

とオペレーティングシステムの組合せについては、オンラインプログラム集に既に構築済みのコードが含まれている。)

**2.8** 練習問題 2.7 における掃除機問題環境のための単純条件反射エージェントを実装せよ。汚れ条件とエージェント位置に関するすべての可能な初期条件について、環境シミュレータを走らせよ。各条件でのエージェント性能得点とその全平均得点を記録せよ。

**2.9** 練習問題 2.7 の掃除機問題について各移動ごとに 1 点減点する修正版を考えよ。

- この環境で単純条件反射エージェントは完璧に合理的でありうるか、説明せよ。
- 内部状態をもつ条件反射エージェントならどうか。そのようなエージェントを設計せよ。
- エージェントが環境内の各広場がきれいか汚れているかを知覚できる場合、上記 a と b に関する答えはどう変わるか。

**2.10** 練習問題 2.7 の掃除機問題環境において、最初の汚れの条件と同様に環境の広がりや境界、障害物など環境の地図に関する未知である修正版を考える（エージェントは上下左右に動くことができるとする）。

- 単純条件反射エージェントはこの環境で完全に合理的であることが可能か、説明せよ。
- 乱数 を用いるエージェント関数付きの単純条件反射エージェントは単純条件反射エージェントより性能が良いか。そのようなエージェントを設計し、いくつかの環境で性能を測定せよ。
- 乱数を用いるエージェントの性能が悪くなる環境を設計できるか。その評価結果を示せ。
- 内部状態つき条件反射エージェントは単純条件反射エージェントより性能が良いか。そのようなエージェントを設計し、いくつかの環境でその性能を測定せよ。このようなタイプの合理的エージェントを設計できるか。

**2.11** 位置センサの代わりに、エージェントが障害物に向かって移動したり環境の境界を横切ろうとしたりすることを検出する“衝突”センサにした場合、練習問題 2.10 に再度取り組め。もし衝突センサが働かなくなった場合、エージェントはどのように行動するか。

**2.12** 上記練習問題の掃除機問題環境はすべて決定的であった。以下の各確率的条件で可能なエージェントプログラムについて論ぜよ。

- マーフィーの法則：25 % の時間にわたって、汚れている場合にきれいにする吸込み行為が失敗し、床がきれいなのにゴミを床にぶちまける。また、もし 10 % の時間は間違ったセンサ出力が得られるという場合には、エージェントプログラムはどのような影響を受けるか。
- 小さい子供：各時間ステップにおいて、きれいな広場それぞれが 10 % の確率で汚るとする。この場合、合理的エージェントの設計を考えられるか。

## 3 探索による問題解決

### 問題解決エージェント

単一の行為では達成できないゴールを達成するため、エージェントはどのように連続した一連の行為を見つけることができるかを考察する。

2 章で議論した単純なエージェントは反射エージェントであった。これは、状態から行為への直接的写像に基づいて行動する。そのようなエージェントはこの写像が記憶するには大きすぎ、学習するにも長くかかりすぎるような環境をうまく扱えない。他方、ゴールに基づいたエージェントは将来の行為とそれによる結果を考慮することでうまく扱うことができる。

この章では、問題解決エージェント (problem-solving agent) とよばれる一種のゴールに基づいたエージェントについて説明する。問題解決エージェントは、望ましい状態にいたる行為列を見つけることによって行うべきことを決める。まず“問題”とその“解”を構成する要素を正確に定義し、そしてその定義を説明するために、いくつかの例を述べる。次にこれらの問題を解くことのできるいくつかの汎用の探索アルゴリズムを述べ、それらの利点を比較する。これらのアルゴリズムは問題の定義以外には問題に関する情報を用いないという意味で、知識なし (uninformed) のアルゴリズムである。4 章では、解を見つけるためにどこを検索したらよいかについて何らかの知識をもつ知識あり (informed) の方法を扱う。

この章は、アルゴリズム解析に関する概念を用いる。漸近的計算量の概念 (すなわち  $O()$  記法) と NP 完全性についてよく知らない読者は、付録 A を参照するとよい。

### 3.1 問題解決エージェント

知的エージェントは、達成度を最大にすることを目的とする。2 章で述べたように、もしエージェントがゴール (goal) を決定することができ、そしてそれを満足させるように目指すことができるならば、これはいくぶん単純化される。エージェントがこれを行う方法と理由についてまず注目してみよう。

休日の旅行でルーマニアの都市 Arad にあるエージェントがいると考えよう。エージェントの達成度は、多くの要因を含んでいる。日焼けの状態を良くしたり、ルーマニア語を上達したり、名所を訪れたり、あるいは Arad の夜 (たいして期待できないが) を楽しみたい

し、二日酔いは避けたいなどである。このときの意思決定の問題は複雑で、多くのトレイドオフを含んでいる。ガイドブックも注意深く読まなくてはならない。ここでエージェントは払戻不可のチケットで翌日 Bucharest から飛び立とうとしていると仮定しよう。エージェントは Bucharest へ向かうというゴール(goal)を採用するのには意味がある。Bucharest に時間どおりに着かなくななるような行為は、それ以上考慮することなく除外することができる。これによつてエージェントの意思決定問題は、どのようにゴールの決定は、エージェントが達成しようとしている目的を制限し、その振舞いを決めるのに役立つ。そのときの状況と達成度に基づいてゴールの定式化(goal formulation)を行うことは、問題解決における最初の段階である。

ゴールを世界状態の集合、すなわちちょうどその集合に含まれる状態になることでゴールが満たされるような状態の集合とみなすことにしてしまう。エージェントがしなければならないのはどのような行為の系列がエージェントをゴール状態の一つに到達されるかを見つけることである。これをを行うためには、どんな種類の行為と状態を最初に決める必要がある。もし“左足を前方に1インチ動かす”あるいは“ハンドルを左へ1度回す”というレベルで行為を考慮しようとするのならば、Bucharest へ行くことは言うまでもなく、おそらく駐車場の出口さえ決して見つけられないだろう。なぜならば、そのような詳細なレベルでは不確かなことが多すぎ、解へいたるのにあまりに多くのステップを必要とするからである。問題の定式化(problem formulation)は、与えられたゴールに対してどんな行為と状態を考慮すべきかを決める過程である。この過程はのちほど、詳細に議論することにする。さしあたり、大きな町から町へドライブするというレベルの行為を考えると仮定しよう。したがつて、考慮する状態はエージェントがある特定の町にいるということに対応する。<sup>1</sup>

エージェントは、今 Bucharest へドライブするというゴールを採用して、そして Arad からどこへ車で行くか考えている。Arad からは三つの道が出ていて、Sibiu へ向かう道と、Timisoara へ向かう道と、Zerind へ向かう道である。これらどれもゴールを達成しない。それで、エージェントがルーマニアの地理に非常に精通しているのでないかぎり、どの道を行けばよいかわからないだろう。<sup>2</sup> 言い換えれば、エージェントは可能な行為のうちのどちらが最良であるかを知らない。なぜならば、エージェントはおのれの行為をとつたときに生じる状態について十分知らないからである。もしエージェントが付加的知識を何ももたらさなければ動けなくなってしまう。せいぜいできることは、でたらめに一つ行為を選ぶことである。

けれども、エージェントが紙かあるいはそのメモリの中にルーマニアの地図をもつてゐるとしても、エージェントは自身がいたることのできる状態と、そこでとどめのできる行為についての情報を知ることができ。あの三つの町のおのの道を通る道のその後の段階を考え、そして最後には Bucharest に着く道すじを見つけようとするために、エージェントはこの情報を使うことができる。ひとたびエージェントが Arad から Bucharest に向かう経路を地図上で見つけたら、その道すじに対応する

<sup>1</sup> 一つの基世界状態はあらゆる面についてそこで起こっていることを規定しているので、これらの忠実のものが必ずしも世界状態の大規模な集合に對応していることに注意せよ。問題解決の状態と世界状態の間の区別を忘れないでおくことは重要である。

<sup>2</sup> 我々は、ほとんどの読者がこのエージェントと同じ立場において、エージェントと同じくらい手がかりがないことを容易に想像できると仮定している。この教育的工具を利用することができないルーマニアの読者にお詫びする。

```

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    action  $\leftarrow$  FIRST(seq)
    seq  $\leftarrow$  REST(seq)
  return action

```

図 3.1 単純な問題解決エージェント。始めにゴールと問題を定式化し、問題を解決することによって、それを一度に一つずつ実行する。これが最後まで終わると、エージェントは別のゴールを定式化し、始めからもう一歩、エージェントが行動を実行していく間、知覚を無視しており、見つけた解が常に機能することに注意して欲しい。

ドライブの行為を実行することによって、そのゴールを達成することができ、一般的に、エージェントが、そのときただちに選択肢が複数あり、その結果導かれる状態の価値がわからぬ場合、まず価値がわかる状態へ導くと調べることで、何をすべきかわかる。そして、最良の行為を選択することができます。

そのような列を探す過程は、探索(search)とよばれる。探索アルゴリズムは問題を入力として受けとり、行為の形の解(solution)を返す。ひとたび解が見つけられたら、それが薦める行為を実行することができる。これは実行(execution)段階とよばれる。こうして図 3.1 で示されるような“定式化、探索、実行”というエージェントの単純なプランを考えることができる。ゴールと解くべき問題を定式化したあとで、エージェントは、それを解くために探索手続きを呼び出す。それからエージェントは、解をガイドとして用いながら行為を行う。つまり、解が薦める次の行為——たいていは行為列の最初の行為——を実行し、それを列から取り去る。解を実行し終わると、エージェントは新たなゴールを定式化する。

次に、まず問題の定式化について述べたあと、本章の多くを SEARCH 関数について述べ以上、本章では議論しない。

詳細な議論に入つていく前に、問題解決エージェントが 2 章で議論したエージェントや環境における行為を実行することができる。これは実行(execution)段階とよばれる。この結果導かれる状態の価値がわかる場合、まず価値がわかる状態へ導くと調べることで、何をすべきかわかる。そして、最良の行為を選択することができます。

そのような列を探す過程は、探索(search)とよばれる。探索アルゴリズムは問題を入力として受けとり、行為の形の解(solution)を返す。ひとたび解が見つけられたら、それが薦める行為を実行することができる。これは実行(execution)段階とよばれる。こうして図 3.1 で示されるような“定式化、探索、実行”というエージェントの単純なプランを考えることができる。ゴールと解くべき問題を定式化したあとで、エージェントは、それを解くために探索手続きを呼び出す。それからエージェントは、解をガイドとして用いながら行為を行う。つまり、解が薦める次の行為——たいていは行為列の最初の行為——を実行し、それを列から取り去る。解を実行し終わると、エージェントは新たなゴールを定式化する。

探索  
実行

道の行為の例であり、そのため予期せぬ出来事には対処できない、さらに、解の実行においては知覚されない！ 計画を実行するエージェントはいかばば目を閉じて実行する。そして何が起るか、完全に確信している（制御系ではこのことを開ループ（open-loop）系とよぶ、知覚を無視することでエージェントと環境のループができるからである）。これららの仮定すべてによって環境は最も簡単に扱うことのできるものになる。本章が本書の中で初めのはうに置かれていることの理由の一つは、このことにある。3.6 節は可観測性と決定性の仮定をゆるめたときにどのようになるかを簡単に見る。12 章と 17 章ではさらには深く考える。

### 明確に定義された問題と解

問題 (problem) は形式的に、四つの要素から定義される。

- エージェントがそこから開始する初期状態 (initial state). たとえば、ルーマニアにいる我々のエージェントの初期状態は  $In(Arad)$  と記述できるだろう。
- エージェントが利用できる可能な行為 (action) の記述。一般的な形式化<sup>3</sup>では後者関数 (successor function) が用いられる。ある特定の状態  $x$  に対して、 $SUCCESSOR-FN(x)$  は順序対  $(action, successor)$  の集合を返す。この順序対に含まれる各  $action$  は状態  $x$  でとることのできる行為 (legal action) の一つであり、各  $successor$  はその行為を取ったときに  $x$  から達することのできる状態である。たとえば、状態  $In(Arad)$  では、ルーマニアの問題の後者関数は次のとおり返すことになる。

$\{(Go(Sibiu), In(Sibiu)), (Go(Timisoara), In(Timisoara)), (Go(Zerind), In(Zerind))\}$

初期状態と後者関数の二つによつて、問題の状態空間 (state space) が暗黙に定義される。すなわち、初期状態から到達できるすべての状態の集合である。状態空間は、状態をノードとし行為をノード間の辺とする (図 3.2 に示したルーマニアの地図は、各道路を両方向おののドライブという行為とみなせば、状態空間グラフとして解釈できる)。状態空間における経路 (path) とは、行為の列つながった状態の列である。

- ある状態がゴール状態かどうかを決定するゴール検査 (goal test)。取りうるゴール状態の集合が明示的に与えられる場合がある。この場合、ゴール検査は与えられた状態がその集合の要素であるかどうかを調べるだけでよい。ルーマニアでのエージェントのゴールは單一要素の集合  $\{In(Bucharest)\}$  である。ゴールは状態を明示的に記述する方法でなく抽象的な性質で規定される場合もある。たとえば、チェスではゴールは相手のキングが攻められ、逃れることのできない、いわゆる “チェックメイト” の状態に達することである。

- 各経路に数値をコストとして与える経路コスト (path cost) 関数。問題解決エージェントはその達成尺度に応じてコスト関数を選ぶ。Bucharest に行こうとしているエージェントでは時間が本質的であり、経路のコストにはその距離を km で与えるのが一つの考え方だろう。この章では経路のコストは経路内の個々の行為のコストの和で記述されると仮定する。状態  $x$  から状態  $y$  へ行く行為  $a$  を行うのにかかるコストをスケーリングする。

