Stuart Russell, Peter Norvig

Künstliche Intelligenz

Ein moderner Ansatz

2. Auflage

Mit Beiträgen von:
John F. Canny
Douglas D. Edwards
Jitendra M. Malik
Sebastian Thrun

ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Kapitel

3 Problemlösung durch Suchen

In diesem Kapitel erfahren Sie, wie ein Agent eine Aktionsfolge ermittelt, mit der er an seine Ziele gelangt, wenn eine einzelne Aktion dafür nicht ausreichend ist.

Die einfachsten in Kapitel 2 beschriebenen Agenten waren die Reflex-Agenten, die ihre Aktionen auf einer direkten Abbildung von Zuständen auf Aktionen basieren lassen. Solche Agenten verhalten sich nicht zufrieden stellend in Umgebungen, für die diese Abbildung zu groß ist, um gespeichert zu werden, und für die es zu lange dauern würde, um sie zu erlernen. Zielbasierte Agenten dagegen können erfolgreich handeln, indem sie zukünftige Aktionen berücksichtigen und dabei in Betracht ziehen, wie wünschenswert ihre Ergebnisse sind.

Dieses Kapitel beschreibt eine Variante eines zielbasierten Agenten, den so genannten **problemlösenden Agenten**. Problemlösende Agenten entscheiden, was zu tun ist, indem sie Aktionsfolgen suchen, die zu wünschenswerten Zuständen führen. Wir beginnen mit einer genauen Definition der Elemente, aus denen sich ein „Problem“ und seine „Lösung“ zusammensetzen, und zeigen mehrere Beispiele, um diese Definitionen zu konkretisieren. Anschließend beschreiben wir mehrere allgemeine Suchalgorithmen, die für die Lösung dieser Probleme verwendet werden können, und vergleichen die Vorteile der einzelnen Algorithmen. Die Algorithmen sind **nicht informiert,** d.h. sie erhalten keine weiteren Informationen über das Problem außer seiner Definition. Kapitel 4 beschäftigt sich mit **informierten** Suchalgorithmen, die Hinweise darauf haben, wo nach Lösungen gesucht werden soll.

Dieses Kapitel verwendet Konzepte aus der Algorithmenanalyse. Leser, die mit den Konzepten der asymptotischen Komplexität (d.h. der $O()$-Notation) und NP-Vollständigkeit nicht vertraut sind, sollten in Anhang A nachlesen.

3.1 Problemlösende Agenten

Intelligente Agenten sollten versuchen, ihr Leistungsmaß zu maximieren. Wie wir in Kapitel 2 erwähnt haben, wird dies manchmal einfacher, wenn der Agent ein **Ziel** annehmen kann und versucht, es zu erreichen. Betrachten wir zuerst, wie und warum ein Agent dies tun sollte.

Stellen Sie sich vor, ein Agent befindet sich in der Innenstadt von Arad in Rumänien und genießt einen Städtereisenurlaub. Das Leistungsmaß des Agenten beinhaltet viele Faktoren: Er will seine Sonnenbräune vertiefen, er will besser Rumänisch lernen, er will sich

alle Sehenswürdigkeiten ansehen, er will das Nachtleben genießen (falls es ein solches gibt), er will einen Kater vermeiden usw. Das Entscheidungsproblem ist komplex und bedingt viele Abwägungen und ein sorgfältiges Studium von Touristenführern. Nehmen wir nun an, der Agent hat ein nicht umtauschbares Flugticket ab Bukarest am nächsten Tag. In diesem Falle ist es sinnvoll, dass der Agent des **Ziel** verfolgt, nach Bukarest zu gelangen. Aktionsfolgen, die dazu führen, dass Bukarest nicht rechtzeitig erreicht werden kann, können ohne weitere Überlegung zurückgewiesen werden, und das Entscheidungsproblem des Agenten wird wesentlich vereinfacht. Ziele helfen, das Verhalten zu organisieren, indem die Etappenziele begrenzt werden, die der Agent zu erreichen versucht. Die **Zielformulierung**, die auf der aktuellen Situation und dem Leistungsmaß des Agenten basiert, ist der erste Schritt zur Problemlösung.

Wir betrachten ein Ziel als eine Menge von Zuständen der Welt – und zwar jener Zustände, in denen das Ziel erfüllt wird. Die Aufgabe des Agenten ist es, herauszufinden, welche Aktionsfolge ihn zu einem Zielzustand bringt. Bevor er das tun kann, muss er entscheiden, welche Aktionen und Zustände berücksichtigt werden sollen. Wenn er versucht, Aktionen auf dem Niveau von „Bewege deinen linken Fuß um einen Zentimeter nach vorne“ oder „Dreh das Steuerrad um ein Grad nach links“ zu berücksichtigen, wird er womöglich nie aus der Parklücke herauskommen, ganz zu schweigen von Bukarest, weil auf dieser Detailebene zu viel Unsicherheit in der Welt besteht und er zu viele Schritte für eine Lösung braucht. Die **Problemformulierung** ist der Prozess, zu entscheiden, welche Aktionen und Zustände berücksichtigt werden sollen, wenn man ein bestimmtes Ziel vor Augen hat. Wir werden diesen Prozess später genauer betrachten. Hier wollen wir annehmen, dass der Agent Aktionen auf der Ebene berücksichtigt, von einer Hauptstadt in eine andere zu fahren. Die Zustände, die er berücksichtigt, sind also, sich in einer bestimmten Stadt zu befinden.[1]

Unser Agent hat das Ziel angenommen, nach Bukarest zu fahren, und überlegt, wie er von Arad aus dorthin gelangt. Es gibt drei Straßen, die aus Arad heraus führen, eine nach Sibiu, eine andere nach Timisoara und eine nach Zerind. Keine davon führt zu dem großen Ziel Bukarest. Wenn der Agent also nicht sehr vertraut mit der rumänischen Geografie ist, weiß er nicht, welcher Straße er folgen soll.[2] Mit anderen Worten, der Agent weiß nicht, welche seiner möglichen Aktionen die beste ist, weil er nicht genug über den Zustand weiß, der aus der Ausführung der einzelnen Aktionen resultiert. Wenn der Agent über kein zusätzliches Wissen verfügt, befindet er sich in einer Sackgasse. Am besten ist es hier für ihn, zufällig eine der Aktionen auszuwählen.

---

1 Beachten Sie, dass jeder dieser „Zustände“ einer großen Zustandsmenge der Welt entspricht, weil ein Zustand der realen Welt jeden Aspekt der Realität spezifiziert. Es ist wichtig, diese Unterscheidung zwischen Zuständen bei der Problemlösung und Zuständen der Welt zu beachten.

2 Wir gehen davon aus, dass es den meisten Lesern nicht anders geht, und Sie können sich selbst vorstellen, wie ratlos unser Agent ist. Wir entschuldigen uns bei den rumänischen Lesern, die diesem pädagogischen Trick nicht folgen können.

Nehmen wir jedoch an, der Agent besitzt eine Straßenkarte von Rumänien, entweder auf Papier oder in seinem Gedächtnis. Der Sinn einer Straßenkarte ist es, den Agenten mit Informationen über die Zustände zu versorgen, in die er geraten könnte, sowie über die Aktionen, die er ausführen kann. Der Agent kann diese Informationen nutzen, um zukünftige Phasen einer hypothetischen Reise durch jede der drei Städte in Betracht zu ziehen, um eine Route zu finden, die irgendwann nach Bukarest führt. Nachdem er auf der Karte einen Pfad von Arad nach Bukarest gefunden hat, kann er sein Ziel erreichen, indem er die Fahraktionen ausführt, die den Schritten der Reise entsprechen. *Im Allgemeinen kann ein Agent mit mehreren unmittelbaren Aktionen unbekannten Werts entscheiden, was zu tun ist, indem er zuerst verschiedene mögliche Aktionsfolgen auswertet, die zu Zuständen mit bekanntem Wert führen, und dann die beste Folge ausführt.*

Dieser Prozess, eine solche Folge zu ermitteln, wird auch als **Suche** bezeichnet. Ein Suchalgorithmus nimmt ein Problem als Eingabe entgegen und gibt eine **Lösung** in Form einer Aktionsfolge zurück. Nachdem eine Lösung gefunden wurde, können die Aktionen ausgeführt werden, die sie empfiehlt. Man spricht auch von der **Ausführungsphase**. Wir haben also einen einfachen „Formulieren-Suchen-Ausführen"-Entwurf für den Agenten, wie in Abbildung 3.1 gezeigt. Nach der Formulierung eines Ziels und eines zu lösenden Problems ruft der Agent eine Suchprozedur auf, um es zu lösen. Anschließend verwendet er die Lösung, um seine Aktionen auszuführen und als Nächstes das zu tun, was die Lösung empfiehlt – in der Regel die erste Aktion der Folge –, und diesen Schritt dann aus der Folge zu entfernen. Nachdem die Lösung ausgeführt wurde, formuliert der Agent ein neues Ziel.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns eine Aktion
   inputs: percept, eine Wahrnehmnung
   static: seq, eine Aktionsfolge, anfänglich leer
           state, eine Beschreibung des gegenwärtigen Zustands der Welt
           goal, ein Ziel, anfänglich Null
           problem, eine Problemformulierung

