

4 知識に基づく探索と検査

ここでは状態空間に関する知識を用いると、暗闇の中を手探りで探すようなアルゴリズムをどのように避けられるかを説明する。

3章では、問題に関する知識がないときに、機械的に新しい状態を生成し、それがゴールを満足するかどうかをテストするという方法によって解を見つけることを説明した。残念なことに、この戦略はほとんどの場合非常に効率が悪い、本章では知識を用いた探索——問題に特有の知識を用いる方法——によって、解をより効率的に見つける方法を説明する。4.1節では3章のアルゴリズムを、知識が与えられた場合へ変更した版を説明し、4.2節では、4.3節と4.4節では、初期状態から機械的に経路を探査する代わりに、複数の現状態を評価し変更することによって状態空間の局所探索(local-search)を行うアルゴリズムを説明する。これらのアルゴリズムは経路のコストは重要でなく、解空間だけが重要であるような問題に適している。局所探索のアルゴリズムのファミリーには統計物理(焼きまし法(simulated annealing))や進化生物学(遺伝的アルゴリズム(genetic algorithm: GA))に触れた方式も含まれている。最後に、4.5節ではオンライン探索(online search)を説明する。これは、エージェントがまったく未知の状態空間に直面したときに行われるものである。

4.1 知識に基づく（ヒューリスティック）探索手法

知識に基づく
探索

評価関数

この節では知識に基づく探索(informed search)——問題の定義だけでなく問題特有の知識を利用する方式——が、知識のない探索に比べてどのように効率的に解を見つけるのかを説明する。これから説明する一般的なアプローチは最良優先探索(best-first search)とよばれてもいる。最良優先探索は、展開していくノードを評価関数(evaluation function) $f(n)$ を用いて選択していくという、一般的なTREE-SEARCHやGRAPH-SEARCHアルゴリズムの具體例である。評価関数はゴールまでの距離を測定するから、伝統的な方法では評価値が“最も低い”ノードが展開先に選ばれる。最良優先探索は優先待ち行列を使って、我々の一般的な探索の枠組みの中で実現される。優先待ち行列は f 値を小さい順に並べた縦^aの集合。

^a 訳注：まだ展開していないノードの集合。

保持するデータ構造である。

“最良優先探索”という名前は尊重すべきである。もし仮に本当に最良のノードがいつも最初に展開できるのであれば、そもそも探索などする必要がない。ゴールに向けてまっすぐ進めばよいだけである。我々にできるのは、評価関数にしたがって最良に“見える”ノードを選ぶだけである。もし評価関数が本当に正確であれば、選ばれるノードは最良である。実際は評価関数の値は真のものから少し離れているので、探索が道に迷ってしまうこともある。それでもここでは“最良優先探索”という名前を使うことにする。“最良に見えるものを優先する探索”という名前は格好悪いからである。BEST-FIRST-SEARCHアルゴリズムには、異なった評価関数を用いるファミリーが存在する！これらはアルゴリズムのキーとなる要素は $h(n)$ で表されるヒューリスティック関数 (heuristic function)²である：

$$h(n) = \text{ノード } n \text{ からゴールノードへ至る最短経路の見積りコスト}$$

たとえば、ルーマニアの場合、AradからBucharestまでの最短経路のコストは、AradからBucharestまでの直線距離だと見積もることができる。

ヒューリスティック関数は、探索アルゴリズムに追加の知識を与える最も普通の方法である。ヒューリスティック関数については4.2節でもとつと深く学習することにする。今のところはヒューリスティック関数を、もし n がゴールノードだとすれば $h(n)=0$ となる、という一つの制約をもつた任意の問題特有の関数だと考えることにする。この節の以下の部分では、探索を導くためにヒューリスティック情報を用いる二つの方法を説明する。

欲張り最良優先探索

欲張り最良優先探索(greedy best-first search)³は、ゴールに一番近いノードが解に早く着きそうであるという根拠で、そのようなノードから展開を試みる。つまり、 $f(n) = h(n)$ というヒューリスティック関数を使ってノードの展開を行うということである。

h_{SLD} および直線距離(straight-line distance)のヒューリスティックを用いて、ルーマニアの道探索問題がどのように進むかを見てよう。もしゴールがBucharestだとすると、Bucharestまでの直線距離を知らなくてはならないが、その情報は図4.1に書かれている。

たとえば、 $h_{SLD}(In(Arad))=366$ である。 h_{SLD} の値は問題の記述自身からは計算できないことを注意しなければならない。さらに、 h_{SLD} は実際の道のりと相関関係があり、役に立つヒューリスティックであることを知るには、ある程度の経験が必要である。

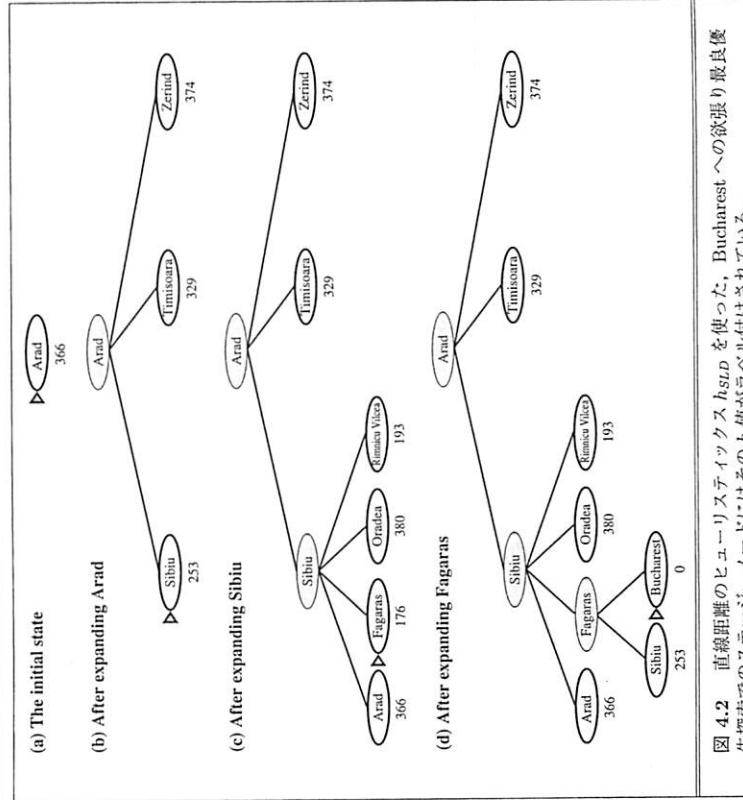
図4.2は h_{SLD} を用いてAradからBucharestへの経路を欲張り最良優先探索している様子を示している。SibiuのほうがZerindやTimisoaraよりBucharestに近いから、Aradから最初に展開されるノードはSibiuになる。次に展開されるノードは、それが最もBucharestに近いFagarasである。次にFagarasはゴールであるBucharestを生成する。この例では、 h_{SLD} を使った欲張り最良優先探索は、解となる経路の上にないノードを展開することなく解を見つけ出している。したがって、この探索コストが極小である。しかし、これは完全に誤りである。

¹ 練習問題4.3では、このファミリーがいくつかの知識なしアルゴリズムを含んでいることを示せと読者に要求する。

² ヒューリスティック関数 $h(n)$ は入力としてノードをとるが、その関数はそのノードの状態にだけ依存する。

³ 本書の第1版ではこれを欲張り探索(greedy search)とよんだ。また、他の本ではこれを最良優先探索(best-first search)とよんでいる。後者の用語についての我々の用語についての説明はPearl(1984)にしたがったものである。

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

図 4.1 h_{SLD} の値——Bucharest への直線距離。図 4.2 直線距離のヒューリスティックス h_{SLD} を使った、Bucharest への欲張り最良優先探索でのステージ。ノードにはその h 値がラベル付けてある。

まず Vasliu に行って——ヒューリスティックスによるとゴールによるとゴールから実際に遠いのであるが、——次に Urziceni, Bucharest に行って Fagaras に行く経路である。このようなケースでは、ヒューリスティックスによって不要なノードが展開されている。さらに、我々が不注意で同じ状態が繰り返されるのを発見しなければ、探索は Neamt と Isasi の間で往復して、正解は見つけられないだろう。

欲張り最良探索はゴールへの全行程の経路を一つ選んで行き止まりにならんといふ点で、深さ優先探索と似ているので、深さ優先探索と同じ欠点をもつ、最適解を見つける保証がないこと、(無限ループに陥って他の道に戻れない可能性があるため) 不完全であることである。欲張り探索の最悪の場合の時間計算量は、 m を探索空間の最大深さとすると $O(b^m)$ となる。しかし良いヒューリスティック関数があれば、空間・時間計算量は実質的に減らすことができる。減らすことのできる量は、その問題とヒューリスティックスの質によって決まる。

A*探索：経路全体の解コストの見積りの最小化

最も優先探索の中で最もよく知られているものは A*探索 (A* search) (“A スター探索”と発音する) とよばれている。この探索では、そのノードに達するまでのコスト $g(n)$ と、そのノードからゴールにいたるまでのコスト $h(n)$ を結合してノードを評価する：

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

$f(n)$ は出发ノードからノード n までの経路のコストで、 $h(n)$ は n からゴールまでの最短距離の見積りコストであるから、

$$f(n) = n \text{ 経由の最短解の見積りコスト}$$

ということになる。したがって、もし最短解を見つけようとするなら、最初に試みるべきもっともらしい候補は $g(n) + f(n)$ が最小となるノードである。この方法は単に合理的なだけないことを明らかにする、それは h 関数がある条件を満たせば、A*探索は完全で最適であることである。

A*は、もし THREE-SEARCH と一緒に用いられるならば、その最適性の解析は簡単である。この場合には、A*はもし $h(n)$ が許容的ヒューリスティック (admissible heuristic)、すなわち、 $h(n)$ がゴールへ至るまでのコストを “過大に見積ることがない” ならば、A*は最適である。許容的ヒューリスティックスは問題の解決コストを実際に小さく見積もるので、本質的に楽観的なヒューリスティックスである。 $g(n)$ は n に達するまでの正確なコストであるから、 $f(n)$ は n 経由の正解のコストを絶対に過大に見積もらないという帰結を得る。

許容的ヒューリスティックスのわかりやすい例は Bucharest へ行くのに用いた直線距離 h_{SLD} である。任意の 2 点を結ぶ最短経路は直線であり、直線は過大な見積りができるないから、直線距離は許容的である。図 4.3 に Bucharest への A*探索の展開の様子を示す。 g の値は図 3.2 のステップコストから計算され、 h_{SLD} の値は図 4.1 で与えられる。Bucharest はステップ (e) の縁に最初に出現するが、その f 値 (450) は Pitesti (417) よりも大きいため、展開対象には選ばれていないことに特に注意する必要がある。言い方を変えると、Pitesti 経由で 417 と同じコストの解があるかもしないため、このアルゴリズムは 450 のコストの解を決めなかつたのである。この例から、もし $h(n)$ が許容的なら TREE-SEARCH を



使ったA*は評容的であることの一般的な証明を導くことができる。ある準最適なノード G_2 が線に現れたらとし、最適解のコストを C^* であるとしよう。すると、 G_2 は準最適であり、 $h(G_2) = 0$ （すべてのゴールノードで成り立つ）であるから、次の式が成り立つ。

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$$

次に最適経路上の他のノード n を考える。たとえば、前の段落の例でのPitestiである（もし正解が存在するなら、そのようなノードが必ず存在する）。もし $h(n)$ が正解経路を完成させるコストを過大に見積もっていないなら、次の式が成り立つ。

$$f(n) = g(n) + h(n) \leq C^*$$

このように、 $f(n) \leq C^* < f(G_2)$ であることを示したので、 G_2 は展開されることはなく

A^* は最適解に復帰する。

もし我々がTREE-SEARCHアルゴリズムの代わりに図3.19のGRAPH-SEARCHアルゴリズムを使おなら、この証明はうまくいかない。GRAPH-SEARCHアルゴリズムは、それが最初に生成されたものでないときは、繰り返される状態へ至る最適解を捨ててしまう恐れがあるので、準最適解を答えとしてしまうかもしれない（練習問題4.4参照）。この問題を解決するには二つの方法がある。第一の解決方法は、GRAPH-SEARCHを拡張して、同一ノードで見つかかった二つの経路のうちでコストが高いほうを捨てる（3.5節の議論を参照）。余分な管理をするのは煩わしいが最適性は保証される。第二の解決方法は、均一コスト探索の場合と同様に、任意の繰り返し状態に至る最適経路を常に最初に調べるようになることである。この性質は、もし $h(n)$ に特別な条件、すなわち無矛盾性（consistency）（単調性（monotonicity）とも言う）条件をつけるなら成立する。すべてのノード n とアクション a で作られた n のすべての後続ノード n' について、 n からゴールに至る見積りコストが、 n' へ至るステップコストと n' からゴールへ至る見積りコストの和より大きくならないなら、ヒューリスティクス $h(n)$ は無矛盾である：

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

この式は、三角形のどの辺も他の二辺の和より大きくなることではないという一般的な三角不等式（triangle inequality）の形をしている。ここでは、三角形は n と、 n' と、 n に最も近いゴールによって作られている。すべての無矛盾であるヒューリックスは許容的でもあることを示す（練習問題4.7）のはかなりやさしい、無矛盾性に関する最も重要な結果は次のことである。GRAPH-SEARCHを使ったA*は、もし $h(n)$ が無矛盾であれば最適である。

無矛盾性は許容性よりも厳しい条件ではあるが、許容的であって無矛盾でないようなヒューリックスを作るのは難しい。この章で議論するすべての許容的なヒューリックスは無矛盾でもある。たとえば h_{SLD} を考えてみよう。我々は、三角形の各辺が直線距離で測定されるなら一般的な三角不等式が成り立つことや、 n と n' の直線距離が $c(n, a, n')$ よりも大きくないことを知っている。そのため、 h_{SLD} は無矛盾性のあるヒューリックスなのである。

無矛盾性に関する他の重要な結論は以下のものである。もし $h(n)$ が無矛盾であれば、どの経路に沿った $f(n)$ の値も減少することはない。この証明は無矛盾性の定義から直接求めることができる。 n' が n の次のノードだとすると、ある a に対して $g(n) = g(n') + c(n, a, n')$ となり、以下の式が得られる。

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

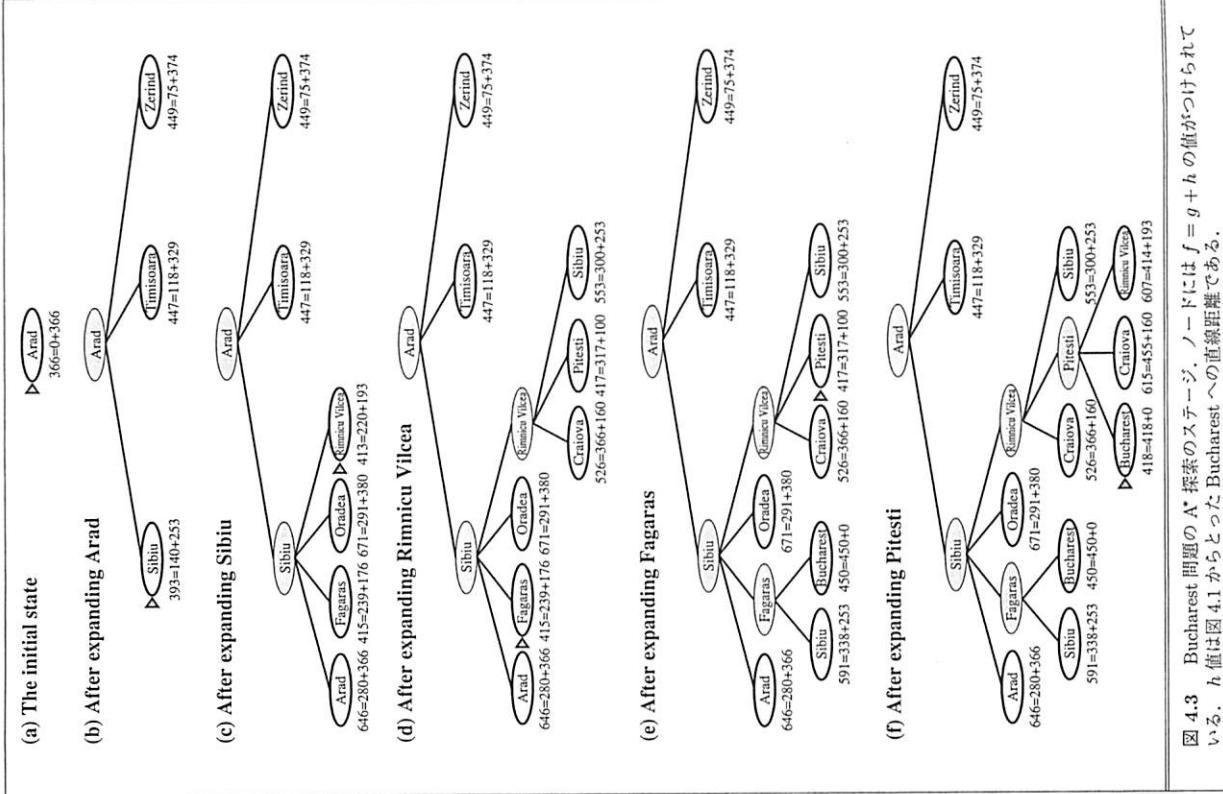


図 4.3 Bucharest 問題の A* 探索のステージ。ノードには $f = g + h$ の値がつけられており、 h 値は図 4.1 からとった Bucharest への直線距離である。