<sup>3</sup> 別の形式化には、ある状態から次の状態を生成するのに用いることのできるオペレータ (operator) の集合を用いる方法がある。

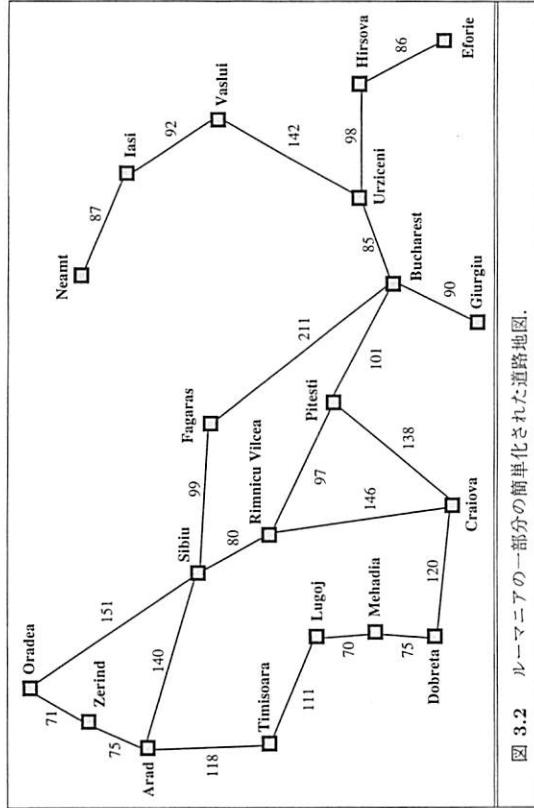


図 3.2 ルーマニアの一部分の簡単化された道路地図。

- ステップコスト (step cost) といい、 $c(x, a, y)$  と記述する。ルーマニア問題のステップコストは各道の距離であり、図 3.2 に示してある。ステップコストは非負であると仮定することになる。<sup>4</sup>
- これまでに与えた要素によって問題を定義することができます。また、これらを合わせて一つのデータ構造にすることで問題解決アルゴリズムへの入力として与えることができる。ある問題の解は初期状態からゴール状態への経路である。解の良さは経路コスト関数で測られる。最適解 (optimal solution) はすべての解の中で最も経路コストの小さい解である。

### 問題の定式化

前の節では、Bucharest へ行くという問題を、初期状態、後者関数、ゴール検査、経路コストで定式化した。この定式化は合理的に見えるが、現実世界の大変多くの面を省略している。状態の記述として我々が採用した  $In(Arad)$  と、現実の田舎旅行を比較してみよう。現実の旅行では、状態には非常に多くのことが含まれる。旅行仲間のことや、ラジオ放送のこと、窓の外の風景、とりしまりが近くにいるかどうか、次の休憩までどのくらいの距離があるか、道路の状態、天気などである。これらはどれも我々の状態記述に含まれない。Bucharest への道を抽象 (abstraction) とよぶ。

状態記述の抽象に加えて行為も抽象する必要がある。ドライブという行為にはたくさんの効果がある。自動車と乗っている人の位置を変えるだけでなく、時間と燃料を費したり、公害を増したり、エージェントを変えたりする（よく言うように、旅は人を大きくする）。我々の形式化では場所の変化のみを考える。他にも省略するたくさんを行

抽象

<sup>4</sup> 負のコストを認めた場合には、練習問題 3.17 で考える。

為がある。ラジオのスイッチを入れる、窓の外を見る、とりしまりのためにスピードを落とすなどである。もちろん、“ハンドルを3度左へ回す”などといったレベルで行為を特定したりしない。

もう少し正確に抽象の適切なレベルについて定義することはできるだろうか？ 詳細な世界状態の大まかな集合と、詳細な行為の列にそれぞれ対応するようには何が選ばれた状態と行為について考えよう。たとえば、AradからSibiu, Rimnicu Vilcea, Pitestiを通ってBucharestへといった経路のような、問題の抽象化された解を考えよう。この抽象的な解は大変な数の詳細な経路に対応する。たとえば、SibiuとRimnicu Vilceaの間をラジオをつけてドライブすることができるだろうし、その後はスイッチを切るかもしれない。任意の抽象的解を詳細な解の一つにつぶして開拓するとき、この抽象化は妥当(valid)であるといふ。十分条件はこうである。“Aradにいる”に対応するすべての詳細な状態に対する抽象化された解に含まれる行為の一つひとつを実行するよ

りも容易であれば抽象化は有用(useful)である。今の場合、平均的なドライバーエージェントにとってさりに探索、計画することなく実行できることから、十分に容易であると言える。良い抽象化は、したがって、妥当性を残しつつできるだけ詳細を取り除き、そして抽象化された行為が容易に実行できるものである。有用な抽象化をする能力がなかったなら、知的エージェントは現実世界に完全に飲み込まれてしまうだろう。

### 3.2 例題

問題解決のアプローチは非常に広範囲のタスク環境に適用されてきた。ここではよく知られたものを、おもちゃの問題(toy problem)と現実世界問題(real-world problem)に区別しながらすることにする。おもちゃの問題は、様々な問題解決方法を説明したり、練習することを意図するもので、簡潔かつ正確な記述で与えられる。つまり、おもちゃの問題はアルゴリズムの性能を比較するために異なった研究者が容易に用いることができる。現実世界問題はその解に人々が実際に興心をもつ問題である。現実世界問題は単一の合意された記述をもたないことが多いが、ここではその形式化について一般的な雰囲気を味わってもらえるよう試みる。

#### おもちゃの問題

はじめに取り上げる問題は2章で紹介した掃除機の世界(vacuum world)(図2.2参照)である。これは次のとおり形式化できる：

- ◇ 状態(state)：エージェントは二つの場所のうち一つをとる。おのおのの場所に埃があるかもしれないし、またはないかもしない。したがって、 $2 \times 2^2 = 8$ 通りの可能な世界状態がある。
- ◇ 初期状態(initial state)：任意の状態を初期状態として指定できる。
- ◇ 後者関数(successor function)：Left(左に動く), Right(右に動く), Suck(吸い込む)の三つの行為の結果である状態を生成する関数である。このときの完全な状

態空間を図3.3に示す。

- ◇ ゴール検査(goal test)：両方の場所に埃がないことを検査する。
  - ◇ 経路コスト(path cost)：各ステップはコスト1を要し、したがって経路コストは経路中のステップの個数である。
- 現実世界と比較すると、このおもちゃの問題は場所も埃も離散的であること、確実に掃除されること、一度掃除されると再び汚れることがないことなどの違いがある(3.6節でこれらの仮説を緩める)。注意すべき重要なことは、状態が、エージェントのいる場所と埃がどの場所にあるかによって決まるということである。n個の場所をもつ大きな環境では $n^{2^n}$ 個の状態をもつ。

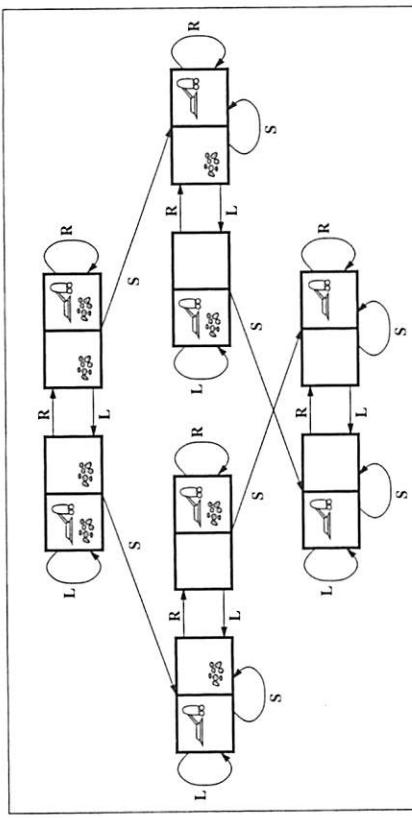


図 3.3 掃除機の世界の状態空間。矢は行為 L = Left, R = Right, S = Suck を表す。

8パズル

8パズル(8-puzzle)（例が図3.4に示されている）は、八つの番号づけられたタイルと一つの空白のスペースからなる3×3のボードから構成される。空白のスペースに隣接したタイルは、そのスペースにずらし入れることができる。目的は、たとえば図の右に示された配置のような特定の目標状態に到達することである。標準的な形式化は次のとおりである：

- ◇ 状態：各状態記述は、八つのタイルと一つの空白の場所が九つの区画のどこにあるかを指定する。
- ◇ 初期状態：任意の状態が初期状態として指定される。任意の目標状態を一つ決める」と、可能な初期状態のうち、ちょうど半分がその目標状態に到達できる（練習問題3.4）。
- ◇ 後者関数：四つの行為（空白を上下左右に動かす）の結果の状態を生成する。
- ◇ ゴール検査：状態が図3.4に示されたゴール状態（他のゴール状態でもよい）と一致するかどうかが検査する。
- ◇ 経路コスト：各ステップはコスト1を要し、経路のコストは経路中のステップの個数である。

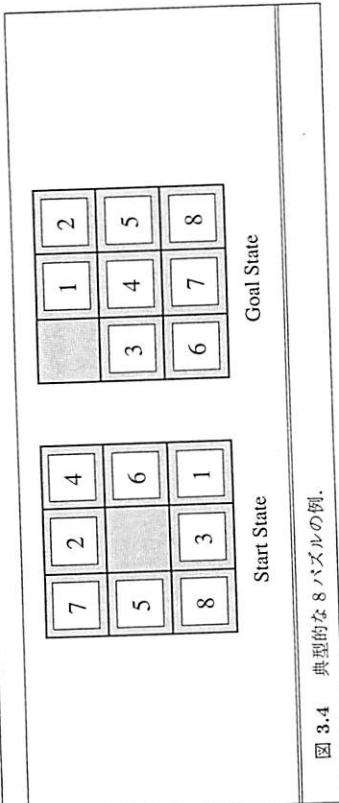


図 3.4 典型的な 8 パズルの例。

ここではどのような抽象化が行われたかを考えよう。行為はその始めと終わりの状態で抽象化されており、タイルをすらしている途中の位置は無視されている。タイルがひつかったときには盤を取り出したり、ナイフに引っかけてタイルを取り出してまた元に戻したりといった行為は抽象化によって考えない、物理的な操作の詳細はすべてなくなり、パズルのルールの記述が残されたわけである。

8 パズルは滑りブロックパズル (sliding-block puzzle) の族に属する。このパズルは AI の新しい探索アルゴリズムを試すための問題としてよく使われる。この一般的なクラスは NP 完全であることが知られている（参考問題 3.5）。一方で、最悪の場合にも本章あるいは次章で述べられる探索アルゴリズムに比べて、はつきりと良い結果を得る方法を見つけることは期待できない。8 パズルは  $9!/2 = 181,440$  通りの到達可能な状態があり、無作為に選ばれた例においては、(4 × 4 の盤上の) 15 パズルは 1.3 兆個の状態があり、無作為に選ばれた初期状態 (初期状態) の最適解を得るために最もも効率的なアルゴリズムを用いて数百万秒かかる。 $(5 \times 5$  の盤上の) 24 パズルには約  $10^{25}$  個の状態があり、無作為に選ばれた初期状態の最適解を得ることは現在の計算機とアルゴリズムを用いてもまだわめて困難である。

8 クイーン問題 (8-queens problem) の目標は、どのクイーンも他のクイーンを攻撃しないように、八つのクイーンをチェス盤に置くことである（クイーンは、同じ列、同じ行、同じ斜めの線上のコマを攻撃する）。図 3.5 は、試みたが失敗した解を示しており、右端の

列のクイーンが、左上のクイーンによって攻撃されている。この問題と  $n$  クイーンの族全体のための効率的な専用のアルゴリズムが存在するけれども、これらは依然探索アルゴリズムのためのおもしろいテスト問題である。主に 2 通りの定式化がある。漸増的定式化 (incremental formulation) は空の状態から始めて状態記述を増やすオペレータを用いる。8 クイーン問題では各行行為は一つのクイーンを状態に追加することになる。完全状態定式化 (complete-state formulation) は盤上に八つのすべてのクイーンがあるところから始め、それらを動かしていくものである。いずれの場合でも最終状態だけが問題なので、経路コストは重要ではない、一つ目の漸増的定式化では次のようになるだろう：

- ◇ 状態： 0 個から 8 個のクイーンの盤上への任意の配置
- ◇ 初期状態： クイーンが一つも置かれていない盤
- ◇ 後者関数： クイーンを一つ、空いたマスに置く。
- ◇ ゴール検査： 8 クイーンが盤上にあり、どれも攻撃しない。

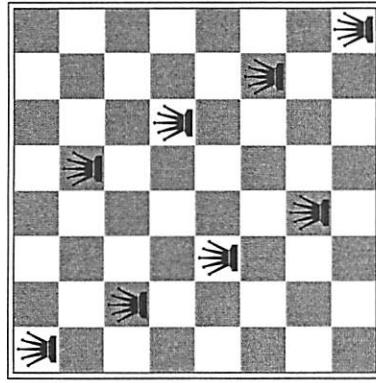


図 3.5 あと少しで解ける 8 クイーンパズル (完全な解は練習とする)。

この定式化では、 $64 \cdot 63 \cdots 57 \approx 3 \times 10^{14}$  通りの異なる列を調べなければならない。すでに攻撃しているマスにはクイーンを置かないようにするともう少し良い定式化が得られる：