state ← UPDATE-STATE(state, percept)
if seq ist leer then do
   goal ← FORMULATE-GOAL(state)
   problem ← FORMULATE-PROBLEM(state, goal)
   seq ← SEARCH(problem)
action ← FIRST(seq)
seq ← REST(seq)
return action
```

Abbildung 3.1: Ein einfacher problemlösender Agent. Als Erstes formuliert er ein Ziel und ein Problem, sucht nach einer Aktionsfolge, die das Problem lösen würde, und führt dann die Aktionen nacheinander aus. Nachdem er damit fertig ist, formulierte ein weiteres Ziel und beginnt von neuem. Beachten Sie, dass er bei der Ausführung der Aktionsfolge seine Wahrnehmungen ignoriert: Er geht davon aus, dass die von ihm gefundene Lösung immer funktioniert.

Zuerst beschreiben wir den Prozess der Problemformulierungen und widmen dann einen Großteil des Kapitels verschiedenen Algorithmen für die SEARCH-Funktion. Wir werden in diesem Kapitel nicht weiter auf die Arbeitsweise der Funktionen UPDATE-STATE und FORMULATE-GOAL eingehen.

Bevor wir ins Detail gehen, wollen wir einen kurzen Exkurs unternehmen, um zu überprüfen, ob problemlösende Agenten in die Diskussion von Agenten und Umgebungen aus Kapitel 2 passen. Der Agentenentwurf in Abbildung 3.1 geht davon aus, dass die Umgebung **statisch** ist, weil die Formulierung und die Lösung des Problems erfolgen, ohne dass Änderungen berücksichtigt werden, die möglicherweise in der Umgebung stattfinden. Der Agentenentwurf geht weiterhin davon aus, dass der Ausgangszustand bekannt ist; ihn zu kennen, ist am einfachsten, wenn die Umgebung **beobachtbar** ist. Die Idee, „alternative Aktionsfolgen" aufzulisten, geht davon aus, dass die Umgebung als **diskret** betrachtet werden kann. Schließlich, und was gleichzeitig auch am wichtigsten ist, geht der Agentenentwurf davon aus, dass die Umgebung **deterministisch** ist. Lösungen für Probleme sind einzelne Aktionsfolgen, die also nicht auf unerwartete Ereignisse reagieren können; darüber hinaus werden die Lösungen ausgeführt, ohne dass die Wahrnehmungen berücksichtigt werden! Ein Agent, der seine Pläne bei geschlossenen Augen ausführt, muss sich schon sehr sicher sein, was passiert. (Steuerungstheoretiker bezeichnen dies auch als **Open-Loop-System**, weil das Ignorieren der Wahrnehmungen die Schleife zwischen dem Agenten und der Umgebung unterbricht.) Alle diese Annahmen bedeuten, dass wir es mit der einfachsten Art von Umgebungen zu tun haben, was ein Grund dafür ist, warum dieses Kapitel so weit vorne im Buch zu finden ist. Abschnitt 3.6 betrachtet kurz, was passiert, wenn wir die Voraussetzungen der Beobachtbarkeit und des Determinismus lockern. Kapitel 12 und 17 bieten genauere Informationen.

3.1.1 Wohldefinierte Probleme und Lösungen

Ein **Problem** kann formal durch vier Komponenten definiert werden:

- dem **Ausgangszustand**, in dem der Agent beginnt. Der Ausgangszustand für unseren Agenten in Rumänien könnte beispielsweise als *In*(*Arad*) beschrieben werden.
- einer Beschreibung der möglichen **Aktionen**, die dem Agenten zur Verfügung stehen. Die gebräuchlichste Formulierung[3] verwendet eine **Nachfolgerfunktion**. Für einen gegebenen Zustand *x* gibt die Funktion SUCCESSOR-FN(*x*) eine Menge sortierter <*Aktion, Nachfolger*>-Paare zurück, wobei jede Aktion eine der erlaubten Aktionen im Zustand *x* ist und jeder Nachfolger ein Zustand, der von *x* aus erreicht werden kann, indem man die Aktion anwendet. Aus dem Zustand *In*(*Arad*) beispielsweise würde die Nachfolgerfunktion für das Rumänien-Problem Folgendes zurückgeben:

  {<*Go*(*Sibiu*), *In*(*Sibiu*)>, <*Go*(*Timisoara*), *In*(*Timisoara*)>, <*Go*(*Zerind*), *In*(*Zerind*)>}

  Zusammen definieren der Ausgangszustand und die Nachfolgerfunktion implizit den **Zustandsraum** des Problems – die Menge aller Zustände, die von diesem Ausgangszustand erreichbar sind. Der Zustandsraum bildet einen Graphen, in dem die Knoten Zustände darstellen und die Pfeile zwischen den Knoten die Aktionen. (Die in Abbil-

[3] Eine alternative Formulierung verwendet eine Menge von **Operatoren**, die auf einen Zustand angewendet werden können, um Nachfolger zu erzeugen.

dung 3.2 gezeigte Landkarte von Rumänien kann als Zustandsraumgraph interpretiert werden, wenn wir annehmen, dass jede Straße für zwei Fahraktionen steht – eine in jede Richtung.) Ein **Pfad** in dem Zustandsraum ist eine Abfolge von Zuständen, die durch eine Aktionsfolge verbunden sind.

- dem **Zieltest**, der entscheidet, ob ein bestimmter Zustand ein Zielzustand ist. Manchmal gibt es ein explizite Menge möglicher Zielzustände, und der Test überprüft einfach, ob der gegebene Zustand einer davon ist. Das Ziel des Agenten in Rumänien ist die aus einem Element bestehende Menge {*In*(*Bucharest*)}. Manchmal wird das Ziel als abstrakte Eigenschaft und nicht als explizite Auflistung einer Menge von Zuständen spezifiziert. Im Schach beispielsweise ist das Ziel, einen Zustand namens „Schach matt“ zu erreichen, das bedeutet, der König des Gegners wird angegriffen und kann sich nicht wehren.
- einer **Pfadkosten**funktion, die jedem Pfad einen numerischen Kostenwert zuweist. Der problemlösende Agent wählt eine Kostenfunktion, die sein eigenes Leistungsmaß reflektiert. Für den Agenten, der versucht, nach Bukarest zu gelangen, spielt hauptsächlich die Zeit eine Rolle; deshalb könnten die Kosten für einen Pfad in seiner Länge in Kilometern angegeben werden. In diesem Kapitel gehen wir davon aus, dass die Kosten eines Pfades als die Summe der Kosten der einzelnen Aktionen entlang des Pfades beschrieben werden können. Die **Schrittkosten** für die Durchführung einer Aktion *a*, um vom Zustand *x* in den Zustand *y* zu gelangen, wird als *c*(*x*, *a*, *y*) ausgedrückt. Die Schrittkosten für Rumänien sind in Abbildung 3.2 als Entfernungen dargestellt. Wir gehen davon aus, dass Schrittkosten nicht negativ sind.[4]

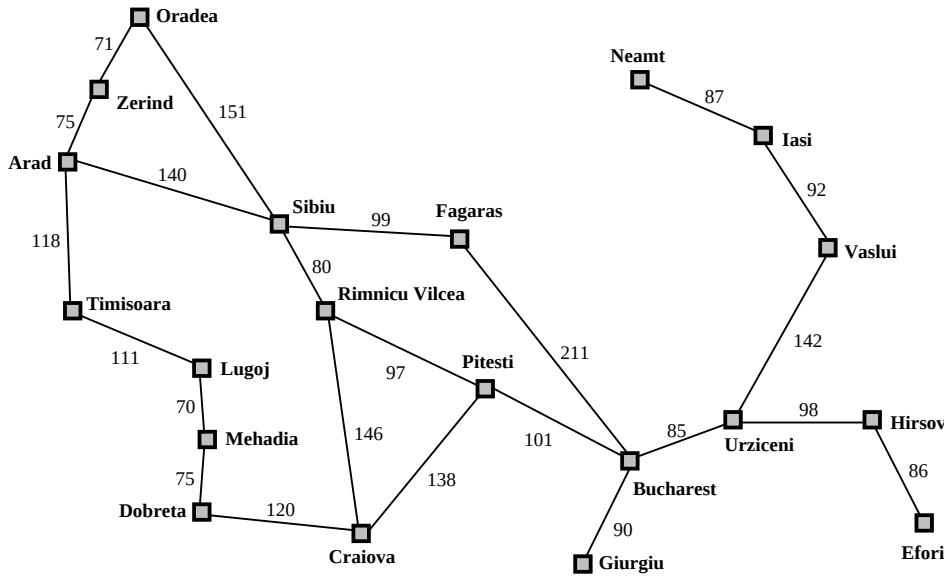

Abbildung 3.2: Eine vereinfachte Straßenkarte eines Teils von Rumänien

[4] Die Auswirkungen negativer Kosten werden in Übung 3.17 genauer betrachtet.

Die oben beschriebenen Elemente definieren ein Problem und können in einer einzelnen Datenstruktur zusammengefasst werden, die einem Problemlösungs-Algorithmus als Eingabe übergeben wird. Eine **Lösung** für ein Problem ist ein Pfad vom Ausgangszustand zum Zielzustand. Die Lösungsqualität wird anhand der Pfadkostenfunktion bewertet, und eine **optimale Lösung** hat die geringsten Pfadkosten aller Lösungen.

3.1.2 Probleme formulieren

Im obigen Abschnitt haben wir die Formulierung des Problems vorgeschlagen, nach Bukarest zu gelangen, indem wir den Ausgangszustand, eine Nachfolgerfunktion, einen Zieltest sowie Pfadkosten angegeben haben. Diese Formulierung erscheint vernünftig, aber viele Aspekte der realen Welt werden darin nicht berücksichtigt. Vergleichen Sie die einfache Zustandsbeschreibung, die wir gewählt haben, *In*(*Arad*), mit einer tatsächlichen Reise durch das Land, wobei der Zustand der Welt so viele weitere Kleinigkeiten beinhaltet: die Reisebegleiter, die Musik im Radio, die Bilder, die man durch das Fenster sieht, die etwaige Anwesenheit von Verkehrspolizisten, die Entfernung zum nächsten Rastplatz, den Straßenzustand, das Wetter usw. All diese Betrachtungen werden in unseren Zustandsbeschreibungen nicht berücksichtigt, weil sie irrelevant für das Problem sind, einen Weg nach Bukarest zu finden. Der Prozess, Details aus einer Repräsentation zu entfernen, wird als **Abstraktion** bezeichnet.

Neben der Abstrahierung der Zustandsbeschreibung müssen wir auch die eigentlichen Aktionen abstrahieren. Eine Fahraktion hat viele Wirkungen. Sie ändert nicht nur die Position des Fahrzeugs und seiner Insassen, sondern benötigt Zeit, verbraucht Treibstoff, verursacht Luftverschmutzung und verändert den Agenten (man sagt ja auch, Reisen bildet). In unserer Formulierung berücksichtigen wir nur die Positionsänderung. Außerdem gibt es viele Aktionen, die wir völlig weglassen: das Einschalten des Radios, das Aus-dem-Fenster-Sehen, das Langsamer-Fahren-weil-eine-Polizeistreife-hinter-uns-ist usw., und natürlich geben wir keine Aktionen an, die sich auf dem Niveau „Steuerrad um 3 Grad nach links drehen“ befinden.

Können wir bei der Definition des geeigneten Abstraktionsgrades präziser sein? Stellen Sie sich die abstrakten Zustände und Aktionen vor, die wir als Entsprechung zu großen Mengen detaillierter Weltzustände und detaillierter Aktionsfolgen gewählt haben. Betrachten wir jetzt eine Lösung für das abstrakte Problem: beispielsweise den Pfad von Arad nach Sibiu nach Rimnicu Vilcea nach Pitesti nach Bukarest. Diese abstrakte Lösung entspricht einer großen Menge detaillierter Pfade. Beispielsweise könnten wir auf der Fahrt zwischen Sibiu und Rimnicu Vilcea das Radio eingeschaltet haben und es für die restliche Reise ausschalten. Die Abstraktion ist *gültig*, wenn wir jede abstrakte Lösung zu einer Lösung der detaillierteren Welt expandieren können; eine ausreichende Bedingung ist, dass es für jeden detaillierten Zustand, zum Beispiel „in Arad“, einen detaillierten Pfad zu einem Zustand gibt, zum Beispiel „in Sibiu“ usw. Die Abstraktion ist sinnvoll, wenn die Ausführung jeder der Aktionen in der Lösung einfacher als das ursprüngliche Problem ist; in diesem Fall sind sie einfach genug, dass sie ohne weitere Suche oder Planung durch einen durchschnittlichen Fahragenten ausgeführt werden können. Die Aus-

wahl einer guten Abstraktion bedingt also, dass so viele Details wie möglich entfernt werden, dennoch die Gültigkeit beibehalten bleibt und sichergestellt ist, dass die abstrakten Aktionen einfach auszuführen sind. Ginge es nicht um die Fähigkeit, sinnvollere Abstraktion zu erzeugen, würden intelligente Agenten von der realen Welt völlig abgedrängt.

3.2 Beispielprobleme

Der problemlösende Ansatz wurde auf einen immensen Bereich von Aufgabenumgebungen angewendet. Wir werden hier einige der bekanntesten Beispiele auflisten und dabei zwischen *Spielproblemen* und *Problemen der realen Welt* unterscheiden. Anhand von **Spielproblemen** sollen verschiedenen Methoden zur Problemlösung demonstriert werden. Sie können in einer präzisen, exakten Beschreibung angegeben werden. Das bedeutet, sie können ganz einfach von den Forschern genutzt werden, um die Leistung von Algorithmen zu vergleichen. Bei einem **Realweltproblem** machen sich die Leute um seine Lösung wirklich Gedanken. Sie haben normalerweise nicht eine einzige Beschreibung, auf die sich alle geeinigt haben, aber wir versuchen, das allgemeine Aussehen ihrer Formulierungen aufzuzeigen.

3.2.1 Spielprobleme

Das erste Beispiel, das wir hier betrachten wollen, ist die in Kapitel 2 eingeführte Staubsaugerwelt (siehe Abbildung 2.2). Sie kann wie folgt als Problem formuliert werden:

- **Zustände:** Der Agent befindet sich an einer von zwei Positionen, die jeweils schmutzig sein können, aber nicht müssen. Es gibt also $2 \times 2^2 = 8$ mögliche Weltzustände.
- **Ausgangszustand:** Jeder beliebige Zustand kann als Ausgangszustand festgelegt werden.
- **Nachfolgerfunktion:** Diese Funktion erzeugt die erlaubten Zustände, die aus der Durchführung der drei Aktionen (*Links*, *Rechts* und *Saugen*) resultieren. Abbildung 3.3 zeigt den vollständigen Zustandsraum.
- **Zieltest:** Überprüft, ob alle Quadrate sauber sind.
- **Pfadkosten:** Jeder Schritt kostet 1, die gesamten Pfadkosten entsprechen also der Anzahl der Schritte im Pfad.

Verglichen mit der realen Welt hat dieses Spielproblem diskrete Positionen, diskreten Schmutz, zuverlässige Reinigung – und es wird nie wieder schmutzig, nachdem geputzt wurde. (In Abschnitt 3.6 werden wir diese Vorgaben lockern.) Ein wichtiger Aspekt ist, dass der Zustand sowohl von der Agentenposition als auch von den Schmutzposition bestimmt wird. Eine größere Umgebungen mit n Positionen hat $n2^n$ Zustände.

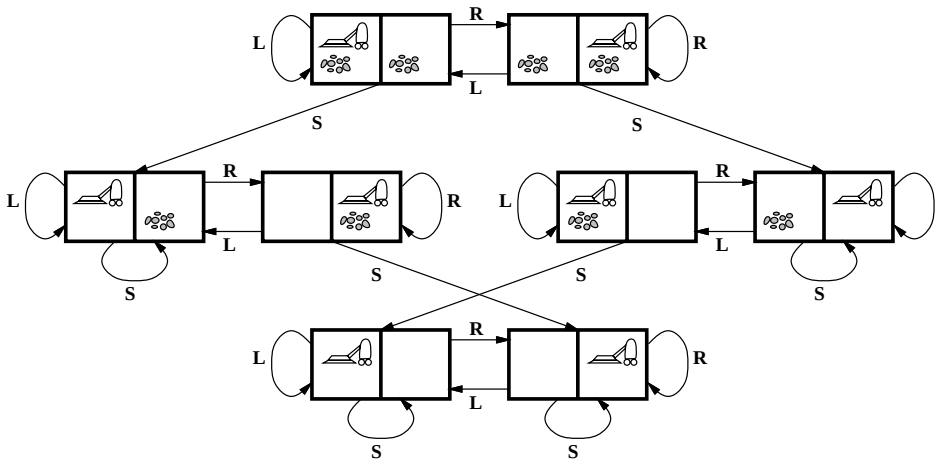

Abbildung 3.3: Der Zustandsraum für die Staubsaugerwelt. Pfeile stehen für Aktionen: L = *Links*, R = *Rechts*, S = *Saugen*.

Das 8-Puzzle, für das Sie in Abbildung 3.4 ein Beispiel sehen, besteht aus einem Spielbrett von 3×3 mit 8 nummerierten Feldern und einem leeren Raum. Ein Feld, das sich neben dem leeren Feld befindet, kann dorthin verschoben werden. Das Ziel dabei ist, einen vorgegebenen Zielzustand zu erreichen, wie beispielsweise rechts in der Abbildung gezeigt. Die Standardformulierung sieht wie folgt aus:

- **Zustände**: Eine Zustandsbeschreibung gibt die Position jedes der acht Felder und des Lehrfeldes in einem der neun Quadrate an.
- **Ausgangszustand**: Jeder Zustand kann als Ausgangszustand festgelegt werden. Beachten Sie, dass jedes beliebige Ziel von genau der Hälfte der möglichen Ausgangszustände erreicht werden kann (Übung 3.4).
- **Nachfolgerfunktion**: Erzeugt die legalen Zustände, die durch Ausführung der vier Aktionen entstehen (die Züge *Links*, *Rechts*, *Oben* oder *Unten*).
- **Zieltest**: Überprüft, ob der Zustand mit der in Abbildung 3.4 gezeigten Zielkonfiguration übereinstimmt. (Es sind auch andere Zielkonfigurationen möglich.)
- **Pfadkosten**: Jeder Schritt kostet 1, die Gesamtpfadkosten entsprechen also der Anzahl der Schritte im Pfad.

Welche Abstraktionen haben wir hier angewendet? Die Aktionen werden auf ihre Anfangs- und Endzustände abstrahiert, wobei die Zwischenpositionen ignoriert werden, während das Feld verschoben wird. Wir haben Aktionen wegabstrahiert wie beispielsweise, dass das Brett geschüttelt wird, wenn sich die Felder verklemmen, oder den Ausbau der Felder mit Hilfe eines Messers und den anschließenden Wiedereinbau. Wir haben also eine Beschreibung der Regeln des Puzzles – und haben alle Details der physischen Bewegung weggelassen.

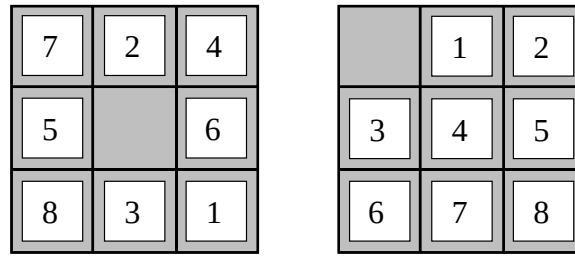

Abbildung 3.4: Ein typisches Beispiel für ein 8-Puzzle

Das 8-Puzzle gehört zur Familie der **Schiebeblock-Puzzles**, die häufig als Testprobleme für neue Suchalgorithmen in der künstlichen Intelligenz verwendet werden. Diese allgemeine Klasse ist NP-vollständig; man erwartet also nicht, Methoden zu finden, die im schlechtesten Fall wesentlich besser sind als die in diesem und dem nächsten Kapitel beschriebenen Suchalgorithmen. Das 8-Puzzle hat 9!/2 = 181.440 erreichbar Zustände und ist einfach zu lösen. Das 15-Puzzle (auf einem Spielbrett mit 4×4 Feldern) hat etwa 1,3 Milliarden Zustände, und beliebige Beispiele können durch die besten Suchalgorithmen in ein paar Millisekunden optimal gelöst werden. Das 24-Puzzle (auf einem Spielbrett mit 5×5 Feldern) hat etwa 10^{25} Zustände, und es ist immer noch schwierig, beliebige Beispiele mit den aktuellen Maschinen und Algorithmen optimal zu lösen.

Des Ziel des **8-Damen-Problems** ist es, acht Damen so auf einem Schachbrett zu platzieren, dass keine der Damen eine andere angreift. (Eine Dame greift jede Figur in derselben Zeile, Spalte oder Diagonalen an.) Abbildung 3.5 zeigt einen Lösungsversuch, der fehlschlägt: Die Dame in der ganz rechten Spalte wird durch die Damen ganz oben links angegriffen.

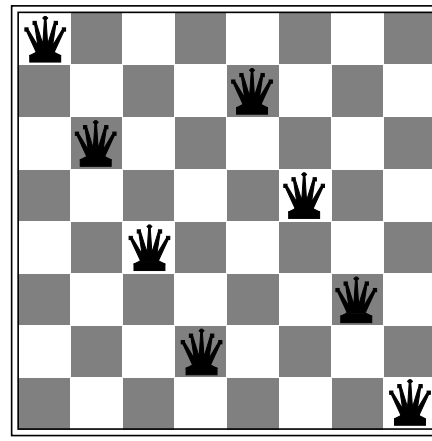

Abbildung 3.5: Versuch einer Lösung für das 8-Damen-Problem.
Sie schlägt fehl. Die Lösung bleibt Ihnen als Übung überlassen.

Obwohl es für dieses Problem effiziente spezielle Algorithmen gibt, ebenso wie für die gesamte *n*-Damen-Familie, bleibt es ein interessantes Testproblem für Suchalgorithmen. Es gibt hauptsächlich zwei Arten der Formulierung: Eine **inkrementelle Formulierung** verwendet Operatoren, die die Zustandsbeschreibung *erweitern*, beginnend mit einem leeren Zustand; für das 8-Damen-Problem bedeutet das, dass jede Aktion dem Zustand eine Dame hinzufügt. Eine **vollständige Zustandsformulierung** beginnt mit allen acht Damen auf dem Brett und verschiebt sie dort. In jedem Fall sind die Pfadkosten nicht von Interesse, weil nur der endgültige Zustand zählt. Die erste inkrementelle Formulierung, die man ausprobieren könnte, sieht wie folgt aus:

- **Zustände**: Eine beliebige Anordnung von 0 bis 8 Damen auf dem Brett ist ein Zustand.
- **Ausgangszustand**: Keine Dame auf dem Brett.
- **Nachfolgerfunktion**: Einem beliebigen Quadrat eine Dame hinzufügen.
- **Zieltest**: Es befinden sich acht Damen auf dem Brett und keine davon ist angegriffen.

Bei dieser Formulierung müssen wir $64 \cdot 63 \cdot \cdot \cdot 57 \approx 3 \times 10^{14}$ mögliche Abfolgen auswerten. Eine bessere Formulierung wäre, zu verbieten, eine Dame auf einem Quadrat zu platzieren, das bereits angegriffen ist:

- **Zustände**: Anordnungen von n Damen ($0 \leq n \leq 8$), eine pro Spalte in den am weitesten links liegenden n Spalten, wobei keine Dame eine andere angreift, sind Zustände.
- **Nachfolgerfunktion**: Einem beliebigen Quadrat in der am weitesten links liegenden leeren Spalte wird eine Dame hinzugefügt, so dass diese nicht durch eine andere Dame angegriffen wird.

Diese Formulierung reduziert den 8-Damen-Zustandsraum von 3×10^{14} auf nur noch 2057, und es ist einfach, Lösungen zu finden. Für 100 Damen dagegen hat die Ausgangsformulierung etwa 10^{400} Zustände, während die verbesserte Formulierung etwa 10^{52} Zustände aufweist (Übung 3.5). Dabei handelt es sich um eine enorme Reduzierung, aber selbst der verbesserte Zustandsraum ist noch zu groß für die Algorithmen in diesem Kapitel, um ihn bewältigen zu können. Kapitel 4 beschreibt die vollständige Zustandsformulierung, und Kapitel 5 stellt einen einfachen Algorithmus vor, der selbst für das Millionen-Damen-Problem eine einfache Lösung erzeugt.

3.2.2 Probleme aus der realen Welt

Wir haben bereits gesehen, wie das **Routensuche-Problem** im Hinblick auf die vorgegebenen Positionen und die Übergänge entlang von Verbindungen zwischen ihnen definiert ist. Algorithmen zur Routensuche werden in zahlreichen Anwendungen eingesetzt, wie beispielsweise beim Routing in Computernetzwerken, bei der Planung militärischer Operationen oder in Planungssystemen für den Flugreiseverkehr. Diese Probleme sind in der Regel komplex zu spezifizieren. Betrachten Sie ein vereinfachtes Beispiel eines Flugverkehrsproblems, das wie folgt spezifiziert ist:

- **Zustände**: Jeder Zustand wird durch eine Position (d.h. einen Flughafen) und die aktuelle Zeit repräsentiert.
- **Ausgangszustand**: Wird durch das Problem spezifiziert.
- **Nachfolgerfunktion**: Gibt den Zustand zurück, der durch die Buchung eines planmäßigen Fluges entsteht (vielleicht weiter spezifiziert durch Sitzklasse und Position), der später als zur aktuellen Zeit plus der Transitzeiten innerhalb des Flughafens vom aktuellen Flughafen zu einem anderen abfliegt.
- **Zieltest**: Gelangen wir innerhalb einer vorgegebenen Zeit ans Ziel?
- **Pfadkosten**: Sind vom Flugpreis, der Wartezeit, der Flugzeit, Zoll- und Einwanderungsprozeduren, Sitzqualität, Tageszeit, Flugzeugtyp, Vielflieger-Meilenbonus usw. abhängig.

Kommerzielle Flugbuchungssysteme verwenden eine Problemformulierung dieser Art – mit vielen zusätzlichen Komplikationen, um die byzantinischen Verkehrsstrukturen zu bewältigen, die die Fluggesellschaften erzeugen. Jeder erfahrene Reisende weiß jedoch, dass nicht alle Flüge planmäßig verlaufen. Ein wirklich gutes System sollte Alternativpläne berücksichtigen – wie beispielsweise Sicherungsreservierungen auf alternativen Flügen –, und zwar in einem Maße, dass diese nach Kosten und der Ausfallwahrscheinlichkeit des ursprünglichen Fluges zu rechtfertigen sind.

Touring-Probleme sind eng verwandt mit Problemen der Routensuche, allerdings mit einem wichtigen Unterschied. Betrachten Sie beispielsweise das Problem „Besuche jede in Abbildung 3.2 gezeigte Stadt mindestens einmal und beginne und ende dabei in Bukarest“. Wie bei der Routensuche entsprechen die Aktionen den Reisen zwischen den benachbarten Städten. Der Zustandsraum unterscheidet sich jedoch erheblich. Jeder Zustand muss nicht nur die aktuelle Position beinhalten, sondern auch *die Menge der Städte, die der Agent besucht hat*. Der Ausgangszustand wäre also „in Bukarest; besucht {Bukarest}“, und ein typischer Zwischenzustand wäre „in Vaslui; besucht {Bukarest, Urziceni,Vaslui}“, und der Zieltest würde überprüfen, ob sich der Agent in Bukarest befindet und alle 20 Städte besucht wurden.

Das Problem des **Handelsreisenden** (**Traveling Salesman Problem**, TSP) ist ein Touring-Problem, wobei jede Stadt genau einmal besucht werden muss. Das Ziel dabei ist, die *kürzeste* Tour zu finden. Das Problem ist als NP-hart bekannt, aber es wurde ein enormer Aufwand betrieben, die Fähigkeiten von TSP-Algorithmen zu verbessern. Neben der Planung von Reisen für Handelsreisende wurden diese Algorithmen auch für andere Aufgaben verwendet, wie beispielsweise Bewegungen automatischer Platinenbestücker oder Verkaufsautomaten in Kaufhäusern.

Ein **VLSI-Layout** bedingt die Positionierung von Millionen von Komponenten und Verbindungen auf einem Chip, um Fläche, Schaltverzögerungen und Streukapazitäten zu minimieren und den Herstellungsgewinn zu maximieren. Das Layoutprogramm kommt nach der logischen Entwurfsphase und ist normalerweise in zwei Teile unterteilt: **Zellen-Layout** und **Kanal-Routing**. Bei einem Zellen-Layout werden die elementaren Komponenten der Schaltung in Zellen gruppiert, die jeweils eine bestimmte Funktion ausführen. Jede Zelle hat einen feststehenden Platzbedarf (Größe und Umriss) und benötigt eine

bestimmte Anzahl an Verbindungen zu allen anderen Zellen. Das Ziel dabei ist, die Zellen so auf dem Chip zu platzieren, dass sie sich nicht überlappen und dass ausreichend viel Platz für die verbindenden Drähte ist, die zwischen den Zellen angeordnet werden. Das Kanal-Routing ermittelt eine bestimmte Route für jeden Draht zwischen den Zellen. Diese Suchprobleme sind extrem komplex, aber es ist durchaus wünschenswert, sie zu lösen. In Kapitel 4 lernen Sie einige Algorithmen kennen, die eine Lösung erzeugen.

Die **Roboternavigation** ist eine Verallgemeinerung des Problems der Routensuche, das wir zuvor beschrieben haben. Statt einer diskreten Menge von Routen kann sich ein Roboter im stetigen Raum mit (im Prinzip) einer unendlichen Menge möglicher Aktionen und Zustände bewegen. Für einen rollenden Roboter, der sich auf einer flachen Oberfläche bewegt, ist der Raum im Wesentlichen zweidimensional. Wenn der Roboter Arme und Beine oder Räder hat, wird der Suchraum mehrdimensional. Man benötigt fortgeschrittene Techniken, um den Suchraum endlich zu machen. Wir werden einige dieser Methoden in Kapitel 25 betrachten. Neben der Komplexität des Problems müssen reale Roboter auch mit Fehlern der Sensoreneingabe und der Motorsteuerung zurechtkommen.

Die **automatische Bauabfolge** komplexer Objekte durch einen Roboter wurde zuerst von FREDDY (Michie, 1972) demonstriert. Seither gab es nur wenig Fortschritte, aber es gibt sie, nämlich im Hinblick darauf, wann der Zusammenbau komplizierter Objekte wie beispielsweise elektrischer Motoren wirtschaftlich durchführbar ist. Bei Problemen des Zusammenbaus von Objekten ist das Ziel, eine Reihenfolge zu finden, in der die Einzelteile zusammengebaut werden. Wird die falsche Reihenfolge gewählt, gibt es später in der Abfolge keine Möglichkeit, ein weiteres Teil einzubauen, ohne bereits erledigte Arbeiten noch einmal rückgängig zu machen. Die Überprüfung eines Schritts in der Abfolge für die Durchführbarkeit ist ein schwieriges geometrisches Suchproblem, das eng mit der Roboternavigation verknüpft ist. Die Ermittlung eines erlaubten Nachfolgers ist also der aufwändigste Teil der Bauabfolge. Jeder praktische Algorithmus muss vermeiden, den gesamten Zustandsraum auszuwerten, und darf nur einen kleinen Teil davon betrachten. Ein weiteres wichtiges Bauproblem ist der **Proteinentwurf**, wobei das Ziel ist, eine Abfolge von Aminosäuren zu finden, die ein dreidimensionales Protein ergeben, das die richtigen Eigenschaften besitzt, um eine Krankheit zu heilen.

In den vergangenen Jahren gab es einen gesteigerten Bedarf an Softwarerobotern, die eine **Internetsuche** durchführen und dabei Fragen auf Antworten, verwandte Informationen oder Einkaufsgelegenheiten suchen. Dies ist eine gute Anwendung von Suchtechniken, weil es einfach ist, das Internet als einen Graphen von Knoten (Seiten) darzustellen, die über Links verknüpft sind. Eine vollständige Beschreibung der Internetsuche finden Sie in Kapitel 10.

3.3 Die Suche nach Lösungen

Nachdem wir einige Probleme formuliert haben, müssen wir sie jetzt lösen. Dazu verwenden wir eine Suche durch den Zustandsraum. Dieses Kapitel beschäftigt sich mit Suchtechniken, die einen expliziten **Suchbaum** verwenden, der durch den Ausgangszustand und die Nachfolgerfunktion erzeugt wird, die in ihrer Kombination den Zustandsraum

definieren. Im Allgemeinen haben wir wahrscheinlich öfter einen Suchgraphen als einem Suchbaum, wenn derselbe Zustand über mehrere Pfade erreicht werden kann. Diese wichtige Komplikation beschreiben wir in Abschnitt 3.5.

Abbildung 3.6 zeigt einige der Expansionen im Suchbaum, um eine Route von Arad nach Bukarest zu finden. Die Wurzel des Suchbaums ist ein **Suchknoten**, der dem Ausgangszustand entspricht, *In*(*Arad*). Im ersten Schritt testen wir, ob dies ein globaler Zustand ist. Offensichtlich ist das nicht der Fall, aber diese Überprüfung ist wichtig, so dass wir Scheinprobleme lösen können, wie beispielsweise „Starte in Arad, gehe nach Arad“. Weil dies kein Zielzustand ist, müssen wir einige weitere Zustände betrachten. Dazu **expandieren** wir den aktuellen Zustand; das bedeutet, wir wenden die Nachfolgerfunktion auf den aktuellen Zustand an und **erzeugen** damit eine neue Menge von Zuständen. In diesem Fall erhalten wir drei neue Zustände: *In*(*Sibiu*), *In*(*Timisoara*) und *In*(*Zerind*). Jetzt müssen wir wählen, welche dieser drei Möglichkeiten weiter betrachtet werden soll.

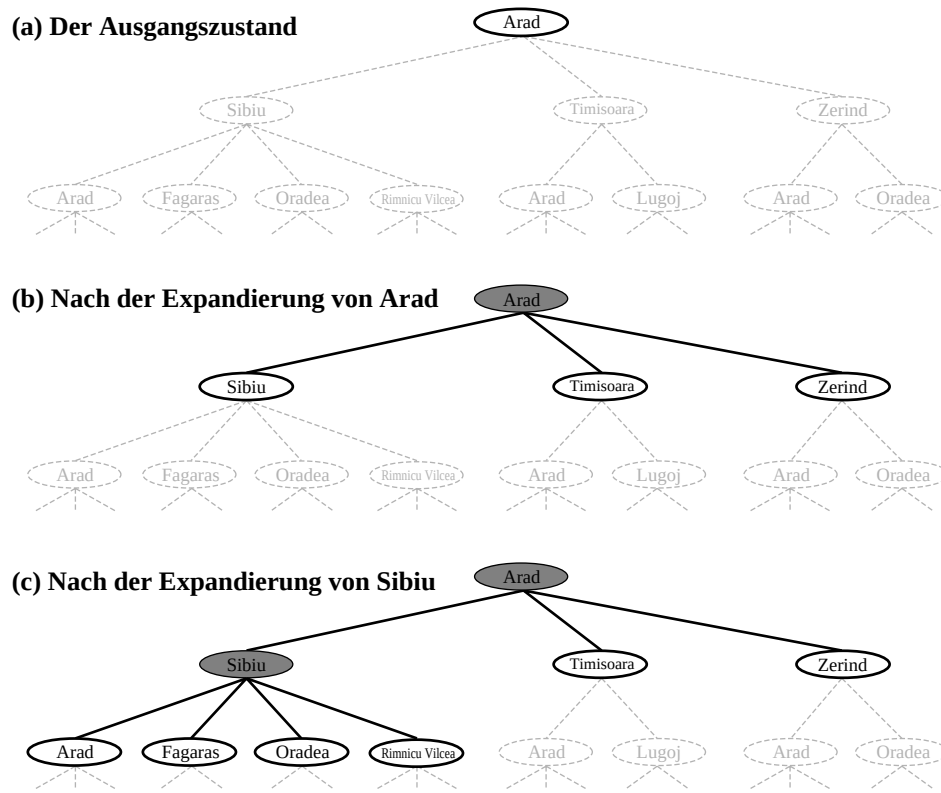

Abbildung 3.6: Partielle Suchbäume für die Ermittlung einer Route von Arad nach Bukarest. Knoten, die expandiert wurden, sind grau schattiert dargestellt; Knoten, die erzeugt, aber noch nicht expandiert wurden, sind fett ausgezeichnet; Knoten, die noch nicht erzeugt wurden, sind mit hellen gestrichelten Linien dargestellt.

Dieses ist die charakteristische Eigenschaft der Suche – es wird jetzt eine Option weiterverfolgt, die anderen werden beiseite gelegt und später betrachtet, falls die erste Suche nicht zu einer Lösung führt. Angenommen, wir verwenden zuerst Sibiu. Wir prüfen, ob es sich dabei um einen Zielzustand handelt (das ist nicht der Fall) und expandieren es dann auf *In*(*Arad*), *In*(*Fagaras*), *In*(*Oradea*) und *In*(*RimnicuVilcea*). Wir können dann einen beliebigen dieser vier Zustände auswählen, oder zurückgehen und Timisoara oder Zerind auswählen. Wir können weiter auswählen, testen und expandieren, bis wir entweder eine Lösung gefunden haben oder bis es keine weiteren Zustände zu expandieren gibt. Die Auswahl, welcher Zustand zu expandieren ist, wird durch die **Suchstrategie** festgelegt. Der allgemeinen Baum-Suchalgorithmus ist in Abbildung 3.7 informell beschrieben.

