

ヒューリスティック探索入門

陣内 佑

March 25, 2020

目次

1	イントロダクション	7
1.1	何故人工知能に探索が必要なのか	8
1.2	本書の射程	10
1.2.1	関連書籍	11
2	状態空間問題 (State-Space Problem)	13
2.1	状態空間問題 (State-Space Problem)	13
2.2	状態空間問題の例	16
2.2.1	グリッド経路探索 (Grid Path-Finding)	16
2.2.2	スライディングタイル (Sliding-tile Puzzle)	18
2.2.3	多重整列問題 (Multiple Sequence Alignment)	19
2.2.4	倉庫番 (Sokoban)	21
2.2.5	巡回セールスパーソン問題 (Traveling Salesperson Problem, TSP)	22
2.3	問題の性質・難しさ	23

2.4	関連文献	25
3	情報なし探索 (Blind Search)	27
3.1	木探索アルゴリズム (Tree Search Algorithm)	28
3.2	グラフ探索アルゴリズム (Graph Search Algorithm)	31
3.3	幅優先探索 (Breadth-First Search)	34
3.4	深さ優先探索 (Depth-First Search)	36
3.4.1	再帰による深さ優先探索	36
3.5	ダイクストラ法 (Dijkstra Algorithm)	37
3.6	関連文献	38
4	ヒューリスティック探索 (Heuristic Search)	41
4.1	ヒューリスティックとは?	41
4.2	ヒューリスティック関数 (Heuristic Function)	42
4.3	A*探索 (A* Search)	46
4.4	ヒューリスティック関数の例	50
4.4.1	グリッド経路探索: マンハッタン距離	50
4.4.2	スライディングタイル: マンハッタン距離	50
4.4.3	巡回セールスパーソン問題: 最小全域木	51
4.5	非最適解の探索	51
4.5.1	重み付き A*探索 (Weighted A*)	52
4.5.2	貪欲最良優先探索 (Greedy Best-First Search)	53
4.5.3	山登り法 (Hill Climbing)	54
4.5.4	強制山登り法 (Enforced Hill Climbing)	55
4.6	上手く行かない場合	56
4.7	関連文献	57
5	グラフ探索のためのデータ構造	59
5.1	オープンリスト (Open List)	60
5.1.1	データ構造の選択	60

5.1.2	タイブレーキング (Tiebreaking)	61
5.2	クローズドリスト (Closed List)	62
5.2.1	ハッシュテーブル (Hash Table)	63
5.2.2	ハッシュ関数 (Hash Function)	64
5.2.3	トランスポジションテーブル (Transposition Table)	65
5.2.4	遅延重複検出 (Delayed Duplicate Detection)	66
5.2.5	関連文献	67
6	時間・空間制限下のヒューリスティック探索	69
6.1	分枝限定法 (Branch-and-Bound)	70
6.2	反復深化深さ優先探索 (Depth First Iterative Deepening)	72
6.2.1	反復深化 A* (Iterative Deepening A*)	75
6.3	両方向探索 (Bidirectional Search)	77
6.4	外部メモリ探索 (External Search)	78
6.4.1	外部メモリ 幅優先探索 (External BrFS)	80
6.5	シンボリック探索 (Symbolic Search)	82
6.5.1	特徴表現 (Symbolic Representation)	83
6.5.2	二分決定グラフ (Binary Decision Diagram)	84
6.5.3	特徴関数による状態空間の探索	85
6.6	新奇性に基づく枝刈り (Novelty-based Pruning)	87
6.6.1	状態の新奇性 (Novelty)	88
6.6.2	幅制限探索 (Width-Based Search)	89
6.6.3	反復幅制限探索 (Iterative Width Search)	90
6.7	並列探索 (Parallel Search)	90
6.7.1	並列化オーバーヘッド (Parallel Overheads)	91
6.7.2	並列 A* (Parallel A*)	93
6.7.3	ハッシュ分配 A* (Hash Distributed A*)	94
7	自動行動計画問題 (Automated Planning Problem)	99
7.1	定義	100

7.2	プランニングドメイン記述言語: Planning Domain Definition Language	101
7.3	古典的プランニング問題の例	103
7.4	ヒューリスティック関数の自動生成	106
7.4.1	ゴールカウントヒューリスティック (Goal Count Heuristic)	107
7.4.2	パターンデータベースヒューリスティック (Pattern Database Heuristic)	107
7.5	関連文献	107

1

イントロダクション

朝起きて、ごはんをよそい、味噌汁を作る。ご飯を食べて、職場に向かう。最寄駅まで歩き、電車に乗って職場への電車に乗る。

何故、人はごはんをよそうことが出来るのだろうか？ごはんをよそうためにしゃもじを右手にとり、茶碗を左手に持つ。炊飯器を空けて、ごはんをかき混ぜる。かき混ぜたらごはんをしゃもじの上に乗せて、茶碗の上に持っていく。しゃもじを回転させると、ごはんは茶碗に落ちる。

とても、とても難しいことをやっていると思わないだろうか？不思議なことに、我々は「ごはんをよそう」と頭にあるだけ(と自覚している)だけなのに、何故かそのために必要な行動を列挙し、一つずつ実行していけるのである。

何故、我々は殆ど頭を(自覚的に)使わずにこのような計画を立てることが出来るのだろうか？何故ごはんをよそうためにお湯を沸かしたり、最寄り駅まで歩いたりする必要はないと分かるのだろうか？それは我々が無限に近い可能な行動の集合の中から直感(ヒューリスティック)的に目的を達成するために必要そうな行動を絞り込むことが出来るからである。

この直感をコンピュータに実装することこそが高度な先読みをし、高度な行動計画を行い実行の出来るシステム、人工知能に肝要なのではないか。

本書で扱うヒューリスティック探索はその仮説に基づき、直感を用いた行動計画を行う研究分野である。

1.1 何故人工知能に探索が必要なのか

グラフ探索アルゴリズムは人工知能に限らず情報科学に多岐に渡って有用な手法である。本書では特に人工知能の要素技術としての問題を扱うために解説する。

人工知能とは何か、と考えることは本書の主眼ではない。人工知能の教科書として有名な Artificial Intelligence: Modern Approach [78] では人工知能と呼ばれる研究は主に以下の4つの目標を目指していると説明している。

1. Think Rationally (合理的に考える)
2. Think Humanly (人間的に考える)
3. Act Rationally (合理的に行動する)
4. Act Humanly (人間的に行動する)

グラフ探索アルゴリズムは主に Think Rationally を実現するための技術である¹。探索によって先読み (lookahead) をし、最も合理的な手を選ぶというのが目的である。

先読みをするという点が機械学習による Think Rationally と異なる点である。機械学習は過去に学習した経験を元に合理的な行動を選ぶというアイデアである。それに対して、探索は未来にどういう経験をするかを先読みして合理的な行動を選ぶ。

探索技術の大きな課題・欠点は世界のモデルを必要とする点である。モデルがないと未来にどういう経験をするかを先読みすることができない。例

¹探索は4つ全てに強く関係しているが本書は主に Think Rationally に注視する

例えば将棋であれば各コマがどのように動き、敵の王を詰ますと勝ち、といった情報を与えなければならない。加えてどの場面の時にどちらがどのくらい有利なのかという場面の評価値がないと強いエージェントは作れない。モデルは完璧である必要はないが、先読みをするためには有用なものでなくてはならない。

なので、探索は世界のモデルが容易に得られる問題において用いられてきた。例えば経路探索問題など地図があればおおよその距離が推定できる。今後の NASA などによる宇宙開発でも探索技術が重要であり続けると考えられている。NASA のウェブページを見ると過去と現在の探索技術を用いたプロジェクトがたくさん紹介されている²。