- ◇ 状態： 一番左から  $n$  列 ( $0 \leq n \leq 8$ ) 目までに、 $n$  個のクイーンを 1 列に一つずつ置き、互いに攻撃しない配置が状態である。
- ◇ 後者関数： クイーンが置かれていない最も左の列で、他のどのクイーンにも攻撃されない任意のマスク一つにクイーンを加える。

この定式化は 8 クイーンの状態空間を  $3 \times 10^{14}$  個からたったの 2,057 個に減らし、解の発見は容易になる。他方、100 クイーンでは最初の定式化で約  $10^{400}$  個の状態があったのにに対し、改良された定式化でも約  $10^{52}$  個の状態がある（練習問題 3.5）。4 章では完全状態定式化を、5 章では 100 万 クイーン問題でも容易に解決できる単純なアルゴリズムを述べる。

### 現実世界問題

#### ルート発見問題

これまでにルート発見問題 (route-finding problem) を、ある場所の集合とそれらの間のリンクに沿った移動によってどのように定義したらよいかについてみてきた。ルート発見のアルゴリズムは様々な応用に用いられている。たとえば、計算機ネットワークにおけるルーティングや軍事作戦の計画、航空会社の運行計画システムである。これらの問題は多くの場合規定するのが難しい、航空会社の運行計画システムについて次のような簡単化された例を考えよう：

- ◇ 状態： 場所（たとえば、ある空港）と現在の時刻によって表現される。
- ◇ 初期状態： 問題に応じて特定される。
- ◇ 後者関数： 現在の時刻に空港内の移動時間を加えた時刻以降に発する、現在の空港から別の空港への定期航空便（その座席クラスやさらにその位置が与えられるごともある）のそれぞれについて、それらの便を利用してしたあととの状態が返される。
- ◇ ゴール検査： あらかじめ指定した時刻までに目的地に到着するかどうか。

### 完全状態定式化

#### 漸増的定式化

この問題によってどのようルート発見問題を解くかについてみてきた。ルート発見のアルゴリズムは様々な応用に用いられている。たとえば、計算機ネットワークにおけるルーティングや軍事作戦の計画、航空会社の運行計画システムである。これらの問題は多くの場合規定するのが難しい、航空会社の運行計画システムについて次のような簡単化された例を考えよう：

- ◇ 状態： 場所（たとえば、ある空港）と現在の時刻によって表現される。
- ◇ 初期状態： 問題に応じて特定される。
- ◇ 後者関数： 現在の時刻に空港内の移動時間を加えた時刻以降に発する、現在の空港から別の空港への定期航空便（その座席クラスやさらにその位置が与えられるごともある）のそれぞれについて、それらの便を利用してしたあととの状態が返される。
- ◇ ゴール検査： あらかじめ指定した時刻までに目的に到着するかどうか。

◇ 経路コスト：金銭的コスト、待ち時間、飛行時間、税関や入出国の手続き、座席の品質、出発時刻、機体のタイプ、乗客向けのマイレージ特典、などが関連する。商用の旅行支援システムはこうした種類の問題形式化を用いており、それに加えて各航空会社が課す複雑な航空料金を扱う必要がある。旅慣れた人なら誰でも知っているように、すべての旅が予定どおりにいくとは限らない。本当に良いシステムは偶発性に対する備えをもつ計画一つまり、代替選択についての予備の予約——についても、そのためのコストが元々の計画が失敗する可能性に見合う限りにおいて、考慮するべきである。

## 巡回問題

巡回問題 (touring problem) はルート発見問題と密接に関係するが、一方、重要な違いがある。たとえば、“Bucharest を出発し、図 3.2 にあるすべての都市を少なくとも 1 回ずつ回って Bucharest に戻りなさい”という問題を考えよう。経路発見問題と同様に、隣合つた都市の間の移動が行動に対応する。しかしながら状態空間はまったく異なる。各状態は現在の場所だけでなく、エージェントがすでに訪れた都市の集合を含む必要がある。したがって、初期状態は “Bucharest にいる；訪問済み (Bucharest, Urziceni, Vaslui)” であり、普通であれば状態 “Vaslui にいる；訪問済み {Bucharest, Urziceni, Vaslui}” を通過する。そして、ゴール検査はエージェントが Bucharest について、20 都市すべてを訪問したかどうかを調べることになる。

巡回セールスマン (セールスマン) 問題 (traveling salesperson problem: TSP) は各都市をちょうど一度ずつ回るようには制限された巡回問題である。目的は最短の巡回路を見つけることである。この問題は NP 困難であることが知られているが、TSP アルゴリズムの能力を改良する膨大な努力が払われてきた。セールスマシンの旅行計画の他に、これらのアルゴリズムは回路基板穿孔機や工場の収納機械の動作計画のような仕事のために使用してきた。

## VLSI レイアウト

VLSI レイアウト (VLSI layout) 問題では何百万の部品とその連結の位置を基板上に配置し、面積、回路の時間遅れ、浮遊容量 (stray capacity) を最小化し、製造収益を最大化することが要求される。このレイアウト問題は論理設計のあとのが工程で、二つの部分、セルレイアウト (cell layout) とチャネルルーティング (channel routing) に分けることができる。セルレイアウトではセルの一つひとつがそれぞの機能を果たすように回路の基本構成要素をセルの中にグループ分けする。各セルは決まつた足形 (サイズと形) をもち、他のセルに決まった数の接続が必要になる。目的はセルをチップ上で互いに重なり合わないよう、そしてセルの間に接続線のための余裕があるように配置することである。チャネルルーティングはセル間の隙間を通して各接続線の経路を見つけることである。これらの探索問題は非常に複雑であるが、確かに解く価値がある。4 章では、これらを解くことのできるアルゴリズムをいくつか調べる。

## ロボット操縦

ロボット操縦 (robot navigation) はすでに述べたルート発見問題の一般化である。ルートのような離散的な集合ではなく、ロボットは (原則的には) 可能な行為と状態の無限集合をもつ連續的な空間で動作する。平らな表面の上を動く周回ロボットでは、空間は本質的には 2 次元である。ロボットが腕や脚あるいは車輪をもち、これらを制御しなければならない場合は、探索空間は多次元になる。高度な技術によって探索空間を有限にすることが必要になる。これらの方のいくつかを 25 章で調べる。現実のロボットはセンサの読み取りや駆動制御の誤差を扱う必要がある。

## 自動組立ての順序付け

複雑な対象のロボットを用いた自動組立ての順序付け (automatic assembly sequencing)

は FREDDY ロボット (Michie, 1972) によって最初に実証された。それ以来の進展はゆっくりであったが確実に進んでおり、電機モータのような複雑な物体の組立てが採算が合うところまできている。組立て問題の目的はある物体の部品の組立ての順序を見つけることである。誤った順序が選ばれると、すでに終わった仕事を取り消さなければその後の部品を付け加えることができないといったことになる。ある順序の一つのステップが実行可能かどうかをチェックすることは、ロボット操縦と密接に関連した複雑な幾何学的探索問題である。言い換えれば、正しい後者を生成することが組立て順序付けてコストのかかる部分である。もう一つ別の重要な組立て問題はタンパク質設計 (protein design) である。その目的はある疾患に効果のあるふさわしい性質をもつ 3 次元構造のタンパク質に量み込まれるアミノ酸系列を見つけることである。

近年、インターネット探索 (internet search) によって、質問の答えや関連する情報を探したり、商業取引をするソフトウェアロボットへの要求が高まっている。インターネットはノード (ページ) とそれらの間のリンクのグラフとして容易に形式化することができます。そのため、これらは探索技術のよい応用である。インターネット探索についての完全な記述は 10 章まで待てほし。

## 3.3 解の探索

## 探索木

いくつかの問題の定式化を行ってきたが、今度はこれを解かなければならぬ。これには状態空間の探索を行う。本章では状態空間を定義する初期状態と後者閑数で生成される探索木 (search tree) を明示的に用いる探索の技術について扱う。一般には、複数の経路によって同じ状態に達することができる場合には探索木でなく、探索グラフが得られることがある。この複雑化は重要であり、3.5 節まで考えるのを待つことにする。

図 3.6 は Arad から Bucharest までの経路を見つけるために探索木を展開した様子を示す。探索木の根は初期状態  $In(Arad)$  に対応する探索ノード (search node) である。最初のステップはこれがゴール状態であるかどうかを検査する。明らかにゴールではないが、“Arad から出発して Arad まで行け”といつたトリッキーな問題を解くためにこれを確認することは重要である。これはゴール状態でないので、他の状態を考える必要がある。そこで現在の状態を展開 (expand) する。すなわち、現在の状態に後者閑数を適用し、新しい状態集合を生成 (generate) する。この場合、三つの新しい状態、 $In(Sibiu)$ ,  $In(Timisoara)$ ,  $In(Zerind)$  が得られる。ここで、これら三つのうちのいずれかを選んで、さらに考えなければならない。

まず一つの可能性を調べる。それで解決されなかつたときを選んだとして。この可能性を生き残置いておく——これが探索の本質である。最初に Sibiu に行くことを選んだとしたときにはこれがゴール状態かどうかを調べる (ゴールではない)。そこで、これを展開し  $In(Anad)$ ,  $In(Fagaras)$ ,  $In(Oradea)$ ,  $In(Rimnicu Vilcea)$  を得る。次にこれら四つのうち一つを選ぶか、あるいは戻って Timisoara か Zerind を選ぶことができる。選択、検査、展開を解が見つかるか、あるいはそれ以上展開されるべき状態がなくなるまで繰り返す。展開する状態の選択は探索戦略 (search strategy) によって決まる。一般的な木の探索アルゴリズムについて、図 3.7 に非形式的に記述した。

状態空間と探索木の区別は重要である。ルート探索問題では状態空間には各都市に対応

してたった 20 の状態しかないが、状態空間内の経路は無限個ある。したがって探索木は無限個のノードをもつことになる。たとえば、三つの経路 Arad-Sibiu, Arad-Sibiu-Arad, Arad-Sibiu-Arad-Sibiu は経路の最初の三つである（よい探索アルゴリズムは明らかに、こうした繰返しの経路を避ける。どのようにして避けるかは 3.5 節で述べる）。

ノードを表現する方法はたくさんあるが、ここでは次の五つの要素を含むデータ構造であることを仮定する：

- STATE (状態)：状態空間中のノードに対応する状態。
- PARENT-NODE (親ノード)：探索木の中で、このノードを生成したノード。
- ACTION (行為)：このノードを生成したとき親ノードに適用された行為。
- PATH-COST (経路コスト)：初期状態からこのノードへの経路のコストで、伝統的に  $g(n)$  と表記する。この経路は親へのポインタで表わされる。

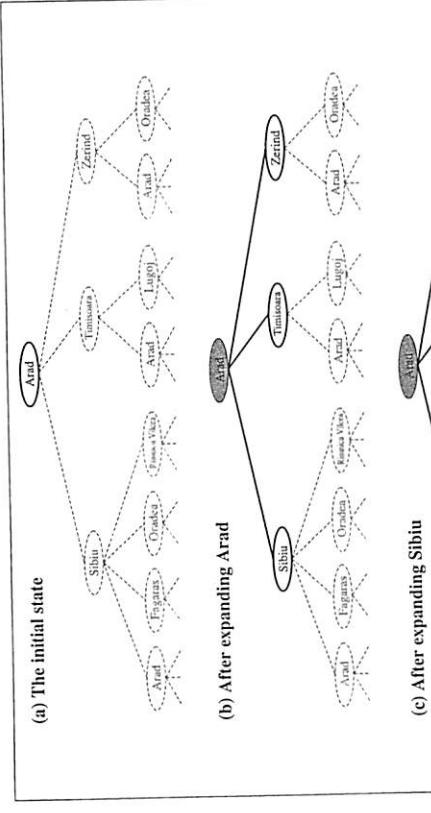


図 3.6 Arad から Bucharest までの経路を見つけるための、部分的な探索木。展開されたノードは影をつけており、生成されたがまだ展開されていないノードは破線で示している。まだ生成されていないノードは破線で示している。

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

図 3.7 一般的な探索アルゴリズムの非形式的な記述。

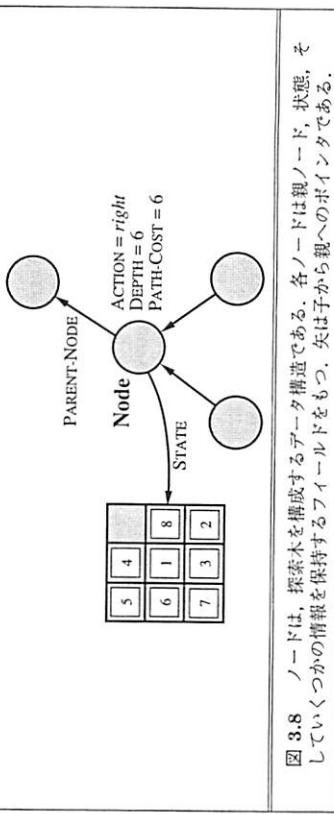


図 3.8 ノードは、探索木を構成するデータ構造である。各ノードは親ノード、状態、そしていくつかの情報を保持するフィールドをもつ。矢は子から親へのポインタである。

