Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

PENGANTAR KECERDASAN BUATAN (AI)

YAYASAN PRIMA AGUS TEKNIK

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

PENGANTAR KECERDASAN BUATAN (AI)

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-634-7227-28-7 (PDF)
9 786347 227287

Pengantar Kecerdasan Buatan (AI)

Penulis :
Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

ISBN : 978-634-7227-28-7

Editor :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

Penyunting :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Anggota IKAPI No: 279 / ALB / JTE / 2023

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. 08122925000
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji syukur saya panjatkan kepada Tuhan Yang Maha Esa atas terselesaikannya buku berjudul "***Pengantar Kecerdasan Buatan (AI)***" ini. Buku ini disusun dengan tujuan memberikan pemahaman mendalam dan komprehensif mengenai dasar-dasar kecerdasan buatan (Artificial Intelligence), mulai dari konsep dasar, sejarah, hingga perkembangan terkini di bidang ini.

Buku ini terbagi dalam beberapa bab yang sistematis, dimulai dengan pendahuluan yang mengulas definisi, dasar, dan sejarah AI. Selanjutnya, pembaca diajak mengenal agen cerdas dan lingkungan tempat agen tersebut beroperasi, serta konsep rasionalitas yang menjadi landasan perilaku agen. Bab pertama memperkenalkan pembaca pada apa itu kecerdasan buatan, memaparkan dasar-dasar dan sejarahnya, serta membahas perkembangan terbaru di bidang ini. Bab ini menjadi fondasi penting bagi pemahaman seluruh isi buku.

Bab kedua menjelaskan agen cerdas—entitas yang dapat berperilaku rasional dalam sebuah lingkungan. Konsep lingkungan, rasionalitas, dan struktur agen diuraikan secara mendetail untuk membekali pembaca memahami agen dalam AI. Bab selanjutnya memberi perhatian khusus pada pendekatan pemecahan masalah melalui teknik pencarian, dari pencarian tanpa informasi hingga heuristik dan algoritma optimasi yang canggih. Pembahasan ini mendalam dan menyertakan contoh aplikasi nyata.

Di bab permainan dan pencarian bersaing, pembaca diajak memahami strategi pengambilan keputusan dalam lingkungan kompetitif dan stokastik, termasuk teknik pemangkasan Alpha-Beta yang meningkatkan efisiensi pencarian. Masalah kepuasan terhadap kendala, agen berbasis logika, dan logika orde pertama menjadi fokus bab berikutnya, menyoroti bagaimana AI dapat memformalkan dan mengelola pengetahuan serta inferensi secara sistematis.

Akhirnya, bab tentang inferensi dalam logika orde pertama menutup pembahasan teoritis dengan penjelasan teknik inferensi maju, seperti unifikasi, forward dan backward chaining, serta resolusi yang menjadi dasar implementasi AI modern.

Selain itu, pembaca akan menemukan pembahasan tentang pemecahan masalah kepuasan terhadap kendala, agen logis, serta logika orde pertama yang merupakan dasar penting dalam representasi dan inferensi pengetahuan dalam AI. Kami juga mengulas teknik inferensi pada logika orde pertama, yang menjadi fondasi algoritma cerdas dalam berbagai aplikasi praktis. Buku ini diakhiri dengan daftar pustaka sebagai sumber referensi bagi pembaca yang ingin mendalami lebih jauh.

Semoga buku ini dapat menjadi pegangan yang bermanfaat bagi mahasiswa, akademisi, dan praktisi di bidang kecerdasan buatan. Kami berharap buku ini dapat menumbuhkan minat dan pemahaman yang lebih mendalam untuk mengeksplorasi dan mengembangkan teknologi AI yang semakin maju. Kami mengucapkan terima kasih kepada semua pihak yang telah membantu dalam penyusunan buku ini. Kritik dan saran yang membangun sangat kami harapkan demi penyempurnaan karya ini di masa mendatang.

Semarang, Agustus 2025

Penulis

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

DAFTAR ISI

BAB 1
PENDAHULUAN

1.1 APA ITU AI?

Kami telah mengklaim bahwa AI itu menarik, tetapi kami belum mengatakan apa itu. Pada Gambar 1.1, kita melihat delapan definisi AI, yang disusun sepanjang dua dimensi. Definisi di atas berkaitan dengan proses berpikir dan penalaran, sedangkan yang di bawah membahas perilaku. Definisi di sebelah kiri mengukur keberhasilan dalam hal kesetiaan terhadap kinerja manusia, sedangkan yang di sebelah kanan mengukur terhadap ukuran kinerja ideal, yang disebut rasionalitas. Suatu sistem adalah rasional jika melakukan "hal yang benar," berdasarkan apa yang diketahuinya.

Secara historis, keempat pendekatan terhadap AI telah diikuti, masing-masing oleh orang yang berbeda dengan metode yang berbeda. Pendekatan yang berpusat pada manusia sebagian harus merupakan ilmu empiris, yang melibatkan pengamatan dan hipotesis tentang perilaku manusia. Pendekatan rasionalis melibatkan kombinasi matematika dan teknik. Berbagai kelompok saling meremehkan dan membantu. Mari kita bahas keempat pendekatan tersebut secara lebih rinci.

Berpikir Secara Manusiawi	**Berpikir Rasional**
"Upaya baru yang menarik untuk membuat komputer berpikir ... seperti mesin yang memiliki pikiran, dalam arti yang sebenarnya." (Haugeland, 1985) "[Otomatisasi] aktivitas yang kita kaitkan dengan pemikiran manusia, aktivitas seperti pengambilan keputusan, pemecahan masalah, pembelajaran .. ." (Bellman, 1978)	"Studi tentang kemampuan mental melalui penggunaan model komputasional." (Charniak dan McDermott, 1985) "Studi tentang komputasi yang memungkinkan persepsi, penalaran, dan tindakan." (Winston, 1992)
Bertindak Secara Manusiawi	**Bertindak Secara Rasional**
"Seni menciptakan mesin yang menjalankan fungsi yang memerlukan kecerdasan saat dilakukan oleh manusia." (Kurzweil, 1990) "Studi tentang cara membuat komputer melakukan hal-hal yang, saat ini, dapat dilakukan manusia dengan lebih baik." (Rich dan Knight, 1991)	"Kecerdasan Komputasional adalah studi tentang desain agen cerdas." (Poole et al., 1998) "AI . . . berkaitan dengan perilaku cerdas dalam artefak." (Nilsson, 1998)

Gambar 1.1 Beberapa definisi kecerdasan buatan, dibagi menjadi empat kategori.

Bertindak Secara Manusiawi: Pendekatan Tes Turing

Tes Turing, yang diusulkan oleh Alan Turing, dirancang untuk memberikan definisi operasional yang memuaskan tentang kecerdasan. Komputer lulus tes jika seorang interogator manusia, setelah mengajukan beberapa pertanyaan tertulis, tidak dapat mengatakan apakah tanggapan tertulis tersebut berasal dari seseorang atau dari komputer. Bab 26 membahas perincian tes dan apakah komputer akan benar-benar cerdas jika lulus. Untuk saat ini, kami

mencatat bahwa memprogram komputer untuk lulus tes yang diterapkan secara ketat menyediakan banyak hal untuk dikerjakan. Komputer harus memiliki kemampuan berikut:

- pemrosesan bahasa alami agar dapat berkomunikasi dengan sukses dalam bahasa Inggris;
- representasi pengetahuan untuk menyimpan apa yang diketahui atau didengarnya;
- penalaran otomatis untuk menggunakan informasi yang tersimpan guna menjawab pertanyaan dan menarik kesimpulan baru;
- pembelajaran mesin untuk beradaptasi dengan keadaan baru dan untuk mendeteksi serta mengekstrapolasi pola.

Uji Turing sengaja menghindari interaksi fisik langsung antara interogator dan komputer, karena simulasi fisik seseorang tidak diperlukan untuk kecerdasan. Namun, apa yang disebut Uji Turing total mencakup sinyal video sehingga interogator dapat menguji kemampuan persepsi subjek, serta kesempatan bagi interogator untuk memasukkan objek fisik "melalui lubang palka." Untuk lulus Uji Turing total, komputer akan membutuhkan:

- visi komputer untuk mengamati objek, dan
- robotika untuk memanipulasi objek dan bergerak.

Enam disiplin ilmu ini menyusun sebagian besar AI, dan Turing layak mendapat penghargaan karena merancang uji yang tetap relevan 60 tahun kemudian. Namun, peneliti AI hanya mencurahkan sedikit upaya untuk lulus Uji Turing, percaya bahwa lebih penting untuk mempelajari prinsip-prinsip dasar kecerdasan daripada menduplikasi contoh. Pencarian "penerbangan buatan" berhasil ketika Wright bersaudara dan yang lainnya berhenti meniru burung dan mulai menggunakan terowongan angin dan belajar tentang aerodinamika. Teks teknik penerbangan tidak mendefinisikan tujuan bidang mereka sebagai pembuatan "mesin yang terbang persis seperti merpati sehingga dapat mengelabui merpati lainnya."

Berpikir Secara Manusiawi: Pendekatan Pemodelan Kognitif

Jika kita akan mengatakan bahwa suatu program berpikir seperti manusia, kita harus memiliki cara untuk menentukan bagaimana manusia berpikir. Kita perlu memahami cara kerja pikiran manusia yang sebenarnya. Ada tiga cara untuk melakukannya: melalui introspeksi—berusaha menangkap pikiran kita sendiri saat pikiran itu melintas; melalui eksperimen psikologis—mengamati seseorang saat beraksi; dan melalui pencitraan otak—mengamati otak saat beraksi.

Setelah kita memiliki teori pikiran yang cukup tepat, teori tersebut dapat diungkapkan sebagai program komputer. Jika perilaku masukan-keluaran program tersebut sesuai dengan perilaku manusia yang sesuai, itu merupakan bukti bahwa beberapa mekanisme program tersebut juga dapat beroperasi pada manusia. Misalnya, Allen Newell dan Herbert Simon, yang mengembangkan GPS, "*General Problem Solver*", tidak puas hanya dengan program mereka yang memecahkan masalah dengan benar. Mereka lebih peduli dengan membandingkan jejak langkah penalarannya dengan jejak subjek manusia yang memecahkan masalah yang sama. Bidang ilmu kognitif interdisipliner menyatukan model komputer dari AI dan teknik eksperimental dari psikologi untuk membangun teori yang tepat dan dapat diuji tentang pikiran manusia.

Ilmu kognitif merupakan bidang yang menarik, layak untuk beberapa buku teks dan setidaknya satu ensiklopedia. Kami sesekali akan mengomentari persamaan atau perbedaan antara teknik AI dan kognisi manusia. Namun, ilmu kognitif yang sebenarnya tentu saja didasarkan pada penyelidikan eksperimental terhadap manusia atau hewan yang sebenarnya. Kami akan membahasnya di buku-buku lain, karena kami berasumsi pembaca hanya memiliki komputer untuk bereksperimen.

Pada masa-masa awal AI, sering kali terjadi kebingungan antara pendekatan: seorang penulis akan berpendapat bahwa suatu algoritme bekerja dengan baik pada suatu tugas dan karenanya merupakan model kinerja manusia yang baik, atau sebaliknya. Penulis modern memisahkan kedua jenis klaim tersebut; perbedaan ini memungkinkan AI dan ilmu kognitif berkembang lebih cepat. Kedua bidang tersebut terus saling melengkapi, terutama dalam visi komputer, yang menggabungkan bukti neurofisiologis ke dalam model komputasional.

Berpikir Secara Rasional: Pendekatan "Hukum-Hukum Pikiran"

Filsuf Yunani Aristoteles adalah salah satu orang pertama yang mencoba mengkodifikasikan "pemikiran yang benar," yaitu, proses penalaran yang tak terbantahkan. Silogismenya menyediakan pola untuk struktur argumen yang selalu menghasilkan kesimpulan yang benar ketika diberikan premis yang benar—misalnya, "Socrates adalah manusia; semua manusia fana; oleh karena itu, Socrates fana." Hukum-hukum pikiran ini seharusnya mengatur operasi pikiran; studi mereka memulai bidang yang disebut logika.

Para ahli logika pada abad ke-19 mengembangkan notasi yang tepat untuk pernyataan tentang semua jenis objek di dunia dan hubungan di antara mereka. (Bandingkan ini dengan notasi aritmatika biasa, yang hanya menyediakan pernyataan tentang angka.) Pada tahun 1965, ada program yang, pada prinsipnya, dapat memecahkan masalah yang dapat dipecahkan yang dijelaskan dalam notasi logis. (Meskipun jika tidak ada solusi, program tersebut mungkin berulang selamanya.) Apa yang disebut tradisi logika dalam kecerdasan buatan berharap untuk membangun program-program tersebut untuk menciptakan sistem cerdas. Ada dua kendala utama untuk pendekatan ini.

Pertama, tidak mudah untuk mengambil pengetahuan informal dan menyatakannya dalam istilah formal yang disyaratkan oleh notasi logis, khususnya ketika pengetahuan tersebut kurang dari 100% pasti. Kedua, ada perbedaan besar antara memecahkan masalah "pada prinsipnya" dan memecahkannya dalam praktik. Bahkan masalah dengan hanya beberapa ratus fakta dapat menghabiskan sumber daya komputasi komputer mana pun kecuali jika komputer tersebut memiliki beberapa panduan tentang langkah penalaran mana yang harus dicoba terlebih dahulu. Meskipun kedua kendala ini berlaku untuk setiap upaya membangun sistem penalaran komputasional, kendala tersebut muncul pertama kali dalam tradisi logika.

Bertindak Secara Rasional: Pendekatan Agen Rasional

Agen hanyalah sesuatu yang bertindak (agen berasal dari bahasa Latin agere, melakukan). Tentu saja, semua program komputer melakukan sesuatu, tetapi agen komputer diharapkan untuk melakukan lebih banyak hal: beroperasi secara mandiri, memahami lingkungannya, bertahan dalam jangka waktu yang lama, beradaptasi dengan perubahan, dan

menciptakan serta mengejar tujuan. Agen rasional adalah agen yang bertindak untuk mencapai hasil terbaik atau, ketika ada ketidakpastian, hasil terbaik yang diharapkan.

Dalam pendekatan "hukum pemikiran" terhadap AI, penekanannya adalah pada kesimpulan yang benar. Membuat kesimpulan yang benar terkadang menjadi bagian dari menjadi agen yang rasional, karena salah satu cara untuk bertindak rasional adalah dengan bernalar secara logis hingga mencapai kesimpulan bahwa tindakan tertentu akan mencapai tujuan seseorang dan kemudian bertindak berdasarkan kesimpulan itu. Di sisi lain, kesimpulan yang benar tidak sepenuhnya rasional; dalam beberapa situasi, tidak ada hal yang terbukti benar untuk dilakukan, tetapi sesuatu tetap harus dilakukan. Ada juga cara bertindak rasional yang tidak dapat dikatakan melibatkan kesimpulan. Misalnya, mundur dari kompor yang panas adalah tindakan refleks yang biasanya lebih berhasil daripada tindakan yang lebih lambat yang diambil setelah pertimbangan yang cermat.

Semua keterampilan yang dibutuhkan untuk Tes Turing juga memungkinkan agen untuk bertindak rasional. Representasi pengetahuan dan penalaran memungkinkan agen untuk mencapai keputusan yang baik. Kita perlu mampu menghasilkan kalimat yang dapat dipahami dalam bahasa alami untuk bertahan hidup dalam masyarakat yang kompleks. Kita perlu belajar tidak hanya untuk pengetahuan, tetapi juga karena hal itu meningkatkan kemampuan kita untuk menghasilkan perilaku yang efektif. Pendekatan agen rasional memiliki dua keunggulan dibanding pendekatan lainnya.

Pertama, pendekatan ini lebih umum daripada pendekatan "hukum pemikiran" karena inferensi yang benar hanyalah salah satu dari beberapa mekanisme yang mungkin untuk mencapai rasionalitas. Kedua, pendekatan ini lebih sesuai untuk pengembangan ilmiah daripada pendekatan yang didasarkan pada perilaku manusia atau pemikiran manusia. Standar rasionalitas didefinisikan dengan baik secara matematis dan sepenuhnya umum, dan dapat "diuraikan" untuk menghasilkan desain agen yang terbukti mencapainya.

Di sisi lain, perilaku manusia sangat cocok untuk satu lingkungan tertentu dan didefinisikan oleh, yah, jumlah total dari semua hal yang dilakukan manusia. Oleh karena itu, buku ini berkonsentrasi pada prinsip-prinsip umum agen rasional dan komponen untuk membangunnya. Kita akan melihat bahwa meskipun masalah tersebut tampak sederhana, berbagai macam masalah muncul ketika kita mencoba menyelesaikannya. Bab 2 menguraikan beberapa masalah ini secara lebih rinci.

Satu hal penting yang perlu diingat: Kita akan segera melihat bahwa mencapai rasionalitas sempurna—selalu melakukan hal yang benar—tidak dapat dilakukan dalam lingkungan yang rumit. Tuntutan komputasional terlalu tinggi. Namun, untuk sebagian besar buku ini, kita akan mengadopsi hipotesis kerja bahwa rasionalitas sempurna adalah titik awal yang baik untuk analisis. Ini menyederhanakan masalah dan menyediakan latar yang tepat untuk sebagian besar materi dasar di bidang ini. Bab 5 dan 17 membahas secara eksplisit masalah rasionalitas terbatas—bertindak dengan tepat ketika tidak ada cukup waktu untuk melakukan semua perhitungan yang mungkin diinginkan.

1.2 DASAR-DASAR KECERDASAN BUATAN

Pada bagian ini, kami menyajikan sejarah singkat disiplin ilmu yang menyumbangkan ide, sudut pandang, dan teknik untuk AI. Seperti sejarah lainnya, sejarah ini dipaksa untuk berkonsentrasi pada sejumlah kecil orang, peristiwa, dan ide serta mengabaikan hal-hal lain yang juga penting. Kami menyusun sejarah berdasarkan serangkaian pertanyaan. Kami tentu tidak ingin memberi kesan bahwa pertanyaan-pertanyaan ini adalah satu-satunya pertanyaan yang dibahas oleh disiplin ilmu atau bahwa semua disiplin ilmu telah berupaya mencapai AI sebagai hasil akhirnya.

Filsafat

- Dapatkah aturan formal digunakan untuk menarik kesimpulan yang valid?
- Bagaimana pikiran muncul dari otak fisik?
- Dari mana pengetahuan berasal?
- Bagaimana pengetahuan mengarah pada tindakan?

Aristoteles (384–322 B.C.), yang patung dadanya muncul di sampul depan buku ini, adalah orang pertama yang merumuskan serangkaian hukum yang tepat yang mengatur bagian rasional dari pikiran. Ia mengembangkan sistem silogisme informal untuk penalaran yang tepat, yang pada prinsipnya memungkinkan seseorang untuk menghasilkan kesimpulan secara mekanis, dengan premis awal yang diberikan.

Jauh kemudian, Ramon Lull (meninggal 1315) memiliki gagasan bahwa penalaran yang berguna sebenarnya dapat dilakukan oleh artefak mekanis. Thomas Hobbes mengusulkan bahwa penalaran seperti komputasi numerik, bahwa "kita menambah dan mengurangi dalam pikiran kita yang diam." Otomatisasi komputasi itu sendiri sudah berjalan dengan baik. Sekitar tahun 1500, Leonardo da Vinci merancang tetapi tidak membangun kalkulator mekanis; rekonstruksi baru-baru ini menunjukkan bahwa desainnya berfungsi.

Mesin hitung pertama yang diketahui dibangun sekitar tahun 1623 oleh ilmuwan Jerman Wilhelm Schickard, meskipun Pascaline, yang dibangun pada tahun 1642 oleh Blaise Pascal, lebih terkenal. Pascal menulis bahwa "mesin aritmatika menghasilkan efek yang tampak lebih dekat dengan pikiran daripada semua tindakan hewan." Gottfried Wilhelm Leibniz membuat perangkat mekanis yang dimaksudkan untuk melakukan operasi pada konsep daripada angka, tetapi cakupannya agak terbatas. Leibniz memang melampaui Pascal dengan membuat kalkulator yang dapat menambah, mengurangi, mengalikan, dan mengambil akar, sedangkan Pascaline hanya dapat menambah dan mengurangi. Beberapa orang berspekulasi bahwa mesin mungkin tidak hanya melakukan perhitungan tetapi benar-benar dapat berpikir dan bertindak sendiri. Dalam bukunya Leviathan tahun 1651, Thomas Hobbes mengusulkan gagasan tentang "hewan buatan", dengan alasan "Karena jantung hanyalah pegas; dan saraf, selain dari begitu banyak tali; dan sendi, selain dari begitu banyak roda." Mengatakan bahwa pikiran beroperasi, setidaknya sebagian, menurut aturan logis, dan membangun sistem fisik yang meniru beberapa aturan tersebut adalah satu hal; mengatakan bahwa pikiran itu sendiri adalah sistem fisik semacam itu adalah hal yang lain. Rene´ Descartes memberikan pembahasan pertama yang jelas tentang perbedaan antara pikiran dan materi serta masalah-masalah yang muncul.

Salah satu masalah dengan konsepsi pikiran yang murni fisik adalah bahwa konsepsi tersebut tampaknya tidak memberikan banyak ruang bagi kehendak bebas: jika pikiran sepenuhnya diatur oleh hukum-hukum fisika, maka pikiran tidak memiliki kehendak bebas lebih dari sekadar batu yang "memutuskan" untuk jatuh ke pusat bumi. Descartes adalah pendukung kuat kekuatan penalaran dalam memahami dunia, sebuah filsafat yang sekarang disebut rasionalisme, dan filsafat yang menganggap Aristoteles dan Leibnitz sebagai anggotanya. Namun, Descartes juga merupakan pendukung dualisme. Ia berpendapat bahwa ada bagian dari pikiran manusia (atau jiwa atau roh) yang berada di luar alam, yang terbebas dari hukum-hukum fisika. Sebaliknya, hewan tidak memiliki kualitas ganda ini; mereka dapat diperlakukan sebagai mesin. Alternatif untuk dualisme adalah materialisme, yang menyatakan bahwa operasi otak menurut hukum-hukum fisika membentuk pikiran. Kehendak bebas hanyalah cara persepsi pilihan-pilihan yang tersedia tampak bagi entitas yang memilih.

Mengingat adanya pikiran fisik yang memanipulasi pengetahuan, masalah berikutnya adalah menetapkan sumber pengetahuan. Gerakan empirisme, yang dimulai dengan Novum Organum karya Francis Bacon, dicirikan oleh diktum John Locke: "Tidak ada sesuatu pun dalam pemahaman, yang tidak ada terlebih dahulu dalam indra." *A Treatise of Human Nature* karya David Hume mengusulkan apa yang sekarang dikenal sebagai prinsip induksi: bahwa aturan umum diperoleh melalui pemaparan terhadap asosiasi berulang-ulang antara unsur-unsurnya.

Berdasarkan karya Ludwig Wittgenstein dan Bertrand Russell, Vienna Circle yang terkenal, yang dipimpin oleh Rudolf Carnap, mengembangkan doktrin positivisme logis. Doktrin ini menyatakan bahwa semua pengetahuan dapat dicirikan oleh teori-teori logis yang terhubung, pada akhirnya, dengan kalimat-kalimat observasi yang sesuai dengan masukan sensorik; dengan demikian positivisme logis menggabungkan rasionalisme dan empirisme. Teori konfirmasi Carnap dan Carl Hempel mencoba menganalisis perolehan pengetahuan dari pengalaman.

Buku Carnap berjudul *The Logical Structure of the World* mendefinisikan prosedur komputasional eksplisit untuk mengekstraksi pengetahuan dari pengalaman-pengalaman dasar. Itu mungkin teori pikiran pertama sebagai proses komputasional. Elemen terakhir dalam gambaran filosofis tentang pikiran adalah hubungan antara pengetahuan dan tindakan. Pertanyaan ini penting bagi AI karena kecerdasan memerlukan tindakan dan juga penalaran. Selain itu, hanya dengan memahami bagaimana tindakan dibenarkan, kita dapat memahami cara membangun agen yang tindakannya dapat dibenarkan (atau rasional). Aristoteles berpendapat (dalam De Motu Animalium) bahwa tindakan dibenarkan oleh hubungan logis antara tujuan dan pengetahuan tentang hasil tindakan (bagian terakhir dari kutipan ini juga muncul di sampul depan buku ini, dalam bahasa Yunani asli):

> Tetapi bagaimana bisa terjadi bahwa berpikir terkadang disertai dengan tindakan dan terkadang tidak, terkadang dengan gerakan, dan terkadang tidak? Tampaknya hal yang hampir sama terjadi seperti dalam kasus penalaran dan membuat kesimpulan tentang objek yang tidak berubah. Tetapi dalam kasus itu, akhirnya adalah proposisi spekulatif ... sedangkan di sini kesimpulan yang dihasilkan dari kedua premis tersebut adalah tindakan yang perlu saya tutupi; jubah adalah penutup. Saya perlu jubah. Apa yang saya butuhkan, harus saya buat; saya perlu jubah. Saya harus membuat jubah. Dan kesimpulannya, "Saya harus membuat jubah," adalah sebuah tindakan.

Dalam Etika Nikomakhean (Buku III. 3, 1112b), Aristoteles lebih jauh menguraikan topik ini, dengan menyarankan sebuah algoritma:

> Kita tidak berunding tentang tujuan, tetapi tentang cara. Karena seorang dokter tidak berunding apakah ia akan menyembuhkan, atau seorang orator tidak berunding apakah ia akan membujuk. Mereka mengasumsikan tujuan dan mempertimbangkan bagaimana dan dengan cara apa tujuan itu dicapai, dan apakah tampaknya mudah dan paling baik dicapai dengan cara itu; sementara jika tujuan dicapai hanya dengan satu cara, mereka mempertimbangkan bagaimana tujuan akan dicapai dengan cara ini dan dengan cara apa tujuan ini akan dicapai, hingga mereka sampai pada penyebab pertama, dan apa yang terakhir dalam urutan analisis tampaknya menjadi yang pertama dalam urutan menjadi. Dan jika kita sampai pada suatu kemustahilan, kita menyerah mencari, misalnya, jika kita membutuhkan uang dan ini tidak dapat diperoleh; tetapi jika sesuatu tampak mungkin, kita mencoba melakukannya.

Algoritma Aristoteles diimplementasikan 2300 tahun kemudian oleh Newell dan Simon dalam program GPS mereka. Kita sekarang akan menyebutnya sistem perencanaan regresi (lihat Bab 10).

Analisis berbasis tujuan memang berguna, tetapi tidak mengatakan apa yang harus dilakukan ketika beberapa tindakan akan mencapai tujuan atau ketika tidak ada tindakan yang akan mencapainya sepenuhnya. Antoine Arnaul dengan tepat menggambarkan rumus kuantitatif untuk memutuskan tindakan apa yang harus diambil dalam kasus seperti ini (lihat Bab 16). Buku Utilitarianisme karya John Stuart Mill mempromosikan gagasan kriteria keputusan rasional di semua bidang aktivitas manusia. Teori keputusan yang lebih formal dibahas di bagian berikut.

Matematika

- Apa aturan formal untuk menarik kesimpulan yang valid?
- Apa yang dapat dihitung?
- Bagaimana kita bernalar dengan informasi yang tidak pasti?

Para filsuf mempertaruhkan beberapa ide dasar AI, tetapi lompatan ke ilmu formal memerlukan tingkat formalisasi matematika dalam tiga bidang dasar: logika, komputasi, dan probabilitas. Gagasan logika formal dapat ditelusuri kembali ke filsuf Yunani kuno, tetapi perkembangan matematikanya benar-benar dimulai dengan karya George Boole, yang menyusun rincian logika proposisional, atau Boolean. Pada tahun 1879, Gottlob Frege memperluas logika Boole untuk mencakup objek dan relasi, menciptakan logika orde pertama yang digunakan saat ini. Alfred Tarski memperkenalkan teori referensi yang menunjukkan cara menghubungkan objek dalam logika dengan objek di dunia nyata.

Langkah selanjutnya adalah menentukan batasan apa yang dapat dilakukan dengan logika dan komputasi. Algoritma nontrivial pertama dianggap sebagai algoritma Euclid untuk menghitung pembagi persekutuan terbesar. Kata algoritma (dan gagasan untuk mempelajarinya) berasal dari al-Khowarazmi, seorang matematikawan Persia abad ke-9, yang tulisannya juga memperkenalkan angka Arab dan aljabar ke Eropa. Boole dan yang lainnya membahas algoritma untuk deduksi logis, dan, pada akhir abad ke-19, upaya sedang dilakukan untuk memformalkan penalaran matematika umum sebagai deduksi logis.

Pada tahun 1930, Kurt Godel menunjukkan bahwa ada prosedur yang efektif untuk membuktikan pernyataan benar apa pun dalam logika orde pertama Frege dan Russell, tetapi logika orde pertama tidak dapat menangkap prinsip induksi matematika yang diperlukan untuk

mengkarakterisasi bilangan asli. Pada tahun 1931, Godel menunjukkan bahwa batasan deduksi memang ada. Teorema ketidaklengkapannya menunjukkan bahwa dalam teori formal apa pun sekuat aritmatika Peano (teori dasar bilangan asli), ada pernyataan benar yang tidak dapat diputuskan dalam arti bahwa pernyataan tersebut tidak memiliki bukti dalam teori tersebut.

Hasil mendasar ini juga dapat diartikan sebagai menunjukkan bahwa beberapa fungsi pada bilangan bulat tidak dapat direpresentasikan oleh suatu algoritma—yaitu, fungsi tersebut tidak dapat dihitung. Hal ini memotivasi Alan Turing untuk mencoba mengkarakterisasi secara tepat fungsi mana yang dapat dihitung—yang mampu dihitung. Gagasan ini sebenarnya sedikit bermasalah karena gagasan tentang perhitungan atau prosedur efektif benar-benar tidak dapat diberikan definisi formal. Namun, tesis Church–Turing, yang menyatakan bahwa mesin Turing mampu menghitung fungsi yang dapat dihitung, secara umum diterima sebagai definisi yang memadai. Turing juga menunjukkan bahwa ada beberapa fungsi yang tidak dapat dihitung oleh mesin Turing. Misalnya, tidak ada mesin yang dapat memberi tahu secara umum apakah program tertentu akan memberikan jawaban pada input tertentu atau berjalan selamanya. Meskipun keterputusan dan keterkomputasian penting untuk memahami komputasi, gagasan tentang keterlacakan memiliki dampak yang lebih besar.

Secara kasar, suatu masalah disebut sulit dipecahkan jika waktu yang dibutuhkan untuk memecahkan contoh-contoh masalah tersebut bertambah secara eksponensial seiring dengan ukuran contoh-contoh tersebut. Perbedaan antara pertumbuhan polinomial dan eksponensial dalam kompleksitas pertama kali ditekankan pada pertengahan tahun 1960an. Hal ini penting karena pertumbuhan eksponensial berarti bahwa bahkan contoh-contoh yang cukup besar tidak dapat dipecahkan dalam waktu yang wajar. Oleh karena itu, seseorang harus berusaha untuk membagi keseluruhan masalah dalam menghasilkan perilaku cerdas menjadi submasalah yang dapat dipecahkan daripada yang sulit dipecahkan.

Bagaimana seseorang dapat mengenali masalah yang sulit dipecahkan? Teori kelengkapan NP, yang dipelopori oleh Steven Cook dan Richard Karp, menyediakan suatu metode. Cook dan Karp menunjukkan keberadaan kelas-kelas besar masalah pencarian dan penalaran kombinatorial kanonik yang bersifat NP-lengkap. Setiap kelas masalah yang dapat direduksi menjadi kelas masalah NP-lengkap cenderung tidak dapat dipecahkan. (Meskipun belum terbukti bahwa masalah NP-lengkap tentu tidak dapat dipecahkan, sebagian besar ahli teori mempercayainya.) Hasil-hasil ini kontras dengan optimisme yang ditunjukkan pers populer saat menyambut komputer pertama—"Otak Super Elektronik" yang "Lebih Cepat dari Einstein!" Meskipun kecepatan komputer meningkat, penggunaan sumber daya yang cermat akan menjadi ciri sistem cerdas. Secara kasar, dunia adalah contoh masalah yang sangat besar! Pekerjaan dalam AI telah membantu menjelaskan mengapa beberapa contoh masalah NP-lengkap sulit, namun yang lainnya mudah.

Selain logika dan komputasi, kontribusi besar ketiga matematika bagi AI adalah teori probabilitas. Gerolamo Cardano dari Italia adalah orang pertama yang merumuskan gagasan probabilitas, menggambarkannya dalam bentuk kemungkinan hasil dari peristiwa perjudian. Pada tahun 1654, Blaise Pascal, dalam suratnya kepada Pierre Fermat, menunjukkan cara

meramal masa depan dari permainan judi yang belum selesai dan menetapkan hasil rata-rata bagi para penjudi.

Probabilitas dengan cepat menjadi bagian yang sangat berharga dari semua ilmu kuantitatif, membantu menangani pengukuran yang tidak pasti dan teori yang tidak lengkap. James Bernoulli, Pierre Laplace, dan yang lainnya memajukan teori tersebut dan memperkenalkan metode statistik baru. Thomas Bayes, yang muncul di sampul depan buku ini, mengusulkan aturan untuk memperbarui probabilitas berdasarkan bukti baru. Aturan Bayes mendasari sebagian besar pendekatan modern terhadap penalaran yang tidak pasti dalam sistem AI.

Ekonomi

- Bagaimana kita harus membuat keputusan agar dapat memaksimalkan keuntungan?
- Bagaimana kita harus melakukannya jika orang lain mungkin tidak setuju?
- Bagaimana kita harus melakukannya jika keuntungannya mungkin masih jauh di masa depan?

Ilmu ekonomi dimulai pada tahun 1776, ketika filsuf Skotlandia Adam Smith menerbitkan *An Inquiry into the Nature and Causes of the Wealth of Nations*. Meskipun orang Yunani kuno dan lainnya telah memberikan kontribusi pada pemikiran ekonomi, Smith adalah orang pertama yang menganggapnya sebagai ilmu, dengan menggunakan gagasan bahwa ekonomi dapat dianggap terdiri dari agen individu yang memaksimalkan kesejahteraan ekonomi mereka sendiri.

Kebanyakan orang menganggap ekonomi sebagai ilmu tentang uang, tetapi para ekonom akan mengatakan bahwa mereka benar-benar mempelajari bagaimana orang membuat pilihan yang mengarah pada hasil yang diinginkan. Ketika McDonald's menawarkan hamburger seharga satu dolar, mereka menegaskan bahwa mereka lebih suka dolar dan berharap pelanggan akan lebih suka hamburger. Perlakuan matematis terhadap “hasil yang lebih disukai” atau utilitas pertama kali diformalkan oleh Le´on Walras (diucapkan “Valrasse”) dan disempurnakan oleh Frank Ramsey dan kemudian oleh John von Neumann dan Oskar Morgenstern dalam buku mereka *The Theory of Games and Economic Behavior*.

Teori keputusan, yang menggabungkan teori probabilitas dengan teori utilitas, menyediakan kerangka formal dan lengkap untuk keputusan (ekonomi atau lainnya) yang dibuat dalam ketidakpastian—yaitu, dalam kasus di mana deskripsi probabilistik secara tepat menangkap lingkungan pembuat keputusan. Ini cocok untuk ekonomi “besar” di mana setiap agen tidak perlu memperhatikan tindakan agen lain sebagai individu. Untuk ekonomi “kecil”, situasinya lebih seperti permainan: tindakan satu pemain dapat secara signifikan memengaruhi utilitas pemain lain (baik secara positif maupun negatif).

Pengembangan teori permainan oleh von Neumann dan Morgenstern mencakup hasil yang mengejutkan bahwa, untuk beberapa permainan, agen rasional harus mengadopsi kebijakan yang (atau setidaknya tampak) acak. Tidak seperti teori keputusan, teori permainan tidak menawarkan resep yang tidak ambigu untuk memilih tindakan. Sebagian besar ekonom tidak membahas pertanyaan ketiga yang tercantum di atas, yaitu, bagaimana membuat keputusan rasional ketika hasil dari tindakan tidak langsung tetapi merupakan hasil dari

beberapa tindakan yang diambil secara berurutan. Topik ini ditelusuri dalam bidang riset operasi, yang muncul dalam Perang Dunia II dari upaya di Inggris untuk mengoptimalkan instalasi radar, dan kemudian menemukan aplikasi sipil dalam keputusan manajemen yang kompleks.

Pekerjaan dalam ekonomi dan riset operasi telah banyak berkontribusi pada gagasan kita tentang agen rasional, namun selama bertahun-tahun penelitian AI berkembang di sepanjang jalur yang sepenuhnya terpisah. Salah satu alasannya adalah kompleksitas yang tampak dalam membuat keputusan rasional. Peneliti AI perintis Herbert Simon memenangkan Penghargaan Nobel dalam bidang ekonomi pada tahun 1978 atas karya awalnya yang menunjukkan bahwa model yang didasarkan pada kepuasan—membuat keputusan yang "cukup baik," daripada menghitung keputusan optimal dengan susah payah—memberikan deskripsi yang lebih baik tentang perilaku manusia yang sebenarnya. Sejak tahun 1990an, telah terjadi kebangkitan minat dalam teknik teori keputusan untuk sistem agen.

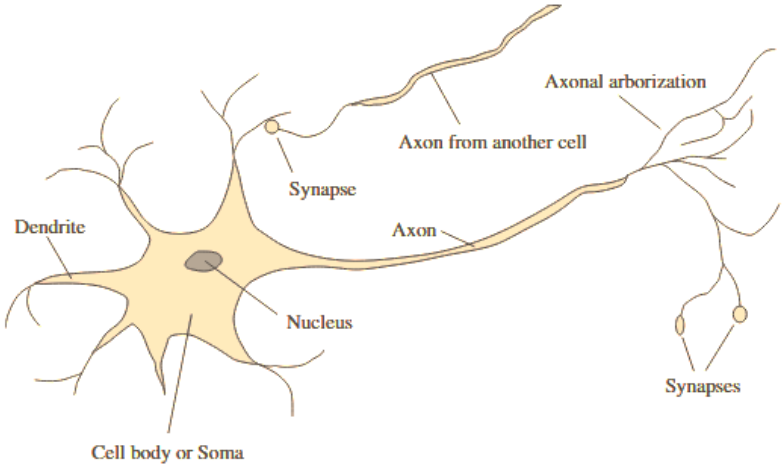
Ilmu Saraf

• Bagaimana otak memproses informasi?

Sains saraf adalah studi tentang sistem saraf, khususnya otak. Meskipun cara pasti otak memungkinkan pikiran menjadi salah satu misteri besar sains, fakta bahwa otak memungkinkan pikiran menjadi kenyataan telah diketahui selama ribuan tahun karena bukti bahwa pukulan keras ke kepala dapat menyebabkan ketidakmampuan mental.

Telah lama diketahui bahwa otak manusia entah bagaimana berbeda; sekitar tahun 335 SM, Aristoteles menulis, "Dari semua hewan, manusia memiliki otak terbesar yang sebanding dengan ukurannya." Namun, baru pada pertengahan abad ke-18 otak secara luas diakui sebagai pusat kesadaran. Sebelumnya, lokasi yang mungkin termasuk jantung dan limpa. Studi Paul Broca tentang afasia (kekurangan bicara) pada pasien yang mengalami kerusakan otak pada tahun 1861 menunjukkan adanya area otak terlokalisasi yang bertanggung jawab atas fungsi kognitif tertentu. Secara khusus, ia menunjukkan bahwa produksi ucapan terlokalisasi pada bagian hemisfer kiri yang sekarang disebut area Broca.

Pada saat itu, diketahui bahwa otak terdiri dari sel-sel saraf, atau neuron, tetapi baru pada tahun 1873 Camillo Golgi mengembangkan teknik pewarnaan yang memungkinkan pengamatan neuron individual di otak (lihat Gambar 1.2). Teknik ini digunakan oleh Santiago Ramon y Cajal dalam studi perintisnya tentang struktur neuron otak. Nicolas Rashevsky adalah orang pertama yang menerapkan model matematika untuk mempelajari sistem saraf.



Gambar 1.2 Bagian-bagian sel saraf atau neuron. Setiap neuron terdiri dari badan sel, atau soma, yang berisi inti sel. Bercabang keluar dari badan sel adalah sejumlah serat yang disebut dendrit dan satu serat panjang yang disebut akson. Akson membentang untuk jarak yang jauh, jauh lebih panjang dari skala yang ditunjukkan dalam diagram ini. Biasanya, akson panjangnya 1 cm (100 kali diameter badan sel), tetapi dapat mencapai hingga 1 meter. Sebuah neuron membuat koneksi dengan 10 hingga 100.000 neuron lain di persimpangan yang disebut sinapsis. Sinyal disebarkan dari neuron ke neuron melalui reaksi elektrokimia yang rumit. Sinyal mengendalikan aktivitas otak dalam jangka pendek dan juga memungkinkan perubahan jangka panjang dalam konektivitas neuron. Mekanisme ini dianggap membentuk dasar untuk pembelajaran di otak. Sebagian besar pemrosesan informasi terjadi di korteks serebral, lapisan luar otak. Unit organisasi dasar tampaknya berupa kolom jaringan dengan diameter sekitar 0,5 mm, berisi sekitar 20.000 neuron dan meluas ke kedalaman penuh korteks sekitar 4 mm pada manusia).

Kini kita memiliki beberapa data tentang pemetaan antara area otak dan bagian tubuh yang dikontrolnya atau tempat mereka menerima masukan sensorik. Pemetaan semacam itu dapat berubah secara radikal selama beberapa minggu, dan beberapa hewan tampaknya memiliki banyak peta. Selain itu, kita tidak sepenuhnya memahami bagaimana area lain dapat mengambil alih fungsi saat satu area rusak. Hampir tidak ada teori tentang bagaimana memori individu disimpan.

Pengukuran aktivitas otak yang utuh dimulai pada tahun 1929 dengan penemuan elektroensefalograf (EEG) oleh Hans Berger. Perkembangan terkini pencitraan resonansi magnetik fungsional (fMRI) memberi para ahli saraf gambar aktivitas otak yang sangat terperinci, yang memungkinkan pengukuran yang berhubungan dengan proses kognitif yang sedang berlangsung dengan cara yang menarik. Hal ini ditambah dengan kemajuan dalam perekaman aktivitas neuron sel tunggal. Neuron individual dapat dirangsang secara elektrik, kimiawi, atau bahkan optik, yang memungkinkan hubungan masukan-keluaran neuronal dipetakan. Meskipun ada kemajuan ini, kita masih jauh dari pemahaman tentang bagaimana proses kognitif sebenarnya bekerja.

Kesimpulan yang benar-benar menakjubkan adalah bahwa sekumpulan sel sederhana dapat mengarah pada pikiran, tindakan, dan kesadaran atau, dalam kata-kata ringkas John Searle, otak menyebabkan pikiran.

	Superkomputer	*Komputer Pribadi*	*Otak Manusia*
Unit Komutasi	10^4 CPUs, 10^{12} transistor	4 CPUs, 10^9 transistor	10^4 neuron
Unit penyimpanan	10^{14} bits RAM	10^{11} bits RAM	10^{14} neuron
	10^{15} bits disk	10^{13} bits disk	10^{15} sinapsis
Waktu Siklus	10^{-9} detik	10^{-9} detik	10^{-3} detik
Operasi/detik	10^{15}	10^{10}	10^{17}
Pembaruan memori/detik	10^{14}	10^{10}	10^{14}

Gambar 1.3 Perbandingan kasar sumber daya komputasi mentah yang tersedia untuk superkomputer IBM BLUE GENE, komputer pribadi khas tahun 2008, dan otak manusia. Angka-angka otak pada dasarnya tetap, sedangkan angka-angka superkomputer telah meningkat dengan faktor 10 setiap 5 tahun atau lebih, yang memungkinkannya mencapai paritas kasar dengan otak. Komputer pribadi tertinggal dalam semua metrik kecuali waktu siklus.

Satu-satunya teori alternatif yang nyata adalah mistisisme: bahwa pikiran beroperasi di suatu alam mistis yang berada di luar ilmu fisika.

Otak dan komputer digital memiliki sifat yang agak berbeda. Gambar 1.3 menunjukkan bahwa komputer memiliki waktu siklus yang sejuta kali lebih cepat daripada otak. Otak mengatasinya dengan penyimpanan dan interkoneksi yang jauh lebih banyak daripada komputer pribadi kelas atas, meskipun superkomputer terbesar memiliki kapasitas yang mirip dengan otak. (Namun, perlu dicatat bahwa otak tampaknya tidak menggunakan semua neuronnya secara bersamaan.)

Futuris banyak menggunakan angka-angka ini, menunjuk pada singularitas yang mendekat di mana komputer mencapai tingkat kinerja manusia super, tetapi perbandingan mentahnya tidak terlalu informatif. Bahkan dengan komputer dengan kapasitas yang hampir tak terbatas, kita tetap tidak akan tahu bagaimana mencapai tingkat kecerdasan otak.

Psikologi

• Bagaimana manusia dan hewan berpikir dan bertindak?

Asal usul psikologi ilmiah biasanya ditelusuri kembali ke karya fisikawan Jerman Hermann von Helmholtz dan muridnya Wilhelm Wundt. Helmholtz menerapkan metode ilmiah untuk mempelajari penglihatan manusia, dan *Handbook of Physiological Optics*-nya bahkan sekarang digambarkan sebagai "risalah tunggal terpenting tentang fisika dan fisiologi penglihatan manusia".

Pada tahun 1879, Wundt membuka laboratorium psikologi eksperimental pertama di Universitas Leipzig. Wundt bersikeras pada eksperimen yang dikontrol dengan cermat, di mana para pekerjanya akan melakukan tugas perseptual atau asosiatif sambil melakukan introspeksi terhadap proses berpikir mereka. Kontrol yang cermat sangat membantu menjadikan psikologi sebagai ilmu, tetapi sifat subjektif dari data membuat kecil kemungkinan bahwa seorang peneliti akan pernah membantah teorinya sendiri.

Di sisi lain, para ahli biologi yang mempelajari perilaku hewan tidak memiliki data introspektif dan mengembangkan metodologi objektif, seperti yang dijelaskan oleh H. S. Jennings dalam karyanya yang berpengaruh *Behavior of the Lower Organisms*. Menerapkan sudut pandang ini pada manusia, gerakan behaviorisme, yang dipimpin oleh John Watson, menolak teori apa pun yang melibatkan proses mental dengan alasan bahwa introspeksi tidak dapat memberikan bukti yang dapat diandalkan.

Para ahli behaviorisme bersikeras hanya mempelajari ukuran objektif dari persepsi (atau stimulus) yang diberikan kepada hewan dan tindakan (atau respons) yang dihasilkannya. Behaviorisme menemukan banyak hal tentang tikus dan merpati tetapi kurang berhasil dalam memahami manusia. Psikologi kognitif, yang memandang otak sebagai perangkat pemrosesan informasi, dapat ditelusuri kembali setidaknya ke karya William James. Helmholtz juga bersikeras bahwa persepsi melibatkan bentuk inferensi logis bawah sadar. Sudut pandang kognitif sebagian besar dikalahkan oleh behaviorisme di Amerika Serikat, tetapi di Unit Psikologi Terapan Cambridge, yang diarahkan oleh Frederic Bartlett, pemodelan kognitif mampu berkembang pesat. *The Nature of Explanation*, oleh murid dan penerus Bartlett, Kenneth Craik, dengan tegas membangun kembali legitimasi istilah-istilah "mental" seperti keyakinan dan tujuan, dengan menyatakan bahwa istilah-istilah tersebut sama ilmiahnya dengan, katakanlah, menggunakan tekanan dan suhu untuk berbicara tentang gas, meskipun keduanya terbuat dari molekul yang tidak memiliki keduanya. Craik menetapkan tiga langkah utama agen berbasis pengetahuan: (1) stimulus harus diterjemahkan ke dalam representasi internal, (2) representasi dimanipulasi oleh proses kognitif untuk memperoleh representasi internal baru, dan (3) representasi tersebut pada gilirannya diterjemahkan kembali menjadi tindakan. Dia menjelaskan dengan jelas mengapa ini merupakan rancangan yang baik bagi seorang agen:

> Jika organisme membawa "model skala kecil" dari realitas eksternal dan tindakan-tindakannya sendiri yang mungkin di dalam kepalanya, ia mampu mencoba berbagai alternatif, menyimpulkan mana yang terbaik di antara semuanya, bereaksi terhadap situasi-situasi masa depan sebelum situasi-situasi itu muncul, memanfaatkan pengetahuan tentang peristiwa-peristiwa masa lalu dalam menghadapi masa kini dan masa depan, dan dalam segala hal bereaksi dengan cara yang jauh lebih lengkap, lebih aman, dan lebih kompeten terhadap keadaan-keadaan darurat yang dihadapinya.

Setelah kematian Craik dalam kecelakaan sepeda pada tahun 1945, karyanya dilanjutkan oleh Donald Broadbent, yang bukunya *Perception and Communication* adalah salah satu karya pertama yang memodelkan fenomena psikologis sebagai pemrosesan informasi. Sementara itu, di Amerika Serikat, pengembangan pemodelan komputer mengarah pada penciptaan bidang ilmu kognitif. Bidang ini dapat dikatakan telah dimulai di sebuah lokakarya pada bulan September 1956 di MIT. (Kita akan melihat bahwa ini hanya dua bulan setelah konferensi di mana AI itu sendiri "lahir.")

Di lokakarya tersebut, George Miller mempresentasikan *The Magic Number Seven*, Noam Chomsky mempresentasikan *Three Models of Language*, dan Allen Newell dan Herbert Simon mempresentasikan *The Logic Theory Machine*. Ketiga makalah berpengaruh ini menunjukkan bagaimana model komputer dapat digunakan untuk membahas psikologi memori, bahasa, dan pemikiran logis, masing-masing. Sekarang menjadi pandangan umum

(meskipun jauh dari universal) di antara para psikolog bahwa "teori kognitif harus seperti program komputer"; artinya, ia harus menggambarkan mekanisme pemrosesan informasi terperinci yang dengannya beberapa fungsi kognitif dapat dilaksanakan.

Rekayasa Komputer

• Bagaimana kita dapat membangun komputer yang efisien?

Agar kecerdasan buatan berhasil, kita memerlukan dua hal: kecerdasan dan artefak. Komputer telah menjadi artefak pilihan. Komputer elektronik digital modern diciptakan secara independen dan hampir bersamaan oleh para ilmuwan di tiga negara yang terlibat dalam Perang Dunia II. Komputer operasional pertama adalah Heath Robinson elektromekanis, yang dibuat pada tahun 1940 oleh tim Alan Turing untuk satu tujuan: menguraikan pesan-pesan Jerman. Pada tahun 1943, kelompok yang sama mengembangkan Colossus, mesin serba guna yang kuat berdasarkan tabung vakum.

Komputer terprogram operasional pertama adalah Z-3, penemuan Konrad Zuse di Jerman pada tahun 1941. Zuse juga menemukan bilangan floating-point dan bahasa pemrograman tingkat tinggi pertama, Plankalkul. Komputer elektronik pertama, ABC, dirakit oleh John Atanasoff dan muridnya Clifford Berry antara tahun 1940 dan 1942 di Iowa State University. Penelitian Atanasoff hanya mendapat sedikit dukungan atau pengakuan; ENIAC, yang dikembangkan sebagai bagian dari proyek militer rahasia di Universitas Pennsylvania oleh tim yang terdiri dari John Mauchly dan John Eckert, terbukti menjadi cikal bakal komputer modern yang paling berpengaruh.

Sejak saat itu, setiap generasi perangkat keras komputer telah menghasilkan peningkatan kecepatan dan kapasitas serta penurunan harga. Kinerja meningkat dua kali lipat setiap 18 bulan atau lebih hingga sekitar tahun 2005, ketika masalah disipasi daya menyebabkan produsen mulai melipatgandakan jumlah inti CPU daripada kecepatan clock. Harapan saat ini adalah bahwa peningkatan daya di masa mendatang akan berasal dari paralelisme masif—konvergensi yang aneh dengan sifat-sifat otak.

Tentu saja, ada perangkat hitung sebelum komputer elektronik. Mesin otomatis paling awal, yang berasal dari abad ke-17, dibahas di halaman 6. Mesin yang dapat diprogram pertama adalah alat tenun, yang dirancang pada tahun 1805 oleh Joseph Marie Jacquard, yang menggunakan kartu berlubang untuk menyimpan instruksi pola yang akan ditenun. Pada pertengahan abad ke-19, Charles Babbage merancang dua mesin, yang tidak pernah ia selesaikan. Difference Engine dimaksudkan untuk menghitung tabel matematika untuk proyek teknik dan ilmiah. Mesin itu akhirnya dibuat dan diperlihatkan berfungsi pada tahun 1991 di Museum Sains di London. Analytical Engine milik Babbage jauh lebih ambisius: mesin itu mencakup memori yang dapat dialamatkan, program yang tersimpan, dan lompatan bersyarat dan merupakan artefak pertama yang mampu melakukan komputasi universal. Rekan kerja Babbage, Ada Lovelace, putri penyair Lord Byron, mungkin adalah programmer pertama di dunia. Ia menulis program untuk Analytical Engine yang belum selesai dan bahkan berspekulasi bahwa mesin itu dapat bermain catur atau menggubah musik.

AI juga berutang budi pada sisi perangkat lunak ilmu komputer, yang telah menyediakan sistem operasi, bahasa pemrograman, dan alat yang dibutuhkan untuk menulis

program modern (dan makalah tentangnya). Namun, ini adalah satu bidang yang utangnya telah dilunasi: pekerjaan di bidang AI telah memelopori banyak ide yang telah kembali ke ilmu komputer arus utama, termasuk pembagian waktu, penerjemah interaktif, komputer pribadi dengan jendela dan tetikus, lingkungan pengembangan cepat, tipe data daftar tertaut, manajemen penyimpanan otomatis, dan konsep utama pemrograman simbolik, fungsional, deklaratif, dan berorientasi objek.

Teori Kendali dan Sibernetika

• Bagaimana artefak dapat beroperasi di bawah kendalinya sendiri?

Ktesibios dari Alexandria (sekitar 250 SM) membuat mesin pengendali mandiri pertama: jam air dengan pengatur yang mempertahankan laju aliran konstan. Penemuan ini mengubah definisi tentang apa yang dapat dilakukan artefak. Sebelumnya, hanya makhluk hidup yang dapat mengubah perilakunya sebagai respons terhadap perubahan lingkungan. Contoh lain dari sistem kendali umpan balik yang mengatur diri sendiri meliputi pengatur mesin uap, yang diciptakan oleh James Watt, dan termostat, yang diciptakan oleh Cornelis Drebbel, yang juga menemukan kapal selam. Teori matematika tentang sistem umpan balik yang stabil dikembangkan pada abad ke-19.

Tokoh utama dalam penciptaan apa yang sekarang disebut teori kendali adalah Norbert Wiener. Wiener adalah seorang matematikawan brilian yang bekerja dengan Bertrand Russel, antara lain, sebelum mengembangkan minat dalam sistem kontrol biologis dan mekanis dan hubungannya dengan kognisi. Seperti Craik (yang juga menggunakan sistem kontrol sebagai model psikologis), Wiener dan koleganya Arturo Rosenblueth dan Julian Bigelow menantang ortodoksi behavioris. Mereka memandang perilaku bertujuan sebagai sesuatu yang muncul dari mekanisme pengaturan yang mencoba meminimalkan "kesalahan"—perbedaan antara keadaan saat ini dan keadaan tujuan.

Pada akhir 1940an, Wiener, bersama dengan Warren McCulloch, Walter Pitts, dan John von Neumann, menyelenggarakan serangkaian konferensi berpengaruh yang mengeksplorasi model matematika dan komputasional baru dari kognisi. Buku Wiener *Cybernetics* menjadi buku terlaris dan menyadarkan masyarakat akan kemungkinan adanya mesin dengan kecerdasan buatan. Sementara itu, di Inggris, W. Ross Ashby memelopori ide serupa. Ashby, Alan Turing, Grey Walter, dan yang lainnya membentuk Ratio Club untuk "mereka yang memiliki ide-ide Wiener sebelum buku Wiener muncul." *Design for a Brain* karya Ashby menguraikan idenya bahwa kecerdasan dapat diciptakan dengan menggunakan perangkat homeostatis yang berisi putaran umpan balik yang sesuai untuk mencapai perilaku adaptif yang stabil.

Teori kontrol modern, khususnya cabang yang dikenal sebagai kontrol optimal stokastik, memiliki tujuan untuk merancang sistem yang memaksimalkan fungsi objektif dari waktu ke waktu. Hal ini kurang lebih sesuai dengan pandangan kita tentang AI: merancang sistem yang berperilaku optimal. Lalu, mengapa AI dan teori kontrol merupakan dua bidang yang berbeda, meskipun para pendirinya memiliki hubungan yang erat? Jawabannya terletak pada hubungan yang erat antara teknik matematika yang familier bagi para peserta dan serangkaian masalah terkait yang tercakup dalam setiap pandangan dunia. Kalkulus dan

aljabar matriks, alat teori kontrol, menghasilkan sistem yang dapat dijelaskan oleh serangkaian variabel kontinu yang tetap, sedangkan AI didirikan sebagian sebagai cara untuk melepaskan diri dari keterbatasan yang dirasakan ini. Alat inferensi logis dan komputasi memungkinkan peneliti AI untuk mempertimbangkan masalah seperti bahasa, visi, dan perencanaan yang sepenuhnya berada di luar lingkup teori kontrol.

Linguistik

• Bagaimana bahasa berhubungan dengan pikiran?

Pada tahun 1957, B. F. Skinner menerbitkan Verbal Behavior. Buku ini merupakan catatan komprehensif dan terperinci tentang pendekatan behavioris terhadap pembelajaran bahasa, yang ditulis oleh pakar terkemuka di bidang tersebut. Namun anehnya, ulasan buku tersebut menjadi sama terkenalnya dengan buku itu sendiri, dan hampir menghilangkan minat terhadap behaviorisme.

Penulis ulasan tersebut adalah ahli bahasa Noam Chomsky, yang baru saja menerbitkan buku tentang teorinya sendiri, *Syntactic Structures*. Chomsky menunjukkan bahwa teori behavioris tidak membahas gagasan kreativitas dalam bahasa—teori tersebut tidak menjelaskan bagaimana seorang anak dapat memahami dan membuat kalimat yang belum pernah didengarnya sebelumnya. Teori Chomsky—berdasarkan model sintaksis yang berasal dari ahli bahasa India Panini (sekitar 350 SM)—dapat menjelaskan hal ini, dan tidak seperti teori sebelumnya, teori tersebut cukup formal sehingga pada prinsipnya dapat diprogram.

Linguistik modern dan AI, kemudian, "lahir" pada waktu yang hampir bersamaan, dan tumbuh bersama, berpotongan dalam bidang hibrida yang disebut linguistik komputasional atau pemrosesan bahasa alami. Masalah pemahaman bahasa segera berubah menjadi jauh lebih rumit daripada yang terlihat pada tahun 1957.

Memahami bahasa membutuhkan pemahaman tentang pokok bahasan dan konteksnya, bukan hanya pemahaman tentang struktur kalimat. Ini mungkin tampak jelas, tetapi tidak banyak dihargai sampai tahun 1960an. Sebagian besar pekerjaan awal dalam representasi pengetahuan (studi tentang cara menempatkan pengetahuan ke dalam bentuk yang dapat dipikirkan oleh komputer) dikaitkan dengan bahasa dan diinformasikan oleh penelitian dalam linguistik, yang pada gilirannya terhubung dengan pekerjaan selama beberapa dekade pada analisis filosofis bahasa.

1.3 SEJARAH KECERDASAN BUATAN

Dengan materi latar belakang yang telah kita miliki, kita siap untuk membahas perkembangan AI itu sendiri.

Masa Awal Kecerdasan Buatan (1943-1955)

Karya pertama yang kini secara umum dikenal sebagai AI dilakukan oleh Warren McCulloch dan Walter Pitts. Mereka menggunakan tiga sumber: pengetahuan tentang fisiologi dasar dan fungsi neuron di otak; analisis formal logika proposisional yang dikemukakan oleh Russell dan Whitehead; dan teori komputasi Turing. Mereka mengusulkan model neuron buatan yang setiap neuronnya dicirikan sebagai "aktif" atau "nonaktif", dengan peralihan ke

"aktif" terjadi sebagai respons terhadap rangsangan oleh sejumlah neuron tetangga yang memadai.

Keadaan neuron dipahami sebagai "secara faktual setara dengan proposisi yang mengusulkan stimulus yang memadai." Mereka menunjukkan, misalnya, bahwa fungsi komputasi apa pun dapat dihitung oleh beberapa jaringan neuron yang terhubung, dan bahwa semua konektif logis (dan, atau, tidak, dll.) dapat diimplementasikan oleh struktur jaringan sederhana. McCulloch dan Pitts juga menyarankan bahwa jaringan yang didefinisikan dengan tepat dapat belajar. Donald Hebb menunjukkan aturan pembaruan sederhana untuk memodifikasi kekuatan koneksi antara neuron. Aturannya, yang sekarang disebut pembelajaran Hebbian, tetap menjadi model yang berpengaruh hingga hari ini. Dua mahasiswa sarjana di Harvard, Marvin Minsky dan Dean Edmonds, membangun komputer jaringan saraf pertama pada tahun 1950. SNARC, demikian sebutannya, menggunakan 3000 tabung vakum dan mekanisme pilot otomatis surplus dari pembom B-24 untuk mensimulasikan jaringan 40 neuron. Kemudian, di Princeton, Minsky mempelajari komputasi universal dalam jaringan saraf. Komite Ph.D.-nya skeptis tentang apakah pekerjaan semacam ini harus dianggap matematika, tetapi von Neumann dilaporkan mengatakan, "Jika tidak sekarang, suatu hari nanti akan dianggap matematika." Minsky kemudian membuktikan teorema-teorema berpengaruh yang menunjukkan keterbatasan penelitian jaringan saraf.

Ada sejumlah contoh awal karya yang dapat dikarakterisasikan sebagai AI, tetapi visi Alan Turing mungkin yang paling berpengaruh. Ia memberikan kuliah tentang topik tersebut sejak tahun 1947 di London Mathematical Society dan mengartikulasikan agenda yang persuasif dalam artikelnya tahun 1950 *“Computing Machinery and Intelligence.”* Di sana, ia memperkenalkan Turing Test, pembelajaran mesin, algoritma genetik, dan pembelajaran penguatan. Ia mengusulkan gagasan Program Anak, dengan menjelaskan “Daripada mencoba membuat program untuk mensimulasikan pikiran orang dewasa, mengapa tidak mencoba membuat program yang mensimulasikan pikiran anak?”

Kelahiran Kecerdasan Buatan (1956)

Princeton adalah rumah bagi tokoh berpengaruh lainnya dalam AI, John McCarthy. Setelah menerima gelar doktor di sana pada tahun 1951 dan bekerja selama dua tahun sebagai instruktur, McCarthy pindah ke Stanford dan kemudian ke Dartmouth College, yang kemudian menjadi tempat lahirnya bidang tersebut secara resmi. McCarthy meyakinkan Minsky, Claude Shannon, dan Nathaniel Rochester untuk membantunya menyatukan para peneliti AS yang tertarik pada teori automata, jaringan saraf, dan studi kecerdasan. Mereka menyelenggarakan lokakarya dua bulan di Dartmouth pada musim panas tahun 1956. Proposal tersebut menyatakan:

> Kami mengusulkan agar studi kecerdasan buatan selama 2 bulan yang melibatkan 10 orang dilakukan selama musim panas tahun 1956 di Dartmouth College di Hanover, New Hampshire. Penelitian ini akan dilanjutkan berdasarkan dugaan bahwa setiap aspek pembelajaran atau fitur kecerdasan lainnya pada prinsipnya dapat dijelaskan dengan sangat tepat sehingga mesin dapat dibuat untuk menirunya. Upaya akan dilakukan untuk menemukan cara membuat mesin menggunakan bahasa, membentuk

abstraksi dan konsep, memecahkan berbagai jenis masalah yang sekarang hanya diperuntukkan bagi manusia, dan meningkatkan kemampuan mereka sendiri. Kami pikir kemajuan yang signifikan dapat dicapai dalam satu atau beberapa masalah ini jika sekelompok ilmuwan yang dipilih dengan cermat mengerjakannya bersama-sama selama musim panas.

Ada 10 peserta secara keseluruhan, termasuk Trenchard More dari Princeton, Arthur Samuel dari IBM, dan Ray Solomonoff serta Oliver Selfridge dari MIT. Dua peneliti dari Carnegie Tech, Allen Newell dan Herbert Simon, justru mencuri perhatian. Meskipun yang lain memiliki ide dan dalam beberapa kasus program untuk aplikasi tertentu seperti pemeriksa, Newell dan Simon sudah memiliki program penalaran, *Logic Theorist* (LT), yang menurut Simon, "Kami telah menciptakan program komputer yang mampu berpikir non-numerik, dan dengan demikian memecahkan masalah pikiran-tubuh yang sudah lama ada."

Segera setelah lokakarya, program tersebut mampu membuktikan sebagian besar teorema dalam Bab 2 *Principia Mathematica* karya Russell dan Whitehead. Russell dilaporkan sangat senang ketika Simon menunjukkan kepadanya bahwa program tersebut telah menghasilkan pembuktian untuk satu teorema yang lebih pendek daripada yang ada di Principia. Para editor *Journal of Symbolic Logic* kurang terkesan; mereka menolak makalah yang ditulis bersama oleh Newell, Simon, dan Logic Theorist. Lokakarya Dartmouth tidak menghasilkan terobosan baru, tetapi memperkenalkan semua tokoh utama satu sama lain.

Selama 20 tahun berikutnya, bidang tersebut akan didominasi oleh orang-orang ini dan mahasiswa serta kolega mereka di MIT, CMU, Stanford, dan IBM. Melihat proposal untuk lokakarya Dartmouth, kita dapat melihat mengapa AI perlu menjadi bidang yang terpisah. Mengapa semua pekerjaan yang dilakukan dalam AI tidak dapat dilakukan dengan nama teori kontrol atau penelitian operasi atau teori keputusan, yang, bagaimanapun, memiliki tujuan yang sama dengan AI? Atau mengapa AI bukan cabang matematika? Jawaban pertama adalah bahwa AI sejak awal merangkul gagasan menduplikasi kemampuan manusia seperti kreativitas, peningkatan diri, dan penggunaan bahasa.

Tidak ada bidang lain yang membahas masalah ini. Jawaban kedua adalah metodologi. AI adalah satu-satunya dari bidang-bidang ini yang jelas merupakan cabang ilmu komputer (meskipun penelitian operasi berbagi penekanan pada simulasi komputer), dan AI adalah satu-satunya bidang yang mencoba membangun mesin yang akan berfungsi secara otonom dalam lingkungan yang kompleks dan berubah.

Antusiasme Awal, Harapan Besar (1952–1969)

Tahun-tahun awal AI penuh dengan keberhasilan—dengan cara yang terbatas. Mengingat komputer dan alat pemrograman primitif pada saat itu dan fakta bahwa hanya beberapa tahun sebelumnya komputer dianggap sebagai sesuatu yang dapat melakukan aritmatika dan tidak lebih, sungguh mengherankan setiap kali komputer melakukan sesuatu yang sedikit pintar.

Lembaga intelektual, pada umumnya, lebih suka percaya bahwa "mesin tidak akan pernah dapat melakukan X." Peneliti AI secara alami menanggapi dengan mendemonstrasikan

satu X demi satu. John McCarthy menyebut periode ini sebagai era "Lihat, Bu, tidak ada tangan!"

Keberhasilan awal Newell dan Simon diikuti dengan General Problem Solver, atau GPS. Tidak seperti *Logic Theorist*, program ini dirancang sejak awal untuk meniru protokol pemecahan masalah manusia. Dalam kelas teka-teki terbatas yang dapat ditanganinya, ternyata urutan program yang mempertimbangkan subtujuan dan tindakan yang mungkin mirip dengan urutan manusia dalam mendekati masalah yang sama.

Jadi, GPS mungkin merupakan program pertama yang mewujudkan pendekatan "berpikir secara manusiawi". Keberhasilan GPS dan program-program berikutnya sebagai model kognisi membuat Newell dan Simon merumuskan hipotesis sistem simbol fisik yang terkenal, yang menyatakan bahwa "sistem simbol fisik memiliki sarana yang diperlukan dan cukup untuk tindakan cerdas umum." Yang mereka maksud adalah bahwa sistem apa pun (manusia atau mesin) yang menunjukkan kecerdasan harus beroperasi dengan memanipulasi struktur data yang terdiri dari simbol-simbol.

Kita akan melihat nanti bahwa hipotesis ini telah ditentang dari banyak arah. Di IBM, Nathaniel Rochester dan rekan-rekannya menghasilkan beberapa program AI pertama. Herbert Gelernter membuat *Geometry Theorem Prover*, yang mampu membuktikan teorema-teorema yang dianggap rumit oleh banyak siswa matematika. Dimulai pada tahun 1952, Arthur Samuel menulis serangkaian program untuk catur (draft) yang akhirnya berhasil dimainkan pada tingkat amatir yang tinggi.

Dalam perjalanannya, ia membantah gagasan bahwa komputer hanya dapat melakukan apa yang diperintahkan: programnya dengan cepat belajar memainkan permainan yang lebih baik daripada pembuatnya. Program tersebut dipertunjukkan di televisi pada bulan Februari 1956, sehingga menciptakan kesan yang kuat. Seperti Turing, Samuel kesulitan menemukan waktu untuk komputer. Bekerja di malam hari, ia menggunakan mesin-mesin yang masih berada di lantai pengujian di pabrik manufaktur IBM. John McCarthy pindah dari Dartmouth ke MIT dan di sana memberikan tiga kontribusi penting dalam satu tahun bersejarah: 1958. Dalam MIT AI Lab Memo No. 1, McCarthy mendefinisikan bahasa tingkat tinggi Lisp, yang menjadi bahasa pemrograman AI yang dominan selama 30 tahun berikutnya. Dengan Lisp, McCarthy memiliki alat yang ia butuhkan, tetapi akses ke sumber daya komputasi yang langka dan mahal juga merupakan masalah serius. Sebagai tanggapan, ia dan orang lain di MIT menciptakan pembagian waktu. Juga pada tahun 1958, McCarthy menerbitkan sebuah makalah berjudul Programs with Common Sense, di mana ia menggambarkan Advice Taker, sebuah program hipotetis yang dapat dilihat sebagai sistem AI lengkap pertama.

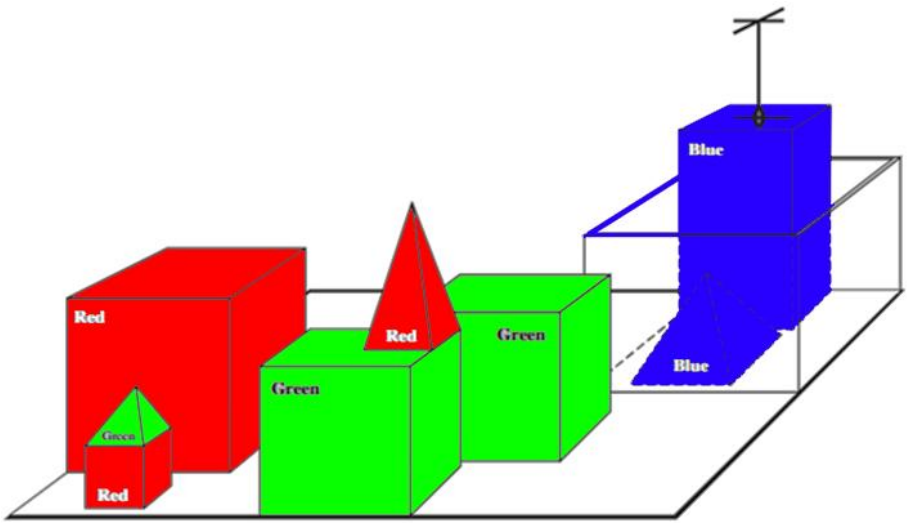
Seperti *Logic Theorist* dan *Geometry Theorem Prover*, program McCarthy dirancang untuk menggunakan pengetahuan untuk mencari solusi untuk masalah. Tetapi tidak seperti yang lain, itu adalah untuk mewujudkan pengetahuan umum tentang dunia. Misalnya, ia menunjukkan bagaimana beberapa aksioma sederhana akan memungkinkan program untuk menghasilkan rencana untuk berkendara ke bandara.

Program ini juga dirancang untuk menerima aksioma baru dalam operasi normal, sehingga memungkinkannya mencapai kompetensi di area baru tanpa diprogram ulang.

Dengan demikian, Advice Taker mewujudkan prinsip-prinsip utama representasi dan penalaran pengetahuan: bahwa penting untuk memiliki representasi formal dan eksplisit dari dunia dan cara kerjanya serta mampu memanipulasi representasi itu dengan proses deduktif. Sungguh luar biasa betapa banyak makalah tahun 1958 yang masih relevan saat ini. Tahun 1958 juga menandai tahun ketika Marvin Minsky pindah ke MIT. Namun, kolaborasi awalnya dengan McCarthy tidak bertahan lama. McCarthy menekankan representasi dan penalaran dalam logika formal, sedangkan Minsky lebih tertarik untuk membuat program bekerja dan akhirnya mengembangkan pandangan anti-logika. Pada tahun 1963, McCarthy memulai lab AI di Stanford.

Rencananya untuk menggunakan logika guna membangun Advice Taker yang terbaik dikembangkan oleh penemuan J. A. Robinson pada tahun 1965 tentang metode resolusi (algoritma pembuktian teorema lengkap untuk logika orde pertama; lihat Bab 9). Pekerjaan di Stanford menekankan metode tujuan umum untuk penalaran logis. Aplikasi logika mencakup sistem tanya jawab dan perencanaan Cordell Green dan proyek robotika Shakey di *Stanford Research Institute* (SRI). Minsky mengawasi serangkaian mahasiswa yang memilih masalah terbatas yang tampaknya memerlukan kecerdasan untuk menyelesaikannya. Domain terbatas ini kemudian dikenal sebagai dunia mikro. Program SAINT James Slagle mampu memecahkan masalah integrasi kalkulus bentuk tertutup yang umum ditemukan pada mata kuliah tahun pertama di perguruan tinggi. Program ANALOGY Tom Evans memecahkan masalah analogi geometri yang muncul dalam tes IQ. Program STUDENT Daniel Bobrow memecahkan masalah cerita aljabar, seperti berikut:

Jika jumlah pelanggan yang diperoleh Tom adalah dua kali kuadrat dari 20 persen jumlah iklan yang dijalankannya, dan jumlah iklan yang dijalankannya adalah 45, berapakah jumlah pelanggan yang diperoleh Tom?



Gambar 1.4 Sebuah adegan dari dunia balok. SHRDLU baru saja menyelesaikan perintah "Temukan balok yang lebih tinggi dari balok yang Anda pegang dan masukkan ke dalam kotak."

Dunia mikro yang paling terkenal adalah dunia balok, yang terdiri dari sekumpulan balok padat yang diletakkan di atas meja (atau lebih sering, simulasi meja), seperti yang ditunjukkan pada Gambar 1.4. Tugas umum di dunia ini adalah menata ulang balok-balok dengan cara tertentu, menggunakan tangan robot yang dapat mengambil satu balok pada satu waktu.

Pekerjaan awal yang membangun jaringan saraf McCulloch dan Pitts juga berkembang pesat. Pekerjaan Winograd dan Cowan menunjukkan bagaimana sejumlah besar elemen secara kolektif dapat mewakili konsep individual, dengan peningkatan yang sesuai dalam ketahanan dan paralelisme. Metode pembelajaran Hebb disempurnakan oleh Bernie Widrow, yang menyebut jaringannya adalines, dan oleh Frank Rosenblatt dengan perceptrons-nya. Teorema konvergensi perceptron mengatakan bahwa algoritme pembelajaran dapat menyesuaikan kekuatan koneksi dari sebuah perceptron untuk mencocokkan data input apa pun, asalkan kecocokan tersebut ada. Topik-topik ini dibahas dalam Bab 20.

Dosis Realitas (1966–1973)

Sejak awal, para peneliti AI tidak malu membuat prediksi tentang keberhasilan mereka yang akan datang. Pernyataan berikut oleh Herbert Simon pada tahun 1957 sering dikutip:

> Bukan tujuan saya untuk mengejutkan atau mengagetkan Anda—tetapi cara paling sederhana yang dapat saya rangkum adalah dengan mengatakan bahwa sekarang ada di dunia mesin yang berpikir, yang belajar, dan yang menciptakan. Terlebih lagi, kemampuan mereka untuk melakukan hal-hal tersebut akan meningkat dengan cepat hingga—di masa depan yang dapat terlihat—jangkauan masalah yang dapat mereka tangani akan sama luasnya dengan jangkauan penerapan pikiran manusia.

Istilah seperti "masa depan yang terlihat" dapat diartikan dengan berbagai cara, tetapi Simon juga membuat prediksi yang lebih konkret: bahwa dalam waktu 10 tahun komputer akan menjadi juara catur, dan teorema matematika yang signifikan akan dibuktikan oleh mesin. Prediksi ini menjadi kenyataan (atau hampir benar) dalam waktu 40 tahun, bukan 10 tahun. Rasa percaya diri Simon yang berlebihan disebabkan oleh kinerja sistem AI awal yang menjanjikan pada contoh-contoh sederhana. Namun, dalam hampir semua kasus, sistem awal ini ternyata gagal total saat dicoba pada pilihan masalah yang lebih luas dan pada masalah yang lebih sulit.

Kesulitan jenis pertama muncul karena sebagian besar program awal tidak mengetahui apa pun tentang pokok bahasannya; mereka berhasil melalui manipulasi sintaksis yang sederhana. Kisah yang khas terjadi pada upaya penerjemahan mesin awal, yang didanai dengan murah hati oleh Dewan Riset Nasional AS dalam upaya untuk mempercepat penerjemahan makalah ilmiah Rusia setelah peluncuran Sputnik pada tahun 1957.

Awalnya dianggap bahwa transformasi sintaksis sederhana berdasarkan tata bahasa Rusia dan Inggris, dan penggantian kata dari kamus elektronik, akan cukup untuk mempertahankan makna kalimat yang tepat. Faktanya adalah bahwa penerjemahan yang akurat membutuhkan pengetahuan latar belakang untuk mengatasi ambiguitas dan menetapkan isi kalimat. Penerjemahan ulang yang terkenal dari "roh itu bersedia tetapi dagingnya lemah" menjadi "vodka itu baik tetapi dagingnya busuk" menggambarkan kesulitan yang dihadapi.

Pada tahun 1966, sebuah laporan oleh komite penasihat menemukan bahwa "belum ada penerjemahan mesin untuk teks ilmiah umum, dan tidak ada yang dalam prospek segera."

Semua pendanaan pemerintah AS untuk proyek penerjemahan akademis dibatalkan. Saat ini, penerjemahan mesin adalah alat yang tidak sempurna tetapi banyak digunakan untuk dokumen teknis, komersial, pemerintah, dan Internet.

Kesulitan jenis kedua adalah banyaknya masalah yang coba dipecahkan oleh AI. Sebagian besar program AI awal memecahkan masalah dengan mencoba berbagai kombinasi langkah hingga solusinya ditemukan. Strategi ini awalnya berhasil karena mikrodunia berisi sangat sedikit objek dan karenanya sangat sedikit tindakan yang mungkin dan urutan solusinya sangat singkat. Sebelum teori kompleksitas komputasional dikembangkan, secara luas dianggap bahwa "meningkatkan skala" ke masalah yang lebih besar hanyalah masalah perangkat keras yang lebih cepat dan memori yang lebih besar.

Optimisme yang menyertai pengembangan pembuktian teorema resolusi, misalnya, segera mereda ketika para peneliti gagal membuktikan teorema yang melibatkan lebih dari beberapa lusin fakta. Fakta bahwa suatu program dapat menemukan solusi pada prinsipnya tidak berarti bahwa program tersebut berisi salah satu mekanisme yang diperlukan untuk menemukannya dalam praktik. Ilusi daya komputasi tanpa batas tidak terbatas pada program pemecahan masalah. Percobaan awal dalam evolusi mesin (sekarang disebut algoritma genetika) didasarkan pada keyakinan yang tidak diragukan lagi benar bahwa dengan membuat serangkaian mutasi kecil yang tepat pada program kode mesin, seseorang dapat menghasilkan program dengan kinerja yang baik untuk tugas tertentu. Idenya adalah untuk mencoba mutasi acak dengan proses seleksi untuk mempertahankan mutasi yang tampaknya berguna. Meskipun menghabiskan ribuan jam waktu CPU, hampir tidak ada kemajuan yang ditunjukkan. Algoritma genetika modern menggunakan representasi yang lebih baik dan telah menunjukkan lebih banyak keberhasilan.

Kegagalan untuk mengatasi "ledakan kombinatorial" merupakan salah satu kritik utama AI yang dimuat dalam laporan Lighthill, yang menjadi dasar keputusan pemerintah Inggris untuk mengakhiri dukungan terhadap penelitian AI di semua universitas kecuali dua. (Tradisi lisan melukiskan gambaran yang agak berbeda dan lebih berwarna, dengan ambisi politik dan permusuhan pribadi yang uraiannya tidak penting.)

Kesulitan ketiga muncul karena beberapa keterbatasan mendasar pada struktur dasar yang digunakan untuk menghasilkan perilaku cerdas. Misalnya, buku Minsky dan Papert, Perceptrons membuktikan bahwa, meskipun perceptron (bentuk sederhana jaringan saraf) dapat ditunjukkan untuk mempelajari apa pun yang dapat mereka wakili, mereka hanya dapat mewakili sangat sedikit. Secara khusus, perceptron dua masukan (dibatasi menjadi lebih sederhana daripada bentuk yang awalnya dipelajari Rosenblatt) tidak dapat dilatih untuk mengenali saat kedua masukannya berbeda. Meskipun hasil mereka tidak berlaku untuk jaringan berlapis yang lebih kompleks, pendanaan penelitian untuk penelitian jaringan saraf segera menyusut hingga hampir tidak ada. Ironisnya, algoritma pembelajaran back-propagation baru untuk jaringan berlapis yang menyebabkan kebangkitan besar dalam penelitian jaringan saraf pada akhir 1980an sebenarnya pertama kali ditemukan pada tahun 1969.

Sistem Berbasis Pengetahuan: Kunci Menuju Kekuatan? (1969–1979)

Gambaran pemecahan masalah yang muncul selama dekade pertama penelitian AI adalah mekanisme pencarian tujuan umum yang mencoba merangkai langkah-langkah penalaran dasar untuk menemukan solusi lengkap. Pendekatan semacam itu disebut metode lemah karena, meskipun umum, pendekatan tersebut tidak dapat ditingkatkan ke contoh masalah yang besar atau sulit. Alternatif untuk metode lemah adalah menggunakan pengetahuan khusus domain yang lebih kuat yang memungkinkan langkah-langkah penalaran yang lebih besar dan dapat lebih mudah menangani kasus-kasus yang biasanya terjadi di area keahlian yang sempit. Orang mungkin mengatakan bahwa untuk memecahkan masalah yang sulit, Anda harus hampir mengetahui jawabannya.

Program DENDRAL merupakan contoh awal pendekatan ini. Program ini dikembangkan di Stanford, tempat Ed Feigenbaum (mantan murid Herbert Simon), Bruce Buchanan (seorang filsuf yang beralih menjadi ilmuwan komputer), dan Joshua Lederberg (ahli genetika peraih Nobel) bekerja sama untuk memecahkan masalah dalam menyimpulkan struktur molekul dari informasi yang diberikan oleh spektrometer massa. Masukan ke program tersebut terdiri dari rumus dasar molekul (misalnya, $C_6H_{13}NO_2$) dan spektrum massa yang memberikan massa berbagai fragmen molekul yang dihasilkan ketika molekul tersebut dibombardir oleh berkas elektron. Misalnya, spektrum massa mungkin mengandung puncak pada m = 15, yang sesuai dengan massa fragmen metil (CH_3). Versi program yang naif menghasilkan semua kemungkinan struktur yang konsisten dengan rumus, lalu memperkirakan spektrum massa apa yang akan diamati untuk masing-masing struktur, dengan membandingkannya dengan spektrum aktual.

Seperti yang mungkin diharapkan, hal ini sulit dilakukan bahkan untuk molekul berukuran sedang. Para peneliti DENDRAL berkonsultasi dengan ahli kimia analitis dan menemukan bahwa mereka bekerja dengan mencari pola puncak yang terkenal dalam spektrum yang menunjukkan substruktur umum dalam molekul. Misalnya, aturan berikut digunakan untuk mengenali subkelompok keton (C=O) (yang beratnya 28):

jika ada dua puncak pada x_1 dan x_2 sehingga

a) $x_1 + x_2 = M + 28$ (M adalah massa seluruh molekul);
b) $x_1 - 28$ adalah puncak yang tinggi;
c) $x_2 - 28$ adalah puncak yang tinggi;
d) Setidaknya satu dari x_1 dan x_2 tinggi.

maka ada subgrup keton

Mengenali bahwa molekul tersebut mengandung substruktur tertentu mengurangi jumlah kandidat yang mungkin secara drastis. DENDRAL sangat kuat karena Semua pengetahuan teoritis yang relevan untuk memecahkan masalah ini telah dipetakan dari bentuk umumnya dalam [komponen prediksi spektrum] (“prinsip pertama”) ke bentuk khusus yang efisien (“resep buku resep”).

Pentingnya DENDRAL adalah bahwa itu adalah sistem intensif pengetahuan pertama yang berhasil: keahliannya berasal dari sejumlah besar aturan tujuan khusus. Sistem

selanjutnya juga menggabungkan tema utama pendekatan Advice Taker McCarthy—pemisahan yang jelas antara pengetahuan (dalam bentuk aturan) dari komponen penalaran. Dengan mengingat pelajaran ini, Feigenbaum dan yang lainnya di Stanford memulai Proyek Pemrograman Heuristik (HPP) untuk menyelidiki sejauh mana metodologi baru sistem pakar dapat diterapkan pada area keahlian manusia lainnya. Upaya besar berikutnya adalah di area diagnosis medis. Feigenbaum, Buchanan, dan Dr. Edward Shortliffe mengembangkan MYCIN untuk mendiagnosis infeksi darah. Dengan sekitar 450 aturan, MYCIN mampu bekerja sebaik beberapa pakar, dan jauh lebih baik daripada dokter junior. MYCIN juga mengandung dua perbedaan utama dari DENDRAL.

Pertama, tidak seperti aturan DENDRAL, tidak ada model teoritis umum yang dapat digunakan untuk menyimpulkan aturan MYCIN. Aturan tersebut harus diperoleh dari wawancara ekstensif dengan para pakar, yang kemudian memperolehnya dari buku teks, pakar lain, dan pengalaman langsung kasus. Kedua, aturan tersebut harus mencerminkan ketidakpastian yang terkait dengan pengetahuan medis. MYCIN memasukkan kalkulus ketidakpastian yang disebut faktor kepastian (lihat Bab 14), yang tampaknya (pada saat itu) sangat sesuai dengan cara dokter menilai dampak bukti pada diagnosis. Pentingnya pengetahuan domain juga tampak jelas dalam bidang pemahaman bahasa alami.

Meskipun sistem SHRDLU milik Winograd untuk memahami bahasa alami telah menimbulkan banyak kegembiraan, ketergantungannya pada analisis sintaksis menyebabkan beberapa masalah yang sama seperti yang terjadi pada karya terjemahan mesin awal. Sistem ini mampu mengatasi ambiguitas dan memahami referensi kata ganti, tetapi ini terutama karena sistem ini dirancang khusus untuk satu area—dunia blok. Beberapa peneliti, termasuk Eugene Charniak, sesama mahasiswa pascasarjana Winograd di MIT, menyarankan bahwa pemahaman bahasa yang kuat akan membutuhkan pengetahuan umum tentang dunia dan metode umum untuk menggunakan pengetahuan itu. Di Yale, ahli bahasa yang beralih menjadi peneliti AI Roger Schank menekankan hal ini, dengan mengklaim, "Tidak ada yang namanya sintaksis," yang membuat banyak ahli bahasa kesal tetapi berhasil memulai diskusi yang bermanfaat. Schank dan murid-muridnya membuat serangkaian program yang semuanya memiliki tugas untuk memahami bahasa alami. Akan tetapi, penekanannya tidak terlalu pada bahasa itu sendiri, tetapi lebih pada masalah representasi dan penalaran dengan pengetahuan yang dibutuhkan untuk memahami bahasa.

Masalah-masalah tersebut meliputi representasi situasi stereotip, mendeskripsikan organisasi memori manusia, dan pemahaman rencana dan tujuan. Pertumbuhan aplikasi yang meluas pada masalah dunia nyata menyebabkan peningkatan permintaan skema representasi pengetahuan yang dapat diterapkan. Sejumlah besar bahasa representasi dan penalaran yang berbeda dikembangkan. Beberapa didasarkan pada logika—misalnya, bahasa Prolog menjadi populer di Eropa, dan keluarga PLANNER di Amerika Serikat. Sementara itu, para ahli lain, mengikuti gagasan Minsky tentang kerangka (frame), mengadopsi pendekatan yang lebih terstruktur, dengan mengumpulkan fakta-fakta tentang jenis objek dan peristiwa tertentu dan mengatur jenis-jenis tersebut ke dalam hierarki taksonomi besar yang serupa dengan taksonomi biologi.

AI Menjadi Sebuah Industri (1980–sekarang)

Sistem pakar komersial pertama yang sukses, R1, mulai beroperasi di Digital Equipment Corporation. Program tersebut membantu mengonfigurasi pesanan untuk sistem komputer baru; pada tahun 1986, program tersebut menghemat biaya perusahaan sekitar Rp 400 miliar per tahun. Pada tahun 1988, kelompok AI DEC telah menerapkan 40 sistem pakar, dan masih banyak lagi yang akan segera diterapkan. DuPont telah menggunakan 100 sistem dan mengembangkan 500 sistem, sehingga menghemat biaya sekitar Rp 100 miliar per tahun. Hampir setiap perusahaan besar di AS memiliki kelompok AI sendiri dan menggunakan atau menyelidiki sistem pakar.

Pada tahun 1981, Jepang mengumumkan proyek "Generasi Kelima", sebuah rencana 10 tahun untuk membangun komputer cerdas yang menjalankan Prolog. Sebagai tanggapan, Amerika Serikat membentuk *Microelectronics and Computer Technology Corporation* (MCC) sebagai konsorsium penelitian yang dirancang untuk memastikan daya saing nasional. Dalam kedua kasus tersebut, AI merupakan bagian dari upaya yang luas, termasuk desain chip dan penelitian antarmuka manusia. Di Inggris, laporan Alvey mengembalikan pendanaan yang dipotong oleh laporan Lighthill. Namun, di ketiga negara tersebut, proyek-proyek tersebut tidak pernah mencapai tujuan ambisius mereka.

Secara keseluruhan, industri AI berkembang pesat dari beberapa juta dolar pada tahun 1980 menjadi miliaran dolar pada tahun 1988, termasuk ratusan perusahaan yang membangun sistem pakar, sistem penglihatan, robot, serta perangkat lunak dan perangkat keras yang dikhususkan untuk tujuan-tujuan ini. Segera setelah itu tibalah periode yang disebut "Musim Dingin AI", di mana banyak perusahaan jatuh di pinggir jalan karena mereka gagal memenuhi janji-janji yang berlebihan.

Kembalinya Jaringan Saraf (1986–sekarang)

Pada pertengahan 1980an, setidaknya empat kelompok berbeda menemukan kembali algoritma pembelajaran back-propagation yang pertama kali ditemukan pada tahun 1969 oleh Bryson dan Ho. Algoritma ini diterapkan pada banyak masalah pembelajaran dalam ilmu komputer dan psikologi, dan penyebaran luas hasil dalam koleksi Parallel Distributed Processing menyebabkan kegembiraan besar. Model-model koneksionis sistem cerdas ini dipandang oleh beberapa orang sebagai pesaing langsung baik untuk model simbolik yang dipromosikan oleh Newell dan Simon maupun pendekatan logika McCarthy dan lainnya.

Mungkin tampak jelas bahwa pada tingkat tertentu manusia memanipulasi simbol—pada kenyataannya, buku Terrence Deacon *The Symbolic Species* menunjukkan bahwa ini adalah karakteristik yang menentukan manusia—tetapi koneksionis yang paling bersemangat mempertanyakan apakah manipulasi simbol memiliki peran penjelasan nyata dalam model kognisi terperinci. Pertanyaan ini masih belum terjawab, tetapi pandangan saat ini adalah bahwa pendekatan koneksionis dan simbolik saling melengkapi, tidak bersaing. Seperti yang terjadi dengan pemisahan AI dan ilmu kognitif, penelitian jaringan saraf modern telah terbagi menjadi dua bidang, satu berkaitan dengan pembuatan arsitektur dan algoritma jaringan yang efektif dan pemahaman sifat matematikanya, yang lainnya berkaitan dengan pemodelan yang cermat terhadap sifat empiris neuron dan kumpulan neuron aktual.

AI Mengadopsi Metode Ilmiah (1987–sekarang)

Beberapa tahun terakhir telah terjadi revolusi baik dalam konten maupun metodologi kerja dalam kecerdasan buatan. Sekarang lebih umum untuk membangun teori yang sudah ada daripada mengusulkan teori yang benar-benar baru, untuk mendasarkan klaim pada teorema yang ketat atau bukti eksperimental yang kuat daripada intuisi, dan untuk menunjukkan relevansi dengan aplikasi dunia nyata daripada contoh mainan. AI didirikan sebagian sebagai pemberontakan terhadap keterbatasan bidang yang sudah ada seperti teori kontrol dan statistik, tetapi sekarang ia merangkul bidang-bidang tersebut. Seperti yang dikatakan David McAllester:

> Pada periode awal AI, tampaknya masuk akal bahwa bentuk-bentuk baru komputasi simbolik, misalnya, bingkai dan jaringan semantik, membuat banyak teori klasik menjadi usang. Hal ini menyebabkan suatu bentuk isolasionisme di mana AI sebagian besar terpisah dari ilmu komputer lainnya.
>
> Isolasionisme ini saat ini sedang ditinggalkan. Ada pengakuan bahwa pembelajaran mesin tidak boleh dipisahkan dari teori informasi, bahwa penalaran yang tidak pasti tidak boleh dipisahkan dari pemodelan stokastik, bahwa pencarian tidak boleh dipisahkan dari pengoptimalan dan kontrol klasik, dan bahwa penalaran otomatis tidak boleh dipisahkan dari metode formal dan analisis statis.

Dalam hal metodologi, AI akhirnya benar-benar berada di bawah metode ilmiah. Agar dapat diterima, hipotesis harus tunduk pada eksperimen empiris yang ketat, dan hasilnya harus dianalisis secara statistik untuk kepentingannya. Sekarang dimungkinkan untuk mereplikasi eksperimen dengan menggunakan repositori bersama dari data uji dan kode.

Bidang pengenalan ucapan menggambarkan pola tersebut. Pada tahun 1970an, berbagai macam arsitektur dan pendekatan yang berbeda dicoba. Banyak di antaranya yang agak *ad hoc* dan rapuh, dan hanya ditunjukkan pada beberapa contoh yang dipilih secara khusus. Dalam beberapa tahun terakhir, pendekatan berdasarkan model Markov tersembunyi (HMM) telah mendominasi bidang tersebut.

Dua aspek HMM relevan. Pertama, pendekatan tersebut didasarkan pada teori matematika yang ketat. Hal ini memungkinkan para peneliti wicara untuk mengembangkan hasil matematika yang telah dikembangkan di bidang lain selama beberapa dekade. Kedua, hasil tersebut dihasilkan melalui proses pelatihan pada korpus besar data wicara nyata. Hal ini memastikan bahwa kinerjanya tangguh, dan dalam uji buta yang ketat, HMM telah meningkatkan skornya secara stabil. Teknologi wicara dan bidang terkait pengenalan karakter tulisan tangan telah melakukan transisi ke aplikasi industri dan konsumen yang tersebar luas. Perhatikan bahwa tidak ada klaim ilmiah bahwa manusia menggunakan HMM untuk mengenali wicara; sebaliknya, HMM menyediakan kerangka matematika untuk memahami masalah dan mendukung klaim teknik bahwa HMM bekerja dengan baik dalam praktik.

Penerjemahan mesin mengikuti jalur yang sama dengan pengenalan wicara. Pada tahun 1950an, ada antusiasme awal untuk pendekatan yang didasarkan pada urutan kata, dengan model yang dipelajari sesuai dengan prinsip teori informasi. Pendekatan itu tidak lagi disukai pada tahun 1960an, tetapi muncul kembali pada akhir tahun 1990an dan sekarang mendominasi bidang tersebut.

Jaringan saraf juga sesuai dengan tren ini. Banyak pekerjaan pada jaringan saraf pada tahun 1980an dilakukan dalam upaya untuk mencari tahu apa yang dapat dilakukan dan

mempelajari bagaimana jaringan saraf berbeda dari teknik "tradisional". Dengan menggunakan metodologi dan kerangka kerja teoritis yang lebih baik, bidang ini sampai pada pemahaman di mana jaringan saraf sekarang dapat dibandingkan dengan teknik yang sesuai dari statistik, pengenalan pola, dan pembelajaran mesin, dan teknik yang paling menjanjikan dapat diterapkan pada setiap aplikasi.

Sebagai hasil dari perkembangan ini, apa yang disebut teknologi penambangan data telah melahirkan industri baru yang kuat. Penalaran Probabilistik dalam Sistem Cerdas karya Judea Pearl menyebabkan penerimaan baru terhadap teori probabilitas dan keputusan dalam AI, menyusul kebangkitan minat yang dilambangkan oleh artikel Peter Cheeseman "*In Defense of Probability*." Formalisme jaringan Bayesian diciptakan untuk memungkinkan representasi yang efisien dari, dan penalaran yang ketat dengan, pengetahuan yang tidak pasti.

Pendekatan ini sebagian besar mengatasi banyak masalah sistem penalaran probabilistik tahun 1960an dan 1970an; sekarang mendominasi penelitian AI pada penalaran tak pasti dan sistem pakar. Pendekatan ini memungkinkan pembelajaran dari pengalaman, dan menggabungkan yang terbaik dari AI klasik dan jaringan saraf. Karya Judea Pearl dan Eric Horvitz dan David Heckerman mempromosikan gagasan sistem pakar normatif: sistem yang bertindak secara rasional menurut hukum teori keputusan dan tidak mencoba meniru langkah-langkah pemikiran pakar manusia. Sistem operasi WindowsTM mencakup beberapa sistem pakar diagnostik normatif untuk mengoreksi masalah. Bab 13 hingga 16 membahas area ini.

Revolusi halus serupa telah terjadi dalam robotika, visi komputer, dan representasi pengetahuan. Pemahaman yang lebih baik tentang masalah dan sifat kompleksitasnya, dikombinasikan dengan kecanggihan matematika yang meningkat, telah menghasilkan agenda penelitian yang dapat diterapkan dan metode yang kuat. Meskipun formalisasi dan spesialisasi yang meningkat menyebabkan bidang-bidang seperti visi dan robotika menjadi agak terisolasi dari AI "arus utama" pada tahun 1990an, tren ini telah berbalik dalam beberapa tahun terakhir karena alat-alat dari pembelajaran mesin khususnya telah terbukti efektif untuk banyak masalah. Proses reintegrasi telah menghasilkan manfaat yang signifikan

Munculnya Agen Cerdas (1995–sekarang)

Mungkin didorong oleh kemajuan dalam memecahkan submasalah AI, para peneliti juga mulai melihat kembali masalah "agen secara keseluruhan". Karya Allen Newell, John Laird, dan Paul Rosenbloom pada SOAR adalah contoh paling terkenal dari arsitektur agen yang lengkap. Salah satu lingkungan terpenting bagi agen cerdas adalah Internet. Sistem AI telah menjadi sangat umum dalam aplikasi berbasis Web sehingga sufiks "-bot" telah memasuki bahasa sehari-hari. Selain itu, teknologi AI mendasari banyak alat Internet, seperti mesin pencari, sistem rekomendasi, dan agregator situs Web.

Salah satu konsekuensi dari upaya membangun agen yang lengkap adalah kesadaran bahwa subbidang AI yang sebelumnya terisolasi mungkin perlu ditata ulang ketika hasilnya akan dihubungkan bersama. Secara khusus, kini banyak yang menyadari bahwa sistem sensorik (penglihatan, sonar, pengenalan suara, dll.) tidak dapat memberikan informasi yang sepenuhnya dapat diandalkan tentang lingkungan. Oleh karena itu, sistem penalaran dan perencanaan harus mampu menangani ketidakpastian. Konsekuensi utama kedua dari

perspektif agen adalah bahwa AI telah ditarik ke dalam kontak yang jauh lebih dekat dengan bidang lain, seperti teori kontrol dan ekonomi, yang juga berurusan dengan agen. Kemajuan terkini dalam kontrol mobil robotik berasal dari campuran pendekatan mulai dari sensor yang lebih baik, integrasi teori kontrol penginderaan, lokalisasi dan pemetaan, serta tingkat perencanaan tingkat tinggi.

Meskipun keberhasilan ini, beberapa pendiri AI yang berpengaruh, termasuk John McCarthy, Marvin Minsky, Nils Nilsson dan Patrick Winston, telah menyatakan ketidakpuasan dengan kemajuan AI. Mereka berpendapat bahwa AI seharusnya tidak terlalu menekankan pada pembuatan versi aplikasi yang terus ditingkatkan yang bagus untuk tugas tertentu, seperti mengemudikan mobil, bermain catur, atau mengenali ucapan. Sebaliknya, mereka percaya bahwa AI seharusnya kembali ke akarnya, yaitu berjuang untuk, dalam kata-kata Simon, "mesin yang berpikir, belajar, dan berkreasi." Mereka menyebut upaya tersebut AI tingkat manusia atau HLAI; simposium pertama mereka diadakan pada tahun 2004. Upaya tersebut akan membutuhkan basis pengetahuan yang sangat besar; membahas dari mana basis pengetahuan ini mungkin berasal.

Gagasan terkait adalah subbidang Kecerdasan Umum Buatan atau AGI, yang mengadakan konferensi pertamanya dan menyelenggarakan Jurnal Kecerdasan Umum Buatan pada tahun 2008. AGI mencari algoritme universal untuk belajar dan bertindak di lingkungan apa pun, dan berakar pada karya Ray Solomonoff, salah satu peserta konferensi Dartmouth asli tahun 1956. Menjamin bahwa apa yang kita buat benar-benar AI yang Ramah juga menjadi perhatian yang akan kita bahas lagi di Bab 26.

Ketersediaan Set Data Yang Sangat Besar (2001–sekarang)

Sepanjang 60 tahun sejarah ilmu komputer, penekanannya adalah pada algoritme sebagai subjek utama studi. Namun, beberapa karya terbaru dalam AI menunjukkan bahwa untuk banyak masalah, lebih masuk akal untuk mengkhawatirkan data dan tidak terlalu pilih-pilih tentang algoritme mana yang akan diterapkan. Hal ini benar karena semakin tersedianya sumber data yang sangat besar: misalnya, triliunan kata dalam bahasa Inggris dan miliaran gambar dari Web; atau miliaran pasangan basa dari sekuens genomik.

Salah satu makalah yang berpengaruh dalam bidang ini adalah karya Yarowsky tentang disambiguasi makna kata: mengingat penggunaan kata "tanaman" dalam sebuah kalimat, apakah itu merujuk pada flora atau pabrik? Pendekatan sebelumnya terhadap masalah ini mengandalkan contoh berlabel manusia yang dikombinasikan dengan algoritme pembelajaran mesin. Yarowsky menunjukkan bahwa tugas tersebut dapat dilakukan, dengan akurasi di atas 96%, tanpa contoh berlabel sama sekali.

Sebaliknya, mengingat korpus teks yang sangat besar yang tidak diberi anotasi dan hanya definisi kamus dari dua makna—"karya, pabrik industri" dan "flora, kehidupan tanaman"—seseorang dapat memberi label contoh dalam korpus, dan dari sana melakukan bootstrap untuk mempelajari pola baru yang membantu memberi label contoh baru. Banko dan Brill menunjukkan bahwa teknik seperti ini berkinerja lebih baik saat jumlah teks yang tersedia berubah dari satu juta kata menjadi satu miliar dan peningkatan kinerja dari penggunaan lebih banyak data melampaui perbedaan apa pun dalam pilihan algoritma;

algoritma biasa-biasa saja dengan 100 juta kata data pelatihan yang tidak berlabel mengungguli algoritma paling terkenal dengan 1 juta kata.

Sebagai contoh lain, Hays dan Efros membahas masalah pengisian lubang pada foto. Misalkan Anda menggunakan Photoshop untuk menutupi mantan teman dari foto grup, tetapi sekarang Anda perlu mengisi area yang ditutupi dengan sesuatu yang cocok dengan latar belakang. Hays dan Efros mendefinisikan sebuah algoritme yang menelusuri kumpulan foto untuk menemukan sesuatu yang cocok. Mereka menemukan kinerja algoritme mereka buruk ketika mereka menggunakan kumpulan yang hanya berisi sepuluh ribu foto, tetapi melampaui ambang batas menjadi kinerja yang sangat baik ketika mereka mengembangkan koleksi tersebut menjadi dua juta foto.

Pekerjaan seperti ini menunjukkan bahwa "kemacetan pengetahuan" dalam AI—masalah tentang cara mengekspresikan semua pengetahuan yang dibutuhkan sistem—dapat dipecahkan dalam banyak aplikasi dengan metode pembelajaran daripada rekayasa pengetahuan yang dikodekan secara manual, asalkan algoritme pembelajaran memiliki cukup data untuk digunakan. Reporter telah memperhatikan lonjakan aplikasi baru dan telah menulis bahwa "Musim Dingin AI" mungkin akan berganti menjadi Musim Semi yang baru. Seperti yang ditulis Kurzweil, “saat ini, ribuan aplikasi AI tertanam dalam infrastruktur setiap industri.”

1.4 KEADAAN SENI TERKINI

Apa yang dapat dilakukan AI saat ini? Jawaban yang ringkas sulit diberikan karena ada begitu banyak aktivitas di begitu banyak subbidang. Berikut ini kami contohkan beberapa aplikasi; yang lain muncul di seluruh buku ini.

Kendaraan robotik: Mobil robotik tanpa pengemudi bernama STANLEY melaju melalui medan kasar gurun Mojave dengan kecepatan 22 mph, menyelesaikan lintasan sepanjang 132 mil sebagai yang pertama memenangkan DARPA Grand Challenge 2005. STANLEY adalah Volkswagen Touareg yang dilengkapi dengan kamera, radar, dan laser rangefinder untuk merasakan lingkungan sekitar dan perangkat lunak bawaan untuk mengendalikan kemudi, pengereman, dan akselerasi. Tahun berikutnya, BOSS dari CMU memenangkan Urban Challenge, mengemudi dengan aman di tengah kemacetan lalu lintas melalui jalan-jalan pangkalan Angkatan Udara yang ditutup, mematuhi peraturan lalu lintas, dan menghindari pejalan kaki serta kendaraan lain.

Pengenalan ucapan: Seorang pelancong yang menelepon United Airlines untuk memesan tiket pesawat dapat melakukan seluruh percakapan yang dipandu oleh sistem pengenalan ucapan dan manajemen dialog otomatis.

Perencanaan dan penjadwalan otonom: Seratus juta mil dari Bumi, program Agen Jarak Jauh NASA menjadi program perencanaan otonom pertama yang mengendalikan penjadwalan operasi untuk wahana antariksa. Agen jarak jauh membuat rencana dari sasaran tingkat tinggi yang ditetapkan dari darat dan memantau pelaksanaan rencana tersebut—mendeteksi, mendiagnosis, dan memulihkan masalah saat terjadi. Program penerus MAPGEN merencanakan operasi harian untuk Mars Exploration Rovers milik NASA, dan MEXAR2

melakukan perencanaan misi—baik perencanaan logistik dan sains—untuk misi Mars Express milik Badan Antariksa Eropa pada tahun 2008.

Permainan catur: DEEP BLUE IBM menjadi program komputer pertama yang mengalahkan juara dunia dalam pertandingan catur saat mengalahkan Garry Kasparov dengan skor 3,5 banding 2,5 dalam pertandingan eksibisi. Kasparov mengatakan bahwa ia merasakan "jenis kecerdasan baru" dalam dirinya. Majalah Newsweek menggambarkan pertandingan itu sebagai "pertahanan terakhir otak." Nilai saham IBM meningkat sebesar Rp 180 triliun. Para juara manusia mempelajari kekalahan Kasparov dan mampu memenangkan beberapa pertandingan pada tahun-tahun berikutnya, tetapi pertandingan manusia-komputer terkini dimenangkan secara meyakinkan oleh komputer.

Pemberantasan spam: Setiap hari, algoritma pembelajaran mengklasifikasikan lebih dari satu miliar pesan sebagai spam, sehingga penerima tidak perlu membuang waktu menghapus apa yang, bagi banyak pengguna, dapat mencakup 80% atau 90% dari semua pesan, jika tidak diklasifikasikan oleh algoritma. Karena para pengirim spam terus memperbarui taktik mereka, pendekatan terprogram statis sulit untuk mengimbanginya, dan algoritma pembelajaran bekerja paling baik.

Perencanaan logistik: Selama krisis Teluk Persia tahun 1991, pasukan AS menggunakan Alat Analisis dan Perencanaan Ulang Dinamis, DART, untuk melakukan perencanaan dan penjadwalan logistik otomatis untuk transportasi. Ini melibatkan hingga 50.000 kendaraan, kargo, dan orang pada satu waktu, dan harus memperhitungkan titik awal, tujuan, rute, dan penyelesaian konflik di antara semua parameter. Teknik perencanaan AI menghasilkan rencana dalam hitungan jam yang akan memakan waktu berminggu-minggu dengan metode lama. *Defense Advanced Research Project Agency* (DARPA) menyatakan bahwa aplikasi tunggal ini lebih dari sekadar membayar kembali investasi DARPA selama 30 tahun dalam AI.

Robotika: iRobot Corporation telah menjual lebih dari dua juta penyedot debu robotik Roomba untuk penggunaan di rumah. Perusahaan tersebut juga menggunakan PackBot yang lebih tangguh di Irak dan Afghanistan, yang digunakan untuk menangani bahan berbahaya, membersihkan bahan peledak, dan mengidentifikasi lokasi penembak jitu.

Terjemahan Mesin: Sebuah program komputer secara otomatis menerjemahkan dari bahasa Arab ke bahasa Inggris, yang memungkinkan penutur bahasa Inggris untuk melihat tajuk utama “Ardogan Menegaskan Bahwa Turki Tidak Akan Menerima Tekanan Apa Pun, Mendesak Mereka untuk Mengakui Siprus.” Program tersebut menggunakan model statistik yang dibangun dari contoh-contoh terjemahan bahasa Arab ke bahasa Inggris dan dari contoh-contoh teks bahasa Inggris yang berjumlah dua triliun kata. Tidak ada ilmuwan komputer dalam tim tersebut yang berbicara bahasa Arab, tetapi mereka memahami statistik dan algoritma pembelajaran mesin.

1.5 RINGKASAN

Bab ini mendefinisikan AI dan menetapkan latar belakang budaya yang menjadi dasar pengembangannya. Berikut ini adalah beberapa poin penting:

- Orang yang berbeda mendekati AI dengan tujuan yang berbeda pula. Dua pertanyaan penting yang perlu ditanyakan adalah: Apakah Anda peduli dengan cara berpikir atau perilaku? Apakah Anda ingin memodelkan manusia atau bekerja berdasarkan standar ideal?
- Dalam buku ini, kami mengadopsi pandangan bahwa kecerdasan terutama berkaitan dengan tindakan rasional. Idealnya, agen cerdas mengambil tindakan terbaik yang memungkinkan dalam suatu situasi. Kami mempelajari masalah membangun agen yang cerdas dalam pengertian ini.
- Para filsuf (yang sudah ada sejak 400 SM) membuat AI dapat dipahami dengan mempertimbangkan gagasan bahwa pikiran dalam beberapa hal seperti mesin, yang beroperasi berdasarkan pengetahuan yang dikodekan dalam beberapa bahasa internal, dan bahwa pikiran dapat digunakan untuk memilih tindakan apa yang harus diambil.
- Matematikawan menyediakan alat untuk memanipulasi pernyataan kepastian logis serta pernyataan probabilistik yang tidak pasti. Mereka juga menetapkan dasar untuk memahami komputasi dan penalaran tentang algoritma.
- Ekonom memformalkan masalah pengambilan keputusan yang memaksimalkan hasil yang diharapkan bagi pembuat keputusan.
- Ahli saraf menemukan beberapa fakta tentang cara kerja otak dan cara otak mirip dan berbeda dari komputer.
- Psikolog mengadopsi gagasan bahwa manusia dan hewan dapat dianggap sebagai mesin pemroses informasi. Ahli bahasa menunjukkan bahwa penggunaan bahasa sesuai dengan model ini.
- Insinyur komputer menyediakan mesin yang semakin canggih yang memungkinkan penerapan AI.
- Teori kontrol berkaitan dengan perancangan perangkat yang bekerja secara optimal berdasarkan umpan balik dari lingkungan. Awalnya, perangkat matematika teori kontrol sangat berbeda dari AI, tetapi kedua bidang tersebut semakin dekat satu sama lain.
- Sejarah AI telah mengalami siklus keberhasilan, optimisme yang salah tempat, dan pengurangan antusiasme serta pendanaan. Ada juga siklus pengenalan pendekatan kreatif baru dan penyempurnaan sistematis yang terbaik.
- AI telah berkembang lebih pesat dalam dekade terakhir karena penggunaan metode ilmiah yang lebih besar dalam bereksperimen dan membandingkan pendekatan.
- Kemajuan terkini dalam memahami dasar teoritis kecerdasan berjalan seiring dengan peningkatan kemampuan sistem nyata. Subbidang AI telah menjadi lebih terintegrasi, dan AI telah menemukan titik temu dengan disiplin ilmu lainnya.

BAB 2
AGEN CERDAS

Kita mulai dengan memeriksa agen, lingkungan, dan hubungan di antara mereka. Pengamatan bahwa beberapa agen berperilaku lebih baik daripada yang lain secara alami mengarah pada gagasan tentang agen rasional—agen yang berperilaku sebaik mungkin. Seberapa baik agen dapat berperilaku tergantung pada sifat lingkungan; beberapa lingkungan lebih sulit daripada yang lain. Kami memberikan kategorisasi kasar lingkungan dan menunjukkan bagaimana sifat lingkungan memengaruhi desain agen yang sesuai untuk lingkungan tersebut. Kami menjelaskan sejumlah rancangan agen "kerangka" dasar, yang akan kami jabarkan di bagian selanjutnya dari buku ini.

2.1 AGEN DAN LINGKUNGAN

Agen adalah apa pun yang dapat dilihat sebagai persepsi terhadap lingkungannya melalui sensor dan bertindak atas lingkungan tersebut melalui aktuator. Ide sederhana ini diilustrasikan dalam Gambar 2.1. Agen manusia memiliki mata, telinga, dan organ lain sebagai sensor dan tangan, kaki, saluran vokal, dan sebagainya sebagai aktuator. Agen robotik mungkin memiliki kamera dan pencari jarak inframerah sebagai sensor dan berbagai motor sebagai aktuator.

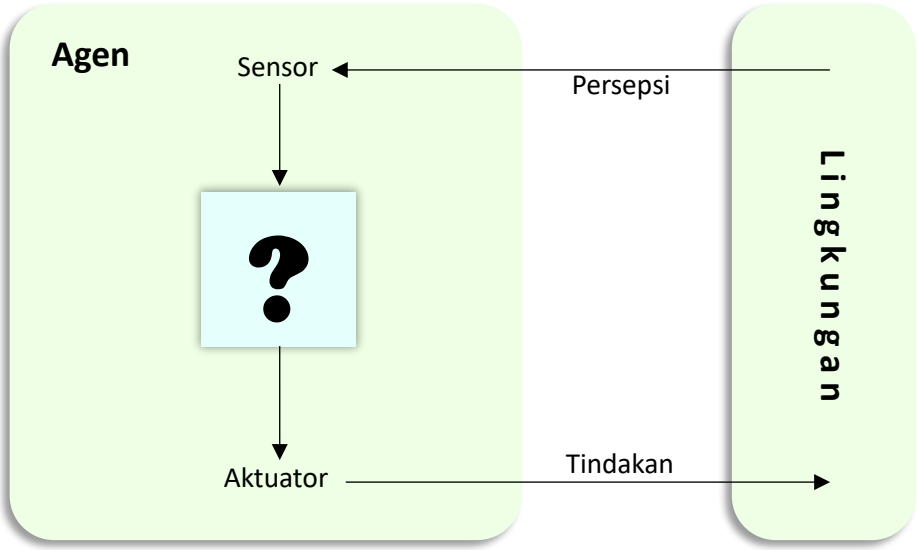
Agen perangkat lunak menerima penekanan tombol, konten file, dan paket jaringan sebagai masukan sensorik dan bertindak atas lingkungan dengan menampilkannya di layar, menulis file, dan mengirim paket jaringan. Kami menggunakan istilah persepsi untuk merujuk pada masukan persepsi agen pada saat tertentu. Urutan persepsi agen adalah riwayat lengkap dari semua hal yang pernah dirasakan agen.

Secara umum, pilihan tindakan agen pada saat tertentu dapat bergantung pada seluruh urutan persepsi yang diamati hingga saat ini, tetapi tidak pada apa pun yang belum dirasakannya. Dengan menentukan pilihan tindakan agen untuk setiap kemungkinan urutan persepsi, kita telah mengatakan lebih atau kurang semua hal yang perlu dikatakan tentang agen. Secara matematis, kita mengatakan bahwa perilaku agen dijelaskan oleh fungsi agen yang memetakan setiap urutan persepsi yang diberikan ke suatu tindakan.

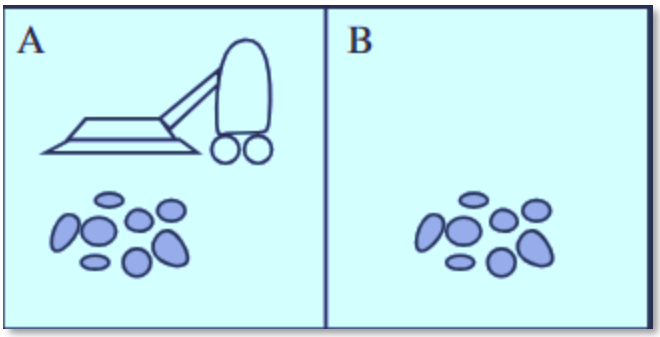
Kita dapat membayangkan menabulasi fungsi agen yang menggambarkan agen yang diberikan; untuk sebagian besar agen, ini akan menjadi tabel yang sangat besar—bahkan tak terbatas, kecuali kita menetapkan batasan pada panjang urutan persepsi yang ingin kita pertimbangkan. Mengingat agen untuk bereksperimen, pada prinsipnya kita dapat membuat tabel ini dengan mencoba semua kemungkinan urutan persepsi dan mencatat tindakan yang dilakukan agen sebagai respons. Tabel tersebut, tentu saja, merupakan karakterisasi eksternal dari agen. Secara internal, fungsi agen untuk agen buatan akan diimplementasikan oleh program agen. Penting untuk menjaga agar kedua gagasan ini tetap berbeda. Fungsi agen adalah deskripsi matematis abstrak; program agen adalah implementasi konkret, yang berjalan dalam beberapa sistem fisik.

Untuk mengilustrasikan gagasan ini, kita menggunakan contoh yang sangat sederhana—dunia penyedot debu yang ditunjukkan pada Gambar 2.2. Dunia ini sangat sederhana sehingga kita dapat menggambarkan semua yang terjadi; dunia ini juga merupakan dunia yang dibuat-buat, sehingga kita dapat menciptakan banyak variasi. Dunia khusus ini hanya memiliki dua lokasi: kotak A dan B. Agen vakum merasakan di kotak mana ia berada dan apakah ada kotoran di kotak tersebut. Agen dapat memilih untuk bergerak ke kiri, bergerak ke kanan, menyedot kotoran, atau tidak melakukan apa pun. Salah satu fungsi agen yang sangat sederhana adalah sebagai berikut: jika kotak saat ini kotor, maka sedot; jika tidak, pindah ke kotak lainnya. Tabulasi sebagian dari fungsi agen ini ditunjukkan pada Gambar 2.3 dan program agen yang mengimplementasikannya muncul pada Gambar 2.8 di halaman 48.

Dengan melihat Gambar 2.3, kita melihat bahwa berbagai agen dunia vakum dapat didefinisikan hanya dengan mengisi kolom sebelah kanan dengan berbagai cara. Pertanyaan yang jelas adalah: Bagaimana cara yang tepat untuk mengisi tabel? Dengan kata lain, apa yang membuat agen baik atau buruk, cerdas atau bodoh? Kami menjawab pertanyaan-pertanyaan ini di bagian berikutnya.



Gambar 2.1 Agen berinteraksi dengan lingkungan melalui sensor dan aktuator.



Gambar 2.2 Dunia penyedot debu dengan hanya dua lokasi.

Urutan Persepsi	*Tindakan*
[A, Bersih]	*Kanan*
[A, Kotor]	*Suck*
[B, Bersih]	*Kiri*
[B, Kotor]	*Suck*
[A, Bersih], [A, Bersih]	*Kanan*
[A, Bersih], [A, Kotor]	*Suck*
⋮	⋮
[A, Bersih], [A, Bersih], [A, Bersih]	*Kanan*
[A, Bersih], [A, Bersih], [A, Kotor]	*Suck*
⋮	⋮

Gambar 2.3 Tabulasi parsial fungsi agen sederhana untuk dunia penyedot debu yang ditunjukkan pada Gambar 2.2.

Sebelum menutup bagian ini, kami harus menekankan bahwa pengertian agen dimaksudkan sebagai alat untuk menganalisis sistem, bukan karakterisasi absolut yang membagi dunia menjadi agen dan non-agen. Seseorang dapat melihat kalkulator genggam sebagai agen yang memilih tindakan menampilkan "4" ketika diberikan urutan persepsi "2 + 2 =," tetapi analisis semacam itu tidak akan membantu pemahaman kita tentang kalkulator. Dalam arti tertentu, semua bidang teknik dapat dilihat sebagai perancangan artefak yang berinteraksi dengan dunia; AI beroperasi pada (apa yang dianggap penulis) ujung spektrum yang paling menarik, di mana artefak memiliki sumber daya komputasi yang signifikan dan lingkungan tugas memerlukan pengambilan keputusan yang tidak sepele.

2.2 PERILAKU BAIK: KONSEP RASIONALITAS

Agen rasional adalah agen yang melakukan hal yang benar—secara konseptual, setiap entri dalam tabel untuk fungsi agen diisi dengan benar. Jelas, melakukan hal yang benar lebih baik daripada melakukan hal yang salah, tetapi apa artinya melakukan hal yang benar?

Kami menjawab pertanyaan lama ini dengan cara lama: dengan mempertimbangkan konsekuensi perilaku agen. Ketika agen ditempatkan di suatu lingkungan, ia menghasilkan serangkaian tindakan sesuai dengan persepsi yang diterimanya. Rangkaian tindakan ini menyebabkan lingkungan mengalami serangkaian keadaan. Jika rangkaian tersebut diinginkan, maka agen telah berkinerja baik. Gagasan tentang keinginan ini ditangkap oleh ukuran kinerja yang mengevaluasi setiap rangkaian keadaan lingkungan yang diberikan.

Perhatikan bahwa kami mengatakan keadaan lingkungan, bukan keadaan agen. Jika kita mendefinisikan keberhasilan dalam hal pendapat agen tentang kinerjanya sendiri, agen dapat mencapai rasionalitas sempurna hanya dengan menipu dirinya sendiri bahwa kinerjanya sempurna. Agen manusia khususnya terkenal karena "rasa iri"—percaya bahwa mereka tidak benar-benar menginginkan sesuatu (misalnya, Hadiah Nobel) setelah tidak mendapatkannya. Jelas, tidak ada satu ukuran kinerja yang pasti untuk semua tugas dan agen; biasanya, seorang desainer akan merancang satu ukuran yang sesuai dengan keadaan. Ini tidak semudah kedengarannya. Pertimbangkan, misalnya, agen penyedot debu dari bagian sebelumnya. Kita

mungkin mengusulkan untuk mengukur kinerja berdasarkan jumlah kotoran yang dibersihkan dalam satu shift delapan jam.

Dengan agen yang rasional, tentu saja, apa yang Anda minta adalah apa yang Anda dapatkan. Agen yang rasional dapat memaksimalkan ukuran kinerja ini dengan membersihkan kotoran, lalu membuang semuanya ke lantai, lalu membersihkannya lagi, dan seterusnya. Ukuran kinerja yang lebih sesuai akan memberi penghargaan kepada agen karena lantainya bersih. Misalnya, satu poin dapat diberikan untuk setiap kotak bersih pada setiap langkah waktu (mungkin dengan penalti untuk listrik yang dikonsumsi dan kebisingan yang dihasilkan). Sebagai aturan umum, lebih baik merancang ukuran kinerja sesuai dengan apa yang sebenarnya diinginkan seseorang di lingkungan tersebut, daripada menurut bagaimana menurut seseorang agen seharusnya berperilaku.

Bahkan ketika jebakan yang jelas dihindari, masih ada beberapa masalah pelik yang harus diurai. Misalnya, gagasan "lantai bersih" dalam paragraf sebelumnya didasarkan pada kebersihan rata-rata dari waktu ke waktu. Namun, kebersihan rata-rata yang sama dapat dicapai oleh dua agen yang berbeda, yang satu melakukan pekerjaan yang biasa-biasa saja sepanjang waktu sementara yang lain membersihkan dengan penuh semangat tetapi mengambil waktu istirahat yang lama.

Mana yang lebih baik mungkin tampak menjadi pokok bahasan yang bagus dalam ilmu kebersihan, tetapi sebenarnya itu adalah pertanyaan filosofis yang mendalam dengan implikasi yang luas. Mana yang lebih baik—kehidupan yang sembrono dengan pasang surut, atau kehidupan yang aman tetapi membosankan? Mana yang lebih baik—ekonomi di mana setiap orang hidup dalam kemiskinan sedang, atau ekonomi di mana sebagian orang hidup berkelimpahan sementara yang lain sangat miskin? Kami meninggalkan pertanyaan-pertanyaan ini sebagai latihan bagi pembaca yang tekun.

Rasionalitas

Apa yang rasional pada waktu tertentu bergantung pada empat hal:

- Ukuran kinerja yang menentukan kriteria keberhasilan.
- Pengetahuan agen sebelumnya tentang lingkungan.
- Tindakan yang dapat dilakukan agen.
- Urutan persepsi agen hingga saat ini.

Hal ini mengarah pada definisi agen rasional:

> *Untuk setiap kemungkinan urutan persepsi, agen rasional harus memilih tindakan yang diharapkan dapat memaksimalkan ukuran kinerjanya, dengan mempertimbangkan bukti yang diberikan oleh urutan persepsi dan pengetahuan bawaan apa pun yang dimiliki agen.*

Pertimbangkan agen penyedot debu sederhana yang membersihkan kotak jika kotor dan pindah ke kotak lain jika tidak; ini adalah fungsi agen yang ditabulasikan pada Gambar 2.3. Apakah ini agen rasional? Tergantung! Pertama, kita perlu mengatakan apa ukuran kinerjanya, apa yang diketahui tentang lingkungan, dan sensor dan aktuator apa yang dimiliki agen tersebut. Mari kita asumsikan yang berikut:

- Ukuran kinerja memberikan satu poin untuk setiap kotak bersih pada setiap langkah waktu, selama "masa hidup" 1000 langkah waktu.

- "Geografi" lingkungan diketahui apriori (Gambar 2.2) tetapi distribusi kotoran dan lokasi awal agen tidak. Kotak bersih tetap bersih dan penghisapan membersihkan kotak saat ini. Tindakan Kiri dan Kanan menggerakkan agen ke kiri dan kanan kecuali jika ini akan membawa agen ke luar lingkungan, dalam hal ini agen tetap di tempatnya.
- Satu-satunya tindakan yang tersedia adalah Kiri, Kanan, dan Hisap.
- Agen tersebut benar-benar memahami lokasinya dan apakah lokasi tersebut mengandung kotoran.

Kami menyatakan bahwa dalam keadaan ini agen tersebut memang rasional; kinerja yang diharapkan setidaknya sama tingginya dengan agen lainnya. Latihan 2.2 meminta Anda untuk membuktikan hal ini.

Kita dapat dengan mudah melihat bahwa agen yang sama akan menjadi tidak rasional dalam keadaan yang berbeda. Misalnya, setelah semua kotoran dibersihkan, agen akan berosilasi maju mundur tanpa perlu; jika ukuran kinerja mencakup penalti satu poin untuk setiap gerakan ke kiri atau kanan, agen tersebut akan berkinerja buruk. Agen yang lebih baik untuk kasus ini tidak akan melakukan apa pun setelah yakin bahwa semua kotak bersih. Jika kotak yang bersih dapat menjadi kotor lagi, agen tersebut harus sesekali memeriksa dan membersihkannya kembali jika diperlukan. Jika geografi lingkungan tidak diketahui, agen perlu menjelajahinya daripada terpaku pada kotak A dan B.

Kemahatahuan, Pembelajaran, dan Otonomi

Kita perlu berhati-hati dalam membedakan antara rasionalitas dan kemahatahuan. Agen yang mahatahu mengetahui hasil aktual dari tindakannya dan dapat bertindak sesuai dengannya; tetapi kemahatahuan tidak mungkin dalam kenyataan. Pertimbangkan contoh berikut: Saya sedang berjalan di sepanjang Champs Elysees suatu hari dan saya melihat seorang teman lama di seberang jalan.

Tidak ada lalu lintas di dekatnya dan saya tidak melakukan hal lain, jadi, karena rasional, saya mulai menyeberang jalan. Sementara itu, pada ketinggian 33.000 kaki, pintu kargo jatuh dari pesawat yang lewat, dan sebelum saya sampai ke seberang jalan, saya tergencet. Apakah saya tidak rasional karena menyeberang jalan? Tidak mungkin obituari saya akan berbunyi "Orang bodoh mencoba menyeberang jalan."

Contoh ini menunjukkan bahwa rasionalitas tidak sama dengan kesempurnaan. Rasionalitas memaksimalkan kinerja yang diharapkan, sementara kesempurnaan memaksimalkan kinerja aktual. Mundur dari persyaratan kesempurnaan bukan hanya soal bersikap adil kepada agen. Intinya adalah jika kita mengharapkan agen melakukan tindakan terbaik setelah kejadian, mustahil untuk merancang agen yang memenuhi spesifikasi ini—kecuali kita meningkatkan kinerja bola kristal atau mesin waktu.

Definisi rasionalitas kita tidak memerlukan kemahatahuan, karena pilihan rasional hanya bergantung pada urutan persepsi hingga saat ini. Kita juga harus memastikan bahwa kita tidak secara tidak sengaja membiarkan agen terlibat dalam aktivitas yang jelas-jelas kurang cerdas. Misalnya, jika agen tidak melihat ke dua arah sebelum menyeberang jalan yang ramai, maka urutan persepsinya tidak akan memberi tahu bahwa ada truk besar yang mendekat dengan kecepatan tinggi. Apakah definisi rasionalitas kita mengatakan bahwa sekarang boleh

menyeberang jalan? Jauh dari itu! Pertama, tidaklah rasional untuk menyeberang jalan mengingat urutan persepsi yang tidak informatif ini: risiko kecelakaan karena menyeberang tanpa melihat terlalu besar.

Kedua, agen yang rasional harus memilih tindakan "melihat" sebelum melangkah ke jalan, karena melihat membantu memaksimalkan kinerja yang diharapkan. Melakukan tindakan untuk mengubah persepsi di masa mendatang—kadang-kadang disebut pengumpulan informasi. Contoh kedua dari pengumpulan informasi diberikan oleh eksplorasi yang harus dilakukan oleh agen pembersih vakum di lingkungan yang awalnya tidak dikenal. Definisi kami mengharuskan agen rasional tidak hanya mengumpulkan informasi tetapi juga belajar sebanyak mungkin dari apa yang dirasakannya. Konfigurasi awal agen dapat mencerminkan beberapa pengetahuan sebelumnya tentang lingkungan, tetapi seiring agen memperoleh pengalaman, hal ini dapat dimodifikasi dan ditambah.

Ada kasus ekstrem di mana lingkungan sepenuhnya diketahui secara apriori. Dalam kasus seperti itu, agen tidak perlu merasakan atau belajar; ia hanya bertindak dengan benar. Tentu saja, agen seperti itu rapuh. Pertimbangkan kumbang kotoran yang rendah hati. Setelah menggali sarangnya dan bertelur, ia mengambil bola kotoran dari tumpukan di dekatnya untuk menutup pintu masuk. Jika bola kotoran itu terlepas dari genggamannya di tengah perjalanan, kumbang itu melanjutkan tugasnya dan berpura-pura menutup sarang dengan bola kotoran yang tidak ada, tanpa pernah menyadari bahwa bola itu hilang.

Evolusi telah membangun asumsi dalam perilaku kumbang, dan ketika asumsi itu dilanggar, perilaku yang tidak berhasil pun terjadi. Tawon sphex sedikit lebih cerdas. Tawon sphex betina akan menggali liang, keluar dan menyengat ulat dan menyeretnya ke liang, memasuki liang lagi untuk memeriksa apakah semuanya baik-baik saja, menyeret ulat ke dalam, dan bertelur. Ulat itu berfungsi sebagai sumber makanan saat telur menetas. Sejauh ini baik-baik saja, tetapi jika seorang ahli entomologi memindahkan ulat beberapa inci jauhnya saat sphex melakukan pemeriksaan, ia akan kembali ke langkah "seret" dari rencananya dan akan melanjutkan rencananya tanpa modifikasi, bahkan setelah puluhan intervensi pemindahan ulat.

Sphex tidak dapat belajar bahwa rencana bawaannya gagal, dan dengan demikian tidak akan mengubahnya. Sejauh agen bergantung pada pengetahuan awal perancangnya daripada persepsinya sendiri, kita katakan bahwa agen tersebut tidak memiliki otonomi. Agen rasional haruslah otonom—ia harus mempelajari apa yang dapat dilakukannya untuk mengimbangi pengetahuan awal yang sebagian atau tidak benar. Misalnya, agen pembersih vakum yang belajar untuk meramalkan di mana dan kapan kotoran tambahan akan muncul akan bekerja lebih baik daripada agen yang tidak melakukannya.

Dalam praktiknya, seseorang jarang membutuhkan otonomi penuh sejak awal: ketika agen tersebut memiliki sedikit atau tidak memiliki pengalaman, ia harus bertindak secara acak kecuali perancang memberikan bantuan. Jadi, seperti evolusi yang memberi hewan cukup banyak refleks bawaan untuk bertahan hidup cukup lama untuk belajar sendiri, akan masuk akal untuk memberi agen kecerdasan buatan beberapa pengetahuan awal serta kemampuan untuk belajar. Setelah pengalaman yang cukup tentang lingkungannya, perilaku agen rasional

dapat menjadi efektif dan independen dari pengetahuan sebelumnya. Oleh karena itu, penggabungan pembelajaran memungkinkan seseorang untuk merancang satu agen rasional yang akan berhasil dalam berbagai macam lingkungan.

2.3 SIFAT LINGKUNGAN

Sekarang setelah kita memiliki definisi rasionalitas, kita hampir siap untuk berpikir tentang membangun agen rasional. Namun, pertama-tama, kita harus berpikir tentang lingkungan tugas, yang pada dasarnya adalah "masalah" yang "ditangani" oleh agen rasional. Kita mulai dengan menunjukkan cara menentukan lingkungan tugas, mengilustrasikan proses dengan sejumlah contoh. Kemudian kita menunjukkan bahwa lingkungan tugas hadir dalam berbagai bentuk. Bentuk lingkungan tugas secara langsung memengaruhi desain yang tepat untuk program agen.

Menentukan Lingkungan Tugas

Dalam pembahasan kita tentang rasionalitas agen penyedot debu sederhana, kita harus menentukan ukuran kinerja, lingkungan, dan aktuator serta sensor agen. Kita mengelompokkan semua ini di bawah judul lingkungan tugas. Bagi yang berpikiran akronim, kita menyebutnya deskripsi PEAS (Kinerja, Lingkungan, Aktuator, Sensor). Dalam merancang agen, langkah pertama harus selalu menentukan lingkungan tugas selengkap mungkin.

Dunia vakum adalah contoh sederhana; Mari kita pertimbangkan masalah yang lebih rumit: pengemudi taksi otomatis. Kita harus menunjukkan, sebelum pembaca menjadi khawatir, bahwa taksi yang sepenuhnya otomatis saat ini agak melampaui kemampuan teknologi yang ada. Tugas mengemudi penuh sangat terbuka. Tidak ada batasan untuk kombinasi keadaan baru yang dapat muncul—alasan lain mengapa kami memilihnya sebagai fokus diskusi. Gambar 2.4 merangkum deskripsi PEAS untuk lingkungan tugas taksi. Kami membahas setiap elemen secara lebih rinci dalam paragraf berikut.

Tipe Agen	Ukuran Kinerja	Lingkungan	Akuator	Sensor
Supir Taksi	Perjalanan Aman, Cepat, Legal, Nyaman, Memaksimalkan Keuntungan	Jalan, Lalu lintas lainnya, Pejalan kaki, Pelanggan	Kemudi, Akselerasi, rem, sinyal, klakson, display	Kamera, Sonar, speedometer, GPS, odometer, akselerometer, sensor mesin, keyboard

Gambar 2.4 Deskripsi PEAS tentang lingkungan tugas untuk taksi otomatis.

Pertama, apa ukuran kinerja yang ingin kita capai dari pengemudi otomatis kita? Kualitas yang diharapkan mencakup mencapai tujuan yang benar; meminimalkan konsumsi bahan bakar dan keausan; meminimalkan waktu atau biaya perjalanan; meminimalkan pelanggaran undang-undang lalu lintas dan gangguan bagi pengemudi lain; memaksimalkan keselamatan dan

kenyamanan penumpang; memaksimalkan keuntungan. Jelas, beberapa dari tujuan ini saling bertentangan, jadi diperlukan pengorbanan.

Selanjutnya, seperti apa lingkungan berkendara yang akan dihadapi taksi? Setiap pengemudi taksi harus berhadapan dengan berbagai jalan, mulai dari jalan pedesaan dan gang perkotaan hingga jalan bebas hambatan 12 jalur. Jalan tersebut berisi lalu lintas lain, pejalan kaki, hewan liar, pekerjaan jalan, mobil polisi, genangan air, dan lubang jalan. Taksi juga harus berinteraksi dengan calon penumpang dan penumpang sebenarnya.

Ada juga beberapa pilihan opsional. Taksi mungkin perlu beroperasi di California Selatan, tempat salju jarang menjadi masalah, atau di Alaska, tempat salju jarang tidak menjadi masalah. Mobil dapat selalu melaju di jalur kanan, atau kita mungkin menginginkannya cukup fleksibel untuk melaju di jalur kiri saat berada di Inggris atau Jepang. Jelas, semakin terbatas lingkungannya, semakin mudah masalah desainnya.

Aktuator untuk taksi otomatis mencakup yang tersedia untuk pengemudi manusia: kontrol atas mesin melalui akselerator dan kontrol atas kemudi dan pengereman. Selain itu, ia memerlukan output ke layar tampilan atau synthesizer suara untuk berbicara kembali kepada penumpang, dan mungkin beberapa cara untuk berkomunikasi dengan kendaraan lain, dengan sopan atau sebaliknya.