探索と機械学習は組み合わせて用いることでより賢い行動が出来るようになると考えられ、研究されている。モンテカルロ木探索とディープラーニングを組み合わせてプロ棋士に勝利した AlphaGo などはまさに探索と機械学習を組み合わせたエージェントの強力さを体現しているといえるだろう [81]。これはディープラーニングによって場面と次の一手の評価値を機械学習で学び、それと探索を組み合わせて良い評価値の場面につながるような手を選んでいくということをしている。もう一つの組み合わせ方としては機械学習によってモデルを学習し、それを使って探索をするという方法がある。前述のように探索にはモデルが必要であるというのが重要な問題である。例えば Atari でディープラーニングを使って探索のためのモデルを学習する研究がある [72, 80, 18]。

このように、探索アルゴリズムは人工知能技術を理解する上で欠かせない分野の一つである。特に最近大きなブレイクスルーのあった機械学習・深層学習とも強いシナジーを持っているため、これから大きな進展があると期待される分野の一つであると言えるだろう。

²<https://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/>

1.2 本書の射程

本書では状態空間問題 (state-space problem) を主な対象として扱う。グラフ探索アルゴリズムはこれに限らず様々な場面で使われるがこの本では状態空間問題に注目する。状態空間問題は与えられたゴールに到達するための行動の列、プラン (plan) を発見する問題である。

本書が主に扱う状態空間問題のうち完全情報 (perfect information) かつ決定論的 (deterministic) モデルである (2 6 章)。

世界を正確に表現することは不可能である。よって、殆どの問題はより解きやすい問題にモデル化され、モデル化された問題を解くことによって解きたい問題を解決するというのがエンジニアリングである。

どのように世界をモデルするかは非常に難しい問題である。モデルを簡単なものにすればするほど解きやすくなるが、簡単に正しいモデルをデザインする・自動生成することは非常に難しい。それだけでなくモデルが人間にとって理解しやすいものであるか、似たような他の問題にも適用可能か、様々なモデルの「良さ」が考えられる。

完全情報とは、エージェントが世界の状態を全て観察できるモデルである。神の目線に立っている。これに対して不完全情報 (partial information) モデルではエージェントは世界の状態を知ることは出来ず、代わりに観察 (observation) をすることで世界の状態の一部を知ることが出来る。実世界で動くロボットなどを考えると不完全情報モデルの方が現実的であるが、多くの問題が完全情報で十二分に表現することが出来る。

決定論的とはエージェントの行動によって世界の状態がどのように遷移するかが一意に (決定論的) に定まることである。非決定論的モデルでは遷移が一意に定まらない。同じ状態から同じアクションを取ったとしても、世界がどのように変化するかは一意に定まらない。非決定論的モデルにおける探索問題は本書では扱わない。興味があれば強化学習の教科書を読むと良い [85]。

本書が扱う完全情報決定論的モデルはシンプルなモデルである。これを対象としてグラフ探索アルゴリズムの解説をする。

1.2.1 関連書籍

ヒューリスティック探索に関連した書籍をいくつか紹介する。Judea Pearl の Heuristic [73] は 1984 年に出版されたこの分野の古典的名著であり、長く教科書として使われている本である。A*探索の基本的な性質の解析が丁寧に書かれているのでとても読みやすい。また、二人ゲームのための探索に多くの紙面を割いている。Stefan Edelkamp and Stefan Schrodل の Heuristic Search Theory and Application [27] はヒューリスティック探索について辞書的に調べられる本である。2010 年出版なので Pearl よりも新しい内容が書かれている。Stuart Russell and Peter Norvig の Artificial Intelligence [78] は人工知能の定番の教科書である。人工知能に興味がある方はこの本を読むべきである。この本は探索・プランニングだけでなく制約充足問題、機械学習、自然言語処理、画像処理など人工知能のさまざまなテーマを広く扱っている。Malik Ghallab, Dana Nau, and Paolo Traverso の Automated Planning and Acting [37] はヒューリスティック探索ではなくプランニングの本である。探索は主に Think rationally のための技術だが、探索をロボットなどの制御に応用するためには考えるだけでなく実際に行動をしなければならない (Act rationally)。この本は探索をロボットの意思決定に使うためのさまざまな技術的な課題とその解決方法を紹介した本である。

状態空間問題 (State-Space Problem)

この章ではまず、2.1 節ではグラフ探索手法が用いられる問題として状態空間問題を定義する。次に 2.2 節で状態空間問題の例をいくつか紹介する。経路探索問題や倉庫番問題など、応用がありつつ、かつ分かりやすい問題を選んだ。これらの問題はすべてヒューリスティック探索研究でベンチマークとして広く使われているものである。

2.1 節における定式化は [78]、[73]、[27] などを参考にしている。本文は入門の内容であるので、研究の詳細が知りたい方はこれらの教科書を読むべきである。

2.1 状態空間問題 (State-Space Problem)

この本では主に初期状態とゴール条件が与えられたとき、ゴール条件を満たすための経路を返す問題を探索する手法を考える。特に本書の主眼は 2 章から 6 章までで扱う状態空間問題 (state-space problem) である。

定義 1 (ユニットコスト状態空間問題、state-space problem): ユニットコスト状態空間問題 $P = (S, A, s, T)$ は状態の集合 S 、初期状態 $s \in S$ 、ゴール集合 $T \in S$ 、アクション集合 $A = a_1, \dots, a_n$ 、 $a_i : S \rightarrow S$ が与えられ、初期状態 s からゴール状態へ遷移させるアクションの列を返す問題である。

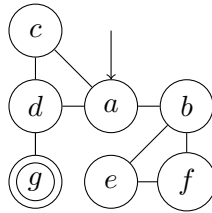


図 2.1: 状態空間問題の例。エージェントはスタート地点 a からゴール地点 g を目指す。

ユニットコスト状態空間問題はグラフにモデルすることで考えやすくなる。ユニットコスト状態空間問題を表す状態空間グラフ (state-space graph) は以下のように定義される。

定義 2 (状態空間グラフ、State-space graph): 問題グラフ $G = (V, E, s, T)$ は状態空間問題 $P = (S, A, s, T)$ に対して以下のように定義される。ノード集合 $V = S$ 、初期ノード $s \in S$ 、ゴールノード集合 T 、エッジ集合 $E \subseteq V \times V$ 。エッジ $u, v \in E$ は $a(u) = v$ となる $a \in A$ が存在する場合に存在し、そしてその場合にのみ存在する (iff)。

状態空間問題の解 (solution) は以下の定義である。

定義 3 (解、Solution): 解 $\pi = (a_1, a_2, \dots, a_k)$ はアクション $a_i \in A$ の (順序付) 配列であり、初期状態 s からゴール状態 $t \in T$ へ遷移させる。すなわち、 $u_i \in S, i \in \{0, 1, \dots, k\}$, $u_0 = s, u_k = t$ が存在し、 $u_i = a_i(u_{i-1})$ となる。

どのような解を見つけたいかは問題に依存する。多くの問題では経路コスト (path cost) の合計を小さくすることを目的とする。

定義 4 (状態空間問題、Weighted state-space problem): 状態空間問題 $P = (S, A, s, T, w)$ はユニットコスト状態空間問題の定義に加え、コスト関数 $w : A \rightarrow \mathbb{R}$ がある。経路 (a_1, \dots, a_k) のコストは $\sum_{i=1}^k w(a_i)$ と定義される。

本書ではこの状態空間問題を主に扱う。状態空間問題のうちコストが定数関数である場合がユニットコスト状態空間問題である。状態空間問題は重み付き (コスト付き) グラフとしてモデルすることが出来る。すなわち、 $G = (V, E, s, T, w)$ は状態空間グラフの定義に加え、エッジの重み $w : E \rightarrow \mathbb{R}$ を持つ。