- DEPTH (深さ)：初期状態から経路に沿ってそのノードへ達するまでのステップ数。ノードと状態の区別を忘れないことは重要である。ノードは探索木を表現するためのデータ構造である。状態は世界の状況に対応する。つまり、ノードは PARENT-NODE ポイントで定義される特定の経路上にあるが、状態はそうではない。さらに、一つの状態が二つの異なる探索経路で生成される場合、二つの異なるノードが同じ世界状態を含んでいることになる。ノードのデータ構造を図 3.8 に図示する。
- 生成されたが、まだ展開されていないノードの集まり——これを縁 (fringe) という——を表現する必要がある。縁の各要素は葉ノード (leaf node)，すなわち木の中で後者をもないノードである。図 3.6 の各木の縁は輪郭を大綱で表したノードからなる。縁の最も簡単な表現はノードの集合である。すると探索戦略は、次に展開されるべきノードをこの集合から選択する関数ということになる。これは概念的に直接的であるが、計算量的にはコストが大きい。なぜなら、この戦略閾値は最良のノードを選択するために集合のすべての要素を見なければならない。そこで、ここではノードの集まりを待ち行列 (queue) で実装することを仮定する。待ち行列には次の操作がある：

- MAKE-QUEUE(*element, ...*) は与えられた要素をもつ待ち行列をつくる。
- EMPTY?(*queue*) は待ち行列に要素が一つもない場合に限り真を返す。
- FIRST(*queue*) は待ち行列の先頭要素を返す。
- REMOVE-FIRST(*queue*) は FIRST(*queue*) を返し、その待ち行列からそれを（先頭要素）を取り除く。
- INSERT(*element, queue*) は要素を待ち行列に加え、できた待ち行列を返す（異なった順序で要素を加える異なる異なったタイプの待ち行列を後で説明する）。
- INSERT-ALL(*elements, queue*) は要素の集合を待ち行列に追加し、できた待ち行列を返す。

これらの定義を用いて、一般的な木の探索アルゴリズムをより形式的に書くことができる。これを図 3.9 に示す。

```

function TREE-SEARCH(problem,fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]),fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node  $\leftarrow$  REMOVE-FIRST(fringe)
    if GOAL-TEST(problem) applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe  $\leftarrow$  INSERT-ALL(EXPAND(node,problem),fringe)

  function EXPAND(node,problem) returns a set of nodes
    successors  $\leftarrow$  the empty set
    for each action,result in SUCCESSOR-FN[problem](STATE[node]) do
      s  $\leftarrow$  a new NODE
      STATE[s]  $\leftarrow$  result
      PARENT-NODE[s]  $\leftarrow$  node
      ACTION[s]  $\leftarrow$  action
      PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node,action,s)
      DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
      add s to successors
    return successors

```

図 3.9 一般的木を探査する TREE-SEARCH アルゴリズム (引数 fringe (様) は空の待ち行列とする必要があること、待ち行列の種類が探索の順序に影響することに注意して欲しい)。関数 SOLUTION は親ボイントを根までたどって得られる行為の列を返す。

#### 問題解決の達成度に関する尺度

問題解決アルゴリズムの出力は failureか、あるいは一つの解である（無限のループに入り込んで、出力を何も返さないアルゴリズムもある）。アルゴリズムの達成度を次の 4通りで評価する：

- ◊ 完全性 (completeness)：そのアルゴリズムは、解が存在する場合は解を見つけることを保証しているか。
- ◊ 最適性 (optimality)：その戦略は 63 ページに定義されている意味で最適な解を見つけるか。
- ◊ 時間複雑さ (time complexity)：探索を達成するためにどの程度の時間が必要か。
- ◊ 空間複雑さ (space complexity)：探索を達成するためにどの程度の記憶領域が必要か。

時間と空間の複雑さは、常に問題の困難さについての何らかの尺度に関して考えられる。理論計算科学では、典型的な尺度は状態グラフのサイズである（ルーマニアの探索プログラムに入力される明示的なデータ構造とみなせるからである）。なぜなら、このグラフ地図はその一例である）。人工知能では、こうしたグラフは初期状態と後者間数によって暗黙的に表現され、しばしば無限である。そこで、複雑さは次の三つの量の関数で表現される。すなわち、任意のノードの後のノードの数の最大値である分枝度 (branching factor) *b*、最も浅いノードのノードの数 *d*、状態空間中の経路の最大の長さ *m* の三つである。

#### 3.4 情報のない探索戦略

時間は探索中に生成されたノード数で<sup>5</sup>、空間には記憶領域に保存される最大のノード数で計られることが多い。

探索アルゴリズムの効果について評価するには、探索コスト (search cost) のみを考えることができる。これは典型的には時間複雑さに依存するが、記憶領域の使用についても含めることができる。あるいは、探索コストと見つかった解の経路コストを組み合わせたトータルコスト (total cost) を考える。あるいは、探索コスト (total cost) は経路を見つける問題では、探索コストは探索に要する時間であり、解のコスト (経路コスト) の全長 (キロメートル) である。つまり、トータルコストを計算するためにはキロメートルとミリ秒を加えなければならない。この二つの間の正式な換算率はないが、この場合は自動車の平均速度を見積もることでキロメートルをミリ秒に換算するのが合理的であろう (なぜならエージェントが気にしているのは時間である)。これによってエージェントは、それ以上の計算でより短い経路を見つけてもかえって非生産的になるような最適のトレードオフを知ることができます。異なるものの間のトレードオフの一般的な問題については 16 章で取り上げる。

#### 3.4 情報のない探索戦略

この節では情報のない探索 (uninformed search) あるいは盲目的探索 (blind search) の見出しで五つの探索戦略を扱う。この言葉はそれらの探索が問題の定義で与えられる以外に状態についてのそれ以上の情報を持たないことを意味する。これらの探索ができることは、後者を生成することと、ゴールとゴールでない状態を区別することだけである。ゴールでない状態の一つが別のものよりも見込みがあるかどうかを知ることのできる戦略は、情報のある探索 (informed search) や、ヒューリスティック探索 (heuristic search) とよばれる。これらについては 4 章で扱う。探索戦略はどれも、ノードの展開される順序によって区別される。

##### 幅優先探索

幅優先探索 (breadth-first search) は、単純な戦略で、まず根ノードを開拓し、その後、根ノードの後者すべてを展開する、と続けるものである。一般に探索木のある深さのすべてのノードよりも前に展開される。幅優先探索は先入れ先出し (first-in first-out: FIFO) の待ち行列を縦に用い、これを空にして TREE-SEARCH を呼び出すことで実現できる。これによって先に探索されたノードが先に展開される。つまり、TREE-SEARCH(*problem*, FIFO-QUEUE()) とよぶことで幅優先探索となる。FIFO 待ち行列はすべての生成された後者を待ち行列の最後におく。このことは浅いノードは深いノードよりも先に展開されることを意味する。図 3.10 は単純な二分木での探索の進み具合を示している。

<sup>5</sup> 代わりに展開されるノードの数で時間計算量を計る教科書もある。これら二つの計り方の違いは萬々倍である。ノードの展開にかかる実行時間は生成されたノード数に伴って増加するようと考えられる。

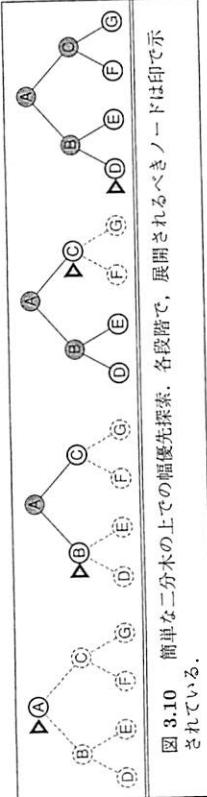


図 3.10 簡単な二分木の上の幅優先探索、各段階で、展開されるべきノードは印で示されている。

前節の四つの基準によって幅優先探索を評価してみよう。完全であることは容易にわかる。もし、最も浅いゴールがある有限の深さ  $d$  にあったとするならば、幅優先探索はそれよりも浅いすべてのノードを展開したあとでこのゴールを見つけることになる（ただし、分岐度  $b$  は有限とする）。最も浅いゴールノードが最適であるとは限らない、技術的に言うならば、経路コストがノードの深さに関する非減少関数であるならば幅優先探索は最適である（たとえば、すべての行為のコストは同じである場合がそうである）。

これまでのところ幅優先探索については良いことばかりであった。これが常に戦略として選択されない理由をみるために、探索を終えるまでに必要な時間と記憶領域について考える必要がある。このため、すべての状態が  $b$  個の後者をもつ仮想的な状態空間を考える。このとき探索木の根は最初のレベルで  $b$  個のノードを生成し、そのそれぞれのノードは  $b$  個のノードをさらに生成し、第二のレベル全体で  $b^2$  個となる。これらのそれぞれがさらに  $b$  個のノードを生成するので第三のレベルでは  $b^3$  個のノードになる、というように続く。ここで解が深さ  $d$  にあったとしよう。最悪の場合、レベル  $d$  の最後のノード以外のすべてを展開する（ゴール自身は展開されない）ことになり、レベル  $d+1$  の  $b^{d+1} - b$  個のノードを生成する。すると、生成されるノードの合計は、

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

である。生成されたノードは、緑にあるかあるいは緑のノードの先祖であるので、すべてが記憶領域に残ることになる。それゆえ空間複雑さは時間複雑さと同じ（正確には、ルートのためにノードが一つ多い）となる。

複雑さの解析をする人は  $O(b^{d+1})$  のような指數の複雑さを心配する（あるいは、挑戦好きな人ならば興奮する）。図 3.11 を見るとその理由がわかるだろう。分岐度  $b = 10$  で、解の深さ  $d$  を様々な値にした場合の幅優先探索に必要な時間と記憶領域を示す。表では 1 秒間に 10,000 個のノードを生成することができ、また、一つのノードのために 1000 バイトの記憶領域を必要とすると仮定した。多くの探索問題は、最近のハソナルコンビュータで実行したとき、おかげで（100 倍から 100 分の 1 程度の範囲で）これらの仮定に当てはまる。

図 3.11 からは二つの教訓を学ぶことができる。第一に、幅優先探索では実行時間よりも必要な記憶領域のほうが深刻である。重要な問題で深さ 8 の解を得るのに 31 時間待つことはそれほどの時間ではないが、そのときに必要なテラバイトの主記憶をもつコンピュータはほとんどない。幸い、記憶領域をこれほど必要としない他の探索戦略がある。

第二の教訓は時間の必要量はやはり大きな要因である。もし問題の解が深さ 12 にあるならば、（上の仮定の下）幅優先探索は（そして、実は情報のない探索はどれも）解を見つけるのに 35 年もの時間を必要とする。一般に指數の複雑さをもつ探索問題は、ごく小さな問題以外は情報のない探索では解くことができない。

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	$10^7$	19 minutes	10 gigabytes
8	$10^9$	31 hours	1 terabytes
10	$10^{11}$	129 days	101 terabytes
12	$10^{13}$	35 years	10 petabytes
14	$10^{15}$	3,523 years	1 exabyte

図 3.11 幅優先探索で必要とされる時間および領域、これらの数値は分岐度  $b = 10$ 、1 秒当たりの展開ノード数 10,000、ノード当たりの必要領域 1000 バイトを仮定した。

### 均一コスト探索

幅優先探索はつねに最も浅い未展開のノードを展開するので、ステップコストがすべて等しい場合には最適である。單純な並張りによって、任意のステップコスト閾数に対して最適なアルゴリズムを見つけることができる。最も浅いノードを展開する代わりに、均一コスト探索 (uniform-cost search) は経路コストが最も小さなノードを展開する。すべてのステップコストが等しい場合はこれは幅優先探索に一致する。

均一コスト探索は経路上のステップ数ではなく、全体のコストを見る。それゆえ、コスト 0 の行為をもつノードがあり、その行為の結果が同じ状態に戻ってしまう場合（たとえば、 $NoOp$  行為）、いつたんそのノードを展開すると無限ループに陥るだろう。もし、すべてのステップの行為がある小さな正の定数  $c$  以上であるなら完全性は保障される。この条件は、経路のコストが常にその経路に沿って進むにつれて増加する性質のためである。この性質のために、アルゴリズムは経路のコストが増加する順序でノードを展開することは容易にわかる。それゆえ、展開のために最初に選択されるノードは最適である (TREE-SEARCH は展開のためには選択したノードにだけゴール検査を適応することを意味する)。この条件は、経路のコストのために十分条件でもある。この条件は、経路のコストが常にその経路を最短経路を見つけるためにこのアルゴリズムを試してほしい。

均一コスト探索は経路の深さではなくコストによってガイドされている。そのためその複雑さを  $b$  と  $d$  で特徴づけることは容易でない。代わりに  $C^*$  を最適なコストとし、すべての行為が少なくとも  $c$  のコストをもつことを仮定する。するとアルゴリズムの最悪の時間と領域の複雑さは  $O(b(b + (C^*/c))$  である。これは  $b^d$  よりもずっと大きい。なぜならば均一コスト探索は長くておそらく有用なステップを探索する前に少ないステップ数の大きな木を探索することがあります。しばしば実際そうする。すべてのステップコストが等しい場合はもちろん  $b^{1+(C^*/c)}$  はちょうど  $b^d$  である。

### 深さ優先探索

深さ優先探索 (depth-first search) は探索木の現在の縁にある最も深いノードを開く。探索の進み具合を図 3.12 に示した。この探索では後者をもたないノードまで、探索木をまっすぐに最も深いレベルにまで進む。そうしたノードは展開されると線からなくなっていくので、探索はまだ展開されていない後者がある次に深いノードに戻る。