```
function TREE-SEARCH(problem, strategy) returns eine Lösung oder einen Fehler
    initialisiere den Suchbaum unter Verwendung des Ausgangszustands von problem
     loop do
          if es gibt keine Expansionskandidaten then return Fehler
          wähle einen Knoten zur Expandierung gemäß strategy aus
          if der Knoten enthält einen Zielzustand then return die entsprechende
                Lösung
          else expandiere den Knoten und füge dem Suchbaum resultierende
                Knoten hinzu
```

Abbildung 3.7: Eine informelle Beschreibung des allgemeinen Baum-Suchalgorithmus

Es ist wichtig, zwischen dem Suchraum und dem Suchbaum zu unterscheiden. Für das Problem der Routensuche gibt es nur 20 Zustände im Zustandsraum, einen für jede Stadt. Es gibt jedoch eine unendliche Anzahl an Pfaden in diesem Zustandsraum, so dass der Suchbaum eine unendliche Menge an Knoten aufweist. Beispielsweise sind die drei Pfade Arad–Sibiu, Arad–Sibiu–Arad, Arad–Sibiu–Arad–Sibiu die ersten drei Pfade einer unendlichen Pfadfolge. (Offensichtlich vermeidet ein guter Suchalgorithmus, solchen wiederholten Pfaden zu folgen; Abschnitt 3.5 zeigt, wie das geht.)

Es gibt viele Möglichkeiten, Knoten darzustellen, aber wir gehen davon aus, dass ein Knoten eine Datenstruktur mit fünf Komponenten ist:

- STATE (Zustand): der Zustand in dem Zustandsraum, dem der Knoten entspricht
- PARENT-NODE (Elternknoten): der Knoten im Suchbaum, der diesen Knoten erzeugt hat
- ACTION (Aktion): die Aktion, die auf den übergeordneten Knoten angewendet wurde, um den Knoten zu erzeugen
- PATH-COST (Pfadkosten): die Kosten, traditionell als *g(n)* bezeichnet, des Pfades vom Ausgangszustand zu dem Knoten, wie er durch die Zeiger des übergeordneten Knotens vorgegeben ist
- DEPTH (Tiefe): die Anzahl der Schritte entlang des Pfades vom Ausgangszustand aus

Beachten Sie unbedingt die Unterscheidung zwischen Knoten und Zuständen. Ein Knoten ist eine Datenstruktur für die „Buchhaltung“, mit deren Hilfe der Suchbaum dargestellt wird. Ein Zustand entspricht einer Konfiguration der Welt. Die Knoten befinden sich damit auf bestimmten Pfaden, wie durch die Zeiger des Elternknotens definiert, während

das für Zustände nicht der Fall ist. Darüber hinaus können zwei unterschiedlichen Knoten denselben Weltzustand enthalten, wenn dieser Zustand über zwei verschiedene Suchpfade erzeugt wird. Die Knoten-Datenstruktur ist in Abbildung 3.8 dargestellt.

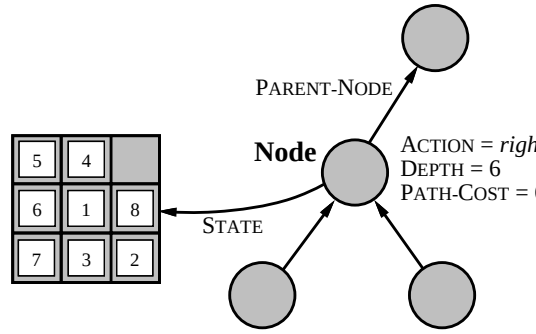

Abbildung 3.8: Knoten sind die Datenstrukturen, aus denen sich der Suchbaum zusammensetzt. Jeder davon hat einen übergeordneten Knoten, einen Zustand sowie verschiedene Felder für die Verwaltung. Pfeile verweisen vom untergeordneten Knoten zum übergeordneten Knoten.

Außerdem müssen wir die Menge der Knoten anzeigen, die erzeugt, aber noch nicht expandiert wurden – diese Menge wird auch als der **Randbereich** (**Fringe**) bezeichnet. Jedes Element dieses Randbereichs ist ein **Blattknoten**, d.h. ein Knoten ohne Nachfolger im Baum. In Abbildung 3.6 besteht der Randbereich jedes Baums aus den Knoten mit fetten Linien. Die einfachste Darstellung des Randbereichs wäre eine Knotenmenge. Die Suchstrategie wäre dann eine Funktion, die den nächsten Knoten, der expandiert werden soll, aus dieser Menge auswählt. Obwohl dies vom Konzept her ganz einfach ist, könnte es rechentechnisch aufwändig sein, weil die Strategiefunktion möglicherweise jedes Element der Menge betrachten muss, um das am besten geeignete auszuwählen. Gehen wir also davon aus, dass die Menge der Knoten als **(Warte-)Schlange (Queue)** implementiert ist. Die Operationen für eine Schlange sehen wie folgt aus:

- MAKE-QUEUE(*element,...*) erzeugt eine Schlange mit den vorgegebenen Elementen.
- EMPTY?(*queue*) gibt nur dann true zurück, wenn in der Schlange keine weiteren Elemente enthalten sind.
- FIRST(*queue*) gibt das erste Element der Schlange zurück.
- REMOVE-FIRST(*queue*) gibt FIRST(*queue*) zurück und entfernt es aus der Schlange.
- INSERT(*element, queue*) fügt ein Element in die Schlange ein und gibt die resultierende Schlange zurück. (Wir werden sehen, dass unterschiedliche Arten von Schlangen die Elemente in unterschiedlichen Reihenfolgen einfügen.)
- INSERT-ALL(*elements, queue*) fügt eine Menge von Elementen in die Schlange ein und gibt die resultierende Schlange zurück.

Mit diesen Definitionen können wir die formalere Version des allgemeinen Baum-Suchalgorithmus schreiben, wie in Abbildung 3.9 gezeigt.