3章で詳解するが、探索アルゴリズムは状態空間グラフのノード・エッジ全てを保持する必要はない。全てのノード・エッジを保持した状態空間グラフを特に明示的状态空間グラフ (explicit state-space graph) と呼ぶとする。このようなグラフは、例えば隣接行列を用いて表すことが出来る。隣接行列 M は行と列の大きさが $|V|$ である正方行列であり、エッジ (v_i, v_j) が存在するならば $M_{i,j} = 1$ 、なければ $M_{i,j} = 0$ とする行列である。このような表現方法の問題点は行列の大きさが $|V|^2$ であるため、大きな状態空間を保持することが出来ないことである。例えば、2.2 節で紹介する 15-puzzle は状態の数が $|V| = 15!/2$ であるため、隣接行列を保持することは現在のコンピュータでは非常に困難である。

そこで、探索アルゴリズムは多くの場合初期ノードとノード展開関数による非明示的状态空間グラフ (implicit state-space graph) で表せられる。

定義 5 (非明示的状态空間グラフ、Implicit state-space graph): 非明示的状态空間グラフ $(s, Goal, Expand, w)$ は初期状態 $s \in V$ 、ゴール条件 $Goal: V \rightarrow B = \{false, true\}$ 、ノード展開関数 $Expand: V \rightarrow 2^V$ 、コスト関数 $w: V \times V \rightarrow \mathbb{R}$ によって与えられる^a。

^a 2^V はノード集合 V のべき集合である。

非明示的状态空間グラフも状態空間問題に対して定義できる。明示的状态空間グラフとの違いは次状態の情報が隣接行列ではなくノード展開関数 $Expand$ の形に表現されている点である。 $Expand$ はある状態からの可能な次の状態の集合を返す関数である。 $Expand$ 関数は明示的に与えられるのではなく、ルールによって与えられることが多い。例えば将棋であれば、将棋のルールによって定められる合法手によって得られる次の状態の集合が $Expand$ 関数によって得られる。多くの場合このノード展開関数は隣接行列よりも小さな情報で表現できる。

本書ではすべてのノードの分枝数が有限であると仮定する (局所有限グラフ、locally finite graph)。また、特に断りがない場合簡単のため $w \geq 0$ を仮定する。

2.2 状態空間問題の例

状態空間問題の例をいくつか紹介する。これらの問題はヒューリスティック探索研究でベンチマークとして広く使われているものである。

2.2.1 グリッド経路探索 (Grid Path-Finding)

グリッド経路探索問題 (grid path-finding problem) は k (多くの場合 $k = 2$) 次元のグリッド上で初期配置からゴール位置までの経路を求める問題である [91]。

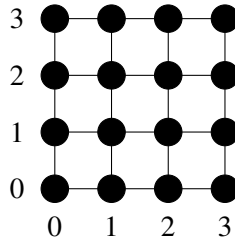


図 2.2: グリッド経路探索問題

グリッドには障害物がおかれ、通れない箇所がある。エージェントが移動できる方向は4方向 ($A = \{up, down, left, right\}$) か8方向 (4方向に加えて斜め移動) とする場合が多い。自由方向 (Any Angle) の問題を扱う研究も存在する [71]。

Web 上に簡単に試せるデモがあるので、参照されたい¹。この本で説明する様々なグラフ探索手法をグリッド経路探索に試すことが出来る。

グリッド経路探索はロボットのモーションプランニングやゲーム AIなどで応用される [5]。ストラテジーゲームなどでユニット (エージェント) を動かすために使われる [20, 84]。またグリッドは様々な問題を経路探索に帰着して解くことができるという意味でも重要である。例えば多重整列問題 (Multiple Sequence Alignment) はグリッド経路探索に帰着して解くことが出来る (節 2.2.3)。ロボットのモーションプランニングも経路探索問題に帰着することが出来ることがある [11]。 k 個の関節の角度を変えながら、現在状態から目的の状態 (Configuration) に近づけたい。各関節の角度をグリッドの各次元で表す。ロボットの物理的な構造のために不可能な角度の組み合わせを障害物の置かれたグリッドとする。このようにモデルを作ると、グリッド上で障害物を避けた経路を計算することで現在状態から目的状態へ関節をうまく動かすモーションプランが発見できる。

¹<http://qiao.github.io/PathFinding.js/visual/>

1	2	3	4
4	5	6	7
7	8	9	10
10	11	12	

図 2.3: 15 パズルのゴール状態の例

2.2.2 スライディングタイル (Sliding-tile Puzzle)

多くの一人ゲームはグラフ探索問題に帰着することが出来る。スライディングタイルはその例であり、ヒューリスティック探索研究においてメジャーなベンチマーク問題でもある (図 2.3) [54]。1 から $(n^2) - 1$ までの数字が振られたタイルが $n \times n$ の正方形に並べられている。正方形には一つだけブランクと呼ばれるタイルのない位置があり、四方に隣り合うタイルのいずれかをその位置に移動する (スライドする) ことが出来る。スライディングタイル問題は、与えられた初期状態からスライドを繰り返し、ゴール状態にたどり着く経路を求める問題である。

スライディングタイルの到達可能な状態の数は $|V| = (n^2)!/2^2$ であり、 n に対して指数的に増加する。可能なアクションは $A = \{up, down, left, right\}$ の 4 つであり、アクションにかかるコストはすべて同じとする。

後述するが、ヒューリスティック探索のためには状態からゴール状態までの距離 (コスト) の下界 (lower bound) が計算できると有用である。スライディングタイルにおける下界の求め方として最もシンプルなものはマンハッタン距離ヒューリスティックである。マンハッタン距離ヒューリスティックは各タイルの現在状態の位置とゴール状態の位置のマンハッタン距離の総和を取る。可能なアクションはすべて一つしかタイルを動かさないで、一回のアクションでマンハッタン距離は最大で 1 しか縮まらない。よって、マン

²スライディングタイルは偶奇性があり、到達不可能な状態がある [54]。

ハッタン距離はゴールまでの距離の下界である。

2.2.3 多重整列問題 (Multiple Sequence Alignment)

生物学・進化学では遺伝子配列・アミノ酸配列の編集距離 (edit distance) を比較することで二個体がどれだけ親しいかを推定することが広く研究されている。多重整列問題 (Multiple Sequence Alignment) (MSA) は複数の遺伝子・アミノ酸配列が与えられた時、それらの配列間の編集距離とその時出来上がった配列を求める問題である。2つの配列に対してそれぞれコストの定義された編集操作を繰り返し、同一の配列に並べ替える手続きをアライメントと呼ぶ。2つの配列の編集距離 (edit distance) は編集操作の合計コストの最小値である。3つ以上の配列における距離の定義は様々考えられるが、例えば全ての配列のペアの編集距離の総和を用いられる。

MSA における可能な編集操作は置換と挿入である。置換は配列のある要素 (DNA かアミノ酸) を別の要素に入れ替える操作であり、挿入は配列のある位置に要素を挿入する操作である。例えば (ATG, TGC, AGC) の3つの配列のアライメントを考える。表 2.2 は置換と編集に対するコストの例である。- は欠損を示し、対応する要素が存在しないことを表す。表 2.1 はこのコスト表を用いたアライメントの例である。このとき、例えば配列 ATG- と -TGC の編集距離は (A,-)、(T,T)、(G,G)、(-,C) のコストの総和であるので、表 2.2 を参照し、 $3 + 0 + 0 + 3 = 6$ である。同様に (ATG-, A-GC) の距離は 9、(-TGC, A-GC) の距離は 6 であるので、3 配列の編集距離は $6 + 9 + 6 = 21$ である。

n 配列の MSA は n 次元のグリッドの経路探索問題に帰着することが出来る [62]。図 2.3 は (ATG) と (TGC) の2つの配列による MSA をグリッド経路探索問題に帰着した例である。状態 $s = (x_1, x_2, \dots, x_n)$ の各変数 x_i は配列 i のどの位置までアライメントを完了したかを表す変数であり、配列 i の長さを l_i とすると定義域は $0 \leq x_i \leq l_i$ である。全てのアライメントが完了した状態 $s = (l_1, l_2, \dots, l_n)$ がゴール状態である。可能なアクション $a = (b_1, b_2, \dots, b_n), (b_i = 0, 1)$ は配列 i に対してそれぞれ欠損を挿入するかどうかであり、配列 i に対して欠損を挿入する場合に $b_i = 0$ 、挿入しない場