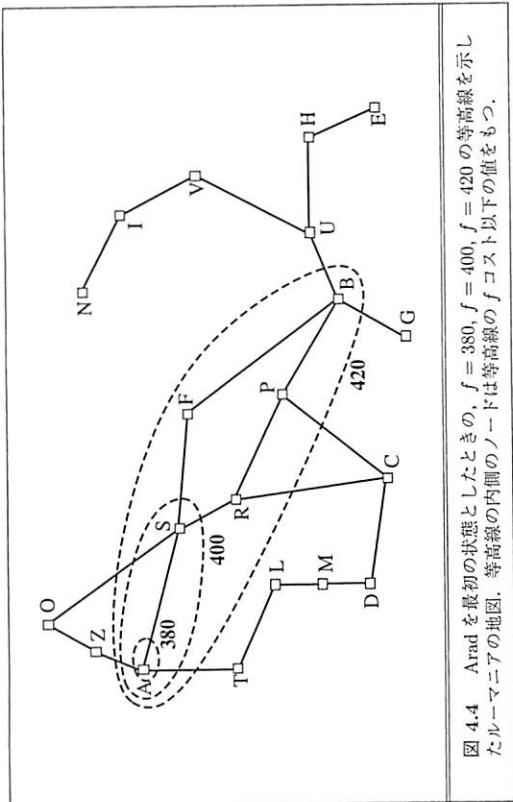


図 4.4 Arad を最初の状態としたときの、 $f = 380$, $f = 400$, $f = 420$ の等高線を示したルーマニアの地図。等高線の内側のノードは等高線の f コスト以下との値をもつ。

このことから、GRAPH-SEARCH を使った A^* で展開されたノードの系列は $f(n)$ が減少しない順序となっている。そのため、展開するためには最初のゴールノードは最適な解である。あとで展開されるすべてのノードのほうがコストがそれ以上だからである。任意の経路に沿って f コストが減少しないということは、地図上の等高線 (contour) と同じように状態空間の等高線を描くことができるということである。図 4.4 はその例である。400 ドラベル付けした等高線の内側では、すべてのノードの $f(n)$ が 400 以下であることが成り立つ。したがって、 A^* は f コストが最小の緑のノードを追加するから、 A^* 探索は開始ノードから、 f コストの増加していく同心の帯内にノードを追加しながら広がっていくことを観察することができる。

一方で、 A^* は $f(n) = 0$ を使った A^* 探索においては、この帯は開始状態のまわり均一コスト探査 ($h(n) = 0$) によるいくつかのノードを展開するかもしれない。

直観的には、見つかった最初の解のが最適解であるのは明らかである。というのは、そのあとのすべての等高線内のゴールノードは f コストが高く、その結果 (すべてのゴールノードは $h(n) = 0$ となるから) g コストが増加する帶を追加するので、 f がゴール状態への経路のコストに等しいような帯に最終的には達する。⁴

効率優遇
枝刈りされた

A^* は $f(n) > C^*$ となるノードを展開しないことに注意しなくてはならない。たとえば、図 4.3 で Timisoara は、それが根ノードの子供としても展開されない。我々は Timisoara 以下の部分木は枝刈りされた (pruned) という言い方をする。 h_{SLD} は許容的だから、このアルゴリズムは最適性を保証しながらこの部分木を安全に無視することができます。確認することなしに可能性を考慮の外におくという枝刈りの概念は、多くの AI 分野で重要なものである。

最後に観察できることは、根ノードから探索経路を拡大していくこの種の最適アルゴリズムの中で、 A^* は与えられたどんなヒューリスティック関数についても効率最適 (optimally efficient) であるという点である。すなわち他の最適アルゴリズムは $(f(n) = C^*)$ であるノードの間で引き分けになる可能性を除いて) A^* より少ないノードを展開することが保証されている。これは、 $f(n) < C^*$ となるすべてのノードを“展開しない”ようなアルゴリズムは、どれも最適解を見つけられない危険を有しているからである。

すべてのそのようなアルゴリズムの中で、 A^* 探索が完全で、最適で、効率最適なのはかなり満足なものである。残念ながら、このことは A^* が我々の探索のすべてのニーズに対する解であることを意味しているわけではない。問題点は、ほとんどの問題において、ゴール等高線の探索空間の中のノードの数は、依然として解の長さに対する指数関数的である。この結論についての証明は本書の範囲外であるが、ヒューリスティック関数のエラーが実際の経路コストの対数よりも早く増加するのではなければ、指数関数的な増加の条件は起りうることが示されている。数学的な記法を使うと、この漸近的な増加の条件は以下のようになる。

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

ここで $h^*(n)$ は n からゴールまでの真のコストである。実用的な用途におけるほとんどすべてのヒューリスティックでは、エラーは少なくとも経路コストに比例し、それにによる指數的な増加によって、ついにはどんなコンピュータでも追いつけなくなる。そのため最適解を見つけることは現実的ではないことが多い。満最適解を高速に見つける A^* の変形版を使ったり、より正確だが厳密には許容的ではないヒューリスティックを設計したりすることもできる。どんな場合にも、良いヒューリスティックを使うと、情報のない探索を用いた場合よりも大きな節約になる。4.2 節では良いヒューリスティックを設計するための課題を説明する。

ところで、計算時間が A^* の主要な欠点といわわけではない。 A^* は (全 GRAPH-SEARCH アルゴリズムと同様に) 生成されたすべてのノードをメモリに記録するので、通常は A^* は時間切れになる前にメモリ空間が不足することになる。そのため、 A^* は多くの大規模プログラムでは実用的ではない。最近開発されているアルゴリズムは実行時間にわざかなコストをかけることによって、最適性や完全性に対することなしにメモリ空間の問題を克服している。これらのアルゴリズムは次に説明する。

メモリ限定ヒューリスティック探索

A^* のメモリ要求を減らす最も単純な方法は反復深化の考え方をヒューリスティック探索に適用することであり、その結果が、反復深化 A^* (IDA*) アルゴリズムである。IDA* と通常の反復深化との大きな違いは、深さによるカットオフ (cutoff) を使うのではなく、 f コスト ($g + h$) によるカットオフを使う点にある。各繰返しにおいて、カットオフ値は、以

⁴ 完全性を示すには、 C^* 以下のコストのノードが有限しかないとすると、これは、すべてのステップコストがある有限値 ϵ より大きく、 b が有限ならば、成り立つ条件である。

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE([problem]),  $\infty$ )
    function RBFS(problem, node, [limit]) returns a solution, or failure and a new f-  