### 深さ優先探索

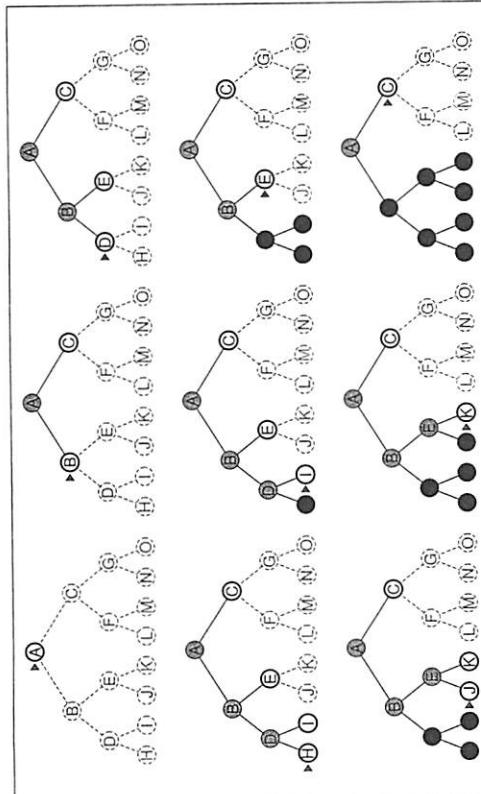


図 3.12 二分木の上の深さ優先探索。展開され、そして縁に子孫がないノードはメモりから取り除かれる。そうしたノードは黒く塗りつぶされている。深さ 3 のノードは後者をもたないこと、M が唯一のゴールノードであることを仮定した。

この戦略は、スタックとして知られる後入れ先出し (last-in-first-out: LIFO) の待ち行列を TREE-SEARCH に使うことで実現することができる。別の TREE-SEARCH の実現法としては、各子供に対して順番にそれ自身を呼び出す再帰関数をもつ深さ優先探索が一般的である（深さに制限をもつ再帰的な深さ優先アルゴリズムを図 3.13 に示す）。

深さ優先探索は記憶領域の要求がきわめて低い、根から葉ノードへのたった一つの経路のみを、その経路上の各ノードのまだ展開されていない兄弟ノードとともに保存すればよい。いったん展開されたノードは、その子孫がすべて完全に探索されると直ちに、記憶領域から削除することができる（図 3.12 参照）。分辨率  $b$ 、最大深さ  $m$  の状態空間に対して深さ優先探索は  $bm + 1$  個のノード分だけの記憶領域が必要とされる。図 3.11 と同じ仮定をもち、ゴールノードと同じ深さのノードは後者をもないと仮定したとき、深さ優先探索は深さ  $d = 12$  で 118 キロバイトを必要とし、これは 10 ベタバイトに対して 100 億倍の領域が稼げる。

深さ優先探索の一つのバリエーションである後戻り探索 (backtracking search) は、用いる記憶領域がさりに少ないので、後戻り探索では、すべての後者ではなく、ただ一つの後者のみを生成する。部分的に展開されたノードには次に生成する後者を記憶する。このため、 $O(bm)$  ではなく、 $O(m)$  の記憶領域で済む。後戻り探索は記憶領域（と時間についても）を節約するためのもう一つ別の仕掛けを備えている。最初に現在の状態をコピーする代わ

### 深さ制限探索

#### 深さ制限探索

りに、その記述を直接修正して後者を生成する。これによって必要な記憶領域は一つの状態記述と  $O(m)$  個の行為を記憶するための領域だけとなる。これが機能するためには、次の後者を生成するときに、各修正を元に戻すことができなければならない。ロボットによる組立て問題のように大きな状態記述をもつ問題の場合には、これらの技術が成否を大きく左右する。

深さ優先探索の欠点は、選択によっては探索木の根に近いところで解がある場合も、間違った選択をすると、大変長い（場合によっては無限）経路にはまりこむ可能性があることである。たとえば、図 3.12 では深さ優先探索は  $C$  がゴールノードであっても、左の部分木のすべてを探索することになる。さらにノード  $J$  もゴールノードである場合は、深さ優先探索はこの  $J$  を解として返す。つまり、深さ優先探索は最適性をもたない、左の部分木の深さに限界がなく、そこに解がない場合は、深さ優先探索は停止せず、したがって完全性ももたない、最悪の場合、深さ優先探索は探索木に含まれる  $O(b^m)$  個のノードすべてを生成する。ここで  $m$  は任意のノードの最大の深さである。 $m$  は  $d$  (=最も浅いところにある解の深さ) よりもずっと大きくなり、木に限界がない場合には無限大になることに注意してほしい。

木に限界がない場合の問題は、あらかじめ深さの限界  $\ell$  を定めた深さ優先探索によつてある程度解決される。つまり、深さ  $\ell$  のノードは後者をもたないものとして扱われる。この方法は深さ制限探索 (depth-limited search) と呼ばれる。深さの限界は無限の経路の問題を解決する。しかし残念ながら、 $\ell < d$  と選択した場合、すなわち、最も浅いゴールが深さ  $\ell = 19$  とするのは一つの選択である。しかし、実際にには地図を注意深くみると、この都市から別の都市へ高々 9 ステップで行けることに気づくだろう。この数は状態空間の直径 (diameter) として知られ、深さの限界としてよい選択であり、深さ制限探索はより効率が良くなる。しかし、多くの問題では問題を解かない限り良い深さの限界について知ることはできない。

深さの限界が問題についての知識に基づく場合もある。たとえば、ルーマニアの地図には 20 の都市がある。したがって、解があるならば最長でも長さ 19 であり、このときは 20 の都市がある。しかし、解があるならば最長でも長さ 19 であり、このときは  $\ell = 19$  とするのは一つの選択である。しかし、実際には地図を注意深くみると、この都市から別の都市へ高々 9 ステップで行けることに気づくだろう。この数は状態空間の直径 (diameter) として知られ、深さの限界としてよい選択であり、深さ制限探索はより効率が良くなる。しかし、多くの問題では問題を解かない限り良い深さの限界について知ることはできない。

深さ制限探索は一般的の木探索アルゴリズムや深さ優先探索の再帰アルゴリズムを少し修正することで実装できる。図 3.13 に再帰的な深さ制限探索の擬似コードを示す。深さ制限探索は 2 種類の失敗で停止する可能性がある。通常の失敗である値 failure は解がなかったことを表し、値 cutoff は深さの限度以内に解がなかったことを示す。

### 反復深化深さ優先探索

反復深化探索 (iterative deepening search) (あるいは反復深化深さ優先探索) は、深さ優先探索と組み合わせて使われる一般的な戦略で、最良の深さ制限を見つける。深さの制限

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    cutoff_occurred? ← false
    if GOAL-TEST[node] = limit then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure

```

図 3.13 深さ制限探索の再帰を用いた実装。

をはじめ 0 にしておき、次に 1、そして 2、とゴールが見つかるまで次第に増やしていく。ゴールは、深さの制限が最も浅いゴールの深さ  $d$  に達したときに見つかることを示す。アルゴリズムを図 3.14 に示す。反復深化探索と幅優先探索の利点をあわせもつている。深さ優先探索と同様に必要な記憶領域は大変少なく、正確には  $O(bd)$  である。また、幅優先探索と同じように分枝度が有限であれば完全であり、経路コストがノードの深さの非減少関数であれば最適である。図 3.15 は二分木の反復深化探索での 4 回の繰返しを示している。

反復深化探索は、状態を何度も生成しており、無駄が多いようには思われる。ところがこのことは、それはほど大きな犠牲ではないことがわかる。探索木が各深さで同じ（あるいは、およそ同じ）分枝度をもつのならば、ほとんどのノードが一番下にあることになり、何度も上のレベルのノードを生成することは大した問題ではないことになる。反復深化探索では、最も下（深さ  $d$ ）のノードは一度だけ生成され、一つ上のレベルは 2 回、と続き、根の子は  $d$  回生成される。したがって生成されるノードの個数の合計は、

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

となり、時間複雑さは  $O(b^d)$  である。幅優先探索が生成するノード数と比較してみよう。

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

幅優先探索は深さ  $d+1$  のノードもいくつか生成するが、反復深化探索は生成しないことに注意する必要がある。結果として反復深化探索は、状態を何度も生成するにもかかわらずに成功する必要がある。

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
inputs: problem, a problem
for depth ← 0 to  $\infty$  do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

```

図 3.14 反復深化探索アルゴリズム。順に深さ制限を増しながら、深さ制限探索を繰り返している。解が見つかったとき、あるいは、深さ制限探索が解がないことを意味する *failure* を返したときに停止する。

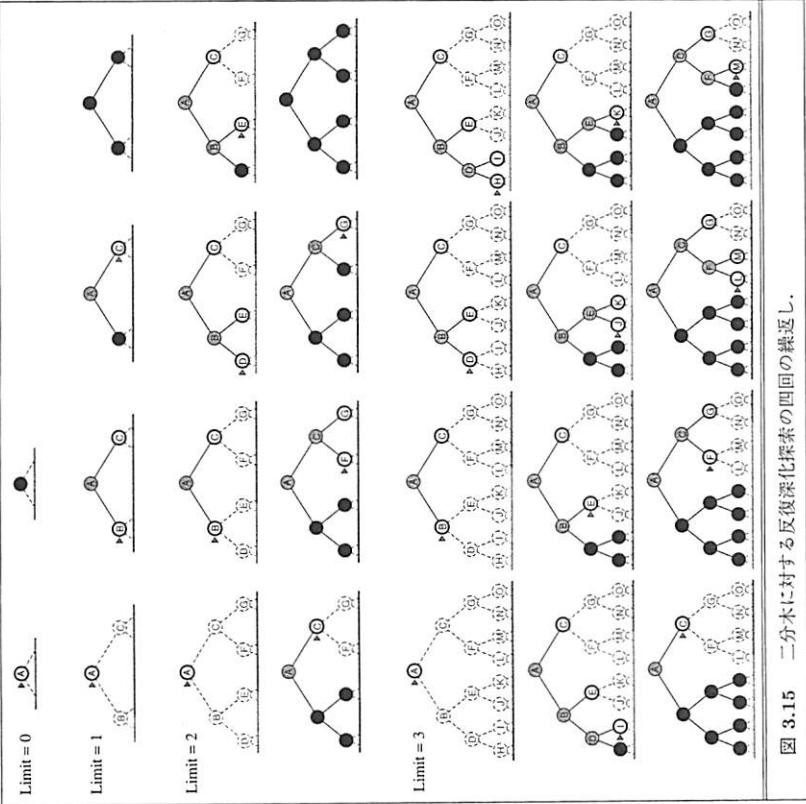


図 3.13 深さ制限探索の再帰を用いた実装。  
図 3.14 反復深化探索アルゴリズム。順に深さ制限を増しながら、深さ制限探索を繰り返している。解が見つかったとき、あるいは、深さ制限探索が解がないことを意味する *failure* を返したときに停止する。

反復深化探索は、各繰返しで新しいノードの層を完全に探索し、その後に次の層に進む点で幅優先探索と類似している。均一コスト探索についても、その最適性の保証を継承しつつ、記憶領域についての要求を避けられることのできる反復版を考えるのも価値のあることがあるように思われる。アイデアは深さの限界を増していく代わりに、経路コストの限界を増していくことである。反復延長探索 (iterative lengthening search) とよぶこの探索アルゴリズムは練習問題 3.11 で検討する。残念ながら、反復延長は均一コスト探索に比べて本質的なオーバーヘッドを招くことになるのがわかる。

反復延長探索



図 3.15 二分木に対する反復深化探索の四回の繰返し。  
図 3.15 二分木に対する反復深化探索の四回の繰返し。

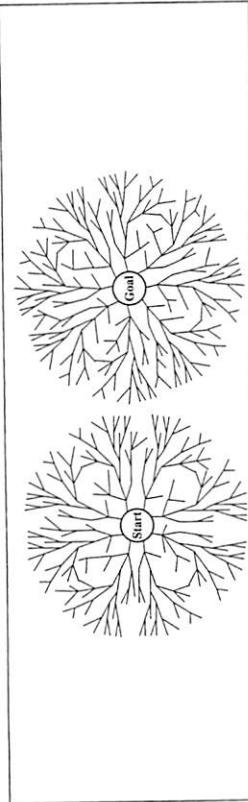


図 3.16 まもなく、Start のノードからの枝が Goal のノードと出会い、成功しようとしている双方向探索の模式図

### 双方向探索

双方向探索のアイデアは二つの方向からの探索を同時にを行うことである。一つは初期状態から前方へもう一つはゴールから後方へ進み、二つの探索が真ん中で出合ったときに停止する（図 3.16）。 $b^{d/2} + b^{d/2}$  は  $b^d$  よりもずっと小さくなる理由である。すなわち、図の二つの小さな円の面積は Start を中心に Goal に達するほどの大きな円の面積よりも小さい。

双方向探索は、二つの探索の両方あるいは片方がそのノードを展開する前にもう一方の探索木の累に入っているかどうかを調べる機能をもつことで実現できる。ノードが他方の探索木に入っているときには、ある問題が深さ  $d = 6$  に解をもち、それぞれの方向の探索を幅優先探索で1回に一つずつ探索するとき、最終の場合に各ノードを生成することになる。各ノードが他方の探索木に含まれるかどうかの確認はハッシュ表を用いることで定数時間で実行でき、したがって、双方向探索の時間複雑さは  $O(b^{d/2})$  である。少なくとも一方の探索木は、ノードが含まれるかどうかの確認のために記憶しておかなければならぬので、領域複雑さも  $O(b^{d/2})$  となる。この領域の必要量が双方向探索の最も重大な欠点である。アルゴリズムは、両方向の探索に横型探索を行つた場合、完全で（そして均一なステップコストであれば）最適である。他の組合せの場合は完全性、最適性のどちらか一方、あるいは双方が失われる。

時間複雑さを抑えることのできる双方向探索は魅力的である。しかし後向き探索はどのように実現できるだろうか。これは思ったよりも難しい。ノード  $n$  を後者にもつすべてのノードを  $n$  の前者（predecessor）と言い、 $Pred(n)$  と書くことにする。双方向探索は  $Pred(n)$  が効率的に計算できる必要がある。状態空間中のすべての行為が反映可能な場合、すなわち  $Pred(n) = Succ(n)$  の場合は最も簡単である。他の場合は本質的な工夫が必要になる。