Sensor dasar untuk taksi akan mencakup satu atau lebih kamera video yang dapat dikontrol sehingga dapat melihat jalan; ia dapat melengkapinya dengan sensor inframerah atau sonar untuk mendeteksi jarak ke mobil lain dan rintangan. Untuk menghindari tilang, taksi harus memiliki speedometer, dan untuk mengendalikan kendaraan dengan benar, terutama di tikungan, ia harus memiliki accelerometer. Untuk menentukan kondisi mekanis kendaraan, ia memerlukan rangkaian sensor mesin, bahan bakar, dan sistem kelistrikan yang biasa. Seperti banyak pengemudi manusia, mungkin diperlukan sistem penentuan posisi global (GPS) agar tidak tersesat. Terakhir, diperlukan papan ketik atau mikrofon agar penumpang dapat meminta tujuan.

Pada Gambar 2.5, kami telah membuat sketsa elemen PEAS dasar untuk sejumlah jenis agen tambahan. Mungkin mengejutkan bagi sebagian pembaca bahwa daftar jenis agen kami mencakup beberapa program yang beroperasi di lingkungan yang sepenuhnya buatan yang ditentukan oleh masukan papan ketik dan keluaran karakter di layar. "Tentunya," seseorang mungkin berkata, "ini bukan lingkungan nyata, bukan?" Faktanya, yang penting bukanlah perbedaan antara lingkungan "nyata" dan "buatan", tetapi kompleksitas hubungan antara perilaku agen, urutan persepsi yang dihasilkan oleh lingkungan, dan ukuran kinerja. Beberapa lingkungan "nyata" sebenarnya cukup sederhana.

Pada Gambar 2.5, kami telah membuat sketsa elemen PEAS dasar untuk sejumlah jenis agen tambahan. Mungkin mengejutkan bagi sebagian pembaca bahwa daftar jenis agen kami mencakup beberapa program yang beroperasi di lingkungan yang sepenuhnya buatan yang ditentukan oleh masukan papan ketik dan keluaran karakter di layar. "Tentunya," seseorang mungkin berkata, "ini bukan lingkungan nyata, bukan?" Faktanya, yang penting bukanlah perbedaan antara lingkungan "nyata" dan "buatan", tetapi kompleksitas hubungan antara perilaku agen, urutan persepsi yang dihasilkan oleh lingkungan, dan ukuran kinerja.

Beberapa lingkungan "nyata" sebenarnya cukup sederhana. Misalnya, robot yang dirancang untuk memeriksa komponen saat komponen tersebut melewati sabuk konveyor dapat menggunakan sejumlah asumsi penyederhanaan: bahwa pencahayaannya selalu tepat, bahwa satu-satunya benda di sabuk konveyor adalah komponen yang jenisnya diketahui, dan bahwa hanya dua tindakan (menerima atau menolak) yang mungkin dilakukan.

Sebaliknya, beberapa agen perangkat lunak (atau robot perangkat lunak atau softbot) ada di domain yang kaya dan tak terbatas. Bayangkan operator situs web softbot yang dirancang untuk memindai sumber berita Internet dan menunjukkan item menarik kepada penggunanya, sambil menjual ruang iklan untuk menghasilkan pendapatan. Agar berhasil, operator tersebut akan memerlukan beberapa kemampuan pemrosesan bahasa alami, ia perlu mempelajari apa yang diminati setiap pengguna dan pengiklan, dan ia perlu mengubah rencananya secara dinamis—misalnya, ketika koneksi untuk satu sumber berita terputus atau ketika yang baru online. Internet adalah lingkungan yang kompleksitasnya menyaingi dunia fisik dan yang penghuninya mencakup banyak agen buatan dan manusia.

Properti Lingkungan Tugas

Rentang lingkungan tugas yang mungkin muncul dalam AI jelas sangat luas. Namun, kita dapat mengidentifikasi sejumlah kecil dimensi yang dengannya lingkungan tugas dapat dikategorikan. Dimensi-dimensi ini menentukan, sebagian besar, desain agen yang tepat dan penerapan masing-masing keluarga teknik utama untuk implementasi agen. Pertama, kami mencantumkan dimensinya, lalu kami menganalisis beberapa lingkungan tugas untuk mengilustrasikan ide-idenya. Definisi di sini bersifat informal; bab-bab selanjutnya memberikan pernyataan dan contoh yang lebih tepat dari setiap jenis lingkungan.

Dapat diamati sepenuhnya vs. dapat diamati sebagian: Jika sensor agen memberinya akses ke status lingkungan yang lengkap pada setiap titik waktu, maka kami katakan bahwa lingkungan tugas dapat diamati sepenuhnya. Lingkungan tugas secara efektif dapat diamati sepenuhnya jika sensor mendeteksi semua aspek yang relevan dengan pilihan tindakan; relevansi, pada gilirannya, bergantung pada ukuran kinerja.

Lingkungan yang dapat diamati sepenuhnya mudah karena agen tidak perlu mempertahankan status internal apa pun untuk melacak dunia. Suatu lingkungan mungkin dapat diamati sebagian karena sensor yang bising dan tidak akurat atau karena bagian dari status tersebut hilang begitu saja dari data sensor—misalnya, agen vakum dengan hanya sensor kotoran lokal tidak dapat mengetahui apakah ada kotoran di kotak lain, dan taksi otomatis tidak dapat melihat apa yang dipikirkan pengemudi lain.

Jika agen tidak memiliki sensor sama sekali, maka lingkungan tersebut tidak dapat diamati. Kita mungkin berpikir bahwa dalam kasus seperti itu, nasib agen tidak ada harapan, tetapi, seperti yang kita bahas di Bab 4, tujuan agen mungkin masih dapat dicapai, terkadang dengan pasti.

Tipe Agen	Ukuran Kinerja	Lingkungan	Akuator	Sensor
Sistem diagnosis medis	Pasien sehat, biaya berkurang	Pasien, Rumah sakit, staff	Tampialn Pertanyaan, tes, Diagnosis, Perawatan, Rujukan	Entry Keyboard, Gejala, Temuan, Jawaban Pasien.
Sistem Analisis, citra satelit	Kategorisasi gambar yang benar	Downlink dari satelit yang mengorbit	Tampilan kategorisasi kejadian	Array piksel warna
Robot pemetik sebagian	Persentase bagian dalam tempat sampah yg benar	Sabuk konveyor dengan bagian-bagian tempat sampah	Lengan dan Tangan bersendi	Kamera, Sonar, Sudut sambungan
Pengendali Kilang	Kemurnian Hasil	Kilang, Operator	Katup, Pompa, pemanas, Display	Sensor Suhu, Tekanan, kimia
Guru Bahasa inggris Interaktif	Nilai siswa pada ujian	Sekumpulan siswa, Lembaga pengujian	Tampilan Latihan, saran, koreksi	Entri papan ketik

Gambar 2.5 Contoh jenis agen dan deskripsi PEAS-nya.

Agen tunggal vs. multiagen: Perbedaan antara lingkungan agen tunggal dan multiagen mungkin tampak cukup sederhana. Misalnya, agen yang memecahkan teka-teki silang sendirian jelas berada dalam lingkungan agen tunggal, sedangkan agen yang bermain catur berada dalam lingkungan dua agen. Namun, ada beberapa masalah yang tidak kentara. Pertama, kami telah menjelaskan bagaimana suatu entitas dapat dipandang sebagai agen, tetapi kami belum menjelaskan entitas mana yang harus dipandang sebagai agen. Apakah agen A (misalnya pengemudi taksi) harus memperlakukan objek B (kendaraan lain) sebagai agen, atau dapatkah objek tersebut diperlakukan hanya sebagai objek yang berperilaku sesuai dengan hukum fisika, yang dianalogikan dengan ombak di pantai atau daun yang tertiup angin? Perbedaan utamanya adalah apakah perilaku B paling baik dijelaskan sebagai memaksimalkan ukuran kinerja yang nilainya bergantung pada perilaku agen A.

Misalnya, dalam catur, entitas lawan B mencoba memaksimalkan ukuran kinerjanya, yang, menurut aturan catur, meminimalkan ukuran kinerja agen A. Dengan demikian, catur adalah lingkungan multiagen yang kompetitif. Di sisi lain, dalam lingkungan pengemudi taksi, menghindari tabrakan memaksimalkan ukuran kinerja semua agen, sehingga merupakan

lingkungan multiagen yang sebagian kooperatif. Lingkungan ini juga sebagian kompetitif karena, misalnya, hanya satu mobil yang dapat menempati tempat parkir. Masalah desain agen dalam lingkungan multiagen sering kali sangat berbeda dari masalah dalam lingkungan agen tunggal; misalnya, komunikasi sering kali muncul sebagai perilaku rasional dalam lingkungan multiagen; dalam beberapa lingkungan kompetitif, perilaku acak bersifat rasional karena menghindari jebakan prediktabilitas.

Deterministik vs. stokastik. Jika keadaan lingkungan berikutnya sepenuhnya ditentukan oleh keadaan saat ini dan tindakan yang dilakukan oleh agen, maka kita katakan lingkungan tersebut deterministik; jika tidak, maka lingkungan tersebut stokastik. Pada prinsipnya, seorang agen tidak perlu khawatir tentang ketidakpastian dalam lingkungan deterministik yang sepenuhnya dapat diamati. (Dalam definisi kami, kami mengabaikan ketidakpastian yang muncul murni dari tindakan agen lain dalam lingkungan multiagen; dengan demikian, sebuah permainan dapat bersifat deterministik meskipun setiap agen mungkin tidak dapat memprediksi tindakan yang lain.) Namun, jika lingkungan tersebut sebagian dapat diamati, maka lingkungan tersebut dapat tampak stokastik. Sebagian besar situasi nyata sangat kompleks sehingga mustahil untuk melacak semua aspek yang tidak teramati; untuk tujuan praktis, situasi tersebut harus diperlakukan sebagai stokastik. Mengemudi taksi jelas bersifat stokastik dalam pengertian ini, karena seseorang tidak akan pernah dapat memprediksi perilaku lalu lintas secara tepat; terlebih lagi, ban seseorang dapat meletus dan mesinnya macet tanpa peringatan. Dunia vakum seperti yang kami gambarkan bersifat deterministik, tetapi variasinya dapat mencakup elemen stokastik seperti kotoran yang muncul secara acak dan mekanisme penghisapan yang tidak dapat diandalkan (Latihan 2.13).

Kita mengatakan suatu lingkungan tidak pasti jika tidak dapat diamati sepenuhnya atau tidak deterministik. Satu catatan akhir: penggunaan kata "stokastik" secara umum menyiratkan bahwa ketidakpastian tentang hasil dikuantifikasi dalam hal probabilitas; lingkungan nondeterministik adalah lingkungan di mana tindakan dicirikan oleh kemungkinan hasilnya, tetapi tidak ada probabilitas yang melekat padanya. Deskripsi lingkungan nondeterministik biasanya dikaitkan dengan ukuran kinerja yang mengharuskan agen untuk berhasil untuk semua kemungkinan hasil tindakannya.

Episodik vs. sekuensial: Dalam lingkungan tugas episodik, pengalaman agen dibagi menjadi beberapa episode atomik. Dalam setiap episode, agen menerima persepsi dan kemudian melakukan satu tindakan. Yang terpenting, episode berikutnya tidak bergantung pada tindakan yang diambil dalam episode sebelumnya. Banyak tugas klasifikasi bersifat episodik. Misalnya, agen yang harus menemukan komponen yang rusak pada jalur perakitan mendasarkan setiap keputusan pada komponen saat ini, terlepas dari keputusan sebelumnya; lebih jauh, keputusan saat ini tidak memengaruhi apakah komponen berikutnya rusak atau tidak.

Di sisi lain, dalam lingkungan berurutan, keputusan saat ini dapat memengaruhi semua keputusan di masa mendatang. Catur dan mengemudi taksi bersifat berurutan: dalam kedua kasus, tindakan jangka pendek dapat memiliki konsekuensi jangka panjang. Lingkungan

episodik jauh lebih sederhana daripada lingkungan berurutan karena agen tidak perlu berpikir ke depan.

Statis vs. dinamis: Jika lingkungan dapat berubah saat agen sedang mempertimbangkan, maka kita katakan lingkungan tersebut dinamis untuk agen tersebut; jika tidak, maka lingkungan tersebut statis. Lingkungan statis mudah dihadapi karena agen tidak perlu terus-menerus melihat dunia saat memutuskan tindakan, juga tidak perlu khawatir tentang berlalunya waktu.

Di sisi lain, lingkungan dinamis terus-menerus menanyakan kepada agen apa yang ingin dilakukannya; jika belum memutuskan, itu berarti memutuskan untuk tidak melakukan apa pun. Jika lingkungan itu sendiri tidak berubah seiring berjalannya waktu tetapi skor kinerja agen berubah, maka kita katakan lingkungan tersebut semidinamis. Mengemudikan taksi jelas dinamis: mobil-mobil lain dan taksi itu sendiri terus bergerak sementara algoritme pengemudian ragu-ragu tentang apa yang harus dilakukan selanjutnya. Catur, ketika dimainkan dengan jam, bersifat semidinamis. Teka-teki silang bersifat statis.

Diskrit vs. kontinu: Perbedaan diskrit/kontinu berlaku untuk keadaan lingkungan, cara waktu ditangani, dan persepsi serta tindakan agen. Misalnya, lingkungan catur memiliki sejumlah keadaan berbeda yang terbatas (tidak termasuk jam). Catur juga memiliki serangkaian persepsi dan tindakan diskrit. Mengemudikan taksi adalah masalah keadaan kontinu dan waktu kontinu: kecepatan dan lokasi taksi serta kendaraan lain bergerak melalui serangkaian nilai kontinu dan melakukannya dengan lancar dari waktu ke waktu. Tindakan mengemudikan taksi juga kontinu (sudut kemudi, dll.). Input dari kamera digital bersifat diskrit, secara tegas, tetapi biasanya diperlakukan sebagai representasi intensitas dan lokasi yang terus berubah.

Diketahui vs. tidak diketahui: Secara tegas, perbedaan ini tidak merujuk pada lingkungan itu sendiri, tetapi pada status pengetahuan agen (atau perancang) tentang "hukum fisika" lingkungan tersebut. Dalam lingkungan yang diketahui, hasil (atau probabilitas hasil jika lingkungan bersifat stokastik) untuk semua tindakan diberikan. Jelas, jika lingkungan tidak diketahui, agen harus mempelajari cara kerjanya agar dapat membuat keputusan yang baik.

Perhatikan bahwa perbedaan antara lingkungan yang diketahui dan tidak diketahui tidak sama dengan perbedaan antara lingkungan yang dapat diamati sepenuhnya dan sebagian. Sangat mungkin bagi lingkungan yang diketahui untuk dapat diamati sebagian—misalnya, dalam permainan kartu solitaire, saya mengetahui aturannya tetapi masih tidak dapat melihat kartu yang belum dibalik. Sebaliknya, lingkungan yang tidak diketahui dapat diamati sepenuhnya—dalam permainan video baru, layar mungkin menampilkan seluruh status permainan tetapi saya masih tidak tahu apa yang dilakukan tombol-tombol tersebut sampai saya mencobanya. Seperti yang mungkin diharapkan, kasus tersulit adalah yang dapat diamati sebagian, multiagen, stokastik, berurutan, dinamis, berkelanjutan, dan tidak diketahui. Mengemudikan taksi sulit dalam semua hal ini, kecuali bahwa sebagian besar lingkungan pengemudi sudah dikenal. Mengemudikan mobil sewaan di negara baru dengan geografi dan peraturan lalu lintas yang tidak dikenal jauh lebih mengasyikkan. Gambar 2.6 mencantumkan

properti sejumlah lingkungan yang sudah dikenal. Perhatikan bahwa jawabannya tidak selalu jelas.

Misalnya, kami menggambarkan robot pemetik komponen sebagai episodik, karena biasanya robot tersebut mempertimbangkan setiap komponen secara terpisah. Namun, jika suatu hari ada banyak komponen yang rusak, robot harus belajar dari beberapa pengamatan bahwa distribusi kerusakan telah berubah, dan harus mengubah perilakunya untuk komponen berikutnya. Kami tidak menyertakan kolom "diketahui/tidak diketahui" karena, seperti yang dijelaskan sebelumnya, ini bukan sepenuhnya properti lingkungan.

Lingkungan Tugas	Tampilan	Agen	Deterministik	Episodik	Statis	Diskrit
Teka – Teki Silang	Sepenuhnya	Single	Deterministik	Berurutan	Statis	Diskrit
Catur dengan Jam	Sebagian	Multi	Deterministik	Berurutan	Semi	Diskrit
Poker	Sebagian	Multi	Stokastik	Berurutan	Statis	Diskrit
Bakgamon	Sepenuhnya	Multi	Stokastik	Berurutan	Statis	Diskrit
Driver Taksi	Sebagian	Multi	Stokastik	Berurutan	Dinamis	Kontinu
Diagnosa Medis	Sebagian	Single	Stokastik	Berurutan	Dinamis	Kontinu
Analisis Gambar	Sepenuhnya	Single	Deterministik	Episodik	Semi	Kontinu
Robot pemetik Bagian	Sebagian	Single	Stokastik	Episodik	Dinamis	Kontinu
Pengendali Kilang	Sebagian	Single	Stokastik	Berurutan	Dinamis	Kontinu
Tutor Bahasa Inggris	Sebagian	Multi	Stokastik	Berurutan	Dinamis	Dsikrit

Gambar 2.6 Contoh lingkungan tugas dan karakteristiknya.

Untuk beberapa lingkungan, seperti catur dan poker, cukup mudah untuk memberi agen pengetahuan lengkap tentang aturan, tetapi tetap menarik untuk mempertimbangkan bagaimana agen dapat belajar memainkan permainan ini tanpa pengetahuan tersebut.

Beberapa jawaban dalam tabel bergantung pada bagaimana lingkungan tugas didefinisikan. Kami telah mencantumkan tugas diagnosis medis sebagai agen tunggal karena proses penyakit pada pasien tidak dimodelkan secara menguntungkan sebagai agen; tetapi sistem diagnosis medis mungkin juga harus berurusan dengan pasien yang membangkang dan staf yang skeptis, sehingga lingkungan tersebut dapat memiliki aspek multiagen. Lebih jauh, diagnosis medis bersifat episodik jika seseorang menganggap tugas tersebut sebagai pemilihan diagnosis yang diberikan daftar gejala; masalahnya berurutan jika tugas tersebut dapat mencakup mengusulkan serangkaian tes, mengevaluasi kemajuan selama pengobatan, dan seterusnya.

Selain itu, banyak lingkungan bersifat episodik pada tingkat yang lebih tinggi daripada tindakan individu agen. Misalnya, turnamen catur terdiri dari serangkaian permainan; Setiap permainan adalah sebuah episode karena (secara umum) kontribusi gerakan dalam satu permainan terhadap kinerja agen secara keseluruhan tidak dipengaruhi oleh gerakan dalam

permainan sebelumnya. Di sisi lain, pengambilan keputusan dalam satu permainan tentu saja berurutan.

Repositori kode yang terkait dengan buku ini (aima.cs.berkeley.edu) mencakup implementasi sejumlah lingkungan, bersama dengan simulator lingkungan tujuan umum yang menempatkan satu atau beberapa agen dalam lingkungan simulasi, mengamati perilaku mereka dari waktu ke waktu, dan mengevaluasinya menurut ukuran kinerja yang diberikan. Eksperimen semacam itu sering kali dilakukan bukan untuk satu lingkungan tetapi untuk banyak lingkungan yang diambil dari kelas lingkungan.

Misalnya, untuk mengevaluasi pengemudi taksi dalam lalu lintas simulasi, kita ingin menjalankan banyak simulasi dengan kondisi lalu lintas, pencahayaan, dan cuaca yang berbeda. Jika kita merancang agen untuk satu skenario, kita mungkin dapat memanfaatkan properti tertentu dari kasus tertentu tetapi mungkin tidak mengidentifikasi desain yang baik untuk mengemudi secara umum. Karena alasan ini, repositori kode juga menyertakan generator lingkungan untuk setiap kelas lingkungan yang memilih lingkungan tertentu (dengan kemungkinan tertentu) untuk menjalankan agen. Misalnya, generator lingkungan vakum menginisialisasi pola kotoran dan lokasi agen secara acak. Kita kemudian tertarik pada kinerja rata-rata agen di kelas lingkungan tersebut. Agen rasional untuk kelas lingkungan tertentu memaksimalkan kinerja rata-rata ini.

2.4 STRUKTUR AGEN

Sejauh ini kita telah membahas agen dengan mendeskripsikan perilaku—tindakan yang dilakukan setelah serangkaian persepsi tertentu. Sekarang kita harus mulai dan membahas cara kerja bagian dalamnya. Tugas AI adalah merancang program agen yang mengimplementasikan fungsi agen—pemetaan dari persepsi ke tindakan. Kita berasumsi program ini akan berjalan pada beberapa jenis perangkat komputasi dengan sensor dan aktuator fisik—kita menyebutnya arsitektur:

$$agen = arsitektur + program.$$

Tentu saja, program yang kita pilih harus sesuai dengan arsitektur. Jika program akan merekomendasikan tindakan seperti Berjalan, arsitekturnya sebaiknya memiliki kaki. Arsitekturnya mungkin hanya PC biasa, atau mungkin mobil robotik dengan beberapa komputer, kamera, dan sensor lain di dalamnya. Secara umum, arsitektur membuat persepsi dari sensor tersedia untuk program, menjalankan program, dan memasukkan pilihan tindakan program ke aktuator saat dibuat. Sebagian besar buku ini membahas tentang perancangan program agen,

Program Agen

Semua program agen yang kami rancang dalam buku ini memiliki kerangka yang sama: program tersebut mengambil persepsi saat ini sebagai input dari sensor dan mengembalikan tindakan ke aktuator. Perhatikan perbedaan antara program agen, yang mengambil persepsi saat ini sebagai input, dan fungsi agen, yang mengambil seluruh riwayat persepsi. Program

agen hanya mengambil persepsi saat ini sebagai input karena tidak ada lagi yang tersedia dari lingkungan; jika tindakan agen perlu bergantung pada seluruh rangkaian persepsi, agen harus mengingat persepsi tersebut.

```
function TABLE-DRIVEN-AGENT(percept) returns an action
    persistent: percepts, a sequence, initially empty
                table, a table of actions, indexed by percept sequences, initially fully specified

append percept to the end of percepts
    action ← LOOKUP(percepts, table)
    return action
```

Gambar 2.7 Program TABLE − DRIVEN − AGENT dipanggil untuk setiap persepsi baru dan mengembalikan tindakan setiap kali. Program ini menyimpan rangkaian persepsi lengkap dalam memori.

Kami menjelaskan program agen dalam bahasa pseudocode sederhana yang didefinisikan. (Repositori kode daring berisi implementasi dalam bahasa pemrograman nyata.) Misalnya, Gambar 2.7 menunjukkan program agen yang agak sepele yang melacak rangkaian persepsi dan kemudian menggunakannya untuk mengindeks ke dalam tabel tindakan untuk memutuskan apa yang harus dilakukan. Tabel—yang contohnya diberikan untuk dunia vakum pada Gambar 2.3—secara eksplisit menggambarkan fungsi agen yang diwujudkan oleh program agen. Untuk membangun agen rasional dengan cara ini, kita sebagai desainer harus membangun tabel yang berisi tindakan yang tepat untuk setiap kemungkinan urutan persepsi.

Adalah instruktif untuk mempertimbangkan mengapa pendekatan berbasis tabel untuk konstruksi agen ditakdirkan gagal. Misalkan P adalah kumpulan kemungkinan persepsi dan misalkan T adalah masa pakai agen (jumlah total persepsi yang akan diterimanya). Tabel pencarian akan berisi entri $\sum_{t=1}^{T} |P|^t$. Pertimbangkan taksi otomatis: input visual dari satu kamera masuk dengan kecepatan sekitar 27 megabita per detik (30 bingkai per detik, 640 × 480 piksel dengan 24 bit informasi warna).

Ini memberikan tabel pencarian dengan lebih dari $10^{250.000.000.000}$ entri untuk perjalanan satu jam. Bahkan tabel pencarian untuk catur—bagian kecil dari dunia nyata yang berperilaku baik—akan memiliki setidaknya 10^{150} entri. Ukuran tabel yang menakutkan ini (jumlah atom di alam semesta yang dapat diamati kurang dari 10^{80}) berarti bahwa (a) tidak ada agen fisik di alam semesta ini yang akan memiliki ruang untuk menyimpan tabel, (b) perancang tidak akan punya waktu untuk membuat tabel, (c) tidak ada agen yang dapat mempelajari semua entri tabel yang benar dari pengalamannya, dan (d) bahkan jika lingkungannya cukup sederhana untuk menghasilkan ukuran tabel yang layak, perancang tetap tidak memiliki panduan tentang cara mengisi entri tabel.

Terlepas dari semua ini, AGEN YANG DIGERAKKAN-TABEL memang melakukan apa yang kita inginkan: ia mengimplementasikan fungsi agen yang diinginkan. Tantangan utama bagi AI adalah mencari tahu cara menulis program yang, sejauh mungkin, menghasilkan perilaku

rasional dari program yang agak kecil daripada dari tabel yang sangat besar. Kami memiliki banyak contoh yang menunjukkan bahwa hal ini dapat dilakukan dengan sukses di area lain: misalnya, tabel akar kuadrat besar yang digunakan oleh para insinyur dan anak sekolah sebelum tahun 1970an kini telah digantikan oleh program lima baris untuk metode Newton yang berjalan pada kalkulator elektronik. Pertanyaannya adalah, dapatkah AI melakukan perilaku cerdas umum seperti yang dilakukan Newton untuk akar kuadrat? Kami yakin jawabannya adalah ya.

Di bagian selanjutnya dari bagian ini, kami menguraikan empat jenis dasar program agen yang mewujudkan prinsip-prinsip yang mendasari hampir semua sistem cerdas:

- Agen refleks sederhana;
- Agen refleks berbasis model;
- Agen berbasis tujuan; dan
- Agen berbasis utilitas.

Setiap jenis program agen menggabungkan komponen-komponen tertentu dengan cara-cara tertentu untuk menghasilkan tindakan. Bagian 2.4. menjelaskan secara umum cara mengubah semua agen ini menjadi agen pembelajaran yang dapat meningkatkan kinerja komponen-komponennya sehingga menghasilkan tindakan yang lebih baik. Terakhir, Bagian 2.4 menjelaskan berbagai cara di mana komponen-komponen itu sendiri dapat direpresentasikan dalam agen. Keragaman ini memberikan prinsip pengorganisasian utama untuk bidang ini dan untuk buku itu sendiri.

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action

    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Gambar 2.8 Program agen untuk agen refleks sederhana dalam lingkungan vakum dua-keadaan. Program ini mengimplementasikan fungsi agen yang ditabulasikan dalam Gambar 2.3.

Agen Refleks Sederhana

Jenis agen yang paling sederhana adalah agen refleks sederhana. Agen ini memilih tindakan berdasarkan persepsi saat ini, mengabaikan sisa riwayat persepsi. Misalnya, agen vakum yang fungsi agennya ditabulasikan dalam Gambar 2.3 adalah agen refleks sederhana, karena keputusannya hanya didasarkan pada lokasi saat ini dan apakah lokasi tersebut mengandung kotoran. Program agen untuk agen ini ditunjukkan pada Gambar 2.8.

Perhatikan bahwa program agen vakum memang sangat kecil dibandingkan dengan tabel yang sesuai. Pengurangan yang paling jelas berasal dari mengabaikan riwayat persepsi, yang memangkas jumlah kemungkinan dari 4^T menjadi hanya 4. Pengurangan kecil lebih lanjut berasal dari fakta bahwa ketika kotak saat ini kotor, tindakan tidak bergantung pada lokasi.

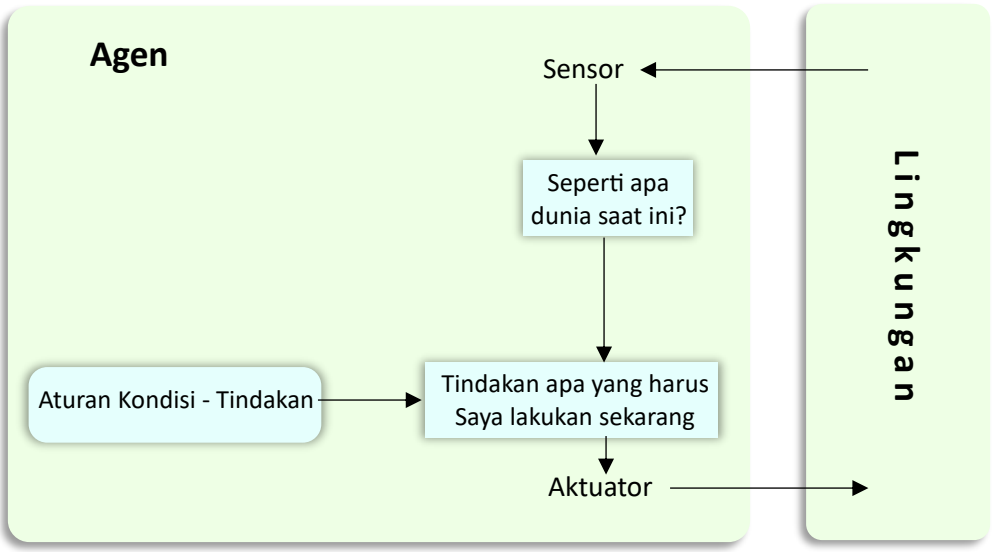
Perilaku refleks sederhana terjadi bahkan di lingkungan yang lebih kompleks. Bayangkan diri Anda sebagai pengemudi taksi otomatis. Jika mobil di depan mengerem dan lampu remnya menyala, maka Anda akan memperhatikan hal ini dan memulai pengereman. Dengan kata lain, beberapa pemrosesan dilakukan pada masukan visual untuk menetapkan kondisi yang kita sebut "Mobil di depan sedang mengerem." Kemudian, ini memicu beberapa koneksi yang ditetapkan dalam program agen ke tindakan "memulai pengereman." Kami menyebut koneksi seperti itu sebagai aturan kondisi-tindakan, yang ditulis sebagai

$$jika\ mobil - di - depan - mengerem\ maka\ mulai - pengereman.$$

Manusia juga memiliki banyak koneksi seperti itu, beberapa di antaranya adalah respons yang dipelajari (seperti untuk mengemudi) dan beberapa di antaranya adalah refleks bawaan (seperti berkedip ketika sesuatu mendekati mata). Dalam buku ini, kami menunjukkan beberapa cara berbeda di mana koneksi tersebut dapat dipelajari dan diimplementasikan. Program pada Gambar 2.8 khusus untuk satu lingkungan vakum tertentu. Pendekatan yang lebih umum dan fleksibel adalah pertama-tama membangun interpreter tujuan umum untuk aturan kondisi-tindakan dan kemudian membuat set aturan untuk lingkungan tugas tertentu.

Gambar 2.9 memberikan struktur program umum ini dalam bentuk skematis, yang menunjukkan bagaimana aturan kondisi-tindakan memungkinkan agen untuk membuat koneksi dari persepsi ke tindakan. (Jangan khawatir jika ini tampak sepele; ini akan menjadi lebih menarik sebentar lagi.) Kami menggunakan persegi panjang untuk menunjukkan status internal terkini dari proses pengambilan keputusan agen, dan oval untuk menunjukkan informasi latar belakang yang digunakan dalam proses tersebut. Program agen, yang juga sangat sederhana, ditunjukkan pada Gambar 2.10.

Fungsi $\mathrm{INTERPRET} - \mathrm{INPUT}$ menghasilkan deskripsi abstrak dari status terkini dari persepsi, dan fungsi $\mathrm{RULE} - \mathrm{MATCH}$ mengembalikan aturan pertama dalam rangkaian aturan yang cocok dengan deskripsi status yang diberikan. Perhatikan bahwa deskripsi dalam hal "aturan" dan "pencocokan" murni bersifat konseptual; implementasi aktual dapat sesederhana kumpulan gerbang logika yang mengimplementasikan rangkaian Boolean.



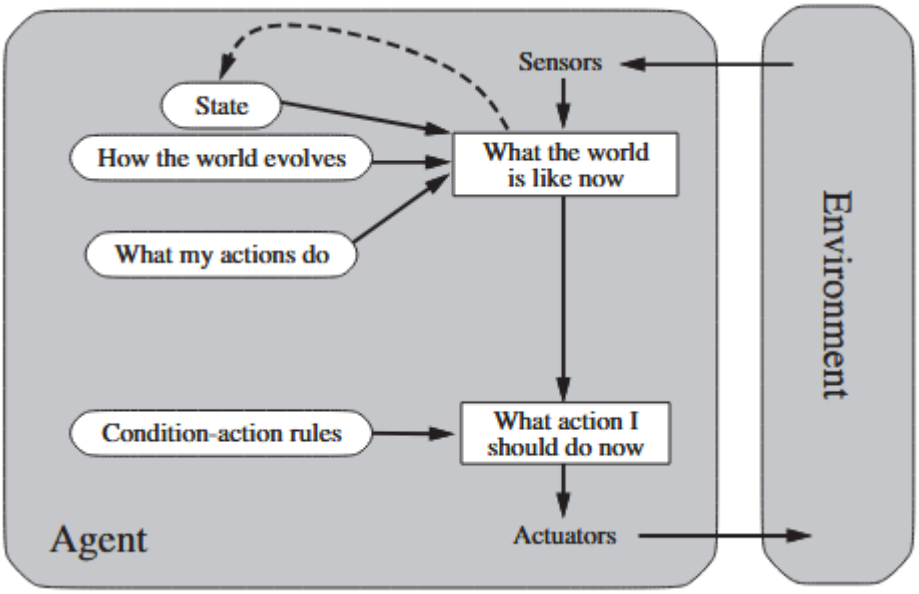
Gambar 2.9 Diagram skema agen refleks sederhana.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
    persistent: rules, a set of condition–action rules

    state ← INTERPRET-INPUT(percept)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
    return action
```

Gambar 2.10 Agen refleks sederhana. Agen ini bertindak berdasarkan aturan yang kondisinya sesuai dengan keadaan saat ini, sebagaimana didefinisikan oleh persepsi.

Agen refleks sederhana memiliki sifat mengagumkan karena sederhana, tetapi ternyata memiliki kecerdasan terbatas. Agen pada Gambar 2.10 akan bekerja hanya jika keputusan yang benar dapat dibuat berdasarkan hanya persepsi terkini—yaitu, hanya jika lingkungan dapat diamati sepenuhnya. Bahkan sedikit ketidakteramatian dapat menyebabkan masalah serius.

Misalnya, aturan pengereman yang diberikan sebelumnya mengasumsikan bahwa kondisi mobil di depan sedang mengerem dapat ditentukan dari persepsi saat ini—satu bingkai video. Ini berfungsi jika mobil di depan memiliki lampu rem yang dipasang di tengah. Sayangnya, model lama memiliki konfigurasi lampu belakang, lampu rem, dan lampu sein yang berbeda, dan tidak selalu memungkinkan untuk mengetahui dari satu gambar apakah mobil sedang mengerem. Agen refleks sederhana yang mengemudi di belakang mobil seperti itu akan mengerem terus-menerus dan tidak perlu, atau, lebih buruk lagi, tidak akan pernah mengerem sama sekali.



Gambar 2.11 Agen refleks berbasis model.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
    persistent: state, the agent's current conception of the world state
                model, a description of how the next state depends on current state and action
                rules, a set of condition–action rules
                action, the most recent action, initially none

    state ← UPDATE-STATE(state, action, percept, model)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
return action
```

Gambar 2.12 Agen refleks berbasis model. Agen ini melacak keadaan dunia saat ini, menggunakan model internal. Kemudian, agen ini memilih tindakan dengan cara yang sama seperti agen refleks.

Kita dapat melihat masalah serupa muncul di dunia vakum. Misalkan agen vakum refleks sederhana tidak memiliki sensor lokasinya dan hanya memiliki sensor kotoran. Agen semacam itu hanya memiliki dua kemungkinan persepsi: [*Kotor*] dan [*Bersih*]. Agen tersebut dapat Menyedot sebagai respons terhadap [*Kotor*]; apa yang harus dilakukannya sebagai respons terhadap [*Bersih*]? Bergerak ke Kiri gagal (selamanya) jika dimulai di kotak A, dan bergerak ke Kanan gagal (selamanya) jika dimulai di kotak B. Loop tak terhingga sering kali tidak dapat dihindari untuk agen refleks sederhana yang beroperasi di lingkungan yang dapat diamati sebagian.

Melarikan diri dari loop tak terhingga dimungkinkan jika agen dapat mengacak tindakannya. Misalnya, jika agen vakum merasakan [*Bersih*], ia mungkin melempar koin untuk memilih antara Kiri dan Kanan. Mudah untuk menunjukkan bahwa agen akan mencapai kotak lainnya dalam rata-rata dua langkah. Kemudian, jika kotak itu kotor, agen akan

membersihkannya dan tugas akan selesai. Oleh karena itu, agen refleks sederhana yang diacak mungkin mengungguli agen refleks sederhana yang deterministik.

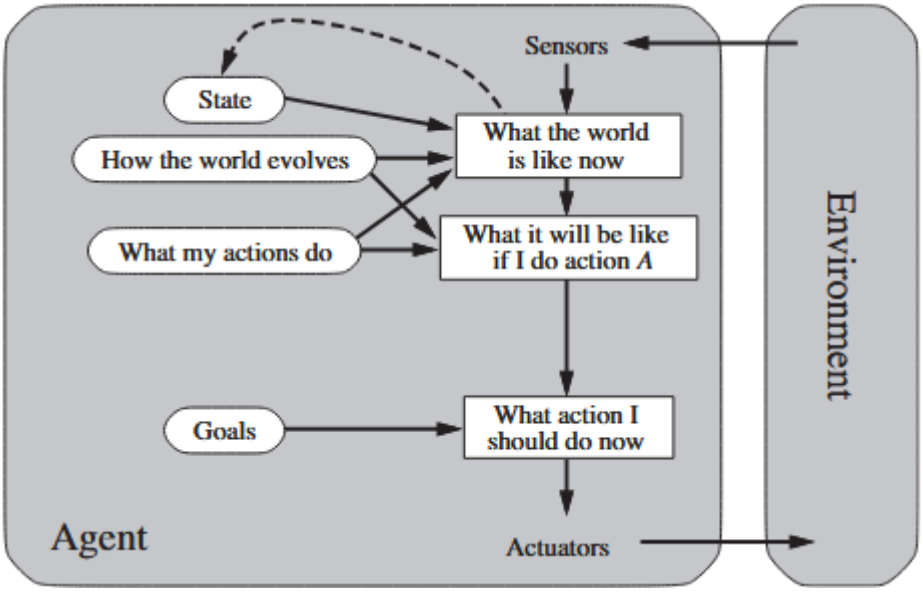
Kami sebutkan di Bagian 2.3 bahwa perilaku acak yang tepat dapat menjadi rasional di beberapa lingkungan multiagen. Di lingkungan agen tunggal, pengacakan biasanya tidak rasional. Ini adalah trik yang berguna yang membantu agen refleks sederhana dalam beberapa situasi, tetapi dalam kebanyakan kasus kita dapat melakukannya jauh lebih baik dengan agen deterministik yang lebih canggih.

Agen Refleks Berbasis Model

Cara paling efektif untuk menangani observabilitas parsial adalah dengan melacak bagian dunia yang tidak dapat dilihatnya sekarang. Artinya, agen harus mempertahankan semacam status internal yang bergantung pada riwayat persepsi dan dengan demikian mencerminkan setidaknya beberapa aspek yang tidak teramati dari status saat ini. Untuk masalah pengereman, status internal tidak terlalu luas—hanya bingkai sebelumnya dari kamera, yang memungkinkan agen mendeteksi saat dua lampu merah di tepi kendaraan menyala atau mati secara bersamaan. Untuk tugas mengemudi lainnya seperti berpindah jalur, agen perlu melacak di mana mobil lain berada jika tidak dapat melihat semuanya sekaligus. Dan agar mengemudi dapat dilakukan, agen perlu melacak di mana kuncinya berada.

Memperbarui informasi status internal ini seiring berjalannya waktu memerlukan dua jenis pengetahuan yang harus dikodekan dalam program agen. Pertama, kita memerlukan beberapa informasi tentang bagaimana dunia berevolusi secara independen dari agen—misalnya, bahwa mobil yang menyalip umumnya akan lebih dekat di belakang daripada beberapa saat yang lalu. Kedua, kita memerlukan beberapa informasi tentang bagaimana tindakan agen itu sendiri memengaruhi dunia—misalnya, bahwa ketika agen memutar roda kemudi searah jarum jam, mobil berbelok ke kanan, atau bahwa setelah berkendara selama lima menit ke utara di jalan bebas hambatan, seseorang biasanya berada sekitar lima mil di utara tempat seseorang berada lima menit yang lalu. Pengetahuan tentang "bagaimana dunia bekerja" ini—baik yang diterapkan dalam rangkaian Boolean sederhana atau dalam teori ilmiah lengkap—disebut model dunia. Agen yang menggunakan model seperti itu disebut agen berbasis model.

Gambar 2.11 memberikan struktur agen refleks berbasis model dengan status internal, yang menunjukkan bagaimana persepsi saat ini digabungkan dengan status internal lama untuk menghasilkan deskripsi terkini dari status saat ini, berdasarkan model agen tentang bagaimana dunia bekerja. Program agen ditunjukkan pada Gambar 2.12. Bagian yang menarik adalah fungsi UPDATE-STATE, yang bertanggung jawab untuk membuat deskripsi status internal baru. Rincian tentang bagaimana model dan status direpresentasikan sangat bervariasi tergantung pada jenis lingkungan dan teknologi tertentu yang digunakan dalam desain agen.



Gambar 2.13 Agen berbasis model dan berbasis tujuan. Agen ini melacak status dunia serta serangkaian tujuan yang ingin dicapainya, dan memilih tindakan yang (pada akhirnya) akan mengarah pada pencapaian tujuan tersebut.

Terlepas dari jenis representasi yang digunakan, jarang mungkin bagi agen untuk menentukan status terkini dari lingkungan yang dapat diamati sebagian secara tepat. Sebaliknya, kotak berlabel "bagaimana dunia sekarang" (Gambar 2.11) mewakili "tebakan terbaik" agen (atau terkadang tebakan terbaik). Misalnya, taksi otomatis mungkin tidak dapat melihat sekeliling truk besar yang berhenti di depannya dan hanya dapat menebak tentang apa yang mungkin menyebabkan penundaan.

Dengan demikian, ketidakpastian tentang status terkini mungkin tidak dapat dihindari, tetapi agen tetap harus membuat keputusan. Poin yang mungkin kurang jelas tentang "keadaan" internal yang dipertahankan oleh agen berbasis model adalah bahwa ia tidak harus menggambarkan "bagaimana dunia saat ini" dalam arti harfiah. Misalnya, taksi mungkin sedang berkendara pulang, dan mungkin ada aturan yang menyuruhnya untuk mengisi bensin dalam perjalanan pulang kecuali jika tangki bensinnya terisi setidaknya setengah.

Meskipun "berkendara pulang" mungkin tampak sebagai aspek dari keadaan dunia, fakta tentang tujuan taksi sebenarnya merupakan aspek dari keadaan internal agen. Jika Anda merasa ini membingungkan, pertimbangkan bahwa taksi tersebut mungkin berada di tempat yang sama persis pada waktu yang sama, tetapi bermaksud untuk mencapai tujuan yang berbeda.

Agen Berbasis Tujuan

Mengetahui sesuatu tentang keadaan lingkungan saat ini tidak selalu cukup untuk memutuskan apa yang harus dilakukan. Misalnya, di persimpangan jalan, taksi dapat berbelok ke kiri, berbelok ke kanan, atau terus melaju lurus. Keputusan yang tepat bergantung pada tujuan taksi. Dengan kata lain, selain deskripsi keadaan saat ini, agen memerlukan semacam informasi tujuan yang menggambarkan situasi yang diinginkan—misalnya, berada di tempat tujuan penumpang. Program agen dapat menggabungkan ini dengan model (informasi yang

sama seperti yang digunakan dalam agen refleks berbasis model) untuk memilih tindakan yang mencapai tujuan. Gambar 2.13 menunjukkan struktur agen berbasis tujuan.

Terkadang pemilihan tindakan berbasis tujuan bersifat langsung—misalnya, ketika kepuasan tujuan langsung diperoleh dari satu tindakan. Terkadang akan lebih rumit—misalnya, ketika agen harus mempertimbangkan rangkaian panjang tikungan dan belokan untuk menemukan cara mencapai tujuan. Pencarian dan perencanaan adalah subbidang AI yang dikhususkan untuk menemukan urutan tindakan yang mencapai tujuan agen. Perhatikan bahwa pengambilan keputusan semacam ini pada dasarnya berbeda dari aturan tindakan kondisi yang dijelaskan sebelumnya, karena melibatkan pertimbangan masa depan—baik "Apa yang akan terjadi jika saya melakukan ini dan itu?" dan "Apakah itu akan membuat saya bahagia?"

Dalam desain agen refleks, informasi ini tidak direpresentasikan secara eksplisit, karena aturan bawaan memetakan langsung dari persepsi ke tindakan. Agen refleks mengerem saat melihat lampu rem. Agen berbasis tujuan, pada prinsipnya, dapat bernalar bahwa jika mobil di depan menyalakan lampu rem, mobil akan melambat. Mengingat cara dunia biasanya berkembang, satu-satunya tindakan yang akan mencapai tujuan untuk tidak menabrak mobil lain adalah mengerem.

Meskipun agen berbasis tujuan tampak kurang efisien, agen ini lebih fleksibel karena pengetahuan yang mendukung keputusannya direpresentasikan secara eksplisit dan dapat dimodifikasi. Jika hujan mulai turun, agen dapat memperbarui pengetahuannya tentang seberapa efektif remnya akan beroperasi; hal ini secara otomatis akan menyebabkan semua perilaku yang relevan diubah agar sesuai dengan kondisi baru.

Di sisi lain, untuk agen refleks, kita harus menulis ulang banyak aturan kondisi-tindakan. Perilaku agen berbasis tujuan dapat dengan mudah diubah untuk menuju tujuan yang berbeda, cukup dengan menentukan tujuan tersebut sebagai tujuan. Aturan agen refleks tentang kapan harus berbelok dan kapan harus lurus hanya akan berfungsi untuk satu tujuan; semuanya harus diganti untuk menuju ke tempat baru.

Agen Berbasis Utilitas

Tujuan saja tidak cukup untuk menghasilkan perilaku berkualitas tinggi di sebagian besar lingkungan. Misalnya, banyak rangkaian tindakan akan membawa taksi ke tujuannya (dengan demikian mencapai tujuan) tetapi beberapa lebih cepat, lebih aman, lebih andal, atau lebih murah daripada yang lain. Sasaran hanya memberikan perbedaan biner kasar antara keadaan "bahagia" dan "tidak bahagia". Ukuran kinerja yang lebih umum seharusnya memungkinkan perbandingan berbagai keadaan dunia menurut seberapa bahagianya hal itu akan membuat agen. Karena "bahagia" tidak terdengar sangat ilmiah, para ekonom dan ilmuwan komputer menggunakan istilah utilitas sebagai gantinya.

Kita telah melihat bahwa ukuran kinerja memberikan skor pada setiap rangkaian keadaan lingkungan tertentu, sehingga dapat dengan mudah membedakan antara cara yang lebih dan kurang diinginkan untuk mencapai tujuan taksi. Fungsi utilitas agen pada dasarnya merupakan internalisasi dari ukuran kinerja. Jika fungsi utilitas internal dan ukuran kinerja

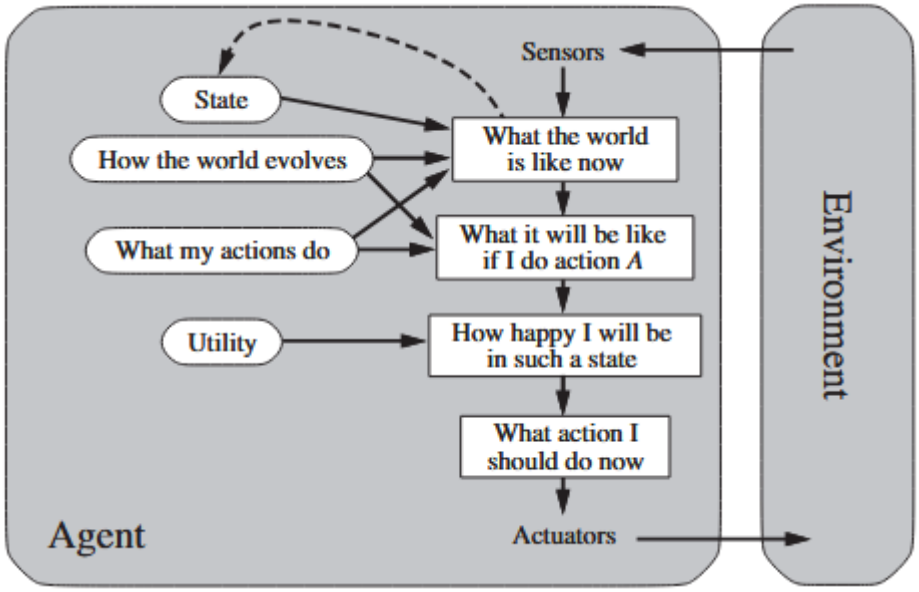
eksternal saling sesuai, maka agen yang memilih tindakan untuk memaksimalkan utilitasnya akan menjadi rasional menurut ukuran kinerja eksternal.

Mari kita tekankan lagi bahwa ini bukan satu-satunya cara untuk menjadi rasional—kita telah melihat program agen rasional untuk dunia vakum (Gambar 2.8) yang tidak tahu apa fungsi utilitasnya—tetapi, seperti agen berbasis tujuan, agen berbasis utilitas memiliki banyak keuntungan dalam hal fleksibilitas dan pembelajaran.

Lebih jauh, dalam dua jenis kasus, tujuan tidak memadai tetapi agen berbasis utilitas masih dapat membuat keputusan rasional. Pertama, ketika ada tujuan yang saling bertentangan, yang hanya beberapa di antaranya dapat dicapai (misalnya, kecepatan dan keselamatan), fungsi utilitas menentukan tradeoff yang sesuai. Kedua, ketika ada beberapa tujuan yang dapat dicapai agen, dan tidak ada satu pun yang dapat dicapai dengan pasti, utilitas menyediakan cara di mana kemungkinan keberhasilan dapat ditimbang terhadap pentingnya tujuan tersebut. Observabilitas parsial dan stokastisitas ada di mana-mana di dunia nyata, dan begitu pula pengambilan keputusan dalam ketidakpastian. Secara teknis, agen berbasis utilitas rasional memilih tindakan yang memaksimalkan utilitas yang diharapkan dari hasil tindakan—yaitu, utilitas yang diharapkan diperoleh agen, secara rata-rata, mengingat probabilitas dan utilitas setiap hasil.

Agen yang memiliki fungsi utilitas eksplisit dapat membuat keputusan rasional dengan algoritme tujuan umum yang tidak bergantung pada fungsi utilitas spesifik yang dimaksimalkan. Dengan cara ini, definisi rasionalitas "global"—yang menetapkan fungsi agen yang memiliki kinerja tertinggi sebagai rasional—diubah menjadi kendala "lokal" pada desain agen rasional yang dapat diekspresikan dalam program sederhana. Struktur agen berbasis utilitas muncul pada Gambar 2.14. Program agen berbasis utilitas, di mana kami merancang agen pembuat keputusan yang harus menangani ketidakpastian yang melekat dalam lingkungan stokastik atau yang dapat diamati sebagian.

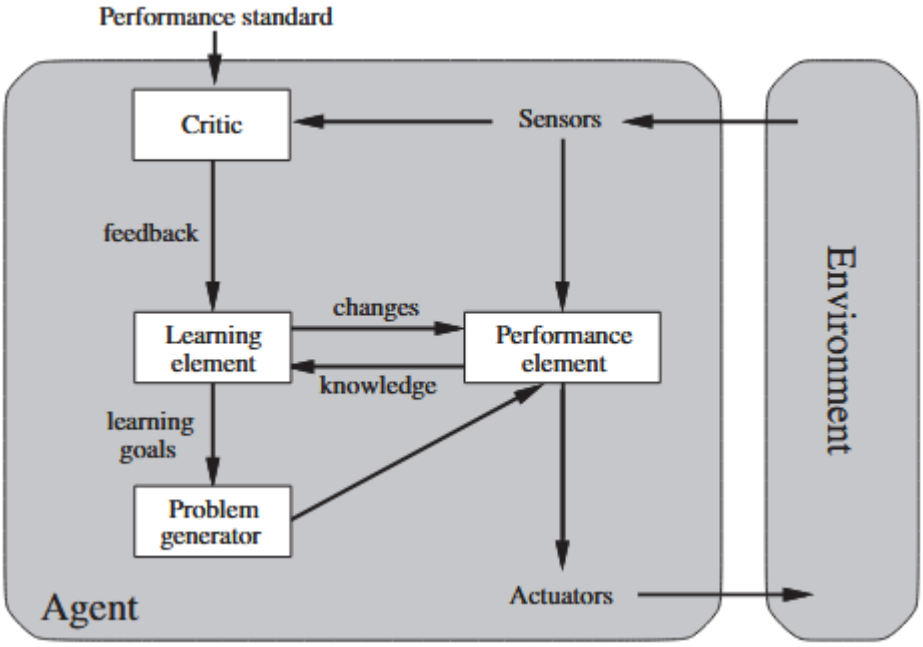
Pada titik ini, pembaca mungkin bertanya-tanya, "Apakah sesederhana itu? Kami hanya membangun agen yang memaksimalkan utilitas yang diharapkan, dan selesai?" Memang benar bahwa agen tersebut akan cerdas, tetapi tidak sederhana. Agen berbasis utilitas harus memodelkan dan melacak lingkungannya, tugas-tugas yang melibatkan banyak penelitian tentang persepsi, representasi, penalaran, dan pembelajaran. Hasil penelitian ini mengisi banyak bab dalam buku ini. Memilih tindakan yang memaksimalkan utilitas juga merupakan tugas yang sulit, yang membutuhkan algoritme cerdik yang mengisi beberapa bab lagi. Bahkan dengan algoritme ini, rasionalitas sempurna biasanya tidak dapat dicapai dalam praktik karena kompleksitas komputasi, seperti yang kami catat di Bab 1.



Gambar 2.14 Agen berbasis model dan utilitas. Agen ini menggunakan model dunia, beserta fungsi utilitas yang mengukur preferensinya di antara berbagai keadaan dunia. Kemudian, agen ini memilih tindakan yang menghasilkan utilitas terbaik yang diharapkan, di mana utilitas yang diharapkan dihitung dengan cara merata-ratakan semua kemungkinan keadaan hasil, yang dibobot dengan probabilitas hasil.

Agen Pembelajaran

Kami telah menjelaskan program agen dengan berbagai metode untuk memilih tindakan. Sejauh ini, kami belum menjelaskan bagaimana program agen terbentuk. Dalam makalah awalnya yang terkenal, Turing mempertimbangkan gagasan untuk benar-benar memprogram mesin cerdasnya secara manual.



Gambar 2.15 Agen pembelajaran umum.

Ia memperkirakan berapa banyak pekerjaan yang mungkin diperlukan dan menyimpulkan, "Beberapa metode yang lebih cepat tampaknya lebih baik." Metode yang ia usulkan adalah membangun mesin pembelajaran dan kemudian mengajarkannya. Di banyak bidang AI, ini sekarang menjadi metode yang lebih disukai untuk menciptakan sistem canggih.

Pembelajaran memiliki keuntungan lain, seperti yang telah kami catat sebelumnya: pembelajaran memungkinkan agen untuk beroperasi di lingkungan yang awalnya tidak dikenal dan menjadi lebih kompeten daripada yang mungkin dimungkinkan oleh pengetahuan awalnya. Di bagian ini, kami secara singkat memperkenalkan gagasan utama agen pembelajaran. Di seluruh buku ini, kami mengomentari peluang dan metode untuk pembelajaran pada jenis agen tertentu. Bagian V membahas lebih dalam tentang algoritme pembelajaran itu sendiri.

Agen pembelajaran dapat dibagi menjadi empat komponen konseptual, seperti yang ditunjukkan pada Gambar 2.15. Perbedaan yang paling penting adalah antara elemen pembelajaran, yang bertanggung jawab untuk melakukan perbaikan, dan elemen kinerja, yang bertanggung jawab untuk memilih tindakan eksternal. Elemen kinerja adalah apa yang sebelumnya kami anggap sebagai keseluruhan agen: elemen ini menerima persepsi dan memutuskan tindakan. Elemen pembelajaran menggunakan umpan balik dari kritikus tentang kinerja agen dan menentukan bagaimana elemen kinerja harus dimodifikasi agar lebih baik di masa mendatang.

Desain elemen pembelajaran sangat bergantung pada desain elemen kinerja. Saat mencoba merancang agen yang mempelajari kemampuan tertentu, pertanyaan pertama bukanlah "Bagaimana saya akan membuatnya mempelajarinya?" tetapi "Elemen kinerja seperti apa yang dibutuhkan agen saya untuk melakukan ini setelah mempelajari caranya?" Dengan desain agen, mekanisme pembelajaran dapat dibangun untuk meningkatkan setiap bagian agen.

Kritikus memberi tahu elemen pembelajaran seberapa baik kinerja agen terhadap standar kinerja yang ditetapkan. Kritikus diperlukan karena persepsi itu sendiri tidak memberikan indikasi keberhasilan agen. Misalnya, program catur dapat menerima persepsi yang menunjukkan bahwa ia telah mengalahkan lawannya, tetapi ia membutuhkan standar kinerja untuk mengetahui bahwa ini adalah hal yang baik; persepsi itu sendiri tidak mengatakannya. Penting agar standar kinerja ditetapkan. Secara konseptual, seseorang harus menganggapnya berada di luar agen sama sekali karena agen tidak boleh mengubahnya agar sesuai dengan perilakunya sendiri.

Komponen terakhir dari agen pembelajaran adalah generator masalah. Ia bertanggung jawab untuk menyarankan tindakan yang akan mengarah pada pengalaman baru dan informatif. Intinya adalah jika elemen kinerja berhasil, ia akan terus melakukan tindakan yang terbaik, berdasarkan apa yang diketahuinya. Namun, jika agen bersedia untuk sedikit mengeksplorasi dan melakukan beberapa tindakan yang mungkin kurang optimal dalam jangka pendek, ia mungkin menemukan tindakan yang jauh lebih baik untuk jangka panjang. Tugas pembuat masalah adalah menyarankan tindakan eksplorasi ini. Inilah yang dilakukan para ilmuwan saat mereka melakukan eksperimen. Galileo tidak menganggap bahwa

menjatuhkan batu dari puncak menara di Pisa itu sendiri merupakan hal yang berharga. Ia tidak mencoba memecahkan batu atau mengubah otak orang yang lewat. Tujuannya adalah mengubah otaknya sendiri dengan mengidentifikasi teori yang lebih baik tentang gerak benda.

Untuk membuat desain keseluruhan lebih konkret, mari kita kembali ke contoh taksi otomatis. Elemen kinerja terdiri dari kumpulan pengetahuan dan prosedur apa pun yang dimiliki taksi untuk memilih tindakan mengemudinya. Taksi keluar ke jalan dan melaju, menggunakan elemen kinerja ini. Kritikus mengamati dunia dan menyampaikan informasi kepada elemen pembelajaran.

Misalnya, setelah taksi berbelok cepat ke kiri melewati tiga lajur lalu lintas, kritikus mengamati bahasa mengejutkan yang digunakan oleh pengemudi lain. Dari pengalaman ini, elemen pembelajaran mampu merumuskan aturan yang menyatakan bahwa ini adalah tindakan yang buruk, dan elemen kinerja dimodifikasi dengan penerapan aturan baru. Pembuat masalah mungkin mengidentifikasi area perilaku tertentu yang perlu ditingkatkan dan menyarankan eksperimen, seperti mencoba rem pada permukaan jalan yang berbeda dalam kondisi yang berbeda.

Elemen pembelajaran dapat membuat perubahan pada komponen "pengetahuan" apa pun yang ditunjukkan dalam diagram agen (Gambar 2.9, 2.11, 2.13, dan 2.14). Kasus yang paling sederhana melibatkan pembelajaran langsung dari urutan persepsi. Pengamatan terhadap pasangan keadaan lingkungan yang berurutan dapat memungkinkan agen untuk mempelajari "Bagaimana dunia berevolusi," dan pengamatan terhadap hasil tindakannya dapat memungkinkan agen untuk mempelajari "Apa yang dilakukan tindakan saya." Misalnya, jika taksi memberikan tekanan pengereman tertentu saat melaju di jalan basah, maka taksi akan segera mengetahui seberapa banyak perlambatan yang sebenarnya dicapai. Jelas, kedua tugas pembelajaran ini lebih sulit jika lingkungan hanya dapat diamati sebagian.

Bentuk pembelajaran dalam paragraf sebelumnya tidak perlu mengakses standar kinerja eksternal—dalam arti tertentu, standar tersebut adalah standar universal untuk membuat prediksi yang sesuai dengan eksperimen. Situasinya sedikit lebih rumit bagi agen berbasis utilitas yang ingin mempelajari informasi utilitas. Misalnya, misalkan agen pengemudi taksi tidak menerima tip dari penumpang yang benar-benar terguncang selama perjalanan.

Standar kinerja eksternal harus memberi tahu agen bahwa hilangnya tip merupakan kontribusi negatif terhadap kinerja keseluruhannya; maka agen mungkin dapat mempelajari bahwa manuver kekerasan tidak berkontribusi pada utilitasnya sendiri. Dalam arti tertentu, standar kinerja membedakan bagian dari persepsi yang masuk sebagai hadiah (atau hukuman) yang memberikan umpan balik langsung pada kualitas perilaku agen. Standar kinerja yang tertanam seperti rasa sakit dan lapar pada hewan dapat dipahami dengan cara ini.

Singkatnya, agen memiliki berbagai komponen, dan komponen tersebut dapat direpresentasikan dalam banyak cara dalam program agen, sehingga tampaknya ada banyak variasi di antara metode pembelajaran. Namun, ada satu tema pemersatu. Pembelajaran pada agen cerdas dapat diringkas sebagai proses modifikasi setiap komponen agen untuk membawa komponen tersebut lebih dekat dengan informasi umpan balik yang tersedia, sehingga meningkatkan kinerja agen secara keseluruhan.

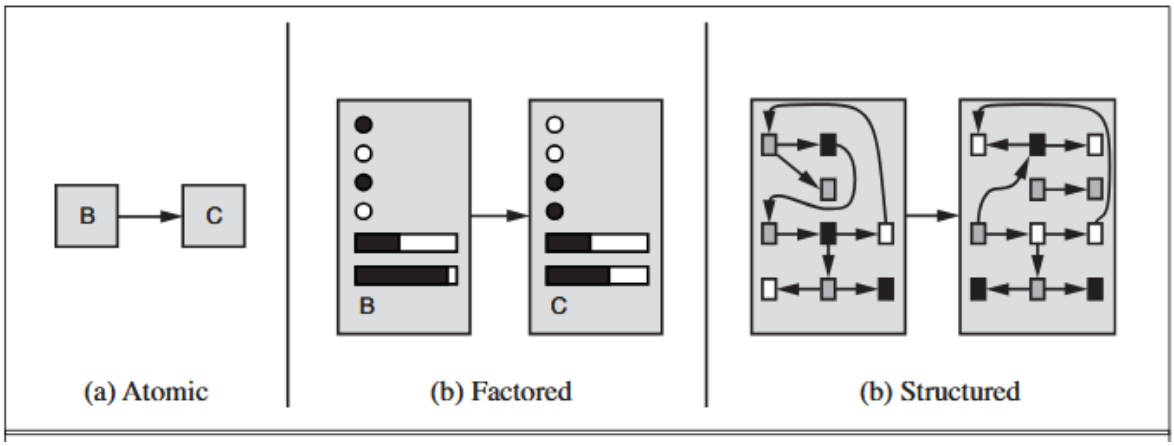
Bagaimana Komponen Program Agen Bekerja

Kami telah menjelaskan program agen (dalam istilah yang sangat sederhana) yang terdiri dari berbagai komponen, yang fungsinya adalah untuk menjawab pertanyaan seperti: "Seperti apa dunia sekarang?" "Tindakan apa yang harus saya lakukan sekarang?" "Apa yang dilakukan tindakan saya?" Pertanyaan berikutnya untuk seorang mahasiswa AI adalah, "Bagaimana komponen-komponen ini bekerja?"

Diperlukan sekitar seribu halaman untuk mulai menjawab pertanyaan itu dengan tepat, tetapi di sini kami ingin menarik perhatian pembaca pada beberapa perbedaan mendasar di antara berbagai cara komponen dapat merepresentasikan lingkungan tempat agen tersebut berada. Secara kasar, kita dapat menempatkan representasi di sepanjang sumbu dengan kompleksitas dan daya ekspresif yang meningkat—atomik, terfaktor, dan terstruktur.

Untuk mengilustrasikan ide-ide ini, ada baiknya untuk mempertimbangkan komponen agen tertentu, seperti yang berhubungan dengan "Apa yang dilakukan tindakan saya." Komponen ini menggambarkan perubahan yang mungkin terjadi di lingkungan sebagai hasil dari tindakan yang diambil, dan Gambar 2.16 memberikan gambaran skematis tentang bagaimana transisi tersebut dapat direpresentasikan. Dalam representasi atom, setiap keadaan dunia tidak dapat dibagi—ia tidak memiliki struktur internal. Pertimbangkan masalah menemukan rute berkendara dari satu ujung negara ke ujung lainnya melalui beberapa rangkaian kota (kami membahas masalah ini pada Gambar 3.2).

Untuk tujuan memecahkan masalah ini, mungkin cukup untuk mereduksi keadaan dunia menjadi hanya nama kota tempat kita berada—satu atom pengetahuan; sebuah "kotak hitam" yang satu-satunya properti yang dapat dikenali adalah identik dengan atau berbeda dari kotak hitam lainnya. Algoritme yang mendasari pencarian dan permainan, model Markov Tersembunyi , dan proses keputusan Markov semuanya bekerja dengan representasi atom—atau, setidaknya, mereka memperlakukan representasi seolah-olah mereka adalah atom.



Gambar 2.16 Tiga cara untuk merepresentasikan status dan transisi di antara status tersebut. (a) Representasi atomik: status (seperti B atau C) adalah kotak hitam tanpa struktur internal; (b) Representasi terfaktor: status terdiri dari vektor nilai atribut; nilai dapat berupa Boolean, bernilai riil, atau salah satu dari sekumpulan simbol yang tetap. (c) Representasi terstruktur:

status mencakup objek, yang masing-masing dapat memiliki atributnya sendiri serta hubungan dengan objek lainnya.

Sekarang pertimbangkan deskripsi fidelitas yang lebih tinggi untuk masalah yang sama, di mana kita perlu memperhatikan lebih dari sekadar lokasi atom di satu kota atau kota lainnya; Kita mungkin perlu memperhatikan berapa banyak gas di dalam tangki, koordinat GPS kita saat ini, apakah lampu peringatan oli berfungsi atau tidak, berapa banyak uang receh yang kita miliki untuk penyeberangan tol, stasiun apa yang ada di radio, dan seterusnya. Representasi yang difaktorkan membagi setiap keadaan menjadi serangkaian variabel atau atribut yang tetap, yang masing-masing dapat memiliki nilai.

Sementara dua keadaan atom yang berbeda tidak memiliki kesamaan—mereka hanyalah kotak hitam yang berbeda—dua keadaan terfaktorkan yang berbeda dapat berbagi beberapa atribut (seperti berada di beberapa lokasi GPS tertentu) dan tidak yang lain (seperti memiliki banyak gas atau tidak memiliki gas); ini membuatnya jauh lebih mudah untuk mengetahui cara mengubah satu keadaan menjadi keadaan lain. Dengan representasi yang difaktorkan, kita juga dapat mewakili ketidakpastian—misalnya, ketidaktahuan tentang jumlah gas di dalam tangki dapat direpresentasikan dengan membiarkan atribut itu kosong. Banyak bidang penting AI didasarkan pada representasi terfaktor, termasuk algoritme pemenuhan kendala (Bab 6), logika proposisional (Bab 7), perencanaan, jaringan Bayesian, dan algoritme pembelajaran mesin.

Untuk banyak tujuan, kita perlu memahami dunia sebagai sesuatu yang saling terkait, bukan sekadar variabel dengan nilai. Misalnya, kita mungkin melihat truk besar di depan kita mundur ke jalan masuk peternakan sapi perah, tetapi seekor sapi lepas dan menghalangi jalan truk. Representasi terfaktor tidak mungkin dilengkapi sebelumnya dengan atribut *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* dengan nilai benar atau salah. Sebaliknya, kita memerlukan representasi terstruktur, di mana objek seperti sapi dan truk serta berbagai hubungan mereka yang bervariasi dapat dijelaskan secara eksplisit. (Lihat Gambar 2.16(c).)

Representasi terstruktur mendasari basis data relasional dan logika orde pertama, model probabilitas orde pertama, pembelajaran berbasis pengetahuan dan sebagian besar pemahaman bahasa alami. Faktanya, hampir semua hal yang diungkapkan manusia dalam bahasa alami menyangkut objek dan hubungan di antara objek tersebut. Seperti yang telah disebutkan sebelumnya, sumbu yang dilalui representasi atomik, terfaktor, dan terstruktur adalah sumbu peningkatan ekspresivitas. Secara kasar, representasi yang lebih ekspresif dapat menangkap, setidaknya secara ringkas, semua hal yang dapat ditangkap oleh representasi yang kurang ekspresif, ditambah beberapa hal lainnya.

Seringkali, bahasa yang lebih ekspresif jauh lebih ringkas; misalnya, aturan catur dapat ditulis dalam satu atau dua halaman bahasa representasi terstruktur seperti logika orde pertama tetapi memerlukan ribuan halaman ketika ditulis dalam bahasa representasi terfaktor seperti logika proposisional. Di sisi lain, penalaran dan pembelajaran menjadi lebih kompleks seiring dengan meningkatnya kekuatan ekspresif representasi. Untuk memperoleh manfaat

representasi ekspresif sekaligus menghindari kekurangannya, sistem cerdas untuk dunia nyata mungkin perlu beroperasi di semua titik sepanjang sumbu secara bersamaan.

2.5 RINGKASAN

Bab ini merupakan semacam tur singkat AI, yang telah kami pahami sebagai ilmu desain agen. Poin-poin utama yang perlu diingat adalah sebagai berikut:

- Agen adalah sesuatu yang merasakan dan bertindak dalam suatu lingkungan. Fungsi agen untuk agen menentukan tindakan yang diambil oleh agen sebagai respons terhadap urutan persepsi apa pun.
- Ukuran kinerja mengevaluasi perilaku agen dalam suatu lingkungan. Agen rasional bertindak untuk memaksimalkan nilai yang diharapkan dari ukuran kinerja, mengingat urutan persepsi yang telah dilihatnya sejauh ini.
- Spesifikasi lingkungan tugas mencakup ukuran kinerja, lingkungan eksternal, aktuator, dan sensor. Dalam mendesain agen, langkah pertama harus selalu menentukan lingkungan tugas selengkap mungkin.
- Lingkungan tugas bervariasi di sepanjang beberapa dimensi penting. Mereka dapat diamati secara penuh atau sebagian, agen tunggal atau multiagen, deterministik atau stokastik, episodik atau berurutan, statis atau dinamis, diskrit atau berkelanjutan, dan diketahui atau tidak diketahui.
- Program agen mengimplementasikan fungsi agen. Ada berbagai desain program agen dasar yang mencerminkan jenis informasi yang dibuat eksplisit dan digunakan dalam proses pengambilan keputusan. Desainnya bervariasi dalam hal efisiensi, kekompakan, dan fleksibilitas. Desain program agen yang tepat bergantung pada sifat lingkungan.
- Agen refleks sederhana merespons persepsi secara langsung, sedangkan agen refleks berbasis model mempertahankan status internal untuk melacak aspek dunia yang tidak terlihat dalam persepsi saat ini. Agen berbasis tujuan bertindak untuk mencapai tujuan mereka, dan agen berbasis utilitas mencoba memaksimalkan "kebahagiaan" yang mereka harapkan.
- Semua agen dapat meningkatkan kinerja mereka melalui pembelajaran.

BAB 3
MENYELESAIKAN MASALAH DENGAN PENCARIAN

Bab ini menjelaskan satu jenis agen berbasis tujuan yang disebut agen pemecahan masalah. Agen pemecahan masalah menggunakan representasi atomik, seperti yang dijelaskan —yaitu, status dunia dianggap sebagai keseluruhan, tanpa struktur internal yang terlihat oleh algoritme pemecahan masalah.

Pembahasan kita tentang pemecahan masalah dimulai dengan definisi yang tepat tentang masalah dan solusinya serta memberikan beberapa contoh untuk mengilustrasikan definisi ini. Kemudian, kami menjelaskan beberapa algoritme pencarian tujuan umum yang dapat digunakan untuk memecahkan masalah ini.

Kita akan melihat beberapa algoritme pencarian tanpa informasi—algoritme yang tidak diberi informasi tentang masalah selain definisinya. Meskipun beberapa algoritme ini dapat memecahkan masalah yang dapat dipecahkan, tidak ada satu pun yang dapat melakukannya secara efisien. Di sisi lain, algoritme pencarian dengan informasi dapat bekerja dengan cukup baik jika diberikan beberapa panduan tentang di mana mencari solusi.

Dalam bab ini, kami membatasi diri pada jenis lingkungan tugas yang paling sederhana, yang solusinya terhadap suatu masalah selalu berupa urutan tindakan yang tetap. Kasus yang lebih umum—di mana tindakan agen di masa mendatang dapat bervariasi tergantung pada persepsi di masa mendatang—dibahas dalam Bab 4.

Bab ini menggunakan konsep kompleksitas asimtotik (yaitu, notasi $O()$) dan kelengkapan NP.

3.1 AGEN PEMECAHAN MASALAH

Agen cerdas diharapkan dapat memaksimalkan ukuran kinerja mereka. Seperti yang telah disebutkan di Bab 2, pencapaian ini terkadang disederhanakan jika agen dapat menetapkan sasaran dan berusaha memenuhinya. Mari kita lihat terlebih dahulu mengapa dan bagaimana agen dapat melakukan ini.

Bayangkan seorang agen di kota Arad, Rumania, sedang menikmati liburan wisata. Ukuran kinerja agen tersebut mencakup banyak faktor: ia ingin memperbaiki warna kulitnya yang kecokelatan, memperbaiki bahasa Rumanianya, menikmati pemandangan, menikmati kehidupan malam (apa adanya), menghindari mabuk, dan sebagainya. Masalah keputusan merupakan masalah yang rumit yang melibatkan banyak pilihan dan pembacaan buku panduan yang cermat. Sekarang, anggaplah agen tersebut memiliki tiket yang tidak dapat dikembalikan untuk terbang dari Bukares keesokan harinya.

Dalam kasus tersebut, masuk akal bagi agen tersebut untuk menetapkan tujuan untuk tiba di Bukares. Tindakan yang tidak mencapai Bukares tepat waktu dapat ditolak tanpa pertimbangan lebih lanjut dan masalah keputusan agen tersebut menjadi sangat sederhana. Sasaran membantu mengatur perilaku dengan membatasi tujuan yang ingin dicapai agen dan dengan demikian tindakan yang perlu dipertimbangkan.

Perumusan sasaran, berdasarkan situasi terkini dan ukuran kinerja agen, merupakan langkah pertama dalam pemecahan masalah. Kita akan menganggap tujuan sebagai serangkaian keadaan dunia—tepatnya keadaan di mana tujuan tersebut terpenuhi. Tugas agen adalah mencari tahu cara bertindak, sekarang dan di masa mendatang, sehingga mencapai keadaan tujuan. Sebelum dapat melakukannya, agen perlu memutuskan (atau kita perlu memutuskan atas namanya) tindakan dan keadaan seperti apa yang harus dipertimbangkan.

Jika agen mempertimbangkan tindakan pada tingkat "maju satu inci kaki kiri" atau "putar roda kemudi satu derajat ke kiri", agen mungkin tidak akan pernah menemukan jalan keluar dari tempat parkir, apalagi ke Bukares, karena pada tingkat detail tersebut terdapat terlalu banyak ketidakpastian di dunia dan akan ada terlalu banyak langkah dalam penyelesaian. Perumusan masalah adalah proses memutuskan tindakan dan keadaan apa yang harus dipertimbangkan, dengan tujuan tertentu.

Kita akan membahas proses ini secara lebih rinci nanti. Untuk saat ini, mari kita asumsikan bahwa agen akan mempertimbangkan tindakan pada tingkat berkendara dari satu kota besar ke kota besar lainnya. Oleh karena itu, setiap keadaan sesuai dengan berada di kota tertentu. Agen kita sekarang telah mengadopsi tujuan berkendara ke Bukares dan sedang mempertimbangkan ke mana harus pergi dari Arad. Tiga jalan keluar dari Arad, satu menuju Sibiu, satu menuju Timisoara, dan satu menuju Zerind. Tak satu pun dari jalan ini mencapai tujuan, jadi kecuali agen tersebut familier dengan geografi Rumania, ia tidak akan tahu jalan mana yang harus ditempuh.

Dengan kata lain, agen tersebut tidak akan tahu tindakan mana yang mungkin terbaik, karena ia belum cukup tahu tentang keadaan yang dihasilkan dari setiap tindakan. Jika agen tidak memiliki informasi tambahan—yaitu, jika lingkungan tidak diketahui dalam pengertian yang didefinisikan dalam Bagian 2.3—maka ia tidak punya pilihan selain mencoba salah satu tindakan secara acak. Situasi yang menyedihkan ini dibahas dalam Bab 4. Namun, misalkan agen memiliki peta Rumania. Inti dari peta adalah untuk memberi agen informasi tentang keadaan yang mungkin terjadi dan tindakan yang dapat diambilnya. Agen dapat menggunakan informasi ini untuk mempertimbangkan tahap-tahap selanjutnya dari perjalanan hipotetis melalui masing-masing dari tiga kota, mencoba menemukan perjalanan yang akhirnya sampai ke Bukares.

Setelah menemukan jalur di peta dari Arad ke Bukares, agen dapat mencapai tujuannya dengan melakukan tindakan mengemudi yang sesuai dengan bagian perjalanan. Secara umum, agen dengan beberapa opsi langsung yang nilainya tidak diketahui dapat memutuskan apa yang harus dilakukan dengan terlebih dahulu memeriksa tindakan mendatang yang akhirnya mengarah ke keadaan yang nilainya diketahui.

Untuk lebih spesifik tentang apa yang kami maksud dengan "memeriksa tindakan mendatang", kami harus lebih spesifik tentang sifat lingkungan, sebagaimana didefinisikan dalam Bagian 2.3. Untuk saat ini, kami berasumsi bahwa lingkungan dapat diamati, sehingga agen selalu mengetahui keadaan saat ini. Bagi agen yang mengemudi di Rumania, masuk akal untuk menganggap bahwa setiap kota di peta memiliki tanda yang menunjukkan keberadaannya kepada pengemudi yang datang.

Kami juga berasumsi bahwa lingkungan itu terpisah, jadi pada keadaan apa pun hanya ada sejumlah tindakan yang terbatas untuk dipilih. Hal ini berlaku untuk bernavigasi di Rumania karena setiap kota terhubung ke sejumlah kecil kota lainnya. Kami akan berasumsi bahwa lingkungan itu diketahui, sehingga agen mengetahui keadaan mana yang dicapai oleh setiap tindakan. (Memiliki peta yang akurat sudah cukup untuk memenuhi kondisi ini untuk masalah navigasi.) Akhirnya, kami berasumsi bahwa lingkungan bersifat deterministik, sehingga setiap tindakan memiliki tepat satu hasil.

Dalam kondisi ideal, hal ini berlaku untuk agen di Rumania—artinya jika ia memilih untuk berkendara dari Arad ke Sibiu, ia akan berakhir di Sibiu. Tentu saja, kondisinya tidak selalu ideal, seperti yang kami tunjukkan di Bab 4. Berdasarkan asumsi ini, solusi untuk setiap masalah adalah serangkaian tindakan yang tetap. "Tentu saja!" seseorang mungkin berkata, "Apa lagi yang bisa dilakukan?" Secara umum, ini bisa menjadi strategi percabangan yang merekomendasikan berbagai tindakan di masa mendatang, tergantung pada persepsi yang muncul. Misalnya, dalam kondisi yang kurang ideal, agen mungkin berencana untuk berkendara dari Arad ke Sibiu lalu ke Rimnicu Vilcea, tetapi mungkin juga perlu memiliki rencana darurat jika tiba secara tidak sengaja di Zerind, bukan di Sibiu. Untungnya, jika agen mengetahui keadaan awal dan lingkungannya diketahui dan deterministik, ia tahu persis di mana ia akan berada setelah tindakan pertama dan apa yang akan dirasakannya. Karena hanya satu persepsi yang mungkin terjadi setelah tindakan pertama, solusinya hanya dapat menentukan satu kemungkinan tindakan kedua, dan seterusnya.

Proses mencari serangkaian tindakan yang mencapai tujuan disebut pencarian. Algoritme pencarian mengambil masalah sebagai masukan dan mengembalikan solusi dalam bentuk serangkaian tindakan. Setelah solusi ditemukan, tindakan yang direkomendasikannya dapat dilakukan. Ini disebut fase eksekusi. Jadi, kita memiliki desain sederhana "formulasikan, cari, jalankan" untuk agen, seperti yang ditunjukkan pada Gambar 3.1.

Setelah merumuskan tujuan dan masalah untuk dipecahkan, agen memanggil prosedur pencarian untuk menyelesaikannya. Kemudian agen menggunakan solusi untuk memandu tindakannya, melakukan apa pun yang direkomendasikan solusi sebagai hal berikutnya yang harus dilakukan—biasanya, tindakan pertama dari urutan—dan kemudian menghapus langkah itu dari urutan. Setelah solusi dijalankan, agen akan merumuskan tujuan baru.

Perhatikan bahwa saat agen menjalankan urutan solusi, ia mengabaikan persepsinya saat memilih tindakan karena ia tahu sebelumnya apa yang akan terjadi. Agen yang menjalankan rencananya dengan mata tertutup, bisa dikatakan, harus cukup yakin dengan apa yang sedang terjadi. Ahli teori kontrol menyebut ini sistem loop terbuka, karena mengabaikan persepsi memutus loop antara agen dan lingkungan.

Pertama-tama kami menjelaskan proses formulasi masalah, dan kemudian mencurahkan sebagian besar bab ini untuk berbagai algoritme untuk fungsi SEARCH. Kami tidak membahas cara kerja fungsi UPDATE − STATE dan FORMULATE − GOAL lebih lanjut dalam bab ini.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent:  seq, an action sequence, initially empty
                 state, some description of the current world state
                 goal, a goal, initially null
                 problem, a problem formulation

state ← UPDATE-STATE(state, percept)
if seq is empty then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    if seq = failure then return a null action
action ← FIRST(seq)
seq ← REST(seq)
return action
```

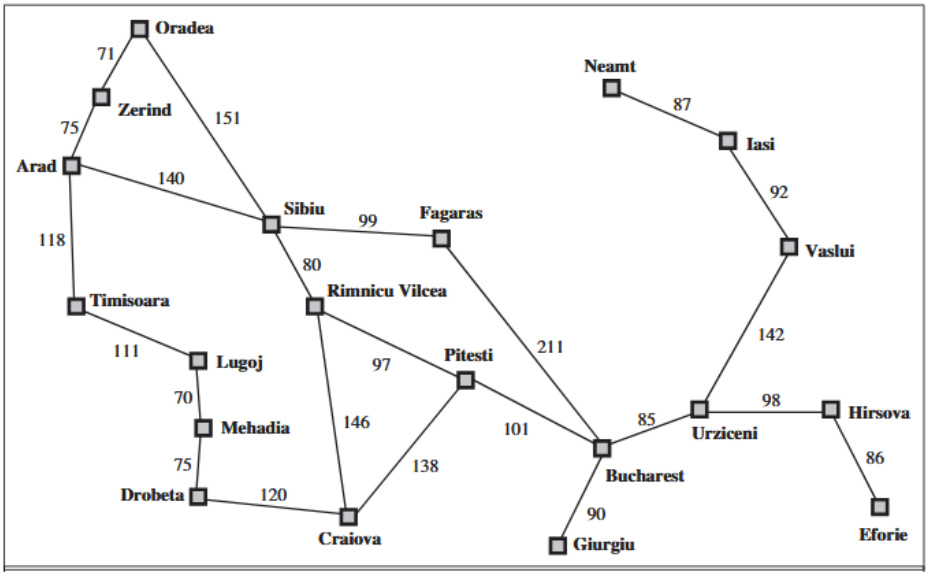
Gambar 3.1 Agen pemecahan masalah sederhana. Pertama-tama ia merumuskan tujuan dan masalah, mencari serangkaian tindakan yang akan menyelesaikan masalah, lalu menjalankan tindakan tersebut satu per satu. Setelah selesai, ia merumuskan tujuan lain dan memulai lagi.

Masalah dan Solusi yang Terdefinisi dengan Baik

Suatu masalah dapat didefinisikan secara formal oleh lima komponen:

- Keadaan awal tempat agen memulai. Misalnya, keadaan awal agen kami di Rumania dapat dideskripsikan sebagai $In(Arad)$.
- Deskripsi tindakan yang mungkin tersedia bagi agen. Diberikan status s tertentu, ACTIONS(s) mengembalikan serangkaian tindakan yang dapat dijalankan di s. Kita katakan bahwa setiap tindakan ini berlaku di s. Misalnya, dari status $In(Arad)$, tindakan yang berlaku adalah $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$.
- Deskripsi tentang apa yang dilakukan setiap tindakan; nama formal untuk ini adalah model transisi, yang ditetapkan oleh fungsi RESULT(s, a) yang mengembalikan status yang dihasilkan dari melakukan tindakan a di status s. Kita juga menggunakan istilah penerus untuk merujuk ke status apa pun yang dapat dicapai dari status tertentu dengan satu tindakan. Misalnya, kita memiliki

$$RESULT(In(Arad), Go(Zerind)) = In(Zerind).$$



Gambar 3.2 Peta jalan sederhana dari sebagian wilayah Rumania.

Bersama-sama, status awal, tindakan, dan model transisi secara implisit mendefinisikan ruang status masalah—serangkaian semua status yang dapat dicapai dari status awal dengan urutan tindakan apa pun. Ruang keadaan membentuk jaringan atau grafik terarah di mana simpul-simpulnya adalah keadaan dan hubungan antara simpul-simpulnya adalah tindakan. (Peta Rumania yang ditunjukkan pada Gambar 3.2 dapat diinterpretasikan sebagai grafik ruang keadaan jika kita melihat setiap jalan sebagai dua tindakan berkendara, satu di setiap arah.) Jalur dalam ruang keadaan adalah urutan keadaan yang dihubungkan oleh urutan tindakan.

- Uji sasaran, yang menentukan apakah keadaan tertentu adalah keadaan tujuan. Terkadang ada serangkaian keadaan tujuan yang mungkin secara eksplisit, dan pengujian tersebut hanya memeriksa apakah keadaan yang diberikan adalah salah satunya. Sasaran agen di Rumania adalah himpunan tunggal $\{Di(Bucharest)\}$. Terkadang tujuan ditentukan oleh properti abstrak, bukan serangkaian status yang disebutkan secara eksplisit. Misalnya, dalam catur, tujuannya adalah mencapai status yang disebut "skakmat", di mana raja lawan diserang dan tidak dapat melarikan diri.
- Fungsi biaya jalur yang menetapkan biaya numerik untuk setiap jalur. Agen pemecahan masalah memilih fungsi biaya yang mencerminkan ukuran kinerjanya sendiri. Bagi agen yang mencoba pergi ke Bukares, waktu adalah hal terpenting, jadi biaya jalur mungkin panjangnya dalam kilometer. Dalam bab ini, kami berasumsi bahwa biaya jalur dapat dijelaskan sebagai jumlah biaya tindakan individual di sepanjang jalur. Biaya langkah untuk mengambil tindakan a dalam status s untuk mencapai status s' dilambangkan dengan $c(s, a, s')$. Biaya langkah untuk Rumania ditunjukkan pada Gambar 3.2 sebagai jarak rute. Kami berasumsi bahwa biaya langkah tidak negatif.

Elemen-elemen sebelumnya mendefinisikan masalah dan dapat dikumpulkan menjadi satu struktur data yang diberikan sebagai masukan ke algoritma pemecahan masalah. Solusi untuk masalah adalah urutan tindakan yang mengarah dari keadaan awal ke keadaan tujuan. Kualitas

solusi diukur dengan fungsi biaya jalur, dan solusi optimal memiliki biaya jalur terendah di antara semua solusi.

Merumuskan Masalah

Pada bagian sebelumnya kami mengusulkan formulasi masalah untuk mencapai Bukares dalam hal keadaan awal, tindakan, model transisi, uji tujuan, dan biaya jalur. Formulasi ini tampaknya masuk akal, tetapi masih merupakan model—deskripsi matematika abstrak—dan bukan hal yang sebenarnya.

Bandingkan deskripsi keadaan sederhana yang telah kami pilih, $In(Arad)$, dengan perjalanan lintas negara yang sebenarnya, di mana keadaan dunia mencakup begitu banyak hal: teman seperjalanan, program radio saat ini, pemandangan di luar jendela, kedekatan petugas penegak hukum, jarak ke tempat peristirahatan berikutnya, kondisi jalan, cuaca, dan sebagainya. Semua pertimbangan ini tidak dimasukkan dalam deskripsi keadaan kita karena tidak relevan dengan masalah menemukan rute ke Bukares. Proses menghilangkan detail dari representasi disebut abstraksi.

Selain mengabstraksikan deskripsi keadaan, kita harus mengabstraksikan tindakan itu sendiri. Tindakan mengemudi memiliki banyak efek. Selain mengubah lokasi kendaraan dan penumpangnya, tindakan itu menghabiskan waktu, mengonsumsi bahan bakar, menghasilkan polusi, dan mengubah agen (seperti yang mereka katakan, perjalanan meluas). Formulasi kita hanya memperhitungkan perubahan lokasi. Selain itu, ada banyak tindakan yang sama sekali tidak kita masukkan: menyalakan radio, melihat ke luar jendela, memperlambat laju kendaraan untuk petugas penegak hukum, dan sebagainya. Dan tentu saja, kita tidak menentukan tindakan pada level "memutar roda kemudi ke kiri sejauh satu derajat."

Dapatkah kita lebih tepat dalam mendefinisikan level abstraksi yang tepat? Pikirkan keadaan dan tindakan abstrak yang telah kita pilih sebagai yang sesuai dengan serangkaian besar keadaan dunia yang terperinci dan urutan tindakan yang terperinci. Sekarang pertimbangkan solusi untuk masalah abstrak: misalnya, jalur dari Arad ke Sibiu ke Rimnicu Vilcea ke Pitesti ke Bucharest. Solusi abstrak ini sesuai dengan sejumlah besar jalur yang lebih terperinci.

Misalnya, kita dapat berkendara dengan radio menyala antara Sibiu dan Rimnicu Vilcea, lalu mematikannya selama sisa perjalanan. Abstraksi tersebut valid jika kita dapat memperluas solusi abstrak apa pun menjadi solusi di dunia yang lebih terperinci; syarat cukupnya adalah bahwa untuk setiap keadaan terperinci yang "di Arad," ada jalur terperinci ke beberapa keadaan yang "di Sibiu," dan seterusnya. Abstraksi berguna jika melaksanakan setiap tindakan dalam solusi lebih mudah daripada masalah awal; dalam kasus ini, tindakan tersebut cukup mudah sehingga dapat dilaksanakan tanpa pencarian atau perencanaan lebih lanjut oleh agen pengemudi rata-rata.

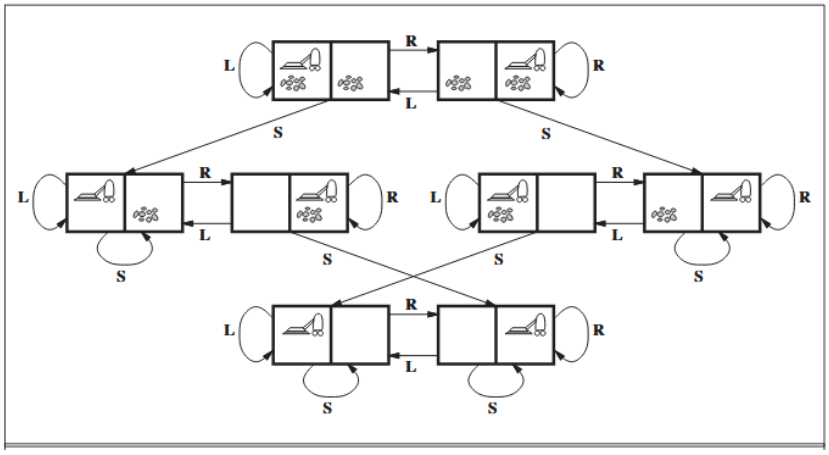
Misalnya, kita dapat berkendara dengan radio menyala antara Sibiu dan Rimnicu Vilcea, lalu mematikannya selama sisa perjalanan. Abstraksi tersebut valid jika kita dapat memperluas solusi abstrak apa pun menjadi solusi di dunia yang lebih terperinci; syarat cukupnya adalah bahwa untuk setiap keadaan terperinci yang "di Arad," ada jalur terperinci ke beberapa keadaan yang "di Sibiu," dan seterusnya. Abstraksi berguna jika melaksanakan setiap tindakan

dalam solusi lebih mudah daripada masalah awal; dalam kasus ini, tindakan tersebut cukup mudah sehingga dapat dilaksanakan tanpa pencarian atau perencanaan lebih lanjut oleh agen pengemudi rata-rata.

Dengan demikian, pemilihan abstraksi yang baik melibatkan penghapusan detail sebanyak mungkin sambil mempertahankan validitas dan memastikan bahwa tindakan abstrak mudah dilaksanakan. Kalau bukan karena kemampuan membangun abstraksi yang berguna, agen cerdas akan benar-benar tenggelam oleh dunia nyata.

3.2 CONTOH MASALAH

Pendekatan pemecahan masalah telah diterapkan pada berbagai macam lingkungan tugas. Kami mencantumkan beberapa yang paling dikenal di sini, dengan membedakan antara masalah mainan dan masalah dunia nyata. Masalah mainan dimaksudkan untuk mengilustrasikan atau melatih berbagai metode pemecahan masalah. Masalah tersebut dapat diberikan deskripsi yang ringkas dan tepat dan karenanya dapat digunakan oleh berbagai peneliti untuk membandingkan kinerja algoritme. Masalah dunia nyata adalah masalah yang solusinya benar-benar diperhatikan orang. Masalah semacam itu cenderung tidak memiliki satu deskripsi yang disepakati, tetapi kita dapat memberikan gambaran umum tentang formulasinya.



Gambar 3.3 Ruang keadaan untuk dunia vakum. Tautan menunjukkan tindakan: L = *Kiri*, R = *Kanan*, S = *Hisap*.

Masalah Mainan

Contoh pertama yang kami periksa adalah dunia vakum yang pertama kali diperkenalkan di Bab 2. (Lihat Gambar 2.2.) Ini dapat diformulasikan sebagai masalah sebagai berikut:

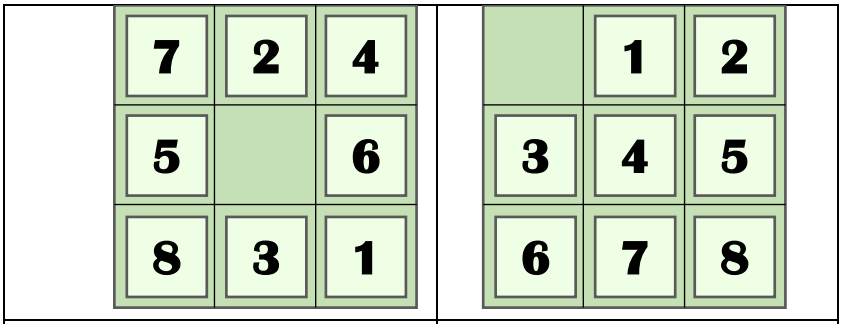
- **Keadaan**: Keadaan ditentukan oleh lokasi agen dan lokasi kotoran. Agen berada di salah satu dari dua lokasi, yang masing-masing mungkin mengandung atau tidak mengandung kotoran. Jadi, ada $2 \times 2^2 = 8$ kemungkinan keadaan dunia. Lingkungan yang lebih besar dengan n lokasi memiliki $n \cdot 2^n$ keadaan.
- **Keadaan awal**: Keadaan apa pun dapat ditetapkan sebagai keadaan awal.

- **Tindakan**: Dalam lingkungan sederhana ini, setiap keadaan hanya memiliki tiga tindakan: *Kiri, Kanan*, dan *Hisap*. Lingkungan yang lebih besar mungkin juga mencakup *Atas* dan *Bawah*.
- **Model transisi**: Tindakan memiliki efek yang diharapkan, kecuali bahwa bergerak ke *Kiri* di kotak paling kiri, bergerak ke *Kanan* di kotak paling kanan, dan Menghisap di kotak bersih tidak memiliki efek. Ruang keadaan lengkap ditunjukkan pada Gambar 3.3.
- **Uji tujuan**: Ini memeriksa apakah semua kotak bersih.
- **Biaya jalur**: Setiap langkah berharga 1, jadi biaya jalur adalah jumlah langkah di jalur tersebut.

Dibandingkan dengan dunia nyata, masalah mainan ini memiliki lokasi yang berbeda, kotoran yang berbeda, pembersihan yang andal, dan tidak akan pernah menjadi lebih kotor. Bab 4 melonggarkan beberapa asumsi ini.

Teka-teki 8, contohnya ditunjukkan pada Gambar 3.4, terdiri dari papan 3 x 3 dengan delapan ubin bernomor dan ruang kosong. Ubin yang berdekatan dengan ruang kosong dapat meluncur ke dalam ruang tersebut. Tujuannya adalah untuk mencapai status tujuan tertentu, seperti yang ditunjukkan di sebelah kanan gambar. Rumusan standarnya adalah sebagai berikut:

- **Status**: Deskripsi status menentukan lokasi masing-masing dari delapan petak dan petak kosong di salah satu dari sembilan petak.
- **Status awal**: Setiap status dapat ditetapkan sebagai status awal. Perhatikan bahwa setiap sasaran yang diberikan dapat dicapai dari tepat setengah dari kemungkinan status awal (Latihan 3.4).
- **Tindakan**: Rumusan paling sederhana mendefinisikan tindakan sebagai gerakan ruang kosong *Kiri, Kanan*, *Atas,* atau *Bawah*. Subset yang berbeda dari ini dimungkinkan tergantung pada di mana petak kosong berada.
- **Model transisi**: Diberikan status dan tindakan, ini mengembalikan status yang dihasilkan; misalnya, jika kita menerapkan *Kiri* ke status awal pada Gambar 3.4, status yang dihasilkan memiliki 5 dan petak kosong yang ditukar.
- **Uji sasaran**: Ini memeriksa apakah status cocok dengan konfigurasi sasaran yang ditunjukkan pada Gambar 3.4. (Konfigurasi sasaran lainnya dimungkinkan.)
- **Biaya jalur**: Setiap langkah berharga 1, jadi biaya jalur adalah jumlah langkah di jalur tersebut.

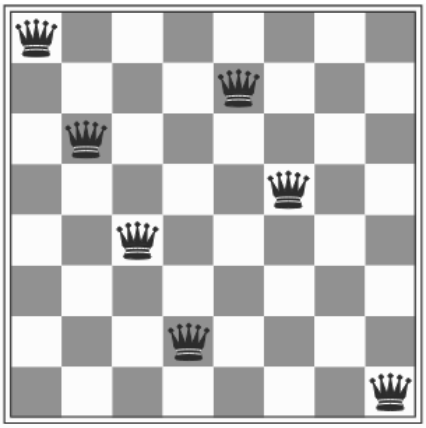


Gambar 3.4 Contoh tipikal teka-teki angka 8.

Abstraksi apa yang telah kita sertakan di sini? Tindakan diabstraksikan ke status awal dan akhir, mengabaikan lokasi antara tempat balok bergeser. Kita telah mengabstraksikan tindakan seperti menggoyangkan papan saat potongan-potongan tersangkut dan mengesampingkan tindakan mengeluarkan potongan-potongan itu dengan pisau dan meletakkannya kembali. Kita hanya memiliki deskripsi aturan teka-teki, menghindari semua detail manipulasi fisik.

Teka-teki 8 termasuk dalam keluarga teka-teki blok geser, yang sering digunakan sebagai masalah uji untuk algoritme pencarian baru dalam AI. Keluarga ini dikenal sebagai NP-lengkap, jadi seseorang tidak berharap menemukan metode yang jauh lebih baik dalam kasus terburuk daripada algoritme pencarian yang dijelaskan dalam bab ini dan bab berikutnya. Teka-teki 8 memiliki $9!/2 = 181.440$ status yang dapat dicapai dan mudah dipecahkan. Teka-teki 15 (pada papan 4×4) memiliki sekitar 1,3 triliun status, dan contoh acak dapat dipecahkan secara optimal dalam beberapa milidetik oleh algoritme pencarian terbaik. Teka-teki 24-bagian (pada papan 5×5) memiliki sekitar 10^{25} keadaan, dan kejadian acak memerlukan waktu beberapa jam untuk diselesaikan secara optimal.

Tujuan dari masalah 8-ratu adalah menempatkan delapan ratu pada papan catur sedemikian rupa sehingga tidak ada ratu yang menyerang yang lain. (Ratu menyerang bidak mana pun di baris, kolom, atau diagonal yang sama.) Gambar 3.5 menunjukkan upaya penyelesaian yang gagal: ratu di kolom paling kanan diserang oleh ratu di kiri atas.



Gambar 3.5 Hampir merupakan solusi untuk masalah 8 ratu. (Solusi dibiarkan sebagai latihan.)

Meskipun ada algoritme tujuan khusus yang efisien untuk masalah ini dan untuk seluruh keluarga n-ratu, masalah ini tetap menjadi masalah pengujian yang berguna untuk algoritme pencarian. Ada dua jenis formulasi utama.

Formulasi inkremental melibatkan operator yang menambah deskripsi status, dimulai dengan status kosong; untuk masalah 8-ratu, ini berarti bahwa setiap tindakan menambahkan ratu ke status tersebut. Formulasi status lengkap dimulai dengan semua 8 ratu di papan dan memindahkannya. Dalam kedua kasus, biaya jalur tidak penting karena hanya status akhir yang diperhitungkan. Formulasi inkremental pertama yang dapat dicoba adalah sebagai berikut:

- **Status**: Setiap susunan 0 hingga 8 ratu di papan adalah status.

- **Status awal**: Tidak ada ratu di papan.
- **Tindakan**: Menambahkan ratu ke kotak kosong mana pun.
- **Model transisi**: Mengembalikan papan dengan ratu yang ditambahkan ke kotak yang ditentukan.
- **Uji tujuan**: 8 ratu ada di papan, tidak ada yang diserang.

Dalam formulasi ini, kita memiliki $64 \cdot 63 \cdots 57 \approx 1{,}8 \times 10^{14}$ kemungkinan urutan untuk diselidiki. Formulasi yang lebih baik akan melarang penempatan ratu di petak mana pun yang sudah diserang:

- **Keadaan**: Semua kemungkinan susunan n ratu $(0 \leq n \leq 8)$, satu per kolom di n kolom paling kiri, tanpa ratu yang menyerang yang lain.
- **Tindakan**: Tambahkan ratu ke petak mana pun di kolom paling kiri yang kosong sehingga tidak diserang oleh ratu lain.

Formulasi ini mengurangi ruang keadaan 8 ratu dari $1{,}8 \times 10^{14}$ menjadi hanya 2.057, dan solusinya mudah ditemukan. Di sisi lain, untuk 100 ratu, pengurangannya adalah dari sekitar 10^{400} keadaan menjadi sekitar 10^{52} keadaan (Latihan 3.5)—peningkatan yang besar, tetapi tidak cukup untuk membuat masalah tersebut dapat dipecahkan. Bagian 4.1 menguraikan formulasi keadaan lengkap, dan Bab 6 memberikan algoritma sederhana yang dapat memecahkan masalah sejuta ratu dengan mudah.

Masalah mainan terakhir kami dirancang oleh Donald Knuth dan menggambarkan bagaimana ruang keadaan tak terbatas dapat muncul. Knuth menduga bahwa, dimulai dengan angka 4, serangkaian operasi faktorial, akar kuadrat, dan dasar akan mencapai bilangan bulat positif yang diinginkan. Misalnya, kita dapat mencapai 5 dari 4 sebagai berikut:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

Definisi masalahnya sangat sederhana:

- **Keadaan**: Angka positif.
- **Keadaan awal**: 4.
- **Tindakan**: Terapkan operasi faktorial, akar kuadrat, atau operasi dasar (faktorial hanya untuk bilangan bulat).
- **Model transisi**: Seperti yang diberikan oleh definisi matematika dari operasi tersebut.
- **Uji tujuan**: Keadaan adalah bilangan bulat positif yang diinginkan.

Sejauh pengetahuan kami, tidak ada batasan seberapa besar angka dapat dibangun dalam proses mencapai target tertentu—misalnya, angka 620.448.401.733.239.439.360.000 dihasilkan dalam ekspresi untuk 5—jadi ruang keadaan untuk masalah ini tidak terbatas.

Ruang keadaan seperti itu sering muncul dalam tugas yang melibatkan pembuatan ekspresi matematika, rangkaian, pembuktian, program, dan objek lain yang didefinisikan secara rekursif.

Masalah di Dunia Nyata

Kita telah melihat bagaimana masalah pencarian rute didefinisikan dalam hal lokasi tertentu dan transisi di sepanjang tautan di antara lokasi tersebut. Algoritme pencarian rute digunakan dalam berbagai aplikasi. Beberapa, seperti situs web dan sistem dalam mobil yang menyediakan petunjuk arah mengemudi, merupakan perluasan yang relatif mudah dari contoh Rumania. Lainnya, seperti perutean aliran video dalam jaringan komputer, perencanaan operasi militer, dan sistem perencanaan perjalanan maskapai, melibatkan spesifikasi yang jauh lebih rumit. Pertimbangkan masalah perjalanan maskapai yang harus dipecahkan oleh situs web perencanaan perjalanan:

- **Negara bagian**: Setiap negara bagian jelas mencakup lokasi (misalnya, bandara) dan waktu saat ini. Lebih jauh, karena biaya tindakan (segmen penerbangan) dapat bergantung pada segmen sebelumnya, basis tarifnya, dan statusnya sebagai domestik atau internasional, negara bagian harus mencatat informasi tambahan tentang aspek "historis" ini.
- **Negara bagian awal**: Ini ditentukan oleh kueri pengguna.
- **Tindakan**: Naik pesawat dari lokasi saat ini, di kelas kursi mana pun, berangkat setelah waktu saat ini, menyisakan cukup waktu untuk transfer di dalam bandara jika diperlukan.
- **Model transisi**: Keadaan yang dihasilkan dari naik pesawat akan memiliki tujuan penerbangan sebagai lokasi saat ini dan waktu kedatangan penerbangan sebagai waktu saat ini.
- **Uji tujuan**: Apakah kita berada di tujuan akhir yang ditentukan oleh pengguna?
- **Biaya jalur**: Ini bergantung pada biaya moneter, waktu tunggu, waktu penerbangan, prosedur bea cukai dan imigrasi, kualitas kursi, waktu, jenis pesawat, penghargaan jarak tempuh frequent-flyer, dan sebagainya.

Sistem saran perjalanan komersial menggunakan rumusan masalah semacam ini, dengan banyak komplikasi tambahan untuk menangani struktur tarif rumit yang diberlakukan oleh maskapai penerbangan. Namun, setiap pelancong berpengalaman tahu bahwa tidak semua perjalanan udara berjalan sesuai rencana. Sistem yang benar-benar bagus harus mencakup rencana darurat—seperti reservasi cadangan pada penerbangan alternatif—sejauh rencana tersebut dibenarkan oleh biaya dan kemungkinan kegagalan rencana awal.

Masalah perjalanan sangat erat kaitannya dengan masalah pencarian rute, tetapi dengan perbedaan penting. Pertimbangkan, misalnya, masalah "Kunjungi setiap kota pada Gambar 3.2 setidaknya satu kali, mulai dan berakhir di Bukares." Seperti halnya pencarian rute, tindakan tersebut sesuai dengan perjalanan antara kota-kota yang berdekatan. Namun, ruang statusnya sangat berbeda. Setiap status harus mencakup tidak hanya lokasi saat ini tetapi juga kumpulan kota yang telah dikunjungi agen. Jadi, status awal adalah $In(Bucharest)$, $Visited(\{Bucharest\})$, status antara yang umum adalah $In(Vaslui)$,

$Visited(\{Bucharest, Urziceni, Vaslui\})$, dan uji sasaran akan memeriksa apakah agen berada di Bucharest dan semua 20 kota telah dikunjungi.

Permasalahan tenaga penjual keliling (TSP) adalah permasalahan perjalanan di mana setiap kota harus dikunjungi tepat satu kali. Tujuannya adalah untuk menemukan perjalanan terpendek. Permasalahan ini dikenal sebagai NP-keras, tetapi sejumlah besar upaya telah dilakukan untuk meningkatkan kemampuan algoritma TSP. Selain merencanakan perjalanan untuk tenaga penjual keliling, algoritma ini telah digunakan untuk tugas-tugas seperti merencanakan pergerakan bor papan sirkuit otomatis dan mesin penyimpanan di lantai pabrik.

Permasalahan tata letak VLSI memerlukan penempatan jutaan komponen dan koneksi pada sebuah chip untuk meminimalkan area, meminimalkan penundaan sirkuit, meminimalkan kapasitansi liar, dan memaksimalkan hasil produksi. Masalah tata letak muncul setelah fase desain logis dan biasanya dibagi menjadi dua bagian: tata letak sel dan perutean saluran. Dalam tata letak sel, komponen primitif sirkuit dikelompokkan ke dalam sel, yang masing-masing menjalankan beberapa fungsi yang dikenal. Setiap sel memiliki tapak tetap (ukuran dan bentuk) dan memerlukan sejumlah koneksi ke setiap sel lainnya.

Tujuannya adalah untuk menempatkan sel pada chip sehingga tidak saling tumpang tindih dan agar ada ruang untuk kabel penghubung yang akan ditempatkan di antara sel. Perutean saluran menemukan rute tertentu untuk setiap kabel melalui celah di antara sel. Masalah pencarian ini sangat rumit, tetapi pasti layak dipecahkan. Nanti di bab ini, kami menyajikan beberapa algoritme yang mampu menyelesaikannya.

Navigasi robot merupakan generalisasi dari masalah pencarian rute yang dijelaskan sebelumnya. Daripada mengikuti serangkaian rute yang terpisah, robot dapat bergerak dalam ruang yang berkesinambungan dengan (pada prinsipnya) serangkaian tindakan dan status yang tak terbatas. Untuk robot melingkar yang bergerak di permukaan datar, ruang pada dasarnya berdimensi dua. Ketika robot memiliki lengan dan kaki atau roda yang juga harus dikontrol, ruang pencarian menjadi berdimensi banyak. Teknik-teknik tingkat lanjut diperlukan hanya untuk membuat ruang pencarian terbatas. Kami meneliti beberapa metode ini di Bab 25. Selain kompleksitas masalah, robot sungguhan juga harus mengatasi kesalahan dalam pembacaan sensor dan kontrol motornya.

Urutan perakitan otomatis objek kompleks oleh robot pertama kali didemonstrasikan oleh FREDDY. Kemajuan sejak saat itu lambat tetapi pasti, sampai pada titik di mana perakitan objek rumit seperti motor listrik layak secara ekonomi. Dalam masalah perakitan, tujuannya adalah untuk menemukan urutan perakitan bagian-bagian dari beberapa objek. Jika urutan yang salah dipilih, tidak akan ada cara untuk menambahkan beberapa bagian nanti dalam urutan tersebut tanpa membatalkan sebagian pekerjaan yang telah dilakukan.

Memeriksa kelayakan suatu langkah dalam urutan tersebut adalah masalah pencarian geometris yang sulit yang terkait erat dengan navigasi robot. Dengan demikian, pembuatan tindakan hukum adalah bagian yang mahal dari urutan perakitan. Setiap algoritme praktis harus menghindari penjelajahan semua kecuali sebagian kecil dari ruang keadaan. Masalah perakitan penting lainnya adalah desain protein, yang tujuannya adalah menemukan urutan

asam amino yang akan dilipat menjadi protein tiga dimensi dengan sifat yang tepat untuk menyembuhkan penyakit tertentu.

3.3 MENCARI SOLUSI

Setelah merumuskan beberapa masalah, sekarang kita perlu menyelesaikannya. Solusi adalah urutan tindakan, jadi algoritme pencarian bekerja dengan mempertimbangkan berbagai kemungkinan urutan tindakan. Kemungkinan urutan tindakan yang dimulai pada status awal membentuk pohon pencarian dengan status awal di akar; cabang adalah tindakan dan simpul sesuai dengan status dalam ruang status masalah. Gambar 3.6 menunjukkan beberapa langkah pertama dalam mengembangkan pohon pencarian untuk menemukan rute dari Arad ke Bukares.

Simpul akar pohon sesuai dengan status awal, $In(Arad)$. Langkah pertama adalah menguji apakah ini status tujuan. (Jelas bukan, tetapi penting untuk memeriksa agar kita dapat memecahkan masalah jebakan seperti "mulai di Arad, sampai di Arad.") Kemudian kita perlu mempertimbangkan untuk mengambil berbagai tindakan. Kita melakukan ini dengan memperluas status saat ini; yaitu, menerapkan setiap tindakan hukum ke status saat ini, sehingga menghasilkan serangkaian status baru. Dalam kasus ini, kita tambahkan tiga cabang dari simpul induk $In(Arad)$ yang mengarah ke tiga simpul anak baru: $In(Sibiu)$, $In(Timisoara)$, dan $In(Zerind)$. Sekarang kita harus memilih mana dari ketiga kemungkinan ini untuk dipertimbangkan lebih lanjut.

Ini adalah inti dari pencarian—menindaklanjuti satu pilihan sekarang dan mengesampingkan yang lain untuk nanti, jika pilihan pertama tidak mengarah ke solusi. Misalkan kita memilih Sibiu terlebih dahulu. Kita periksa untuk melihat apakah itu adalah keadaan tujuan (bukan) dan kemudian kembangkan untuk mendapatkan $In(Arad), In(Fagaras), In(Oradea), dan\ In(RimnicuVilcea)$. Kita kemudian dapat memilih salah satu dari keempat ini atau kembali dan memilih Timisoara atau Zerind.

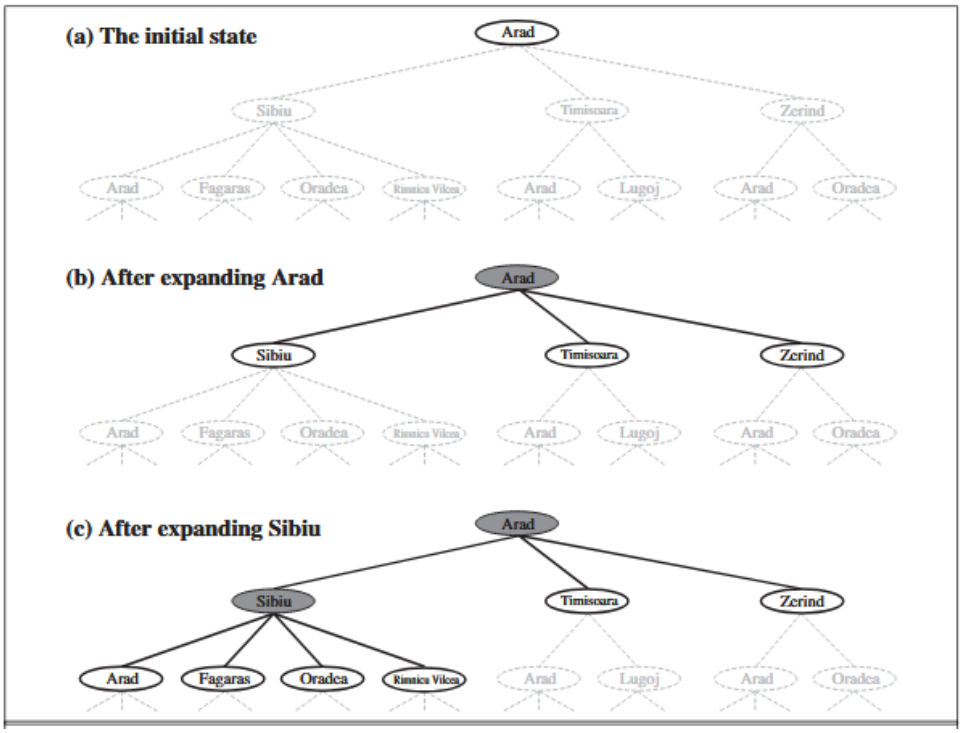
Masing-masing dari keenam simpul ini adalah simpul daun, yaitu simpul tanpa anak di pohon. Kumpulan semua simpul daun yang tersedia untuk perluasan pada titik tertentu disebut batas. (Banyak penulis menyebutnya daftar terbuka, yang secara geografis kurang menggugah dan kurang akurat, karena struktur data lain lebih cocok daripada daftar.) Pada Gambar 3.6, batas setiap pohon terdiri dari simpul-simpul dengan garis tepi tebal. Proses perluasan simpul pada batas tersebut berlanjut hingga solusi ditemukan atau tidak ada lagi status yang perlu diperluas. Algoritma TREE-SEARCH umum ditunjukkan secara informal pada Gambar 3.7. Semua algoritma pencarian memiliki struktur dasar ini; semuanya bervariasi terutama menurut cara mereka memilih status mana yang akan diperluas berikutnya—yang disebut strategi pencarian.

Pembaca yang jeli akan melihat satu hal aneh tentang pohon pencarian yang ditunjukkan pada Gambar 3.6: pohon tersebut mencakup jalur dari Arad ke Sibiu dan kembali ke Arad lagi! Kita katakan bahwa $In(Arad)$ adalah status berulang dalam pohon pencarian, yang dalam kasus ini dihasilkan oleh jalur melingkar. Mempertimbangkan jalur melingkar tersebut berarti bahwa pohon pencarian lengkap untuk Rumania tidak terbatas karena tidak

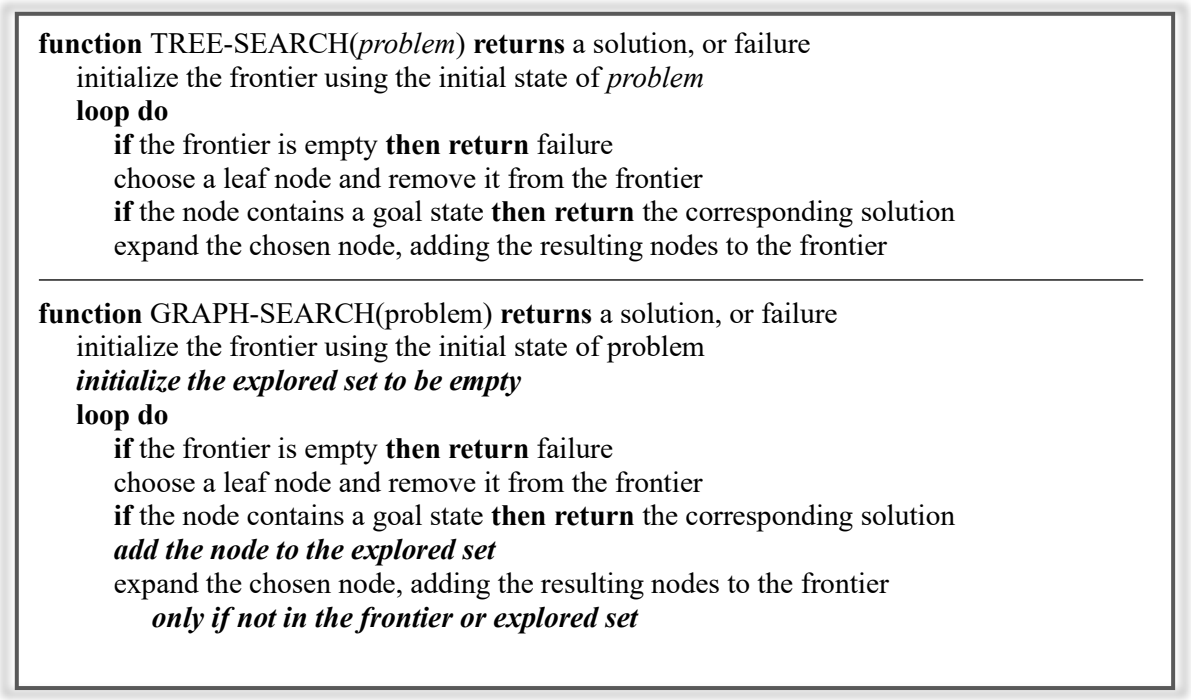
ada batasan seberapa sering seseorang dapat melintasi suatu lingkaran. Di sisi lain, ruang keadaan—peta yang ditunjukkan pada Gambar 3.2—hanya memiliki 20 keadaan. Seperti yang kita bahas di Bagian 3.4, loop dapat menyebabkan algoritma tertentu gagal, sehingga masalah yang dapat dipecahkan menjadi tidak dapat dipecahkan. Untungnya, tidak perlu mempertimbangkan jalur loop. Kita dapat mengandalkan lebih dari sekadar intuisi untuk ini: karena biaya jalur bersifat aditif dan biaya langkah tidak negatif, jalur loop ke keadaan tertentu tidak akan pernah lebih baik daripada jalur yang sama dengan loop yang dihilangkan.

Jalur loop adalah kasus khusus dari konsep jalur redundan yang lebih umum, yang ada setiap kali ada lebih dari satu cara untuk berpindah dari satu keadaan ke keadaan lain. Pertimbangkan jalur Arad–Sibiu (panjang 140 km) dan Arad–Zerind–Oradea–Sibiu (panjang 297 km). Jelas, jalur kedua bersifat redundan—itu hanya cara yang lebih buruk untuk mencapai keadaan yang sama. Jika Anda khawatir tentang pencapaian tujuan, tidak ada alasan untuk menyimpan lebih dari satu jalur ke setiap status tertentu, karena status tujuan apa pun yang dapat dicapai dengan memperluas satu jalur juga dapat dicapai dengan memperluas jalur lainnya.

Dalam beberapa kasus, adalah mungkin untuk mendefinisikan masalah itu sendiri sehingga dapat menghilangkan jalur yang berulang. Misalnya, jika kita merumuskan masalah 8 ratu sehingga ratu dapat ditempatkan di kolom mana pun, maka setiap status dengan n ratu dapat dicapai melalui n! jalur yang berbeda; tetapi jika kita merumuskan kembali masalah tersebut sehingga setiap ratu baru ditempatkan di kolom paling kiri yang kosong, maka setiap status dapat dicapai hanya melalui satu jalur.



Gambar 3.6 Pohon pencarian parsial untuk menemukan rute dari Arad ke Bucharest. Node yang telah diperluas diberi warna; node yang telah dibuat tetapi belum diperluas diberi garis tebal; node yang belum dibuat ditampilkan dalam garis putus-putus samar.



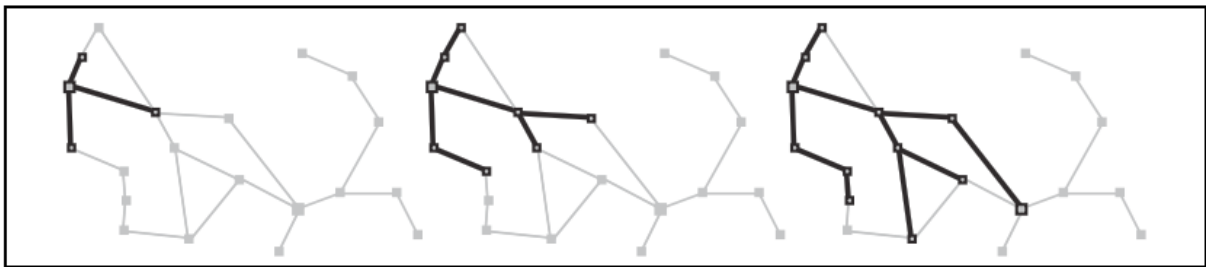
Gambar 3.7 Deskripsi informal dari algoritma pencarian pohon dan pencarian grafik secara umum. Bagian-bagian dari PENCARIAN GRAFIK yang ditandai dengan huruf tebal miring merupakan tambahan yang diperlukan untuk menangani status yang berulang.

Dalam kasus lain, jalur redundan tidak dapat dihindari. Ini mencakup semua masalah yang tindakannya dapat dibalik, seperti masalah pencarian rute dan teka-teki blok geser. Pencarian rute pada kisi persegi panjang (seperti yang digunakan nanti untuk Gambar 3.9) adalah contoh yang sangat penting dalam permainan komputer.

Dalam kisi seperti itu, setiap status memiliki empat penerus, jadi pohon pencarian dengan kedalaman d yang mencakup status berulang memiliki 4^d daun; tetapi hanya ada sekitar $2d^2$ status berbeda dalam d langkah dari status apa pun. Untuk d = 20, ini berarti sekitar satu triliun simpul tetapi hanya sekitar 800 status berbeda. Jadi, mengikuti jalur redundan dapat menyebabkan masalah yang dapat dipecahkan menjadi sulit dipecahkan. Ini berlaku bahkan untuk algoritme yang tahu cara menghindari loop tak terbatas.

Seperti kata pepatah, *algoritme yang melupakan riwayatnya ditakdirkan untuk mengulanginya*. Cara untuk menghindari penjelajahan jalur redundan adalah dengan mengingat di mana seseorang telah berada. Untuk melakukan ini, kami melengkapi algoritma TREE-SEARCH dengan struktur data yang disebut set yang dieksplorasi (juga dikenal sebagai daftar tertutup), yang mengingat setiap simpul yang diperluas.

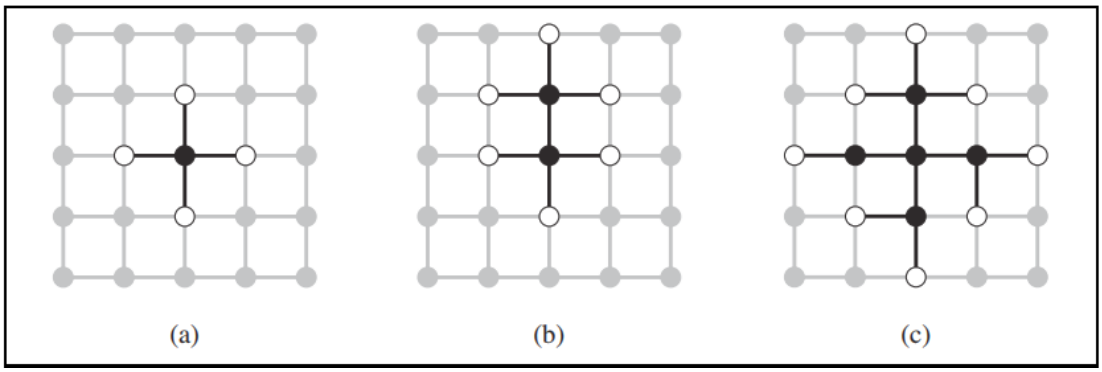
Simpul yang baru dibuat yang cocok dengan simpul yang dibuat sebelumnya—simpul dalam set yang dieksplorasi atau batas wilayah—dapat dibuang alih-alih ditambahkan ke batas wilayah. Algoritma baru, yang disebut GRAPH-SEARCH, ditampilkan secara informal pada Gambar 3.7. Algoritma khusus dalam bab ini mengacu pada desain umum ini.



Gambar 3.8 Urutan pohon pencarian yang dihasilkan oleh pencarian grafik pada masalah Rumania pada Gambar 3.2. Pada setiap tahap, kami telah memperpanjang setiap jalur sebanyak satu langkah. Perhatikan bahwa pada tahap ketiga, kota paling utara (Oradea) telah menjadi jalan buntu: kedua penerusnya telah dieksplorasi melalui jalur lain.

Jelas, pohon pencarian yang dibangun oleh algoritma GRAPH-SEARCH berisi paling banyak satu salinan dari setiap status, sehingga kita dapat menganggapnya sebagai menumbuhkan pohon langsung pada grafik ruang status, seperti yang ditunjukkan pada Gambar 3.8. Algoritma memiliki properti bagus lainnya: batas wilayah memisahkan grafik ruang status menjadi wilayah yang dieksplorasi dan wilayah yang belum dieksplorasi, sehingga setiap jalur dari status awal ke status yang belum dieksplorasi harus melewati status di batas wilayah.

Properti ini diilustrasikan dalam Gambar 3.9. Karena setiap langkah memindahkan suatu keadaan dari perbatasan ke wilayah yang dieksplorasi sambil memindahkan beberapa keadaan dari wilayah yang belum dieksplorasi ke perbatasan, kita melihat bahwa algoritma tersebut secara sistematis memeriksa keadaan di ruang keadaan, satu per satu, hingga menemukan solusi.

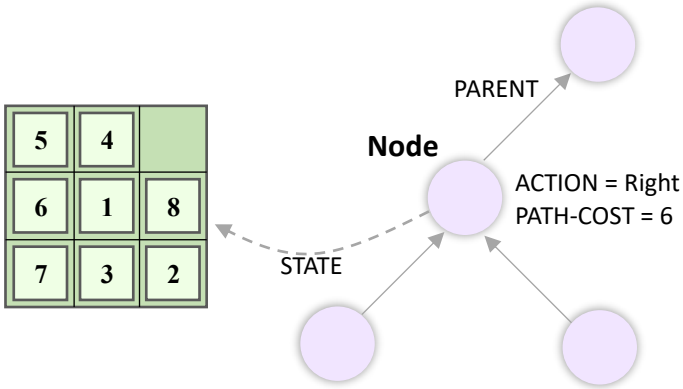


Gambar 3.9 Sifat pemisahan PENCARIAN GRAFIK, diilustrasikan pada masalah kisi persegi panjang. Batas (simpul putih) selalu memisahkan wilayah ruang keadaan yang dieksplorasi (simpul hitam) dari wilayah yang belum dieksplorasi (simpul abu-abu). Pada (a), hanya akar yang telah diperluas. Pada (b), satu simpul daun telah diperluas. Pada (c), penerus akar yang tersisa telah diperluas dalam urutan searah jarum jam.

Infrastruktur untuk Algoritme Penelusuran

Algoritme penelusuran memerlukan struktur data untuk melacak pohon penelusuran yang sedang dibangun. Untuk setiap simpul n dari pohon, kita memiliki struktur yang berisi empat komponen:

- $n.STATE$: status dalam ruang status yang sesuai dengan simpul;
- $n.PARENT$: simpul dalam pohon penelusuran yang menghasilkan simpul ini;
- $n.ACTION$: tindakan yang diterapkan pada induk untuk menghasilkan simpul;
- n.PATH-COST: biaya, yang secara tradisional dilambangkan dengan $g(n)$, dari jalur dari status awal ke simpul, sebagaimana ditunjukkan oleh penunjuk induk.



Gambar 3.10 Node adalah struktur data tempat pohon pencarian dibangun. Setiap node memiliki induk, status, dan berbagai bidang pencatatan. Anak panah menunjuk dari anak ke induk.

Mengingat komponen untuk simpul induk, mudah untuk melihat cara menghitung komponen yang diperlukan untuk simpul anak. Fungsi CHILD-NODE mengambil simpul induk dan tindakan dan mengembalikan simpul anak yang dihasilkan:

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action),
        PARENT = parent, ACTION = action,
        PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Struktur data node digambarkan dalam Gambar 3.10. Perhatikan bagaimana pointer PARENT merangkai node menjadi struktur pohon. Pointer ini juga memungkinkan jalur solusi diekstraksi saat node tujuan ditemukan; kami menggunakan fungsi SOLUTION untuk mengembalikan urutan tindakan yang diperoleh dengan mengikuti pointer parent kembali ke akar.

Hingga saat ini, kami belum terlalu berhati-hati untuk membedakan antara node dan status, tetapi dalam penulisan algoritme terperinci, penting untuk membuat perbedaan itu. Node adalah struktur data pembukuan yang digunakan untuk merepresentasikan pohon pencarian. Status sesuai dengan konfigurasi dunia. Jadi, node berada di jalur tertentu, seperti

yang ditentukan oleh pointer PARENT, sedangkan status tidak. Lebih jauh, dua node yang berbeda dapat berisi status dunia yang sama jika status itu dihasilkan melalui dua jalur pencarian yang berbeda.

Sekarang setelah kami memiliki node, kami perlu tempat untuk menyimpannya. Batas harus disimpan sedemikian rupa sehingga algoritme pencarian dapat dengan mudah memilih node berikutnya untuk diperluas sesuai dengan strategi yang disukainya. Struktur data yang tepat untuk ini adalah antrean. Operasi pada antrian adalah sebagai berikut:

- EMPTY?$(queue)$ mengembalikan true hanya jika tidak ada lagi elemen dalam antrian.
- POP$(queue)$ menghapus elemen pertama dari antrian dan mengembalikannya.
- INSERT$(element, queue)$ memasukkan elemen dan mengembalikan antrian yang dihasilkan.

Antrean dicirikan oleh urutan penyimpanan simpul yang disisipkan. Tiga varian umum adalah antrean masuk pertama, keluar pertama atau FIFO, yang memunculkan elemen antrean tertua; antrean masuk terakhir, keluar pertama atau LIFO (juga dikenal sebagai tumpukan), yang memunculkan elemen antrean terbaru; dan antrean prioritas, yang memunculkan elemen antrean dengan prioritas tertinggi menurut beberapa fungsi pemesanan.

Set yang dieksplorasi dapat diimplementasikan dengan tabel hash untuk memungkinkan pemeriksaan yang efisien untuk status yang berulang. Dengan implementasi yang baik, penyisipan dan pencarian dapat dilakukan dalam waktu yang hampir konstan, tidak peduli berapa banyak status yang disimpan. Seseorang harus berhati-hati untuk mengimplementasikan tabel hash dengan gagasan yang tepat tentang kesetaraan antara status. Misalnya, dalam masalah penjual keliling, tabel hash perlu mengetahui bahwa set kota yang dikunjungi {Bucharest,Urziceni,Vaslui} sama dengan {Urziceni,Vaslui,Bucharest}.

Terkadang hal ini dapat dicapai dengan mudah dengan menegaskan bahwa struktur data untuk status harus dalam beberapa bentuk kanonik; yaitu, status yang secara logis ekuivalen harus dipetakan ke struktur data yang sama. Dalam kasus status yang dijelaskan oleh himpunan, misalnya, representasi bit-vektor atau daftar yang diurutkan tanpa pengulangan akan menjadi kanonik, sedangkan daftar yang tidak diurutkan tidak akan menjadi kanonik.

Mengukur Kinerja Pemecahan Masalah

Sebelum kita masuk ke dalam desain algoritma pencarian tertentu, kita perlu mempertimbangkan kriteria yang mungkin digunakan untuk memilih di antara kriteria tersebut. Kita dapat mengevaluasi kinerja algoritma dengan empat cara:

- **Kelengkapan**: Apakah algoritma dijamin akan menemukan solusi ketika ada solusi?
- **Optimalitas**: Apakah strategi menemukan solusi optimal?
- **Kompleksitas waktu**: Berapa lama waktu yang dibutuhkan untuk menemukan solusi?
- **Kompleksitas ruang**: Berapa banyak memori yang dibutuhkan untuk melakukan pencarian?

Kompleksitas waktu dan ruang selalu dipertimbangkan sehubungan dengan beberapa ukuran kesulitan masalah. Dalam ilmu komputer teoritis, ukuran tipikal adalah ukuran grafik ruang keadaan, $|V| + |E|$, di mana V adalah himpunan titik sudut (simpul) grafik dan E adalah himpunan sisi (tautan). Ini sesuai ketika grafik adalah struktur data eksplisit yang menjadi input

ke program pencarian. (Peta Rumania adalah contohnya.) Dalam AI, grafik sering kali direpresentasikan secara implisit oleh keadaan awal, tindakan, dan model transisi dan sering kali tak terbatas. Karena alasan ini, kompleksitas dinyatakan dalam tiga kuantitas: b, faktor percabangan atau jumlah maksimum penerus dari setiap simpul; d, kedalaman simpul tujuan paling dangkal (yaitu, jumlah langkah di sepanjang jalur dari akar); dan m, panjang maksimum jalur apa pun di ruang keadaan.

Waktu sering kali diukur dalam hal jumlah simpul yang dihasilkan selama pencarian, dan ruang dalam hal jumlah maksimum simpul yang disimpan dalam memori. Untuk sebagian besar, kami menggambarkan kompleksitas waktu dan ruang untuk pencarian di pohon; untuk grafik, jawabannya bergantung pada seberapa "redundan" jalur dalam ruang keadaan. Untuk menilai efektivitas algoritma pencarian, kita dapat mempertimbangkan biaya pencarian saja—yang biasanya bergantung pada kompleksitas waktu tetapi juga dapat mencakup istilah untuk penggunaan memori—atau kita dapat menggunakan biaya total, yang menggabungkan biaya pencarian dan biaya jalur solusi yang ditemukan. Untuk masalah menemukan rute dari Arad ke Bukares, biaya pencarian adalah jumlah waktu yang dibutuhkan oleh pencarian dan biaya solusi adalah total panjang jalur dalam kilometer.

Jadi, untuk menghitung biaya total, kita harus menambahkan milidetik dan kilometer. Tidak ada "nilai tukar resmi" antara keduanya, tetapi mungkin masuk akal dalam kasus ini untuk mengubah kilometer menjadi milidetik dengan menggunakan perkiraan kecepatan rata-rata mobil (karena waktu adalah hal yang penting bagi agen). Hal ini memungkinkan agen untuk menemukan titik tradeoff optimal di mana perhitungan lebih lanjut untuk menemukan jalur yang lebih pendek menjadi kontraproduktif.

3.4 STRATEGI PENCARIAN TANPA INFORMASI

Bagian ini membahas beberapa strategi pencarian yang termasuk dalam judul pencarian tanpa informasi (juga disebut pencarian buta). Istilah ini berarti bahwa strategi tersebut tidak memiliki informasi tambahan tentang status di luar yang disediakan dalam definisi masalah. Yang dapat dilakukannya hanyalah menghasilkan penerus dan membedakan status tujuan dari status non-tujuan. Semua strategi pencarian dibedakan berdasarkan urutan perluasan simpul. Strategi yang mengetahui apakah satu status non-tujuan "lebih menjanjikan" daripada yang lain disebut pencarian dengan informasi atau strategi pencarian heuristik; strategi tersebut dibahas dalam Bagian 3.5.

Pencarian Breadth-First

Pencarian breadth-first adalah strategi sederhana di mana simpul akar diperluas terlebih dahulu, kemudian semua penerus simpul akar diperluas berikutnya, kemudian penerusnya, dan seterusnya. Secara umum, semua simpul diperluas pada kedalaman tertentu di pohon pencarian sebelum simpul apa pun di tingkat berikutnya diperluas. Pencarian breadth-first adalah contoh dari algoritma pencarian grafik umum (Gambar 3.7) di mana simpul yang paling dangkal yang belum dikembangkan dipilih untuk perluasan. Hal ini dicapai dengan sangat mudah dengan menggunakan antrean FIFO untuk perbatasan. Jadi, simpul baru (yang selalu lebih dalam dari induknya) masuk ke bagian belakang antrean, dan simpul lama,

yang lebih dangkal dari simpul baru, akan diperluas terlebih dahulu. Ada satu perubahan kecil pada algoritma pencarian grafik umum, yaitu bahwa pengujian tujuan diterapkan ke setiap simpul saat simpul tersebut dibuat, bukan saat simpul tersebut dipilih untuk perluasan.

Keputusan ini dijelaskan di bawah ini, di mana kami membahas kompleksitas waktu. Perhatikan juga bahwa algoritma tersebut, mengikuti pola umum untuk pencarian grafik, membuang jalur baru apa pun ke status yang sudah ada di perbatasan atau set yang dieksplorasi; mudah untuk melihat bahwa jalur tersebut harus setidaknya sedalam jalur yang sudah ditemukan. Jadi, pencarian breadth-first selalu memiliki jalur paling dangkal ke setiap simpul di perbatasan. Pseudocode diberikan pada Gambar 3.11. Gambar 3.12 menunjukkan kemajuan pencarian pada pohon biner sederhana.

Bagaimana peringkat pencarian breadth-first menurut empat kriteria dari bagian sebelumnya? Kita dapat dengan mudah melihat bahwa pencarian breadth-first sudah lengkap—jika simpul tujuan paling dangkal berada pada kedalaman tertentu d, pencarian breadth-first pada akhirnya akan menemukannya setelah menghasilkan semua simpul yang lebih dangkal (asalkan faktor percabangan b terbatas). Perhatikan bahwa segera setelah simpul tujuan dihasilkan, kita tahu bahwa simpul tersebut adalah simpul tujuan paling dangkal karena semua simpul yang lebih dangkal pasti sudah dihasilkan dan gagal dalam uji tujuan. Sekarang, simpul tujuan paling dangkal belum tentu merupakan simpul yang optimal; secara teknis, pencarian breadth-first optimal jika biaya jalur merupakan fungsi yang tidak menurun dari kedalaman simpul. Skenario yang paling umum adalah bahwa semua tindakan memiliki biaya yang sama.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

    node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?( frontier) then return failure
        node ← POP( frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Gambar 3.11 Pencarian breadth-first pada grafik.

Sejauh ini, berita tentang pencarian breadth-first cukup bagus. Berita tentang waktu dan ruang tidak begitu bagus. Bayangkan mencari pohon seragam di mana setiap status memiliki b penerus. Akar pohon pencarian menghasilkan b simpul pada level pertama, yang masing-masing menghasilkan b simpul lagi, dengan total b^2 pada level kedua.

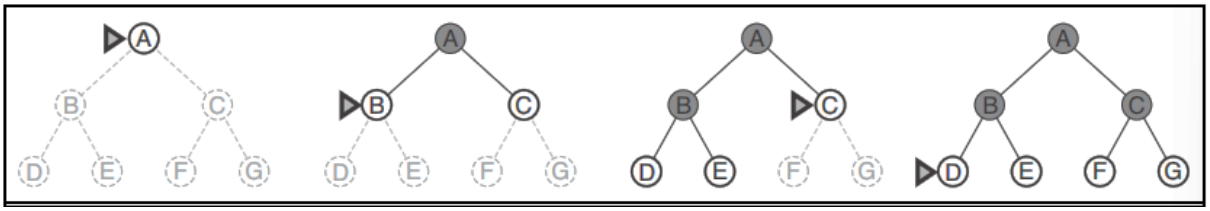
Masing-masing menghasilkan b simpul lagi, menghasilkan b^3 simpul pada level ketiga, dan seterusnya. Sekarang anggaplah solusinya berada pada kedalaman d. Dalam kasus terburuk, itu adalah simpul terakhir yang dihasilkan pada level itu. Maka jumlah total simpul yang dihasilkan adalah

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

(Jika algoritme menerapkan uji sasaran ke simpul saat dipilih untuk perluasan, alih-alih saat dibuat, seluruh lapisan simpul pada kedalaman d akan diperluas sebelum sasaran terdeteksi dan kompleksitas waktu akan menjadi $O(b^{d+1})$

Sedangkan untuk kompleksitas ruang: untuk semua jenis penelusuran grafik, yang menyimpan setiap simpul yang diperluas dalam set yang dieksplorasi, kompleksitas ruang selalu berada dalam faktor b dari kompleksitas waktu. Khususnya untuk penelusuran grafik breadth-first, setiap simpul yang dibuat tetap berada dalam memori.
Akan ada $O(b^{d-1})$ simpul dalam set yang dieksplorasi dan $O(b^d)$ simpul di perbatasan, jadi kompleksitas ruang adalah $O(b^d)$, yaitu, didominasi oleh ukuran perbatasan. Beralih ke penelusuran pohon tidak akan menghemat banyak ruang, dan dalam ruang keadaan dengan banyak jalur redundan, peralihan dapat menghabiskan banyak waktu.

Batasan kompleksitas eksponensial seperti $O(b^d)$ menakutkan. Gambar 3.13 menunjukkan alasannya. Tabel ini mencantumkan, untuk berbagai nilai kedalaman solusi d, waktu dan memori yang diperlukan untuk pencarian breadth-first dengan faktor percabangan $b = 10$. Tabel ini mengasumsikan bahwa 1 juta node dapat dihasilkan per detik dan bahwa sebuah node memerlukan penyimpanan sebesar 1000 byte. Banyak masalah pencarian yang secara kasar sesuai dengan asumsi ini (dengan faktor kurang lebih 100) ketika dijalankan pada komputer pribadi modern.



Gambar 3.12 Pencarian breadth-first pada pohon biner sederhana. Pada setiap tahap, simpul yang akan diperluas berikutnya ditunjukkan dengan penanda.

Kedalaman	Node		Waktu	Memory	
2	110	0.11	Milisecond	107	Kilobyte
4	11.110	11	Milisecond	10.6	Megabyte
6	10^6	1.1	Detik	1	Gigabyte
8	10^8	2	Menit	103	Gigabyte
10	10^{10}	3	Jam	10	Terabyte
12	10^{12}	13	Hari	1	Petabyte
14	10^{14}	3.5	Tahun	99	Petabyte

16	10^{16}	350 tahun		10 exabyte

Gambar 3.13 Waktu dan kebutuhan memori untuk pencarian breadth-first. Angka yang ditampilkan mengasumsikan faktor percabangan b = 10; 1 juta node/detik; 1000 byte/node.

Dua pelajaran dapat dipetik dari Gambar 3.13. Pertama, kebutuhan memori merupakan masalah yang lebih besar untuk pencarian breadth-first daripada waktu eksekusi. Seseorang mungkin menunggu 13 hari untuk solusi dari masalah penting dengan kedalaman pencarian 12, tetapi tidak ada komputer pribadi yang memiliki petabyte memori yang dibutuhkannya. Untungnya, strategi lain memerlukan lebih sedikit memori.

Pelajaran kedua adalah bahwa waktu masih merupakan faktor utama. Jika masalah Anda memiliki solusi pada kedalaman 16, maka (dengan asumsi kami) akan memakan waktu sekitar 350 tahun untuk pencarian breadth-first (atau pencarian tanpa informasi apa pun) untuk menemukannya. Secara umum, *masalah pencarian kompleksitas eksponensial tidak dapat dipecahkan dengan metode tanpa informasi untuk semua kejadian kecuali yang terkecil.*

Pencarian Biaya Seragam

Jika semua biaya langkah sama, pencarian breadth-first adalah optimal karena selalu memperluas simpul yang paling dangkal dan belum diperluas. Dengan perluasan sederhana, kita dapat menemukan algoritma yang optimal dengan fungsi biaya langkah apa pun. Alih-alih memperluas simpul yang paling dangkal, pencarian biaya seragam memperluas simpul n dengan biaya jalur terendah $g(n)$. Ini dilakukan dengan menyimpan frontier sebagai antrean prioritas yang diurutkan berdasarkan g. Algoritma ditunjukkan pada Gambar 3.14.

Selain pengurutan antrean berdasarkan biaya jalur, ada dua perbedaan signifikan lainnya dari pencarian breadth-first. Yang pertama adalah bahwa pengujian tujuan diterapkan ke simpul saat simpul tersebut dipilih untuk perluasan (seperti dalam algoritma pencarian grafik generik yang ditunjukkan pada Gambar 3.7) dan bukan saat simpul tersebut pertama kali dibuat. Alasannya adalah bahwa simpul tujuan pertama yang dibuat mungkin berada di jalur yang kurang optimal. Perbedaan kedua adalah bahwa pengujian ditambahkan jika jalur yang lebih baik ditemukan ke simpul yang saat ini berada di frontier.

Kedua modifikasi ini berperan dalam contoh yang ditunjukkan pada Gambar 3.15, di mana masalahnya adalah untuk pergi dari Sibiu ke Bukares. Penerus Sibiu adalah Rimnicu Vilcea dan Fagaras, dengan biaya masing-masing 80 dan 99. Node dengan biaya terendah, Rimnicu Vilcea, diperluas berikutnya, menambahkan Pitesti dengan biaya $80 + 97 = 177$. Node dengan biaya terendah sekarang adalah Fagaras, jadi diperluas, menambahkan Bukares dengan biaya $99 + 211 = 310$.

Sekarang node tujuan telah dibuat, tetapi pencarian dengan biaya seragam terus berjalan, memilih Pitesti untuk perluasan dan menambahkan jalur kedua ke Bukares dengan biaya $80 + 97 + 101 = 278$. Sekarang algoritme memeriksa untuk melihat apakah jalur baru ini lebih baik daripada yang lama; ya, jadi yang lama dibuang. Bukares, sekarang dengan biaya-g 278, dipilih untuk perluasan dan solusinya dikembalikan. Mudah untuk melihat bahwa pencarian biaya seragam secara umum optimal. Pertama, kita mengamati bahwa setiap kali pencarian biaya seragam memilih simpul n untuk perluasan, jalur optimal ke simpul tersebut

telah ditemukan. (Jika tidak demikian halnya, harus ada simpul perbatasan lain n' pada jalur optimal dari simpul awal ke n, berdasarkan sifat pemisahan grafik pada Gambar 3.9; menurut definisi, n' akan memiliki biaya g yang lebih rendah daripada n dan akan dipilih terlebih dahulu.)

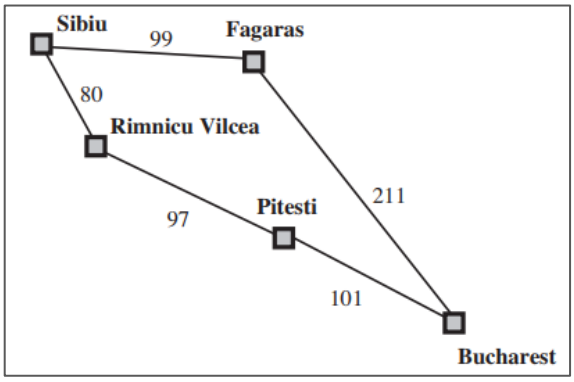
Kemudian, karena biaya langkah tidak negatif, jalur tidak akan pernah menjadi lebih pendek saat simpul ditambahkan. Kedua fakta ini secara bersama-sama menyiratkan bahwa pencarian biaya seragam memperluas simpul dalam urutan biaya jalur optimalnya. Oleh karena itu, simpul tujuan pertama yang dipilih untuk perluasan harus menjadi solusi optimal. Pencarian biaya seragam tidak peduli dengan jumlah langkah yang dimiliki suatu jalur, tetapi hanya dengan total biayanya. Oleh karena itu, algoritma akan terjebak dalam loop tak terbatas jika terdapat jalur dengan urutan tindakan tanpa biaya yang tak terbatas—misalnya, urutan tindakan $NoOp$. Kelengkapan dijamin asalkan biaya setiap langkah melebihi beberapa konstanta positif kecil ϵ.

Pencarian biaya seragam dipandu oleh biaya jalur dan bukan kedalaman, sehingga kompleksitasnya tidak mudah dicirikan dalam hal b dan d. Sebaliknya, misalkan C^* adalah biaya solusi optimal, dan asumsikan bahwa setiap tindakan membutuhkan biaya setidaknya ϵ. Maka kompleksitas waktu dan ruang terburuk dari algoritma tersebut adalah $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, yang dapat jauh lebih besar daripada b^d. Ini karena pencarian biaya seragam dapat menjelajahi pohon besar dengan langkah-langkah kecil sebelum menjelajahi jalur yang melibatkan langkah-langkah besar dan mungkin berguna. Ketika semua biaya langkah sama, $b^{1+\lfloor C^*/\epsilon \rfloor}$ sama dengan b^{d+1}.

Ketika semua biaya langkah sama, penelusuran biaya seragam mirip dengan penelusuran breadth-first, kecuali bahwa penelusuran breadth-first berhenti segera setelah menghasilkan sasaran, sedangkan penelusuran biaya seragam memeriksa semua simpul pada kedalaman sasaran untuk melihat apakah ada yang berbiaya lebih rendah; dengan demikian, penelusuran biaya seragam melakukan lebih banyak pekerjaan dengan memperluas simpul pada kedalaman d secara tidak perlu.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?( frontier) then return failure
    node ← POP( frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Gambar 3.14 Pencarian biaya seragam pada grafik. Algoritme ini identik dengan algoritme pencarian grafik umum pada Gambar 3.7, kecuali penggunaan antrean prioritas dan penambahan pemeriksaan ekstra jika jalur yang lebih pendek ke status batas ditemukan. Struktur data untuk batas perlu mendukung pengujian keanggotaan yang efisien, sehingga harus menggabungkan kemampuan antrean prioritas dan tabel hash.



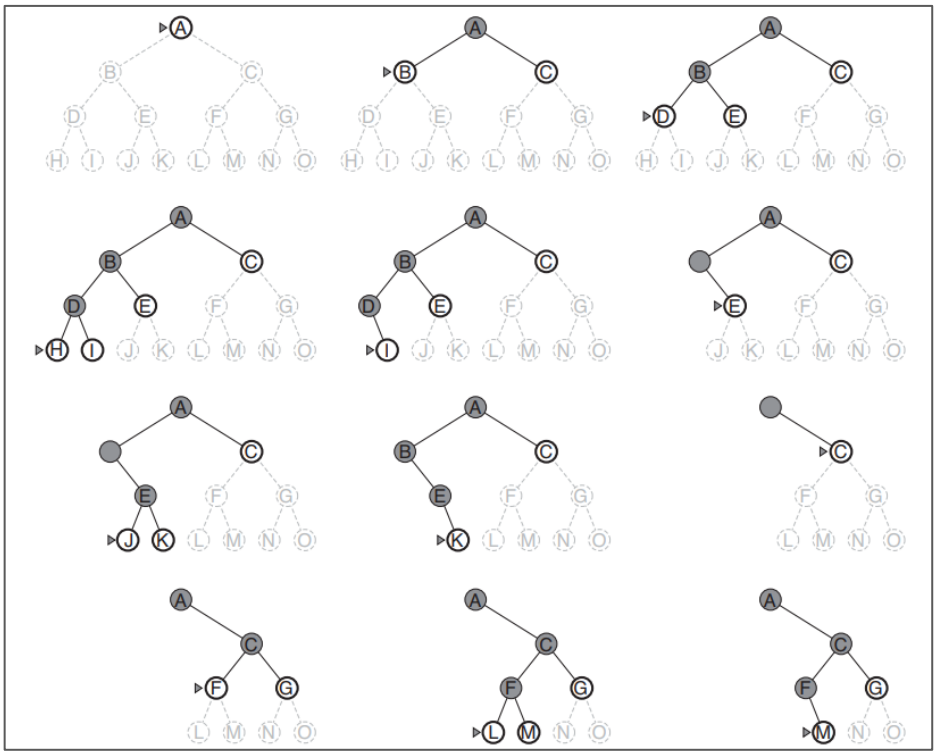
Gambar 3.15 Bagian ruang negara Rumania, dipilih untuk mengilustrasikan pencarian biaya seragam.

Penelusuran Depth-First

Pencarian kedalaman-pertama selalu memperluas simpul terdalam di batas pohon pencarian saat ini. Kemajuan pencarian diilustrasikan dalam Gambar 3.16. Pencarian segera berlanjut ke tingkat terdalam pohon pencarian, di mana simpul-simpul tersebut tidak memiliki penerus. Saat simpul-simpul tersebut diperluas, simpul-simpul tersebut dikeluarkan dari batas, sehingga pencarian "mundur" ke simpul terdalam berikutnya yang masih memiliki penerus yang belum dijelajahi.

Algoritma pencarian kedalaman-pertama adalah contoh dari algoritma pencarian grafik dalam Gambar 3.7; sedangkan pencarian kedalaman-pertama menggunakan antrean FIFO, pencarian kedalaman-pertama menggunakan antrean LIFO. Antrean LIFO berarti bahwa simpul yang paling baru dibuat dipilih untuk perluasan. Ini harus menjadi simpul terdalam yang belum diperluas karena simpul tersebut lebih dalam dari induknya—yang, pada gilirannya, adalah simpul terdalam yang belum diperluas saat dipilih.

Sebagai alternatif untuk implementasi bergaya GRAPH-SEARCH, pencarian depth-first biasanya diimplementasikan dengan fungsi rekursif yang memanggil dirinya sendiri pada setiap anaknya secara bergantian. (Algoritma depth-first rekursif yang menyertakan batas kedalaman ditunjukkan pada Gambar 3.17.)



Gambar 3.16 Pencarian mendalam pada pohon biner. Wilayah yang belum dijelajahi ditampilkan dalam warna abu-abu muda. Node yang dijelajahi tanpa keturunan di perbatasan dihapus dari memori. Node pada kedalaman 3 tidak memiliki penerus dan M adalah satu-satunya node tujuan.

Properti penelusuran mendalam sangat bergantung pada apakah versi penelusuran grafik atau penelusuran pohon digunakan. Versi penelusuran grafik, yang menghindari status berulang dan jalur redundan, lengkap dalam ruang status terbatas karena pada akhirnya akan memperluas setiap simpul. Di sisi lain, versi penelusuran pohon tidak lengkap—misalnya, dalam Gambar 3.6 algoritme akan mengikuti loop Arad–Sibiu–Arad–Sibiu selamanya.
Penelusuran pohon mendalam dapat dimodifikasi tanpa biaya memori tambahan sehingga memeriksa status baru terhadap status pada jalur dari akar ke simpul saat ini; ini menghindari loop tak terbatas dalam ruang status terbatas tetapi tidak menghindari proliferasi jalur redundan. Dalam ruang status tak terbatas, kedua versi gagal jika jalur non-tujuan tak terbatas ditemukan. Misalnya, dalam masalah Knuth 4, penelusuran mendalam akan terus menerapkan operator faktorial selamanya.

Untuk alasan yang sama, kedua versi tersebut tidak optimal. Misalnya, pada Gambar 3.16, pencarian mendalam pertama akan menjelajahi seluruh sub-pohon kiri bahkan jika simpul C adalah simpul tujuan. Jika simpul J juga merupakan simpul tujuan, maka pencarian mendalam pertama akan mengembalikannya sebagai solusi, bukan C, yang akan menjadi solusi yang lebih baik; oleh karena itu, pencarian mendalam pertama tidaklah optimal.

Kompleksitas waktu pencarian grafik depth-first dibatasi oleh ukuran ruang status (yang tentu saja bisa tak terbatas). Di sisi lain, pencarian pohon depth-first dapat menghasilkan semua simpul $O(bm)$ di pohon pencarian, di mana m adalah kedalaman maksimum setiap

simpul; ini bisa jauh lebih besar daripada ukuran ruang status. Perhatikan bahwa m sendiri bisa jauh lebih besar daripada d (kedalaman solusi paling dangkal) dan tak terbatas jika pohon tidak dibatasi.

Sejauh ini, pencarian depth-first tampaknya tidak memiliki keunggulan yang jelas dibandingkan pencarian breadth-first, jadi mengapa kita memasukkannya? Alasannya adalah kompleksitas ruang. Untuk pencarian grafik, tidak ada keuntungan, tetapi pencarian pohon depth-first hanya perlu menyimpan satu jalur dari akar ke simpul daun, bersama dengan simpul saudara yang tersisa yang belum diperluas untuk setiap simpul di jalur tersebut. Setelah simpul diperluas, simpul tersebut dapat dihapus dari memori segera setelah semua turunannya telah dieksplorasi sepenuhnya.

Untuk ruang keadaan dengan faktor percabangan b dan kedalaman maksimum m, penelusuran kedalaman-pertama memerlukan penyimpanan hanya $O(bm)$ simpul. Dengan menggunakan asumsi yang sama seperti pada Gambar 3.13 dan mengasumsikan bahwa simpul pada kedalaman yang sama dengan simpul tujuan tidak memiliki penerus, kami menemukan bahwa penelusuran kedalaman-pertama akan memerlukan 156 kilobyte, bukan 10 eksabita pada kedalaman $d = 16$, faktor ruang yang 7 triliun kali lebih sedikit.

Hal ini telah menyebabkan penerapan penelusuran pohon kedalaman-pertama sebagai pekerja keras dasar dari banyak bidang AI, termasuk kepuasan kendala (Bab 6), kepuasan proposisional (Bab 7), dan pemrograman logika (Bab 9). Untuk sisa bagian ini, kami berfokus terutama pada versi penelusuran pohon dari penelusuran kedalaman-pertama. Varian penelusuran kedalaman-pertama yang disebut penelusuran backtracking menggunakan lebih sedikit memori. (Lihat Bab 6 untuk detail lebih lanjut.) Dalam backtracking, hanya satu penerus yang dihasilkan pada satu waktu, bukan semua penerus; setiap simpul yang diperluas sebagian mengingat penerus mana yang akan dihasilkan berikutnya. Dengan cara ini, hanya diperlukan memori $O(m)$ dan bukan $O(bm)$. Pencarian backtracking memfasilitasi trik lain yang menghemat memori (dan menghemat waktu): gagasan menghasilkan penerus dengan memodifikasi deskripsi status saat ini secara langsung dan bukan menyalinnya terlebih dahulu. Ini mengurangi kebutuhan memori menjadi hanya satu deskripsi status dan tindakan $O(m)$. Agar ini berhasil, kita harus dapat membatalkan setiap modifikasi saat kita kembali untuk menghasilkan penerus berikutnya. Untuk masalah dengan deskripsi status yang besar, seperti perakitan robotik, teknik ini sangat penting untuk keberhasilan.

Pencarian Terbatas Kedalaman

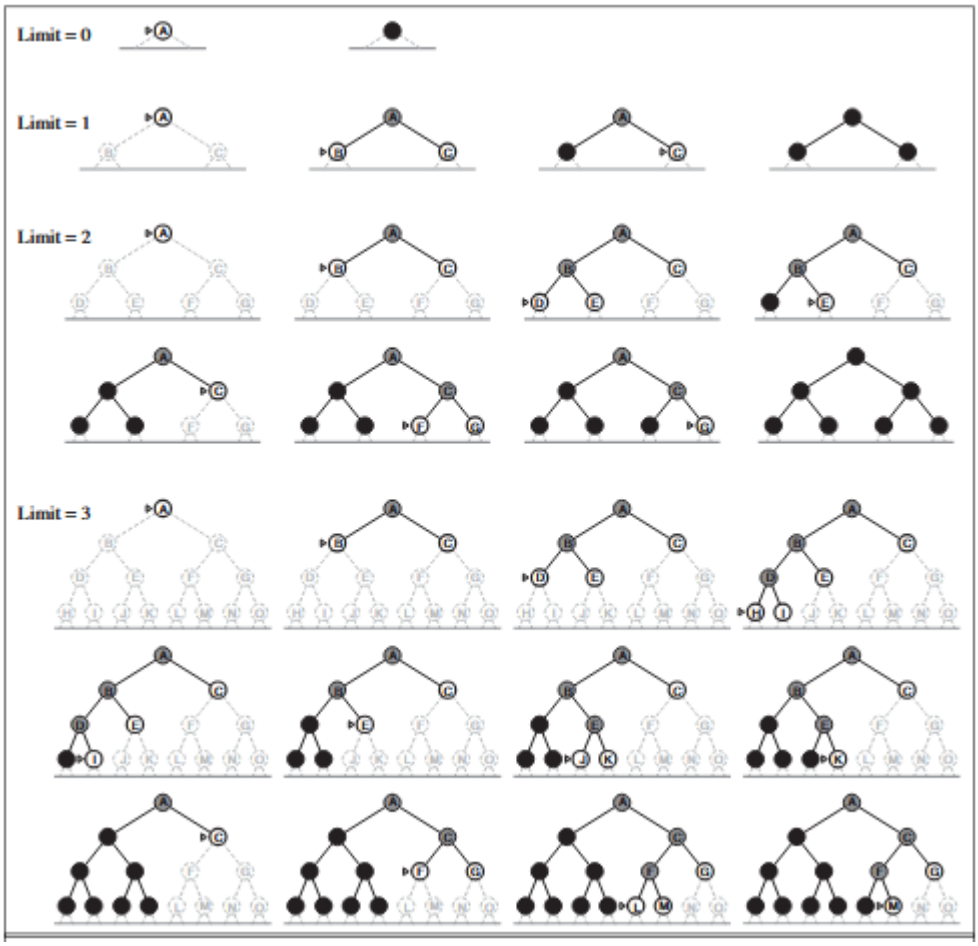
Kegagalan memalukan dari pencarian mendalam pertama dalam ruang keadaan tak terbatas dapat diatasi dengan menyediakan pencarian mendalam pertama dengan batas kedalaman yang telah ditentukan sebelumnya ℓ. Artinya, simpul pada kedalaman ℓ diperlakukan seolah-olah tidak memiliki penerus. Pendekatan ini disebut pencarian terbatas kedalaman. Batas kedalaman memecahkan masalah jalur tak terbatas.

Sayangnya, hal itu juga memperkenalkan sumber ketidaklengkapan tambahan jika kita memilih $\ell < d$, yaitu, tujuan paling dangkal berada di luar batas kedalaman. (Ini mungkin terjadi ketika d tidak diketahui.) Pencarian terbatas kedalaman juga tidak akan optimal jika kita memilih $\ell > d$. Kompleksitas waktunya adalah $O(b^{\ell})$ dan kompleksitas ruangnya adalah

$O(b\ell)$. Pencarian mendalam pertama dapat dilihat sebagai kasus khusus dari pencarian terbatas kedalaman dengan $\ell = \infty$.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? ← false
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit − 1)
            if result = cutoff then cutoff occurred? ← true
            else if result ≠ failure then return result
            if cutoff occurred? then return cutoff else return failure
```