        cost limit
        if GOAL-TEST[problem]/STATE[node] then return node
        successors  $\leftarrow$  EXPAND(node, problem)
        if successors is empty then return failure,  $\infty$ 
        for each s in successors do
            f[s]  $\leftarrow$  max(g(s) + h(s), f[node])
        repeat
            best  $\leftarrow$  the lowest f-value node in successors
            if f[best] > f_limit then return failure, f[best]
            alternative  $\leftarrow$  the second-lowest f-value among successors
            result, f[best]  $\leftarrow$  RBFS(problem, best, min(f_limit, alternative))
            if result  $\neq$  failure then return result
    endfunction

```

図 4.5 繰返し最良優先探索のアルゴリズム。

前の繰返しでのカットオフを超したノードの *f* コストの最小値である。IDA*は均一のステップコストをもつ多くの問題で実用的であり、並べられたノードの待ち行列を保持することによる実質的なオーバヘッドを避けることができる。残念なことに、IDA*は、練習問題 3.11 で述べたような均一コスト探索の反復版と同じように、実数値のコストに伴う問題点をもっている。この節では RBFS と MA* というメモリ限定期間の、より最近の二つのアルゴリズムを簡単に説明する。

再帰的最良探索 (recursive best-first search: RBFS) は、通常の最良優先探索のオペレーションを模倣しながら、線形の空間だけを使うというアルゴリズムである。このアルゴリズムは図 4.5 に示されている。この構造は、再帰的深さ優先探索の構造と類似しているが、現在の経路を無限に探索するではなく、現在のノードのすべての祖先のノードから行ける別の経路の *f* 値の最適値を調べる。もし現在のノードを超えると、再帰処理を後戻りして他の経路に移る。再帰処理を後戻りするとき、RBFS は経路に沿った各ノードの *f* 値を、その子供の *f* 値の一一番良い値に置き換える。このようにして、RBFS は忘れた部分木の一番良い葉ノードの *f* 値を記憶するので、少しあとにあってもその部分木を再度展開する価値があるかどうかを決めることができる。図 4.6 は RBFS がどのようにして Bucharest に達するかを示している。

RBFS は IDA* よりもやや効率的であるが、それでもなお過度なノード再生成の問題をもつ。図 4.6 の例では、RBFS は最初に Rîmnicu Vilcea 経由の経路にしたがうが、“心変わりして” Fagaras を試み、さらにまた心変わりしてもともに戻る。このような心変わりは、現在の最良経路が拡張されたびに *f* 値が増加する良いチャンスがある——*h* は通常はゴールに近いノードではなくに樂観的ではない——ために生じるのである。特に探索空間が大きいたときは、心変わりが起こると、二番目に良い経路が一番の経路となり、それに合わせるために探索は後戻りしなくてはならない。各々の心変わりは IDA* の繰返しに相当し、心変わりをすると最良経路を再構築してもう一つノードを展開するために、忘れられたノードの再展開を多数回必要とするかもしれない。

A* と同様に、ヒューリスティック関数 *h*(*n*) が許容的ならば、RBFS は最適なアルゴリ

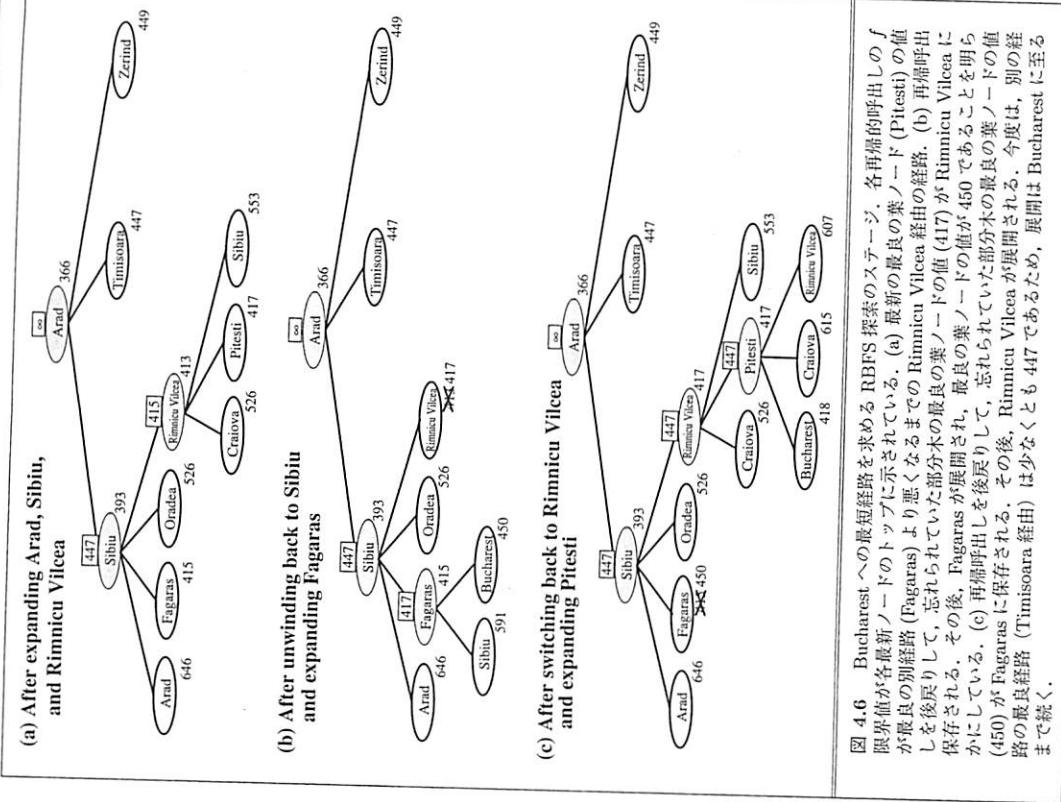


図 4.6 Bucharest への最短経路を求める RBFS 探索のステージ。各再帰的呼び出しの限界値が各最新ノードのトップに示されている。(a) 最新の最良の葉ノード (Pitești) の値が最良の別経路 (Fagaras) より悪くなるまでの Rîmnicu Vilcea 経由の経路。(b) 再帰呼び出しを後戻りして、忘れられた部分木の最良の葉ノードの値 (417) が Rîmnicu Vilcea に保存される。その後、Fagaras が展開され、最良の葉ノードの値が 450 であることを明らかにしている。(c) 再帰呼び出しを後戻りして、忘れられた部分木の最良の葉ノードの値 (450) が Fagaras に保存される。その後、Rîmnicu Vilcea が展開される。今度は、別の経路の最良経路 (Timisoara 経由) は少なくとも 447 であるため、展開は Bucharest に至るまで続く。

ズムである。その空間的な複雑さは $O(bd)$ であるが、その時間的な複雑さは特徴を述べるのが難しい。といふのは、時間的な複雑さは、ヒューリスティック関数の正確さと、ノードが展開されると最適経路の交換がどのくらい頻繁に起るか、の両方に依存するからである。IDA* も RBFS もグラフの探索の複雑さは指數関数的に増加しうる (3.5 節参照)。それは、これらの方は現在の経路上にない場合の状態の繰返しをチェックすることができるないからである。そのため、これらの方は同じ状態を何度も調べることがある。IDA* も RBFS もメモリを“少ししか使わなすぎ”という問題をもっている。繰返しの

間、IDA*は最新の δ コストの限界値といふ一つの数字だけしか保持していない。RBFSはメモリ上にもう少し多い情報をもつてゐるが、それでも $O(bd)$ のメモリしか使っていない。もう少し多くのメモリが使えたとしても、RBFSはそれを利用する方法をもっていない。このようなことから、使えるメモリをすべて使うことは賢いことだと思われる。これを行う二つのアルゴリズムは MA* (メモリ制限 A*) (memory-bounded A*) と SMA* (単純化 MA*) (simplified MA*) である。ここではより単純な——これは良いことである——SMA*を説明しよう。SMA*はメモリが一杯になるまで最適な葉ノードを展開しながら、A*と同じように進行する。メモリが一杯になった時点では、古いノードを捨てることに新しいノードを探索木に追加することはできない。SMA*は常に f 値が一番高い、“最悪な”葉ノードを捨てる。次にSMA*はRBFSと同じように、忘れられたノードの値を親ノードに戻す。このようにして、忘れた部分木の祖先はその部分木の最適経路の品質を知る。この情報を使って、SMA*は、他のすべての経路がすでに忘れてしまった経路よりも悪そうに見えるときだけ、その部分木を再生成する。言い換えると、ノード n のすべての後続ノードが忘れたならば、 n からどこへ進むかわからなくなるが、それで n から先に進むのに価値がどれほどあるかについてのアイデアをもつてているということである。

完全なアルゴリズムをここで再現するには複雑すぎるが⁵、一言述べておくべきことがある。SMA*は最良の葉ノードを展開し、最悪の葉ノードを消去することを述べた。もしすべての葉ノードが同じ f 値をもっていたとするどうなるだろうか？ このとき、このアルゴリズムは同じノードに対して削除と展開の両方を選択するかもしれない。SMA*ではこの問題を最新の最良の葉ノードを展開し、最古の最悪の葉ノードを消去することによつて解決している。たった一つのノードによって、両方が同じノードになるかもしれない。そのようなときは、現在の探索木は、メモリのすべてを埋めるような、ルートノードから葉ノードへの一絆路に違いない。その葉ノードがゴールノードでなければ、 “たとえそれが最良経路の上にあつたとしても”， 使えるメモリだけでは解に到達できない。したがって、そのノードは、その後続ノードがないかのように正しく捨てることができる。

SMA*は、到達できる解があるとき、すなわち一番浅いゴールノードへの深さ d が(ノードで表現したときの)メモリサイズよりも小さいとき完全である。SMA*はどれかの最適解が到達可能なときに最適である。どの最適解も到達可能でなければ、最良の到達可能な解を計算する。現実的な方をすると、SMA*は最適な解を見つけるのに一番良い汎用のアルゴリズムと言つてよい。特に状態空間がグラフ構造であり、ステップのコストが均一でなく、ノードの生成がオープンとクローズのリストを管理する余計なオーバヘッドに比べて高価なときに最もである。

しかしながら非常に困難な問題のときは、SMA*は解経路の候補の集合の間を、メモリに収まる部分集合の範囲で、連続的に行ったり来たりせざるをえないことがしばしば起こる(これはディスクページシングシステムにおけるスラッシング(thrashing)の問題に似ている)。同じノードを繰返し生成するのに特別の時間が必要であるということは、メモリが無限ならば A*で実際に解できた問題が、SMA*では解けないということである。すなわち、メモリの制限があると、計算時間の観点から問題が解決できないことがあります。時

間にメモリの間のトレードオフを説明する理論はないが、これは避けられない問題のように思われる。これから抜け出す唯一の方法は、最適性という条件を捨てる。

より良い探索のための学習

我々は横型探索や欲張り最良探索などのように、コンピュータ科学者によって設計され定着した戦略を説明してきた。エージェントはより良く探索する方法を学習することができただろうか。その答えはイエスであり、その方法はメタレベル状態空間 (metalevel state space) とよばれる重要な概念によるものである。メタレベル状態空間での各状態は、Romania のようなオブジェクトレベル状態空間 (object-level state space) で探索をしているプログラムの内部(計算)状態を把握している。たとえば、A*アルゴリズムの内部状態は現在の探索木からなっている。メタレベル状態空間での各アクションは内部状態を変更する計算ステップである。たとえば、A*の各計算ステップは葉ノードを展開し、その後ノードをその木に追加する。このように、図4.3は探索木がだんだん大きくなる系列を示しているが、この図は、経路上の各状態がオブジェクトレベルの探索木であるようなメタレベル状態空間での経路を表したものと考えることができる。

さて、図4.3での経路は五つのステップからなっている。その中にはFagarasを展開した1ステップが入っているが、これはそんなに役に立っていない、より難しい問題では、そのようなミスのステップがたくさん起こり、メタレベル学習 (metalevel learning) アルゴリズムはこれらの経験から、有望でない部分木を展開するのを避けるよう学習することができます。この種の学習に使われる技法は21章で説明する。学習の目的は、計算の費用と経路コストのトレードオフがあるとき、問題解決のトータルコスト (total cost) を最小にすることがある。

4.2 ヒューリスティック関数

この節では、ヒューリスティックス一般の特性に光を当てたため、8バスルのためのヒューリスティックを取り上げる。
8バスルとは、ヒューリスティック探索問題の最も初期の例題の一つである。3.2節で述べたように、バスルの目的はタイルを空いたスペースに縦にスライドさせてゴール配置に一致させることにある(図4.7)。
ランダムに作り出した8バスルの平均の解コストは約22ステップである。分岐度(は)は3である(空いたスペースが中央なら4になり、角だと2であり、辺では3である)。ということは、深さ22のしらみつぶし探索はおよそ $3^{22} \approx 3.1 \times 10^{10}$ 状態になるということである。状態の繰返しに注意するだけでこの値を170,000程度まで削減できる。到達できる別個の状態が $9!/2 = 181,440$ しかないからである(練習問題3.4参照)。これは管理できる値ではあるが、15バスルではこの値はおよそ 10^{13} にもなる。そこで、次になすべきことは良いヒューリスティック関数を見つけることである。もしA*を使って最短解を求めたいのであれば、ゴールまでのステップ数を決して過大に見積もらないようにヒューリスティック関数が必要である。15バスルにおけるそのようなヒューリスティックスには長

⁵ ラフな記述は本書の第1版でなされている。



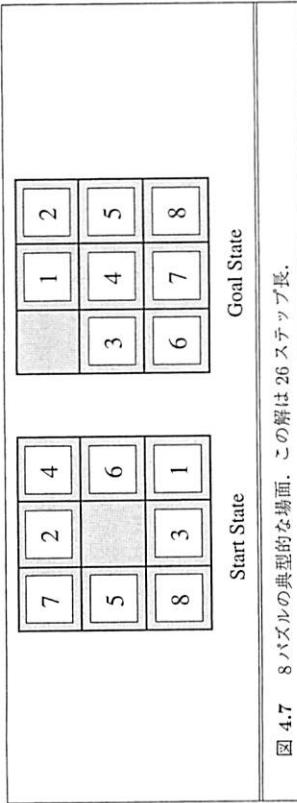


図 4.7 8 パズルの典型的な場面。この解は 26 ステップ長。

い歴史がある。ここでは二つのよく使われる候補を示す。

- h_1 = ゴールの位置にないタイルの数。図 4.7 は 8 枚のタイル全部がゴールの位置にないので、出発状態では $h_1 = 8$ となる。正位置からはずれたタイルは少なくとも 1 回は動かなければいけないから、 h_1 は滑容的ヒューリスティックである。
- h_2 = ゴール状態からのタイルの距離の和。タイルは対角線上を動けないので、計算する距離は縦と横の距離の和である。この距離は街角距離 (city block distance) またはマンハッタン距離 (Manhattan distance) とよばれる。 h_2 も許容的である。いかなる操作もタイルを目標へ 1 ステップ近づけることしかできないからである。出発状態のタイル 1 から 8 のマンハッタン距離は

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

になる。

我々が望むように、どちらのヒューリスティック関数も真的解のコスト 26 を過剰に見積もることはない。

効率におけるヒューリスティックの正確さの影響

ヒューリスティックスの質を特徴づける一つの方法は有効分岐度 (effective branching factor) b^* を使うことである。もし特定の問題において A^* で作られたノードの数の総数が N で、解の深さが d ならば、 b^* は深さ d の均一の木が $N+1$ のノードをもつために必要な分岐度である。したがって、

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

である。たとえば、 A^* が 52 のノードを用いて深さ 5 の解を見つけたとすれば、有効分岐度は 1.92 となる。あるヒューリスティックスに対する有効分岐度は、問題事例に応じて変化するが、十分に難い問題ではその値は普通はほぼ一定である。そのため少數の問題で b^* を実験的に見積もると、そのヒューリスティックスの全体的な有用性を判断するための良いガイドとなる。うまく設計されたヒューリスティックでは b^* は 1 に近い数字になり、かなり大きな問題でも解くことができる。

ヒューリスティック関数 h_1 と h_2 をテストするために、解の長さが 2 から 24 まで (それが解の長さについて 100 個) の問題 1,200 個をランダムに生成し、反復深化と、 h_1 と h_2 の A^* 木探索を用いてそれらを解いた。図 4.8 は各手法で展開した平均ノード数と有効

分岐度である。この結果は、 h_2 が h_1 より優れていること、反復深化探索 (IDS) よりはる

かに優れていることを示している。長さ 14 の我々の解では、 h_2 を用いて A^* を行った場合

は情報なしの反復深化に比べて 30,000 倍も効率的である。

d	Search Cost			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2,45	1.79	1.79
4	112	13	12	2,87	1.48	1.45
6	680	20	18	2,73	1.34	1.30
8	6384	39	25	2,80	1.33	1.24
10	47127	93	39	2,79	1.38	1.22
12	3644035	227	73	2,78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

図 4.8 ITERATIVE-DEEPENING-SEARCH アルゴリズムと h_1 , h_2 の A^* アルゴリズムの探索コストと有効分岐度の比較。データは 8 バスルの解の長さごとの 100 の問題の平均をとっている。

h_2 は常に h_1 より優れているのであるうかと尋ねるかもしない。答えはイエスである。二つのヒューリスティックスの定義から、いかなるノード n においても $h_2(n) \geq h_1(n)$ が成り立つことを示すのは容易である。このとき h_2 は h_1 より優位に立つ (dominate) といふ。優位はそのまま効率につながり、 h_2 を使った A^* よりも平均で少ないうノードしか展開しない ($f(n) = C^*$ であるようないくつかのノードで起こりうることを除けば)。このことは以下の簡単な議論からわかる。100 ページでの $f(n) < C^*$ となるすべてのノードが必ず展開されるという観察を思い出そう。これは、 $h(n) < C^* - g(n)$ のすべてのノードで等しい ($f(n) = C^*$) であるから、 A^* で展開されるすべてのノードは h_1 を使っても展開され、 h_1 ではさらに他のノードも同じように展開されないかもしれない。したがって、過大な見積りをせず、またヒューリスティックの計算時間が大きすぎない限り、大きい値のヒューリスティック関数を用いたほうが常に良い結果が得られる。

許容的なヒューリスティック関数の考察

h_1 (異なるタイル位置の数) と h_2 (マンハッタン距離) がともに 8 パズルのヒューリスティックスとしてかなり良いものであることを見てきた。ではどのようにして h_2 を作るのだろうか。コンピュータがこのようなヒューリスティック関数を機械的に作ることは可能なのだろうか?

h_1 と h_2 は 8 パズルの残りの経路の長さを見積もつたのであるが、これらは 8 パズルの単純化版での完全に正確な経路長にもなっている。もしこのパズルのルールを、隣接した空きスペースにしかタイルを動かせないという代わりに、どこへでも動かせるようになれば、 h_1 は最短解のステップ数を正確に与えている。同様にルールを、たとえすでにタイルが置いてある場所であっても、どの方向にも一つ動かせると言えたとすれば、 h_2 は最短解のステップを正確に与えている。行為に対する制限を減らした問題のことを弱条件問題 (relaxed problem)

(lem) とよぶ、弱条件問題の最適解のコストは、もとの問題の許容的なヒューリスティックスになっている。というのは、定義から、もとの問題の最適解は弱条件問題の解でもあり、そのためにもとの問題の最適解は弱条件問題での最適解と少なくとも同程度にコストがかかるに違いないからである。求まったヒューリスティックスは弱問題の正確なコストであるから、三角不等式を満足し、無矛盾 (consistent) である (99 ページを参照)。

無矛盾

もし問題の定義が形式言語で書き下ろされていれば、弱条件問題を作成的に作ることができることである。⁶たとえば、もし 8 パズルの行為が、

$A \in B$ の左右上下に隣接しかつ、 B が空いていれば、 A は B へタイルを動かす

ことができる。一つもしくはそれ以上の条件を取り除いて三つの弱条件問題を作ることができる。

(a) A が B の隣であれば A から B へタイルを動かすことができる。

(b) B が空いていれば A から B へタイルを動かすことができる。

(c) A から B へタイルを動かすことができる。

(a) から、我々は h_2 (マンハッタン距離) を作ることができる。もし各タイルを順番に目標に向かって動かすなら、 h_2 は適切なスコアになるからである。(b) から導かれるヒューリスティックスは練習問題 4.9 で議論される。(c) から h_1 (異なったタイル位置の数) を導くことができる。これは、もしタイルが「目的の場所」へ 1 ステップで動かせるなら、 h_1 が適切なスコアとなるからである。このような技法で生成された弱条件問題は“探索することなく”解くことができるのは大変重要なことには注意すべきである。というのは、弱条件問題のルールでは、この問題は八つの独立した部分問題に分割することができるからである。もし弱条件問題が解けないほど難しいなら、それに対応したヒューリスティックスは獲得するのにコストがかかることがある。⁷

ABSOLVER というプログラムは、“弱条件問題方式”や他の様々な手法を用いて問題定義からヒューリスティックスを自動的に作り出す (Prideditis, 1993)。ABSOLVER は 8 パズルでこれまでのヒューリスティックスより良い新しいヒューリスティックスを発見し、さらに有名なルービックキューブで最初の有用なヒューリスティックスを発見した。新しいヒューリスティックス開数を作るときの問題は、しばしば“明らかに最も”的”ヒューリスティックスを作り損うことである。もしある問題に対して、 $h_1 \dots h_m$ という許容的ヒューリスティックスが使えたとして、そのどれもが他に対しても優位でないと、どれを選べばよいのであらうか？ これから明らかにするように、どれか一つを選ぶ必要はないのである。

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

と定義すれば最も優良のヒューリスティックスが得られる。この合成ヒューリスティックスは問題となっているノードごとに最も正確な開数を用いる。それぞれの要素ヒューリスティックスが許容的なので h も許容的である。 h が無矛盾であることを証明するのも容易である。しかも、 h はすべての要素ヒューリスティックスより優位にある。許容的なヒューリスティックスは、与えられた問題の部分問題 (subproblem) の解コストの合計がまだに全問題を解くコストの下限にあることは簡単にわかる。これがが分離パラメータベース (disjoint pattern databases) の裏にあるアイデアである。このようにヒューリスティックスが使われる場合に使うことである。

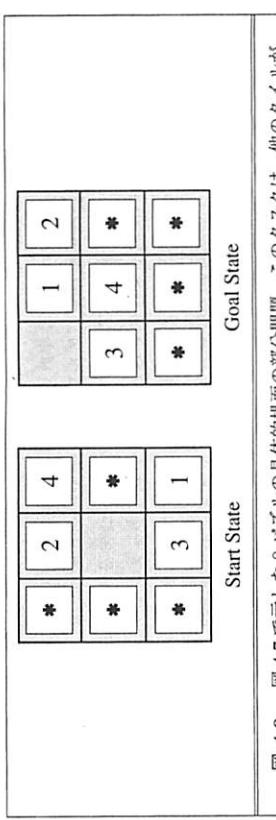


図 4.9 図 4.7 で示した 8 パズルの具体的な場面の部分問題。このタスクは、他のタイルがどうなるかは気にせずに、タイル 1, 2, 3, 4 を正しい位置に移動することである。

からも導くことができる。たとえば、図 4.9 における 8 パズルの部分問題の例を表している。この部分問題はタイル 1, 2, 3, 4 を正しい位置にすることである。この部分問題の最適解のコストは完全な問題のコストの下限であることは明らかである。あるケースでは、このヒューリスティックスはマンハッタン距離よりも実質的に正確であることが明らかになっている。

バターンデータベース (pattern databases) の裏にあるアイデアは、すべての可能な部分問題——我々の例では、四つのタイルと空白の可能なすべての配置——に対して正確な解コストを記憶させることである（この部分問題を解く際に他の四つのタイルの位置は無関係であるが、これらのタイルを動かすにはコストがかかることに注意しなくてはならない）。そして、探索の間に出会った完全な状態のそれぞれでデータベース中の対応する部分問題構成を単に参照することによって、許容可能なヒューリスティックス h_{DB} を計算する。そのデータベース自体は、ゴール状態から後向きに探索し、新しいパートンに出会うたびにコストを記録することによって構築されている。この探索にかかる総コストはその後の多くの具体的問題で回収される。

1-2-3-4 のタイルの選択は任意に選んだものである。5-6-7-8 や 2-4-6-8 などのタイルでもデータベースを作ることができる。どのデータベースも許容的ヒューリスティックスをもち、すでに説明したように、その最大値をとることによって、これらのヒューリスティックスを結合することができる。このように結合されたヒューリスティックスはマンハッタン距離よりも正確である。ランダムな 15 パズルを解くのに作られるノードの数を 1,000 分の 1 のオーダで減少することができる。

1-2-3-4 のデータベースから求まつたヒューリスティックスと 5-6-7-8 のデータベースから求まつたヒューリスティックスは重なりがないから“たしあわせる”ことができる。ではないかと考えるかもしれない。たしかに、まだ許容的なヒューリスティックスであろうか？ 答えはノーである。与えられた状態に対する 1-2-3-4 の部分問題の解と 5-6-7-8 の解はほとんど確実にいくつかの動きを共有し、1-2-3-4 は 5-6-7-8 に触れることがなく動かすことや、その逆はできなさそうだからである。しかし、これらの動きをカウントしないければどうであろうか？ すなわち、1-2-3-4 部分問題を解くのにトータルコストを記録するのではなく、1-2-3-4 に関する動きの数だけをカウントするのである。すると、二つのコストの合計がまだに全問題を解くコストの下限にあることは簡単にわかる。これがが分離パラメータベース (disjoint pattern databases) の裏にあるアイデアである。このようにヒューリスティックスを使うことによって、全部の構築盤探査をさせることで獲得することができる。

分離パターンデータベース

⁶ 8 章と 11 章で、このタスクに適した形式言語を説明する。この形式言語では、操作可能な形式的記述をすることで、弱条件問題の構築を自動的に行うことができる。今のこところは、日本語を用いる。（原注：英語）
⁷ 完全なヒューリスティックスは h に“こいつ”と“全部の構築盤探査をさせることで獲得することができる。”このようにヒューリスティックスがマハッタン距離と、ランダム 15 パズルの場合、作られるノード数がトレードオフの関係にある。



離の場合に比べて 10,000 分の 1 のオーダで減少し、数ミリ秒で解くことができる。24 バスルの場合、およそ 100 万倍のスピードアップが得られる。

タイルを滑らすバスルでは、タイルのそれぞれの動きが一つの部分問題にしか影響しないように分割することができ、一時に一つのタイルだけしか動かせないので、分離ハターンデータベースが使えるのである。ルビックキューブの問題では、一つの動きが 26 の立方体のうち 8 か 9 の立方体に影響するので、このような分割はできない。今のところ、このような問題でどのように分離データベースを定義するかは明らかではない。

経験からのヒューリスティクスの学習

ヒューリスティック関数 $h(n)$ はノード n の状態から始まる解のコストを見積もることが想定されている。エージェントはどのようにしてそのような関数を作るのだろうか？ 一つの方法は前の節で述べたように、最適な解が簡単に求まるよう弱条件問題を考えることだった。別のある方法は経験から学習することである。ここでは“経験”とは、たとえば多くの 8 パズルを解くことである。8 パズル問題などの最適解も、 $h(n)$ が学習できるような例を提供する。それぞれの例は、解経路からの状態とその点からの解の実コストからなる。これらの例から帰納学習 (inductive learning) アルゴリズムを用いて、探索の間に発生する他の状態の解コストを（運が良ければ）予測するような関数 $h(n)$ を作ることができる。ニューラルネットや決定木や他の方法を使って、これを実行する技法は 18 章で示されている（21 章で説明する強化学習を使うこともできる）。

帰納推論による学習方式は、單に生の状態記述が与えられたときよりも、その評価に関する状態特徴 (feature) が与えられたときに一番よく働く。たとえば、“位置が異なっているタイルの数”という特徴は、ゴールからの実距離を予測するのに役立つかもしれない。この性質を $x_1(n)$ とよぼう。我々は 100 のランダムに生成された 8 パズルの配置を得て、実際の解コストの統計を集めることができた。我々は $x_1(n)$ が 5 であるときは平均の解コストが 14 である、などを見つけることができた。これらのデータが与えられると、 x_1 の値は $h(n)$ を予測するのに使うことができる。もちろん、複数の特徴を使うこともできる。二番目の特徴 $x_2(n)$ は“ゴール状態でも隣接しているタイルの対の数”的なものでよい、どのように $x_1(n)$ と $x_2(n)$ を組み合わせて $h(n)$ を予測するのだろうか？ よく使われるアプローチは線形結合を使うことである。

$$h(n) = c_1x_1(n) + c_2x_2(n)$$

定数 c_1 と c_2 は解コストにおいて実データにうまく適合するように調整される、おそらく、 c_1 を正の数とし、 c_2 を負の数とすべきである。

4.3 局所探索アルゴリズムと最適化問題

我々が今まで見てきた探索アルゴリズムは、探索空間をシステムティックに調べるようには設計されている。システムティック性は、一つ以上の経路を記憶し、経路上の各点においてどの候補を調べ、どの候補を調べていないかを記録することで実現されることであります。そのため、そのゴールまでの経路もその問題の解になっている。

4.3 局所探索アルゴリズムと最適化問題

しかし、多くの問題ではゴールへの経路は重要ではない、たとえば、8 クイーン問題 (66 ページをみよ)において、クイーンの最終配置が大事なのであって、クイーンが配置されていった順番は大事ではない。このクラスの問題には、集積回路の設計、工場のレイアウト、ジョブショップスケジューリング、自動プログラミング、通信ネットワークの最適化、乗物のルート設定、ポートフォリオの管理などの多くの重要な応用問題が含まれれる。もしゴールまでの経路が重要でなければ、経路をまったく心配しないという異なるたったクラスのアルゴリズムを考えることもできる。局所探索 (local search) アルゴリズムは、(複数の経路を考えるのでなく) 単一の現状態 (current state) を使って動作し、一般にその状態の隣接状態にのみ働く。通常は、その探索したがった経路は保持されない。局所探索アルゴリズムはシステムティックではないが、二つの重要な利点がある。(1) 非常に少ないメモリ——通常は定数個のメモリ——しか使わない。(2) システマティックなアルゴリズムでは適切でないような膨大な状態空間や、無限の（連続）状態空間においても合理的な解を見つけることができる。

局所探索アルゴリズムは、ゴールを探さなければなく、純粋な最適化問題 (optimization problem) を解くにも有効である。これは目的関数 (objective function) にしたがって最も良い状態を見つけることを目的とするものである。多くの最適化問題は 3 章で紹介した“標準的な”探索モデルには合わない。たとえば、自然界には、ダーウィンの進化論がそれを最適化しようとしているとみなせるかもしれない目的関数——生殖の適合性——がある。しかし、この問題には“ゴールテスト”も“経路コスト”も存在しない。

局所探索を理解するために、(図 4.10 に示すように) 状態空間俯瞰図 (state space landscape) を考えることが非常に役に立つことを示す。俯瞰図は“（状態で定義される）位置”と“（ヒューリスティックコスト関数が目的関数の値で定義される）高度”の二つをもつている。もし高度がコストに相当するものならば、その目的は一番低い谷——大局的最小値 (global minimum)——を見つけることになる。もし高度が目的関数に相当するものならば、その目的は一番高いピーク——大局的最大値 (global maximum)——を見つけることになる。(両者は、マイナス記号を挿入するだけで相互に変換することができる。) 局所探索アルゴリズムはこの俯瞰図を探査する。完全な (complete) 局所探索アルゴリズムはゴールが存在すれば必ずゴールを見つける。最適化 (optimal) アルゴリズムは必ず大局的最小値/最大値を見つける。

山登り探索

山登り (hill-climbing) 探索アルゴリズムを図 4.11 に示す。これは、値が増える方向に単に連続的に動く——すなわち山登りする——ループである。これは、その隣接地のはうが高い値をとることがないような“ピーク”に達すると停止する。このアルゴリズムは探索木を保持していないので、現在のノードのデータ構造は状態とその目的関数の値だけを記録すればよい。山登り法は現在の状態の直接の隣接地を越えて先読みみをすることはない。これは記憶喪失のときには無い霧の中でエベレスト山の頂上を探索することに似ている。山登り法を説明するのに 66 ページで紹介した 8 クイーン問題 (8-queens problem) を使う。局所探索アルゴリズムは通常、八つのクイーンが各列に一つずつ配置されているような完全な状態記述 (complete-state formulation) を使う。後者関数は一つのクイーンを同じ別の位置に移動することによって作り出すことができる状態をすべて計算する（ここで

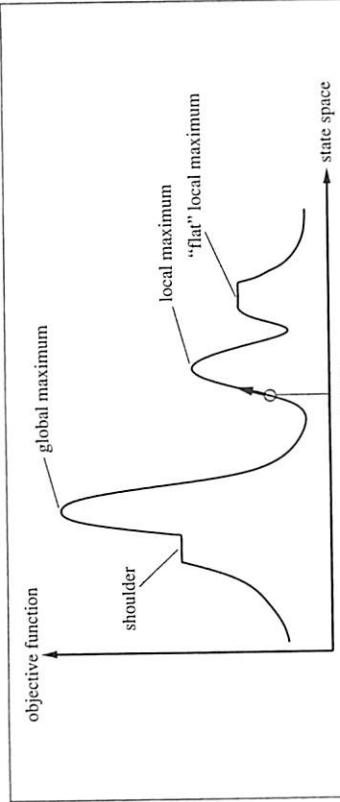


図 4.10 上方向が目的関数に対応する 1 次元の状態空間の俯瞰図。この目的は大局的最大値を見つけることである。山登り探索は矢印で示したように、現在状態を改良するよう変更する。いろいろな地形的な特徴が言葉で定義されている。