```
function TREE-SEARCH(problem, fringe) returns eine Lösung oder einen Fehler

   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if EMPTY?(fringe) then return Fehler
      node ← REMOVE-FIRST(fringe)
      if GOAL-TEST[problem] angewendet auf STATE[node] erfolgreich
         then return SOLUTION(node)
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns eine Knotenmenge

   successors ← die leere Menge
   for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
      s ← ein neuer Knoten
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      füge s zu successors hinzu
   return successors
```

Abbildung 3.9: Der allgemeine Baum-Suchalgorithmus. (Beachten Sie, dass das Argument *fringe* eine leere Schlange sein muss und dass sich der Typ der Schlange auf die Reihenfolge der Suche auswirkt.) Die Funktion SOLUTION gibt die Folge der Aktionen zurück, die man erhält, indem man die Zeiger der übergeordneten Knoten zurück bis zur Wurzel verfolgt.

3.3.1 Leistungsbewertung für die Problemlösung

Die Ausgabe eines problemlösenden Algorithmus ist entweder ein Versagen oder eine Lösung. (Einige Algorithmen verbleiben möglicherweise in einer Endlosschleife und geben nie eine Ausgabe zurück.) Wir werden die Leistung eines Algorithmus auf viererlei Arten bewerten:

- **Vollständigkeit**: Findet der Algorithmus garantiert eine Lösung, wenn es eine solche gibt?
- **Optimalität**: Findet die Strategie die optimale Lösung, wie in Abschnitt 3.1.1 definiert?
- **Zeitkomplexität**: Wie lange dauert es, eine Lösung zu finden?
- **Speicherkomplexität**: Wie viel Speicher wird für die Suche benötigt?

Zeit- und Speicherkomplexität werden immer im Hinblick auf ein bestimmtes Schwierigkeitsmaß des Problems betrachtet. In der theoretischen Informatik ist das typische Maß die Größe des Zustandsraumgraphen, weil der Graph als explizite Datenstruktur betrachtet wird, die die Eingabe für das Suchprogramm darstellt. (Ein Beispiel dafür ist die Straßenkarte von Rumänien.) In der künstlichen Intelligenz, wo der Graph implizit durch den Ausgangszustand und die Nachfolgerfunktion repräsentiert wird und häufig unendlich ist, wird die Komplexität im Hinblick auf drei quantitative Mengen ausgedrückt: *b*, den **Ver-**

zweigungsfaktor (**branching factor**) oder die maximale Anzahl der Nachfolger jedes Knotens; *d*, die Tiefe (*depth*) des flachsten Knotens und *m*, die maximale Länge eines beliebigen Pfads im Zustandsraum.

Zeit wird häufig im Hinblick auf die Anzahl der während der Suche erzeugten[5] Knoten gemessen, und der Speicherplatz im Hinblick auf die maximale Anzahl im Speicher abgelegter Knoten.

Um die Effektivität eines Suchalgorithmus abschätzen zu können, können wir einfach nur die **Suchkosten** betrachten – die in der Regel von der Zeitkomplexität abhängig sind, wir können aber auch einen Term für die Speicherbenutzung berücksichtigen – oder wir können die **Gesamtkosten** verwenden, die die Suchkosten und die Pfadkosten der gefundenen Lösung kombinieren. Für das Problem, eine Route von Arad nach Bukarest zu finden, sind die Suchkosten die Zeitdauer, die die Suche benötigt, und die Lösungskosten die Gesamtlänge des Pfades in Kilometern. Um also die Gesamtkosten zu berechnen, müssen wir Kilometer und Millisekunden addieren. Es gibt keinen „offiziellen Wechselkurs“ zwischen den beiden, aber es könnte in diesem Fall sinnvoll sein, Kilometer in Millisekunden umzuwandeln, indem man eine Schätzung der Durchschnittsgeschwindigkeit des Autos verwendet (weil sich der Agent hauptsächlich um die Zeit kümmert). Das ermöglicht dem Agenten, eine optimale Abwägung zu finden, ab wann eine weitere Berechnungen zur Ermittlung eines kürzeren Pfades kontraproduktiv wird. Das allgemeinere Problem der Abwägung und zwischen verschiedenen Gütern wird in Kapitel 16 genauer betrachtet.

3.4 Uninformierte Suchstrategien

Dieser Abschnitt deckt die fünf Suchstrategien ab, die wir unter der Überschrift der **uninformierten Suche** (auch als **blinde Suche** bezeichnet) finden. Der Begriff bedeutet, dass sie keine zusätzlichen Informationen über Zustände besitzen außer den in der Problemdefinition vorgegebenen. Alles, was sie tun können, ist, Nachfolger zu erzeugen und einen Zielzustand von einem Nichtzielzustand zu unterscheiden. Strategien, die wissen, ob ein Nichtzielzustand „viel versprechender“ als ein anderer ist, werden auch als **informierte Suchstrategien** oder **heuristische Suchstrategien** bezeichnet. Sie werden in Kapitel 4 genauer beschrieben. Alle Suchstrategien werden nach der *Reihenfolge* unterschieden, in der die Knoten expandiert werden.

3.4.1 Breitensuche (Breadth-first search)

Die **Breitensuche** ist eine einfache Strategie, wobei der Wurzelknoten als Erster expandiert wird, dann alle Nachfolger des Wurzelknotens und dann *deren* Nachfolger usw. Im Allgemeinen werden zuerst alle Knoten einer bestimmten Tiefe im Suchbaum expandiert, bevor Knoten in der nächsten Ebene expandiert werden.

---

[5] In einigen Artikeln wird die Zeit auch im Hinblick auf die Anzahl der Knotenexpandierungen angegeben. Die beiden Messungen unterscheiden sich höchstens um einen Faktor *b*. Wir haben den Eindruck, dass die Ausführungszeit einer Knotenexpandierung mit der Anzahl der in dieser Expandierung erzeugten Knoten steigt.

Die Breitensuche kann durch Aufruf von TREE-SEARCH mit einem leeren Randbereich implementiert werden. Dabei handelt es sich um eine FIFO-Schlange (first-in-first-out), wobei sichergestellt wird, dass die Knoten, die zuerst besucht werden, als Erste expandiert werden. Mit anderen Worten, der Aufruf von TREE-SEARCH (*problem*, FIFO-QUEUE()) erzeugt eine Breitensuche. Die FIFO-Schlange legt alle neu erzeugten Nachfolger am Ende der Schlange ab, d.h. flachere Knoten werden vor tieferen Knoten expandiert. Abbildung 3.10 zeigt den Verlauf einer Suche in einem einfachen Binärbaum.

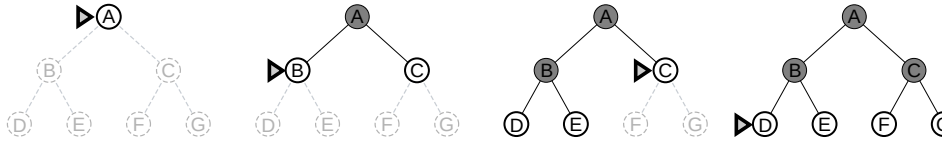

Abbildung 3.10: Breitensuche für einen einfachen Binärbaum. In jeder Phase ist der nächste zu expandierende Knoten durch einen Pfeil gekennzeichnet.

Wir werden die Breitensuche unter Anwendung der vier Kriterien aus dem vorigen Abschnitt bewerten. Es ist leicht ersichtlich, dass sie *vollständig* ist – wenn sich der flachste Zielknoten auf einer endlichen Tiefe d befindet, findet ihn die Breitensuche irgendwann, nachdem alle flacheren Knoten expandiert wurden (vorausgesetzt der Verzweigungsfaktor b ist endlich). Der *flachste* Zielknoten ist nicht unbedingt der *optimale*; technisch betrachtet ist die Breitensuche optimal, wenn die Pfadkosten eine nichtfallende Funktion der Knotentiefe darstellen (wenn beispielsweise alle Aktionen dieselben Kosten haben).

Bisher waren alle Informationen über die Breitensuche positiv. Um zu erkennen, warum es sich dabei nicht immer um die Strategie der Wahl handelt, müssen wir betrachten, wie lange es dauert, eines Suche vollständig durchzuführen, und wie viel Speicher dafür benötigt wird. Dazu betrachten wir einen hypothetischen Zustandsraum, wo jeder Zustand b Nachfolger hat. Die Wurzel des Suchbaums erzeugt b Knoten auf der ersten Ebene, die jeweils b weitere Knoten erzeugen, was auf der zweiten Ebene insgesamt b^2 Knoten ergibt. Jeder *dieser* Knoten erzeugt b weitere Knoten, womit wir auf der dritten Ebene b^3 Knoten erhalten usw. Nehmen wir an, die Lösung befindet sich in der Tiefe d. Im schlimmsten Fall expandieren wir alle bis auf den letzten Knoten auf der Ebene d (weil das eigentliche Ziel nicht expandiert wird), womit wir $b^{d+1} - b$ Knoten auf Ebene $d + 1$ erhalten. Die Gesamtzahl der erzeugten Knoten beträgt damit:

$$b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Jeder erzeugte Knoten muss im Speicher bleiben, weil er entweder Teil des Randbereichs ist oder ein Vorfahre eines Randbereichknotens. Die Speicherkomplexität ist damit dieselbe wie die Zeitkomplexität (plus ein Knoten für die Wurzel).

Wer sich mit der Komplexitätsanalyse auskennt, macht sich sofort Sorgen, wenn er exponentielle Komplexitätsgrenzen wie etwa $O(b^{d+1})$ sieht (oder freut sich, weil er es als Herausforderung betrachtet). Abbildung 3.11 zeigt, warum das so ist. Sie listet die Zeit und den Speicher auf, die für eine Breitensuche mit einem Verzweigungsfaktor von $b = 10$ für verschiedene Werte der Lösungstiefe d benötigt werden. Die Tabelle geht davon aus, dass pro Sekunde 10.000 Knoten erzeugt werden können und für einen Knoten 1000 Byte

Speicherplatz erforderlich sind. Viele Suchprobleme können in etwa in diese Annahme eingeordnet werden (plus/minus einem Faktor von 100), wenn sie auf einem modernen PC ausgeführt werden.

Aus Abbildung 3.11 lernen wir zwei Dinge: Erstens, *die Speicheranforderungen sind für die Breitensuche ein größeres Problem als die Ausführungszeit.* 31 Stunden sind nicht zu lang, wenn es sich um die Lösung eines wichtigen Problems der Tiefe 8 handelt, aber nur wenige Computer haben das Terabyte Arbeitsspeicher, das dafür erforderlich ist. Glücklicherweise gibt es andere Suchstrategien, für die weniger Speicherplatz erforderlich ist.

Die zweite Lektion ist, dass die Zeitanforderungen immer noch einen wesentlichen Faktor darstellen. Wenn Ihr Problem eine Lösung in der Tiefe 12 aufweist, dann dauert es (unseren Annahmen nach) 35 Jahre, bis die Breitensuche (oder eine andere uninformierte Suche) sie gefunden hat. Im Allgemeinen *können Suchprobleme mit exponentieller Komplexität nicht von uninformierten Methoden gelöst werden, außer es handelt sich dabei um sehr kleine Beispiele.*

Tiefe	Knoten	Zeit	Speicher
2	1100	11 Sekunden	1 Megabyte
4	111100	11 Sekunden	100 Megabyte
6	10^7	19 Minuten	10 Gigabyte
8	10^9	31 Stunden	1 Terabyte
10	10^{11}	129 Tage	100 Terabyte
12	10^{13}	35 Jahre	10 Petabyte
14	10^{15}	3523 Jahre	1 Exabyte

Abbildung 3.11: Zeit- und Speicheranforderungen für die Breitensuche. Die hier gezeigten Zahlen setzen einen Verzweigungsfaktor von b = 10 voraus; 10.000 Knoten/Sekunde; 1000 Byte/Knoten.

3.4.2 Suche mit einheitlichen Kosten (Uniform-cost-Suche)

Die Breitensuche ist optimal, wenn alle Schrittkosten gleich sind, weil sie immer den *flachsten* nicht expandierten Knoten expandiert. Mit einer einfachen Erweiterung finden wir einen Algorithmus, der optimal für jede Schrittkostenfunktion ist. Statt den flachsten Knoten zu expandieren, expandiert die **Suche mit einheitlichen Kosten** den Knoten *n* mit den *geringsten Pfadkosten*. Beachten Sie, dass dies identisch mit einer Breitensuche ist, wenn alle Schrittkosten gleich sind.

Bei der Suche mit einheitlichen Kosten geht es nicht um die *Anzahl* der Schritte, die ein Pfad aufweist, sondern nur um die Gesamtkosten. Aus diesem Grund verbleibt sie in einer Endlosschleife, wenn sie irgendwann einen Knoten expandiert, der eine Aktion mit null

Kosten aufweist, die zum selben Zustand zurückführt (zum Beispiel eine *NoOp*-Aktion). Wir können Vollständigkeit garantieren, vorausgesetzt die Kosten jedes Schritts sind größerer oder gleich einer kleinen positiven Konstante ϵ. Diese Bedingung ist ebenfalls ausreichend, um die *Optimalität* sicherzustellen. Sie bedeutet, dass die Kosten für einen Pfad immer steigen, solange wir den Pfad verfolgen. Aus dieser Eigenschaft erkennt man leicht, dass der Algorithmus Knoten in der Reihenfolge steigender Pfadkosten expandiert. Aus diesem Grund ist der erste Zielknoten, der zur Expansion ausgewählt wird, die optimale Lösung. (Beachten Sie, dass TREE-SEARCH den Zieltest immer auf die Knoten anwendet, die für die Expansion ausgewählt sind.) Wir empfehlen, den Algorithmus auszuprobieren, um den kürzesten Pfad nach Bukarest zu finden.

Die Suche mit einheitlichen Kosten wird durch Pfadkosten und nicht durch Tiefe gesteuert; deshalb kann ihre Komplexität nicht ganz einfach im Hinblick auf b und d charakterisiert werden. Seien stattdessen C^* die Kosten der optimalen Lösung, und nehmen wir an, dass jede Aktion mindestens ϵ kostet. Dann ist die schlechteste Zeit- und Speicherkomplexität des Algorithmus gleich $O(b^{\lceil C^*/\epsilon \rceil})$, was sehr viel größer als b^d sein kann. Das liegt daran, dass die Suche mit einheitlichen Kosten große Bäume kleiner Schritte untersuchen kann, und dies auch häufig tut, bevor sie Pfade erkundet, die große und vielleicht sinnvolle Schritte beinhalten. Wenn alle Schritte gleich viel kosten, ist natürlich $b^{\lceil C^*/\epsilon \rceil}$ gleich b^d.

3.4.3 Tiefensuche (Depth-first search)

Die **Tiefensuche** expandiert immer die tiefsten Knoten in dem aktuellen Randbereich des Suchbaums. Abbildung 3.12 zeigt den Verlauf dieser Suche. Die Suche geht unmittelbar auf die tiefste Ebene des Suchbaums, wo die Knoten keinen Nachfolger haben. Wenn diese Knoten expandiert werden, werden sie aus dem Randbereich entfernt, so dass die Suche dann zum nächstflacheren Knoten weitergeht, der noch nicht erkundete Nachfolger aufweist.

Diese Strategie kann durch TREE-SEARCH mit einer LIFO-Schlange (last-in-first-out), auch als Stack bezeichnet, implementiert werden. Als Alternative zur TREE-SEARCH-Implementierung ist es üblich, die Tiefensuche mit einer rekursiven Funktion zu implementieren, die sich für jeden ihrer Nachfahren selbst aufruft. (Abbildung 3.13 zeigt einen neuen rekursiven Depth-first-Algorithmus, der eine Begrenzung für die Tiefe aufweist.)

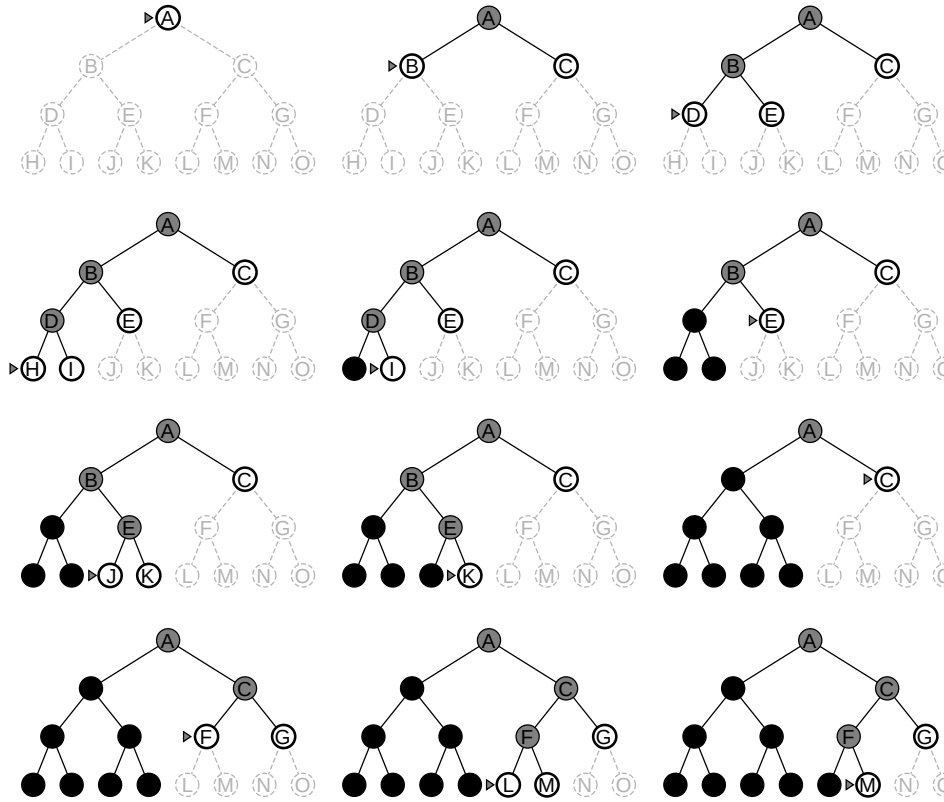

Abbildung 3.12: Tiefensuche in einem Binärbaum. Knoten, die expandiert wurden und keine Nachfolger im Randbereich aufweisen, können aus dem Speicher entfernt werden; sie sind schwarz dargestellt. Für Knoten auf der Tiefe 3 geht man davon aus, dass sie keine Nachfolger haben, und *M* ist der einzige Zielknoten.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns eine Lösung oder Fehler/
Abbruch
   return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns eine Lösung oder Fehler/
Abbruch
   cutoff_occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return Abbruch
   else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = Abbruch then cutoff_occurred= ← true
      else if result ≠ Fehler then return result
   if cutoff_occurred? then return Abbruch else return Fehler
```

Abbildung 3.13: Eine rekursive Implementierung der tiefenbeschränkten Suche

Die Tiefensuche hat sehr gemäßigte Speicheranforderungen. Sie muss nur einen einzigen Pfad von der Wurzel zu einem Blattknoten speichern, zusammen mit den verbleibenden nicht expandierten Geschwisterknoten für jeden Knoten auf dem Pfad. Nachdem ein Knoten expandiert wurde, kann er aus dem Speicher entfernt werden, sobald alle seine Nachfolger vollständig erkundet wurden (siehe Abbildung 3.12). Für einen Zustandsraum mit Verzweigungsfaktor b und einer maximalen Tiefe m benötigt die Tiefensuche nur $bm + 1$ Knoten. Unter Verwendung derselben Annahmen wie in Abbildung 3.13 und vorausgesetzt, dass Knoten mit derselben Tiefe wie der Zielknoten keinen Nachfolger haben, stellen wir fest, dass für die Tiefensuche 118 KB anstelle von 10 Petabyte bei einer Tiefe von $d = 12$ erforderlich sind. Das ist ein Faktor von 10 Milliarden mal weniger Speicherplatz.