表 2.1: 多重配列問題 (MSA)

Sequence1	A	T	G	-
Sequence2	-	T	G	C
Sequence3	A	-	G	C

表 2.2: MSA の塩基配列のコスト表

	A	T	G	C	-
A	0	3	3	3	3
T	3	0	3	3	3
G	3	3	0	3	3
C	3	3	3	0	3
-	3	3	3	3	0

合は $b_i = 1$ となる。状態 s に対してアクション a を適用した後の状態 s' は $s' = (x_1 + b_1, x_2 + b_2, \dots, x_n + b_n)$ となる。図 2.3 は初期状態 $s = (0, 0)$ に対して $a = (1, 0)$ を適用している。これは (A), (-) までアライメントを進めた状態に対応する。次に $a = (1, 1)$ が適用され、アライメントは (A,T), (-,T) という状態に遷移する。このようにして $(0, 0)$ から (l_1, l_2) までたどり着くまでの最小コストを求める。

MSA は可能なアクションの数が配列の数 n に対して指数的に増える ($2^n - 1$) 点が難しい。アミノ酸配列が対象である場合はコストの値が幅広い点も難しい [74]。MSA は生物学研究に役立つというモチベーションから非常に熱心に研究されており、様々な定式化による解法が知られている。詳しくは [88, 39, 28] を参照されたい。

表 2.3: MSA のグリッド経路探索問題への帰着

	A	T	G	-
T	→	↘		
G			↘	
C				↓

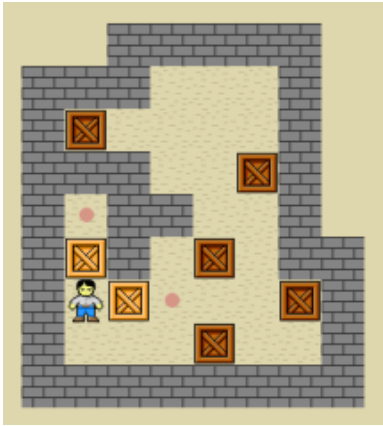


図 2.4: 倉庫番

2.2.4 倉庫番 (Sokoban)

倉庫番 (Sokoban) は倉庫の荷物を押していくことで指定された位置に置くというパズルゲームである。現在でも様々なゲームの中で親しまれている [55, 21]。プレイヤーは「荷物の後ろに回って押す」ことしか出来ず、引っ張ったり、横から動かしたりすることが出来ない。また、荷物の上を通ることも出来ない。PSPACE-complete であることが知られている [21]。

状態の表現方法は2通りあり、一つはグリッドの各位置に何が置いてあるかを変数とする方法である。もうひとつはプレイヤー、各荷物の位置に対

してそれぞれ変数を割り当てる方法である。可能なアクションは `move-up`, `move-left`, `move-down`, `move-right`, `push-up`, `push-left`, `push-down`, `push-right` の 8 通りである。 `move-*` はプレイヤーが動くアクションに対応し、コストは 0 である。 `push-*` は荷物を押すアクションであり、正のアクションコストが割当てられている。よって、倉庫番はなるべく荷物を押す回数を少なくして荷物を目的の位置に動かすことが目的となる。

グラフ探索問題として倉庫番を考えるとときに重要であるのは、倉庫番は不可逆なアクション (irreversible action) が存在することである。全てのアクション $a \in A$ に対して $a^{-1} \in A$ が存在し、 $a(a^{-1}(s)) = s$ かつ $a^{-1}(a(s)) = s$ となる場合、問題は可逆 (reversible) であると呼ぶ。例えばグリッド経路探索やスライディングタイルは可逆である。可逆な問題は対応するアクションのコストが同じであれば無向グラフとしてモデルすることもでき、初期状態から到達できる状態は、すべて初期状態に戻ることが出来る。一方、不可逆な問題ではこれが保証されず、デッドエンドにはまる可能性がある (2.3 節)。

倉庫番では荷物を押すことは出来ても引っ張ることが出来ないため、不可逆な問題である。例えば、荷物を部屋の隅に置いてしまうと戻すことが出来ないため、詰み状態に陥る可能性がある問題である。このような性質を持つ問題では特にグラフ探索による先読みが効果的である。

倉庫番のもうひとつ重要な性質はゼロコストアクション (zero-cost action) の存在である。ゼロコストアクションはコストが 0 のアクションである。倉庫番のアクションのうち `move-up`, `move-left`, `move-down`, `move-right` はコストゼロ ($w(e) = 0$) のアクションである。へたなアルゴリズムを実行すると無限に無駄なアクションを繰り返し続けるということもありうるだろう。

2.2.5 巡回セールスパーソン問題 (Traveling Salesperson Problem, TSP)

巡回セールスパーソン問題は、地図上にある都市を全て回り最初の都市に戻るときの最短距離の経路を求める問題である [6]。 n 個の都市があるとして (非最適解含む) 解の数は $(n - 1)!/2$ 個である。可能なアクションは「都

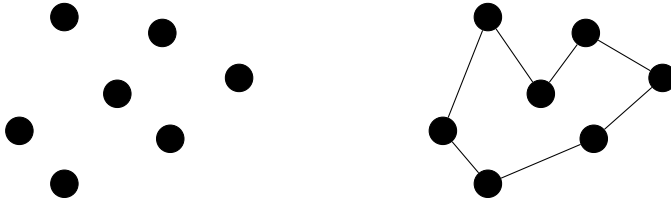


図 2.5: 巡回セールスパーソン問題

市 $i \in \{1..n\}$ を訪れる」であり、一度訪れた都市には行けない。TSP のゴール条件はすべての都市を訪れることである。よって、何も考えずに n 回アクションを実行すれば、とりあえず解を得ることが出来る。しかし最適解を得ることは難しく、NP 完全問題であることが知られている。TSP はヒューリスティック探索に限らず、様々なアプローチで研究されている実用的に非常に重要なドメインである [6]。TSP について特に詳しく知りたい方はそちらの教科書を参照されたい。

2.3 問題の性質・難しさ

本書で定義した状態空間問題は小さなモデルである。完全情報であり、状態遷移は決定論的である。それでも NP 困難問題であり、難しい問題は難しい。この節は問題の難しさがどのような要素に左右されるかを列挙する。

1. 状態空間の大きさ

状態空間の大きさ $|S|$ は大きい程概して問題は難しくなる。特に状態空間が無限である場合深さ優先探索などのアルゴリズムは停止しない場合がある。例えば状態変数に実数が含まれる場合、状態空間の大きさは無限になる。

2. 分枝度

ある状態 s の分枝度 (branching factor) はそのノードの子ノードの数を指す。特に状態空間問題の分枝度は、すべての状態の分枝度の平均を指す。ただし多くの場合平均を厳密に求めることはなく、おおよその平均を指して分枝度をすることが多い。分枝度が大きいほど問題は難しいとは限らない。分枝度が多いほどグラフが密である、つまりエッジの数が多いことに対応する。分枝度を b とすると、あるノード s の子ノードの数は b 個であり、孫ノードの数は b^2 である。 s からの深さ d のノードは b^d 個である。