```

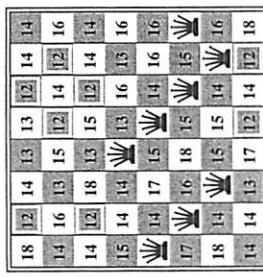
function Hill-Climbing(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
neighbor, a node
current←MAKE-NODE(INITIAL-STATE[problem])
loop do
  neighbor←a highest-valued successor of current
  if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
  current←neighbor

```

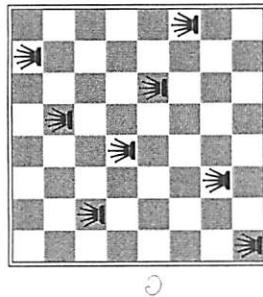
図 4.11 山登り探索アルゴリズム（最急勾配（steepest ascent）版）。これは最も基本的な局所探索法である。各ステップにおいて、最新のノードが最良の隣接ノードに置き換わる。この版では、これは最も高い値をもつたノードに置き換わるということである。しかし、もしヒューリスティックなコスト見積り h が使われるなら、最も低い h をもつたノードが見つけられることがある。

で各状態は $8 \times 7 = 56$ の後続状態をもつ）。ヒューリスティックコスト関数 h は、直接的か間接的かを問わず互いに攻撃し合うクイーンの対の数である。この関数の大局部的な最小値は完全な解のときだけにゼロである。図 4.12 (a) は $h = 17$ のときの状態を示している。この図にはすべての後続値と、その最良値が $h = 12$ であることが示されている。山登りアルゴリズムは最良の後続状態が二つ以上あるときは、通常はその中でランダムに選ぶ。

山登りは欲張り局所探索（greedy local search）とよばれることがある。これは次にどこへ行くかを前もって考えずに良い隣接状態に移るからである。欲張りは死に値する七つの罪の一ひとつだと考えられているが、欲張りアルゴリズムはしばしばうまく働くことを明らかにする。山登りは、悪い状態を改良することが容易なので、解に向かって非常に急速に進行する。たとえば、図 4.12 (a) の状態から図 4.12 (b) の状態に達するのに 5 ステップしかからない。図 4.12 (b) は $h = 1$ であり、解のすぐ近くである。しかし残念ながら山登りは次の理由で行き詰まることがしばしばある：



(a)



(b)

図 4.12 (a) ヒューリスティックのコスト見積りが $h = 17$ であるような 8 クイーンの状態、クイーンをその列内で移動したときの可能な後続状態の h の値が示されている。最も動きに印がついている。(b) 8 クイーンの状態空間における局所的小値、この状態では $h = 1$ となるが、その後続状態はどれもこれより大きな値となる。

◇ 局所的最大 (local maxima)：局所的最大は、隣接する状態よりも高い値をもつが、全局的最適値よりも小さなビーグルのことである。山登りアルゴリズムは局所的最大の周囲に達するとビーグルに向かって進んでしまって、どこへも行けなくなってしまう。図 4.10 はこの問題を図解している。より具体的に言うと、図 4.12 (b) の状態は局所的最大（コスト h では局所的最小）であり、一つのクイーンをどのように動かしても状態は悪くなってしまう。

◇ 尾根 (ridge)：尾根は図 4.13 に示している。尾根は局所的最大が連続することによってできるもので、欲張りアルゴリズムで導いていくことが非常に難しい。

◇ 高原 (plateaux)：高原とは、評価関数がフラットであるような状態空間の併置図の領域である。高原は、そこから登り坂が存在しないようなフラットな局所的最大や、そこからの改善が可能な肩 (shoulder) になることもある（図 4.10 参照）。山登り法は高原を脱出する道を見つけられないとされない。

どの場合も、アルゴリズムはそれ以上改善できないところまで進む。ランダムに作られた 8 クイーン問題からスタートして、最も激な登りを選ぶ山登りは回数にして 86% の問題事例が行き詰まり、14% しか解けなかった。アルゴリズムは早く動作し、成功するときは平均して 4 ステップであり、行き詰まるときは 3 ステップである。これは $8^8 \approx 1700$ 万状態ある状態空間にしては悪くない。

図 4.11 のアルゴリズムは、一番良い後続状態が現在の状態と同じ値をもつような高原に達すると停止する。図 4.10 に示すように進み続ける、つまり高原が実際は肩だという希望をもつて横移動 (sideways move) を許すことは良い考え方だろうか？ その答えは通常はイエスだが注意が必要である。もし登りの動きがないときには、そのアルゴリズムが肩でないフラットな局所的最大値に達するたびに無限ループに陥ってしまう。その一つのよく使われる解決法は、連続して横移動をする数に制限を設けることである。たとえば、図 4.12 (a) の状態から図 4.12 (b) の状態に達する横移動を 100 まで認めるようにする。これによって、山登りで解くことができる問題の比率が 14% から 94% にアップする。その成功は

肩

横移動

欲張り局所探索

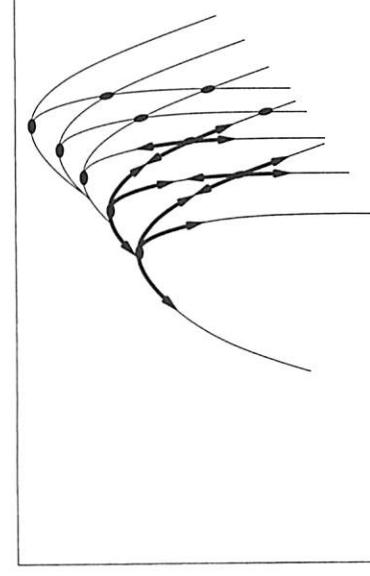


図 4.13 なぜ山登り法において尾根が困難な問題を引き起こすかを示す図。状態の格子（黒い円）が左から右へ登ついく尾根に重ねて表示され、直接には結合していない局所的最大値の列を作っている。それぞれの局所的最大値は下方向を指す。

コストにも現れ、このアルゴリズムは成功する場合は平均しておよそ 21 ステップであり、失敗する場合は 64 ステップである。

山登り法の変形版がたくさん考案されている。確率的山登り法 (stochastic hill climbing) は登る動きの中からランダムに選択する。この選択の確率は登りの険しさに応じて変えることができる。この方法は一番険しい登り坂を選択する方法に比べて、一般的には収束が遅いが、ある状態俯瞰図ではこのほうが良い解を得られる場合がある。第一選択山登り法 (first-choice hill climbing) は、現在よりも良い状態を見つかるまでランダムに後続の状態を作ることによつて、確率的な山登り法を実現する。これは状態が多くの（たとえば、数千の）後続状態をもつときは良い戦略である。練習問題 4.16 は、これを調べるように読者に要求している。

今まで説明してきた山登り法は不完全である。すなわち、局所的な最大値で行き詰まる可能性があるため、解があつてもそれを見つけられないことがしばしばある。ランダム再スタート山登り法 (random-restart hill climbing) は“もし最初に成功しなかつたとしても、何度も何度も試みよ”という有名な格言を採用している。この方法はランダムに初期状態⁸を作り出しして一連の山登り探索を行い、ゴールが見つかったら終了する。これは 1 回に近い確率で完全である。これは偶然にゴール状態を初期状態として作つてしまふかもしれないという明らかなる理由からである。もし山登り探索をするごとに p の確率で成功するなら、必要な再スタートの期待値は $1/p$ である。横移動が許されない 8 ケイーンの例では、 $p \approx 0.14$ であり、そのため、目標を見つけるのにおよそ 7 回の繰返し（6 回の失敗と 1 回の成功）を必要とする。ステップの期待値は 1 回の成功する繰返しコストと $(1-p)/p$ と失敗コストの積をしたものであり、およそ 22 ステップとなる。横移動を許すならば、平均して $1/0.94 \approx 1.06$ の繰返しが必要となり、 $(1 \times 21) + (0.06/0.94) \times 64 \approx 25$ ステップとなる。したがつて、8 ケイーンではランダム再スタート山登り法は実際に非常に効率的である。

シミュレーテッドアニーリング探索

ある、300 万ケイーンでも、このアプローチは 1 分以内に解を見つけることができる。⁹ 山登り法の成功は状態空間の俯瞰図の形に依存する。もし局所的最大値や高原が少ししかなければ、ランダム再スタート山登り法は良い解を非常に高速で見つけることができる。その一方、実問題の多くは平らな床の上にヤマアラシの家族がいて、そのヤマアラシのそれをの針の上にミニチュアのヤマアラシがいて、というのが無限に続くような俯瞰図の形をしている。通常は、NP 困難な問題は行き詰まりを起こすような局所的最大値が指数的な数だけある。それにもかかわらず、少ない数の再スタートによって、かなり良い局所的最大値を見つけることもしばしばある。

山登り法は、低い値をもつ（または高いコストの）状態に向けて“坂を下る”動きを“決して”せず、局所探索に陥る可能性があるから不完全であることがわかっている。それに対し、後続状態の集合の中から一様にランダムに後続状態を選んで動くといふ、純粋なランダムウォーカーは完全であるが著しく効率が悪い。したがつて、効率性と完全性が成る、焼きなましまし法 (simulated annealing) はそのようなアルゴリズムである。冶金学においてアニーリング (annealing) とは金属やガラスを高温に熱し、徐々に冷やすことによって鍛えたり硬くしたりするプロセスである。こうすることによって、金属が合体してエネルギーが低い結晶状態になるのである。焼きなましまし法 (gradient descent) (すなわち、コストの最小化) に切り替え、ピンポン球をでこぼこの面の一一番深い裂け目に入れることを想像してみよう。もし球を回転させただけだと局所的最小値で止まってしまう。もし面をゆすると、球を局所的最小値から球を取り出させることができる。そのコツは、局所的最大値から球を取り出させるくらい激しくゆするが、大局的最大値から外に出るほどは激しくない程度にゆすることである。焼きなましまし法の解は、激しくゆする（高温時に）ことから始めて次第にゆする強さを減らす（より低温時に）ことである。

焼きなましまし法アルゴリズム（図 4.14）の最も内側のループは山登り法とよく似ている。しかしながら、最良の動作を選ぶ代わりに、こちらはランダムな動作を選ぶ。もしその動作が状態を良くするなら、その動作はいつも採用される。状態を良くしないなら、このアルゴリズムはより小さい確率でこの動作を採用する。この確率は動作の“悪さ”——評価が悪くなつたことを示す ΔE の値——の指數オーダーで小さくなる。この確率は“温度” T が低くなつても低下する。最初に温度が高いときは受け入れやすいが、 T が低下するにつれて受け入れにくくなる。もし T が十分にゆっくりと低下するようになると、このアルゴリズムは 1 に近い確率で大局的な最大値を見つけることを証明することができる。

焼きなましまし法は最初 1980 年代の初期に、VLSI のレイアウト問題を解くために盛んに使われた。それ以来、工場のスケジューリングや他の大規模最適化問題に広く使われてきた。練習問題 4.16 ではクイーンパズルについて、焼きなましまし法の効率とランダム再スタート山登り法との特徴から、ランダム状態を生成すること自体が困難な問題である。

⁸ 路點の内に特定された状態から、ランダム状態を生成すること自体が困難な問題である。

⁹ Luby, Sinclair, and Zuckerman (1993) は、ある場合には、特定の時間の経過後にランダム探索アルゴリズムを再スタートするのが一番良いことと、このことが個々の探索を無限に続けるよりもこの例がある。