Eine Variante der Tiefensuche ist die so genannte **Backtracking-Suche**, die noch weniger Speicher benötigt. Beim Backtracking wird jeweils nur ein einziger Nachfolger erzeugt und nicht alle Nachfolger auf einmal; jeder partiell expandierte Knoten merkt sich, welcher Nachfolger als Nächstes erzeugt werden soll. Auf diese Weise wird nur $O(m)$ Speicher an Stelle von $O(bm)$ Speicher benötigt. Die Backtracking-Suche ermöglicht einen weiteren Trick zur Speicherersparnis (und Zeitersparnis): die Idee, einen Nachfolger zu erzeugen, indem man die aktuelle Zustandsbeschreibung direkt abändert, statt sie zuerst zu kopieren. Damit werden die Speicheranforderungen auf nur eine einzige Zustandsbeschreibung und $O(m)$ Aktionen reduziert. Damit dies funktioniert, müssen wir in der Lage sein, Veränderungen rückgängig zu machen, wenn wir zurückgehen, um den nächsten Nachfolger zu erzeugen. Für Probleme mit großen Zustandsbeschreibungen, wie beispielsweise bei roboterbedienten Fließbändern, sind diese Techniken kritisch für den Erfolg.

Der Nachteil der Tiefensuche ist, dass sie eine falsche Auswahl treffen kann und möglicherweise in einem sehr langen (oder sogar unendlichen) Pfad stecken bleibt, während eine andere Auswahl zu einer Lösung in der Nähe der Wurzel des Suchbaums geführt hätte. In Abbildung 3.12 beispielsweise untersucht die Tiefensuche den gesamten linken Suchbaum, selbst wenn der Knoten C ein Zielknoten ist. Wäre der Knoten J ebenfalls ein Zielknoten, würde die Tiefensuche ihn als Lösung zurückgegeben; sie ist also nicht optimal. Wäre der linke Unterbaum unendlich tief, enthielte aber keine Lösungen, würde die Tiefensuche niemals terminieren; sie ist also nicht vollständig. Im schlimmsten Fall erzeugt die Tiefensuche alle $O(b^m)$ Knoten im Suchbaum, wobei m die maximale Tiefe jedes Knotens darstellt. Beachten Sie, dass m sehr viel größer als d sein kann (die Tiefe der flachsten Lösung) und unendlich ist, wenn der Baum keine Begrenzung aufweist.

3.4.4 Tiefenbeschränkte Suche (Depth-limited search)

Das Problem unbegrenzter Bäume kann abgeschwächt werden, indem man eine Tiefensuche mit vorgegebener Tiefenbegrenzung l bereitstellt. Das bedeutet, die Knoten der Tiefe l werden behandelt, als hätten sie keinen Nachfolger. Dieser Ansatz wird auch als **tiefenbeschränkte Suche** bezeichnet. Die Tiefenbegrenzung löst das Problem unendlich langer Pfade. Leider führt sie auch eine zusätzliche Quelle der Unvollständigkeit ein, wenn wir $l < d$ wählen, d. h. das flachste Ziel liegt unterhalb der Tiefenbegrenzung. (Das ist nicht unwahrscheinlich, wenn d unbekannt ist.) Die tiefenbeschränkte Suche ist außer-

dem nicht optimal, wenn wir $l > d$ wählen. Ihre Zeitkomplexität ist $O(b^l)$, die Speicherkomplexität $O(bl)$. Die Tiefensuche kann als Sonderfall der tiefenbeschränkten Suche mit $l = \infty$ betrachtet werden.

Manchmal können die Tiefenbegrenzungen auf einem Wissen über das Problem begründet werden. Auf der Landkarte von Rumänien beispielsweise gibt es 20 Städte. Wir wissen also, wenn es eine Lösung gibt, muss sie im schlechtesten Fall die Länge 19 aufweisen; $l = 19$ ist also eine mögliche Wahl. Hätten wir die Landkarte jedoch sorgfältig betrachtet, so hätten wir erkannt, dass jede Stadt von jeder anderen Stadt in höchstens neun Schritten erreicht werden kann. Diese Zahl, auch als **Durchmesser** des Zustandsraums bezeichnet, bietet eine bessere Tiefenbegrenzung, die zu einer effizienteren tiefenbeschränkten Suche führt. Für die meisten Probleme kennen wir jedoch keine gute Tiefenbegrenzung, bevor wir sie nicht gelöst haben.

Die tiefenbeschränkte Suche kann als einfaches Variante des allgemeinen Baum-Suchalgorithmus oder des rekursiven Tiefensuche-Suchalgorithmus implementiert werden. Wir zeigen den Pseudocode für die rekursive tiefenbeschränkte Suche in Abbildung 3.13. Beachten Sie, dass die tiefenbeschränkte Suche mit zwei Fehlertypen terminieren kann: Der Standardfehlerwert (*failure*) zeigt an, dass es keine Lösung gibt; der Abbruch- *(cutoff-*) Wert zeigt an, dass es keine Lösung innerhalb der Tiefenbegrenzung gibt.

3.4.5 Iterativ vertiefte Tiefensuche (iterative deepening depth-first search)

Die **iterativ vertiefte Tiefensuche** ist eine allgemeine Strategie, die häufig in Kombination mit der Tiefensuche verwendet wird, die die beste Tiefenbegrenzung ermittelt. Dazu erhöht sie schrittweise die Begrenzung – zuerst 0, dann 1, dann 2 usw., bis ein Ziel gefunden ist. Das passiert, wenn die Tiefenbegrenzung d erreicht, die Tiefe des flachsten Zielknotens. Abbildung 3.14 zeigt den zugehörigen Algorithmus. Die iterative Vertiefung kombiniert die Vorteile der Tiefen- und der Breitensuche. Bei der Tiefensuche sind ihre Speicheranforderungen recht gemäßigt, genauer gesagt $O(bd)$. Wie die Breitensuche ist sie vollständig, wenn der Verzweigungsfaktor endlich ist, und optimal, wenn die Pfadkosten eine nichtfallende Funktion der Knotentiefe sind. Abbildung 3.15 zeigt vier Iterationen von ITERATIVE-DEEPENING-SEARCH für einen binären Suchbaum, wobei die Lösung in der vierten Iteration gefunden wird.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns eine Lösung oder einen Fehler
   inputs: problem, ein Problem

   for depth ← 0 to ∞ do
   result ← DEPTH-LIMITED-SEARCH(problem, depth)
   if result ≠ Abbruch then return result
```

Abbildung 3.14: Der Suchalgorithmus für die iterative Vertiefung, die wiederholt eine tiefenbeschränkte Suche mit steigenden Begrenzungen anwendet. Er terminiert, sobald eine Lösung gefunden ist oder wenn die tiefenbeschränkte Suche einen Fehler zurückgibt, d. h. es keine Lösung gibt.

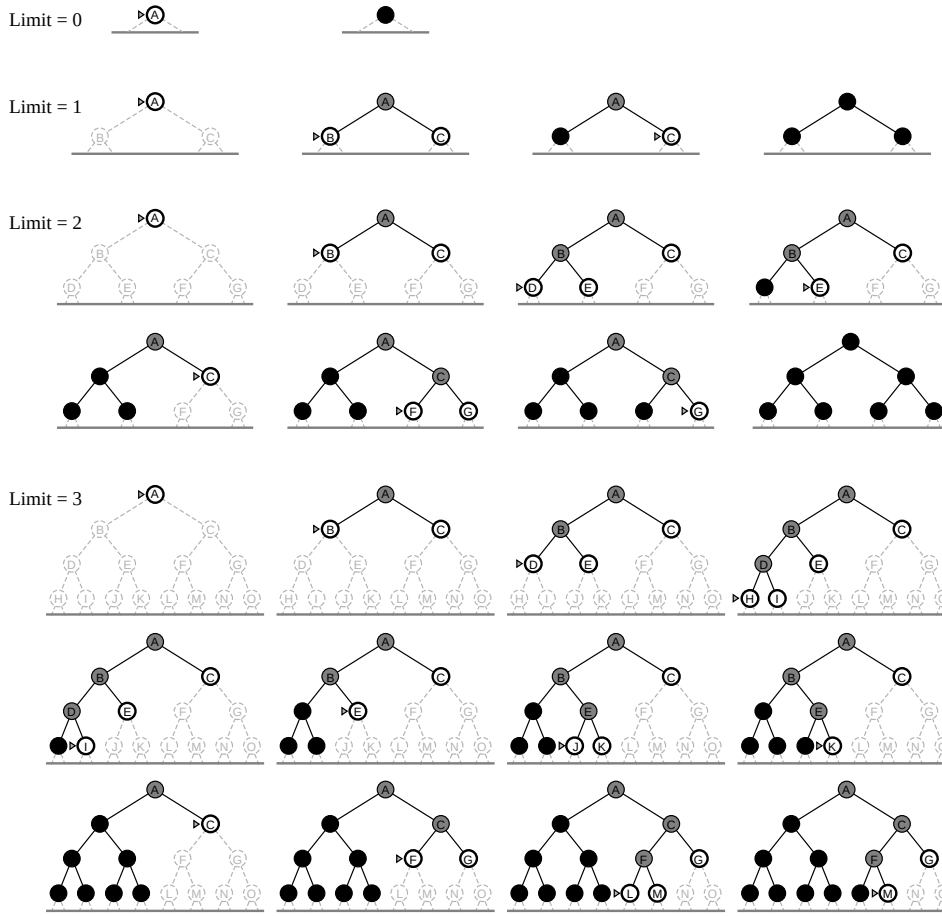

Abbildung 3.15: Vier Iterationen der Suche mit iterativer Vertiefung für einen Binärbaum

Die Suche durch iterative Vertiefung scheint vielleicht Verschwendung zu sein, weil die Zustände mehrfach erzeugt werden. Es stellt sich jedoch heraus, dass dies nicht sehr aufwändig ist. Der Grund dafür ist, dass sich bei einem Suchbaum mit demselben (oder fast demselben) Verzweigungsfaktor auf jeder Ebene die meisten Knoten auf der untersten Ebene befinden, so dass es keine Rolle spielt, wenn die oberen Ebenen mehrfach erzeugt werden. Bei einer iterativ vertiefenden Suche werden die Knoten auf der untersten Ebene (Tiefe d) einmal erzeugt, die auf der nächsthöheren Ebene zweimal usw. bis zu den unmittelbaren Nachfolgern der Wurzel, die d-mal erzeugt werden. Die Gesamtzahl der erzeugten Knoten ist also:

$$N(\text{iterativ vertiefende Suche}) = d(b) + (d-1)b^2 + \ldots + (1)b^d$$

Das bedeutet eine Zeitkomplexität von $O(b^d)$. Wir können dies mit den Knoten vergleichen, die durch eine Tiefensuche erzeugt werden:

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

Beachten Sie, dass die Breitensuche einige Knoten auf der Tiefe d+1 erzeugt, während das bei der iterativen Vertiefung nicht der Fall ist. Das Ergebnis ist, dass die iterative Vertiefung letztlich schneller ist als die Breitensuche, obwohl die Zustände wiederholt erzeugt werden. Für $b = 10$ und $d = 5$ beispielsweise sind die Zahlen gleich:

$$N(\text{iterativ vertiefende Suche}) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$

$$N(\text{Breitensuche}) = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100$$

Im Allgemeinen ist die iterative Vertiefung die bevorzugte uninformierte Suchmethode, wenn es einen großen Suchraum gibt und die Tiefe der Lösung nicht bekannt ist.

Die Suche mit iterativer Vertiefung ist analog zur Breitensuche, weil sie in jedem Iterationsschritt eine vollständige Ebene neuer Knoten auswertet, bevor sie zur nächsten Ebene weitergeht. Es scheint sinnvoll zu sein, ein iteratives Analogon zur Suche mit einheitlichen Kosten zu entwickeln, wobei die Optimalitätsgarantie des letztgenannten Algorithmus geerbt wird, während seine Speicheranforderungen vermieden werden. Die Idee dabei ist, statt steigender Tiefenbegrenzungen steigende Pfadkostenbegrenzungen zu verwenden. In dem resultierenden Algorithmus, auch als **iterativ verlängernde Suche** (**iterative lengthening**) bezeichnet, ist in Übung 3.11 dargestellt. Es stellt sich leider heraus, dass beim iterativen Lengthening ein wesentlicher Mehraufwand im Vergleich zur Suche mit einheitlichen Kosten entsteht.

3.4.6 Bidirektionale Suche

Die Idee bei der **bidirektionalen Suche** ist, zwei gleichzeitige Suchvorgänge auszuführen: einen vom Ausgangszustand aus, den anderen vom Ziel aus, und beides anzuhalten, sobald sich die beiden Suchprozesse in der Mitte treffen (Abbildung 3.16). Die Motivation dabei ist, dass $b^{d/2} + b^{d/2}$ sehr viel kleiner als b^d ist, oder in der Abbildung, dass die Fläche der beiden kleinen Kreise kleiner als die Fläche eines großen Kreises ist, dessen Mittelpunkt am Start liegt und dessen Radius bis zum Ziel verläuft.

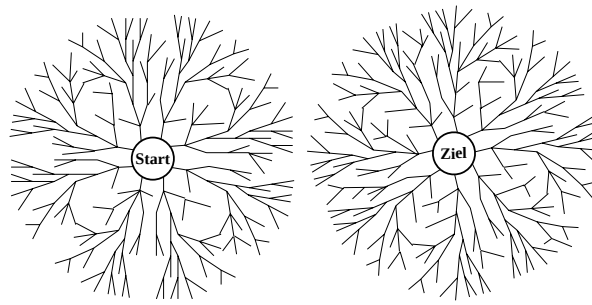

Abbildung 3.16: Schematische Ansicht einer bidirektionalen Suche, die erfolgreich ist, wenn eine Verzweigung vom Startknoten auf eine Verzweigung vom Zielknoten trifft.

Die bidirektionale Suche wird implementiert, indem man veranlasst, dass einer oder beide Suchläufe jeden Knoten überprüfen, bevor dieser expandiert wird, um festzustellen, ob er in der Verzweigung des anderen Suchbaums liegt; in diesem Fall wurde eine Lösung gefunden. Hat ein Problem beispielsweise die Lösungstiefe $d = 6$ und jede Richtung führt jeweils eine Breitensuche für einen Knoten aus, treffen sich die beiden Suchen im schlechtesten Fall, wenn jede Suche alle bis auf den letzten der Knoten der Tiefe 3 expandiert hat. Für $b = 10$ bedeutet das insgesamt 22.200 erstellte Knoten, im Vergleich zu 11.111.100 Knoten für eine Standardbreitensuche. Die Überprüfung, ob ein Knoten Teil des anderen Suchbaums ist, kann in einer konstanten Zeit mit Hilfe einer Hash-Tabelle durchgeführt werden. Die Zeitkomplexität der bidirektionalen Suche ist also $O(b^{d/2})$. Mindestens einer der Suchbäume muss im Speicher gehalten werden, so dass die Überprüfung auf die Mitgliedschaft erfolgen kann. Daraus ergibt sich eine Speicherkomplexität von ebenfalls $O(b^{d/2})$. Diese Speicheranforderung ist die größte Schwäche der bidirektionalen Suche. Der Algorithmus ist vollständig und optimal (für einheitliche Schrittkosten), wenn beide Suchen Breitensuchen sind; bei anderen Konditionen wird möglicherweise die Vollständigkeit, die Optimalität oder beides geopfert.

Die reduzierte Zeitkomplexität macht die bidirektionale Suche attraktiv, aber wie suchen wir rückwärts? Das ist nicht ganz so einfach, wie es sich anhört. Seien die **Vorgänger** eines Knotens n, $Pred(n)$, alle Knoten, die n als Nachfolger haben. Für die bidirektionale Suche ist es erforderlich, dass $Pred(n)$ effizient berechnet werden kann. Der einfachste Fall ist, wenn alle Aktionen im Zustandsraum auch umkehrbar sind, so dass gilt $Pred(n) = Succ(n)$. Für andere Fälle kann ein wesentlicher Aufwand entstehen.

Betrachten Sie die Frage, was wir meinen, wenn wir „das Ziel“ sagen, während wir „rückwärts vom Ziel aus“ suchen. Für das 8-Puzzle und für die Suche nach einer Route in Rumänien gibt es nur einen Zielzustand; die Rückwärtssuche ist also vergleichbar mit der Vorwärtssuche. Gibt es mehrere explizit aufgelistete Zielzustände – beispielsweise die beiden schmutzfreien Zielzustände in Abbildung 3.3 –, können wir einen neuen Dummy-Zielzustand konstruieren, dessen unmittelbare Vorgänger alle eigentlichen Zielzustände sind. Alternativ können einige redundante Knotenerstellungen vermieden werden, indem man die Menge der Zielzustände als einen einzigen Zustand betrachtet, wobei jeder Vorgänger auch eine Zustandsmenge ist – insbesondere die Menge der Zustände, die einen entsprechenden Nachfolger in der Menge der Zielzustände haben. (Siehe auch Abschnitt 3.6.)

Der schwierigste Fall für die bidirektionale Suche ist, wenn der Zielvergleich nur eine implizite Beschreibung einer möglicherweise großen Menge von Zielzuständen erzeugt – beispielsweise alle Zustände, die beim Schach den Zieltest „Schach matt“ erfüllen. Eine Rückwärtssuche müsste kompakte Beschreibungen „aller Zustände, die durch den Zug m_1 zum Schachmatt führen“ konstruieren usw.; und diese Beschreibungen müssen mit den Zuständen verglichen werden, die die Vorwärtssuche erzeugt. Es gibt keine allgemeine Möglichkeit, dies effizient zu erledigen.

3.4.7 Vergleich uninformierter Suchstrategien

Abbildung 3.17 vergleicht die Suchstrategien im Hinblick auf die vier Bewertungskriterien, die in Abschnitt 3.4 eingeführt wurden.

Kriterium	Breitensuche	Einheitliche Kosten	Tiefensuche	Beschränkte Tiefensuche	Iterative Vertiefung	Bidirektional (falls möglich)
Vollständig?	Ja[a]	Ja[a, b]	Nein	Nein	Ja[a]	Ja[a, d]
Zeit	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Speicher	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Ja[c]	Ja	Nein	Nein	Ja[c]	Ja[c, d]

Abbildung 3.17: Bewertung der Suchstrategien. *b* ist der Verzweigungsfaktor, *d* die Tiefe der flachsten Lösung, *m* die maximale Tiefe des Suchbaums, *l* die Tiefenbegrenzung. Die hochgestellten Symbole haben die folgenden Bedeutungen: [a]: vollständig, wenn *b* endlich ist; [b]: vollständig, wenn die Schrittkosten $\geq \varepsilon$ für jedes positive ε sind; [c]: optimal, wenn alle Schrittkosten identisch sind; [d]: wenn beide Richtungen eine Breitensuche verwenden.

3.5 Wiederholte Zustände vermeiden

Bisher haben wir eine der wichtigsten Komplikationen des Suchprozesses ignoriert: die Möglichkeit, Zeit zu verschwenden, indem man Zustände expandiert, die bereits besucht und zuvor expandiert wurden. Für einige Probleme tritt diese Möglichkeit niemals auf; der Zustandsraum ist ein Baum, und es gibt nur einen Pfad zu jedem Zustand. Die effiziente Formulierung des 8-Damen-Problems (wo jede neue Dame in der nächsten linken leeren Spalte platziert wird) ist zum größten Teil aus eben diesem Grund effizient, weil jeder Zustand nur über einen Pfad erreicht werden kann. Wenn wir das 8-Damen-Problem formulieren, so dass eine Dame in jeder beliebigen Spalte platziert werden kann, kann jeder Zustand bei *n* Damen durch *n*! verschiedene Pfade erreicht werden.

Für manche Probleme sind wiederholte Zustände unvermeidbar. Das schließt alle Probleme ein, wobei die Aktionen umkehrbar sind, wie beispielsweise Probleme zur Routensuche oder Schiebeblock-Puzzles. Die Suchbäume für diese Probleme sind unendlich, aber wenn wir einige der wiederholten Zustände kürzen, können wir den Suchbaum auf endliche Größe bringen und damit nur den Teil des Baums erzeugen, der den Zustandsraumgraphen aufspannt. Betrachtet man den Suchbaum nur bis zu einer bestimmten Tiefe, findet man ganz einfach Fälle, wobei durch die Eliminierung wiederholter Zustände eine exponentielle Reduzierung der Suchkosten entsteht. Im Extremfall wird ein Zustandsraum der Größe d+1 (Abbildung 3.18(a)) zu einem Baum mit 2^d Blättern (Abbildung 3.18(b)). Ein

realistischeres Beispiel ist das in Abbildung 3.18(c) für ein rechteckiges Gitter gezeigte. Bei einem Gitter hat jeder Zustand vier Nachfolger, der Suchbaum inklusive wiederholter Zustände hat also 4^d Blätter; es gibt jedoch nur etwa $2d^2$ verschiedene Zustände innerhalb von d Schritten in jedem beliebigen Zustand. Für $d = 20$ ergeben sich dafür etwa eine Trillion Knoten, aber nur etwa 800 verschiedene Zustände.

Wiederholte Zustände können dann bewirken, dass ein lösbares Problem zu einem unlösbaren Problem wird, wenn der Algorithmus es nicht erkennt. Die Erkennung bedeutet normalerweise, dass der Knoten, der expandiert werden soll, mit denen verglichen wird, die bereits expandiert wurden; wird eine Übereinstimmung festgestellt, hat der Algorithmus zwei Pfade zum selben Zustand erkannt und kann einen davon verwerfen.

Für die Tiefensuche sind die einzigen Knoten im Speicher diejenigen auf dem Pfad von der Wurzel zum aktuellen Knoten. Ein Vergleich dieser Knoten mit dem aktuellen Knoten erlaubt dem Algorithmus, verschleifte Pfade zu erkennen, die sofort verworfen werden können. Das ist praktisch, um sicherzustellen, dass endliche Zustandsräume nicht zu unendlichen Suchbäumen werden, nur weil Schleifen vorliegen; leider wird damit nicht die exponentielle Fortpflanzung von Pfaden, die keine Schleife enthalten, in Problemen wie den in Abbildung 3.18 gezeigten vermieden. Die einzige Möglichkeit, sie zu vermeiden, ist, mehr Knoten im Speicher aufzubewahren. Es gibt eine grundlegende Abwägung zwischen Speicher und Zeit. *Algorithmen, die ihren Verlauf vergessen, werden diesen sehr wahrscheinlich wiederholen.*

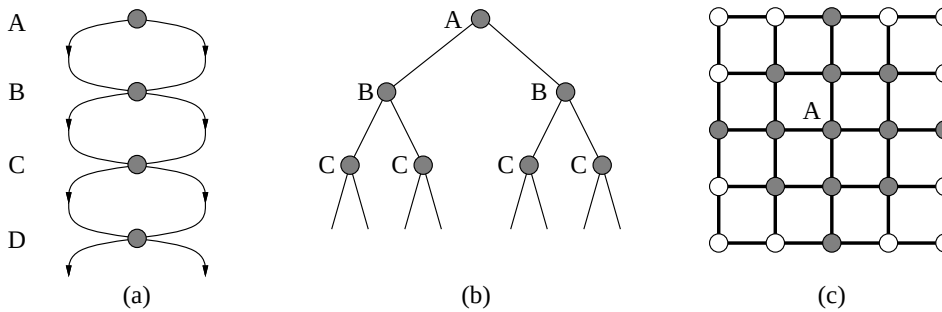

Abbildung 3.18: Zustandsräume, die einen exponentiell größeren Suchbaum erzeugen. (a) ein Zustandsraum, in dem zwei mögliche Aktionen von A nach B führen, zwei von B nach C usw. Der Zustandsraum enthält $d + 1$ Zustände, wobei d die maximale Tiefe darstellt. (b) Der entsprechende Suchbaum, der 2^d Verzweigungen hat, die den 2^d Pfaden durch den Raum entsprechen. (c) Ein rechteckiges Gitter. Zustände innerhalb von zwei Schritten vom Ausgangszustand (a) sind grau dargestellt.

Wenn sich ein Algorithmus jeden Zustand merkt, den er besucht hat, kam man davon ausgehen, dass er den Zustandsraumgraphen direkt erkundet. Wir können den allgemeinen TREE-SEARCH-Algorithmus abändern, so dass er eine Datenstruktur beinhaltet, die als die **geschlossene Liste** bezeichnet wird, die jeden erweiterten Knoten speichert. (Der Randbereich nicht erweiterter Knoten wird manchmal auch als die **offene Liste** bezeichnet.) Wenn der aktuelle Knoten mit einem Knoten aus der geschlossenen Liste übereinstimmt, wird er nicht erweitert, sondern verworfen. Der neue Algorithmus wird als GRAPH-SEARCH (Abbildung 3.19) bezeichnet. Bei Problemen mit vielen wiederholten Zuständen

ist GRAPH-SEARCH sehr viel effizienter als TREE-SEARCH. Die Worst-case-Zeit und die Speicheranforderungen sind proportional zur Größe des Zustandsraums. Das kann sehr viel kleiner als $O(b^d)$ sein.