Gambar 3.17 Implementasi rekursif pencarian pohon dengan batasan kedalaman.



Gambar 3.19 Empat iterasi pencarian pendalaman iteratif pada pohon biner.

Terkadang, batas kedalaman dapat didasarkan pada pengetahuan tentang masalah tersebut. Misalnya, pada peta Rumania terdapat 20 kota. Oleh karena itu, kita tahu bahwa jika ada solusi, panjangnya harus 19 paling panjang, jadi $\ell = 19$ adalah pilihan yang memungkinkan. Namun, jika kita mempelajari peta dengan saksama, kita akan menemukan bahwa kota mana pun dapat dicapai dari kota mana pun dalam paling banyak 9 langkah. Angka ini, yang dikenal sebagai diameter ruang keadaan, memberi kita batas kedalaman yang lebih baik, yang mengarah pada pencarian terbatas kedalaman yang lebih efisien. Namun, untuk sebagian besar masalah, kita tidak akan mengetahui batas kedalaman yang baik hingga kita menyelesaikan masalah tersebut.

Pencarian terbatas kedalaman dapat diimplementasikan sebagai modifikasi sederhana pada algoritma pencarian pohon atau grafik umum. Atau, dapat diimplementasikan sebagai algoritma rekursif sederhana seperti yang ditunjukkan pada Gambar 3.17. Perhatikan bahwa pencarian terbatas kedalaman dapat berakhir dengan dua jenis kegagalan: nilai kegagalan standar menunjukkan tidak ada solusi; nilai batas menunjukkan tidak ada solusi dalam batas kedalaman.

Pencarian Mendalam Pertama yang Berulang

Pencarian mendalam pertama yang berulang (atau pencarian mendalam pertama yang berulang) adalah strategi umum, yang sering digunakan dalam kombinasi dengan pencarian pohon mendalam pertama, yang menemukan batas kedalaman terbaik. Hal ini dilakukan dengan meningkatkan batas secara bertahap—pertama 0, kemudian 1, kemudian 2, dan seterusnya—hingga suatu sasaran ditemukan. Hal ini akan terjadi ketika batas kedalaman mencapai d, kedalaman simpul sasaran yang paling dangkal.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Gambar 3.18 Algoritma pencarian mendalam berulang, yang berulang kali menerapkan pencarian terbatas kedalaman dengan batasan yang meningkat. Algoritma ini berakhir saat solusi ditemukan atau jika pencarian terbatas kedalaman menghasilkan kegagalan, yang berarti tidak ada solusi.

Algoritme tersebut ditunjukkan pada Gambar 3.18. Pendalaman berulang menggabungkan manfaat pencarian mendalam pertama dan pencarian luas pertama. Seperti pencarian mendalam pertama, persyaratan memorinya sederhana: $O(bd)$ tepatnya. Seperti pencarian luas pertama, hal ini lengkap ketika faktor percabangan terbatas dan optimal ketika biaya jalur adalah fungsi yang tidak menurun dari kedalaman simpul. Gambar 3.19 menunjukkan empat iterasi PENCARIAN-MENDALAM-ITERATIF pada pohon pencarian biner, di mana solusinya ditemukan pada iterasi keempat.

Pencarian pendalaman berulang mungkin tampak boros karena status dibuat beberapa kali. Ternyata ini tidak terlalu mahal. Alasannya adalah bahwa dalam pohon pencarian dengan

faktor percabangan yang sama (atau hampir sama) di setiap level, sebagian besar simpul berada di level terbawah, jadi tidak terlalu menjadi masalah jika level atas dibuat beberapa kali.

Dalam pencarian pendalaman berulang, simpul di level terbawah (kedalaman d) dibuat satu kali, simpul di level berikutnya dari bawah dibuat dua kali, dan seterusnya, hingga anak dari akar, yang dibuat d kali. Jadi jumlah total simpul yang dibuat dalam kasus terburuk adalah

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

yang memberikan kompleksitas waktu $O(b^d)$—secara asimtotik sama dengan pencarian breadth-first. Ada beberapa biaya tambahan untuk menghasilkan level atas beberapa kali, tetapi tidak besar. Misalnya, jika $b = 10$ dan $d = 5$, angkanya adalah

$$N(\text{IDS}) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$
$$N(\text{BFS}) = 10 + 100 + 1.000 + 10.000 + 100.000 = 111.110$$

Jika Anda benar-benar khawatir tentang pengulangan, Anda dapat menggunakan pendekatan hibrida yang menjalankan penelusuran breadth-first hingga hampir semua memori yang tersedia dikonsumsi, lalu menjalankan pendalaman iteratif dari semua simpul di perbatasan. Secara umum, pendalaman iteratif adalah metode penelusuran tak berinformasi yang lebih disukai ketika ruang penelusuran besar dan kedalaman solusi tidak diketahui.

Penelusuran pendalaman iteratif dianalogikan dengan penelusuran breadth-first karena ia menjelajahi lapisan lengkap simpul baru di setiap iterasi sebelum beralih ke lapisan berikutnya. Tampaknya ada baiknya mengembangkan analog iteratif untuk penelusuran biaya seragam, mewarisi jaminan optimalitas algoritme yang terakhir sambil menghindari persyaratan memorinya. Idenya adalah menggunakan peningkatan batas biaya jalur alih-alih peningkatan batas kedalaman. Algoritme yang dihasilkan, yang disebut penelusuran pemanjangan iteratif, Ternyata, sayangnya, pemanjangan iteratif menimbulkan overhead yang substansial dibandingkan dengan penelusuran biaya seragam.

Pencarian Dua Arah

Ide di balik pencarian dua arah adalah menjalankan dua pencarian simultan—satu maju dari status awal dan yang lainnya mundur dari tujuan—dengan harapan bahwa kedua pencarian bertemu di tengah (Gambar 3.20). Motivasinya adalah bahwa $b^{d/2} + b^{d/2}$ jauh lebih kecil daripada b^d, atau pada gambar, luas dua lingkaran kecil lebih kecil daripada luas satu lingkaran besar yang berpusat di awal dan mencapai tujuan.

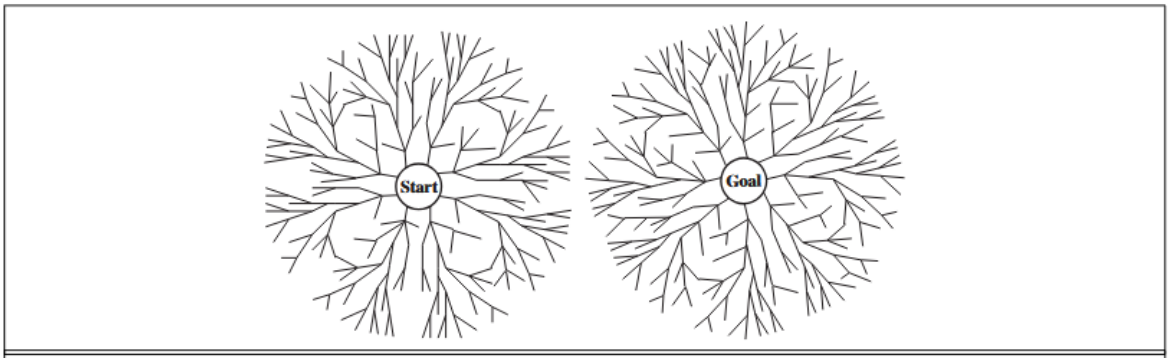
Pencarian dua arah diimplementasikan dengan mengganti uji tujuan dengan pemeriksaan untuk melihat apakah batas dari dua pencarian berpotongan; jika demikian, solusi telah ditemukan. (Penting untuk menyadari bahwa solusi pertama yang ditemukan mungkin tidak optimal, bahkan jika kedua pencarian sama-sama bersifat breadth-first; beberapa pencarian tambahan diperlukan untuk memastikan tidak ada jalan pintas lain

melintasi celah tersebut.) Pemeriksaan dapat dilakukan ketika setiap node dibuat atau dipilih untuk ekspansi dan, dengan tabel hash, akan membutuhkan waktu yang konstan.

Misalnya, jika suatu masalah memiliki kedalaman solusi $d = 6$, dan setiap arah menjalankan pencarian breadth-first satu node pada satu waktu, maka dalam kasus terburuk kedua pencarian bertemu ketika mereka telah menghasilkan semua node pada kedalaman 3. Untuk $b = 10$, ini berarti total 2.220 generasi node, dibandingkan dengan $1.111.110$ untuk pencarian breadth-first standar. Dengan demikian, kompleksitas waktu pencarian dua arah menggunakan pencarian breadth-first di kedua arah adalah $O(b^{d/2})$

Kompleksitas ruang juga $O(b^{d/2})$. Kita dapat mengurangi ini sekitar setengah jika salah satu dari dua pencarian dilakukan dengan pendalaman iteratif, tetapi setidaknya salah satu batas harus disimpan dalam memori sehingga pemeriksaan persimpangan dapat dilakukan. Persyaratan ruang ini adalah kelemahan paling signifikan dari pencarian dua arah.

Pengurangan dalam kompleksitas waktu membuat pencarian dua arah menjadi menarik, tetapi bagaimana kita mencari secara mundur? Ini tidak semudah kedengarannya. Misalkan pendahulu dari suatu keadaan x adalah semua keadaan yang memiliki x sebagai penerusnya. Pencarian dua arah memerlukan metode untuk menghitung pendahulu. Ketika semua tindakan dalam ruang keadaan dapat dibalik, pendahulu dari x hanyalah penerusnya. Kasus lain mungkin memerlukan kecerdikan yang substansial.



Gambar 3.20 Tampilan skema pencarian dua arah yang akan berhasil ketika cabang dari simpul awal bertemu dengan cabang dari simpul tujuan.

Pertimbangkan pertanyaan tentang apa yang kita maksud dengan "tujuan" dalam mencari "mundur dari tujuan." Untuk teka-teki 8 dan untuk menemukan rute di Rumania, hanya ada satu keadaan tujuan, jadi pencarian mundur sangat mirip dengan pencarian maju.

Jika ada beberapa keadaan tujuan yang tercantum secara eksplisit—misalnya, dua keadaan tujuan bebas kotoran pada Gambar 3.3—maka kita dapat membangun keadaan tujuan tiruan baru yang pendahulu langsungnya adalah semua keadaan tujuan yang sebenarnya. Tetapi jika tujuannya adalah deskripsi abstrak, seperti tujuan bahwa "tidak ada ratu yang menyerang ratu lain" dalam masalah n-ratu, maka pencarian dua arah sulit digunakan.

Membandingkan Strategi Pencarian yang tidak Terinformasi

Gambar 3.21 membandingkan strategi pencarian berdasarkan empat kriteria evaluasi yang ditetapkan dalam Bagian 3.3.2. Perbandingan ini berlaku untuk versi pencarian pohon. Untuk pencarian grafik, perbedaan utamanya adalah bahwa pencarian mendalam pertama bersifat lengkap untuk ruang keadaan terbatas dan bahwa kompleksitas ruang dan waktu dibatasi oleh ukuran ruang keadaan.

Kriteria	Luas Pertama	Biaya Seragam	Kedalaman Pertama	Batasan Kedalaman	Pendalaman Berulang	Dua Arah (Jika ada)
Lengkap?	Ya^{a}	Ya^{a,c}	Tidak	Tidak	Ya^{a}	Ya^{a,d}
Waktu	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Ruang	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Ya^{c}	Ya	Tidak	Tidak	Ya^{c}	Ya^{c,d}

Gambar 3.21 Evaluasi strategi pencarian pohon. b adalah faktor percabangan; d adalah kedalaman solusi paling dangkal; m adalah kedalaman maksimum pohon pencarian; l adalah batas kedalaman. Peringatan superskrip adalah sebagai berikut: a lengkap jika b terbatas; b lengkap jika biaya langkah $\geq \epsilon$ untuk ϵ positif; c optimal jika semua biaya langkah identik; d jika kedua arah menggunakan pencarian breadth-first.

3.5 STRATEGI PENCARIAN BERDASARKAN INFORMASI (HEURISTIK)

Bagian ini menunjukkan bagaimana strategi pencarian berdasar informasi—yang menggunakan pengetahuan khusus masalah di luar definisi masalah itu sendiri—dapat menemukan solusi lebih efisien daripada strategi tanpa informasi. Pendekatan umum yang kami pertimbangkan disebut pencarian terbaik-pertama. Pencarian terbaik-pertama adalah contoh algoritma PENCARIAN POHON atau PENCARIAN GRAFIK umum di mana sebuah simpul dipilih untuk perluasan berdasarkan fungsi evaluasi, $f(n)$. Fungsi evaluasi ditafsirkan sebagai estimasi biaya, sehingga simpul dengan evaluasi terendah diperluas terlebih dahulu. Implementasi pencarian grafik terbaik-pertama identik dengan pencarian biaya seragam (Gambar 3.14), kecuali untuk penggunaan f sebagai ganti g untuk mengurutkan antrean prioritas.

Pilihan f menentukan strategi pencarian. (Misalnya, seperti yang ditunjukkan Latihan 3.21, pencarian pohon best-first mencakup pencarian depth-first sebagai kasus khusus.) Sebagian besar algoritme best-first mencakup fungsi heuristik, yang dilambangkan $h(n)$, sebagai komponen f: $h(n)$ = estimasi biaya jalur termurah dari status di simpul n ke status tujuan. (Perhatikan bahwa $h(n)$ mengambil simpul sebagai input, tetapi, tidak seperti $g(n)$, ia hanya bergantung pada status di simpul tersebut.) Misalnya, di Rumania, seseorang dapat memperkirakan biaya jalur termurah dari Arad ke Bukares melalui jarak garis lurus dari Arad ke Bukares.

Fungsi heuristik adalah bentuk paling umum di mana pengetahuan tambahan tentang masalah diberikan kepada algoritme pencarian. Kami mempelajari heuristik secara lebih mendalam di Bagian 3.6. Untuk saat ini, kami menganggapnya sebagai fungsi yang arbitrer, nonnegatif, dan spesifik masalah, dengan satu kendala: jika n adalah simpul tujuan, maka

$h(n) = 0$. Sisa bagian ini membahas dua cara untuk menggunakan informasi heuristik guna memandu pencarian.

Pencarian Greedy Best-First

Pencarian Greedy best-first8 mencoba memperluas simpul yang paling dekat dengan tujuan, dengan alasan bahwa hal ini kemungkinan akan mengarah ke solusi dengan cepat. Jadi, ia mengevaluasi simpul hanya dengan menggunakan fungsi heuristik; yaitu, $f(n) = h(n)$. Mari kita lihat cara kerjanya untuk masalah pencarian rute di Rumania; kami menggunakan heuristik jarak garis lurus, yang akan kami sebut h_{SLD}. Jika tujuannya adalah Bukares, kami perlu mengetahui jarak garis lurus ke Bukares, yang ditunjukkan pada Gambar 3.22. Misalnya, $h_{SLD}(In(Arad)) = 366$. Perhatikan bahwa nilai h_{SLD} tidak dapat dihitung dari deskripsi masalah itu sendiri.

Selain itu, diperlukan sejumlah pengalaman untuk mengetahui bahwa h_{SLD} berkorelasi dengan jarak jalan yang sebenarnya dan, oleh karena itu, merupakan heuristik yang berguna. Gambar 3.23 menunjukkan kemajuan pencarian greedy best-first menggunakan h_{SLD} untuk menemukan jalur dari Arad ke Bucharest. Node pertama yang akan diperluas dari Arad adalah Sibiu karena lebih dekat ke Bucharest daripada Zerind atau Timisoara. Node berikutnya yang akan diperluas adalah Fagaras karena paling dekat. Fagaras pada gilirannya menghasilkan Bucharest, yang merupakan tujuannya.

Untuk masalah khusus ini, pencarian greedy best-first menggunakan h_{SLD} menemukan solusi tanpa pernah memperluas node yang tidak berada di jalur solusi; oleh karena itu, biaya pencariannya minimal. Namun, hal itu tidak optimal: jalur melalui Sibiu dan Fagaras ke Bukares 32 kilometer lebih panjang daripada jalur melalui Rimnicu Vilcea dan Pitesti. Ini menunjukkan mengapa algoritme itu disebut "serakah"—pada setiap langkah, algoritme itu mencoba sedekat mungkin dengan tujuan. Pencarian pohon terbaik-pertama yang serakah juga tidak lengkap bahkan dalam ruang keadaan terbatas, seperti pencarian kedalaman-pertama. Pertimbangkan masalah perjalanan dari Iasi ke Fagaras.

Heuristik menyarankan agar Neamt diperluas terlebih dahulu karena paling dekat dengan Fagaras, tetapi itu jalan buntu. Solusinya adalah pergi dulu ke Vaslui—langkah yang sebenarnya lebih jauh dari tujuan menurut heuristik—lalu melanjutkan ke Urziceni, Bukares, dan Fagaras. Algoritme tidak akan pernah menemukan solusi ini, karena perluasan Neamt mengembalikan Iasi ke perbatasan, Iasi lebih dekat ke Fagaras daripada Vaslui, dan karenanya Iasi akan diperluas lagi, yang mengarah ke loop tak terbatas. (Versi pencarian grafik lengkap dalam ruang terbatas, tetapi tidak dalam ruang tak terbatas.)

Kompleksitas waktu dan ruang terburuk untuk versi pohon adalah $O(b^m)$, di mana m adalah kedalaman maksimum ruang pencarian. Namun, dengan fungsi heuristik yang baik, kompleksitas dapat dikurangi secara substansial. Jumlah pengurangan bergantung pada masalah tertentu dan kualitas heuristik.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380

Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Gambar 3.22 Nilai h_{SLD} —jarak garis lurus ke Bukares.

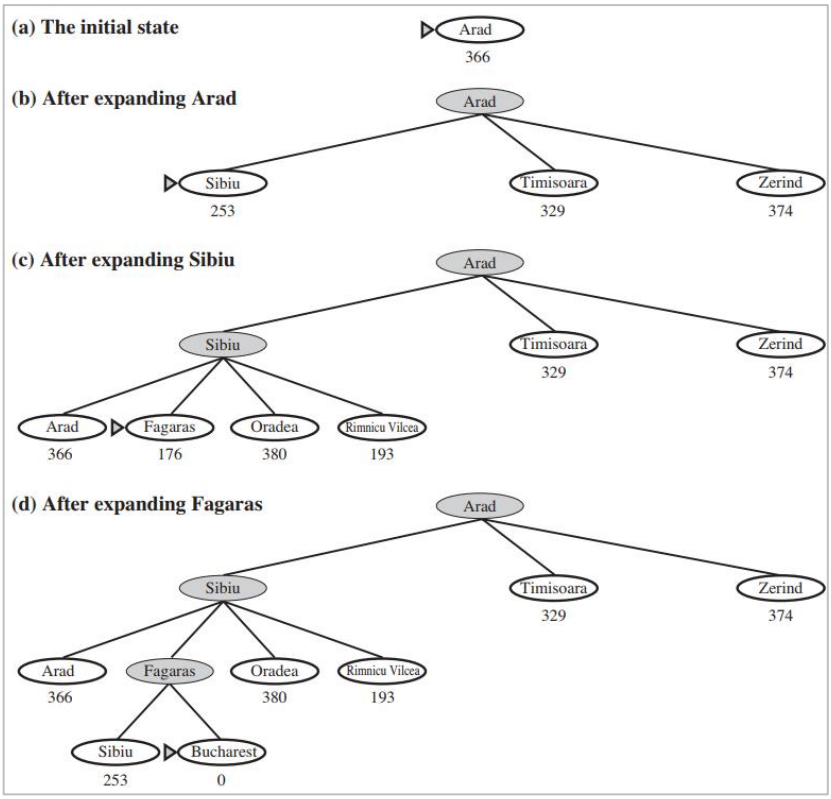
Pencarian A*: Meminimalkan Total Estimasi Biaya Solusi

Bentuk pencarian best-first yang paling dikenal luas disebut pencarian $\mathbf{A}^*$ (diucapkan "pencarian A-star"). Pencarian ini mengevaluasi simpul dengan menggabungkan $g(n)$, biaya untuk mencapai simpul, dan $h(n)$, biaya untuk berpindah dari simpul ke tujuan:

$$f(n) = g(n) + h(n) .$$

Karena $g(n)$ memberikan biaya lintasan dari simpul awal ke simpul n, dan $h(n)$ adalah estimasi biaya lintasan termurah dari n ke tujuan, kita memperoleh

$f(n)$ = estimasi biaya solusi termurah melalui n .



Gambar 3.23 Tahapan dalam pencarian pohon terbaik-pertama yang rakus untuk Bucharest dengan heuristik jarak garis lurus h_{SLD} Node diberi label dengan nilai h-nya.

Jadi, jika kita mencoba mencari solusi termurah, hal yang masuk akal untuk dicoba terlebih dahulu adalah simpul dengan nilai $g(n) + h(n)$ terendah. Ternyata strategi ini lebih dari sekadar masuk akal: asalkan fungsi heuristik $h(n)$ memenuhi kondisi tertentu, pencarian A^* bersifat lengkap dan optimal. Algoritmanya identik dengan UNIFORM-COST-SEARCH kecuali A^* menggunakan $g + h$, bukan g.

Kondisi untuk Optimalitas: Admisibilitas dan Konsistensi

Kondisi pertama yang kita perlukan untuk optimalitas adalah $h(n)$ menjadi heuristik yang dapat diterima. Heuristik yang dapat diterima adalah heuristik yang tidak pernah melebih-lebihkan biaya untuk mencapai tujuan. Karena $g(n)$ adalah biaya aktual untuk mencapai n di sepanjang jalur saat ini, dan $f(n) = g(n) + h(n)$, kita memiliki konsekuensi langsung bahwa $f(n)$ tidak pernah melebih-lebihkan biaya sebenarnya dari solusi di sepanjang jalur saat ini melalui n.

Heuristik yang dapat diterima pada dasarnya optimis karena menganggap biaya penyelesaian masalah lebih rendah dari yang sebenarnya. Contoh nyata dari heuristik yang dapat diterima adalah jarak garis lurus h_{SLD} yang kita gunakan untuk menuju Bukares. Jarak garis lurus dapat diterima karena jalur terpendek antara dua titik mana pun adalah garis lurus, sehingga garis lurus tidak dapat dilebih-lebihkan.

Pada Gambar 3.24, kami menunjukkan kemajuan pencarian pohon A^* untuk Bucharest. Nilai g dihitung dari biaya langkah pada Gambar 3.2, dan nilai h_{SLD} diberikan pada Gambar 3.22. Perhatikan khususnya bahwa Bucharest pertama kali muncul di garis batas pada langkah (e), tetapi tidak dipilih untuk perluasan karena biaya f-nya (450) lebih tinggi daripada biaya Pitesti (417). Cara lain untuk mengatakan ini adalah bahwa mungkin ada solusi melalui Pitesti yang biayanya serendah 417, sehingga algoritme tidak akan menerima solusi yang biayanya 450.

Kondisi kedua yang sedikit lebih kuat disebut konsistensi (atau terkadang monotonisitas) hanya diperlukan untuk aplikasi A^* pada pencarian grafik. Heuristik $h(n)$ konsisten jika, untuk setiap simpul n dan setiap penerus n' dari n yang dihasilkan oleh tindakan apa pun a, perkiraan biaya untuk mencapai tujuan dari n tidak lebih besar dari biaya langkah untuk mencapai n' ditambah perkiraan biaya untuk mencapai tujuan dari n':

$$h(n) \leq c(n, a, n') + h(n') .$$

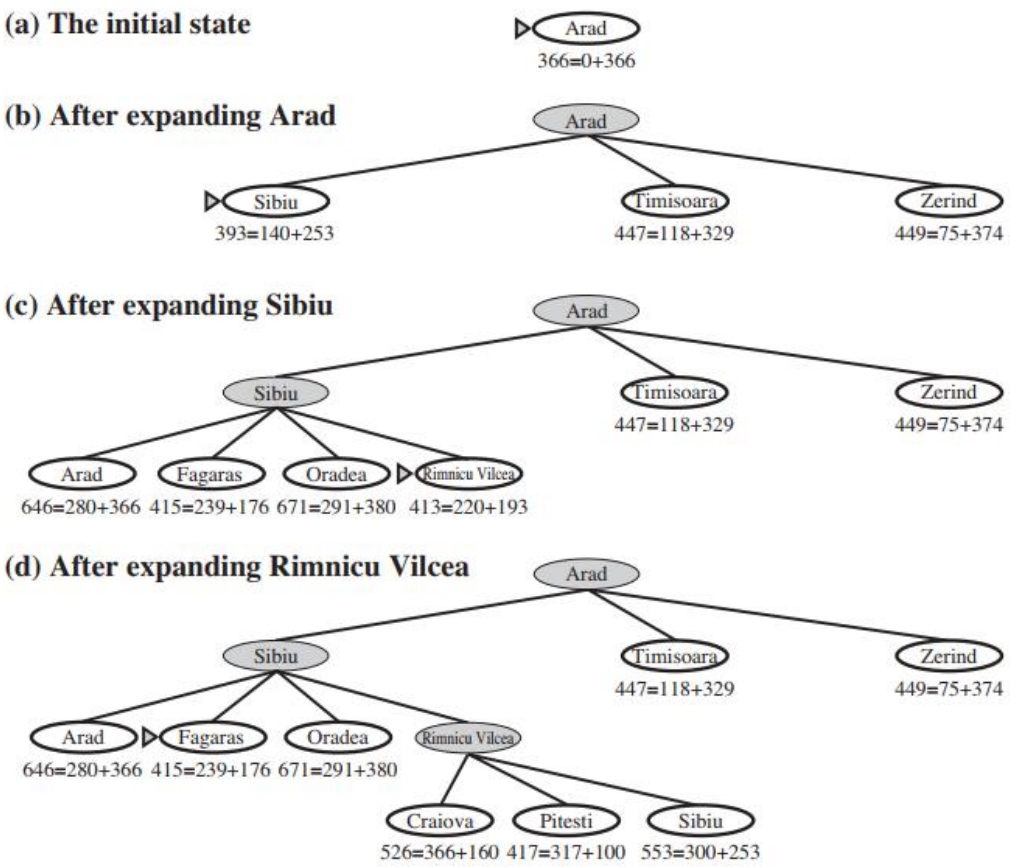
Ini adalah bentuk pertidaksamaan segitiga umum, yang menetapkan bahwa setiap sisi segitiga tidak boleh lebih panjang dari jumlah kedua sisi lainnya. Di sini, segitiga dibentuk oleh n, n', dan tujuan G_n yang paling dekat dengan n. Untuk heuristik yang dapat diterima, pertidaksamaan tersebut sangat masuk akal: jika ada rute dari n ke G_n melalui n' yang lebih murah daripada $h(n)$, itu akan melanggar sifat bahwa $h(n)$ adalah batas bawah biaya untuk mencapai G_n.

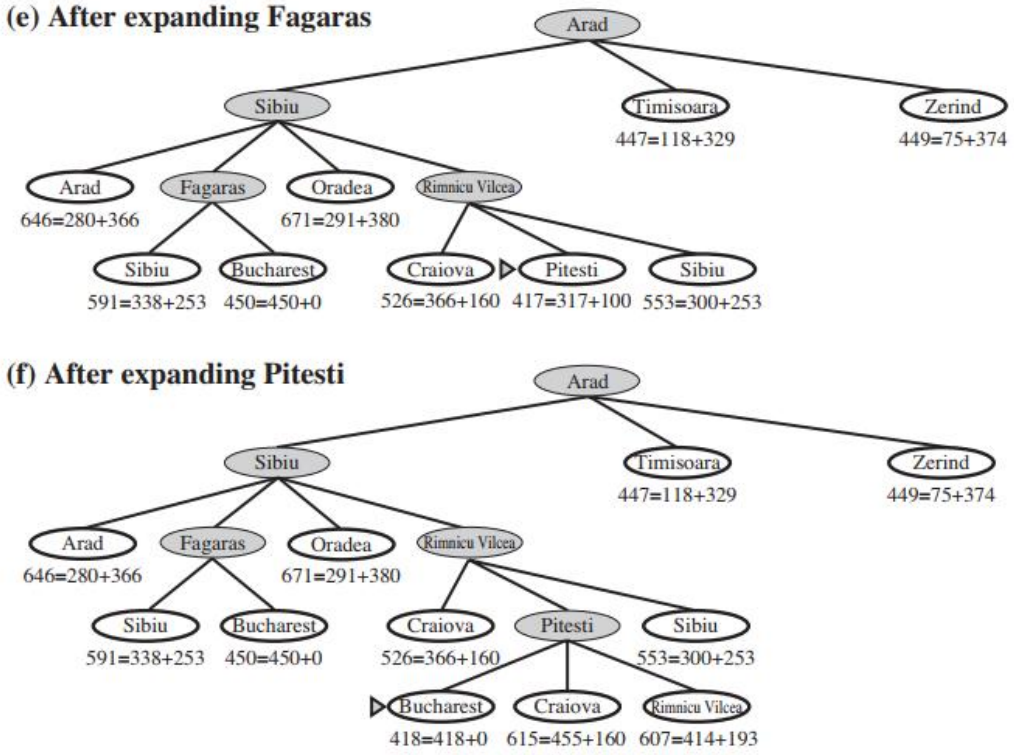
Cukup mudah untuk menunjukkan (Latihan 3.29) bahwa setiap heuristik yang konsisten juga dapat diterima. Oleh karena itu, konsistensi merupakan persyaratan yang lebih ketat daripada penerimaan, tetapi seseorang harus bekerja cukup keras untuk meramu heuristik

yang dapat diterima tetapi tidak konsisten. Semua heuristik yang dapat diterima yang kita bahas dalam bab ini juga konsisten. Pertimbangkan, misalnya, hSLD. Kita tahu bahwa ketidaksetaraan segitiga umum terpenuhi ketika setiap sisi diukur dengan jarak garis lurus dan bahwa jarak garis lurus antara n dan n' tidak lebih besar dari $c(n, a, n')$. Oleh karena itu, hSLD adalah heuristik yang konsisten.

Optimalitas A*

Seperti yang telah disebutkan sebelumnya, A* memiliki properti berikut: versi pencarian pohon dari A* optimal jika $h(n)$ dapat diterima, sedangkan versi pencarian grafik optimal jika $h(n)$ konsisten. Kami menunjukkan klaim kedua dari kedua klaim ini karena lebih berguna. Argumen tersebut pada dasarnya mencerminkan argumen untuk optimalitas pencarian biaya seragam, dengan g digantikan oleh
ƒ —seperti dalam algoritma A* itu sendiri.



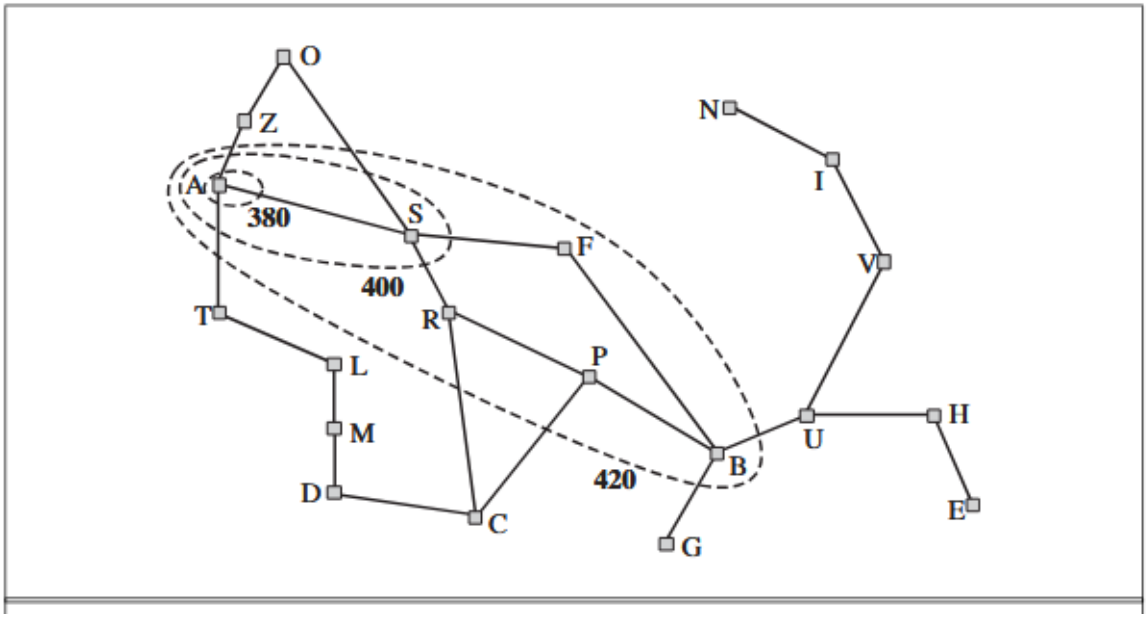


Gambar 3.24 Tahapan dalam pencarian A* untuk Bucharest. Node diberi label dengan $f = g + h$. Nilai h adalah jarak garis lurus ke Bucharest yang diambil dari Gambar 3.22.

Langkah pertama adalah menetapkan yang berikut: jika $h(n)$ konsisten, maka nilai $f(n)$ di sepanjang jalur mana pun tidak menurun. Bukti mengikuti langsung dari definisi konsistensi. Misalkan n' adalah penerus n; maka $g(n') = g(n) + c(n, a, n')$ untuk beberapa aksi a, dan kita memiliki:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n).$$

Langkah berikutnya adalah membuktikan bahwa setiap kali A* memilih simpul n untuk ekspansi, jalur optimal ke simpul tersebut telah ditemukan. Jika tidak demikian, harus ada simpul perbatasan lain n' pada jalur optimal dari simpul awal ke n, berdasarkan sifat pemisahan grafik pada Gambar 3.9; karena f tidak menurun di sepanjang jalur mana pun, n' akan memiliki biaya f yang lebih rendah daripada n dan akan dipilih terlebih dahulu.



Gambar 3.25 Peta Rumania yang menunjukkan kontur pada $f = 380, f = 400$, dan $f = 420$, dengan Arad sebagai negara bagian awal. Node di dalam kontur tertentu memiliki biaya f yang lebih kecil atau sama dengan nilai kontur.

Dari dua pengamatan sebelumnya, dapat disimpulkan bahwa urutan simpul yang diekspansi oleh A* menggunakan PENCARIAN-GRAFIK berada dalam urutan yang tidak menurun dari $f(n)$. Oleh karena itu, simpul tujuan pertama yang dipilih untuk perluasan harus merupakan solusi optimal karena f adalah biaya sebenarnya untuk simpul tujuan (yang memiliki $h = 0$) dan semua simpul tujuan selanjutnya akan setidaknya sama mahalnya.

Fakta bahwa biaya-f tidak menurun di sepanjang jalur apa pun juga berarti bahwa kita dapat menggambar kontur di ruang status, seperti kontur di peta topografi. Gambar 3.25 menunjukkan sebuah contoh. Di dalam kontur berlabel 400, semua simpul memiliki $f(n)$ kurang dari atau sama dengan 400, dan seterusnya. Kemudian, karena A* memperluas simpul perbatasan dengan biaya-f terendah, kita dapat melihat bahwa pencarian A* menyebar dari simpul awal, menambahkan simpul dalam pita konsentris dengan biaya-f yang meningkat. Dengan pencarian biaya seragam (pencarian A* menggunakan $h(n) = 0$), pita akan menjadi "lingkaran" di sekitar status awal. Dengan heuristik yang lebih akurat, pita akan meregang ke arah status tujuan dan menjadi lebih terfokus secara sempit di sekitar jalur optimal. Jika C^* adalah biaya jalur solusi optimal, maka kita dapat mengatakan yang berikut:

- A* mengembangkan semua simpul dengan $f(n) < C^*$.
- A* kemudian dapat mengembangkan beberapa simpul tepat pada "kontur tujuan" (di mana $f(n) = C^*$) sebelum memilih simpul tujuan.

Kelengkapan mengharuskan hanya ada simpul yang jumlahnya terbatas dengan biaya kurang dari atau sama dengan C^*, suatu kondisi yang benar jika semua biaya langkah melebihi beberapa ϵ terbatas dan jika b terbatas.

Perhatikan bahwa A* tidak mengembangkan simpul dengan $f(n) > C^*$—misalnya, Timisoara tidak dikembangkan pada Gambar 3.24 meskipun merupakan anak dari akar. Kita

katakan bahwa sub-pohon di bawah Timisoara dipangkas; karena h_{SLD} dapat diterima, algoritme dapat mengabaikan sub-pohon ini dengan aman sambil tetap menjamin keoptimalan. Konsep pemangkasan—menghilangkan kemungkinan dari pertimbangan tanpa harus memeriksanya—penting bagi banyak bidang AI.

Satu pengamatan terakhir adalah bahwa di antara algoritme optimal jenis ini—algoritme yang memperluas jalur pencarian dari akar dan menggunakan informasi heuristik yang sama—A* efisien secara optimal untuk setiap heuristik konsisten yang diberikan. Artinya, tidak ada algoritme optimal lain yang dijamin akan memperluas lebih sedikit simpul daripada A* (kecuali mungkin melalui pemecahan seri di antara simpul dengan $f(n) = C*$). Ini karena algoritme apa pun yang tidak memperluas semua simpul dengan $f(n) < C*$ berisiko kehilangan solusi optimal.

Pencarian A* yang lengkap, optimal, dan efisien secara optimal di antara semua algoritme tersebut cukup memuaskan. Sayangnya, itu tidak berarti bahwa A* adalah jawaban untuk semua kebutuhan pencarian kita. Masalahnya adalah, untuk sebagian besar masalah, jumlah status dalam ruang pencarian kontur tujuan masih eksponensial dalam panjang solusi. Rincian analisis berada di luar cakupan buku ini, tetapi hasil dasarnya adalah sebagai berikut. Untuk masalah dengan biaya langkah konstan, pertumbuhan waktu proses sebagai fungsi kedalaman solusi optimal d dianalisis dalam hal kesalahan absolut atau kesalahan relatif heuristik. Kesalahan absolut didefinisikan sebagai $\Delta \equiv h^* - h$, di mana h* adalah biaya aktual untuk mencapai tujuan dari akar, dan kesalahan relatif didefinisikan sebagai $\epsilon \equiv (h^* - h)/h^*$.

Hasil kompleksitas sangat bergantung pada asumsi yang dibuat tentang ruang keadaan. Model paling sederhana yang dipelajari adalah ruang keadaan yang memiliki satu tujuan dan pada dasarnya adalah pohon dengan tindakan yang dapat dibalik. (Teka-teki ke-8 memenuhi asumsi pertama dan ketiga ini.) Dalam kasus ini, kompleksitas waktu A* bersifat eksponensial dalam kesalahan absolut maksimum, yaitu, $O(b^{\Delta})$ Untuk biaya langkah konstan, kita dapat menulisnya sebagai $O(b^{\epsilon d})$, di mana d adalah kedalaman solusi. Untuk hampir semua heuristik dalam penggunaan praktis, kesalahan absolut setidaknya proporsional dengan biaya jalur h^*, jadi ϵ konstan atau bertambah dan kompleksitas waktu bersifat eksponensial dalam d. Kita juga dapat melihat efek dari heuristik yang lebih akurat: $O(b^{\epsilon d}) = O((b^{\epsilon})^d)$ jadi faktor percabangan efektif (didefinisikan lebih formal di bagian berikutnya) adalah b^{ϵ}.

Ketika ruang keadaan memiliki banyak keadaan tujuan—terutama keadaan tujuan yang mendekati optimal—proses pencarian dapat menyimpang dari jalur optimal dan ada biaya tambahan yang proporsional dengan jumlah tujuan yang biayanya berada dalam faktor ϵ dari biaya optimal. Akhirnya, dalam kasus umum grafik, situasinya bahkan lebih buruk. Dapat ada banyak keadaan secara eksponensial dengan $f(n) < C^*$ bahkan jika kesalahan absolut dibatasi oleh suatu konstanta.

Misalnya, pertimbangkan versi dunia vakum tempat agen dapat membersihkan kotak mana pun dengan biaya satuan tanpa harus mengunjunginya: dalam kasus tersebut, kotak dapat dibersihkan dalam urutan apa pun. Dengan N kotak yang awalnya kotor, ada 2^N keadaan tempat beberapa himpunan bagian telah dibersihkan dan semuanya berada di jalur solusi

optimal—dan karenanya memenuhi $f(n) < C^*$ —bahkan jika heuristik memiliki kesalahan 1. Kompleksitas A* sering kali membuatnya tidak praktis untuk bersikeras menemukan solusi optimal.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE),∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    successors ←[ ]
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure, ∞
    for each s in successors do /* update f with value from previous search, if any */
        s.f ← max(s.g + s.h, node.f )
    loop do
        best ← the lowest f-value node in successors
        if best.f > f_limit then return failure, best.f
        alternative ←the second-lowest f −value among successors
        result, best.f ← RBFS(problem, best, min( f_limit, alternative))
        if result ≠ failure then return result
```

Gambar 3.26 Algoritma untuk pencarian terbaik-pertama secara rekursif.

Seseorang dapat menggunakan varian A* yang menemukan solusi suboptimal dengan cepat, atau seseorang terkadang dapat merancang heuristik yang lebih akurat tetapi tidak sepenuhnya dapat diterima. Bagaimanapun, penggunaan heuristik yang baik tetap memberikan penghematan yang sangat besar dibandingkan dengan penggunaan pencarian yang tidak berdasar. Di Bagian 3.6, kita akan membahas pertanyaan tentang merancang heuristik yang baik. Namun, waktu komputasi bukanlah kelemahan utama A*. Karena ia menyimpan semua simpul yang dihasilkan dalam memori (seperti halnya semua algoritma PENCARIAN-GRAFIK), A* biasanya kehabisan ruang jauh sebelum kehabisan waktu. Karena alasan ini, A* tidak praktis untuk banyak masalah berskala besar. Namun, ada algoritma yang mengatasi masalah ruang tanpa mengorbankan keoptimalan atau kelengkapan, dengan sedikit biaya dalam waktu eksekusi. Kita akan membahasnya selanjutnya.

Pencarian Heuristik yang Dibatasi Memori

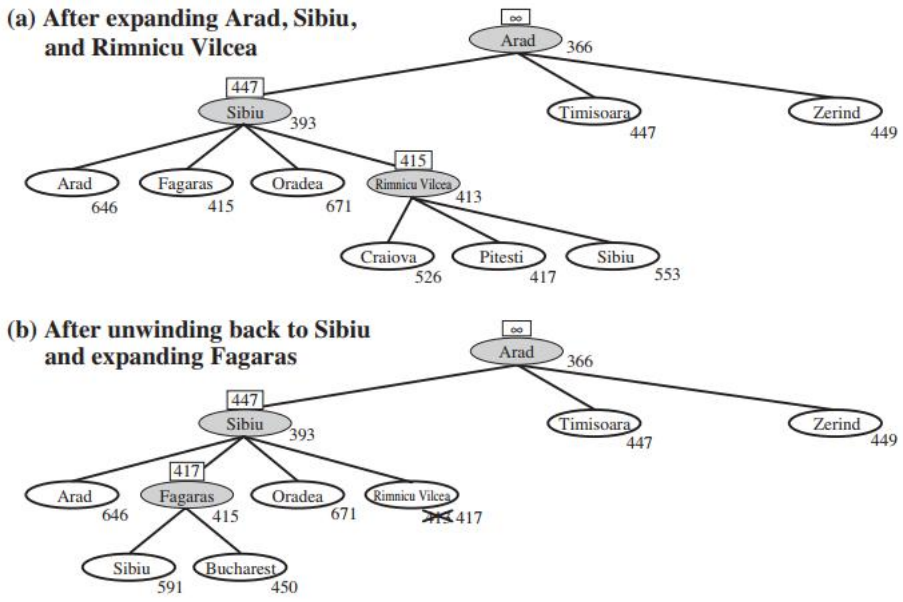
Cara paling sederhana untuk mengurangi kebutuhan memori untuk A* adalah mengadaptasi ide pendalaman iteratif ke konteks pencarian heuristik, yang menghasilkan algoritma A* (IDA*) pendalaman iteratif. Perbedaan utama antara IDA* dan pendalaman iteratif standar adalah bahwa batas yang digunakan adalah biaya $f(g + h)$ dan bukan kedalaman; pada setiap iterasi, nilai batas adalah biaya f terkecil dari setiap simpul yang melampaui batas pada iterasi sebelumnya. IDA* praktis untuk banyak masalah dengan biaya langkah unit dan menghindari overhead substansial yang terkait dengan menjaga antrean simpul yang diurutkan.

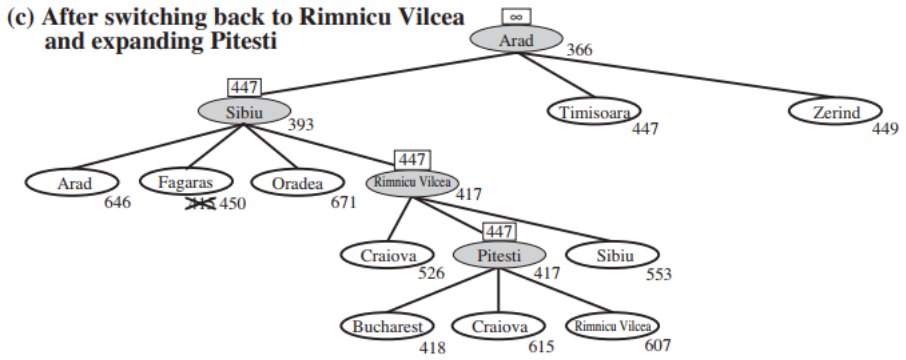
Sayangnya, ia mengalami kesulitan yang sama dengan biaya bernilai riil seperti halnya versi iteratif pencarian biaya seragam. Bagian ini secara singkat memeriksa dua algoritma terikat memori lainnya, yang disebut RBFS dan MA*. Pencarian terbaik-pertama rekursif (RBFS) adalah algoritma rekursif sederhana yang mencoba meniru operasi pencarian terbaik-pertama standar, tetapi hanya menggunakan ruang linier. Algoritma tersebut ditunjukkan pada Gambar 3.26.

Strukturnya mirip dengan pencarian kedalaman-pertama rekursif, tetapi alih-alih melanjutkan tanpa batas ke jalur saat ini, ia menggunakan variabel batas f untuk melacak nilai-f dari jalur alternatif terbaik yang tersedia dari setiap leluhur simpul saat ini. Jika simpul saat ini melampaui batas ini, rekursi akan kembali ke jalur alternatif. Saat rekursi berakhir, RBFS mengganti nilai-f dari setiap simpul di sepanjang jalur dengan nilai cadangan—nilai-f terbaik dari anak-anaknya. Dengan cara ini, RBFS mengingat nilai-f dari daun terbaik di sub-pohon yang terlupakan dan karena itu dapat memutuskan apakah perlu memperluas kembali sub-pohon di lain waktu.

Gambar 3.27 menunjukkan bagaimana RBFS mencapai Bucharest. RBFS agak lebih efisien daripada IDA*, tetapi masih mengalami regenerasi simpul yang berlebihan. Dalam contoh pada Gambar 3.27, RBFS mengikuti jalur melalui Rimnicu Vilcea, lalu "berubah pikiran" dan mencoba Fagaras, lalu berubah pikiran lagi. Perubahan pikiran ini terjadi karena setiap kali jalur terbaik saat ini diperluas, nilai f-nya cenderung meningkat—h biasanya kurang optimis untuk simpul yang lebih dekat ke tujuan.

Ketika ini terjadi, jalur terbaik kedua mungkin menjadi jalur terbaik, jadi pencarian harus mundur untuk mengikutinya. Setiap perubahan pikiran sesuai dengan iterasi IDA* dan dapat memerlukan banyak perluasan ulang simpul yang terlupakan untuk membuat ulang jalur terbaik dan memperluasnya ke satu simpul lagi.





Gambar 3.27 Tahapan dalam pencarian RBFS untuk rute terpendek ke Bukares. Nilai batas f untuk setiap panggilan rekursif ditampilkan di atas setiap simpul saat ini, dan setiap simpul diberi label dengan biaya f-nya. (a) Jalur melalui Rimnicu Vilcea diikuti hingga daun terbaik saat ini (Pitesti) memiliki nilai yang lebih buruk daripada jalur alternatif terbaik (Fagaras). (b) Rekursi terlepas dan nilai daun terbaik dari sub-pohon yang terlupakan (417) dicadangkan ke Rimnicu Vilcea; kemudian Fagaras diperluas, memperlihatkan nilai daun terbaik sebesar 450. (c) Rekursi terlepas dan nilai daun terbaik dari sub-pohon yang terlupakan (450) dicadangkan ke Fagaras; kemudian Rimnicu Vilcea diperluas. Kali ini, karena jalur alternatif terbaik (melalui Timisoara) berbiaya sedikitnya 447, perluasan berlanjut ke Bukares.

Seperti pencarian pohon A^*, RBFS adalah algoritma optimal jika fungsi heuristik $h(n)$ dapat diterima. Kompleksitas ruangnya linear pada kedalaman solusi optimal terdalam, tetapi kompleksitas waktunya agak sulit untuk dikarakterisasi: hal itu bergantung pada akurasi fungsi heuristik dan pada seberapa sering jalur terbaik berubah saat simpul diperluas. IDA^* dan RBFS mengalami masalah karena menggunakan terlalu sedikit memori. Di antara iterasi, IDA^* hanya menyimpan satu angka: batas biaya *f* saat ini. RBFS menyimpan lebih banyak informasi dalam memori, tetapi hanya menggunakan ruang linear: bahkan jika lebih banyak memori tersedia, RBFS tidak memiliki cara untuk memanfaatkannya. Karena mereka melupakan sebagian besar dari apa yang telah mereka lakukan, kedua algoritme tersebut mungkin berakhir dengan memperluas kembali status yang sama berkali-kali. Lebih jauh, mereka mengalami peningkatan kompleksitas yang berpotensi eksponensial yang terkait dengan jalur redundan dalam grafik (lihat Bagian 3.3).

Oleh karena itu, tampaknya masuk akal untuk menggunakan semua memori yang tersedia. Dua algoritme yang melakukan ini adalah **MA*** (A^* yang dibatasi memori) dan **SMA*** (MA^* yang disederhanakan). **SMA***—ya—lebih sederhana, jadi kami akan menjelaskannya. SMA^* berjalan seperti A^*, memperluas daun terbaik hingga memori penuh. Pada titik ini, SMA^* tidak dapat menambahkan simpul baru ke pohon pencarian tanpa membuang simpul lama. SMA^*selalu membuang simpul daun terburuk—simpul dengan nilai *f* tertinggi. Seperti RBFS, SMA^*kemudian mencadangkan nilai simpul yang terlupakan ke induknya.

Dengan cara ini, leluhur dari subpohon yang terlupakan mengetahui kualitas jalur terbaik di subpohon tersebut. Dengan informasi ini, SMA^* membuat ulang subpohon hanya ketika semua jalur lain telah terbukti terlihat lebih buruk daripada jalur yang telah dilupakannya. Cara lain untuk mengatakan ini adalah, jika semua keturunan dari simpul n

dilupakan, maka kita tidak akan tahu ke mana harus pergi dari n, tetapi kita masih akan memiliki gambaran tentang seberapa berharganya untuk pergi ke mana saja dari n.

Algoritme lengkapnya terlalu rumit untuk direproduksi di sini, tetapi ada satu hal yang perlu disebutkan. Kami mengatakan bahwa SMA^* memperluas daun terbaik dan menghapus daun terburuk. Bagaimana jika semua simpul daun memiliki nilai f yang sama? Untuk menghindari pemilihan simpul yang sama untuk penghapusan dan perluasan, SMA^* memperluas daun terbaik terbaru dan menghapus daun terburuk tertua. Ini bertepatan ketika hanya ada satu daun, tetapi dalam kasus itu, pohon pencarian saat ini harus menjadi jalur tunggal dari akar ke daun yang mengisi semua memori. Jika daun bukan simpul tujuan, maka bahkan jika berada di jalur solusi optimal, solusi itu tidak dapat dicapai dengan memori yang tersedia.

Oleh karena itu, simpul dapat dibuang persis seperti jika tidak memiliki penerus. SMA^* lengkap jika ada solusi yang dapat dicapai—yaitu, jika d, kedalaman simpul tujuan yang paling dangkal, lebih kecil dari ukuran memori (dinyatakan dalam simpul). SMA^*optimal jika ada solusi optimal yang dapat dicapai; jika tidak, SMA^* mengembalikan solusi terbaik yang dapat dicapai. Secara praktis, SMA^* adalah pilihan yang cukup tangguh untuk menemukan solusi optimal, terutama ketika ruang keadaan adalah grafik, biaya langkah tidak seragam, dan pembuatan simpul mahal dibandingkan dengan biaya overhead untuk mempertahankan batas dan set yang dieksplorasi. Namun, pada masalah yang sangat sulit, sering kali SMA^* dipaksa untuk beralih maju mundur secara terus-menerus di antara banyak jalur solusi kandidat, yang hanya sebagian kecilnya yang dapat masuk ke dalam memori. (Ini menyerupai masalah thrashing dalam sistem paging disk.) Kemudian waktu tambahan yang diperlukan untuk regenerasi berulang dari simpul yang sama berarti bahwa masalah yang secara praktis dapat dipecahkan oleh A^*, mengingat memori yang tidak terbatas, menjadi tidak dapat dipecahkan untuk SMA^*. Artinya, keterbatasan memori dapat membuat masalah menjadi tidak dapat dipecahkan dari sudut pandang waktu komputasi. Meskipun tidak ada teori saat ini yang menjelaskan tradeoff antara waktu dan memori, tampaknya ini adalah masalah yang tidak dapat dihindari. Satu-satunya jalan keluar adalah dengan mengabaikan persyaratan optimalitas.

Belajar Mencari dengan Lebih Baik

Kami telah menyajikan beberapa strategi tetap—breadth-first, greedy best-first, dan seterusnya—yang telah dirancang oleh ilmuwan komputer. Bisakah agen belajar cara mencari dengan lebih baik? Jawabannya adalah ya, dan metode tersebut bertumpu pada konsep penting yang disebut ruang status metalevel. Setiap status dalam ruang status metalevel menangkap status internal (komputasional) dari sebuah program yang mencari dalam ruang status level objek seperti Romania.

Misalnya, status internal algoritma A^* terdiri dari pohon pencarian saat ini. Setiap tindakan dalam ruang status metalevel adalah langkah komputasi yang mengubah status internal; misalnya, setiap langkah komputasi dalam A^* memperluas simpul daun dan menambahkan penerusnya ke pohon. Dengan demikian, Gambar 3.24, yang menunjukkan urutan pohon pencarian yang semakin besar, dapat dilihat sebagai penggambaran jalur dalam

ruang status metalevel di mana setiap status pada jalur tersebut adalah pohon pencarian level objek.

Sekarang, jalur dalam Gambar 3.24 memiliki lima langkah, termasuk satu langkah, perluasan Fagaras, yang tidak terlalu membantu. Untuk masalah yang lebih sulit, akan ada banyak kesalahan langkah seperti itu, dan algoritma pembelajaran metalevel dapat belajar dari pengalaman ini untuk menghindari penjelajahan sub-pohon yang tidak menjanjikan. Teknik yang digunakan untuk pembelajaran semacam ini dijelaskan dalam Bab 21. Tujuan pembelajaran adalah meminimalkan total biaya pemecahan masalah, dengan mengorbankan biaya komputasi dan biaya jalur.

3.6 FUNGSI HEURISTIK

Pada bagian ini, kita akan membahas heuristik untuk teka-teki 8, untuk menjelaskan hakikat heuristik secara umum. Teka-teki 8 adalah salah satu masalah pencarian heuristik paling awal. Seperti yang disebutkan dalam Bagian 3.2, tujuan teka-teki adalah menggeser petak secara horizontal atau vertikal ke dalam ruang kosong hingga konfigurasinya sesuai dengan konfigurasi tujuan (Gambar 3.28).

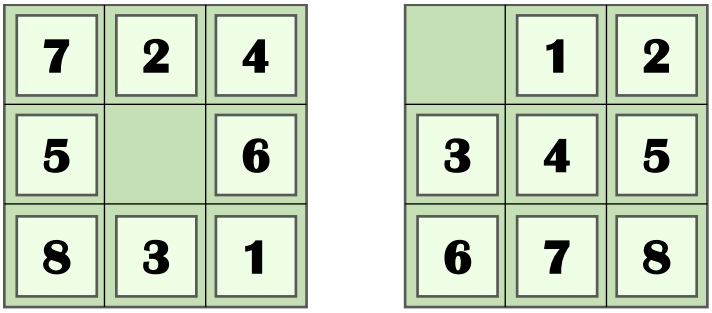
Biaya solusi rata-rata untuk contoh teka-teki 8 yang dibuat secara acak adalah sekitar 22 langkah. Faktor percabangannya sekitar 3. (Ketika ubin kosong berada di tengah, empat gerakan mungkin dilakukan; ketika berada di sudut, dua gerakan; dan ketika berada di sepanjang tepi, tiga gerakan.) Ini berarti bahwa pencarian pohon menyeluruh hingga kedalaman 22 akan melihat sekitar $3^{22} \approx 3{,}1 \times 10^{10}$ status. Pencarian grafik akan memotongnya dengan faktor sekitar 170.000 karena hanya $9!/2 = 181.440$ status berbeda yang dapat dicapai. (Lihat Latihan 3.4.)

Ini adalah angka yang dapat dikelola, tetapi angka yang sesuai untuk teka-teki ke-15 kira-kira 10^{13}, jadi urutan berikutnya adalah menemukan fungsi heuristik yang baik. Jika kita ingin menemukan solusi terpendek dengan menggunakan A^*, kita memerlukan fungsi heuristik yang tidak pernah melebih-lebihkan jumlah langkah menuju tujuan. Ada sejarah panjang heuristik semacam itu untuk teka-teki ke-15; berikut dua kandidat yang umum digunakan:

- h_1= jumlah ubin yang salah tempat. Untuk Gambar 3.28, kedelapan ubin semuanya berada di luar posisi, jadi kondisi awal akan memiliki $h_1 = 8$. h_1 adalah heuristik yang dapat diterima karena jelas bahwa ubin apa pun yang tidak pada tempatnya harus dipindahkan setidaknya sekali.
- h_2 = jumlah jarak ubin dari posisi tujuannya. Karena ubin tidak dapat bergerak sepanjang diagonal, jarak yang akan kita hitung adalah jumlah jarak horizontal dan vertikal. Ini terkadang disebut jarak blok kota atau jarak Manhattan. h_2 juga dapat diterima karena semua gerakan hanya dapat memindahkan satu ubin satu langkah lebih dekat ke tujuan. Ubin 1 hingga 8 di kondisi awal memberikan jarak Manhattan sebesar

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

Seperti yang diharapkan, keduanya tidak melebih-lebihkan biaya solusi sebenarnya, yaitu 26.



Negarai Mulai Negara Tujuan

Gambar 3.28 Contoh umum teka-teki 8. Solusinya terdiri dari 26 langkah.

Pengaruh Akurasi Heuristik pada Kinerja

Salah satu cara untuk mengkarakterisasi kualitas heuristik adalah faktor percabangan efektif b^*. Jika jumlah total simpul yang dihasilkan oleh A* untuk masalah tertentu adalah N dan kedalaman solusi adalah d, maka b^* adalah faktor percabangan yang harus dimiliki pohon seragam dengan kedalaman d agar dapat memuat $N + 1$ simpul. Jadi,

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d .$$

Misalnya, jika A* menemukan solusi pada kedalaman 5 menggunakan 52 simpul, maka faktor percabangan efektif adalah 1,92. Faktor percabangan efektif dapat bervariasi di antara contoh masalah, tetapi biasanya cukup konstan untuk masalah yang cukup sulit. (Keberadaan faktor percabangan efektif mengikuti hasil yang disebutkan sebelumnya, yaitu jumlah simpul yang diperluas oleh A* tumbuh secara eksponensial dengan kedalaman solusi.) Oleh karena itu, pengukuran eksperimental b^* pada sekumpulan kecil masalah dapat memberikan panduan yang baik untuk kegunaan heuristik secara keseluruhan. Heuristik yang dirancang dengan baik akan memiliki nilai b^* mendekati 1, yang memungkinkan masalah yang cukup besar untuk dipecahkan dengan biaya komputasi yang wajar.

Untuk menguji fungsi heuristik h_1 dan h_2, kami menghasilkan 1200 masalah acak dengan panjang solusi dari 2 hingga 24 (100 untuk setiap angka genap) dan memecahkannya dengan pencarian pendalaman berulang dan dengan pencarian pohon A* menggunakan h_1 dan h_2. Gambar 3.29 memberikan jumlah rata-rata simpul yang dihasilkan oleh setiap strategi dan faktor percabangan efektif. Hasilnya menunjukkan bahwa h_2 lebih baik daripada h_1, dan jauh lebih baik daripada menggunakan pencarian pendalaman berulang. Bahkan untuk masalah kecil dengan $d = 12$, A* dengan h_2 adalah 50.000 kali lebih efisien daripada pencarian pendalaman iteratif tanpa informasi.

	Biaya Pencarian (Node yang dihasilkan)			Faktor Percabangan yang Effektif		
d	IDS	$(A^*)h_1$	$(A^*)h_2$	IDS	$(A^*)h_1$	$(A^*)h_2$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Gambar 3.29 Perbandingan biaya pencarian dan faktor percabangan efektif untuk algoritma ITERATIVE-DEEPENING-SEARCH dan A* dengan h_1, h_2. Data dirata-ratakan dari 100 contoh teka-teki 8 untuk setiap panjang solusi d.

Orang mungkin bertanya apakah h_2 selalu lebih baik daripada h_1. Jawabannya adalah "Pada dasarnya, ya." Mudah untuk melihat dari definisi kedua heuristik bahwa, untuk setiap node n, $h_2(n) \geq h_1(n)$. Dengan demikian, kita katakan bahwa h_2 mendominasi h_1. Dominasi diterjemahkan langsung ke dalam efisiensi: A* menggunakan h_2 tidak akan pernah memperluas lebih banyak node daripada A* menggunakan h_1 (kecuali mungkin untuk beberapa node dengan $f(n) = C^*$.

Argumennya sederhana. Ingat kembali pengamatan bahwa setiap node dengan $f(n) < C^*$pasti akan diperluas. Ini sama dengan mengatakan bahwa setiap node dengan $h(n) < C^* - g(n)$ pasti akan diperluas. Tetapi karena h_2 setidaknya sebesar h_1 untuk semua node, setiap node yang pasti diperluas oleh pencarian A* dengan h_2 juga pasti akan diperluas dengan h_1, dan h_1 dapat menyebabkan node lain juga diperluas. Oleh karena itu, secara umum lebih baik menggunakan fungsi heuristik dengan nilai yang lebih tinggi, asalkan konsisten dan waktu komputasi untuk heuristik tidak terlalu lama.

Menghasilkan Heuristik yang Dapat Diterima dari Masalah yang Dilonggarkan

Kita telah melihat bahwa h_1 (petak yang salah tempat) dan h_2 (jarak Manhattan) merupakan heuristik yang cukup baik untuk teka-teki 8 dan bahwa h_2 lebih baik. Bagaimana seseorang dapat menemukan h_2? Apakah mungkin komputer menciptakan heuristik seperti itu secara mekanis? h_1 dan h_2 merupakan estimasi panjang lintasan yang tersisa untuk teka-teki 8, tetapi keduanya juga merupakan panjang lintasan yang sangat akurat untuk versi teka-teki yang disederhanakan.

Jika aturan teka-teki diubah sehingga petak dapat bergerak ke mana saja, bukan hanya ke petak kosong yang berdekatan, maka h_1 akan memberikan jumlah langkah yang tepat dalam solusi terpendek. Demikian pula, jika petak dapat bergerak satu petak ke arah mana pun,

bahkan ke petak yang ditempati, maka h_2 akan memberikan jumlah langkah yang tepat dalam solusi terpendek. Masalah dengan lebih sedikit pembatasan pada tindakan disebut masalah yang dilonggarkan.

Grafik ruang keadaan dari masalah yang dilonggarkan merupakan supergraf dari ruang keadaan asli karena penghapusan pembatasan menciptakan sisi tambahan dalam grafik. Karena masalah yang dilonggarkan menambahkan sisi ke ruang keadaan, setiap solusi optimal dalam masalah asli, menurut definisi, juga merupakan solusi dalam masalah yang dilonggarkan; tetapi masalah yang dilonggarkan mungkin memiliki solusi yang lebih baik jika sisi yang ditambahkan menyediakan jalan pintas. Oleh karena itu, biaya solusi optimal untuk masalah yang dilonggarkan adalah heuristik yang dapat diterima untuk masalah asli. Lebih jauh, karena heuristik yang diturunkan adalah biaya yang tepat untuk masalah yang dilonggarkan, ia harus mematuhi ketidaksetaraan segitiga dan karenanya konsisten.

Jika definisi masalah ditulis dalam bahasa formal, adalah mungkin untuk membangun masalah yang dilonggarkan secara otomatis. Misalnya, jika 8 tindakan teka-teki dijelaskan sebagai

Sebuah ubin dapat bergerak dari kotak A ke kotak B jika

A berdekatan secara horizontal atau vertikal dengan B dan B kosong,

kita dapat menghasilkan tiga masalah yang dilonggarkan dengan menghapus satu atau kedua kondisi:

a) Sebuah ubin dapat bergerak dari kotak A ke kotak B jika A berdekatan dengan B.
b) Sebuah ubin dapat bergerak dari kotak A ke kotak B jika B kosong.
c) Sebuah petak dapat bergerak dari kotak A ke kotak B.

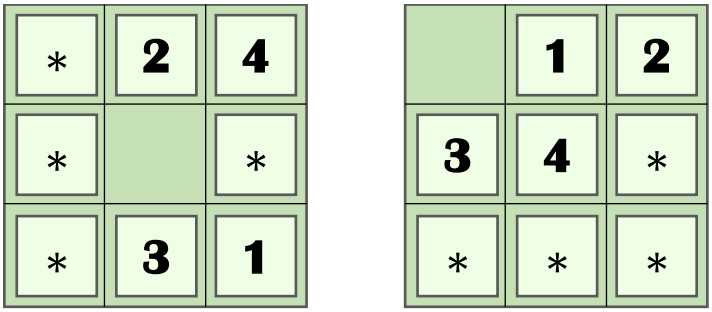
Dari (a), kita dapat memperoleh h_2 (jarak Manhattan). Alasannya adalah bahwa h_2 akan menjadi skor yang tepat jika kita memindahkan setiap petak secara bergiliran ke tujuannya. Dari (c), kita dapat memperoleh h_1 (petak yang salah tempat) karena akan menjadi skor yang tepat jika petak dapat bergerak ke tujuan yang dituju dalam satu langkah. Perhatikan bahwa sangat penting bahwa masalah yang dilonggarkan yang dihasilkan oleh teknik ini dapat dipecahkan pada dasarnya tanpa pencarian, karena aturan yang dilonggarkan memungkinkan masalah tersebut didekomposisi menjadi delapan submasalah yang independen. Jika masalah yang dilonggarkan sulit dipecahkan, maka nilai heuristik yang sesuai akan mahal untuk diperoleh.

Sebuah program yang disebut ABSOLVER dapat menghasilkan heuristik secara otomatis dari definisi masalah, menggunakan metode "masalah yang dilonggarkan" dan berbagai teknik lainnya. ABSOLVER menghasilkan heuristik baru untuk teka-teki 8 teka-teki yang lebih baik daripada heuristik yang sudah ada sebelumnya dan menemukan heuristik pertama yang berguna untuk teka-teki Rubik's Cube yang terkenal.

Satu masalah dengan menghasilkan fungsi heuristik baru adalah bahwa seseorang sering gagal mendapatkan satu heuristik "yang jelas terbaik". Jika kumpulan heuristik yang dapat diterima $h_1 \ldots h_m$ tersedia untuk suatu masalah dan tidak ada satu pun yang

mendominasi yang lain, mana yang harus kita pilih? Ternyata, kita tidak perlu membuat pilihan. Kita dapat memiliki yang terbaik dari semua dunia, dengan mendefinisikan:

$$h(n) = max\{h_1(n), \ldots, h_m(n)\}.$$



Negarai Mulai Negara Tujuan

Gambar 3.30 Submasalah dari contoh 8 teka-teki yang diberikan pada Gambar 3.28. Tugasnya adalah menempatkan ubin 1, 2, 3, dan 4 pada posisi yang benar, tanpa perlu khawatir tentang apa yang terjadi pada ubin lainnya.

Heuristik komposit ini menggunakan fungsi mana pun yang paling akurat pada simpul yang dimaksud. Karena heuristik komponen dapat diterima, h dapat diterima; mudah juga untuk membuktikan bahwa h konsisten. Lebih jauh, h mendominasi semua heuristik komponennya.

Menghasilkan Heuristik yang Dapat Diterima dari Submasalah: Basis Data Pola

Heuristik yang dapat diterima juga dapat diperoleh dari biaya solusi submasalah dari masalah yang diberikan. Misalnya, Gambar 3.30 menunjukkan submasalah dari contoh 8 teka-teki pada Gambar 3.28. Submasalah melibatkan penempatan petak 1, 2, 3, 4 pada posisi yang benar. Jelas, biaya solusi optimal dari submasalah ini adalah batas bawah pada biaya masalah yang lengkap. Ternyata lebih akurat daripada jarak Manhattan dalam beberapa kasus. Ide di balik basis data pola adalah untuk menyimpan biaya solusi yang tepat ini untuk setiap kemungkinan contoh submasalah—dalam contoh kita, setiap kemungkinan konfigurasi dari empat petak dan petak kosong. (Lokasi dari empat petak lainnya tidak relevan untuk tujuan penyelesaian submasalah, tetapi pergerakan petak-petak tersebut dihitung terhadap biaya.)

Kemudian kita menghitung heuristik h_{DB} yang dapat diterima untuk setiap status lengkap yang ditemukan selama pencarian hanya dengan mencari konfigurasi submasalah yang sesuai dalam basis data. Basis data itu sendiri dibangun dengan mencari kembali dari tujuan dan mencatat biaya setiap pola baru yang ditemukan; biaya pencarian ini diamortisasi selama banyak contoh masalah berikutnya.

Pilihan 1-2-3-4 cukup arbitrer; kita juga dapat membangun basis data untuk 5-6-7-8, untuk 2-4-6-8, dan seterusnya. Setiap basis data menghasilkan heuristik yang dapat diterima, dan heuristik ini dapat digabungkan, seperti yang dijelaskan sebelumnya, dengan mengambil nilai maksimum. Heuristik gabungan semacam ini jauh lebih akurat daripada jarak Manhattan; jumlah simpul yang dihasilkan saat memecahkan 15 teka-teki acak dapat dikurangi dengan faktor 1000.

Orang mungkin bertanya-tanya apakah heuristik yang diperoleh dari basis data 1-2-3-4 dan 5-6-7-8 dapat ditambahkan, karena kedua submasalah tersebut tampaknya tidak tumpang tindih. Apakah ini masih memberikan heuristik yang dapat diterima? Jawabannya tidak, karena solusi dari submasalah 1-2-3-4 dan submasalah 5-6-7-8 untuk keadaan tertentu hampir pasti akan berbagi beberapa gerakan—tidak mungkin 1-2-3-4 dapat dipindahkan ke tempatnya tanpa menyentuh 5-6-7-8, dan sebaliknya. Tetapi bagaimana jika kita tidak menghitung gerakan tersebut? Artinya, kita tidak mencatat total biaya untuk memecahkan submasalah 1-2-3-4, tetapi hanya jumlah gerakan yang melibatkan 1-2-3-4. Maka mudah untuk melihat bahwa jumlah dari kedua biaya tersebut masih merupakan batas bawah biaya penyelesaian seluruh masalah. Inilah ide di balik basis data pola terpisah. Dengan basis data tersebut, dimungkinkan untuk menyelesaikan 15 teka-teki acak dalam beberapa milidetik—jumlah simpul yang dihasilkan berkurang dengan faktor 10.000 dibandingkan dengan penggunaan jarak Manhattan. Untuk 24 teka-teki, percepatan sekitar satu juta faktor dapat diperoleh.

Basis data pola terpisah berfungsi untuk teka-teki ubin geser karena masalahnya dapat dibagi sedemikian rupa sehingga setiap gerakan hanya memengaruhi satu submasalah—karena hanya satu ubin yang dipindahkan pada satu waktu. Untuk masalah seperti Kubus Rubik, pembagian semacam ini sulit karena setiap gerakan memengaruhi 8 atau 9 dari 26 kubus. Cara yang lebih umum untuk mendefinisikan heuristik aditif yang dapat diterima telah diusulkan yang berlaku untuk kubus Rubik, tetapi cara tersebut belum menghasilkan heuristik yang lebih baik daripada heuristik nonaditif terbaik untuk masalah tersebut.

Mempelajari Heuristik dari Pengalaman

Fungsi heuristik $h(n)$ seharusnya memperkirakan biaya solusi yang dimulai dari keadaan di simpul n. Bagaimana agen dapat menyusun fungsi tersebut? Satu solusi telah diberikan di bagian sebelumnya—yaitu, untuk merancang masalah yang lebih mudah yang solusi optimalnya dapat ditemukan dengan mudah. Solusi lainnya adalah belajar dari pengalaman. "Pengalaman" di sini berarti memecahkan banyak teka-teki 8, misalnya. Setiap solusi optimal untuk masalah 8 teka-teki memberikan contoh-contoh yang dapat dipelajari dari $h(n)$.

Setiap contoh terdiri dari keadaan dari jalur solusi dan biaya aktual solusi dari titik tersebut. Dari contoh-contoh ini, algoritma pembelajaran dapat digunakan untuk menyusun fungsi $h(n)$ yang dapat (dengan keberuntungan) memprediksi biaya solusi untuk keadaan lain yang muncul selama pencarian. Teknik untuk melakukan hal ini menggunakan jaringan saraf, pohon keputusan.

Metode pembelajaran induktif bekerja paling baik ketika dilengkapi dengan fitur-fitur keadaan yang relevan untuk memprediksi nilai keadaan, daripada hanya dengan deskripsi keadaan mentah. Misalnya, fitur "jumlah petak yang salah tempat" mungkin berguna dalam memprediksi jarak sebenarnya suatu keadaan dari tujuan. Mari kita sebut fitur ini $x_1(n)$. Kita dapat mengambil 100 konfigurasi 8-teka-teki yang dibuat secara acak dan mengumpulkan statistik tentang biaya solusi aktualnya.

Kita mungkin menemukan bahwa ketika $x_1(n)$ adalah 5, biaya solusi rata-rata adalah sekitar 14, dan seterusnya. Berdasarkan data ini, nilai x_1 dapat digunakan untuk memprediksi

$h(n)$. Tentu saja, kita dapat menggunakan beberapa fitur. Fitur kedua $x_2(n)$ mungkin adalah "jumlah pasangan petak yang berdekatan yang tidak berdekatan dalam keadaan tujuan." Bagaimana $x_1(n)$ dan $x_2(n)$ harus dikombinasikan untuk memprediksi $h(n)$? Pendekatan yang umum adalah menggunakan kombinasi linear:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) .$$

Konstanta c_1 dan c_2 disesuaikan untuk memberikan kecocokan terbaik dengan data aktual pada biaya solusi. Kita mengharapkan c_1 dan c_2 menjadi positif karena petak yang salah tempat dan pasangan yang berdekatan yang salah membuat masalah lebih sulit dipecahkan. Perhatikan bahwa heuristik ini memenuhi kondisi bahwa $h(n) = 0$ untuk status tujuan, tetapi tidak selalu dapat diterima atau konsisten.

3.7 RINGKASAN

Bab ini memperkenalkan metode yang dapat digunakan agen untuk memilih tindakan dalam lingkungan yang deterministik, dapat diamati, statis, dan sepenuhnya diketahui. Dalam kasus seperti itu, agen dapat menyusun urutan tindakan yang mencapai tujuannya; proses ini disebut pencarian.

- Sebelum agen dapat mulai mencari solusi, tujuan harus diidentifikasi dan masalah yang didefinisikan dengan baik harus dirumuskan.
- Masalah terdiri dari lima bagian: keadaan awal, serangkaian tindakan, model transisi yang menggambarkan hasil tindakan tersebut, fungsi uji tujuan, dan fungsi biaya jalur. Lingkungan masalah direpresentasikan oleh ruang keadaan. Jalur melalui ruang keadaan dari keadaan awal ke keadaan tujuan adalah solusi.
- Algoritma pencarian memperlakukan keadaan dan tindakan sebagai atomik: mereka tidak mempertimbangkan struktur internal apa pun yang mungkin dimilikinya.
- Algoritma PENCARIAN-POHON umum mempertimbangkan semua jalur yang mungkin untuk menemukan solusi, sedangkan algoritma PENCARIAN-GRAF menghindari pertimbangan jalur yang redundan.
- Algoritma pencarian dinilai berdasarkan kelengkapan, keoptimalan, kompleksitas waktu, dan kompleksitas ruang. Kompleksitas bergantung pada b, faktor percabangan dalam ruang keadaan, dan d, kedalaman solusi paling dangkal.
- Metode pencarian tanpa informasi hanya memiliki akses ke definisi masalah. Algoritma dasarnya adalah sebagai berikut:
  - Pencarian breadth-first memperluas simpul paling dangkal terlebih dahulu; pencarian ini lengkap, optimal untuk biaya langkah satuan, tetapi memiliki kompleksitas ruang eksponensial.
  - Pencarian biaya seragam memperluas simpul dengan biaya jalur terendah, g(n), dan optimal untuk biaya langkah umum.
  - Pencarian depth-first memperluas simpul terdalam yang belum diperluas terlebih dahulu. Pencarian ini tidak lengkap maupun optimal, tetapi memiliki

kompleksitas ruang linier. Pencarian terbatas kedalaman menambahkan batas kedalaman.
  - Pencarian pendalaman berulang memanggil pencarian depth-first dengan batas kedalaman yang meningkat hingga tujuan ditemukan. Pencarian ini lengkap, optimal untuk biaya langkah satuan, memiliki kompleksitas waktu yang sebanding dengan pencarian breadth-first, dan memiliki kompleksitas ruang linier.
  - Pencarian dua arah dapat mengurangi kompleksitas waktu secara drastis, tetapi tidak selalu dapat diterapkan dan mungkin memerlukan terlalu banyak ruang.
- Metode pencarian yang terinformasi mungkin memiliki akses ke fungsi heuristik $h(n)$ yang memperkirakan biaya solusi dari n.
  - Algoritma pencarian best-first generik memilih simpul untuk perluasan menurut fungsi evaluasi.
  - Pencarian best-first yang serakah memperluas simpul dengan $h(n)$ minimal. Ini tidak optimal tetapi sering kali efisien.
  - Pencarian A^* memperluas simpul dengan $f(n) = g(n) + h(n)$ minimal. A^* lengkap dan optimal, asalkan $h(n)$ dapat diterima (untuk PENCARIAN-POHON) atau konsisten (untuk PENCARIAN-GRAF). Kompleksitas ruang A^* masih mahal.
  - RBFS (recursive best-first search) dan SMA^* (simplified memory-bounded A^*) adalah algoritma pencarian yang tangguh dan optimal yang menggunakan memori dalam jumlah terbatas; jika diberi waktu yang cukup, mereka dapat memecahkan masalah yang tidak dapat dipecahkan A^* karena kehabisan memori.
- Kinerja algoritma pencarian heuristik bergantung pada kualitas fungsi heuristik. Seseorang terkadang dapat membangun heuristik yang baik dengan melonggarkan definisi masalah, dengan menyimpan biaya solusi yang telah dihitung sebelumnya untuk submasalah dalam basis data pola, atau dengan belajar dari pengalaman dengan kelas masalah.

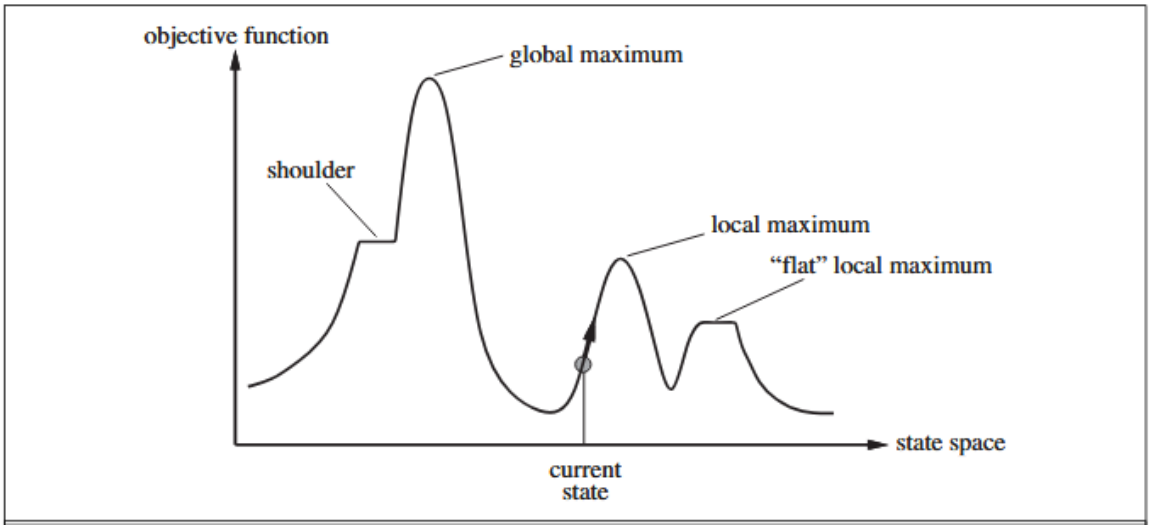
BAB 4
MELAMPAUI PENCARIAN KLASIK

4.1 ALGORITMA PENCARIAN LOKAL DAN MASALAH OPTIMASI

Algoritma pencarian yang telah kita lihat sejauh ini dirancang untuk mengeksplorasi ruang pencarian secara sistematis. Sistematisitas ini dicapai dengan menyimpan satu atau beberapa jalur dalam memori dan dengan mencatat alternatif mana yang telah dieksplorasi di setiap titik di sepanjang jalur tersebut. Ketika suatu tujuan ditemukan, jalur menuju tujuan tersebut juga merupakan solusi untuk masalah tersebut. Namun, dalam banyak masalah, jalur menuju tujuan tersebut tidak relevan. Misalnya, dalam masalah 8 ratu, yang penting adalah konfigurasi akhir ratu, bukan urutan penambahannya. Properti umum yang sama berlaku untuk banyak aplikasi penting seperti desain sirkuit terpadu, tata letak lantai pabrik, penjadwalan bengkel kerja, pemrograman otomatis, pengoptimalan jaringan telekomunikasi, perutean kendaraan, dan manajemen portofolio.

Jika jalur menuju tujuan tidak menjadi masalah, kita dapat mempertimbangkan kelas algoritme yang berbeda, yaitu algoritme yang sama sekali tidak mempedulikan jalur. Algoritme pencarian lokal beroperasi menggunakan satu simpul arus (bukan beberapa jalur) dan umumnya hanya bergerak ke tetangga simpul tersebut. Biasanya, jalur yang diikuti oleh pencarian tidak disimpan. Meskipun algoritme pencarian lokal tidak sistematis, algoritme ini memiliki dua keuntungan utama: (1) algoritme ini menggunakan sedikit memori—biasanya dalam jumlah yang konstan; dan (2) algoritme ini sering kali dapat menemukan solusi yang masuk akal dalam ruang keadaan yang besar atau tak terbatas (kontinu) yang tidak cocok untuk algoritme sistematis.

Selain menemukan tujuan, algoritme pencarian lokal berguna untuk memecahkan masalah optimasi murni, yang tujuannya adalah menemukan keadaan terbaik menurut fungsi objektif. Banyak masalah optimasi tidak sesuai dengan model pencarian "standar" yang diperkenalkan dalam Bab 3. Misalnya, alam menyediakan fungsi objektif—kebugaran reproduksi—yang dapat dilihat sebagai upaya untuk mengoptimalkan evolusi Darwin, tetapi tidak ada "uji tujuan" dan tidak ada "biaya jalur" untuk masalah ini. Untuk memahami pencarian lokal, kami rasa penting untuk mempertimbangkan lanskap ruang-keadaan (seperti pada Gambar 4.1). Sebuah lanskap memiliki "lokasi" (didefinisikan oleh keadaan) dan "ketinggian" (didefinisikan oleh nilai fungsi biaya heuristik atau fungsi tujuan). Jika ketinggian sesuai dengan biaya, maka tujuannya adalah untuk menemukan lembah terendah—minimum global; jika ketinggian sesuai dengan fungsi tujuan, maka tujuannya adalah untuk menemukan puncak tertinggi—maksimum global. (Anda dapat mengonversi dari satu ke yang lain hanya dengan memasukkan tanda minus.) Algoritme pencarian lokal menjelajahi lanskap ini. Algoritme pencarian lokal yang lengkap selalu menemukan tujuan jika ada; algoritme optimal selalu menemukan minimum/maksimum global.



Gambar 4.1 Bentang alam ruang-keadaan satu dimensi yang elevasinya sesuai dengan fungsi objektif. Tujuannya adalah untuk menemukan nilai maksimum global. Pencarian pendakian bukit memodifikasi keadaan saat ini untuk mencoba memperbaikinya, seperti yang ditunjukkan oleh tanda panah. Berbagai fitur topografi didefinisikan dalam teks.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Gambar 4.2 Algoritma pencarian hill-climbing, yang merupakan teknik pencarian lokal paling dasar. Pada setiap langkah, simpul saat ini digantikan oleh tetangga terbaik; dalam versi ini, artinya tetangga dengan NILAI tertinggi, tetapi jika estimasi biaya heuristik h digunakan, kita akan menemukan tetangga dengan h terendah.

Pencarian Pendakian Bukit

Algoritme pencarian pendakian bukit (versi pendakian paling curam) ditunjukkan pada Gambar 4.2. Algoritme ini hanyalah sebuah lingkaran yang terus bergerak ke arah peningkatan nilai—yaitu, menanjak. Algoritme ini berakhir saat mencapai "puncak" di mana tidak ada tetangga yang memiliki nilai lebih tinggi. Algoritme ini tidak memelihara pohon pencarian, jadi struktur data untuk simpul saat ini hanya perlu mencatat status dan nilai fungsi tujuan. Pendakian bukit tidak melihat ke depan melampaui tetangga langsung dari status saat ini. Ini menyerupai upaya menemukan puncak Gunung Everest dalam kabut tebal sambil menderita amnesia.

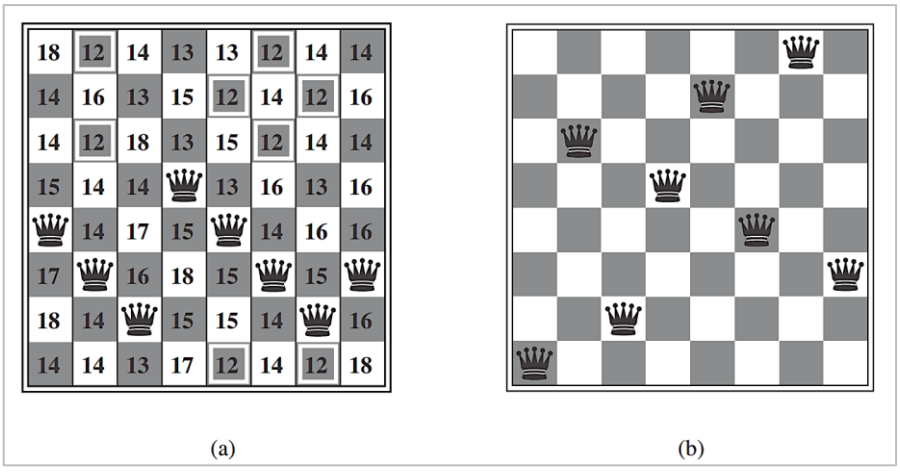
Untuk mengilustrasikan pendakian bukit, kita akan menggunakan masalah 8 ratu. Algoritme pencarian lokal biasanya menggunakan formulasi status lengkap, di mana setiap status memiliki 8 ratu di papan, satu per kolom. Penerus status adalah semua status yang

mungkin dihasilkan dengan memindahkan satu ratu ke kotak lain di kolom yang sama (jadi setiap status memiliki $8 \times 7 = 56$ penerus).

Fungsi biaya heuristik h adalah jumlah pasangan ratu yang saling menyerang, baik secara langsung maupun tidak langsung. Nilai minimum global fungsi ini adalah nol, yang hanya terjadi pada solusi sempurna. Gambar 4.3(a) menunjukkan keadaan dengan $h = 17$. Gambar tersebut juga menunjukkan nilai semua penerusnya, dengan penerus terbaik memiliki $h = 12$. Algoritme hill-climbing biasanya memilih secara acak di antara kumpulan penerus terbaik jika ada lebih dari satu.

Pendakian bukit terkadang disebut pencarian lokal serakah karena mengambil status tetangga yang baik tanpa berpikir ke depan tentang ke mana harus pergi berikutnya. Meskipun keserakahan dianggap sebagai salah satu dari tujuh dosa mematikan, ternyata algoritma serakah sering kali bekerja dengan cukup baik. Pendakian bukit sering kali membuat kemajuan cepat menuju solusi karena biasanya cukup mudah untuk memperbaiki status yang buruk. Misalnya, dari status pada Gambar 4.3(a), hanya dibutuhkan lima langkah untuk mencapai status pada Gambar 4.3(b), yang memiliki $h = 1$ dan hampir menjadi solusi. Sayangnya, pendakian bukit sering kali macet karena alasan berikut:

- **Maksimum lokal**: maksimum lokal adalah puncak yang lebih tinggi daripada setiap status tetangganya tetapi lebih rendah daripada maksimum global. Algoritma pendakian bukit yang mencapai sekitar maksimum lokal akan ditarik ke atas menuju puncak tetapi kemudian akan macet tanpa tujuan lain. Gambar 4.1 mengilustrasikan masalah tersebut secara skematis. Lebih konkretnya, status pada Gambar 4.3(b) adalah maksimum lokal (yaitu, minimum lokal untuk biaya h); setiap gerakan ratu tunggal memperburuk situasi.
- **Punggungan**: punggungan ditunjukkan pada Gambar 4.4. Punggungan menghasilkan urutan titik maksimum lokal yang sangat sulit untuk dinavigasi oleh algoritme serakah.
- **Dataran tinggi**: dataran tinggi adalah area datar dari lanskap ruang-negara. Itu bisa menjadi titik maksimum lokal yang datar, yang darinya tidak ada jalan keluar menanjak, atau bahu jalan, yang darinya kemajuan dimungkinkan. (Lihat Gambar 4.1.) Pencarian pendakian bukit mungkin tersesat di dataran tinggi.

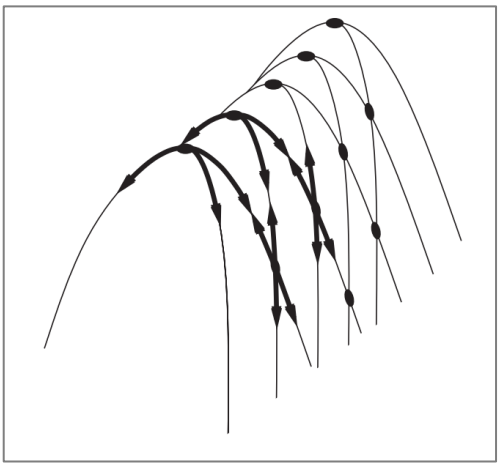


Gambar 4.3 (a) Keadaan 8 ratu dengan estimasi biaya heuristik $h = 17$, yang menunjukkan nilai h untuk setiap kemungkinan penerus yang diperoleh dengan memindahkan ratu di dalam kolomnya. Pergerakan terbaik ditandai. (b) Minimum lokal di ruang keadaan 8 ratu; keadaan memiliki $h = 1$ tetapi setiap penerus memiliki biaya yang lebih tinggi.

Dalam setiap kasus, algoritme mencapai titik di mana tidak ada kemajuan yang dibuat. Dimulai dari negara bagian 8-ratu yang dibuat secara acak, pendakian bukit dengan pendakian paling curam macet 86% dari waktu, hanya menyelesaikan 14% dari contoh masalah. Ia bekerja dengan cepat, hanya mengambil 4 langkah rata-rata ketika berhasil dan 3 ketika macet—lumayan untuk ruang negara bagian dengan $88 \approx 17$ juta negara bagian. Algoritme pada Gambar 4.2 berhenti jika mencapai plateau di mana penerus terbaik memiliki nilai yang sama dengan status saat ini.

Bukankah merupakan ide yang baik untuk terus berjalan—untuk mengizinkan pergerakan menyamping dengan harapan bahwa plateau tersebut benar-benar merupakan bahu jalan, seperti yang ditunjukkan pada Gambar 4.1? Jawabannya biasanya ya, tetapi kita harus berhati-hati. Jika kita selalu mengizinkan pergerakan menyamping ketika tidak ada pergerakan menanjak, loop tak terbatas akan terjadi setiap kali algoritme mencapai maksimum lokal datar yang bukan bahu jalan.

Salah satu solusi umum adalah dengan membatasi jumlah pergerakan menyamping berturut-turut yang diizinkan. Misalnya, kita dapat mengizinkan hingga, katakanlah, 100 pergerakan menyamping berturut-turut dalam masalah 8-ratu. Ini meningkatkan persentase contoh masalah yang diselesaikan dengan mendaki bukit dari 14% menjadi 94%. Keberhasilan datang dengan biaya: algoritme tersebut menghitung rata-rata sekitar 21 langkah untuk setiap contoh yang berhasil dan 64 untuk setiap kegagalan.



Gambar 4.4 Ilustrasi mengapa punggung bukit menyebabkan kesulitan saat mendaki bukit. Kisi-kisi keadaan (lingkaran hitam) ditumpangkan pada punggung bukit yang menjulang dari kiri ke kanan, menciptakan serangkaian titik maksimum lokal yang tidak terhubung langsung satu sama lain. Dari setiap titik maksimum lokal, semua tindakan yang tersedia mengarah ke bawah bukit.

Banyak varian pendakian bukit telah ditemukan. Pendakian bukit stokastik memilih secara acak dari antara gerakan menanjak; probabilitas pemilihan dapat bervariasi dengan kecuraman gerakan menanjak. Ini biasanya konvergen lebih lambat daripada pendakian paling curam, tetapi di beberapa lanskap negara bagian, ia menemukan solusi yang lebih baik. Pendakian bukit pilihan pertama menerapkan pendakian bukit stokastik dengan menghasilkan penerus secara acak hingga satu yang dihasilkan lebih baik daripada keadaan saat ini. Ini adalah strategi yang baik ketika suatu negara memiliki banyak (misalnya, ribuan) penerus.

Algoritme pendakian bukit yang dijelaskan sejauh ini tidak lengkap—mereka sering gagal menemukan tujuan ketika ada karena mereka dapat terjebak pada maksimum lokal. Pendakian bukit acak-ulang mengadopsi pepatah terkenal, "Jika pada awalnya Anda tidak berhasil, coba, coba lagi." Ia melakukan serangkaian pencarian pendakian bukit dari keadaan awal yang dihasilkan secara acak, hingga tujuan ditemukan. Ia secara sepele lengkap dengan probabilitas mendekati 1, karena ia pada akhirnya akan menghasilkan keadaan tujuan sebagai keadaan awal.

Jika setiap pencarian pendakian bukit memiliki probabilitas p untuk berhasil, maka jumlah restart yang diharapkan adalah $1/p$. Untuk contoh 8 ratu tanpa gerakan menyamping yang diizinkan, $p \approx 0{,}14$, jadi kita memerlukan sekitar 7 iterasi untuk menemukan sasaran (6 kegagalan dan 1 keberhasilan). Jumlah langkah yang diharapkan adalah biaya satu iterasi yang berhasil ditambah $(1-p)/p$ dikalikan biaya kegagalan, atau sekitar 22 langkah secara keseluruhan. Ketika kita mengizinkan gerakan menyamping, $1/0{,}94 \approx 1{,}06$ iterasi diperlukan rata-rata dan $(1 \times 21) + (0{,}06/0{,}94) \times 64 \approx 25$ langkah. Untuk 8 ratu, pendakian bukit dengan restart acak sangat efektif. Bahkan untuk tiga juta ratu, pendekatan ini dapat menemukan solusi dalam waktu kurang dari satu menit.

Keberhasilan pendakian bukit sangat bergantung pada bentuk lanskap ruang-keadaan: jika terdapat sedikit titik maksimum dan plateaux lokal, pendakian bukit dengan memulai ulang secara acak akan menemukan solusi yang baik dengan sangat cepat. Di sisi lain, banyak masalah nyata memiliki lanskap yang lebih mirip keluarga landak botak yang tersebar luas di lantai datar, dengan landak mini yang hidup di ujung setiap jarum landak, tanpa batas. Masalah NP-keras biasanya memiliki jumlah titik maksimum lokal eksponensial untuk dicakup. Meskipun demikian, titik maksimum lokal yang cukup baik sering kali dapat ditemukan setelah sejumlah kecil restart.

Simulasi Annealing

Algoritma pendakian bukit yang tidak pernah melakukan gerakan "menurun" menuju keadaan dengan nilai yang lebih rendah (atau biaya yang lebih tinggi) dijamin tidak lengkap, karena dapat macet pada titik maksimum lokal. Sebaliknya, jalan yang benar-benar acak—yaitu, bergerak ke penerus yang dipilih secara acak dari kumpulan penerus—sudah lengkap tetapi sangat tidak efisien. Oleh karena itu, tampaknya masuk akal untuk mencoba menggabungkan pendakian bukit dengan jalan acak dengan cara tertentu yang menghasilkan efisiensi dan kelengkapan.

Simulasi anil adalah salah satu algoritma tersebut. Dalam metalurgi, anil adalah proses yang digunakan untuk melunakkan atau mengeraskan logam dan kaca dengan

memanaskannya ke suhu tinggi dan kemudian secara bertahap mendinginkannya, sehingga memungkinkan material mencapai keadaan kristal berenergi rendah. Untuk menjelaskan simulasi anil, kami mengalihkan sudut pandang kami dari pendakian bukit ke penurunan gradien (yaitu, meminimalkan biaya) dan membayangkan tugas memasukkan bola pingpong ke celah terdalam di permukaan yang bergelombang.

Jika kita membiarkan bola menggelinding, bola akan berhenti di minimum lokal. Jika kita menggoyang permukaan, kita dapat memantulkan bola keluar dari minimum lokal. Triknya adalah menggoyang cukup keras untuk memantulkan bola keluar dari minimum lokal tetapi tidak cukup keras untuk melepaskannya dari minimum global. Solusi simulasi annealing adalah dengan memulai dengan mengocok dengan keras (yaitu, pada suhu tinggi) dan kemudian secara bertahap mengurangi intensitas pengocokan (yaitu, menurunkan suhu).

Lingkaran terdalam dari algoritma simulasi annealing (Gambar 4.5) cukup mirip dengan pendakian bukit. Namun, alih-alih memilih langkah terbaik, ia memilih langkah acak. Jika langkah tersebut memperbaiki situasi, maka langkah tersebut selalu diterima. Jika tidak, algoritma menerima langkah tersebut dengan beberapa probabilitas kurang dari 1. Probabilitas menurun secara eksponensial dengan "keburukan" langkah tersebut—jumlah ΔE yang memperburuk evaluasi. Probabilitas juga menurun saat "suhu" T turun: langkah "buruk" lebih mungkin diizinkan di awal saat T tinggi, dan menjadi lebih tidak mungkin saat T menurun. Jika jadwal menurunkan T cukup lambat, algoritme akan menemukan optimum global dengan probabilitas mendekati 1.

Simulasi annealing pertama kali digunakan secara luas untuk memecahkan masalah tata letak VLSI pada awal 1980an. Simulasi ini telah diterapkan secara luas pada penjadwalan pabrik dan tugas pengoptimalan skala besar lainnya.

Pencarian Berkas Lokal

Menyimpan hanya satu simpul dalam memori mungkin tampak sebagai reaksi ekstrem terhadap masalah keterbatasan memori. Algoritme pencarian berkas lokal melacak k status, bukan hanya satu. Algoritme ini dimulai dengan k status yang dihasilkan secara acak. Pada setiap langkah, semua penerus dari semua k status dihasilkan. Jika ada yang menjadi tujuan, algoritme berhenti. Jika tidak, algoritme memilih k penerus terbaik dari daftar lengkap dan mengulang.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs:  problem, a problem
             schedule, a mapping from time to "temperature"

    current ← MAKE-NODE(problem.INITIAL-STATE)
    for t = 1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.VALUE – current.VALUE
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
```

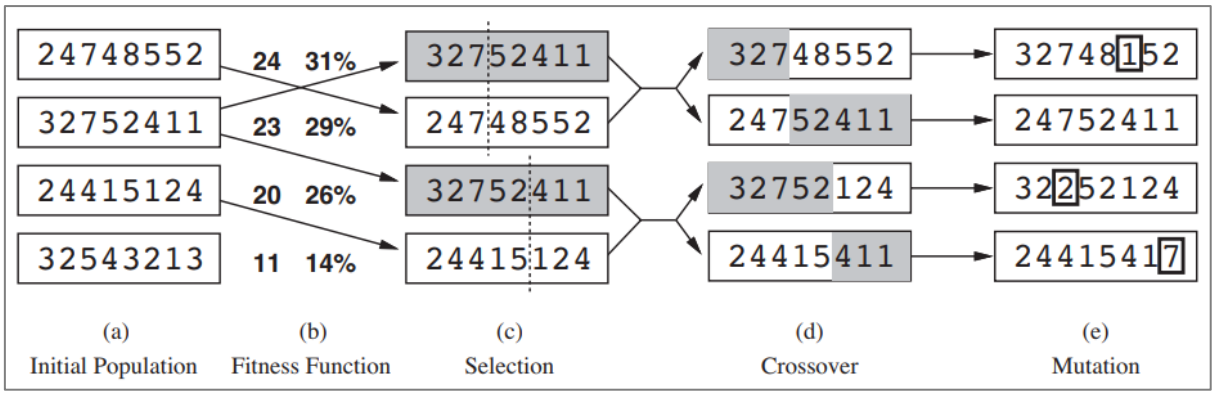
Gambar 4.5 Algoritma simulasi anil, versi pendakian bukit stokastik di mana beberapa gerakan menurun diperbolehkan. Gerakan menurun diterima dengan mudah di awal jadwal anil dan kemudian lebih jarang seiring berjalannya waktu. Masukan jadwal menentukan nilai suhu T sebagai fungsi waktu.

Sekilas, pencarian berkas lokal dengan k status mungkin tampak tidak lebih dari sekadar menjalankan k restart acak secara paralel, bukan berurutan. Faktanya, kedua algoritme tersebut sangat berbeda. Dalam pencarian restart acak, setiap proses pencarian berjalan secara independen dari yang lain. Dalam pencarian berkas lokal, informasi yang berguna diteruskan di antara utas pencarian paralel. Akibatnya, status yang menghasilkan penerus terbaik berkata kepada yang lain, "Kemarilah, rumput lebih hijau!" Algoritme tersebut dengan cepat meninggalkan pencarian yang tidak membuahkan hasil dan memindahkan sumber dayanya ke tempat yang paling banyak mengalami kemajuan.

Dalam bentuknya yang paling sederhana, pencarian berkas lokal dapat mengalami kekurangan keragaman di antara k status—status tersebut dapat dengan cepat terkonsentrasi di wilayah kecil ruang status, sehingga pencarian tersebut tidak lebih dari sekadar versi pendakian bukit yang mahal. Varian yang disebut pencarian berkas stokastik, yang analog dengan pendakian bukit stokastik, membantu mengatasi masalah ini. Alih-alih memilih k terbaik dari kumpulan kandidat penerus, pencarian berkas stokastik memilih k penerus secara acak, dengan probabilitas memilih penerus tertentu sebagai fungsi peningkatan nilainya. Pencarian berkas stokastik memiliki beberapa kemiripan dengan proses seleksi alam, di mana "penerus" (keturunan) dari suatu "keadaan" (organisme) mengisi generasi berikutnya sesuai dengan "nilainya" (kebugaran).

Algoritma Genetika

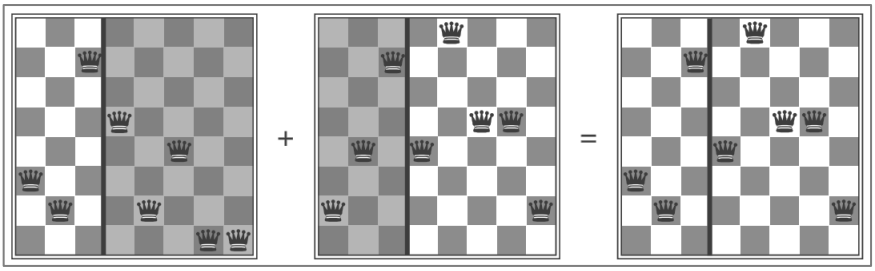
Algoritma genetika (atau GA) adalah varian dari pencarian berkas stokastik di mana keadaan penerus dihasilkan dengan menggabungkan dua keadaan induk, bukan dengan memodifikasi satu keadaan. Analogi dengan seleksi alam sama seperti dalam pencarian berkas stokastik, kecuali bahwa sekarang kita berurusan dengan reproduksi seksual, bukan reproduksi aseksual.



Gambar 4.6 Algoritma genetik, diilustrasikan untuk rangkaian digit yang mewakili status 8 ratu. Populasi awal pada (a) diurutkan berdasarkan fungsi kebugaran pada (b), menghasilkan

pasangan untuk perkawinan pada (c). Mereka menghasilkan keturunan pada (d), yang mengalami mutasi pada (e).

Seperti pencarian berkas, GA dimulai dengan sekumpulan k status yang dihasilkan secara acak, yang disebut populasi. Setiap status, atau individu, direpresentasikan sebagai string di atas alfabet terbatas—paling umum, string 0 dan 1. Misalnya, status 8 ratu harus menentukan posisi 8 ratu, masing-masing dalam kolom 8 kotak, dan karenanya memerlukan $8 \times \log_2 8 = 24$ bit. Atau, status dapat direpresentasikan sebagai 8 digit, masing-masing dalam rentang dari 1 hingga 8. (Kami akan menunjukkan nanti bahwa kedua penyandian berperilaku berbeda.) Gambar 4.6(a) menunjukkan populasi empat string 8 digit yang mewakili status 8 ratu.



Gambar 4.7 Kedelapan negara ratu yang sesuai dengan dua orang tua pertama pada Gambar 4.6(c) dan keturunan pertama pada Gambar 4.6(d). Kolom yang diarsir hilang pada langkah persilangan dan kolom yang tidak diarsir dipertahankan.

Produksi status generasi berikutnya ditunjukkan pada Gambar 4.6(b)–(e). Dalam (b), setiap status dinilai oleh fungsi objektif, atau (dalam terminologi GA) fungsi kebugaran. Fungsi kebugaran harus mengembalikan nilai yang lebih tinggi untuk keadaan yang lebih baik, jadi, untuk masalah 8 ratu, kami menggunakan jumlah pasangan ratu yang tidak menyerang, yang memiliki nilai 28 sebagai solusi. Nilai dari keempat keadaan tersebut adalah 24, 23, 20, dan 11. Dalam varian algoritma genetika ini, probabilitas untuk dipilih untuk bereproduksi berbanding lurus dengan skor kebugaran, dan persentasenya ditunjukkan di samping skor mentah.

Dalam (c), dua pasang dipilih secara acak untuk reproduksi, sesuai dengan probabilitas dalam (b). Perhatikan bahwa satu individu dipilih dua kali dan satu tidak dipilih sama sekali. Untuk setiap pasangan yang akan dikawinkan, titik persilangan dipilih secara acak dari posisi dalam untaian. Dalam Gambar 4.6, titik persilangan berada setelah digit ketiga dalam pasangan pertama dan setelah digit kelima dalam pasangan kedua.

Dalam (d), keturunannya sendiri dibuat dengan menyilangkan untaian induk pada titik persilangan. Misalnya, anak pertama dari pasangan pertama mendapatkan tiga digit pertama dari orang tua pertama dan digit yang tersisa dari orang tua kedua, sedangkan anak kedua mendapatkan tiga digit pertama dari orang tua kedua dan sisanya dari orang tua pertama. Keadaan 8 ratu yang terlibat dalam langkah reproduksi ini ditunjukkan pada Gambar 4.7.

Contoh tersebut menunjukkan bahwa ketika dua keadaan orang tua sangat berbeda, operasi persilangan dapat menghasilkan keadaan yang jauh dari keadaan orang tua mana pun. Sering kali populasi cukup beragam di awal proses, jadi persilangan (seperti simulasi annealing)

sering kali mengambil langkah besar dalam ruang keadaan di awal proses pencarian dan langkah-langkah yang lebih kecil di kemudian hari ketika sebagian besar individu cukup mirip.

Terakhir, dalam (e), setiap lokasi tunduk pada mutasi acak dengan probabilitas independen yang kecil. Satu digit bermutasi pada keturunan pertama, ketiga, dan keempat. Dalam masalah 8 ratu, ini sesuai dengan memilih ratu secara acak dan memindahkannya ke kotak acak di kolomnya. Gambar 4.8 menjelaskan algoritma yang mengimplementasikan semua langkah ini. Seperti pencarian berkas stokastik, algoritma genetik menggabungkan kecenderungan menanjak dengan eksplorasi acak dan pertukaran informasi di antara utas pencarian paralel. Keuntungan utama, jika ada, dari algoritma genetik berasal dari operasi persilangan. Namun, secara matematis dapat ditunjukkan bahwa, jika posisi kode genetik awalnya diubah dalam urutan acak, persilangan tidak memberikan keuntungan apa pun. Secara intuitif, keuntungan tersebut berasal dari kemampuan persilangan untuk menggabungkan blok huruf besar yang telah berevolusi secara independen untuk melakukan fungsi yang berguna, sehingga meningkatkan tingkat ketelitian pengoperasian pencarian. Misalnya, bisa jadi menempatkan tiga ratu pertama di posisi 2, 4, dan 6 (di mana mereka tidak saling menyerang) merupakan blok berguna yang dapat digabungkan dengan blok lain untuk membangun solusi.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
   inputs: population, a set of individuals
           FITNESS-FN, a function that measures the fitness of an individual

   repeat
      new_population ← empty set
      for i = 1 to SIZE(population) do
         x ← RANDOM-SELECTION(population, FITNESS-FN)
         y ← RANDOM-SELECTION(population, FITNESS-FN)
         child ← REPRODUCE(x, y)
         if (small random probability) then child ← MUTATE(child)
         add child to new_population
      population ← new_population
   until some individual is fit enough, or enough time has elapsed
   return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
inputs: x, y, parent individuals

n ← LENGTH(x); c ←random number from 1 to n
return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Gambar 4.8 Algoritma genetik. Algoritma ini sama dengan yang digambarkan pada Gambar 4.6, dengan satu variasi: dalam versi yang lebih populer ini, setiap perkawinan dua orang tua hanya menghasilkan satu keturunan, bukan dua.

Teori algoritma genetik menjelaskan cara kerjanya dengan menggunakan ide skema, yang merupakan substring di mana beberapa posisi dapat dibiarkan tidak ditentukan. Misalnya,

skema 246***** menggambarkan semua status 8-ratu di mana tiga ratu pertama berada di posisi 2, 4, dan 6, berturut-turut. String yang cocok dengan skema (seperti 24613578) disebut contoh skema. Dapat ditunjukkan bahwa jika kebugaran rata-rata contoh skema berada di atas rata-rata, maka jumlah contoh skema dalam populasi akan tumbuh seiring waktu.

Jelas, efek ini tidak mungkin signifikan jika bit yang berdekatan sama sekali tidak terkait satu sama lain, karena akan ada beberapa blok bersebelahan yang memberikan manfaat yang konsisten. Algoritma genetik bekerja paling baik ketika skema sesuai dengan komponen solusi yang bermakna. Misalnya, jika string adalah representasi antena, maka skema dapat mewakili komponen antena, seperti reflektor dan deflektor. Komponen yang baik kemungkinan besar akan baik dalam berbagai desain yang berbeda. Hal ini menunjukkan bahwa keberhasilan penggunaan algoritma genetika memerlukan rekayasa representasi yang cermat.

Dalam praktiknya, algoritma genetika telah memberikan dampak yang luas pada masalah pengoptimalan, seperti tata letak sirkuit dan penjadwalan pekerjaan. Saat ini, tidak jelas apakah daya tarik algoritma genetika muncul dari kinerjanya atau dari asal-usulnya yang menarik secara estetika dalam teori evolusi. Masih banyak pekerjaan yang harus dilakukan untuk mengidentifikasi kondisi di mana algoritma genetika bekerja dengan baik.

4.2 PENCARIAN LOKAL DALAM RUANG BERKESINAMBUNGAN

Pada Bab 2, kami menjelaskan perbedaan antara lingkungan diskrit dan berkelanjutan, dengan menunjukkan bahwa sebagian besar lingkungan dunia nyata bersifat berkelanjutan. Namun, tidak ada satu pun algoritme yang telah kami jelaskan (kecuali untuk pendakian bukit pilihan pertama dan simulasi anil) yang dapat menangani ruang keadaan dan tindakan berkelanjutan, karena keduanya memiliki faktor percabangan tak terbatas.

Bagian ini memberikan pengantar yang sangat singkat tentang beberapa teknik pencarian lokal untuk menemukan solusi optimal dalam ruang berkelanjutan. Literatur tentang topik ini sangat luas; banyak teknik dasar yang berasal dari abad ke-17, setelah pengembangan kalkulus oleh Newton dan Leibniz. Kami menemukan penggunaan teknik ini di beberapa tempat dalam buku ini, termasuk bab tentang pembelajaran, visi, dan robotika.