```

function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                    next, a node
                    T, a "temperature" controlling the probability of downward steps

current  $\leftarrow$  MAKE-NODE([INITIAL-STATE][problem])
for t  $\leftarrow$  1 to  $\infty$  do
  T  $\leftarrow$  schedule[t]
  if T = 0 then return current
  next  $\leftarrow$  a randomly selected successor of current
   $\Delta E \leftarrow \text{VALUE}[\text{next}] - \text{VALUE}[\text{current}]$ 
  if  $\Delta E > 0$  then current  $\leftarrow$  next
  else current  $\leftarrow$  next only with probability  $e^{\Delta E / T}$ 

```

図 4.14 焼きなまし法(探索アルゴリズム)で、いくつかの下向きの移動が許されている確率的山登り法の版。下向きの動きは焼きなまし法のスケジュールの中であらかじめ許されおり、時間が経過するにつれてその動きは少なくなる。入力引数のスケジュールは時間の関数として値 *T* を決定する。

ト山登り法の効率の比較を求めることが求められる。

局所的ビーム探索

最初は、*k* 個の状態を用いた局所的ビーム探索は、メモリの制約問題への極端な対応かもしれない。実際には、この二つのアルゴリズムは一つの状態だけではなく、*k* 個の状態を記録する。このアルゴリズムは *k* 個のランダムに作られた状態から始まる。各ステップにおいて、*k* 個のすべての状態について、すべての後続状態が作られる。もしその一つがゴール状態なら、このアルゴリズムは停止する。ゴール状態がなければ、完全なリストから良い順に *k* 個までの後続状態を選んで繰り返す。

最初は、*k* 個の状態を用いた局所的ビーム探索は、*k* 回のランダム再スタートを順番にやる代わりに並列に実行するにすぎないと思うかもしれない。実際には、この二つのアルゴリズムはかなり異なるものである。ランダム再スタート探索では、各探索ステップは他とは独立に実行される。局所的ビーム探索では、*k* 個の並列探索処理の間で有用な情報が伝達される。たとえば、もしある状態が複数の良い後続状態をもち、他の *k* - 1 個の状態が良くない後続状態しかないとき、最初の状態は他の状態へ“ここへ来なさい、こちらの車のほうがよけい青いよ!”とびがける効果が生じる。このアルゴリズムはただちに役に立たない探索をやめて、一番改善がなされているところへ資源を移す。

局所的ビーム探索は一番単純な形式では、*k* 個の状態の間の多様性のなさという問題が起きるかもしれない。これは *k* 個の状態が状態空間の狭い範囲に急速に集中され、この探索が山登り法の高価な版と大差なくなるという問題である。確率的ビーム探索(stochastic beam search)によばれる変形版は、確率的山登り法と類似し、この問題を多少は解決するのを助けることができる。確率的ビーム探索は、候補となる後続状態の候補の集合から最も *k* 個を選ぶ代わりに、後続状態からの選択確率がその値の増加関数になるように、*k* 個の後

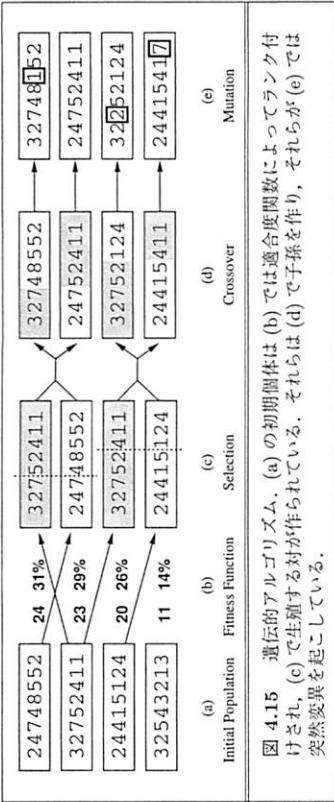


図 4.15 遺伝的アルゴリズム。(a) の初期個体は (b) では適合度関数によってランク付けされ、(c) で生殖する対が作られている。それらは (d) で子孫を作り、それらが (e) では突然変異を起こしている。

競状態をランダムに選択する、確率的ビーム探索は、“状態”(生物)の“後続”(子孫)はその“値”(適合性)に応じて次世代の人口が決まるという自然選択のプロセスと似ているところがある。

遺伝的アルゴリズム

遺伝的アルゴリズム(genetic algorithm: GA)は、一つの状態を修正することによるので後続状態を作られるという確率ビーム探索はなく、二つの親状態を結合することによって後続状態を作られるという確率ビーム探索の変形である。自然選択への類比は確率的ビーム探索と同じであるが、無性生殖ではなく有性生殖となっているところが異なる。

ビーム探索の場合と同様に、GA は個体群(population)と呼ばれる、ランダムに生成された *k* 個の状態集合から始まる。各状態すなわち個体(individual)は有限のアルファベットの文字列——多くの場合は 0 と 1 の列——として表される。たとえば、8 クイーンの状態は八つのクイーンの列の八つの正方形の位置を記述するもので、 $8 \times \log_2 8 = 24$ bit が必要とする。または、この状態は 1 から 8 までの 8 進数で表すことができる(あとで、この二つの符号化は異なる振舞いをすることを説明する)。図 4.15 (a) は八つのクイーンの状態を表す四つの 8 進数の文字列の個体群を示している。

次世代の状態の繁殖状況が図 4.15 (b)～(e) に示されている。(b) では、それぞれの状態は評価関数、すなわち(GA の用語で)適合度関数(fitness function)を使って評価づけされている。適合度関数はより良い状態に対してもより高い値をとるようにしなければいけないので、8 クイーン問題では我々は互いに攻撃し合わないクイーンのペアの数を使っている。この値は解においては 28 となる。図の四つの状態の値は 24, 23, 20, 11 である。遺伝的アルゴリズムのこの変形版では、生殖のために選ばれる確率をこの適合のスコアと正比例するようとしている。そのパーセンテージはもとの状態の値の横に示している。

(c) では、生殖を行うため、(b) の確率に合わせて、二つのペアがランダムに選ばれる。個体は 2 度選択されるものとまったく選択されないものとがあることに注意する必要がある。11 生殖を行うペアのぞれぞれで、交叉(crossover)する点が文字列の位置からランダムに選ばれる。図 4.15 (d) では交叉の点は最初のペアの三番目の文字の直後と、二番目のペアの文 又

¹¹ この選択ルールにはいろいろな変形がある。ある數値よりも低い個体をすべて捨ててしまうという摘み取り(culling)方式はランダム方式より早く収束するのを示すことができる(Baum et al., 1995)。

¹⁰ 局所的ビーム探索は経路に基づくアルゴリズムのビーム探索(beam search)を適用したものである。

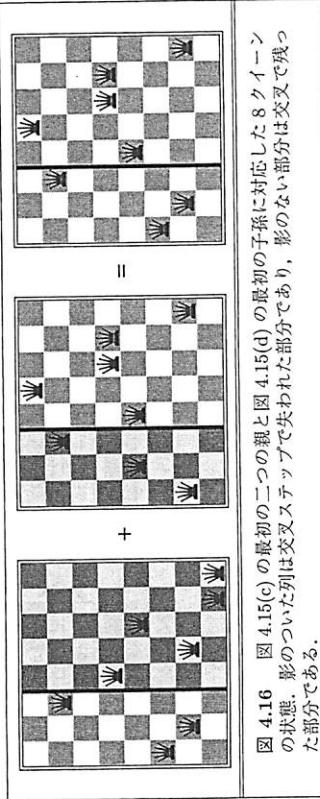


図 4.16 図 4.15(c) の最初の二つの親と図 4.15(d) の最初の子孫に対応した 8 クイーンの状態、影のついた列は交叉ステップで失われた部分であり、影のない部分は交叉で残った部分である。

五番目の文字の直後である。¹²

(d) では、交叉する点で親の文字列を交叉させることで子孫の文字列が作られている。たとえば、最初のペアの最初の子供は第一の親から 3 文字をもらい、第二の親から残りの文字をもらっている。また、二番目の子供は第二の親から 3 文字をもらい、第一の親から残りの文字をもらっている。この生道プロセスにおける 8 クイーンの状態は図 4.16 に示している。この例は、二つの親がまったく違うものであるとき、交叉の操作をすると、どちらの親からも遠い状態を作り出すことができる。このプロセスの初期に個体群が非常に多様であることが示している。この交叉は（焼きなまし法と同じように）探索処理の初期においては大きな進化を遂げ、あとになつてはほとんどどの個体が似ているときには小さな進化となることが多い。

最後に (e) では、文字列の各位置で小さな独立の確率でランダムな突然変異 (mutation) が生じている。一番目と三番目と四番目の子孫で 1 文字に突然変異が生じている。8 クイーン問題では、これはクイーンをランダムに選び、その列のランダムな位置に移動することに相当する。図 4.17 はこれらすべてのステップを実行するアルゴリズムを示している。確率的ビーム探索と同じように、遺伝的アルゴリズムはランダム探査による山登りと、並列探索間の情報交換を結合したものである。遺伝的アルゴリズムのほうに利点があるとすれば、それは交叉の操作によるものである。しかし、遺伝特号の位置が初期にランダムに並べ替えられると、交叉の利点はないことが数学的に証明できる。直観的には、遺伝的アルゴリズムの利点は、独立に進化して有用な機能をもった文字の固まりを交叉させ、探索が実行する粒度のレベルを高める、という交叉の能力からきているのである。たとえば、最初にクイーンを 2, 4, 6 の位置におくこと（これらは互いに攻撃しない）は、解を作るために他のプロックと結合できる有用なプロックである。

遺伝的アルゴリズムの理論によると、スキーマ (schema) の考え方を使ってどのようにこのアルゴリズムが働くかを説明できる。スキーマとは、そのいくつかの部分が未定義であるような部分文字列である。たとえばスキーマ 246***** は、8 クイーンにおいて、最初の三つのクイーンが 2, 4, 6 の位置にあるようなすべての状態を表している。そのスキーマに適合する文字列 (24613578 のように) は、そのスキーマのインスタンス (instance) と呼ばれる。もし、スキーマのインスタンスの平均適合度が平均値より大きければ、個体群の