```
function GRAPH-SEARCH(problem, fringe) returns eine Lösung oder einen Fehler

   closed ← eine leere Menge
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if EMPTY?(fringe) then return Fehler
      node ← REMOVE-FIRST(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      if STATE[node] gehört nicht zu closed then
         füge STATE[node] zu closed hinzu
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Abbildung 3.19: Der allgemeine Graph-Search-Algorithmus. Die Menge *closed* kann mit Hilfe einer Hash-Tabelle implementiert werden, um die Überprüfung auf wiederholte Zustände effizienter zu machen. Dieser Algorithmus setzt voraus, dass der erste Pfad zu einem Zustand *s* der billigste ist (siehe Text).

Es kann sehr kompliziert sein, Optimalität für eine Graphensuche zu erzielen. Wir haben bereits gesagt, dass, wenn ein wiederholter Zustand erkannt wird, der Algorithmus zwei Pfade zum selben Zustand gefunden hat. Der Algorithmus GRAPH-SEARCH in Abbildung 3.19 verwirft immer den *zuletzt entdeckten* Pfad; wenn der neu entdeckte Pfad kürzer als der ursprüngliche ist, könnte GRAPH-SEARCH offenbar eine optimale Lösung verpassen. Glücklicherweise können wir zeigen (Übung 3.12), dass dies nicht passieren kann, wenn entweder eine Suche mit einheitlichen Kosten oder eine Breitensuche mit konstanten Schrittkosten verwendet wird; diese beiden optimalen Baum-Suchstrategien sind also auch optimale Graphen-Suchstrategien. Die iterativ vertiefende Graphensuche dagegen verwendet eine Tiefensuche-Expandierung und kann leicht einem nicht optimalen Pfad zu einem Knoten folgen, bevor sie den optimalen findet. Aus diesem Grund muss die Suche mit iterativ vertieftem Graphen also überprüfen, ob ein neu erkannter Pfad zu einem Knoten besser als der ursprüngliche ist, und wenn dies der Fall ist, muss sie möglicherweise die Tiefen und die Pfadkosten der Nachfolger dieses Knotens revidieren.

Beachten Sie, dass die Verwendung einer geschlossenen Liste bedeutet, dass für die Tiefensuche und für die iterativ vertiefende Suche keine linearen Speicheranforderungen mehr gelten. Weil der Algorithmus GRAPH-SEARCH jeden Knoten im Speicher behält, sind einige Suchen auf Grund von Speicherbeschränkungen nicht durchführbar.

3.6 Suche mit partieller Information

In Abschnitt 3.3 sind wir davon ausgegangen, dass die Umgebung vollständig beobachtbar und deterministisch ist und der Agent weiß, welche Wirkung die einzelnen Aktionen haben. Aus diesem Grund kann der Agent genau berechnen, welche Zustände aus einer beliebigen Folge von Aktionen resultieren, und er weiß immer, in welchem Zustand er

sich befindet. Seine Wahrnehmung bietet nach einer Aktion keine neuen Informationen. Was passiert, wenn das Wissen über die Zustände oder die Aktionen unvollständig ist? Wir stellen fest, dass unterschiedliche Arten der Unvollständigkeit zu drei verschiedenen Problemstellungen führen:

1. **Sensorlose Probleme** (auch als angepasste Probleme bezeichnet): Wenn der Agent überhaupt keine Sensoren besitzt, könnte er sich (soweit er es erkennen kann) in einem von mehreren möglichen Ausgangszuständen befinden, und jede Aktion könnte deshalb zu einem von mehreren möglichen Nachfolgezuständen führen.
2. **Kontingenzprobleme**: Wenn die Umgebung teilweise überschaubar ist oder Aktionen unsicher sind, bietet die Wahrnehmung des Agenten nach jeder Aktion neue Informationen. Jede mögliche Wahrnehmung definiert eine neue Möglichkeit, für die geplant werden muss. Ein Problem wird als **adversarial** bezeichnet, wenn die Unsicherheit durch die Aktionen eines anderen Agenten verursacht wird.
3. **Exploration der Probleme**: Wenn die Zustände und Aktionen der Umgebung unbekannt sind, muss der Agent versuchen, sie zu entdecken. Explorationsprobleme können als Extremfall von Kontingenzproblemen bezeichnet werden.

Als Beispiel verwenden wir die Umgebung der Staubsaugerwelt. Sie wissen, dass der Zustandsraum acht Zustände hat, wie in Abbildung 3.20 gezeigt. Es gibt drei Aktionen – *Links*, *Rechts* und *Saugen* –, und das Ziel ist, den gesamten Schmutz zu entfernen (Zustand 7 und 8). Wenn die Umgebung beobachtbar, deterministisch und vollständig bekannt ist, so ist das Problem durch jeden der beschriebenen Algorithmen ganz einfach zu lösen. Ist der Ausgangszustand beispielsweise 5, erreicht die Aktionsfolge [*Rechts, Saugen*] einen Zielzustand, 8. Der restliche Abschnitt beschäftigt sich mit sensorlosen und Kontingenzversionen des Problems. Explorationsprobleme werden in Abschnitt 4.5 beschrieben, adversariale Probleme in Kapitel 6.

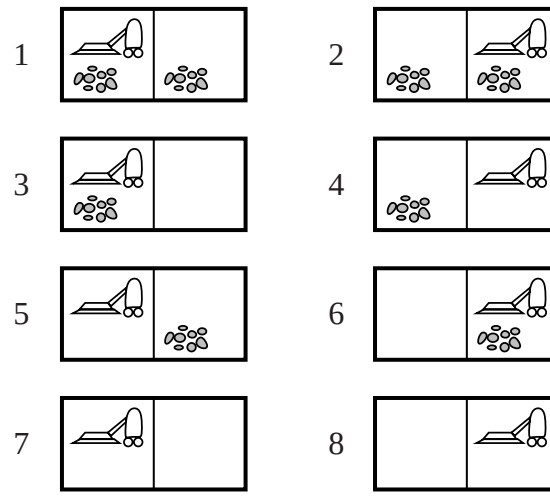

Abbildung 3.20: Die acht möglichen Zustände der Staubsaugerwelt

3.6.1 Sensorlose Probleme

Angenommen, der Staubsauger-Agent kennt alle Wirkungen seiner Aktionen, aber er besitzt keine Sensoren. Er weiß dann nur, dass sein Ausgangszustand aus der Menge {1, 2, 3, 4, 5, 6, 7, 8} stammt. Man könnte annehmen, die Vorhersage des Agenten ist hoffnungslos, aber tatsächlich kommt er ganz gut zurecht. Weil er weiß, welche Aktionen er durchführen kann, kann er beispielsweise berechnen, dass die Aktion *Rechts* bewirkt, dass er in einen der Zustände {2, 4, 6, 8} gelangt und die Aktionsfolge [*Rechts*, *Saugen*] immer zu einem der Zustände {4, 8} führt. Die Folge [*Rechts*, *Saugen*, *Links*, *Saugen*] schließlich garantiert, den Zielzustand 7 zu erreichen, unabhängig von dem vorliegenden Startzustand. Wir sagen, der Agent kann die Welt in den Zielzustand **zwingen**, selbst wenn er nicht weiß, von wo aus er gestartet ist. Zusammenfassend können wir sagen, wenn die Welt nicht vollständig beobachtbar ist, muss der Agent anhand der Zustandsmengen schließen, in die er gelangen könnte, und nicht anhand von einzelnen Zuständen. Wir bezeichnen jede dieser Zustandsmengen als **Glaubenszustand**, die den aktuellen Glauben des Agenten im Hinblick auf die möglichen physischen Zustände darstellt, in denen er sich befinden könnte. (In einer vollständig beobachtbaren Umgebung enthält jeder Glaubenszustand genau einen physischen Zustand.)

Um sensorlose Probleme zu lösen, suchen wir im Raum der Glaubenszustände und nicht der physischen Zustände. Der Ausgangszustand ist ein Glaubenszustand, und jede Aktion bildet von einem Glaubenszustand in einen anderen Glaubenszustand ab. Eine Aktion wird auf einen Glaubenszustand angewendet, indem die Ergebnisse der Anwendung dieser Aktion auf jeden physischen Zustand im Glaubenszustand zusammengefasst werden. Ein Pfad verbindet jetzt mehrere Glaubenszustände, und eine Lösung ist jetzt ein Pfad, der zu einem Glaubenszustand führt, *dessen Elemente alle* Zielzustände sind. Abbildung 3.21 zeigt den erreichbaren Glaubenszustandsraum für die deterministische, sensorlose Staubsaugerwelt. Es gibt nur zwölf erreichbare Glaubenszustände, aber der gesamte Glaubenszustandsraum enthält jede mögliche Menge physischer Zustände, d.h. $2^8 = 256$ Glaubenszustände. Im Allgemeinen gilt, wenn der physische Zustandsraum S Zustände hat, hat der Glaubenszustandsraum 2^S Glaubenszustände.

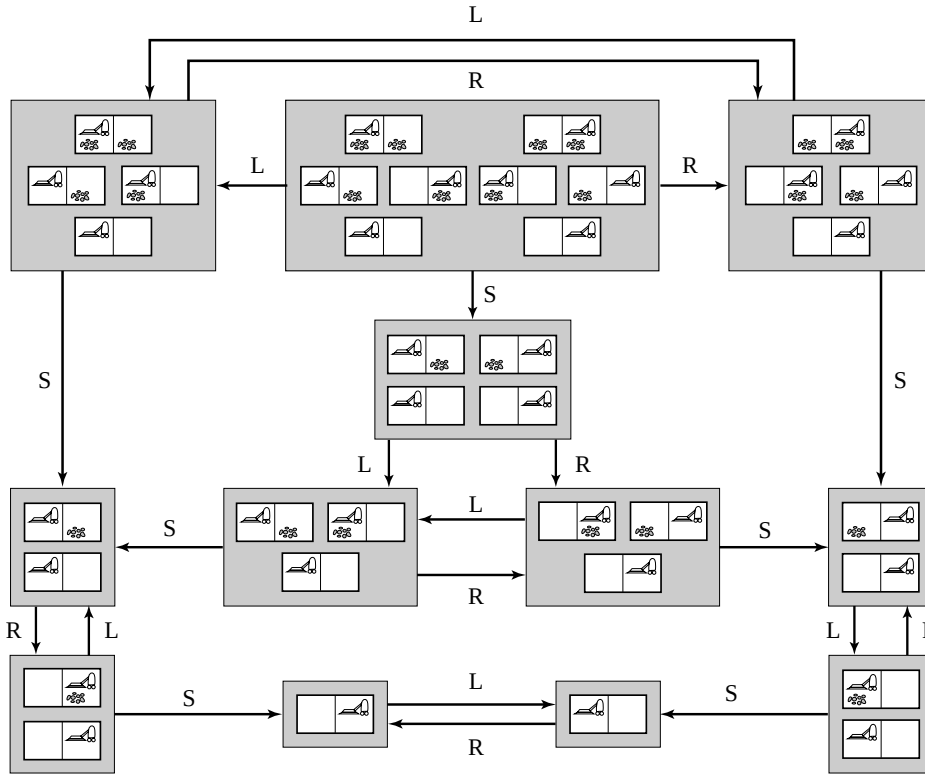

Abbildung 3.21: Der erreichbare Teil des Glaubenszustandsraums für die deterministische, sensorlose Staubsaugerwelt. Jedes grau unterlegte Feld entspricht einem einzelnen Glaubenszustand. Der Agent befindet sich zu jedem beliebigen Zeitpunkt in einem bestimmten Glaubenszustand, weiß aber nicht, in welchem physischen Zustand er sich befindet. Der anfängliche Glaubenszustand (völliges Unwissen) ist das obere Feld in der Mitte. Aktionen werden durch beschriftete Pfeile dargestellt. Schleifen wurden der Übersichtlichkeit halber weggelassen.