Kami mulai dengan sebuah contoh. Misalkan kita ingin menempatkan tiga bandara baru di mana saja di Rumania, sehingga jumlah kuadrat jarak dari setiap kota pada peta (Gambar 3.2) ke bandara terdekatnya diminimalkan. Ruang keadaan kemudian didefinisikan oleh koordinat bandara: $(x_1, y_1), (x_2, y_2)$, dan (x_3, y_3). Ini adalah ruang enam dimensi; kita juga mengatakan bahwa keadaan didefinisikan oleh enam variabel. (Secara umum, keadaan didefinisikan oleh vektor variabel n-dimensi, $\boldsymbol{x}$.)

Bergerak di ruang ini sesuai dengan memindahkan satu atau lebih bandara di peta. Fungsi objektif $f = (x_1, y_1, x_2, y_2, x_3, y_3)$ relatif mudah dihitung untuk keadaan tertentu setelah kita menghitung kota terdekat. Biarkan C_i menjadi himpunan kota yang bandara terdekatnya (dalam keadaan saat ini) adalah bandara i. Kemudian, di lingkungan keadaan saat ini, di mana C_i tetap konstan, kita memiliki

$$f = (x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i + y_c)^2 \qquad (4.1)$$

Ekspresi ini benar secara lokal, tetapi tidak secara global karena himpunan Ci adalah fungsi (tidak kontinu) dari keadaan.

Salah satu cara untuk menghindari masalah kontinu adalah dengan mendiskritkan lingkungan setiap keadaan. Misalnya, kita hanya dapat memindahkan satu bandara pada satu waktu baik dalam arah x maupun y dengan jumlah tetap $\pm\delta$. Dengan 6 variabel, ini memberikan 12 kemungkinan penerus untuk setiap keadaan. Kita kemudian dapat menerapkan salah satu algoritma pencarian lokal yang dijelaskan sebelumnya. Kita juga dapat menerapkan pendakian bukit stokastik dan simulasi anil secara langsung, tanpa mendiskritkan ruang. Algoritma ini memilih penerus secara acak, yang dapat dilakukan dengan menghasilkan vektor acak dengan panjang δ.

Banyak metode mencoba menggunakan gradien lanskap untuk menemukan nilai maksimum. Gradien fungsi tujuan adalah vektor ∇f yang memberikan besaran dan arah kemiringan paling curam. Untuk masalah kita, kita memiliki

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

Dalam beberapa kasus, kita dapat menemukan nilai maksimum dengan menyelesaikan persamaan $\nabla f = 0$. (Ini dapat dilakukan, misalnya, jika kita hanya menempatkan satu bandara; solusinya adalah rata-rata aritmatika dari semua koordinat kota.) Namun, dalam banyak kasus, persamaan ini tidak dapat diselesaikan dalam bentuk tertutup. Misalnya, dengan tiga bandara, ekspresi untuk gradien bergantung pada kota mana yang paling dekat dengan setiap bandara dalam kondisi saat ini. Ini berarti kita dapat menghitung gradien secara lokal (tetapi tidak secara global); misalnya,

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_i} (x_i + x_c) \qquad (4.2)$$

Dengan ekspresi gradien yang benar secara lokal, kita dapat melakukan pendakian bukit dengan tanjakan paling curam dengan memperbarui status terkini sesuai dengan rumus

$$\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\alpha} \nabla \boldsymbol{f}(\mathbf{x}),$$

di mana α adalah konstanta kecil yang sering disebut ukuran langkah. Dalam kasus lain, fungsi objektif mungkin tidak tersedia dalam bentuk yang dapat dibedakan sama sekali—misalnya, nilai dari sekumpulan lokasi bandara tertentu mungkin ditentukan dengan menjalankan beberapa paket simulasi ekonomi berskala besar. Dalam kasus tersebut, kita dapat menghitung

apa yang disebut gradien empiris dengan mengevaluasi respons terhadap kenaikan dan penurunan kecil di setiap koordinat. Pencarian gradien empiris sama dengan pendakian bukit dengan tanjakan paling curam dalam versi ruang keadaan yang didiskritisasi.

Tersembunyi di balik frasa "α adalah konstanta kecil" terdapat berbagai macam metode untuk menyesuaikan α. Masalah dasarnya adalah, jika α terlalu kecil, terlalu banyak langkah yang diperlukan; jika α terlalu besar, pencarian dapat melampaui batas maksimum. Teknik pencarian garis mencoba mengatasi dilema ini dengan memperluas arah gradien saat ini—biasanya dengan menggandakan α berulang kali—hingga f mulai menurun lagi. Titik di mana hal ini terjadi menjadi keadaan saat ini yang baru. Ada beberapa aliran pemikiran tentang bagaimana arah baru harus dipilih pada titik ini.

Untuk banyak masalah, algoritma yang paling efektif adalah metode Newton–Raphson yang sudah dikenal luas. Ini adalah teknik umum untuk menemukan akar fungsi—yaitu, memecahkan persamaan dalam bentuk $g(x) = 0$. Metode ini bekerja dengan menghitung estimasi baru untuk akar x menurut rumus Newton.

$$x \leftarrow x - g(x)/g'(x)$$

Untuk menemukan nilai maksimum atau minimum f, kita perlu menemukan x sehingga gradiennya adalah nol (yaitu, $\nabla f(\mathrm{x}) = 0$. Dengan demikian, $g(x)$ dalam rumus Newton menjadi $\nabla f(\mathrm{x})$, dan persamaan pembaruan dapat ditulis dalam bentuk matriks-vektor sebagai

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_{\boldsymbol{f}}^{-\mathbf{1}}(\mathbf{x})\boldsymbol{\nabla f}(\mathbf{x})$$

di mana $\mathbf{H}_f(\mathbf{x})$ adalah matriks Hessian turunan kedua, yang elemen-elemennya H_{ij} diberikan oleh $\partial^2 f/\partial x_i \partial x_j$. Untuk contoh bandara kita, kita dapat melihat dari Persamaan (4.2) bahwa $\mathbf{H}_f(\mathbf{x})$ sangat sederhana: elemen-elemen off-diagonal adalah nol dan elemen-elemen diagonal untuk bandara i hanya dua kali jumlah kota di C_i. Perhitungan sesaat menunjukkan bahwa satu langkah pembaruan memindahkan bandara i langsung ke pusat massa C_i, yang merupakan minimum dari ekspresi lokal untuk f dari Persamaan (4.1). Namun, untuk masalah berdimensi tinggi, menghitung entri n^2 dari Hessian dan menginversinya mungkin mahal, sehingga banyak versi perkiraan dari metode Newton–Raphson telah dikembangkan.

Metode pencarian lokal mengalami maksima lokal, ridge, dan plateaux dalam ruang keadaan kontinu sama seperti dalam ruang diskrit. Restart acak dan simulasi annealing dapat digunakan dan sering kali membantu. Namun, ruang kontinu berdimensi tinggi merupakan tempat besar yang mudah membuat orang tersesat.

Topik terakhir yang berguna jika Anda mengetahuinya adalah optimisasi terbatas. Masalah optimisasi terbatas jika solusinya harus memenuhi beberapa batasan ketat pada nilai variabel. Misalnya, dalam masalah penentuan lokasi bandara, kita dapat membatasi lokasi agar berada di dalam Rumania dan di daratan (bukan di tengah danau). Kesulitan masalah optimisasi terbatas bergantung pada sifat batasan dan fungsi objektif. Kategori yang paling terkenal adalah masalah pemrograman linier, yang batasannya harus berupa pertidaksamaan

linier yang membentuk himpunan cembung dan fungsi objektifnya juga linier. Kompleksitas waktu pemrograman linier bersifat polinomial dalam jumlah variabel.

Pemrograman linier mungkin merupakan kelas masalah optimisasi yang paling banyak dipelajari dan paling berguna. Ini adalah kasus khusus dari masalah optimisasi cembung yang lebih umum, yang memungkinkan wilayah batasan menjadi wilayah cembung apa pun dan tujuan menjadi fungsi apa pun yang cembung di dalam wilayah batasan. Dalam kondisi tertentu, masalah optimasi konveks juga dapat dipecahkan secara polinomial dan dapat dilakukan dalam praktik dengan ribuan variabel. Beberapa masalah penting dalam pembelajaran mesin dan teori kontrol dapat diformulasikan sebagai masalah optimasi konveks (lihat Bab 20).

4.3 PENCARIAN DENGAN TINDAKAN NONDETERMINISTIK

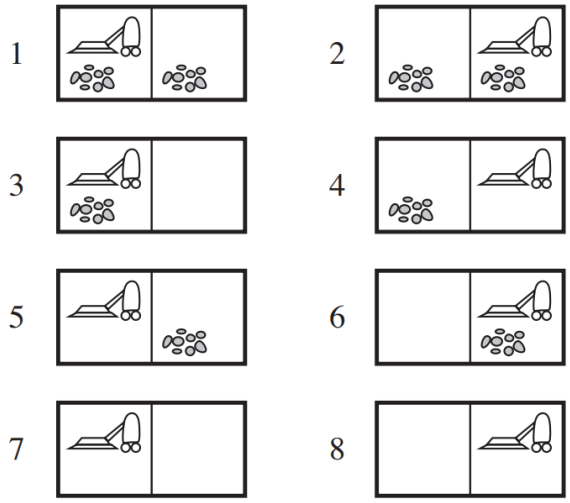
Dalam Bab 3, kami berasumsi bahwa lingkungan sepenuhnya dapat diamati dan deterministik dan bahwa agen mengetahui apa saja dampak dari setiap tindakan. Oleh karena itu, agen dapat menghitung dengan tepat keadaan mana yang dihasilkan dari setiap rangkaian tindakan dan selalu mengetahui keadaan mana yang sedang dialaminya. Persepsinya tidak memberikan informasi baru setelah setiap tindakan, meskipun tentu saja persepsi tersebut memberi tahu agen keadaan awal.

Ketika lingkungan dapat diamati sebagian atau tidak deterministik (atau keduanya), persepsi menjadi berguna. Dalam lingkungan yang dapat diamati sebagian, setiap persepsi membantu mempersempit serangkaian kemungkinan keadaan yang mungkin dialami agen, sehingga memudahkan agen untuk mencapai tujuannya. Ketika lingkungan tidak deterministik, persepsi memberi tahu agen kemungkinan hasil tindakannya yang sebenarnya telah terjadi.

Dalam kedua kasus tersebut, persepsi masa depan tidak dapat ditentukan sebelumnya dan tindakan agen di masa depan akan bergantung pada persepsi masa depan tersebut. Jadi solusi untuk suatu masalah bukanlah suatu urutan melainkan suatu rencana kontinjensi (juga dikenal sebagai strategi) yang menentukan apa yang harus dilakukan tergantung pada persepsi yang diterima. Di bagian ini, kami meneliti kasus nondeterminisme, dengan menunda observabilitas parsial ke Bagian 4.4.

Dunia vakum yang tidak menentu

Sebagai contoh, kami menggunakan dunia vakum, yang pertama kali diperkenalkan di Bab 2 dan didefinisikan sebagai masalah pencarian di Bagian 3.2. Ingat kembali bahwa ruang keadaan memiliki delapan keadaan, seperti yang ditunjukkan pada Gambar 4.9. Ada tiga tindakan—Kiri, Kanan, dan Hisap—dan tujuannya adalah untuk membersihkan semua kotoran (keadaan 7 dan 8). Jika lingkungan dapat diamati, deterministik, dan sepenuhnya diketahui, maka masalah tersebut dapat dipecahkan dengan mudah oleh salah satu algoritma di Bab 3 dan solusinya adalah urutan tindakan. Misalnya, jika keadaan awal adalah 1, maka urutan tindakan [Hisap, Kanan,Hisap] akan mencapai keadaan tujuan, 8.



Gambar 4.9 Delapan kemungkinan keadaan dunia vakum; keadaan 7 dan 8 merupakan keadaan tujuan.

Sekarang anggaplah kita memperkenalkan nondeterminisme dalam bentuk penyedot debu yang kuat tetapi tidak menentu. Dalam dunia vakum yang tidak menentu, tindakan Hisap bekerja sebagai berikut:

- Ketika diterapkan pada kotak yang kotor, tindakan tersebut membersihkan kotak tersebut dan terkadang membersihkan kotoran di kotak yang berdekatan juga.
- Ketika diterapkan pada kotak yang bersih, tindakan tersebut terkadang mengendapkan kotoran di karpet.

Untuk memberikan formulasi yang tepat dari masalah ini, kita perlu menggeneralisasi gagasan tentang model transisi dari Bab 3. Alih-alih mendefinisikan model transisi dengan fungsi HASIL yang mengembalikan satu status, kita menggunakan fungsi HASIL yang mengembalikan serangkaian status hasil yang mungkin. Misalnya, dalam dunia vakum yang tidak menentu, tindakan Hisap dalam status 1 mengarah ke status dalam rangkaian $\{5, 7\}$—kotoran di kotak sebelah kanan mungkin atau mungkin tidak disedot.

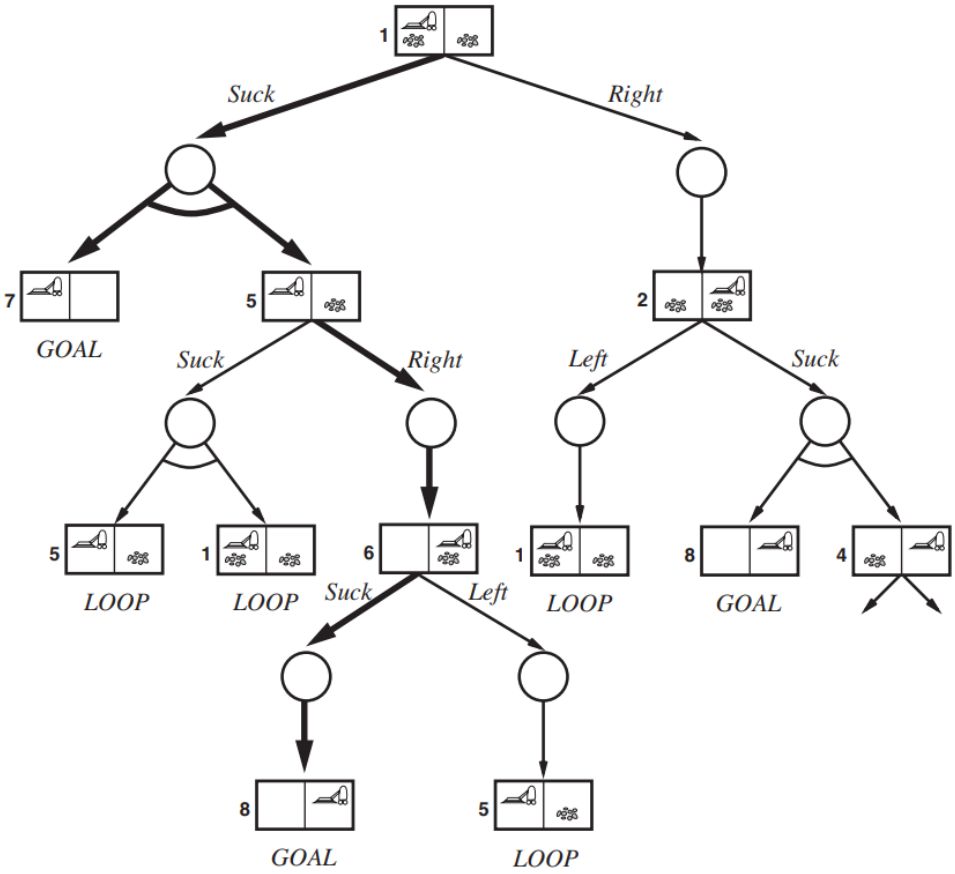
Kita juga perlu menggeneralisasi gagasan tentang solusi untuk masalah tersebut. Misalnya, jika kita mulai dari keadaan 1, tidak ada satu urutan tindakan yang dapat menyelesaikan masalah. Sebaliknya, kita memerlukan rencana kontinjensi seperti berikut:

$$[Menghisap, jika\ Keadaan\ = 5\ maka\ [Benar, Menghisap]\ jika\ tidak\ [\,]]\,. \qquad (4.3)$$

Jadi, solusi untuk masalah nondeterministik dapat berisi pernyataan if–then–else yang bersarang; ini berarti bahwa mereka adalah pohon dan bukan urutan. Ini memungkinkan pemilihan tindakan berdasarkan kontinjensi yang muncul selama eksekusi. Banyak masalah di dunia nyata dan fisik adalah masalah kontinjensi karena prediksi yang tepat tidak mungkin dilakukan. Karena alasan ini, banyak orang tetap membuka mata saat berjalan-jalan atau mengemudi.

Pohon Pencarian AND–OR

Pertanyaan berikutnya adalah bagaimana menemukan solusi kontingen untuk masalah nondeterministik. Seperti dalam Bab 3, kita mulai dengan membangun pohon pencarian, tetapi di sini pohon-pohon tersebut memiliki karakter yang berbeda. Dalam lingkungan deterministik, satu-satunya percabangan diperkenalkan oleh pilihan agen sendiri di setiap keadaan. Kita menyebut simpul-simpul ini sebagai simpul OR. Dalam dunia vakum, misalnya, pada simpul OR agen memilih *Kiri* atau *Kanan atau Hisap*. Dalam lingkungan nondeterministik, percabangan juga diperkenalkan oleh pilihan hasil lingkungan untuk setiap tindakan. Kita menyebut simpul-simpul ini sebagai simpul AND.



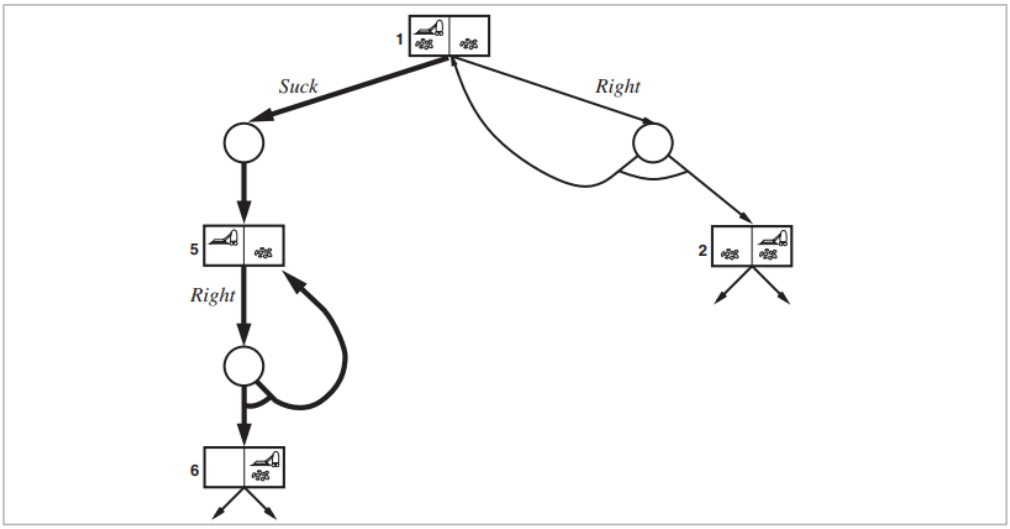
Gambar 4.10 Dua tingkat pertama dari pohon pencarian untuk dunia vakum yang tidak menentu. Node status adalah node OR tempat beberapa tindakan harus dipilih. Pada node AND, yang ditunjukkan sebagai lingkaran, setiap hasil harus ditangani, seperti yang ditunjukkan oleh busur yang menghubungkan cabang-cabang yang keluar. Solusi yang ditemukan ditunjukkan dengan garis tebal.

Misalnya, tindakan Hisap dalam keadaan 1 mengarah ke keadaan dalam himpunan $\{5, 7\}$, sehingga agen perlu menemukan rencana untuk keadaan 5 dan untuk keadaan 7. Kedua jenis simpul ini bergantian, mengarah ke pohon AND–OR seperti yang diilustrasikan dalam Gambar 4.10. Solusi untuk masalah pencarian AND–OR adalah subpohon yang (1) memiliki simpul tujuan di setiap daun, (2) menetapkan satu tindakan di setiap simpul OR-nya, dan (3)

mencakup setiap cabang hasil di setiap simpul AND-nya. Solusinya ditunjukkan dengan garis tebal pada gambar; solusinya sesuai dengan rencana yang diberikan dalam Persamaan (4.3). (Rencana tersebut menggunakan notasi if–then–else untuk menangani cabang AND, tetapi ketika ada lebih dari dua cabang pada satu simpul, mungkin lebih baik menggunakan konstruksi kasus.)

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
    OR-SEARCH(problem.INITIAL-STATE, problem, [ ])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
    if problem.GOAL-TEST(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan ≠ failure then return [action | plan]
    return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
    for each si in states do
        plani ← OR-SEARCH(si, problem, path)
        if plani = failure then return failure
    return [if s1 then plan1 else if s2 then plan2 else . . . if sn−1 then plann−1 else plann]
```

Gambar 4.11 Algoritma untuk mencari grafik AND–OR yang dihasilkan oleh lingkungan nondeterministik. Algoritma ini mengembalikan rencana bersyarat yang mencapai status tujuan dalam semua keadaan. (Notasi $[x|\ell]$ mengacu pada daftar yang dibentuk dengan menambahkan objek x ke bagian depan daftar ℓ.)



Gambar 4.12 Bagian dari grafik pencarian untuk dunia vakum licin, di mana kami telah menunjukkan (beberapa) siklus secara eksplisit. Semua solusi untuk masalah ini adalah rencana siklus karena tidak ada cara untuk bergerak dengan andal.

Memodifikasi agen pemecahan masalah dasar yang ditunjukkan pada Gambar 3.1 untuk mengeksekusi solusi kontingen semacam ini mudah dilakukan. Seseorang juga dapat mempertimbangkan desain agen yang agak berbeda, di mana agen dapat bertindak sebelum menemukan rencana yang terjamin dan menangani beberapa kontingensi hanya saat muncul selama eksekusi. Jenis interleaving pencarian dan eksekusi ini juga berguna untuk masalah eksplorasi (lihat Bagian 4.5) dan untuk permainan (lihat Bab 5).

Gambar 4.11 memberikan algoritma rekursif yang mengutamakan kedalaman untuk pencarian grafik AND–OR. Salah satu aspek utama algoritma adalah cara menangani siklus, yang sering muncul dalam masalah nondeterministik (misalnya, jika suatu tindakan terkadang tidak memiliki efek atau jika efek yang tidak diinginkan dapat diperbaiki). Jika keadaan saat ini identik dengan keadaan pada jalur dari akar, maka ia kembali dengan kegagalan. Ini tidak berarti bahwa tidak ada solusi dari keadaan saat ini; ini hanya berarti bahwa jika ada solusi nonsiklis, solusi tersebut harus dapat dicapai dari inkarnasi sebelumnya dari keadaan saat ini, sehingga inkarnasi baru dapat dibuang.

Dengan pemeriksaan ini, kami memastikan bahwa algoritma berakhir di setiap ruang keadaan terbatas, karena setiap jalur harus mencapai tujuan, jalan buntu, atau keadaan berulang. Perhatikan bahwa algoritma tidak memeriksa apakah keadaan saat ini merupakan pengulangan keadaan pada beberapa jalur lain dari akar, yang penting untuk efisiensi.

Grafik AND–OR juga dapat dieksplorasi dengan metode breadth-first atau best-first. Konsep fungsi heuristik harus dimodifikasi untuk memperkirakan biaya solusi kontingen daripada urutan, tetapi gagasan tentang penerimaan berlaku dan ada analog dari algoritma A$*$ untuk menemukan solusi optimal. Petunjuk diberikan dalam catatan bibliografi di akhir bab.

Coba, Coba Lagi

Pertimbangkan dunia vakum licin, yang identik dengan dunia vakum biasa (tidak tak menentu) kecuali bahwa tindakan gerakan terkadang gagal, meninggalkan agen di lokasi yang sama. Misalnya, bergerak ke Kanan dalam keadaan 1 mengarah ke himpunan keadaan {1, 2}. Gambar 4.12 menunjukkan bagian dari grafik pencarian; jelas, tidak ada lagi solusi asiklik dari keadaan 1, dan PENCARIAN-GRAFIK-AND-OR- akan kembali dengan kegagalan. Namun, ada solusi siklik, yaitu terus mencoba ke Kanan hingga berhasil. Kita dapat mengekspresikan solusi ini dengan menambahkan label untuk menunjukkan beberapa bagian dari rencana dan menggunakan label itu nanti alih-alih mengulang rencana itu sendiri.

Jadi, solusi siklik kita adalah

$$[Suck, L1 : Kanan, jika\ Keadaan = 5\ maka\ L1\ jika\ tidak, Suck].$$

(Sintaks yang lebih baik untuk bagian perulangan dari rencana ini adalah "while $State = 5$ do $Right$.") Secara umum, rencana siklik dapat dianggap sebagai solusi asalkan setiap daun adalah keadaan tujuan dan bahwa daun dapat dicapai dari setiap titik dalam rencana. Realisasi utamanya adalah bahwa perulangan dalam ruang keadaan kembali ke keadaan L diterjemahkan menjadi perulangan dalam rencana kembali ke titik di mana subrencana untuk keadaan L dieksekusi.

Dengan definisi solusi siklik, agen yang mengeksekusi solusi tersebut pada akhirnya akan mencapai tujuan asalkan setiap hasil dari tindakan nondeterministik akhirnya terjadi. Apakah kondisi ini masuk akal? Itu tergantung pada alasan nondeterminisme. Jika tindakan melempar dadu, maka masuk akal untuk menganggap bahwa pada akhirnya angka enam akan dilempar. Jika tindakan yang dilakukan adalah memasukkan kunci kartu hotel ke dalam kunci pintu, tetapi tidak berhasil pada percobaan pertama, maka mungkin kunci tersebut akan berhasil pada akhirnya, atau mungkin salah memasukkan kunci (atau salah kamar!).

Setelah tujuh atau delapan kali mencoba, kebanyakan orang akan berasumsi bahwa masalahnya ada pada kunci dan akan kembali ke meja resepsionis untuk mendapatkan kunci baru. Salah satu cara untuk memahami keputusan ini adalah dengan mengatakan bahwa rumusan masalah awal (dapat diamati, tidak deterministik) ditinggalkan demi rumusan yang berbeda (dapat diamati sebagian, deterministik) di mana kegagalan dikaitkan dengan sifat kunci yang tidak dapat diamati.

4.4 PENCARIAN DENGAN PENGAMATAN SEBAGIAN

Sekarang kita beralih ke masalah pengamatan parsial, di mana persepsi agen tidak cukup untuk menentukan keadaan yang tepat. Seperti yang dicatat di awal bagian sebelumnya, jika agen berada dalam salah satu dari beberapa kemungkinan keadaan, maka suatu tindakan dapat mengarah pada salah satu dari beberapa kemungkinan hasil—bahkan jika lingkungannya deterministik.

Konsep utama yang diperlukan untuk memecahkan masalah yang dapat diamati sebagian adalah keadaan keyakinan, yang mewakili keyakinan agen saat ini tentang kemungkinan keadaan fisik yang mungkin dialaminya, mengingat urutan tindakan dan persepsi hingga saat itu. Kita mulai dengan skenario paling sederhana untuk mempelajari keadaan keyakinan, yaitu ketika agen tidak memiliki sensor sama sekali; kemudian kita menambahkan penginderaan parsial serta tindakan nondeterministik.

Pencarian Tanpa Pengamatan

Ketika persepsi agen tidak memberikan informasi sama sekali, kita memiliki apa yang disebut masalah tanpa sensor atau terkadang masalah konforman. Awalnya, orang mungkin berpikir agen tanpa sensor tidak memiliki harapan untuk menyelesaikan masalah jika tidak tahu keadaannya; pada kenyataannya, masalah tanpa sensor cukup sering dapat dipecahkan. Selain itu, agen tanpa sensor dapat sangat berguna, terutama karena tidak bergantung pada sensor yang bekerja dengan baik.

Dalam sistem manufaktur, misalnya, banyak metode cerdik telah dikembangkan untuk mengarahkan komponen dengan benar dari posisi awal yang tidak diketahui dengan menggunakan serangkaian tindakan tanpa penginderaan sama sekali. Biaya penginderaan yang tinggi adalah alasan lain untuk menghindarinya: misalnya, dokter sering meresepkan antibiotik spektrum luas daripada menggunakan rencana kontingensi untuk melakukan tes darah yang mahal, kemudian menunggu hasilnya kembali, dan kemudian meresepkan antibiotik yang lebih spesifik dan mungkin rawat inap karena infeksi telah berkembang terlalu jauh. Kita dapat membuat versi dunia vakum tanpa sensor.

Asumsikan bahwa agen mengetahui geografi dunianya, tetapi tidak mengetahui lokasinya atau distribusi kotorannya. Dalam kasus tersebut, status awalnya dapat berupa elemen apa pun dari himpunan $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Sekarang, pertimbangkan apa yang terjadi jika mencoba tindakan $Kanan$. Ini akan menyebabkannya berada di salah satu status $\{2, 4, 6, 8\}$—agen sekarang memiliki lebih banyak informasi! Lebih jauh lagi, urutan tindakan $[Kanan, Hisap]$ akan selalu berakhir di salah satu status $\{4, 8\}$. Akhirnya, urutan $[Kanan, Hisap, Kiri, Hisap]$ dijamin akan mencapai status tujuan 7 tidak peduli apa pun status awalnya. Kita katakan bahwa agen dapat memaksa dunia ke status 7.

Untuk memecahkan masalah tanpa sensor, kita mencari di ruang status keyakinan daripada status fisik. Perhatikan bahwa dalam ruang status keyakinan, masalahnya sepenuhnya dapat diamati karena agen selalu mengetahui status keyakinannya sendiri. Lebih jauh lagi, solusinya (jika ada) selalu berupa urutan tindakan. Hal ini karena, seperti dalam soal-soal biasa di Bab 3, persepsi yang diterima setelah setiap tindakan sepenuhnya dapat diprediksi—persepsi tersebut selalu kosong! Jadi tidak ada kemungkinan yang perlu direncanakan. Hal ini berlaku bahkan jika lingkungannya tidak deterministik.

Sangatlah bermanfaat untuk melihat bagaimana masalah pencarian status keyakinan dibangun. Misalkan masalah fisik dasar P didefinisikan oleh $ACTIONS_P, RESULT_P$, $GOAL-TEST_P$, dan $STEP-COST_P$. Maka kita dapat mendefinisikan masalah tanpa sensor yang sesuai sebagai berikut:

- **Status keyakinan**: Seluruh ruang status keyakinan berisi setiap kemungkinan set status fisik. Jika P memiliki N status, maka masalah tanpa sensor memiliki hingga 2^N status, meskipun banyak yang mungkin tidak dapat dicapai dari status awal.
- **Status awal**: Biasanya set semua status di P, meskipun dalam beberapa kasus agen akan memiliki lebih banyak pengetahuan daripada ini.
- **Tindakan**: Ini sedikit rumit. Misalkan agen berada dalam status keyakinan $b = \{s1, s2\}$, tetapi $ACTIONS_P\ (s_1) \neq ACTIONS_P\ (s_2)$; maka agen tidak yakin tindakan mana yang sah. Jika kita berasumsi bahwa tindakan ilegal tidak mempunyai dampak terhadap lingkungan, maka aman untuk mengambil gabungan semua tindakan dalam salah satu keadaan fisik dalam keadaan keyakinan saat ini b:

$$ACTION(b) = \bigcup_{s \in b} ACTION_P\ (s).$$

  Di sisi lain, jika tindakan ilegal mungkin merupakan akhir dunia, lebih aman untuk hanya mengizinkan persimpangan, yaitu, serangkaian tindakan yang legal di semua negara bagian. Untuk dunia vakum, setiap negara bagian memiliki tindakan hukum yang sama, sehingga kedua metode memberikan hasil yang sama.
- **Model transisi**: Agen tidak tahu negara bagian mana dalam negara bagian keyakinan yang merupakan yang benar; jadi sejauh yang diketahuinya, ia mungkin sampai ke salah satu negara bagian yang dihasilkan dari penerapan tindakan ke salah satu negara

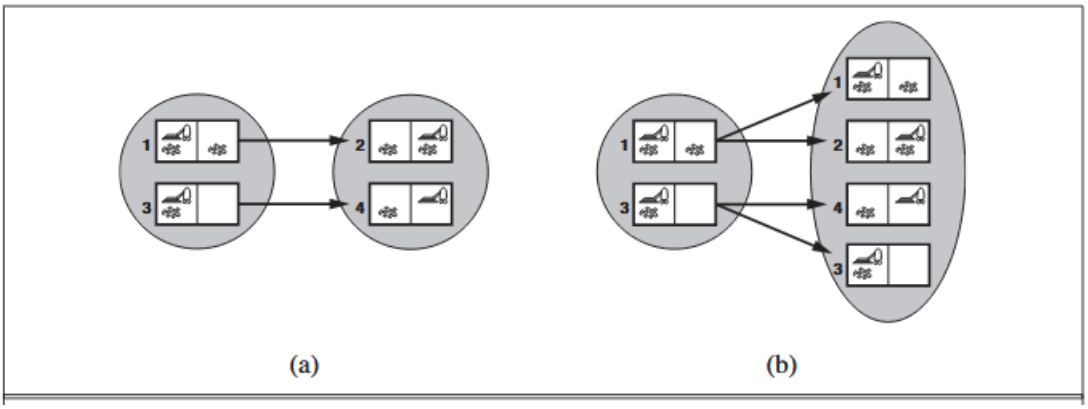
bagian fisik dalam negara bagian keyakinan. Untuk tindakan deterministik, serangkaian negara bagian yang mungkin dicapai adalah

$$b' = RESULT(b, a) = \{s' : s' = RESULT_P(s, a) \; and \; s \in b\}. \qquad (4.4)$$

Dengan tindakan deterministik, b' tidak pernah lebih besar dari b. Dengan nondeterminisme, kita memiliki

$$\begin{aligned} b' = RESULT(b, a) \quad &= \{s' : s' \in RESULTS_P(s, a) \; and \; s \in b\} \\ &= \bigcup_{s \in b} RESULTS_P(s, a), \end{aligned}$$

yang mungkin lebih besar dari b, seperti yang ditunjukkan pada Gambar 4.13. Proses menghasilkan status keyakinan baru setelah tindakan disebut langkah prediksi; notasi $b' = PREDICT_P(b, a)$ akan berguna.



Gambar 4.13 (a) Memprediksi keadaan keyakinan berikutnya untuk dunia vakum tanpa sensor dengan tindakan deterministik, Kanan. (b) Prediksi untuk keadaan keyakinan dan tindakan yang sama dalam versi licin dari dunia vakum tanpa sensor.

- **Uji tujuan**: Agen menginginkan rencana yang pasti berhasil, yang berarti bahwa status keyakinan memenuhi tujuan hanya jika semua status fisik di dalamnya memenuhi $GOAL - TEST_P$. Agen mungkin secara tidak sengaja mencapai tujuan lebih awal, tetapi tidak akan tahu bahwa ia telah melakukannya.
- **Biaya jalur**: Ini juga rumit. Jika tindakan yang sama dapat memiliki biaya yang berbeda di berbagai status, maka biaya untuk melakukan tindakan dalam status keyakinan tertentu dapat berupa salah satu dari beberapa nilai. (Ini menimbulkan kelas masalah baru, yang kami bahas dalam Latihan 4.9.) Untuk saat ini, kami berasumsi bahwa biaya tindakan sama di semua status dan karenanya dapat ditransfer langsung dari masalah fisik yang mendasarinya.

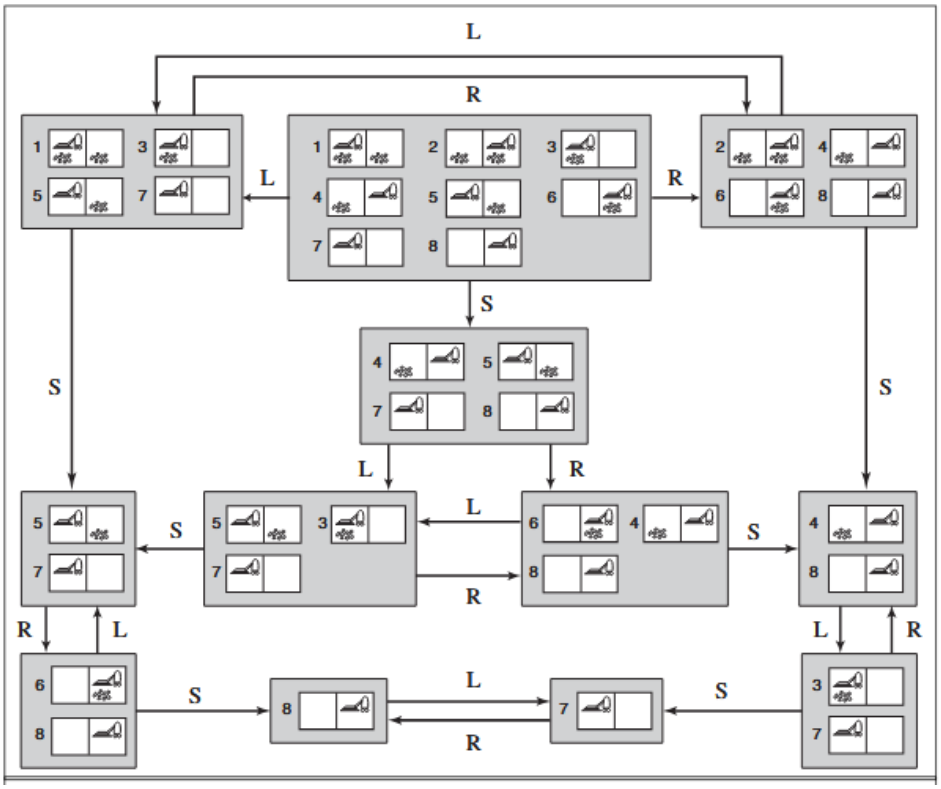
Gambar 4.14 menunjukkan ruang status keyakinan yang dapat dicapai untuk dunia vakum deterministik tanpa sensor. Hanya ada 12 status keyakinan yang dapat dicapai dari $2^8 = 256$

kemungkinan status keyakinan. Definisi sebelumnya memungkinkan konstruksi otomatis formulasi masalah status keyakinan dari definisi masalah fisik yang mendasarinya. Setelah ini selesai, kita dapat menerapkan salah satu algoritme pencarian di Bab 3.

Bahkan, kita dapat melakukan sedikit lebih dari itu. Dalam pencarian grafik "biasa", status yang baru dihasilkan diuji untuk melihat apakah status tersebut identik dengan status yang sudah ada. Ini juga berlaku untuk status keyakinan; misalnya, pada Gambar 4.14, urutan tindakan $[Hisap, Kiri, Hisap]$ yang dimulai pada status awal mencapai status keyakinan yang sama dengan $[Kanan, Kiri, Hisap]$, yaitu, $\{5, 7\}$. Sekarang, perhatikan status keyakinan yang dicapai oleh $[Kiri]$, yaitu, $\{1, 3, 5, 7\}$. Jelas, ini tidak identik dengan $\{5, 7\}$, tetapi ini adalah superset. Mudah untuk membuktikan (Latihan 4.8) bahwa jika urutan tindakan adalah solusi untuk status keyakinan b, maka itu juga merupakan solusi untuk setiap subset dari b.

Oleh karena itu, kita dapat membuang jalur yang mencapai $\{1, 3, 5, 7\}$ jika $\{5, 7\}$ telah dihasilkan. Sebaliknya, jika $\{1, 3, 5, 7\}$ telah dihasilkan dan ditemukan dapat dipecahkan, maka setiap subset, seperti $\{5, 7\}$, dijamin dapat dipecahkan. Tingkat pemangkasan ekstra ini dapat secara dramatis meningkatkan efisiensi pemecahan masalah tanpa sensor. Namun, bahkan dengan peningkatan ini, pemecahan masalah tanpa sensor seperti yang telah kami jelaskan jarang layak dalam praktik. Kesulitannya bukan pada luasnya ruang status keyakinan—meskipun secara eksponensial lebih besar daripada ruang status fisik yang mendasarinya; dalam kebanyakan kasus, faktor percabangan dan panjang solusi dalam ruang status keyakinan dan ruang status fisik tidak begitu berbeda. Kesulitan yang sebenarnya terletak pada ukuran setiap status keyakinan. Misalnya, status keyakinan awal untuk dunia vakum 10×10 berisi 100×2^{100} atau sekitar 10^{32} status fisik—terlalu banyak jika kita menggunakan representasi atomik, yang merupakan daftar status yang eksplisit.

Salah satu solusinya adalah merepresentasikan status keyakinan dengan deskripsi yang lebih ringkas. Dalam bahasa Inggris, kita dapat mengatakan agen mengetahui “Tidak Ada” dalam status awal; setelah bergerak ke Kiri, kita dapat mengatakan, “Tidak ada di kolom paling kanan,” dan seterusnya. Bab 7 menjelaskan cara melakukan ini dalam skema representasi formal. Pendekatan lain adalah menghindari algoritme pencarian standar, yang memperlakukan status keyakinan sebagai kotak hitam seperti status masalah lainnya. Sebaliknya, kita dapat melihat ke dalam status keyakinan dan mengembangkan algoritme pencarian status keyakinan tambahan yang membangun solusi satu status fisik pada satu waktu. Misalnya, dalam dunia vakum tanpa sensor, status keyakinan awal adalah $\{1, 2, 3, 4, 5, 6, 7, 8\}$, dan kita harus menemukan urutan tindakan yang berfungsi di semua 8 status tersebut.



Gambar 4.14 Bagian yang dapat dijangkau dari ruang status keyakinan untuk dunia vakum deterministik tanpa sensor. Setiap kotak yang diarsir sesuai dengan satu status keyakinan. Pada titik mana pun, agen berada dalam status keyakinan tertentu tetapi tidak tahu di status fisik mana ia berada. Status keyakinan awal (ketidaktahuan total) adalah kotak tengah atas. Tindakan direpresentasikan oleh tautan berlabel. Self-loop dihilangkan demi kejelasan.

Kita dapat melakukannya dengan terlebih dahulu menemukan solusi yang berfungsi untuk status 1; kemudian kita memeriksa apakah solusi tersebut berfungsi untuk status 2; jika tidak, kembali dan temukan solusi yang berbeda untuk status 1, dan seterusnya. Sama seperti pencarian AND–OR yang harus menemukan solusi untuk setiap cabang pada simpul AND, algoritme ini harus menemukan solusi untuk setiap status dalam status keyakinan; perbedaannya adalah bahwa pencarian AND–OR dapat menemukan solusi yang berbeda untuk setiap cabang, sedangkan pencarian status keyakinan inkremental harus menemukan satu solusi yang berfungsi untuk semua status.

Keuntungan utama dari pendekatan inkremental adalah pendekatan ini biasanya dapat mendeteksi kegagalan dengan cepat—ketika status keyakinan tidak dapat dipecahkan, biasanya sebagian kecil dari status keyakinan, yang terdiri dari beberapa status pertama yang diperiksa, juga tidak dapat dipecahkan. Dalam beberapa kasus, hal ini mengarah pada percepatan yang proporsional dengan ukuran status keyakinan, yang mungkin sama besarnya dengan ruang status fisik itu sendiri.

Bahkan algoritme solusi yang paling efisien pun tidak banyak berguna jika tidak ada solusi. Banyak hal yang tidak dapat dilakukan tanpa penginderaan. Misalnya, teka-teki 8 tanpa sensor tidak mungkin dilakukan. Di sisi lain, sedikit penginderaan dapat memberikan hasil yang besar. Misalnya, setiap contoh teka-teki 8 dapat dipecahkan jika hanya satu kotak yang

terlihat—solusinya melibatkan pemindahan setiap petak secara bergantian ke kotak yang terlihat dan kemudian melacak lokasinya.

Pencarian dengan Observasi

Untuk masalah umum yang dapat diamati sebagian, kita harus menentukan bagaimana lingkungan menghasilkan persepsi untuk agen. Misalnya, kita dapat mendefinisikan dunia vakum dengan penginderaan lokal sebagai dunia di mana agen memiliki sensor posisi dan sensor kotoran lokal tetapi tidak memiliki sensor yang mampu mendeteksi kotoran di kotak lain.

Spesifikasi masalah formal mencakup fungsi $PERCEPT(s)$ yang mengembalikan persepsi yang diterima dalam keadaan tertentu. (Jika penginderaan bersifat nondeterministik, maka kita menggunakan fungsi $PERCEPTS$ yang mengembalikan serangkaian persepsi yang mungkin.) Misalnya, dalam dunia vakum dengan penginderaan lokal, $PERCEPT$ dalam keadaan 1 adalah $[A, Kotor\ ]$. Masalah yang dapat diamati sepenuhnya adalah kasus khusus di mana $PERCEPT(s) = s$ untuk setiap keadaan s, sedangkan masalah tanpa sensor adalah kasus khusus di mana $PERCEPT(s) = null$.

Jika observasi bersifat parsial, biasanya beberapa keadaan dapat menghasilkan persepsi tertentu. Misalnya, persepsi $[A, Dirty\ ]$ dihasilkan oleh keadaan 3 dan juga oleh keadaan 1. Oleh karena itu, jika ini adalah persepsi awal, keadaan keyakinan awal untuk dunia vakum dengan penginderaan lokal adalah $\{1, 3\}$. TINDAKAN, BIAYA LANGKAH, dan UJI TUJUAN dibangun dari masalah fisik yang mendasarinya seperti halnya untuk masalah tanpa sensor, tetapi model transisinya sedikit lebih rumit. Kita dapat menganggap transisi dari satu keadaan keyakinan ke keadaan berikutnya untuk tindakan tertentu terjadi dalam tiga tahap, seperti yang ditunjukkan pada Gambar 4.15:

- Tahap prediksi sama seperti untuk masalah tanpa sensor: jika tindakan a berada dalam keadaan keyakinan b, keadaan keyakinan yang diprediksi adalah $\hat{b} = PREDIKSI(b, a)$.
- Tahap prediksi observasi menentukan serangkaian persepsi o yang dapat diamati dalam keadaan keyakinan yang diprediksi:

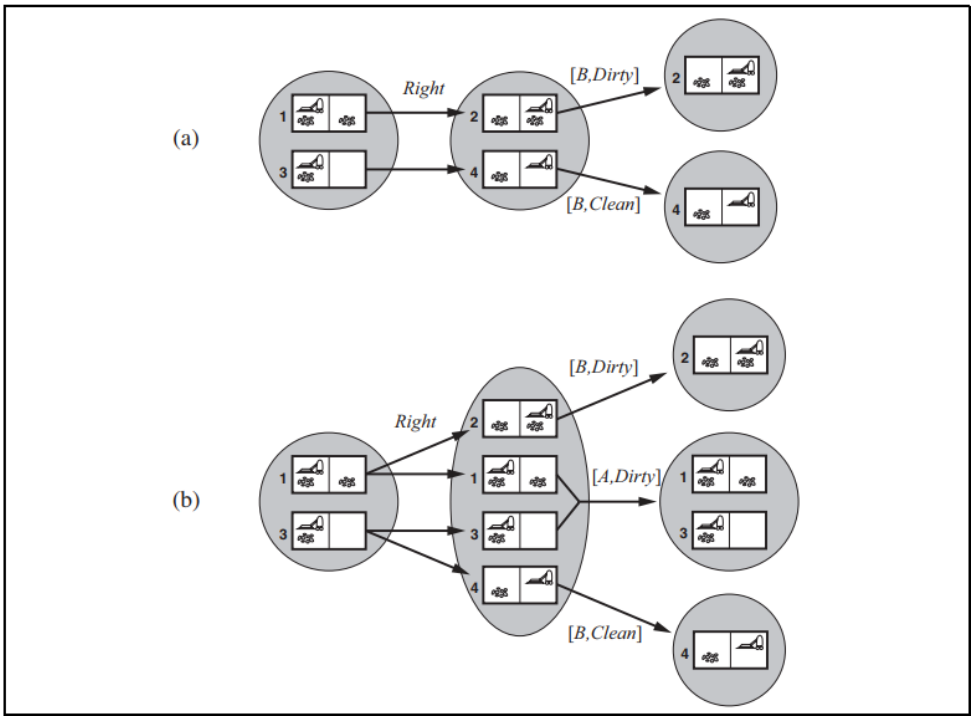
$$POSSIBLE - PERCEPTS(\hat{b}) = \{o: o = PERCEPT(s) and\ s \in \hat{b}\}.$$

- Tahap pembaruan menentukan, untuk setiap kemungkinan persepsi, status keyakinan yang akan dihasilkan dari persepsi tersebut. Status keyakinan baru b_o hanyalah serangkaian status dalam $\hat{b}$ yang dapat menghasilkan persepsi tersebut:

$$b_o = UPDATE(\hat{b}, o) = \{s: o = PERCEPT(s)\ and\ s \in \hat{b}\}.$$

Perhatikan bahwa setiap status keyakinan yang diperbarui b_o tidak boleh lebih besar dari status keyakinan yang diprediksi $\hat{b}$; pengamatan hanya dapat membantu mengurangi ketidakpastian dibandingkan dengan kasus tanpa sensor. Selain itu, untuk

penginderaan deterministik, status keyakinan untuk berbagai kemungkinan persepsi akan terpisah, membentuk partisi dari status keyakinan yang diprediksi sebelumnya.

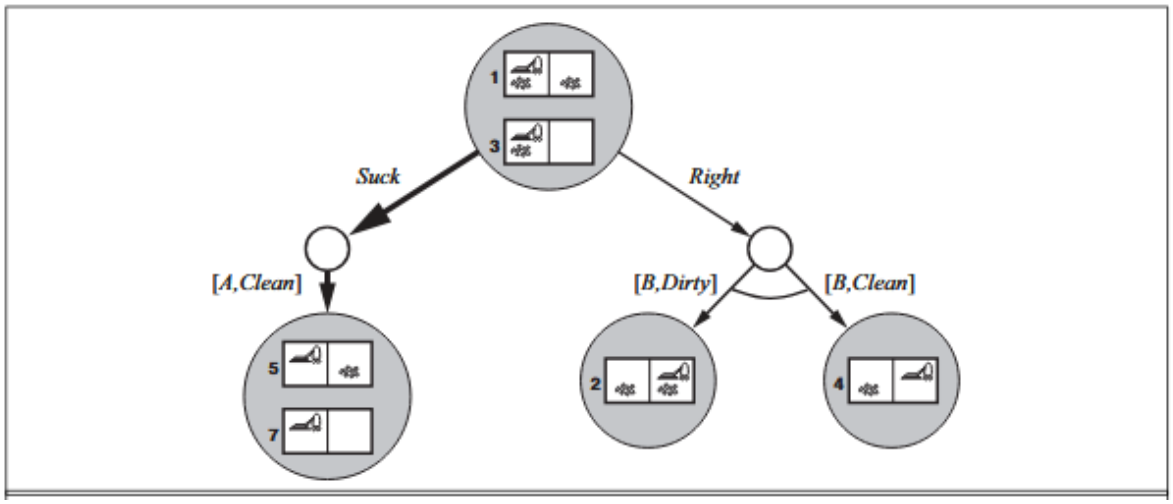


Gambar 4.15 Dua contoh transisi di dunia vakum dengan penginderaan lokal. (a) Di dunia deterministik, Kanan diterapkan dalam status keyakinan awal, menghasilkan status keyakinan baru dengan dua kemungkinan status fisik; untuk status tersebut, persepsi yang mungkin adalah $[B, Kotor]$ dan $[B, Bersih]$, yang mengarah ke dua status keyakinan, yang masing-masing merupakan satu kesatuan. (b) Di dunia licin, Kanan diterapkan dalam status keyakinan awal, menghasilkan status keyakinan baru dengan empat status fisik; untuk status tersebut, persepsi yang mungkin adalah $[A, Kotor]$, $[B, Kotor]$, dan $[B, Bersih]$, yang mengarah ke tiga status keyakinan seperti yang ditunjukkan.

Dengan menggabungkan ketiga tahap ini, kita memperoleh kemungkinan keadaan keyakinan yang dihasilkan dari tindakan tertentu dan kemungkinan persepsi selanjutnya:

$$RESULTS(b,a) = \{b_o : b_o = UPDATE(PREDICT(b,a),o) \; and \; o \in POSSIBLE - PERCEPTS(PREDICT(b,a))\}. \quad (4.5)$$

Sekali lagi, ketidakpastian dalam masalah yang dapat diamati sebagian berasal dari ketidakmampuan untuk memprediksi secara tepat persepsi mana yang akan diterima setelah bertindak; ketidakpastian yang mendasari dalam lingkungan fisik dapat berkontribusi pada ketidakmampuan ini dengan memperluas status keyakinan pada tahap prediksi, yang mengarah ke lebih banyak persepsi pada tahap pengamatan.



Gambar 4.16 Tingkat pertama dari pohon pencarian AND–OR untuk masalah di dunia vakum penginderaan lokal; Suck adalah langkah pertama dari solusinya.

Memecahkan Masalah yang dapat Diamati Sebagian

Bagian sebelumnya menunjukkan cara memperoleh fungsi HASIL untuk masalah status keyakinan yang tidak deterministik dari masalah fisik yang mendasari dan fungsi PERSEPSI. Dengan formulasi seperti itu, algoritma pencarian AND–OR pada Gambar 4.11 dapat diterapkan secara langsung untuk memperoleh solusi. Gambar 4.16 menunjukkan bagian dari pohon pencarian untuk dunia vakum penginderaan lokal, dengan asumsi persepsi awal $[A, Dirty\ ]$. Solusinya adalah rencana bersyarat

$$[Suck, Right, if\ Bstate = \{6\}\ then\ Suck\ else[\ ]].$$

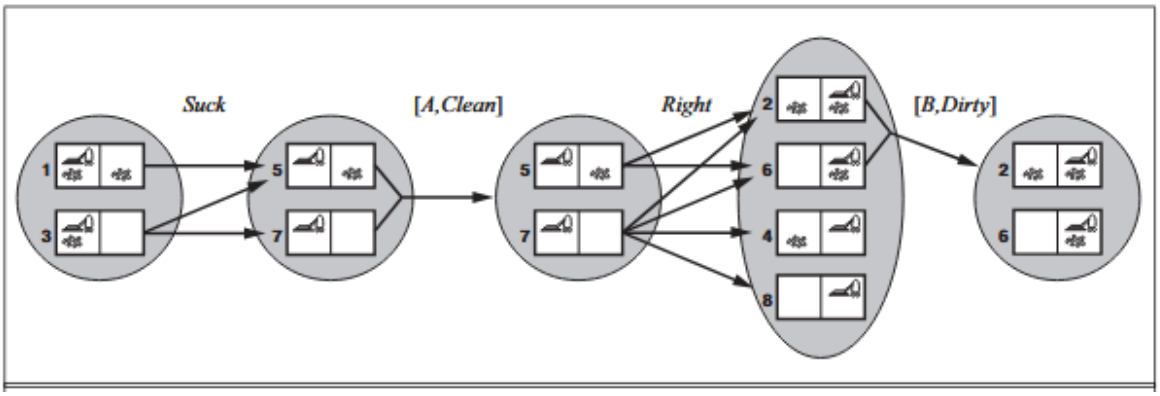
Perhatikan bahwa, karena kami menyediakan masalah status keyakinan pada algoritme penelusuran AND–OR, algoritme tersebut mengembalikan rencana bersyarat yang menguji status keyakinan alih-alih status aktual. Ini sebagaimana mestinya: dalam lingkungan yang dapat diamati sebagian, agen tidak akan dapat menjalankan solusi yang memerlukan pengujian status aktual.

Seperti halnya algoritme penelusuran standar yang diterapkan pada masalah tanpa sensor, algoritme penelusuran AND–OR memperlakukan status keyakinan sebagai kotak hitam, sama seperti status lainnya. Kita dapat meningkatkannya dengan memeriksa status keyakinan yang dihasilkan sebelumnya yang merupakan bagian atau superset dari status saat ini, sama seperti untuk masalah tanpa sensor. Kita juga dapat memperoleh algoritme penelusuran inkremental, yang analog dengan yang dijelaskan untuk masalah tanpa sensor, yang memberikan percepatan substansial atas pendekatan kotak hitam.

Agen untuk lingkungan yang dapat diamati sebagian

Desain agen pemecahan masalah untuk lingkungan yang dapat diamati sebagian cukup mirip dengan agen pemecahan masalah sederhana pada Gambar 3.1: agen merumuskan masalah, memanggil algoritma pencarian (seperti AND-OR-GRAPH-SEARCH) untuk menyelesaikannya, dan mengeksekusi solusinya. Ada dua perbedaan utama. Pertama, solusi

untuk masalah akan menjadi rencana bersyarat, bukan urutan; jika langkah pertama adalah ekspresi if–then–else, agen perlu menguji kondisi di bagian if dan mengeksekusi bagian then atau bagian else sesuai dengan itu.



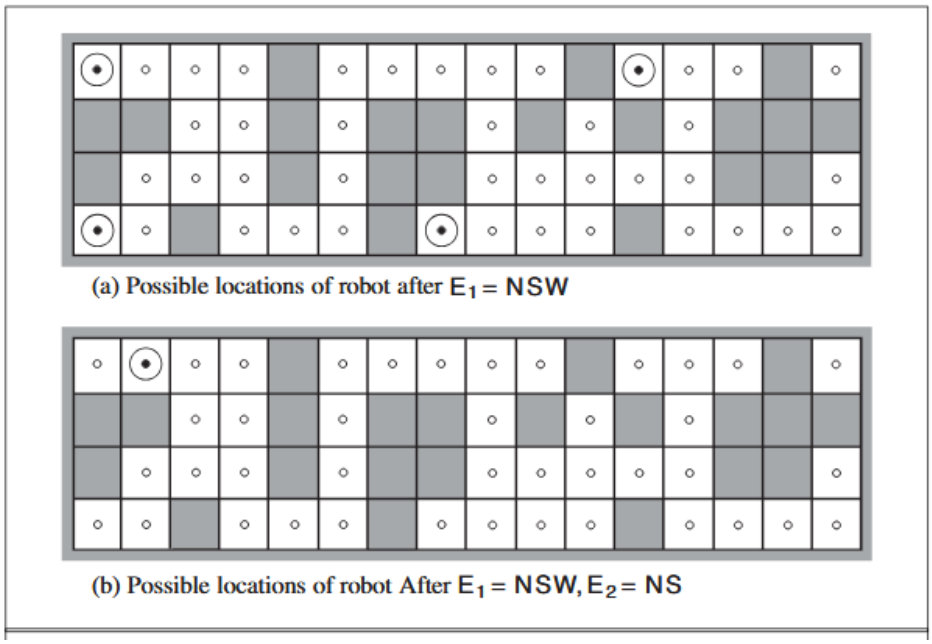
Gambar 4.17 Dua siklus prediksi–pembaruan pemeliharaan keadaan keyakinan di dunia vakum taman kanak-kanak dengan penginderaan lokal.

Kedua, agen perlu mempertahankan status keyakinannya saat melakukan tindakan dan menerima persepsi. Proses ini menyerupai proses prediksi–observasi–pembaruan dalam Persamaan (4.5) tetapi sebenarnya lebih sederhana karena persepsi diberikan oleh lingkungan, bukan dihitung oleh agen. Dengan status keyakinan awal b, tindakan a, dan persepsi o, status keyakinan baru adalah:

$$b' = UPDATE(PREDICT(b, a), o). \qquad (4.6)$$

Gambar 4.17 menunjukkan status keyakinan yang dipertahankan di dunia vakum taman kanak-kanak dengan penginderaan lokal, di mana kotak mana pun dapat menjadi kotor kapan saja kecuali agen tersebut secara aktif membersihkannya pada saat itu.

Di lingkungan yang dapat diamati sebagian—yang mencakup sebagian besar lingkungan dunia nyata—mempertahankan status keyakinan seseorang merupakan fungsi inti dari sistem cerdas apa pun. Fungsi ini dikenal dengan berbagai nama, termasuk pemantauan, penyaringan, dan estimasi status. Persamaan (4.6) disebut sebagai estimator status rekursif karena menghitung status keyakinan baru dari yang sebelumnya daripada dengan memeriksa seluruh urutan persepsi. Jika agen tidak ingin "tertinggal", perhitungan harus dilakukan secepat persepsi masuk. Saat lingkungan menjadi lebih kompleks, perhitungan pembaruan yang tepat menjadi tidak layak dan agen harus menghitung status keyakinan perkiraan, mungkin dengan berfokus pada implikasi persepsi untuk aspek lingkungan yang menjadi perhatian saat ini. Sebagian besar pekerjaan pada masalah ini telah dilakukan untuk lingkungan stokastik, keadaan kontinu dengan alat teori probabilitas, seperti yang dijelaskan dalam Bab 15. Di sini kami akan menunjukkan contoh dalam lingkungan diskrit dengan sensor deterministik dan tindakan nondeterministik.



Gambar 4.18 Kemungkinan posisi robot, ⊙, (a) setelah satu pengamatan $E_1 = NSW$ dan (b) setelah pengamatan kedua $E_2 = NS$. Bila sensor tidak bersuara dan model transisi akurat, tidak ada kemungkinan lokasi lain bagi robot yang konsisten dengan urutan dua pengamatan ini.

Contoh tersebut menyangkut robot dengan tugas lokalisasi: mencari tahu di mana ia berada, dengan peta dunia dan serangkaian persepsi dan tindakan. Robot kami ditempatkan di lingkungan seperti labirin pada Gambar 4.18. Robot dilengkapi dengan empat sensor sonar yang memberi tahu apakah ada rintangan—dinding luar atau kotak hitam pada gambar—di masing-masing dari empat arah kompas. Kami berasumsi bahwa sensor memberikan data yang benar-benar tepat, dan bahwa robot memiliki peta lingkungan yang benar. Namun sayangnya sistem navigasi robot rusak, jadi ketika menjalankan tindakan Bergerak, ia bergerak secara acak ke salah satu kotak yang berdekatan. Tugas robot adalah menentukan lokasinya saat ini.

Misalkan robot baru saja dinyalakan, jadi ia tidak tahu di mana ia berada. Dengan demikian, status keyakinan awalnya b terdiri dari himpunan semua lokasi. Robot menerima persepsi NSW, yang berarti ada rintangan di utara, barat, dan selatan, dan melakukan pembaruan menggunakan persamaan $b_o = UPDATE(b)$, menghasilkan 4 lokasi yang ditunjukkan pada Gambar 4.18(a). Anda dapat memeriksa labirin untuk melihat bahwa hanya empat lokasi tersebut yang menghasilkan persepsi NWS.

$$UPDATE(PREDICT(UPDATE(b, NSW), Move), NS).$$

Dengan tindakan nondetermnistik, langkah $PREDICT$ mengembangkan status keyakinan, tetapi langkah $UPDATE$ mengecilkannya kembali—selama persepsi memberikan beberapa informasi identifikasi yang berguna. Terkadang persepsi tidak banyak membantu untuk pelokalan: Jika ada satu atau lebih koridor timur-barat yang panjang, maka robot dapat

menerima rangkaian persepsi NS yang panjang, tetapi tidak pernah tahu di mana di koridor tersebut ia berada.

4.5 AGEN PENCARIAN ONLINE DAN LINGKUNGAN YANG TIDAK DIKETAHUI

Sejauh ini kami telah berkonsentrasi pada agen yang menggunakan algoritme pencarian offline. Mereka menghitung solusi lengkap sebelum melangkah ke dunia nyata dan kemudian mengeksekusi solusi tersebut. Sebaliknya, agen pencarian online menyelingi perhitungan dan tindakan: pertama-tama ia mengambil tindakan, kemudian mengamati lingkungan dan menghitung tindakan berikutnya.

Pencarian online merupakan ide yang bagus dalam domain dinamis atau semidinamis—domain tempat terdapat penalti untuk duduk diam dan melakukan komputasi terlalu lama. Pencarian online juga membantu dalam domain nondeterministik karena memungkinkan agen untuk memfokuskan upaya komputasinya pada kemungkinan yang benar-benar muncul daripada kemungkinan yang mungkin terjadi tetapi mungkin tidak akan terjadi. Tentu saja, ada tradeoff: semakin banyak agen merencanakan ke depan, semakin jarang ia akan menemukan dirinya dalam kesulitan tanpa dayung.

Pencarian online merupakan ide yang diperlukan untuk lingkungan yang tidak diketahui, tempat agen tidak mengetahui keadaan apa yang ada atau apa yang dilakukan tindakannya. Dalam kondisi ketidaktahuan ini, agen menghadapi masalah eksplorasi dan harus menggunakan tindakannya sebagai eksperimen untuk belajar cukup banyak agar pertimbangan menjadi berharga.

Contoh kanonik pencarian daring adalah robot yang ditempatkan di gedung baru dan harus menjelajahinya untuk membangun peta yang dapat digunakan untuk berpindah dari titik A ke titik B. Metode untuk melarikan diri dari labirin—pengetahuan yang dibutuhkan untuk calon pahlawan zaman dahulu—juga merupakan contoh algoritma pencarian daring. Namun, eksplorasi spasial bukanlah satu-satunya bentuk eksplorasi.

Pertimbangkan bayi yang baru lahir: ia memiliki banyak kemungkinan tindakan tetapi tidak mengetahui hasil dari tindakan tersebut, dan ia hanya mengalami beberapa kemungkinan keadaan yang dapat dicapainya. Penemuan bayi secara bertahap tentang cara kerja dunia, sebagian, merupakan proses pencarian daring.

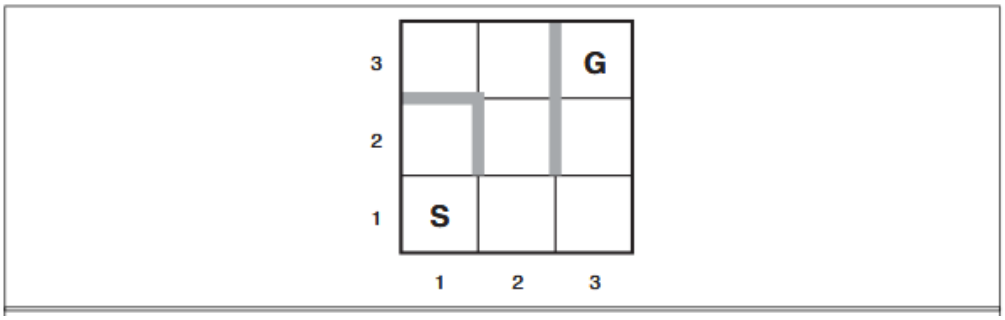
Masalah Pencarian Daring

Masalah pencarian daring harus dipecahkan oleh agen yang melakukan tindakan, bukan dengan komputasi murni. Kami mengasumsikan lingkungan yang deterministik dan dapat diamati sepenuhnya (Bab 17 melonggarkan asumsi ini), tetapi kami menetapkan bahwa agen hanya mengetahui hal berikut:

- $TINDAKAN(s)$, yang mengembalikan daftar tindakan yang diizinkan dalam status s;
- Fungsi biaya langkah $c(s, a, s')$—perhatikan bahwa ini tidak dapat digunakan hingga agen mengetahui bahwa s' adalah hasilnya; dan
- $UJI - TUJUAN(s)$.

Perhatikan khususnya bahwa agen tidak dapat menentukan HASIL, a kecuali dengan benar-benar berada di s dan melakukan a. Misalnya, dalam masalah labirin yang ditunjukkan pada

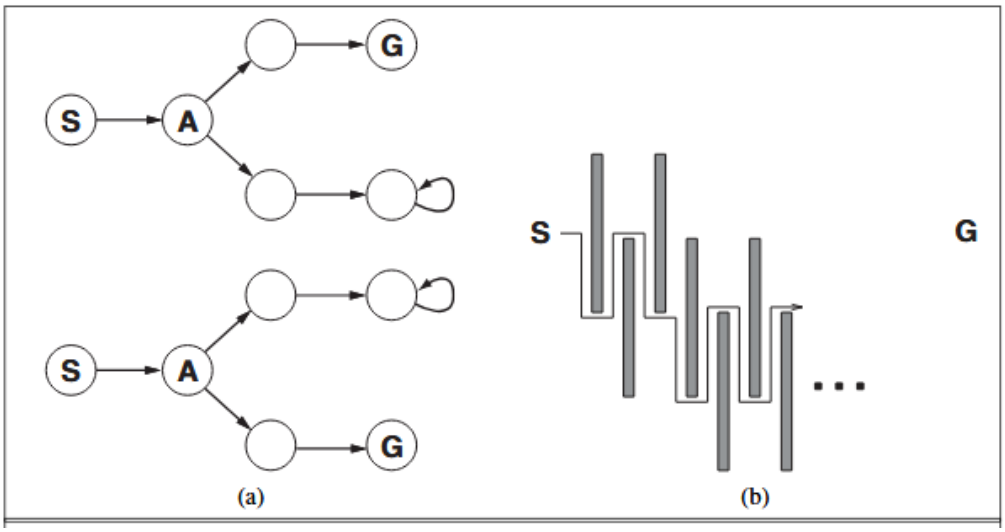
Gambar 4.19, agen tidak tahu bahwa $Naik$ dari (1,1) mengarah ke (1,2); atau, setelah melakukan itu, ia tidak tahu bahwa $Turun$ akan membawanya kembali ke (1,1). Tingkat ketidaktahuan ini dapat dikurangi dalam beberapa aplikasi—misalnya, penjelajah robot mungkin tahu cara kerja tindakan pergerakannya dan hanya tidak mengetahui lokasi rintangan.



Gambar 4.19 Masalah labirin sederhana. Agen mulai di S dan harus mencapai G tetapi tidak mengetahui apa pun tentang lingkungannya.

Akhirnya, agen mungkin memiliki akses ke fungsi heuristik yang dapat diterima $h(s)$ yang memperkirakan jarak dari keadaan saat ini ke keadaan tujuan. Misalnya, pada Gambar 4.19, agen mungkin mengetahui lokasi tujuan dan dapat menggunakan heuristik jarak Manhattan. Biasanya, tujuan agen adalah untuk mencapai keadaan tujuan sambil meminimalkan biaya. (Tujuan lain yang mungkin adalah hanya untuk menjelajahi seluruh lingkungan.)

Biaya adalah total biaya jalur dari jalur yang sebenarnya dilalui agen. Merupakan hal yang umum untuk membandingkan biaya ini dengan biaya jalur dari jalur yang akan diikuti agen jika mengetahui ruang pencarian sebelumnya—yaitu, jalur terpendek yang sebenarnya (atau eksplorasi lengkap terpendek). Dalam bahasa algoritma daring, ini disebut rasio kompetitif; kami ingin rasio ini sekecil mungkin.



Gambar 4.20 (a) Dua ruang keadaan yang mungkin menuntun agen pencarian daring ke jalan buntu. Setiap agen akan gagal di setidaknya satu dari ruang-ruang ini. (b) Lingkungan dua dimensi yang dapat menyebabkan agen pencarian daring mengikuti rute yang tidak efisien

untuk mencapai tujuan. Apa pun pilihan yang diambil agen, musuh akan memblokir rute tersebut dengan dinding tipis dan panjang lainnya, sehingga jalur yang diikuti jauh lebih panjang daripada jalur terbaik yang memungkinkan.

Meskipun ini terdengar seperti permintaan yang masuk akal, mudah untuk melihat bahwa rasio kompetitif terbaik yang dapat dicapai adalah tak terbatas dalam beberapa kasus. Misalnya, jika beberapa tindakan tidak dapat diubah—yaitu, tindakan tersebut mengarah ke keadaan yang tidak ada tindakan yang mengarah kembali ke keadaan sebelumnya—pencarian daring mungkin secara tidak sengaja mencapai keadaan buntu yang tidak ada keadaan tujuan yang dapat dicapai. Mungkin istilah "secara tidak sengaja" tidak meyakinkan—bagaimanapun juga, mungkin ada algoritme yang kebetulan tidak mengambil jalan buntu saat menjelajah. Klaim kami, lebih tepatnya, adalah bahwa tidak ada algoritme yang dapat menghindari jalan buntu di semua ruang keadaan.

Perhatikan dua ruang keadaan buntu pada Gambar 4.20(a). Bagi algoritme pencarian daring yang telah mengunjungi keadaan S dan A, kedua ruang keadaan tersebut tampak identik, sehingga harus membuat keputusan yang sama di keduanya. Oleh karena itu, ia akan gagal di salah satunya. Ini adalah contoh argumen lawan—kita dapat membayangkan lawan membangun ruang keadaan sementara agen menjelajahinya dan menempatkan tujuan dan jalan buntu di mana pun yang dipilihnya.

Jalan buntu merupakan kesulitan nyata bagi eksplorasi robot—tangga, jalan landai, tebing, jalan satu arah, dan semua jenis medan alam menghadirkan peluang untuk tindakan yang tidak dapat diubah. Untuk membuat kemajuan, kita cukup berasumsi bahwa ruang keadaan dapat dieksplorasi dengan aman—yaitu, beberapa keadaan tujuan dapat dicapai dari setiap keadaan yang dapat dicapai. Ruang keadaan dengan tindakan yang dapat dibalik, seperti labirin dan 8 teka-teki, dapat dilihat sebagai grafik tak berarah dan jelas dapat dieksplorasi dengan aman.

Bahkan dalam lingkungan yang dapat dieksplorasi dengan aman, tidak ada rasio kompetitif terbatas yang dapat dijamin jika ada jalur dengan biaya tak terbatas. Hal ini mudah ditunjukkan dalam lingkungan dengan tindakan yang tidak dapat diubah, tetapi pada kenyataannya hal ini tetap berlaku untuk kasus yang dapat dibalik juga, seperti yang ditunjukkan Gambar 4.20(b). Karena alasan ini, kinerja algoritme pencarian daring biasanya dideskripsikan dalam hal ukuran seluruh ruang keadaan, bukan hanya kedalaman tujuan yang paling dangkal.

Agen Pencarian Daring

Setelah setiap tindakan, agen daring menerima persepsi yang memberitahukan status apa yang telah dicapainya; dari informasi ini, agen dapat melengkapi peta lingkungannya. Peta saat ini digunakan untuk memutuskan ke mana harus pergi selanjutnya. Perpaduan antara perencanaan dan tindakan ini berarti bahwa algoritme pencarian daring sangat berbeda dari algoritme pencarian luring yang telah kita lihat sebelumnya. Misalnya, algoritme luring seperti A$*$ dapat memperluas simpul di satu bagian ruang dan kemudian segera memperluas simpul di bagian lain ruang, karena perluasan simpul melibatkan simulasi daripada tindakan nyata.

Di sisi lain, algoritme daring dapat menemukan penerus hanya untuk simpul yang ditempatinya secara fisik. Untuk menghindari perjalanan melintasi pohon untuk memperluas simpul berikutnya, tampaknya lebih baik memperluas simpul dalam urutan lokal. Pencarian kedalaman-pertama memiliki properti ini karena (kecuali saat menelusuri kembali) simpul berikutnya yang diperluas adalah anak dari simpul sebelumnya yang diperluas.

Agen pencarian kedalaman-pertama daring ditunjukkan pada Gambar 4.21. Agen ini menyimpan petanya dalam tabel, $RESULT[s, a]$, yang mencatat status yang dihasilkan dari pelaksanaan tindakan a dalam status s. Setiap kali tindakan dari status saat ini belum dieksplorasi, agen mencoba tindakan tersebut. Kesulitan muncul ketika agen telah mencoba semua tindakan dalam suatu status.

Dalam penelusuran mendalam offline, status tersebut hanya dihapus dari antrean; dalam penelusuran online, agen harus menelusuri kembali secara fisik. Dalam penelusuran mendalam, ini berarti kembali ke status tempat agen terakhir kali memasuki status saat ini. Untuk mencapainya, algoritme menyimpan tabel yang mencantumkan, untuk setiap status, status pendahulu yang belum ditelusuri kembali oleh agen. Jika agen telah kehabisan status yang dapat ditelusuri kembali, maka penelusurannya selesai.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  if s is not null then
      result[s, a] ← s'
      add s to the front of unbacktracked[s']
  if untried[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Gambar 4.21 Agen pencarian daring yang menggunakan eksplorasi mendalam. Agen ini hanya berlaku di ruang status di mana setiap tindakan dapat "dibatalkan" oleh tindakan lain.

Kami menyarankan agar pembaca menelusuri kemajuan ONLINE-DFS-AGENT saat diterapkan pada labirin yang diberikan dalam Gambar 4.19. Cukup mudah untuk melihat bahwa agen, dalam kasus terburuk, akan berakhir dengan melintasi setiap tautan dalam ruang status tepat dua kali. Untuk eksplorasi, ini optimal; untuk menemukan tujuan, di sisi lain, rasio kompetitif agen bisa sangat buruk jika ia melakukan perjalanan panjang ketika ada tujuan tepat di sebelah keadaan awal. Varian daring dari pendalaman iteratif memecahkan masalah ini;

untuk lingkungan yang merupakan pohon seragam, rasio kompetitif agen tersebut adalah konstanta kecil. Karena metodenya untuk melacak balik, ONLINE-DFS-AGENT hanya bekerja di ruang keadaan tempat tindakan dapat dibalik. Ada algoritme yang sedikit lebih rumit yang bekerja di ruang keadaan umum, tetapi tidak ada algoritme tersebut yang memiliki rasio kompetitif terbatas.

Pencarian Lokal Daring

Seperti pencarian mendalam, pencarian mendaki bukit memiliki sifat lokalitas dalam perluasan simpulnya. Faktanya, karena hanya menyimpan satu status terkini dalam memori, pencarian mendaki bukit sudah menjadi algoritma pencarian daring! Sayangnya, pencarian ini tidak terlalu berguna dalam bentuknya yang paling sederhana karena membiarkan agen berada pada posisi maksimum lokal tanpa tujuan. Selain itu, restart acak tidak dapat digunakan, karena agen tidak dapat memindahkan dirinya sendiri ke status baru. Alih-alih restart acak, seseorang dapat mempertimbangkan untuk menggunakan random walk untuk menjelajahi lingkungan. Random walk hanya memilih secara acak salah satu tindakan yang tersedia dari status terkini; preferensi dapat diberikan pada tindakan yang belum dicoba. Mudah untuk membuktikan bahwa random walk pada akhirnya akan menemukan tujuan atau menyelesaikan penjelajahannya, asalkan ruangnya terbatas.

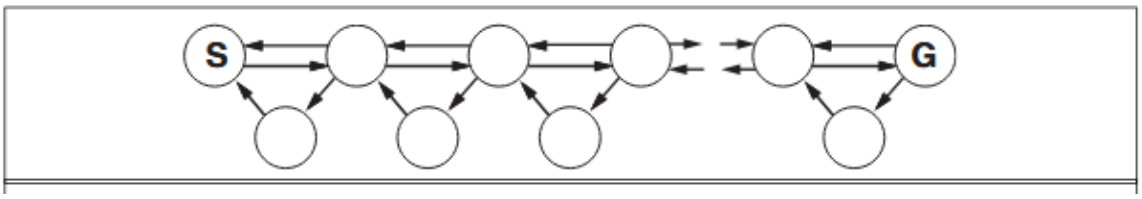
Di sisi lain, prosesnya bisa sangat lambat. Gambar 4.22 menunjukkan lingkungan tempat jalan acak akan mengambil langkah-langkah eksponensial untuk menemukan tujuan karena, pada setiap langkah, kemajuan mundur dua kali lebih mungkin terjadi daripada kemajuan maju. Tentu saja, contoh ini dibuat-buat, tetapi ada banyak ruang keadaan dunia nyata yang topologinya menyebabkan "jebakan" semacam ini untuk jalan acak.

Menambah pendakian bukit dengan memori daripada keacakan ternyata menjadi pendekatan yang lebih efektif. Ide dasarnya adalah menyimpan "estimasi terbaik saat ini" $H(s)$ dari biaya untuk mencapai tujuan dari setiap keadaan yang telah dikunjungi. $H(s)$ awalnya hanya berupa estimasi heuristik $h(s)$ dan diperbarui saat agen memperoleh pengalaman dalam ruang keadaan. Gambar 4.23 menunjukkan contoh sederhana dalam ruang keadaan satu dimensi. Dalam (a), agen tampaknya terjebak dalam minimum lokal yang datar pada keadaan yang diarsir.

Daripada tetap di tempatnya, agen harus mengikuti apa yang tampaknya menjadi jalur terbaik menuju tujuan mengingat estimasi biaya saat ini untuk tetangganya. Perkiraan biaya untuk mencapai tujuan melalui tetangga s' adalah biaya untuk mencapai s' ditambah perkiraan biaya untuk mencapai tujuan dari sana—yaitu, $c(s, a, s') + H(s')$.

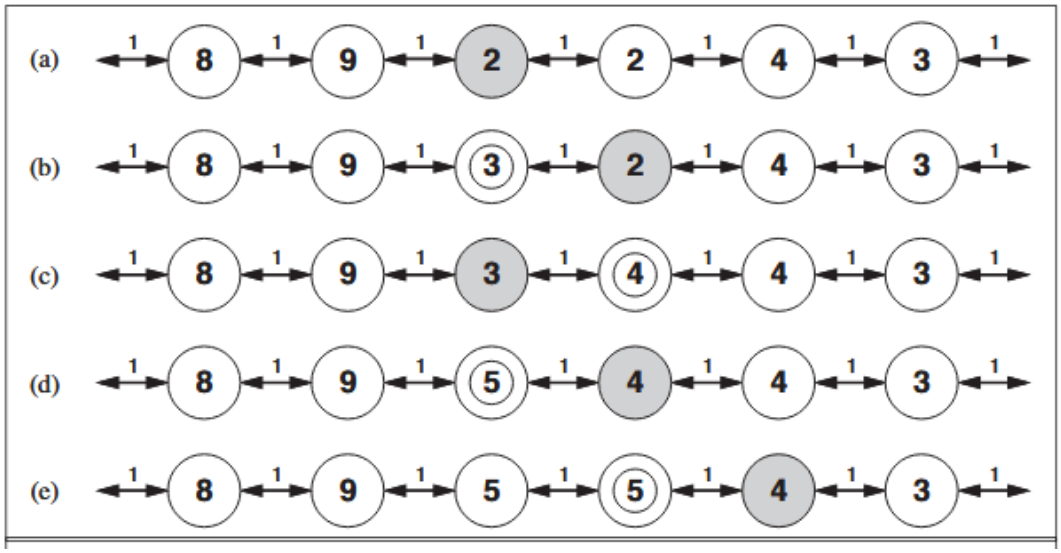
Dalam contoh tersebut, ada dua tindakan, dengan perkiraan biaya 1 + 9 dan 1 + 2, jadi tampaknya yang terbaik adalah bergerak ke kanan. Sekarang, jelas bahwa perkiraan biaya 2 untuk keadaan yang diarsir terlalu optimis. Karena langkah terbaik menghabiskan biaya 1 dan mengarah ke keadaan yang setidaknya berjarak 2 langkah dari tujuan, keadaan yang diarsir harus setidaknya berjarak 3 langkah dari tujuan, jadi H-nya harus diperbarui sebagaimana mestinya, seperti yang ditunjukkan pada Gambar 4.23(b). Melanjutkan proses ini, agen akan bergerak maju mundur dua kali lagi, memperbarui H setiap kali dan "meratakan" minimum lokal hingga keluar ke kanan.

Agen yang menerapkan skema ini, yang disebut pembelajaran A∗ waktu nyata (LRTA∗), ditunjukkan pada Gambar 4.24. Seperti ONLINE-DFS-AGENT, ia membangun peta lingkungan dalam tabel hasil. Ia memperbarui estimasi biaya untuk keadaan yang baru saja ditinggalkannya, lalu memilih langkah yang "tampaknya terbaik" menurut estimasi biayanya saat ini. Satu detail penting adalah bahwa tindakan yang belum dicoba dalam keadaan s selalu diasumsikan mengarah langsung ke tujuan dengan biaya sekecil mungkin, yaitu $h(s)$. Optimisme dalam ketidakpastian ini mendorong agen untuk mengeksplorasi jalur baru yang mungkin menjanjikan.



Gambar 4.22 Suatu lingkungan di mana suatu perjalanan acak akan memerlukan banyak langkah secara eksponensial untuk menemukan tujuan.

Agen LRTA∗ dijamin akan menemukan tujuan di lingkungan yang terbatas dan dapat dieksplorasi dengan aman. Namun, tidak seperti A∗, agen ini tidak lengkap untuk ruang keadaan tak terbatas—ada beberapa kasus di mana agen ini dapat tersesat tak terbatas. Agen ini dapat mengeksplorasi lingkungan dengan n keadaan dalam langkah $O(n^2)$ dalam kasus terburuk, tetapi sering kali jauh lebih baik. Agen LRTA∗ hanyalah salah satu dari sekumpulan besar agen daring yang dapat didefinisikan dengan menentukan aturan pemilihan tindakan dan aturan pembaruan dengan berbagai cara.



Gambar 4.23 Lima iterasi LRTA∗ pada ruang status satu dimensi. Setiap status diberi label dengan $H(s)$, estimasi biaya saat ini untuk mencapai tujuan, dan setiap tautan diberi label dengan biaya langkahnya. Status yang diarsir menandai lokasi agen, dan estimasi biaya yang diperbarui pada setiap iterasi dilingkari.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Gambar 4.24 LRTA*-AGENT memilih tindakan berdasarkan nilai status tetangga, yang diperbarui saat agen bergerak di ruang status.

Pembelajaran dalam Pencarian Daring

Ketidaktahuan awal agen pencarian daring menyediakan beberapa peluang untuk pembelajaran. Pertama, agen mempelajari "peta" lingkungan—lebih tepatnya, hasil dari setiap tindakan di setiap status—hanya dengan mencatat setiap pengalaman mereka. (Perhatikan bahwa asumsi lingkungan deterministik berarti bahwa satu pengalaman cukup untuk setiap tindakan.)

Kedua, agen pencarian lokal memperoleh estimasi biaya setiap status yang lebih akurat dengan menggunakan aturan pembaruan lokal, seperti dalam LRTA$*$. Dalam Bab 21, kami menunjukkan bahwa pembaruan ini akhirnya konvergen ke nilai eksak untuk setiap status, asalkan agen menjelajahi ruang status dengan cara yang benar. Setelah nilai eksak diketahui, keputusan optimal dapat diambil hanya dengan beralih ke penerus berbiaya terendah—dengan kata lain, pendakian bukit murni merupakan strategi yang optimal.

Jika Anda mengikuti saran kami untuk melacak perilaku ONLINE-DFS-AGENT di lingkungan Gambar 4.19, Anda akan melihat bahwa agen tersebut tidak terlalu cerdas. Misalnya, setelah melihat bahwa aksi $Naik$ bergerak dari (1,1) ke (1,2), agen masih tidak tahu bahwa aksi $Turun$ kembali ke (1,1) atau bahwa aksi $Naik$ juga bergerak dari (2,1) ke (2,2), dari (2,2) ke (2,3), dan seterusnya. Secara umum, kita ingin agen mempelajari bahwa $Naik$ meningkatkan koordinat-y kecuali jika ada dinding yang menghalangi, bahwa $Turun$ menguranginya, dan seterusnya. Agar ini terjadi, kita memerlukan dua hal. Pertama, kita memerlukan representasi formal dan dapat dimanipulasi secara eksplisit untuk aturan umum semacam ini; sejauh ini, kita telah menyembunyikan informasi di dalam kotak hitam yang disebut fungsi RESULT. Bagian III dikhususkan untuk masalah ini. Kedua, kita memerlukan

algoritme yang dapat membangun aturan umum yang sesuai dari pengamatan khusus yang dilakukan oleh agen.

4.6 RINGKASAN

Bab ini telah mengkaji algoritme pencarian untuk masalah di luar kasus "klasik" dalam menemukan jalur terpendek menuju tujuan dalam lingkungan diskrit yang dapat diamati dan deterministik.

- Metode pencarian lokal seperti hill climbing beroperasi pada formulasi status lengkap, dengan hanya menyimpan sejumlah kecil simpul dalam memori. Beberapa algoritme stokastik telah dikembangkan, termasuk simulasi annealing, yang menghasilkan solusi optimal saat diberikan jadwal pendinginan yang sesuai.
- Banyak metode pencarian lokal juga berlaku untuk masalah dalam ruang kontinu. Masalah pemrograman linier dan pengoptimalan konveks mematuhi batasan tertentu pada bentuk ruang status dan sifat fungsi objektif, dan menerima algoritme waktu polinomial yang sering kali sangat efisien dalam praktik.
- Algoritme genetika adalah pencarian hill-climbing stokastik di mana populasi status yang besar dipertahankan. Status baru dihasilkan oleh mutasi dan oleh crossover, yang menggabungkan pasangan status dari populasi.
- Dalam lingkungan nondeterministik, agen dapat menerapkan pencarian AND–OR untuk menghasilkan rencana kontingen yang mencapai tujuan terlepas dari hasil mana yang terjadi selama eksekusi.
- Ketika lingkungan dapat diamati sebagian, status keyakinan mewakili serangkaian status yang mungkin dialami agen.
- Algoritme pencarian standar dapat diterapkan langsung ke ruang status keyakinan untuk memecahkan masalah tanpa sensor, dan pencarian AND–OR status keyakinan dapat memecahkan masalah umum yang dapat diamati sebagian. Algoritme inkremental yang membangun solusi status demi status dalam status keyakinan sering kali lebih efisien.
- Masalah eksplorasi muncul ketika agen tidak memiliki gambaran tentang status dan tindakan lingkungannya. Untuk lingkungan yang dapat dieksplorasi dengan aman, agen pencarian daring dapat membangun peta dan menemukan tujuan jika ada. Memperbarui estimasi heuristik dari pengalaman menyediakan metode yang efektif untuk keluar dari minimum lokal.

BAB 5
PENCARIAN YANG BERSAING

5.1 PERMAINAN

Bab 2 memperkenalkan lingkungan multiagen, di mana setiap agen perlu mempertimbangkan tindakan agen lain dan bagaimana tindakan tersebut memengaruhi kesejahteraannya sendiri. Ketidakpastian agen lain ini dapat menimbulkan kemungkinan dalam proses pemecahan masalah agen, seperti yang dibahas dalam Bab 4. Dalam bab ini kita membahas lingkungan kompetitif, di mana tujuan agen saling bertentangan, sehingga menimbulkan masalah pencarian yang bertentangan—yang sering dikenal sebagai permainan.

Teori permainan matematika, cabang ilmu ekonomi, memandang setiap lingkungan multiagen sebagai sebuah permainan, asalkan dampak dari masing-masing agen terhadap agen lainnya adalah "signifikan," terlepas dari apakah agen tersebut kooperatif atau kompetitif. Dalam AI, permainan yang paling umum adalah jenis yang agak terspesialisasi—apa yang oleh para ahli teori permainan disebut permainan deterministik, bergiliran, dua pemain, zero-sum dengan informasi sempurna (seperti catur).

Dalam terminologi kami, ini berarti lingkungan deterministik, yang dapat diamati sepenuhnya, di mana dua agen bertindak secara bergantian dan di mana nilai utilitas di akhir permainan selalu sama dan berlawanan. Misalnya, jika satu pemain memenangkan permainan catur, pemain lainnya pasti kalah. Pertentangan antara fungsi utilitas agen inilah yang membuat situasi menjadi bermusuhan.

Permainan telah melibatkan kemampuan intelektual manusia—terkadang hingga tingkat yang mengkhawatirkan—selama peradaban ada. Bagi para peneliti AI, sifat abstrak permainan menjadikannya subjek yang menarik untuk dipelajari. Keadaan permainan mudah direpresentasikan, dan agen biasanya dibatasi pada sejumlah kecil tindakan yang hasilnya ditentukan oleh aturan yang tepat. Permainan fisik, seperti kroket dan hoki es, memiliki deskripsi yang jauh lebih rumit, rentang tindakan yang jauh lebih luas, dan aturan yang agak tidak tepat yang mendefinisikan legalitas tindakan. Dengan pengecualian sepak bola robot, permainan fisik ini belum banyak menarik minat komunitas AI.

Permainan, tidak seperti kebanyakan masalah mainan yang dipelajari dalam Bab 3, menarik karena terlalu sulit dipecahkan. Misalnya, catur memiliki faktor percabangan rata-rata sekitar 35, dan permainan sering kali mencapai 50 langkah oleh setiap pemain, sehingga pohon pencarian memiliki sekitar 35^{100} atau 10^{154} simpul (meskipun grafik pencarian "hanya" memiliki sekitar 10^{40} simpul yang berbeda). Oleh karena itu, permainan, seperti dunia nyata, memerlukan kemampuan untuk membuat beberapa keputusan bahkan ketika menghitung keputusan optimal tidak layak dilakukan. Permainan juga sangat merugikan inefisiensi. Sementara implementasi pencarian A$*$ yang setengahnya efisien akan membutuhkan waktu dua kali lebih lama untuk berjalan hingga selesai, program catur yang setengahnya efisien dalam menggunakan waktu yang tersedia mungkin akan gagal total, jika hal-hal lain sama. Oleh

karena itu, penelitian tentang permainan telah melahirkan sejumlah ide menarik tentang cara memanfaatkan waktu sebaik mungkin.

Kita mulai dengan definisi langkah optimal dan algoritma untuk menemukannya. Kemudian kita melihat teknik untuk memilih langkah yang baik ketika waktu terbatas. Pemangkasan memungkinkan kita untuk mengabaikan bagian-bagian dari pohon pencarian yang tidak membuat perbedaan pada pilihan akhir, dan fungsi evaluasi heuristik memungkinkan kita untuk memperkirakan utilitas sebenarnya dari suatu keadaan tanpa melakukan pencarian yang lengkap.

Bagian 5.5 membahas permainan seperti backgammon yang mencakup unsur peluang; kami juga membahas bridge, yang mencakup unsur-unsur informasi yang tidak sempurna karena tidak semua kartu terlihat oleh setiap pemain. Akhirnya, kami melihat bagaimana program permainan canggih menghadapi perlawanan manusia dan arah untuk pengembangan di masa mendatang.

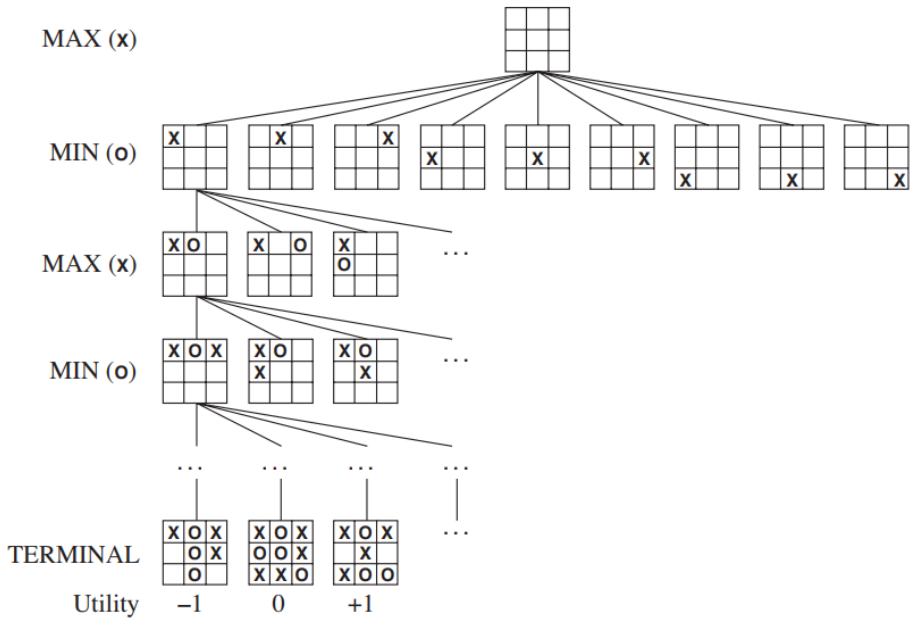
Pertama-tama kami mempertimbangkan permainan dengan dua pemain, yang kami sebut MAX dan MIN karena alasan yang akan segera menjadi jelas. MAX bergerak terlebih dahulu, dan kemudian mereka bergiliran bergerak hingga permainan berakhir. Di akhir permainan, poin diberikan kepada pemain yang menang dan penalti diberikan kepada yang kalah. Suatu permainan dapat didefinisikan secara formal sebagai semacam masalah pencarian dengan unsur-unsur berikut:

- S_0: Keadaan awal, yang menentukan bagaimana permainan disiapkan di awal.
- $PLAYER(s)$: Menentukan pemain mana yang memiliki langkah dalam suatu keadaan.
- ACTIONS(s): Mengembalikan rangkaian langkah yang sah dalam suatu keadaan.
- $RESULT(s, a)$: Model transisi, yang menentukan hasil dari suatu langkah.
- $TERMINAL - TEST(s)$: Tes terminal, yang bernilai benar ketika permainan berakhir dan salah jika tidak. Keadaan di mana permainan telah berakhir disebut keadaan terminal.
- $UTILITY(s, p)$: Fungsi utilitas (juga disebut fungsi objektif atau fungsi pembayaran), menentukan nilai numerik akhir untuk permainan yang berakhir dalam keadaan terminal s untuk pemain p. Dalam catur, hasilnya adalah menang, kalah, atau seri, dengan nilai $+1, 0$, atau $\frac{1}{2}$. Beberapa permainan memiliki berbagai kemungkinan hasil yang lebih luas; pembayaran dalam backgammon berkisar dari 0 hingga $+192$. Permainan zero-sum (secara membingungkan) didefinisikan sebagai permainan di mana total pembayaran untuk semua pemain sama untuk setiap contoh permainan. Catur adalah zero-sum karena setiap permainan memiliki pembayaran $0 + 1, 1 + 0$ atau $\frac{1}{2} + \frac{1}{2}$. "Jumlah konstan" akan menjadi istilah yang lebih baik, tetapi zero-sum adalah tradisional dan masuk akal jika Anda membayangkan setiap pemain dikenakan biaya masuk sebesar $\frac{1}{2}$.

Keadaan awal, fungsi ACTIONS, dan fungsi RESULT mendefinisikan pohon permainan untuk permainan tersebut—pohon di mana simpulnya adalah keadaan permainan dan tepinya adalah gerakan. Gambar 5.1 menunjukkan bagian dari pohon permainan untuk tic-tac-toe (nol

dan silang). Dari keadaan awal, MAX memiliki sembilan gerakan yang mungkin. Permainan berganti-ganti antara MAX yang menempatkan X dan MIN yang menempatkan O hingga kita mencapai simpul daun yang sesuai dengan keadaan terminal sehingga satu pemain memiliki tiga dalam satu baris atau semua kotak terisi. Angka pada setiap simpul daun menunjukkan nilai utilitas keadaan terminal dari sudut pandang MAX; nilai tinggi diasumsikan baik untuk MAX dan buruk untuk MIN (begitulah asal nama pemain).

Untuk tic-tac-toe, pohon permainan relatif kecil—kurang dari $9! = 362.880$ simpul terminal. Namun, untuk catur, ada lebih dari 10^{40} simpul, jadi pohon permainan sebaiknya dianggap sebagai konstruksi teoritis yang tidak dapat kita wujudkan di dunia nyata. Namun, terlepas dari ukuran pohon permainan, tugas MAX adalah mencari langkah yang baik. Kami menggunakan istilah pohon pencarian untuk pohon yang ditumpangkan pada pohon permainan penuh, dan memeriksa cukup banyak simpul untuk memungkinkan pemain menentukan langkah apa yang harus dilakukan.

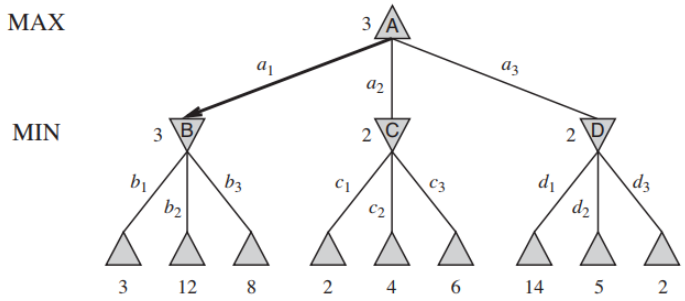


Gambar 5.1 Pohon permainan (sebagian) untuk permainan tic-tac-toe. Node teratas adalah status awal, dan MAX bergerak terlebih dahulu, menempatkan X di kotak kosong. Kami menunjukkan bagian dari pohon, memberikan gerakan bergantian oleh MIN (O) dan MAX (X), hingga akhirnya mencapai status terminal, yang dapat diberi utilitas sesuai dengan aturan permainan.

5.2 KEPUTUSAN OPTIMAL DALAM PERMAINAN

Dalam masalah pencarian normal, solusi optimal akan menjadi serangkaian tindakan yang mengarah ke status tujuan—status akhir yang merupakan kemenangan. Dalam pencarian yang bersifat adversarial, MIN memiliki sesuatu untuk dikatakan tentangnya. Oleh karena itu, MAX harus menemukan **strategi** kontingen, yang menentukan langkah MAX dalam status awal, lalu langkah MAX dalam status yang dihasilkan dari setiap kemungkinan respons oleh

MIN, lalu langkah MAX dalam status yang dihasilkan dari setiap kemungkinan respons oleh MIN terhadap langkah tersebut, dan seterusnya.



Gambar 5.2 Pohon permainan dua lapis. Node Δ adalah "simpul MAX," yang merupakan giliran MAX untuk bergerak, dan node ∇ adalah "simpul MIN." Node terminal menunjukkan nilai utilitas untuk MAX; node lainnya diberi label dengan nilai minimumnya. Langkah terbaik MAX di akar adalah a_1, karena mengarah ke keadaan dengan nilai minimum tertinggi, dan balasan terbaik MIN adalah b_1, karena mengarah ke keadaan dengan nilai minimum terendah.

Ini persis sama dengan algoritma pencarian AND–OR (Gambar 4.11) dengan MAX memainkan peran OR dan MIN setara dengan AND. Secara kasar, strategi optimal menghasilkan hasil yang setidaknya sama baiknya dengan strategi lainnya saat seseorang bermain dengan lawan yang tidak pernah salah. Kami mulai dengan menunjukkan cara menemukan strategi optimal ini.

Bahkan permainan sederhana seperti tic-tac-toe terlalu rumit bagi kita untuk menggambar seluruh pohon permainan dalam satu halaman, jadi kita akan beralih ke permainan sepele pada Gambar 5.2. Kemungkinan langkah untuk MAX pada simpul akar diberi label a_1, a_2 dan a_3. Kemungkinan balasan untuk a_1 untuk MIN adalah b_1, b_2, b_3., dan seterusnya. Permainan khusus ini berakhir setelah satu langkah masing-masing oleh MAX dan MIN. (Dalam istilah permainan, kita katakan bahwa pohon ini memiliki satu langkah dalam, yang terdiri dari dua setengah langkah, yang masing-masing disebut **ply**.) Utilitas status terminal dalam permainan ini berkisar dari 2 hingga 14.

Dengan pohon permainan, strategi optimal dapat ditentukan dari nilai minimum setiap simpul, yang kita tulis sebagai MINIMAX(n). Nilai minimum simpul adalah utilitas (untuk MAX) untuk berada dalam status yang sesuai, dengan asumsi bahwa kedua pemain bermain secara optimal dari sana hingga akhir permainan. Jelas, nilai minimum status terminal hanyalah utilitasnya. Lebih jauh lagi, jika diberi pilihan, MAX lebih suka bergerak ke keadaan nilai maksimum, sedangkan MIN lebih suka keadaan nilai minimum. Jadi, kita memiliki yang berikut:

$$MINIMAX(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL} - \text{TEST}(s) \\ max_{a \in Actions\,(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ min_{a \in Actions\,(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Mari kita terapkan definisi ini pada pohon permainan pada Gambar 5.2. Node terminal pada level paling bawah mendapatkan nilai utilitasnya dari fungsi UTILITY permainan. Node MIN pertama, berlabel B, memiliki tiga status penerus dengan nilai 3, 12, dan 8, sehingga nilai minimumnya adalah 3. Demikian pula, dua node MIN lainnya memiliki nilai minimum 2. Node akar adalah node MAX; status penerusnya memiliki nilai minimum 3, 2, dan 2; sehingga memiliki nilai minimum 3. Kita juga dapat mengidentifikasi **keputusan minimum** pada akar: tindakan a_1 adalah pilihan optimal untuk MAX karena mengarah ke status dengan nilai minimum tertinggi. Definisi permainan optimal untuk MAX ini mengasumsikan bahwa MIN juga bermain secara optimal—ia memaksimalkan *kasus terburuk* untuk MAX. Bagaimana jika MIN tidak bermain secara optimal? Maka mudah untuk menunjukkan bahwa MAX akan bermain lebih baik. Strategi lain terhadap lawan yang suboptimal mungkin lebih baik daripada strategi minimax, tetapi strategi ini tentu saja lebih buruk terhadap lawan yang optimal.

Algoritma minimax

Algoritma minimax (Gambar 5.3) menghitung keputusan minimax dari status saat ini. Algoritma ini menggunakan perhitungan rekursif sederhana dari nilai minimax setiap status penerus, yang secara langsung menerapkan persamaan yang menentukan. Rekursi berlanjut hingga ke daun pohon, dan kemudian nilai minimax **dicadangkan** melalui pohon saat rekursi berakhir. Misalnya, pada Gambar 5.2, algoritma pertama-tama melakukan rekursi hingga ke tiga simpul kiri bawah dan menggunakan fungsi UTILITY pada simpul-simpul tersebut untuk menemukan bahwa nilainya masing-masing adalah 3, 12, dan 8. Kemudian, algoritma ini mengambil nilai minimum dari nilai-nilai ini, 3, dan mengembalikannya sebagai nilai cadangan dari simpul B. Proses serupa menghasilkan nilai cadangan 2 untuk C dan 2 untuk D. Akhirnya, kita mengambil nilai maksimum 3, 2, dan 2 untuk mendapatkan nilai cadangan 3 untuk simpul akar.

Algoritma minimax melakukan eksplorasi mendalam yang lengkap dari pohon permainan. Jika kedalaman maksimum pohon adalah m dan terdapat b gerakan yang sah di setiap titik, maka kompleksitas waktu dari algoritma minimax adalah $O(b^m)$. Kompleksitas ruang adalah $O(bm)$ untuk algoritma yang menghasilkan semua tindakan sekaligus, atau $O(m)$ untuk algoritma yang menghasilkan tindakan satu per satu. Untuk permainan nyata, tentu saja, biaya waktu sama sekali tidak praktis, tetapi algoritma ini berfungsi sebagai dasar untuk analisis matematis permainan dan untuk algoritma yang lebih praktis.

Keputusan optimal dalam permainan multipemain

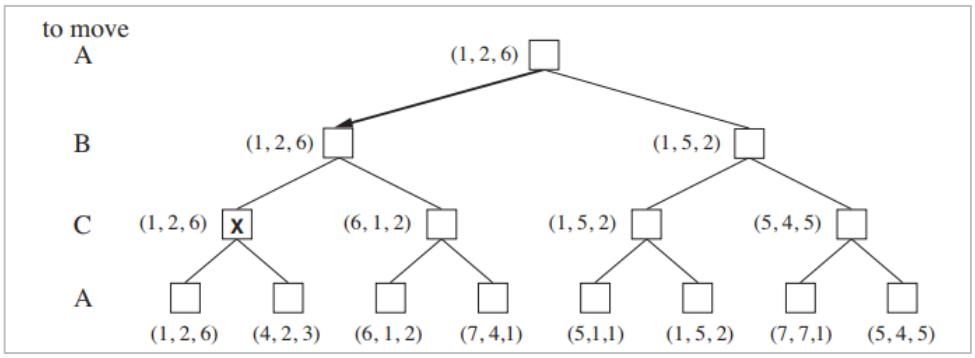
Banyak permainan populer yang memungkinkan lebih dari dua pemain. Mari kita telaah cara memperluas ide minimax ke permainan multipemain. Ini mudah dari sudut pandang teknis, tetapi menimbulkan beberapa masalah konseptual baru yang menarik. Pertama, kita perlu mengganti nilai tunggal untuk setiap simpul dengan vektor nilai. Misalnya,

dalam permainan tiga pemain dengan pemain A, B, dan C, sebuah vektor (v_A, v_B, v_C.) dikaitkan dengan setiap simpul. Untuk keadaan terminal, vektor ini memberikan utilitas keadaan dari sudut pandang setiap pemain. (Dalam permainan dua pemain dengan jumlah nol, vektor dua elemen dapat disederhanakan menjadi satu nilai karena nilainya selalu berlawanan.) Cara paling sederhana untuk mengimplementasikan hal ini adalah dengan mengembalikan vektor utilitas menggunakan fungsi UTILITY.

Sekarang kita harus mempertimbangkan status nonterminal. Pertimbangkan simpul bertanda X di pohon permainan yang ditunjukkan pada Gambar 5.4. Dalam status tersebut, pemain C memilih apa yang harus dilakukan. Kedua pilihan mengarah ke status terminal dengan vektor utilitas $\langle v_A = 1, v_B = 2,\ v_C = 6\rangle$ dan $\langle v_A = 4, v_B = 2,\ v_C = 3\rangle$. Karena 6 lebih besar dari 3, C harus memilih langkah pertama. Ini berarti bahwa jika status X tercapai, permainan berikutnya akan mengarah ke status terminal dengan utilitas $\langle v_A = 1, v_B = 2, v_C = 6\rangle$. Oleh karena itu, nilai cadangan X adalah vektor ini.

```
function MINIMAX-DECISION(state) returns an action
    return arg max_{a∈ACTIONS_s} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

Gambar 5.3 Algoritma untuk menghitung keputusan minimax. Algoritma ini mengembalikan tindakan yang sesuai dengan langkah terbaik yang mungkin, yaitu langkah yang mengarah pada hasil dengan utilitas terbaik, dengan asumsi bahwa lawan bermain untuk meminimalkan utilitas. Fungsi $\mathrm{MAX} - \mathrm{VALUE}$ dan $\mathrm{MIN} - \mathrm{VALUE}$ menelusuri seluruh pohon permainan, hingga ke daun, untuk menentukan nilai cadangan dari suatu keadaan. Notasi $argmax_{a \in S} f(a)$ menghitung elemen a dari himpunan S yang memiliki nilai maksimum $f(a)$



Gambar 5.4 Tiga lapisan pertama dari pohon permainan dengan tiga pemain (A, B, C). Setiap simpul diberi label dengan nilai dari sudut pandang setiap pemain. Langkah terbaik ditandai di akarnya.

Nilai cadangan dari simpul n selalu merupakan vektor utilitas dari status penerus dengan nilai tertinggi untuk pemain yang memilih di n. Siapa pun yang memainkan gim multipemain, seperti Diplomasi, dengan cepat menyadari bahwa lebih banyak hal yang terjadi daripada dalam gim dua pemain. Gim multipemain biasanya melibatkan **aliansi**, baik formal maupun informal, di antara para pemain. Aliansi dibuat dan dihancurkan saat permainan berlangsung. Bagaimana kita memahami perilaku seperti itu? Apakah aliansi merupakan konsekuensi alami dari strategi optimal untuk setiap pemain dalam gim multipemain? Ternyata memang bisa.

Misalnya, misalkan A dan B berada di posisi lemah dan C berada di posisi kuat. Maka sering kali lebih baik bagi A dan B untuk menyerang C daripada saling menyerang, agar C tidak menghancurkan mereka masing-masing secara individual. Dengan cara ini, kolaborasi muncul dari perilaku yang sepenuhnya egois. Tentu saja, begitu C melemah di bawah serangan gabungan, aliansi kehilangan nilainya, dan A atau B dapat melanggar perjanjian. Dalam beberapa kasus, aliansi eksplisit hanya membuat konkret apa yang akan terjadi. Dalam kasus lain, stigma sosial melekat pada pemutusan aliansi, sehingga pemain harus menyeimbangkan keuntungan langsung dari pemutusan aliansi dengan kerugian jangka panjang karena dianggap tidak dapat dipercaya.

Jika permainannya tidak zero-sum, maka kolaborasi juga dapat terjadi hanya dengan dua pemain. Misalkan, misalnya, ada keadaan terminal dengan utilitas $\langle v_A = 1000, v_B = 1000 \rangle$ dan bahwa 1000 adalah utilitas tertinggi yang mungkin untuk setiap pemain. Maka strategi optimalnya adalah bagi kedua pemain untuk melakukan segala yang mungkin untuk mencapai keadaan ini—yaitu, para pemain akan secara otomatis bekerja sama untuk mencapai tujuan yang diinginkan bersama.

5.3 PEMANGKASAN ALPHA–BETA

Masalah dengan pencarian minimax adalah jumlah status permainan yang harus diperiksa bersifat eksponensial dalam kedalaman pohon. Sayangnya, kita tidak dapat menghilangkan eksponen, tetapi ternyata kita dapat memotongnya menjadi dua secara efektif.

Triknya adalah bahwa adalah mungkin untuk menghitung keputusan minimax yang benar tanpa melihat setiap simpul dalam pohon permainan.

Artinya, kita dapat meminjam ide **pemangkasan** dari Bab 3 untuk menghilangkan bagian besar pohon dari pertimbangan. Teknik khusus yang kita periksa disebut **pemangkasan alfa–beta**. Ketika diterapkan pada pohon minimax standar, ia mengembalikan langkah yang sama seperti yang akan dilakukan minimax, tetapi memangkas cabang-cabang yang tidak mungkin memengaruhi keputusan akhir.

Pertimbangkan lagi pohon permainan dua lapis dari Gambar 5.2. Mari kita bahas perhitungan keputusan optimal sekali lagi, kali ini dengan memperhatikan dengan saksama apa yang kita ketahui di setiap titik dalam proses tersebut. Langkah-langkahnya dijelaskan dalam Gambar 5.5. Hasilnya adalah kita dapat mengidentifikasi keputusan minimax tanpa pernah mengevaluasi dua simpul daun. Cara lain untuk melihat ini adalah sebagai penyederhanaan rumus untuk MINIMAX. Misalkan dua penerus simpul C yang tidak dievaluasi pada Gambar 5.5 memiliki nilai x dan y. Maka nilai simpul akar diberikan oleh

$$\begin{aligned} MINIMAX(root) &= \max(\min(3,12,8), \min(2,\mathrm{x},\mathrm{y}), \min(14,5,2)) \\ &= \max(3, \min(2,\mathrm{x},\mathrm{y}), 2) \\ &= \max(3, z, 2) \qquad \text{where z } = \min(2,\mathrm{x},\mathrm{y}) \leq 2 \\ &= 3. \end{aligned}$$

Dengan kata lain, nilai akar dan keputusan minimum tidak bergantung pada nilai daun yang dipangkas x dan y. Pemangkasan alfa–beta dapat diterapkan pada pohon dengan kedalaman berapa pun, dan sering kali memungkinkan untuk memangkas seluruh subpohon, bukan hanya daun. Prinsip umumnya adalah ini: pertimbangkan simpul n di suatu tempat di pohon (lihat Gambar 5.6), sehingga Pemain memiliki pilihan untuk pindah ke simpul tersebut. Jika Pemain memiliki pilihan yang lebih baik m baik di simpul induk n atau di titik pilihan mana pun yang lebih jauh ke atas, maka n *tidak akan pernah tercapai dalam permainan yang sebenarnya*. Jadi, setelah kita mengetahui cukup banyak tentang n (dengan memeriksa beberapa turunannya) untuk mencapai kesimpulan ini, kita dapat memangkasnya.

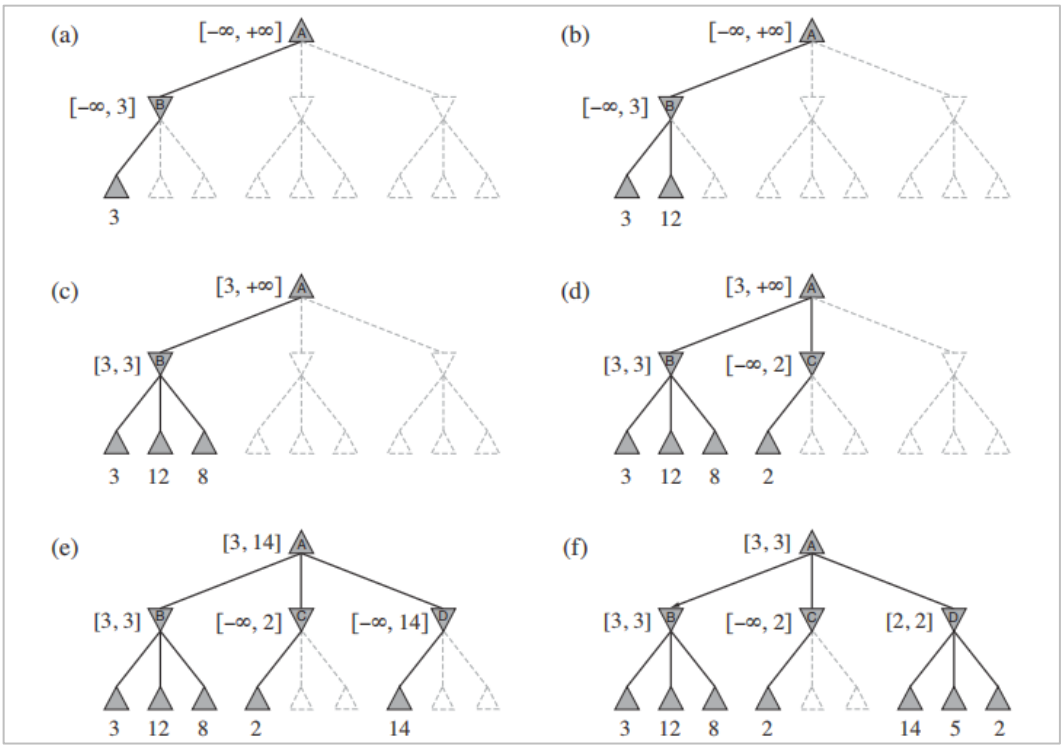
Ingat bahwa pencarian minimum mengutamakan kedalaman, jadi pada satu waktu kita hanya perlu mempertimbangkan simpul di sepanjang satu jalur di pohon. Pemangkasan alfa–beta mendapatkan namanya dari dua parameter berikut yang menjelaskan batasan pada nilai cadangan yang muncul di mana saja di sepanjang jalur:

α = nilai pilihan terbaik (yaitu, nilai tertinggi) yang telah kita temukan sejauh ini di titik pilihan mana pun di sepanjang jalur untuk MAX.

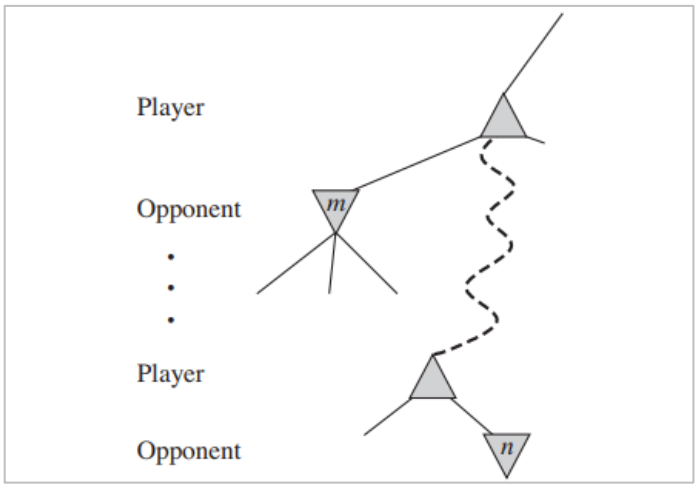
β = nilai pilihan terbaik (yaitu, nilai terendah) yang telah kita temukan sejauh ini di titik pilihan mana pun di sepanjang jalur untuk MIN.

Pencarian alfa-beta memperbarui nilai α dan β saat berjalan dan memangkas cabang yang tersisa di simpul (yaitu, mengakhiri panggilan rekursif) segera setelah nilai simpul saat ini

diketahui lebih buruk daripada nilai α atau β saat ini untuk MAX atau MIN, masing-masing. Algoritme lengkap diberikan dalam Gambar 5.7. Kami mendorong Anda untuk melacak perilakunya saat diterapkan ke pohon dalam Gambar 5.5.



Gambar 5.5 Tahapan dalam perhitungan keputusan optimal untuk pohon permainan pada Gambar 5.2. Pada setiap titik, kami menunjukkan rentang nilai yang mungkin untuk setiap simpul. (a) Daun pertama di bawah *B* memiliki nilai 3. Oleh karena itu, *B*, yang merupakan simpul MIN, memiliki nilai *paling banyak* 3. (b) Daun kedua di bawah *B* memiliki nilai 12; MIN akan menghindari langkah ini, jadi nilai *B* masih *paling banyak* 3. (c) Daun ketiga di bawah *B* memiliki nilai 8; kita telah melihat semua status penerus *B*, jadi nilai *B* tepat 3. Sekarang, kita dapat menyimpulkan bahwa nilai akarnya paling sedikit 3, karena MAX memiliki pilihan bernilai 3 di akarnya. (d) Daun pertama di bawah *C* memiliki nilai 2. Jadi, *C*, yang merupakan simpul MIN, memiliki nilai *paling banyak* 2. Namun, kita tahu bahwa *B* bernilai 3, jadi MAX tidak akan pernah memilih *C*. Jadi, tidak ada gunanya melihat status penerus *C* yang lain. Ini adalah contoh pemangkasan alfa–beta. (e) Daun pertama di bawah *D* memiliki nilai 14, jadi *D* bernilai *paling banyak* 14. Ini masih lebih tinggi dari alternatif terbaik MAX (yaitu, 3), jadi kita perlu terus mengeksplorasi status penerus *D*. Perhatikan juga bahwa kita sekarang memiliki batas pada semua penerus akar, jadi nilai akar juga paling banyak 14. (f) Penerus kedua *D* bernilai 5, jadi sekali lagi kita perlu terus mengeksplorasi. Penerus ketiga bernilai 2, jadi sekarang D bernilai tepat 2. Keputusan MAX di akar adalah pindah ke B, yang memberikan nilai 3.



Gambar 5.6 Kasus umum untuk pemangkasan alfa–beta. Jika m lebih baik daripada n untuk Pemain, kita tidak akan pernah mencapai n dalam permainan.

Urutan gerakan

Efektivitas pemangkasan alfa–beta sangat bergantung pada urutan pemeriksaan status. Misalnya, pada Gambar 5.5(e) dan (f), kita tidak dapat memangkas penerus *D* sama sekali karena penerus terburuk (dari sudut pandang MIN) dihasilkan terlebih dahulu. Jika penerus ketiga *D* dihasilkan terlebih dahulu, kita akan dapat memangkas dua lainnya. Ini menunjukkan bahwa mungkin ada baiknya mencoba memeriksa terlebih dahulu penerus yang kemungkinan besar terbaik.

Jika ini dapat dilakukan, maka ternyata alfa–beta hanya perlu memeriksa $O(b^{m/2})$ simpul untuk memilih gerakan terbaik, bukan $O(b^m)$ untuk minimum. Ini berarti bahwa faktor percabangan efektif menjadi $\sqrt{b}$, bukan b—untuk catur, sekitar 6, bukan 35. Dengan kata lain, alfa–beta dapat memecahkan pohon kira-kira dua kali lebih dalam dari minimum dalam jumlah waktu yang sama. Jika penerus diperiksa secara acak, bukan berdasarkan urutan terbaik, jumlah total simpul yang diperiksa akan menjadi sekitar $O(b^{3m/4})$ untuk *b* sedang. Untuk catur, fungsi pemesanan yang cukup sederhana (seperti mencoba tangkapan terlebih dahulu, lalu ancaman, lalu langkah maju, dan kemudian langkah mundur) akan membawa Anda ke dalam sekitar faktor 2 dari hasil kasus terbaik $O(b^{m/2})$.

Menambahkan skema urutan langkah yang dinamis, seperti mencoba terlebih dahulu langkah-langkah yang ditemukan sebagai yang terbaik di masa lalu, membawa kita cukup dekat ke batas teoritis. Masa lalu bisa jadi adalah langkah sebelumnya—seringkali ancaman yang sama tetap ada—atau bisa berasal dari eksplorasi sebelumnya atas langkah saat ini. Salah satu cara untuk memperoleh informasi dari langkah saat ini adalah dengan pencarian pendalaman berulang. Pertama, cari 1 lapis dalam dan catat jalur langkah terbaik.