```

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

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

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

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

```

図 4.17 遺伝的アルゴリズム、このアルゴリズムは図 4.15 で図示したものに一つの変形を行つたものと同じである。より一般的な版では、二つの親の各生殖は二つの子孫ではなく一つの子孫だけを作り出す。

中におけるそのスキーマのインスタンスの数は時間とともに増加することを示すことができる。もし隣合うビットが全体として無関係ならば、一貫した利益をもたらすような隣接するブロックはほとんどないので、この効果は重要ではない。遺伝的アルゴリズムはスキーマが解の重要な部品となっているときに一番よくはたらく。たとえば、もし文字列がアンテナの表現となっているなら、スキーマは反射板や偏針儀のようなアンテナの部品を表現しているかもかもしれない。良い部品はいろいろ異なる設計に対しても良い部品となる傾向が高い。このことにより、遺伝的アルゴリズムを使つてうまくいくには、表現についての注意深い技法が必要となることがわかる。

実際、遺伝的アルゴリズムは回路のレイアウトやジョブシフタリングのような最適化問題に幅広いインパクトを与えた。現在のところ、遺伝的アルゴリズムの人気がその効率からきているのか、進化理論での審美的に満足できる起源からきているのかは明らかでない。遺伝的アルゴリズムがうまく働くための条件を見つけるため、しなくてはならない多くの研究が残されている。

4.4 連続空間における局所検索

¹² 符号化が問題となるのはここである。もしハフマの代わりに 24 ビット符号化が用いられたら、交叉の行われる点は 2/3 の確率で一つの数字の内部となる。その結果、その数字に本質的に任意の突然変異が起こる。

進化と探索

進化(evolution)の理論は Charles Darwin の *On the Origin of Species by Means of Natural Selection* ([自然淘汰による種の起源]) (1859) で展開された。その中のアイデアは以下のようすに単純である。生殖において (突然変異 (mutation) として知られている) 変異が起り、生殖個体の適合度の効果におよそ比例して後継の世代が保存される。

Darwin の理論は、器官の特徴がどのように受け継がれたり変形したりするかの知識なしに展開された。これらのプロセスを支配している確率法則は最初に Gregor Mendel (1866) によって見いだされた。彼は人工授精と彼がよんだ方法を使ってスイトーピーに実験を行った修道僧である。もっととどくなつて、Watson and Crick (1953) が DNA 分子の構造とそのアルファベット AGTC (アデニン, グアニン, チミン, シトシン) を見つけた。その標準モデルでは、文字列中のいくつかの地点の突然変異によっても、“交叉”(両親の DNA の長い文字列部分を結合することによって子孫の DNA が生成される) によっても変異が生じる。局所探索アルゴリズムとの類比については、すでに以下のように説明している。確率的ピーム探索と進化の重要な差異は性的な生殖を利用するかどうかである。これによつて後継者は唯一の生物からではなく、複数の生物から後継者があられる。しかし、実際の進化はほとんど遺伝的アルゴリズムが評しているよりもっと豊富である。たとえば、突然変異には、反転や、複製や、DNA の大きな塊の移動を含めることができ。また、あるウイルスは DNA をある生物から借りて別の生物へ挿入することができ。さらに、ゲノムの中に自分自身を何千回もコピーするだけの置換可能な遺伝子もある。生殖相手となりうるが遺伝子を伝搬しながらの細胞を毒し、それにより自分の複製のチャンスを大きくするような遺伝子さえある。最も重要なのは、それによってゲノムが複製され器官に翻訳されるようなメカニズムを遺伝子が自分で符号化していることである。遺伝的アルゴリズムでは、これらのメカニズムは操作される文字列の中では表現されない別のプログラムとなつている。

Darwin 流の進化は、探索ヒューリスティックスを少しも改良することなく盲目的に 10^{45} もの生物を生成してしまい、効率の悪いメカニズムのように思えるかもしない。しかし、Darwin の 50 年前に、別の偉大なフランスの自然学者の Jean Lamarck (1809) は進化的理論を提倡した。この理論では生物の一生の間に適応によって獲得した性質はその子孫に伝達される。そのようなプロセスは効率的であるが、自然に起こることはなさそうである。もっと最近では、James Baldwin (1896) は表面的には似た理論を提倡した。その理論は生物の一生の間に学習した振舞いが進化の比率を加速することができるというものである。Lamarck の理論と異なり、Baldwin の理論は Darwin の進化と完全につじつまが合う。というのは、彼の理論は、遺伝子の構成で許された可能な振舞いの集合の中に局所的最適値を見つけた個体に働く選択圧に依存しているからである。近代のコンピューターミュレーションによって、“通常の”進化によって内部の効率の値が実際の適合度に相關しているような生物を作れるなら、“Baldwin 効果”が現実のものであることを確認している。

かたった——後者関数は多くの場合無限に多くの状態を計算してしまう！ この節では、連続空間での最適解を見つけるための局所探索技法を非常に簡単に紹介する。この話題に関する文献は数多い。その多くの基本的な技法は、17世紀に Newton と Leibniz による計算法の開発のあとに起源をもつ。¹³ これらの技法の利用については、学習やビジョンやロボティクスに関する章など、この本の多くの場所で見つけられる。すなわち、実世界を扱うあらゆる場面で。

例から始めよう。Romania のどこかに三つの新しい空港を、地図 (図 3.2) 上の各都市から最寄りの空港までの距離の 2 乗の和が最小になるように設置したいとする。状態空間は空港の座標 $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ で定義する。これは“6 次元”的な空間であり、状態は六つの変数 (variable) で定義されるとも言う (一般に、状態は n 次元のベクトル変数 x で定義される)。この空間の中を動くことは、地図上で一つ以上の空港を移動することに相当する。目的関数 $f(x_1, y_1, x_2, y_2, x_3, y_3)$ は、最も近い都市を計算してしまえば、おのおの特定の状態でこの値を計算するのは比較的やさしいが、一般的にこれを書き下すのはコソッがいる。

¹³ 本章では多変数の計算やベクトル計算の基礎知識が役に立つ。

傾き

連続の問題を避ける一つの方法は、單に各状態の隣接関係を離散化 (discretization) することである。たとえば、一度には、一つの空港を x 方向か y 方向に固定量 δ だけ動かせただけとする。変数が 6 だとすると、これは各状態に 12 の後続状態があることになる。これに今まで述べた局所探索アルゴリズムのどれでも使うことができる。また空間を離散化することなしに、確率的山登り法や焼きなまし法を直に使うこともできる。これらのアルゴリズムは後続状態をランダムに選ぶ。これは長さ δ のベクトルをランダムに作ることによって実現される。

最大値を見つけるために、偏微の傾き (gradient) を使う方式がたくさんある。目的関数の傾きは最大の傾斜の大きさと方向を与えるベクトル ∇f である。我々の問題においては ∇f は以下のようになる。

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

ある場合には、方程式 $\nabla f = 0$ を解くことによって最大値を求めることができる (たとえば、もしたまた一つの空港を設置するなら、これが可能になる)。この方程式の解はすべての都市の座標の数値的な平均値となる)。しかしその場合には、この方程式を閉形式で解くことはできない。たとえば、三つの空港の場合、傾きの式は現在の状態でどの都市が各空港に近いかに依存する。これは、我々が傾きを大局的にではなく局所的にに計算できることを意味している。そうだとすれば、次の式で現在の状態を更新することで最急勾配山登り法を実行することができる。ここで α は小さな定数である。

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

別の場合には、目的関数は微分方程式の形ではまったく書けないかもしれない。たとえば、空港の位置に関する特定集合の値は、大規模な経済ミニュレーションパッケージを走らせるところで決定されるかもしれない。これらの場合には、各座標の微少な増減に対する反応を評価することによって、いわゆる経験的傾き (empirical gradient) を求めることができます。経験的傾き探索は状態空間の離散版での最急勾配山登り法と同じである。

経験的傾き

が存在する、基本的な問題は、もし α が小さすぎるなら、非常に多くのステップが必要になること、また、 α が大きすぎるなら、探索は最大値を超してしまいうかもしれないことである。ラインサーチ法 (line search) の技術は、 f が再び下がり始めるまで現在の勾配の方向を拡張することによって——通常は α を繰り返し倍増することによって——このジレンマを克服しようとする。これが生じる地点が新しい現在状態となる、この地点でどのように新しい方向が選ばれるかについては、いくつかの流派の考え方がある。

多くの問題では、最も効率的なアルゴリズムは尊重すべきニュートン-ラフソン法 (Newton-Raphson method) である。これは関数の解を求める、つまり $g(x) = 0$ 形式の方程式を解決する一般的な技法である。これは以下のニュートンの公式にしたがって解 x の新しい推定値を計算することで実行される。

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

f の最大値や最小値を求めるためには傾きがゼロ (すなわち $\nabla f(\mathbf{x}) = 0$) になるような \mathbf{x} を見つける必要がある。このようにニュートンの公式での $g(x)$ は $\nabla f(\mathbf{x})$ となり、更新の式は行列形式で以下のように書くことができる。

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

ここで、 $\mathbf{H}_f(\mathbf{x})$ は 2 次導関数のヘッセ (Hessian) 行列であり、その要素 H_{ij} は $\partial^2 f / \partial x_i \partial x_j$ で与えられるものである。ヘシアンは n^2 の要素をもつので、ニュートン-ラフソン法は高次元空間のときは高価となり、多くの近似解法が開発されている。

局所探索手法は離散的空間の場合と同様に、連続状態空間で局所的最大値や尾根や高原の悪影響を受ける、ランダム再スタート法や焼きなまし法を使うこともでき、それらは多くの場合役に立つ。しかし、高次元の連続空間はすぐに迷ってしまうような大きな場所なのである。

ちょっとしか知らない人でも役立つような最後の問題は制約つき最適化 (constrained optimization) である。最適化問題は、その解が変数のとる値に厳しい制約を満たさなくてはならないとき、制約つきである。たとえば、空港の配置問題では、空港の位置は Romania 内部で砂漠地帯 (湖の真ん中でなく) に制限される。制約つき最適化問題の難しさは副約や目的関数の性質による。その最も知られたカテゴリは線形計画法 (linear programming: LP) 問題である。この問題においては、制約は 凸 領域を形成する線形不等式ではなくてはならないし、目的関数も線形でなくてはならない。線形計画法問題は変数の数に応じて多项式時間で解くことができる。二次計画法や、二次円錐計画法など、異なるタイプの制約や目的関数をもつ問題も研究されている。

4.5 オンライン探索エージェントと未知環境

今までオフライン探索 (offline search) アルゴリズムを使うエージェントに説明を集中しておいた。エージェントは実世界に足を踏み入れる前に完全な解を計算し (図 3.1 参照)，その後で自分の知覚に頼らずにその解を実行する。それに対し、オンライン探索 (online search)¹⁴ は計算機科学の分野では、入力データが到着したら、完全なデータ集合が描うる

行為を行い、次に環境を観測し次の行為の計算を行う。オンライン探索は、動的な分野や準動的な分野——長い時間ぶらぶらしていたり計算したりする」と罰則がつく分野——では良いアイデアである。オンライン探索は確率的分野ではさらにより良いアイデアである。一般にオンライン探索は起こりうることをすべて考慮した指標的に大きな予測プランに行き当たらざるを考えないが、オンライン探索は実際に起きただけを考えればよい、たとえば、チエスをするエージェントは、ゲームの完全な流れを書き出す前に最初の手を長く考えるようにと良い助言を受ける。

オンライン探索は探査問題 (exploration problem) を実行するのに必要なアイデアである。探査問題ではエージェントは状態や行為はわかっていない、未知の状態でエージェントは次に何をするべきかを決める実験として行為をしなければならないので、計算と行為を交互にさはさまなければならない。

オンライン探索の単純な代表例には、ロボットがある。それは新しい建物の中に配置され、A から B へ移動するのに使えるような地図を作るために建物を探査しなくてはならない、迷路から抜け出す方法——古代の野心的な英雄が必要とした知識——はオンライン探索アルゴリズムの例もある。しかし、空間的な探査だけが探査の方法ではない。生まれたばかりの赤ん坊を考えてみよう。赤ん坊には多くの行為をする可能性があるが、その行為の結果については知らない、赤ん坊は自分が達成することができるわずかな可能状態のみ経験することになる。世界がどのようになっているかを赤ん坊が徐々に発見していくことが、ある部分ではオンライン探索プロセスである。

オンライン探査問題

オンライン探査問題は、純粋な計算プロセスによってではなく、エージェントの実行動作によって解決されるのである。エージェントは次の関数だけを知っていると仮定する：

- ACTIONS(s)：状態 s で許された行為のリストを計算する。
- ステップコスト関数 $c(s, a, s')$ ——エージェントが s' が結果だと知るまでは、この関数は利用できないことに注意せよ。
- GOAL-TEST(s)。

エージェントは以前に訪問した状態を常に認識できると仮定し、さらにもエージェントの行動が決定的であるとの仮定する。(この二つの仮定は 17 章で弱められる。) 最後に、エージェントは現在の状態からゴール状態への距離を見積もる評価的ヒューリスティックス関数 $h(s)$ にアクセスするかもしれない。たとえば、図 4.18 では、エージェントはゴール位置だけがわかつていない。

¹⁴ “オンライン”という用語は計算機科学の分野では、入力データが到着したら、完全なデータ集合が描うる伴つことなく実行しなければならないようなアルゴリズムをさすのによく使われる。

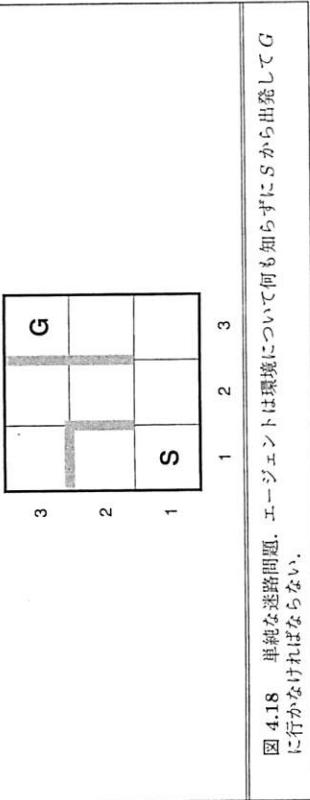


図 4.18 単純な迷路問題。エージェントは環境について何も知らずに S から出発して G に行かなければならぬ。

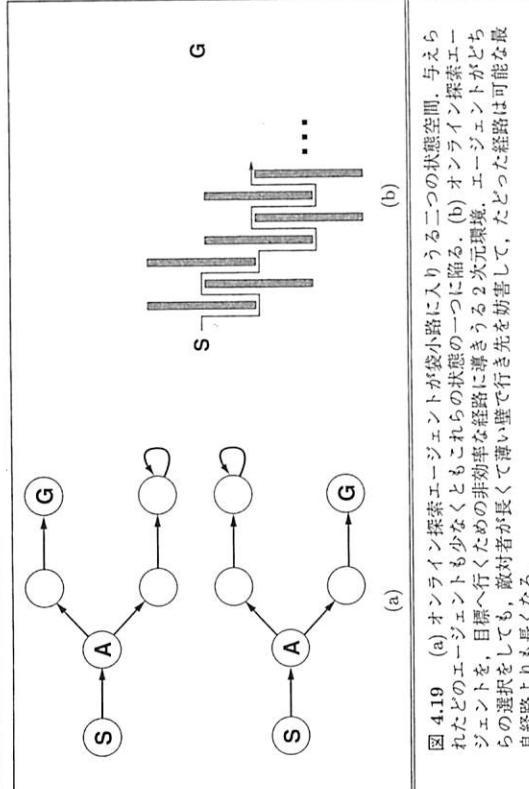


図 4.19 (a) オンライン探索エージェントが袋小路に入りうる二つの状態空間。与えられたどのエージェントも少なくともこれらの状態の一つに陥る。(b) オンライン探索エージェントを、目標へ行くための非効率的な経路に導きうる 2 次元環境。エージェントがどちらの選択をしても、敵対者が長くて薄い壁で行き先を妨害して、たどった経路は可能な最良経路よりも長くなる。

置を知つておれば、マンハッタン距離のヒューリスティックスを使うことができるかもしない。

典型的には、エージェントの目標は、コストを最小にしながらゴール状態に達することである。(別の目標が、單に環境全体を探査することであることもあリうる。) このコストはエージェントが実際にお訪れる経路のトータルな経路コストである。このコストを、もしエージェントが探索空間を事前に知つたら使つたであらう経路のコスト——すなわち実際の最短経路のコスト（または、最短の完全探索）——と比較することとはよく行われる。オンラインアルゴリズムの言語では、これは競争率 (competitive ratio) とよばれる。この値はできるだけ小さくしたい。

これは合理的な要求のようにも聞こえるが、ある場合には、達成できる一番良い競争率の値が無限大になることがある。たとえば、もしある動作が逆戻りできないものであるなら、オンライン探索によって、そこからゴール状態に行けない袋小路状態に偶然に陥つてしまふかもしれない。おそらく“偶然に”という言葉は説得力がないと思うだろう。



対立論証

安全に探し可能

う——ひょっとして、探査したときには袋小路の経路をとることがないようなアルゴリズムがあるかもしれないと思うだろう。より正確に言うと、我々の主張は、すべての状態空間において袋小路を避けうるアルゴリズムはないということである。図 4.19 (a) の二つの袋小路を考えてみよう。状態 S と A を訪問してきたオンライン探索アルゴリズムにおいては、二つの状態空間は同一に見えるので、両方に同じ決定をしなければならない。したがつて、どちらかの一つで失敗する。これは対立論証 (adversary argument) の一例である——我々はそのエージェントが状態空間を探査している間、状態空間を構築し、好きなどころにゴールや袋小路をおくことができる敵対者を想像することができます。

袋小路はロボット探査で実際に起る困難な問題である。階段や、スロープや、崖や、あらゆる種類の自然の地形によって、後戻りできない行為をする機会が生じてしまう。先に進むために、状態空間が安全に探査可能 (safely explorable) である——到達できるどの状態からもあるゴール状態に到達できる——と仮定する。迷路や 8 パズルのように、戻れる行為を伴う状態空間は無向グラフとみなせて、明らかに安全に探査可能である。

安全に探査可能である環境であってさえ、制限されないコストをもつ経路があるなら、競争比に制限があることを保証することはできない。後戻りできない行為をもつ環境ではこれを証明するのはやさしいが、図 4.19 (b) で示すように、後戻りできる場合でも、これは成り立つ。このため、オンライン探索アルゴリズムの効率を述べるときは、車に最も浅いゴール状態に関するではなく、全状態空間のサイズに関するのが普通である。

オンライン探索エージェント

オンラインエージェントは行為を行なうたびに、エージェントがどのような状態に達したかを示す知覚情報を受け取る。この情報から、エージェントは環境の地図を充実させることができ、新しい地図は次にどこへ行くかを決めるのに用いられる。このようなプランニングと行為を相互にさしはさむことは、オンライン探索が今までに見てきたオフライン探索とまったく異なることを意味している。たとえば、 A^* のようなオフラインアルゴリズムは、空間の一部分のノードを開拓し、直後に空間の別の部分のノードを開拓する、という能力をもつ。というのは、ノードの展開は実際の行為ではなく、模擬の行為だからである。一方、オンラインアルゴリズムは、エージェントが物理的に占めているノードを開拓することができるだけである。次のノードを開拓するのに木金盤のすべての進路をたどるのを避けるには、ノードを“局所的な順序”で展開するほうが良いように思われる。深さ優先探索は（後戻りのときを除けば）展開される前のノードの子供なので、まさしくこの性質をもつっている。

オンライン深さ優先探索エージェントを図 4.20 に示す。このエージェントは地図を表す result[a, s] の中に記憶する。この表は、状態 s で行為 a を実行した結果の状態を記録するものである。現在の状態から探査されていない行為があるときはいつもエージェントはその行為を実行しようとする。エージェントがある状態ですべての行為を実行してしまったら、問題が生じる。オンライン深さ優先探索の場合は、単に待ち行列からこの状態を削除するだけである。オンライン探索の場合は、物理的に後戻りをしなくてはならない。深さ優先探索の場合、これはエージェントが最も最近入った最新状態からもとの状態に戻ることを意味している。これは、各状態において、エージェントがまだ戻りしていない前状態を列挙している表を保持することで実現される。もしエージェントが後戻りできる状態