Unsere Beschreibung der sensorlosen Probleme ist bisher von deterministischen Aktionen ausgegangen, aber die Analyse bleibt im Wesentlichen unverändert, wenn die Umgebung nicht deterministisch ist, d.h. wenn die Aktionen mehrere mögliche Ergebnisse haben können. Der Grund dafür ist, dass der Agent beim Fehlen von Sensoren keine Möglichkeit hat, zu erkennen, welches Ergebnis eigentlich aufgetreten ist, so dass die verschiedenen möglichen Ergebnisse nur weitere physische Zustände im Nachfolger-Glaubenszustand sind. Angenommen, die Umgebung gehorcht Murphys Gesetz: Die *Saugen*-Aktion schüttet *manchmal* Schmutz auf dem Teppich aus, aber nur dann, *wenn sich dort noch kein Schmutz befindet*.[6] Wird also *Saugen* im physischen Zustand 4 (siehe Abbildung 3.20) angewendet, gibt es zwei mögliche Ergebnisse: Zustände 2 und 4. Wendet

[6] Wir nehmen an, dass die meisten Leser ähnlichen Problemen gegenüberstehen und sich solidarisch mit unserem Agenten zeigen. Wir entschuldigen uns bei Besitzern moderner, effizienterer Haushaltsgeräte, die den Nutzen dieser pädagogischen Geräte nicht verstehen.

man *Saugen* auf den Ausgangs-Glaubensraum {1, 2, 3, 4, 5, 6, 7, 8} an, führt das jetzt zu dem Glaubenszustand, der die Vereinigung aus den Ergebnismengen für die acht physischen Zustände darstellt. Wenn wir dies berechnen, stellen wir fest, dass der neue Glaubenszustand gleich {1, 2, 3, 4, 5, 6, 7, 8} ist. Für einen sensorlosen Agenten in der Welt von Murphys Gesetz lässt die Aktion *Saugen* den Glaubenszustand unverändert! Tatsächlich ist das Problem nicht lösbar (siehe Übung 3.18). Intuitiv kann man sagen, der Grund dafür ist, dass der Agent nicht erkennen kann, ob das aktuelle Quadrat schmutzig ist, und dass er deshalb auch nicht erkennen kann, ob die Aktion *Saugen* es reinigt oder noch mehr Schmutz erzeugt.

3.6.2 Kontingenzprobleme

Wenn die Umgebung so gestaltet ist, dass der Agent neue Informationen von seinen Sensoren erhält, nachdem er eine Aktion ausgeführt hat, steht der Agent einem **Kontingenzproblem** gegenüber. Die Lösung für ein Kontingenzproblem nimmt häufig die Form eines *Baums* an, wobei jeder Zweig ausgewählt sein kann, abhängig von den Wahrnehmungen, die bis zu diesem Punkt im Baum empfangen wurden. Angenommen, der Agent befindet sich in der Welt von Murphys Gesetz und hat einen Positionssensor und einen Schmutzsensor, aber keinen Sensor, der Schmutz in anderen Quadraten erkennt. Die Wahrnehmung [*L, Schmutzig*] bedeutet also, dass sich der Agent in einem der Zustände {1, 3} befindet. Der Agent könnte die Aktionsfolge [*Saugen, Rechts, Saugen*] formulieren. Durch das *Saugen* würde sich der Zustand in einen von {5, 7} ändern, und eine Drehung nach rechts würde dann den Zustand auf einen aus {6, 8} ändern. Die Ausführung der letzten *Saugen*-Aktion im Zustand 6 führt uns zu Zustand 8, einem Ziel, aber ihre Ausführung in Zustand 8 könnte uns zurück nach Zustand 6 bringen (nach Murphys Gesetz), und dann schlägt der Plan fehl.

Betrachtet man den Glaubenszustandsraum für diese Version des Problems, kann einfach erkannt werden, dass keine feste Aktionsfolge eine Lösung für dieses Problem garantiert. Es gibt jedoch eine Lösung, wenn wir nicht auf einer festen Aktionsfolge bestehen:

[*Saugen, Rechts,* ***if*** [*R, Schmutzig*] ***then*** *Saugen*]

Das erweitert den Lösungsraum, so dass die Möglichkeit der Auswahl von Aktionen abhängig von den Möglichkeiten erfolgt, die während der Ausführung entstehen. Viele Probleme in der realen, physischen Welt sind Kontingenzprobleme, weil eine genauere Vorhersage nicht möglich ist. Aus diesem Grund halten viele Leute ihre Augen offen, während sie gehen oder fahren.

Kontingenzprobleme erlauben manchmal rein sequenzielle Lösungen. Betrachten Sie beispielsweise eine *vollständig beobachtbare* Welt nach Murphys Gesetz. Kontingenzen entstehen, wenn der Agent eine *Saugen*-Aktion in einem sauberen Quadrat ausführt, weil dann in diesem Quadrat Dreck abgelegt werden kann, aber nicht muss. Solange der Agent dies niemals tut, entstehen keine Kontingenzen, und es gibt für jeden Ausgangszustand eine sequenzielle Lösung (Übung 3.18).

Die Algorithmen für Kontingenzprobleme sind komplizierter als die Standard-Suchalgorithmen in diesem Kapitel; sie werden in Kapitel 12 beschrieben. Kontingenzprobleme führen auch zu einem etwas anderen Entwurf der Agenten, weil der Agent handeln kann, *bevor* er einen garantiert funktionierenden Plan hat. Das ist praktisch, denn statt jede Möglichkeit im Voraus zu betrachten, die während der Ausführung auftreten *könnte*, ist es häufig besser, zuerst zu handeln und dann zu prüfen, *ob* Kontingenzen auftreten. Der Agent kann dann fortfahren, das Problem zu lösen, und die zusätzliche Information berücksichtigen. Diese Art Verzahnung von Suche und Ausführung ist auch für Explorationsprobleme (siehe Abschnitt 4.5) und Spiele (siehe Kapitel 6) nützlich.

Zusammenfassung

Dieses Kapitel hat Methoden vorgestellt, die ein Agent anwenden kann, um Aktionen in Umgebungen auszuwählen, die deterministisch, beobachtbar, statisch und vollständig bekannt sind. In diesen Fällen kann der Agent Aktionsfolgen konstruieren, mit denen er seine Ziele erreicht; dieser Prozess wird als **Suche** bezeichnet.

- Bevor ein Agent beginnen kann, nach Lösungen zu suchen, muss er sein **Ziel** formulieren und anhand des Ziels ein **Problem** formulieren.
- Ein Problem besteht aus vier Komponenten: dem **Ausgangszustand**, einer Menge von **Aktionen**, einer **Zieltest**funktion und einer **Pfadkosten**funktion. Die Umgebung des Problems wird durch einen **Zustandsraum** dargestellt. Einen **Pfad** durch den Zustandsraum vom Ausgangszustand zu einem Zielzustand ist eine **Lösung**.
- Ein einzelner allgemeiner TREE-SEARCH-Algorithmus kann für die Lösung beliebiger Probleme genutzt werden; spezifische Varianten des Algorithmus verwenden unterschiedliche Strategien.
- Suchalgorithmen werden nach **Vollständigkeit**, **Optimalität**, **Zeit**- und **Speicherkomplexität** beurteilt. Die Komplexität ist von b abhängig, dem Verzweigungsfaktor im Zustandsraum, sowie von d, der Tiefe der flachsten Lösung.
- Die **Breitensuche** wählt den flachsten nicht erweiterten Knoten im Suchbaum zur Expandierung aus. Sie ist vollständig, optimal für einheitliche Schrittkosten und weist eine Zeit- und Speicherkomplexität von $O(b^d)$ auf. Auf Grund der Speicherkomplexität ist sie für die meisten Fälle nicht geeignet. Die **Suche mit einheitlichen Kosten** ist mit der Breitensuche vergleichbar, expandiert aber den Knoten mit den geringsten Pfadkosten, $g(n)$. Sie ist vollständig und optimal, wenn die Kosten für jeden Schritt eine positive Grenze ε überschreiten.
- Die **Tiefensuche** wählt den tiefsten nicht expandierten Knoten im Suchbaum zur Expandierung aus. Sie ist weder vollständig noch optimal und hat eine Zeitkomplexität von $O(b^m)$ und eine Speicherkomplexität von $O(bm)$, wobei m die maximale Tiefe jedes Pfads im Zustandsraum darstellt.
- Die **tiefenbeschränkte Suche** legt eine feste Begrenzung für die Tiefe der Tiefensuche fest.

- Die **iterativ vertiefende Suche** ruft die tiefenbeschränkte Suche mit steigenden Begrenzungen auf, bis ein Ziel gefunden wurde. Sie ist vollständig, optimal für einheitliche Schrittkosten und weist eine Zeitkomplexität von $O(b^d)$ sowie eine Speicherkomplexität von $O(bd)$ auf.
- Die **bidirektionale Suche** kann die Zeitkomplexität wesentlich reduzieren, aber sie ist nicht immer anwendbar und kann zu viel Speicher verbrauchen.
- Wenn es sich bei dem Zustandsraum um einen Graphen und nicht um einen Baum handelt, kann es sich auszahlen, auf wiederholte Zustände im Suchbaum zu überprüfen. Der GRAPH-SEARCH-Algorithmus eliminiert alle Doppelzustände.
- Wenn die Umgebung partiell beobachtbar ist, kann der Agent Suchalgorithmen im Raum von **Glaubenszuständen** anwenden oder im Raum der Mengen möglicher Zustände, in denen sich der Agent befinden könnte. In einigen Fällen kann eine einzige Lösungsfolge konstruiert werden; in anderen Fällen braucht der Agent einen **Kontingenzplan**, um möglicherweise auftretende unbekannte Umstände bearbeiten zu können.

Bibliografische und historische Hinweise

Die meisten der in diesem Kapitel analysierten Zustandsraum-Suchprobleme haben eine lange Geschichte in der Literatur und sind weniger banal, als sie vielleicht erscheinen mögen. Das Problem mit den Missionaren und den Kannibalen aus Übung 3.9 wurde detailliert von Amarel (1968) analysiert. Es wurde früher in der KI von Simon und Newell (1961) sowie in der Operationsforschung von Bellman und Dreyfus (1962) betrachtet. Studien wie diese sowie die Arbeit von Newell und Simon zum Logic Theorist (1957) und GPS (1961) führten zur Einrichtung von Suchalgorithmen als die wichtigsten Waffen in dem Feldzug der KI-Forscher von 1960 und zur Einrichtung der Problemlösung als die KI-Aufgabe überhaupt. Leider wurde im Hinblick auf die Automatisierung der Problemformulierungsschritte sehr wenig getan. Eine neuere Behandlung der Problemdarstellung und Abstraktion, inklusive KI-Programme, die diese Aufgaben (teilweise) selbst durchführen, findet sich in Knoblock (1990).

Das 8-Puzzle ist ein jüngerer Verwandter des 15-Puzzles, das um 1870 von dem berühmten amerikanischen Spieledesigner Sam Loyd (1959) erfunden wurde. Das 15-Puzzle erlangte in den Vereinigten Staaten schnell eine immense Popularität, vergleichbar mit der neueren Sensation, die durch den Zauberwürfel von Rubik ausgelöst wurde. Außerdem zog es auch schnell die Aufmerksamkeit von Mathematikern auf sich (Johnson und Story 1879; Tait, 1880). Die Herausgeber des *American Journal of Mathematics* berichteten: „Das 15-Puzzle wurde in den letzten paar Wochen beim amerikanischen Publikum immer bekannter, und man kann sagen, dass es die Aufmerksamkeit von neun von zehn Personen beiderlei Geschlechts sowie aller Altersstufen und Schichten auf sich gezogen hat. Das hätte jedoch die Herausgeber nicht beeinflusst, einen Artikel zu einem solchen Thema im *American Journal of Mathematics* zu veröffentlichen, wäre nicht...“ (es folgt ein Überblick über das neue mathematische Interesse an dem 15-Puzzle). Eine ausführliche Analyse des 8-Puzzles wurde mit Hilfe eines Computers von Schofield (1967) vor-

genommen. Ratner und Warmuth (1986) zeigten, dass die allgemeine $n \times n$-Version des 15-Puzzles zur Klasse der NP-vollständigen Probleme gehört.

Das 8-Damen-Problem wurde zuerst 1848 anonym im deutschen Schachmagazin *Schach* veröffentlicht; später wurde es einem gewissen Max Bezzel zugeordnet. Es wurde erneut 1850 veröffentlicht, als es die Aufmerksamkeit des großen Mathematikers Carl Friedrich Gauss auf sich gezogen hatte, der versuchte, alle möglichen Lösungen aufzulisten, jedoch nur 72 fand. 1850 veröffentlichte Nauck alle 92 Lösungen. Netto (1901) verallgemeinerte das Problem auf n Damen, und Abramson und Yung (1989) fanden einen Algorithmus mit $O(n)$.

Jedes der Suchprobleme aus der realen Welt, die in diesem Kapitel aufgelistet wurden, hat einen wesentlichen Forschungsaufwand verursacht. Methoden zur Auswahl optimaler Flüge bleiben größtenteils proprietär, aber Carl de Marcken (persönliche Kommunikation) hat gezeigt, dass die Preisgebung und die Beschränkungen von Flugtickets so kompliziert geworden sind, dass das Problem der Auswahl eines optimalen Fluges formal *nicht entscheidbar* ist. Das Problem des Handlungsreisenden ist ein kombinatorisches Standardproblem der theoretischen Informatik (Lawler, 1985; Lawler et al., 1992). Karp (1972) bewies, dass das Problem des Handlungsreisenden NP-hart sei, aber es wurden effektive heuristische Annäherungsmethoden entwickelt (Lin und Kernighan, 1973). Arora (1998) leitete ein vollständig polynomiales Annäherungsschema für Euklidsche TSPs ab. VLSI-Layout-Methoden werden von Shahookar und Mazumder (1991) erforscht, und in VLSI-Journalen erschienen zahlreiche Artikel zur Layout-Optimierung. Roboternavigation und Montageprobleme werden in Kapitel 25 beschrieben.

Algorithmen der uninformierten Suche für die Problemlösung sind ein zentrales Thema der klassischen Informatik (Horowitz und Sahni, 1978) und Operationsforschung (Dreyfus, 1969); Deo und Pang (1984) und Gallo und Pallottino (1988) beschreiben neuere Forschungen. Die Breitensuche wurde von Moore (1959) für die Lösung von Labyrinthen formuliert. Die Methode der **dynamischen Programmierung** (Bellman und Dreyfus, 1962), die systematisch alle Lösungen für alle Unterprobleme steigender Längen aufzeichnet, kann als eine Art von Breitensuche für Graphen betrachtet werden. Der Zwei-Punkt-kürzester-Pfad-Algorithmus (two-point shortest-path algorithm) von Dijkstra (1959) ist der Ursprung der Suche mit einheitlichen Kosten.

Eine Version der iterativen vertiefenden Suche, die darauf ausgelegt war, die Schachuhr effizient zu nutzen, wurde zuerst von Slate und Atkin (1977) im Schachspielprogramm CHESS 4.5 verwendet, aber die Anwendung auf die Kürzester-Pfad-Graphen-Suche geht auf Korf (1985a) zurück. Die bidirektionale Suche, die von Pohl (1969, 1971) eingeführt wurde, kann in einigen Fällen ebenfalls effektiv sein.

Teilweise beobachtbare und nichtdeterministische Umgebungen wurden innerhalb des Problemlösungsansatzes noch nicht weiter untersucht. Einige Effizienzaspekte in der Glaubenszustandssuche wurden von Genesereth und Nourbakhsh (1993) untersucht. Koenig und Simmons (1998) betrachteten die Roboternavigation aus einer unbekannten Ausgangsposition, und Erdman und Mason (1988) untersuchten das Problem der Robotermanipulation ohne Sensoren unter Verwendung einer stetigen Form der Glaubenszustandssuche. Die Kontingenzsuche wurde innerhalb des Planungsteilbereichs untersucht (siehe Kapitel 12). Größtenteils wurden Planung und Aktionen mit unsicheren Informa-

tionen unter Verwendung von Werkzeugen der Wahrscheinlichkeits- und Entscheidungstheorie betrachtet (siehe Kapitel 17).

Die Lehrbücher von Nilsson (1971, 1980) sind gute allgemeine Informationsquellen zu klassischen Suchalgorithmen. Einen umfassenden und aktuelleren Überblick finden Sie bei Korf (1988). Artikel über neue Suchalgorithmen – die bemerkenswerterweise immer noch entwickelt werden – erscheinen in Journalen wie beispielsweise *Artificial Intelligence*.

Übungen

3.1 Definieren Sie in eigenen Worten die folgenden Begriffe: Zustand, Zustandsraum, Suchbaum, Suchknoten, Ziel, Aktion, Nachfolgerfunktion und Verzweigungsfaktor.