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state,−∞,+ ∞)
    return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state,α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a),α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

```
function MAX-VALUE(state,α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a),α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

Gambar 5.7 Algoritma pencarian alfa-beta. Perhatikan bahwa rutinitas ini sama dengan fungsi MINIMAX pada Gambar 5.3, kecuali untuk dua baris di masing-masing MIN-VALUE dan MAX-VALUE yang mempertahankan α dan β (dan pembukuan untuk meneruskan parameter ini).

Kemudian cari 1 lapis lebih dalam, tetapi gunakan jalur yang terekam untuk menginformasikan urutan langkah. Seperti yang kita lihat di Bab 3, pendalaman berulang pada pohon permainan eksponensial hanya menambahkan sebagian kecil waktu pencarian total, yang dapat lebih dari sekadar dibuat dari urutan langkah yang lebih baik. Langkah terbaik sering disebut gerakan pembunuh dan mencobanya terlebih dahulu disebut heuristik gerakan pembunuh.

Di Bab 3, kita mencatat bahwa status berulang di pohon pencarian dapat menyebabkan peningkatan eksponensial dalam biaya pencarian. Dalam banyak permainan, status berulang sering terjadi karena **transposisi**—permutasi berbeda dari urutan langkah yang berakhir di posisi yang sama. Misalnya, jika Putih memiliki satu langkah, a_1, yang dapat dijawab oleh Hitam dengan b_1 dan langkah yang tidak terkait a_2 di sisi lain papan yang dapat dijawab oleh b_2, maka urutan $[a_1, b_1, a_2, b_2]$ dan $[a_2, b_2, a_1, b_1]$ keduanya berakhir di posisi yang sama.

Sebaiknya evaluasi posisi yang dihasilkan disimpan dalam tabel hash saat pertama kali ditemui sehingga kita tidak perlu menghitungnya ulang pada kejadian berikutnya. Tabel hash dari posisi yang terlihat sebelumnya secara tradisional disebut tabel transposisi; pada dasarnya identik dengan daftar yang dieksplorasi dalam PENCARIAN GRAFIK (Bagian 3.3). Menggunakan tabel transposisi dapat memiliki efek dramatis, terkadang hingga menggandakan kedalaman pencarian yang dapat dicapai dalam catur. Di sisi lain, jika kita mengevaluasi satu juta node per

detik, pada titik tertentu tidak praktis untuk menyimpan semuanya dalam tabel transposisi. Berbagai strategi telah digunakan untuk memilih node mana yang akan disimpan dan mana yang akan dibuang.

5.4 KEPUTUSAN WAKTU NYATA YANG TIDAK SEMPURNA

Algoritma minimax menghasilkan seluruh ruang pencarian permainan, sedangkan algoritma alpha–beta memungkinkan kita untuk memangkas sebagian besarnya. Namun, alpha–beta masih harus mencari hingga ke status terminal setidaknya untuk sebagian dari ruang pencarian. Kedalaman ini biasanya tidak praktis, karena langkah harus dilakukan dalam jumlah waktu yang wajar—biasanya paling lama beberapa menit.

Makalah *Claude Shannon Programming a Computer for Playing Chess* (1950) mengusulkan agar program menghentikan pencarian lebih awal dan menerapkan **fungsi evaluasi** heuristik ke status dalam pencarian, yang secara efektif mengubah simpul nonterminal menjadi daun terminal. Dengan kata lain, sarannya adalah mengubah minimax atau alpha–beta dengan dua cara: mengganti fungsi utilitas dengan fungsi evaluasi heuristik EVAL, yang memperkirakan utilitas posisi, dan mengganti uji terminal dengan **uji batas** yang memutuskan kapan akan menerapkan EVAL. Hal ini memberi kita nilai minimum heuristik berikut untuk keadaan s dan kedalaman maksimum *d*:

$$\mathrm{H-MINIMAX}(s,d) = \begin{cases} \mathrm{UTILITY}(s) & \text{if } \mathrm{CUTOFF-TEST}(s,d) \\ max_{a \in Actions\,(s)} \mathrm{MINIMAX}\big(\mathrm{RESULT}(s,a)\big) & \text{if } \mathrm{PLAYER}(s) = \mathrm{MAX} \\ min_{a \in Actions\,(s)} \mathrm{MINIMAX}\big(\mathrm{RESULT}(s,a)\big) & \text{if } \mathrm{PLAYER}(s) = \mathrm{MIN} \end{cases}$$

Fungsi evaluasi

Fungsi evaluasi mengembalikan *estimasi* utilitas permainan yang diharapkan dari posisi tertentu, sama seperti fungsi heuristik Bab 3 mengembalikan estimasi jarak ke gawang. Ide tentang estimator bukanlah hal baru ketika Shannon mengusulkannya. Selama berabad-abad, pemain catur (dan penggemar permainan lainnya) telah mengembangkan cara untuk menilai nilai suatu posisi karena manusia bahkan lebih terbatas dalam jumlah pencarian yang dapat mereka lakukan daripada program komputer. Harus jelas bahwa kinerja program permainan sangat bergantung pada kualitas fungsi evaluasinya. Fungsi evaluasi yang tidak akurat akan mengarahkan agen ke posisi yang ternyata kalah. Bagaimana tepatnya kita merancang fungsi evaluasi yang baik?

Pertama, fungsi evaluasi harus mengurutkan status terminal dengan cara yang sama seperti fungsi utilitas sebenarnya: status yang menang harus dievaluasi lebih baik daripada seri, yang pada gilirannya harus lebih baik daripada kekalahan. Jika tidak, agen yang menggunakan fungsi evaluasi mungkin salah meskipun dapat melihat ke depan hingga akhir permainan. Kedua, perhitungan tidak boleh memakan waktu terlalu lama! (Intinya adalah untuk mencari lebih cepat.) Ketiga, untuk status nonterminal, fungsi evaluasi harus berkorelasi kuat dengan peluang menang yang sebenarnya.

Seseorang mungkin bertanya-tanya tentang frasa "peluang menang." Bagaimanapun, catur bukanlah permainan untung-untungan: kita mengetahui keadaan saat ini dengan pasti, dan tidak ada dadu yang terlibat. Namun, jika pencarian harus dihentikan pada keadaan nonterminal, maka algoritme tersebut tentu tidak pasti tentang hasil akhir dari keadaan tersebut. Jenis ketidakpastian ini disebabkan oleh keterbatasan komputasi, bukan informasi. Mengingat jumlah komputasi terbatas yang dapat dilakukan fungsi evaluasi untuk keadaan tertentu, yang terbaik yang dapat dilakukannya adalah menebak hasil akhir.

Mari kita buat gagasan ini lebih konkret. Sebagian besar fungsi evaluasi bekerja dengan menghitung berbagai **fitur** keadaan—misalnya, dalam catur, kita akan memiliki fitur untuk jumlah pion putih, pion hitam, ratu putih, ratu hitam, dan seterusnya. Fitur-fitur tersebut, jika digabungkan, menentukan berbagai **kategori atau kelas ekivalensi** keadaan: keadaan dalam setiap kategori memiliki nilai yang sama untuk semua fitur. Misalnya, satu kategori berisi semua permainan akhir dua pion vs. satu pion. Secara umum, setiap kategori yang diberikan akan berisi beberapa status yang mengarah pada kemenangan, beberapa yang mengarah pada seri, dan beberapa yang mengarah pada kekalahan. Fungsi evaluasi tidak dapat mengetahui status mana yang mana, tetapi dapat mengembalikan satu nilai yang mencerminkan proporsi status dengan setiap hasil. Misalnya, anggaplah pengalaman kami menunjukkan bahwa 72% status yang ditemui dalam kategori dua pion vs. satu pion mengarah pada kemenangan (utilitas $+1$); 20% ke kekalahan (0), dan 8% ke seri $(1/2)$. Maka evaluasi yang wajar untuk status dalam kategori tersebut adalah **nilai yang diharapkan**: $(0{,}72 \times +1) + (0{,}20 \times 0) + (0{,}08 \times 1/2) = 0{,}76$. Pada prinsipnya, nilai yang diharapkan dapat ditentukan untuk setiap kategori, yang menghasilkan fungsi evaluasi yang berfungsi untuk status apa pun. Seperti halnya status terminal, fungsi evaluasi tidak perlu mengembalikan nilai yang diharapkan sebenarnya selama urutan statusnya sama.

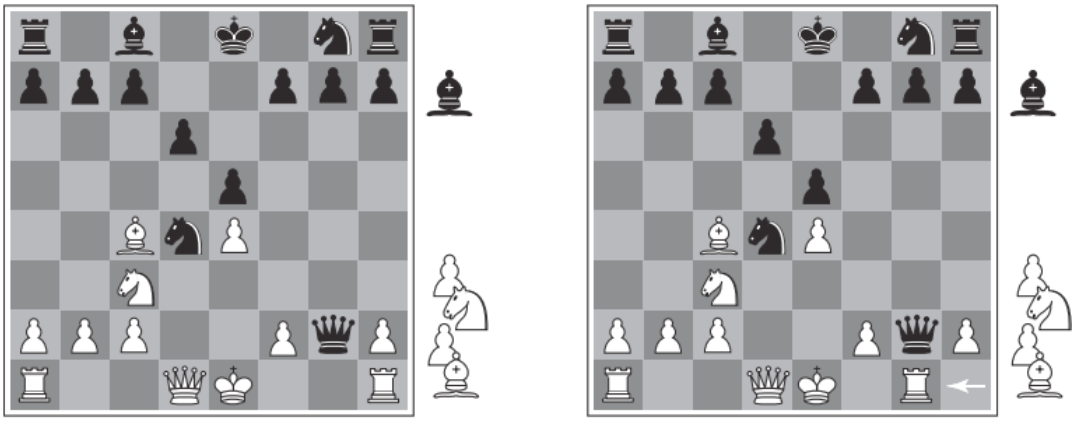
Dalam praktiknya, analisis semacam ini memerlukan terlalu banyak kategori dan karenanya terlalu banyak pengalaman untuk memperkirakan semua kemungkinan menang. Sebaliknya, sebagian besar fungsi evaluasi menghitung kontribusi numerik terpisah dari setiap fitur dan kemudian **menggabungkannya** untuk menemukan nilai total. Misalnya, buku catur pengantar memberikan **nilai material** perkiraan untuk setiap buah catur: setiap pion bernilai 1, kuda atau gajah bernilai 3, benteng bernilai 5, dan ratu bernilai 9. Fitur lain seperti "struktur pion yang baik" dan "keselamatan raja" mungkin bernilai setengah pion, misalnya. Nilai fitur ini kemudian dijumlahkan untuk mendapatkan evaluasi posisi.

Keuntungan aman yang setara dengan pion memberikan kemungkinan besar untuk menang, dan keuntungan aman yang setara dengan tiga pion seharusnya memberikan kemenangan yang hampir pasti, seperti yang diilustrasikan dalam Gambar 5.8(a). Secara matematis, fungsi evaluasi semacam ini disebut **fungsi linier tertimbang** karena dapat dinyatakan sebagai

$$\text{Eval}(s) = \omega_1 f_1(s) + \omega_2 f_2(s) + \cdots + \omega_n f_n(s) = \sum_{i=1}^{n} \omega_i f_i(s) ,$$

di mana setiap w_i w_i adalah bobot dan setiap f_i adalah fitur posisi. Untuk catur, f_i dapat berupa angka dari setiap jenis buah catur di papan, dan w_i dapat berupa nilai buah catur (1 untuk pion, 3 untuk gajah, dst.).

Menjumlahkan nilai fitur tampaknya merupakan hal yang wajar untuk dilakukan, tetapi sebenarnya hal itu melibatkan asumsi yang kuat: bahwa kontribusi setiap fitur tidak bergantung pada nilai fitur lainnya. Misalnya, menetapkan nilai 3 untuk gajah mengabaikan fakta bahwa gajah lebih kuat di akhir permainan, ketika mereka memiliki banyak ruang untuk bermanuver.



(a) White to move (b) White to move

Gambar 5.8 Dua posisi catur yang hanya berbeda pada posisi benteng di kanan bawah. Pada (a), Hitam memiliki keunggulan seekor kuda dan dua pion, yang seharusnya cukup untuk memenangkan permainan. Pada (b), Putih akan menangkap ratu, yang memberinya keunggulan yang seharusnya cukup kuat untuk menang.

Karena alasan ini, program terkini untuk catur dan permainan lainnya juga menggunakan kombinasi fitur nonlinier. Misalnya, sepasang gajah mungkin bernilai sedikit lebih dari dua kali lipat nilai gajah tunggal, dan gajah bernilai lebih tinggi di akhir permainan (yaitu, saat fitur nomor langkah tinggi atau fitur *jumlah buah yang tersisa* rendah).

Pembaca yang cermat akan menyadari bahwa fitur dan bobot bukanlah bagian dari aturan catur! Fitur dan bobot tersebut berasal dari pengalaman bermain catur manusia selama berabad-abad. Dalam permainan yang tidak menyediakan pengalaman semacam ini, bobot fungsi evaluasi dapat diperkirakan dengan teknik pembelajaran mesin. Yang melegakan, penerapan teknik ini pada catur telah mengonfirmasi bahwa gajah memang bernilai sekitar tiga pion.

Memotong pencarian

Langkah berikutnya adalah memodifikasi ALPHA – BETA – SEARCH sehingga akan memanggil fungsi EVAL heuristik saat pencarian harus dihentikan. Kami mengganti dua baris pada Gambar 5.7 yang menyebutkan TERMINAL – TEST dengan baris berikut:

if $\text{CUTOFF} - \text{TEST}(state, depth)$ **then return** $\text{EVAL}(state)$

Kita juga harus mengatur beberapa pembukuan sehingga $depth$ saat ini bertambah pada setiap panggilan rekursif. Pendekatan paling mudah untuk mengendalikan jumlah pencarian adalah dengan menetapkan batas kedalaman tetap sehingga $\text{CUTOFF} - \text{TEST}(state, depth)$ mengembalikan $true$ untuk semua $depth$ yang lebih besar dari beberapa kedalaman tetap d. (Ia juga harus mengembalikan $true$ untuk semua status terminal, seperti yang dilakukan $\text{TERMINAL} - \text{TEST}.$) Kedalaman d dipilih sehingga gerakan dipilih dalam waktu yang dialokasikan. Pendekatan yang lebih kuat adalah menerapkan pendalaman iteratif. (Lihat Bab 3.) Ketika waktu habis, program mengembalikan gerakan yang dipilih oleh pencarian terdalam yang diselesaikan. Sebagai bonus, pendalaman iteratif juga membantu dengan urutan gerakan.

Pendekatan sederhana ini dapat menyebabkan kesalahan karena sifat perkiraan fungsi evaluasi. Pertimbangkan lagi fungsi evaluasi sederhana untuk catur berdasarkan keuntungan material. Misalkan program mencari hingga batas kedalaman, mencapai posisi pada Gambar 5.8(b), di mana Hitam unggul dengan satu kuda dan dua pion. Program akan melaporkan ini sebagai nilai heuristik keadaan, dengan demikian menyatakan bahwa keadaan tersebut merupakan kemenangan yang mungkin bagi Hitam. Namun, langkah Putih berikutnya menangkap ratu Hitam tanpa kompensasi. Oleh karena itu, posisi tersebut benar-benar dimenangkan oleh Putih, tetapi ini hanya dapat dilihat dengan melihat ke depan satu langkah lagi.

Jelas, diperlukan uji batas yang lebih canggih. Fungsi evaluasi harus diterapkan hanya pada posisi yang **tenang**—yaitu, tidak mungkin menunjukkan perubahan nilai yang liar dalam waktu dekat. Dalam catur, misalnya, posisi di mana penangkapan yang menguntungkan dapat dilakukan tidak tenang untuk fungsi evaluasi yang hanya menghitung material. Posisi yang tidak tenang dapat diperluas lebih jauh hingga posisi tenang tercapai. Pencarian tambahan ini disebut **pencarian ketenangan**; terkadang dibatasi untuk hanya mempertimbangkan jenis gerakan tertentu, seperti gerakan tangkap, yang akan dengan cepat menyelesaikan ketidakpastian dalam posisi.

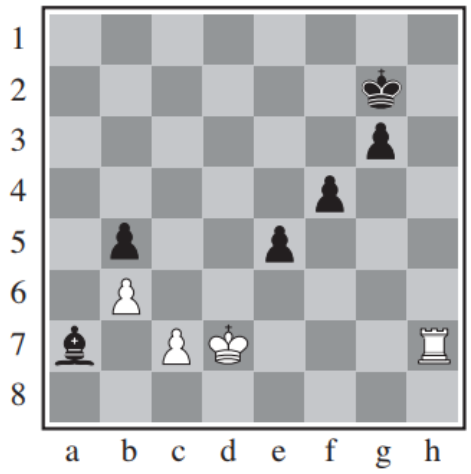
Efek horizon lebih sulit dihilangkan. Efek ini muncul ketika program menghadapi gerakan lawan yang menyebabkan kerusakan serius dan pada akhirnya tidak dapat dihindari, tetapi dapat dihindari sementara dengan menunda taktik. Perhatikan permainan catur pada Gambar 5.9. Jelas bahwa tidak ada cara bagi gajah hitam untuk melarikan diri. Misalnya, benteng putih dapat menangkapnya dengan bergerak ke h1, lalu a1, lalu a2; penangkapan pada kedalaman 6 lapis. Namun, Hitam memiliki serangkaian gerakan yang mendorong penangkapan gajah "melewati horizon." Misalkan Hitam mencari hingga kedalaman 8 lapis. Sebagian besar gerakan oleh Hitam akan mengarah pada penangkapan gajah pada akhirnya, dan dengan demikian akan ditandai sebagai gerakan "buruk". Namun, Hitam akan mempertimbangkan untuk melakukan skak raja putih dengan pion di e4. Ini akan menyebabkan raja menangkap pion. Sekarang Hitam akan mempertimbangkan untuk melakukan skak lagi, dengan pion di f5, yang mengarah pada penangkapan pion lainnya. Itu

menghabiskan 4 ply, dan dari sana 4 ply yang tersisa tidak cukup untuk menangkap gajah. Hitam berpikir bahwa garis permainan telah menyelamatkan gajah dengan mengorbankan dua pion, padahal yang sebenarnya dilakukannya hanyalah mendorong penangkapan gajah yang tak terelakkan melampaui cakrawala yang dapat dilihat Hitam.

Satu strategi untuk mengurangi **efek cakrawala** adalah **ekstensi tunggal**, langkah yang "jelas lebih baik" daripada semua langkah lain dalam posisi tertentu. Setelah ditemukan di mana saja di pohon selama pencarian, langkah tunggal ini diingat. Ketika pencarian mencapai batas kedalaman normal, algoritme memeriksa untuk melihat apakah ekstensi tunggal adalah langkah yang sah; jika ya, algoritme memungkinkan langkah tersebut dipertimbangkan. Ini membuat pohon lebih dalam, tetapi karena akan ada beberapa ekstensi tunggal, itu tidak menambahkan banyak simpul total ke pohon.

Pemangkasan ke depan

Sejauh ini, kita telah berbicara tentang pemotongan pencarian pada tingkat tertentu dan tentang melakukan pemangkasan alfa–beta yang terbukti tidak berpengaruh pada hasil (setidaknya berkenaan dengan nilai evaluasi heuristik). **Pemangkasan ke depan** juga dapat dilakukan, yang berarti bahwa beberapa langkah pada simpul tertentu dipangkas segera tanpa pertimbangan lebih lanjut. Jelas, sebagian besar manusia yang bermain catur hanya mempertimbangkan beberapa langkah dari setiap posisi (setidaknya secara sadar). Salah satu pendekatan untuk pemangkasan ke depan adalah **pencarian balok**: pada setiap ply, pertimbangkan hanya "balok" dari n langkah terbaik (menurut fungsi evaluasi) daripada mempertimbangkan semua langkah yang mungkin.



Gambar 5.9 Efek horizon. Dengan Hitam yang bergerak, gajah hitam pasti akan kalah. Namun, Hitam dapat mencegah kejadian itu dengan menghentikan raja putih dengan pionnya, yang memaksa raja untuk menangkap pion. Hal ini mendorong hilangnya gajah yang tak terelakkan melewati horizon, dan dengan demikian pengorbanan pion dilihat oleh algoritma pencarian sebagai langkah yang baik, bukan langkah yang buruk.

Sayangnya, pendekatan ini agak berbahaya karena tidak ada jaminan bahwa langkah terbaik tidak akan terpangkas. Algoritma PROBCUT, atau pemotongan probabilistik adalah versi pemangkasan maju dari penelusuran alfa-beta yang menggunakan statistik yang

diperoleh dari pengalaman sebelumnya untuk mengurangi kemungkinan langkah terbaik akan terpangkas. Penelusuran $alfa - beta$ memangkas setiap simpul yang terbukti berada di luar jendela (α, β) saat ini. PROBCUT juga memangkas simpul yang mungkin berada di luar jendela. Ia menghitung probabilitas ini dengan melakukan penelusuran dangkal untuk menghitung nilai cadangan v dari sebuah simpul dan kemudian menggunakan pengalaman sebelumnya untuk memperkirakan seberapa besar kemungkinan skor v pada kedalaman d di pohon akan berada di luar (α, β).

Buro menerapkan teknik ini pada program Othello-nya, LOGISTELLO, dan menemukan bahwa versi programnya dengan PROBCUT mengalahkan versi reguler sebanyak 64% dari waktu, bahkan ketika versi reguler diberi waktu dua kali lebih banyak. Menggabungkan semua teknik yang dijelaskan di sini menghasilkan program yang dapat memainkan catur yang kredibel (atau permainan lainnya). Mari kita asumsikan kita telah menerapkan fungsi evaluasi untuk catur, uji batas yang wajar dengan pencarian ketenangan, dan tabel transposisi yang besar. Mari kita asumsikan juga bahwa, setelah berbulan-bulan melakukan bit-bashing yang membosankan, kita dapat menghasilkan dan mengevaluasi sekitar satu juta node per detik pada PC terbaru, yang memungkinkan kita untuk mencari sekitar 200 juta node per langkah di bawah kendali waktu standar (tiga menit per langkah).

Faktor percabangan untuk catur adalah sekitar 35, rata-rata, dan 35^5 adalah sekitar 50 juta, jadi jika kita menggunakan pencarian minimax, kita hanya dapat melihat ke depan sekitar lima plies. Meskipun tidak tidak kompeten, program seperti itu dapat dengan mudah dikelabui oleh pemain catur manusia rata-rata, yang terkadang dapat merencanakan enam atau delapan plies ke depan. Dengan pencarian alfa-beta kita mendapatkan sekitar 10 plies, yang menghasilkan tingkat permainan ahli. Bagian 5.8 menjelaskan teknik pemangkasan tambahan yang dapat memperluas kedalaman pencarian efektif hingga sekitar 14 plies. Untuk mencapai status grandmaster, kita memerlukan fungsi evaluasi yang disetel secara ekstensif dan basis data besar berisi gerakan pembukaan dan akhir permainan yang optimal.

Pencarian versus penelusuran

Entah mengapa tampaknya berlebihan bagi program catur untuk memulai permainan dengan mempertimbangkan pohon berisi satu miliar status permainan, hanya untuk menyimpulkan bahwa ia akan menggerakkan pionnya ke e4. Buku-buku yang menjelaskan permainan yang baik dalam pembukaan dan akhir permainan catur telah tersedia selama sekitar satu abad. Oleh karena itu, tidak mengherankan bahwa banyak program permainan menggunakan *penelusuran tabel* daripada menelusuri pembukaan dan akhir permainan.

Untuk pembukaan, komputer sebagian besar mengandalkan keahlian manusia. Saran terbaik dari para ahli manusia tentang cara memainkan setiap pembukaan disalin dari buku dan dimasukkan ke dalam tabel untuk digunakan komputer. Namun, komputer juga dapat mengumpulkan statistik dari basis data permainan yang dimainkan sebelumnya untuk melihat urutan pembukaan mana yang paling sering menghasilkan kemenangan. Pada langkah-langkah awal, hanya ada sedikit pilihan, dan dengan demikian banyak komentar ahli dan permainan sebelumnya yang dapat diambil. Biasanya setelah sepuluh langkah, kita berakhir pada posisi

yang jarang terlihat, dan program harus beralih dari penelusuran tabel ke penelusuran. Menjelang akhir permainan, posisi yang mungkin semakin sedikit lagi, dan dengan demikian peluang untuk melakukan pencarian pun semakin besar. Namun, di sinilah komputer memiliki keahlian: analisis komputer terhadap permainan akhir jauh melampaui apa pun yang dicapai oleh manusia. Seorang manusia dapat memberi tahu Anda strategi umum untuk memainkan permainan akhir raja-dan-benteng-versus-raja (KRK): kurangi mobilitas raja lawan dengan menekannya ke salah satu sisi papan, gunakan raja Anda untuk mencegah lawan lolos dari tekanan.

Akhir permainan lainnya, seperti raja, gajah, dan kuda versus raja (KBNK), sulit dikuasai dan tidak memiliki deskripsi strategi yang ringkas. Di sisi lain, komputer dapat sepenuhnya memecahkan permainan akhir dengan membuat **kebijakan**, yang merupakan pemetaan dari setiap kemungkinan keadaan ke langkah terbaik dalam keadaan tersebut. Kemudian, kita dapat mencari langkah terbaik daripada menghitungnya ulang. Seberapa besar tabel pencarian KBNK nantinya? Ternyata ada 462 cara untuk menempatkan dua raja di papan tanpa harus berdekatan. Setelah raja ditempatkan, ada 62 petak kosong untuk gajah, 61 untuk kuda, dan dua pemain yang mungkin untuk melangkah selanjutnya, jadi hanya ada $462 \times 62 \times 61 \times 2 = 3.494.568$ posisi yang mungkin. Beberapa di antaranya adalah skakmat; tandai seperti itu dalam tabel. Kemudian lakukan pencarian minimax retrograde: balikkan aturan catur untuk melakukan unmoves alih-alih bergerak.

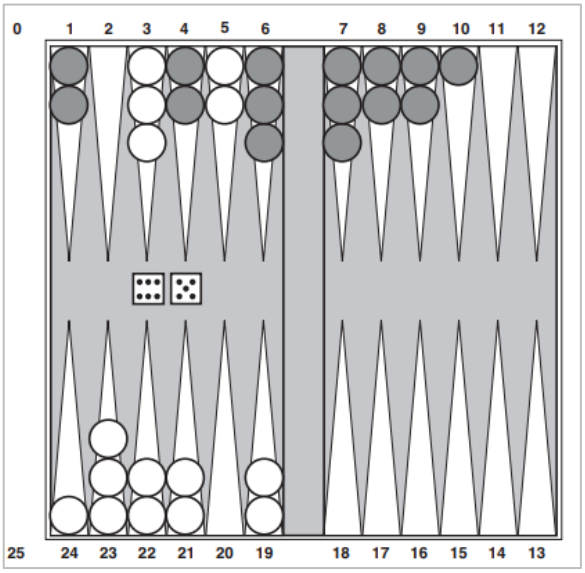
Setiap langkah oleh Putih yang, tidak peduli langkah apa yang ditanggapi Hitam, berakhir di posisi yang ditandai sebagai menang, juga harus menang. Lanjutkan pencarian ini hingga semua $3.494.568$ posisi diselesaikan sebagai menang, kalah, atau seri, dan Anda memiliki tabel pencarian yang sempurna untuk semua permainan akhir KBNK. Dengan menggunakan teknik ini dan serangkaian trik optimasi, Ken Thompson (1986, 1996) dan Lewis Stiller (1992, 1996) memecahkan semua permainan akhir catur dengan hingga lima buah dan beberapa dengan enam buah, sehingga dapat diakses melalui Internet. Stiller menemukan satu kasus di mana mat yang dipaksakan ada tetapi memerlukan 262 gerakan; hal ini menimbulkan kekhawatiran karena aturan catur mengharuskan gerakan tangkap atau pion terjadi dalam 50 gerakan. Karya selanjutnya oleh Marc Bourzutschky dan Yakov Konoval memecahkan semua permainan akhir enam buah tanpa pion dan beberapa permainan akhir tujuh buah; ada permainan akhir KQNKRBN yang dengan permainan terbaik memerlukan 517 gerakan hingga tangkapan, yang kemudian mengarah ke mat.

Jika kita dapat memperluas tabel akhir permainan catur dari 6 buah menjadi 32, maka Putih akan tahu pada langkah pembukaan apakah itu akan menjadi menang, kalah, atau seri. Hal ini belum terjadi sejauh ini untuk catur, tetapi telah terjadi untuk dam, seperti yang dijelaskan di bagian catatan sejarah.

5.5 PERMAINAN STOKASTIK

Dalam kehidupan nyata, banyak kejadian eksternal yang tidak terduga dapat menempatkan kita dalam situasi yang tidak terduga. Banyak permainan mencerminkan ketidakpastian ini dengan memasukkan elemen acak, seperti melempar dadu. Kami

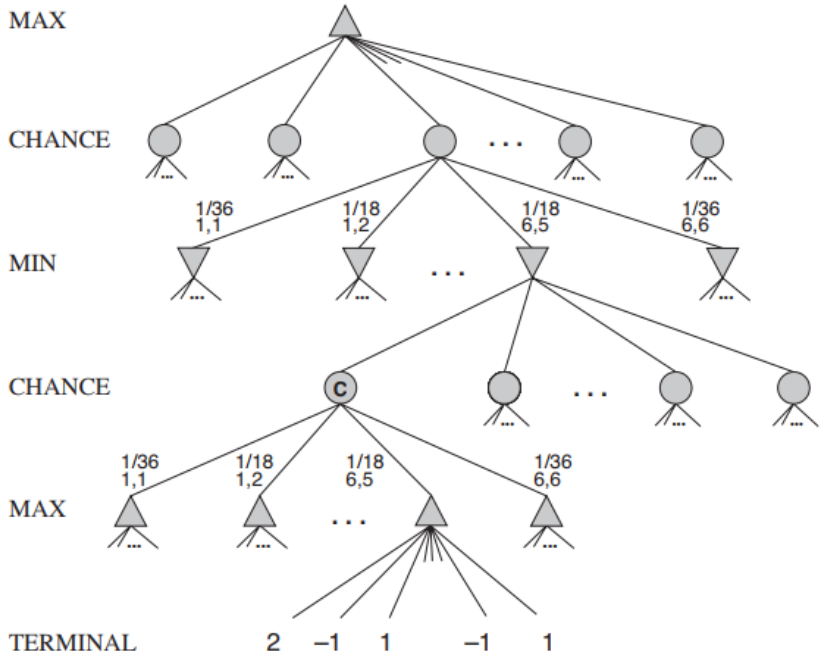
menyebutnya **permainan stokastik**. Backgammon adalah permainan khas yang menggabungkan keberuntungan dan keterampilan. Dadu dilempar di awal giliran pemain untuk menentukan langkah yang sah. Dalam posisi backgammon pada Gambar 5.10, misalnya, Putih telah melempar angka 6–5 dan memiliki empat kemungkinan langkah.



Gambar 5.10 Posisi khas permainan backgammon. Tujuan permainan ini adalah untuk memindahkan semua bidak dari papan. Putih bergerak searah jarum jam menuju angka 25, dan Hitam bergerak berlawanan arah jarum jam menuju angka 0. Sebuah bidak dapat bergerak ke posisi mana pun kecuali jika ada beberapa bidak lawan di sana; jika ada satu lawan, bidak tersebut akan ditangkap dan harus memulai dari awal. Pada posisi yang ditunjukkan, Putih telah melempar dadu 6–5 dan harus memilih di antara empat langkah yang sah: (5–10,5–11), (5–11,19–24), (5–10,10–16), dan (5–11,11–16), di mana notasi (5–11,11–16) berarti memindahkan satu bidak dari posisi 5 ke 11, lalu memindahkan satu bidak dari 11 ke 16.

Meskipun Putih tahu apa langkah sahnya sendiri, Putih tidak tahu apa yang akan digulirkan Hitam dan dengan demikian tidak tahu apa langkah sah Hitam. Itu berarti Putih tidak dapat membuat pohon permainan standar seperti yang kita lihat dalam catur dan tic-tac-toe. Pohon permainan dalam backgammon harus menyertakan simpul peluang selain simpul MAKS dan MIN. **Simpul peluang** ditunjukkan sebagai lingkaran pada Gambar 5.11.

Cabang-cabang yang mengarah dari setiap simpul peluang menunjukkan kemungkinan lemparan dadu; setiap cabang diberi label dengan lemparan dan probabilitasnya. Ada 36 cara untuk melempar dua dadu, masing-masing sama kemungkinannya; tetapi karena 6–5 sama dengan 5–6, hanya ada 21 lemparan berbeda. Enam ganda (1–1 hingga 6–6) masing-masing memiliki probabilitas 1/36, jadi kita katakan P (1–1) = 1/36. 15 lemparan berbeda lainnya masing-masing memiliki probabilitas 1/18.



Gambar 5.11 Pohon permainan skematik untuk posisi backgammon.

Langkah selanjutnya adalah memahami cara membuat keputusan yang tepat. Jelas, kita tetap ingin memilih langkah yang mengarah ke posisi terbaik. Namun, posisi tidak memiliki nilai minimum yang pasti. Sebaliknya, kita hanya dapat menghitung **nilai yang diharapkan** dari suatu posisi: rata-rata dari semua kemungkinan hasil dari simpul peluang.

Hal ini membawa kita untuk menggeneralisasi **nilai minimum** untuk permainan deterministik ke nilai minimum yang diharapkan untuk permainan dengan simpul peluang. Simpul terminal dan simpul MAX dan MIN (yang lemparan dadunya diketahui) bekerja dengan cara yang persis sama seperti sebelumnya. Untuk simpul peluang, kita menghitung nilai yang diharapkan, yang merupakan jumlah nilai dari semua hasil, yang dibobot dengan probabilitas setiap tindakan peluang:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL} - \text{TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\,\text{EXPECTIMINIMAX}\big(\text{RESULT}(s,r)\big) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

di mana r merupakan kemungkinan lemparan dadu (atau kejadian peluang lainnya) dan $\text{RESULT}(s,r)$ adalah keadaan yang sama dengan s, dengan fakta tambahan bahwa hasil lemparan dadu adalah r.

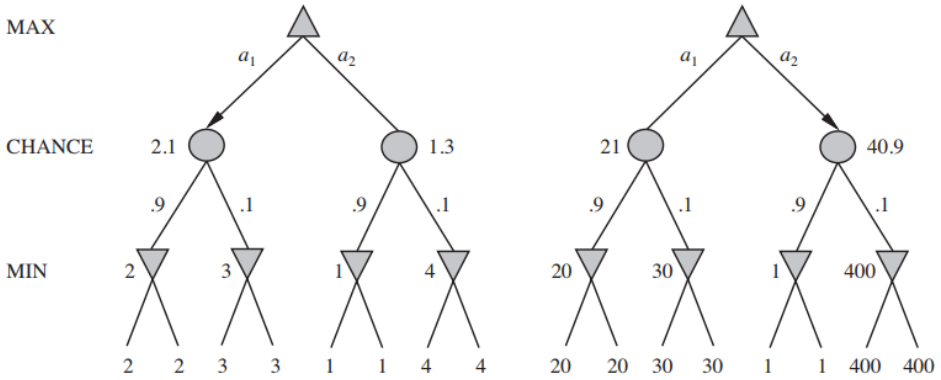
Fungsi evaluasi untuk permainan untung-untungan

Seperti halnya minimax, perkiraan yang jelas untuk dilakukan dengan expectiminimax adalah dengan menghentikan pencarian di beberapa titik dan menerapkan fungsi evaluasi pada setiap daun. Seseorang mungkin berpikir bahwa fungsi evaluasi untuk permainan seperti

backgammon harus sama seperti fungsi evaluasi untuk catur—mereka hanya perlu memberikan skor yang lebih tinggi untuk posisi yang lebih baik.

Namun pada kenyataannya, keberadaan simpul peluang berarti bahwa seseorang harus lebih berhati-hati tentang apa arti nilai evaluasi. Gambar 5.12 menunjukkan apa yang terjadi: dengan fungsi evaluasi yang menetapkan nilai [1, 2, 3, 4] pada daun, langkah a_1 adalah yang terbaik; dengan nilai [1, 20, 30, 400], langkah a_2 adalah yang terbaik.

Oleh karena itu, program tersebut berperilaku sangat berbeda jika kita membuat perubahan dalam skala beberapa nilai evaluasi! Ternyata untuk menghindari sensitivitas ini, fungsi evaluasi harus berupa transformasi linier positif dari probabilitas menang dari suatu posisi (atau, lebih umum, dari utilitas yang diharapkan dari posisi tersebut). Ini adalah sifat penting dan umum dari situasi yang melibatkan ketidakpastian.



Gambar 5.12 Transformasi pemeliharaan tatanan pada nilai daun mengubah langkah terbaik.

Jika program mengetahui terlebih dahulu semua lemparan dadu yang akan terjadi selama sisa permainan, menyelesaikan permainan dengan dadu akan sama seperti menyelesaikan permainan tanpa dadu, yang dilakukan mini-max dalam waktu $O(b^m)$, di mana b adalah faktor percabangan dan m adalah kedalaman maksimum pohon permainan. Karena expectiminimax juga mempertimbangkan semua kemungkinan urutan lemparan dadu, maka akan dibutuhkan $O(b^m n^m)$, di mana n adalah jumlah lemparan yang berbeda.

Bahkan jika kedalaman pencarian dibatasi pada kedalaman kecil d, biaya tambahan dibandingkan dengan minimax membuatnya tidak realistis untuk mempertimbangkan melihat ke depan sangat jauh dalam sebagian besar permainan untung-untungan. Dalam backgammon, n adalah 21 dan b biasanya sekitar 20, tetapi dalam beberapa situasi dapat mencapai 4000 untuk lemparan dadu yang ganda. Tiga kali lipat mungkin adalah semua yang dapat kita lakukan.

Cara lain untuk memikirkan masalah ini adalah ini: keuntungan dari alpha–beta adalah mengabaikan perkembangan di masa mendatang yang tidak akan terjadi, mengingat permainan terbaik. Jadi, ia berkonsentrasi pada kemungkinan kejadian. Dalam permainan dadu, tidak ada urutan langkah yang mungkin, karena agar langkah tersebut terjadi, dadu harus keluar dengan cara yang benar terlebih dahulu agar sah. Ini adalah masalah umum setiap kali ketidakpastian muncul: kemungkinan berlipat ganda secara drastis, dan menyusun

rencana tindakan yang terperinci menjadi tidak ada gunanya karena dunia mungkin tidak akan berjalan sesuai rencana.

Mungkin Anda pernah berpikir bahwa pemangkasan alfa-beta dapat diterapkan pada pohon permainan dengan simpul peluang. Ternyata bisa. Analisis untuk simpul MIN dan MAX tidak berubah, tetapi kita juga dapat memangkas simpul peluang, dengan sedikit kecerdikan. Perhatikan simpul peluang C pada Gambar 5.11 dan apa yang terjadi pada nilainya saat kita memeriksa dan mengevaluasi anak-anaknya. Apakah mungkin menemukan batas atas nilai C sebelum kita melihat semua anaknya? (Ingatlah bahwa inilah yang dibutuhkan alfa–beta untuk memangkas simpul dan subpohonnya.) Sekilas, mungkin tampak mustahil karena nilai C adalah rata-rata dari nilai anak-anaknya, dan untuk menghitung rata-rata sekumpulan angka, kita harus melihat semua angka. Namun, jika kita menetapkan batasan pada nilai-nilai yang mungkin dari fungsi utilitas, maka kita dapat mencapai batasan untuk rata-rata tanpa melihat setiap angka. Misalnya, katakanlah bahwa semua nilai utilitas berada di antara —2 dan +2; maka nilai simpul daun dibatasi, dan pada gilirannya kita dapat menetapkan batas atas pada nilai simpul peluang tanpa melihat semua anaknya.

Alternatifnya adalah melakukan **simulasi Monte Carlo** untuk mengevaluasi posisi. Mulailah dengan algoritma pencarian alfa–beta (atau lainnya). Dari posisi awal, minta algoritma memainkan ribuan permainan melawan dirinya sendiri, menggunakan lemparan dadu acak. Dalam kasus backgammon, persentase kemenangan yang dihasilkan telah terbukti menjadi perkiraan yang baik dari nilai posisi, bahkan jika algoritma tersebut memiliki heuristik yang tidak sempurna dan hanya mencari beberapa plies. Untuk permainan dengan dadu, jenis simulasi ini disebut **rollout.**

5.6 PERMAINAN YANG DAPAT DIAMATI SEBAGIAN

Catur sering kali digambarkan sebagai perang dalam bentuk miniatur, tetapi tidak memiliki setidaknya satu karakteristik utama dari perang sesungguhnya, yaitu, dapat diamati sebagian. Dalam "kabut perang", keberadaan dan disposisi unit musuh sering kali tidak diketahui hingga terungkap melalui kontak langsung. Akibatnya, peperangan mencakup penggunaan pengintai dan mata-mata untuk mengumpulkan informasi dan penggunaan penyembunyian dan gertakan untuk membingungkan musuh. Permainan yang dapat diamati sebagian memiliki karakteristik ini dan dengan demikian secara kualitatif berbeda dari permainan yang dijelaskan di bagian sebelumnya.

5.6 KRIEGSPIEL: CATUR YANG DAPAT DIAMATI SEBAGIAN

Dalam permainan yang dapat diamati sebagian yang deterministik, ketidakpastian tentang keadaan papan sepenuhnya muncul dari kurangnya akses ke pilihan yang dibuat oleh lawan. Kelas ini mencakup permainan anak-anak seperti Kapal Perang (di mana kapal setiap pemain ditempatkan di lokasi yang tersembunyi dari lawan tetapi tidak bergerak) dan Stratego (di mana lokasi bidak diketahui tetapi jenis bidak disembunyikan). Kita akan menelaah permainan **Kriegspiel**, varian catur yang dapat diamati sebagian, di mana bidak dapat bergerak tetapi sama sekali tidak terlihat oleh lawan.

Aturan Kriegspiel adalah sebagai berikut: Putih dan Hitam masing-masing melihat papan yang hanya berisi bidak mereka sendiri. Seorang wasit, yang dapat melihat semua bidak, mengadili permainan dan secara berkala membuat pengumuman yang didengar oleh kedua pemain. Pada gilirannya, Putih mengusulkan kepada wasit setiap langkah yang akan sah jika tidak ada bidak hitam. Jika langkah tersebut sebenarnya tidak sah (karena bidak hitam), wasit mengumumkan "ilegal." Dalam kasus ini, Putih dapat terus mengusulkan langkah hingga langkah yang sah ditemukan—dan mempelajari lebih lanjut tentang lokasi bidak Hitam dalam prosesnya. Setelah langkah yang sah diajukan, wasit mengumumkan satu atau beberapa hal berikut: "Tangkap di petak X" jika ada tangkapan, dan "Skak oleh D" jika raja hitam dalam keadaan skak, di mana D adalah arah skak, dan dapat berupa salah satu dari "Kuda," "Pangkat," "Berkas," "Diagonal panjang," atau "Diagonal pendek." (Dalam kasus skak yang ditemukan, wasit dapat membuat dua pengumuman "Skak".)

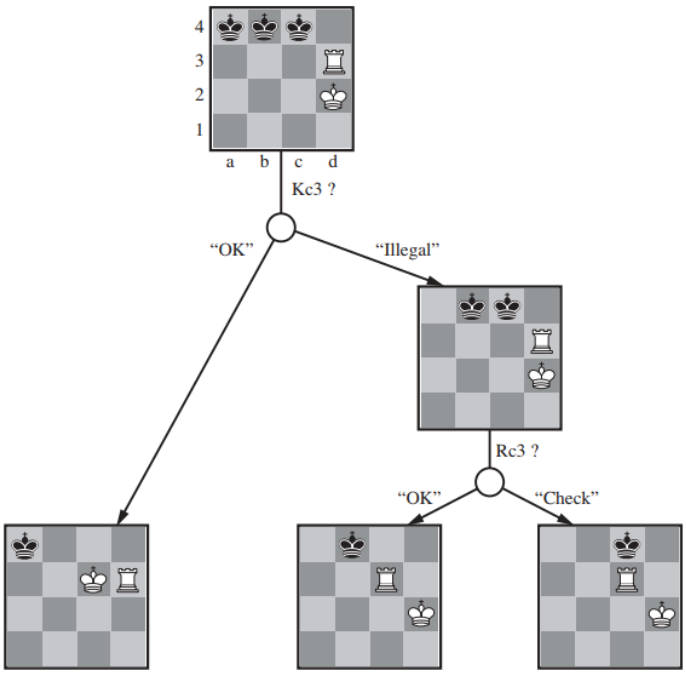
Jika Hitam skakmat atau jalan buntu, wasit mengatakan demikian; jika tidak, giliran Hitam untuk bergerak. Kriegspiel mungkin tampak sangat mustahil, tetapi manusia mengelolanya dengan cukup baik dan program komputer mulai mengejar ketinggalan. Ada baiknya mengingat gagasan tentang **status keyakinan** sebagaimana didefinisikan dalam Bagian 4.4 dan diilustrasikan dalam Gambar 4.14—himpunan semua status papan yang mungkin secara logis berdasarkan riwayat lengkap persepsi hingga saat ini. Awalnya, status keyakinan Putih adalah satuan karena bidak Hitam belum bergerak. Setelah Putih membuat langkah dan Hitam merespons, status keyakinan Putih berisi 20 posisi karena Hitam memiliki 20 balasan untuk setiap langkah Putih. Melacak status keyakinan saat permainan berlangsung adalah masalah estimasi status, yang langkah pembaruannya diberikan dalam Persamaan (4.6). Kita dapat memetakan estimasi status Kriegspiel secara langsung ke kerangka kerja yang dapat diamati sebagian dan nondeterministik dari Bagian 4.4 jika kita menganggap lawan sebagai sumber nondeterminisme; yaitu, HASIL langkah Putih disusun dari hasil (yang dapat diprediksi) dari langkah Putih sendiri dan hasil yang tidak dapat diprediksi yang diberikan oleh balasan Hitam/

Dengan status keyakinan saat ini, Putih mungkin bertanya, "Bisakah saya memenangkan permainan?" Untuk permainan yang dapat diamati sebagian, gagasan tentang **strategi** diubah; alih-alih menentukan langkah yang harus dilakukan untuk setiap kemungkinan langkah yang mungkin dilakukan lawan, kita memerlukan langkah untuk setiap kemungkinan urutan persepsi yang mungkin diterima. Bagi Kriegspiel, strategi kemenangan, atau **skakmat terjamin**, adalah strategi yang, untuk setiap kemungkinan urutan persepsi, mengarah ke skakmat aktual untuk setiap kemungkinan keadaan papan dalam keadaan keyakinan saat ini, terlepas dari bagaimana lawan bergerak. Dengan definisi ini, keadaan keyakinan lawan tidak relevan—strategi harus berhasil bahkan jika lawan dapat melihat semua bidak. Ini sangat menyederhanakan perhitungan. Gambar 5.13 menunjukkan bagian dari skakmat terjamin untuk akhir permainan KRK (raja dan benteng melawan raja). Dalam kasus ini, Hitam hanya memiliki satu bidak (raja), jadi keadaan keyakinan untuk Putih dapat ditunjukkan dalam satu papan dengan menandai setiap kemungkinan posisi raja Hitam.

Algoritma pencarian AND-OR umum dapat diterapkan pada ruang keadaan keyakinan untuk menemukan skakmat terjamin. Algoritma keadaan keyakinan inkremental yang disebutkan dalam bagian itu sering menemukan skakmat tengah permainan hingga kedalaman 9—mungkin jauh melampaui kemampuan pemain manusia. Selain skakmat yang terjamin, Kriegspiel mengakui konsep yang sama sekali baru yang tidak masuk akal dalam permainan yang sepenuhnya dapat diamati: skakmat probabilistik. Skakmat semacam itu masih diperlukan untuk bekerja di setiap keadaan papan dalam keadaan keyakinan; mereka bersifat probabilistik sehubungan dengan pengacakan gerakan pemain yang menang. Untuk mendapatkan ide dasarnya, pertimbangkan masalah menemukan raja hitam yang sendirian hanya dengan menggunakan raja putih. Hanya dengan bergerak secara acak, raja putih pada akhirnya akan bertabrakan dengan raja hitam bahkan jika yang terakhir mencoba menghindari nasib ini, karena Hitam tidak dapat terus menebak gerakan mengelak yang tepat tanpa batas. Dalam terminologi teori probabilitas, deteksi terjadi dengan probabilitas 1. Akhir permainan KBNK—raja, gajah, dan kuda melawan raja—dimenangkan dalam pengertian ini; Putih memberi Hitam urutan pilihan acak yang tak terbatas, untuk salah satunya Hitam akan menebak dengan salah dan mengungkapkan posisinya, yang mengarah ke skakmat.

Akhir permainan KBBK, di sisi lain, dimenangkan dengan probabilitas $1 - \epsilon$. Putih dapat memaksakan kemenangan hanya dengan membiarkan salah satu gajahnya tidak terlindungi selama satu langkah. Jika Hitam kebetulan berada di tempat yang tepat dan menangkap gajah (langkah yang akan kalah jika gajah dilindungi), permainan berakhir seri. Putih dapat memilih untuk melakukan langkah berisiko di beberapa titik yang dipilih secara acak di tengah rangkaian yang sangat panjang, sehingga mengurangi ϵ menjadi konstanta yang sangat kecil, tetapi tidak dapat mengurangi ϵ menjadi nol.

Sangat jarang skakmat yang dijamin atau probabilistik dapat ditemukan dalam kedalaman yang wajar, kecuali di akhir permainan. Terkadang strategi skakmat berhasil untuk beberapa keadaan papan dalam keadaan keyakinan saat ini tetapi tidak untuk yang lain. Mencoba strategi seperti itu mungkin berhasil, yang mengarah pada **skakmat yang tidak disengaja**—tidak disengaja dalam arti Putih tidak dapat mengetahui bahwa itu akan menjadi skakmat—jika bidak Hitam kebetulan berada di tempat yang tepat. (Kebanyakan skakmat dalam permainan antara manusia bersifat kebetulan.) Gagasan ini secara alami mengarah pada pertanyaan tentang seberapa besar kemungkinan suatu strategi akan menang, yang pada gilirannya mengarah pada pertanyaan tentang seberapa besar kemungkinan setiap kondisi papan dalam kondisi keyakinan saat ini merupakan kondisi papan yang sebenarnya.



Gambar 5.13 Bagian dari skakmat yang dijamin dalam permainan akhir KRK, ditunjukkan pada papan yang diperkecil. Dalam keadaan keyakinan awal, raja Hitam berada di salah satu dari tiga lokasi yang mungkin. Dengan kombinasi gerakan penyelidikan, strategi mempersempitnya menjadi satu. Penyelesaian skakmat dibiarkan sebagai latihan.

Kecenderungan pertama seseorang mungkin adalah mengusulkan bahwa semua keadaan papan dalam keadaan keyakinan saat ini memiliki kemungkinan yang sama—tetapi ini tidak mungkin benar. Pertimbangkan, misalnya, keadaan keyakinan Putih setelah langkah pertama Hitam dalam permainan. Menurut definisi (dengan asumsi bahwa Hitam bermain secara optimal), Hitam pasti telah memainkan langkah yang optimal, jadi semua keadaan papan yang dihasilkan dari langkah yang kurang optimal seharusnya diberi probabilitas nol. Argumen ini juga tidak sepenuhnya benar, karena *tujuan setiap pemain bukan hanya untuk memindahkan bidak ke petak yang tepat tetapi juga untuk meminimalkan informasi yang dimiliki lawan tentang lokasinya*. Memainkan strategi "optimal" yang dapat diprediksi akan memberikan informasi kepada lawan.

Oleh karena itu, permainan yang optimal dalam permainan yang dapat diamati sebagian memerlukan kemauan untuk bermain secara acak. (Inilah sebabnya mengapa inspektur kebersihan restoran melakukan kunjungan inspeksi acak.) Ini berarti sesekali memilih langkah yang mungkin tampak "secara intrinsik" lemah—tetapi langkah tersebut memperoleh kekuatan dari sifatnya yang sangat tidak dapat diprediksi, karena lawan tidak mungkin menyiapkan pertahanan apa pun terhadapnya. Dari pertimbangan ini, tampaknya probabilitas yang terkait dengan status papan dalam status keyakinan saat ini hanya dapat dihitung dengan strategi acak yang optimal; pada gilirannya, menghitung strategi tersebut tampaknya memerlukan pengetahuan tentang probabilitas berbagai status papan. Teka-teki ini dapat dipecahkan dengan mengadopsi gagasan teori permainan tentang solusi ekuilibrium,

yang akan kita bahas lebih lanjut di Bab 17. Ekuilibrium menentukan strategi acak yang optimal untuk setiap pemain.

Namun, menghitung ekuilibrium sangatlah mahal, bahkan untuk permainan kecil, dan tidak mungkin dilakukan untuk Kriegspiel. Saat ini, desain algoritme yang efektif untuk permainan Kriegspiel umum merupakan topik penelitian terbuka. Sebagian besar sistem melakukan lookahead dengan kedalaman terbatas di ruang status keyakinan mereka sendiri, mengabaikan status keyakinan lawan. Fungsi evaluasi menyerupai fungsi untuk permainan yang dapat diamati tetapi mencakup komponen untuk ukuran status keyakinan—semakin kecil semakin baik!

Permainan kartu

Permainan kartu menyediakan banyak contoh observabilitas parsial *stokastik*, di mana informasi yang hilang dihasilkan secara acak. Misalnya, dalam banyak permainan, kartu dibagikan secara acak di awal permainan, dengan setiap pemain menerima kartu yang tidak terlihat oleh pemain lain. Permainan tersebut termasuk bridge, whist, hearts, dan beberapa bentuk poker.

Sekilas, mungkin tampak bahwa permainan kartu ini seperti permainan dadu: kartu dibagikan secara acak dan menentukan langkah yang tersedia untuk setiap pemain, tetapi semua "dadu" dilempar di awal! Meskipun analogi ini ternyata salah, ini menunjukkan algoritma yang efektif: pertimbangkan semua kemungkinan pembagian kartu yang tidak terlihat; selesaikan masing-masing seolah-olah itu adalah permainan yang dapat diamati sepenuhnya; dan kemudian pilih langkah yang memiliki hasil terbaik yang dirata-ratakan dari semua pembagian. Misalkan setiap pembagian s terjadi dengan probabilitas $P(s)$; maka langkah yang kita inginkan adalah

$$\operatorname{argmax}_a \sum_s P(s)\ \text{MINIMAX}(\text{RESULT}(s,a))\ . \tag{5.1}$$

Di sini, kami menjalankan MINIMAX yang tepat jika memungkinkan secara komputasi; jika menjalankan $\text{H}-\text{MINIMAX}$. Sekarang, dalam kebanyakan permainan kartu, jumlah kemungkinan pembagian kartu cukup besar. Misalnya, dalam permainan bridge, setiap pemain hanya melihat dua dari empat tangan; ada dua tangan yang tidak terlihat yang masing-masing berisi 13 kartu, jadi jumlah pembagiannya adalah $\binom{26}{13} = 10,400,600.$

Memecahkan bahkan satu pembagian kartu saja cukup sulit, jadi memecahkan sepuluh juta tidak mungkin dilakukan. Sebaliknya, kami menggunakan perkiraan Monte Carlo: alih-alih menjumlahkan semua pembagian kartu, kami mengambil sampel acak dari N pembagian kartu, di mana probabilitas pembagian kartu s muncul dalam sampel sebanding dengan $P(s)$:

$$\operatorname{argmax}_a \frac{1}{N}\sum_{i=1}^{N} \text{MINIMAX}(\text{RESULT}(s_i,a))\ . \tag{5.2}$$

(Perhatikan bahwa $P(s)$ tidak muncul secara eksplisit dalam penjumlahan, karena sampel sudah ditarik sesuai dengan $P(s)$.*)* Saat N tumbuh besar, jumlah atas sampel acak cenderung ke nilai yang tepat, tetapi bahkan untuk N yang cukup kecil —misalnya, 100 hingga 1.000— metode ini memberikan perkiraan yang baik. Ini juga dapat diterapkan pada permainan deterministik seperti Kriegspiel, dengan perkiraan $P(s)$ yang wajar.

Untuk permainan seperti whist dan hearts, di mana tidak ada fase penawaran atau taruhan sebelum permainan dimulai, setiap pembagian akan memiliki kemungkinan yang sama sehingga nilai $P(s)$ semuanya sama. Untuk bridge, permainan didahului oleh fase penawaran di mana setiap tim menunjukkan berapa banyak trik yang diharapkan akan dimenangkan. Karena pemain menawar berdasarkan kartu yang mereka pegang, pemain lain mempelajari lebih lanjut tentang probabilitas setiap pembagian. Mempertimbangkan hal ini dalam memutuskan cara bermain kartu itu sulit, karena alasan yang disebutkan dalam uraian kami tentang Kriegspiel: pemain dapat menawar sedemikian rupa untuk meminimalkan informasi yang disampaikan kepada lawan mereka. Meski begitu, pendekatan ini cukup efektif untuk bridge, seperti yang kami tunjukkan di Bagian 5.7.

Strategi yang dijelaskan dalam Persamaan 5.1 dan 5.2 terkadang disebut sebagai *averaging over clairvoyance* karena mengasumsikan bahwa permainan akan dapat diamati oleh kedua pemain segera setelah langkah pertama. Meskipun menarik secara intuitif, strategi ini dapat menyesatkan seseorang. Pertimbangkan cerita berikut:

> *Hari 1:* Jalan A mengarah ke tumpukan emas; Jalan B mengarah ke percabangan. Ambil percabangan kiri dan Anda akan menemukan tumpukan emas yang lebih besar, tetapi ambil percabangan kanan dan Anda akan tertabrak bus.
>
> *Hari 2:* Jalan A mengarah ke tumpukan emas; Jalan B mengarah ke percabangan. Ambil percabangan kanan dan Anda akan menemukan tumpukan emas yang lebih besar, tetapi ambil percabangan kiri dan Anda akan tertabrak bus.
>
> *Hari 3*: Jalan A mengarah ke tumpukan emas; Jalan B mengarah ke percabangan. Salah satu cabang percabangan mengarah ke tumpukan emas yang lebih besar, tetapi ambil percabangan yang salah dan Anda akan tertabrak bus. Sayangnya Anda tidak tahu percabangan mana yang mana.

Merata-ratakan kewaskitaan mengarah pada penalaran berikut: pada Hari ke-1, *B* adalah pilihan yang tepat; pada Hari ke-2, *B* adalah pilihan yang tepat; pada Hari ke-3, situasinya sama dengan Hari ke-1 atau Hari ke-2, jadi *B* pasti masih menjadi pilihan yang tepat.

Sekarang kita dapat melihat bagaimana rata-rata kewaskitaan gagal: ia tidak mempertimbangkan keadaan keyakinan yang akan dialami agen setelah bertindak. Keadaan keyakinan ketidaktahuan total tidak diinginkan, terutama ketika salah satu kemungkinannya adalah kematian yang pasti. Karena mengasumsikan bahwa setiap status masa depan akan secara otomatis menjadi status pengetahuan yang sempurna, pendekatan tersebut tidak akan pernah memilih tindakan yang *mengumpulkan informasi* (seperti langkah pertama pada Gambar 5.13); juga tidak akan memilih tindakan yang menyembunyikan informasi dari lawan atau memberikan informasi kepada mitra karena mengasumsikan bahwa mereka sudah

mengetahui informasi tersebut; dan tidak akan pernah **menggertak** dalam poker, karena mengasumsikan lawan dapat melihat kartunya.

5.7 PROGRAM PERMAINAN TERKINI

Pada tahun 1965, matematikawan Rusia Alexander Kronrod menyebut catur sebagai "lalat buah kecerdasan buatan." John McCarthy tidak setuju: sementara ahli genetika menggunakan lalat buah untuk membuat penemuan yang berlaku untuk biologi secara lebih luas, AI telah menggunakan catur untuk melakukan hal yang setara dengan mengembangbiakkan lalat buah yang sangat cepat. Mungkin analogi yang lebih baik adalah bahwa catur bagi AI seperti balap motor Grand Prix bagi industri mobil: program permainan canggih adalah mesin yang sangat cepat dan sangat optimal yang menggabungkan kemajuan teknik terbaru, tetapi tidak banyak digunakan untuk berbelanja atau berkendara di luar jalan raya. Meskipun demikian, balapan dan permainan menghasilkan kegembiraan dan aliran inovasi yang terus-menerus yang telah diadopsi oleh masyarakat luas. Di bagian ini kita melihat apa yang diperlukan untuk menjadi yang teratas dalam berbagai permainan

Catur: Program catur DEEP BLUE IBM, yang sekarang sudah tidak digunakan lagi, terkenal karena mengalahkan juara dunia Garry Kasparov dalam pertandingan eksibisi yang dipublikasikan secara luas. Deep Blue berjalan pada komputer paralel dengan 30 prosesor IBM RS/6000 yang melakukan pencarian alfa-beta. Bagian yang unik adalah konfigurasi 480 prosesor catur VLSI kustom yang melakukan pembuatan langkah dan pengurutan langkah untuk beberapa level terakhir dari pohon, dan mengevaluasi simpul daun. Deep Blue mencari hingga 30 miliar posisi per langkah, mencapai kedalaman 14 secara rutin. Kunci keberhasilannya tampaknya adalah kemampuannya untuk menghasilkan ekstensi tunggal di luar batas kedalaman untuk garis-garis gerakan pemaksaan/paksaan yang cukup menarik. Dalam beberapa kasus pencarian mencapai kedalaman 40 plies. Fungsi evaluasi memiliki lebih dari 8000 fitur, banyak di antaranya menggambarkan pola buah catur yang sangat spesifik.

Sebuah "buku pembukaan" sekitar 4000 posisi digunakan, serta basis data 700.000 permainan grandmaster yang darinya rekomendasi konsensus dapat diekstraksi. Sistem ini juga menggunakan basis data akhir permainan yang besar dari posisi terpecahkan yang berisi semua posisi dengan lima buah catur dan banyak dengan enam buah catur. Basis data ini memiliki efek memperluas kedalaman pencarian efektif secara substansial, yang memungkinkan Deep Blue bermain dengan sempurna dalam beberapa kasus bahkan ketika masih banyak langkah lagi untuk mencapai skakmat.

Keberhasilan DEEP BLUE memperkuat kepercayaan yang berlaku umum bahwa kemajuan dalam permainan komputer terutama berasal dari perangkat keras yang semakin canggih—pandangan yang didorong oleh IBM. Namun, peningkatan algoritme telah memungkinkan program yang berjalan pada PC standar untuk memenangkan Kejuaraan Catur Komputer Dunia. Berbagai heuristik pemangkasan digunakan untuk mengurangi faktor percabangan efektif menjadi kurang dari 3 (dibandingkan dengan faktor percabangan aktual sekitar 35). Yang terpenting di antaranya adalah heuristik **langkah nol**, yang menghasilkan batas bawah yang baik pada nilai posisi, menggunakan pencarian dangkal di mana lawan dapat

bergerak dua kali di awal. Batas bawah ini sering kali memungkinkan pemangkasan alfa-beta tanpa biaya pencarian mendalam penuh. Yang juga penting adalah **pemangkasan sia-sia**, yang membantu memutuskan terlebih dahulu langkah mana yang akan menyebabkan pemutusan beta pada simpul penerus.

HYDRA dapat dilihat sebagai penerus DEEP BLUE. HYDRA berjalan pada kluster 64 prosesor dengan 1 gigabyte per prosesor dan dengan perangkat keras khusus dalam bentuk chip FPGA (Field Programmable Gate Array). HYDRA mencapai 200 juta evaluasi per detik, hampir sama dengan Deep Blue, tetapi HYDRA mencapai 18 plies lebih dalam daripada hanya 14 karena penggunaan heuristik null move dan forward prunes yang agresif. RYBKA, pemenang Kejuaraan Catur Komputer Dunia 2008 dan 2009, dianggap sebagai pemain komputer terkuat saat ini. Ia menggunakan prosesor Intel Xeon 8-core 3,2 GHz yang siap pakai, tetapi sedikit yang diketahui tentang desain programnya. Keunggulan utama RYBKA tampaknya adalah fungsi evaluasinya, yang telah disetel oleh pengembang utamanya, Master Internasional Vasik Rajlich, dan sedikitnya tiga grandmaster lainnya.

Pertandingan terkini menunjukkan bahwa program catur komputer teratas telah mengungguli semua pesaing manusia. (Lihat catatan sejarah untuk detailnya.)

Dam: Jonathan Schaeffer dan rekan-rekannya mengembangkan CHINOOK, yang berjalan pada PC biasa dan menggunakan pencarian alfa-beta. Chinook mengalahkan juara manusia yang sudah lama dalam pertandingan singkat pada tahun 1990, dan sejak tahun 2007 CHINOOK telah mampu bermain dengan sempurna dengan menggunakan pencarian alfa-beta yang dikombinasikan dengan basis data 39 triliun posisi akhir permainan.

Othello, yang juga disebut Reversi, mungkin lebih populer sebagai permainan komputer daripada permainan papan. Permainan ini memiliki ruang pencarian yang lebih kecil daripada catur, biasanya 5 hingga 15 langkah legal, tetapi keahlian evaluasi harus dikembangkan dari awal. Pada tahun 1997, program LOGISTELLO mengalahkan juara dunia manusia, Takeshi Murakami, dengan enam permainan tanpa gol. Secara umum diakui bahwa manusia tidak sebanding dengan komputer di Othello.

Backgammon: Bagian 5.5 menjelaskan mengapa penyertaan ketidakpastian dari lemparan dadu membuat pencarian mendalam menjadi kemewahan yang mahal. Sebagian besar pekerjaan pada backgammon telah digunakan untuk meningkatkan fungsi evaluasi. Gerry Tesauro (1992) menggabungkan pembelajaran penguatan dengan jaringan saraf untuk mengembangkan evaluator yang sangat akurat yang digunakan dengan pencarian hingga kedalaman 2 atau 3. Setelah memainkan lebih dari satu juta permainan pelatihan melawan dirinya sendiri, program Tesauro, TD-GAMMON, bersaing dengan pemain manusia teratas. Pendapat program tentang langkah pembukaan permainan dalam beberapa kasus telah mengubah kebijaksanaan yang diterima secara radikal.

Go adalah permainan papan paling populer di Asia. Karena papannya berukuran 19 × 19 dan langkah diperbolehkan masuk ke (hampir) setiap kotak kosong, faktor percabangan dimulai pada 361, yang terlalu menakutkan untuk metode pencarian alfa-beta biasa. Selain itu, sulit untuk menulis fungsi evaluasi karena kendali wilayah sering kali sangat tidak dapat diprediksi hingga akhir permainan. Oleh karena itu, program-program teratas, seperti MOGO,

menghindari pencarian alfa-beta dan sebagai gantinya menggunakan peluncuran Monte Carlo. Triknya adalah memutuskan langkah apa yang akan dilakukan selama peluncuran. Tidak ada pemangkasan yang agresif; semua langkah memungkinkan. Metode UCT (batas kepercayaan atas pada pohon) bekerja dengan membuat langkah acak dalam beberapa iterasi pertama, dan seiring waktu memandu proses pengambilan sampel untuk lebih memilih langkah yang telah menghasilkan kemenangan dalam sampel sebelumnya. Beberapa trik ditambahkan, termasuk aturan *berbasis pengetahuan* yang menyarankan gerakan tertentu setiap kali pola tertentu terdeteksi dan *pencarian lokal terbatas* untuk memutuskan pertanyaan taktis. Beberapa program juga menyertakan teknik khusus dari **teori permainan kombinatorial** untuk menganalisis akhir permainan. Teknik-teknik ini menguraikan posisi menjadi subposisi yang dapat dianalisis secara terpisah dan kemudian digabungkan (Berlekamp dan Wolfe, 1994; Muller, 2003). Solusi optimal yang diperoleh dengan cara ini telah mengejutkan banyak pemain Go profesional, yang mengira mereka telah bermain secara optimal selama ini. Program Go saat ini bermain di tingkat master pada papan 9 × 9 yang diperkecil, tetapi masih pada tingkat amatir tingkat lanjut pada papan penuh.

Bridge adalah permainan kartu dengan informasi yang tidak sempurna: kartu pemain disembunyikan dari pemain lain. Bridge juga merupakan permainan multipemain dengan empat pemain, bukan dua, meskipun para pemain dipasangkan menjadi dua tim. Seperti pada Bagian 5.6, permainan optimal dalam permainan yang dapat diamati sebagian seperti bridge dapat mencakup elemen pengumpulan informasi, komunikasi, dan pertimbangan probabilitas yang cermat. Banyak dari teknik ini digunakan dalam program Bridge Baron, yang memenangkan kejuaraan bridge komputer tahun 1997. Meskipun tidak dimainkan secara optimal, Bridge Baron adalah salah satu dari sedikit sistem permainan yang berhasil menggunakan rencana hierarkis yang rumit yang melibatkan ide-ide tingkat tinggi, seperti **finessing** dan **squeezing**, yang sudah dikenal oleh pemain bridge.

Program GIB memenangkan kejuaraan bridge komputer tahun 2000 dengan cukup meyakinkan menggunakan metode Monte Carlo. Sejak saat itu, program pemenang lainnya telah mengikuti jejak GIB. Inovasi utama GIB adalah menggunakan **generalisasi berbasis penjelasan** untuk menghitung dan menyimpan aturan umum untuk permainan optimal dalam berbagai kelas situasi standar daripada mengevaluasi setiap situasi secara individual.

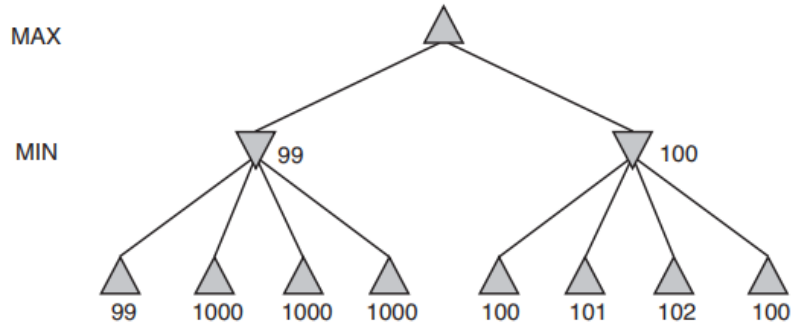
Misalnya, dalam situasi di mana satu pemain memiliki kartu $A - K - Q - J - 4 - 3 - 2$ dari satu jenis dan pemain lain memiliki $10 - 9 - 8 - 7 - 6 - 5$, ada $7 \times 6 = 42$ cara pemain pertama dapat memimpin dari jenis itu dan pemain kedua dapat mengikutinya. Namun, GIB memperlakukan situasi ini hanya sebagai dua: pemain pertama dapat memimpin kartu tinggi atau kartu rendah; kartu yang dimainkan secara tepat tidak menjadi masalah. Dengan pengoptimalan ini (dan beberapa lainnya), GIB dapat memecahkan kesepakatan 52 kartu yang dapat diamati sepenuhnya **tepat** dalam waktu sekitar satu detik. Ketepatan taktis GIB menutupi ketidakmampuannya untuk bernalar tentang informasi. Ia finis di urutan ke-12 dari 35 peserta dalam kontes par (yang hanya melibatkan permainan kartu, bukan penawaran) pada kejuaraan dunia manusia tahun 1998, jauh melampaui ekspektasi banyak pakar manusia.

Ada beberapa alasan mengapa GIB bermain pada level pakar dengan simulasi Monte Carlo, sedangkan program Kriegspiel tidak. Pertama, evaluasi GIB terhadap versi permainan yang dapat diamati sepenuhnya bersifat tepat, menelusuri pohon permainan secara lengkap, sedangkan program Kriegspiel mengandalkan heuristik yang tidak tepat. Namun, yang jauh lebih penting adalah fakta bahwa dalam permainan bridge, sebagian besar ketidakpastian dalam informasi yang dapat diamati sebagian berasal dari keacakan pembagian kartu, bukan dari permainan lawan yang saling bertentangan. Simulasi Monte Carlo menangani keacakan dengan baik, tetapi tidak selalu menangani strategi dengan baik, terutama ketika strategi tersebut melibatkan nilai informasi.

Scrabble: Kebanyakan orang menganggap bagian tersulit dari Scrabble adalah menemukan kata-kata yang bagus, tetapi berdasarkan kamus resmi, ternyata cukup mudah untuk memprogram generator langkah untuk menemukan langkah dengan skor tertinggi (Gordon, 1994). Namun, itu tidak berarti permainannya sudah selesai: hanya mengambil langkah dengan skor tertinggi setiap giliran akan menghasilkan pemain yang bagus tetapi tidak ahli. Masalahnya adalah Scrabble sebagian dapat diamati dan bersifat stokastik: Anda tidak tahu huruf apa yang dimiliki pemain lain atau huruf apa yang akan Anda ambil selanjutnya. Jadi, bermain Scrabble dengan baik menggabungkan kesulitan backgammon dan bridge. Meskipun demikian, pada tahun 2006, program QUACKLE mengalahkan mantan juara dunia, David Boys, dengan skor 3–2.

5.8 PENDEKATAN ALTERNATIF

Karena menghitung keputusan optimal dalam permainan sulit dilakukan dalam banyak kasus, semua algoritme harus membuat beberapa asumsi dan perkiraan. Pendekatan standar, yang didasarkan pada mini-max, fungsi evaluasi, dan alpha–beta, hanyalah salah satu cara untuk melakukannya. Mungkin karena pendekatan ini telah lama dikerjakan, pendekatan standar mendominasi metode lain dalam permainan turnamen. Beberapa orang percaya bahwa hal ini telah menyebabkan permainan menjadi terpisah dari arus utama penelitian AI: pendekatan standar tidak lagi menyediakan banyak ruang untuk wawasan baru tentang pertanyaan umum tentang pengambilan keputusan. Di bagian ini, kita akan melihat alternatifnya.



Gambar 5.14 Pohon permainan dua lapis yang minimax heuristiknya mungkin menghasilkan kesalahan.

Pertama, mari kita pertimbangkan minimax heuristik. Ia memilih langkah optimal dalam pohon pencarian yang diberikan *asalkan evaluasi simpul daun benar-benar tepat*. Kenyataannya, evaluasi biasanya adalah estimasi kasar dari nilai posisi dan dapat dianggap memiliki galat besar yang terkait dengannya. Gambar 5.14 menunjukkan pohon permainan dua lapis yang minimax-nya menyarankan untuk mengambil cabang kanan karena $100 > 99$. Itu adalah langkah yang benar jika semua evaluasinya benar. Namun tentu saja fungsi evaluasi hanya perkiraan. Misalkan evaluasi setiap simpul memiliki galat yang independen dari simpul lain dan didistribusikan secara acak dengan mean nol dan deviasi standar σ. Kemudian ketika $\sigma = 5$, cabang kiri sebenarnya lebih baik 71% dari waktu, dan 58% dari waktu ketika $\sigma = 2$. Intuisi di balik ini adalah bahwa cabang kanan memiliki empat simpul yang mendekati 99; jika kesalahan dalam evaluasi salah satu dari keempatnya membuat cabang kanan turun di bawah 99, maka cabang kiri lebih baik.

Pada kenyataannya, keadaan sebenarnya lebih buruk dari ini karena kesalahan dalam fungsi evaluasi tidak independen. Jika kita salah dalam satu simpul, kemungkinan besar simpul di dekatnya di pohon juga akan salah. Fakta bahwa simpul berlabel 99 memiliki saudara kandung berlabel 1000 menunjukkan bahwa sebenarnya simpul itu mungkin memiliki nilai benar yang lebih tinggi. Kita dapat menggunakan fungsi evaluasi yang mengembalikan distribusi probabilitas atas nilai-nilai yang mungkin, tetapi sulit untuk menggabungkan distribusi ini dengan benar, karena kita tidak akan memiliki model yang baik dari ketergantungan yang sangat kuat yang ada di antara nilai-nilai simpul saudara kandung.

Selanjutnya, kita pertimbangkan algoritma pencarian yang menghasilkan pohon. Tujuan perancang algoritma adalah untuk menentukan perhitungan yang berjalan cepat dan menghasilkan langkah yang baik. Algoritma alfa-beta dirancang tidak hanya untuk memilih langkah yang baik tetapi juga untuk menghitung batas pada nilai-nilai semua langkah yang sah. Untuk melihat mengapa informasi tambahan ini tidak diperlukan, pertimbangkan posisi di mana hanya ada satu langkah yang sah. Pencarian alfa-beta masih akan menghasilkan dan mengevaluasi pohon pencarian yang besar, yang memberi tahu kita bahwa satu-satunya langkah adalah langkah terbaik dan memberinya nilai. Namun, karena kita harus tetap melakukan langkah tersebut, mengetahui nilai langkah tersebut tidak ada gunanya. Demikian pula, jika ada satu langkah yang jelas bagus dan beberapa langkah yang sah tetapi menyebabkan kekalahan cepat, kita tidak ingin alfa-beta membuang-buang waktu untuk menentukan nilai yang tepat untuk satu langkah yang bagus. Lebih baik melakukan langkah dengan cepat dan menyimpan waktu untuk nanti. Ini mengarah pada gagasan tentang utilitas perluasan simpul. Algoritme pencarian yang baik harus memilih perluasan simpul dengan utilitas tinggi—yaitu, yang cenderung mengarah pada penemuan langkah yang jauh lebih baik. Jika tidak ada perluasan simpul yang utilitasnya lebih tinggi daripada biayanya (dalam hal waktu), maka algoritme harus berhenti mencari dan melakukan langkah. Perhatikan bahwa ini berfungsi tidak hanya untuk situasi favorit yang jelas tetapi juga untuk kasus gerakan simetris, yang tidak ada jumlah pencarian yang akan menunjukkan bahwa satu gerakan lebih baik daripada yang lain.

Jenis penalaran tentang komputasi apa yang harus dilakukan ini disebut **metareasoning** (penalaran tentang penalaran). Ini berlaku tidak hanya untuk permainan, tetapi untuk semua jenis penalaran. Semua perhitungan dilakukan untuk mencoba mencapai keputusan yang lebih baik, semuanya memiliki biaya, dan semuanya memiliki kemungkinan menghasilkan peningkatan tertentu dalam kualitas keputusan. Alfa-beta menggabungkan jenis metareasoning yang paling sederhana, yaitu, teorema yang menyatakan bahwa cabang-cabang tertentu dari pohon dapat diabaikan tanpa kehilangan apa pun. Hal ini mungkin dilakukan dengan jauh lebih baik.

Terakhir, mari kita periksa kembali sifat pencarian itu sendiri. Algoritme untuk pencarian heuristik dan untuk permainan menghasilkan urutan keadaan konkret, dimulai dari keadaan awal dan kemudian menerapkan fungsi evaluasi. Jelas, ini bukan cara manusia bermain game. Dalam catur, seseorang sering memiliki tujuan tertentu dalam pikiran—misalnya, menjebak ratu lawan—dan dapat menggunakan tujuan ini untuk *secara selektif* menghasilkan rencana yang masuk akal untuk mencapainya. Penalaran atau perencanaan yang diarahkan pada tujuan semacam ini terkadang menghilangkan pencarian kombinatorial sama sekali. PARADISE karya David Wilkins (1980) adalah satu-satunya program yang berhasil menggunakan penalaran yang diarahkan pada tujuan dalam catur: program tersebut mampu memecahkan beberapa masalah catur yang memerlukan kombinasi 18 langkah. Hingga saat ini belum ada pemahaman yang baik tentang cara *menggabungkan* kedua jenis algoritme tersebut menjadi sistem yang tangguh dan efisien, meskipun Bridge Baron mungkin merupakan langkah ke arah yang benar. Sistem yang terintegrasi sepenuhnya akan menjadi pencapaian yang signifikan tidak hanya untuk penelitian permainan tetapi juga untuk penelitian AI secara umum, karena sistem tersebut akan menjadi dasar yang baik untuk agen cerdas umum.

5.9 RINGKASAN

Kami telah melihat berbagai permainan untuk memahami apa arti permainan optimal dan untuk memahami cara bermain dengan baik dalam praktik. Gagasan yang paling penting adalah sebagai berikut:

- Suatu permainan dapat didefinisikan berdasarkan **keadaan awal** (bagaimana papan disiapkan), **tindakan** hukum di setiap keadaan, **hasil** dari setiap tindakan, **uji terminal** (yang menyatakan kapan permainan berakhir), dan **fungsi utilitas** yang berlaku untuk keadaan terminal.
- Dalam permainan zero-sum dua pemain dengan **informasi sempurna**, algoritme **minimax** dapat memilih langkah optimal dengan penghitungan kedalaman pertama dari pohon permainan.
- Algoritme pencarian **alfa–beta** menghitung langkah optimal yang sama dengan minimax, tetapi mencapai efisiensi yang jauh lebih besar dengan menghilangkan subpohon yang terbukti tidak relevan.

- Biasanya, tidak mungkin untuk mempertimbangkan seluruh pohon permainan (bahkan dengan alfa–beta), jadi kami perlu menghentikan pencarian di beberapa titik dan menerapkan **fungsi evaluasi** heuristik yang memperkirakan utilitas suatu keadaan.
- Banyak program permainan menghitung terlebih dahulu tabel langkah terbaik di awal dan akhir permainan sehingga mereka dapat mencari langkah tersebut alih-alih mencarinya.
- Permainan untung-untungan dapat ditangani dengan perluasan algoritma minimax yang mengevaluasi **simpul peluang** dengan mengambil utilitas rata-rata dari semua anaknya, yang dibobot dengan probabilitas setiap anaknya.
- Permainan optimal dalam permainan dengan **informasi yang tidak sempurna**, seperti Kriegspiel dan bridge, memerlukan penalaran tentang **status keyakinan** saat ini dan masa depan setiap pemain. Perkiraan sederhana dapat diperoleh dengan merata-ratakan nilai suatu tindakan pada setiap kemungkinan konfigurasi informasi yang hilang.
- Program bahkan telah mengalahkan pemain manusia yang hebat dalam permainan seperti catur, dam, dan Othello. Manusia tetap unggul dalam beberapa permainan dengan informasi yang tidak sempurna, seperti poker, bridge, dan Kriegspiel, dan dalam permainan dengan faktor percabangan yang sangat besar dan sedikit pengetahuan heuristik yang baik, seperti Go.

BAB 6
MASALAH KEPUASAN TERHADAP KENDALA

Bab 3 dan 4 mengeksplorasi gagasan bahwa masalah dapat dipecahkan dengan mencari dalam ruang status. Status ini dapat dievaluasi dengan heuristik khusus domain dan diuji untuk melihat apakah status tersebut merupakan status tujuan. Namun, dari sudut pandang algoritme pencarian, setiap status bersifat atomik, atau tidak dapat dibagi—kotak hitam tanpa struktur internal.

Bab ini menjelaskan cara untuk memecahkan berbagai macam masalah dengan lebih efisien. Kami menggunakan **representasi terfaktor** untuk setiap status: sekumpulan variabel, yang masing-masing memiliki nilai. Masalah terpecahkan ketika setiap variabel memiliki nilai yang memenuhi semua kendala pada variabel tersebut. Masalah yang dijelaskan dengan cara ini disebut **masalah kepuasan kendala**, atau CSP.

Algoritme pencarian CSP memanfaatkan struktur status dan menggunakan heuristik tujuan umum daripada heuristik khusus masalah untuk memungkinkan penyelesaian masalah yang kompleks. Ide utamanya adalah menghilangkan sebagian besar ruang pencarian sekaligus dengan mengidentifikasi kombinasi variabel/nilai yang melanggar batasan.

6.1 MENDEFINISIKAN MASALAH PEMENUHAN KENDALA

Masalah pemenuhan kendala terdiri dari tiga komponen, X, D, dan C:

X adalah sekumpulan variabel, $\{X_1, \dots, X_n\}$.

D adalah sekumpulan domain, $\{D_1, \dots, D_n\}$satu untuk setiap variabel.

C adalah sekumpulan kendala yang menentukan kombinasi nilai yang diizinkan.

Setiap domain D_i terdiri dari sekumpulan nilai yang diizinkan, $\{v_1, \dots, v_n\}$ untuk variabel X_i. Setiap kendala C_i terdiri dari pasangan ⟨*scope, rel*⟩, di mana scope adalah tuple variabel yang berpartisipasi dalam kendala dan *rel* adalah relasi yang menentukan nilai yang dapat diambil oleh variabel tersebut. Relasi dapat direpresentasikan sebagai daftar eksplisit semua tuple nilai yang memenuhi kendala, atau sebagai relasi abstrak yang mendukung dua operasi: menguji apakah suatu tuple merupakan anggota relasi dan menghitung anggota relasi. Misalnya, jika X_1 dan X_2 keduanya memiliki domain $\{A, B\}$, maka kendala yang menyatakan bahwa kedua variabel harus memiliki nilai yang berbeda dapat ditulis sebagai $\langle(X_1, X_2), [(A, B), (B, A)]\rangle$ atau sebagai $\langle(X_1, X_2), X_1 \neq X_2\rangle$.

Untuk menyelesaikan CSP, kita perlu menentukan ruang keadaan dan pengertian solusi. Setiap status dalam CSP didefinisikan dengan **penugasan** nilai untuk beberapa atau semua variabel $\{X_i = v_i, X_j = v_j, \dots\}$ Penugasan yang tidak melanggar batasan apa pun disebut penugasan **konsisten** atau legal. **Penugasan lengkap** adalah penugasan di mana setiap variabel ditugaskan, dan **solusi** untuk CSP adalah penugasan konsisten dan lengkap. **Penugasan parsial** adalah penugasan yang hanya menetapkan nilai untuk beberapa variabel.

Pewarnaan peta

Misalkan, setelah bosan dengan Rumania, kita melihat peta Australia yang menunjukkan setiap negara bagian dan teritorinya (Gambar 6.1(a)). Kita diberi tugas untuk

mewarnai setiap wilayah dengan warna merah, hijau, atau biru sedemikian rupa sehingga tidak ada wilayah tetangga yang memiliki warna yang sama. Untuk merumuskan ini sebagai CSP, kita mendefinisikan variabel sebagai wilayah

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

Domain setiap variabel adalah himpunan $D_i = (Red, green, blue)$. Kendala mengharuskan daerah tetangga memiliki warna yang berbeda. Karena ada sembilan tempat yang berbatasan dengan daerah lain, maka ada sembilan kendala:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,$$
$$WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Di sini kita menggunakan singkatan; $SA \neq WA$ adalah jalan pintas untuk $\langle(SA, WA), SA \neq WA\rangle$, di mana $SA \neq WA$ dapat disebutkan secara lengkap sebagai

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$$

Ada banyak kemungkinan solusi untuk masalah ini, seperti

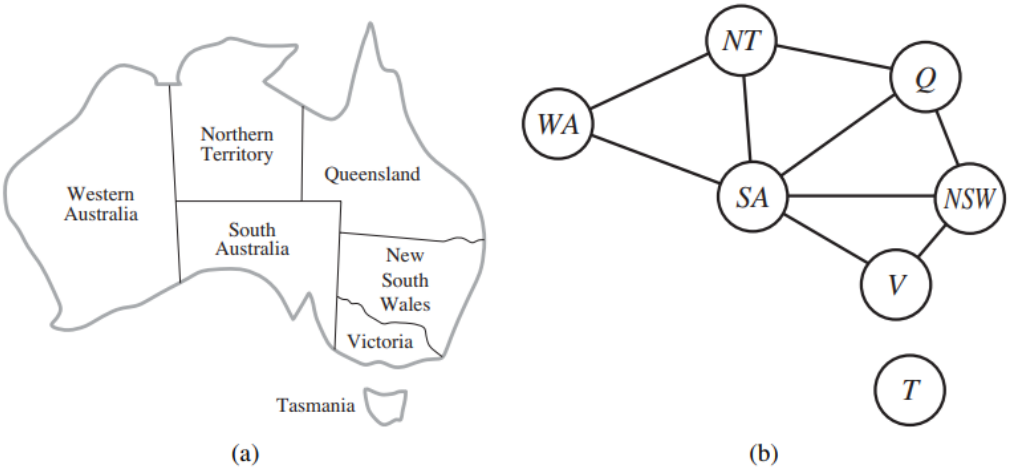
$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}$$

Akan membantu jika memvisualisasikan CSP sebagai **grafik kendala**, seperti yang ditunjukkan pada Gambar 6.1(b). Node grafik sesuai dengan variabel masalah, dan tautan menghubungkan dua variabel yang berpartisipasi dalam kendala.

Mengapa merumuskan masalah sebagai CSP? Salah satu alasannya adalah bahwa CSP menghasilkan representasi alami untuk berbagai macam masalah; jika Anda sudah memiliki sistem penyelesaian CSP, sering kali lebih mudah untuk menyelesaikan masalah dengan menggunakannya daripada merancang solusi khusus menggunakan teknik pencarian lain. Selain itu, penyelesai CSP dapat lebih cepat daripada penelusur ruang-keadaan karena penyelesai CSP dapat dengan cepat menghilangkan bagian besar dari ruang pencarian. Misalnya, setelah kita memilih $\{SA = biru\}$ dalam masalah Australia, kita dapat menyimpulkan bahwa tidak satu pun dari lima variabel tetangga dapat mengambil nilai biru. Tanpa memanfaatkan propagasi kendala, prosedur pencarian harus mempertimbangkan $3^5 = 243$ penugasan untuk lima variabel tetangga; dengan propagasi kendala, kita tidak perlu mempertimbangkan *biru* sebagai nilai, jadi kita hanya memiliki $2^5 = 32$ penugasan untuk dilihat, pengurangan sebesar 87%.

Dalam pencarian ruang-keadaan biasa, kita hanya dapat bertanya: apakah keadaan khusus ini merupakan tujuan? Tidak? Bagaimana dengan yang ini? Dengan CSP, setelah kita mengetahui bahwa penugasan parsial bukanlah solusi, kita dapat segera membuang penyempurnaan lebih lanjut dari penugasan parsial. Lebih jauh, kita dapat melihat mengapa

penugasan tersebut bukanlah solusi—kita melihat variabel mana yang melanggar batasan—sehingga kita dapat memfokuskan perhatian pada variabel yang penting. Akibatnya, banyak masalah yang sulit dipecahkan untuk pencarian ruang keadaan biasa dapat diselesaikan dengan cepat jika diformulasikan sebagai CSP.



Gambar 6.1 (a) Negara bagian dan teritori utama Australia. Pewarnaan peta ini dapat dilihat sebagai masalah pemenuhan kendala (constraint fulfillment problem/CSP). Tujuannya adalah untuk menetapkan warna pada setiap wilayah sehingga tidak ada wilayah tetangga yang memiliki warna yang sama. (b) Masalah pewarnaan peta direpresentasikan sebagai grafik kendala.

Penjadwalan bengkel

Pabrik memiliki masalah dalam menjadwalkan pekerjaan selama sehari, yang tunduk pada berbagai kendala. Dalam praktiknya, banyak dari masalah ini dipecahkan dengan teknik CSP. Pertimbangkan masalah penjadwalan perakitan mobil. Seluruh pekerjaan terdiri dari tugas-tugas, dan kita dapat memodelkan setiap tugas sebagai variabel, di mana nilai setiap variabel adalah waktu dimulainya tugas, yang dinyatakan sebagai bilangan bulat menit. Kendala dapat menyatakan bahwa satu tugas harus dilakukan sebelum tugas lainnya—misalnya, roda harus dipasang sebelum penutup roda dipasang—dan hanya beberapa tugas yang dapat dilakukan sekaligus. Kendala juga dapat menentukan bahwa suatu tugas memerlukan waktu tertentu untuk diselesaikan.

Kita pertimbangkan bagian kecil dari perakitan mobil, yang terdiri dari 15 tugas: memasang as roda (depan dan belakang), memasang keempat roda (kanan dan kiri, depan dan belakang), mengencangkan mur untuk setiap roda, memasang penutup roda, dan memeriksa perakitan akhir. Kita dapat merepresentasikan tugas dengan 15 variabel:

$$X = \{ Axle_F, Axle_B, Whell_{RF}, Whell_{LF}, Whell_{RB}, Whell_{LB}, Nuts_{RF}, Nuts_{LF} Nuts_{RB} \\ Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect \}.$$

Nilai setiap variabel adalah waktu dimulainya tugas. Selanjutnya, kami merepresentasikan batasan **prioritas** antara tugas-tugas individual. Setiap kali tugas T_1 harus

terjadi sebelum tugas T_2, dan tugas T_2 membutuhkan durasi d_1 untuk diselesaikan, kami menambahkan batasan aritmatika dalam bentuk

$$T_1 + d_1 \leq T_2$$

Dalam contoh kita, as harus sudah berada di tempatnya sebelum roda dipasang, dan butuh waktu 10 menit untuk memasang as, jadi kita menulis

$$Axle_F + 10 \leq Axle_{RF}; \;\; Axle_F + 10 \leq Axle_{LF}$$
$$Axle_B + 10 \leq Axle_{RB}; \;\; Axle_B + 10 \leq Axle_{LB}$$

Selanjutnya kita katakan bahwa, untuk setiap roda, kita harus memasang roda (yang memakan waktu 1 menit), lalu mengencangkan mur (2 menit), dan terakhir memasang penutup roda (1 menit, tetapi belum terwakili):

$$Axle_{RF} + 10 \leq Axle_{RF}; \;\; Axle_{RF} + 10 \leq Axle_{RF}$$
$$Axle_{LF} + 10 \leq Axle_{LF}; \;\; Axle_{LF} + 10 \leq Axle_{LF}$$
$$Axle_{RB} + 10 \leq Axle_{RB}; \;\; Axle_{RB} + 10 \leq Axle_{RB}$$
$$Axle_{LB} + 10 \leq Axle_{LB}; \;\; Axle_{LB} + 10 \leq Axle_{LB}$$

Misalkan kita memiliki empat pekerja untuk memasang roda, tetapi mereka harus berbagi satu alat yang membantu menempatkan as roda pada tempatnya. Kita memerlukan batasan disjungtif untuk mengatakan bahwa $Axle_F$ dan $Axle_B$ tidak boleh tumpang tindih dalam waktu; salah satu harus lebih dulu atau yang lain yang harus lebih dulu:

$$(Axle_F + 10 \leq Axle_B) \quad or \quad (Axle_F + 10 \leq Axle_F).$$

Ini tampak seperti kendala yang lebih rumit, menggabungkan aritmatika dan logika. Namun, kendala ini masih dapat disederhanakan menjadi sekumpulan pasangan nilai yang dapat diambil oleh $Axle_F$ dan $Axle_B$. Kita juga perlu menegaskan bahwa pemeriksaan dilakukan terakhir dan memakan waktu 3 menit. Untuk setiap variabel kecuali Inspect, kita tambahkan kendala dalam bentuk $X + d_X \leq Inspect$. Terakhir, misalkan ada persyaratan untuk menyelesaikan seluruh perakitan dalam waktu 30 menit. Kita dapat mencapainya dengan membatasi domain semua variabel:

$$D_i = \{1, 2, 3, \ldots .27\}.$$

Masalah khusus ini mudah dipecahkan, tetapi CSPs telah diterapkan pada masalah penjadwalan job-shop seperti ini dengan ribuan variabel. Dalam beberapa kasus, terdapat kendala rumit yang sulit ditentukan dalam formalisme CSP, dan teknik perencanaan yang lebih maju digunakan.

Variasi formalisme CSP

Jenis CSP yang paling sederhana melibatkan variabel yang memiliki domain diskrit dan terbatas. Masalah pewarnaan peta dan penjadwalan dengan batas waktu keduanya termasuk dalam jenis ini. Masalah 8 ratu yang dijelaskan dalam Bab 3 juga dapat dilihat sebagai CSP domain terbatas, di mana variabel $Q_1, \ldots, Q_8$adalah posisi setiap ratu di kolom $1, \ldots, 8$ dan setiap variabel memiliki domain $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Domain diskrit dapat tak terbatas, seperti himpunan bilangan bulat atau string. (Jika kita tidak menetapkan batas waktu pada masalah penjadwalan pekerjaan, akan ada jumlah waktu mulai yang tak terbatas untuk setiap variabel.) Dengan domain tak terbatas, tidak mungkin lagi untuk menggambarkan kendala dengan menghitung semua kombinasi nilai yang diizinkan. Sebaliknya, bahasa kendala harus digunakan yang memahami kendala seperti $T_1 + d_1 \leq T_2$ secara langsung, tanpa menghitung set pasangan nilai yang diizinkan untuk (T_1, T_2). Algoritma solusi khusus (yang tidak kita bahas di sini) ada untuk kendala linier pada variabel integer—yaitu, kendala, seperti yang baru saja diberikan, di mana setiap variabel hanya muncul dalam bentuk linier. Dapat ditunjukkan bahwa tidak ada algoritma untuk memecahkan kendala nonlinier umum pada variabel integer.

Masalah pemenuhan kendala dengan domain kontinu umum terjadi di dunia nyata dan dipelajari secara luas dalam bidang riset operasi. Misalnya, penjadwalan eksperimen pada Teleskop Luar Angkasa Hubble memerlukan pengaturan waktu pengamatan yang sangat tepat; awal dan akhir setiap pengamatan dan manuver adalah variabel bernilai kontinu yang harus mematuhi berbagai kendala astronomi, preseden, dan daya. Kategori CSP domain kontinu yang paling terkenal adalah masalah pemrograman linier, di mana kendala harus berupa persamaan atau ketidaksetaraan linier. Masalah pemrograman linier dapat diselesaikan dalam polinomial waktu dalam jumlah variabel. Masalah dengan berbagai jenis kendala dan fungsi objektif juga telah dipelajari—pemrograman kuadrat, pemrograman kerucut orde kedua, dan seterusnya.

Selain memeriksa jenis variabel yang dapat muncul dalam CSP, ada baiknya untuk melihat jenis kendala. Jenis yang paling sederhana adalah **kendala unary**, yang membatasi nilai satu variabel. Misalnya, dalam masalah pewarnaan peta, mungkin saja warga Australia Selatan tidak akan menoleransi warna hijau; Kita dapat menyatakannya dengan kendala unary $\langle (SA), SA \neq Green \rangle$

Kendala **biner menghubungkan** dua variabel. Misalnya, $SA \neq NSW$ adalah kendala biner. CSP biner adalah CSP yang hanya memiliki kendala biner; CSP biner dapat direpresentasikan sebagai grafik kendala, seperti pada Gambar 6.1(b).

Kita juga dapat menggambarkan kendala tingkat tinggi, seperti menegaskan bahwa nilai Y berada di antara X dan Z, dengan kendala terner Antara(X, Y, Z).

Kendala yang melibatkan sejumlah variabel yang sembarangan disebut **kendala global.** (Namanya tradisional tetapi membingungkan karena tidak perlu melibatkan semua variabel dalam suatu masalah). Salah satu kendala global yang paling umum adalah *Alldiff*, yang menyatakan bahwa semua variabel yang terlibat dalam kendala harus memiliki nilai yang berbeda. Dalam soal Sudoku, semua variabel dalam baris atau kolom harus memenuhi

kendala *Alldiff*. Contoh lain diberikan oleh teka-teki **kriptoaritmatika**. (Lihat Gambar 6.2(a).) Setiap huruf dalam teka-teki kriptoaritmatika mewakili digit yang berbeda. Untuk kasus pada Gambar 6.2(a), ini akan direpresentasikan sebagai kendala global *Alldiff* (F, T, U, W, R, O). Kendala penjumlahan pada empat kolom teka-teki dapat ditulis sebagai kendala n-ary berikut:

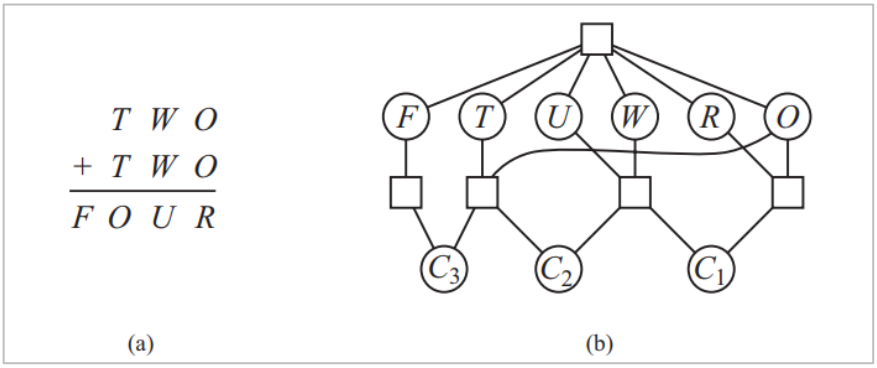
$$O + O = R + 10 \cdot C_{10}$$
$$C_{10} + W + W = U + 10 \cdot C_{100}$$
$$C_{100} + T + T = O + 10 \cdot C_{1000}$$
$$C_{1000} = F,$$

di mana C_{10}, C_{100} dan C_{1000} adalah variabel bantu yang mewakili digit yang dibawa ke kolom puluhan, ratusan, atau ribuan. Kendala ini dapat direpresentasikan dalam **hipergraf kendala**, seperti yang ditunjukkan pada Gambar 6.2(b). Hipergraf terdiri dari simpul biasa (lingkaran pada gambar) dan hipernode (kotak), yang mewakili kendala n-ary.

Atau, seperti yang diminta Latihan 6.6 untuk dibuktikan, setiap kendala domain-hingga dapat direduksi menjadi sekumpulan kendala biner jika cukup banyak variabel bantu yang diperkenalkan, sehingga kita dapat mengubah CSP apa pun menjadi satu dengan hanya kendala biner; ini membuat algoritme lebih sederhana. Cara lain untuk mengubah CSP n-ary menjadi biner adalah transformasi **grafik ganda**: buat grafik baru di mana akan ada satu variabel untuk setiap kendala dalam grafik asli, dan satu kendala biner untuk setiap pasangan kendala dalam grafik asli yang berbagi variabel. Misalnya, jika grafik asli memiliki variabel $\{X, Y, Z\}$ dan kendala $\langle (X, Y, Z), C_1 \rangle$ dan $\langle (X, Y\ ), C_2 \rangle$ maka grafik dual akan memiliki variabel $\{C_1, C_2\}$ dengan kendala biner $\langle (X, Y), R_1 \rangle$, di mana (X, Y) adalah variabel bersama dan R_1 adalah relasi baru yang mendefinisikan kendala antara variabel bersama, sebagaimana ditentukan oleh C_1 dan C_2 asli.

Namun, ada dua alasan mengapa kita mungkin lebih menyukai kendala global seperti Alldiff daripada serangkaian kendala biner. Pertama, lebih mudah dan lebih sedikit rawan kesalahan untuk menulis deskripsi masalah menggunakan Alldiff. Kedua, adalah mungkin untuk merancang algoritma inferensi tujuan khusus untuk kendala global yang tidak tersedia untuk serangkaian kendala yang lebih primitif. Kami menjelaskan algoritma inferensi ini di Bagian 6.2.5. Kendala yang telah kami jelaskan sejauh ini semuanya merupakan kendala absolut, yang jika dilanggar akan menyingkirkan solusi potensial. Banyak CSP dunia nyata menyertakan kendala preferensi yang menunjukkan solusi mana yang lebih disukai. Misalnya, dalam masalah penjadwalan kelas universitas, ada kendala absolut bahwa tidak ada profesor yang dapat mengajar dua kelas pada saat yang sama. Namun, kami juga dapat mengizinkan kendala preferensi: Prof. R mungkin lebih suka mengajar di pagi hari, sedangkan Prof. N lebih suka mengajar di sore hari. Jadwal yang mengharuskan Prof. R mengajar pada pukul 2 siang akan tetap menjadi solusi yang diizinkan (kecuali jika Prof. R kebetulan menjadi ketua departemen) tetapi tidak akan menjadi solusi yang optimal. Kendala preferensi sering kali dapat dikodekan sebagai biaya pada penugasan variabel individual—misalnya, menetapkan

slot sore untuk Prof. R menghabiskan biaya 2 poin terhadap fungsi tujuan keseluruhan, sedangkan slot pagi menghabiskan biaya 1. Dengan formulasi ini, CSP dengan preferensi dapat dipecahkan dengan metode pencarian optimasi, baik berbasis jalur atau lokal. Kami menyebut masalah seperti itu sebagai masalah optimasi kendala, atau COP. Masalah pemrograman linier melakukan optimasi semacam ini.



Gambar 6.2 (a) Masalah kriptoaritmatika. Setiap huruf mewakili digit yang berbeda; tujuannya adalah untuk menemukan substitusi digit untuk huruf sehingga hasil penjumlahannya benar secara aritmatika, dengan batasan tambahan bahwa tidak boleh ada angka nol di depan. (b) Hipergraf kendala untuk masalah kriptoaritmatika, yang menunjukkan kendala *Alldiff* (kotak persegi di bagian atas) serta kendala penambahan kolom (empat kotak persegi di tengah). Variabel C_1, C_2, dan C_3 mewakili digit bawaan untuk tiga kolom.

6.2 PROPAGASI KENDALA: INFERENSI DALAM CSP

Dalam penelusuran ruang-keadaan biasa, suatu algoritme hanya dapat melakukan satu hal: penelusuran. Dalam CSP terdapat pilihan: suatu algoritme dapat menelusuri (memilih penugasan variabel baru dari beberapa kemungkinan) atau melakukan jenis **inferensi** tertentu yang disebut **propagasi kendala**: menggunakan kendala untuk mengurangi jumlah nilai legal untuk suatu variabel, yang pada gilirannya dapat mengurangi nilai legal untuk variabel lain, dan seterusnya. Propagasi kendala dapat dijalin dengan penelusuran, atau dapat dilakukan sebagai langkah praproses, sebelum penelusuran dimulai. Terkadang praproses ini dapat menyelesaikan seluruh masalah, jadi penelusuran tidak diperlukan sama sekali.

Ide utamanya adalah **konsistensi lokal**. Jika kita memperlakukan setiap variabel sebagai simpul dalam grafik (lihat Gambar 6.1(b)) dan setiap kendala biner sebagai busur, maka proses penerapan konsistensi lokal di setiap bagian grafik menyebabkan nilai yang tidak konsisten dihilangkan di seluruh grafik. Ada berbagai jenis konsistensi lokal, yang sekarang akan kita bahas secara bergantian.

Konsistensi simpul

Sebuah variabel tunggal (yang sesuai dengan simpul dalam jaringan CSP) **konsisten terhadap simpul** jika semua nilai dalam domain variabel tersebut memenuhi batasan unary variabel tersebut. Misalnya, dalam varian masalah pewarnaan peta Australia (Gambar 6.1) di mana penduduk Australia Selatan tidak menyukai warna hijau, variabel *SA* dimulai dengan domain $\{merah, hijau, biru\}$, dan kita dapat membuatnya konsisten terhadap simpul dengan

menghilangkan warna $hijau$, sehingga SA tetap memiliki domain tereduksi $\{merah, biru\}$. Kita katakan bahwa suatu jaringan konsisten terhadap simpul jika setiap variabel dalam jaringan tersebut konsisten terhadap simpul.

Selalu mungkin untuk menghilangkan semua batasan unary dalam suatu CSP dengan menjalankan konsistensi simpul. Dimungkinkan juga untuk mengubah semua batasan n-ary menjadi batasan biner. Oleh karena itu, adalah umum untuk mendefinisikan penyelesai CSP yang hanya bekerja dengan batasan biner; kita membuat asumsi tersebut untuk sisa bab ini, kecuali jika disebutkan.

Konsistensi busur

Sebuah variabel dalam CSP konsisten busur jika setiap nilai dalam domainnya memenuhi batasan biner variabel tersebut. Secara lebih formal, X_i konsisten busur terhadap variabel lain X_j jika untuk setiap nilai dalam domain Di saat ini terdapat beberapa nilai dalam domain Dj yang memenuhi batasan biner pada busur (X_i, X_j). Sebuah jaringan konsisten busur jika setiap variabel konsisten busur dengan setiap variabel lainnya. Misalnya, perhatikan batasan $Y = X^2$ di mana domain dari X dan Y adalah himpunan digit. Kita dapat menulis batasan ini secara eksplisit sebagai

$$\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9))\} \rangle .$$

Untuk membuat X konsisten busur terhadap Y, kita kurangi domain X menjadi $\{0, 1, 2, 3\}$. Jika kita juga membuat Y konsisten terhadap X, maka domain Y menjadi $\{0, 1, 4, 9\}$ dan seluruh CSP konsisten.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
    inputs: csp, a binary CSP with components (X, D, C)
    local variables: queue, a queue of arcs, initially all the arcs in csp
    ---------------------------------------------------------------------
    while queue is not empty do
        (X_i, X_j) ← REMOVE-FIRST(queue)
        if REVISE(csp, X_i, X_j) then
            if size of D_i = 0 then return false
            for each X_k in X_i.NEIGHBORS - {X_j} do
                add (X_k, X_i) to queue
    return true
    ---------------------------------------------------------------------
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
    revised ← false
    for each x in D_i do
        if no value y in D_j allows (x ,y) to satisfy the constraint between X_i and X_j then
            delete x from D_i
            revised ← true
    return revised
```

Gambar 6.3 Algoritma konsistensi busur AC-3. Setelah menerapkan AC-3, setiap busur

konsisten terhadap busur, atau beberapa variabel memiliki domain kosong, yang menunjukkan bahwa CSP tidak dapat dipecahkan. Nama "AC-3" digunakan oleh penemu algoritma karena merupakan versi ketiga yang dikembangkan.

Di sisi lain, konsistensi busur tidak dapat membantu masalah pewarnaan peta Australia. Pertimbangkan kendala ketidaksetaraan berikut pada (SA, WA):

$$\{(merah, hijau), (merah, biru), (hijau, merah), (hijau, biru), (biru, merah), (biru, hijau)\}.$$

Apa pun nilai yang Anda pilih untuk SA (atau untuk WA), ada nilai yang valid untuk variabel lainnya. Jadi, menerapkan konsistensi busur tidak berpengaruh pada domain dari kedua variabel tersebut.

Algoritme yang paling populer untuk konsistensi busur disebut AC-3 (lihat Gambar 6.3). Untuk membuat setiap variabel konsisten busur, algoritme AC-3 mempertahankan antrean busur untuk dipertimbangkan. (Sebenarnya, urutan pertimbangan tidak penting, jadi struktur data sebenarnya adalah satu set, tetapi tradisi menyebutnya antrean.) Awalnya, antrean berisi semua busur di CSP. AC-3 kemudian mengeluarkan busur sembarang (X_i, X_j) dari antrean dan membuat X_i konsisten busur sehubungan dengan X_j. Jika ini membiarkan Di tidak berubah, algoritme hanya beralih ke busur berikutnya. Tetapi jika ini merevisi Di (membuat domain lebih kecil), maka kita menambahkan ke antrean semua busur (X_k, X_i) di mana X_k adalah tetangga X_i. Kita perlu melakukan itu karena perubahan pada D_i mungkin memungkinkan pengurangan lebih lanjut dalam domain D_k, bahkan jika kita sebelumnya telah mempertimbangkan X_k. Jika D_i direvisi menjadi nol, maka kita tahu seluruh CSP tidak memiliki solusi yang konsisten, dan AC-3 dapat segera mengembalikan kegagalan. Jika tidak, kita terus memeriksa, mencoba menghapus nilai dari domain variabel hingga tidak ada lagi busur dalam antrean. Pada titik itu, kita dibiarkan dengan CSP yang setara dengan CSP asli—keduanya memiliki solusi yang sama—tetapi CSP yang konsisten dengan busur dalam banyak kasus akan lebih cepat dicari karena variabelnya memiliki domain yang lebih kecil.

Kompleksitas AC-3 dapat dianalisis sebagai berikut. Asumsikan CSP dengan n variabel, masing-masing dengan ukuran domain paling banyak d, dan dengan c kendala biner (busur). Setiap busur (X_k, X_i) dapat dimasukkan dalam antrean hanya d kali karena X_i memiliki paling banyak d nilai untuk dihapus. Pengecekan konsistensi busur dapat dilakukan dalam waktu $O(d^2)$, jadi kita memperoleh total waktu terburuk $O(cd^3)$.

Adalah mungkin untuk memperluas pengertian konsistensi busur untuk menangani kendala n-ary daripada hanya kendala biner; ini disebut konsistensi busur umum atau terkadang konsistensi hiperbusur, tergantung pada penulisnya. Suatu variabel X_i adalah **konsisten busur umum** sehubungan dengan kendala n-ary jika untuk setiap nilai v dalam domain X_i terdapat tupel nilai yang merupakan anggota kendala, memiliki semua nilainya diambil dari domain variabel yang sesuai, dan memiliki komponen X_i yang sama dengan v. Misalnya, jika semua variabel memiliki domain $\{0, 1, 2, 3\}$, maka untuk membuat variabel X konsisten dengan kendala $X < Y < Z$, kita harus menghilangkan 2 dan 3 dari domain X karena kendala tidak dapat dipenuhi ketika X adalah 2 atau 3.

Konsistensi jalur

Konsistensi busur dapat sangat membantu dalam mengurangi domain variabel, terkadang menemukan solusi (dengan mengurangi setiap domain ke ukuran 1) dan terkadang menemukan bahwa CSP tidak dapat dipecahkan (dengan mengurangi beberapa domain ke ukuran 0). Namun untuk jaringan lain, konsistensi busur gagal membuat cukup banyak inferensi. Pertimbangkan masalah pewarnaan peta di Australia, tetapi hanya dengan dua warna yang diizinkan, merah dan biru. Konsistensi busur tidak dapat melakukan apa pun karena setiap variabel sudah konsisten dengan busur: masing-masing dapat berwarna merah dengan biru di ujung busur lainnya (atau sebaliknya). Namun jelas tidak ada solusi untuk masalah tersebut: karena Australia Barat, Teritori Utara, dan Australia Selatan semuanya saling bersentuhan, kita memerlukan setidaknya tiga warna untuk mereka saja.

Konsistensi busur memperketat domain (kendala unary) menggunakan busur (kendala biner). Untuk membuat kemajuan pada masalah seperti pewarnaan peta, kita memerlukan gagasan konsistensi yang lebih kuat. Konsistensi jalur memperketat kendala biner dengan menggunakan kendala implisit yang disimpulkan dengan melihat tiga variabel. Himpunan dua variabel $\{X_i, X_j\}$ konsisten lintasan terhadap variabel ketiga X_m jika, untuk setiap penugasan $\{X_i = a, X_j = b\}$ yang konsisten dengan batasan pada $\{X_i, X_j\}$, terdapat penugasan ke X_m yang memenuhi batasan pada $\{X_i, X_m\}$ dan $\{X_m, X_j\}$. Ini disebut konsistensi lintasan karena kita dapat menganggapnya sebagai lintasan dari X_i ke X_j dengan X_m di tengahnya.

Mari kita lihat bagaimana konsistensi lintasan berlaku dalam pewarnaan peta Australia dengan dua warna. Kita akan membuat himpunan $\{WA, SA\}$ konsisten lintasan terhadap NT. Kita mulai dengan menghitung penugasan konsisten ke himpunan tersebut. Dalam kasus ini, hanya ada dua: {WA = $merah$, SA = $biru$} dan {WA = $biru$, SA = $merah$}. Kita dapat melihat bahwa dengan kedua penugasan ini, NT tidak dapat berwarna merah maupun biru (karena akan berbenturan dengan WA maupun SA). Karena tidak ada pilihan yang valid untuk NT, kami menghilangkan kedua penugasan, dan kami berakhir dengan tidak ada penugasan yang valid untuk $\{WA, SA\}$. Oleh karena itu, kami tahu bahwa tidak ada solusi untuk masalah ini. Algoritme PC-2 mencapai konsistensi jalur dengan cara yang sama seperti AC-3 mencapai konsistensi busur. Karena sangat mirip, kami tidak menunjukkannya di sini.

Konsistensi K

Bentuk propagasi yang lebih kuat dapat didefinisikan dengan gagasan **konsistensi k**. CSP bersifat k −konsisten jika, untuk setiap himpunan variabel $k - 1$ dan untuk setiap penugasan yang konsisten pada variabel tersebut, nilai yang konsisten selalu dapat ditetapkan pada variabel ke-k mana pun. Konsistensi 1 menyatakan bahwa, jika diberikan himpunan yang kosong, kita dapat membuat setiap himpunan variabel tunggal menjadi konsisten: inilah yang kita sebut konsistensi simpul. Konsistensi 2 sama dengan konsistensi busur. Untuk jaringan kendala biner, konsistensi 3 sama dengan konsistensi jalur.

CSP bersifat k −konsisten kuat jika bersifat k −konsisten dan juga $(k - 1)$ −konsisten, $(k - 2)$ −konsisten, ... hingga ke 1-konsisten. Sekarang anggaplah kita memiliki CSP dengan n simpul dan membuatnya sangat n −konsisten (yaitu, sangat k-konsisten untuk $k = n$). Kita kemudian dapat memecahkan masalah sebagai berikut:

Pertama, kita memilih nilai yang konsisten untuk X_1. Kita kemudian dijamin dapat memilih nilai untuk X_2 karena grafiknya konsisten 2, untuk X_3 karena konsisten 3, dan seterusnya. Untuk setiap variabel X_i, kita hanya perlu mencari melalui nilai d dalam domain untuk menemukan nilai yang konsisten dengan $X_1 \ldots X_{i-1}$. Kita dijamin menemukan solusi dalam waktu $O(n^2 d)$. Tentu saja, tidak ada makan siang gratis: setiap algoritma untuk menetapkan konsistensi-n harus mengambil waktu eksponensial dalam n dalam kasus terburuk. Lebih buruk lagi, konsistensi$-n$ juga membutuhkan ruang yang eksponensial dalam n. Masalah memori bahkan lebih parah daripada waktu. Dalam praktiknya, menentukan tingkat pengecekan konsistensi yang tepat sebagian besar merupakan ilmu empiris. Dapat dikatakan bahwa praktisi umumnya menghitung konsistensi-2 dan lebih jarang konsistensi-3.

Kendala global

Ingatlah bahwa kendala global adalah kendala yang melibatkan sejumlah variabel yang sembarangan (tetapi tidak harus semua variabel). Kendala global sering terjadi dalam masalah nyata dan dapat ditangani oleh algoritme tujuan khusus yang lebih efisien daripada metode tujuan umum yang dijelaskan sejauh ini. Misalnya, kendala *Alldiff* menyatakan bahwa semua variabel yang terlibat harus memiliki nilai yang berbeda (seperti dalam masalah kriptoaritmatika di atas dan teka-teki Sudoku di bawah).

Satu bentuk sederhana deteksi ketidakkonsistenan untuk kendala *Alldiff* bekerja sebagai berikut: jika m variabel terlibat dalam kendala, dan jika mereka memiliki n kemungkinan nilai yang berbeda secara keseluruhan, dan $m > n$, maka kendala tidak dapat dipenuhi. Ini mengarah ke algoritme sederhana berikut: Pertama, hapus variabel apa pun dalam kendala yang memiliki domain tunggal, dan hapus nilai variabel itu dari domain variabel yang tersisa. Ulangi selama ada variabel tunggal. Jika pada titik mana pun domain kosong dihasilkan atau ada lebih banyak variabel daripada nilai domain yang tersisa, maka ketidakkonsistenan telah terdeteksi.

Metode ini dapat mendeteksi ketidakkonsistenan dalam penugasan $\{WA = merah, NSW = merah\}$ untuk Gambar 6.1. Perhatikan bahwa variabel SA, NT, dan Q secara efektif terhubung oleh kendala *Alldiff* karena setiap pasangan harus memiliki dua warna yang berbeda. Setelah menerapkan AC-3 dengan penugasan parsial, domain setiap variabel direduksi menjadi {*hijau, biru*}. Artinya, kita memiliki tiga variabel dan hanya dua warna, sehingga kendala *Alldiff* dilanggar. Dengan demikian, prosedur konsistensi sederhana untuk kendala orde tinggi terkadang lebih efektif daripada menerapkan konsistensi busur ke serangkaian kendala biner yang setara. Ada algoritme inferensi yang lebih kompleks untuk *Alldiff* yang menyebarkan lebih banyak kendala tetapi lebih mahal secara komputasi untuk dijalankan.

Kendala orde tinggi penting lainnya adalah **kendala sumber daya**, terkadang disebut kendala paling tinggi. Misalnya, dalam masalah penjadwalan, misalkan $P_1, \ldots, P_4$ menunjukkan jumlah personel yang ditugaskan untuk masing-masing dari empat tugas. Kendala yang mengharuskan tidak lebih dari 10 personel ditugaskan secara total ditulis sebagai Paling Banyak $(10, P_1, P_2, P_3, P_4)$. Kita dapat mendeteksi ketidakkonsistenan hanya dengan memeriksa jumlah nilai minimum domain saat ini; misalnya, jika setiap variabel memiliki domain {3, 4, 5,

6}, kendala Paling Banyak tidak dapat dipenuhi. Kita juga dapat menegakkan konsistensi dengan menghapus nilai maksimum domain mana pun jika tidak konsisten dengan nilai minimum domain lainnya. Jadi, jika setiap variabel dalam contoh kita memiliki domain {2, 3, 4, 5, 6}, nilai 5 dan 6 dapat dihapus dari setiap domain.

Untuk masalah besar yang terbatas sumber dayanya dengan nilai integer—seperti masalah logistik yang melibatkan pemindahan ribuan orang dalam ratusan kendaraan—biasanya tidak mungkin untuk merepresentasikan domain setiap variabel sebagai sekumpulan besar integer dan secara bertahap mengurangi himpunan tersebut dengan metode pengecekan konsistensi.

Sebaliknya, domain direpresentasikan oleh batas atas dan bawah dan dikelola oleh **propagasi batas**. Misalnya, dalam masalah penjadwalan maskapai penerbangan, misalkan ada dua penerbangan, F_1 dan F_2, yang pesawatnya memiliki kapasitas masing-masing 165 dan 385. Domain awal untuk jumlah penumpang pada setiap penerbangan kemudian adalah:

$$D_1 = [0.165] \text{ dan } D_2 = [0.385]$$

Sekarang misalkan kita memiliki kendala tambahan bahwa kedua penerbangan tersebut harus membawa 420 orang: $F_1 + F_2 = 420$. Dengan menyebarkan kendala batas, kita mengurangi domain menjadi:

$$D_1 = [35.165] \text{ dan } D_2 = [255.385]$$

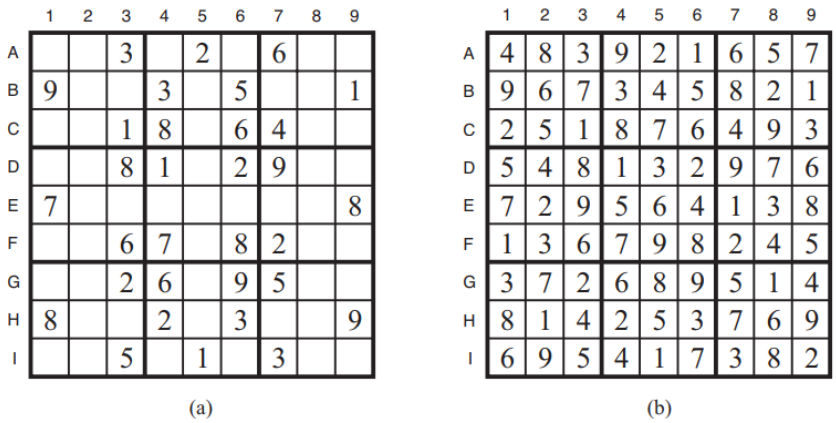
Kita katakan bahwa CSP **konsisten batas** jika untuk setiap variabel X, dan untuk nilai batas bawah dan batas atas X, terdapat beberapa nilai Y yang memenuhi kendala antara X dan Y untuk setiap variabel Y. Jenis propagasi batas ini banyak digunakan dalam masalah kendala praktis.

Contoh Sudoku

Teka-teki **Sudoku** yang populer telah memperkenalkan jutaan orang pada masalah pemenuhan kendala, meskipun mereka mungkin tidak mengenalinya. Papan Sudoku terdiri dari 81 kotak, beberapa di antaranya awalnya diisi dengan angka dari 1 hingga 9. Teka-teki tersebut adalah untuk mengisi semua kotak yang tersisa sedemikian rupa sehingga tidak ada angka yang muncul dua kali dalam baris, kolom, atau kotak 3×3 mana pun (lihat Gambar 6.4). Baris, kolom, atau kotak disebut **unit**.

Teka-teki Sudoku yang dicetak di koran dan buku teka-teki memiliki sifat bahwa ada tepat satu solusi. Meskipun beberapa soal mungkin sulit dipecahkan secara manual, dan memakan waktu puluhan menit, bahkan soal Sudoku yang paling sulit pun dapat diselesaikan oleh pemecah CSP dalam waktu kurang dari $0{,}1$ detik.

Teka-teki Sudoku dapat dianggap sebagai CSP dengan 81 variabel, satu untuk setiap kotak. Kami menggunakan nama variabel $A1$ hingga $A9$ untuk baris atas (kiri ke kanan), hingga $I1$ hingga $I9$ untuk baris bawah. Kotak kosong memiliki domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ dan kotak yang telah diisi sebelumnya memiliki domain yang terdiri dari satu nilai.



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Gambar 6.4 (a) Teka-teki Sudoku dan (b) solusinya.

Selain itu, ada 27 kendala Alldiff yang berbeda: satu untuk setiap baris, kolom, dan kotak yang berisi 9 kotak.

$$Alldiff\ (A1, A2, A3, A4, A5, A6, A7, A8, A9)$$
$$Alldiff\ (B1, B2, B3, B4, B5, B6, B7, B8, B9)$$
$$\ldots$$
$$Alldiff\ (A1, B1, C1, D1, E1, F1, G1, H1, I1)$$
$$Alldiff\ (A2, B2, C2, D2, E2, F2, G2, H2, I2)$$
$$\ldots$$
$$Alldiff\ (A1, A2, A3, B1, B2, B3, C1, C2, C3)$$
$$Alldiff\ (A4, A5, A6, B4, B5, B6, C4, C5, C6)$$
$$\ldots$$

Mari kita lihat sejauh mana konsistensi busur dapat membawa kita. Asumsikan bahwa kendala Alldiff telah diperluas menjadi kendala biner (seperti $A1 \neq A2$) sehingga kita dapat menerapkan algoritma AC-3 secara langsung. Pertimbangkan variabel $E6$ dari Gambar 6.4(a)—kotak kosong antara angka 2 dan 8 di kotak tengah. Dari kendala di kotak, kita dapat menghapus tidak hanya 2 dan 8 tetapi juga 1 dan 7 dari domain $E6$. Dari kendala di kolomnya, kita dapat menghilangkan 5, 6, 2, 8, 9, dan 3. Itu menyisakan $E6$ dengan domain {4}; dengan kata lain, kita mengetahui jawaban untuk $E6$. Sekarang perhatikan variabel $I6$ —kotak di kotak tengah bawah yang dikelilingi oleh 1, 3, dan 3. Dengan menerapkan konsistensi busur di kolomnya, kita menghilangkan 5, 6, 2, 4 (karena sekarang kita tahu $E6$ pasti 4), 8, 9, dan 3. Kita menghilangkan 1 dengan konsistensi busur dengan $I5$, dan kita hanya memiliki nilai 7 di domain $I6$. Sekarang ada 8 nilai yang diketahui di kolom 6, jadi konsistensi busur dapat menyimpulkan bahwa $A6$ pasti 1. Inferensi berlanjut sepanjang garis ini, dan akhirnya, AC-3 dapat memecahkan seluruh teka-teki—semua variabel memiliki domain yang direduksi menjadi satu nilai, seperti yang ditunjukkan pada Gambar 6.4(b).

Tentu saja, Sudoku akan segera kehilangan daya tariknya jika setiap teka-teki dapat dipecahkan dengan penerapan mekanis AC-3, dan memang AC-3 hanya berfungsi untuk teka-teki Sudoku yang termudah. Yang sedikit lebih sulit dapat dipecahkan oleh PC-2, tetapi dengan

biaya komputasi yang lebih besar: ada 255.960 kendala jalur yang berbeda untuk dipertimbangkan dalam teka-teki Sudoku. Untuk memecahkan teka-teki yang paling sulit dan untuk membuat kemajuan yang efisien, kita harus lebih pintar.

Memang, daya tarik teka-teki Sudoku bagi pemecah manusia adalah kebutuhan untuk menjadi lebih banyak akal dalam menerapkan strategi inferensi yang lebih kompleks. Para penggemar memberi mereka nama yang berwarna-warni, seperti "tripleks telanjang". Strategi itu bekerja sebagai berikut: dalam unit apa pun (baris, kolom atau kotak), temukan tiga kotak yang masing-masing memiliki domain yang berisi tiga angka yang sama atau subset dari angka-angka tersebut. Misalnya, tiga domain mungkin adalah $\{1, 8\}$, $\{3, 8\}$, dan $\{1, 3, 8\}$. Dari situ kita tidak tahu kotak mana yang berisi $1, 3$, atau 8, tetapi kita tahu bahwa ketiga angka itu harus didistribusikan di antara ketiga kotak tersebut. Oleh karena itu, kita dapat menghilangkan $1, 3$, dan 8 dari domain setiap kotak lain dalam unit tersebut.

Menarik untuk dicatat seberapa jauh kita dapat melangkah tanpa mengatakan banyak hal yang khusus untuk Sudoku. Tentu saja kita harus mengatakan bahwa ada 81 variabel, bahwa domainnya adalah angka 1 hingga 9, dan bahwa ada 27 kendala *Alldiff*. Namun, di luar itu, semua strategi—konsistensi busur, konsistensi lintasan, dsb.—berlaku secara umum untuk semua CSP, bukan hanya untuk soal Sudoku. Bahkan triple telanjang sebenarnya adalah strategi untuk menegakkan konsistensi kendala *Alldiff* dan tidak ada hubungannya dengan Sudoku itu sendiri. Inilah kekuatan formalisme CSP: untuk setiap area soal baru, kita hanya perlu mendefinisikan soal dalam bentuk kendala; kemudian mekanisme penyelesaian kendala umum dapat mengambil alih.

6.3 PENCARIAN BACKTRACKING UNTUK CSP

Persoalan Sudoku dirancang untuk dipecahkan dengan inferensi atas kendala. Namun, banyak CSP lain yang tidak dapat dipecahkan hanya dengan inferensi; akan tiba saatnya kita harus mencari solusi. Di bagian ini, kita akan membahas algoritma pencarian backtracking yang bekerja pada penugasan parsial; di bagian berikutnya, kita akan membahas algoritma pencarian lokal atas penugasan lengkap.

Kita dapat menerapkan pencarian standar dengan batasan kedalaman (dari Bab 3). Suatu status akan menjadi penugasan parsial, dan suatu tindakan akan menambahkan $var = nilai$ ke penugasan. Namun, untuk CSP dengan n variabel berukuran domain d, kita akan segera menyadari sesuatu yang buruk: faktor percabangan di tingkat atas adalah nd karena nilai d apa pun dapat ditetapkan ke variabel n apa pun. Di tingkat berikutnya, faktor percabangan adalah $(n - 1)d$, dan seterusnya untuk n tingkat. Kita akan membuat pohon dengan $n! \cdot d^n$ daun, meskipun hanya ada d^n kemungkinan penugasan lengkap! Rumusan kita yang tampaknya masuk akal tetapi naif mengabaikan sifat penting yang umum untuk semua CSP: komutatif. Suatu masalah bersifat komutatif jika urutan penerapan dari serangkaian tindakan yang diberikan tidak berpengaruh pada hasilnya.

CSP bersifat **komutatif** karena ketika menetapkan nilai ke variabel, kita mencapai penugasan parsial yang sama tanpa mempedulikan urutannya. Oleh karena itu, kita hanya perlu mempertimbangkan satu variabel di setiap simpul di pohon pencarian. Misalnya, di

simpul akar pohon pencarian untuk mewarnai peta Australia, kita mungkin membuat pilihan antara SA = merah, SA = hijau, dan SA = biru, tetapi kita tidak akan pernah memilih antara SA = merah dan WA = biru. Dengan pembatasan ini, jumlah daun adalah dn, seperti yang kita harapkan.

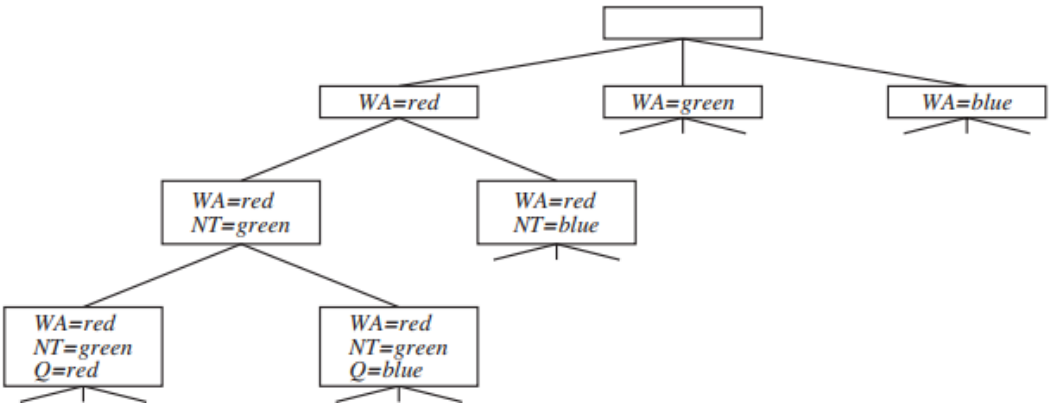
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)
function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var , assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var , value)
            if inferences = failure then
                add inferences to assignment
                result ← BACKTRACK(assignment, csp)
                if result = failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```

Gambar 6.5 Algoritma backtracking sederhana untuk masalah pemenuhan kendala. Algoritma ini dimodelkan pada pencarian rekursif depth-first dari Bab 3. Dengan memvariasikan fungsi SELECT − UNASSIGNED − VARIABLE dan ORDER − DOMAIN − VALUES, kita dapat menerapkan heuristik tujuan umum yang dibahas dalam teks. Fungsi INFERENCE secara opsional dapat digunakan untuk memaksakan konsistensi busur, lintasan, atau k, sesuai keinginan. Jika pilihan nilai menyebabkan kegagalan (diperhatikan baik oleh INFERENCE atau oleh BACKTRACK), maka penetapan nilai (termasuk yang dibuat oleh INFERENCE) dihapus dari penetapan saat ini dan nilai baru dicoba.

Istilah **pencarian backtracking** digunakan untuk pencarian mendalam yang memilih nilai untuk satu variabel pada satu waktu dan melakukan backtracking saat variabel tidak memiliki nilai legal yang tersisa untuk ditetapkan. Algoritme tersebut ditunjukkan pada Gambar 6.5. Algoritme tersebut berulang kali memilih variabel yang tidak ditetapkan, lalu mencoba semua nilai dalam domain variabel tersebut secara bergantian, mencoba menemukan solusi. Jika ketidakkonsistenan terdeteksi, maka BACKTRACK mengembalikan kegagalan, yang menyebabkan panggilan sebelumnya mencoba nilai lain. Bagian dari pohon pencarian untuk masalah Australia ditunjukkan pada Gambar 6.6, di mana kami telah menetapkan variabel dalam urutan WA, NT, Q, ...Karena representasi CSP distandarisasi, tidak perlu menyediakan BACKTRACKING − SEARCH dengan status awal khusus domain, fungsi tindakan, model transisi, atau uji tujuan. Perhatikan bahwa BACKTRACKING − SEARCH hanya menyimpan satu representasi dari suatu status dan mengubah representasi tersebut alih-alih membuat yang baru.

Pada Bab 3, kami memperbaiki kinerja buruk algoritma pencarian yang tidak terinformasi dengan menyediakan fungsi heuristik khusus domain yang berasal dari pengetahuan kami tentang masalah tersebut. Ternyata kami dapat memecahkan CSP secara efisien tanpa pengetahuan khusus domain tersebut. Sebaliknya, kami dapat menambahkan beberapa kecanggihan pada fungsi yang tidak ditentukan pada Gambar 6.5, menggunakannya untuk menjawab pertanyaan berikut:

1. Variabel mana yang harus ditetapkan berikutnya (SELECT – UNASSIGNED – VARIABLE), dan dalam urutan apa nilainya harus dicoba (ORDER – DOMAIN – VALUES)?
2. Inferensi apa yang harus dilakukan pada setiap langkah dalam pencarian (INFERENCE)?
3. Ketika pencarian sampai pada penugasan yang melanggar batasan, dapatkah pencarian menghindari pengulangan kegagalan ini?



Gambar 6.6 Bagian dari pohon pencarian untuk masalah pewarnaan peta pada Gambar 6.1.

Subbagian berikut menjawab setiap pertanyaan ini secara bergantian.

Urutan variabel dan nilai

Algoritma backtracking berisi garis

$$var \leftarrow SELECT - UNASSIGNED - VARIABLE(csp)\,.$$

Strategi paling sederhana untuk $SELECT - UNASSIGNED - VARIABLE$ adalah memilih variabel tak tertugaskan berikutnya secara berurutan, {X_1, X_2,.. .}. Pengurutan variabel statis ini jarang menghasilkan pencarian yang paling efisien. Misalnya, setelah penugasan untuk $WA = red$ dan $NT = green$ pada Gambar 6.6, hanya ada satu nilai yang mungkin untuk SA, jadi masuk akal untuk menetapkan $SA = blue$ berikutnya daripada menetapkan Q. Faktanya, setelah SA ditugaskan, pilihan untuk Q, NSW, dan V semuanya dipaksakan. Ide intuitif ini—memilih variabel dengan nilai "legal" paling sedikit—disebut heuristik **nilai-tersisa-minimum** (MRV). Ini juga disebut sebagai "variabel paling dibatasi" atau heuristik "gagal-pertama", yang terakhir karena memilih variabel yang paling mungkin menyebabkan kegagalan segera, sehingga memangkas pohon pencarian. Jika beberapa variabel X tidak memiliki nilai legal yang tersisa, heuristik MRV akan memilih X dan kegagalan akan segera

terdeteksi—menghindari pencarian yang tidak ada gunanya melalui variabel lain. Heuristik MRV biasanya berkinerja lebih baik daripada pemesanan acak atau statis, terkadang dengan faktor 1.000 atau lebih, meskipun hasilnya sangat bervariasi tergantung pada masalahnya.

Heuristik MRV sama sekali tidak membantu dalam memilih wilayah pertama yang akan diwarnai di Australia, karena awalnya setiap wilayah memiliki tiga warna legal. Dalam kasus ini, **heuristik derajat** berguna. Heuristik ini mencoba mengurangi faktor percabangan pada pilihan mendatang dengan memilih variabel yang terlibat dalam jumlah kendala terbesar pada variabel lain yang tidak ditetapkan. Pada Gambar 6.1, SA adalah variabel dengan derajat tertinggi, 5; variabel lainnya memiliki derajat 2 atau 3, kecuali T, yang memiliki derajat 0. Faktanya, setelah SA dipilih, penerapan heuristik derajat memecahkan masalah tanpa langkah yang salah—Anda dapat memilih warna yang konsisten di setiap titik pilihan dan tetap mendapatkan solusi tanpa mundur. Heuristik nilai-minimum-tersisa biasanya merupakan panduan yang lebih kuat, tetapi heuristik derajat dapat berguna sebagai pemecah seri.

Setelah variabel dipilih, algoritme harus memutuskan urutan untuk memeriksa nilainya. Untuk ini, heuristik **nilai-paling-tidak-memaksa** dapat efektif dalam beberapa kasus. Ia lebih menyukai nilai yang mengesampingkan pilihan paling sedikit untuk variabel tetangga dalam grafik kendala. Misalnya, anggaplah bahwa dalam Gambar 6.1 kita telah menghasilkan penugasan parsial dengan $WA = Red$ dan $NT = green$ dan pilihan kita berikutnya adalah untuk Q. Biru akan menjadi pilihan yang buruk karena menghilangkan nilai legal terakhir yang tersisa untuk tetangga $Q, SA.$ Oleh karena itu, heuristik nilai-paling-tidak-memaksa lebih menyukai merah daripada biru. Secara umum, heuristik mencoba memberikan fleksibilitas maksimum untuk penugasan variabel berikutnya. Tentu saja, jika kita mencoba menemukan semua solusi untuk suatu masalah, bukan hanya yang pertama, maka urutannya tidak penting karena kita harus mempertimbangkan setiap nilai. Hal yang sama berlaku jika tidak ada solusi untuk masalah tersebut.

Mengapa pemilihan variabel harus gagal terlebih dahulu, tetapi pemilihan nilai harus gagal terakhir? Ternyata, untuk berbagai macam masalah, urutan variabel yang memilih variabel dengan jumlah nilai tersisa minimum membantu meminimalkan jumlah simpul di pohon pencarian dengan memangkas bagian pohon yang lebih besar lebih awal. Untuk urutan nilai, triknya adalah kita hanya memerlukan satu solusi; oleh karena itu masuk akal untuk mencari nilai yang paling mungkin terlebih dahulu. Jika kita ingin menghitung semua solusi daripada hanya menemukan satu, maka urutan nilai tidak akan relevan.

Pencarian dan inferensi yang saling menyisipkan

Sejauh ini kita telah melihat bagaimana AC-3 dan algoritme lain dapat menyimpulkan reduksi dalam domain variabel *sebelum* kita memulai pencarian. Namun, inferensi dapat menjadi lebih kuat selama pencarian: setiap kali kita memilih nilai untuk variabel, kita memiliki peluang baru untuk menyimpulkan reduksi domain baru pada variabel tetangga.

Salah satu bentuk inferensi yang paling sederhana disebut **pemeriksaan maju**. Setiap kali variabel X ditetapkan, proses pemeriksaan maju menetapkan konsistensi busur untuknya: untuk setiap variabel Y yang tidak ditetapkan yang terhubung ke X dengan batasan, hapus dari domain Y nilai apa pun yang tidak konsisten dengan nilai yang dipilih untuk X. Karena

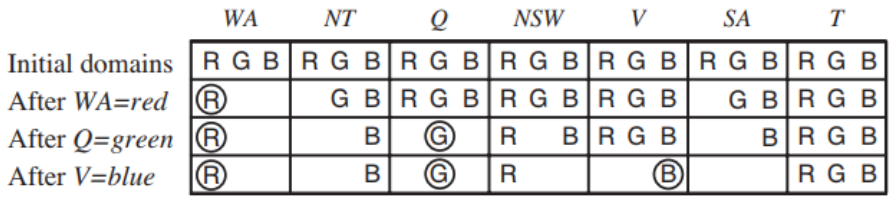
pemeriksaan maju hanya melakukan inferensi konsistensi busur, tidak ada alasan untuk melakukan pemeriksaan maju jika kita telah melakukan konsistensi busur sebagai langkah praproses.

Gambar 6.7 menunjukkan kemajuan pencarian backtracking pada CSP Australia dengan pemeriksaan maju. Ada dua hal penting yang perlu diperhatikan tentang contoh ini. Pertama, perhatikan bahwa setelah $WA = red$ dan $Q = green$ ditetapkan, domain NT dan SA direduksi menjadi satu nilai; kita telah menghilangkan percabangan pada variabel-variabel ini sama sekali dengan menyebarkan informasi dari WA dan Q. Hal kedua yang perlu diperhatikan adalah bahwa setelah $V = blue$, domain *SA* kosong. Oleh karena itu, pemeriksaan ke depan telah mendeteksi bahwa penetapan parsial $\{WA = red, Q = green, V = blue\}$ tidak konsisten dengan batasan masalah, dan oleh karena itu algoritme akan segera melakukan penelusuran balik.

Untuk banyak masalah, pencarian akan lebih efektif jika kita menggabungkan heuristik MRV dengan pemeriksaan ke depan. Perhatikan Gambar 6.7 setelah menetapkan $\{WA = red\}$. Secara intuitif, tampaknya penetapan itu membatasi tetangganya, NT dan SA, jadi kita harus menangani variabel-variabel itu berikutnya, dan kemudian semua variabel lainnya akan sesuai. Itulah yang terjadi dengan MRV: NT dan SA memiliki dua nilai, jadi salah satunya dipilih terlebih dahulu, lalu yang lain, lalu Q, NSW, dan V secara berurutan. Akhirnya T masih memiliki tiga nilai, dan salah satunya berfungsi. Kita dapat melihat pemeriksaan ke depan sebagai cara yang efisien untuk menghitung informasi yang dibutuhkan heuristik MRV secara bertahap untuk melakukan tugasnya.

Meskipun pemeriksaan ke depan mendeteksi banyak ketidakkonsistenan, pemeriksaan tersebut tidak mendeteksi semuanya. Masalahnya adalah pemeriksaan tersebut membuat variabel saat ini konsisten terhadap busur, tetapi tidak melihat ke depan dan membuat semua variabel lainnya konsisten terhadap busur. Misalnya, perhatikan baris ketiga Gambar 6.7. Hal tersebut menunjukkan bahwa ketika WA berwarna red dan Q berwarna $green$, baik NT maupun SA dipaksa menjadi $blue$. Pemeriksaan ke depan tidak melihat cukup jauh ke depan untuk menyadari bahwa ini adalah ketidakkonsistenan: NT dan SA berdekatan sehingga tidak dapat memiliki nilai yang sama.

Algoritme yang disebut MAC (untuk **Mempertahankan Konsistensi Busur (MAC**)) mendeteksi ketidakkonsistenan ini. Bahasa Indonesia: Setelah variabel X_i diberi nilai, prosedur INFERENCE memanggil AC-3, tetapi alih-alih antrean semua busur di CSP, kita mulai dengan hanya busur (X_i, X_j) untuk semua X_j yang merupakan variabel tak ditetapkan yang merupakan tetangga X_i. Dari sana, AC-3 melakukan propagasi kendala dengan cara biasa, dan jika ada variabel yang domainnya direduksi menjadi himpunan kosong, panggilan ke AC-3 gagal dan kita tahu untuk segera melakukan backtrack. Kita dapat melihat bahwa MAC secara ketat lebih kuat daripada pemeriksaan maju karena pemeriksaan maju melakukan hal yang sama seperti MAC pada busur awal dalam antrean MAC; tetapi tidak seperti MAC, pemeriksaan maju tidak menyebarkan kendala secara rekursif ketika perubahan dilakukan pada domain variabel.



Gambar 6.7 Kemajuan penelusuran pewarnaan peta dengan pemeriksaan maju. $WA = red$ ditetapkan terlebih dahulu; kemudian pemeriksaan maju menghapus merah dari domain variabel tetangga NT dan SA. Setelah $Q = green$ ditetapkan, hijau dihapus dari domain NT, SA, dan NSW. Setelah $V = blue$ ditetapkan, biru dihapus dari domain *NSW* dan SA, sehingga SA tidak memiliki nilai yang sah.

Penelusuran balik cerdas: Melihat ke belakang

Algoritme $\text{BACKTRACKING} - \text{SEARCH}$ pada Gambar 6.5 memiliki kebijakan yang sangat sederhana untuk apa yang harus dilakukan ketika cabang penelusuran gagal: kembali ke variabel sebelumnya dan coba nilai yang berbeda untuknya. Ini disebut **penelusuran balik kronologis** karena titik keputusan terkini ditinjau ulang. Dalam subbagian ini, kami mempertimbangkan kemungkinan yang lebih baik. Pertimbangkan apa yang terjadi ketika kami menerapkan penelusuran balik sederhana pada Gambar 6.1 dengan variabel tetap yang mengurutkan Q, NSW, V, T, SA, WA, NT. Misalkan kami telah menghasilkan penugasan parsial $\{Q = red, NSW = green, V = blue, T = red\}$. Ketika kami mencoba variabel berikutnya, SA, kami melihat bahwa setiap nilai melanggar batasan. Kami kembali ke T dan mencoba warna baru untuk Tasmania! Jelas ini konyol—mewarnai ulang Tasmania tidak mungkin menyelesaikan masalah dengan Australia Selatan.

Pendekatan yang lebih cerdas untuk melakukan backtracking adalah dengan melakukan backtracking ke variabel yang mungkin dapat memperbaiki masalah tersebut—variabel yang bertanggung jawab untuk membuat salah satu nilai SA yang mungkin menjadi tidak mungkin. Untuk melakukan ini, kita akan melacak sekumpulan penugasan yang berkonflik dengan beberapa nilai untuk SA. Kumpulan tersebut (dalam kasus ini $\{Q = red, NSW = green, V = blue, \}$), disebut sebagai set konflik untuk SA. Metode backjumping melakukan backtracking ke penugasan terbaru dalam set konflik; dalam kasus ini, backjumping akan melompati Tasmania dan mencoba nilai baru untuk V. Metode ini mudah diimplementasikan dengan modifikasi pada BACKTRACK sehingga ia mengakumulasikan set konflik sambil memeriksa nilai yang sah untuk ditetapkan. Jika tidak ditemukan nilai yang sah, algoritme tersebut harus mengembalikan elemen terbaru dari set konflik tersebut beserta indikator kegagalan.

Pembaca yang jeli akan menyadari bahwa pemeriksaan maju dapat menyediakan set konflik tanpa pekerjaan tambahan: setiap kali pemeriksaan maju berdasarkan penugasan $X = x$ menghapus nilai dari domain Y, pemeriksaan tersebut harus menambahkan $X = x$ ke set konflik Y. Jika nilai terakhir dihapus dari domain Y, maka penugasan dalam set konflik Y ditambahkan ke set konflik X. Kemudian, ketika kita sampai di Y, kita langsung tahu di mana harus melakukan backtrack jika diperlukan.

Pembaca yang jeli akan menyadari sesuatu yang aneh: backjumping terjadi ketika setiap nilai dalam domain berkonflik dengan penugasan saat ini; tetapi pemeriksaan maju mendeteksi kejadian ini dan mencegah pencarian untuk mencapai simpul tersebut! Bahkan, dapat ditunjukkan bahwa setiap cabang yang dipangkas dengan backjumping juga dipangkas dengan pemeriksaan maju. Oleh karena itu, backjumping sederhana menjadi berlebihan dalam pencarian pemeriksaan maju atau, memang, dalam pencarian yang menggunakan pemeriksaan konsistensi yang lebih kuat, seperti MAC.

Bahasa Indonesia: Meskipun ada pengamatan pada paragraf sebelumnya, ide di balik backjumping tetap bagus: untuk mundur berdasarkan alasan kegagalan. Backjumping memperhatikan kegagalan ketika domain variabel menjadi kosong, tetapi dalam banyak kasus cabang sudah hancur jauh sebelum ini terjadi. Pertimbangkan lagi penugasan parsial $\{WA = red, NSW = red\}$ (yang, dari diskusi kita sebelumnya, tidak konsisten). Misalkan kita mencoba $T = red$ berikutnya dan kemudian menetapkan $NT, Q, V, SA.$ Kita tahu bahwa tidak ada penugasan yang dapat bekerja untuk keempat variabel terakhir ini, jadi akhirnya kita kehabisan nilai untuk dicoba di NT. Sekarang, pertanyaannya adalah, di mana untuk mundur? Backjumping tidak dapat bekerja, karena NT memang memiliki nilai yang konsisten dengan variabel yang ditugaskan sebelumnya—NT tidak memiliki set konflik lengkap dari variabel sebelumnya yang menyebabkannya gagal. Akan tetapi, kita tahu bahwa keempat variabel NT, Q, V, dan SA, jika digabungkan, gagal karena sekumpulan variabel sebelumnya, yang pasti merupakan variabel yang secara langsung berkonflik dengan keempat variabel tersebut. Hal ini mengarah pada pengertian yang lebih mendalam tentang sekumpulan konflik untuk variabel seperti NT: sekumpulan variabel sebelumnya itulah yang menyebabkan NT, bersama dengan variabel-variabel berikutnya, tidak memiliki solusi yang konsisten. Dalam kasus ini, sekumpulan tersebut adalah WA dan NSW, jadi algoritme harus melakukan backtracking ke NSW dan melewati Tasmania. Algoritme backjumping yang menggunakan sekumpulan konflik yang didefinisikan dengan cara ini disebut **backjumping yang diarahkan oleh konflik**.

Sekarang kita harus menjelaskan bagaimana sekumpulan konflik baru ini dihitung. Metodenya sebenarnya cukup sederhana. Kegagalan "terminal" dari cabang pencarian selalu terjadi karena domain variabel menjadi kosong; variabel tersebut memiliki sekumpulan konflik standar. Dalam contoh kita, SA gagal, dan sekumpulan konfliknya adalah (misalnya) $\{WA, NT, Q\}$. Bahasa Indonesia: Kami kembali melompat ke Q, dan Q menyerap set konflik dari SA (minus Q itu sendiri, tentu saja) ke dalam set konflik langsungnya sendiri, yaitu $\{NT, NSW\}$; set konflik yang baru adalah $\{WA, NT, NSW\}$. Artinya, tidak ada solusi dari Q dan seterusnya, mengingat penugasan sebelumnya ke $\{WA, NT, NSW\}$. Oleh karena itu, kami kembali melacak ke NT, yang terbaru dari ini. NT menyerap $\{WA, NT, NSW\} - \{NT\}$ ke dalam set konflik langsungnya sendiri $\{WA\}$, menghasilkan $\{WA, NSW\}$ (seperti yang dinyatakan dalam paragraf sebelumnya). Sekarang algoritme kembali melompat ke NSW, seperti yang kita harapkan. Untuk meringkas: biarkan X_j menjadi variabel saat ini, dan biarkan $conf(X_j)$ menjadi set konfliknya. Jika setiap nilai yang mungkin untuk X_j gagal, kembali melompat ke variabel terbaru X_i di $conf(X_j)$, dan tetapkan

$$conf(X_i) \longleftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$

Ketika kita mencapai kontradiksi, backjumping dapat memberi tahu kita seberapa jauh kita harus mundur, jadi kita tidak membuang waktu mengubah variabel yang tidak akan memperbaiki masalah. Namun, kita juga ingin menghindari masalah yang sama lagi. Ketika pencarian mencapai kontradiksi, kita tahu bahwa beberapa subset dari set konflik bertanggung jawab atas masalah tersebut. Pembelajaran kendala adalah gagasan untuk menemukan set variabel minimum dari set konflik yang menyebabkan masalah. Set variabel ini, beserta nilai-nilai yang sesuai, disebut no-good. Kita kemudian mencatat no-good, baik dengan menambahkan kendala baru ke CSP atau dengan menyimpan cache no-good yang terpisah.

Misalnya, perhatikan status $\{WA = red, NT = green, Q = blue\}$ di baris bawah Gambar 6.6. Pemeriksaan ke depan dapat memberi tahu kita bahwa status ini adalah no-good karena tidak ada penugasan yang valid ke SA. Dalam kasus khusus ini, mencatat no-good tidak akan membantu, karena begitu kita memangkas cabang ini dari pohon pencarian, kita tidak akan pernah menemukan kombinasi ini lagi. Namun, anggaplah bahwa pohon pencarian pada Gambar 6.6 sebenarnya merupakan bagian dari pohon pencarian yang lebih besar yang dimulai dengan terlebih dahulu menetapkan nilai untuk V dan T. Maka akan bermanfaat untuk mencatat $\{WA = red, NT = green, Q = blue\}$ sebagai tidak baik karena kita akan mengalami masalah yang sama lagi untuk setiap kemungkinan set penugasan ke V dan T.