ゴールから探索するという場合のゴールが一体何を意味するかが問題である。しかしながらマニアでの経路探索ではゴール状態はただ一つであるので、後向き探索は前向き探索ほとんどまったく同じである。明示的に列挙できるゴールが複数個ある場合、たとえば、図 3.3 の二つの壁のないゴール状態のような場合、ダミーのゴールを一つ作り、その前者が実際のゴール状態すべてであるようにすることができる。別の方法としては、ゴール状態の集合を单一の状態とみなすことや冗長なノード生成を避けようというものがある。

### 情報のない探索戦略の比較

図 3.17 に 3.4 節で述べた四つの評価基準にしたがった探索戦略の比較をあげた。

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>a</sup>	Yes <sup>a,b</sup>	No	No	Yes <sup>a</sup>	Yes <sup>a,d</sup>
Time	$O(b^{d+1})$	$O(b^{1+[C^*/\epsilon]})$	$O(b^m)$	$O(b^d)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+[C^*/\epsilon]})$	$O(bm)$	$O(bt)$	$O(bt)$	$O(b^{d/2})$
Optimal?	Yes <sup>c</sup>	Yes <sup>c</sup>	No	No	Yes <sup>c</sup>	Yes <sup>c,d</sup>

図 3.17 探索戦略の評価。 $b$  は分岐度、 $d$  は最も浅い解の深さ、 $m$  は探索木の最大の深さ、<sup>a</sup> は深さ制限である。上付き文字の<sup>a</sup> は有限であれば完全であること、<sup>b</sup> はステップコストがすべて同一であれば最適であること、<sup>c</sup> は両方向の探索に幅優先探索を用いたときの結果であることをそれぞれ意味する。

### 3.5 繰返し状態の回避

これまでのところ、探索プロセスの最も重要な困難の一つを完全に無視してきた。すなわち、すでに現れて展開された状態を再び展開し、時間を無駄にする可能性である。問題によってはこうしたこととは絶対に起きないこともある。状態空間が木であり各状態への経路が唯一つであるような場合である。8 ケイーン問題の効率よい定式化（すなわち、新しいケイーンがケイーンの置かれていない最も左の列に置かれる方法）が効率の良い理由の大部分はこれである。このとき、どの状態もただ一つの経路に沿つてのみ達することができる。もし 8 ケイーン問題をクイーンをどの列においてもよいという形式で定式化するなら、 $n$  個のケイーンを含む状態はそれぞれ  $n!$  通りの経路によって達することができてしまう。状態の繰返しが避けられない問題もある。経路発見問題や滑りプロックパズルのような行為を反転できる問題はすべてそうである。これらの問題の探索木は無限である。しかし、もし繰り返される状態のいくつかを刈ることができるれば、状態空間グラフの全域木となる部分だけを生成するように、探索木を有限サイズにすることができる。ある決まった深さまでの探索木を考えると、状態の繰返しを削除することで探索コストを指數的に削減できる場合があることは容易にわかる。極端な場合、サイズが  $d + 1$  の状態空間（図 3.18 (a)）は  $2^d$  個の葉をもつ探索木（図 3.18 (b)）となる。もう少し現実的な例は図 3.18 (c) に示さ

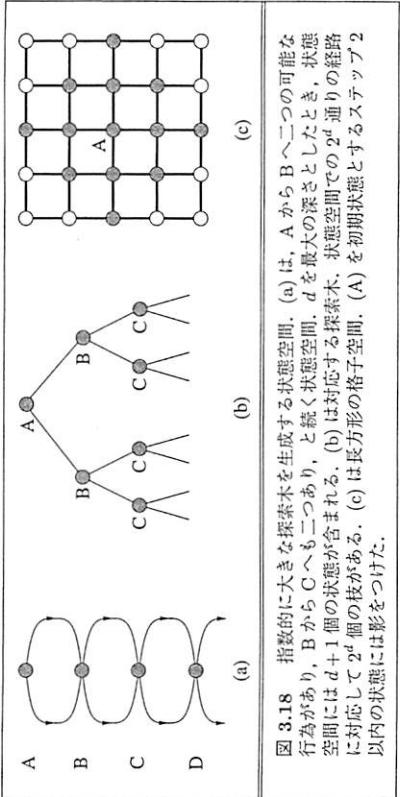


図 3.18 指数的に大きな探索木を生成する状態空間。 (a) は、A から B へ二つの可能な行ががあり、B から C へも二つあり、と続く状態空間。 $d$  を最大の深さとしたとき、状態空間には  $d+1$  個の状態が含まれる。(b) は対応する探索木、(c) は長方形の格子空間。(A) を初期状態とするステップ 2 以内の状態には影をつけた。

長方形格子

されているような長方形格子 (rectangular grid) である。格子では各状態は四つの後者をもち、状態の繰返しを含む探索木は  $4^d$  個の葉をもつ。しかし、ある状態から  $d$  ステップの範囲にあるのは約  $d+1$  個の状態のみである。 $d=20$  では約一兆個ものノードとなる一方、状態数は約 800 である。

状態の繰返しはアルゴリズムがそれを検知しない場合、解くことのできる状態とすでに展開された状態を比較することである。もし一致が見つかればアルゴリズムは同じ状態への二つの経路を見つけたことになり、そのうちの一つを捨てることができる。

深さ優先探索では記憶しているノードは根から現在のノードまでの経路上のものだけである。この記憶しているノードと現在のノードを比較することでアルゴリズムは経路のループを検知することができ、そのような経路はただちに捨てることができる。こうすることでも有限の状態空間がループのために無限の探索木にならないことを保証するのに有用である。しかし、残念ながら図 3.18 のような問題ではループにならない経路であっても指數的な増殖を避けることはできない。それを避けるにはもう少し多くのノードを記憶するしかない。時間と空間の根本的なトレードオフがある。過去を忘れてしまうアルゴリズムは繰返しの運命にある。

アルゴリズムが訪れた状態をすべて記憶しているならば、状態空間を直接探索しているとみなすことができる。一般の TREE-SEARCH アルゴリズムに、展開したすべてのノードを保存する closed list と呼ばれるデータ構造を含めるように修正することができます(未展開のノードである縁は open list とよばれることもある)。現在のノードが closed list の中のあるノードに一致した場合、それを展開せずに捨てる。この新しいアルゴリズムは GRAPH-SEARCH とよばれる(図 3.19)。状態の繰返しが多い問題では GRAPH-SEARCH は TREE-SEARCH よりもずっと効率が良い、最悪の場合の時間と領域の必要量は状態空間のサイズに比例する。これは  $O(b^d)$  よりも非常に小さい可能性がある。

グラフの探索は最適性に問題がある。すでに述べたように状態の繰返しが見つかったとき、アルゴリズムは同じ状態への二つの経路を見つけることになる。図 3.19 の GRAPH-SEARCH アルゴリズムはいつも新しく見つけた経路を捨てる。もし新しく見つけた経路が元のものよりも短い場合には、明らかに最適な解を失うことになる。幸い、均一コスト探索や定数

```

function GRAPH-SEARCH(problem,fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE([INITIAL-STATE[problem]]) ,fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL?-TEST(problem)[STATE[node]] then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node,problem),fringe)

```

図 3.19 一般的なグラフを探索する GRAPH-SEARCH のアルゴリズム。集合 *closed* は、状態の繰返しを効率よく検査できるハッシュ表を用いて実装することができる。このアルゴリズムでは状態 *s* への最初の経路が最もコストが低いことを仮定した(本文を見よ)。

ステップコストの状況で幅優先探索を使つた場合にはこうしたことにはならないことが分かる(練習問題 3.12)。そのため、これら二つの最適な木の探索戦略は最適なグラフの探索戦略でもある。他方、反復深化探索は深さ優先展開していくので最適な経路を見つける前に最適ではない経路にそつてノードを見つけることが起きやすい。そのため、反復深化を使ったグラフの探索は、あるノードへの新しく見つけた経路が元のものよりも良いかどうかを調べ、もし良いならばそのノードの子孫すべての深さと経路コストを修正する必要がある。

closed list を用いるということは、深さ優先探索や反復深化探索はもはや線形の領域複雑ではなくなるということに注意する必要がある。GRAPH-SEARCH アルゴリズムはすべてのノードを記憶しておくので、探索によっては記憶の制限のために実現不可能になる場合がある。

### 3.6 部分的知識をもつ探索

3.3 節では、環境が完全に観測できること、決定的であること、そしてエージェントが各行為のもたらす効果を知つていていることを仮定した。それゆえ、エージェントはどんな行為の系列からでも、そこから結論される状態を正確に計算することができ、そして今いる状態をいつでも知っている。行為を行つたあとどの知識をもたらさない。状態や行為についての知識が不完全であるとどのようなことが起きるだろうか。不完全さの違いに応じて三種類の異なる問題があることがある:

- センサなし問題(sensorless problem)(あるいは、順応問題(conformant problem)) : エージェントがセンサをまったくもたない場合、(わかっている範囲で) いくつかのエージェントがセンサをもつた場合、(わかつてない範囲で) いくつかのエージェントが初期状態のうちの一つにいることになる。行為は、それゆえ可能な後者状態の一つに導くことになる。
- 偶然性問題(contingency problem) : 環境が部分的にのみ観測できるか、あるいは行為が不確定な場合、各行為のとのエージェントの知覚は新しい情報を与える。可能

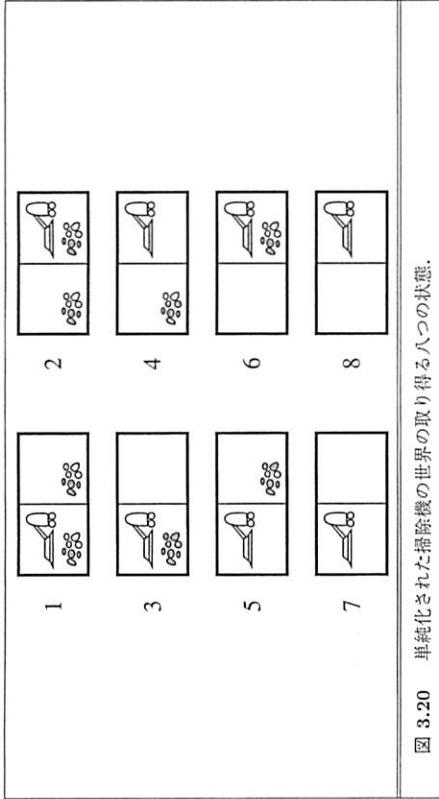


図 3.20 単純化された掃除機の世界の取り得る八つの状態。

な知覚のそれぞれはそれに備えて計画をたてるべき偶発事象の範囲を限定する。不確実さが他のエージェントの行為に起因する場合、エージェントはそれらを発見するために行動しなければならない。探査問題は敵対的 (adversarial) であるとよばれる。

3. 探査問題 (exploration problem)：環境中の状態と行為が未知の場合、エージェントはそれらを発見するために行動しなければならない。探査問題は偶然性問題の極端な場合であると見ることができる。

例として掃除機の世界の環境を用いる。図 3.20 にあるように状態空間には八つの状態があることを思い出してください。行為は *Left*, *Right*, *Suck* の三つで、ゴールはすべての埃を吸い取ること（すなわち状態 7 または 8 にすること）である。もし環境が観測可能であり、決定的であり、そして完全に既知であるならば問題は明らかにこれまでに述べてきたどのアルゴリズムを使っても解くことができる。たとえば初期状態が 5 であるならば、行為列 [*Right*, *Suck*] でゴール状態 8 に達することができ、この節の残りの部分では、この問題がセンサなし問題と偶然性問題である場合について考える。探査問題については 4.5 節で、敵対問題については 6 章で扱う。

### センサなし問題

掃除機エージェントが行為がもたらす効果はすべて知っているが、センサをもたない場合を考えよう。すると、エージェントは初期状態が集合 {1, 2, 3, 4, 5, 6, 7, 8} のうちの一つであることしか知らない。エージェントの状況は望みがないと思われるかもしれない。しかし、実際ににはきわめてうまくいく。エージェントの状況は行為が何をもたらすかを知っているので、たとえば行為 *Right* の結果、{2, 4, 6, 8} のいずれかの状態になること、行為列 [*Right*, *Suck*] の結果、{4, 8} のどちらかで終わることを計算できる。そして最終的に列 [*Right*, *Suck*, *Left*, *Suck*] は、始めた状態が何であろうと状態 7 に達することを保証する。エージェントは自身がどの状態から開始したかを知らないでも、世界を状態 7 に強制する (*coerce*) ことができる。要するに、世界が完全には知覚できないとき、エージェントは単一の状態ではなく、それが達する可能性のある状態の集合を推論する必要がある。このような状態の集合をを