3.2 Erklären Sie, warum einer Zielformulierung eine Problemformulierung folgen muss.

3.3 Angenommen, LEGAL-ACTIONS(s) beschreibt die Menge aller Aktionen, die im Zustand s erlaubt sind, und RESULT(a, s) beschreibt den Zustand, der aus der Ausführung einer erlaubten Aktion a im Zustand s resultiert. Definieren Sie SUCESSOR-FN im Hinblick auf LEGAL-ACTIONS und RESULT und umgekehrt.

3.4 Zeigen Sie, dass die 8-Puzzle-Zustände in zwei disjunkte Mengen unterteilt sind, so dass kein Zustand in einer Menge in einen Zustand in der anderen Menge umgewandelt werden kann, egal wie viele Züge dafür aufgewendet werden. (Hinweis: Siehe Berlekamp et al. (1982).) Leiten Sie eine Prozedur ab, mit der Sie erkennen, zu welcher Klasse ein bestimmter Zustand gehört, und erklären Sie, warum dies für die Erzeugung beliebiger Zustände sinnvoll ist.

3.5 Betrachten Sie das n-Damen-Problem unter Verwendung der „effizienten“ inkrementellen Formulierung, die im Anschluss an Abbildung 3.5 diskutiert wird. Erklären Sie, warum die Größe des Zustandsraums mindestens $\sqrt[3]{n!}$ beträgt, und schätzen Sie das größte n ab, für das eine umfassende Exploration machbar ist. (Hinweis: Leiten Sie eine Untergrenze für den Verzweigungsfaktor ab, indem Sie die maximale Anzahl der Quadrate betrachten, die eine Dame in jeder Spalte angreifen kann.)

3.6 Führt ein endlicher Zustandsraum immer zu einem endlichen Suchbaum? Wie verhält es sich bei einem endlichen Zustandsraum, der ein Baum ist? Können Sie Genaueres darüber sagen, welche Arten von Zustandsräumen immer zu endlichen Suchbäumen führen? (Übernommen aus Bender, 1996.)

3.7 Geben Sie Ausgangszustand, Zieltest, Nachfolgerfunktion und Kostenfunktion für jede der folgenden Aufgaben an. Wählen Sie eine Formulierung, die präzise genug ist, um implementiert zu werden.

a. Sie sollen eine Landkarte mit nur vier Farben so einfärben, dass zwei benachbarte Regionen nicht dieselbe Farbe haben.
b. Ein 30 cm großer Affe befindet sich in einem Raum, in dem Bananen von der 80 cm hohen Decke hängen. Er möchte eine Banane pflücken. Im Raum befinden sich zwei stapelbare, bewegliche und besteigbare 30 cm hohe Würfel.

c. Sie haben ein Programm, das die Meldung „Ungültiger Eingabedatensatz“ ausgibt, wenn eine bestimmte Datei mit Eingabedatensätzen eingelesen wird. Sie wissen, dass die Verarbeitung jedes einzelnen Datensatzes unabhängig von den anderen Datensätzen ist. Sie wollen erkennen, welcher Datensatz fehlerhaft ist.
d. Sie haben drei Behälter mit jeweils 12, 8 und 3 Litern Aufnahmevolumen und ein Wasserfass. Sie können die Behälter füllen und ihren Inhalt entweder in die anderen Behälter oder auf den Boden gießen. Sie wollen genau 1 Liter abmessen

3.8 Betrachten Sie einen Zustandsraum, wo der Ausgangszustand Nummer 1 ist und die Nachfolgerfunktion für den Zustand n zwei Zustände mit den Nummern $2n$ und $2n + 1$ zurückgibt.
a. Zeichnen Sie den Teil des Zustandsraums für die Zustände 1 bis 15.
b. Angenommen, der Zielzustand ist 11. Listen Sie die Reihenfolge auf, in der die Knoten bei einer Breitensuche, tiefenbeschränkten Suche mit der Grenze 2 und iterativ vertiefenden Suche besucht werden.
c. Wäre eine bidirektionale Suche für dieses Problem geeignet? Wenn ja, beschreiben Sie detailliert, wie sie funktioniert.
d. Welcher Verzweigungsfaktor liegt bei der bidirektionalen Suche für jede Richtung vor?
e. Führt die Antwort auf (c) zu einer neuen Formulierung des Problems, die Ihnen erlaubt, das Problem zu lösen, fast ohne eine Suche vom Zustand 1 zu einem vorgegebenen Zielzustand zu gelangen?

3.9 Das Problem der Missionare und Kannibalen wird im Allgemeinen wie folgt formuliert: Drei Missionare und drei Kannibalen befinden sich auf einer Seite eines Flusses, und sie haben ein Boot, das ein oder zwei Menschen aufnehmen kann. Suchen Sie eine Möglichkeit, alle auf die andere Seite zu bringen, ohne je eine Gruppe von Missionaren auf einer Seite zu lassen, wo ihnen die Kannibalen zahlenmäßig überlegen sind. Dieses Problem ist in der KI sehr bekannt, weil es Thema des ersten Artikels war, der zur Problemformulierung aus analytischer Sicht erschien (Amarel, 1968).
a. Formulieren Sie das Problem präzise, und treffen Sie nur die Unterscheidungen, die für die Sicherstellung einer gültigen Lösung erforderlich sind. Zeichnen Sie eine Skizze des vollständigen Zustandsraums.
b. Implementieren und lösen Sie das Problem optimal unter Verwendung eines geeigneten Suchalgorithmus. Ist es sinnvoll, auf wiederholte Zustände zu überprüfen?
c. Warum glauben Sie, ist es für die Menschen zu schwierig, dieses Rätsel zu lösen, wenn man überlegt, dass der Zustandsraum so einfach ist?

3.10 Implementieren Sie zwei Versionen der Nachfolgerfunktion für das 8-Puzzle: Die eine Version soll alle Nachfolger gleichzeitig erzeugen, indem die 8-Puzzle-Datenstruktur kopiert und bearbeitet wird, und die andere soll nur jeweils einen neuen Nachfolger erzeugen, wenn sie aufgerufen wird, indem sie den übergeordneten Zustand direkt bearbeitet (und die Bearbeitungen gegebenenfalls rückgängig macht). Schreiben Sie Versionen der iterativ vertiefenden Tiefensuche, die diese Funktionen verwenden, und vergleichen Sie ihre Leistung.

3.11 Im letzten Abschnitt von Kapitel 3.4.5 haben wir die **iterativ vertiefende Suche** erwähnt, ein iteratives Analogon zur Suche mit einheitlichen Kosten. Die Idee dabei ist, steigende Obergrenzen für Pfadkosten zu verwenden. Wird ein Knoten erzeugt, dessen Pfadkosten die aktuelle Obergrenze überschreiten, wird er sofort verworfen. Für jede neue Iteration wird die Obergrenze auf die niedrigsten Pfadkosten der in der vorherigen Iteration verworfenen Knoten gesetzt.

a. Zeigen Sie, dass dieser Algorithmus optimal für allgemeine Pfadkosten ist.
b. Betrachten Sie einen einheitlichen Baum mit Verzweigungsfaktor b, einer Lösungstiefe d und einheitlichen Schrittkosten. Wie viele Iterationen sind für die iterative Vertiefung erforderlich?
c. Betrachten Sie jetzt die Schrittkosten, die aus dem stetigen Bereich [0,1] mit einem Minimum an positiven Kosten ε entstehen. Wie viele Iterationen sind im schlimmsten Fall erforderlich?
d. Implementieren Sie den Algorithmus und wenden Sie ihn auf Instanzen des 8-Puzzles und des Handlungsreisenden-Problems an. Vergleichen Sie die Leistung des Algorithmus mit der der Suche mit einheitlichen Kosten und kommentieren Sie Ihre Ergebnisse.

3.12 Beweisen Sie, dass die Suche mit einheitlichen Kosten und die Breitensuche mit konstanten Schrittkosten optimal sind, wenn sie für den GRAPH-SEARCH-Algorithmus eingesetzt werden. Zeigen Sie einen Zustandsraum mit konstanten Schrittkosten, in dem GRAPH-SEARCH unter Verwendung iterativer Vertiefung eine suboptimale Lösung findet.

3.13 Beschreiben Sie einen Zustandsraum, in dem sich eine iterativ vertiefende Suche sehr viel schlechter verhält als eine Breitensuche (z.B. $O(n^2)$ im Vergleich zu $O(n)$).

3.14 Schreiben Sie ein Programm, das als Eingabe die URLs von zwei Webseiten entgegennimmt und einen Pfad der Links von der einen zur anderen ermittelt. Welche Suchstrategie ist am besten geeignet? Ist eine Verwendung der bidirektionalen Suche sinnvoll? Könnte eine Suchmaschine verwendet werden, um eine Vorgängerfunktion zu implementieren?

3.15 Betrachten Sie das Problem, den kürzesten Pfad zwischen zwei Punkten auf einer Ebene zu finden, die konvexe polygonale Hindernisse aufweist, wie in Abbildung 3.22 gezeigt. Dies ist eine Idealisierung des Problems, das ein Roboter lösen muss, um sich seinen Weg in einer unordentlichen Umgebung zu schaffen.

a. Angenommen, der Zustandsraum besteht aus allen Positionen (x,y) in der Ebene. Wie viele Zustände gibt es? Wie viele Pfade gibt es zu dem Ziel?

b. Erklären Sie kurz, warum der kürzeste Pfad von einer Polygonkante zur nächsten in der Ebene aus geradlinigen Segmenten bestehen muss, die einige der Kanten der Polygone beinhalten. Definieren Sie jetzt einen guten Zustandsraum. Wie groß ist dieser Zustandsraum?

c. Definieren Sie die erforderlichen Funktionen, um das Suchproblem zu implementieren, einschließlich einer Nachfolgerfunktion, die eine Kante als Eingabe entgegennimmt und die Menge der Kanten zurückgibt, die in einer geraden Linie von der vorgegebenen Kante aus erreicht werden können. (Vergessen Sie nicht die Nachbarn im selben Polygon.) Verwenden Sie die Luftliniendistanz für die heuristische Funktion.

d. Wenden Sie einen oder mehrere der Algorithmen aus diesem Kapitel an, um Probleme in diesem Bereich zu lösen, und beschreiben Sie ihre Leistung.

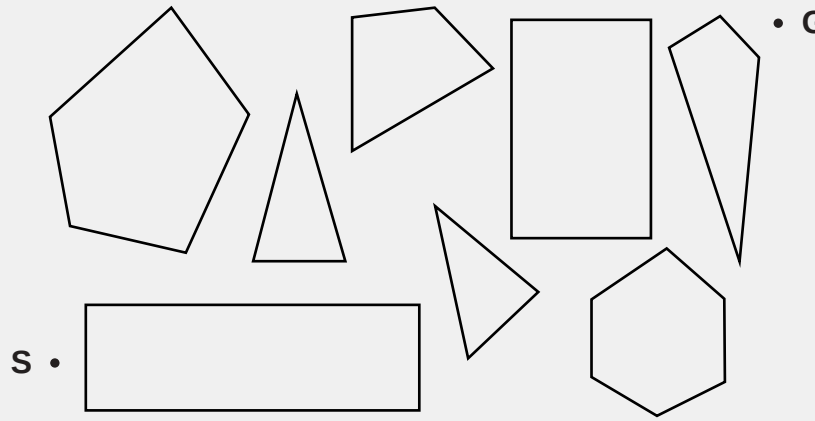

Abbildung 3.22: Ein Szenario mit polygonalen Hindernissen

3.16 Wir können das Navigationsproblem aus Übung 3.15 wie folgt in eine Umgebung umwandeln:

- Die Wahrnehmung ist eine Liste der Positionen der sichtbaren Kanten *relativ zu dem Agenten*. Die Wahrnehmung beinhaltet *nicht* die Position des Roboters! Der Roboter muss seine eigene Position aus der Karte lernen; für den Moment können Sie davon ausgehen, dass jede Position eine andere „Perspektive“ hat.

- Jede Aktion ist ein Vektor, der einen geradlinigen Pfad beschreibt, dem gefolgt werden soll. Ist der Pfad ohne Hindernis, ist die Aktion erfolgreich; andernfalls hält der Roboter an der Stelle an, wo sein Pfad sich mit dem ersten Hindernis kreuzt. Gibt der Agent einen Vektor ohne jede Bewegung zurück und befindet er sich am Ziel (das feststehend und bekannt ist), soll die Umgebung den Agenten an eine beliebige Position (aber nicht innerhalb eines Hindernisses) versetzen.
- Das Leistungsmaß berechnet dem Agenten 1 Punkt für jede durchquerte Distanzeinheit und belohnt in jeweils mit 1000 Punkten, wenn das Ziel erreicht ist.

a. Implementieren Sie diese Umgebung und einen Problemlösungs-Agenten dafür. Der Agent muss ein neues Problem formulieren, nachdem er in die Umgebung zurückversetzt wurde, wobei er seine aktuelle Position ermitteln muss.
b. Dokumentieren Sie die Leistung Ihres Agenten (indem Sie den Agenten bei der Bewegung geeignete Kommentare erzeugen lassen) und zeichnen Sie seine Leistung über 100 Durchgänge auf.
c. Ändern Sie die Umgebung so ab, dass der Agent in 30 Prozent der Zeit an einem nicht erwünschten Ziel ankommt (zufällig aus den anderen (gegebenenfalls vorhandenen) sichtbaren Kanten ausgewählt, andernfalls überhaupt keine Bewegung). Dies ist ein einfaches Modell der Bewegungsfehler eines realen Roboters. Ändern Sie den Agenten so ab, dass er beim Erkennen eines solchen Fehlers feststellt, wo er sich befindet, und dann einen Plan entwickelt, sich an den alten Punkt zu bewegen und den alten Plan wieder aufzunehmen. Beachten Sie, dass es manchmal auch fehlschlagen kann, an die vorherige Position zurückzugelangen! Zeigen Sie ein Beispiel dafür, wie der Agent erfolgreich zwei aufeinander folgende Bewegungsfehler überwindet und dennoch das Ziel erreicht.
d. Probieren Sie jetzt zwei andere Auflösungsschemas für einen Fehler aus: (1) suchen Sie nach der am nächsten gelegenen Kante auf der ursprünglichen Route; (2) planen Sie die Route zum Ziel von der neuen Position aus neu. Vergleichen Sie die Leistung der drei Wiederherstellungsschemas. Würde die Verwendung von Suchkosten den Vergleich beeinflussen?
e. Angenommen, es gibt Positionen, von denen aus die Perspektive identisch ist. (Beispielsweise wenn die Welt ein Raster mit quadratischen Hindernissen ist.) Welcher Art Problem steht der Agent jetzt gegenüber? Wie sehen Lösungen aus?

3.17 In Kapitel 3.1.1 haben wir gesagt, dass wir keine Probleme mit negativen Pfadkosten betrachten wollen. In dieser Übung werden wir weiter darauf eingehen.

a. Angenommen, Aktionen können beliebig große negative Kosten haben; erklären Sie, warum diese Möglichkeit jeden optimalen Algorithmus zwingen würde, den gesamten Zustandsraum zu durchsuchen.
b. Hilft es, wenn wir darauf bestehen, dass Schrittkosten größer oder gleich einer negativen Konstanten c sein müssen? Betrachten Sie sowohl Bäume als auch Graphen.

c. Angenommen, es gibt eine Menge von Operatoren, die keine Schleifen bilden, so dass die Ausführung der Menge in einer beliebigen Reihenfolge keine eigentlichen Änderungen an den Zuständen erzeugt. Was bedeutet es für das optimale Verhalten eines Agenten in einer solchen Umgebung, wenn alle diese Operatoren negative Kosten haben?

d. Man kann sich leicht Operatoren mit hohen negativen Kosten vorstellen, sogar in Bereichen wie der Routensuche. Einige Straßenzüge haben vielleicht einen so schönen Ausblick, dass dies die normalen Kosten im Hinblick auf Zeit und Treibstoff bei weitem übertrifft. Erklären Sie präzise innerhalb des Kontexts der Zustandsraumsuche, warum Menschen nicht unendlich lange in Aussichtsschleifen fahren, und erklären Sie, wie Sie den Zustandsraum und die Operatoren für die Routensuche definieren, so dass auch künstliche Agenten Schleifen vermeiden können.

e. Können Sie sich einen realen Bereich vorstellen, in dem Schrittkosten Schleifen verursachen?

3.18 Betrachten Sie die sensorlose 2-Positionen-Staubsauger-Welt unter Murphys Gesetz. Zeichnen Sie den Glaubenszustandsraum, der von dem Ausgangs-Glaubenszustand $\{1, 2, 3, 4, 5, 6, 7, 8\}$ erreicht werden kann, und erklären Sie, warum das Problem nicht lösbar ist. Zeigen Sie auch, dass es eine Lösungsfolge für jeden möglichen Ausgangszustand gibt, wenn die Welt vollständig beobachtbar ist.

3.19 Betrachten Sie das Staubsaugerwelt-Problem, das in Abbildung 2.2 definiert wurde.

a. Welcher der in diesem Kapitel definierten Algorithmen ist für dieses Problem geeignet? Sollte der Algorithmus auf wiederholte Zustände überprüfen?

b. Wenden Sie den von Ihnen gewählten Algorithmus an, um eine optimale Aktionsfolge für eine 3×3-Welt zu berechnen, in deren Ausgangszustand Schmutz in den drei oberen Quadraten liegt und der Agent sich in der Mitte befindet.

c. Konstruieren Sie einen Suchagenten für die Staubsaugerwelt und bewerten Sie seine Leistung in einer Menge von 3×3-Welten mit einer Wahrscheinlichkeit von 0,2 von Schmutz in jedem Quadrat. Berücksichtigen Sie die Suchkosten sowie die Pfadkosten bei der Leistungsbewertung und verwenden Sie einen vernünftigen Umrechnungssatz.

d. Vergleichen Sie Ihren besten Suchagenten mit einem einfachen zufälligen Reflex-Agenten, der saugt, wenn Schmutz vorhanden ist, und sich andernfalls zufällig bewegt.

e. Betrachten Sie, was passiert, wenn die Welt auf $n \times n$ vergrößert wird. Inwieweit variiert die Leistung des Suchagenten und des Reflex-Agenten mit der Größe von n?