Tidak baik dapat digunakan secara efektif dengan pemeriksaan maju atau dengan lompatan mundur. Pembelajaran kendala adalah salah satu teknik terpenting yang digunakan oleh pemecah CSP modern untuk mencapai efisiensi pada masalah yang kompleks.

6.4 PENCARIAN LOKAL UNTUK CSP

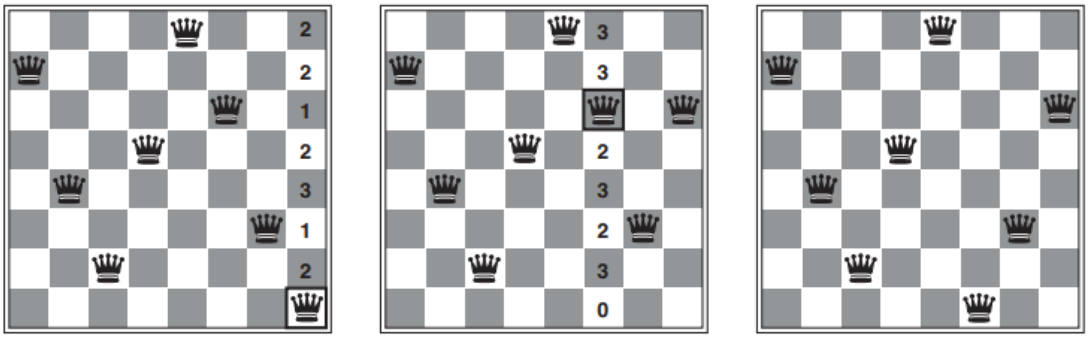
Algoritme pencarian lokal terbukti efektif dalam memecahkan banyak CSP. Algoritme tersebut menggunakan formulasi status lengkap: status awal menetapkan nilai untuk setiap variabel, dan pencarian mengubah nilai satu variabel pada satu waktu. Misalnya, dalam masalah 8 ratu (lihat Gambar 4.3), status awal mungkin berupa konfigurasi acak 8 ratu dalam 8 kolom, dan setiap langkah memindahkan satu ratu ke posisi baru di kolomnya. Biasanya, tebakan awal melanggar beberapa batasan. Inti dari pencarian lokal adalah untuk menghilangkan batasan yang dilanggar.

Dalam memilih nilai baru untuk suatu variabel, heuristik yang paling jelas adalah memilih nilai yang menghasilkan jumlah konflik minimum dengan variabel lain—heuristik konflik minimum. Algoritme tersebut ditunjukkan pada Gambar 6.8 dan penerapannya pada masalah 8 ratu didiagramkan pada Gambar 6.9.

```
function MIN-CONFLICTS(csp, max steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
            max steps, the number of steps allowed before giving up

    current ← an initial complete assignment for csp
    for i = 1 to max steps do
        if current is a solution for csp then return current
        var ← a randomly chosen conflicted variable from csp.VARIABLES
        value ← the value v for var that minimizes CONFLICTS(var , v, current, csp)
        set var = value in current
    return failure
```

Gambar 6.8 Algoritma MIN-CONFLICTS untuk menyelesaikan CSP dengan pencarian lokal. Keadaan awal dapat dipilih secara acak atau dengan proses penugasan greedy yang memilih nilai konflik minimal untuk setiap variabel secara bergantian. Fungsi CONFLICTS menghitung jumlah kendala yang dilanggar oleh nilai tertentu, mengingat sisa penugasan saat ini.

Konflik minimum secara mengejutkan efektif untuk banyak CSP. Hebatnya, pada masalah n-ratu, jika Anda tidak menghitung penempatan awal ratu, waktu proses konflik-min secara kasar tidak bergantung pada ukuran masalah. Ia bahkan memecahkan masalah sejuta ratu dalam rata-rata 50 langkah (setelah penugasan awal). Pengamatan yang luar biasa ini merupakan stimulus yang mengarah pada banyak penelitian pada tahun 1990-an tentang pencarian lokal dan perbedaan antara masalah mudah dan sulit, yang kita bahas di Bab 7. Secara kasar, n-ratu mudah untuk pencarian lokal karena solusinya terdistribusi secara padat di seluruh ruang keadaan. Konflik-min juga berfungsi dengan baik untuk masalah sulit. Misalnya, ia telah digunakan untuk menjadwalkan pengamatan untuk Teleskop Luar Angkasa Hubble, mengurangi waktu yang dibutuhkan untuk menjadwalkan pengamatan seminggu dari tiga minggu (!) menjadi sekitar 10 menit.



Gambar 6.9 Solusi dua langkah menggunakan konflik minimum untuk masalah 8 ratu. Pada setiap tahap, ratu dipilih untuk dipindahkan ke kolomnya. Jumlah konflik (dalam hal ini, jumlah ratu penyerang) ditunjukkan di setiap kotak. Algoritme memindahkan ratu ke kotak konflik minimum, memutuskan seri secara acak.

Semua teknik pencarian lokal dari Bagian 4.1 adalah kandidat untuk aplikasi ke CSP, dan beberapa di antaranya terbukti sangat efektif. Bentang alam CSP di bawah heuristik konflik minimum biasanya memiliki serangkaian plateaux. Mungkin ada jutaan penugasan variabel yang hanya berjarak satu konflik dari solusi. Pencarian plateau—yang memungkinkan perpindahan ke status lain dengan skor yang sama—dapat membantu pencarian lokal menemukan jalan keluar dari plateau ini. Pengembaraan di plateau ini dapat diarahkan dengan **pencarian tabu**: menyimpan daftar kecil status yang baru saja dikunjungi dan melarang algoritme untuk kembali ke status tersebut. Simulasi annealing juga dapat digunakan untuk keluar dari plateaux.

Teknik lain, yang disebut **pembobotan kendala**, dapat membantu memusatkan pencarian pada kendala penting. Setiap kendala diberi bobot numerik, Wi, awalnya semuanya 1. Pada setiap langkah pencarian, algoritme memilih pasangan variabel/nilai untuk diubah yang akan menghasilkan bobot total terendah dari semua kendala yang dilanggar. Bobot kemudian disesuaikan dengan menambah bobot setiap kendala yang dilanggar oleh penugasan saat ini. Hal ini memiliki dua manfaat: menambahkan topografi ke plateaux, memastikan bahwa perbaikan dari kondisi saat ini dapat dilakukan, dan juga, seiring waktu, menambah bobot pada kendala yang terbukti sulit dipecahkan.

Keuntungan lain dari pencarian lokal adalah dapat digunakan dalam pengaturan daring saat masalah berubah. Hal ini khususnya penting dalam masalah penjadwalan. Jadwal penerbangan seminggu mungkin melibatkan ribuan penerbangan dan puluhan ribu penugasan personel, tetapi cuaca buruk di satu bandara dapat membuat jadwal tidak dapat dilaksanakan. Kami ingin memperbaiki jadwal dengan jumlah perubahan minimum. Hal ini dapat dilakukan dengan mudah dengan algoritma pencarian lokal yang dimulai dari jadwal saat ini. Pencarian mundur dengan serangkaian kendala baru biasanya memerlukan lebih banyak waktu dan mungkin menemukan solusi dengan banyak perubahan dari jadwal saat ini.

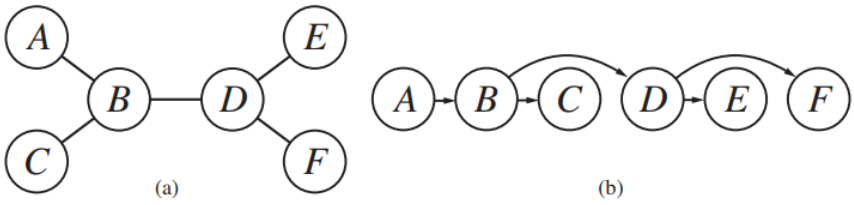
6.5 STRUKTUR MASALAH

Di bagian ini, kami meneliti cara-cara di mana struktur masalah, sebagaimana yang direpresentasikan oleh grafik kendala, dapat digunakan untuk menemukan solusi dengan cepat. Sebagian besar pendekatan di sini juga berlaku untuk masalah lain selain CSP, seperti penalaran probabilistik. Lagi pula, satu-satunya cara yang mungkin kita harapkan untuk menghadapi dunia nyata adalah dengan menguraikannya menjadi banyak submasalah. Melihat kembali grafik kendala untuk Australia (Gambar 6.1(b), diulang sebagai Gambar 6.12(a)), satu fakta menonjol: Tasmania tidak terhubung ke daratan utama. Secara intuitif, jelas bahwa pewarnaan Tasmania dan pewarnaan daratan utama adalah **submasalah yang independen**—solusi apa pun untuk daratan utama yang dikombinasikan dengan solusi apa pun untuk Tasmania menghasilkan solusi untuk seluruh peta. Independensi dapat dipastikan hanya dengan menemukan komponen-**komponen yang terhubung** dari grafik kendala. Setiap komponen sesuai dengan submasalah CSP_i. Jika penugasan S_i adalah solusi CSP_i, maka $\cup_i S_i$ adalah solusi dari $\cup_i CSP_i$. Mengapa ini penting? Pertimbangkan hal berikut: misalkan

setiap CSP_i memiliki c variabel dari total n variabel, di mana c adalah konstanta. Maka ada n/c submasalah, yang masing-masing memerlukan paling banyak pekerjaan d^c untuk diselesaikan, di mana d adalah ukuran domain. Oleh karena itu, pekerjaan total adalah $O(d^c n/c)$, yang linier dalam n; tanpa dekomposisi, pekerjaan total adalah $)(d^n)$, yang eksponensial dalam n.

Mari kita buat ini lebih konkret: membagi CSP Boolean dengan 80 variabel menjadi empat submasalah mengurangi waktu penyelesaian kasus terburuk dari masa hidup alam semesta hingga kurang dari satu detik. Submasalah yang sepenuhnya independen memang lezat, tetapi langka. Untungnya, beberapa struktur grafik lainnya juga mudah dipecahkan. Misalnya, grafik kendala adalah **pohon** ketika dua variabel apa pun dihubungkan hanya oleh satu jalur. Kami menunjukkan bahwa *setiap CSP berstruktur pohon dapat diselesaikan dalam waktu linier dalam jumlah variabel.* Kuncinya adalah gagasan baru tentang konsistensi, yang disebut **konsistensi busur terarah** atau DAC. CSP didefinisikan sebagai konsisten busur arah di bawah urutan variabel $X_1, X_2, \ldots, X_n$ jika dan hanya jika setiap X_i konsisten busur dengan setiap X_i untuk $j > i$.

Untuk menyelesaikan CSP berstruktur pohon, pertama-tama pilih variabel apa pun untuk menjadi akar pohon, dan pilih urutan variabel sedemikian rupa sehingga setiap variabel muncul setelah induknya di pohon. Urutan seperti itu disebut pengurutan topologi. Gambar 6.10(a) menunjukkan contoh pohon dan (b) menunjukkan satu kemungkinan urutan. Setiap pohon dengan n simpul memiliki $n - 1$ busur, jadi kita dapat membuat grafik ini konsisten busur arah dalam $O(n)$ langkah, yang masing-masing harus membandingkan hingga d nilai domain yang mungkin untuk dua variabel, untuk total waktu $O(nd^2)$. Setelah kita memiliki grafik konsisten busur arah, kita dapat menelusuri daftar variabel dan memilih nilai yang tersisa. Karena setiap tautan dari induk ke anaknya konsisten, kita tahu bahwa untuk nilai apa pun yang kita pilih untuk induk, akan ada nilai valid yang tersisa untuk dipilih untuk anaknya. Itu berarti kita tidak perlu mundur; kita dapat bergerak secara linear melalui variabel. Algoritme lengkap ditunjukkan pada Gambar 6.11.



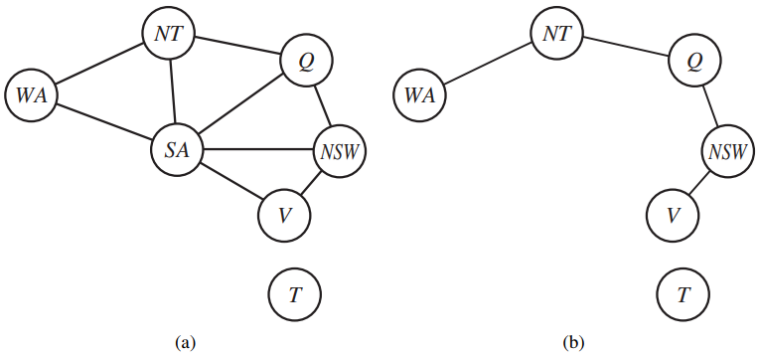
Gambar 6.10 (a) Grafik kendala dari CSP berstruktur pohon. (b) Urutan linier variabel yang konsisten dengan pohon dengan A sebagai akarnya. Ini dikenal sebagai urutan topologi variabel.

Sekarang setelah kita memiliki algoritme yang efisien untuk pohon, kita dapat mempertimbangkan apakah grafik kendala yang lebih umum dapat disederhanakan menjadi pohon. Ada dua cara utama untuk melakukannya, satu berdasarkan penghapusan simpul dan satu berdasarkan penggabungan simpul.

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
inputs: *csp*, a CSP with components X, D, C

$n \leftarrow$ number of variables in X
$assignment \leftarrow$an empty assignment
$root \leftarrow$ any variable in X
$X \leftarrow$ TOPOLOGICALSORT(X , $root$)
for $j = n$ **down to 2 do**
  MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
  if it cannot be made consistent **then return** $failure$
for $i = 1$ **to** n **do**
  $assignment[X_i] \leftarrow$any consistent value from D_i
  if there is no consistent value **then return** $failure$
return $assignment$

Gambar 6.11 Algoritma TREE-CSP-SOLVER untuk menyelesaikan CSP berstruktur pohon. Jika CSP memiliki solusi, kita akan menemukannya dalam waktu linier; jika tidak, kita akan mendeteksi kontradiksi.



Gambar 6.12 (a) Grafik kendala asli dari Gambar 6.1. (b) Grafik kendala setelah penghapusan SA.

Pendekatan pertama melibatkan penetapan nilai pada beberapa variabel sehingga variabel yang tersisa membentuk pohon. Perhatikan grafik kendala untuk Australia, yang ditunjukkan lagi pada Gambar 6.12(a). Jika kita dapat menghapus Australia Selatan, grafik akan menjadi pohon, seperti pada (b). Untungnya, kita dapat melakukannya (pada grafik, bukan benua) dengan menetapkan nilai untuk SA dan menghapus dari domain variabel lain nilai apa pun yang tidak konsisten dengan nilai yang dipilih untuk SA. Sekarang, solusi apa pun untuk CSP setelah SA dan kendalanya dihapus akan konsisten dengan nilai yang dipilih untuk SA. (Ini berfungsi untuk CSP biner; situasinya lebih rumit dengan kendala tingkat tinggi.) Oleh karena itu, kita dapat memecahkan pohon yang tersisa dengan algoritme yang diberikan di atas dan dengan demikian memecahkan seluruh masalah. Tentu saja, dalam kasus umum (berbeda dengan pewarnaan peta), nilai yang dipilih untuk SA bisa jadi salah, jadi kita perlu mencoba setiap kemungkinan nilai. Algoritme umumnya adalah sebagai berikut:

1. Pilih subset S dari variabel CSP sehingga grafik kendala menjadi pohon setelah penghapusan S. S disebut **siklus cutset**.

2. Untuk setiap kemungkinan penugasan ke variabel dalam S yang memenuhi semua kendala pada S,
    (a) hapus dari domain variabel yang tersisa nilai apa pun yang tidak konsisten dengan penugasan untuk S, dan
    (b) Jika CSP yang tersisa memiliki solusi, kembalikan bersama dengan penugasan untuk S.

Jika siklus cutset memiliki ukuran c, maka total waktu proses adalah $O(d^c \cdot (n - c)d^2)$: kita harus mencoba setiap kombinasi dc nilai untuk variabel dalam S, dan untuk setiap kombinasi kita harus memecahkan masalah pohon berukuran $n - c$. Jika grafiknya "hampir pohon," maka c akan kecil dan penghematan atas penelusuran balik langsung akan sangat besar. Namun, dalam kasus terburuk, c bisa sebesar $(n - 2)$. Menemukan set potong siklus terkecil adalah NP-keras, tetapi beberapa algoritma aproksimasi yang efisien diketahui. Pendekatan algoritmik secara keseluruhan disebut **pengkondisian set** potong, yang digunakan untuk penalaran tentang probabilitas.

Pendekatan kedua didasarkan pada konstruksi **dekomposisi pohon** dari grafik kendala menjadi satu set submasalah yang terhubung. Setiap submasalah dipecahkan secara independen, dan solusi yang dihasilkan kemudian digabungkan. Seperti kebanyakan algoritma bagi-dan-kuasai, ini berfungsi dengan baik jika tidak ada submasalah yang terlalu besar. Gambar 6.13 menunjukkan dekomposisi pohon dari masalah pewarnaan peta menjadi lima submasalah. Dekomposisi pohon harus memenuhi tiga persyaratan berikut:

- Setiap variabel dalam masalah asli muncul dalam setidaknya satu submasalah.
- Jika dua variabel dihubungkan oleh kendala dalam masalah asli, keduanya harus muncul bersama-sama (bersama dengan kendala) dalam setidaknya satu submasalah.
- Jika sebuah variabel muncul dalam dua submasalah di pohon, variabel tersebut harus muncul di setiap submasalah di sepanjang jalur yang menghubungkan submasalah tersebut.

Dua kondisi pertama memastikan bahwa semua variabel dan kendala terwakili dalam dekomposisi. Kondisi ketiga tampak agak teknis, tetapi hanya mencerminkan kendala bahwa setiap variabel yang diberikan harus memiliki nilai yang sama di setiap submasalah tempat variabel tersebut muncul; tautan yang menghubungkan submasalah di pohon memberlakukan kendala ini. Misalnya, SA muncul di keempat submasalah yang terhubung pada Gambar 6.13. Anda dapat memverifikasi dari Gambar 6.12 bahwa dekomposisi ini masuk akal.

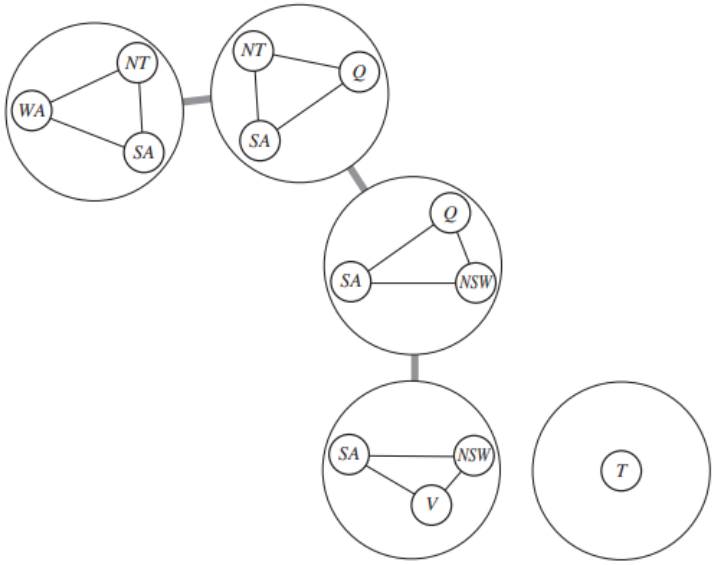
Kami memecahkan setiap submasalah secara independen; jika ada yang tidak memiliki solusi, kami tahu seluruh masalah tidak memiliki solusi. Jika kami dapat memecahkan semua submasalah, maka kami mencoba membangun solusi global sebagai berikut. Pertama, kami melihat setiap submasalah sebagai "variabel mega" yang domainnya adalah kumpulan semua solusi untuk submasalah tersebut. Misalnya, submasalah paling kiri pada Gambar 6.13 adalah masalah pewarnaan peta dengan tiga variabel dan karenanya memiliki enam solusi—satu adalah $\{WA = red, SA = blue, NT = green\}$. Kemudian, kami memecahkan kendala yang menghubungkan submasalah, menggunakan algoritma efisien untuk pohon yang diberikan sebelumnya. Kendala antara submasalah hanya menegaskan bahwa solusi submasalah setuju

pada variabel bersama mereka. Misalnya, diberikan solusi $\{WA = red, SA = blue, NT = green\}$ untuk submasalah pertama, satu-satunya solusi konsisten untuk submasalah berikutnya adalah $\{SA = blue, NT = green, Q = red\}$.

Grafik kendala yang diberikan mengakui banyak dekomposisi pohon; dalam memilih dekomposisi, tujuannya adalah untuk membuat submasalah sekecil mungkin. **Lebar pohon** dari dekomposisi pohon grafik adalah satu kurang dari ukuran submasalah terbesar; lebar pohon grafik itu sendiri didefinisikan sebagai lebar pohon minimum di antara semua dekomposisi pohonnya. Jika grafik memiliki lebar pohon w dan kita diberi dekomposisi pohon yang sesuai, maka masalah tersebut dapat dipecahkan dalam waktu $O(nd^{w+1})$. Oleh karena itu, CSP *dengan grafik kendala dengan lebar pohon terbatas dapat dipecahkan dalam waktu polinomial*. Sayangnya, menemukan dekomposisi dengan lebar pohon minimal adalah NP-keras, tetapi ada metode heuristik yang bekerja dengan baik dalam praktik.

Sejauh ini, kita telah melihat struktur grafik kendala. Mungkin ada struktur penting dalam nilai variabel juga. Pertimbangkan masalah pewarnaan peta dengan n warna. Untuk setiap solusi yang konsisten, sebenarnya ada satu set $n!$ solusi yang dibentuk dengan mengubah nama warna. Misalnya, pada peta Australia kita tahu bahwa WA, NT, dan SA semuanya harus memiliki warna yang berbeda, tetapi ada $3! = 6$ cara untuk menetapkan tiga warna ke tiga wilayah ini. Ini disebut **simetri nilai**. Kita ingin mengurangi ruang pencarian dengan faktor $n!$ dengan memecah simetri. Kita melakukan ini dengan memperkenalkan **kendala pemecahan simetri.** Untuk contoh kita, kita mungkin menerapkan kendala pemesanan sewenang-wenang, $NT < SA < WA$, yang mengharuskan tiga nilai berada dalam urutan abjad. Kendala ini memastikan bahwa hanya satu dari $n!$ solusi yang mungkin: $\{NT = blue, SA = green, WA = red\}$.

Untuk pewarnaan peta, mudah untuk menemukan kendala yang menghilangkan simetri, dan secara umum adalah mungkin untuk menemukan kendala yang menghilangkan semua kecuali satu solusi simetris dalam waktu polinomial, tetapi sangat sulit untuk menghilangkan semua simetri di antara kumpulan nilai antara selama pencarian. Dalam praktiknya, pemecahan simetri nilai telah terbukti penting dan efektif pada berbagai masalah.



Gambar 6.13 Dekomposisi pohon dari grafik kendala pada Gambar 6.12(a).

6.6 RINGKASAN

- Masalah pemenuhan kendala (CSP) merepresentasikan suatu keadaan dengan serangkaian pasangan variabel/nilai dan merepresentasikan kondisi untuk suatu solusi dengan serangkaian kendala pada variabel. Banyak masalah dunia nyata yang penting dapat dideskripsikan sebagai CSP.
- Sejumlah teknik inferensi menggunakan kendala untuk menyimpulkan pasangan variabel/nilai mana yang konsisten dan mana yang tidak. Ini termasuk node, arc, path, dan k-consistency.
- Pencarian backtracking, suatu bentuk pencarian depth-first, umumnya digunakan untuk memecahkan CSP. Inferensi dapat dijalin dengan pencarian.
- Nilai-nilai minimum yang tersisa dan heuristik derajat adalah metode domain-independen untuk memutuskan variabel mana yang akan dipilih berikutnya dalam pencarian backtracking. Heuristik nilai-paling-tidak-memaksa membantu dalam memutuskan nilai mana yang akan dicoba terlebih dahulu untuk variabel tertentu. Backtracking terjadi ketika tidak ada penugasan legal yang dapat ditemukan untuk suatu variabel. Lompatan balik yang diarahkan oleh konflik menelusuri kembali secara langsung ke sumber masalah.
- Pencarian lokal menggunakan heuristik konflik-min juga telah diterapkan pada masalah pemenuhan kendala dengan sangat sukses.
- Kompleksitas penyelesaian CSP sangat terkait dengan struktur grafik kendalanya. Masalah berstruktur pohon dapat diselesaikan dalam waktu linier. Pengondisian set potongan dapat mereduksi CSP umum menjadi CSP berstruktur pohon dan cukup efisien jika set potongan kecil dapat ditemukan. Teknik dekomposisi pohon mengubah CSP menjadi pohon submasalah dan efisien jika lebar pohon grafik kendala kecil.

BAB 7
AGEN LOGIS

Manusia, tampaknya, mengetahui berbagai hal; dan apa yang mereka ketahui membantu mereka melakukan berbagai hal. Ini bukan pernyataan kosong. Mereka membuat klaim kuat tentang bagaimana kecerdasan manusia dicapai—bukan melalui mekanisme refleks semata, tetapi melalui proses penalaran yang beroperasi pada **representasi** internal pengetahuan. Dalam AI, pendekatan terhadap kecerdasan ini diwujudkan dalam **agen berbasis pengetahuan**.

Agen pemecahan masalah dari Bab 3 dan 4 mengetahui berbagai hal, tetapi hanya dalam pengertian yang sangat terbatas dan tidak fleksibel. Misalnya, model transisi untuk teka-teki 8—pengetahuan tentang apa yang dilakukan tindakan—tersembunyi di dalam kode khusus domain dari fungsi RESULT. Ini dapat digunakan untuk memprediksi hasil tindakan, tetapi tidak untuk menyimpulkan bahwa dua petak tidak dapat menempati ruang yang sama atau bahwa status dengan paritas ganjil tidak dapat dicapai dari status dengan paritas genap. Representasi atomik yang digunakan oleh agen pemecahan masalah juga sangat terbatas. Dalam lingkungan yang dapat diamati sebagian, satu-satunya pilihan agen untuk merepresentasikan apa yang diketahuinya tentang keadaan saat ini adalah dengan mencantumkan semua kemungkinan keadaan konkret—prospek yang tidak ada harapan dalam lingkungan yang besar.

Bab 6 memperkenalkan gagasan merepresentasikan keadaan sebagai penugasan nilai ke variabel; ini adalah langkah ke arah yang benar, yang memungkinkan beberapa bagian agen bekerja dengan cara yang tidak bergantung pada domain dan memungkinkan algoritme yang lebih efisien. Dalam bab ini dan bab-bab berikutnya, kita mengambil langkah ini hingga ke kesimpulan logisnya, bisa dikatakan—kita mengembangkan logika sebagai kelas representasi umum untuk mendukung agen berbasis pengetahuan. Agen semacam itu dapat menggabungkan dan menggabungkan kembali informasi agar sesuai dengan berbagai tujuan. Sering kali, proses ini dapat sangat jauh dari kebutuhan saat ini—seperti ketika seorang matematikawan membuktikan teorema atau seorang astronom menghitung harapan hidup bumi. Agen berbasis pengetahuan dapat menerima tugas baru dalam bentuk tujuan yang dijelaskan secara eksplisit; mereka dapat mencapai kompetensi dengan cepat dengan diberi tahu atau mempelajari pengetahuan baru tentang lingkungan; dan mereka dapat beradaptasi dengan perubahan lingkungan dengan memperbarui pengetahuan yang relevan.

7.1 AGEN BERBASIS PENGETAHUAN

Komponen utama agen berbasis pengetahuan adalah basis pengetahuannya, atau KB. Basis pengetahuan adalah serangkaian kalimat. (Di sini, "$SENTENCE$" digunakan sebagai istilah teknis. Istilah ini terkait tetapi tidak identik dengan kalimat dalam bahasa Inggris dan bahasa alami lainnya.) Setiap kalimat diungkapkan dalam bahasa yang disebut bahasa representasi pengetahuan dan mewakili beberapa pernyataan tentang dunia. Terkadang kita

menyebut kalimat dengan nama aksioma, ketika kalimat tersebut dianggap sudah ada tanpa diturunkan dari kalimat lain.

Pasti ada cara untuk menambahkan kalimat baru ke basis pengetahuan dan cara untuk menanyakan apa yang diketahui. Nama standar untuk operasi ini masing-masing adalah TELL dan ASK. Kedua operasi tersebut dapat melibatkan inferensi—yaitu, memperoleh kalimat baru dari kalimat lama. Inferensi harus mematuhi persyaratan bahwa ketika seseorang TELLed pertanyaan dari basis pengetahuan, jawabannya harus mengikuti dari apa yang telah diberitahukan (atau DIBERITAHUKAN) ke basis pengetahuan sebelumnya. Nanti di bab ini, kita akan membahas lebih rinci tentang kata penting "follow". Untuk saat ini, anggaplah bahwa proses inferensi tidak boleh mengarang hal-hal yang tidak diinginkan saat berjalan.

Gambar 7.1 menunjukkan garis besar program agen berbasis pengetahuan. Seperti semua agen kita, program ini mengambil persepsi sebagai input dan mengembalikan tindakan. Agen mempertahankan basis pengetahuan, KB, yang mungkin awalnya berisi beberapa **pengetahuan latar belakang.**

Setiap kali program agen dipanggil, ia melakukan tiga hal. Pertama, ia TELLs basis pengetahuan tentang apa yang ia rasakan. Kedua, ia ASKs basis pengetahuan tentang tindakan apa yang harus dilakukan. Dalam proses menjawab pertanyaan ini, penalaran ekstensif dapat dilakukan tentang keadaan dunia saat ini, tentang hasil dari kemungkinan urutan tindakan, dan seterusnya. Ketiga, program agen TELLs basis pengetahuan tentang tindakan yang dipilih, dan agen menjalankan tindakan tersebut.

Detail bahasa representasi disembunyikan di dalam tiga fungsi yang mengimplementasikan antarmuka antara sensor dan aktuator di satu sisi dan sistem representasi dan penalaran inti di sisi lain. $\text{MAKE}-\text{PERCEPT}-\text{SENTENCE}$ menyusun kalimat yang menegaskan bahwa agen mempersepsikan persepsi yang diberikan pada waktu yang diberikan. $\text{MAKE}-\text{ACTION}-\text{QUERY}$ menyusun kalimat yang menanyakan tindakan apa yang harus dilakukan pada waktu saat ini. Terakhir, $\text{MAKE}-\text{ACTION}-\text{SENTENCE}$ menyusun kalimat yang menegaskan bahwa tindakan yang dipilih telah dilaksanakan. Rincian mekanisme inferensi disembunyikan di dalam TELL dan ASK. Bagian selanjutnya akan mengungkap rincian ini.

Agen pada Gambar 7.1 tampak sangat mirip dengan agen dengan status internal yang dijelaskan pada Bab 2. Namun, karena definisi TELL dan ASK, agen berbasis pengetahuan bukanlah program sembarangan untuk menghitung tindakan. Agen ini sesuai dengan deskripsi pada tingkat pengetahuan, di mana kita hanya perlu menentukan apa yang diketahui agen dan apa tujuannya, untuk memperbaiki perilakunya. Misalnya, taksi otomatis mungkin memiliki tujuan untuk membawa penumpang dari San Francisco ke Marin County dan mungkin tahu bahwa Jembatan Golden Gate adalah satu-satunya penghubung antara kedua lokasi tersebut. Kemudian kita dapat mengharapkannya untuk menyeberangi Jembatan Golden Gate karena ia tahu bahwa itu akan mencapai tujuannya. Perhatikan bahwa analisis ini tidak bergantung pada cara kerja taksi di tingkat implementasi. Tidak masalah apakah pengetahuan geografisnya diimplementasikan sebagai daftar tertaut atau peta piksel, atau apakah ia bernalar dengan

memanipulasi rangkaian simbol yang disimpan dalam register atau dengan menyebarkan sinyal bising dalam jaringan neuron.

Agen berbasis pengetahuan dapat dibangun hanya dengan TELLs apa yang perlu diketahuinya. Dimulai dengan basis pengetahuan yang kosong, perancang agen dapat TELLs kalimat satu per satu hingga agen tahu cara beroperasi di lingkungannya. Ini disebut pendekatan deklaratif untuk membangun sistem. Sebaliknya, pendekatan prosedural mengodekan perilaku yang diinginkan secara langsung sebagai kode program. Pada tahun 1970-an dan 1980-an, pendukung kedua pendekatan tersebut terlibat dalam perdebatan sengit. Kita sekarang memahami bahwa agen yang sukses sering kali menggabungkan elemen deklaratif dan prosedural dalam desainnya, dan bahwa pengetahuan deklaratif sering kali dapat dikompilasi menjadi kode prosedural yang lebih efisien. Kita juga dapat menyediakan agen berbasis pengetahuan dengan mekanisme yang memungkinkannya belajar sendiri.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time
  TELL(KB, MAKE-PERCEPT-SENTENCE(percept,t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action,t))
  t ← t + 1
  return action
```

Gambar7.1 Agen berbasis pengetahuan generik. Diberikan sebuah persepsi, agen menambahkan persepsi tersebut ke basis pengetahuannya, meminta basis pengetahuan untuk melakukan tindakan terbaik, dan memberi tahu basis pengetahuan bahwa ia telah melakukan tindakan tersebut.

7.2 DUNIA WUMPUS

Di bagian ini, kami menggambarkan lingkungan tempat agen berbasis pengetahuan dapat menunjukkan nilai mereka. **Dunia wumpus** adalah gua yang terdiri dari ruangan-ruangan yang dihubungkan oleh lorong-lorong. Di suatu tempat di dalam gua itu, bersembunyi wumpus yang mengerikan, binatang buas yang memakan siapa saja yang memasuki ruangannya. Wumpus dapat ditembak oleh agen, tetapi agen tersebut hanya memiliki satu anak panah. Beberapa ruangan berisi lubang tanpa dasar yang akan menjebak siapa saja yang masuk ke ruangan-ruangan ini (kecuali wumpus, yang terlalu besar untuk jatuh). Satu-satunya fitur yang meringankan dari lingkungan yang suram ini adalah kemungkinan menemukan setumpuk emas. Meskipun dunia wumpus agak jinak menurut standar permainan komputer modern, dunia ini menggambarkan beberapa poin penting tentang kecerdasan.

Contoh dunia wumpus ditunjukkan pada Gambar 7.2. Definisi yang tepat dari lingkungan tugas diberikan, seperti yang disarankan di Bagian 2.3, oleh deskripsi PEAS:

- **Ukuran kinerja**: +1000 untuk memanjat keluar dari gua dengan emas, –1000 untuk jatuh ke dalam lubang atau dimakan oleh wumpus, –1 untuk setiap tindakan yang

diambil dan –10 untuk menghabiskan anak panah. Permainan berakhir ketika agen mati atau ketika agen memanjat keluar dari gua.

- **Lingkungan**: Kotak-kotak kamar berukuran 4 × 4. Agen selalu memulai di kotak berlabel [1,1], menghadap ke kanan. Lokasi emas dan wumpus dipilih secara acak, dengan distribusi seragam, dari kotak-kotak selain kotak awal. Selain itu, setiap kotak selain awal dapat berupa lubang, dengan probabilitas 0,2.
- **Aktuator**: Agen dapat bergerak Maju, Belok Kiri sejauh 90^o, atau Belok Kanan sejauh 90^o . Agen akan mati dengan menyedihkan jika memasuki petak yang berisi lubang atau wumpus hidup. (Memasuki petak dengan wumpus mati aman, meskipun bau.) Jika agen mencoba bergerak maju dan menabrak dinding, maka agen tidak akan bergerak. Tindakan Grab dapat digunakan untuk mengambil emas jika berada di petak yang sama dengan agen. Tindakan Shoot dapat digunakan untuk menembakkan anak panah dalam garis lurus ke arah yang dihadapi agen. Anak panah terus melesat hingga mengenai (dan karenanya membunuh) wumpus atau mengenai dinding. Agen hanya memiliki satu anak panah, jadi hanya tindakan Shoot pertama yang memiliki efek. Terakhir, tindakan Climb dapat digunakan untuk memanjat keluar dari gua, tetapi hanya dari petak [1,1].
- **Sensor**: Agen memiliki lima sensor, yang masing-masing memberikan sedikit informasi:
  - –Di petak yang berisi wumpus dan di petak yang berdekatan secara langsung (tidak diagonal), agen akan merasakan Bau Busuk.
  - –Di petak yang berbatasan langsung dengan lubang, agen akan merasakan Angin Semilir.
  - –Di petak tempat emas berada, agen akan merasakan Kilauan.
  - – Saat agen berjalan ke dinding, ia akan merasakan Benturan.
  - – Saat wumpus terbunuh, ia mengeluarkan Jeritan memilukan yang dapat dirasakan di mana saja di dalam gua.

  Persepsi akan diberikan ke program agen dalam bentuk daftar lima simbol; misalnya, jika ada bau busuk dan angin sepoi-sepoi, tetapi tidak ada kilauan, benturan, atau jeritan, program agen akan mendapatkan [*Bau, Angin Semilir, Tidak Ada, Tidak Ada, Tidak Ada*].

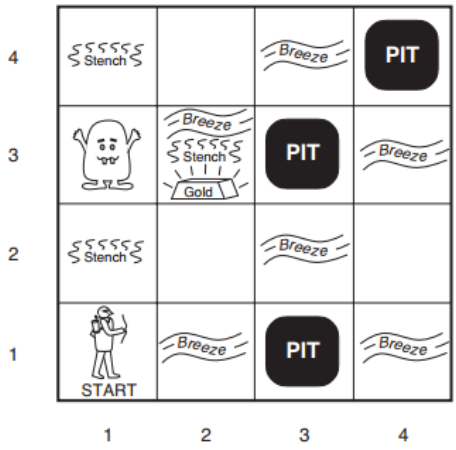
Kita dapat mengkarakterisasi lingkungan wumpus di sepanjang berbagai dimensi yang diberikan dalam Bab 2. Jelas, lingkungan tersebut bersifat diskret, statis, dan agen tunggal. (Untungnya, wumpus tidak bergerak.) Lingkungan tersebut berurutan, karena hadiah mungkin datang hanya setelah banyak tindakan diambil. Lingkungan tersebut dapat diamati sebagian, karena beberapa aspek keadaan tidak dapat dilihat secara langsung: lokasi agen, kondisi kesehatan wumpus, dan ketersediaan anak panah. Mengenai lokasi lubang dan wumpus: kita dapat memperlakukannya sebagai bagian keadaan yang tidak teramati yang kebetulan tidak dapat diubah—dalam hal ini, model transisi untuk lingkungan tersebut sepenuhnya diketahui; atau kita dapat mengatakan bahwa model transisi itu sendiri tidak diketahui karena agen tidak mengetahui tindakan *Forward* mana yang fatal—dalam hal ini, menemukan lokasi lubang dan wumpus melengkapi pengetahuan agen tentang model transisi.

Bagi agen di lingkungan tersebut, tantangan utamanya adalah ketidaktahuannya tentang konfigurasi lingkungan tersebut; mengatasi ketidaktahuan ini tampaknya memerlukan penalaran logis. Dalam sebagian besar contoh dunia wumpus, agen tersebut dapat mengambil emas dengan aman. Kadang-kadang, agen tersebut harus memilih antara pulang dengan tangan hampa dan mempertaruhkan kematian untuk menemukan emas tersebut. Sekitar 21% lingkungan tersebut sama sekali tidak adil, karena emas tersebut berada di dalam lubang atau dikelilingi oleh lubang.

Mari kita saksikan agen wumpus berbasis pengetahuan menjelajahi lingkungan yang ditunjukkan pada Gambar 7.2. Kami menggunakan bahasa representasi pengetahuan informal yang terdiri dari penulisan simbol-simbol dalam kotak (seperti pada Gambar 7.3 dan 7.4).

Basis pengetahuan awal agen tersebut berisi aturan-aturan lingkungan, seperti yang dijelaskan sebelumnya; khususnya, ia mengetahui bahwa ia berada di [1,1] dan bahwa [1,1] adalah kotak yang aman; kami menyatakannya dengan "A" dan "OK," masing-masing, di kotak [1,1].

Persepsi pertama adalah [*None, None, None, None, None*], yang darinya agen dapat menyimpulkan bahwa kotak-kotak tetangganya, [1,2] dan [2,1], bebas dari bahaya—kotak-kotak tersebut baik-baik saja. Gambar 7.3(a) menunjukkan status pengetahuan agen pada titik ini.

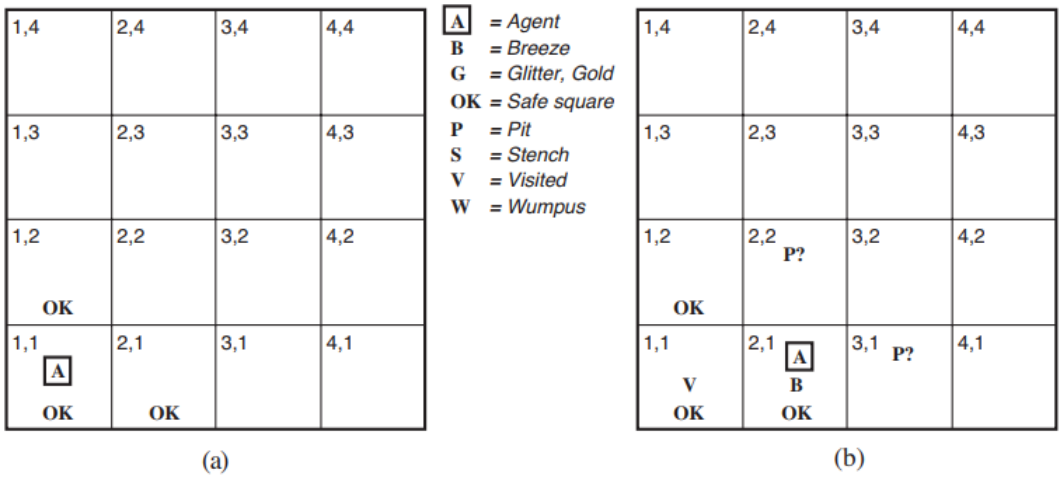


Gambar 7.2 Dunia wumpus pada umumnya. Agen berada di pojok kiri bawah, menghadap ke kanan.

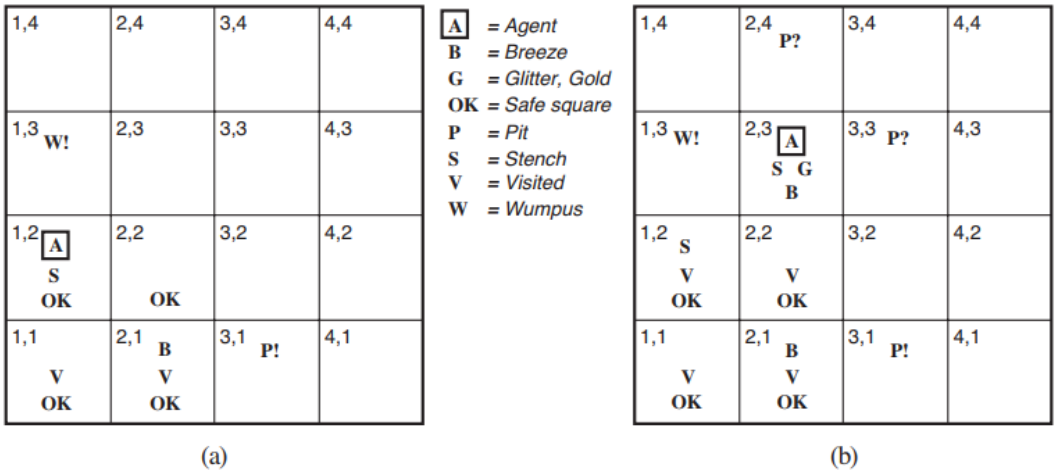
Agen yang berhati-hati hanya akan bergerak ke petak yang diketahuinya aman. Mari kita asumsikan agen memutuskan untuk bergerak maju ke [2,1]. Agen merasakan angin sepoi-sepoi (dilambangkan dengan "B") di [2,1], jadi pasti ada lubang di petak tetangga. Lubang itu tidak mungkin berada di [1,1], menurut aturan permainan, jadi pasti ada lubang di [2,2] atau [3,1] atau keduanya. Notasi "P?" pada Gambar 7.3(b) menunjukkan kemungkinan lubang di petak-petak tersebut. Pada titik ini, hanya ada satu petak yang diketahui aman dan belum dikunjungi. Jadi agen yang bijaksana akan berbalik, kembali ke [1,1], lalu melanjutkan ke [1,2].

Agen merasakan bau busuk di [1,2], yang menghasilkan keadaan pengetahuan yang ditunjukkan pada Gambar 7.4(a). Bau busuk di [1,2] berarti pasti ada wumpus di dekatnya. Namun, menurut aturan permainan, wumpus tidak dapat berada di [1,1] dan tidak dapat

berada di [2,2] (atau agen akan mendeteksi bau busuk saat berada di [2,1]). Oleh karena itu, agen dapat menyimpulkan bahwa wumpus berada di [1,3]. Notasi W! menunjukkan kesimpulan ini. Selain itu, tidak adanya angin di [1,2] menyiratkan bahwa tidak ada lubang di [2,2]. Namun, agen telah menyimpulkan bahwa pasti ada lubang di [2,2] atau [3,1], jadi ini berarti lubang itu pasti berada di [3,1]. Ini adalah kesimpulan yang cukup sulit, karena menggabungkan pengetahuan yang diperoleh di waktu yang berbeda di tempat yang berbeda dan bergantung pada tidak adanya persepsi untuk membuat satu langkah penting.



Gambar 7.3 Langkah pertama yang diambil oleh agen di dunia wumpus. (a) Situasi awal, setelah persepsi [$None, None, None, None, None$]. (b) Setelah satu gerakan, dengan persepsi [$None, Breeze, None, None, None$].



Gambar 7.4 Dua tahap selanjutnya dalam perkembangan agen. (a) Setelah langkah ketiga, dengan persepsi [$Stench, None, None, None, None$]. (b) Setelah langkah kelima, dengan persepsi [$Stench, Breeze, Glitter, None, None$].

Agen kini telah membuktikan kepada dirinya sendiri bahwa tidak ada lubang atau lubang di [2,2], jadi tidak apa-apa untuk pindah ke sana. Kami tidak menunjukkan status pengetahuan agen di [2,2]; kami hanya berasumsi bahwa agen berbalik dan pindah ke [2,3], sehingga menghasilkan Gambar 7.4(b). Di [2,3], agen mendeteksi kilauan, jadi ia harus

mengambil emas dan kemudian kembali ke rumah. Perhatikan bahwa dalam setiap kasus di mana agen menarik kesimpulan dari informasi yang tersedia, kesimpulan itu dijamin benar jika informasi yang tersedia benar. Ini adalah sifat dasar penalaran logis. Di sisa bab ini, kami menjelaskan cara membangun agen logis yang dapat mewakili informasi dan menarik kesimpulan seperti yang dijelaskan dalam paragraf sebelumnya.

7.3 LOGIKA

Bagian ini merangkum konsep dasar representasi dan penalaran logis. Ide-ide indah ini tidak bergantung pada bentuk-bentuk logika tertentu. Oleh karena itu, kami menunda detail teknis bentuk-bentuk tersebut hingga bagian berikutnya, dengan menggunakan contoh aritmatika biasa yang sudah dikenal.

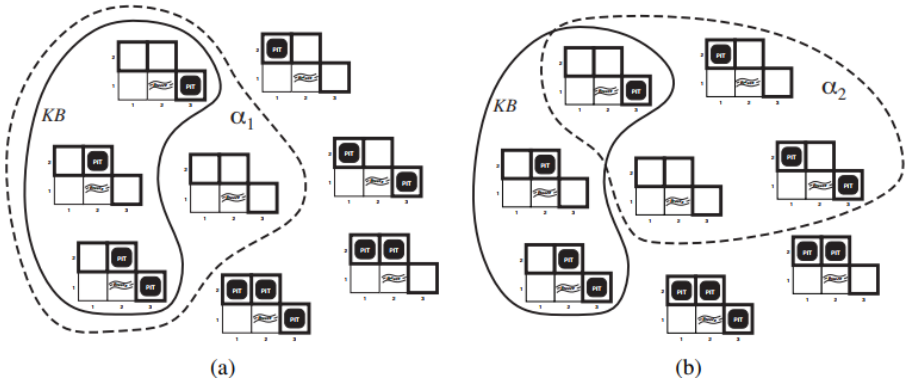
Di Bagian 7.1, kami mengatakan bahwa basis pengetahuan terdiri dari kalimat-kalimat. Kalimat-kalimat ini diungkapkan menurut sintaksis bahasa representasi, yang menentukan semua kalimat yang terbentuk dengan baik. Gagasan sintaksis cukup jelas dalam aritmatika biasa: "$x + y$ = 4" adalah kalimat yang terbentuk dengan baik, sedangkan "$x4y+$ =" tidak.

Logika juga harus mendefinisikan semantik atau makna kalimat. Semantik mendefinisikan kebenaran setiap kalimat sehubungan dengan setiap kemungkinan dunia. Misalnya, semantik untuk aritmatika menetapkan bahwa kalimat "$x + y = 4$" adalah benar di dunia tempat x adalah 2 dan y adalah 2, tetapi salah di dunia tempat x adalah 1 dan y adalah 1. Dalam logika standar, setiap kalimat harus benar atau salah di setiap kemungkinan dunia—tidak ada "di antara keduanya."

Ketika kita perlu bersikap tepat, kita menggunakan istilah **model** sebagai ganti "kemungkinan dunia." Sementara kemungkinan dunia dapat dianggap sebagai lingkungan (yang berpotensi) nyata tempat agen mungkin berada atau mungkin tidak berada, model adalah abstraksi matematis, yang masing-masing hanya memperbaiki kebenaran atau kesalahan setiap kalimat yang relevan. Secara informal, kita dapat menganggap kemungkinan dunia sebagai, misalnya, memiliki x pria dan y wanita yang duduk di meja bermain bridge, dan kalimat $x + y = 4$ adalah benar ketika totalnya ada empat orang. Secara formal, kemungkinan model hanyalah semua kemungkinan penugasan bilangan riil ke variabel x dan y. Setiap penugasan tersebut menentukan kebenaran setiap kalimat aritmatika yang variabelnya adalah x dan y. Jika kalimat α benar dalam model m, kita katakan bahwa m **memenuhi** α atau terkadang m **adalah model** α. Kita menggunakan notasi $M(\alpha)$ untuk mengartikan himpunan semua model α.

Sekarang setelah kita memiliki pengertian tentang kebenaran, kita siap untuk berbicara tentang penalaran logis. Ini melibatkan hubungan konsekuensi logis antara kalimat—gagasan bahwa sebuah kalimat mengikuti secara logis dari kalimat lain. Dalam notasi matematika, kita menulis

$$\boldsymbol{\alpha} \models \boldsymbol{\beta}$$



Gambar 7.5 Model-model yang mungkin untuk keberadaan lubang-lubang di kotak [1,2], [2,2], dan [3,1]. KB yang sesuai dengan pengamatan tidak ada apa-apa di [1,1] dan ada angin sepoi-sepoi di [2,1] ditunjukkan oleh garis utuh. (a) Garis putus-putus menunjukkan model α_1 (tidak ada lubang di [1,2]). (b) Garis putus-putus menunjukkan model α_2 (tidak ada lubang di [2,2]).

yang berarti bahwa kalimat α mensyaratkan kalimat β. Definisi formal dari pensyaratan adalah ini: $\alpha \models \beta$ jika dan hanya jika, dalam setiap model di mana α benar, β juga benar. Dengan menggunakan notasi yang baru saja diperkenalkan, kita dapat menulis

$$\alpha \models \beta \; if \; and \; only \; if \; M(\alpha) \subseteq M(\beta) \, .$$

(Perhatikan arah ⊆ di sini: jika $\alpha \models \beta$, maka α adalah pernyataan yang lebih kuat daripada β: pernyataan ini mengesampingkan lebih banyak kemungkinan dunia.) Hubungan konsekuensi sudah dikenal dalam aritmatika; kita senang dengan gagasan bahwa kalimat $x = 0$ mensyaratkan kalimat xy = 0. Jelas, dalam model apa pun di mana x adalah nol, maka xy adalah nol (terlepas dari nilai y).

Kita dapat menerapkan jenis analisis yang sama pada contoh penalaran dunia wumpus yang diberikan di bagian sebelumnya. Pertimbangkan situasi pada Gambar 7.3(b): agen tidak mendeteksi apa pun di [1,1] dan angin sepoi-sepoi di [2,1]. Persepsi ini, dikombinasikan dengan pengetahuan agen tentang aturan dunia wumpus, membentuk KB. Agen tertarik (antara lain) pada apakah kotak yang berdekatan [1,2], [2,2], dan [3,1] berisi lubang. Masing-masing dari tiga kotak mungkin berisi atau tidak berisi lubang, jadi (untuk tujuan contoh ini) ada 2^3 = 8 model yang mungkin. Delapan model ini ditunjukkan pada Gambar 7.5.

KB dapat dianggap sebagai serangkaian kalimat atau sebagai kalimat tunggal yang menegaskan semua kalimat individual. KB salah dalam model yang bertentangan dengan apa yang diketahui agen—misalnya, KB salah dalam model apa pun di mana [1,2] berisi lubang, karena tidak ada angin di [1,1]. Faktanya, hanya ada tiga model di mana KB benar, dan ini ditunjukkan dikelilingi oleh garis padat pada Gambar 7.5. Sekarang mari kita pertimbangkan dua kemungkinan kesimpulan:

$$\alpha_1 = \text{“}There \; is \; no \; pit \; in \; [1,2].\text{”}$$

$$\alpha_1 = \text{“}There\ is\ no\ pit\ in\ [2,2].\text{”}$$

Kami telah mengelilingi model α_1 dan α_2 dengan garis putus-putus pada Gambar 7.5(a) dan 7.5(b), masing-masing. Dengan pemeriksaan, kami melihat yang berikut:

dalam setiap model di mana *KB* benar, α_1 juga benar.

Oleh karena itu, KB |= α_1: tidak ada pit dalam [1,2]. Kita juga dapat melihat bahwa:

dalam beberapa model di mana KB benar, α_2 salah.

Oleh karena itu, KB | $\neq \alpha_1$: agen tidak dapat menyimpulkan bahwa tidak ada pit dalam [2,2]. (Juga tidak dapat menyimpulkan bahwa ada pit dalam [2,2].)

Contoh sebelumnya tidak hanya menggambarkan konsekuensi tetapi juga menunjukkan bagaimana definisi konsekuensi dapat diterapkan untuk memperoleh kesimpulan—yaitu, untuk melakukan **inferensi logis**. Algoritma inferensi yang diilustrasikan pada Gambar 7.5 disebut **pengecekan model**, karena ia menghitung semua model yang mungkin untuk memeriksa apakah α benar dalam semua model di mana KB benar, yaitu, bahwa *M (KB) ⊆ M (α).*

Dalam memahami konsekuensi dan inferensi, mungkin akan membantu jika kita menganggap himpunan semua konsekuensi KB sebagai tumpukan jerami dan α sebagai jarum. Konsekuensi seperti jarum yang berada di tumpukan jerami; inferensi seperti menemukannya. Perbedaan ini diwujudkan dalam beberapa notasi formal: jika algoritma inferensi i dapat memperoleh α dari KB, kita menulis

$$KB \vdash_i \alpha\ ,$$

yang diucapkan “α diperoleh dari KB oleh i” atau “i memperoleh α dari KB .”

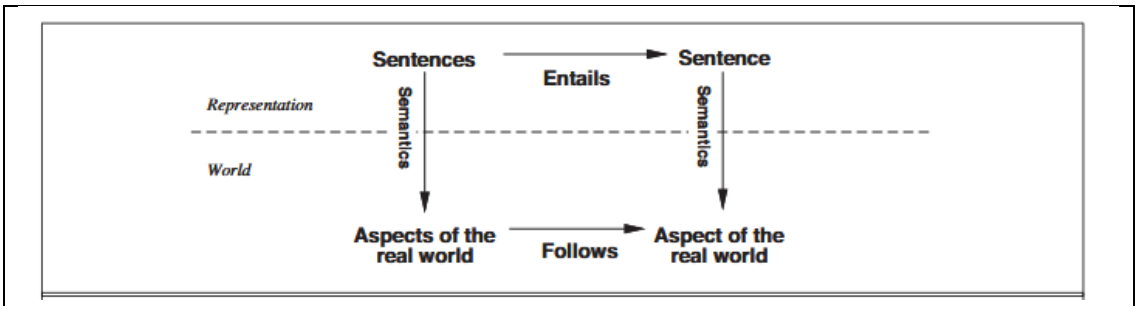
Algoritma inferensi yang hanya memperoleh kalimat-kalimat yang diinduksi disebut sebagai kalimat yang sehat atau yang mempertahankan kebenaran. Kesehatian adalah sifat yang sangat diinginkan. Prosedur inferensi yang tidak tepat pada dasarnya mengarang hal-hal yang terjadi—ia mengumumkan penemuan jarum yang tidak ada. Mudah untuk melihat bahwa pemeriksaan model, jika dapat diterapkan, adalah prosedur yang tepat.

Sifat **kelengkapan** juga diinginkan: suatu algoritme inferensi dikatakan lengkap jika dapat memperoleh kalimat apa pun yang disiratkan. Untuk tumpukan jerami sungguhan, yang luasnya terbatas, tampak jelas bahwa pemeriksaan sistematis selalu dapat memutuskan apakah jarum berada di dalam tumpukan jerami. Namun, untuk banyak basis pengetahuan, tumpukan jerami konsekuensinya tidak terbatas, dan kelengkapan menjadi masalah penting.[5] Untungnya, ada prosedur inferensi lengkap untuk logika yang cukup ekspresif untuk menangani banyak basis pengetahuan.

Kami telah menjelaskan suatu proses penalaran yang kesimpulannya dijamin benar di dunia mana pun tempat premisnya benar; khususnya, *jika KB benar di dunia nyata, maka setiap kalimat α yang diperoleh dari KB melalui prosedur inferensi yang tepat juga benar di dunia nyata.* Jadi, sementara proses inferensi beroperasi pada "sintaks"—konfigurasi fisik internal seperti bit dalam register atau pola blip listrik di otak—proses tersebut sesuai dengan

hubungan dunia nyata di mana beberapa aspek dunia nyata adalah kasus6 berdasarkan aspek lain dari dunia nyata yang menjadi kasus. Korespondensi antara dunia dan representasi ini diilustrasikan dalam Gambar 7.6.

Isu terakhir yang perlu dipertimbangkan adalah **landasan**—hubungan antara proses penalaran logis dan lingkungan nyata tempat agen berada. Secara khusus, *bagaimana kita tahu bahwa KB benar di dunia nyata?* (Bagaimanapun, KB hanyalah "sintaks" di dalam kepala agen.) Ini adalah pertanyaan filosofis yang telah banyak ditulis dalam buku. (Lihat Bab 26.) Jawaban sederhananya adalah bahwa sensor agen menciptakan hubungan tersebut. Misalnya, agen dunia wumpus kita memiliki sensor bau. Program agen menciptakan kalimat yang sesuai setiap kali ada bau. Kemudian, setiap kali kalimat itu ada di basis pengetahuan, kalimat itu benar di dunia nyata. Dengan demikian, makna dan kebenaran kalimat persepsi ditentukan oleh proses penginderaan dan konstruksi kalimat yang menghasilkannya. Bagaimana dengan pengetahuan agen lainnya, seperti keyakinannya bahwa wumpus menyebabkan bau di kotak yang berdekatan? Ini bukan representasi langsung dari satu persepsi, tetapi aturan umum—yang mungkin berasal dari pengalaman persepsi tetapi tidak identik dengan pernyataan pengalaman tersebut. Aturan umum seperti ini dihasilkan oleh proses konstruksi kalimat yang disebut pembelajaran, yang merupakan subjek Bagian V. Pembelajaran bisa saja salah. Bisa jadi wumpus menyebabkan bau *kecuali pada tanggal 29 Februari di tahun kabisat*, yaitu saat mereka mandi. Dengan demikian, KB mungkin tidak benar di dunia nyata, tetapi dengan prosedur pembelajaran yang baik, ada alasan untuk optimis.



Gambar 7.6 Kalimat merupakan konfigurasi fisik agen, dan penalaran merupakan proses membangun konfigurasi fisik baru dari konfigurasi lama. Penalaran logis harus memastikan bahwa konfigurasi baru mewakili aspek dunia yang benar-benar mengikuti aspek yang diwakili oleh konfigurasi lama.

7.4 LOGIKA PROPOSISI: LOGIKA YANG SANGAT SEDERHANA

Sekarang kami menyajikan logika sederhana tetapi kuat yang disebut logika proposisional. Kami membahas sintaksis logika proposisional dan semantiknya—cara menentukan kebenaran kalimat. Kemudian kami melihat implikasi—hubungan antara kalimat dan kalimat lain yang mengikutinya—dan melihat bagaimana hal ini mengarah pada algoritme sederhana untuk inferensi logis. Tentu saja, semuanya terjadi di dunia wumpus.

Sintaksis

Sintaksis logika proposisional mendefinisikan kalimat-kalimat yang diperbolehkan. Kalimat-kalimat atomik terdiri dari satu simbol proposisi. Setiap simbol tersebut mewakili

proposisi yang bisa benar atau salah. Kita menggunakan simbol-simbol yang dimulai dengan huruf kapital dan boleh berisi huruf-huruf lain atau subskrip, misalnya: $P, Q, R, W_{1,3}$, dan $North$. Nama-nama tersebut bersifat acak tetapi sering dipilih agar memiliki nilai mnemonik—kita menggunakan $W_{1,3}$ untuk mewakili proposisi yang menjadi wumpus dalam [1,3]. (Ingatlah bahwa simbol-simbol seperti $W_{1,3}$ bersifat atomik, yaitu, W, 1 dan 3 bukanlah bagian-bagian simbol yang bermakna.) Ada dua simbol proposisi dengan makna tetap: Benar adalah proposisi yang selalu benar dan Salah adalah proposisi yang selalu salah. **Kalimat-kalimat kompleks** dibangun dari kalimat-kalimat yang lebih sederhana, menggunakan tanda kurung dan **konjungsi logis**. Ada lima konjungsi yang umum digunakan:

- $\neg$ (tidak). Kalimat seperti $\neg W_{1,3}$ disebut **negasi** dari $W_{1,3}$. **Literal** bisa berupa kalimat atomik (**literal positif**) atau kalimat atomik yang dinegasikan (**literal negatif**).
- $\wedge$ (dan). Kalimat yang konektif utamanya adalah $\wedge$, seperti $W_{1,3} \wedge P_{3,1}$, **disebut konjungsi**; bagian-bagiannya adalah **konjungsi**. (Huruf $\wedge$ tampak seperti "A" untuk "Dan.")
- $\vee$ (atau). Kalimat yang menggunakan $\vee$, seperti $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, merupakan **disjungsi** dari **disjungsi** ($W_{1,3} \wedge P_{3,1}$) dan $W_{2,2}$. (Secara historis, $\vee$ berasal dari bahasa Latin "vel," yang berarti "atau." Bagi kebanyakan orang, lebih mudah mengingat $\vee$ sebagai $\wedge$ terbalik.)
- $\Rightarrow$ (Implikasi). Kalimat seperti $W_{1,3} \wedge P_{3,1} \Rightarrow \neg W_{2,2}$ disebut **implikasi** (atau kondisional). **Premis** atau **antesedennya** adalah $(W_{1,3} \wedge P_{3,1})$, dan **konklusi** atau **konsekuennya** adalah $\neg W_{2,2}$. Implikasi juga dikenal sebagai **aturan** atau pernyataan **jika-maka**. Simbol implikasi terkadang ditulis dalam buku-buku lain sebagai $\supset$ atau $\rightarrow$
- $\Leftrightarrow$ (jika dan hanya jika). Kalimat $W_{1,3} \Leftrightarrow \neg W_{2,2}$ adalah bikondisional. Beberapa buku lain menulis ini sebagai $\equiv$.

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} &\rightarrow \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} &\rightarrow (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
&\mid \neg\ \mathit{Sentence} \\
&\mid \mathit{Sentence} \wedge \mathit{Sentence} \\
&\mid \mathit{Sentence} \vee \mathit{Sentence} \\
&\mid \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
&\mid \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Gambar 7.7 Tata bahasa BNF (Bentuk Backus–Naur) kalimat dalam logika proposisional, beserta urutan operator, dari tertinggi ke terendah.

Gambar 7.7 memberikan tata bahasa formal logika proposisional; Tata bahasa BNF sendiri ambigu; kalimat dengan beberapa operator dapat diurai oleh tata bahasa dengan berbagai cara. Untuk menghilangkan ambiguitas, kami mendefinisikan prioritas untuk setiap

operator. Operator "tidak" (¬) memiliki prioritas tertinggi, yang berarti bahwa dalam kalimat $\neg A \wedge B$, ¬ terikat paling erat, sehingga menghasilkan padanan $\neg(A) \wedge B$ daripada $\neg(A \wedge B)$. (Notasi untuk aritmatika biasa sama: $-2 + 4$ adalah 2, bukan -6.) Jika ragu, gunakan tanda kurung untuk memastikan interpretasi yang tepat. Tanda kurung siku memiliki arti yang sama dengan tanda kurung; pilihan tanda kurung siku atau tanda kurung semata-mata untuk memudahkan manusia membaca kalimat.

Semantik

Setelah menentukan sintaksis logika proposisional, sekarang kita tentukan semantiknya. Semantik mendefinisikan aturan untuk menentukan kebenaran kalimat sehubungan dengan model tertentu. Dalam logika proposisional, model hanya menetapkan nilai kebenaran—benar atau salah—untuk setiap simbol proposisi. Misalnya, jika kalimat dalam basis pengetahuan menggunakan simbol proposisi $P_{1,2}, P_{2,2}$ dan $P_{3,1}$, maka salah satu model yang mungkin adalah

$$m_1 = \{P_{1,2} = false, P_{2,2} = false. P_{3,1} = true\}.$$

Dengan tiga simbol proposisi, ada $2^3 = 8$ model yang mungkin—tepatnya seperti yang digambarkan pada Gambar 7.5. Namun, perhatikan bahwa model tersebut murni objek matematika tanpa hubungan yang diperlukan dengan dunia wumpus. $P_{1,2}$ hanyalah sebuah simbol; itu mungkin berarti "ada lubang di [1,2]" atau "Saya di Paris hari ini dan besok." Semantik untuk logika proposisional harus menentukan cara menghitung nilai kebenaran dari setiap kalimat, yang diberikan sebuah model. Ini dilakukan secara rekursif.

Semua kalimat dibangun dari kalimat atomik dan lima konektif; oleh karena itu, kita perlu menentukan cara menghitung kebenaran kalimat atomik dan cara menghitung kebenaran kalimat yang dibentuk dengan masing-masing dari lima konektif. Kalimat atomik mudah:

- Benar adalah benar dalam setiap model dan Salah adalah salah dalam setiap model.
- Nilai kebenaran dari setiap simbol proposisi lainnya harus ditentukan secara langsung dalam model. Misalnya, dalam model m_1 yang diberikan sebelumnya, $P_{1,2}$ adalah salah.

Untuk kalimat kompleks, kita memiliki lima aturan, yang berlaku untuk setiap subkalimat P dan Q dalam setiap model m (di sini "iff" berarti "jika dan hanya jika"):

- $\neg P$ benar jika P salah dalam m.
- $P \wedge Q$ benar jika P dan Q benar dalam m.
- $P \vee Q$ benar jika P atau Q benar dalam m.
- $P \Rightarrow Q$ benar kecuali P benar dan Q salah dalam m.
- $P \Leftrightarrow Q$ benar jika P dan Q keduanya benar atau keduanya salah dalam m.

Aturan tersebut juga dapat dinyatakan dengan **tabel kebenaran** yang menentukan nilai kebenaran kalimat kompleks untuk setiap kemungkinan penugasan nilai kebenaran pada komponen-komponennya. Tabel kebenaran untuk lima konjungsi diberikan dalam Gambar 7.8. Dari tabel-tabel ini, nilai kebenaran kalimat s apa pun dapat dihitung sehubungan dengan model m apa pun dengan evaluasi rekursif sederhana. Misalnya, kalimat $\neg P_{1,2} \wedge (P_{2,2} \vee$

$P_{3,1}$), yang dievaluasi dalam m_1, menghasilkan $true \wedge (false \vee true) = true \wedge true = true$. Latihan 7.3 meminta Anda untuk menulis algoritma $PL - TRUE?(s, m)$, yang menghitung nilai kebenaran kalimat logika proposisional s dalam model m.

Tabel kebenaran untuk "and," "or," dan "not" sangat sesuai dengan intuisi kita tentang kata-kata bahasa Inggris. Poin utama yang mungkin membingungkan adalah bahwa $P \vee Q$ bernilai $true$ ketika P bernilai true atau Q bernilai true *atau keduanya*. Konjungtif lain, yang disebut "exclusive or" ("xor" singkatnya), menghasilkan $false$ ketika kedua disjunct bernilai true. Tidak ada konsensus mengenai simbol untuk $exclusive\ or$; beberapa pilihan adalah V̇ atau ≠ atau ⊕.

P	*Q*	¬P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ Q
Salah	Salah	Benar	Salah	Salah	Benar	Benar
Salah	Benar	Benar	Salah	Benar	Benar	Salah
Benar	Salah	Salah	Salah	Benar	Salah	Salah
Benar	Benar	Salah	Benar	Benar	Benar	Benar

Gambar 7.8 Tabel kebenaran untuk lima konjungsi logis. Untuk menggunakan tabel tersebut guna menghitung, misalnya, nilai $P \vee Q$ saat P benar dan Q salah, pertama-tama cari baris di sebelah kiri tempat P benar dan Q salah (baris ketiga). Kemudian cari di baris tersebut di bawah kolom $P \vee Q$ untuk melihat hasilnya: *benar*.

Tabel kebenaran untuk ⇒ mungkin tidak sepenuhnya sesuai dengan pemahaman intuitif seseorang tentang "P menyiratkan Q" atau "jika P maka Q." Untuk satu hal, logika proposisional tidak memerlukan hubungan sebab akibat atau relevansi antara P dan Q. Kalimat "5 ganjil menyiratkan Tokyo adalah ibu kota Jepang" adalah kalimat yang benar dari logika proposisional (berdasarkan interpretasi normal), meskipun itu adalah kalimat bahasa Inggris yang sangat ganjil. Hal membingungkan lainnya adalah bahwa implikasi apa pun benar jika antesedennya salah. Misalnya, "*5* genap menyiratkan Sam pintar" adalah benar, terlepas dari apakah Sam pintar atau tidak. Ini tampaknya aneh, tetapi masuk akal jika Anda menganggap "$P \Rightarrow Q$" sebagai pernyataan, "Jika P benar, maka saya mengklaim bahwa Q benar. Jika tidak, saya tidak membuat klaim apa pun." Satu-satunya cara agar kalimat ini *salah* adalah jika P benar tetapi Q salah.

Bikondisional, $P \Leftrightarrow Q$, benar jika $P \Rightarrow Q$ dan $Q \Rightarrow P$ keduanya benar. Dalam bahasa Inggris, ini sering ditulis sebagai "P jika dan hanya jika Q." Banyak aturan dalam dunia wumpus yang paling baik ditulis menggunakan ⇔. Misalnya, sebuah kotak adalah breezy jika kotak di sebelahnya memiliki lubang, dan sebuah kotak adalah breezy hanya jika kotak di sebelahnya memiliki lubang. Jadi kita memerlukan bikondisional,

$$B_1 \Leftrightarrow (P_{1,2} \vee P_{2,1}),$$

di mana $B_{1,1}$ berarti ada angin sepoi-sepoi di [1,1].

Basis pengetahuan sederhana

Sekarang setelah kita mendefinisikan semantik untuk logika proposisional, kita dapat membangun basis pengetahuan untuk dunia wumpus. Kita fokus terlebih dahulu pada aspek-aspek yang tidak dapat diubah dari dunia wumpus, dan meninggalkan aspek-aspek yang dapat diubah untuk bagian selanjutnya. Untuk saat ini, kita memerlukan simbol-simbol berikut untuk setiap lokasi $[x, y]$:

$P_{x,y}$ benar jika ada lubang di $[x, y]$.

$W_{x,y}$ benar jika ada wumpus di $[x, y]$, hidup atau mati.

$B_{x,y}$ benar jika agen merasakan angin sepoi-sepoi di $[x, y]$.

$S_{x,y}$ benar jika agen merasakan bau busuk di $[x, y]$.

Kalimat yang kita tulis akan cukup untuk memperoleh $\neg P_{1,2}$(tidak ada lubang di [1,2]), seperti yang dilakukan secara informal di Bagian 7.3. Kita memberi label setiap kalimat R_i sehingga kita dapat merujuknya:

- Tidak ada lubang di [1,1]:

  R_1: $\neg P_{1,1}$.

- Sebuah kotak berangin sepoi-sepoi jika dan hanya jika ada lubang di kotak di sebelahnya. Ini harus dinyatakan untuk setiap kotak; untuk saat ini, kita hanya menyertakan kotak yang relevan:

  R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.

  R_3: $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$.

- Kalimat sebelumnya benar di semua dunia wumpus. Sekarang kita sertakan persepsi angin untuk dua kotak pertama yang dikunjungi di dunia spesifik tempat agen berada, yang mengarah ke situasi pada Gambar 7.3(b).

  R_4: $\neg B_{1,1}$.

  R_5 : $B_{2,1}$.

Prosedur inferensi sederhana

Tujuan kita sekarang adalah memutuskan apakah $KB \models \alpha$ untuk beberapa kalimat α. Misalnya, apakah $\neg P_{1,2}$ disyaratkan oleh KB kita? Algoritme pertama kita untuk inferensi adalah pendekatan pemeriksaan model yang merupakan implementasi langsung dari definisi konsekuensi: menghitung model, dan memeriksa apakah α benar dalam setiap model di mana KB benar. Model adalah penugasan benar atau salah untuk setiap simbol proposisi. Kembali ke contoh dunia wumpus kita, simbol proposisi yang relevan adalah $B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}$ dan $P_{3,1}$. Dengan tujuh simbol, ada 2^7 = 128 model yang mungkin; dalam tiga di antaranya, *KB* benar (Gambar 7.9). Dalam ketiga model tersebut, $\neg P_{1,2}$ benar, maka tidak ada pit dalam [1,2]. Di sisi lain, $P_{2,2}$ benar dalam dua dari tiga model dan salah dalam satu model, jadi kita belum dapat mengetahui apakah ada celah dalam [2,2].

Gambar 7.9 mereproduksi dalam bentuk yang lebih tepat penalaran yang diilustrasikan dalam Gambar 7.5. Algoritma umum untuk memutuskan konsekuensi dalam logika proposisional ditunjukkan dalam Gambar 7.10. Seperti algoritma $\text{BACKTRACKING} - \text{SEARCH}, \text{TT} - \text{ENTAILS?}$ melakukan enumerasi rekursif dari ruang terbatas penugasan ke simbol. Algoritma tersebut **baik** karena mengimplementasikan secara langsung definisi konsekuensi, dan **lengkap** karena berfungsi untuk KB dan α apa pun dan selalu berakhir—hanya ada model terbatas yang harus diperiksa.

Tentu saja, "banyak terbatas" tidak selalu sama dengan "sedikit." Jika KB dan α berisi n simbol secara keseluruhan, maka ada 2^n model. Dengan demikian, kompleksitas waktu algoritma tersebut adalah $O(2^n)$. (Kompleksitas ruang hanya O(n) karena enumerasi bersifat mendalam.) Nanti dalam bab ini kami akan menunjukkan algoritme yang jauh lebih efisien dalam banyak kasus. Sayangnya, implikasi proposisional bersifat ko-NP-lengkap (yaitu, mungkin tidak lebih mudah daripada NP-lengka), *jadi setiap algoritme inferensi yang diketahui untuk logika proposisional memiliki kompleksitas kasus terburuk yang eksponensial dalam ukuran masukan.*

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Gambar 7.9 Tabel kebenaran yang dibangun untuk basis pengetahuan yang diberikan dalam teks. *KB* bernilai benar jika R_1 hingga R_5 bernilai benar, yang hanya muncul di 3 dari 128 baris (yang digarisbawahi di kolom sebelah kanan). Di semua 3 baris, $P_{1,2}$ bernilai salah, jadi tidak ada pit di [1,2]. Di sisi lain, mungkin ada (atau mungkin tidak) pit di [2,2].

```
function TT-ENTAILS?(KB,α) returns true or false
    inputs: KB, the knowledge base, a sentence in propositional logic
            α, the query, a sentence in propositional logic

    symbols ← a list of the proposition symbols in KB and α
    return TT-CHECK-ALL(KB,α, symbols, { })

function TT-CHECK-ALL(KB,α, symbols , model) returns true or false
    if EMPTY?(symbols) then
        if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
                    else return true // when KB is false, always return true
    else do
        P ← FIRST(symbols)
        rest ← REST(symbols)
return (TT-CHECK-ALL(KB,α, rest, model ∪ {P = true})
            and
            TT-CHECK-ALL(KB,α, rest, model ∪ {P = false }))
```

Gambar 7.10 Algoritma enumerasi tabel kebenaran untuk memutuskan konsekuensi proposisional. (TT adalah singkatan dari tabel kebenaran.) PL-TRUE? mengembalikan true jika kalimat berlaku dalam model. Variabel model mewakili model parsial—penugasan ke beberapa simbol. Kata kunci "dan" digunakan di sini sebagai operasi logis pada dua argumennya, yang mengembalikan $true$ atau $false$.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Gambar 7.11 Kesetaraan logika standar. Simbol α, β, dan γ merupakan kalimat acak dari logika proposisional.

7.5 PEMBUKTIAN TEOREMA PROPOSISI

Sejauh ini, kita telah menunjukkan cara menentukan konsekuensi dengan *pemeriksaan model*: menghitung model dan menunjukkan bahwa kalimat tersebut harus berlaku di semua model. Di bagian ini, kita menunjukkan bagaimana konsekuensi dapat dilakukan dengan **pembuktian teorema**—menerapkan aturan inferensi secara langsung ke kalimat-kalimat dalam basis pengetahuan kita untuk membangun pembuktian kalimat yang diinginkan tanpa

berkonsultasi dengan model. Jika jumlah model banyak tetapi panjang pembuktiannya pendek, maka pembuktian teorema dapat lebih efisien daripada pemeriksaan model.

Sebelum kita menyelami detail algoritma pembuktian teorema, kita memerlukan beberapa konsep tambahan yang terkait dengan konsekuensi. Konsep pertama adalah kesetaraan logis: dua kalimat α dan β secara logis setara jika keduanya benar dalam kumpulan model yang sama. Kita menulisnya sebagai $\alpha \equiv \beta$. Misalnya, kita dapat dengan mudah menunjukkan (menggunakan tabel kebenaran) bahwa $P \wedge Q$ dan $Q \wedge P$ secara logis setara; kesetaraan lainnya ditunjukkan pada Gambar 7.11. Kesetaraan ini memainkan peran yang sama dalam logika seperti halnya identitas aritmatika dalam matematika biasa. Definisi alternatif kesetaraan adalah sebagai berikut: dua kalimat α dan β apa pun setara hanya jika masing-masing dari mereka memerlukan yang lain:

$$\alpha \equiv \beta \; jika \; dan \; hanya \; jika \; \alpha \models \beta \; dan \; \beta \models \alpha \; .$$

Konsep kedua yang kita perlukan adalah **validitas**. Sebuah kalimat valid jika benar dalam semua model. Misalnya, kalimat $P \vee \neg P$ valid. Kalimat yang valid juga dikenal sebagai **tautologi**—kalimat tersebut tentu benar. Karena kalimat *Benar* benar dalam semua model, setiap kalimat yang valid secara logis setara dengan *Benar*. Apa gunanya kalimat yang valid? Dari definisi kita tentang implikasi, kita dapat memperoleh **teorema deduksi**, yang sudah dikenal oleh orang Yunani kuno:

Untuk setiap kalimat α dan β, $\alpha \models \beta$ jika dan hanya jika kalimat $(\alpha \Rightarrow \beta)$ valid.

Oleh karena itu, kita dapat memutuskan apakah $\alpha \models \beta$ dengan memeriksa apakah $(\alpha \Rightarrow \beta)$ benar dalam setiap model—yang pada dasarnya adalah apa yang dilakukan oleh algoritma inferensi pada Gambar 7.10—atau dengan membuktikan bahwa $(\alpha \Rightarrow \beta)$ setara dengan Benar. Sebaliknya, teorema deduksi menyatakan bahwa setiap kalimat implikasi yang valid menggambarkan inferensi yang sah.

Konsep terakhir yang kita perlukan adalah Validitas. Suatu kalimat dapat dipenuhi jika benar dalam, atau dipenuhi oleh, beberapa model. Misalnya, basis pengetahuan yang diberikan sebelumnya,$(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, dapat dipenuhi karena ada tiga model yang benar, seperti yang ditunjukkan pada Gambar 7.9. Kepuasan dapat diperiksa dengan menghitung model yang mungkin hingga ditemukan satu yang memenuhi kalimat tersebut. Masalah penentuan kepuasan kalimat dalam logika proposisional—masalah **SAT**—adalah masalah pertama yang terbukti NP-lengkap. Banyak masalah dalam ilmu komputer benar-benar merupakan masalah kepuasan. Misalnya, semua masalah kepuasan kendala dalam Bab 6 menanyakan apakah kendala tersebut dapat dipenuhi oleh beberapa penugasan.

Keabsahan dan kepuasan tentu saja saling terkait: α valid jika ча tidak dapat dipenuhi; sebaliknya, α dapat dipenuhi jika $\neg\alpha$ tidak valid. Kita juga memperoleh hasil yang berguna berikut:

$$\alpha \models \beta \; jika \; dan \; hanya \; jika \; kalimat \; (\alpha \wedge \neg\beta) \; tidak \; memuaskan.$$

Membuktikan β dari α dengan memeriksa ketidakterpuasan $(\alpha \wedge \neg\beta)$ sama persis dengan teknik pembuktian matematika standar *reductio ad absurdum* (secara harfiah, "reduksi menjadi hal yang absurd"). Ini juga disebut pembuktian dengan **sanggahan** atau pembuktian dengan **kontradiksi**. Seseorang menganggap kalimat β salah dan menunjukkan bahwa ini mengarah pada kontradiksi dengan aksioma α yang diketahui. Kontradiksi ini persis seperti yang dimaksud dengan mengatakan bahwa kalimat $(\alpha \wedge \neg\beta)$ tidak dapat dipenuhi.

Inferensi dan pembuktian

Bagian ini membahas aturan inferensi yang dapat diterapkan untuk memperoleh **pembuktian**—rantai kesimpulan yang mengarah ke tujuan yang diinginkan. Aturan yang paling terkenal disebut **Modus Ponens** (bahasa Latin untuk modus yang menegaskan) dan ditulis

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta}$$

Notasi tersebut berarti bahwa, setiap kali kalimat dalam bentuk $\alpha \Rightarrow \beta$ dan α diberikan, maka kalimat β dapat disimpulkan. Misalnya, jika $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ dan $(WumpusAhead \wedge WumpusAlive)$ diberikan, maka Shoot dapat disimpulkan.

Aturan inferensi lain yang berguna adalah **Eliminasi-Dan**, yang menyatakan bahwa, dari sebuah konjungsi, konjungsi mana pun dapat disimpulkan:

$$\frac{\alpha \wedge \beta}{\alpha}$$

Misalnya, dari $(WumpusAhead \wedge WumpusAlive)$, $WumpusAlive$ dapat disimpulkan.

Dengan mempertimbangkan kemungkinan nilai kebenaran α dan β, seseorang dapat dengan mudah menunjukkan bahwa Modus Ponens dan Eliminasi-Dan adalah benar sekali dan untuk selamanya. Aturan-aturan ini kemudian dapat digunakan dalam setiap contoh tertentu di mana mereka berlaku, menghasilkan kesimpulan yang benar tanpa perlu menghitung model.

Semua kesetaraan logis dalam Gambar 7.11 dapat digunakan sebagai aturan inferensi. Misalnya, kesetaraan untuk eliminasi bikondisional menghasilkan dua aturan inferensi

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Tidak semua aturan inferensi bekerja di kedua arah seperti ini. Misalnya, kita tidak dapat menjalankan Modus Ponens di arah yang berlawanan untuk memperoleh $\alpha \Rightarrow \beta$ dan α dari β.

Mari kita lihat bagaimana aturan inferensi dan kesetaraan ini dapat digunakan di dunia wumpus. Kita mulai dengan basis pengetahuan yang berisi $\neg R_1$ hingga R_5 dan menunjukkan

cara membuktikan $P_{1,2}$, yaitu, tidak ada pit di [1,2]. Pertama, kita terapkan eliminasi bikondisional ke R_2 untuk memperoleh

$$R_6: \left(B_{1,1} \Rightarrow \left(P_{1,2} \vee P_{2,1}\right)\right) \wedge \left(\left(P_{1,2} \vee P_{2,1}\right) \Rightarrow B_{1,1}\right)$$

Kemudian kita terapkan Eliminasi-And pada R_6 untuk memperoleh

$$R_7: \left(\left(P_{1,2} \vee P_{2,1}\right) \Rightarrow B_{1,1}\right)$$

Kesetaraan logis untuk kontraposisi memberikan

$$R_8: \left(\neg B_{1,1} \Rightarrow \left(\neg P_{1,2} \vee P_{2,1}\right)\right)$$

Sekarang kita dapat menerapkan Modus Ponens dengan R_8 dan persepsi R_4 (yaitu, $\neg B_{1,1}$), untuk memperoleh

$$R_9: \neg\left(P_{1,2} \vee P_{2,1}\right)$$

Akhirnya, kita menerapkan aturan De Morgan, yang memberikan kesimpulan

$$R_{10}: \neg P_{1,2} \wedge \neg P_{2,1}$$

Artinya, baik [1,2] maupun [2,1] tidak mengandung pit.

Kita menemukan bukti ini dengan tangan, tetapi kita dapat menerapkan salah satu algoritma pencarian di Bab 3 untuk menemukan urutan langkah yang merupakan bukti. Kita hanya perlu mendefinisikan masalah bukti sebagai berikut:

- INITIAL STATE: basis pengetahuan awal.
- ACTIONS: rangkaian tindakan terdiri dari semua aturan inferensi yang diterapkan pada semua kalimat yang cocok dengan bagian atas aturan inferensi.
- RESULT: hasil dari suatu tindakan adalah menambahkan kalimat di bagian bawah aturan inferensi.
- GOAL: sasaran adalah keadaan yang memuat kalimat yang ingin kita buktikan.

Dengan demikian, pencarian bukti merupakan alternatif untuk menghitung model. Dalam banyak kasus praktis, *menemukan bukti dapat lebih efisien karena bukti dapat mengabaikan proposisi yang tidak relevan, tidak peduli berapa banyak jumlahnya*. Misalnya, bukti yang diberikan sebelumnya yang mengarah ke $\neg P_{1,2} \wedge \neg P_{2,1}$tidak menyebutkan proposisi $B_{2,1}$, $P_{1,1}$, $P_{2,2}$, atau $P_{3,1}$. Mereka dapat diabaikan karena proposisi tujuan, $P_{1,2}$, hanya muncul dalam kalimat R_2; proposisi lain dalam R_2 hanya muncul dalam R_4 dan R_2; jadi R_1, R_3, dan R_5 tidak memiliki pengaruh pada bukti. Hal yang sama akan berlaku bahkan jika kita menambahkan sejuta kalimat lagi ke basis pengetahuan; algoritma tabel kebenaran sederhana, di sisi lain, akan kewalahan oleh ledakan model eksponensial.

Salah satu sifat terakhir dari sistem logika adalah **monotonisitas**, yang menyatakan bahwa himpunan kalimat yang terkandung hanya dapat bertambah seiring dengan penambahan informasi ke dalam basis pengetahuan. Untuk kalimat α dan β,

$$if\ KB \models \alpha\ then\ KB \wedge \beta \models \alpha$$

Misalnya, anggaplah basis pengetahuan berisi pernyataan tambahan β yang menyatakan bahwa ada tepat delapan lubang di dunia. Pengetahuan ini dapat membantu agen menarik kesimpulan **tambahan**, tetapi tidak dapat membatalkan kesimpulan α yang telah disimpulkan—seperti kesimpulan bahwa tidak ada lubang di [1,2]. Monotonisitas berarti bahwa aturan inferensi dapat diterapkan setiap kali premis yang sesuai ditemukan di basis pengetahuan—kesimpulan aturan harus *mengikuti apa pun yang ada di basis pengetahuan*.

Pembuktian dengan resolusi

Kami berpendapat bahwa aturan inferensi yang dibahas sejauh ini masuk akal, tetapi kami belum membahas pertanyaan tentang *kelengkapan* untuk algoritme inferensi yang menggunakannya. Algoritme pencarian seperti pencarian pendalaman berulang (halaman 89) lengkap dalam arti bahwa mereka akan menemukan tujuan yang dapat dicapai, tetapi jika aturan inferensi yang tersedia tidak memadai, maka tujuan tersebut tidak dapat dicapai—tidak ada bukti yang hanya menggunakan aturan inferensi tersebut. Misalnya, jika kami menghapus aturan eliminasi bikondisional, pembuktian di bagian sebelumnya tidak akan berhasil. Bagian saat ini memperkenalkan satu aturan inferensi, **resolusi**, yang menghasilkan algoritme inferensi lengkap saat digabungkan dengan algoritme penelusuran lengkap apa pun.

Kita mulai dengan menggunakan versi sederhana dari aturan resolusi di dunia wumpus. Mari kita pertimbangkan langkah-langkah yang mengarah ke Gambar 7.4(a): agen kembali dari [2,1] ke [1,1] lalu pergi ke [1,2], di mana ia merasakan bau busuk, tetapi tidak ada angin. Kita tambahkan fakta berikut ke basis pengetahuan:

$R_{11}: \neg B_{1,2}$

$R_{12}: B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$

Melalui proses yang sama yang menghasilkan R_{10} sebelumnya, kita sekarang dapat menyimpulkan tidak adanya lubang di [2,2] dan [1,3] (ingat bahwa [1,1] sudah diketahui tidak memiliki lubang):

$R_{13}: \neg P_{2,2}$

$R_{14}: \neg P_{1,3}$

Kita juga dapat menerapkan eliminasi bikondisional pada R_3, diikuti oleh Modus Ponens dengan R_5, untuk memperoleh fakta bahwa terdapat pit pada [1,1], [2,2], atau [3,1]:

$R_{15}: P_{1,1} \vee P_{2,2} \vee P_{3,1}$

Sekarang datanglah aplikasi pertama dari aturan resolusi: literal $\neg P_{2,2}$ di R_{13} diselesaikan dengan literal $P_{2,2}$ di R_{15} untuk memberikan **resolven**

$R_{16}: P_{1,1} \vee P_{3,1}$

Dalam bahasa Inggris; jika ada pit di salah satu dari [1,1], [2,2], dan [3,1] dan tidak ada di [2,2], maka pit tersebut ada di [1,1] atau [3,1]. Demikian pula, literal $\neg P_{1,1}$ di R_1 beresolusi dengan literal $P_{1,1}$ di R_{16} untuk memberikan

R_{17}: $P_{3,1}$

Dalam bahasa Inggris: jika ada pit di [1,1] atau [3,1] dan tidak ada di [1,1], maka pit tersebut ada di [3,1]. Dua langkah inferensi terakhir ini adalah contoh aturan inferensi **resolusi unit**,

$$\frac{\ell_1 \vee \cdots \vee \ell_{k,} \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \cdots \vee \ell_K}$$

di mana setiap ℓ adalah literal dan $\ell 1$ dan m adalah literal yang saling melengkapi (yaitu, yang satu adalah negasi dari yang lain). Dengan demikian, aturan resolusi unit mengambil **klausa**—disjungsi literal—dan literal dan menghasilkan klausa baru. Perhatikan bahwa satu literal dapat dilihat sebagai disjungsi dari satu literal, yang juga dikenal sebagai **klausa unit.** Aturan resolusi unit dapat digeneralisasikan ke aturan **resolusi** penuh,

$$\frac{\ell_1 \vee \cdots \vee \ell_{k,} \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \cdots \vee \ell_K \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

di mana ℓ_i dan m_j adalah literal komplementer. Ini menyatakan bahwa resolusi mengambil dua klausa dan menghasilkan klausa baru yang berisi semua literal dari dua klausa asli kecuali dua literal komplementer. Misalnya, kita memiliki

$$\frac{P_{1,1} \vee P_{3,1} \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

Ada satu aspek teknis lagi dari aturan resolusi: klausa yang dihasilkan harus hanya berisi satu salinan dari setiap literal. Penghapusan beberapa salinan literal disebut **pemfaktoran**. Misalnya, jika kita menyelesaikan $(A \vee B)$ dengan $(A \vee \neg B)$, kita memperoleh $(A \vee A)$, yang direduksi menjadi hanya A.

Kebenaran aturan resolusi dapat dilihat dengan mudah dengan mempertimbangkan literal ℓ_i yang komplementer terhadap literal m_j dalam klausa lainnya. Jika ℓ_i benar, maka m_j salah, dan karenanya $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \cdots \vee m_n$ harus benar, karena $m_1 \vee \cdots \vee m_n$ diberikan. Jika ℓ_i salah, maka $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$ harus benar karena $\ell_i \vee \cdots \vee \ell_k$ diberikan. Sekarang ℓ_i benar atau salah, jadi salah satu dari kesimpulan ini berlaku—tepat seperti yang dinyatakan dalam aturan resolusi.

Yang lebih mengejutkan tentang aturan resolusi adalah bahwa aturan ini membentuk dasar bagi serangkaian prosedur inferensi lengkap. *Pembukti teorema berbasis resolusi dapat, untuk setiap kalimat α dan β dalam logika proposisional, memutuskan apakah* $\alpha \vDash \beta$. Dua subbagian berikutnya menjelaskan bagaimana resolusi mencapai hal ini.

Bentuk normal konjungtif

Aturan resolusi hanya berlaku untuk klausa (yaitu, disjungsi literal), jadi tampaknya hanya relevan untuk basis pengetahuan dan kueri yang terdiri dari klausa. Lalu, bagaimana

aturan ini dapat mengarah pada prosedur inferensi lengkap untuk semua logika proposisional? Jawabannya adalah *bahwa setiap kalimat logika proposisional secara logis setara dengan konjungsi klausa.* Kalimat yang dinyatakan sebagai konjungsi klausa dikatakan dalam **bentuk normal konjungtif** atau **CNF** (lihat Gambar 7.14). Sekarang kami menjelaskan prosedur untuk mengonversi ke CNF. Kami mengilustrasikan prosedur tersebut dengan mengubah kalimat $B_{1,1}$ e ($P_{1,2}$ V $P_{2,1}$) menjadi CNF. Langkah-langkahnya adalah sebagai berikut:

1. Hilangkan ⇔, ganti $\alpha \Leftrightarrow \beta$ dengan $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

   $$\left(B_{1,1} \Rightarrow \left(P_{1,2} \vee P_{2,1}\right)\right)\left(\left(P_{1,2} \vee P_{2,1}\right) \Rightarrow B_{1,1}\right)$$

2. Hilangkan ⇒, ganti $\alpha \Rightarrow \beta$ dengan $\neg\alpha \vee \beta$:

   $$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})(\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. CNF mengharuskan ¬ muncul hanya dalam bentuk literal, jadi kita "memindahkan ¬ ke dalam" dengan penerapan berulang dari kesetaraan berikut dari Gambar 7.11:

   $\neg(\neg\alpha) \equiv \alpha$ (Eliminasi Negasi Ganda)
   $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (De Morgan)
   $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (De Morgan)

   Dalam contoh ini, kita hanya memerlukan satu penerapan aturan terakhir:

   $$\left(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}\right) \wedge \left(\left(P_{1,2} \vee \neg P_{2,1}\right) \vee B_{1,1}\right)$$

4. Sekarang kita memiliki kalimat yang berisi operator Λ dan V bertingkat yang diterapkan pada literal. Kita terapkan hukum distribusi dari Gambar 7.11, mendistribusikan V ke Λ sedapat mungkin.

   $$\left(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}\right) \wedge \left(\neg P_{1,2} \vee B_{1,1}\right) \wedge \left(\neg P_{1,2} \vee B_{1,1}\right)$$

   Kalimat asli sekarang dalam CNF, sebagai konjungsi tiga klausa. Jauh lebih sulit dibaca, tetapi dapat digunakan sebagai input untuk prosedur resolusi.

Algoritma resolusi

Prosedur inferensi berdasarkan resolusi bekerja dengan menggunakan prinsip pembuktian dengan kontradiksi. Yaitu, untuk menunjukkan bahwa $KB \vDash \alpha$ kami menunjukkan bahwa $(KB \wedge \neg\alpha)$ tidak memuaskan. Kami melakukan ini dengan membuktikan kontradiksi. Algoritma resolusi ditunjukkan pada Gambar 7.12.

Pertama, $(KB \wedge \neg\alpha)$ diubah menjadi CNF. Kemudian, aturan resolusi diterapkan pada klausa yang dihasilkan. Setiap pasangan yang berisi literal komplementer diselesaikan untuk menghasilkan klausa baru, yang ditambahkan ke set jika belum ada. Proses berlanjut hingga salah satu dari dua hal terjadi:

- Tidak ada klausa baru yang dapat ditambahkan, dalam hal ini KB tidak memerlukan α; atau,

- dua klausa diselesaikan untuk menghasilkan klausa kosong, dalam hal ini KB memerlukan α.

Klausul kosong—disjungsi tanpa disjungsi—setara dengan *Salah* karena disjungsi hanya benar jika setidaknya salah satu disjungsinya benar. Cara lain untuk melihat bahwa klausa kosong mewakili kontradiksi adalah dengan mengamati bahwa hal itu muncul hanya dari penyelesaian dua klausa unit komplementer seperti P dan $\neg P$) .

Kita dapat menerapkan prosedur penyelesaian pada inferensi yang sangat sederhana di dunia wumpus. Ketika agen berada di [1,1], tidak ada angin sepoi-sepoi, jadi tidak akan ada lubang di kotak-kotak tetangga. Basis pengetahuan yang relevan adalah

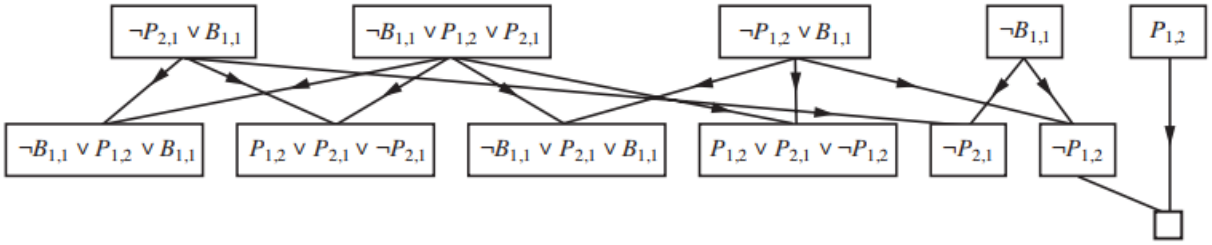
$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

dan kami ingin membuktikan α yang, katakanlah, $\neg P_{1,2}$. Ketika kami mengonversi $(KB \wedge \neg\alpha)$ke CNF, kami memperoleh klausa yang ditunjukkan di bagian atas Gambar 7.13. Baris kedua gambar menunjukkan klausa yang diperoleh dengan menyelesaikan pasangan di baris pertama. Kemudian, ketika $P_{1,2}$ diselesaikan dengan $\neg P_{1,2}$, kami memperoleh klausa kosong, yang ditunjukkan sebagai kotak kecil. Pemeriksaan Gambar 7.13 mengungkapkan bahwa banyak langkah penyelesaian tidak ada gunanya. Misalnya, klausa $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$setara dengan *True* ∨ $P_{1,2}$ yang setara dengan True. Menyimpulkan bahwa True adalah benar tidak terlalu membantu. Oleh karena itu, klausa apa pun di mana dua literal komplementer muncul dapat dibuang.

function PL-RESOLUTION(KB, α) **returns** true or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new ←{}
  loop do
    for each pair of clauses C_i, C_j **in** clauses **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if new ⊆ clauses **then return** *false*
    clauses ← *clauses* ∪*new*

Gambar 7.12 Algoritma resolusi sederhana untuk logika proposisional. Fungsi PL-RESOLVE mengembalikan himpunan semua klausa yang mungkin diperoleh dengan menyelesaikan dua masukannya.



Gambar 7.13 Penerapan sebagian PL-RESOLUTION pada inferensi sederhana di dunia wumpus. $\neg P_{1,2}$ ditunjukkan mengikuti empat klausa pertama di baris paling atas.

Kelengkapan resolusi

Untuk menyimpulkan pembahasan kita tentang resolusi, sekarang kita tunjukkan mengapa PL-RESOLUTION bersifat lengkap. Untuk melakukannya, kita perkenalkan **penutupan resolusi** $RC(S)$ dari sekumpulan klausa S, yang merupakan sekumpulan semua klausa yang dapat diturunkan dengan penerapan berulang aturan resolusi pada klausa di S atau turunannya. Penutupan resolusi adalah apa yang PL-RESOLUTION hitung sebagai nilai akhir dari *klausa* variabel. Mudah untuk melihat bahwa $RC(S)$ haruslah terbatas, karena hanya ada banyak klausa berbeda yang dapat dibangun dari simbol $P_1 \ldots P_k$ yang muncul di S. (Perhatikan bahwa ini tidak akan benar tanpa langkah pemfaktoran yang menghilangkan beberapa salinan literal.) Oleh karena itu, PL-RESOLUTION selalu berakhir.

Teorema kelengkapan untuk resolusi dalam logika proposisional disebut **teorema resolusi dasar:**

> Jika sekumpulan klausa tidak dapat dipenuhi, maka penutupan resolusi klausa tersebut memuat klausa kosong.

Teorema ini dibuktikan dengan menunjukkan kontraposisinya: jika penutup $RC(S)$ tidak memuat klausa kosong, maka S dapat dipenuhi. Bahkan, kita dapat membangun model untuk S dengan nilai kebenaran yang sesuai untuk $P_1 \ldots P_k$. Prosedur konstruksinya adalah sebagai berikut:

Untuk i dari 1 hingga k,

- Jika klausa dalam $RC(S)$ memuat literal $\neg P_i$ dan semua literal lainnya salah berdasarkan penugasan yang dipilih untuk $P_1 \ldots P_{i-1}$, maka tetapkan *salah* untuk P_i
- Jika tidak, tetapkan *benar* untuk P_i.

Penugasan untuk $P_1 \ldots P_k$ ini adalah model S. Untuk melihatnya, asumsikan kebalikannya—bahwa, pada tahap i tertentu dalam deret, penugasan simbol P_i menyebabkan beberapa klausa C menjadi salah. Agar ini terjadi, semua literal lainnya dalam C harus sudah dipalsukan oleh penugasan ke $P_1 \ldots P_{i-1}$ Dengan demikian, C sekarang harus terlihat seperti $(false \vee false \vee \cdots false \vee P_i)$ atau seperti $false \vee false \vee \cdots false \vee \neg P_i$ Jika hanya satu dari keduanya yang ada di $RC(S)$, maka algoritma akan menetapkan nilai kebenaran yang sesuai ke P_i untuk menjadikan C benar, jadi C hanya dapat dipalsukan jika kedua klausa ini ada di $RC(S)$.

Sekarang, karena $RC(S)$ ditutup di bawah resolusi, ia akan memuat resolvent dari kedua klausa ini, dan resolvent itu akan memiliki semua literalnya yang telah dipalsukan oleh penugasan ke $P_1 \ldots P_{i-1}$ Ini bertentangan dengan asumsi kita bahwa klausa pertama yang dipalsukan muncul pada tahap i. Oleh karena itu, kita telah membuktikan bahwa konstruksi tersebut tidak pernah memalsukan klausa di $RC(S)$; yaitu, ia menghasilkan model $RC(S)$ dan dengan demikian model Sitself (karena S terdapat di *RC*(S)*)*.

Klausa Horn dan klausa pasti

Kelengkapan penyelesaian menjadikannya metode inferensi yang sangat penting. Namun, dalam banyak situasi praktis, kekuatan penyelesaian penuh tidak diperlukan. Beberapa basis pengetahuan dunia nyata memenuhi batasan tertentu pada bentuk kalimat yang dikandungnya, yang memungkinkan mereka menggunakan algoritme inferensi yang lebih terbatas dan efisien.

Salah satu bentuk terbatas tersebut adalah **klausa pasti**, yang merupakan disjungsi literal yang salah satunya positif. Misalnya, klausa $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ adalah klausa pasti, sedangkan $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ bukan.

Klausul Horn yang sedikit lebih umum adalah **klausa Horn**, yang merupakan disjungsi literal yang *paling banyak salah satunya positif*. Jadi, semua klausa pasti adalah klausa Horn, seperti halnya klausa tanpa literal positif; ini disebut **klausa tujuan**. Klausa Horn ditutup dalam penyelesaian: jika Anda menyelesaikan dua klausa Horn, Anda mendapatkan kembali klausa Horn. Basis pengetahuan yang hanya berisi klausa pasti menarik karena tiga alasan:

1. Setiap klausa pasti dapat ditulis sebagai implikasi yang premisnya merupakan konjungsi literal positif dan simpulannya merupakan literal positif tunggal. Misalnya, klausa pasti $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ dapat ditulis sebagai implikasi $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1})$. Dalam bentuk implikasi, kalimat tersebut lebih mudah dipahami: kalimat tersebut menyatakan bahwa jika agen berada di [1,1] dan ada angin sepoi-sepoi, maka [1,1] adalah angin sepoi-sepoi. Dalam bentuk Horn, premis disebut badan dan simpulannya disebut kepala. Kalimat yang terdiri dari literal positif tunggal, seperti $L_{1,1}$, disebut fakta. Kalimat tersebut juga dapat ditulis dalam bentuk implikasi sebagai $True \Rightarrow L_{1,1}$, tetapi lebih mudah untuk menulis $L_{1,1}$ saja.
2. Inferensi dengan klausa Horn dapat dilakukan melalui algoritma **forward-chaining** dan **backward-chaining**, yang akan kami jelaskan selanjutnya. Kedua algoritma ini alami, karena langkah-langkah inferensi jelas dan mudah diikuti oleh manusia. Jenis inferensi ini adalah dasar untuk **pemrograman logika**, yang dibahas dalam Bab 9.
3. Memutuskan implikasi dengan klausa Horn dapat dilakukan dalam waktu yang linier dalam ukuran basis pengetahuan—kejutan yang menyenangkan.

$$
\begin{aligned}
CNFSentence &\rightarrow Clause_1 \wedge \cdots \wedge Clause_n \\
Clause &\rightarrow Literal_1 \vee \cdots \vee Literal_m \\
Literal &\rightarrow Symbol \mid \neg Symbol \\
Symbol &\rightarrow P \mid Q \mid R \mid \ldots \\
HornClauseForm &\rightarrow DefiniteClauseForm \mid GoalClauseForm \\
DefiniteClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow Symbol \\
GoalClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow False
\end{aligned}
$$

Gambar 7.14 Tata bahasa untuk bentuk normal konjungtif, klausa Horn, dan klausa pasti. Klausa seperti $A \wedge B \Rightarrow C$ masih merupakan klausa pasti jika ditulis sebagai $\neg A \vee \neg B \vee C$, tetapi hanya yang pertama yang dianggap sebagai bentuk kanonik untuk klausa pasti. Satu kelas lagi adalah kalimat $k - CNF$, yang merupakan kalimat CNF di mana setiap klausa memiliki paling banyak k literal.

Forward dan backward chaining

Algoritma forward-chaining $\text{PL} - \text{FC} - \text{ENTAILS?}\,(KB, q)$ menentukan apakah satu simbol proposisi q —kueri—diintisarikan oleh basis pengetahuan klausa-klausa tertentu. Algoritma ini dimulai dari fakta-fakta yang diketahui (literal positif) dalam basis pengetahuan. Jika semua premis implikasi diketahui, maka kesimpulannya ditambahkan ke kumpulan fakta yang diketahui. Misalnya, jika $L_{1,1}$ dan *Breeze* diketahui dan $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$)ada di basis pengetahuan, maka $B_{1,1}$ dapat ditambahkan. Proses ini berlanjut hingga kueri q ditambahkan atau hingga tidak ada lagi inferensi yang dapat dibuat. Algoritme terperinci ditunjukkan pada Gambar 7.15; hal utama yang perlu diingat adalah bahwa algoritme tersebut berjalan dalam waktu linier.

Cara terbaik untuk memahami algoritme adalah melalui contoh dan gambar. Gambar 7.16(a) menunjukkan basis pengetahuan sederhana klausa Horn dengan A dan B sebagai fakta yang diketahui. Gambar 7.16(b) menunjukkan basis pengetahuan yang sama yang digambar sebagai grafik AND–OR (lihat Bab 4). Dalam grafik **AND–OR**, beberapa tautan yang dihubungkan oleh busur menunjukkan konjungsi—setiap tautan harus dibuktikan—sementara beberapa tautan tanpa busur menunjukkan disjungsi—setiap tautan dapat dibuktikan. Mudah untuk melihat cara kerja forward chaining dalam grafik. Daun yang diketahui (di sini, A dan B) ditetapkan, dan inferensi disebarkan ke atas grafik sejauh mungkin. Di mana pun konjungsi muncul, penyebaran menunggu hingga semua konjungsi diketahui sebelum melanjutkan. Pembaca didorong untuk mengerjakan contoh tersebut secara terperinci.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
    inputs: KB, the knowledge base, a set of propositional definite clauses
            q, the query, a proposition symbol
    count ← a table, where count[c] is the number of symbols in c's premise
    inferred ← a table, where inferred[s] is initially false for all symbols
    agenda ←a queue of symbols, initially symbols known to be true in KB

    while agenda is not empty do
        p ← POP(agenda)
        if p = q then return true
        if inferred[p] = false then
            inferred[p] ← true
            for each clause c in KB where p is in c.PREMISE do
                decrement count[c]
                if count[c]=0 then add c.CONCLUSION to agenda
    return false
```

Gambar 7.15 Algoritma forward-chaining untuk logika proposisional. Agenda melacak simbol yang diketahui benar tetapi belum "diproses." Tabel hitungan melacak berapa banyak premis dari setiap implikasi yang belum diketahui. Setiap kali simbol baru p dari agenda diproses, hitungan dikurangi satu untuk setiap implikasi yang premis p-nya muncul (mudah diidentifikasi dalam waktu konstan dengan pengindeksan yang sesuai.) Jika hitungan mencapai nol, semua premis implikasi diketahui, sehingga kesimpulannya dapat ditambahkan ke agenda. Akhirnya, kita perlu melacak simbol mana yang telah diproses; simbol yang sudah ada dalam set simbol yang disimpulkan tidak perlu ditambahkan ke agenda lagi. Ini menghindari pekerjaan yang berlebihan dan mencegah loop yang disebabkan oleh implikasi seperti $P \Rightarrow Q$ dan $Q \Rightarrow P$.

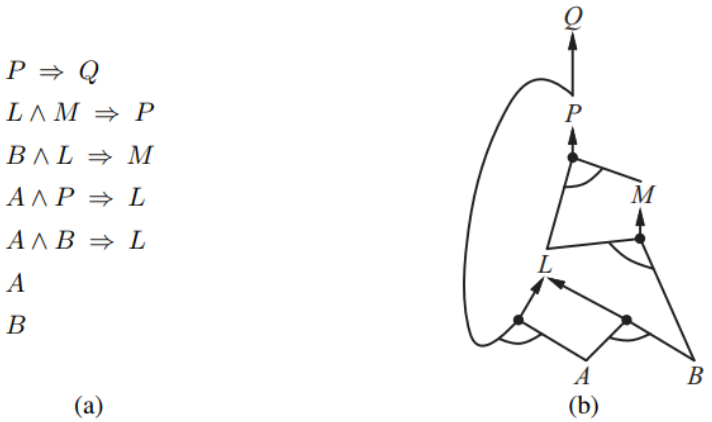
Mudah untuk melihat bahwa forward chaining itu benar: setiap inferensi pada dasarnya adalah penerapan Modus Ponens. Forward chaining juga **lengkap**: setiap kalimat atomik yang diikutsertakan akan diturunkan. Cara termudah untuk melihat ini adalah dengan mempertimbangkan status akhir tabel yang disimpulkan (setelah algoritme mencapai **titik tetap** di mana tidak ada inferensi baru yang mungkin). Tabel tersebut berisi true untuk setiap simbol yang disimpulkan selama proses, dan **false** untuk semua simbol lainnya. Kita dapat melihat tabel tersebut sebagai model logis; lebih jauh, *setiap klausa pasti dalam KB asli adalah benar dalam model ini.*

Untuk melihat ini, asumsikan kebalikannya, yaitu bahwa beberapa klausa $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ salah dalam model tersebut. Maka $a_1 \wedge \ldots \wedge a_k$ harus benar dalam model tersebut dan *b* harus salah dalam model tersebut. Namun ini bertentangan dengan asumsi kita bahwa algoritme telah mencapai titik tetap! Oleh karena itu, kita dapat menyimpulkan bahwa himpunan kalimat atomik yang disimpulkan pada titik tetap tersebut mendefinisikan model KB asli. Lebih jauh, setiap kalimat atom q yang disyaratkan oleh KB harus benar dalam semua modelnya dan khususnya dalam model ini. Oleh karena itu, setiap kalimat atom q yang disyaratkan harus disimpulkan oleh algoritme.

Penalaran rantai maju adalah contoh konsep umum penalaran berbasis data—yaitu, penalaran yang fokus perhatiannya dimulai dengan data yang diketahui. Ini dapat digunakan dalam agen untuk memperoleh kesimpulan dari persepsi yang masuk, sering kali tanpa memikirkan pertanyaan tertentu. Misalnya, agen wumpus mungkin TELL persepsinya ke basis pengetahuan menggunakan algoritme rantai maju inkremental yang fakta-fakta baru dapat ditambahkan ke agenda untuk memulai kesimpulan baru. Pada manusia, sejumlah penalaran berbasis data terjadi saat informasi baru tiba. Misalnya, jika saya berada di dalam ruangan dan mendengar hujan mulai turun, mungkin terpikir oleh saya bahwa piknik akan dibatalkan. Namun, mungkin tidak terpikir oleh saya bahwa kelopak ketujuh belas pada mawar terbesar di taman tetangga saya akan basah; manusia menjaga forward chaining di bawah kendali yang cermat, agar tidak dibanjiri dengan konsekuensi yang tidak relevan.

Algoritma backward-chaining, seperti namanya, bekerja mundur dari kueri. Jika kueri q diketahui benar, maka tidak ada pekerjaan yang diperlukan. Jika tidak, algoritma menemukan implikasi tersebut dalam basis pengetahuan yang kesimpulannya adalah q . Jika semua premis dari salah satu implikasi tersebut dapat dibuktikan benar (dengan backward chaining), maka q benar. Ketika diterapkan pada kueri Q pada Gambar 7.16, ia bekerja mundur ke bawah grafik hingga mencapai serangkaian fakta yang diketahui, A dan B, yang menjadi dasar untuk pembuktian. Algoritma ini pada dasarnya identik dengan algoritma AND – OR – GRAPH – SEARCH pada Gambar 4.11. Seperti halnya forward chaining, implementasi yang efisien berjalan dalam waktu linier.

Backward chaining adalah bentuk **penalaran yang diarahkan pada tujuan**. Ini berguna untuk menjawab pertanyaan-pertanyaan spesifik seperti "Apa yang harus saya lakukan sekarang?" dan "Di mana kunci saya?" Seringkali, biaya backward chaining *jauh lebih kecil* daripada biaya linear dalam ukuran basis pengetahuan, karena prosesnya hanya menyentuh fakta-fakta yang relevan.



Gambar 7.16 (a) Seperangkat klausa Horn. (b) Grafik AND–OR yang sesuai.

7.6 PEMERIKSAAN MODEL PROPOSISI YANG EFEKTIF

Pada bagian ini, kami menguraikan dua keluarga algoritme yang efisien untuk inferensi proposisional umum berdasarkan pemeriksaan model: Satu pendekatan berdasarkan penelusuran backtracking, dan satu lagi pada penelusuran hill-climbing lokal. Algoritme ini

merupakan bagian dari "teknologi" logika proposisional. Bagian ini dapat dibaca sekilas pada bacaan pertama bab ini.

Algoritme yang kami uraikan adalah untuk memeriksa kepuasan: masalah SAT. (Seperti yang telah disebutkan sebelumnya, pengujian konsekuensi, $\alpha \models \beta$, dapat dilakukan dengan menguji ketidakpuasan $\alpha \wedge \neg\beta$.) Kami telah mencatat hubungan antara menemukan model yang memuaskan untuk kalimat logis dan menemukan solusi untuk masalah kepuasan kendala, jadi mungkin tidak mengherankan bahwa kedua keluarga algoritme tersebut sangat mirip dengan algoritme penelusuran balik dari Bagian 6.3 dan algoritme pencarian lokal dari Bagian 6.4. Namun, keduanya sangat penting karena begitu banyak masalah kombinatorial dalam ilmu komputer dapat disederhanakan menjadi pemeriksaan kepuasan kalimat proposisional. Setiap peningkatan dalam algoritme kepuasan memiliki konsekuensi besar bagi kemampuan kita untuk menangani kompleksitas secara umum.

Algoritme penelusuran balik yang lengkap

Algoritme pertama yang kami pertimbangkan sering disebut algoritme **Davis–Putnam,** berdasarkan makalah penting oleh Martin Davis dan Hilary Putnam (1960). Algoritma ini sebenarnya adalah versi yang dijelaskan oleh Davis, Logemann, dan Loveland (1962), jadi kita akan menyebutnya DPLL berdasarkan inisial keempat penulis. DPLL mengambil kalimat dalam bentuk normal konjungtif—satu set klausa sebagai input. Seperti $BACKTRACKING-\text{SEARCH}$ dan $\text{TT}-\text{ENTAILS?}$, pada dasarnya ini adalah enumerasi rekursif dan mendalam dari model-model yang mungkin. Ini mewujudkan tiga perbaikan atas skema sederhana $\text{TT}-\text{ENTAILS?}$:

- Penghentian dini: Algoritma mendeteksi apakah kalimat itu harus benar atau salah, bahkan dengan model yang sebagian sudah selesai. Sebuah klausa benar jika ada literal yang benar, bahkan jika literal lainnya belum memiliki nilai kebenaran; oleh karena itu, kalimat secara keseluruhan dapat dinilai benar bahkan sebelum modelnya selesai. Misalnya, kalimat $(A \vee B) \wedge (A \vee C)$ bernilai benar jika A bernilai benar, terlepas dari nilai B dan C. Demikian pula, kalimat bernilai salah jika ada klausa yang bernilai salah, yang terjadi ketika setiap literalnya bernilai salah. Sekali lagi, ini dapat terjadi jauh sebelum modelnya selesai. Penghentian awal menghindari pemeriksaan seluruh sub-pohon dalam ruang pencarian.
- Heuristik simbol murni: Simbol murni adalah simbol yang selalu muncul dengan "tanda" yang sama di semua klausa. Misalnya, dalam tiga klausa $(A \vee \neg B), (\neg B \vee \neg C)$ dan $(A \vee C)$, simbol A bernilai murni karena hanya literal positif yang muncul, B bernilai murni karena hanya literal negatif yang muncul, dan C tidak murni. Mudah untuk melihat bahwa jika sebuah kalimat memiliki model, maka ia memiliki model dengan simbol murni yang ditetapkan untuk menjadikan literalnya bernilai benar, karena hal itu tidak akan pernah dapat menjadikan klausa bernilai salah. Perhatikan bahwa, dalam menentukan kemurnian simbol, algoritme dapat mengabaikan klausa yang sudah diketahui benar dalam model yang dibangun sejauh ini. Misalnya, jika model berisi $B = wrong$, maka klausa $(\neg B \vee \neg C)$ sudah benar, dan dalam klausa yang tersisa C hanya muncul sebagai literal positif; oleh karena itu C menjadi murni.

- Heuristik klausa unit: Klausa unit didefinisikan sebelumnya sebagai klausa dengan hanya satu literal. Dalam konteks DPLL, ini juga berarti klausa di mana semua literal kecuali satu sudah ditetapkan salah oleh model. Misalnya, jika model berisi $B = true$, maka $(\neg B \vee \neg C)$ disederhanakan menjadi $\neg C$, yang merupakan klausa unit. Jelas, agar klausa ini benar, C harus ditetapkan menjadi salah. Heuristik klausa unit menetapkan semua simbol tersebut sebelum bercabang pada sisanya. Salah satu konsekuensi penting dari heuristik adalah bahwa setiap upaya untuk membuktikan (dengan sanggahan) literal yang sudah ada dalam basis pengetahuan akan langsung berhasil. Perhatikan juga bahwa menetapkan satu klausa unit dapat membuat klausa unit lain—misalnya, ketika C ditetapkan ke false, $(C \vee A)$ menjadi klausa unit, yang menyebabkan true ditetapkan ke A. "Rangkaian" penetapan paksa ini disebut propagasi unit. Ini menyerupai proses forward chaining dengan klausa pasti, dan memang, jika ekspresi CNF hanya berisi klausa pasti maka DPLL pada dasarnya mereplikasi forward chaining.

```
function DPLL-SATISFIABLE?(s) returns true or false
    inputs: s, a sentence in propositional logic

    clauses ← the set of clauses in the CNF representation of s
    symbols ← a list of the proposition symbols in s
    return DPLL(clauses, symbols, { })

_____________________________________________________________

function DPLL(clauses, symbols, model) returns true or false

if every clause in clauses is true in model then return true
    if some clause in clauses is false in model then return false
    P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
    if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
    P, value ← FIND-UNIT-CLAUSE(clauses, model)
    if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
    P ← FIRST(symbols); rest ← REST(symbols)
    return DPLL(clauses, rest, model ∪ {P=true}) or
           DPLL(clauses, rest, model ∪ {P=false}))
```

Gambar 7.17 Algoritma DPLL untuk memeriksa kepuasan kalimat dalam logika proposisional. Ide di balik $\text{FIND} - \text{PURE} - \text{SYMBOL}$ dan $\text{FIND} - \text{UNIT} - \text{CLAUSE}$ dijelaskan dalam teks; masing-masing mengembalikan simbol (atau null) dan nilai kebenaran untuk ditetapkan pada simbol tersebut. Seperti $\text{TT} - \text{ENTAILS?}$, DPLL beroperasi pada model parsial.

Algoritma DPLL ditunjukkan pada Gambar 7.17, yang memberikan kerangka penting dari proses pencarian.

Yang tidak ditunjukkan pada Gambar 7.17 adalah trik yang memungkinkan penyelesai SAT untuk meningkatkan skala ke masalah yang lebih besar. Menariknya, sebagian besar trik ini sebenarnya agak umum, dan kita telah melihatnya sebelumnya dalam bentuk lain:

1. **Analisis komponen** (seperti yang terlihat pada Tasmania di CSP): Karena DPLL menetapkan nilai kebenaran pada variabel, rangkaian klausa dapat dipisahkan menjadi subset terpisah, yang disebut **komponen**, yang tidak memiliki variabel yang tidak ditetapkan. Dengan cara yang efisien untuk mendeteksi saat ini terjadi, penyelesai dapat memperoleh kecepatan yang cukup besar dengan mengerjakan setiap komponen secara terpisah.
2. **Pengurutan variabel dan nilai** (seperti yang terlihat pada Bagian 6.3.1 untuk CSP): Implementasi DPLL sederhana kami menggunakan pengurutan variabel yang sembarangan dan selalu mencoba nilai benar sebelum salah. **Heuristik derajat** menyarankan pemilihan variabel yang paling sering muncul di antara semua klausa yang tersisa.
3. **Penelusuran balik cerdas** (seperti yang terlihat di Bagian 6.3 untuk CSP): Banyak masalah yang tidak dapat diselesaikan dalam waktu berjam-jam dengan penelusuran balik kronologis dapat diselesaikan dalam hitungan detik dengan penelusuran balik cerdas yang menelusuri kembali hingga ke titik konflik yang relevan. Semua penyelesai SAT yang melakukan penelusuran balik cerdas menggunakan beberapa bentuk **pembelajaran klausa konflik** untuk mencatat konflik sehingga konflik tersebut tidak akan terulang lagi di kemudian hari dalam pencarian. Biasanya sekumpulan konflik berukuran terbatas disimpan, dan konflik yang jarang digunakan dihilangkan.
4. **Pengulangan acak**: Terkadang penelusuran tampaknya tidak mengalami kemajuan. Dalam kasus ini, kita dapat memulai dari atas pohon pencarian, daripada mencoba untuk melanjutkan. Setelah memulai ulang, berbagai pilihan acak (dalam pemilihan variabel dan nilai) dibuat. Klausa yang dipelajari pada penelusuran pertama dipertahankan setelah memulai ulang dan dapat membantu memangkas ruang pencarian. Pengulangan tidak menjamin bahwa solusi akan ditemukan lebih cepat, tetapi mengurangi varians pada waktu penyelesaian.
5. **Pengindeksan yang cerdas** (seperti yang terlihat dalam banyak algoritme): Metode percepatan yang digunakan dalam DPLL itu sendiri, serta trik yang digunakan dalam penyelesai modern, memerlukan pengindeksan cepat untuk hal-hal seperti "himpunan klausa di mana variabel Xi muncul sebagai literal positif." Tugas ini menjadi rumit karena algoritme hanya tertarik pada klausa yang belum terpenuhi oleh penugasan sebelumnya ke variabel, sehingga struktur pengindeksan harus diperbarui secara dinamis saat komputasi berlangsung.

Dengan penyempurnaan ini, penyelesai modern dapat menangani masalah dengan puluhan juta variabel. Mereka telah merevolusi bidang-bidang seperti verifikasi perangkat keras dan verifikasi protokol keamanan, yang sebelumnya memerlukan pembuktian yang rumit dan dipandu secara manual.

Algoritma pencarian lokal

Kita telah melihat beberapa algoritma pencarian lokal sejauh ini dalam buku ini, termasuk $\text{HILL} - \text{CLIMBING}$ dan $\text{SIMULATED} - \text{ANNEALING}$. Algoritma ini dapat diterapkan secara langsung pada masalah kepuasan, asalkan kita memilih fungsi evaluasi yang tepat. Karena tujuannya adalah untuk menemukan penugasan yang memenuhi setiap klausa, fungsi evaluasi yang menghitung jumlah klausa yang tidak terpenuhi akan menyelesaikan pekerjaan. Faktanya, ini adalah ukuran yang digunakan oleh algoritma $\text{MIN} - \text{CONFLICTS}$ untuk CSP. Semua algoritma ini mengambil langkah-langkah dalam ruang penugasan lengkap, membalik nilai kebenaran satu simbol pada satu waktu. Ruang tersebut biasanya berisi banyak titik minimum lokal, yang untuk menghindarinya diperlukan berbagai bentuk keacakan. Dalam beberapa tahun terakhir, telah banyak eksperimen untuk menemukan keseimbangan yang baik antara keserakahan dan keacakan.

Salah satu algoritma paling sederhana dan paling efektif yang muncul dari semua pekerjaan ini disebut WALKSAT (Gambar 7.18). Pada setiap iterasi, algoritma memilih klausa yang tidak terpenuhi dan memilih simbol dalam klausa tersebut untuk dibalik. Algoritma memilih secara acak di antara dua cara untuk memilih simbol yang akan dibalik: (1) langkah "$min - conflicts$" yang meminimalkan jumlah klausa yang tidak terpenuhi dalam keadaan baru dan (2) langkah "$random\ walk$" yang memilih simbol secara acak.

Ketika $WALKSAT$ mengembalikan model, kalimat input memang memuaskan, tetapi ketika mengembalikan kegagalan, ada dua kemungkinan penyebab: kalimat tersebut tidak memuaskan atau kita perlu memberi algoritma lebih banyak waktu. Jika kita menetapkan $max\_flips = \infty\ dan\ p > 0, WALKSAT$ pada akhirnya akan mengembalikan model (jika ada), karena langkah $random\ walk$ pada akhirnya akan menemukan solusi. Sayangnya, jika $\max\ \_flips$ tak terhingga dan kalimatnya tidak memuaskan, maka algoritma tidak akan pernah berakhir!