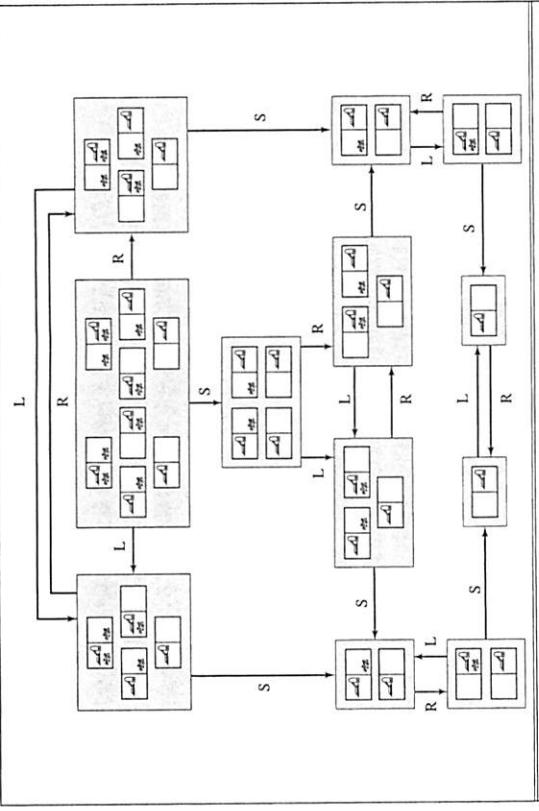


図 3.21 決定的で、センサなしの掃除機の世界での到達可能な範囲の信念状態空間。影つきの箱のおのが一つの信念状態である。どの場所においてもエージェントは特定の信念状態に入るのではなく、その中のどの物理状態に入るのは知らない。初期信念状態（まったく知らない状態）は最上部の中央の箱である。行為はラベル付の矢で示した。自己ループは簡潔にするため、省略した。

信念状態 (belief state) とよぶ。これは現在いる可能性のある物理的状態についてのエージェントの信念を表す（完全に観測できる環境では、各信念は單一の物理的状態を含む）。センサなし問題を解くためには、物理的状態ではなく信念状態の空間で探索を行う。初期状態はある信念状態であり、行為は信念状態を別の信念状態に写像する。信念状態に含まれる物理状態の一ひとつに行為を適用した結果を一つに合わせることで、行為を信念状態に適用した結果が得られる。経路はいくつかの信念状態をつなぐものであり、解はゴール状態だけを要素にもつ信念状態への経路である。図 3.21 は決定的でセンサなしの掃除機の世界の到達可能な信念状態空間を示す。ここには 12 個の到達可能な信念状態のみがあるが、信念状態の全体は物理状態のすべての可能な集合、すなわち  $2^S$  個の信念状態をもつ。信金が感

れ今までのセンサなし問題の議論では決定的な行為を仮定してきた。しかし、環境が非決定的、すなわち行為がいくつかの結果をもたらす可能性がある場合であっても、解析は本質的には変わらない。これは、センサがない場合は、実際にどんな結果が起きたのか知る方法がなく、そのためたらざる可の結果は、ちょうど後者である信念状態に追加的に加える物理状態と考えるしかない。たとえば、環境が次のようないマーフィーの法則にしたがうとしよう。*Suck* の行為は、すでにカーペットに埃がないときにはカーペットに埃を残すことがある。<sup>6</sup> すなわち、物理状態 4 (図 3.20 参照) で *Suck* が適用され

<sup>6</sup> 多くの読者はこうした問題に出会ったことがあると仮定している。我々のエージェントの状況に共感していただけだろう。最新で効率的な家電を所有していて、この教材が利用できない読者はお詫びしたい。

ると、結果は二つの可能性、状態 2 と 4 がある。初期の信念状態  $\{1, 2, 3, 4, 5, 6, 7, 8\}$  に適用する場合、*Suck* はこれら八つの物理状態からの可能な結果の合併集合からなる信念状態を引き起こすことになる。これを計算すると、新しい信念状態は  $\{1, 2, 3, 4, 5, 6, 7, 8\}$  である。そのため、このマーフィーの法則をもつ世界でのセンサなしエージェントにとっては、*Suck* は信念状態を何も変えてくれない。実際、この問題は解くことができない（練習問題 3.18 参照）。直観的にはエージェントは現在のマスに埃があるかどうかをわからず、そのため、*Suck* の行為が埃を片づけるか、それとも作り出してしまうかを知ることができないからである。

### 偶然性問題

#### 偶然性問題

行為のあとエージェントがセンサから新しい情報を得られる環境である場合、エージェントは偶然性問題 (contingency problem) に直面する。偶然性問題への解はしばしば木を構成する、木の枝は、その時点までに得た知識に応じて選択されるものである。たとえば、エージェントがマーフィーの法則の世界において、位置センサと局所的なセンサを備えているが、そのセンサは隣のマスの埃は感知しないとしよう。たとえば知識  $[L, Dirty]$  はエージェントが  $\{1, 3\}$  のどちらかの状態にいることを意味する。このエージェントは行為  $[Suck, Right, Suck]$  を生成するかもしれない。すると、*Suck* は  $\{5, 7\}$  のいずれかの状態に変化させ、そして *Right* によって  $\{6, 8\}$  のどちらかに移る。状態 6 にいる場合、最後の行為 *Suck* によってゴール状態 8 になるが、状態 8 にいる場合は（マーフィーの法則により）状態 6 に戻ってしまうかもしれない。この場合は失敗である。

$[Suck, Right, if [R, Dirty] then Suck]$

これによって実行中の偶然性にしたがって行為を選択する可能性を含むように解空間を拡張する。実際の物理的問題では正確な予測は不可能であり、その多くは偶然性問題である。

これと同じ理由で多くの人々は歩いたり運転したりするとき、目を大きく開いている。

偶然性問題は純粹に直列の解を許す場合がある。たとえば、完全に観測可能なマーフィーの法則の世界を考えよう。エージェントがきれいなマスで行為 *Suck* をを行うと偶然性が現れる。すなわち、そのマスに埃が残るかもしれないし、残らないかもしれない。エージェントがこれをやめなければならない限りは偶然性は現れず、すべての初期状態からの直列な解が存在する（練習問題 3.18）。

偶然性問題に対するアルゴリズムはこの章で述べた標準的な探索アルゴリズムよりも複雑になる。それらは 12 章で述べる。偶然性問題には少し違ったエージェント設計が適していいる。つまり、確固とした計画を立てる前に行動するエージェントである。これは有用である。行動の間に起こるかもしれないすべての偶然性を前もつて考慮するよりも、むしろ試しに行動してみてどんな偶然性が起きたかを見てみるほうが、しばしば良いからである。

エージェントは新たに得た情報を考慮して問題に取り組みつづけることができる。このように探索と実行を行うこと (interleaving) は探査問題 (4.5 節参照) やゲーム (6 章参照) にも有用である。

- 問題は四つの部分からなる：初期状態、行為の集合、ゴール検査関数、経路コスト関数である。問題の環境は状態空間として表現される。状態空間の中の初期状態からゴール状態までの経路がある。

- 一般的なアルゴリズムである TREE-SEARCH は、任意の問題を解くために用いることができる。このアルゴリズムを個々に変形することで異なる戦略を具体化することができる。

- 探索アルゴリズムは、完全性、最適性、時間複雑さを基準にして評価される。複雑さは状態空間の分枝度  $b$  と最も深い解の深さ  $d$  に依存する。

- 幅優先探索は探索木の中の未展開の最も浅いノードを選んで展開する。これは完全で、単位ステップコストに対しては最適であり、時間および空間複雑さは  $O(b^d)$  である。空間複雑さは多くの場合で非現実的となる。均一コスト探索は幅優先探索と類似しているが、最小の経路コスト  $g(n)$  をもつノードを選んで展開する。完全であり、各ステップのコストがある正の値  $\epsilon$  を超えるならば最適である。

- 深さ優先探索は探索木の中の最も深い未展開ノードを選んで展開する。完全性、最適性はいずれも満たさない。時間複雑さは  $O(b^m)$ 、空間複雑さは  $O(mn)$  である。ここで  $m$  は状態空間中の任意の経路の中での最大の深さである。

- 深さ制限探索は深さ優先探索に、ある決まった深さの限界を与えるものである。
- 反復深化探索はゴールが見つかるまで深さの限界値を増加させながら何度も深さ制限探索を行う。それは完全であり、単位ステップコストに対しては最適である。時間複雑さは  $O(b^d)$ 、空間複雑さは  $O(bd)$  である。
- 双方指向探索は時間複雑さを劇的に減少させるが、いつも適用できない、また、多くの場合で必要な空間が大きすぎる。

- 状態空間が木ではなくグラフの場合、探索木の中での状態の繰返しをチェックするこ

- とに利がある。GRAPH-SEARCH アルゴリズムは重複した状態をすべて取り除く。

- 環境が部分的にのみ観測できる場合、エージェントは信念状態、すなわち、エージェ

- ントがいる可能性のある状態の集合、の空間に探索アルゴリズムを適用することができます。

- 单一の列として解が作られる場合がある一方、起きうる未知の状況に備える偶然性を考慮した計画を必要とすることがある。

## 文献と歴史ノート

本章で分析した状態空間探索の大部分は、文献に長い歴史があり、見かけほど簡単ではない。練習問題 3.9 で取り上げた宣教師と人食い人種の問題は Amarel (1968) が詳細に分析した。この問題はこれ以前に, Simon and Newell (1961) が人工知能において Bellman and Dreyfus (1962) がオペレーションズリサーチにおいて検討した。これらの人他, Newell and Simon の Logic Theorist(1957) や GPS (1961) のような研究は 1960 年代の人工知能研究での主要な武器として探索アルゴリズムの確立に、また、標準的な人工知能問題として問題解決の確立に導いた。残念ながら問題の形式化のステップの自動化についてはほとんど研究されなかった。問題の表現や抽象化については、こうしたこと（部分的に）自ら行う人工知能プログラムを含めて、最近の研究が Knoblock (1990) にある。

15 パズルは有名なアメリカ人ゲームデザイナ Sam Loyd (1959) によって 1870 年代に発明されたもので、8 パズルはこれの小型版である。15 パズルは合衆国で一瞬にして大変な人気を得た。これは、最近のルービックキューブの流行に匹敵する。また、数学者の興味もすばやく引き付けている (Johnson and Story, 1879; Tait, 1880)。その論文の *American Journal of Mathematics* のエディタは次のように書いている。“この数週間, 15 パズルはアメリカ大衆に急速に広がっており、男女の別、年齢、階級を問わず、10 人中 9 人の関心を引き付けているといつても過言ではない。しかしこのような事態が、エディタがこの話題に関する論文を *American Journal of Mathematics* に掲載する判断を左右したわけではない、掲載理由は…”（以下、15 パズルの数学的な関心の理由の要約が続く）。

8 ハズルの金解の分析はコンピュータの助けを借りて Schofield (1967) が行った。Ratner and Warmuth (1986) は 15 パズルを一般化した  $n \times n$  版が NP 完全問題のクラスに属することを示した。

8 クイーン問題は最初、ドイツのチェス雑誌 *Schach* に 1848 年匿名で発表された。その後これは Max Bazzel の作とされた。1850 年に再出版された際、著名な数学者 Carl Friedrich Gauss の注意を引いた。Gauss はすべての解を挙げようと試みたが、72 個の解しか見つけられなかった。その後 1850 年、Nauck が 92 個すべての解を出版した。Netto (1901) は問題を  $n$  クイーンに一般化し、Abramson and Yung (1989) が  $O(n)$  のアルゴリズムを見つけている。

本章にあげた現実世界探索問題はどうしても、多くの研究で課題となってきた。航空路線での便の最適選択の方法は大部分が専売的であるが、Carl de Marchen (私信) は航空券と諦めは互いに絡まっており、最適な航空便の選択は形式的には決定不能であることを示した。巡回セールスマン問題は理論計算機科学の標準的な組合せ最適化問題である (Lawler, 1985; Lawler, Lenstra, Kan, and Shmoys, 1992)。巡回セールスマン問題が NP 困難であることが Karp (1972) によって証明されたが、効果的な発見的近似手法が (Lin and Kernighan, 1973) によって開発された。Arora (1998) はユーリッド空間での巡回セールスマン問題に対する完全な多項式時間近似の方法を考察した。VLSI のレイアウトの方法は Shahookar and Mazumder (1991) で述べられている。また、多くのレイアウト方法が VLSI の学術雑誌に発表されている。ロボット操縦と組立ての問題は 25 章で議論する。

問題解決のための情報のない探索アルゴリズムは古典的な計算機科学 (Horowitz and

Sahni, 1978) やオペレーションズリサーチ (Dreyfus, 1969) の中心的話題である。Deo and Pang (1984) や Gallo and Pallottino (1988) はより最近のサーベイである。幅優先探索は Moore (1959) が迷路を解くために定式化された。すべての部分問題の解をその長さを増しながら体系的に記録する動的計画法 (dynamic programming) (Bellman and Dreyfus, 1962) はグラフにおける横型探索の一つの形式と見ることができる。Dijkstra (1959) の二点間の最短経路のアルゴリズムは均一コスト探索の原型である。

反復深化のあるバージョンはチェスで時間を有効に使ったために Slatte and Atkin (1977) によって CHESS 4.5 のゲームプログラムで初めて導入された。しかし、グラフの最短経路探索への応用は Korf (1985) に負う。双方向探索は Pohl (1969, 1971) に導入されており、これもいくつかの事例で大変効果的である。

部分観測の環境や非決定的な環境は問題解決の方法論としてはそれほど深くは研究されていない。Genesereth and Nourbakhsh (1993) は信念状態探索についての効率に関するいくつかの点を研究している。Koenig and Simmons (1998) は未知の初期地点からロボット操縦について研究した。Erdmann and Mason (1988) はセンサなしでロボットを探査する問題を連続空間で定式化された信念状態探索を使って研究した。偶然性探索は計画問題の分野で研究されてきている (12 章参照)。不確かな情報のもとでの計画と行動については、大部分が確率論と意思決定理論の道具を用いて扱われている (17 章参照)。Nilsson (1971, 1980) の教科書は古典的な探索アルゴリズムについての一般的な情報源である。包括的で新しいサーベイは Korf (1988) にある。新しい探索アルゴリズム——驚くことに発見はまだ続いている——は, *Artificial Intelligence* のような雑誌に発表されている。