```

function ONLINE-DFS-AGENT( $s'$ ) returns an action
inputs:  $s'$ , a percept that identifies the current state
static:  $result$ , a table, indexed by action and state, initially empty
        $unexplored$ , a table that lists, for each visited state, the actions not yet tried
        $unbacktracked$ , a table that lists, for each visited state, the backtracks not
          yet tried
 $s, a$ , the previous state and action, initially null
if  $GOAL-TEST(s')$  then return  $stop$ 
if  $s'$  is a new state then  $unexplored[s'] \leftarrow ACTIONS(s')$ 
if  $s'$  is not null then do
   $result[a, s] \leftarrow s'$ 
  add  $s$  to the front of  $unbacktracked[s']$ 
  if  $unexplored[s']$  is empty then return  $stop$ 
  else  $a \leftarrow$  an action  $b$  such that  $result[b, s'] = \text{Pop}(unbacktracked[s'])$ 
   $s \leftarrow s'$ 
return  $a$ 

```

図 4.20 深さ優先探査を利用するオンライン探索エージェント。このエージェントは双方向の探索空間でのみ利用可能である。

を実行しなくなったら、その探索は完了する。

読者は、ONLINE-DFS-AGENT（オンライン深さ優先探索エージェント）が図 4.18 の迷路に適用されたときに、どのように実行されているかをたどってみることを勧めた。このエージェントが、最悪のときには、状態空間のすべてのリンクをちょうど 2 回ずつたどつて終了することを簡単にしておらず、それは最適である。一方、ゴール状態を見つけることをからすると、初期状態の直後にゴール状態があるような場合、このエージェントは、ゴールから離れて長い探索にはまると、その競争比はどんどん悪くなる。繰返し深化アルゴリズムのオンライン版はこの問題を解決する。均一の木の環境のもとでは、そのようなエージェントの競争比は小さな定数だからである。ONLINE-DFS-AGENT は後戻りの機能をもつので、行為が後戻り可能な状態空間でのみ作用する。一般的な状態空間で作用するようなもう少し複雑なアルゴリズムもあるが、そのようなアルゴリズムは競争比に制限がない。

オンライン局所探索

深さ優先探索と同じように、山登り探索 (hill-climbing search) はノードの展開を局所的に行う性質をもっている。実際のところ、山登り探索はメモリ中に現在の状態しか記録しないため、これは“すでに”オンライン探索アルゴリズムとなっている！この方法は他にどこにも進めないような局所的最大値にいるエージェントをそのままにするので、残念ながらその単純な形のままではあまり役に立たない。さらに、このエージェントはそれ自身では新しい状態に移動できないので、ランダムな再スタートも行えない。

ランダムな再スタートをする代わりに、環境の中をランダムウォーカー (random walk) しながら探査することも考えられる。ランダムウォーカーは單に現在の状態で実行可能な行為

の中から一つをランダムに選ぶだけである。まだ実行していない行為の間に優先性をつけられることもできる。空間が有限であれば、ランダムウォーカーはゴール状態を“ついには”見つけるか、または、探査をし続けるかのいずれかである。一方、このプロセスは非常に遅くなるかもしれない、図 4.21 はランダムウォーカーするとゴール状態を見つけるまでに指數関数的に多くのステップを要するような環境を表している。これは各ステップで後戻りの進行が前向きの進行の 2 倍起こりやすいからである。もちろん、これは人工的に作った例であるが、そのトボロジーによって、ランダムウォーカーするところの種の“罠”を生じるような実世界の状態空間はたくさんある。

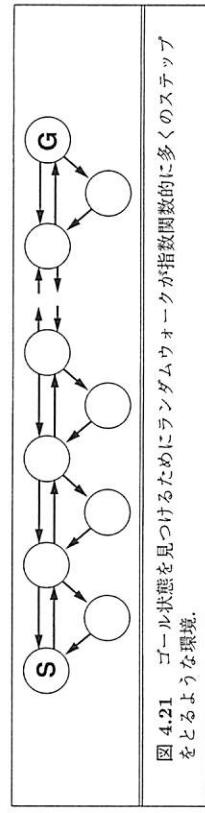


図 4.21 ゴール状態を見つけるためにランダムウォーカーが指數関数的に多くのステップをとるような環境。

山登り法を拡張するには、ランダムさの点ではなく、メモリの点で拡張するほうがより効率的であることを明瞭にする。その基本的なアイデアは、今まで訪れた各状態からゴール状態に達するコストの中で“現在の最良の見積り” $H(s)$ を記憶することである。 $H(s)$ は単にヒューリスティックスの見積り $h(s)$ から始まり、エージェントが状態空間の中で経験を積むにつれて更新される。図 4.22 は单纯な 1 次元の状態空間の例を示している。(a) では、エージェントは影をつけた状態のところでフロットな局所的最小値になつて行き詰まっているようである。エージェントはそこにとどまるのではなく、その隣の状態の最新のコスト見積りに基づいて、ゴール状態への最良経路と思われるほうに進むべきである。隣の状態 s' を通ってゴール状態に達するコストの見積りは、 s' へ行くコストとそこからゴールにいたるコストの和、すなわち $c(s, a, s') + H(s')$ 、である。この例では、見積りコストが $1 + 9 > 1 + 2$ となる二つの行為があるので、右に動いたほうが良いようである。ここで、影をつけた状態のコスト見積りが 2 というものは楽観的すぎたことが明らかになっている。最良の移動コストが 1 であり、それにによってゴール状態から少なくとも 2 ステップかかる状態に行くので、影をつけた状態はゴール状態から少なくとも 3 ステップとなる。そこで、図 4.22 (b) に示すように、 H が更新され、このプロセスを続けると、エージェントはさらには回りたりして右に“抜け出す”まで H を毎回更新して局所的最小値をひきのばす。

この図式を実装するエージェントは、学習 RTA*(LRTA*) とよばれるものであり、図 4.23 に示している。ONLINE-DFS-AGENT の場合と同じように、このエージェントは result 表を使って環境の地図を作る。このエージェントは、離れたばかりの状態のコスト見積りを更新し、現在のコスト見積りにしたがって、“最もらしい”行為を選択する。重要なことは、状態 s で試されていない行為はいつも可能な最小コスト、つまり $h(s)$ 、でゴール状態に直接導くものと仮定していることである。この不確実なもとの樂観主義 (optimism of the樂觀主義) 不確実なもとで

¹⁵ 無限の場合はもととトリッキーである。ランダムウォーカーは無限の 1 次元格子や 2 次元格子のときは完全でないが、3 次元格子のときは完全ではない！後者の場合、ランダムウォーカーが出発点に戻る確率は約 0.3405 にすぎない（一般的な紹介としては Hughes (1995) を見よ）。

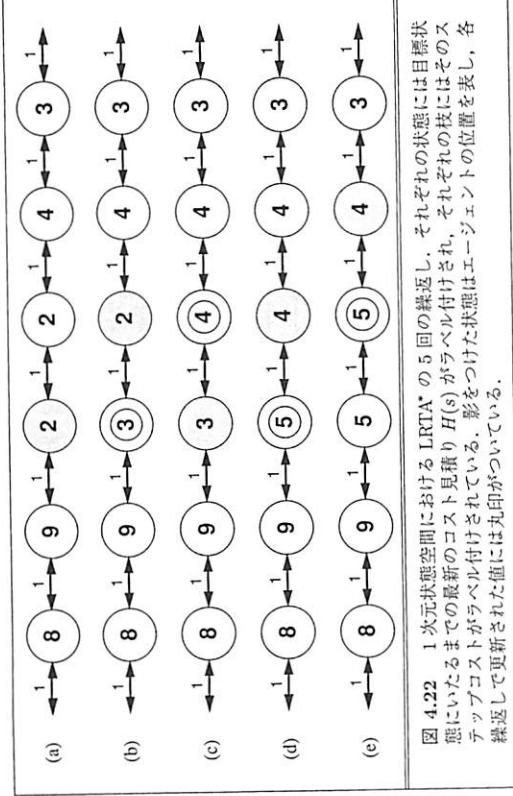


図 4.22 1 次元状態空間における LRTA* の 5 回の繰返し。それぞれの状態には目標状態にいたるまでの最新のコスト見積り $H(s)$ がラベル付けされ、それぞれの枝にはそのステップコストがラベル付けされている。影をつけた状態はエージェントの位置を表し、各繰返しで更新された値には丸印がついている。

```
function LRTA*-AGENT( $s'$ ) returns an action
inputs:  $s'$ , a percept that identifies the current state
static:  $result$ , a table, indexed by action and state, initially empty
 $H$ , a table of cost estimates indexed by state, initially empty
 $s$ ,  $a$ , the previous state and action, initially null
if GOAL-TEST( $s'$ ) then return stop
if  $s'$  is a new state (not in  $H$ ) then  $H[s'] \leftarrow h(s')$ 
unless  $s$  is null
   $result[a, s] \leftarrow s'$ 
   $H[s] \leftarrow \min_{b \in Actions(s)} LRTA^*-Cost(s, b, result[b, s], H)$ 
 $a \leftarrow$  an action  $b$  in  $Actions(s')$  that minimizes  $LRTA^*-Cost(s', b, result[b, s'], H)$ 
 $s \leftarrow s'$ 
return  $a$ 
```