3. デッドエンド

問題によってはある状態に到達するともう問題を解くことは出来ないというシチュエーションがある。例えば倉庫番は荷物を角においてしまうともう動かすことができない。これによってもう問題がクリアできなくなるということがある。このような問題では状態空間を広く探索し、デッドエンド状態のみを探索し続けるということをうまく避ける必要がある。例えば 4.5.2 節の貪欲最良優先探索はデッドエンドに入ってしまうとなかなか抜け出せないかもしれない。概してデッドエンドがある問題では状態空間を広く探索する手法、探索済みの状態を記録する手法が有利であり、局所探索手法はうまくいかないことがある。

4. 解の存在

当然解が存在しない問題もありうる。本書で紹介するアルゴリズムは解が存在しない場合非常に時間がかかるものが多い³。一部のアルゴリズムは解が存在しない場合永遠に停止しない場合がある。そのような場合、アルゴリズムは解が存在しないと示せば理想的である。そのため解が存在しないことを検出するアルゴリズムの研究もされている [9, 47]。

³一般にどのようなアルゴリズムを使っても解が存在しない状態空間問題は難しい。

2.4 関連文献

本章で定義した状態空間問題は基本となる定義であり、完全情報であり状態遷移が決定論的であることを仮定した。状態遷移が決定論的ではなく確率的であると仮定したモデルはマルコフ過程問題 (Markov Decision Process Problem) (MDP) と呼ばれている。状態空間問題は MDP の特殊な場合である。MDP は強化学習 (reinforcement learning) における問題モデルとしても広く使われている。MDP におけるプランニング問題を解くためにはグラフ探索アルゴリズムも使えるが、動的計画法も用いられる [75]。MDP からさらに不完全情報問題に拡張したものを部分観測マルコフ過程問題 (partially observable Markov decision process problem) (POMDP) と呼ぶ。POMDP におけるプランニング問題の厳密解は Belief space プランニング [56] によって求められるが、多くの場合計算困難 (intractable) であるので近似手法が用いられる。

本書では状態空間問題の解をゴールに到達するまでの経路と定義した。状態空間問題のもう一つの定式化として、状態とアクションの組に対して報酬 (reward) ($R : (S, A) \rightarrow \mathbb{R}$) を定義し、報酬を最大化する経路を求める問題がある。この定式化は特に強化学習で用いられる。

情報なし探索 (Blind Search)

1 章では様々な状態空間問題を紹介したが、それぞれの問題の解法はどれも沢山研究されている。一つの指針としては、ある問題に特化した解法を研究することでその問題をより高速に解くというモチベーションがある。これは例えば MSA のように重要なアプリケーションがある問題の場合に特に熱心に研究されることが多い。一方、なるべく広い範囲の問題に対して適用可能な手法を研究するというモチベーションもある。この章で紹介する手法は問題特化のアルゴリズムよりもパフォーマンスに劣るが、問題の知識をあまり必要とせず、さまざまな問題に適用できる。

1 章で紹介した状態空間問題を広く扱うことの出来る手法としてグラフ探索アルゴリズムがある。本章では最もシンプルな問題（ドメイン）の知識を利用しない探索を紹介する。情報なし探索 (Blind Search) は状態空間グラフのみに注目し、背景にある問題に関する知識を一切使わないアルゴリズムである。このような探索では 1. 重複検知を行うか 2. ノードの展開順序が重要になる。重複検出は訪問済みの状態を保存しておくことで同じ状態を繰り返し探索することを防ぐ手法である。対価としては、メモリの消費量が非常に大きくなることにある。ノードの展開順序とは、例えば幅優先探索・深さ優先探索などのバリエーションを指す。効率的な展開順序は問題によって大きく

異なり、問題を選べばこれらの手法によって十分に効率的な探索を行うことが出来る。これらの探索手法は競技プログラミングでもよく解法として使われる [82]。また、いわゆるコーディング面接でもグラフ探索アルゴリズムは頻出である [67]。情報なし探索は [19] の 22 章 Elementary Graph Algorithms にも詳しく説明されている。

3.1 木探索アルゴリズム (Tree Search Algorithm)

木探索アルゴリズムはグラフ探索アルゴリズムの基礎となるフレームワークであり、本文で紹介する手法のほとんどがこのフレームワークを基礎としているといえる。アルゴリズム 1 は木探索の疑似コードである。

木探索はオープンリスト¹と呼ばれるノードの集合を Priority queue に保持する。探索の開始時には、初期状態のみがオープンリストに入っている。木探索は、このオープンリストから一つノード u を選び、ゴール条件を満たしているかを確認する。満たしていれば初期状態から u への経路を返す。満たしていなければ、そのノードを展開する。展開とは、そのノードの子ノードを列挙し、オープンリストに入れることを指す。

各ノードの注目すると、ノードは 1. 未生成、2. 生成済み、3. 展開済みと状態が遷移していく。

1. 未生成ノード (Undiscovered nodes): 状態空間内のまだ生成されていないノード。非明示的グラフでは情報は保持されていない。
2. 生成済みノード (Generated nodes): オープンリストに一度でも入れられたノード。後述するグラフ探索ではクローズドリストに入れられる。
3. 展開済みノード (Expanded nodes): *Expand* を実行し終えたノード。子ノードがすべて生成済みノードになる。オープンリストからは取り除

¹歴史的な経緯でリストと呼ばれているが、データ構造がリストで実装されるという意味ではない。効率的なデータ構造は 5 章で紹介する。

アルゴリズム 1: 木探索 (Implicit Tree Search)