## 練習問題

3.1 次の用語を自分の言葉で定義せよ。状態、状態空間、探索木、探索ノード、ゴール、行為、後者閑数、分枝度。

3.2 ゴールを定式化したあとに問題を定式化しなければならない理由を説明せよ。

3.3 LEGAL-ACTIONS( $s$ ) が状態  $s$  での正しい行為の集合を、RESULT( $a, s$ ) が状態  $s$  で ACTION  $a$  を行った結果の状態を、それぞれ表すものとする。SUCCESSOR-FN を用いて LEGAL-ACTIONS と RESULT を用いて定義せよ。また逆に、SUCCESSOR-FN を用いて LEGAL-ACTIONS と RESULT を定義せよ。

3.4 8 パズルの状態が次のような排列的な二つの状態集合に分割されることを示せ。すなわち、一方の集合の状態はいくつの動作をさせても他の集合の状態に移ることのできないような集合である（ヒント：Berlekamp, Conway, and Guy (1982) 参照）。与えられた状態に対しそれがどちらのクラスに属すかを手書きを考案し、ランダムな状態を生成するのにこれを用いることができることを説明せよ。

3.5 67 ページの効率の良い漸增的な形式化を用いて  $n$  クイーンの問題を検討せよ。状態空間の大きさが少なくとも  $\sqrt[3]{n!}$  となることを説明し、網羅的に探査することができる最大の  $n$  を見積もれ（ヒント：各列について、クイーンが攻撃されるマスの最大数を考えるごとで、分枝度の下界を導く）。

3.6 有限の状態空間からは常に有限の探索木が導かれるかどうかを考えよ。木であるような有限の状態空間ではどうか、さらに厳密にどんなタイプの状態空間が常に有限の状態空間を導くか考えよ (Bender, 1996 より)。

3.7 次のそれぞれに対して、初期状態、ゴール検査、後者閾数、コスト閾数を与える、実装できる程度に明確な定式化を選択せよ。

- 平面地図を4色のみを使って塗り分けたい。ただし、隣接する領域は同じ色で塗つてはならない。
- 3フィートの背丈の猿が一匹、部屋にいる。部屋にはバナナが8フィートの天井からぶら下がっている。猿はこれが欲しい。部屋には積み上げること、移動させること、そして登ることのできる3フィートの高さの木枠が二つある。
- ある種の入力レコードを与えると“不正な入力レコードです”とメッセージを出力するプログラムがある。各レコードの処理は互いに独立であることがわかつているとする。どのレコードが不正であるかを知りたい。
- 12ガロン、8ガロン、3ガロンを観ることのできる三つの壺と、水の蛇口がある。壺を満たしたり、それを他の壺に移して空にしたり、地面に空けたりすることができる。ちょうど1ガロンを割りたい。

3.8 初期状態が数1であり、状態  $n$  に対して後者閾数は二つの状態、数  $2n$  と  $2n+1$  を返すような、状態空間を考える。

- 状態1から15までの部分の状態空間を描け。
- ゴール状態が11であるとする。幅優先探索、深さ制限3の深さ制限探索、反復深化探索のそれぞれが訪問する順序でノードを書き出せ。
- この問題には双方向探索は適切であるかを考えよ。適切である場合、どのように動作するかを詳細に述べよ。
- 双方向探索の各方向の分枝度を与えるよ。
- (c)への解答は状態1からゴール状態まで達する問題をほとんど探索することなく解くことができるような再定式化を示唆しないか考えよ。

3.9 宣教師と人食い人種 (missionaries and cannibals) の問題は通常次のとおりである。3人の宣教師と3人の人食い人種が川の同じ岸にいる。そこには、1人か2人が乗ることのできるボートが停留している。常に、1カ所にいる宣教師の数を人食い人種の数が上回らないよう保ちつつ、全員が反対側の岸に行くことのできる方法を見つけたい。この問題は人工知能では有名である。それは、分析的な観点から問題の定式化を行った最初の論文 (Amarel, 1968) の課題であったからである。

- 精密に、かつ妥当な解を確かめるために必要な区別のみをして問題を定式化せよ。また、完全な状態空間を描け。
- 適切な探索アルゴリズムを用いて、この問題を実装し最適に解け。状態の繰返しを検査することは有効かどうかを考えよ。
- このパズルの状態空間は大変単純であるが、解くためには大きな時間がかかることなぜか？

3.10 8パズルの2種類の後者閾数を実装せよ。一つは8パズルのデータをコピーし、それを修正することで一度にすべての後者を生成するもの、もう一つは呼び出されたときに一度に一つずつ生成し、親の状態を直接修正する（そして必要に応じて修正をやり直す）ことでこれを実現したもの。これらの閾数を用いた反復深化深さ優先探索のバージョンをそれぞれ書き、その動作を比較せよ。

- 3.11 79ページで反復延長探索 (iterative lengthening search)、すなわち均一コスト探索の反復版に触れた。経路コストの制限を順に増加させていくというアイデアである。現在の制限を超えた経路コストをもつノードが生成されたときは、ただちにこれを捨てる。次の繰返しでは、前の繰返しで捨てられたノードのうち、最小の経路コストを制限とする。
- このアルゴリズムは一般的の経路コストに対する最適であることを示せ。
  - 分枝度が  $b$ 、解の深さが  $d$  であり、単位ステップコストをもつ、一様な木を考える。繰返し延長の回数は何度必要であるか、考え方よ。
  - ステップコストが連続的な区間  $[0, 1]$  にあり、ただし最小は正のコスト  $\epsilon$  であるとする。このとき、最悪では何回の繰返しが必要であるか考え方よ。
  - このアルゴリズムを実装し、8パズルと巡回セールスマン問題の具体例に適用せよ。このアルゴリズムと均一コスト探索の振舞いを比較し、結果を考察せよ。

3.12 均一コスト探索と、定数ステップコストをもつ幅優先探索について、GRAPH-SEARCH アルゴリズムを用いた場合、最適であることを証明せよ。また、反復深化を用いる GRAPH-SEARCH が最適ではない解を見つけてしまおうな定数ステップコストをもつ状態空間を与えるよ。

- 3.13 反復深化探索の振舞いが深さ優先探索よりも極端に悪くなる状態空間をあげよ（たとえば、 $O(n)$ ）に対して  $O(n^2)$  となるように）。
- 3.14 二つのウェブページのURLを入力として取り、一方から他方へつながるリンクの経路を見つけるプログラムを書け。どんな探索戦略が適切であるか答えよ。及方向探索は良い考え方かどうかを考えよ。前者閾数を実装するために何かサーチエンジンを用いるか検討せよ。

3.15 図3.22に示されているような凸多角形の障害物をいくつかを含む平面上で、2点間の最短経路を見つける問題を考えたい。これは迷合った環境でロボットがその進む道を見つけなければならないような問題を理想化したものである。

- 平面上のすべての点  $(x, y)$  からなる状態空間を考える。いくつの状態が存在するか？ゴールへの経路がいくつ存在するか？
- 一つの多角形の頂点から、そこにある多角形の他の頂点への最短経路は、多角形の頂点のいくつかをつなぐ線分からなることを、簡単に説明せよ。これを考慮の上、良い状態空間を定義せよ。この状態空間の大きさを考えよ。
- この探索問題を実装するために必要な閾数を定義せよ。この閾数には、頂点を入力するとその頂点から直線で達することのできる頂点の集合が返る後者閾数が含まれる（同じ多角形の頂点の隣接を忘れないように）。直線距離をヒューリスティック閾数に用いなさい。

- d. 本章で述べた一つ以上のアルゴリズムを適用し、この問題を一定の範囲で解いてみよ。また、結果について考察せよ。

- e. 見え方がまったく同じ場所がある場合を考える（たとえば、正方形の障害物をもつ、格子状の世界を考えてみよ）。この場合、エージェントにはどんな課題があるか考えよ、その解決法はどのようになるか、検討せよ。

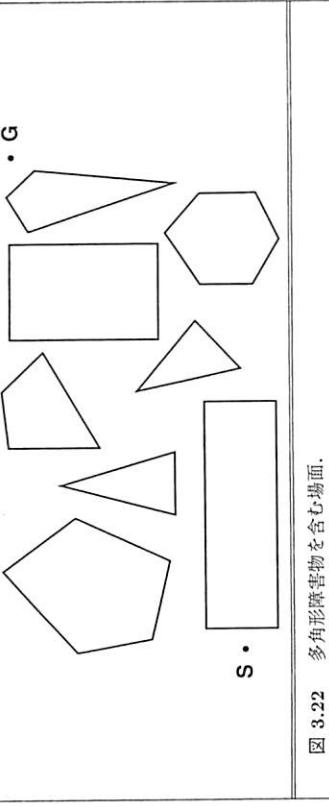


図 3.22 多角形障害物を含む場面。



3.16 練習問題 3.15 の経路探索の問題の環境を次のように考える。

- 知覚を、エージェントから見える頂点が、エージェントに相対的位置のリストとして与えられるものとする。この知覚には当のロボットの位置は含まれない。ロボットは自身の位置を地図から学ばなければならない。さしあたり、それぞれの位置ではそれ程異なった見え方となると仮定することとする。

- 各行為を、それに沿って移動する直線を表すべきトルトとする。その道が障害物に当たらなければ、行為は成功である。そうでない場合は、ロボットはその道と最初の障害物との交点で停止する。エージェントがゼロ移動ベクトルを出しし、かつ、ゴールにいるならば（ゴールは固定されており、既知）、環境によってエージェントは瞬時に（障害物の内部は除く）ランダムな位置に移動させられる。
- 評価尺度として移動した単位距離ごとに 1 点をエージェントに課し、ゴールに達すると 1000 点の褒賞を与える。

- この環境と問題解決エージェントを実装せよ。エージェントは瞬時移動が起ころうに、現在位置の発見を含めて、新しい問題を定式化する必要がある。
- (エージェントがどう動いたか、適切な記録をとることで) エージェントのパフォーマンスを記述せよ、100 エピソードのパフォーマンスについてまとめてよ。
- 30%の確率でエージェントは意図しない方向に移動してしまうよう環境を修正しなさい（見える範囲に他の頂点があればそこからランダムに選んで、なければ移動しない）。これは実ロボットの動作エラーの荒っぽいモデルである。エラーを検知したときは今どこにいるのかを見つけ、元の計画に復帰できるように元いた所に戻るような計画を作るよう、エージェントを修正せよ。元いた所に戻るという行為 b<sub>b</sub>、また失敗することがありますことに注意せよ。エージェントが 2 回の連続したエラーを克服してゴールに到達する例を示せよ。
- さらには 2 種類のエラー回復の方法を試みたい：(1) 元々の経路上の最も近い頂点に向かう、(2) 新しい位置からゴールへの計画を立て直す。3 種類のエラー回復の方法の振舞いを比較せよ。探索コストを考えた場合は、どうであるかも比較せよ。

- 3.17 63 ページで、負の経路コストをもつ問題については考慮しないと述べた。この問題でこれについて深く検討する。

- 行為は任意に大きな負のコストをとることができるとする。この場合、任意の最適なアルゴリズムは状態空間の全体を探索しなければならない。この理由を説明せよ。
- ステップコストが、ある負の定数 c 以上であるとわかっていることで、何らかの改善があるか考えよ。木の場合と、グラフの場合について考えよ。
- ループを作るオペレータの集合がある場合を考える。つまり、この集合に含まれるオペレータをある順序で実行した場合、状態には正味何の変化もない。これらのオペレータがすべて負のコストをもつ場合、この環境でのエージェントの振舞いの最適性について、何が帰結されるか考えよ。
- 経路発見のような領域であっても、負の大きなコストをもつオペレータについて想像することは容易である。たとえば、いくらかの寄り道が時間や燃料の意味での通常のコストを上回って余りあるだけの美しい風景を見せてくれるかもしれない。人が風景を見るドライブをいつまでも無限定にすることがない理由を、状態空間探索の文脈で、厳密な用語を用いて説明せよ。また、人工的なエージェントもループをしないように経路発見をするためには状態空間とオペレータをどのように定義したらしいか、説明せよ。
- ステップコストがループを引き起こしてしまうような、現実の領域について考えよ。

- 3.18 センサなしで、マーフィーの法則にしたがう二つの位置をもつ掃除機の世界を考える。初期信念状態 {1, 2, 3, 4, 5, 6, 7, 8} から達することのできる信念状態空間を描き、この問題は解くことができない理由を説明せよ。また、世界が完全に観測できる場合はどんな初期状態に対しても解の列があることを示せ。



- 3.19 図 3.2.2 に定義された掃除機の世界について考える。
- 本章で定義されたアルゴリズムのうち、どれがこの問題に適しているかを考えよ。そのアルゴリズムで状態の繰返しを検査すべきかどうかを考えよ。
  - 3 × 3 の世界で、上の三つのマスに埃があり、エージェントが中央のマスにいる状態を初期状態にした問題に対し、上で選んだアルゴリズムを適用し、最適な行動を計算せよ。
  - 掃除機の世界の探索エージェントを構築し、各マスで 0.2 の確率で埃が現れる 3 × 3 の世界について、その振舞いを評価せよ。評価尺度には、経路コストだけでなく探索コストも合理的な交換レートで含めること。
  - 最も優秀なエージェントを、埃があつたら吸い込み、なかつたらランダムに移動する単純なランダム反射エージェントと比較せよ。
  - 世界が  $n \times n$  に拡張された場合、どうなるかを考えよ。選択したエージェントの振舞いと、反射エージェントの振舞いが  $n$  に応じてどう変化するか考えよ。