Karena alasan ini, $WALKSAT$ sangat berguna ketika kita mengharapkan adanya solusi—misalnya, masalah yang dibahas dalam Bab 3 dan 6 biasanya memiliki solusi. Di sisi lain, $WALKSAT$ tidak selalu dapat mendeteksi ketidakpuasan, yang diperlukan untuk memutuskan konsekuensi. Misalnya, agen tidak dapat menggunakan $WALKSAT$ dengan andal untuk membuktikan bahwa sebuah kotak aman di dunia wumpus.

```
function WALKSAT(clauses, p, max flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
          if model satisfies clauses then return model
          clause ← a randomly selected clause from clauses that is false in model
          with probability p flip the value in model of a randomly selected symbol from clause
          else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Gambar 7.18 Algoritma WALKSAT untuk memeriksa kepuasan dengan membalik nilai variabel secara acak. Ada banyak versi algoritma ini.

Lanskap masalah SAT acak

Beberapa masalah SAT lebih sulit daripada yang lain. Masalah yang mudah dapat dipecahkan dengan algoritme lama apa pun, tetapi karena kita tahu bahwa SAT bersifat NP-lengkap, setidaknya beberapa contoh masalah harus memerlukan waktu proses eksponensial. Dalam Bab 6, kita melihat beberapa penemuan mengejutkan tentang jenis masalah tertentu. Misalnya, masalah n-queens—yang dianggap cukup rumit untuk algoritme penelusuran $back - track$—ternyata mudah untuk metode penelusuran lokal, seperti $min - conflicts$. Ini karena solusi didistribusikan sangat padat dalam ruang penugasan, dan setiap penugasan awal dijamin memiliki solusi di dekatnya. Jadi, n-queens mudah karena **dibatasi secara kurang.**

Ketika kita melihat masalah kepuasan dalam bentuk normal konjungtif, masalah yang dibatasi secara kurang adalah masalah dengan klausa yang relatif sedikit yang membatasi variabel. Misalnya, berikut adalah kalimat 3-CNF yang dibuat secara acak dengan lima simbol dan lima klausa:

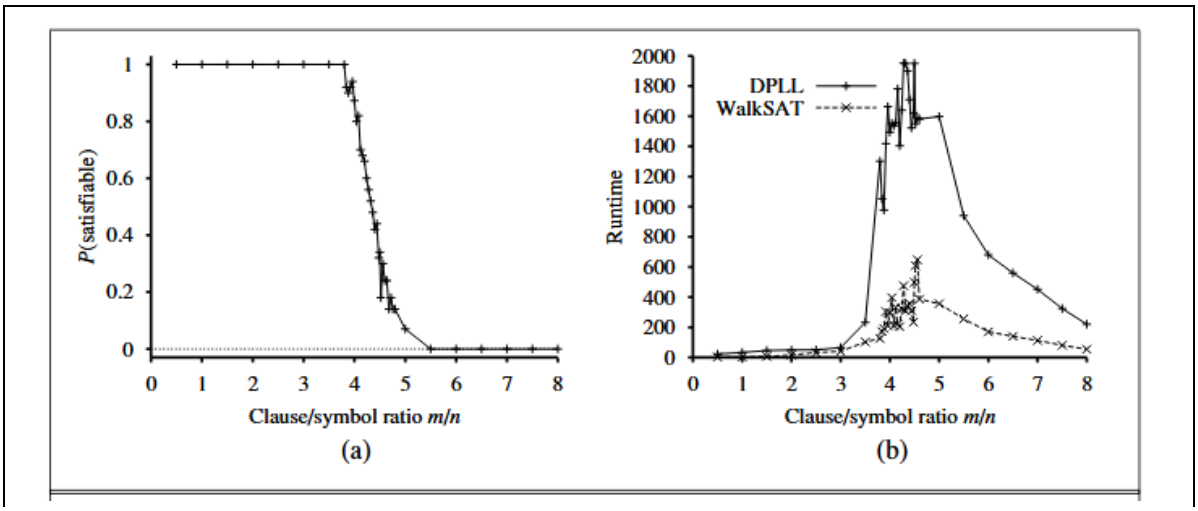
$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$$

Enam belas dari 32 kemungkinan penugasan adalah model kalimat ini, jadi, rata-rata, hanya diperlukan dua tebakan acak untuk menemukan model. Ini adalah masalah kepuasan yang mudah, seperti kebanyakan masalah yang kurang dibatasi. Di sisi lain, masalah yang *terlalu dibatasi* memiliki banyak klausa yang relatif terhadap jumlah variabel dan cenderung tidak memiliki solusi.

Untuk melampaui intuisi dasar ini, kita harus mendefinisikan dengan tepat bagaimana kalimat acak dihasilkan. Notasi $CNF_k(m, n)$ menunjukkan kalimat k-CNF dengan m klausa dan n simbol, di mana klausa dipilih secara seragam, independen, dan tanpa penggantian dari antara semua klausa dengan k literal berbeda, yang positif atau negatif secara acak. (Sebuah simbol mungkin tidak muncul dua kali dalam sebuah klausa, dan juga tidak boleh sebuah klausa muncul dua kali dalam sebuah kalimat.)

Dengan sumber kalimat acak, kita dapat mengukur probabilitas kepuasan. Gambar 7.19(a) memplot probabilitas untuk $CNF_3(m, 50)$, yaitu kalimat dengan 50 variabel dan 3 literal per klausa, sebagai fungsi rasio klausa/simbol, m/n. Seperti yang kita harapkan, untuk m/n kecil probabilitas kepuasan mendekati 1, dan pada m/n besar probabilitas mendekati 0. Probabilitas turun cukup tajam sekitar m/n = 4,3. Secara empiris, kita menemukan bahwa "tebing" tetap di tempat yang kira-kira sama (untuk $k = 3$) dan menjadi lebih tajam dan lebih tajam saat n meningkat. Secara teoritis, **dugaan ambang kepuasan** mengatakan bahwa untuk setiap $k \geq 3$, ada rasio ambang r_k sedemikian rupa sehingga, saat n menuju tak terhingga, probabilitas bahwa $CNF_k(n, rn)$ dapat dipenuhi menjadi 1 untuk semua nilai r di bawah ambang, dan 0 untuk semua nilai di atas. Dugaan tersebut masih belum terbukti.

Sekarang setelah kita memiliki gambaran yang baik tentang di mana letak masalah yang memuaskan dan yang tidak memuaskan, pertanyaan berikutnya adalah, di mana letak masalah yang sulit? Ternyata masalah-masalah tersebut juga sering berada pada nilai ambang batas. Gambar 7.19(b) menunjukkan bahwa masalah dengan 50 simbol pada nilai ambang batas 4,3 sekitar 20 kali lebih sulit dipecahkan daripada masalah dengan rasio 3,3. Masalah dengan batasan yang kurang paling mudah dipecahkan (karena sangat mudah untuk menebak solusinya); masalah dengan batasan yang terlalu banyak tidak semudah masalah dengan batasan yang kurang, tetapi masih jauh lebih mudah daripada masalah yang berada tepat di ambang batas.



Gambar 7.19 (a) Grafik yang menunjukkan probabilitas bahwa kalimat 3-CNF acak dengan $n = 50$ simbol dapat memuaskan, sebagai fungsi rasio klausa/simbol m/n. (b) Grafik median waktu proses (diukur dalam jumlah panggilan rekursif ke DPLL, proksi yang baik) pada kalimat 3-CNF acak. Soal yang paling sulit memiliki rasio klausa/simbol sekitar 4,3.

7.7 AGEN BERDASARKAN LOGIKA PROPOSISI

Di bagian ini, kami menyatukan apa yang telah kami pelajari sejauh ini untuk membangun agen dunia wumpus yang menggunakan logika proposisional. Langkah pertama adalah memungkinkan agen untuk menyimpulkan, sejauh mungkin, keadaan dunia

berdasarkan riwayat persepsinya. Ini memerlukan penulisan model logis lengkap tentang efek tindakan. Kami juga menunjukkan bagaimana agen dapat melacak dunia secara efisien tanpa kembali ke riwayat persepsi untuk setiap inferensi. Terakhir, kami menunjukkan bagaimana agen dapat menggunakan inferensi logis untuk membangun rencana yang dijamin akan mencapai tujuannya.

Keadaan dunia saat ini

Sebagaimana dinyatakan di awal bab, agen logis beroperasi dengan menyimpulkan apa yang harus dilakukan dari basis pengetahuan kalimat tentang dunia. Basis pengetahuan tersebut terdiri dari aksioma—pengetahuan umum tentang cara kerja dunia—dan kalimat persepsi yang diperoleh dari pengalaman agen di dunia tertentu. Di bagian ini, kami fokus pada masalah menyimpulkan keadaan terkini dunia wumpus

Agen mengetahui bahwa kotak awal tidak berisi lubang ($\neg P_{1,1}$) dan tidak ada wumpus ($\neg W_{1,1}$). Lebih jauh, untuk setiap kotak, ia mengetahui bahwa kotak itu berangin jika dan hanya jika kotak di sebelahnya memiliki lubang; dan kotak itu berbau jika dan hanya jika kotak di sebelahnya memiliki wumpus. Jadi, kami menyertakan banyak kalimat dalam bentuk berikut:

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$
...

Agen juga mengetahui bahwa ada tepat satu wumpus. Hal ini diungkapkan dalam dua bagian. Pertama, kita harus mengatakan bahwa setidaknya ada satu wumpus:

$W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,3} \vee W_{4,4}$

Kemudian, kita harus mengatakan bahwa *paling banyak ada satu* wumpus. Untuk setiap pasangan lokasi, kita tambahkan kalimat yang menyatakan bahwa paling sedikit satu di antaranya harus bebas wumpus:

$\neg W_{1,1} \vee \neg W_{1,2}$
$\neg W_{1,1} \vee \neg W_{1,3}$
...
$\neg W_{4,3} \vee \neg W_{4,4}$

Sejauh ini, semuanya baik-baik saja. Sekarang mari kita pertimbangkan persepsi agen. Jika saat ini ada bau busuk, seseorang mungkin mengira bahwa proposisi Bau busuk harus ditambahkan ke basis pengetahuan. Namun, ini tidak sepenuhnya benar: jika tidak ada bau busuk pada langkah waktu sebelumnya, maka $\neg Stench$ sudah akan dinyatakan, dan pernyataan baru hanya akan menghasilkan kontradiksi. Masalah terpecahkan ketika kita menyadari bahwa suatu persepsi menyatakan sesuatu hanya tentang waktu saat ini. Jadi, jika langkah waktu (seperti yang diberikan kepada $\text{MAKE} - \text{PERCEPT} - \text{SENTENCE}$ pada Gambar 7.1) adalah 4, maka kita menambahkan $Stench$ ke basis pengetahuan, alih-alih $Stench$—dengan rapi

menghindari kontradiksi dengan $\neg Stench$. Hal yang sama berlaku untuk persepsi angin sepoi-sepoi, benturan, kilauan, dan teriakan.

Gagasan untuk mengaitkan proposisi dengan langkah waktu meluas ke aspek apa pun di dunia yang berubah seiring waktu. Misalnya, basis pengetahuan awal mencakup $L_{1,1}^0$ —agen berada di kotak [1, 1] pada waktu 0—serta *FacingEast 0, HaveArrow 0, dan WumpusAlive0*. Kami menggunakan kata **fluent** (dari bahasa Latin fluens, mengalir) untuk merujuk pada aspek dunia yang berubah. "Fluent" adalah sinonim untuk "variabel keadaan," dalam pengertian yang dijelaskan dalam pembahasan representasi terfaktor. Simbol yang dikaitkan dengan aspek permanen dunia tidak memerlukan superskrip waktu dan terkadang disebut **variabel atemporal**.

Kita dapat menghubungkan persepsi bau busuk dan angin secara langsung ke properti kotak tempat mereka dialami melalui lokasi fluent sebagai berikut. Untuk setiap langkah waktu t dan setiap kotak $[x, y]$, kami menegaskan:

$$L_{x,y}^t \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L_{x,y}^t \Rightarrow (Stench^t \Leftrightarrow S_{x,y})$$

Sekarang, tentu saja, kita memerlukan aksioma yang memungkinkan agen melacak fluent seperti $L_{x,y}^t$ Fluent ini berubah sebagai hasil dari tindakan yang diambil oleh agen, jadi, dalam terminologi Bab 3, kita perlu menuliskan **model transisi** dunia wumpus sebagai serangkaian kalimat logis.

Pertama, kita memerlukan simbol proposisi untuk kemunculan tindakan. Seperti halnya persepsi, simbol-simbol ini diindeks berdasarkan waktu; dengan demikian, $Forward^0$ berarti bahwa agen mengeksekusi tindakan Forward pada waktu 0. Berdasarkan konvensi, persepsi untuk langkah waktu tertentu terjadi terlebih dahulu, diikuti oleh tindakan untuk langkah waktu tersebut, diikuti oleh transisi ke langkah waktu berikutnya.

Untuk menggambarkan bagaimana dunia berubah, kita dapat mencoba menulis **aksioma efek** yang menentukan hasil dari suatu tindakan pada langkah waktu berikutnya. Misalnya, jika agen berada di lokasi [1, 1] menghadap ke timur pada waktu 0 dan bergerak *Forward*, hasilnya adalah agen berada di kotak [2, 1] dan tidak lagi berada di [1, 1]:

$$L_{1,1}^0 \wedge FacingEast^0 \wedge Forward^0 \Rightarrow \left(L_{2,1}^1 \wedge \neg L_{1,1}^1\right). \quad (7.1)$$

Kita akan memerlukan satu kalimat seperti itu untuk setiap kemungkinan langkah waktu, untuk setiap 16 kotak, dan setiap empat orientasi. Kita juga akan memerlukan kalimat serupa untuk tindakan lainnya: $Grab, Shoot, Climb, TurnLeft,$ dan $TurnRight$.

Misalkan agen memutuskan untuk bergerak Maju pada waktu 0 dan menegaskan fakta ini ke dalam basis pengetahuannya. Mengingat aksioma efek dalam Persamaan (7.1), dikombinasikan dengan pernyataan awal tentang keadaan pada waktu 0, agen sekarang dapat menyimpulkan bahwa ia berada dalam [2, 1]. Yaitu, $ASK(KB, L_{2,1}^1)$= *benar*. Sejauh ini, baik-baik saja. Sayangnya, berita di tempat lain kurang baik: jika kita $ASK(KB, HaveArrow^1)$, jawabannya $False$, yaitu, agen tidak dapat membuktikan bahwa ia masih memiliki anak

panah; ia juga tidak dapat membuktikan bahwa ia tidak memilikinya! Informasi tersebut telah hilang karena aksioma efek gagal menyatakan apa yang tetap *tidak berubah* sebagai hasil dari suatu tindakan. Kebutuhan untuk melakukan hal ini menimbulkan **masalah kerangka**. Salah satu solusi yang mungkin untuk masalah kerangka adalah dengan menambahkan **aksioma kerangka** yang secara eksplisit menyatakan semua proposisi yang tetap sama. Misalnya, untuk setiap waktu t kita akan memiliki

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$
$$Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$

di mana kita secara eksplisit menyebutkan setiap proposisi yang tetap tidak berubah dari waktu t ke waktu $t + 1$ di bawah tindakan Forward. Meskipun agen sekarang tahu bahwa ia masih memiliki anak panah setelah bergerak maju dan bahwa wumpus belum mati atau hidup kembali, penyebaran aksioma kerangka tampaknya sangat tidak efisien. Di dunia dengan m tindakan berbeda dan n fluent, himpunan aksioma kerangka akan berukuran $O(mn)$. Manifestasi khusus dari masalah kerangka ini terkadang disebut **masalah kerangka representasional.** Secara historis, masalah ini merupakan masalah penting bagi peneliti AI; kami mengeksplorasinya lebih lanjut dalam catatan di akhir bab.

Masalah kerangka representasional penting karena dunia nyata memiliki sangat banyak fluent, untuk membuatnya lebih halus. Untungnya bagi kita manusia, setiap tindakan biasanya tidak mengubah lebih dari sejumlah kecil k dari fluent tersebut—dunia menunjukkan **lokalitas**. Memecahkan masalah kerangka representasional memerlukan pendefinisian model transisi dengan himpunan aksioma berukuran $O(mk)$ daripada berukuran $O(mn)$. Ada pula **masalah kerangka inferensial:** masalah memproyeksikan hasil dari rencana tindakan t langkah ke depan dalam waktu $O(kt)$ alih-alih $O(nt)$.

Solusi untuk masalah ini melibatkan perubahan fokus seseorang dari menulis aksioma tentang tindakan ke menulis aksioma tentang $fluent$. Jadi, untuk setiap fluent F , kita akan memiliki aksioma yang mendefinisikan nilai kebenaran F^{t+1} dalam hal fluent (termasuk F itu sendiri) pada waktu t dan tindakan yang mungkin terjadi pada waktu t. Sekarang, nilai kebenaran F^{t+1} dapat ditetapkan dalam salah satu dari dua cara: baik tindakan pada waktu t menyebabkan F menjadi benar pada $t + 1$, atau F sudah benar pada waktu t dan tindakan pada waktu t tidak menyebabkannya menjadi salah. Aksioma bentuk ini disebut aksioma status penerus dan memiliki skema ini:

$$F^{t+1} \Leftrightarrow FActionCausesF^t \vee (F^{t+1} \wedge \neg ActionCausesF^t)$$

Salah satu aksioma status penerus yang paling sederhana adalah aksioma untuk $HaveArrow$. Karena tidak ada tindakan untuk memuat ulang, bagian $ActionCausesF^t$ hilang dan kita tinggal dengan

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t) \qquad (7.2)$$

Untuk lokasi agen, aksioma status penerus lebih rumit. Misalnya, $L_{1,1}^{t+1}$ benar jika (a) agen bergerak Maju dari [1, 2] saat menghadap selatan, atau dari [2, 1] saat menghadap barat; atau

(b) $L_{1,1}^{t}$ sudah benar dan tindakan tidak menyebabkan gerakan (baik karena tindakan tidak Maju atau karena tindakan menabrak dinding). Ditulis dalam logika proposisional, ini menjadi:

$$L_{1,1}^{t+1} \Leftrightarrow \left(L_{1,1}^{t} \wedge (Forward^{t} \vee Bump^{t+1})\right) \vee \left(L_{1,2}^{t} \wedge (South^{t} \vee Forward^{t})\right) \vee \left(L_{2,1}^{t} \wedge (West^{t} \vee Forward^{t})\right) \quad (7.3)$$

Dengan serangkaian aksioma status penerus yang lengkap dan aksioma lain yang tercantum di awal bagian ini, agen akan dapat BERTANYA dan menjawab pertanyaan apa pun yang dapat dijawab tentang status dunia saat ini. Misalnya, di Bagian 7.2 urutan awal persepsi dan tindakan adalah

$$\begin{aligned}
&\neg Stench^{0} \wedge \neg Breeze^{0} \wedge \neg Glitter^{0} \wedge \neg Bump^{0} \wedge \neg Scream^{0} \;;\; Forward^{0} \\
&\neg Stench^{1} \wedge Breeze^{1} \wedge \neg Glitter^{1} \wedge \neg Bump^{1} \wedge \neg Scream^{1} \;;\; TurnRight^{1} \\
&\neg Stench^{2} \wedge Breeze^{2} \wedge \neg Glitter^{2} \wedge \neg Bump^{2} \wedge \neg Scream^{2} \;;\; TurnRight^{2} \\
&\neg Stench^{3} \wedge Breeze^{3} \wedge \neg Glitter^{3} \wedge \neg Bump^{3} \wedge \neg Scream^{3} \;;\; Forward^{3} \\
&\neg Stench^{4} \wedge \neg Breeze^{4} \wedge \neg Glitter^{4} \wedge \neg Bump^{4} \wedge \neg Scream^{4} \;;\; TurnRight^{4} \\
&\neg Stench^{5} \wedge \neg Breeze^{5} \wedge \neg Glitter^{5} \wedge \neg Bump^{5} \wedge \neg Scream^{5} \;;\; Forward^{5} \\
&Stench^{6} \wedge \neg Breeze^{6} \wedge \neg Glitter^{6} \wedge \neg Bump^{6} \wedge \neg Scream^{6}
\end{aligned}$$

Pada titik ini, kita memiliki $\text{ASK}(KB, L_{1,2}^{6}) = True$, jadi agen tahu di mana itu. Selain itu, $\text{ASK}(KB, W_{1,3}) = True$ dan $\text{ASK}(KB, P_{3,1}) = True$, jadi agen telah menemukan wumpus dan salah satu lubang. Pertanyaan terpenting bagi agen adalah apakah kotak itu OK untuk dimasuki, yaitu, kotak itu tidak berisi lubang atau wumpus yang hidup. Akan lebih mudah untuk menambahkan aksioma untuk ini, yang memiliki bentuk.

$$OK_{x,y}^{1} \Leftrightarrow \neg P_{x,y} \wedge \neg\left(W_{x,y}\, WumpusAlive^{t}\right)$$

Akhirnya, $ASK(KB, OK_{2,2}^{6}) = True$, jadi kuadrat [2, 2] OK untuk dimasuki. Faktanya, dengan algoritma inferensi yang lengkap dan masuk akal seperti DPLL, agen dapat menjawab pertanyaan apa pun yang dapat dijawab tentang kuadrat mana yang OK—dan dapat melakukannya hanya dalam beberapa milidetik untuk dunia wumpus kecil hingga sedang.

Memecahkan masalah kerangka representasional dan inferensial merupakan langkah maju yang besar, tetapi masalah yang berbahaya tetap ada: kita perlu memastikan bahwa semua prasyarat yang diperlukan dari suatu tindakan berlaku agar tindakan tersebut memiliki efek yang diinginkan. Kami katakan bahwa tindakan Maju menggerakkan agen maju kecuali ada dinding yang menghalangi, tetapi ada banyak pengecualian tidak biasa lainnya yang dapat menyebabkan tindakan gagal: agen mungkin tersandung dan jatuh, terkena serangan jantung, terbawa oleh kelelawar raksasa, dll. Menentukan semua pengecualian ini disebut masalah kualifikasi. Tidak ada solusi lengkap dalam logika; perancang sistem harus menggunakan

penilaian yang baik dalam memutuskan seberapa rinci mereka ingin menentukan model mereka, dan detail apa yang ingin mereka abaikan.

Agen hibrida

Kemampuan untuk menyimpulkan berbagai aspek keadaan dunia dapat dikombinasikan secara langsung dengan aturan kondisi-tindakan dan dengan algoritma pemecahan masalah dari Bab 3 dan 4 untuk menghasilkan **agen hibrida** untuk dunia wumpus. Gambar 7.20 menunjukkan satu cara yang mungkin untuk melakukan ini. Program agen memelihara dan memperbarui basis pengetahuan serta rencana terkini. Basis pengetahuan awal berisi aksioma *atemporal*—aksioma yang tidak bergantung pada t, seperti aksioma yang menghubungkan kesejukan kotak dengan keberadaan lubang. Pada setiap langkah waktu, kalimat persepsi baru ditambahkan bersama dengan semua aksioma yang bergantung pada t, seperti aksioma status penerus. (Bagian berikutnya menjelaskan mengapa agen tidak memerlukan aksioma untuk langkah waktu mendatang.) Kemudian, agen menggunakan inferensi logis, dengan MENGAJUKAN pertanyaan pada basis pengetahuan, untuk menentukan kotak mana yang aman dan mana yang belum dikunjungi.

Bagian utama program agen menyusun rencana berdasarkan prioritas tujuan yang menurun. Pertama, jika ada kilauan, program menyusun rencana untuk meraih emas, mengikuti rute kembali ke lokasi awal, dan memanjat keluar dari gua. Jika tidak, jika tidak ada rencana saat ini, program merencanakan rute ke kotak aman terdekat yang belum dikunjungi, memastikan rute hanya melewati kotak aman. Perencanaan rute dilakukan dengan pencarian $\mathrm{A}*$, bukan dengan ASK.

Jika tidak ada petak aman untuk dijelajahi, langkah berikutnya—jika agen masih memiliki anak panah—adalah mencoba membuat petak aman dengan menembak salah satu lokasi wumpus yang memungkinkan. Ini ditentukan dengan menanyakan di mana $\mathrm{ASK}(KB, \neg W_{x,y}) = False$—yaitu, di mana tidak diketahui bahwa tidak ada wumpus. Fungsi $\mathrm{PLAN-SHOT}$ (tidak ditampilkan) menggunakan $\mathrm{PLAN-ROUTE}$ untuk merencanakan urutan tindakan yang akan menyelaraskan tembakan ini. Jika ini gagal, program mencari petak untuk dijelajahi yang tidak terbukti tidak aman—yaitu, petak yang $\mathrm{ASK}(KB, \neg OK_{x,y}^{t})$ mengembalikan nilai salah. Jika tidak ada petak seperti itu, maka misi tidak mungkin dilakukan dan agen mundur ke [1, 1] dan memanjat keluar dari gua.

Estimasi status logis

Program agen pada Gambar 7.20 bekerja dengan cukup baik, tetapi memiliki satu kelemahan utama: seiring berjalannya waktu, biaya komputasi yang terlibat dalam panggilan ke ASK akan terus bertambah. Hal ini terjadi terutama karena inferensi yang diperlukan harus kembali lebih jauh dan lebih jauh dalam waktu dan melibatkan lebih banyak simbol proposisi. Jelas, hal ini tidak berkelanjutan—kita tidak dapat memiliki agen yang waktunya untuk memproses setiap persepsi bertambah sebanding dengan lamanya masa pakainya! Yang benar-benar kita butuhkan adalah waktu pembaruan yang konstan—yaitu, tidak bergantung pada t. Jawaban yang jelas adalah menyimpan, atau menyimpan, hasil inferensi, sehingga proses inferensi pada langkah waktu berikutnya dapat dibangun berdasarkan hasil langkah sebelumnya alih-alih harus memulai lagi dari awal.

Seperti yang kita lihat di Bagian 4.4, riwayat persepsi di masa lalu dan semua konsekuensinya dapat digantikan oleh **status keyakinan**—yaitu, beberapa representasi dari himpunan semua kemungkinan status dunia saat ini. Proses memperbarui status keyakinan saat persepsi baru muncul disebut **estimasi status**. Sementara di Bagian 4.4 status keyakinan adalah daftar status yang eksplisit, di sini kita dapat menggunakan kalimat logis yang melibatkan simbol proposisi yang terkait dengan langkah waktu saat ini, serta simbol atemporal. Misalnya, kalimat logis

$$WumpusAlive^1 \wedge L_{2,1}^1 \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \tag{7.2}$$

mewakili himpunan semua keadaan pada waktu 1 saat wumpus hidup, agen berada di [2, 1], kotak itu berangin, dan ada lubang di [3, 1] atau [2, 2] atau keduanya.

Mempertahankan keadaan keyakinan yang tepat sebagai rumus logis ternyata tidak mudah. Jika ada n simbol fasih untuk waktu t, maka ada 2^n keadaan yang mungkin—yaitu, penugasan nilai kebenaran pada simbol-simbol tersebut. Sekarang, himpunan keadaan keyakinan adalah pangkat (himpunan semua himpunan bagian) dari himpunan keadaan fisik. Ada 2^n keadaan fisik, maka ada 2^{2^n} keadaan keyakinan. Bahkan jika kita menggunakan pengodean rumus logis yang paling ringkas, dengan setiap keadaan keyakinan direpresentasikan oleh angka biner unik, kita akan memerlukan angka dengan $\log_2(2^{2^n}) = 2^n$ bit untuk memberi label keadaan keyakinan saat ini.

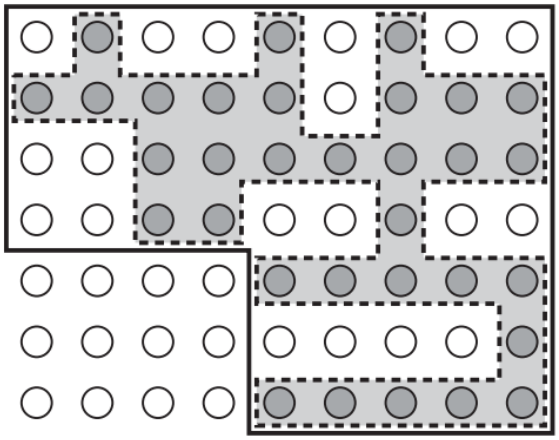
Artinya, estimasi keadaan yang tepat mungkin memerlukan rumus logis yang ukurannya eksponensial dalam jumlah simbol. Satu skema yang sangat umum dan alami untuk estimasi keadaan perkiraan adalah dengan merepresentasikan keadaan keyakinan sebagai konjungsi literal, yaitu, rumus 1-CNF. Untuk melakukan ini, program agen hanya mencoba membuktikan X^t dan $\neg X^t$ untuk setiap simbol X^t (serta setiap simbol atemporal yang nilai kebenarannya belum diketahui), mengingat keadaan keyakinan pada $t - 1$. Konjungsi literal yang dapat dibuktikan menjadi keadaan keyakinan baru, dan keadaan keyakinan sebelumnya dibuang.Penting untuk dipahami bahwa skema ini dapat kehilangan beberapa informasi seiring berjalannya waktu. Misalnya, jika kalimat dalam Persamaan (7.4) adalah keadaan keyakinan yang sebenarnya, maka baik $P_{3,1}$ maupun $P_{2,2}$ tidak akan dapat dibuktikan secara individual dan keduanya tidak akan muncul dalam keadaan keyakinan 1-CNF. Di sisi lain, karena setiap literal dalam status keyakinan 1-CNF dibuktikan dari status keyakinan sebelumnya, dan status keyakinan awal adalah pernyataan yang benar, kita tahu bahwa seluruh status keyakinan 1-CNF harus benar.

```
function HYBRID-WUMPUS-AGENT(percept) returns an action
    inputs: percept, a list, [stench,breeze,glitter ,bump,scream]
    persistent: KB, a knowledge base, initially the atemporal "wumpus physics"
                t, a counter, initially 0, indicating time
                plan, an action sequence, initially empty

TELL(KB, MAKE-PERCEPT-SENTENCE(percept,t))
TELL the KB the temporal "physics" sentences for time t
safe ← {[x , y] : ASK(KB,OK^t_{x,y}) = true}
if ASK(KB,Glitter^t) = true then
    plan ← [Grab]+PLAN-ROUTE(current, {[1,1]}, safe)+[Climb]
if plan is empty then
    unvisited ← {[x , y] : ASK(KB, Ltx,y) = false for all t^1 ≤ t}
    plan ← PLAN-ROUTE(current, unvisited ∩safe, safe)
if plan is empty and ASK(KB,HaveArrow^t) = true then
    possible wumpus ← {[x , y] : ASK(KB,¬ W_{x,y}) = false}
    plan ← PLAN-SHOT(current, possible wumpus, safe)
if plan is empty then // no choice but to take a risk
    not unsafe ← {[x , y] : ASK(KB,¬ OK^1_{x,y} ) = false}
    plan ← PLAN-ROUTE(current, unvisited ∩not unsafe, safe)
if plan is empty then
    plan ← PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
action ← POP(plan)
TELL(KB, MAKE-ACTION-SENTENCE(action,t))
t ← t + 1
return action
```

```
function PLAN-ROUTE(current,goals,allowed) returns an action sequence
    inputs: current, the agent's current position
            goals, a set of squares; try to plan a route to one of them
            allowed, a set of squares that can form part of the route

    problem ← ROUTE-PROBLEM(current, goals,allowed)
    return A*-GRAPH-SEARCH(problem)
```

Gambar 7.20 Program agen hibrida untuk dunia wumpus. Program ini menggunakan basis pengetahuan proposisional untuk menyimpulkan keadaan dunia, dan kombinasi pencarian pemecahan masalah dan kode khusus domain untuk memutuskan tindakan apa yang harus diambil.

Dengan demikian, *himpunan status yang mungkin yang direpresentasikan oleh status keyakinan 1-CNF mencakup semua status yang sebenarnya mungkin diberikan riwayat persepsi yang lengkap*. Seperti yang diilustrasikan dalam Gambar 7.21, status keyakinan 1-CNF bertindak sebagai amplop luar sederhana, atau **perkiraan konservatif**, di sekitar status keyakinan yang tepat. Kita melihat gagasan perkiraan konservatif terhadap himpunan yang rumit ini sebagai tema yang berulang di banyak bidang AI.



Gambar 7.21 Penggambaran status keyakinan 1-CNF (garis tepi tebal) sebagai perkiraan konservatif yang dapat direpresentasikan secara sederhana terhadap status keyakinan yang tepat (bergelombang) (wilayah yang diarsir dengan garis tepi putus-putus). Setiap kemungkinan dunia ditunjukkan sebagai lingkaran; yang diarsir konsisten dengan semua persepsi.

Membuat rencana dengan inferensi proposisional

Agen pada Gambar 7.20 menggunakan inferensi logis untuk menentukan kotak mana yang aman, tetapi menggunakan pencarian A$*$ untuk membuat rencana. Di bagian ini, kami menunjukkan cara membuat rencana dengan inferensi logis. Ide dasarnya sangat sederhana:

1. Buat kalimat yang mencakup
   a) $Init^0$, kumpulan pernyataan tentang keadaan awal;
   b) $Transition^1 \ldots Transition^t$, aksioma keadaan penerus untuk semua tindakan yang mungkin pada setiap waktu hingga waktu maksimum t;
   c) pernyataan bahwa tujuan tercapai pada waktu t: $HaveGold^t \wedge ClimbedOut^t$
2. Sajikan seluruh kalimat kepada penyelesai SAT. Jika penyelesai menemukan model yang memuaskan, maka tujuan dapat dicapai; jika kalimat tidak memuaskan, maka masalah perencanaan tidak mungkin.
3. Dengan asumsi model ditemukan, ekstrak dari model tersebut variabel-variabel yang mewakili tindakan dan ditetapkan $True$. Bersama-sama mereka mewakili rencana untuk mencapai tujuan.

Prosedur perencanaan proposisional, SATPLAN, ditunjukkan pada Gambar 7.22. Prosedur ini mengimplementasikan ide dasar yang baru saja diberikan, dengan satu perubahan. Karena agen tidak tahu berapa banyak langkah yang diperlukan untuk mencapai tujuan, algoritme mencoba setiap kemungkinan jumlah langkah t, hingga beberapa panjang rencana maksimum yang dapat dibayangkan T_{max}. Dengan cara ini, dijamin akan menemukan rencana terpendek jika ada. Karena cara SATPLAN mencari solusi, pendekatan ini tidak dapat digunakan dalam lingkungan yang dapat diamati sebagian; SATPLAN hanya akan menetapkan variabel yang tidak dapat diamati ke nilai yang dibutuhkannya untuk membuat solusi.

```
function SATPLAN(init, transition, goal, T max) returns solution or failure
    inputs: init, transition, goal, constitute a description of the problem
            T max, an upper limit for plan length

    for t = 0 to T max do
        cnf ← TRANSLATE-TO-SAT(init, transition, goal,t)
        model ← SAT-SOLVER(cnf )
        if model is not null then
            return EXTRACT-SOLUTION(model)
    return failure
```

Gambar 7.22 Algoritma SATPLAN. Masalah perencanaan diterjemahkan ke dalam kalimat CNF di mana tujuan dinyatakan berlaku pada langkah waktu t yang tetap dan aksioma disertakan untuk setiap langkah waktu hingga t. Jika algoritma pemuasan menemukan model, maka rencana diekstraksi dengan melihat simbol-simbol proposisi yang merujuk pada tindakan dan ditetapkan benar dalam model. Jika tidak ada model, maka proses diulang dengan tujuan dipindahkan satu langkah kemudian.

Langkah kunci dalam menggunakan SATPLAN adalah konstruksi basis pengetahuan. Jika dilihat sepintas, aksioma dunia wumpus di Bagian 7.7 mungkin tampak cukup untuk langkah 1(a) dan 1(b) di atas. Akan tetapi, ada perbedaan signifikan antara persyaratan untuk konsekuensi (seperti yang diuji oleh ASK) dan persyaratan untuk kepuasan. Pertimbangkan, misalnya, lokasi agen, awalnya [1, 1], dan anggaplah tujuan agen yang tidak ambisius adalah berada di [2, 1] pada waktu 1. Basis pengetahuan awal berisi $L^0_{1,1}$ dan tujuannya adalah $L^1_{2,1}$. Dengan menggunakan ASK, kita dapat membuktikan $L^1_{2,1}$ jika *Forward*^{0} dinyatakan, dan, yang meyakinkan, kita tidak dapat membuktikan $L^1_{2,1}$ jika, katakanlah, *Shoot*^{0} dinyatakan sebagai gantinya. Sekarang, SATPLAN akan menemukan rencana [*Forward*^{0}]; sejauh ini, baik-baik saja. Sayangnya, SATPLAN juga menemukan rencana [*Shoot*^{0}]. Bagaimana ini bisa terjadi? Untuk mengetahuinya, kami memeriksa model yang dibangun SATPLAN: model tersebut mencakup penugasan $L^0_{2,1}$, yaitu, agen dapat berada di [2, 1] pada waktu 1 dengan berada di sana pada waktu 0 dan menembak. Seseorang mungkin bertanya, "Bukankah kita mengatakan agen berada di [1, 1] pada waktu 0?" Ya, kami melakukannya, tetapi kami tidak memberi tahu agen bahwa agen tidak dapat berada di dua tempat sekaligus!

Untuk implikasi, $L^0_{2,1}$ tidak diketahui dan, oleh karena itu, tidak dapat digunakan dalam pembuktian; untuk pembuktian yang memuaskan, di sisi lain, $L^0_{2,1}$ tidak diketahui dan karenanya dapat ditetapkan ke nilai apa pun yang membantu menjadikan tujuan tersebut benar. Atas alasan ini, SATPLAN merupakan alat debugging yang baik untuk basis pengetahuan karena alat ini mengungkap tempat-tempat yang tidak memiliki pengetahuan. Dalam kasus khusus ini, kita dapat memperbaiki basis pengetahuan dengan menegaskan bahwa, pada setiap langkah waktu, agen berada tepat di satu lokasi, menggunakan kumpulan kalimat yang mirip dengan yang digunakan untuk menegaskan keberadaan tepat satu wumpus. untuk lokasi

menangani langkah waktu berikutnya. Perbaikan yang sama juga berfungsi untuk memastikan agen hanya memiliki satu orientasi.

Namun, SATPLAN memiliki lebih banyak kejutan. Yang pertama adalah menemukan model dengan tindakan yang tidak mungkin, seperti menembak tanpa anak panah. Untuk memahami alasannya, kita perlu melihat lebih cermat apa yang dikatakan aksioma status penerus (seperti Persamaan (7.3)) tentang tindakan yang prasyaratnya tidak terpenuhi. Aksioma memang memprediksi dengan benar bahwa tidak akan terjadi apa pun saat tindakan tersebut dijalankan, tetapi tidak mengatakan bahwa tindakan tersebut tidak dapat dijalankan! Untuk menghindari pembuatan rencana dengan tindakan ilegal, kita harus menambahkan **aksioma prasyarat** yang menyatakan bahwa kemunculan tindakan mengharuskan prasyarat terpenuhi. Misalnya, kita perlu mengatakan, untuk setiap waktu t, bahwa.

$$Shoot^t \Rightarrow HaveArrow^t$$

Ini memastikan bahwa jika rencana memilih tindakan Tembak kapan saja, agen harus memiliki anak panah pada saat itu. Kejutan kedua SATPLAN adalah terciptanya rencana dengan beberapa tindakan simultan. Misalnya, SATPLAN mungkin menghasilkan model di mana *Forward*^{0} dan *Shoot*^{0} bernilai benar, yang tidak diperbolehkan. Untuk menghilangkan masalah ini, kami memperkenalkan **aksioma pengecualian tindakan**: untuk setiap pasangan tindakan A_i^t dan A_j^t kami menambahkan aksioma

$$\neg A_i^t \vee \neg A_j^t$$

Dapat ditunjukkan bahwa berjalan maju dan menembak pada saat yang sama tidaklah terlalu sulit dilakukan, sedangkan, katakanlah, menembak dan meraih pada saat yang sama agak tidak praktis. Dengan memaksakan aksioma pengecualian tindakan hanya pada pasangan tindakan yang benar-benar saling mengganggu, kita dapat memungkinkan rencana yang mencakup beberapa tindakan simultan—dan karena SATPLAN menemukan rencana hukum terpendek, kita dapat yakin bahwa ia akan memanfaatkan kemampuan ini.

Singkatnya, SATPLAN menemukan model untuk kalimat yang berisi keadaan awal, tujuan, aksioma keadaan penerus, aksioma prasyarat, dan aksioma pengecualian tindakan. Dapat ditunjukkan bahwa kumpulan aksioma ini cukup, dalam arti bahwa tidak ada lagi "solusi" yang palsu. Model apa pun yang memenuhi kalimat proposisional akan menjadi rencana yang valid untuk masalah awal. Teknologi penyelesaian SAT modern membuat pendekatan ini cukup praktis. Misalnya, penyelesai gaya DPLL tidak mengalami kesulitan dalam menghasilkan solusi 11 langkah untuk contoh dunia wumpus yang ditunjukkan pada Gambar 7.2.

Bagian ini telah menjelaskan pendekatan deklaratif untuk konstruksi agen: agen bekerja dengan menggabungkan kalimat penegasan dalam basis pengetahuan dan melakukan inferensi logis. Pendekatan ini memiliki beberapa kelemahan yang tersembunyi dalam frasa seperti "untuk setiap waktu t" dan "untuk setiap kuadrat $[x, y]$." Untuk agen praktis apa pun, frasa ini harus diimplementasikan oleh kode yang menghasilkan contoh skema kalimat umum

secara otomatis untuk disisipkan ke dalam basis pengetahuan. Untuk dunia wumpus dengan ukuran yang wajar—yang sebanding dengan gim komputer berukuran kecil—kita mungkin memerlukan papan berukuran 100×100 dan 1000 langkah waktu, yang mengarah ke basis pengetahuan dengan puluhan atau ratusan juta kalimat. Hal ini tidak hanya menjadi tidak praktis, tetapi juga menggambarkan masalah yang lebih dalam: kita mengetahui sesuatu tentang dunia wumpus—yaitu, bahwa "fisika" bekerja dengan cara yang sama di semua kotak dan semua langkah waktu—yang tidak dapat kita ungkapkan secara langsung dalam bahasa logika proposisional. Untuk memecahkan masalah ini, kita memerlukan bahasa yang lebih ekspresif, bahasa di mana frasa seperti "untuk setiap waktu t" dan "untuk setiap kotak $[x, y]$" dapat ditulis dengan cara yang alami. Logika orde pertama, yang dijelaskan dalam Bab 8, adalah bahasa seperti itu; dalam logika orde pertama, dunia wumpus dengan ukuran dan durasi apa pun dapat dijelaskan dalam sekitar sepuluh kalimat, bukan sepuluh juta atau sepuluh triliun.

7.8 RINGKASAN

Kami telah memperkenalkan agen berbasis pengetahuan dan telah menunjukkan cara mendefinisikan logika yang dengannya agen tersebut dapat bernalar tentang dunia. Poin-poin utamanya adalah sebagai berikut:

- Agen cerdas membutuhkan pengetahuan tentang dunia untuk mencapai keputusan yang baik.
- Pengetahuan terkandung dalam agen dalam bentuk kalimat dalam bahasa representasi pengetahuan yang disimpan dalam basis pengetahuan.
- Agen berbasis pengetahuan terdiri dari basis pengetahuan dan mekanisme inferensi. Agen beroperasi dengan menyimpan kalimat tentang dunia dalam basis pengetahuannya, menggunakan mekanisme inferensi untuk menyimpulkan kalimat baru, dan menggunakan kalimat ini untuk memutuskan tindakan apa yang harus diambil.
- Bahasa representasi didefinisikan oleh sintaksisnya, yang menentukan struktur kalimat, dan semantiknya, yang mendefinisikan kebenaran setiap kalimat di setiap kemungkinan dunia atau model.
- Hubungan konsekuensi antara kalimat sangat penting untuk pemahaman kita tentang penalaran. Kalimat α mensyaratkan kalimat lain β jika β benar di semua dunia tempat α benar. Definisi yang setara mencakup validitas kalimat $\alpha \Rightarrow \beta$ dan ketidakpuasan kalimat $\alpha \wedge \neg\beta$.
- Inferensi adalah proses memperoleh kalimat baru dari kalimat lama. Algoritma inferensi yang baik hanya memperoleh kalimat yang disyaratkan; algoritma yang lengkap memperoleh semua kalimat yang disyaratkan.
- Logika proposisional adalah bahasa sederhana yang terdiri dari simbol proposisi dan konektif logis. Logika ini dapat menangani proposisi yang diketahui benar, diketahui salah, atau sama sekali tidak diketahui.

- Kumpulan model yang mungkin, dengan kosakata proposisional yang tetap, adalah terbatas, sehingga keterikatan dapat diperiksa dengan menghitung model. Algoritma inferensi pemeriksaan model yang efisien untuk logika proposisional mencakup metode penelusuran balik dan pencarian lokal dan sering kali dapat memecahkan masalah besar dengan cepat.
- Aturan inferensi adalah pola inferensi yang baik yang dapat digunakan untuk menemukan bukti. Aturan resolusi menghasilkan algoritma inferensi yang lengkap untuk basis pengetahuan yang dinyatakan dalam bentuk normal konjungtif. Forward chaining dan backward chaining adalah algoritma penalaran yang sangat alami untuk basis pengetahuan dalam bentuk Horn.
- Metode pencarian lokal seperti WALKSAT dapat digunakan untuk menemukan solusi. Algoritma tersebut baik tetapi tidak lengkap.
- Estimasi status logis melibatkan pemeliharaan kalimat logis yang menggambarkan serangkaian status yang mungkin konsisten dengan riwayat observasi. Setiap langkah pembaruan memerlukan inferensi menggunakan model transisi lingkungan, yang dibangun dari aksioma status penerus yang menentukan bagaimana setiap fasih berubah.
- Keputusan dalam agen logis dapat dibuat dengan penyelesaian SAT: menemukan model yang mungkin yang menentukan urutan tindakan mendatang yang mencapai tujuan. Pendekatan ini hanya berfungsi untuk lingkungan yang sepenuhnya dapat diamati atau tanpa sensor.
- Logika proposisional tidak dapat diskalakan ke lingkungan dengan ukuran tak terbatas karena tidak memiliki kekuatan ekspresif untuk menangani waktu, ruang, dan pola hubungan universal di antara objek secara ringkas.

BAB 8
LOGIKA ORDE PERTAMA

8.1 REPRESENTASI DITINJAU KEMBALI

Pada bagian ini, kami membahas hakikat bahasa representasi. Pembahasan kami memotivasi pengembangan logika orde pertama, bahasa yang jauh lebih ekspresif daripada logika proposisional yang diperkenalkan pada Bab 7. Kami melihat logika proposisional dan jenis bahasa lain untuk memahami apa yang berhasil dan apa yang gagal. Pembahasan kami akan bersifat sepintas, meringkas pemikiran, percobaan, dan kesalahan selama berabad-abad menjadi beberapa paragraf.

Bahasa pemrograman (seperti C++ atau Java atau Lisp) sejauh ini merupakan kelas bahasa formal terbesar yang umum digunakan. Program itu sendiri, dalam arti langsung, hanya

mewakili proses komputasi. Struktur data dalam program dapat mewakili fakta; misalnya, sebuah program dapat menggunakan larik 4 × 4 untuk mewakili isi dunia wumpus. Dengan demikian, pernyataan bahasa pemrograman $Dunia\ [2,2] \leftarrow\ Lubang$ adalah cara yang cukup alami untuk menegaskan bahwa ada lubang di kotak [2,2]. (Representasi semacam itu dapat dianggap $ad\ hoc$; sistem basis data dikembangkan secara tepat untuk menyediakan cara yang lebih umum dan tidak bergantung pada domain untuk menyimpan dan mengambil fakta.) Bahasa pemrograman tidak memiliki mekanisme umum untuk memperoleh fakta dari fakta lain; setiap pembaruan pada struktur data dilakukan dengan prosedur khusus domain yang detailnya diperoleh oleh programmer dari pengetahuannya sendiri tentang domain tersebut. Pendekatan prosedural ini dapat dikontraskan dengan sifat deklaratif logika proposisional, di mana pengetahuan dan inferensi terpisah, dan inferensi sepenuhnya tidak bergantung pada domain.

Kelemahan kedua dari struktur data dalam program (dan basis data, dalam hal ini) adalah tidak adanya cara mudah untuk mengatakan, misalnya, "Ada lubang di [2,2] atau [3,1]" atau "Jika wumpus ada di [1,1] maka dia tidak ada di [2,2]." Program dapat menyimpan satu nilai untuk setiap variabel, dan beberapa sistem memungkinkan nilai tersebut menjadi "tidak diketahui," tetapi mereka tidak memiliki ekspresi yang diperlukan untuk menangani informasi parsial.

Logika proposisional adalah bahasa deklaratif karena semantiknya didasarkan pada hubungan kebenaran antara kalimat dan kemungkinan dunia. Logika ini juga memiliki daya ekspresif yang cukup untuk menangani informasi parsial, menggunakan disjungsi dan negasi. Logika proposisional memiliki sifat ketiga yang diinginkan dalam bahasa representasi, yaitu komposisionalitas. Dalam bahasa komposisionalitas, makna kalimat merupakan fungsi dari makna bagian-bagiannya. Misalnya, makna "$S_{1,4} \wedge S_{1,2}$" terkait dengan makna "$S_{1,4}$" dan "$S_{1,2}$" Akan sangat aneh jika "$S_{1,4}$" berarti ada bau busuk di kotak [1,4] dan "$S_{1,2}$" berarti ada bau busuk di kotak [1,2], tetapi "$S_{1,4} \wedge S_{1,2}$" berarti Prancis dan Polandia bermain imbang 1–1 dalam pertandingan kualifikasi hoki es minggu lalu. Jelas, nonkomposisionalitas membuat sistem penalaran menjadi jauh lebih sulit. Namun, seperti yang kita lihat di Bab 7, logika proposisional tidak memiliki kekuatan ekspresif untuk menggambarkan lingkungan dengan banyak objek secara ringkas. Misalnya, kita dipaksa untuk menulis aturan terpisah tentang angin sepoi-sepoi dan lubang untuk setiap kotak, seperti

$$B_{1,1} \Leftrightarrow \left(P_{1,2} \vee P_{2,1}\right).$$

Di sisi lain, dalam bahasa Inggris, tampaknya cukup mudah untuk mengatakan, sekali dan untuk selamanya, “Kotak yang berdekatan dengan lubang itu berangin sepoi-sepoi.” Sintaksis dan semantik bahasa Inggris entah bagaimana memungkinkan untuk menggambarkan lingkungan secara ringkas.

Bahasa Pemikiran

Bahasa alami (seperti bahasa Inggris atau Spanyol) memang sangat ekspresif. Kami berhasil menulis hampir seluruh buku ini dalam bahasa alami, dengan hanya sesekali beralih

ke bahasa lain (termasuk logika, matematika, dan bahasa diagram). Ada tradisi panjang dalam linguistik dan filsafat bahasa yang memandang bahasa alami sebagai bahasa representasi pengetahuan deklaratif. Jika kita dapat mengungkap aturan bahasa alami, kita dapat menggunakannya dalam sistem representasi dan penalaran serta memperoleh manfaat dari miliaran halaman yang telah ditulis dalam bahasa alami.

Pandangan modern tentang bahasa alami adalah bahwa bahasa alami berfungsi sebagai media komunikasi, bukan representasi murni. Ketika seorang pembicara menunjuk dan berkata, “Lihat!” pendengar akan tahu bahwa, katakanlah, Superman akhirnya muncul di atas atap rumah. Namun, kita tidak ingin mengatakan bahwa kalimat "Lihat!" mewakili fakta itu. Sebaliknya, makna kalimat bergantung pada kalimat itu sendiri dan pada konteks di mana kalimat itu diucapkan. Jelas, seseorang tidak dapat menyimpan kalimat seperti "Lihat!" dalam basis pengetahuan dan berharap untuk memulihkan maknanya tanpa juga menyimpan representasi konteks—yang menimbulkan pertanyaan tentang bagaimana konteks itu sendiri dapat direpresentasikan. Bahasa alami juga mengalami ambiguitas, masalah bagi bahasa representasi. Seperti yang dikatakan Pinker: "Ketika orang berpikir tentang musim semi, tentunya mereka tidak bingung apakah mereka sedang memikirkan musim atau sesuatu yang berbau harum—dan jika satu kata dapat berhubungan dengan dua pikiran, pikiran tidak dapat menjadi kata-kata."

Hipotesis Sapir–Whorf yang terkenal mengklaim bahwa pemahaman kita tentang dunia sangat dipengaruhi oleh bahasa yang kita gunakan. Whorf menulis, “Kita memilah-milah alam, mengaturnya menjadi konsep, dan menganggapnya penting sebagaimana yang kita lakukan, terutama karena kita merupakan pihak dalam kesepakatan untuk mengaturnya dengan cara ini—kesepakatan yang berlaku di seluruh komunitas tutur kita dan dikodifikasikan dalam pola bahasa kita.” Memang benar bahwa komunitas tutur yang berbeda membagi dunia secara berbeda. Orang Prancis memiliki dua kata “chaise” dan “fauteuil,” untuk konsep yang dibahas oleh penutur bahasa Inggris dengan satu kata: “chair.” Namun, penutur bahasa Inggris dapat dengan mudah mengenali kategori fauteuil dan memberinya nama—kira-kira “open-arm chair”—jadi apakah bahasa benar-benar membuat perbedaan? Whorf terutama mengandalkan intuisi dan spekulasi, tetapi dalam tahun-tahun berikutnya kita benar-benar memiliki data nyata dari studi antropologis, psikologis, dan neurologis.

Misalnya, dapatkah Anda mengingat frasa mana dari dua frasa berikut yang menjadi pembuka Bagian 8.1?

“Dalam bagian ini, kita membahas hakikat bahasa representasi .. .”

“Bagian ini membahas topik bahasa representasi pengetahuan .. .”

Wanner melakukan percobaan serupa dan menemukan bahwa subjek membuat pilihan yang tepat pada tingkat peluang—sekitar 50% dari waktu—tetapi mengingat isi bacaan dengan akurasi lebih dari 90%. Ini menunjukkan bahwa orang memproses kata-kata untuk membentuk semacam representasi nonverbal.

Yang lebih menarik adalah kasus di mana suatu konsep sama sekali tidak ada dalam suatu bahasa. Penutur bahasa Aborigin Australia Guugu Yimithirr tidak memiliki kata untuk arah relatif, seperti depan, belakang, kanan, atau kiri. Sebaliknya, mereka menggunakan arah

absolut, dengan mengatakan, misalnya, padanan dari "lengan utara saya sakit." Perbedaan bahasa ini membuat perbedaan dalam perilaku: penutur bahasa Guugu Yimithirr lebih baik dalam menavigasi di medan terbuka, sementara penutur bahasa Inggris lebih baik dalam menempatkan garpu di sebelah kanan piring.

Bahasa juga tampaknya memengaruhi pikiran melalui fitur tata bahasa yang tampaknya sewenang-wenang seperti jenis kelamin kata benda. Misalnya, "jembatan" adalah maskulin dalam bahasa Spanyol dan feminin dalam bahasa Jerman. Boroditsky meminta subjek untuk memilih kata sifat dalam bahasa Inggris untuk menggambarkan foto jembatan tertentu. Penutur bahasa Spanyol memilih kata sifat *big*, *dangerous*, *strong*, dan *towering*, sedangkan penutur bahasa Jerman memilih *beautiful*, *elegant*, *fragile*, dan *slender*. Kata-kata dapat berfungsi sebagai titik jangkar yang memengaruhi cara kita memandang dunia. Loftus dan Palmer menunjukkan kepada subjek percobaan sebuah film tentang kecelakaan mobil. Subjek yang ditanya "Seberapa cepat mobil-mobil itu melaju saat mereka saling bersentuhan?" melaporkan kecepatan rata-rata 32 km/jam, sementara subjek yang ditanya pertanyaan dengan kata "smashed" alih-alih "contacted" melaporkan kecepatan 41 km/jam untuk mobil yang sama dalam film yang sama.

Dalam sistem penalaran logika orde pertama yang menggunakan CNF, kita dapat melihat bahwa bentuk linguistik $''\neg(A \vee B)''$ dan $''\neg A \wedge \neg B''$ adalah sama karena kita dapat melihat ke dalam sistem dan melihat bahwa kedua kalimat tersebut disimpan sebagai bentuk CNF kanonik yang sama. Bisakah kita melakukan itu dengan otak manusia? Sampai saat ini jawabannya adalah "tidak," tetapi sekarang menjadi "mungkin." Mitchell $et\ al.$ menempatkan subjek dalam mesin fMRI *(functional magnetic resonance imaging)*, menunjukkan kepada mereka kata-kata seperti "seledri," dan mencitrakan otak mereka. Para peneliti kemudian dapat melatih program komputer untuk memprediksi, dari citra otak, kata apa yang telah ditunjukkan kepada subjek. Diberi dua pilihan (misalnya, "seledri" atau "pesawat terbang"), sistem memprediksi dengan benar 77% dari waktu. Sistem ini bahkan dapat memprediksi pada tingkat di atas peluang untuk kata-kata yang belum pernah dilihatnya dalam gambar fMRI sebelumnya (dengan mempertimbangkan gambar kata-kata terkait) dan untuk orang-orang yang belum pernah dilihatnya sebelumnya (membuktikan bahwa fMRI mengungkapkan beberapa tingkat representasi umum di antara orang-orang). Jenis pekerjaan ini masih dalam tahap awal, tetapi fMRI (dan teknologi pencitraan lainnya seperti elektrofisiologi intrakranial (Sahin *et al.,* 2009)) menjanjikan untuk memberi kita ide-ide yang jauh lebih konkret tentang seperti apa representasi pengetahuan manusia.

Dari sudut pandang logika formal, merepresentasikan pengetahuan yang sama dalam dua cara yang berbeda sama sekali tidak membuat perbedaan; fakta-fakta yang sama akan dapat diperoleh dari salah satu representasi. Namun, dalam praktiknya, satu representasi mungkin memerlukan lebih sedikit langkah untuk memperoleh kesimpulan, yang berarti bahwa seorang pemikir dengan sumber daya terbatas dapat mencapai kesimpulan menggunakan satu representasi tetapi tidak yang lain. Untuk tugas-tugas nondeduktif seperti belajar dari pengalaman, hasil tentu saja bergantung pada bentuk representasi yang digunakan. Kami tunjukkan di Bab 18 bahwa ketika program pembelajaran

mempertimbangkan dua teori dunia yang mungkin, yang keduanya konsisten dengan semua data, cara yang paling umum untuk memutus ikatan adalah memilih teori yang paling ringkas—dan itu tergantung pada bahasa yang digunakan untuk merepresentasikan teori. Dengan demikian, pengaruh bahasa pada pemikiran tidak dapat dihindari bagi agen mana pun yang melakukan pembelajaran.

Menggabungkan yang Terbaik dari Bahasa Formal dan Alami

Kita dapat mengadopsi dasar logika proposisional—semantik deklaratif dan komposisional yang tidak bergantung pada konteks dan tidak ambigu—dan membangun logika yang lebih ekspresif di atas dasar itu, meminjam ide representasional dari bahasa alami sambil menghindari kekurangannya. Ketika kita melihat sintaksis bahasa alami, elemen yang paling jelas adalah kata benda dan frasa kata benda yang merujuk ke objek (kotak, lubang, wumpuses) dan kata kerja dan frasa kata kerja yang merujuk ke hubungan di antara objek (berangin, berdekatan dengan, menembak). Beberapa hubungan ini adalah fungsi—hubungan di mana hanya ada satu "nilai" untuk "input" tertentu. Mudah untuk mulai membuat daftar contoh objek, hubungan, dan fungsi:

- Objek: orang, rumah, angka, teori, Ronald McDonald, warna, pertandingan bisbol, perang, abad ...
- Hubungan: ini bisa berupa hubungan unary atau properti seperti merah, bulat, palsu, prima, bertingkat .. ., atau hubungan n-ary yang lebih umum seperti saudara dari, lebih besar dari, di dalam, bagian dari, memiliki warna, terjadi setelah, memiliki, berada di antara, ...
- Fungsi: ayah dari, sahabat, inning ketiga, satu lebih dari, awal dari ...
  Memang, hampir semua pernyataan dapat dianggap merujuk pada objek dan properti atau hubungan. Berikut ini beberapa contohnya:
- "Satu tambah dua sama dengan tiga."
  Objek: satu, dua, tiga, satu tambah dua; Relasi: sama dengan; Fungsi: tambah. ("Satu tambah dua" adalah nama untuk objek yang diperoleh dengan menerapkan fungsi "tambah" pada objek "satu" dan "dua." "Tiga" adalah nama lain untuk objek ini.)
- "Kotak yang berdekatan dengan wumpus berbau busuk."
  Objek: wumpus, kotak; Properti: berbau busuk; Relasi: berdekatan.
- "Raja Jahat John memerintah Inggris pada tahun 1200."
  Objek: John, Inggris, 1200; Relasi: memerintah; Properti: jahat, raja.

Bahasa logika tingkat pertama, yang sintaksis dan semantiknya kita definisikan di bagian berikutnya, dibangun di sekitar objek dan relasi. Bahasa ini sangat penting bagi matematika, filsafat, dan kecerdasan buatan justru karena bidang-bidang tersebut—dan memang, sebagian besar kehidupan manusia sehari-hari—dapat dianggap berguna sebagai bidang yang berhubungan dengan objek dan relasi di antara objek-objek tersebut. Logika tingkat pertama juga dapat mengungkapkan fakta tentang beberapa atau semua objek di alam semesta. Hal ini memungkinkan seseorang untuk merepresentasikan hukum atau aturan umum, seperti pernyataan "Kotak yang berdekatan dengan wumpus berbau busuk.

"Perbedaan utama antara logika proposisional dan logika tingkat pertama terletak pada komitmen ontologis yang dibuat oleh setiap bahasa—yaitu, apa yang diasumsikannya tentang hakikat *realitas.* Secara matematis, komitmen ini diungkapkan melalui hakikat model formal yang dengannya kebenaran kalimat didefinisikan. Misalnya, logika proposisional mengasumsikan bahwa ada fakta yang berlaku atau tidak berlaku di dunia. Setiap fakta dapat berada dalam salah satu dari dua keadaan: benar atau salah, dan setiap model menetapkan benar atau salah untuk setiap simbol proposisi (lihat Bagian 7.4.2). Logika tingkat pertama mengasumsikan lebih banyak lagi; yaitu, bahwa dunia terdiri dari objek-objek dengan hubungan tertentu di antara mereka yang berlaku atau tidak berlaku. Model formal karenanya lebih rumit daripada model untuk logika proposisional. Logika tujuan khusus membuat komitmen ontologis lebih jauh lagi; misalnya, logika temporal mengasumsikan bahwa fakta berlaku pada waktu tertentu dan bahwa waktu tersebut (yang dapat berupa titik atau interval) diurutkan. Dengan demikian, logika tujuan khusus memberikan jenis objek tertentu (dan aksioma tentangnya) status "kelas satu" dalam logika, daripada sekadar mendefinisikannya dalam basis pengetahuan. Logika tingkat tinggi memandang hubungan dan fungsi yang dirujuk oleh logika tingkat pertama sebagai objek dalam dirinya sendiri. Hal ini memungkinkan seseorang untuk membuat pernyataan tentang semua hubungan—misalnya, seseorang dapat ingin mendefinisikan apa artinya hubungan bersifat transitif. Tidak seperti kebanyakan logika tujuan khusus, logika tingkat tinggi secara ketat lebih ekspresif daripada logika tingkat pertama, dalam arti bahwa beberapa kalimat logika tingkat tinggi tidak dapat diungkapkan oleh sejumlah kalimat logika tingkat pertama yang terbatas.

Logika juga dapat dicirikan oleh komitmen epistemologisnya—kemungkinan status pengetahuan yang dimungkinkannya sehubungan dengan setiap fakta. Dalam logika proposisional dan logika tingkat pertama, sebuah kalimat mewakili sebuah fakta dan pelakunya percaya bahwa kalimat itu benar, percaya bahwa kalimat itu salah, atau tidak memiliki pendapat. Logika-logika ini karenanya memiliki tiga kemungkinan status pengetahuan mengenai kalimat apa pun. Sistem yang menggunakan teori probabilitas, di sisi lain, dapat memiliki tingkat keyakinan apa pun, mulai dari 0 (ketidakpercayaan total) hingga 1 (keyakinan total). Misalnya, agen dunia wumpus probabilistik mungkin percaya bahwa wumpus berada di [1,3] dengan probabilitas 0,75. Komitmen ontologis dan epistemologis dari lima logika yang berbeda dirangkum dalam Gambar 8.1.

Bahasa	Komitmen Ontologis (Apa yang ada di dunia)	Komitmen Epistemologis (Apa yang diyakini agen tentang fakta)
Logika proposisional	Fakta	benar/salah/tidak diketahui
Logika orde pertama	fakta, objek, hubungan	benar/salah/tidak diketahui
Logika temporal	fakta, objek, hubungan, waktu	benar/salah/tidak diketahui
Teori probabilitas	fakta	tingkat keyakinan $\in$ [0, 1]
Logika fuzzy	fakta dengan tingkat kebenaran $\in$ [0, 1]	nilai interval yang diketahui

Gambar 8.1 Bahasa formal dan komitmen ontologis dan epistemologisnya

Pada bagian berikutnya, kita akan membahas detail logika orde pertama. Sama seperti mahasiswa fisika yang membutuhkan keakraban dengan matematika, mahasiswa AI harus mengembangkan bakat untuk bekerja dengan notasi logika. Di sisi lain, penting juga untuk tidak terlalu peduli dengan hal-hal spesifik dari notasi logika—bagaimanapun juga, ada lusinan versi yang berbeda. Hal utama yang harus diingat adalah bagaimana bahasa memfasilitasi representasi yang ringkas dan bagaimana semantiknya mengarah pada prosedur penalaran yang baik.

8.2 SINTAKSIS DAN SEMANTIK LOGIKA ORDE PERTAMA

Kita mulai bagian ini dengan menentukan secara lebih tepat cara di mana kemungkinan dunia logika tingkat pertama mencerminkan komitmen ontologis terhadap objek dan relasi. Kemudian kita perkenalkan berbagai elemen bahasa, menjelaskan semantiknya seiring berjalannya waktu.

Model untuk Logika Tingkat Pertama

Ingat kembali dari Bab 7 bahwa model bahasa logika adalah struktur formal yang membentuk kemungkinan dunia yang sedang dipertimbangkan. Setiap model menghubungkan kosakata kalimat logika dengan elemen-elemen kemungkinan dunia, sehingga kebenaran kalimat apa pun dapat ditentukan. Dengan demikian, model untuk logika proposisional menghubungkan simbol proposisi dengan nilai kebenaran yang telah ditetapkan sebelumnya. Model untuk logika tingkat pertama jauh lebih menarik. Pertama, mereka memiliki objek di dalamnya! Domain model adalah kumpulan objek atau elemen domain yang dikandungnya. Domain harus tidak kosong—setiap kemungkinan dunia harus berisi setidaknya satu objek. Secara matematis, tidak masalah apa objek-objek ini—yang penting adalah berapa banyak objek yang ada di setiap model tertentu—tetapi untuk tujuan pedagogis, kita akan menggunakan contoh konkret. Gambar 8.2 menunjukkan model dengan lima objek: Richard si Hati Singa, Raja Inggris dari tahun 1189 hingga 1199; adik laki-lakinya, Raja John yang jahat, yang memerintah dari tahun 1199 hingga 1215; kaki kiri Richard dan John; dan sebuah mahkota.

Objek-objek dalam model tersebut dapat saling terkait dalam berbagai cara. Dalam gambar, Richard dan John adalah saudara. Secara formal, relasi hanyalah himpunan tupel objek yang saling terkait. (Tupel adalah kumpulan objek yang disusun dalam urutan tetap dan ditulis dengan tanda kurung siku di sekeliling objek.) Jadi, relasi persaudaraan dalam model ini adalah himpunan.

$$\{\langle \text{Richard the Lionheart}, \text{King John}\rangle, \langle \text{King John}, \text{Richard the Lionheart}\rangle\}. \quad (8.1)$$

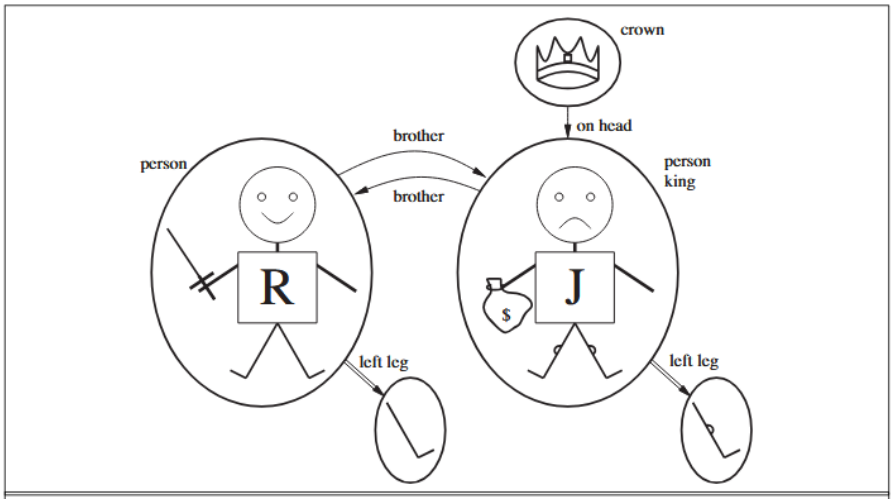
(Di sini kami telah menamai objek dalam bahasa Inggris, tetapi Anda dapat, jika Anda mau, mengganti nama objek dengan gambar secara mental.) Mahkota berada di kepala Raja John, jadi relasi "di kepala" hanya berisi satu tupel, ⟨mahkota, Raja John⟩. Relasi "saudara" dan "di kepala" adalah relasi biner—yaitu, keduanya menghubungkan pasangan objek. Model

tersebut juga berisi relasi unary, atau properti: properti "orang" berlaku untuk Richard dan John; properti "raja" hanya berlaku untuk John (mungkin karena Richard sudah meninggal saat ini); dan properti "mahkota" hanya berlaku untuk mahkota.

Jenis relasi tertentu sebaiknya dianggap sebagai fungsi, karena objek tertentu harus terkait dengan tepat satu objek dengan cara ini. Misalnya, setiap orang memiliki satu kaki kiri, jadi model tersebut memiliki fungsi "kaki kiri" unary yang mencakup pemetaan berikut:

$$\begin{aligned} &\langle \text{Richard the Lionheart} \rangle \rightarrow \text{Richard left leg} \\ &\langle \text{King John} \rangle \rightarrow \text{John's left leg.} \end{aligned} \tag{8.2}$$

Secara tegas, model dalam logika orde pertama memerlukan fungsi total, yaitu, harus ada nilai untuk setiap tupel masukan. Jadi, mahkota harus memiliki kaki kiri dan begitu pula setiap kaki kiri.



Gambar 8.2 Sebuah model yang berisi lima objek, dua relasi biner, tiga relasi unary (ditunjukkan dengan label pada objek), dan satu fungsi unary, kaki kiri.

Ada solusi teknis untuk masalah yang sulit ini yang melibatkan objek "tak terlihat" tambahan yang merupakan kaki kiri dari segala sesuatu yang tidak memiliki kaki kiri, termasuk dirinya sendiri. Untungnya, selama seseorang tidak membuat pernyataan tentang kaki kiri dari hal-hal yang tidak memiliki kaki kiri, hal-hal teknis ini tidak penting. Sejauh ini, kami telah menjelaskan elemen-elemen yang mengisi model untuk logika orde pertama. Bagian penting lainnya dari sebuah model adalah hubungan antara elemen-elemen tersebut dan kosakata kalimat logis, yang akan kami jelaskan selanjutnya.

Simbol dan Interpretasi

Sekarang kita beralih ke sintaks logika tingkat pertama. Pembaca yang tidak sabar dapat memperoleh deskripsi lengkap dari tata bahasa formal pada Gambar 8.3. Elemen sintaksis dasar logika tingkat pertama adalah simbol yang mewakili objek, relasi, dan fungsi. Oleh karena itu, simbol-simbol tersebut hadir dalam tiga jenis: simbol konstan, yang mewakili objek;

simbol predikat, yang mewakili relasi; dan simbol fungsi, yang mewakili fungsi. Kita mengadopsi konvensi bahwa simbol-simbol ini akan dimulai dengan huruf kapital. Misalnya, kita dapat menggunakan simbol konstan $Richard$ dan $John$; simbol predikat $Brother, OnHead, Person, King$, dan $Crown$; dan simbol fungsi $LeftLeg$. Seperti halnya simbol proposisi, pilihan nama sepenuhnya terserah pada pengguna. Setiap simbol predikat dan fungsi hadir dengan aritas yang menentukan jumlah argumen.

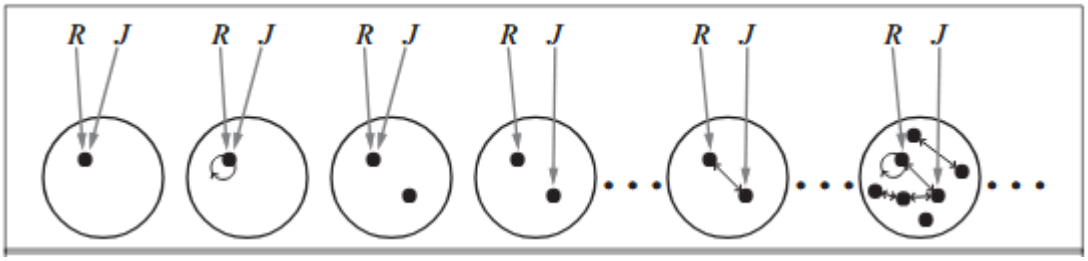
Seperti dalam logika proposisional, setiap model harus menyediakan informasi yang diperlukan untuk menentukan apakah kalimat yang diberikan benar atau salah. Dengan demikian, selain objek, relasi, dan fungsi, setiap model menyertakan interpretasi yang menentukan secara pasti objek, relasi, dan fungsi mana yang dirujuk oleh simbol konstanta, predikat, dan fungsi. Salah satu interpretasi yang mungkin untuk contoh kita—yang oleh ahli logika disebut interpretasi yang dimaksud—adalah sebagai berikut:

- $Richard$ merujuk pada Richard si Hati Singa dan John merujuk pada Raja John yang jahat.
- *Brother* merujuk pada relasi persaudaraan, yaitu, himpunan tupel objek yang diberikan dalam Persamaan (8.1); $OnHead$ merujuk pada relasi "di atas kepala" yang berlaku antara mahkota dan Raja John; $Person, King$, dan $Crown$ merujuk pada himpunan objek yang merupakan person, raja, dan mahkota.
- $LeftLeg$ merujuk pada fungsi "kaki kiri", yaitu, pemetaan yang diberikan dalam Persamaan (8.2).

Tentu saja, ada banyak interpretasi lain yang mungkin. Misalnya, satu interpretasi memetakan Richard ke mahkota dan John ke kaki kiri Raja John. Ada lima objek dalam model, jadi ada 25 kemungkinan interpretasi hanya untuk simbol konstan Richard dan John. Perhatikan bahwa tidak semua objek perlu memiliki nama—misalnya, interpretasi yang dimaksud tidak menyebutkan mahkota atau kaki. Mungkin juga suatu objek memiliki beberapa nama; ada interpretasi di mana Richard dan John merujuk pada mahkota. Jika Anda merasa kemungkinan ini membingungkan, ingatlah bahwa, dalam logika proposisional, sangat mungkin untuk memiliki model di mana Cloudy dan Sunny keduanya benar; tugas basis pengetahuan adalah menyingkirkan model yang tidak konsisten dengan pengetahuan kita.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
&\mid \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term}, \ldots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} &: \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Gambar 8.3 Sintaks logika orde pertama dengan kesetaraan, ditetapkan dalam bentuk Backus–Naur (lihat halaman 1060 jika Anda tidak familier dengan notasi ini). Urutan operator ditetapkan, dari tertinggi ke terendah. Urutan kuantifier sedemikian rupa sehingga kuantifier berlaku atas semua yang ada di sebelah kanannya.



Gambar 8.4 Beberapa anggota himpunan semua model untuk suatu bahasa dengan dua simbol konstan, R dan J, dan satu simbol relasi biner. Penafsiran setiap simbol konstan ditunjukkan oleh panah abu-abu. Dalam setiap model, objek terkait dihubungkan oleh panah.

Singkatnya, model dalam logika orde pertama terdiri dari sekumpulan objek dan interpretasi yang memetakan simbol konstan ke objek, simbol predikat ke relasi pada objek tersebut, dan simbol fungsi ke fungsi pada objek tersebut. Sama seperti logika proposisional, implikasi, validitas, dan sebagainya didefinisikan dalam bentuk semua model yang mungkin. Untuk mendapatkan gambaran tentang seperti apa kumpulan semua model yang mungkin, lihat Gambar 8.4.

Gambar tersebut menunjukkan bahwa model bervariasi dalam jumlah objek yang dikandungnya—dari satu hingga tak terbatas—dan dalam cara simbol konstan dipetakan ke objek. Jika ada dua simbol konstan dan satu objek, maka kedua simbol tersebut harus merujuk ke objek yang sama; tetapi ini masih dapat terjadi bahkan dengan lebih banyak objek. Ketika ada lebih banyak objek daripada simbol konstan, beberapa objek tidak akan memiliki nama. Karena jumlah model yang mungkin tidak terbatas, pengecekan implikasi dengan enumerasi semua model yang mungkin tidak layak untuk logika orde pertama (tidak seperti logika proposisional). Bahkan jika jumlah objek dibatasi, jumlah kombinasinya bisa sangat besar. (Lihat Latihan 8.5.) Untuk contoh pada Gambar 8.4, terdapat 137.506.194.466 model dengan enam objek atau kurang.

Istilah

Suatu istilah adalah ekspresi logika yang merujuk pada suatu objek. Oleh karena itu, simbol konstan adalah istilah, tetapi tidak selalu mudah untuk memiliki simbol yang berbeda untuk menamai setiap objek. Misalnya, dalam bahasa Inggris, kita mungkin menggunakan ekspresi "King John's left leg" daripada memberi nama pada kakinya. Untuk itulah simbol fungsi digunakan: alih-alih menggunakan simbol konstan, kita menggunakan $LeftLeg(John)$. Dalam kasus umum, istilah kompleks dibentuk oleh simbol fungsi yang diikuti oleh daftar istilah dalam tanda kurung sebagai argumen terhadap simbol fungsi. Penting untuk diingat bahwa istilah kompleks hanyalah jenis nama yang rumit. Ini bukan "panggilan subrutin" yang "mengembalikan nilai." Tidak ada subrutin $LeftLeg$ yang mengambil seseorang sebagai input dan mengembalikan kaki. Kita dapat bernalar tentang kaki kiri (misalnya, dengan menyatakan aturan umum bahwa setiap orang memilikinya dan kemudian menyimpulkan bahwa John pasti memilikinya) tanpa pernah memberikan definisi tentang $LeftLeg$. Ini adalah sesuatu yang tidak dapat dilakukan dengan subrutin dalam bahasa pemrograman.

Semantik formal istilah-istilah itu mudah dipahami. Pertimbangkan istilah $f(t_1, \ldots, t_n)$. Simbol fungsi f merujuk ke beberapa fungsi dalam model (sebut saja F); istilah-istilah argumen merujuk ke objek-objek dalam domain (sebut saja $d_1, \ldots, d_n$); dan istilah secara keseluruhan merujuk ke objek yang merupakan nilai fungsi F yang diterapkan ke $d_1, \ldots, d_n$. Misalnya, anggaplah simbol fungsi $LeftLeg$ merujuk ke fungsi yang ditunjukkan dalam Persamaan (8.2) dan John merujuk ke Raja John, maka $LeftLeg(John)$ merujuk ke kaki kiri Raja John. Dengan cara ini, interpretasi tersebut menetapkan referen setiap istilah.

Kalimat Atomik

Sekarang setelah kita memiliki kedua istilah untuk merujuk ke objek dan simbol predikat untuk merujuk ke relasi, kita dapat menggabungkan keduanya untuk membuat kalimat atomik yang menyatakan fakta. Kalimat atomik (atau singkatnya atom) dibentuk dari simbol predikat yang secara opsional diikuti oleh daftar istilah dalam tanda kurung, seperti

$$Brother(Richard, John).$$

Hal ini menyatakan, berdasarkan interpretasi yang dimaksudkan yang diberikan sebelumnya, bahwa Richard si Hati Singa adalah saudara Raja John. Kalimat atomik dapat memiliki istilah kompleks sebagai argumen. Jadi,

$$Married(Father(Richard), Mother(John))$$

menyatakan bahwa ayah Richard si Hati Singa menikah dengan ibu Raja John (sekali lagi, berdasarkan interpretasi yang sesuai).

Kalimat atomik benar dalam model tertentu jika relasi yang dirujuk oleh simbol predikat berlaku di antara objek yang dirujuk oleh argumen.

Kalimat Kompleks

Kita dapat menggunakan konjungsi logis untuk menyusun kalimat yang lebih kompleks, dengan sintaksis dan semantik Yang sama seperti dalam kalkulus proposisional. Berikut adalah empat kalimat yang benar dalam model Gambar 8.2 berdasarkan interpretasi yang kita maksud:

$\neg Brother(LeftLeg(Richard), John)$
$Brother(Richard, John) \wedge Brother(John, Richard)$
$King(Richard) \vee King(John)$
$\neg King(Richard) \Rightarrow King(John)$.

Kuantifier

Setelah kita memiliki logika yang memperbolehkan objek, wajar saja jika kita ingin mengekspresikan properti dari seluruh koleksi objek, alih-alih menyebutkan objek berdasarkan nama. Kuantifier memungkinkan kita melakukan ini. Logika orde pertama berisi dua kuantifier standar, yang disebut universal dan eksistensial.

Kuantifikasi Universal (∀)

Ingat kesulitan yang kita hadapi di Bab 7 dengan ekspresi aturan umum dalam logika proposisional. Aturan seperti "Kotak yang berdekatan dengan wumpus berbau busuk" dan "Semua raja adalah orang" adalah dasar dari logika orde pertama. Kita membahas yang pertama di Bagian 8.3. Aturan kedua, "Semua raja adalah orang," ditulis dalam logika orde pertama sebagai

$$\forall\, x\ King(x) \Rightarrow Person(x)\ .$$

∀ biasanya diucapkan sebagai “Untuk semua .. .”. (Ingat bahwa huruf A terbalik berarti “semua.”) Jadi, kalimat tersebut berbunyi, “Untuk semua x, jika x adalah raja, maka x adalah orang.” Simbol x disebut variabel. Berdasarkan konvensi, variabel adalah huruf kecil. Variabel adalah istilah tersendiri, dan dengan demikian juga dapat berfungsi sebagai argumen fungsi—misalnya, $LeftLeg(x)$. Istilah tanpa variabel disebut istilah dasar.

Secara intuitif, kalimat $\forall\ x\ P$, di mana P adalah ekspresi logis apa pun, menyatakan bahwa P benar untuk setiap objek x. Lebih tepatnya, $\forall\ x\ P$ benar dalam model tertentu jika P benar dalam semua kemungkinan interpretasi yang diperluas yang dibangun dari interpretasi yang diberikan dalam model, di mana setiap interpretasi yang diperluas menentukan elemen domain yang dirujuk oleh x.

Ini terdengar rumit, tetapi sebenarnya ini hanyalah cara hati-hati untuk menyatakan makna intuitif dari kuantifikasi universal. Perhatikan model yang ditunjukkan pada Gambar 8.2 dan interpretasi yang dimaksudkan yang menyertainya. Kita dapat memperluas interpretasi dalam lima cara:

$$
\begin{aligned}
x &\rightarrow Richard\ the\ Lionheart,\\
x &\rightarrow King\ John,\\
x &\rightarrow Richard's\ left\ leg,\\
x &\rightarrow John's\ left\ leg,\\
x &\rightarrow the\ crown.
\end{aligned}
$$

Kalimat yang dikuantifikasi secara universal $\forall\ x\ King(x)\ \Rightarrow\ Person(x)$ adalah benar dalam model asli jika kalimat $King(x)\ \Rightarrow\ Person(x)$ adalah benar di bawah masing-masing dari lima interpretasi yang diperluas. Artinya, kalimat yang dikuantifikasi secara universal setara dengan menegaskan lima kalimat berikut:

Richard si Hati Singa adalah seorang raja ⇒ Richard si Hati Singa adalah seorang manusia.

Raja John adalah seorang raja ⇒ Raja John adalah seorang manusia.

Kaki kiri Richard adalah seorang raja ⇒ Kaki kiri Richard adalah seorang manusia.

Kaki kiri John adalah seorang raja ⇒ Kaki kiri John adalah seorang manusia.

Mahkota adalah seorang raja ⇒ mahkota adalah seorang manusia.

Mari kita cermati rangkaian pernyataan ini dengan saksama. Karena, dalam model kita, Raja John adalah satu-satunya raja, kalimat kedua menegaskan bahwa ia adalah seorang pribadi, seperti yang kita harapkan. Namun, bagaimana dengan empat kalimat lainnya, yang tampaknya mengklaim tentang kaki dan mahkota? Apakah itu bagian dari makna "Semua raja adalah pribadi"? Faktanya, empat pernyataan lainnya benar dalam model tersebut, tetapi tidak mengklaim apa pun tentang kualifikasi pribadi kaki, mahkota, atau bahkan Richard. Ini karena tidak satu pun dari objek ini adalah seorang raja. Dengan melihat tabel kebenaran untuk ⇒ (Gambar 7.8 pada halaman 246), kita melihat bahwa implikasinya benar setiap kali premisnya salah—terlepas dari kebenaran kesimpulannya. Jadi, dengan menegaskan kalimat yang dikuantifikasi secara universal, yang setara dengan menegaskan seluruh daftar implikasi individual, kita akhirnya menegaskan kesimpulan aturan hanya untuk objek-objek yang premisnya benar dan tidak mengatakan apa pun tentang individu-individu yang premisnya salah. Dengan demikian, definisi tabel kebenaran dari ⇒ ternyata sempurna untuk menulis aturan umum dengan kuantifier universal.

Kesalahan umum, yang sering dilakukan bahkan oleh pembaca tekun yang telah membaca paragraf ini beberapa kali, adalah menggunakan konjungsi alih-alih implikasi. Kalimat

$$\forall\, x\ King(x)\ \wedge\ Person(x)$$

akan setara dengan menegaskan

Richard si Hati Singa adalah seorang raja Λ Richard si Hati Singa adalah seorang manusia, Raja John adalah seorang raja Λ Raja John adalah seorang manusia,
Kaki kiri Richard adalah seorang raja Λ Kaki kiri Richard adalah seorang manusia,
dan seterusnya. Jelas, ini tidak menggambarkan apa yang kita inginkan.

Kuantifikasi Eksistensial ($\exists$)

Kuantifikasi universal membuat pernyataan tentang setiap objek. Demikian pula, kita dapat membuat pernyataan tentang beberapa objek di alam semesta tanpa menamainya, dengan menggunakan kuantifier eksistensial. Untuk mengatakan, misalnya, bahwa Raja John memiliki mahkota di kepalanya, kita menulis

$$\exists\, x\ Crown(x)\ \Lambda\ OnHead(x, John).$$

$\exists x$ diucapkan “Ada x sedemikian rupa sehingga .. .” atau “Untuk beberapa x...”.

Secara intuitif, kalimat $\exists x\ P$ mengatakan bahwa P benar untuk setidaknya satu objek x. Lebih tepatnya, $\exists x\ P$ benar dalam model tertentu jika P benar dalam setidaknya satu interpretasi yang diperluas yang menetapkan x ke elemen domain. Yaitu, setidaknya satu dari yang berikut ini benar:

Richard si Hati Singa adalah mahkota Λ Richard si Hati Singa ada di kepala John;
Raja John adalah mahkota Λ Raja John ada di kepala John;
Kaki kiri Richard adalah mahkota Λ Kaki kiri Richard ada di kepala John;
Kaki kiri John adalah mahkota Λ Kaki kiri John ada di kepala John;
Mahkota adalah mahkota Λ Mahkota ada di kepala John.

Pernyataan kelima benar dalam model tersebut, jadi kalimat asli yang dikuantifikasi secara eksistensial benar dalam model tersebut. Perhatikan bahwa, menurut definisi kita, kalimat tersebut juga akan benar dalam model di mana Raja John mengenakan dua mahkota. Ini sepenuhnya konsisten dengan kalimat asli "Raja John memiliki mahkota di kepalanya."Sama seperti $\Rightarrow$ yang tampaknya menjadi konektif alami untuk digunakan dengan $\forall$, Λ adalah konektif alami untuk digunakan dengan $\exists$. Menggunakan Λ sebagai konektif utama dengan $\forall$ menghasilkan pernyataan yang terlalu kuat dalam contoh di bagian sebelumnya; menggunakan $\Rightarrow$ dengan $\exists$ biasanya menghasilkan pernyataan yang sangat lemah. Perhatikan kalimat berikut:

$$\exists x\ Crown(x) \Rightarrow\ OnHead(x, John)\ .$$

Di permukaan, ini mungkin tampak seperti penafsiran yang wajar dari kalimat kita. Dengan menerapkan semantik, kita melihat bahwa kalimat tersebut menyatakan bahwa setidaknya salah satu pernyataan berikut ini benar:

Richard si Hati Singa adalah mahkota ⇒ Richard si Hati Singa ada di kepala John;

Raja John adalah mahkota ⇒ Raja John ada di kepala John;

Kaki kiri Richard adalah mahkota ⇒ Kaki kiri Richard ada di kepala John;

dan seterusnya. Sekarang implikasinya benar jika premis dan kesimpulannya benar, atau jika premisnya salah. Jadi jika Richard si Hati Singa bukan mahkota, maka pernyataan pertama benar dan eksistensial terpenuhi. Jadi, kalimat implikasi yang dikuantifikasi secara eksistensial benar setiap kali objek apa pun gagal memenuhi premis; karenanya kalimat seperti itu sebenarnya tidak banyak bicara sama sekali.

Kuantifier Bersarang

Kita sering ingin mengekspresikan kalimat yang lebih kompleks menggunakan beberapa quantifier. Kasus yang paling sederhana adalah ketika quantifier berjenis sama. Misalnya, "Saudara laki-laki adalah saudara kandung" dapat ditulis sebagai

$$\forall x \; \forall y \; Brother(x, y) \; \Rightarrow \; Sibling(x, y) \; .$$

Kuantifier berurutan dengan tipe yang sama dapat ditulis sebagai satu kuantifier dengan beberapa variabel. Misalnya, untuk mengatakan bahwa hubungan persaudaraan adalah hubungan simetris, kita dapat menulis

$$\forall x, y \; Sibling(x, y) \; e \; Sibling(y, x) \; .$$

Dalam kasus lain, kita akan memiliki campuran. "Setiap orang mencintai seseorang" berarti bahwa untuk setiap orang, ada seseorang yang dicintainya:

$$\forall x \; \exists \; y \; Loves(x, y) \; .$$

Di sisi lain, untuk mengatakan "Ada seseorang yang dicintai oleh semua orang," kita menulis

$$\exists \; y \; \forall \; x \; Loves(x, y).$$

Oleh karena itu, urutan kuantifikasi sangat penting. Akan menjadi lebih jelas jika kita menyisipkan tanda kurung. $\forall x \; (\exists y \; Loves(x, y))$ mengatakan bahwa setiap orang memiliki sifat tertentu, yaitu, sifat bahwa mereka mencintai seseorang. Di sisi lain, $\exists y \; (\forall x \; Loves(x, y))$ mengatakan bahwa seseorang di dunia ini memiliki sifat tertentu, yaitu sifat dicintai oleh semua orang.

Kebingungan dapat muncul ketika dua quantifier digunakan dengan nama variabel yang sama. Pertimbangkan kalimat

$$\forall\, x\ (Crown(x)\ V\ (\exists\, x\ Brother(Richard\ , x)))\ .$$

Di sini x dalam $Brother(Richard, x)$ dikuantifikasi secara eksistensial. Aturannya adalah bahwa variabel tersebut termasuk dalam quantifier terdalam yang menyebutkannya; maka variabel tersebut tidak akan tunduk pada kuantifikasi lain. Cara lain untuk memikirkannya adalah ini: $\exists x\ Brother(Richard\ , x)$ adalah kalimat tentang Richard (bahwa ia memiliki saudara laki-laki), bukan tentang x; jadi meletakkan $\forall x$ di luarnya tidak akan berpengaruh. Kalimat tersebut dapat ditulis dengan baik sebagai $\exists z\ Brother(Richard, z)$. Karena hal ini dapat menjadi sumber kebingungan, kita akan selalu menggunakan nama variabel yang berbeda dengan quantifier bersarang.

Hubungan antara ∀ dan ∃

Kedua kuantifier tersebut sebenarnya saling terkait erat, melalui negasi. Menegaskan bahwa semua orang tidak menyukai parsnip sama dengan menegaskan bahwa tidak ada orang yang menyukainya, dan sebaliknya:

$\forall x\ \neg Likes(x, Parsnips\ )$ setara dengan $\neg\exists x\ Likes(x, Parsnips\ )$.

Kita dapat melangkah lebih jauh: "semua orang menyukai es krim" berarti tidak ada orang yang tidak menyukai es krim:

$\forall x\ Likes(x, IceCream\ )$ setara dengan $\neg\exists x\ \neg Likes(x, IceCream\ )$.

Karena ∀ benar-benar merupakan konjungsi atas semesta objek dan ∃ merupakan disjungsi, tidak mengherankan bahwa keduanya mematuhi aturan De Morgan. Aturan De Morgan untuk kalimat terkuantifikasi dan tidak terkuantifikasi adalah sebagai berikut:

$$\begin{aligned}
\forall\, x\ \neg P\ &\equiv\ \neg\exists\, x\ P & \neg(P\ \vee\ Q)\ &\equiv\ \neg P\ \wedge\ \neg Q\\
\neg\forall\, x\ P\ &\equiv\ \exists\, x\ \neg P & \neg(P\ \wedge\ Q)\ &\equiv\ \neg P\ \vee\ \neg Q\\
\forall\, x\ P\ &\equiv\ \neg\exists\, x\ \neg P & P\ \wedge\ Q\ &\equiv\ \neg(\neg P\ \vee\ \neg Q)\\
\exists\, x\ P\ &\equiv\ \neg\forall\, x\ \neg P & P\ \vee\ Q\ &\equiv\ \neg(\neg P\ \wedge\ \neg Q)\ .
\end{aligned}$$

Kesetaraan

Logika tingkat pertama mencakup satu cara lagi untuk membuat kalimat atomik, selain menggunakan predikat dan istilah seperti yang dijelaskan sebelumnya. Kita dapat menggunakan simbol kesetaraan untuk menandakan bahwa dua istilah merujuk pada objek yang sama. Misalnya,

$$Father(John)\ =\ Henry$$

mengatakan bahwa objek yang dirujuk oleh $Father(John)$ dan objek yang dirujuk oleh $Henry$ adalah sama. Karena interpretasi menentukan referen dari istilah apa pun, menentukan kebenaran kalimat kesetaraan hanyalah masalah melihat bahwa referen dari dua istilah tersebut adalah objek yang sama.

Simbol kesetaraan dapat digunakan untuk menyatakan fakta tentang fungsi tertentu, seperti yang baru saja kita lakukan untuk simbol $Father$. Simbol ini juga dapat digunakan dengan negasi untuk menegaskan bahwa dua istilah bukanlah objek yang sama. Untuk mengatakan bahwa Richard memiliki setidaknya dua saudara laki-laki, kita akan menulis

$$\exists x, y\ Brother(x, Richard\ )\ \Lambda\ Brother(y, Richard\ )\ \Lambda\ \neg(x = y).$$

Kalimat

$$\exists x, y\ Brother(x, Richard\ )\ \Lambda\ Brother(y, Richard)$$

tidak memiliki makna yang dimaksudkan. Secara khusus, hal itu benar dalam model Gambar 8.2, di mana Richard hanya memiliki satu saudara laki-laki. Untuk melihat ini, pertimbangkan interpretasi yang diperluas di mana x dan y diberikan kepada Raja John. Penambahan $\neg(x = y)$ mengesampingkan model tersebut. Notasi $x \neq y$ terkadang digunakan sebagai singkatan untuk $\neg(x = y)$.

Semantik Alternatif?

Melanjutkan contoh dari bagian sebelumnya, misalkan kita percaya bahwa Richard memiliki dua saudara laki-laki, John dan Geoffrey. Bisakah kita menangkap keadaan ini dengan menegaskan

$$Brother(John, Richard)\ \Lambda\ Brother(Geoffrey, Richard)\ ? \qquad (8.3)$$

Tidak sepenuhnya. Pertama, pernyataan ini benar dalam model di mana Richard hanya memiliki satu saudara laki-laki—kita perlu menambahkan $John \neq Geoffrey$. Kedua, kalimat tersebut tidak mengesampingkan model di mana Richard memiliki lebih banyak saudara selain John dan Geoffrey. Dengan demikian, terjemahan yang benar dari "Saudara Richard adalah John dan Geoffrey" adalah sebagai berikut:

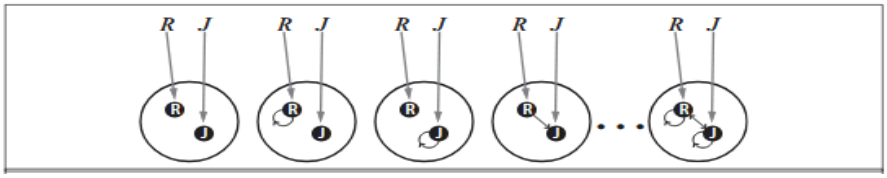
$$Brother(John, Richard)\ \Lambda\ Brother(Geoffrey, Richard)\ \Lambda\ John \neq Geoffrey$$
$$\Lambda \forall x\ Brother(x, Richard) \Rightarrow (x = John\ V\ x = Geoffrey).$$

Untuk banyak tujuan, ini tampaknya jauh lebih rumit daripada ekspresi bahasa alami yang sesuai. Akibatnya, manusia dapat membuat kesalahan dalam menerjemahkan pengetahuan mereka ke dalam logika tingkat pertama, yang mengakibatkan perilaku tidak intuitif dari sistem penalaran logis yang menggunakan pengetahuan tersebut. Dapatkah kita merancang semantik yang memungkinkan ekspresi logis yang lebih lugas?

Satu usulan yang sangat populer dalam sistem basis data berfungsi sebagai berikut. Pertama, kami bersikeras bahwa setiap simbol konstan merujuk ke objek yang berbeda—yang disebut asumsi nama-unik. Kedua, kami berasumsi bahwa kalimat atomik yang tidak diketahui kebenarannya sebenarnya salah—asumsi dunia tertutup. Terakhir, kami menerapkan penutupan domain, yang berarti bahwa setiap model tidak memuat lebih banyak elemen domain daripada yang diberi nama oleh simbol konstanta. Berdasarkan semantik yang dihasilkan, yang kami sebut semantik basis data untuk membedakannya dari semantik standar logika orde pertama, kalimat Persamaan (8.3) memang menyatakan bahwa dua saudara Richard adalah John dan Geoffrey. Semantik basis data juga digunakan dalam sistem pemrograman logika, seperti yang dijelaskan dalam Bagian 9.4.5.

Adalah instruktif untuk mempertimbangkan himpunan semua model yang mungkin berdasarkan semantik basis data untuk kasus yang sama seperti yang ditunjukkan pada Gambar 8.4. Gambar 8.5 menunjukkan beberapa model, mulai dari model tanpa tupel yang memenuhi relasi hingga model dengan semua tupel yang memenuhi relasi. Dengan dua objek, ada empat tupel dua elemen yang mungkin, jadi ada $2^4 = 16$ subset tupel yang berbeda yang dapat memenuhi relasi. Jadi, ada 16 model yang mungkin secara keseluruhan—jauh lebih sedikit daripada model yang tak terhingga jumlahnya untuk semantik orde pertama standar. Di sisi lain, semantik basis data memerlukan pengetahuan pasti tentang apa yang terkandung di dunia.

Contoh ini memunculkan poin penting: tidak ada satu semantik yang "tepat" untuk logika. Kegunaan semantik yang diusulkan bergantung pada seberapa ringkas dan intuitifnya semantik tersebut dalam mengekspresikan jenis pengetahuan yang ingin kita tulis, dan pada seberapa mudah dan alaminya mengembangkan aturan inferensi yang sesuai. Semantik basis data paling berguna ketika kita yakin tentang identitas semua objek yang dijelaskan dalam basis pengetahuan dan ketika kita memiliki semua fakta yang ada; dalam kasus lain, semantik basis data cukup janggal. Untuk sisa bab ini, kita mengasumsikan semantik standar sambil mencatat contoh-contoh di mana pilihan ini mengarah pada ekspresi yang rumit.



Gambar 8.5 Beberapa anggota himpunan semua model untuk bahasa dengan dua simbol konstan, R dan J, dan satu simbol relasi biner, dalam semantik basis data. Penafsiran simbol konstan bersifat tetap, dan ada objek yang berbeda untuk setiap simbol konstan.

8.3 MENGGUNAKAN LOGIKA ORDE PERTAMA

Sekarang setelah kita mendefinisikan bahasa logika ekspresif, sekarang saatnya mempelajari cara menggunakannya. Cara terbaik untuk melakukannya adalah melalui contoh. Kita telah melihat beberapa kalimat sederhana yang mengilustrasikan berbagai aspek sintaksis

logika; di bagian ini, kita menyediakan representasi yang lebih sistematis dari beberapa **domain** sederhana. Dalam representasi pengetahuan, domain hanyalah sebagian dari dunia yang ingin kita ungkapkan sejumlah pengetahuannya. Kita mulai dengan deskripsi singkat tentang antarmuka $TELL/ASK$ untuk basis pengetahuan orde pertama. Kemudian kita melihat domain hubungan keluarga, angka, himpunan, dan daftar, dan dunia wumpus. Bagian berikutnya berisi contoh yang lebih substansial (sirkuit elektronik) dan Bab 12 membahas segala sesuatu di alam semesta.

Pernyataan dan Kueri dalam Logika Orde Pertama

Kalimat ditambahkan ke basis pengetahuan menggunakan $TELL$, persis seperti dalam logika proposisional. Kalimat seperti itu disebut pernyataan. Misalnya, kita dapat menegaskan bahwa John adalah seorang raja, Richard adalah seorang manusia, dan semua raja adalah manusia:

$$Tell(KB, King(John)).$$
$$Tell(KB, Person(Richard)).$$
$$Tell(KB, \forall x\ King(x)\ \Rightarrow\ Person(x)).$$

Kita dapat mengajukan pertanyaan pada basis pengetahuan menggunakan ASK. Misalnya,

$$ASK(KB, King(John))$$

mengembalikan true. Pertanyaan yang diajukan dengan ASK disebut query atau tujuan. Secara umum, setiap query yang secara logis disyaratkan oleh basis pengetahuan harus dijawab dengan tegas. Misalnya, mengingat dua pernyataan sebelumnya, query

$$ASK\big(KB, Person(John)\big)$$

juga harus mengembalikan true. Kita dapat mengajukan pertanyaan kuantitatif, seperti

$$ASK(KB, \exists x\ Person(x)).$$

Jawabannya memang benar, tetapi mungkin ini tidak membantu seperti yang kita harapkan. Ini seperti menjawab "Bisakah Anda memberi tahu saya jam berapa?" dengan "Ya." Jika kita ingin tahu nilai x yang membuat kalimat tersebut benar, kita akan memerlukan fungsi yang berbeda, ASKVARS, yang kita panggil dengan

$$ASKVARS(KB, Person(x))$$

dan yang menghasilkan aliran jawaban. Dalam kasus ini akan ada dua jawaban: $\{x/John\}$ dan $\{x/Richard\}$. Jawaban seperti itu disebut substitusi atau daftar pengikatan. ASKVARS biasanya disediakan untuk basis pengetahuan yang hanya terdiri dari klausa Horn, karena

dalam basis pengetahuan tersebut setiap cara untuk membuat kueri menjadi benar akan mengikat variabel ke nilai tertentu. Itu tidak terjadi dengan logika orde pertama; jika KB telah diberi tahu $King(John)\ V\ King(Richard\ )$, maka tidak ada pengikatan ke x untuk kueri $\exists x\ King(x)$, meskipun kueri tersebut benar.

Domain Kekerabatan

Contoh pertama yang kami pertimbangkan adalah domain hubungan keluarga, atau kekerabatan. Domain ini mencakup fakta seperti "Elizabeth adalah ibu dari Charles" dan "Charles adalah ayah dari William" dan aturan seperti "Nenek seseorang adalah ibu dari orang tuanya."

Jelas, objek dalam domain kami adalah orang. Kami memiliki dua predikat unary, $Male$ dan $Female$. Hubungan kekerabatan—orang tua, persaudaraan, perkawinan, dan sebagainya—diwakili oleh predikat biner: $Parent, Sibling,\ Brother, Sister,\ Child,$ $Daughter, Son,$ $Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt,$ dan $Uncle$. Kita menggunakan fungsi untuk $Mother$ dan $Father$, karena setiap orang memiliki tepat satu dari masing-masing fungsi ini (setidaknya menurut rancangan alam).

Kita dapat menelusuri setiap fungsi dan predikat, menuliskan apa yang kita ketahui dalam bentuk simbol-simbol lainnya. Misalnya, ibu seseorang adalah orangtua perempuannya:

$$\forall m, c\ Mother(c) = m \iff Female(m)\ \Lambda\ Parent(m, c)\ .$$

Suami seseorang adalah pasangan laki-laki seseorang:

$$\forall w, h\ Husband(h, w) \iff Male(h)\ \Lambda\ Spouse(h, w)\ .$$

Laki-laki dan perempuan adalah kategori yang terpisah:

$$\forall x\ Male(x) \iff \neg Female(x)\ .$$

Orangtua dan anak adalah hubungan terbalik:

$$\forall p, c\ Parent(p, c) \Leftrightarrow Child(c, p)\ .$$

Kakek-nenek adalah orangtua dari orangtua seseorang:

$$\forall g, c\ Grandparent(g, c) \Leftrightarrow \exists p\ Parent(g, p) \wedge Parent(p, c).$$

lain dari orang tua seseorang:

$$\forall x, y\ Sibling(x, y) \Leftrightarrow x \neq y \wedge \exists p\ (Parent(p, x) \wedge Parent(p, y)..$$

Kita dapat melanjutkan beberapa halaman lagi seperti ini, dan Latihan 8.14 meminta Anda untuk melakukan hal itu.

Masing-masing kalimat ini dapat dilihat sebagai aksioma domain kekerabatan, seperti yang dijelaskan dalam Bagian 7.1. Aksioma umumnya dikaitkan dengan domain matematika murni—kita akan melihat beberapa aksioma untuk angka sebentar lagi—tetapi aksioma diperlukan di semua domain. Aksioma menyediakan informasi faktual dasar yang darinya kesimpulan yang berguna dapat diturunkan. Aksioma kekerabatan kita juga merupakan definisi; aksioma memiliki bentuk $\forall x, y\ P(x, y) \Leftrightarrow \cdots$. Fungsi $Mother$ dan predikat $Husband, Male, Parent, Grandparent$, dan $Sibling$ dalam bentuk predikat lainnya. Definisi kami "berakhir" pada satu set predikat dasar $(Child, Spouse, dan\ Female)$ yang pada akhirnya mendefinisikan yang lainnya. Ini adalah cara alami untuk membangun representasi domain, dan ini analog dengan cara paket perangkat lunak dibangun oleh definisi subrutin yang berurutan dari fungsi pustaka primitif. Perhatikan bahwa tidak selalu ada satu set predikat primitif yang unik; kita juga bisa menggunakan $Parent, Spouse, dan\ Male$. Di beberapa domain, seperti yang kami tunjukkan, tidak ada set dasar yang dapat diidentifikasi dengan jelas.

Tidak semua kalimat logis tentang domain adalah aksioma. Beberapa adalah teorema—yaitu, mereka disyaratkan oleh aksioma. Misalnya, perhatikan pernyataan bahwa persaudaraan bersifat simetris:

$$\forall x, y\ Sibling(x, y) \Leftrightarrow Sibling(y, x)\ .$$

Apakah ini aksioma atau teorema? Faktanya, ini adalah teorema yang mengikuti secara logis dari aksioma yang mendefinisikan persaudaraan. Jika kita BERTANYA pada basis pengetahuan kalimat ini, maka hasilnya akan $true$. Dari sudut pandang logika murni, basis pengetahuan hanya perlu berisi aksioma dan tidak ada teorema, karena teorema tidak menambah rangkaian kesimpulan yang mengikuti dari basis pengetahuan. Dari sudut pandang praktis, teorema sangat penting untuk mengurangi biaya komputasi dalam memperoleh kalimat baru. Tanpa teorema, sistem penalaran harus mulai dari prinsip dasar setiap saat, seperti fisikawan yang harus memperoleh kembali aturan kalkulus untuk setiap masalah baru.

Tidak semua aksioma adalah definisi. Beberapa memberikan informasi yang lebih umum tentang predikat tertentu tanpa membentuk definisi. Memang, beberapa predikat tidak memiliki definisi lengkap karena kita tidak cukup tahu untuk mengkarakterisasikannya secara lengkap. Misalnya, tidak ada cara pasti yang jelas untuk melengkapi kalimat

$$\forall x\ Person(x) \Leftrightarrow \ldots$$

Untungnya, logika orde pertama memungkinkan kita untuk menggunakan predikat $Person$ tanpa mendefinisikannya secara lengkap. Sebaliknya, kita dapat menulis spesifikasi parsial dari properti yang dimiliki setiap orang dan properti yang menjadikan sesuatu sebagai orang:

$$\forall\ x\ Person(x) \Rightarrow \ldots$$
$$\forall\ x \ldots \Rightarrow\ Person(x)\ .$$

Aksioma juga dapat berupa "fakta biasa," seperti $Male(Jim)$ dan $Spouse\,(Jim, Laura)$. Fakta-fakta tersebut membentuk deskripsi dari contoh masalah tertentu, yang memungkinkan pertanyaan-pertanyaan tertentu untuk dijawab. Jawaban atas pertanyaan-pertanyaan ini kemudian akan menjadi teorema yang mengikuti aksioma. Sering kali, seseorang menemukan bahwa jawaban yang diharapkan tidak tersedia—misalnya, dari $Spouse(Jim, Laura)$ seseorang berharap (berdasarkan hukum di banyak negara) untuk dapat menyimpulkan $\neg Spouse(George, Laura)$; tetapi ini tidak mengikuti aksioma yang diberikan sebelumnya—bahkan setelah kita menambahkan $Jim \neq George$ seperti yang disarankan di Bagian 8.2.8. Ini adalah tanda bahwa ada aksioma yang hilang. Latihan 8.8 meminta pembaca untuk menyediakannya.

Angka, Himpunan, dan Daftar

Angka mungkin merupakan contoh paling jelas tentang bagaimana sebuah teori besar dapat dibangun dari inti aksioma yang sangat kecil. Kami menjelaskan di sini teori bilangan asli atau bilangan bulat non-negatif. Kami memerlukan predikat $NatNum$ yang akan berlaku untuk bilangan asli; kami memerlukan satu simbol konstanta, 0; dan kami memerlukan satu simbol fungsi, S (penerus). Aksioma Peano mendefinisikan bilangan asli dan penjumlahan. Bilangan asli didefinisikan secara rekursif:

$$NatNum(0)\,.$$
$$\forall n\; NatNum(n) \;\Rightarrow\; NatNum(S(n))\,.$$

Artinya, 0 adalah bilangan asli, dan untuk setiap objek n, jika n adalah bilangan asli, maka $S(n)$ adalah bilangan asli. Jadi bilangan asli adalah 0, $S(0)$, S(S(0)), dan seterusnya. (Setelah membaca Bagian 8.2.8, Anda akan melihat bahwa aksioma ini memperbolehkan bilangan asli lain selain bilangan asli yang biasa; lihat Latihan 8.12.) Kita juga memerlukan aksioma untuk membatasi fungsi penerus:

$$\forall\, n\; 0 \;\neq=\; S(n)\,.$$
$$\forall\, m, n\; m \;\neq=\; n \;\Rightarrow\; S(m) \neq=\; S(n).$$

Sekarang kita dapat mendefinisikan penjumlahan dalam bentuk fungsi penerus:

$$\forall\, m\; NatNum(m) \;\Rightarrow\; +(0, m) \;=\; m\,.$$
$$\forall\, m, n\; NatNum(m) \;\wedge\; NatNum(n) \;\Rightarrow\; +(S(m), n) \;=\; S(+(m, n))\,.$$

Aksioma pertama ini menyatakan bahwa menambahkan 0 ke sembarang bilangan asli m akan menghasilkan m itu sendiri. Perhatikan penggunaan simbol fungsi biner "+" dalam suku $+(m, 0)$; dalam matematika biasa, suku tersebut akan ditulis $m + 0$ menggunakan notasi **i**nfiks. (Notasi yang telah kita gunakan untuk logika orde pertama disebut **prefiks**.) Untuk

membuat kalimat kita tentang bilangan lebih mudah dibaca, kita mengizinkan penggunaan notasi infiks. Kita juga dapat menulis $S(n)$ sebagai $n + 1$, sehingga aksioma kedua menjadi

$$\forall m, n\ NatNum(m) \wedge NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1\ .$$

Aksioma ini mengurangi penjumlahan menjadi penerapan berulang fungsi penerus.

Penggunaan notasi infiks adalah contoh gula sintaksis, yaitu, perluasan atau singkatan dari sintaksis standar yang tidak mengubah semantik. Setiap kalimat yang menggunakan gula dapat "dihilangkan gulanya" untuk menghasilkan kalimat yang setara dalam logika orde pertama biasa.

Setelah kita memiliki penjumlahan, mudah untuk mendefinisikan perkalian sebagai penjumlahan berulang, eksponensiasi sebagai perkalian berulang, pembagian dan sisa bilangan bulat, bilangan prima, dan seterusnya. Dengan demikian, keseluruhan teori bilangan (termasuk kriptografi) dapat dibangun dari satu konstanta, satu fungsi, satu predikat, dan empat aksioma.

Domain himpunan juga mendasar bagi matematika serta penalaran akal sehat. (Faktanya, teori bilangan dapat didefinisikan dalam bentuk teori himpunan.) Kita ingin dapat merepresentasikan himpunan individual, termasuk himpunan kosong. Kita memerlukan cara untuk membangun himpunan dengan menambahkan elemen ke suatu himpunan atau mengambil gabungan atau irisan dua himpunan. Kita ingin mengetahui apakah suatu elemen merupakan anggota himpunan dan kita ingin membedakan himpunan dari objek yang bukan himpunan.

Kita akan menggunakan kosakata teori himpunan yang umum sebagai gula sintaksis. Himpunan kosong adalah suatu konstanta yang ditulis sebagai $\{\}$. Ada satu predikat unary, Himpunan, yang berlaku untuk himpunan. Predikat binernya adalah $x \in s$ (x adalah anggota himpunan s) dan $s_1 \subseteq s_2$ (himpunan s_1 adalah bagian himpunan, tidak harus milik sendiri, dari himpunan s_2). Fungsi binernya adalah $s_1 \cap s_2$ (irisan dua himpunan), $s_1 \cup s_2$ (gabungan dua himpunan), dan $\{x|s\}$ (himpunan yang dihasilkan dari elemen x yang berdampingan dengan himpunan s). Satu set aksioma yang mungkin adalah sebagai berikut:

1. Satu-satunya set adalah set kosong dan set yang dibuat dengan menempelkan sesuatu ke suatu set:

$$\forall\ s\ Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2\ Set(s_2) \wedge s = \{x \mid s_2\}$$

2. Set kosong tidak memiliki elemen yang ditempelkan ke dalamnya. Dengan kata lain, tidak ada cara untuk menguraikan {} menjadi set yang lebih kecil dan elemen:

$$\neg\exists x, s\ \{x|s\} = \{\}\ .$$

3. Menempelkan elemen yang sudah ada di dalam set tidak memiliki efek:

$$\forall x, s \; x \in s \Leftrightarrow s = \{x|s\}.$$

4. Satu-satunya anggota suatu set adalah elemen yang ditempelkan ke dalamnya. Kita nyatakan ini secara rekursif, dengan mengatakan bahwa x adalah anggota s jika dan hanya jika s sama dengan himpunan s_2 yang berdampingan dengan elemen y, di mana y sama dengan x atau x adalah anggota s_2:

$$\forall x, s \; x \in s \Leftrightarrow \exists y, s_2 \left(s = \{ y \mid s_2 \} \wedge (x = y \vee x \in s_2)\right).$$

5. Suatu himpunan merupakan bagian dari himpunan lain jika dan hanya jika semua anggota himpunan pertama merupakan anggota himpunan kedua:

$$\forall s_1, s_2 \; s_1 \subseteq s_2 \Leftrightarrow (\forall x \; x \in s_1 \Rightarrow \mathrm{x} \in s_2).$$

6. Dua himpunan sama jika dan hanya jika masing-masing merupakan bagian dari himpunan lainnya:

$$\forall s_1, s_2 (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge (s_2 \subseteq s_1$$

7. Sebuah objek berada di irisan dua himpunan jika dan hanya jika objek tersebut merupakan anggota kedua himpunan:

$$\forall x, s_1, s_2 \; x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2).$$

8. Sebuah objek berada di gabungan dua himpunan jika dan hanya jika objek tersebut merupakan anggota salah satu himpunan:

$$\forall x, s_1, s_2 \; x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2)$$

Daftar mirip dengan himpunan. Perbedaannya adalah daftar diurutkan dan elemen yang sama dapat muncul lebih dari satu kali dalam sebuah daftar. Kita dapat menggunakan kosakata Lisp untuk daftar: Nil adalah daftar konstan tanpa elemen; $Cons$, $Append$, $First$, dan $Rest$ adalah fungsi; dan Find adalah predikat yang melakukan untuk daftar apa yang dilakukan $Member$ untuk himpunan. $List?$ adalah predikat yang hanya berlaku untuk daftar. Seperti halnya himpunan, penggunaan gula sintaksis dalam kalimat logis yang melibatkan daftar adalah hal yang umum. Daftar kosong adalah []. Istilah $Cons(x, y)$, di mana y adalah daftar yang tidak kosong, ditulis $[x|y]$. Istilah $Cons(x, Nil\,)$ (yaitu, daftar yang berisi elemen x) ditulis sebagai $[x]$. Daftar yang berisi beberapa elemen, seperti $[A, B, C]$, sesuai dengan istilah bersarang $Cons(A, Cons\,(B, Cons\,(C, Nil\,)))$. Latihan 8.16 meminta Anda untuk menuliskan aksioma untuk daftar.

Dunia Wumpus

Beberapa aksioma logika proposisional untuk dunia wumpus diberikan dalam Bab 7. Aksioma orde pertama dalam bagian ini jauh lebih ringkas, menangkap dengan cara alami apa yang ingin kita katakan. Ingat bahwa agen wumpus menerima vektor persepsi dengan lima elemen. Kalimat orde pertama yang sesuai yang disimpan dalam basis pengetahuan harus menyertakan persepsi dan waktu terjadinya; jika tidak, agen akan bingung tentang kapan melihat apa. Kami menggunakan bilangan bulat untuk langkah waktu. Kalimat persepsi yang umum adalah

$$Percept([Stench, Breeze, Glitter, None, None],5) .$$

Di sini, $Percept$ adalah predikat biner, dan $Stench$ dan seterusnya adalah konstanta yang ditempatkan dalam daftar. Tindakan di dunia wumpus dapat direpresentasikan dengan istilah logis:

$$Turn\ (Right), Turn\ (Left), Forward, Shoot, Grab, Climb.$$

Untuk menentukan mana yang terbaik, program agen mengeksekusi kueri

$$ASKVARS(\exists a\ BestAction(a, 5)) ,$$

yang mengembalikan daftar pengikatan seperti $\{a/Grab\}$. Program agen kemudian dapat mengembalikan $Grab$ sebagai tindakan yang harus diambil. Data persepsi mentah menyiratkan fakta-fakta tertentu tentang status saat ini. Misalnya:

$$\forall\ t, s, g, m, c\ Percept([s, Breeze, g, m, c], t)\ \Rightarrow\ Breeze(t),$$

$$\forall\ t, s, b, m, c\ Percept([s, b, Glitter, m, c], t)\ \Rightarrow\ Glitter(t) ,$$

dan seterusnya. Aturan-aturan ini menunjukkan bentuk sepele dari proses penalaran yang disebut persepsi, yang kita pelajari secara mendalam di Bab 24. Perhatikan kuantifikasi selama waktu t. Dalam logika proposisional, kita memerlukan salinan setiap kalimat untuk setiap langkah waktu.

Perilaku "refleks" sederhana juga dapat diimplementasikan oleh kalimat implikasi yang dikuantifikasi.
Misalnya, kita memiliki

$$\forall\ t\ Glitter\ (t)\ \Rightarrow\ BestAction(Grab, t).$$

Mengingat persepsi dan aturan dari paragraf sebelumnya, ini akan menghasilkan kesimpulan yang diinginkan $BestAction(Grab, 5)$—yaitu, Grab adalah hal yang benar untuk dilakukan.

Kita telah merepresentasikan input dan output agen; sekarang saatnya merepresentasikan lingkungan itu sendiri. Mari kita mulai dengan objek. Kandidat yang jelas adalah kotak, lubang, dan wumpus. Kita dapat menamai setiap kotak— $Square_{1,2}$ dan seterusnya—tetapi fakta bahwa $Square_{1,2}$ dan $Square_{1,3}$ berdekatan harus menjadi fakta "tambahan", dan kita memerlukan satu fakta tersebut untuk setiap pasangan kotak. Lebih baik menggunakan istilah kompleks di mana baris dan kolom muncul sebagai bilangan bulat; misalnya, kita cukup menggunakan istilah daftar $[1,2]$. Kedekatan dua kotak dapat didefinisikan sebagai

$$\forall\, x, y, a, b\ Adjacent([x, y], [a, b]) \Leftrightarrow$$
$$(x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)) .$$

Kita dapat menamai setiap lubang, tetapi ini tidak tepat karena alasan yang berbeda: tidak ada alasan untuk membedakan lubang-lubang. Lebih mudah menggunakan predikat unary Pit yang berlaku untuk kotak-kotak yang berisi lubang-lubang. Terakhir, karena ada tepat satu wumpus, $Wumpus$ yang konstan sama bagusnya dengan predikat unary (dan mungkin lebih bermartabat dari sudut pandang wumpus).

Lokasi agen berubah seiring waktu, jadi kita menulis $At(Agent\,, s, t)$ yang berarti bahwa agen berada di kotak s pada waktu t. Kita dapat menentukan lokasi wumpus dengan 6t $At(Wumpus, [2, 2], t)$. Kita kemudian dapat mengatakan bahwa objek hanya dapat berada di satu lokasi pada satu waktu:

$$\forall\, x, s_1, s_2\ t\ At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2.$$

Berdasarkan lokasinya saat ini, agen dapat menyimpulkan sifat-sifat kotak dari sifat-sifat persepsinya saat ini. Misalnya, jika agen berada di kotak dan merasakan angin sepoi-sepoi, maka kotak itu berangin:

$$\forall s, t\ At(Agent, s, t)\ \wedge\ Breeze(t)\ \Rightarrow\ Breezy(s)\ .$$

Penting untuk mengetahui bahwa sebuah *kotak* berangin karena kita tahu bahwa lubang-lubang tidak dapat bergerak. Perhatikan bahwa $Breezy$ tidak memiliki argumen waktu.

Setelah menemukan tempat-tempat yang berangin (atau bau) dan, yang sangat penting, tidak berangin (atau tidak bau), agen dapat menyimpulkan di mana lubang-lubang itu berada (dan di mana wumpus berada). Sementara logika proposisional memerlukan aksioma terpisah untuk setiap kotak (lihat R_2 dan R_3 pada halaman 247) dan akan memerlukan serangkaian aksioma yang berbeda untuk setiap tata letak geografis dunia, logika orde pertama hanya memerlukan satu aksioma:

$$\forall s\ Breezy(s)\ \Leftrightarrow\ \exists r\ Adjacent(r, s)\ \wedge\ Pit(r)\ . \qquad (8.4)$$

Demikian pula, dalam logika orde pertama kita dapat mengukur dari waktu ke waktu, jadi kita hanya memerlukan satu aksioma status penerus untuk setiap predikat, daripada salinan yang berbeda untuk setiap langkah waktu. Misalnya, aksioma untuk anak panah (Persamaan (7.2) pada halaman 267) menjadi.

$$\forall\, t\; HaveArrow(t\,+\,1)\;\Leftrightarrow\;(HaveArrow(t)\;\wedge\;\neg Action(Shoot,t))$$

Dari kedua contoh kalimat ini, kita dapat melihat bahwa formulasi logika orde pertama tidak kurang ringkas daripada deskripsi bahasa Inggris asli yang diberikan dalam Bab 7. Pembaca diundang untuk membangun aksioma analog untuk lokasi dan orientasi agen; dalam kasus ini, aksioma tersebut mengukur ruang dan waktu. Seperti dalam kasus estimasi status proposisional, agen dapat menggunakan inferensi logis dengan aksioma semacam ini untuk melacak aspek-aspek dunia yang tidak diamati secara langsung. Bab 10 membahas lebih dalam tentang subjek aksioma status penerus orde pertama dan penggunaannya untuk membangun rencana.

8.4 REKAYASA PENGETAHUAN DALAM LOGIKA ORDE PERTAMA

Bagian sebelumnya menggambarkan penggunaan logika orde pertama untuk merepresentasikan pengetahuan dalam tiga domain sederhana. Bagian ini menggambarkan proses umum konstruksi basis pengetahuan—proses yang disebut rekayasa pengetahuan. Seorang insinyur pengetahuan adalah seseorang yang menyelidiki domain tertentu, mempelajari konsep apa yang penting dalam domain tersebut, dan membuat representasi formal dari objek dan hubungan dalam domain tersebut. Kami menggambarkan proses rekayasa pengetahuan dalam domain sirkuit elektronik yang seharusnya sudah cukup familiar, sehingga kami dapat berkonsentrasi pada isu representasional yang terlibat. Pendekatan yang kami ambil cocok untuk mengembangkan basis pengetahuan tujuan khusus yang domainnya dibatasi dengan cermat dan rentang kuerinya diketahui sebelumnya. Basis pengetahuan tujuan umum, yang mencakup berbagai pengetahuan manusia dan dimaksudkan untuk mendukung tugas-tugas seperti pemahaman bahasa alami, dibahas dalam Bab 12.

Proses Rekayasa Pengetahuan

Proyek rekayasa pengetahuan sangat bervariasi dalam hal konten, cakupan, dan kesulitan, tetapi semua proyek tersebut mencakup langkah-langkah berikut:

i. $Mengidentifikasi\ tugas$. Insinyur pengetahuan harus menggambarkan berbagai pertanyaan yang akan didukung oleh basis pengetahuan dan jenis fakta yang akan tersedia untuk setiap contoh masalah tertentu. Misalnya, apakah basis pengetahuan wumpus harus dapat memilih tindakan atau harus menjawab pertanyaan hanya tentang isi lingkungan? Apakah fakta sensor akan mencakup lokasi saat ini? Tugas akan menentukan pengetahuan apa yang harus direpresentasikan untuk menghubungkan contoh masalah dengan jawaban. Langkah ini analog dengan proses PEAS untuk merancang agen di Bab 2.

ii. $Merakit\ pengetahuan\ yang\ relevan.$ Insinyur pengetahuan mungkin sudah menjadi pakar dalam domain tersebut, atau mungkin perlu bekerja dengan pakar sungguhan untuk mengekstrak apa yang mereka ketahui—proses yang disebut akuisisi pengetahuan. Pada tahap ini, pengetahuan tidak direpresentasikan secara formal. Idenya adalah untuk memahami cakupan basis pengetahuan, sebagaimana ditentukan oleh tugas, dan untuk memahami cara kerja domain sebenarnya. Untuk dunia wumpus, yang didefinisikan oleh serangkaian aturan buatan, pengetahuan yang relevan mudah diidentifikasi. (Namun, perhatikan bahwa definisi kedekatan tidak diberikan secara eksplisit dalam aturan dunia wumpus.) Untuk domain nyata, masalah relevansi bisa jadi cukup sulit—misalnya, sistem untuk simulasi desain VLSI mungkin perlu atau tidak perlu memperhitungkan kapasitansi liar dan efek kulit.

iii. $Tentukan\ kosakata\ predikat, fungsi, dan\ konstanta.$ Yaitu, terjemahkan konsep penting tingkat domain ke dalam nama tingkat logika. Ini melibatkan banyak pertanyaan tentang gaya rekayasa pengetahuan. Seperti gaya pemrograman, ini dapat berdampak signifikan pada keberhasilan proyek pada akhirnya. Misalnya, haruskah lubang diwakili oleh objek atau oleh predikat unary pada kotak? Haruskah orientasi agen berupa fungsi atau predikat? Haruskah lokasi wumpus bergantung pada waktu? Setelah pilihan dibuat, hasilnya adalah kosakata yang dikenal sebagai ontologi domain. Kata ontologi berarti teori khusus tentang hakikat keberadaan atau eksistensi. Ontologi menentukan jenis hal apa yang ada, tetapi tidak menentukan sifat dan hubungan spesifiknya.

iv. $Kodekan\ pengetahuan\ umum\ tentang\ domain.$ Insinyur pengetahuan menuliskan aksioma untuk semua istilah kosakata. Ini menentukan (sejauh mungkin) makna istilah, yang memungkinkan pakar untuk memeriksa konten. Sering kali, langkah ini mengungkap kesalahpahaman atau kesenjangan dalam kosakata yang harus diperbaiki dengan kembali ke langkah 3 dan mengulangi prosesnya.

v. $Mengodekan\ deskripsi\ dari\ contoh\ masalah\ tertentu.$ Jika ontologi dipikirkan dengan baik, langkah ini akan mudah. Ini akan melibatkan penulisan kalimat atom sederhana tentang contoh konsep yang sudah menjadi bagian dari ontologi. Untuk agen logis, contoh masalah disediakan oleh sensor, sedangkan basis pengetahuan "tanpa tubuh" disediakan dengan kalimat tambahan dengan cara yang sama seperti program tradisional disediakan dengan data masukan.

vi. $Ajukan\ pertanyaan\ ke\ prosedur\ inferensi\ dan\ dapatkan\ jawaban.$ Di sinilah imbalannya: kita dapat membiarkan prosedur inferensi beroperasi pada aksioma dan fakta khusus masalah untuk memperoleh fakta yang ingin kita ketahui. Dengan demikian, kita terhindar dari kebutuhan untuk menulis algoritma solusi khusus aplikasi.

vii. $Debug\ basis\ pengetahuan.$ Sayangnya, jawaban atas pertanyaan jarang benar pada percobaan pertama. Lebih tepatnya, jawaban akan benar untuk $basis\ pengetahuan\ sebagaimana\ tertulis$, dengan asumsi bahwa prosedur inferensi itu benar, tetapi jawaban itu tidak akan menjadi jawaban yang diharapkan pengguna. Misalnya, jika sebuah aksioma tidak ada, beberapa pertanyaan tidak akan

dapat dijawab dari basis pengetahuan. Proses debugging yang cukup besar dapat terjadi. Aksioma yang tidak ada atau aksioma yang terlalu lemah dapat dengan mudah diidentifikasi dengan memperhatikan tempat-tempat di mana rantai penalaran berhenti secara tiba-tiba. Misalnya, jika basis pengetahuan menyertakan aturan diagnostik (lihat Latihan 8.13) untuk menemukan wumpus,

$$\forall s\ Smelly(s) \Rightarrow Adjacent(Home(Wumpus), s)\ ,$$

alih-alih bikondisional, maka agen tidak akan pernah dapat membuktikan tidak adanya wumpuses. Aksioma yang salah dapat diidentifikasi karena merupakan pernyataan yang salah tentang dunia. Misalnya, kalimat

$$\forall x\ NumOfLegs(x, 4) \Rightarrow Mammal(x)$$

salah untuk reptil, amfibi, dan, yang lebih penting, tabel. Kekeliruan kalimat ini dapat ditentukan secara independen dari basis pengetahuan lainnya. Sebaliknya, kesalahan umum dalam sebuah program terlihat seperti ini:

$$offset = position + 1.$$

Tidak mungkin untuk mengetahui apakah pernyataan ini benar tanpa melihat bagian program lainnya untuk melihat apakah, misalnya, offset digunakan untuk merujuk ke posisi saat ini, atau ke posisi di luar posisi saat ini, atau apakah nilai posisi diubah oleh pernyataan lain sehingga offset juga harus diubah lagi.

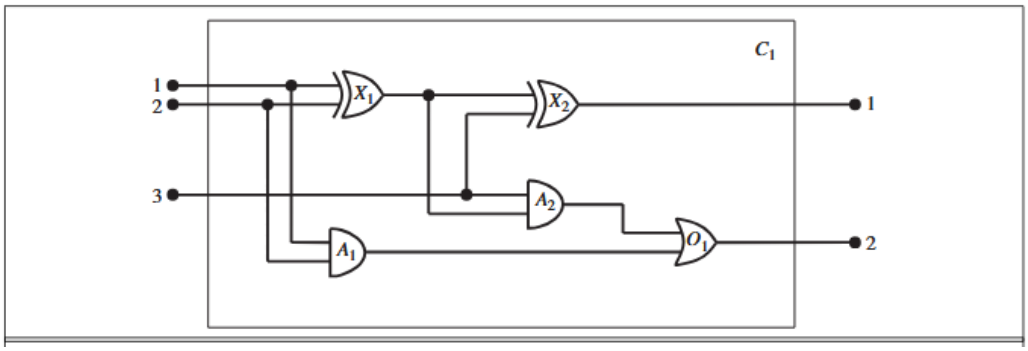
Untuk memahami proses tujuh langkah ini dengan lebih baik, sekarang kita terapkan pada contoh yang diperluas—ranah sirkuit elektronik.

Ranah Sirkuit Elektronik

Kita akan mengembangkan ontologi dan basis pengetahuan yang memungkinkan kita untuk bernalar tentang sirkuit digital seperti yang ditunjukkan pada Gambar 8.6. Kita mengikuti proses tujuh langkah untuk rekayasa pengetahuan.

Mengidentifikasi Tugas

Ada banyak tugas penalaran yang terkait dengan sirkuit digital. Pada tingkat tertinggi, seseorang menganalisis fungsionalitas sirkuit. Misalnya, apakah sirkuit pada Gambar 8.6 benar-benar menjumlah dengan benar? Jika semua masukan tinggi, berapa keluaran gerbang A2? Pertanyaan tentang struktur sirkuit juga menarik. Misalnya, apa saja gerbang yang terhubung ke terminal masukan pertama? Apakah rangkaian tersebut mengandung loop umpan balik? Ini akan menjadi tugas kita di bagian ini. Ada beberapa tingkat analisis yang lebih rinci, termasuk yang terkait dengan penundaan waktu, area rangkaian, konsumsi daya, biaya produksi, dan sebagainya. Setiap tingkat ini akan memerlukan pengetahuan tambahan.



Gambar 8.6 Rangkaian digital C1, yang dimaksudkan sebagai penjumlah penuh satu bit. Dua masukan pertama adalah dua bit yang akan dijumlahkan, dan masukan ketiga adalah bit pembawa. Keluaran pertama adalah penjumlahan, dan keluaran kedua adalah bit pembawa untuk penjumlah berikutnya. Rangkaian tersebut berisi dua gerbang XOR, dua gerbang AND, dan satu gerbang OR.

Kumpulkan Pengetahuan yang Relevan

Apa yang kita ketahui tentang rangkaian digital? Untuk tujuan kita, rangkaian tersebut terdiri dari kabel dan gerbang. Sinyal mengalir sepanjang kabel ke terminal input gerbang, dan setiap gerbang menghasilkan sinyal pada terminal output yang mengalir sepanjang kabel lainnya. Untuk menentukan sinyal apa yang akan dihasilkan, kita perlu mengetahui bagaimana gerbang mengubah sinyal inputnya. Ada empat jenis gerbang: gerbang AND, OR, dan XOR memiliki dua terminal input, dan gerbang NOT memiliki satu. Semua gerbang memiliki satu terminal output. Rangkaian, seperti gerbang, memiliki terminal input dan output.

Untuk bernalar tentang fungsionalitas dan konektivitas, kita tidak perlu berbicara tentang kabel itu sendiri, jalur yang dilaluinya, atau persimpangan tempat kabel-kabel itu bertemu. Yang penting adalah koneksi antar terminal—kita dapat mengatakan bahwa satu terminal keluaran terhubung ke terminal masukan lain tanpa harus mengatakan apa yang sebenarnya menghubungkannya. Faktor-faktor lain seperti ukuran, bentuk, warna, atau biaya berbagai komponen tidak relevan dengan analisis kita.

Jika tujuan kita adalah sesuatu selain memverifikasi desain di tingkat gerbang, ontologinya akan berbeda. Misalnya, jika kita tertarik untuk men-debug sirkuit yang rusak, maka mungkin ide yang bagus untuk menyertakan kabel dalam ontologi, karena kabel yang rusak dapat merusak sinyal yang mengalir melaluinya. Untuk mengatasi kesalahan pengaturan waktu, kita perlu menyertakan penundaan gerbang. Jika kita tertarik untuk merancang produk yang akan menguntungkan, maka biaya sirkuit dan kecepatannya relatif terhadap produk lain di pasaran akan menjadi penting.

Tentukan Kosakata

Kita sekarang tahu bahwa kita ingin berbicara tentang sirkuit, terminal, sinyal, dan gerbang. Langkah berikutnya adalah memilih fungsi, predikat, dan konstanta untuk merepresentasikannya. Pertama, kita perlu mampu membedakan gerbang satu sama lain dan dari objek lainnya. Setiap gerbang direpresentasikan sebagai objek yang diberi nama oleh sebuah konstanta, yang tentangnya kita tegaskan bahwa gerbang tersebut adalah gerbang

dengan, katakanlah, $Gate(X_1)$. Perilaku setiap gerbang ditentukan oleh tipenya: salah satu konstanta AND, OR, XOR, atau NOT. Karena gerbang memiliki tepat satu tipe, maka fungsi berikut sesuai: $Type(X_1) = XOR$. Rangkaian, seperti gerbang, diidentifikasi oleh predikat: $Circuit(C_1)$.

Selanjutnya kita pertimbangkan terminal, yang diidentifikasi oleh predikat $Terminal(x)$. Gerbang atau rangkaian dapat memiliki satu atau lebih terminal input dan satu atau lebih terminal output. Kita menggunakan fungsi $In(1, X_1)$ untuk menunjukkan terminal input pertama untuk gerbang X_1. Fungsi serupa Out digunakan untuk terminal output. Fungsi $Arity(c, i, j)$ menyatakan bahwa rangkaian c memiliki i terminal input dan j terminal output. Konektivitas antar gerbang dapat direpresentasikan oleh predikat, $Connected$, yang mengambil dua terminal sebagai argumen, seperti dalam $Connected(Out(1, X_1), In(1, X_2))$.

Terakhir, kita perlu mengetahui apakah sinyal aktif atau tidak. Salah satu kemungkinan adalah menggunakan predikat unary, $On(t)$, yang benar ketika sinyal di terminal aktif. Namun, hal ini membuatnya agak sulit untuk mengajukan pertanyaan seperti "Berapa saja nilai sinyal yang mungkin di terminal keluaran rangkaian C_1?" Oleh karena itu, kami memperkenalkan dua nilai sinyal, 1 dan 0, dan fungsi $Signal\ (t)$ sebagai objek yang menunjukkan nilai sinyal untuk terminal t.

Encode Pengetahuan Umum Tentang Domain

Salah satu tanda bahwa kita memiliki ontologi yang baik adalah bahwa kita hanya memerlukan beberapa aturan umum, yang dapat dinyatakan dengan jelas dan ringkas. Ini semua adalah aksioma yang akan kita perlukan:

1. Jika dua terminal terhubung, maka keduanya memiliki sinyal yang sama:
   $\forall t_1, t_2\ Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2)$.

2. Sinyal pada setiap terminal adalah 1 atau 0:
   $\forall t\ Terminal(t) \Rightarrow Signal(t) = 1 \vee Signal(t) = 0$

3. Terhubung bersifat komutatif:
   $\forall t_1, t_2\ Connected(t_1, t_2) \Leftrightarrow Connected(t_2, t_1)$

4. Ada empat jenis gerbang:
   $\forall\ g\ Gate(g) \wedge k = Type(g) \Rightarrow k = AND \vee k = OR \vee k = XOR \vee k = NOT$.

5. Output gerbang AND adalah 0 jika dan hanya jika salah satu inputnya adalah 0:
   $\forall\ g\ Gate(g) \wedge Type(g) = AND \Rightarrow Signal\big(Out(1, g)\big) = 0 \Leftrightarrow \exists n\ Signal\big(In(n, g)\big) = 0$

6. Output gerbang OR adalah 1 jika dan hanya jika salah satu inputnya adalah 1:
   $\forall\ g\ Gate(g) \wedge Type(g) = OR \Rightarrow Signal\big(Out(1, g)\big) = 1 \Leftrightarrow \exists n\ Signal\big(In(n, g)\big) = 1$

7. Output gerbang XOR adalah 1 jika dan hanya jika inputnya berbeda:
   $\forall\, g\ Gate(g) \wedge Type(g) = XOR \Rightarrow$
   $Signal\big(Out(1,g)\big) = 1 \Leftrightarrow\ Signal\big(In(1,g)\big) \neq Signal\big(In(2,g)\big)$

8. Output gerbang NOT berbeda dari inputnya:
   $\forall\, g\ Gate(g) \wedge Type(g) = NOT \Rightarrow$
   $Signal\big(Out(1,g)\big) \neq Signal\big(In(1,g)\big)$

9. Gerbang-gerbang (kecuali NOT) memiliki dua input dan satu output.
   $\forall\, g\ Gate(g) \wedge Type(g) = NOT \Rightarrow Arity(g,1,1)$.
   $\forall\, g\ Gate(g) \wedge k = Type(g) \wedge (k = AND \vee k = OR \vee k = XOR \Rightarrow$
   $Arity(g,2,1)$

10. Suatu rangkaian memiliki terminal, hingga aritas masukan dan keluarannya, dan tidak ada yang melampaui aritasnya:

    $\forall\, c,i,j\ Circuit(c) \wedge Arity(c,i,j) \Rightarrow$
    $\forall n\ (n \leq i \Rightarrow Terminal\big(In(c,n)\big)) \wedge (n > i \Rightarrow In(c,n) = Nothing) \wedge$
    $\forall\, n\ (n \leq j \Rightarrow Terminal\big(Out(c,n)\big)) \wedge (n > j \Rightarrow Out(c,n) = Nothing)$

11. Gerbang, terminal, sinyal, tipe gerbang, dan $Nothing$ semuanya berbeda.
    $\forall\, g,t\ (Gate(g) \wedge Terminal(t) \Rightarrow$
    $g \neq t \neq 1 \neq 0 \neq OR \neq AND \neq XOR \neq NOT \neq Nothing.$

12. Gerbang adalah rangkaian.
    $\forall\, g\ (Gate(g) \Rightarrow Circuit(g)$

Kodekan Contoh Masalah Spesifik

Rangkaian yang ditunjukkan pada Gambar 8.6 dikodekan sebagai rangkaian C_1 dengan deskripsi berikut. Pertama, kita kategorikan rangkaian dan gerbang komponennya:

$Circuit(C_1) \wedge Arity(C_1,3,2)$
$Gate(X_1) \wedge Type(X_1) = XOR$
$Gate(X_2) \wedge Type(X_2) = XOR$
$Gate(A_1) \wedge Type(A_1) = AND$
$Gate(A_2) \wedge Type(A_2) = AND$
$Gate(O_1) \wedge Type(O_1) = OR.$

Kemudian, kami menunjukkan hubungan di antara keduanya:

$Connected(Out(1,X_1),In(1,X_2))$ $Connected(In(1,C_1),In(1,X_1))$
$Connected(Out(1,X_1),In(2,A_2))$ $Connected(In(1,C_1),In(1,A_1))$
$Connected(Out(1,A_2),In(1,O_1))$ $Connected(In(2,C_1),In(2,X_1))$
$Connected(Out(1,A_1),In(2,O_1))$ $Connected(In(2,C_1),In(2,A_1))$
$Connected(Out(1,X_2),Out(1,C_1))$ $Connected(In(3,C_1),In(2,X_2))$
$Connected(Out(1,O_1),Out(2,C_1))$ $Connected(In(3,C_1),In(1,A_2)).$

Ajukan Pertanyaan Pada Prosedur Inferensi

Kombinasi masukan apa yang akan menyebabkan keluaran pertama C_1 (bit penjumlahan) menjadi 0 dan keluaran kedua C_1 (bit pembawa) menjadi 1?

$$\exists i_1, i_2, i_3\ Signal\big(In(1,C_1)\big) = i_1 \wedge Signal\big(In(2,C_1)\big) = i_2 \wedge Signal\big(In(2,C_1)\big) = i_3 \wedge Signal\big(Out(1,C_1)\big) = 0 \wedge Signal\big(Out(2,C_1)\big) = 1.$$

Jawabannya adalah substitusi untuk variabel i_1, i_2, dan i_3 sehingga kalimat yang dihasilkan disiratkan oleh basis pengetahuan. ASKVARS akan memberi kita tiga substitusi seperti itu:

$$\{i_1/1, i_2/1, i_3/0\}\ \{i_1/1, i_2/0, i_3/1\}\ \{i_1/0, i_2/1, i_3/1\}\ .$$

Apa saja kemungkinan set nilai semua terminal untuk rangkaian penjumlah?

$$\exists i_1, i_2, i_3, o_1, o_2\ \ Signal\big(In(1,C_1)\big) = i_1 \wedge Signal\big(In(2,C_1)\big) = i_2 \wedge Signal\big(In(3,C_1)\big) = i_3 \wedge Signal\big(Out(1,C_1)\big) = o_1 \wedge Signal\big(Out(2,C_1)\big) = o_2.$$

Kueri terakhir ini akan mengembalikan tabel input-output lengkap untuk perangkat, yang dapat digunakan untuk memeriksa apakah perangkat benar-benar menambahkan inputnya dengan benar. Ini adalah contoh sederhana verifikasi sirkuit. Kita juga dapat menggunakan definisi sirkuit untuk membangun sistem digital yang lebih besar, yang prosedur verifikasinya dapat dilakukan dengan cara yang sama. (Lihat Latihan 8.26.) Banyak domain yang cocok untuk pengembangan basis pengetahuan terstruktur yang sama, di mana konsep yang lebih kompleks didefinisikan di atas konsep yang lebih sederhana.

Debug Basis Pengetahuan

Kita dapat mengganggu basis pengetahuan dengan berbagai cara untuk melihat jenis perilaku salah apa yang muncul. Misalnya, anggaplah kita gagal membaca Bagian 8.2.8 dan karena itu lupa menyatakan bahwa $1 \neq 0$. Tiba-tiba, sistem tidak akan dapat membuktikan output apa pun untuk sirkuit, kecuali untuk kasus input 000 dan 110. Kita dapat menentukan masalahnya dengan menanyakan output dari setiap gerbang. Misalnya, kita dapat menanyakan

$$\exists i_1, i_2, o\ \ Signal\big(In(1,C_1)\big) = i_1 \wedge Signal\big(In(2,C_1)\big) = i_2 \wedge Signal\big(Out(1,X_1)\big),$$

yang menunjukkan bahwa tidak ada keluaran yang diketahui pada X_1 untuk kasus masukan 10 dan 01. Kemudian, kita melihat aksioma untuk gerbang XOR, sebagaimana diterapkan pada X_1:

$$Signal\big(Out(1, X_1)\big) = 1 \Leftrightarrow Signal\big(In(1, X_1)\big) \neq Signal\big(In(2, X_1)\big).$$

Jika inputnya diketahui, katakanlah, 1 dan 0, maka ini disederhanakan menjadi

$$Signal\big(Out(1, X_1)\big) = 1 \Leftrightarrow 1 \neq 0$$

Sekarang permasalahannya menjadi jelas: sistem tidak dapat menyimpulkan bahwa $Signal\big(Out(1, X_1)\big) = 1$, jadi kita perlu memberitahunya bahwa $1 \neq 0$.

8.5 RINGKASAN

Bab ini memperkenalkan logika tingkat pertama, bahasa representasi yang jauh lebih kuat daripada logika proposisional. Poin-poin pentingnya adalah sebagai berikut:

- Bahasa representasi pengetahuan harus bersifat deklaratif, komposisional, ekspresif, independen terhadap konteks, dan tidak ambigu.
- Logika berbeda dalam komitmen ontologis dan komitmen epistemologisnya. Sementara logika proposisional hanya berkomitmen pada keberadaan fakta, logika tingkat pertama berkomitmen pada keberadaan objek dan hubungan dan dengan demikian memperoleh kekuatan ekspresif.
- Sintaksis logika tingkat pertama dibangun di atas logika proposisional. Ia menambahkan istilah untuk mewakili objek, dan memiliki kuantifier universal dan eksistensial untuk membangun pernyataan tentang semua atau beberapa nilai yang mungkin dari variabel yang dikuantifikasi.
- Dunia atau model yang mungkin untuk logika orde pertama mencakup sekumpulan objek dan interpretasi yang memetakan simbol konstan ke objek, simbol predikat ke relasi di antara objek, dan simbol fungsi ke fungsi pada objek.
- Kalimat atomik benar hanya jika relasi yang dinamai oleh predikat berlaku di antara objek yang dinamai oleh istilah. Interpretasi yang diperluas, yang memetakan variabel kuantifier ke objek dalam model, mendefinisikan kebenaran kalimat yang dikuantifikasi.
- Mengembangkan basis pengetahuan dalam logika orde pertama memerlukan proses yang cermat dalam menganalisis domain, memilih kosakata, dan mengodekan aksioma yang diperlukan untuk mendukung inferensi yang diinginkan.

BAB 9
INFERENSI DALAM LOGIKA ORDE PERTAMA

Bab 7 menunjukkan bagaimana inferensi yang baik dan lengkap dapat dicapai untuk logika proposisional. Dalam bab ini, kami memperluas hasil tersebut untuk memperoleh algoritme yang dapat menjawab pertanyaan yang dapat dijawab yang dinyatakan dalam logika orde pertama. Bagian 9.1 memperkenalkan aturan inferensi untuk kuantifier dan menunjukkan cara mereduksi inferensi orde pertama menjadi inferensi proposisional, meskipun dengan biaya yang sangat besar. Bagian 9.2 menjelaskan gagasan **penyatuan**, menunjukkan bagaimana hal itu dapat digunakan untuk membangun aturan inferensi yang bekerja secara langsung dengan kalimat orde pertama. Kami kemudian membahas tiga keluarga utama algoritme inferensi orde pertama. **Rantai maju** dan aplikasinya pada **basis data deduktif** dan **sistem produksi** dibahas dalam Bagian 9.3; **rantai mundur** dan sistem **pemrograman logika** dikembangkan dalam Bagian 9.4. Rantai maju dan mundur bisa sangat efisien, tetapi hanya berlaku untuk basis pengetahuan yang dapat dinyatakan sebagai kumpulan klausa Horn. Kalimat orde pertama umum memerlukan **pembuktian teorema** berbasis resolusi, yang dijelaskan dalam Bagian 9.5.

9.1 INFERENSI PROPOSISIONAL VS. INFERENSI ORDE PERTAMA

Bagian ini dan bagian berikutnya memperkenalkan ide-ide yang mendasari sistem inferensi logis modern. Kita mulai dengan beberapa aturan inferensi sederhana yang dapat diterapkan pada kalimat dengan kuantifier untuk memperoleh kalimat tanpa kuantifier. Aturan-aturan ini secara alami mengarah pada gagasan bahwa inferensi *orde pertama* dapat dilakukan dengan mengubah basis pengetahuan menjadi logika *proposisional* dan menggunakan inferensi *proposisional*, yang sudah kita ketahui cara melakukannya. Bagian berikutnya menunjukkan jalan pintas yang jelas, yang mengarah pada metode inferensi yang memanipulasi kalimat orde pertama secara langsung.

Aturan inferensi untuk kuantifier

Mari kita mulai dengan kuantifier universal. Misalkan basis pengetahuan kita berisi aksioma cerita rakyat standar yang menyatakan bahwa semua raja yang tamak itu jahat:

$$\forall\, x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$

Maka tampaknya cukup diperbolehkan untuk menyimpulkan salah satu kalimat berikut ini:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$$
$$\vdots$$

Aturan **Universal Instantiation** (disingkat **UI**) menyatakan bahwa kita dapat menyimpulkan kalimat apa pun yang diperoleh dengan mensubstitusikan **istilah dasar** (istilah tanpa variabel) untuk variabel tersebut. Untuk menuliskan aturan inferensi secara formal, kita menggunakan gagasan **substitusi** yang diperkenalkan di Bagian 8.3. Biarkan SUBST(θ, α) menunjukkan hasil penerapan substitusi θ ke kalimat α. Maka aturan tersebut ditulis

$$\frac{\forall\, v\ \alpha}{SUBST(\{v/g\}, \alpha)}$$

untuk setiap variabel v dan suku dasar g. Misalnya, tiga kalimat yang diberikan sebelumnya diperoleh dengan substitusi $\{x/John\ \}, \{x/Richard\ \}, dan\ \{x/Father\ (John)\}$.

Dalam aturan **Instansiasi Eksistensial**, variabel digantikan oleh satu *simbol konstanta baru*. Pernyataan formalnya adalah sebagai berikut: untuk setiap kalimat α, variabel v, dan simbol konstanta k yang tidak muncul di tempat lain dalam basis pengetahuan,

$$\frac{\exists\, v\ \alpha}{\mathrm{SUBST}(\{v/k\}, \alpha)}$$

Misalnya dari kalimat

$$\exists x\ Crown(x)\ \wedge\ OnHead(x, John)$$

kita bisa menyimpulkan kalimatnya

asalkan C_1 tidak muncul di tempat lain dalam basis pengetahuan. Pada dasarnya, kalimat eksistensial menyatakan ada

$$Crown(C_1) \wedge OnHead(C_1, John)$$

beberapa objek yang memenuhi suatu kondisi, dan menerapkan aturan instansiasi eksistensial hanya memberikan nama pada objek tersebut. Tentu saja, nama itu tidak boleh sudah menjadi milik objek lain. Matematika memberikan contoh yang bagus: misalkan kita menemukan bahwa ada angka yang sedikit lebih besar dari 2,71828 dan memenuhi persamaan $d(x^y)/dy = x^y$ untuk x. Kita dapat memberi angka ini nama, seperti e, tetapi akan menjadi kesalahan untuk memberinya nama objek yang sudah ada, seperti π. Dalam logika, nama baru tersebut disebut **konstanta Skolem**. Instansiasi Eksistensial adalah kasus khusus dari proses yang lebih umum yang disebut **skolemisasi**, yang kita bahas di Bagian 9.5.

Sementara Instansiasi Universal dapat diterapkan berkali-kali untuk menghasilkan banyak konsekuensi yang berbeda, Instansiasi Eksistensial dapat diterapkan sekali, dan kemudian kalimat yang dikuantifikasi secara eksistensial dapat dibuang. Misalnya, kita tidak lagi memerlukan $\exists\, x\ Kill(x, Victim)$ setelah kita menambahkan kalimat $Kill(Murderer, Victim)$. Secara tegas, basis pengetahuan baru tidak secara logis setara

dengan yang lama, tetapi dapat ditunjukkan **secara inferensial** setara dalam arti bahwa basis pengetahuan asli dapat dipenuhi secara tepat ketika basis pengetahuan asli dapat dipenuhi.

Reduction to propositional inference

kita memiliki aturan untuk menyimpulkan kalimat yang tidak terkuantifikasi dari kalimat yang terkuantifikasi, menjadi mungkin untuk mereduksi inferensi tingkat pertama menjadi inferensi proposisional. Di bagian ini kita memberikan ide-ide utama; rinciannya diberikan di Bagian 9.5. Ide pertama adalah, seperti halnya kalimat yang terkuantifikasi secara eksistensial dapat digantikan oleh satu perwujudan, kalimat yang terkuantifikasi secara universal dapat digantikan oleh himpunan *semua perwujudan* yang mungkin. Misalnya, anggaplah basis pengetahuan kita hanya berisi kalimat-kalimat

$$\begin{aligned} &\forall\, x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x) \\ &King(John) \\ &Greedy(John) \\ &Brother(Richard, John)\ . \end{aligned} \tag{9.1}$$

Kemudian kami menerapkan UI pada kalimat pertama menggunakan semua kemungkinan substitusi istilah dasar dari kosakata basis pengetahuan—dalam kasus ini, $\{x/John\}$ dan $\{x/Richard\}$. Kami memperoleh

$$\begin{aligned} &King(John) \wedge Greedy(John) \Rightarrow Evil(John) \\ &King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard) \end{aligned}$$

dan kita membuang kalimat yang dikuantifikasi secara universal. Sekarang, basis pengetahuan pada dasarnya bersifat proposisional jika kita melihat kalimat atom dasar—$King\ (John)$, $Greedy\ (John)$, dan seterusnya—sebagai simbol proposisi. Oleh karena itu, kita dapat menerapkan salah satu algoritma proposisional lengkap di Bab 7 untuk memperoleh kesimpulan seperti $Evil(John)$.

Teknik **proposisionalisasi** ini dapat dibuat sepenuhnya umum, seperti yang kita tunjukkan di Bagian 9.5; yaitu, setiap basis pengetahuan dan kueri tingkat pertama dapat diproposisionalkan sedemikian rupa sehingga konsekuensinya dipertahankan. Dengan demikian, kita memiliki prosedur keputusan lengkap untuk konsekuensi ... atau mungkin tidak. Ada masalah: ketika basis pengetahuan menyertakan simbol fungsi, himpunan substitusi istilah dasar yang mungkin tidak terbatas! Misalnya, jika basis pengetahuan menyebutkan simbol Father, maka istilah bersarang yang tak terbatas jumlahnya seperti $Father\ (Father\ (Father\ (John)))$ dapat dibangun. Algoritma proposisional kita akan mengalami kesulitan dengan himpunan kalimat yang sangat besar.

Untungnya, ada teorema terkenal dari Jacques Herbrand (1930) yang menyatakan bahwa jika sebuah kalimat disyaratkan oleh basis pengetahuan orde pertama yang asli, maka ada pembuktian yang melibatkan hanya sebagian kecil dari basis pengetahuan yang diproposisikan. Karena setiap sebagian tersebut memiliki kedalaman maksimum penumpukan

di antara istilah-istilah dasarnya, kita dapat menemukan sebagian tersebut dengan terlebih dahulu menghasilkan semua perwujudan dengan simbol-simbol konstan ($Richard\ dan\ John$), kemudian semua istilah dengan kedalaman *1* ($Father\ (Richard)$ dan $Father\ (John)$), kemudian semua istilah dengan kedalaman 2, dan seterusnya, hingga kita dapat membangun pembuktian proposisional dari kalimat yang disyaratkan.

Kita telah membuat sketsa pendekatan untuk inferensi orde pertama melalui proposisionalisasi yang **lengkap**—yaitu, setiap kalimat yang disyaratkan dapat dibuktikan. Ini adalah pencapaian besar, mengingat ruang model yang mungkin tidak terbatas. Di sisi lain, kita tidak tahu sampai pembuktian dilakukan bahwa kalimat tersebut disyaratkan! Apa yang terjadi ketika kalimat tersebut tidak disyaratkan? Dapatkah kita mengetahuinya? Nah, untuk logika orde pertama, ternyata kita tidak bisa. Prosedur pembuktian kita bisa terus berlanjut, menghasilkan istilah-istilah yang semakin bersarang dalam, tetapi kita tidak akan tahu apakah itu terjebak dalam lingkaran yang tidak ada harapan atau apakah pembuktian akan segera muncul. Ini sangat mirip dengan masalah penghentian untuk mesin Turing. Alan Turing (1936) dan Alonzo Church (1936) keduanya membuktikan, dengan cara yang agak berbeda, keniscayaan keadaan ini. *Pertanyaan tentang konsekuensi untuk logika orde pertama bersifat* ***semidecidable****—yaitu, ada algoritme yang mengatakan ya untuk setiap kalimat yang diinisiasi, tetapi tidak ada algoritme yang juga mengatakan tidak untuk setiap kalimat yang tidak diinisiasi.*

9.2 UNIFIKASI DAN PENGANGKATAN

Bagian sebelumnya menjelaskan pemahaman tentang inferensi orde pertama yang ada hingga awal tahun 1960-an. Pembaca yang jeli (dan tentu saja ahli logika komputasional awal tahun 1960-an) akan menyadari bahwa pendekatan proposisionalisasi agak tidak efisien. Misalnya, mengingat kueri $Evil(x)$ dan basis pengetahuan dalam Persamaan (9.1), tampaknya tidak masuk akal untuk menghasilkan kalimat seperti $King(Richard) \Lambda\ Greedy\ (Richard\ ) \Rightarrow Evil(Richard\ )$. Memang, inferensi $Evil(John)$ dari kalimat:

$$\forall\ x\ King(x)\ \Lambda\ Greedy(x)\ \Rightarrow\ Evil(x)$$
$$King(John)$$
$$Greedy(John)$$

tampaknya sangat jelas bagi manusia. Sekarang kami tunjukkan cara membuatnya sangat jelas bagi komputer.

Aturan inferensi tingkat pertama

Inferensi bahwa John jahat—yaitu, bahwa $\{x/John\}$ memecahkan kueri $Jahat(x)$ —berfungsi seperti ini: untuk menggunakan aturan bahwa raja yang tamak itu jahat, temukan beberapa x sehingga x adalah raja dan x tamak, lalu simpulkan bahwa x ini jahat. Secara lebih umum, jika ada beberapa substitusi θ yang membuat setiap konjungsi premis implikasi identik dengan kalimat yang sudah ada dalam basis pengetahuan, maka kita dapat

menegaskan kesimpulan implikasi, setelah menerapkan θ. Dalam kasus ini, substitusi $\theta = \{x/John\}$ mencapai tujuan itu.

Kita sebenarnya dapat membuat langkah inferensi bekerja lebih keras. Misalkan bahwa alih-alih mengetahui $Greedy(John)$, kita tahu bahwa setiap orang tamak:

$$\forall\, y\ Greedy(y) \tag{9.2}$$

Maka kita masih ingin dapat menyimpulkan bahwa $Evil(John)$, karena kita tahu bahwa John adalah seorang raja (yang sudah ada) dan John serakah (karena semua orang serakah). Yang kita perlukan agar ini berhasil adalah menemukan substitusi untuk variabel-variabel dalam kalimat implikasi dan untuk variabel-variabel dalam kalimat-kalimat yang ada dalam basis pengetahuan. Dalam kasus ini, menerapkan substitusi $\{x/John\ , y/John\}$ pada premis implikasi $King(x)$ dan $Greedy(x)$ dan kalimat-kalimat basis pengetahuan $Raja\ (John)$ dan $Greedy(y)$ akan membuat keduanya identik. Dengan demikian, kita dapat menyimpulkan kesimpulan implikasinya.

Proses inferensi ini dapat ditangkap sebagai aturan inferensi tunggal yang kita sebut **Modus Ponens Umum:** Untuk kalimat atomik p_i, p_i' dan q, di mana terdapat substitusi θ sehingga $\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i')$, untuk semua i,

$$\frac{p_1', p_2', \ldots, p_n', (p_1 \wedge p_2 \wedge \ldots \wedge\ p_n \Rightarrow\ q)}{\text{SUBST}(\theta, q)}$$

Ada $n + 1$ premis untuk aturan ini: n kalimat atomik p_i' dan satu implikasi. Kesimpulannya adalah hasil penerapan substitusi θ pada konsekuen q. Untuk contoh kita:

$p_1'\ is\ King(John)$	$p_1\ is\ King(x)$
$p_2'\ is\ Greedy(y)$	$p_2\ is\ Greedy(x)$
$\theta\ is\ \{x/John, y/John\}$	$q\ is\ Evil(x)$
$SUBST(\theta, q)\ is\ Evil(John)$	

Mudah untuk menunjukkan bahwa Modus Ponens Umum adalah aturan inferensi yang baik. Pertama, kita mengamati bahwa, untuk setiap kalimat p (yang variabelnya diasumsikan dikuantifikasi secara universal) dan untuk setiap substitusi θ,

$$p \vDash \text{SUBST}(\theta, p)$$

berlaku melalui Universal Instantiation. Hal ini berlaku khususnya untuk θ yang memenuhi kondisi aturan Generalized Modus Ponens. Jadi, dari p_1' ... p_n' kita dapat menyimpulkan

$$\text{SUBST}(\theta, p_1') \wedge \ldots \wedge \text{SUBST}(\theta, p_n')$$

dan dari implikasi $p_1 \wedge \ldots \wedge p_n \Rightarrow\ q$ kita dapat menyimpulkan

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge\ \text{SUBST}(\theta, p_n)\ \Rightarrow\ \text{SUBST}(\theta, q)$$

Sekarang, θ dalam Modus Ponens Umum didefinisikan sedemikian rupa sehingga $\text{SUBST}(\theta, p_1') = \text{SUBST}(\theta, p_1)$, untuk semua i; oleh karena itu kalimat pertama dari kedua kalimat ini sama persis dengan premis kalimat kedua. Oleh karena itu, $\text{SUBST}(\theta, q)$ diikuti oleh Modus Ponens.

Modus Ponens Umum adalah versi **terangkat** dari Modus Ponens—ia menaikkan Modus Ponens dari logika proposisional dasar (bebas variabel) ke logika orde pertama. Kita akan melihat di sisa bab ini bahwa kita dapat mengembangkan versi terangkat dari algoritma forward chaining, backward chaining, dan resolusi yang diperkenalkan di Bab 7. Keuntungan utama dari aturan inferensi terangkat atas proposisionalisasi adalah bahwa aturan tersebut hanya membuat substitusi yang diperlukan untuk memungkinkan inferensi tertentu berlanjut.

Penyatuan

Aturan inferensi terangkat memerlukan pencarian substitusi yang membuat ekspresi logika yang berbeda tampak identik. Proses ini disebut **penyatuan** dan merupakan komponen utama dari semua algoritma inferensi orde pertama. Algoritma UNIFY mengambil dua kalimat dan mengembalikan **pemersatu** untuk kedua kalimat tersebut jika ada:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) .$$

Mari kita lihat beberapa contoh tentang bagaimana UNIFY seharusnya berperilaku. Misalkan kita memiliki kueri $AskVars(Knows(John, x))$: siapa yang John kenal? Jawaban untuk kueri ini dapat ditemukan dengan menemukan semua kalimat dalam basis pengetahuan yang menyatu dengan $Knows(John, x)$. Berikut adalah hasil penyatuan dengan empat kalimat berbeda yang mungkin ada dalam basis pengetahuan:

$$\text{UNIFY}(Knows(John, x), Knows(John, Jane)) = \{x/Jane\}$$
$$\text{UNIFY}(Knows(John, x), Knows(y, Bill)) = \{x/Bill, y/John\}$$
$$\text{UNIFY}(Knows(John, x), Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}$$
$$\text{UNIFY}(Knows(John, x), Knows(x, Elizabeth)) = fail$$

Penyatuan terakhir gagal karena x tidak dapat mengambil nilai $John$ dan $Elizabeth$ pada saat yang sama. Sekarang, ingat bahwa $Knows(x, Elizabeth)$ berarti "Semua orang tahu Elizabeth," jadi kita seharusnya dapat menyimpulkan bahwa John tahu $Elizabeth$. Masalah muncul hanya karena kedua kalimat tersebut kebetulan menggunakan nama variabel yang sama, x. Masalah tersebut dapat dihindari dengan **menstandardisasi** salah satu dari dua kalimat yang disatukan, yang berarti mengganti nama variabelnya untuk menghindari bentrokan nama. Misalnya, kita dapat mengganti nama x dalam $Knows(x, Elizabeth)$ menjadi X_{17} (nama variabel baru) tanpa mengubah artinya. Sekarang penyatuan akan berfungsi:

$$UNIFY(Knows(John, x), Knows(x_{17}, Elizabeth)) = \{x/Elizabeth, x_{17}/John\}$$

Ada satu komplikasi lagi: kita mengatakan bahwa UNIFY harus mengembalikan substitusi yang membuat kedua argumen terlihat sama. Namun, bisa saja ada lebih dari satu pemersatu seperti itu. Misalnya, $\text{UNIFY}(Knows\ (John, x), Knows\ (y, z))$ dapat

mengembalikan $\{y/John\ , x/z\}$ atau $\{y/John, x/John\ , z/John\ \}$. Pemersatu pertama menghasilkan $Knows\ (John, z)$ sebagai hasil penyatuan, sedangkan yang kedua menghasilkan $Knows\ (John, John)$. Hasil kedua dapat diperoleh dari yang pertama dengan substitusi tambahan {z/John }; kita katakan bahwa pemersatu pertama lebih umum daripada yang kedua, karena pemersatu tersebut memberikan lebih sedikit batasan pada nilai variabel. Ternyata, untuk setiap pasangan ekspresi yang dapat disatukan, ada satu **pemersatu paling umum** (atau MGU) yang unik hingga penggantian nama dan substitusi variabel. (Misalnya, $\{x/John\}$ dan $\{y/John\}$ dianggap ekuivalen, seperti halnya $\{x/John, y/John\ \}$ dan $\{x/John\ , y/x\}$.) Dalam kasus ini adalah $\{y/John, x/z\}$.