Input : 非明示的状态空間グラフ $(s, Goal, Expand, w)$, プライオリティ関数 f

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $Open \leftarrow \{s\}, d(s) \leftarrow 0, g(s) \leftarrow 0;$ 
2 while  $Open \neq \emptyset$  do
3    $u \leftarrow \arg \min_{u' \in Open} f(u');$ 
4    $Open \leftarrow Open \setminus \{u\};$ 
5   if  $Goal(u)$  then
6     return  $Path(u);$ 
7   for each  $v \in Expand(u)$  do
8      $Open \leftarrow Open \cup \{u\};$ 
9      $d(v) \leftarrow d(u) + 1;$ 
10     $g(v) \leftarrow g(u) + w(u, v);$ 
11     $parent(v) \leftarrow u;$ 
12 return  $\emptyset;$ 

```

かれる。

図 3.1 は探索アルゴリズムが $\{a, b, c\}$ を展開し終えた時点でのノードの分類の例である。 $\{a, b, c\}$ は展開済みノードであり、同時に生成済みノードである。これらのノードは後述するグラフ探索ではクローズドリストと呼ばれるデータ構造に入れられ、保持される。これらのノードの子ノードがすべて生成される。 $\{d, e, f\}$ は生成済みノードであるが展開済みではない。探索アルゴリズムのオープンリストにはこれらのノードが入っている。 $\{g, h, i, j\}$ は未生成のノードであり、アルゴリズムからは見えない未発見のノードである。例えば次に d を展開すると、展開済みノードは $\{a, b, c, d\}$ となり、新たに $\{g, h\}$ が未生成ノードから生成済みノードに遷移する。このように探索が

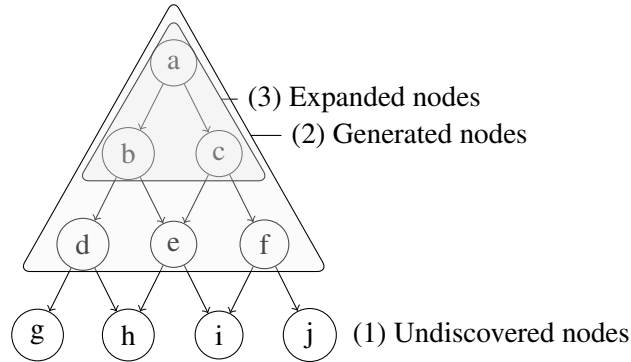


図 3.1: 未生成・生成済み・展開済みノードの例。木探索は生成済みノードのうち展開済みでないものをひとつ取り出し、そのノードを展開し子ノードを得る。新しく得られた子ノードは未生成ノードから生成済みノードとなり、展開したノードは展開済みノードになる。

進行していくことによってノードは順次状態遷移していく。

初期状態からノード n への最小ステップ数を深さ d と呼び、最小経路コストを g 値と呼ぶ。すべてのアクションのコストが 1 のドメインであれば任意の n に対して $d(n) = g(n)$ が成り立つ。状態を更新すると同時に g 値を更新する。これによって解を発見した時に解ノードの g 値が解のコストとなる。なお、状態 s に対して適用可能なアクションの集合 $A(s)$ は与えられていると仮定する。

$Path(u)$ はノードに対して初期状態からそこへ到達するまでの経路を返す関数である。 $(u, parent(u), parent(parent(u)), \dots, s)$ のように再帰的に $parent$ を初期状態まで辿る。

紛らわしいが、木探索アルゴリズムは木だけでなくグラフ一般を探索するアルゴリズムである。木探索の強みは生成済みノードのうち展開済みではないもののみをオープンリストに保持すればよいことにある。未生成ノード、展開済みノードはメモリ上に保持する必要がない。一方この問題は、一度展開したノードが再び Expand によって現れた場合再展開 (reexpansion) をす

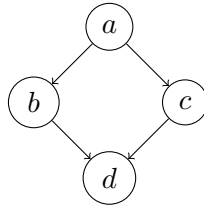


図 3.2: ノードの再展開の例

ることになる。図 3.2 の例ではノード d に二通りの経路で到達できるが、木探索では二回目に d に到達したとき、すでに到達済みであることを把握できない。そのため同じノードを再び生成・展開することになる。このように木探索は複数の経路で到達可能なノードがあるほど (グラフがより木から遠いほど) 同じノードを何度も再展開することになり、効率が悪くなってしまう。

また、木探索アルゴリズムは状態数が有限であっても停止しない場合がある。これらが問題になるような問題ドメインである場合は後述する重複検出を使うグラフ探索 (3.2 節) を使うと良いだろう。

オープンリストはプライオリティキューであり、どの順番でノードを取り出すかを決めなければならない。プライオリティ f の選択は探索アルゴリズムの性能に強く関係する。本章の 3.3 節以降、及び 4 章はこのプライオリティをどうデザインするかについて議論をする。

3.2 グラフ探索アルゴリズム (Graph Search Algorithm)

明示的グラフのあるノードが初期状態から複数の経路でたどり着ける場合、同じ状態を表すノードが木探索による非明示的グラフに複数現れるということが生じる。このようなノードを重複 (duplicate) と呼ぶ。ノードの重複は計算資源を消費してしまうので、効率的な重複検出 (duplicate detection) の方法は重要な研究分野である。本書ではノードの重複検出を行う探索アルゴリズムを狭義にグラフ探索アルゴリズムと呼び、重複検出を行わない探索を狭義

に木探索アルゴリズムと呼ぶ。

アルゴリズム 2: グラフ探索 (Implicit Graph Search)

Input : 非明示的状态空間グラフ $(s, Goal, Expand, w)$ 、プライオリティ関数 f
Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $Closed \leftarrow \emptyset, Open \leftarrow \{s\}, d(s) \leftarrow 0, g(s) \leftarrow 0;$ 
2 while  $Open \neq \emptyset$  do
3    $u \leftarrow \arg \min_{u' \in Open} f(u');$ 
4    $Open \leftarrow Open \setminus \{u\};$ 
5   if  $Goal(u)$  then
6     return  $Path(u);$ 
7   for each  $v \in Expand(u)$  do
8     if  $v \notin Closed$  or  $g(u) + w(u, v) < g(v)$  then
9        $Open \leftarrow Open \cup \{v\};$ 
10       $d(v) \leftarrow d(u) + 1;$ 
11       $g(v) \leftarrow g(u) + w(u, v);$ 
12       $parent(v) \leftarrow u;$ 
13     if  $v \notin Closed$  then
14        $Closed \leftarrow Closed \cup \{v\};$ 
15 return  $\emptyset;$ 

```

重複検出のためには生成されたノードをクローズドリスト (closed list) に保存する。一度クローズドリストに入れられたノードはずっとクローズドリストに保持される。ノード展開関数から子ノードが生成されたら、その子ノードと同じ状態を保持するノードがクローズドリストに存在するかを確認する。もし存在しなければ、そのノードは重複ではない。なのでそのノードをオープンリストに加える。存在した場合の処理は少しややこしい。新たに生成さ

れたノード n の g 値のほうが先に生成されクローズドリストにあるノード n' の g 値よりも小さい場合が存在する。このとき、 n をそのまま捨ててしまうと、そのノードの g 値が本来の値よりも大きく評価されてしまう。

g 値をそのノードに到達できる既知の最小コストにするためには、まずクローズドリストに保存されているノードの g 値を $g(n')$ から $g(n)$ に更新しなければならない。加えて、ノード n を再展開 (reexpansion) しなければならない。ノード n の子ノード c は n' の子ノードとして展開されていたわけであるが、そのとき $g(c) = g(n') + w(n', c)$ として計算された。この値は $g(c) = g(n) + w(n, c)$ に更新しなければならない。 $w(n', c) = w(n, c)$ なので、 $g(n') - g(n)$ だけ g 値が小さくなる。なので、 c の子ノードも再展開をする必要がある。そしてそのまた子ノードも。。。。というように、再展開が生じるとそこから先のノードをすべて再展開する必要がある。これはかなり大きなコストになることが多いので、可能な限り避けたい処理である。

重複が存在した場合に必ずノードを捨てることのできる場合も存在する。まず、解の最適性が不要でない場合 g 値を更新する必要はない。 g 値が過大に評価されても解経路は解経路のままであり、ただ解経路のコストが大きくなるだけである。また、例えば幅優先探索では探索の過程で生成されるノードの d 値は単調増加する。もしユニットコストドメインならば g 値も単調増加である。つまりノード n と重複したノード n' がクローズドリストにあったとすると、 $g(n) \geq g(n')$ が成立する。この場合、解最適性を保ったまま n を安全に捨てることのできる。また、状態空間グラフが木である場合は重複が発生しない。なお、後述する A*探索 4.3 ではある条件を満たせば再展開は行わずに解の最適性が満たせることが知られている。これが A*探索が state-of-the-art として重要視されている理由である。

ここで「ノード」と「状態」の言葉の使い分けに注意したい。状態とは状態空間問題における状態 s である。ノードは状態 s を含み、 f 値、 g 値の情報を含む。重複検出を行わない木探索の場合、同じ状態を保持するノードが2つ以上存在しうる。重複検知は同じ状態を保持するノードをマージする処理に相当する。この処理を行うと同じノードに複数の経路で到達するよう

になり、グラフは木ではなくなる。

重複検出の問題はメモリの使用量である。重複検出を行うためには生成済みノードをすべてクローズドリストに保存しなければならない。なので展開済みノードの数に比例した空間が必要になる。クローズドリストの効率的な実装については 5.2 節で議論をする。

なお、重複検出はノードが生成されたときではなく、ノードが展開される時に遅らせることができる。オープンリストには重複したノードが含まれることになるが、ノードの展開時には重複をチェックするので重複したノードの展開は防げる、ということである。これは遅延重複検出 (delayed duplicate detection) と呼ばれ、5.2.4 節で議論をする。

3.3 幅優先探索 (Breadth-First Search)

探索のパフォーマンスにおいて重要になるのはどのようにして次に展開するノードを選択するかにある。ヒューリスティック探索の研究の非常に大きな部分はここに費やされているといえる。シンプルかつ強力なノード選択方法は First-in-first-out (FIFO) である。あるいは幅優先探索と呼ぶ。