```
function LRTA*-COST( $s, a, s', H$ ) returns a cost estimate
if  $s'$  is undefined then return  $h(s)$ 
else return  $c(s, a, s') + H[s']$ 
```

図 4.23 LRTA*-AGENT が隣接の状態の値によって行動を選択する。隣接状態の値はエージェントが状態空間を動くと更新される。

under certainty) によって、エージェントは新しくて有望そうな経路を探査するのである。LRTA* エージェントは有限で安全に探査可能などんな空間でもゴール状態を見つけることが保証されている。しかし A* と異なり、無限の状態空間ではこのエージェントは完全ではない、無限に迷ってしまう可能性がある場合がある。最悪の場合は、 n 状態の環境を探査するのに $O(n^2)$ ステップかかりうるが、多くの場合はそれよりはるかに良い。LRTA* エージェントは、行動選択ルールと更新ルールを異なる方法で特定して定義されるオンラインエージェントの大きなファミリーの一つにすぎない、このファミリーはもともとは確率的

環境のために開発されたものであるが、これは 21 章で説明する。

オンライン探索エージェントでの学習

オンライン探索エージェントは初めは何の知識もないが、学習の様々な機会をもつ。第一に、エージェントは、自分の経験を記録することによって、環境の“地図”——より正確には各状態での各行為の出力——を学習する（環境が決定性をもつと仮定することは、各行について 1 回の経験で十分であることを意味していることに注意せよ）。第二に、局所探索エージェントは、LRTA* のような局所更新ルールを使って、各状態の値のより正確な見積りを得る。21 章ではこれらの更新により、エージェントが正しい方法で状態空間を探査するなら、各状態で正しい値に収束することを示す。一度、正確な値がわかつると、最大値をもつ後続状態に移動する——すなわち純粋な山登りが最適戦略となる——ことによつて、最適な決定がなされる。

もし読者が図 4.18 の環境で ONLINE-DFS-AGENT の行動をたどるべきだという助言にしたがつたのなら、このエージェントがそんなに賢くないことに気がついていることだろう。たとえば、このエージェントが、 Up 行為によって (1,1) から (1,2) へ進むことをみたあとでも、 $Down$ 行為が (1,1) に戻すこのエージェントは、 Up 行為によって (2,1) から (2,2) へ、(2,2) から (2,3) へなどのように進むこともわからず、一般的に、エージェントには、 Up 行為が壁の位置にいるのでなければ、 y 座標を増加させることや、 $Down$ が y 座標を減少させることなどを学習してほしいと思うだろう。これを実現するには二つのことが必要である。第一に、これらの種類の一般ルールの形式のかつ明確に操作可能な表現法が必要である。今まで、我々は後者関数と呼ばれるブラックボックスの中にこの情報を隠していた。第 3 部ではこの問題を説明する。第二に、このエージェントが行う特定の観測結果から適切な一般ルールを導くアルゴリズムを必要とする。これらは 18 章述べる。

4.6 まとめ

この章では、探索コストを下げるヒューリスティクスの応用について調べた。ヒューリスティクスを使うたくさんのアルゴリズムを見てきたが、良いヒューリスティクスを使つてさえ、最適性は探索コストの観点から法外な値になることを見いだした。

- 最良優先探索は（ある計測値にしたがって）最小コストをもつような未展開のノードを展開するために選択する GRAPH-SEARCH である。最良探索アルゴリズムは典型的にはノード n からの解のコストを見積もるヒューリスティクス関数 $h(n)$ を使う。欲張り最良優先探索は $h(n)$ が極小となるノードを展開する。これは最適ではないが、多くの場合は効率的である。
- A* 探索は $f(n) = g(n) + h(n)$ が最も小さなノードを展開する。もし $h(n)$ が許容的 (TREE-SEARCH の場合) か、無矛盾 (GRAPH-SEARCH の場合) であることを保証するならば、A* は完全で最適である。A* の空間の複雑さはまだ非常に高い。

- ヒューリスティック探索アルゴリズムの効率はヒューリスティック関数の質に依存する。良いヒューリスティックスは、問題の定義を弱めたり、バターンデータベース中の部分問題の解決コストをあらかじめ計算したり、その問題のクラスで経験を学習したりすることによって作られることがある。
- RBFS や SMA* は、制限された量のメモリを使うロバストで最適な探索アルゴリズムである。十分な時間があれば、これらのアルゴリズムは、メモリがあふれたために A*では解くことができない問題を解くことができる。
- 山登り法のような局所探索法は、メモリ中に少數のノードのみを記録することによって、完全な状態の定式化の上ではたらく。焼きなまし法のようにいくつかの確率的アルゴリズムが開発された。これらは適切な冷却スケジュールが与えられると最適解を返す。連續空間の問題を解くために、多くの局所探索手法も使うことができる。
- 遺伝的アルゴリズムは、状態の多くの個体集合が保持されている確率的山登り法である。突然変異や個体集合からの状態の対を結合する交叉によって新しい状態が作られる。
- 探査問題はエージェントがその環境における状態や行為について何も知らないときに発生する。安全に探査できる環境では、オンライン探索エージェントは地図を作り、ゴール状態があるならそれを発見することができる。経験からヒューリスティックの見積りを更新することによって、局所的最大値から遅れる効率的な方法を得ることができる。

問題解決にヒューリスティック情報を使うことは Simon and Newell (1958) による初期の論文に記述されている。しかし、“ヒューリスティック探索”という言葉やゴール状態への距離の見積りを行うヒューリスティック関数の利用についてはもう少しあとになる (Newell and Ernst, 1965; Lin, 1965). Doran and Michie (1966) は、特に 8 パズルや 15 パズルのような、多くの問題への応用としてヒューリスティック探索の実験的な研究を集中的に行つた。Doran and Michie はヒューリスティック探索における、経路長や“浸透度”(経路長と、今までに探査されたノードの総数との比) の理論的解析を行つたが、現在の経路長から得られる情報を無視したうに思われる。A*アルゴリズムは、現在の経路長をヒューリスティック関数に統合したものであるが、Hart, Nilsson, and Raphael (1968) によって開発され、のちに少し修正された (Hart *et al.*, 1972). Dechter and Pearl (1985) は A*の最適な効率性を実証した。

文献と歴史ノート

オリジナルな A* の論文はヒューリスティック関数についての無矛盾性の条件を導入した。より単純な代用手法として、単調条件が Pohl (1977) によって導入されたが、Pearl (1984) はこれら二つは同等であることを示した。A* より前に提案された多くのアルゴリズムはオープンリストやクローズドリストと同等のものを用いた。それらのアルゴリズムの中に、オーブンリストや深さ優先探索や単一コスト探索 (Bellman, 1957; Dijkstra, 1959) が含まれている。特に Bellman の研究では最適アルゴリズムを単純化するために、経路コストを付加することが重要であることを示した。

A*には多くの変形版がある。Pohl (1973) は “動的重み付け (dynamic weighting)” を使うことを提案した。これは、A*で用いられた $f(n) = g(n) + h(n)$ という単純な合計をする代わりに、評価関数として最新の経路長とヒューリスティック関数の重み付けをした $f_w(n) = w_g g(n) + w_h h(n)$ を用いる。重み w_g と w_h は探索が進むにつれて動的に調節される。Pohl のアルゴリズムは ϵ 許容——すなわち、最適解の $1 + \epsilon$ 倍以内の解を見つけることが保証されている——として表現することができる。ここで、 ϵ はこのアルゴリズムに与えられるパラメータである。同じ性質は A_e^* アルゴリズム (Pearl, 1984) でも示されており、このアルゴリズムは、最小 f コストのノードの $1 + \epsilon$ 倍以内の f コストであれば、どんなノードも繋がら選択することができる。この選択は探索コストを最小化するのに行なうこともできる。

A*や他の状態空間探索アルゴリズムはオペレーションズリサーチで広く使われている “分歧限定法” に密接な関連がある (Lawler, Lawler, and Wood, 1966)。状態空間探索と分岐限定法の関係は深く研究されている (Kumar and Kanal, 1983; Nau *et al.*, 1984; Kumar *et al.*, 1988), Martelli and Montanari (1978) は動的計画法 (17 章参照) と、あるタイプの状態空間探索との間のつながりを示した。Kumar and Kanal (1988) は CDP (合成決定プロセス, composite decision process) の名のもとで、ヒューリスティック探索と動的計画法と分岐限定法の “大統合” を図っている。

1950 年代後期と 1960 年代初期では、コンピュータは高々数千ワードのメインメモリしかもつていなかったので、メモリ制限のヒューリスティック探索は初期の研究課題だった。最も初期の探索プログラムの一つであるグラフ探索器 Graph Traverser (Doran and Michie, 1966) はメモリの制限のため最も優先探索に合わせたオペレータを明らかにしている。IDA* (Korf, 1985a, 1985b) は最初に広く使われた最適な、メモリ制限のあるヒューリスティック探索アルゴリズムであり、非常に多くの変形版が開発された。IDA* の効率の解析や実数値のヒューリスティックスをもつときのその難しさの解析は Patric *et al.* (1992) でなされている。

RBFS (Korf, 1991, 1993) は実際にには図 4.5 で示されたアルゴリズムよりも複雑である。図 4.5 のアルゴリズムは逐次展開 (iterative expansion) または IE (Russell, 1992) と呼ばれ、独立に開発されたアルゴリズムに近い。RBFS は上限と同じように下限も用いる。二つのアルゴリズムは許容可能なヒューリスティックスでは同じように振る舞うが、RBFS は非階級的なヒューリスティックスでも最良優先の順でノードを開拓する。一番良い別の経路を保持するアイデアは、Bratko (1986) の Prolog による A* のエレガントな実装や、DTA* アルゴリズム (Russell and Wefald, 1991) などの早期に現れている。後者のほうはメタレベルの状態空間やメタレベルの学習についても論じている。MA* アルゴリズムは Chakrabarti *et al.* (1989) に現れた。SAM*, または単純 MA* は、IE (Russell, 1992) の比較アルゴリズムとして、MA* を実装する中で考案された。Kaindl and Khorsand (1994) は実質的に前のアルゴリズムより高速な双向探索アルゴ

リズムを作り出すためにSMA*を応用した。Korf and Zhang (2000)は分割統治アルゴリズムを説明し、Zhou and Hansen (2002)はメモリ制限付きA*グラフ探索を紹介している。Korf (1995)はメモリ制限探索法のサーベイを行っている。

弱条件問題によって許容的ヒューリスティックスを導けるというアイデアは Held and Karp (1970)によるセミナー論文中に現れている。彼らはTSPを解くのに最小スパンニングラフ木を用いた(練習問題4.8参照)。

弱条件問題化プロセスの自動化は、Mostow (Mostow and Priedtis, 1989)の初期の仕事の上になされ、Priedtis (1993)によつて成功裏に実装された。許容的ヒューリスティックスを導くのに排反バターンデータベースを使用することは Gasser (1995)と Culberson and Schaeffer (1998)によつてなされた。排反バターンデータベースは Korf and Felner (2004)に記述された。ヒューリスティックスの確率的な解釈は Pearl (1984)と Hansson and Mayer (1989)によつて深く研究された。

ヒューリスティックスとヒューリスティック探索アルゴリズムに関する最ももわかりやすいのは Pearl (1984)の *Heuristics*といふ教科書である。この本は、A*の幅広い分派や変形をカバーし、それらの形式的な性質について厳格な証明をしている。Kanal and Kumar (1988)はヒューリスティック探索の重要な事柄の作品集となつている。探索アルゴリズムの新しい研究結果はジャーナル *Artificial Intelligence*に定期的に掲載されている。

局所探索の技法は数学や計算機科学の分野で長い歴史をもつていて、実際、ニュートン-ラフソン法 (Newton, 1671; Raphson, 1690) は、勾配情報が使えるような証明をしていている。Kanal and Kumar (1988)はヒューリスティック探索の重要な事柄の作品集となつている。Brent (1973)はそのような情報をわめて効率的な局所探索方式ととらえることができる。Berm探索は局所探索アルゴリズムとして紹介したが、HARPYシステム (Lowther, 1976)の音声認識のための動的計画法の横幅が制限された変形版として始まつたものである。関連のアルゴリズムは Pearl (1984, 5章)によつて深く解析されている。

局所探索の課題は、n クイーン (Minton *et al.*, 1992) や論理的推論 (Selman *et al.*, 1992)による巨大な制約充足問題の驚くほど良い成果や、ランダムさと並列探索とその他の改良を統合することによって、近年、再び盛んになってきた。Christos Papadimitriou が“新時代”的アルゴリズムとよんだルネサンスは、理論的な計算機科学者 (Koutsoupias and Papadimitriou, 1992; Aldous and Vazirani, 1994)にも関心を引き起こした。オペレーシヨンズリサーチの分野では、ターブー探索 (tabu search) とよばれる、山登り法の変形版が人気を得ている (Glover, 1989; Glover and Laguna, 1997)。人間の限られた短期記憶のモデルにしたがい、このアルゴリズムは再び訪れることができない k 回の訪問済み状態のターブーリストを保持する。グラフの探索が効率をよくしたように、このアルゴリズムはターブーリストを使って局所的小値を免れることができる。山登り法の別の有用な改良は STAGE アルゴリズム (Boyan and Moore, 1998) である。このアイデアは、俯瞰の全体像をつかむため、ランダム再スタート山登り法で見つかった局所的最大値を使うことである。このアルゴリズムは滑らかな表面を局所的最大値の集合に当たはめ、そのあと、その表面の大局部最大値を解析的に計算する。これは新しい再出発点となる。このアルゴリズムは実際の困難な問題で働くことが示された。Gomes *et al.* (1998) はシステムティックな後戻りアルゴリズムの実行時間の分布が幅広い分布 (heavy-tailed distribution) となることが多いことを示した。この分布は、もし実行時間が通常の分布ならば、非常に長時間の実行時間となる確率は予想したより大きくなることを意味している。これにより、ランダムな再ス

タートに理論的な正当化を与える。

焼きなまし法は、最初に Kirkpatrick *et al.* (1983) によって記述された。彼らはメトロポリスアルゴリズム (Metropolis algorithm) (これは物理学で複雑系のシミュレーションをするのに用いられたものであり (Metropolis *et al.*, 1953))、おそらく Los Alamos のデーターパーティで発明されたものである) から直接、アイデアを借りている。焼きなまし法は今やそれが一つの研究分野であり、毎年、数百の論文が出版されている。

連続空間で最適解を見つけることは、最適化理論 (optimization theory) や最適制御理論 (optimal control theory) や変動計算 (calculus of variations) を含むいろいろな分野の重要な課題である。適切な (そして実際的な) 入門書が Press *et al.* (2002) や Bishop (1995) により与えられている。線形計画法 (linear programming: LP) はコンピュータの最初の応用の一つである。シンプレスアルゴリズム (simplex algorithm) (Wood and Dantzig, 1949; Dantzig, 1949) は、最悪の場合には複雑さが指數関数的になるが、いまだに使われている。Karmarkar (1984) は LP のための実際的な多項式時間アルゴリズムを開発した。 Sewall Wright (1931) による、適合度輪廓図 (fitness landscape) の概念に関する研究は遺伝的アルゴリズムの開発の重要な先駆けであった。1950年代に Box (1957) や Friedman (1959) を含む何人かの統計学者は最適化のための進化的技術を用いた。しかし、Rechenberg (1965, 1973) が翼の最適化問題を解くのに進化的戦略 (evolution strategy) を導入するまでの手法は人気を得ることはなかった。1960年代と 1970年代に、John Holland (1975) は、有用なツールとして、また、適合や生命などの理解を拡張する方式として、遺伝的アルゴリズムを擁護した (Holland, 1995)。人工生命 (artificial life) の研究の動き (Langton, 1995) は、遺伝的アルゴリズムの出力を問題の解ではなく生物としてみると、このアイデアを 1 ステップ先に進めた。この分野では Hinton and Nowlan (1987) や Ackley and Littman (1991) により、Baldwin 効果が示唆するものを明らかにするための多くの研究が行われた。進化についての一般的な背景については Smith and Szathmary (1999) を強く勧める。

遺伝的アルゴリズムと他のアプローチ (特に確率的山登り法)との多くの比較により、遺伝的アルゴリズムは吸収東が遅いということがわかつた (O'Reilly and Oppacher, 1994; Mitchell *et al.*, 1996; Juels and Wattenberg, 1996; Baluja, 1997)。そのような認識は GA のミニマニエニティでは普遍的に信じられているわけではない。しかし、個体群に基づく探索をペイズ学習 (20章参照) の近似として理解しようとする GA コミュニティでは、この分野とその批判 (Pelikan *et al.*, 1999) の間のギャップを埋めるのに役立つかもしれない。二次動的システム (quadratic dynamical system) の理論も GA の効率を説明できるかもしれない (Rahani *et al.*, 1998)。アンテナ設計に GA を利用した例として Lohn *et al.* (2001) を、巡回セールスマン問題に適用した例として Larrañaga *et al.* (1999) を見よ。

遺伝的プログラミング (genetic programming) の分野は遺伝的アルゴリズムと密接な関係にある。主な相違点は、突然変異や結合がなされる表現がビット文字列ではなく、プログラムであるという点である。このプログラムは表現木の形で表される。その表現は Lisp のような標準言語の形式でもよいし、回路やロボット制御器などを表現するための特別に設計されたものでもよい。交叉は部分文字列ではなく、部分木の切り貼りを含んでいる。この形式で交叉することによって、子孫が整形式の表現であることが保証される。これは、もしプログラムが文字列として扱われたら立たないことである。

遺伝的プログラミングの最近の関心事は John Koza (Koza, 1992, 1994) の研究に刺激されたものであるが、これは少なくとも Friedberg (1958) の機械語や Fogel *et al.* (1966) による有限状態オートマトンの初期の実験にさかのばるものである。遺伝的アルゴリズムと同じように、技法の効率に関しては議論がある。Koza *et al.* (1999) は遺伝的プログラミングを使って、回路のデバイスの自動設計のいろいろな実験を記述している。

オイラーグラフ オイラーグラフ (Eulerian graph) (すなわち、各ノードで入ってくる枝と出でていく枝が同数であるグラフ) のより一般的な探索が Hierholzer (1873) によるアルゴリズムで解かれた。任意のグラフの完全なアルゴリズムの最初の研究は Deng and Papadimitriou (1990) によってなされた。彼らは完全な一般的アルゴリズムを開発したが、一般グラフを探索するのに競争率を制限することが不可能であることを示した。Papadimitriou and Yannakakis (1991) は幾何学的な経路計画の環境 (ここではすべての行為が後戻り可能) の、ゴールへの経路発見の問題を研究した。彼らは、正方形の障害物では競争比を小さくすることが可能であるが、一般的な長方形の障害物では競争率に制限を設けることはできないことを示した (図 4.19 参照)。

LRTA* アルゴリズムは、エージェントが探索の後で一定時間内に行方を行わなければならぬ環境 (2人プレイゲームでは非常に一般的な状況である) での実時間探索 (real-time search) の研究の一部として、Korf (1990) によって開発された。実際、LRTA* は確率的環境での強化学習アルゴリズムの特殊ケースなのである (Barto *et al.*, 1995)。不確実状態での最適化のポリシー——常に最も近い未訪問状態に向かう——は、知識がないときは、単純な深さ優先探索よりも効率の悪い探索パターンになるかもしない (Koenig, 2000)。

Dasgupta *et al.* (1994) はオンライン繰返し深化探索はヒューリスティック情報がないときに、均一の木のゴール発見を最適な効率性で行うことを示している。LRTA* のテーマの、知識を利用する様々な変形版で、グラフの既知の部分の中で探索と更新を行う方式をとるものが開発された (Pemberton and Korf, 1992)。しかし、ヒューリスティック情報を使つたとき、最適な効率でゴールをどのよう発見するかについてはまだ良い考えがない。

この章では並列探索 (parallel search) アルゴリズムの課題はカバーしていない。その原因の一つは、並列コンピュータのアーキテクチャについて長い議論が必要となるからでもある。並列探索は AI と理論的な計算幾何学の双方ににとって重要な課題になつてきている。AI の文献についての簡単な紹介を Mahanti and Daniels (1993) に見ることができる。

練習問題

4.1 直線距離のヒューリスティックスを使って、Lugoj から Bucharest へ行く問題へ A* 探索を適用したときの操作をトレースせよ。すなわち、このアルゴリズムが考慮するノードの列と、各ノードの f, g, h のスコアを示せ。

4.2 ヒューリスティック経路アルゴリズム (heuristic path algorithm) は目的関数が $f(n) = (2-w)g(n) + wh(n)$ である最良優先探索である。このアルゴリズムは w がどのようになつたときに最適になることが保証されるか? (h が許容的であることを仮定してよい。) このアルゴリズムは $w=0$ のとき、どのような探索が行われるか? $w=1$ のときはどうか? $w=2$ のときはどうか?

4.3 次のそれぞれの言明を証明せよ:

- a. 横幅優先探索は均一コストの探索の特殊なケースである。
- b. 橫幅優先探索も深さ優先探索も均一コスト探索も最良優先探索の特殊なケースである。
- c. 均一コスト探索は A* 探索の特殊なケースである。

4.4 GRAPH-SEARCH を用いて、評価的だが一貫性のない $h(n)$ 関数を使い、A* が準最適解を計算するような状態空間を考え出せ。

4.5 96 ページにおいて、Iasi から Fagaras へ行く問題では直線距離のヒューリスティックスを使うと、欲張り最良優先探索が迷うことを見た。しかし、Fagaras から Iasi へ行くという反対の問題では、このヒューリスティックスは完全である。このヒューリスティックスがどちら方向でも誤るような問題はあるか?

4.6 8 バスルでときどき過見積りするようなヒューリスティックスを考え、特定の問題でのヒューリスティックスがどのように準最適解に導くかを示せ (使いたければコンピュータの助けを借りてもよい)。もし、 h が c よりも大きくなるような過見積りをしないないう反対の問題では、このヒューリスティックスは完全である。このヒューリスティックスがどちら方向でも誤るような問題はあるか?

4.7 もしヒューリスティックスが無矛盾なら、それは許容的でなければならないことを証明せよ。無矛盾のある許容的なヒューリスティックスを構成せよ。

4.8 巡回セールスマン問題 (TSP) は最小スパンニング木 (minimum spanning tree: MST) のヒューリスティックスを使って解くことができる。このヒューリスティックスは、すでに作られた部分的な訪問ルートが与えられたときに、それを完成させるのに要するコストを見積もるために用いられる。都市集合の MST コストは、すべての都市を結する木のリンクコストの和の最小のものである。

- a. TSP を弱問題化した版からこのヒューリスティックスがどのように導かれるかを示せ。
- b. MST ヒューリスティックスは直線距離より優れていることを示せ。
- c. 都市が単位正方形内のランダムな点として表わされるような TSP の例題を生成するプログラムを書け。

- d. MST を構成するため、文献から効率的なアルゴリズムを見つけ出し、それを許容的な探索アルゴリズムとともに用いて TSP の例題を解け。

4.9 108 ベージで、もし正方形 B が空白なら正方形 A から B へタイルを移動してもよい、という 8 パズルの弱条件問題を定義した。この正確な解答は Gaschnig のヒューリスティック (Gaschnig's heuristic) を定義する。なぜ Gaschnig のヒューリスティックが少なくとも h_1 (間違えたタイル数) と同程度の正確さをもつのかを説明し、このヒューリスティックが h_1 と h_2 (マンハッタン距離) の両方より正確であるようなケースを示せ。Gaschnig のヒューリスティックを効率よく計算する方法を示唆することができるか？

4.10 8 パズルの二つのヒューリスティックスはこれを改良すると主張している。たとえば、Nilsson (1971), Mostow and Priedtis (1989), Hansson *et al.* (1992) を参照せよ。これらのヒューリスティックスを実装し、その結果のアルゴリズムを比較して、これららの主張を確認せよ。

4.11 以下の特殊ケースの結果として生じるアルゴリズムの名前を示せ：

- $k = 1$ となる局所的ビーム探索。
- 一つの初期状態と、保持される状態数に制限がない場合の局所的ビーム探索。
- 常に $T = 0$ となっているとき（さらに終了テストを省略しているとき）の焼きなまし法。
- 個体数が $N = 1$ のときの遺伝的アルゴリズム。

4.12 問題に対する良い評価関数がないが、良い比較方法があることがある。たとえば、数値を割り当てることなしに一方のノードが他方のノードより良いことを決める方法があることがある。最も優先探索のためにはこれで十分なことを示せ。A*にも類似のものはあるか？

4.13 LRTA* の計算時間の複雑さと空間の複雑さとの関連を述べよ。

4.14 エージェントが図 4.18 に示したような 3×3 の迷路の環境にいるものとする。このエージェントは自分の初期位置が (1,1) であることや、ゴール位置が (3,3) であることや、壁で妨害されない限り四つの行為 *Up*, *Down*, *Left*, *Right* が通常の効果をもつことを知つてある。このエージェントはどこに内なる壁があるかは知らない、どんな与えられた状態においても、エージェントは実行できる行為の集合を理解している。エージェントはその状態が、過去に訪問したことのある状態か新しい状態かを識別することもできる。

a. 初期の信念状態にすべての可能な環境構成が含まれているような信念状態空間でのオンライン探索として、このオンライン探索をどのようにみることができるかを説明せよ。この初期の信念状態はどのくらい大きいのか？ 信念状態空間はどのくらい大きいのか？

- 初期状態でいくつの知覚が可能か？
- この問題のたまたま選んだプランの最初のいくつかの分歧について述べよ。完全なプランは（大雑把に）どのくらい大きいのか？

このたまたま選んだプランは、与えられた記述に適合するすべての可能な環境の解であることに注意しなければならない。したがって、未知の環境でさえ、探索と実行の交互実行は厳密には必要ない。



4.15 ここでは、練習問題 4.8 で定義されたタイプの TSP を解くために局所探索方式の利用法を調べる。

- TSP を解くのに山登り探索のアプローチを考察せよ。この結果を MST ヒューリスティック（練習問題 4.8）による A* アルゴリズムにより得られた最適解と比較せよ。
- 巡回セールスマン問題に対する遺伝的アルゴリズムのアプローチを考えよ。その結果を他のアプローチのものと比較せよ。表現に関する助言を求めて Larrañaga *et al.* (1999) を参照してもよい。



4.16 8 パズルや 8 クイーンの事例を多数作りだし、それらを山登り法（最急上昇で最初の変種を選択する）、ランダム再スタートの山登り法や焼きなまし法で解け（解けるものであれば）、探索コストと問題が解ける確率を計測し、最適の解コストに対する結果についてコメントせよ。



4.17 ここでは、例として図 3.22 の環境を用いて、ロボットのナビゲーションでの山登り法を調べる。

- 山登り法を用いて練習問題 3.16 を繰り返せ。あなたのエージェントは局所的最小値で行き詰まるだろうか？ 凸形状の妨害物の場合行き詰まることがありうるだろうか？
- エージェントが行き詰まるような非凸形状の環境を作れ。
- 次どこへ行くかを決めるのに深さ 1 の探索をする代わりに、深さ k の探索をするよう山登り法を変更せよ。このアルゴリズムは最良の k ステップの経路を見つけ、それにしたがって 1 ステップ行い、このプロセスを繰り返さなくてはならない。
- 新しいアルゴリズムが局所的最小値から逃れることを保証するようなものが存在するか？
- このケースでエージェントが LRTA* によってどのように局所的最小値から抜け出せるかを説明せよ。



4.18 8 パズル（マンハッタン距離を利用）と TSP (MST を利用——練習問題 4.8 をみよ）のランダムに作られた問題集合で A* と RBFS を比較せよ。自分の結果について議論せよ。8 パズルでヒューリスティック値に小さな乱数を加えたら RBFS の効率に何が起こるか？