```
function UNIFY(x , y, θ) returns a substitution to make x and y identical
    inputs: x , a variable, constant, list, or compound expression
            y, a variable, constant, list, or compound expression
            θ, the substitution built up so far (optional, defaults to empty)

    if θ = failure then return failure
    else if x = y then return θ
    else if VARIABLE?(x ) then return UNIFY-VAR(x , y, θ)
    else if VARIABLE?(y) then return UNIFY-VAR(y, x , θ)
    else if COMPOUND?(x ) and COMPOUND?(y) then
        return UNIFY(x .ARGS, y.ARGS, UNIFY(x .OP, y.OP, θ))
    else if LIST?(x ) and LIST?(y) then
        return UNIFY(x .REST, y.REST, UNIFY(x .FIRST, y.FIRST, θ))
    else return failure

function UNIFY-VAR(var , x , θ) returns a substitution

    if {var/val} ∈ θ then return UNIFY(val, x , θ)
    else if {x/val} ∈ θ then return UNIFY(var , val, θ)
    else if OCCUR-CHECK?(var , x ) then return failure
    else return add {var /x } to θ
```

Gambar 9.1 Algoritma penyatuan. Algoritma ini bekerja dengan membandingkan struktur masukan, elemen demi elemen. Substitusi θ yang merupakan argumen untuk UNIFY dibangun di sepanjang jalan dan digunakan untuk memastikan bahwa perbandingan selanjutnya konsisten dengan pengikatan yang ditetapkan sebelumnya. Dalam ekspresi gabungan seperti $F(A, B)$, bidang OP memilih simbol fungsi F dan bidang ARGS memilih daftar argumen (A, B).

Algoritme untuk menghitung pemersatu paling umum ditunjukkan pada Gambar 9.1. Prosesnya sederhana: jelajahi dua ekspresi secara rekursif secara bersamaan "berdampingan," membangun pemersatu di sepanjang jalan, tetapi gagal jika dua titik yang sesuai dalam struktur tidak cocok. Ada satu langkah mahal: saat mencocokkan variabel dengan istilah kompleks, seseorang harus memeriksa apakah variabel itu sendiri muncul di dalam istilah; jika ya, pencocokan gagal karena tidak ada pemersatu yang konsisten yang dapat dibangun.

Misalnya, $S(x)$ tidak dapat menyatukan dengan $S(S(x))$. Apa yang disebut **pemeriksaan terjadi** ini membuat kompleksitas seluruh algoritme kuadratik dalam ukuran ekspresi yang disatukan. Beberapa sistem, termasuk semua sistem pemrograman logika, cukup menghilangkan pemeriksaan terjadi dan terkadang membuat inferensi yang tidak masuk akal sebagai hasilnya; sistem lain menggunakan algoritma yang lebih kompleks dengan kompleksitas waktu linier.

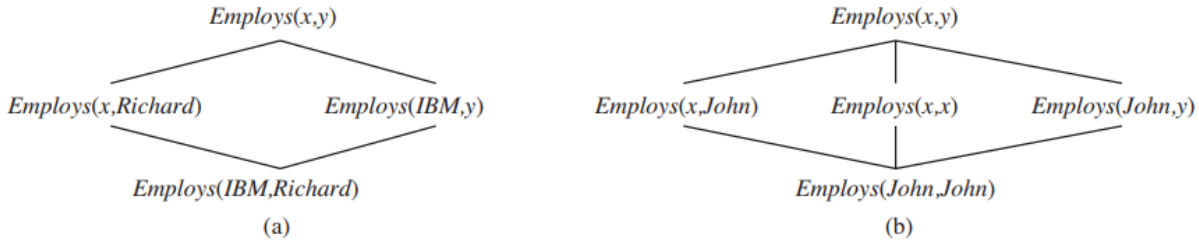
Penyimpanan dan pengambilan

Fungsi TELL dan ASK yang digunakan untuk menginformasikan dan menginterogasi basis pengetahuan adalah fungsi STORE dan FETCH yang lebih primitif. $\mathrm{STORE}(s)$ menyimpan kalimat s ke dalam basis pengetahuan dan $\mathrm{FETCH}(q)$ mengembalikan semua pemersatu sehingga kueri q menyatu dengan beberapa kalimat dalam basis pengetahuan. Masalah yang kami gunakan untuk mengilustrasikan penyatuan—menemukan semua fakta yang menyatu dengan $Knows(John, x)$—adalah contoh FETCHing.

Cara paling sederhana untuk mengimplementasikan STORE dan FETCH adalah dengan menyimpan semua fakta dalam satu daftar panjang dan menyatukan setiap kueri terhadap setiap elemen daftar. Proses seperti itu tidak efisien, tetapi berhasil, dan itu semua yang Anda perlukan untuk memahami sisa bab ini. Sisa bagian ini menguraikan cara-cara untuk membuat pengambilan lebih efisien; bagian ini dapat dilewati pada bacaan pertama. Kita dapat membuat FETCH lebih efisien dengan memastikan bahwa penyatuan hanya dicoba dengan kalimat-kalimat yang memiliki beberapa peluang untuk menyatukan. Misalnya, tidak ada gunanya mencoba menyatukan $Knows(John, x)$ dengan $Brother\ (Richard\ , John)$. Kita dapat menghindari penyatuan tersebut dengan **mengindeks** fakta-fakta dalam basis pengetahuan. Skema sederhana yang disebut **pengindeksan predikat** menempatkan semua fakta $Knows$ dalam satu keranjang dan semua fakta $Brother$ di keranjang lain. Keranjang-keranjang tersebut dapat disimpan dalam tabel hash untuk akses yang efisien.

Pengindeksan predikat berguna ketika ada banyak simbol predikat tetapi hanya beberapa klausa untuk setiap simbol. Namun, terkadang predikat memiliki banyak klausa. Misalnya, anggaplah bahwa otoritas pajak ingin melacak siapa yang mempekerjakan siapa, menggunakan predikat $Employs\ (x, y)$. Ini akan menjadi keranjang yang sangat besar dengan mungkin jutaan pemberi kerja dan puluhan juta karyawan. Menjawab kueri seperti $Employs\ (x, Richard\ )$ dengan pengindeksan predikat akan memerlukan pemindaian seluruh keranjang.

Untuk kueri khusus ini, akan membantu jika fakta diindeks baik oleh predikat maupun argumen kedua, mungkin menggunakan kunci tabel hash gabungan. Kemudian kita dapat dengan mudah membuat kunci dari kueri dan mengambil fakta-fakta yang menyatu dengan kueri tersebut. Untuk kueri lain, seperti $Employs\ (IBM, y)$, kita perlu mengindeks fakta dengan menggabungkan predikat dengan argumen pertama. Oleh karena itu, fakta dapat disimpan di bawah beberapa kunci indeks, membuatnya langsung dapat diakses oleh berbagai kueri yang mungkin menyatu dengannya.



Gambar 9.2 (a) Kisi subsumsi yang simpul terendahnya adalah $Employs\ (IBM, Richard)$. (b) Kisi subsumsi untuk kalimat $Employs\ (John, John)$.

Mengingat kalimat yang akan disimpan, dimungkinkan untuk membuat indeks untuk *semua kemungkinan* kueri yang menyatu dengannya. Untuk fakta $Employs\ (IBM, Richard)$, kuerinya adalah

$Employs(IBM, Richard)$	Apakah IBM mempekerjakan Richard?
$Employs(x, Richard)$	Siapa yang mempekerjakan Richard? Siapa
$Employs(IBM, y)$	yang dipekerjakan IBM?
$Employs(x, y)$	Siapa yang mempekerjakan siapa?

Kueri-kueri ini membentuk **kisi subsumsi**, seperti yang ditunjukkan pada Gambar 9.2(a). Kisi tersebut memiliki beberapa properti yang menarik. Misalnya, anak dari setiap simpul dalam kisi diperoleh dari induknya melalui substitusi tunggal; dan turunan umum "tertinggi" dari dua simpul mana pun adalah hasil penerapan pemersatu paling umum mereka. Bagian kisi di atas fakta dasar apa pun dapat dibangun secara sistematis. Sebuah kalimat dengan konstanta berulang memiliki kisi yang sedikit berbeda, seperti yang ditunjukkan pada Gambar 9.2(b). Simbol fungsi dan variabel dalam kalimat yang akan disimpan memperkenalkan struktur kisi yang lebih menarik lagi.

Skema yang telah kami jelaskan bekerja dengan sangat baik setiap kali kisi berisi sejumlah kecil simpul. Namun, untuk predikat dengan n argumen, kisi berisi $O(2^n)$ simpul. Jika simbol fungsi diizinkan, jumlah simpul juga eksponensial dalam ukuran istilah dalam kalimat yang akan disimpan. Hal ini dapat menyebabkan banyaknya indeks. Pada titik tertentu, manfaat pengindeksan akan lebih kecil daripada biaya penyimpanan dan pemeliharaan semua indeks. Kita dapat menanggapinya dengan mengadopsi kebijakan tetap, seperti memelihara indeks hanya pada kunci yang terdiri dari predikat ditambah setiap argumen, atau dengan menggunakan kebijakan adaptif yang membuat indeks untuk memenuhi permintaan jenis kueri yang diajukan. Bagi sebagian besar sistem AI, jumlah fakta yang akan disimpan cukup kecil sehingga pengindeksan yang efisien dianggap sebagai masalah yang terpecahkan. Bagi basis data komersial, di mana fakta berjumlah miliaran, masalah tersebut telah menjadi subjek studi intensif dan pengembangan teknologi.

9.3 FORWARD CHAINING

Algoritma forward-chaining untuk klausa definit proposisional diberikan di Bagian 7.5. Idenya sederhana: mulai dengan kalimat atomik dalam basis pengetahuan dan terapkan

Modus Ponens ke arah maju, tambahkan kalimat atomik baru, hingga tidak ada lagi inferensi yang dapat dibuat. Di sini, kami menjelaskan bagaimana algoritma diterapkan pada klausa definit orde pertama. Klausa definit seperti $Situasi \Rightarrow Respons$ sangat berguna untuk sistem yang membuat inferensi sebagai respons terhadap informasi yang baru tiba. Banyak sistem dapat didefinisikan dengan cara ini, dan forward chaining dapat diimplementasikan dengan sangat efisien.

Klausa definit orde pertama

Klausul definit orde pertama sangat mirip dengan klausa definit proposisional klausa definit adalah disjungsi literal yang $tepat\ satu\ di\ antaranya\ positif$. Klausa definit bersifat atomik atau merupakan implikasi yang antesedennya adalah konjungsi literal positif dan yang konsekuensinya adalah literal positif tunggal. Berikut ini adalah klausa pasti tingkat pertama:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y)$$

Tidak seperti literal proposisional, literal orde pertama dapat menyertakan variabel, yang dalam hal ini variabel tersebut diasumsikan terkuantifikasi secara universal. (Biasanya, kami menghilangkan kuantifier universal saat menulis klausa pasti.) Tidak setiap basis pengetahuan dapat diubah menjadi sekumpulan klausa pasti karena pembatasan literal positif tunggal, tetapi banyak yang bisa. Pertimbangkan masalah berikut:

> Hukum menyatakan bahwa menjual senjata ke negara musuh merupakan kejahatan bagi warga Amerika. Negara Nono, musuh Amerika, memiliki beberapa rudal, dan semua rudalnya dijual kepadanya oleh Kolonel West, yang merupakan warga Amerika.

Kami akan membuktikan bahwa West adalah seorang penjahat. Pertama, kami akan merepresentasikan fakta-fakta ini sebagai klausa pasti orde pertama. Bagian berikutnya menunjukkan bagaimana algoritma forward-chaining memecahkan masalah tersebut.

"... menjual senjata ke negara musuh merupakan kejahatan bagi warga Amerika":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) . \quad (9.3)$$

"Nono ... punya beberapa rudal." Kalimat $\exists\, x\ Owns(Nono, x) \wedge Missile(x)$ diubah menjadi dua klausa pasti melalui Instansiasi Eksistensial, yang memperkenalkan konstanta baru M_1:

$$Owns(Nono, M_1) \quad (9.4)$$
$$Missile(M_1) \quad (9.5)$$

"Semua rudalnya dijual kepadanya oleh Kolonel West":

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) . \quad (9.6)$$

Kita juga perlu tahu bahwa rudal adalah senjata:

$$Missile(x) \Rightarrow Weapon(x) \quad (9.7)$$

dan kita harus tahu bahwa musuh Amerika dianggap sebagai "musuh":

$$Enemy(x, America) \Rightarrow Hostile(x) . \quad (9.8)$$

"Barat, siapa orang Amerika .. .":

$$American(West) . \quad (9.9)$$

"Negara Nono, musuh Amerika .. .":

$$Enemy(Nono, America) . \quad (9.10)$$

Basis pengetahuan ini tidak mengandung simbol fungsi dan karenanya merupakan contoh dari kelas basis pengetahuan **Datalog**. Datalog adalah bahasa yang dibatasi pada klausa pasti orde pertama tanpa simbol fungsi. Datalog mendapatkan namanya karena dapat mewakili jenis pernyataan yang biasanya dibuat dalam basis data relasional. Kita akan melihat bahwa tidak adanya simbol fungsi membuat inferensi jauh lebih mudah.

Algoritma forward-chaining sederhana

Algoritma forward-chaining pertama yang kita pertimbangkan adalah yang sederhana, ditunjukkan pada Gambar 9.3. Dimulai dari fakta yang diketahui, ia memicu semua aturan yang premisnya terpenuhi, menambahkan kesimpulannya ke fakta yang diketahui. Proses ini berulang hingga kueri terjawab (dengan asumsi bahwa hanya satu jawaban yang diperlukan) atau tidak ada fakta baru yang ditambahkan. Perhatikan bahwa fakta bukanlah "baru" jika itu hanya **penggantian nama** fakta yang diketahui. Satu kalimat adalah penggantian nama yang lain jika keduanya identik kecuali untuk nama variabel. Misalnya, $Likes(x, IceCream)$ dan $Likes(y, IceCream)$ merupakan penggantian nama satu sama lain karena keduanya hanya berbeda dalam pilihan x atau y; maknanya identik: semua orang suka es krim.

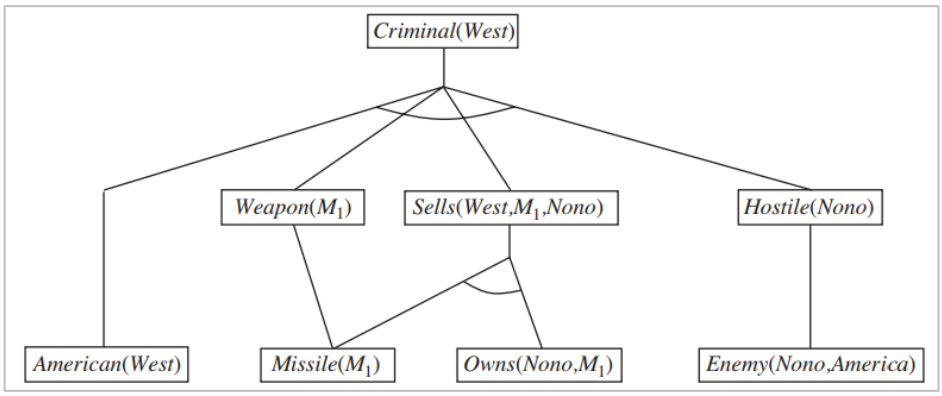
Kami menggunakan masalah kejahatan kami untuk mengilustrasikan cara kerja FOL-FC-ASK. Kalimat implikasinya adalah (9.3), (9.6), (9.7), dan (9.8). Diperlukan dua iterasi:

- Pada iterasi pertama, aturan (9.3) memiliki premis yang tidak terpenuhi.
  Aturan (9.6) terpenuhi dengan $\{x/M_1\}$, dan $Sells(West, M_1, Nono)$ ditambahkan.
  Aturan (9.7) terpenuhi dengan $\{x/M_1\}$, dan $Weapon(M_1)$ ditambahkan.
  Aturan (9.8) terpenuhi dengan $\{x/Nono\}$, dan $Hostile(Nono)$ ditambahkan.
- Pada iterasi kedua, aturan (9.3) terpenuhi dengan $\{x/Barat, y/M_1, z/Nono\}$, dan $Criminal(West)$ ditambahkan.

function FOL-FC-ASK(KB, α) **returns** a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    $new \leftarrow \{\ \}$
    for each $rule$ **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
      for each θ such that SUBST$(\theta, p_1 \wedge \ldots \wedge p_n) =$ SUBST$(\theta, p_1 \wedge \ldots \wedge p'_n)$.
          for some $p'_1 \ldots p'_n$ in KB
      $q' \leftarrow$ SUBST(θ, q)
      if q' does not unify with some sentence already in KB or new **then**
        add q' to new
        $\emptyset \leftarrow$ UNIFY(q, α)
        if $\emptyset$ is not $fail$ **then return** $\emptyset$
    add new to KB
return $false$

Gambar 9.3 Algoritma forward-chaining yang secara konseptual mudah dipahami, tetapi sangat tidak efisien. Pada setiap iterasi, algoritma ini menambahkan ke KB semua kalimat atomik yang dapat disimpulkan dalam satu langkah dari kalimat implikasi dan kalimat atomik yang sudah ada di KB. Fungsi STANDARDIZE − VARIABLES mengganti semua variabel dalam argumennya dengan variabel baru yang belum pernah digunakan sebelumnya.



Gambar 9.4 Pohon pembuktian yang dihasilkan oleh forward chaining pada contoh kejahatan. Fakta awal muncul di level bawah, fakta yang disimpulkan pada iterasi pertama di level tengah, dan fakta yang disimpulkan pada iterasi kedua di level atas.

Gambar 9.4 menunjukkan pohon pembuktian yang dihasilkan. Perhatikan bahwa tidak ada inferensi baru yang mungkin pada titik ini karena setiap kalimat yang dapat disimpulkan dengan forward chaining sudah termuat secara eksplisit dalam KB. Basis pengetahuan seperti itu disebut titik tetap dari proses inferensi. Titik tetap yang dicapai dengan forward chaining dengan klausa pasti orde pertama mirip dengan yang dicapai untuk forward chaining proposisional; perbedaan utamanya adalah bahwa titik tetap orde pertama dapat mencakup kalimat atomik yang dikuantifikasi secara universal.

FOL-FC-ASK mudah dianalisis. Pertama, ini masuk **akal**, karena setiap inferensi hanyalah penerapan Modus Ponens Umum, yang masuk akal. Kedua, ini **lengkap** untuk basis pengetahuan klausa pasti; yaitu, ini menjawab setiap kueri yang jawabannya disyaratkan oleh basis pengetahuan klausa pasti apa pun. Untuk basis pengetahuan Datalog, yang tidak mengandung simbol fungsi, pembuktian kelengkapan cukup mudah. Kita mulai dengan menghitung jumlah kemungkinan fakta yang dapat ditambahkan, yang menentukan jumlah maksimum iterasi. Misalkan k adalah **aritas** maksimum (jumlah argumen) dari setiap predikat, p adalah jumlah predikat, dan n adalah jumlah simbol konstan. Jelas, tidak boleh ada lebih dari pn^{k} fakta dasar yang berbeda, jadi setelah iterasi sebanyak ini algoritme harus mencapai titik tetap. Kemudian kita dapat membuat argumen yang sangat mirip dengan pembuktian kelengkapan untuk rantai maju proposisional. Rincian tentang cara melakukan transisi dari kelengkapan proposisional ke kelengkapan orde pertama diberikan untuk algoritme resolusi di Bagian 9.5.

Untuk klausa pasti umum dengan simbol fungsi, FOL-FC-ASK dapat menghasilkan fakta baru yang tak terhingga banyaknya, jadi kita perlu lebih berhati-hati. Untuk kasus di mana jawaban untuk kalimat kueri q diperlukan, kita harus mengacu pada teorema Herbrand untuk menetapkan bahwa algoritme akan menemukan bukti. (Lihat Bagian 9.5 untuk kasus resolusi.) Jika kueri tidak memiliki jawaban, algoritme dapat gagal untuk dihentikan dalam beberapa kasus. Misalnya, jika basis pengetahuan mencakup aksioma Peano

$$NatNum(0)$$
$$\forall\, n\; NatNum(n) \;\Rightarrow\; NatNum(S(n))\,,$$

kemudian forward chaining menambahkan $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$, dan seterusnya. Masalah ini tidak dapat dihindari secara umum. Seperti halnya logika orde pertama umum, konsekuensi dengan klausa pasti bersifat semidecidable.

Forward chaining yang efisien

Algoritma forward chaining pada Gambar 9.3 dirancang untuk kemudahan pemahaman daripada untuk efisiensi operasi. Ada tiga kemungkinan sumber inefisiensi. Pertama, "loop dalam" dari algoritma melibatkan pencarian semua kemungkinan pemersatu sehingga premis aturan menyatu dengan serangkaian fakta yang sesuai dalam basis pengetahuan. Ini sering disebut **pencocokan pola** dan bisa sangat mahal. Kedua, algoritma memeriksa ulang setiap aturan pada setiap iterasi untuk melihat apakah premisnya terpenuhi, bahkan jika sangat sedikit penambahan yang dilakukan pada basis pengetahuan pada setiap iterasi. Akhirnya, algoritma mungkin menghasilkan banyak fakta yang tidak relevan dengan tujuan. Kami membahas masing-masing masalah ini secara bergantian.

Mencocokkan aturan dengan fakta yang diketahui

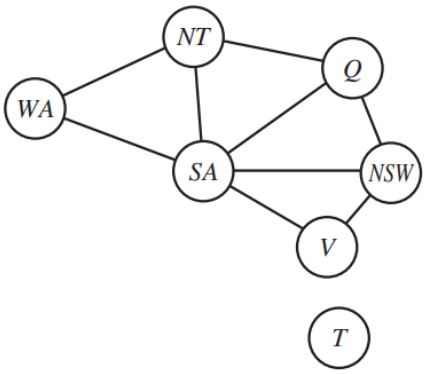
Masalah mencocokkan premis aturan dengan fakta dalam basis pengetahuan mungkin tampak cukup sederhana. Misalnya, anggaplah kita ingin menerapkan aturan

$$Missile(x) \Rightarrow Weapon(x) .$$

Kemudian kita perlu menemukan semua fakta yang menyatu dengan Missile(x); dalam basis pengetahuan yang diindeks dengan tepat, ini dapat dilakukan dalam waktu konstan per fakta. Sekarang pertimbangkan aturan seperti.

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono) .$$

Sekali lagi, kita dapat menemukan semua objek yang dimiliki oleh Nono dalam waktu konstan per objek; kemudian, untuk setiap objek, kita dapat memeriksa apakah itu rudal. Namun, jika basis pengetahuan berisi banyak objek yang dimiliki oleh Nono dan sangat sedikit rudal, akan lebih baik untuk menemukan semua rudal terlebih dahulu dan kemudian memeriksa apakah rudal tersebut dimiliki oleh Nono. Ini adalah masalah **urutan konjungsi**: temukan urutan untuk menyelesaikan konjungsi premis aturan sehingga total biaya diminimalkan. Ternyata menemukan urutan optimal adalah NP-keras, tetapi heuristik yang baik tersedia. Misalnya, heuristik **nilai-nilai-tersisa-minimum** (MRV) yang digunakan untuk CSP dalam Bab 6 akan menyarankan untuk memesan konjungsi untuk mencari rudal terlebih dahulu jika rudal yang dimiliki oleh Nono lebih sedikit daripada objek.



(a)

$$Diff(wa, nt) \wedge Diff(wa, sa) \wedge$$
$$Diff(nt, q) \wedge Diff(nt, sa) \wedge$$
$$Diff(q, nsw) \wedge Diff(q, sa) \wedge$$
$$Diff(nsw, v) \wedge Diff(nsw, sa) \wedge$$
$$Diff(v, sa) \Rightarrow Colorable()$$
$$Diff(Red, Blue) \quad Diff(Red, Green)$$
$$Diff(Green, Red) \quad Diff(Green, Blue)$$
$$Diff(Blue, Red) \quad Diff(Blue, Green)$$

(b)

Gambar 9.5 (a) Grafik kendala untuk mewarnai peta Australia. (b) CSP pewarnaan peta dinyatakan sebagai klausa tunggal yang pasti. Setiap wilayah peta direpresentasikan sebagai variabel yang nilainya dapat berupa salah satu konstanta $Red, Green\ atau\ Blue$.

Hubungan antara pencocokan pola dan pemenuhan kendala sebenarnya sangat dekat. Kita dapat melihat setiap konjungsi sebagai kendala pada variabel yang dikandungnya—misalnya, $Missile(x)$ adalah kendala unary pada x. Dengan memperluas ide ini, $kita\ dapat\ mengekspresikan\ setiap\ CSP\ domain - hingga\ sebagai\ klausa\ tunggal\ yang\ pasti\ bersama\ dengan\ beberapa\ fakta\ dasar\ terkait.$ Pertimbangkan masalah pewarnaan peta dari Gambar 6.1, yang ditunjukkan lagi pada Gambar 9.5(a). Formulasi yang setara sebagai klausa tunggal yang pasti diberikan pada Gambar 9.5(b). Jelas, kesimpulan *Colorable* () dapat disimpulkan hanya jika CSP memiliki solusi. Karena CSP secara umum menyertakan

masalah 3-SAT sebagai kasus khusus, kita dapat menyimpulkan bahwa $mencocokkan\ klausa\ tertentu\ dengan\ serangkaian\ fakta\ adalah\ NP-keras.$

Mungkin tampak agak menyedihkan bahwa forward chaining memiliki masalah pencocokan NP-keras dalam loop dalamnya. Ada tiga cara untuk menghibur diri:

- Kita dapat mengingatkan diri sendiri bahwa sebagian besar aturan dalam basis pengetahuan dunia nyata berukuran kecil dan sederhana (seperti aturan dalam contoh kejahatan kita) daripada berukuran besar dan rumit (seperti formulasi CSP pada Gambar 9.5). Di dunia basis data, adalah umum untuk menganggap bahwa baik ukuran aturan maupun aritas predikat dibatasi oleh suatu konstanta dan hanya mengkhawatirkan **kompleksitas data**—yaitu, kompleksitas inferensi sebagai fungsi dari jumlah fakta dasar dalam basis pengetahuan. Mudah untuk menunjukkan bahwa kompleksitas data dari forward chaining bersifat polinomial.
- Kita dapat mempertimbangkan subkelas aturan yang pencocokannya efisien. Pada dasarnya setiap klausa Datalog dapat dilihat sebagai penentu CSP, sehingga pencocokan akan dapat ditangani tepat saat CSP yang sesuai dapat ditangani. Bab 6 menjelaskan beberapa keluarga CSP yang dapat ditangani. Misalnya, jika grafik kendala (grafik yang simpulnya adalah variabel dan tautannya adalah kendala) membentuk pohon, maka CSP dapat diselesaikan dalam waktu linier. Hasil yang persis sama berlaku untuk pencocokan aturan. Misalnya, jika kita menghapus Australia Selatan dari peta pada Gambar 9.5, klausa yang dihasilkan adalah

  $$Diff\ (wa, nt)\ \wedge\ Diff\ (nt, q)\ \wedge\ Diff\ (q, nsw)\ \wedge\ Diff\ (nsw, v)\ \Rightarrow\ Colorable()$$

  yang sesuai dengan CSP tereduksi yang ditunjukkan pada Gambar 6.12 di halaman 224. Algoritma untuk memecahkan CSP berstruktur pohon dapat diterapkan secara langsung pada masalah pencocokan aturan.
- Kita dapat mencoba menghilangkan upaya pencocokan aturan yang berlebihan dalam algoritma forward-chaining, seperti yang dijelaskan selanjutnya.

Incremental forward chaining

Ketika kita menunjukkan cara kerja forward chaining pada contoh kejahatan, kita curang; khususnya, kita menghilangkan beberapa pencocokan aturan yang dilakukan oleh algoritma yang ditunjukkan pada Gambar 9.3. Misalnya, pada iterasi kedua, aturan

$$Missile(x)\ \Rightarrow\ Weapon(x)$$

cocok dengan Missile(M_1) (lagi), dan tentu saja kesimpulan $Weapon(M_1)$ sudah diketahui jadi tidak terjadi apa-apa. Pencocokan aturan yang berulang seperti itu dapat dihindari jika kita membuat pengamatan berikut: *Setiap fakta baru yang disimpulkan pada iterasi t harus diturunkan dari setidaknya satu fakta baru yang disimpulkan pada iterasi t* − 1. Ini benar karena setiap inferensi yang tidak memerlukan fakta baru dari iterasi $t - 1$ sudah dapat dilakukan pada iterasi $t - 1$.

Pengamatan ini secara alami mengarah pada algoritma forward-chaining inkremental di mana, pada iterasi t, kita memeriksa aturan hanya jika premisnya mencakup konjungsi p'_i yang menyatu dengan fakta p' yang baru disimpulkan pada iterasi $t-1$. Langkah pencocokan aturan kemudian memperbaiki pi agar cocok dengan p', tetapi memungkinkan konjungsi lain dari aturan tersebut untuk cocok dengan fakta dari iterasi sebelumnya. Algoritma ini menghasilkan fakta yang sama persis pada setiap iterasi seperti algoritma pada Gambar 9.3, tetapi jauh lebih efisien.

Dengan pengindeksan yang sesuai, mudah untuk mengidentifikasi semua aturan yang dapat dipicu oleh fakta apa pun, dan memang banyak sistem nyata beroperasi dalam mode "pembaruan" di mana forward chaining terjadi sebagai respons terhadap setiap fakta baru yang DIBERITAHUKAN ke sistem. Inferensi mengalir melalui serangkaian aturan hingga titik tetap tercapai, dan kemudian proses dimulai lagi untuk fakta baru berikutnya.

Biasanya, hanya sebagian kecil aturan dalam basis pengetahuan yang benar-benar dipicu oleh penambahan fakta tertentu. Ini berarti bahwa banyak pekerjaan yang berlebihan dilakukan dalam membangun kecocokan parsial yang memiliki beberapa premis yang tidak terpenuhi secara berulang. Contoh kejahatan kita agak terlalu kecil untuk menunjukkan hal ini secara efektif, tetapi perhatikan bahwa kecocokan parsial dibangun pada iterasi pertama antara aturan

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

dan fakta $American(West)$. Pencocokan parsial ini kemudian dibuang dan dibangun kembali pada iterasi kedua (ketika aturan berhasil). Akan lebih baik untuk mempertahankan dan secara bertahap melengkapi pencocokan parsial saat fakta baru tiba, daripada membuangnya.

Algoritma **rete** [3] adalah yang pertama mengatasi masalah ini. Algoritma tersebut memproses terlebih dahulu serangkaian aturan dalam basis pengetahuan untuk membangun semacam jaringan aliran data di mana setiap simpul adalah literal dari premis aturan. Pengikatan variabel mengalir melalui jaringan dan disaring ketika gagal mencocokkan literal. Jika dua literal dalam aturan berbagi variabel—misalnya, $Sells(x, y, z) \wedge Hostile(z)$ dalam contoh kejahatan—maka pengikatan dari setiap literal disaring melalui simpul kesetaraan. Pengikatan variabel yang mencapai simpul untuk literal n-ary seperti $Sells(x, y, z)$ mungkin harus menunggu pengikatan agar variabel lain terbentuk sebelum proses dapat dilanjutkan. Pada titik tertentu, status jaringan rete menangkap semua kecocokan parsial aturan, sehingga menghindari banyak perhitungan ulang.

Jaringan rete, dan berbagai perbaikannya, telah menjadi komponen utama dari apa yang disebut **sistem produksi**, yang merupakan salah satu sistem forward-chaining paling awal yang digunakan secara luas.[4] Sistem XCON (awalnya disebut R1) dibangun dengan arsitektur sistem produksi. XCON berisi beberapa ribu aturan untuk merancang konfigurasi komponen komputer bagi pelanggan Digital Equipment Corporation. Itu adalah salah satu keberhasilan komersial pertama yang jelas dalam bidang sistem pakar yang sedang berkembang. Banyak

sistem serupa lainnya telah dibangun dengan teknologi dasar yang sama, yang telah diimplementasikan dalam bahasa tujuan umum OPS-5.

Sistem produksi juga populer dalam **arsitektur kognitif**—yaitu, model penalaran manusia—seperti ACT dan SOAR. Dalam sistem seperti itu, "memori kerja" sistem memodelkan memori jangka pendek manusia, dan produksi merupakan bagian dari memori jangka panjang. Pada setiap siklus operasi, produksi dicocokkan dengan memori kerja fakta. Produksi yang kondisinya terpenuhi dapat menambah atau menghapus fakta dalam memori kerja. Berbeda dengan situasi umum dalam basis data, sistem produksi sering kali memiliki banyak aturan dan relatif sedikit fakta. Dengan teknologi pencocokan yang dioptimalkan dengan tepat, beberapa sistem modern dapat beroperasi secara real time dengan puluhan juta aturan.

Fakta yang tidak relevan

Sumber inefisiensi terakhir dalam forward chaining tampaknya intrinsik terhadap pendekatan tersebut dan juga muncul dalam konteks proposisional. Forward chaining membuat semua inferensi yang diperbolehkan berdasarkan fakta-fakta yang diketahui, *meskipun inferensi tersebut tidak relevan dengan tujuan yang sedang dibahas.* Dalam contoh kejahatan kita, tidak ada aturan yang mampu menarik kesimpulan yang tidak relevan, sehingga kurangnya keterarahan bukanlah masalah. Dalam kasus lain (misalnya, jika banyak aturan menggambarkan kebiasaan makan orang Amerika dan harga rudal), FOL-FC-ASK akan menghasilkan banyak kesimpulan yang tidak relevan.

Salah satu cara untuk menghindari penarikan kesimpulan yang tidak relevan adalah dengan menggunakan backward chaining, seperti yang dijelaskan dalam Bagian 9.4. Solusi lain adalah membatasi forward chaining ke subset aturan yang dipilih, seperti dalam PL-FC-ENTAILS?. Pendekatan ketiga telah muncul di bidang **basis data deduktif**, yang merupakan basis data berskala besar, seperti basis data relasional, tetapi yang menggunakan forward chaining sebagai alat inferensi standar alih-alih kueri SQL. Idenya adalah menulis ulang set aturan, menggunakan informasi dari tujuan, sehingga hanya pengikatan variabel yang relevan—yang termasuk dalam apa yang disebut **set ajaib**—yang dipertimbangkan selama inferensi maju. Misalnya, jika tujuannya adalah $Criminal(West)$, aturan yang menyimpulkan $Criminal(x)$ akan ditulis ulang untuk menyertakan konjungsi tambahan yang membatasi nilai x:

$$Magic(x) \land American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$$

Fakta $Magic(West)$ juga ditambahkan ke KB. Dengan cara ini, meskipun basis pengetahuan berisi fakta tentang jutaan orang Amerika, hanya Kolonel West yang akan dipertimbangkan selama proses inferensi maju. Proses lengkap untuk mendefinisikan himpunan magic dan menulis ulang basis pengetahuan terlalu rumit untuk dibahas di sini, tetapi ide dasarnya adalah untuk melakukan semacam inferensi mundur "generik" dari tujuan untuk mengetahui pengikatan variabel mana yang perlu dibatasi. Oleh karena itu, pendekatan himpunan magic dapat dianggap sebagai semacam hibrida antara inferensi maju dan praproses mundur.

9.4 BACKWARD CHAINING

Keluarga utama kedua dari algoritma inferensi logis menggunakan pendekatan **backward chaining** yang diperkenalkan di Bagian 7.5 untuk klausa tertentu. Algoritma ini bekerja mundur dari tujuan, merangkai melalui aturan untuk menemukan fakta yang diketahui yang mendukung pembuktian. Kami menjelaskan algoritma dasar, dan kemudian kami menjelaskan bagaimana ia digunakan dalam **pemrograman logika**, yang merupakan bentuk penalaran otomatis yang paling banyak digunakan. Kami juga melihat bahwa backward chaining memiliki beberapa kelemahan dibandingkan dengan forward chaining, dan kami melihat cara untuk mengatasinya. Akhirnya, kami melihat hubungan dekat antara pemrograman logika dan masalah pemenuhan kendala.

Algoritma backward-chaining

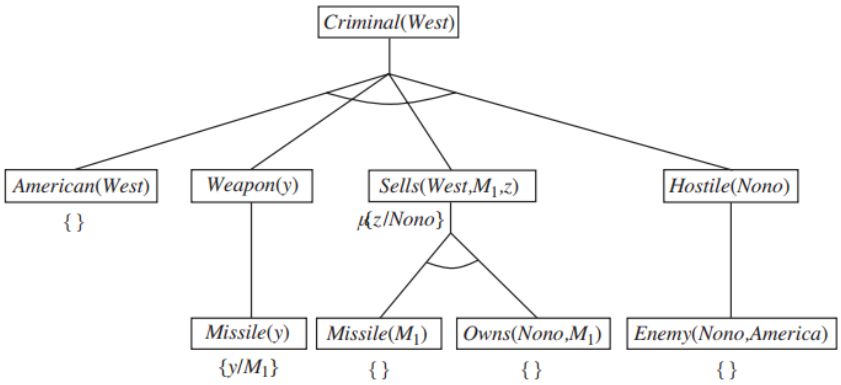
Gambar 9.6 menunjukkan algoritma backward-chaining untuk klausa tertentu. FOL-BC-ASK(KB, *goal*) akan dibuktikan jika basis pengetahuan berisi klausa dalam bentuk $lhs \Rightarrow goal$, di mana lhs (sisi kiri) adalah daftar konjungsi. Fakta atomik seperti $American(West)$ dianggap sebagai klausa yang lhs-nya adalah daftar kosong. Kini, kueri yang berisi variabel dapat dibuktikan dengan berbagai cara. Misalnya, kueri $Person(x)$ dapat dibuktikan dengan substitusi $\{x/John\}$ dan juga dengan $\{x/Richard\}$. Jadi, kami menerapkan FOL-BC-ASK sebagai **generator**—fungsi yang menghasilkan beberapa hasil, yang setiap kali menghasilkan satu kemungkinan hasil.

Backward chaining adalah sejenis penelusuran AND/OR—bagian OR karena kueri tujuan dapat dibuktikan dengan aturan apa pun dalam basis pengetahuan, dan bagian AND karena semua konjungsi dalam lhs klausa harus dibuktikan. FOL-BC-OR bekerja dengan mengambil semua klausa yang mungkin menyatu dengan tujuan, menstandardisasi variabel dalam klausa menjadi variabel baru, dan kemudian, jika rhs klausa memang menyatu dengan tujuan, membuktikan setiap konjungsi dalam lhs, menggunakan FOL-BC-AND. Fungsi tersebut pada gilirannya bekerja dengan membuktikan setiap konjungsi secara bergantian, melacak substitusi yang terakumulasi saat kita melakukannya. Gambar 9.7 adalah pohon pembuktian untuk memperoleh $Criminal\ (West\ )$ dari kalimat (9.3) hingga (9.10).

Backward chaining, sebagaimana telah kami tulis, jelas merupakan algoritma pencarian mendalam. Ini berarti bahwa kebutuhan ruangnya bersifat linier dalam ukuran pembuktian (mengabaikan, untuk saat ini, ruang yang dibutuhkan untuk mengumpulkan solusi). Ini juga berarti bahwa backward chaining (tidak seperti forward chaining) mengalami masalah dengan status berulang dan ketidaklengkapan. Kami akan membahas masalah ini dan beberapa solusi potensial, tetapi pertama-tama kami akan menunjukkan bagaimana backward chaining digunakan dalam sistem pemrograman logika.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
    return FOL-BC-OR(KB, query, { })

---

generator FOL-BC-OR(KB, goal, θ) yields a substitution
    for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
        (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
        for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
            yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
    if θ = failure then return
    else if LENGTH(goals)=0 then yield θ
    else do
        first,rest ← FIRST(goals), REST(goals)
        for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
            for each θ" in FOL-BC-AND(KB, rest, θ') do
                yield θ"
```

Gambar9.6 Algoritma backward-chaining sederhana untuk basis pengetahuan tingkat pertama.



Gambar 9.7 Pohon pembuktian yang dibangun dengan backward chaining untuk membuktikan bahwa West adalah seorang kriminal. Pohon tersebut harus dibaca secara mendalam terlebih dahulu, dari kiri ke kanan. Untuk membuktikan $Criminal(West)$, kita harus membuktikan empat konjungsi di bawahnya. Beberapa di antaranya ada di basis pengetahuan, dan yang lainnya memerlukan backward chaining lebih lanjut. Pengikatan untuk setiap penyatuan yang berhasil ditunjukkan di samping subtujuan yang sesuai. Perhatikan bahwa setelah satu subtujuan dalam konjungsi berhasil, substitusinya diterapkan ke subtujuan berikutnya. Jadi, pada saat FOL-BC-ASK sampai ke konjungsi terakhir, yang awalnya $Hostile(z)$, z sudah terikat ke $Nono$.

Pemrograman logika

Pemrograman logika adalah teknologi yang cukup mendekati perwujudan ideal deklaratif yang dijelaskan dalam Bab 7: bahwa sistem harus dibangun dengan

mengekspresikan pengetahuan dalam bahasa formal dan bahwa masalah harus dipecahkan dengan menjalankan proses inferensi pada pengetahuan tersebut. Idealnya diringkas dalam persamaan Robert Kowalski,

$$Algorithm = Logic + Control$$

Prolog adalah bahasa pemrograman logika yang paling banyak digunakan. Bahasa ini digunakan terutama sebagai bahasa pembuatan prototipe cepat dan untuk tugas manipulasi simbol seperti menulis kompiler dan mengurai bahasa alami. Banyak sistem pakar telah ditulis dalam Prolog untuk bidang hukum, medis, keuangan, dan bidang lainnya.

Program Prolog adalah serangkaian klausa pasti yang ditulis dalam notasi yang agak berbeda dari logika orde pertama standar. Prolog menggunakan huruf kapital untuk variabel dan huruf kecil untuk konstanta—kebalikan dari konvensi kita untuk logika. Koma memisahkan konjungsi dalam klausa, dan klausa ditulis "mundur" dari yang biasa kita gunakan; alih-alih $A \wedge B \Rightarrow C$ dalam Prolog kita memiliki $\mathrm{C} :\text{-}\ \mathrm{A}, \mathrm{B}$. Berikut adalah contoh tipikal:

$$\mathrm{criminal(X)} :- \mathrm{american(X)}, \mathrm{weapon(Y)}, \mathrm{sells(X, Y, Z)}, \mathrm{hostile(Z)}.$$

Notasi [E|L] menunjukkan suatu daftar yang elemen pertamanya adalah E dan sisanya adalah L. Berikut adalah program Prolog untuk append(X,Y,Z), yang berhasil jika daftar Z adalah hasil penambahan daftar X dan Y:

$$\mathrm{append}([\,], \mathrm{Y}, \mathrm{Y}).$$
$$\mathrm{append}([\mathrm{A}|\mathrm{X}], \mathrm{Y}, [\mathrm{A}|\mathrm{Z}]) :- \mathrm{append}(\mathrm{X}, \mathrm{Y}, \mathrm{Z}).$$

Dalam bahasa Inggris, kita dapat membaca klausa-klausa ini sebagai (1) menambahkan daftar kosong dengan daftar Y menghasilkan daftar Y yang sama dan (2) [A|Z] adalah hasil dari menambahkan [A|X] ke Y, asalkan Z adalah hasil dari menambahkan X ke Y. Dalam kebanyakan bahasa tingkat tinggi, kita dapat menulis fungsi rekursif serupa yang menjelaskan cara menambahkan dua daftar. Namun, definisi Prolog sebenarnya jauh lebih kuat, karena ia menjelaskan *hubungan* yang berlaku di antara tiga argumen, daripada *fungsi* yang dihitung dari dua argumen. Misalnya, kita dapat menanyakan $\mathrm{append(X, Y, [1,2])}$: dua daftar apa yang dapat ditambahkan untuk menghasilkan $[1,2]$? Kita mendapatkan kembali solusinya

$$\mathrm{X} = [] \qquad \mathrm{Y} = [1,2];$$
$$\mathrm{X} = [1] \qquad \mathrm{Y} = [2];$$
$$\mathrm{X} = [1,2] \qquad \mathrm{Y} = []$$

Eksekusi program Prolog dilakukan melalui depth-first backward chaining, di mana klausa dicoba dalam urutan penulisannya di basis pengetahuan. Beberapa aspek Prolog berada di luar inferensi logis standar:

- Prolog menggunakan semantik basis data alih-alih semantik orde pertama, dan ini tampak dalam penanganannya terhadap kesetaraan dan negasi.
- Ada serangkaian fungsi bawaan untuk aritmatika. Literal yang menggunakan simbol fungsi ini "dibuktikan" dengan mengeksekusi kode alih-alih melakukan inferensi lebih lanjut. Misalnya, tujuan "X adalah $4 + 3$" berhasil dengan X terikat ke 7. Di sisi lain, tujuan "5 adalah $X + Y$" gagal, karena fungsi bawaan tidak melakukan penyelesaian persamaan yang sewenang-wenang
- Ada predikat bawaan yang memiliki efek samping saat dieksekusi. Ini termasuk predikat input–output dan predikat assert/retract untuk memodifikasi basis pengetahuan. Predikat semacam itu tidak memiliki padanan dalam logika dan dapat menghasilkan hasil yang membingungkan—misalnya, jika fakta dinyatakan dalam cabang pohon pembuktian yang akhirnya gagal.
- **Pemeriksaan kejadian** dihilangkan dari algoritma penyatuan Prolog. Ini berarti bahwa beberapa kesimpulan yang tidak tepat dapat dibuat; ini hampir tidak pernah menjadi masalah dalam praktik.
- Prolog menggunakan pencarian rantai mundur kedalaman-pertama tanpa pemeriksaan untuk rekursi tak terbatas. Ini membuatnya sangat cepat ketika diberikan serangkaian aksioma yang tepat, tetapi tidak lengkap ketika diberikan yang salah.

Desain Prolog merupakan kompromi antara deklaratif dan efisiensi eksekusi—sejauh efisiensi dipahami pada saat Prolog dirancang.

Implementasi program logika yang efisien

Eksekusi program Prolog dapat terjadi dalam dua mode: ditafsirkan dan dikompilasi. Interpretasi pada dasarnya sama dengan menjalankan algoritma FOL-BC-ASK dari Gambar 9.6, dengan program sebagai basis pengetahuan. Kami mengatakan "pada dasarnya" karena interpreter Prolog berisi berbagai perbaikan yang dirancang untuk memaksimalkan kecepatan. Di sini kami hanya mempertimbangkan dua hal.

Pertama, implementasi kami harus secara eksplisit mengelola iterasi atas kemungkinan hasil yang dihasilkan oleh masing-masing subfungsi. Interpreter Prolog memiliki struktur data global, tumpukan **titik pilihan**, untuk melacak berbagai kemungkinan yang kami pertimbangkan dalam FOL-BC-OR. Tumpukan global ini lebih efisien, dan memudahkan penelusuran kesalahan, karena debugger dapat bergerak ke atas dan ke bawah tumpukan.

Kedua, implementasi FOL-BC-ASK kami yang sederhana menghabiskan banyak waktu untuk menghasilkan substitusi. Alih-alih secara eksplisit membuat substitusi, Prolog memiliki variabel logika yang mengingat pengikatannya saat ini. Pada setiap titik waktu, setiap variabel dalam program tidak terikat atau terikat pada suatu nilai. Bersama-sama, variabel dan nilai ini secara implisit mendefinisikan substitusi untuk cabang pembuktian saat ini. Memperluas jalur hanya dapat menambahkan pengikatan variabel baru, karena upaya untuk menambahkan pengikatan yang berbeda untuk variabel yang sudah terikat mengakibatkan kegagalan penyatuan. Ketika jalur dalam pencarian gagal, Prolog akan kembali ke titik pilihan sebelumnya, dan kemudian mungkin harus melepaskan beberapa variabel. Ini dilakukan dengan melacak semua variabel yang telah diikat dalam tumpukan yang disebut **jejak**. Karena

setiap variabel baru diikat oleh UNIFY-VAR, variabel tersebut didorong ke jejak. Ketika suatu tujuan gagal dan saatnya untuk kembali ke titik pilihan sebelumnya, masing-masing variabel tidak terikat karena dihapus dari jejak.

Bahkan interpreter Prolog yang paling efisien memerlukan beberapa ribu instruksi mesin per langkah inferensi karena biaya pencarian indeks, penyatuan, dan membangun tumpukan panggilan rekursif. Pada dasarnya, penafsir selalu bertindak seolah-olah belum pernah melihat program tersebut sebelumnya; misalnya, ia harus menemukan klausa yang sesuai dengan tujuan. Di sisi lain, program Prolog yang dikompilasi adalah prosedur inferensi untuk serangkaian klausa tertentu, sehingga ia mengetahui klausa mana yang sesuai dengan tujuan. Prolog pada dasarnya menghasilkan pembuktian teorema mini untuk setiap predikat yang berbeda, sehingga menghilangkan banyak overhead penafsiran. Rutinitas penyatuan juga dapat dikodekan **secara terbuka** untuk setiap panggilan yang berbeda, sehingga menghindari analisis eksplisit terhadap struktur istilah. (Untuk detail penyatuan yang dikodekan secara terbuka)

```
procedure APPEND(ax , y, az , continuation)

    trail ← GLOBAL-TRAIL-POINTER()
    if ax = [ ] and UNIFY(y, az ) then CALL(continuation)
    RESET-TRAIL(trail)
    a, x , z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
    if UNIFY(ax , [a | x ]) and UNIFY(az , [a | z ]) then APPEND(x , y, z , continuation)
```

Gambar 9.8 Pseudocode yang menggambarkan hasil kompilasi predikat Append. Fungsi NEW-VARIABLE mengembalikan variabel baru, berbeda dari semua variabel lain yang digunakan sejauh ini. Prosedur CALL(*continuation*) melanjutkan eksekusi dengan kelanjutan yang ditentukan.

Rangkaian instruksi komputer masa kini kurang cocok dengan semantik Prolog, sehingga sebagian besar kompiler Prolog mengompilasi ke dalam bahasa perantara daripada langsung ke bahasa mesin. Bahasa perantara yang paling populer adalah Warren Abstract Machine, atau WAM, yang dinamai menurut David H. D. Warren, salah satu pelaksana kompiler Prolog pertama. WAM adalah rangkaian instruksi abstrak yang cocok untuk Prolog dan dapat diinterpretasikan atau diterjemahkan ke dalam bahasa mesin. Kompiler lain menerjemahkan Prolog ke dalam bahasa tingkat tinggi seperti Lisp atau C, lalu menggunakan kompiler bahasa tersebut untuk menerjemahkannya ke bahasa mesin. Misalnya, definisi predikat Append dapat dikompilasi ke dalam kode yang ditunjukkan pada Gambar 9.8. Beberapa hal yang perlu disebutkan:

- Daripada harus mencari klausa Append di basis pengetahuan, klausa tersebut menjadi prosedur dan inferensi dilakukan hanya dengan memanggil prosedur tersebut.

- Seperti yang dijelaskan sebelumnya, pengikatan variabel saat ini disimpan dalam jejak. Langkah pertama dari prosedur ini menyimpan status jejak saat ini, sehingga dapat dipulihkan oleh RESET-TRAIL jika klausa pertama gagal. Ini akan membatalkan semua ikatan yang dihasilkan oleh panggilan pertama ke UNIFY.
- Bagian tersulit adalah penggunaan **kelanjutan** untuk mengimplementasikan titik pilihan. Anda dapat menganggap kelanjutan sebagai pengemasan prosedur dan daftar argumen yang bersama-sama menentukan apa yang harus dilakukan selanjutnya setiap kali tujuan saat ini berhasil. Tidak akan berhasil jika hanya kembali dari prosedur seperti APPEND ketika tujuan berhasil, karena dapat berhasil dalam beberapa cara, dan masing-masing harus dieksplorasi. Argumen kelanjutan memecahkan masalah ini karena dapat dipanggil setiap kali tujuan berhasil. Dalam kode APPEND, jika argumen pertama kosong dan argumen kedua menyatu dengan yang ketiga, maka predikat APPEND telah berhasil. Kami kemudian MEMANGGIL kelanjutan, dengan ikatan yang sesuai di jejak, untuk melakukan apa pun yang harus dilakukan selanjutnya. Misalnya, jika panggilan ke APPEND berada di tingkat atas, kelanjutan akan mencetak ikatan variabel.

Sebelum Warren mengerjakan kompilasi inferensi dalam Prolog, pemrograman logika terlalu lambat untuk penggunaan umum. Kompiler oleh Warren dan yang lainnya memungkinkan kode Prolog mencapai kecepatan yang kompetitif dengan C pada berbagai tolok ukur standar (Van Roy, 1990). Tentu saja, fakta bahwa seseorang dapat menulis perencana atau pengurai bahasa alami dalam beberapa lusin baris Prolog membuatnya agak lebih diinginkan daripada C untuk membuat prototipe sebagian besar proyek penelitian AI skala kecil.

Paralelisasi juga dapat memberikan percepatan yang substansial. Ada dua sumber utama paralelisme. Yang pertama, disebut **paralelisme-OR**, berasal dari kemungkinan tujuan yang menyatu dengan banyak klausa berbeda dalam basis pengetahuan. Masing-masing memunculkan cabang independen dalam ruang pencarian yang dapat mengarah ke solusi potensial, dan semua cabang tersebut dapat dipecahkan secara paralel. Yang kedua, disebut **paralelisme-AND**, berasal dari kemungkinan memecahkan setiap konjungsi dalam badan implikasi secara paralel. Paralelisme AND lebih sulit dicapai, karena solusi untuk seluruh konjungsi memerlukan pengikatan yang konsisten untuk semua variabel. Setiap cabang konjungtif harus berkomunikasi dengan cabang lain untuk memastikan solusi global.

Inferensi redundan dan loop tak terbatas

Sekarang kita beralih ke titik lemah Prolog: ketidakcocokan antara pencarian mendalam dan pohon pencarian yang mencakup status berulang dan jalur tak terbatas. Pertimbangkan program logika berikut yang memutuskan apakah ada jalur antara dua titik pada grafik berarah:

$$\text{path}(X, Z) :- \text{link}(X, Z).$$
$$\text{path}(X, Z) :- \text{path}(X, Y), \text{link}(Y, Z).$$

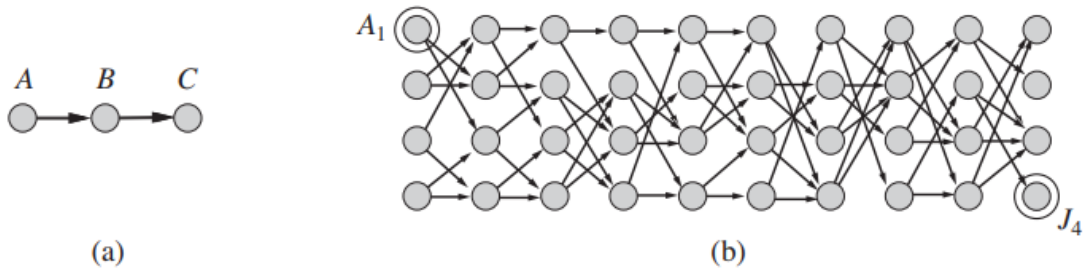
Grafik tiga simpul sederhana, yang dijelaskan oleh fakta $\text{link}(a, b)$ dan $\text{link}(b, c)$, ditunjukkan pada Gambar 9.9(a). Dengan program ini, query $\text{path}(a, c)$ menghasilkan pohon pembuktian *yang* ditunjukkan pada Gambar 9.10(a). Di sisi lain, jika kita menempatkan dua klausa dalam urutan

$$\text{path}(X, Z) :- \text{path}(X, Y), \text{link}(Y, Z).$$
$$\text{path}(X, Z) :- \text{link}(X, Z).$$

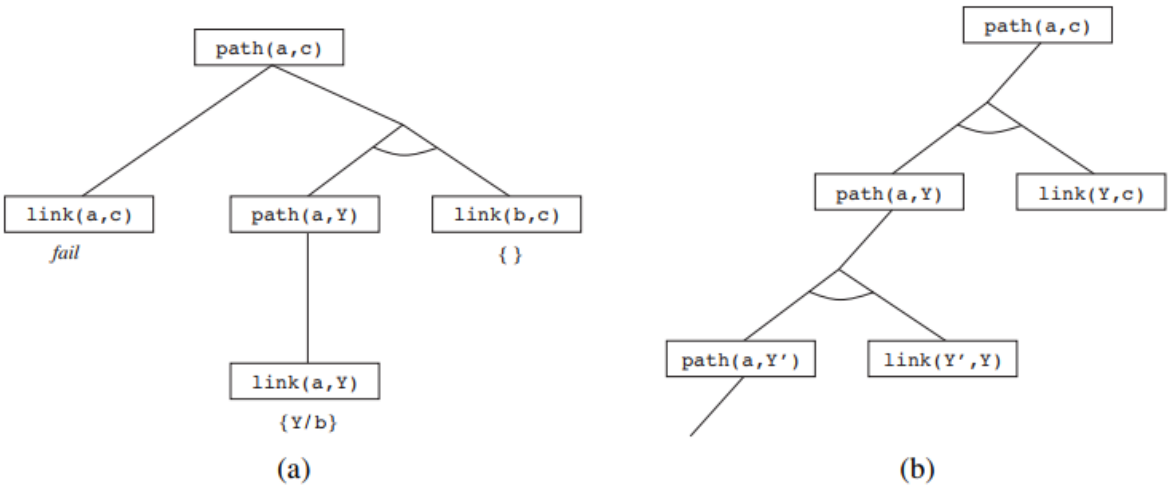
maka Prolog mengikuti jalur tak terbatas yang ditunjukkan pada Gambar 9.10(b). Oleh karena itu Prolog **tidak lengkap** sebagai pembuktian teorema untuk klausa tertentu—bahkan untuk program Datalog, seperti yang ditunjukkan contoh ini—karena, untuk beberapa basis pengetahuan, ia gagal membuktikan kalimat yang disyaratkan. Perhatikan bahwa forward chaining tidak mengalami masalah ini: setelah $\text{link}(a, b), \text{link}(b, c), \text{dan link}(a, c)$ disimpulkan, forward chaining berhenti.

Depth-first backward chaining juga memiliki masalah dengan komputasi yang berlebihan. Misalnya, ketika menemukan jalur dari A_1 ke J_4 pada Gambar 9.9(b), Prolog melakukan 877 inferensi, yang sebagian besar melibatkan pencarian semua jalur yang mungkin ke simpul tempat tujuan tidak dapat dicapai. Ini mirip dengan masalah keadaan berulang yang dibahas dalam Bab 3. Jumlah total inferensi dapat eksponensial dalam jumlah fakta dasar yang dihasilkan. Jika kita menerapkan forward chaining sebagai gantinya, paling banyak n^2 fakta $\text{jalur}(X, Y)$ dapat dihasilkan dengan menghubungkan n simpul. Untuk masalah pada Gambar 9.9(b), hanya diperlukan 62 inferensi.

Forward chaining pada masalah pencarian grafik adalah contoh **pemrograman dinamis**, di mana solusi untuk submasalah dibangun secara bertahap dari solusi submasalah yang lebih kecil dan disimpan dalam cache untuk menghindari perhitungan ulang. Kita dapat memperoleh efek serupa dalam sistem backward chaining menggunakan **memoisasi**—yaitu, menyimpan solusi untuk subtujuan saat ditemukan dan kemudian menggunakan kembali solusi tersebut saat subtujuan berulang, daripada mengulang perhitungan sebelumnya.



Gambar 9.9 (a) Menemukan jalur dari A ke C dapat menyebabkan Prolog menjadi loop tak terbatas. (b) Grafik yang setiap simpulnya terhubung ke dua penerus acak di lapisan berikutnya. Menemukan jalur dari A_1 ke J_4 memerlukan 877 inferensi.



Gambar 9.10 (a) Bukti adanya jalur dari *A* ke *C*. (b) Pohon bukti tak terhingga dihasilkan ketika klausa berada dalam urutan yang "salah".

Ini adalah pendekatan yang diambil oleh sistem **pemrograman logika tabel**, yang menggunakan mekanisme penyimpanan dan pengambilan yang efisien untuk melakukan memoisasi. Pemrograman logika tabel menggabungkan keterarahan tujuan backward chaining dengan efisiensi pemrograman dinamis forward chaining. Ini juga lengkap untuk basis pengetahuan Datalog, yang berarti bahwa programmer tidak perlu khawatir tentang loop tak terbatas. (Masih mungkin untuk mendapatkan perulangan tak terhingga dengan predikat seperti $\mathrm{Father}(\mathrm{X}, \mathrm{Y})$ yang merujuk pada jumlah objek yang berpotensi tak terbatas.)

Semantik basis data Prolog

Prolog menggunakan semantik basis data, seperti yang dibahas di Bagian 8.2. Asumsi nama unik menyatakan bahwa setiap konstanta Prolog dan setiap istilah dasar merujuk ke objek yang berbeda, dan asumsi dunia tertutup menyatakan bahwa satu-satunya kalimat yang benar adalah kalimat yang disyaratkan oleh basis pengetahuan. Tidak ada cara untuk menyatakan bahwa suatu kalimat salah dalam Prolog. Hal ini membuat Prolog kurang ekspresif dibandingkan logika orde pertama, tetapi merupakan bagian dari apa yang membuat Prolog lebih efisien dan lebih ringkas. Pertimbangkan pernyataan Prolog berikut tentang beberapa penawaran kursus:

$$Course(CS, 101), Course(CS, 102), Course(CS, 106), Course(EE, 101). \qquad (9.11)$$

Berdasarkan asumsi nama unik, CS dan EE berbeda (seperti halnya 101, 102, dan 106), jadi ini berarti ada empat mata kuliah yang berbeda. Berdasarkan asumsi dunia tertutup tidak ada mata kuliah lain, jadi ada tepat empat mata kuliah. Namun, jika ini adalah pernyataan dalam FOL dan bukan dalam Prolog, maka yang dapat kita katakan adalah bahwa ada mata kuliah antara satu dan tak terhingga. Itu karena pernyataan (dalam FOL) tidak menyangkal kemungkinan bahwa mata kuliah lain yang tidak disebutkan juga ditawarkan, juga tidak mengatakan bahwa mata kuliah yang disebutkan berbeda satu sama lain. Jika kita ingin menerjemahkan Persamaan (9.11) ke dalam FOL, kita akan mendapatkan ini:

$$Course(d,n) \Leftrightarrow (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101)\,. \quad (9.12)$$

Ini disebut **penyelesaian** Persamaan (9.11). Persamaan ini menyatakan dalam FOL gagasan bahwa paling banyak ada empat mata kuliah. Untuk menyatakan dalam FOL gagasan bahwa paling sedikit ada empat mata kuliah, kita perlu menulis penyelesaian predikat kesetaraan:

$$x = y \Leftrightarrow (x = CS \wedge y = CS) \vee (x = EE \wedge y = EE) \vee (x = 101 \wedge y = 101) \vee (x = 102 \wedge y = 102) \vee (x = 106 \wedge y = 106)\,.$$

Penyelesaian ini berguna untuk memahami semantik basis data, tetapi untuk tujuan praktis, jika masalah Anda dapat dijelaskan dengan semantik basis data, akan lebih efisien untuk bernalar dengan Prolog atau sistem semantik basis data lainnya, daripada menerjemahkannya ke dalam FOL dan bernalar dengan pembuktian teorema FOL yang lengkap.

Pemrograman logika kendala

Dalam pembahasan kita tentang forward chaining (Bagian 9.3), kita menunjukkan bagaimana masalah pemenuhan kendala (CSP) dapat dikodekan sebagai klausa yang pasti. Prolog standar memecahkan masalah tersebut dengan cara yang persis sama seperti algoritma backtracking yang diberikan pada Gambar 6.5.

Karena backtracking menghitung domain variabel, ia hanya berfungsi untuk CSPs **domain-hingga**. Dalam istilah Prolog, harus ada sejumlah solusi terbatas untuk setiap tujuan dengan variabel yang tidak terikat. (Misalnya, goal $\mathrm{diff(Q,SA)}$, yang menyatakan bahwa Queensland dan Australia Selatan harus memiliki warna yang berbeda, memiliki enam solusi jika tiga warna diperbolehkan.) CSP domain tak terbatas—misalnya, dengan variabel bernilai integer atau riil—memerlukan algoritme yang sangat berbeda, seperti propagasi batas atau pemrograman linier.

Pertimbangkan contoh berikut. Kami mendefinisikan $\mathrm{segitiga(X,Y,Z)}$ sebagai predikat yang berlaku jika tiga argumen adalah angka yang memenuhi ketidaksetaraan segitiga:

$$triangle(X,Y,Z) :-$$
$$X > 0, Y > 0, Z > 0, X + Y >= Z, Y + Z >= X, X + Z >= Y.$$

Jika kita menanyakan Prolog $\mathrm{segitiga(3,4,5)}$, maka berhasil. Di sisi lain, jika kita menanyakan $\mathrm{segitiga(3,4,Z)}$, tidak akan ditemukan solusi, karena subgoal $\mathrm{Z} >= 0$ tidak dapat ditangani oleh Prolog; kita tidak dapat membandingkan nilai yang tidak terikat dengan 0.

Pemrograman logika kendala (CLP) memungkinkan variabel dibatasi daripada dibatasi. Solusi CLP adalah kumpulan kendala paling spesifik pada variabel kueri yang dapat diturunkan dari basis pengetahuan. Misalnya, solusi untuk kueri $\mathrm{segitiga(3,4,Z)}$ adalah kendala $7 >= Z >= 1$. Program logika standar hanyalah kasus khusus CLP di mana kendala solusi harus berupa kendala kesetaraan—yaitu, pengikatan.

Sistem CLP menggabungkan berbagai algoritme penyelesaian kendala untuk kendala yang diizinkan dalam bahasa tersebut. Misalnya, sistem yang memungkinkan ketidaksetaraan linier pada variabel bernilai riil mungkin menyertakan algoritme pemrograman linier untuk menyelesaikan kendala tersebut. Sistem CLP juga mengadopsi pendekatan yang jauh lebih fleksibel untuk menyelesaikan kueri pemrograman logika standar. Misalnya, alih-alih penelusuran balik kiri-ke-kanan yang mendalam, sistem tersebut dapat menggunakan salah satu algoritme yang lebih efisien yang dibahas dalam Bab 6, termasuk pengurutan konjungsi heuristik, lompatan balik, pengkondisian set potongan, dan sebagainya. Oleh karena itu, sistem CLP menggabungkan elemen algoritma pemenuhan kendala, pemrograman logika, dan basis data deduktif.

Beberapa sistem yang memungkinkan programmer memiliki kontrol lebih besar atas urutan pencarian untuk inferensi telah ditetapkan. Bahasa MRS memungkinkan programmer untuk menulis **metarules** guna menentukan konjungsi mana yang dicoba terlebih dahulu. Pengguna dapat menulis aturan yang menyatakan bahwa tujuan dengan variabel paling sedikit harus dicoba terlebih dahulu atau dapat menulis aturan khusus domain untuk predikat tertentu.

9.5 RESOLUSI

Keluarga sistem logika terakhir dari tiga keluarga kita didasarkan pada **resolusi**. Kita lihat di halaman 250 bahwa resolusi proposisional menggunakan sanggahan adalah prosedur inferensi lengkap untuk logika proposisional. Di bagian ini, kita menjelaskan cara memperluas resolusi ke logika orde pertama.

Bentuk normal konjungtif untuk logika orde pertama

Seperti dalam kasus proposisional, resolusi orde pertama mengharuskan kalimat berada dalam **bentuk normal konjungtif** (CNF)—yaitu, konjungsi klausa, di mana setiap klausa merupakan disjungsi literal. Literal dapat berisi variabel, yang diasumsikan terkuantifikasi secara universal. Misalnya, kalimat

$$\forall\, x\ American(x) \ \wedge\ Weapon(y) \ \wedge\ Sells(x, y, z) \ \wedge\ Hostile(z) \ \Rightarrow\ Criminal(x)$$

menjadi, dalam CNF,

$$\neg American(x) \ \vee\ \neg Weapon(y) \ \vee\ \neg Sells(x, y, z) \ \vee\ \neg Hostile(z) \ \vee\ Criminal(x)\,.$$

Setiap kalimat logika orde pertama dapat diubah menjadi kalimat CNF yang ekuivalen secara inferensial. Secara khusus, kalimat CNF akan tidak memuaskan ketika kalimat aslinya tidak memuaskan, jadi kita memiliki dasar untuk melakukan pembuktian dengan kontradiksi pada kalimat CNF.

Prosedur konversi ke CNF mirip dengan kasus proposisional, yang kita lihat di halaman 253. Perbedaan utama muncul dari kebutuhan untuk menghilangkan kuantifier eksistensial.

Kita mengilustrasikan prosedur tersebut dengan menerjemahkan kalimat "Setiap orang yang mencintai semua hewan dicintai oleh seseorang," atau

$$\forall\, x\, [\forall\, y\, Animal(y) \;\Rightarrow\; Loves(x,y)] \;\Rightarrow\; [\exists\, y\, Loves(y,x)]\,.$$

Langkah-langkahnya adalah sebagai berikut:

- **Menghilangkan implikasi:**

$$\forall\, x\, [\neg\forall\, y\, \neg Animal(y) \;\vee\; Loves(x,y)] \;\vee\; [\exists\, y\, Loves(y,x)]$$

- **Pindahkan ч ke dalam**: Selain aturan umum untuk konektif yang dinegasikan, kita memerlukan aturan untuk kuantifier yang dinegasikan. Jadi, kita memiliki

$$\neg\forall\, x\; p\; becomes\; \exists\, x\; \neg p$$
$$\neg\exists\, x\; p\; becomes\; \forall\, x\; \neg p\,.$$

  Kalimat kita melalui transformasi berikut:

$$\forall\, x\, [\exists\, y\, \neg(\neg Animal(y) \;\vee\; Loves(x,y))] \;\vee\; [\exists\, y\, Loves(y,x)]\,.$$
$$\forall\, x\, [\exists\, y\, \neg\neg Animal(y) \;\wedge\; \neg Loves(x,y)] \;\vee\; [\exists\, y\, Loves(y,x)]\,.$$
$$\forall\, x\, [\exists\, y\, Animal(y) \;\wedge\; \neg Loves(x,y)] \;\vee\; [\exists\, y\, Loves(y,x)]\,.$$

  Perhatikan bagaimana kuantifier universal (∀ y) dalam premis implikasi telah menjadi kuantifier eksistensial. Kalimat tersebut sekarang berbunyi "Entah ada hewan yang tidak dicintai x, atau (jika tidak demikian) seseorang mencintai x." Jelas, makna kalimat aslinya telah dipertahankan.

- **Standarisasi variabel:** Untuk kalimat seperti $(\exists\, xP\,(x))\, V(\exists\, x\, Q(x))$ yang menggunakan nama variabel yang sama dua kali, ubah nama salah satu variabel. Ini menghindari kebingungan nanti saat kita menghilangkan kuantifier. Jadi, kita memiliki

$$\forall\, x\, [\exists\, y\, Animal(y) \;\wedge\; \neg Loves(x,y)] \;\vee\; [\exists\, z\, Loves(z,x)]\,.$$

- **Skolemisasi: Skolemisasi** adalah proses menghilangkan kuantifier eksistensial dengan eliminasi. Dalam kasus sederhana, ini seperti aturan Instansiasi Eksistensial dari Bagian 9.1: terjemahkan $\exists\, x\; P\;(x)$ menjadi $P\;(A)$, di mana A adalah konstanta baru. Namun, kita tidak dapat menerapkan Instansiasi Eksistensial pada kalimat kita di atas karena tidak cocok dengan pola $\exists\; v\; \alpha$; hanya bagian kalimat yang cocok dengan pola tersebut. Jika kita secara membabi buta menerapkan aturan tersebut pada dua bagian yang cocok, kita memperoleh

$$\forall\, x\, [Animal(A) \;\wedge\; \neg Loves(x,A)] \;\vee\; Loves(B,x)\,,$$

  yang memiliki makna yang sepenuhnya salah: kalimat itu mengatakan bahwa setiap orang gagal mencintai hewan tertentu A atau dicintai oleh entitas tertentu B. Faktanya,

kalimat asli kita memperbolehkan setiap orang gagal mencintai hewan yang berbeda atau dicintai oleh orang yang berbeda. Jadi, kita ingin entitas Skolem bergantung pada x dan z:

$$\forall x\ [Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(z), x)\ .$$

Di sini F dan G adalah **fungsi Skolem**. Aturan umumnya adalah bahwa argumen fungsi Skolem adalah semua variabel yang dikuantifikasi secara universal yang dalam lingkupnya kuantifier eksistensial muncul. Seperti halnya Instansiasi Eksistensial, kalimat yang di-Skolemisasi dapat dipenuhi tepat ketika kalimat aslinya dapat dipenuhi.

- **Hilangkan kuantifier universal**: Pada titik ini, semua variabel yang tersisa harus dikuantifikasi secara universal. Selain itu, kalimat tersebut setara dengan kalimat di mana semua kuantifier universal telah dipindahkan ke kiri. Oleh karena itu, kita dapat menghilangkan kuantifier universal:

$$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(z), x)\ .$$

- **Distribusikan** $\vee$ **ke** $\wedge$:

$$\big[Animal\big(F(x)\big) \vee Loves(G(z), x)\big] \wedge \big[\neg Loves\big(x, F(x)\big) \vee Loves(G(z), x)\big].$$

Langkah ini mungkin juga memerlukan perataan konjungsi dan disjungsi yang bersarang.

Kalimat tersebut sekarang dalam CNF dan terdiri dari dua klausa. Kalimat tersebut sama sekali tidak dapat dibaca. (Mungkin akan membantu untuk menjelaskan bahwa fungsi Skolem $F\ (x)$ merujuk pada hewan yang mungkin tidak dicintai oleh x, sedangkan $G(z)$ merujuk pada seseorang yang mungkin mencintai x.) Untungnya, manusia jarang perlu melihat kalimat CNF—proses penerjemahan dapat dengan mudah diotomatisasi.

Aturan inferensi resolusi

Aturan resolusi untuk klausa orde pertama hanyalah versi terangkat dari aturan resolusi proposisional. Dua klausa, yang diasumsikan distandarisasi secara terpisah sehingga tidak berbagi variabel, dapat diselesaikan jika mengandung literal komplementer. Literal proposisional bersifat komplementer jika salah satunya merupakan negasi dari yang lain; literal orde pertama bersifat komplementer jika salah satunya $menyatu\ dengan$ negasi yang lain. Jadi, kita memiliki

$$\frac{\ell_1 \mathrm{V} \ldots \mathrm{V} \ell_{K,} \qquad m_1 \mathrm{V} \ldots \mathrm{V} m_n}{SUBST(\theta, \ell_1 \mathrm{V} \ldots \mathrm{V} \ell_{i-1} \mathrm{V} \ell_{i+1} \mathrm{V} \ldots \mathrm{V} \ell_k \mathrm{V} m_1 \mathrm{V} \ldots \mathrm{V} m_{J-1} V m_{j+1} V \ldots V m_n}$$

di mana UNIFY(l_i, $\neg m_i$)= θ. Misalnya, kita dapat menyelesaikan dua klausa

$$[Animal(F(x)) \vee Loves(G(x), x)]\ and\ [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

dengan menghilangkan literal komplementer $Loves(G(x), x)$ dan $\neg Loves(u, v)$, dengan pemersatu $\theta = \{u/G(x), v/x\}$, untuk menghasilkan klausa **resolven**

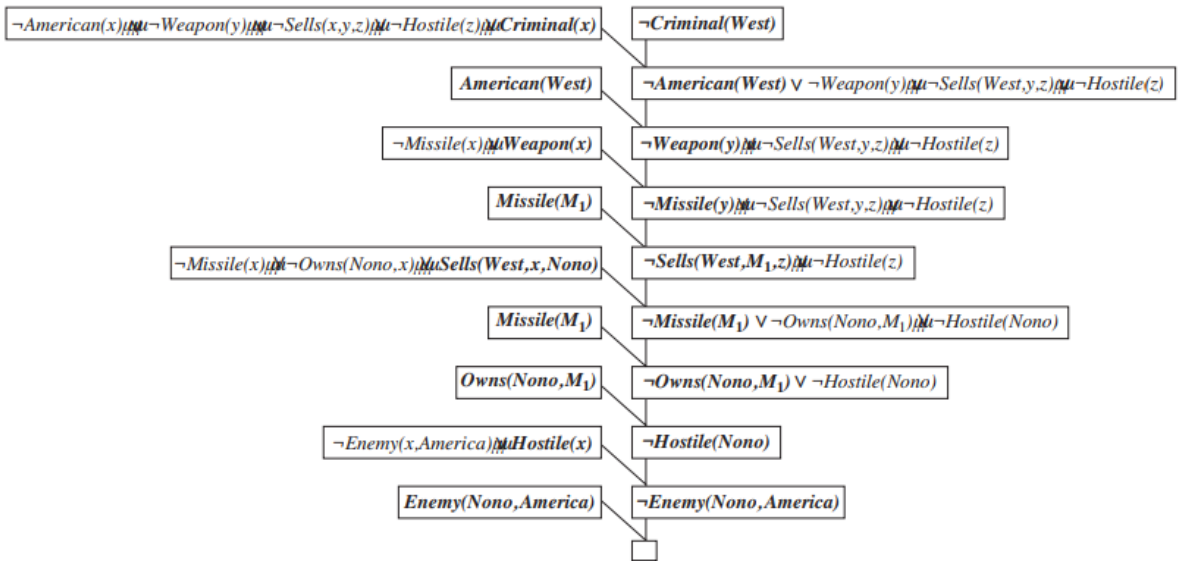
$$[Animal(F(x)) \lor \neg Kills(G(x), x)] .$$

Aturan ini disebut aturan **resolusi biner** karena aturan ini menyelesaikan tepat dua literal. Aturan resolusi biner itu sendiri tidak menghasilkan prosedur inferensi yang lengkap. Aturan resolusi penuh menyelesaikan himpunan bagian literal dalam setiap klausa yang tidak dapat disatukan. Pendekatan alternatif adalah memperluas **pemfaktoran**—penghapusan literal yang berlebihan—ke kasus orde pertama. Pemfaktoran proposisional mereduksi dua literal menjadi satu jika keduanya $identik$; pemfaktoran orde pertama mereduksi dua literal menjadi satu jika keduanya $tidak\ dapat\ disatukan$. Pemersatu harus diterapkan ke seluruh klausa. Kombinasi resolusi biner dan pemfaktoran bersifat lengkap.

Contoh pembuktian

Resolusi membuktikan bahwa $KB \models \alpha$ dengan membuktikan $KB \land \neg\alpha$ tidak memuaskan, yaitu, dengan menurunkan klausa kosong. Pendekatan algoritmik identik dengan kasus proposisional, yang dijelaskan dalam Gambar 7.12, jadi kita tidak perlu mengulanginya di sini. Sebaliknya, kita memberikan dua contoh pembuktian. Yang pertama adalah contoh kejahatan dari Pasal 9.3. Kalimat dalam CNF adalah

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x, y, z) \lor \neg Hostile(z) \lor Criminal(x)$$
$$\neg Missile(x) \lor \neg Owns(Nono, x) \lor Sells(West, x, Nono)$$
$$\neg Enemy(x, America) \lor Hostile(x)$$
$$\neg Missile(x) \lor Weapon(x)$$
$$Owns(Nono, M1) \qquad Missile(M1)$$
$$American(West) \qquad Enemy(Nono, America) .$$



Gambar 9.11 Sebuah bukti resolusi bahwa West adalah seorang penjahat. Pada setiap langkah, huruf-huruf yang menyatukan dicetak tebal.

Kami juga menyertakan tujuan yang dinegasikan $\neg Criminal(West)$. Bukti resolusi ditunjukkan pada Gambar 9.11. Perhatikan strukturnya: "tulang punggung" tunggal yang

dimulai dengan klausa tujuan, diselesaikan terhadap klausa dari basis pengetahuan hingga klausa kosong dihasilkan. Ini adalah karakteristik resolusi pada basis pengetahuan klausa Horn. Faktanya, klausa di sepanjang tulang punggung utama berkorespondensi persis dengan nilai berurutan dari variabel tujuan dalam algoritma rantai mundur Gambar 9.6. Ini karena kami selalu memilih untuk menyelesaikan dengan klausa yang literal positifnya menyatu dengan literal paling kiri dari klausa "saat ini" pada tulang punggung; inilah yang terjadi dalam rantai mundur. Jadi, rantai mundur hanyalah kasus khusus resolusi dengan strategi kontrol tertentu untuk memutuskan resolusi mana yang akan dilakukan selanjutnya.

Contoh kedua kami menggunakan Skolemisasi dan melibatkan klausa yang bukan klausa pasti. Ini menghasilkan struktur bukti yang agak lebih rumit. Dalam bahasa Inggris, masalahnya adalah sebagai berikut:

> Setiap orang yang mencintai semua hewan dicintai oleh seseorang.
> Siapa pun yang membunuh binatang tidak akan dicintai oleh siapa pun.
> Jack mencintai semua binatang.
> Jack atau Curiosity yang membunuh kucing bernama Tuna.
> Apakah Curiosity yang membunuh kucing itu?

Pertama, kita ungkapkan kalimat asli, beberapa pengetahuan latar belakang, dan tujuan G yang dinegasikan dalam logika orde pertama:

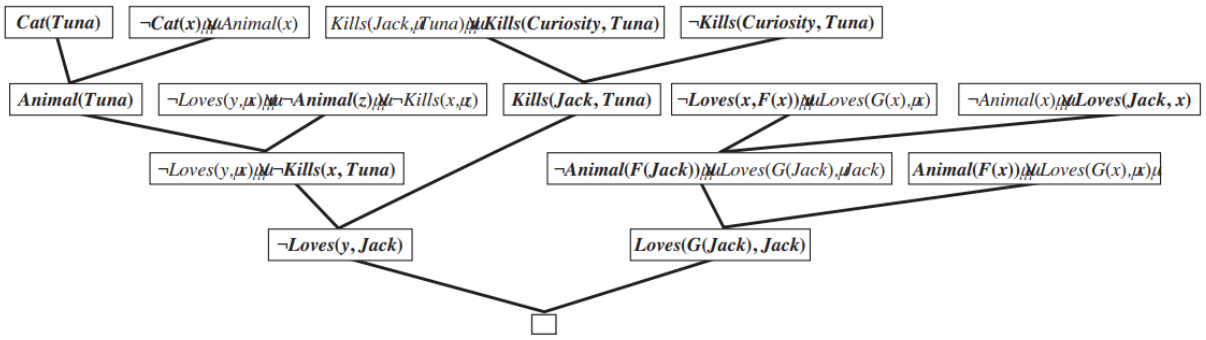
A. $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$
B. $\forall x\ [\exists z\ Animal(z) \wedge Kills(x,z)] \Rightarrow [\forall y \neg Loves(y,x)]$
C. $\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$
D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
E. $Cat(Tuna)$
F. $\forall x\ Cat(x) \Rightarrow Animal(x)$
¬G. $\neg\forall x\ Cat(x) \Rightarrow Animal(x)$

Sekarang kita terapkan prosedur konversi untuk mengonversi setiap kalimat menjadi CNF:

A1. $Animal(F(x)) \vee Loves(G(x),x)$
A2. $\neg Loves(x,F(x)) \vee Loves(G(x),x)$
B. $\neg Loves(y,x) \vee \neg Animal(z) \vee \neg Kills(x,z)$
C. $\neg Animal(x) \vee Loves(Jack,x)$
D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
E. $Cat(Tuna)$
F. $\neg Cat(x) \vee Animal(x)$
¬G. $\neg Kills(Curiosity,Tuna)$

Bukti resolusi bahwa Curiosity membunuh kucing diberikan pada Gambar 9.12. Dalam bahasa Inggris, buktinya dapat diparafrasakan sebagai berikut:

Misalkan Curiosity tidak membunuh Tuna. Kita tahu bahwa Jack atau Curiosity yang melakukannya; jadi Jack pastilah yang melakukannya. Nah, Tuna adalah kucing dan kucing adalah hewan, jadi Tuna adalah hewan. Karena siapa pun yang membunuh hewan tidak dicintai oleh siapa pun, kita tahu bahwa tidak ada yang mencintai Jack. Di sisi lain, Jack mencintai semua hewan, jadi ada yang mencintainya; jadi kita memiliki kontradiksi. Jadi, Curiosity membunuh kucing.



Gambar 9.12 Bukti resolusi bahwa Curiosity membunuh kucing. Perhatikan penggunaan faktorisasi dalam derivasi klausa $Loves\ (G(Jack\ ), Jack\ )$. Perhatikan juga di kanan atas, penyatuan $Loves\ (x, F\ (x))$ dan $Loves(Jack, x)$ hanya dapat berhasil setelah variabel-variabel distandarisasi secara terpisah.

Bukti tersebut menjawab pertanyaan "Apakah Curiosity membunuh kucing?" tetapi sering kali kita ingin mengajukan pertanyaan yang lebih umum, seperti "Siapa yang membunuh kucing?" Resolusi dapat melakukan ini, tetapi butuh sedikit usaha lebih untuk mendapatkan jawabannya. Sasarannya adalah $\exists\ w\ \ Kills(w, Tuna\ )$, yang, ketika dinegasikan, menjadi $\neg Kills(w, Tuna\ )$ dalam CNF. Mengulangi pembuktian pada Gambar 9.12 dengan sasaran baru yang dinegasikan, kita memperoleh pohon pembuktian yang serupa, tetapi dengan substitusi $\{w/Curiosity\ \}$ dalam salah satu langkah. Jadi, dalam kasus ini, mencari tahu siapa yang membunuh kucing hanyalah masalah melacak pengikatan untuk variabel kueri dalam pembuktian.

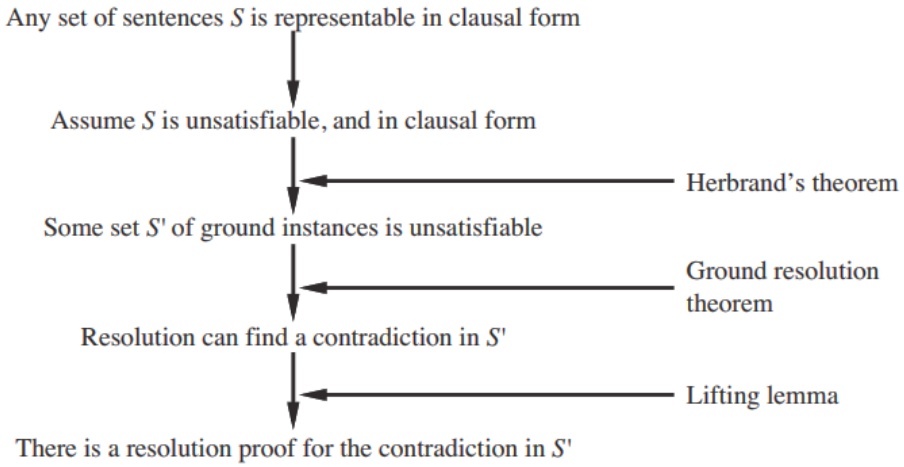
Sayangnya, resolusi dapat menghasilkan **pembuktian yang tidak konstruktif** untuk sasaran eksistensial. Misalnya, $\neg Kills(w, Tuna\ )$ diselesaikan dengan $Kills(Jack, Tuna)$ $V\ Kills(Curiosity, Tuna)$ untuk memberikan $Kills(Jack, Tuna)$, yang diselesaikan lagi dengan $\neg Kills(w, Tuna\ )$ untuk menghasilkan klausa kosong. Perhatikan bahwa w memiliki dua pengikatan berbeda dalam pembuktian ini; resolusi memberi tahu kita bahwa, ya, seseorang membunuh Tuna—baik Jack atau Curiosity. Ini bukan kejutan besar! Salah satu solusinya adalah membatasi langkah resolusi yang diizinkan sehingga variabel kueri hanya dapat diikat satu kali dalam pembuktian yang diberikan; kemudian kita harus dapat menelusuri kembali kemungkinan pengikatan. Solusi lain adalah menambahkan **literal jawaban** khusus ke tujuan yang dinegasikan, yang menjadi $\neg Kills(w, Tuna)\ V\ Answer\ (w)$. Sekarang, proses resolusi menghasilkan jawaban setiap kali klausa dihasilkan yang hanya berisi satu literal jawaban. Untuk pembuktian pada Gambar 9.12, ini adalah $Answer(Curiosity\ )$. Bukti

nonkonstruktif akan menghasilkan klausa $Jawaban\ (Keingintahuan)\ V\ Jawaban\ (Jack)$, yang bukan merupakan jawaban.

Kelengkapan penyelesaian

Bagian ini memberikan bukti kelengkapan penyelesaian. Bukti ini dapat dilewati dengan aman oleh mereka yang bersedia mempercayainya. Kami menunjukkan bahwa penyelesaian bersifat **sanggahan-lengkap**, yang berarti bahwa jika serangkaian kalimat tidak dapat dipenuhi, maka penyelesaian akan selalu dapat menghasilkan kontradiksi. Penyelesaian tidak dapat digunakan untuk menghasilkan semua konsekuensi logis dari serangkaian kalimat, tetapi dapat digunakan untuk menetapkan bahwa kalimat tertentu disyaratkan oleh serangkaian kalimat tersebut. Oleh karena itu, bukti ini dapat digunakan untuk menemukan semua jawaban untuk pertanyaan tertentu, $\neg Q(x)$, dengan membuktikan bahwa $KB \wedge \neg Q(x)$ tidak dapat dipenuhi.

Kami menganggap bahwa kalimat apa pun dalam logika orde pertama (tanpa kesetaraan) dapat ditulis ulang sebagai serangkaian klausa dalam CNF. Hal ini dapat dibuktikan dengan induksi pada bentuk kalimat, menggunakan kalimat atomik sebagai kasus dasar (Davis dan Putnam, 1960). Oleh karena itu, tujuan kami adalah untuk membuktikan hal berikut: *jika S adalah himpunan klausa yang tidak dapat dipenuhi, maka penerapan sejumlah langkah resolusi yang terbatas pada S akan menghasilkan kontradiksi.*



Gambar 9.13 Struktur bukti kelengkapan untuk resolusi.

Sketsa pembuktian kami mengikuti pembuktian asli Robinson dengan beberapa penyederhanaan dari Genesereth dan Nilsson (1987). Struktur dasar pembuktian (Gambar 9.13) adalah sebagai berikut:

1. Pertama, kami mengamati bahwa jika S tidak dapat dipenuhi, maka ada himpunan tertentu dari $contoh\ dasar$ klausa S sehingga himpunan ini juga tidak dapat dipenuhi (teorema Herbrand).
2. Kami kemudian mengacu pada **teorema resolusi dasar** yang diberikan dalam Bab 7, yang menyatakan bahwa resolusi proposisional bersifat lengkap untuk kalimat dasar.
3. Kami kemudian menggunakan **lemma pengangkatan** untuk menunjukkan bahwa, untuk setiap pembuktian resolusi proposisional menggunakan himpunan kalimat

dasar, ada pembuktian resolusi orde pertama yang sesuai menggunakan kalimat orde pertama yang darinya kalimat dasar diperoleh.

Untuk melaksanakan langkah pertama, kita memerlukan tiga konsep baru:

- **Semesta Herbrand**: Jika S adalah himpunan klausa, maka H_S, semesta Herbrand dari *S*, adalah himpunan semua istilah dasar yang dapat dibangun dari yang berikut:
  a. Simbol fungsi dalam *S*, jika ada.
  b. Simbol konstanta dalam *S*, jika ada; jika tidak ada, maka simbol konstanta *A*.

  Misalnya, jika *S* hanya memuat klausa $\neg P\ (x, F\ (x, A))\ V\ ч Q(x, A)\ V\ R(x, B)$, maka H_S adalah himpunan tak terhingga dari istilah dasar berikut:

  $$\{A, B, F\ (A, A), F\ (A, B), F\ (B, A), F\ (B, B), F\ (A, F\ (A, A)), \ldots\}.$$
- **Kejenuhan**: Jika *S* adalah himpunan klausa dan *P* adalah himpunan istilah dasar, maka *P* (*S*), kejenuhan *S* terhadap *P*, adalah himpunan semua klausa dasar yang diperoleh dengan menerapkan semua substitusi istilah dasar yang mungkin konsisten dalam *P* dengan variabel dalam *S*.
- **Basis Herbrand**: Kejenuhan himpunan *S* klausa terhadap semesta Herbrand-nya disebut basis Herbrand dari *S*, ditulis sebagai $H_S(S)$. Misalnya, jika *S* hanya memuat klausa yang baru saja diberikan, maka $H_S(S)$ adalah himpunan klausa tak terhingga

$$\{\neg P(A, F(A, A))\ \vee\ \neg Q(A, A)\ \vee\ R(A, B),$$
$$\neg P(B, F(B, A))\ \vee\ \neg Q(B, A)\ \vee\ R(B, B),$$
$$\neg P(F(A, A), F(F(A, A), A))\ \vee\ \neg Q(F(A, A), A)\ \vee\ R(F(A, A), B),$$
$$\neg P(F(A, B), F(F(A, B), A))\ \vee\ \neg Q(F(A, B), A)\ \vee\ R(F(A, B), B), \ldots\}$$

Definisi-definisi ini memungkinkan kita untuk menyatakan bentuk **teorema Herbrand** (Herbrand, 1930):

> Jika himpunan *S* klausa tidak memuaskan, maka ada himpunan bagian terbatas dari $H_S(S)$ yang juga tidak memuaskan.

Misalkan *S*' menjadi himpunan bagian terbatas dari kalimat dasar. Sekarang, kita dapat mengacu pada teorema resolusi dasar (halaman 255) untuk menunjukkan bahwa **penutupan resolusi** *RC (S')* memuat klausa kosong. Artinya, menjalankan resolusi proposisional hingga tuntas pada S' akan menghasilkan kontradiksi.

Sekarang setelah kita menetapkan bahwa selalu ada pembuktian resolusi yang melibatkan beberapa himpunan bagian terbatas dari basis Herbrand S, langkah selanjutnya adalah menunjukkan bahwa ada pembuktian resolusi menggunakan klausa S itu sendiri, yang belum tentu merupakan klausa dasar. Kita mulai dengan mempertimbangkan satu penerapan aturan resolusi. Robinson menyatakan lemma ini:

> Misalkan C_1 dan C_2 adalah dua klausa tanpa variabel bersama, dan misalkan C' dan C' adalah contoh dasar dari C_1 dan C_2. Jika C' adalah resolven dari C' dan C', maka ada klausa *C* sehingga (1) *C* adalah resolven dari C_1 dan C_2 dan (2) C' adalah contoh dasar dari *C*.

Ini disebut **lemma pengangkatan**, karena ia mengangkat langkah pembuktian dari klausa dasar ke klausa orde pertama umum. Untuk membuktikan lemma pengangkatan dasarnya, Robinson harus menciptakan penyatuan dan memperoleh semua sifat dari sebagian besar pemersatu umum. Daripada mengulang pembuktian di sini, kami cukup mengilustrasikan lemma:

$$C1 = \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$$
$$C2 = \neg N(G(y), z) \vee P(H(y), z)$$
$$C'_1 = \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B)$$
$$C'_2 = \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A))$$
$$C = \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B)$$
$$C = \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B)\,.$$

Kita melihat bahwa *C*' memang merupakan contoh dasar dari *C*. Secara umum, agar *C′* dan *C′* memiliki resolven, keduanya harus dibangun dengan terlebih dahulu menerapkan pada C_1 dan C_2 pemersatu paling umum dari sepasang literal komplementer di C_1 dan C_2. Dari lemma pengangkatan, mudah untuk memperoleh pernyataan serupa tentang urutan penerapan aturan resolusi:

> Untuk klausa *C′* apa pun dalam penutupan resolusi *S′*, ada klausa *C* dalam penutupan resolusi *S* sedemikian rupa sehingga *C′* merupakan contoh dasar dari *C* dan derivasi *C* memiliki panjang yang sama dengan derivasi *C′*.

Dari fakta ini, dapat disimpulkan bahwa jika klausa kosong muncul dalam penutupan resolusi *S′*, klausa tersebut juga harus muncul dalam penutupan resolusi *S*. Hal ini karena klausa kosong tidak dapat menjadi contoh dasar dari klausa lainnya. Untuk rekapitulasi: kita telah menunjukkan bahwa jika *S* tidak dapat dipenuhi, maka ada derivasi terbatas dari klausa kosong menggunakan aturan resolusi. Penghapusan pembuktian teorema dari klausa dasar ke klausa tingkat pertama memberikan peningkatan kekuatan yang besar. Peningkatan ini berasal dari fakta bahwa pembuktian tingkat pertama perlu membuat variabel hanya sejauh yang diperlukan untuk pembuktian, sedangkan metode klausa dasar diperlukan untuk memeriksa sejumlah besar pembuatan variabel yang sembarangan.

Kesetaraan

Tidak ada satu pun metode inferensi yang dijelaskan sejauh ini dalam bab ini yang menangani pernyataan dalam bentuk x = y. Tiga pendekatan berbeda dapat diambil. Pendekatan pertama adalah dengan melakukan aksiomatisasi kesetaraan—untuk menuliskan kalimat tentang hubungan kesetaraan dalam basis pengetahuan. Kita perlu mengatakan bahwa kesetaraan bersifat refleksif, simetris, dan transitif, dan kita juga harus mengatakan bahwa kita dapat mengganti persamaan dengan persamaan dalam predikat atau fungsi apa

pun. Jadi, kita memerlukan tiga aksioma dasar, dan kemudian satu untuk setiap predikat dan fungsi:

$$\forall x\, x = x$$
$$\forall x, y\, x = y \Rightarrow y = x$$
$$\forall x, y, z\, x = y \wedge y = z \Rightarrow x = z$$

$$\forall x, y\, x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y))$$
$$\forall x, y\, x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y))$$
$$\vdots$$
$$\forall w, x, y, z\, w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z))$$
$$\forall w, x, y, z\, w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z))$$
$$\vdots$$

Dengan kalimat-kalimat ini, prosedur inferensi standar seperti resolusi dapat melakukan tugas-tugas yang memerlukan penalaran kesetaraan, seperti memecahkan persamaan matematika. Namun, aksioma-aksioma ini akan menghasilkan banyak kesimpulan, kebanyakan dari mereka tidak membantu pembuktian. Jadi telah ada pencarian cara-cara yang lebih efisien untuk menangani kesetaraan. Salah satu alternatifnya adalah menambahkan aturan inferensi daripada aksioma. Aturan yang paling sederhana, **demodulasi**, mengambil klausa unit $x = y$ dan beberapa klausa α yang berisi istilah x, dan menghasilkan klausa baru yang dibentuk dengan mensubstitusi y untuk x dalam α. Ini berfungsi jika istilah dalam α menyatu dengan x; tidak perlu sama persis dengan x. Perhatikan bahwa demodulasi bersifat terarah; diberikan $x = y, x$ selalu diganti dengan y, tidak pernah sebaliknya. Itu berarti bahwa demodulasi dapat digunakan untuk menyederhanakan ekspresi menggunakan demodulator seperti $x + 0 = x$ atau $x^1 = x$. Sebagai contoh lain, diberikan

$$Father\,(Father\,(x)) = PaternalGrandfather\,(x)$$
$$Birthdate(Father\,(Father\,(Bella)), 1926)$$

kita dapat menyimpulkan dengan demodulasi

$$Birthdate(PaternalGrandfather\,(Bella), 1926)\,.$$

Secara lebih formal, kita memiliki

- **Demodulasi**: Untuk setiap suku x, y, dan z, di mana z muncul di suatu tempat dalam literal mi dan di mana UNIFY$(x, z) = \theta$,

$$\frac{x = y, \qquad m_1 \mathrm{V} \cdots \mathrm{V} m_n}{SUB(SUBST(\theta, x), SUBST(\theta, y), m_1\, \mathrm{V} \cdots \mathrm{V} m_n\,)}$$

  di mana SUBST adalah substitusi biasa dari daftar pengikatan, dan SUB(x, y, m) berarti mengganti x dengan y di mana pun x muncul dalam m.

Aturan tersebut juga dapat diperluas untuk menangani klausa non-unit di mana literal kesetaraan muncul:

- **Paramodulasi:** Untuk setiap istilah x, y, dan z, di mana z muncul di suatu tempat dalam literal mi, dan di mana UNIFY$(x, z) = \theta$,

$$\frac{\ell_1 \mathrm{V} \cdots \mathrm{V} \ell_k \mathrm{V}\, x = y, \qquad m_1\, \mathrm{V} \cdots \mathrm{V} m_n}{SUB(SUBST(\theta, x), SUBST(\theta, y), SUBST(\theta, \ell_1 \mathrm{V} \cdots \mathrm{V} \ell_k \mathrm{V} m_1 \mathrm{V} \cdots \mathrm{V} m_N}$$

Misalnya saja dari

$$P(F(x, B), x) \vee Q(x) \textbf{ and } F(A, y) = y \vee R(y)$$

kita memiliki θ = UNIFY$(F\ (A, y), F\ (x, B)) = \{x/A, y/B\}$, dan kita dapat menyimpulkan dengan paramodulasi kalimat tersebut

$$P(B, A) \vee Q(A) \vee R(B)\,.$$

Paramodulasi menghasilkan prosedur inferensi lengkap untuk logika orde pertama dengan kesetaraan.

Pendekatan ketiga menangani penalaran kesetaraan sepenuhnya dalam algoritme penyatuan yang diperluas. Artinya, suku-suku dapat disatukan jika $terbukti$ sama di bawah beberapa substitusi, di mana "terbukti" memungkinkan penalaran kesetaraan. Misalnya, suku-suku $1 + 2$ dan $2 + 1$ biasanya tidak dapat disatukan, tetapi algoritme penyatuan yang mengetahui bahwa $x + y = y + x$ dapat menyatukannya dengan substitusi kosong. Penyatuan persamaan semacam ini dapat dilakukan dengan algoritme efisien yang dirancang untuk aksioma tertentu yang digunakan (komutativitas, asosiatif, dan sebagainya) daripada melalui inferensi eksplisit dengan aksioma tersebut. Pembuktian teorema yang menggunakan teknik ini terkait erat dengan sistem CLP yang dijelaskan dalam Bagian 9.4.

Strategi resolusi

Kita tahu bahwa penerapan berulang dari aturan inferensi resolusi pada akhirnya akan menemukan bukti jika ada. Dalam subbagian ini, kita memeriksa strategi yang membantu menemukan bukti secara efisien. Preferensi unit: Strategi ini lebih suka melakukan resolusi di mana salah satu kalimatnya adalah literal tunggal (juga dikenal sebagai klausa unit). Ide di balik strategi ini adalah bahwa kita mencoba menghasilkan klausa kosong, jadi mungkin ide yang bagus untuk lebih memilih inferensi yang menghasilkan klausa yang lebih pendek. Menyelesaikan kalimat unit (seperti P) dengan kalimat lain (seperti ¬P V ¬QVR) selalu menghasilkan klausa (dalam hal ini, ¬Q V R) yang lebih pendek daripada klausa lainnya.

Ketika strategi preferensi unit pertama kali dicoba untuk inferensi proposisional pada tahun 1964, hal itu menyebabkan percepatan yang dramatis, sehingga memungkinkan untuk membuktikan teorema yang tidak dapat ditangani tanpa preferensi. Resolusi unit adalah

bentuk resolusi terbatas di mana setiap langkah resolusi harus melibatkan klausa unit. Resolusi unit umumnya tidak lengkap, tetapi lengkap untuk klausa Horn.

Pembuktian resolusi unit pada klausa Horn menyerupai forward chaining. Pembuktian teorema OTTER, menggunakan bentuk pencarian terbaik-pertama. Fungsi heuristiknya mengukur "bobot" setiap klausa, di mana klausa yang lebih ringan lebih disukai. Pilihan heuristik yang tepat tergantung pada pengguna, tetapi umumnya, bobot klausa harus berkorelasi dengan ukuran atau tingkat kesulitannya. Klausa unit diperlakukan sebagai ringan; pencarian dengan demikian dapat dilihat sebagai generalisasi dari strategi preferensi unit.

Seperangkat dukungan: Preferensi yang mencoba resolusi tertentu terlebih dahulu bermanfaat, tetapi secara umum lebih efektif untuk mencoba menghilangkan beberapa resolusi potensial sama sekali. Misalnya, kita dapat menegaskan bahwa setiap langkah resolusi melibatkan setidaknya satu elemen dari serangkaian klausa khusus—seperangkat dukungan. Resolvent kemudian ditambahkan ke dalam serangkaian dukungan. Jika serangkaian dukungan kecil relatif terhadap seluruh basis pengetahuan, ruang pencarian akan berkurang drastis.

Kita harus berhati-hati dengan pendekatan ini karena pilihan yang buruk untuk serangkaian dukungan akan membuat algoritme tidak lengkap. Akan tetapi, jika kita memilih himpunan pendukung S sehingga kalimat-kalimat yang tersisa dapat dipenuhi secara bersama-sama, maka penyelesaian himpunan pendukung selesai.

Misalnya, seseorang dapat menggunakan kueri yang dinegasikan sebagai himpunan pendukung, dengan asumsi bahwa basis pengetahuan asli konsisten. (Lagipula, jika tidak konsisten, maka fakta bahwa kueri yang mengikutinya adalah hampa.) Strategi himpunan pendukung memiliki keuntungan tambahan dalam menghasilkan pohon pembuktian yang diarahkan pada tujuan yang sering kali mudah dipahami oleh manusia.

Resolusi masukan: Dalam strategi ini, setiap resolusi menggabungkan salah satu kalimat masukan (dari KB atau kueri) dengan beberapa kalimat lain. Pembuktian pada Gambar 9.11 hanya menggunakan resolusi masukan dan memiliki bentuk karakteristik berupa "tulang punggung" tunggal dengan kalimat tunggal yang digabungkan ke tulang punggung. Jelas, ruang pohon pembuktian dengan bentuk ini lebih kecil daripada ruang semua grafik pembuktian.

Dalam basis pengetahuan Horn, Modus Ponens adalah semacam strategi resolusi masukan, karena menggabungkan implikasi dari KB asli dengan beberapa kalimat lain. Jadi, tidak mengherankan bahwa resolusi masukan bersifat lengkap untuk basis pengetahuan yang berbentuk Horn, tetapi tidak lengkap dalam kasus umum. Strategi resolusi linier adalah generalisasi kecil yang memungkinkan P dan Q diselesaikan bersama baik jika P ada di KB asli atau jika P adalah leluhur Q di pohon pembuktian. Resolusi linier bersifat lengkap.

Subsumsi: Metode subsumsi menghilangkan semua kalimat yang disubsumsi oleh (yaitu, lebih spesifik daripada) kalimat yang ada di KB. Misalnya, jika $P\ (x)$ ada di KB, maka tidak ada gunanya menambahkan $P(A)$dan bahkan lebih tidak masuk akal lagi menambahkan $P(A) \vee Q(B)$. Subsumsi membantu menjaga KB tetap kecil dan dengan demikian membantu menjaga ruang pencarian tetap kecil.

Penggunaan praktis pembuktian teorema resolusi

Pembuktian teorema dapat diterapkan pada masalah yang terlibat dalam sintesis dan verifikasi perangkat keras dan perangkat lunak. Dengan demikian, penelitian pembuktian teorema dilakukan di bidang desain perangkat keras, bahasa pemrograman, dan rekayasa perangkat lunak—tidak hanya dalam AI.

Dalam kasus perangkat keras, aksioma menggambarkan interaksi antara sinyal dan elemen sirkuit. Pemikir logis yang dirancang khusus untuk verifikasi telah mampu memverifikasi seluruh CPU, termasuk sifat pengaturan waktunya. Pembuktian teorema AURA telah diterapkan untuk merancang sirkuit yang lebih kompak daripada desain sebelumnya. Dalam kasus perangkat lunak, penalaran tentang program cukup mirip dengan penalaran tentang tindakan, seperti dalam Bab 7: aksioma menggambarkan prasyarat dan efek dari setiap pernyataan.

Sintesis formal algoritma merupakan salah satu penggunaan pertama pembuktian teorema, sebagaimana diuraikan oleh Cordell Green (1969a), yang mengembangkan ide-ide awal Herbert Simon (1963). Idenya adalah untuk membuktikan teorema secara konstruktif yang menyatakan bahwa "ada program p yang memenuhi spesifikasi tertentu." Meskipun sintesis deduktif yang sepenuhnya otomatis, sebagaimana disebut, belum dapat diterapkan untuk pemrograman tujuan umum, sintesis deduktif yang dipandu secara manual telah berhasil merancang beberapa algoritma baru dan canggih. Sintesis program tujuan khusus, seperti kode komputasi ilmiah, juga merupakan bidang penelitian yang aktif.

Teknik serupa kini diterapkan pada verifikasi perangkat lunak oleh sistem seperti pemeriksa model SPIN. Misalnya, program kendali wahana antariksa Agen Jarak Jauh diverifikasi sebelum dan sesudah penerbangan. Algoritma enkripsi kunci publik RSA dan algoritma pencocokan string Boyer–Moore telah diverifikasi dengan cara ini.

9.6 RINGKASAN

Kami telah menyajikan analisis inferensi logis dalam logika orde pertama dan sejumlah algoritme untuk melakukannya.

- Pendekatan pertama menggunakan aturan inferensi (instansiasi universal dan instansiasi eksistensial) untuk mengajukan masalah inferensi. Biasanya, pendekatan ini lambat, kecuali domainnya kecil.
- Penggunaan penyatuan untuk mengidentifikasi substitusi yang tepat untuk variabel menghilangkan langkah instansiasi dalam pembuktian orde pertama, sehingga proses menjadi lebih efisien dalam banyak kasus.
- Versi Modus Ponens yang diangkat menggunakan penyatuan untuk menyediakan aturan inferensi yang alami dan kuat, Modus Ponens yang digeneralisasi. Algoritme forward-chaining dan backward-chaining menerapkan aturan ini ke set klausa yang pasti.
- Modus Ponens yang digeneralisasi lengkap untuk klausa yang pasti, meskipun masalah konsekuensinya semidecidable. Untuk basis pengetahuan Datalog yang terdiri dari klausa yang pasti bebas fungsi, konsekuensinya dapat diputuskan.

- Forward chaining digunakan dalam basis data deduktif, yang dapat dikombinasikan dengan operasi basis data relasional. Ia juga digunakan dalam sistem produksi, yang melakukan pembaruan efisien dengan set aturan yang sangat besar. Forward chaining bersifat lengkap untuk Datalog dan berjalan dalam waktu polinomial.
- Backward chaining digunakan dalam sistem pemrograman logika, yang menggunakan teknologi kompiler canggih untuk menyediakan inferensi yang sangat cepat. Backward chaining mengalami inferensi yang berlebihan dan loop tak terbatas; hal ini dapat diatasi dengan memoisasi.
- Prolog, tidak seperti logika orde pertama, menggunakan dunia tertutup dengan nama unik asumsi dan negasi sebagai kegagalan. Hal ini menjadikan Prolog bahasa pemrograman yang lebih praktis, tetapi membawanya lebih jauh dari logika murni.
- Aturan inferensi resolusi umum menyediakan sistem pembuktian lengkap untuk logika orde pertama, menggunakan basis pengetahuan dalam bentuk normal konjungtif.
- Ada beberapa strategi untuk mengurangi ruang pencarian sistem resolusi tanpa mengorbankan kelengkapan. Salah satu isu terpenting adalah menangani kesetaraan; kami menunjukkan bagaimana demodulasi dan paramodulasi dapat digunakan.
- Pembuktian teorema berbasis resolusi yang efisien telah digunakan untuk membuktikan teorema matematika yang menarik dan untuk memverifikasi dan mensintesis perangkat lunak dan perangkat keras.

DAFTAR PUSTAKA

Abdulameer, M., Mansoor, M. M., Alchuban, M., Rashed, A., Al-Showaikh, F., & Hamdan, A. (2022). The impact of artificial intelligence (AI) on the development of accounting and auditing profession. In *Technologies, Artificial Intelligence and the Future of Learning Post-COVID-19: The Crucial Role of International Accreditation* (pp. 201-213). Cham: Springer International Publishing.

Banh, L., & Strobel, G. (2023). Generative artificial intelligence. *Electronic Markets*, *33*(1), 63.

Baratelli, G., & Colleoni, E. (2022). Does artificial intelligence (AI) enabled recruitment improve employer branding?. *International Journal of Business and Management*.

Basha, M. (2023). Impact of artificial intelligence on marketing. *East Asian Journal of Multidisciplinary Research*, *2*(3), 993-1004.

Chiu, T. K., Meng, H., Chai, C. S., King, I., Wong, S., & Yam, Y. (2021). Creation and evaluation of a pretertiary artificial intelligence (AI) curriculum. *IEEE Transactions on Education*, *65*(1), 30-39.

Choudhary, O. P., Infant, S. S., Chopra, H., & Manuta, N. (2025). Exploring the potential and limitations of artificial intelligence in animal anatomy. *Annals of Anatomy-Anatomischer Anzeiger*, *258*, 152366.

Emetaram, E., & Uchime, H. N. (2021). Impact of artificial intelligence (AI) on accountancy profession. *Journal of Accounting and Financial Management*, *7*(2), 15-25.

Ertel, W. (2024). *Introduction to artificial intelligence*. Springer Nature.

Esmaeilzadeh, P. (2024). Challenges and strategies for wide-scale artificial intelligence (AI) deployment in healthcare practices: A perspective for healthcare organizations. *Artificial Intelligence in Medicine*, *151*, 102861.

Esplugas, M. (2023). The use of artificial intelligence (AI) to enhance academic communication, education and research: a balanced approach. *Journal of Hand Surgery (European Volume)*, *48*(8), 819-822.

Harrison Jr, J. H., Gilbertson, J. R., Hanna, M. G., Olson, N. H., Seheult, J. N., Sorace, J. M., & Stram, M. N. (2021). Introduction to artificial intelligence and machine learning for pathology. *Archives of pathology & laboratory medicine*, *145*(10), 1228-1254.\

Holm, S., Stanton, C., & Bartlett, B. (2021). A new argument for no-fault compensation in health care: the introduction of artificial intelligence systems. *Health care analysis*, *29*(3), 171-188.

Ibrahim, K. S. M. H., Huang, Y. F., Ahmed, A. N., Koo, C. H., & El-Shafie, A. (2022). A review of the hybrid artificial intelligence and optimization modelling of hydrological streamflow forecasting. *Alexandria Engineering Journal*, *61*(1), 279-303.

Kathikar, A., Nair, A., Lazarine, B., Sachdeva, A., & Samtani, S. (2023, October). Assessing the vulnerabilities of the open-source artificial intelligence (ai) landscape: A large-scale analysis of the hugging face platform. In *2023 IEEE International Conference on Intelligence and Security Informatics (ISI)* (pp. 1-6). IEEE.

Khalaf, W. S., Morgan, R. N., & Elkhatib, W. F. (2025). Clinical microbiology and artificial intelligence: Different applications, challenges, and future prospects. *Journal of Microbiological Methods*, *232*, 107125.

Lim, S. S., Phan, T. D., Law, M., Goh, G. S., Moriarty, H. K., Lukies, M. W., ... & Clements, W. (2022). Non-radiologist perception of the use of artificial intelligence (AI) in diagnostic medical imaging reports. *Journal of medical imaging and radiation oncology*, *66*(8), 1029-1034.

Liu, Y., Xu, Y., & Song, R. (2024). Transforming User Experience (UX) through Artificial Intelligence (AI) in interactive media design.

Mannuru, N. R., Shahriar, S., Teel, Z. A., Wang, T., Lund, B. D., Tijani, S., ... & Vaidya, P. (2023). Artificial intelligence in developing countries: The impact of generative artificial intelligence (AI) technologies for development. *Information Development*, 02666669231200628.

Monteith, S., Glenn, T., Geddes, J., Whybrow, P. C., Achtyes, E., & Bauer, M. (2022). Expectations for artificial intelligence (AI) in psychiatry. *Current Psychiatry Reports*, *24*(11), 709-721.

Okolie, J. A. (2024). Introduction of machine learning and artificial intelligence in biofuel technology. *Current Opinion in Green and Sustainable Chemistry*, *47*, 100928.

Peng, C., Zhou, X., & Liu, S. (2022). An introduction to artificial intelligence and machine learning for online education. Mobile Networks and Applications, 27(3), 1147-1150.

Rakuasa, H. (2023). Integration of artificial intelligence in geography learning: Challenges and opportunities. *Sinergi International Journal of Education*, *1*(2), 75-83.

Rizvi, A. T., Haleem, A., Bahl, S., & Javaid, M. (2021). Artificial intelligence (AI) and its applications in Indian manufacturing: a review. *Current Advances in Mechanical Engineering: Select Proceedings of ICRAMERD 2020*, 825-835.

Rodrigues, L., Pereira, J., da Silva, A. F., & Ribeiro, H. (2023). The impact of artificial intelligence on audit profession.

Santoni de Sio, F. (2024). Artificial intelligence and the future of work: Mapping the ethical issues. *The Journal of Ethics*, *28*(3), 407-427.

Stahl, B. C., Rodrigues, R., Santiago, N., & Macnish, K. (2022). A European Agency for Artificial Intelligence: Protecting fundamental rights and ethical values. *Computer Law & Security Review*, *45*, 105661.

Su, J., & Zhong, Y. (2022). Artificial Intelligence (AI) in early childhood education: Curriculum design and future directions. *Computers and Education: Artificial Intelligence*, *3*, 100072.

Su, J., Ng, D. T. K., & Chu, S. K. W. (2023). Artificial intelligence (AI) literacy in early childhood education: The challenges and opportunities. *Computers and Education: Artificial Intelligence*, *4*, 100124.

Tjoa, S., Temper, P. K. M., Temper, M., Zanol, J., Wagner, M., & Holzinger, A. (2022, December). AIRMan: An artificial intelligence (AI) risk management system. In *2022 International Conference on Advanced Enterprise Information System (AEIS)* (pp. 72-81). IEEE.

van Leeuwen, K. G., de Rooij, M., Schalekamp, S., van Ginneken, B., & Rutten, M. J. (2022). How does artificial intelligence in radiology improve efficiency and health outcomes?. *Pediatric radiology*, *52*(11), 2087-2093.

Wang, Y., Fu, E. Y., Zhai, X., Yang, C., & Pei, F. (2024). Introduction of artificial intelligence. In Intelligent building fire safety and smart firefighting (pp. 65-97). Cham: Springer Nature Switzerland.

Xia, Q., Chiu, T. K., Lee, M., Sanusi, I. T., Dai, Y., & Chai, C. S. (2022). A self-determination theory (SDT) design approach for inclusive and diverse artificial intelligence (AI) education. Computers & education, 189, 104582.

Xiang, J. (2023). Promoting innovation and practice of introduction to artificial intelligence course with curriculum civics as a handle. *Asian Journal of Education and Social Studies*, *41*(3), 30-36.

Yim, I. H. Y., & Su, J. (2025). Artificial intelligence (AI) learning tools in K-12 education: A scoping review. *Journal of Computers in Education*, *12*(1), 93-131.

Zarei, M., Mamaghani, H. E., Abbasi, A., & Hosseini, M. S. (2024). Application of artificial intelligence in medical education: A review of benefits, challenges, and solutions. *Medicina Clínica Práctica*, *7*(2), 100422.

Zhang, D. (2024). The pathway to curb greenwashing in sustainable growth: The role of artificial intelligence. *Energy Economics*, *133*, 107562.

PENGANTAR KECERDASAN BUATAN (AI)

Dr. Ir. Agus Wibowo, M.Kom, M.Si, MM

BIO DATA PENULIS

Penulis memiliki berbagai disiplin ilmu yang diperoleh dari Universitas Diponegoro (UNDIP) Semarang. dan dari Universitas Kristen Satya Wacana (UKSW) Salatiga. Disiplin ilmu itu antara lain teknik elektro, komputer, manajemen dan ilmu sosiologi. Penulis memiliki pengalaman kerja pada industri elektronik dan sertifikasi keahlian dalam bidang Jaringan Internet, Telekomunikasi, Artificial Intelligence, Internet Of Things (IoT), Augmented Reality (AR), Technopreneurship, Internet Marketing dan bidang pengolahan dan analisa data (komputer statistik).

Penulis adalah pendiri dari Universitas Sains dan Teknologi Komputer (Universitas STEKOM) dan juga seorang dosen yang memiliki Jabatan Fungsional Akademik Lektor Kepala (Associate Professor) yang telah menghasilkan puluhan Buku Ajar ber ISBN, HAKI dari beberapa karya cipta dan Hak Paten pada produk IPTEK. Sejak tahun 2023 penulis tercatat sebagai Dosen luar biasa di Fakultas Ekonomi & Bisnis (FEB) Universitas Diponegoro Semarang. Penulis juga terlibat dalam berbagai organisasi profesi dan industri yang terkait dengan dunia usaha dan industri, khususnya dalam pengembangan sumber daya manusia yang unggul untuk memenuhi kebutuhan dunia kerja secara nyata.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-634-7227-28-7 (PDF)
9 786347 227287