幅優先探索の手順は非常に単純であり、FIFO の順に *Open* から取り出せばいいだけである。これをもう少し大きな視点で、どのようなノードを優先して探索しているのかを考えてみたい。初期状態から現在状態にたどり着くまでの経路の長さをノードの d 値と定義する。すると、幅優先探索のプライオリティ関数 f は d 値と一致する。

$$f_{\text{brfs}}(s) = d(s) \quad (3.1)$$

ユニットコスト問題である場合、更に g 値とも一致する ($f_{\text{brfs}}(s) = d(s) = g(s)$)。

幅優先探索のメリットは最初に発見した解が最短経路長の解であることである。問題がユニットコストドメインであれば、最短経路が最小コスト経

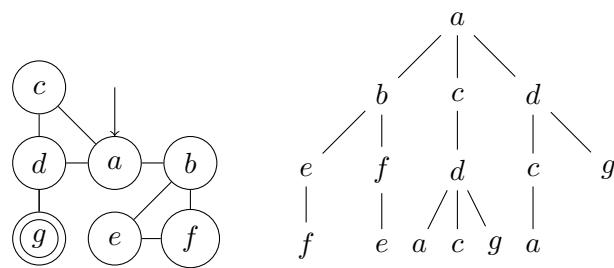


図 3.3: 探索木

路であるので、最適解が得られる。なお、後述する Best First Search と区別するため、Breadth-First Search の略称は BrFS を用いることがある (Best First Search は BFS となる)。

重複検出を用いた幅優先探索で図 2.1 の問題を解こうとすると、オープンリスト、クローズドリストの中身は表 3.1 のように遷移する。図 3.3 の探索木を見比べながら確認してほしい。

表 3.1: 重複検出を用いた幅優先グラフ探索のオープンリスト・クローズドリスト ([27] より)

ステップ	ノードの選択	オープンリスト	クローズドリスト	コメント
1	{}	{a}	{}	
2	a	{b,c,d}	{a}	
3	b	{c,d,e,f}	{a,b}	
4	c	{d,e,f}	{a,b,c}	
5	d	{e,f,g}	{a,b,c,d}	
6	e	{f,g}	{a,b,c,d,e}	
7	f	{g}	{a,b,c,d,e,f}	
8	g	{}	{a,b,c,d,e,f,g}	ゴール

3.4 深さ優先探索 (Depth-First Search)

幅優先探索が幅を優先するのに対して深さ優先探索はもっとも深いノードを優先して探索する。

$$f_{\text{dfs}}(s) = -d(s) \quad (3.2)$$

深さ優先探索は解がある一定の深さにあることが既知である場合に有効である。例えば TSP は全ての街を回ったときのみが解であるので、街の数が n であれば全ての解の経路長が n である。このような問題を幅優先探索で解こうとすると、解は最も深いところにしかないので、最後の最後まで解が一つも得られないということになる。一方、深さ優先探索なら n 回目の展開で一つ目の解を見つけることが出来る。表 3.2 は図 2.1 の問題で重複検出ありの深さ優先探索を行った場合のオープンリスト・クローズドリストの遷移を示した。図 3.3 と合わせてノードが展開される順序を確認すると良い。

深さ優先探索は無限グラフにおいて、解が存在しても永遠に停止しない場合がある。幅優先探索であれば解がある場合、いずれそれを発見する (解の深さを d^* とすると、 $d(s) \leq d^*$ であるノードの数は有限であるので)。しかし深さ優先探索は停止しない場合がある。

良い解、最適解を見つけない場合でも深さ優先探索が有用である場合がある。早めに一つ解が見つけられると、その解よりも質が悪い解にしかつながらないノードを枝刈り (pruning) することが出来る。ノード n を枝刈りするとは、ノード n をオープンリストに加えずそのまま捨てることを指す。つまりアルゴリズム 1 における $Open \leftarrow Open \cup \{v\}$ をスキップする。このような枝刈りを用いた探索アルゴリズムを分枝限定法 (Branch-and-Bound) と呼ぶ。

3.4.1 再帰による深さ優先探索

上述の実装はオープンリストを利用した深さ優先探索である。アルゴリズム 1 を元にした深さ優先探索の実装は効率的ではないことが多い。深さ優先探

表 3.2: 重複検出を用いた深さ優先グラフ探索のオープンリスト・クローズドリスト ([27] より)

ステップ	ノードの選択	オープンリスト	クローズドリスト	コメント
1	{}	{a}	{}	
2	a	{b,c,d}	{a}	
3	b	{e,f,c,d}	{a,b}	
4	e	{f,c,d}	{a,b,e}	
5	f	{c,d}	{a,b,e,f}	
6	c	{d}	{a,b,e,f,c}	
7	d	{g}	{a,b,e,f,c,d}	
8	g	{}	{a,b,e,f,c,d,g}	ゴール

索は再帰によって効率的に実装することができる (アルゴリズム 3)。再帰実装ではオープンリストがないことに注目したい。再帰を利用することでデータ構造を明に保存する必要がなくなり、キャッシュ効率が良くなることがある。幅優先探索も同様に再帰によって実装することが出来るが、効率的であるケースはあまりない。

3.5 ダイクストラ法 (Dijkstra Algorithm)

ダイクストラ法 (Dijkstra’s Algorithm) はグラフ探索アルゴリズムの一種であり、グラフ理論の教科書などでも登場する情報科学全体に多岐に渡り重要とされるアルゴリズムである [22]。例えばネットワークルーティングにおける link state algorithm などに Dijkstra が使われる [68]。ダイクストラ法はグラフ探索において g 値が最も小さいノードを優先して展開するアルゴリズムと説明することができる。

アルゴリズム 3: DFS: 再帰による深さ優先探索 (Depth-First Search)

Input : 非明示的状态空間グラフ $P = (s, Goal, Expand, w)$ 、前
状態 s'

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ
 \emptyset

```

1 if Goal( $s$ ) then
2   | return  $s$ ;
3 for each  $u \in Expand(s) \setminus \{s'\}$  do
4   |  $v \leftarrow DFS(u, Goal, Expand, w, s)$ ;
5   | if  $v \neq \emptyset$  then
6   |   | return ( $s, DFS(v, Goal, Expand, w, u)$ )
7 return  $\emptyset$ ;
```

$$f_{\text{bfs}}(s) = g(n) \quad (3.3)$$

ダイクストラ法は重複検出を行うグラフ探索アルゴリズムである。ダイクストラ法は非負コストグラフにおいて最短経路を返す。ユニットコストドメインでは $\forall n (g(n) = d(n))$ であるため、幅優先探索と同じ動作をする。フィボナッチヒープを用いてオープンリストを実装したダイクストラ法は $O(|E| + |V| \log |V|)$ 時間であることが知られている [34]。そのため、後述するヒューリスティック関数が得られない問題においてはとりあえずダイクストラ法を試してみることは有効である。

3.6 関連文献

ダイクストラ法はコストが負のエッジを持つ場合にうまくいかない。負のエッジを含む問題を解くための手法としてはベルマン-フォード法が有名である [12, 32]。

ダイクストラ法などの情報なし探索は組み合わせ最適化 (combinatorial optimization) の手法として使われることも多い [86]。

No Free Lunch 定理 [90] はコンピュータサイエンスの多くの最適化問題で言及される定理である。状態空間問題における No Free Lunch 定理の主張はざっくりと説明すると以下である。

定理 1: すべての可能なコスト関数による状態空間問題のインスタンスの集合を考える。この問題集合に対する平均性能はすべての探索アルゴリズムで同じである。

つまり、問題の知識が何もなければ「効率的なアルゴリズム」というものは存在しない。問題に対して知っている知識を利用することによってはじめて探索を効率的にすることができる。すなわち、状態空間問題のインスタンスの一部分で性能を犠牲にすることで、他のインスタンス集合への性能を向上させることができる。例えば解の長さが 20 以下であるという知識があるとすれば、探索を深さ 20 までで打ち切ると良いと考えられる。この探索アルゴリズムは解の長さが 20 以下であるインスタンスに対しての性能は良くなるが、解の長さが 20 より大きいインスタンスの性能は著しく悪くなる。このように知識を利用して探索の方向性を変えることがグラフ探索では重要になる。それが次章で扱うヒューリスティック探索の肝である。

ヒューリスティック探索 (Heuristic Search)

3章では問題の知識を利用しないグラフ探索手法について解説した。本章では問題の知識を利用することでより効率的なグラフ探索を行う手法、特にヒューリスティック探索について解説する。

4.1 ヒューリスティックとは？

経路探索問題をダイクストラ法で解くことを考えよう。図 4.1 のグリッド経路探索問題で $(0, 0)$ の位置から $(3, 0)$ の位置まで移動するため経路を求める問題を考えよう。このときダイクストラ法が探索していく範囲は図 4.1 の灰色のエリアにあるノードになる。しかし人間が経路探索を行うときにこんなに広い領域を探索しないだろう。グリッドの右側、ゴールのある方向に向かってのみ探索するだろう。なぜか。それは人間が問題の特徴を利用して、このノードを展開したほうがよいだろう、このノードは展開しなくてよいだろう、という直感を働かせているからである。問題の特徴を利用してノードの有望さをヒューリスティック関数 (heuristic function) として定量化し、それを探索に利用したアルゴリズムをヒューリスティック探索 (heuristic search) と呼ぶ。

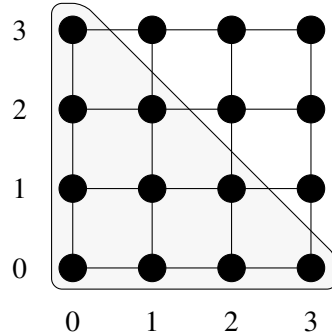


図 4.1: グリッド経路探索におけるダイクストラ法

4.2 ヒューリスティック関数 (Heuristic Function)

ヒューリスティック関数は状態の有望さの評価値であり、多くの場合その状態からゴールまでの最短距離の見積もりであることが多い [41]。

定義 6 (ヒューリスティック関数、heuristic function): ヒューリスティック関数 h はノードの評価関数である。 $h : V \rightarrow \mathbb{R}_{\geq 0}$

ヒューリスティックの値が小さいノードほどゴールに近いと推測できるので、探索ではヒューリスティック値が小さいノードを優先して展開する。ヒューリスティック関数の値をそのノードの h 値と呼ぶ。

ヒューリスティック関数の望ましい性質として、まず正確である方が望ましい。すなわち、 h 値が実際のゴールまでの最短距離に近いほど、有用な情報であると言える。ノード n からゴールまでの正しい最短コストを h^* とする。ヒューリスティック関数 h が任意の n に対して $h(n) = h^*(n)$ である場合、完璧なヒューリスティック (Perfect Heuristic) と呼ぶ。完璧なヒューリスティックがある場合、ほとんどの場合その問題を一瞬で解くことができる。現実には完璧なヒューリスティックはなかなか得られないが、ヒューリスティック

関数がこれに近いほど必要な展開ノード数が小さいことが多い[45]。反対に役に立たないヒューリスティック関数は $h(n) = 0$ などの定数関数である。これはどのノードに対してもゴールまでの距離が同じだと推測しているということであり、つまり何も主張をしていない。定数関数をヒューリスティックに使った探索をブラインド探索 (blind search) と呼ぶ。3 章で扱った情報なし探索はヒューリスティック探索の特別な場合と考えることができる。

もう一つ望ましい性質は h 値が最適解コストの下界である場合である。4.3 章で解説するが、 h 値が最短距離の下界である場合、それを用いた効率的な探索アルゴリズム (A*探索、重み付き A*探索) において解コストに理論的保証が得られることが広く知られている。 h 値が常に最適解コストの下界であるヒューリスティック関数を許容的なヒューリスティック (admissible heuristic) と呼ぶ。

定義 7 (許容的なヒューリスティック, admissible heuristic): ヒューリスティック関数 h は最適解のコストの下界である場合、許容的である。すなわち、全てのノード $u \in V$ に対して $h(u) \leq h^*(u)$ が成り立つ。

ただし、 $h^*(u)$ はノード u からゴールノード集合 T のいずれかへたどり着くための最短経路である。

一般に、許容的なヒューリスティックを得る方法としては、元問題の緩和問題 (relaxed problem) を解き、その最適解コストをヒューリスティック値とすることである。ある問題の緩和問題とは、解集合に元の問題の解を含む問題を指す。要するに元の問題より簡単な問題である¹。

もう一つ有用なヒューリスティックは無矛盾なヒューリスティック (consistent heuristic) である。

定義 8 (無矛盾なヒューリスティック, consistent heuristic): ヒューリス

¹ 解が多いほど簡単であるとは一概には言えないが

ティック関数 h は全てのノードのペア (u, v) に対して

$$h(u) \leq h(v) + k(u, v) \quad (4.1)$$

が成り立つ場合、無矛盾である。 $k(u, v)$ は u から v への最小コストである (経路がない場合 $k = \infty$ とする)。

無矛盾性は特に 4.3 章で後述する A*探索において探索の効率性に重要な性質である。また、無矛盾なヒューリスティックのうちゴールノードの h 値が 0 となるヒューリスティックは許容的である。

定理 2: ゴールノード $n \in T$ に対して $h(n) = 0$ となる無矛盾なヒューリスティックは許容的なヒューリスティックである。

Proof. すべての状態 $u \in S$ に対して以下が成り立つ。

$$h(u) \leq h(n) + k(u, n) \quad (4.2)$$

$$= k(u, n) \quad (4.3)$$

$$= h^*(u) \quad (4.4)$$

よって $h(s) \leq h^*(s)$ より許容的である。 \square

ゴールノードに対してヒューリスティック値が 0 になるヒューリスティック関数を作ることは簡単である。単純に $Goal(s)$ ならば $h(s) = 0$ とすればよい。ヒューリスティックの無矛盾性は証明することは難しそうであるが、実はシンプルな導き方がある。そのためにまず単調なヒューリスティック (monotone heuristic) を定義する。

定義 9 (単調なヒューリスティック、monotone heuristic): ヒューリスティック関数 h は全てのエッジ $e = (u, v) \in E$ に対して $h(u) \leq h(v) + w(u, v)$

が成り立つ場合、無矛盾である。

単調なヒューリスティックは無矛盾性の定義と比較して、ノードのペア (u, v) が直接繋がっているという制約がある分、弱い性質に思えるかもしれない。しかし実は単調性と無矛盾性は同値である。

定理 3: 単調性と無矛盾性は同値である。

証明: 無矛盾性であるならば単調性であることは自明である。単調性が成り立つ場合に無矛盾であることを示す。 (u, v) に経路がない場合 $k = \infty$ なので

$$h(n_0) \leq h(n_1) + w(n_0, n_1) \quad (4.5)$$

$$\leq h(n_2) + w(n_0, n_1) + w(n_1, n_2) \quad (4.6)$$

$$\dots \quad (4.7)$$

$$\leq h(n_k) + \sum_{i=0..k-1} (w(n_i, n_{i+1})) \quad (4.8)$$

$$(4.9)$$

$u = n_0, v = n_k$ とすると無矛盾性が成り立つ。

よって、無矛盾性を示すために必要なのは直接つながっているノードのペアに対して $h(u) \leq h(v) + w(u, v)$ が成り立つことを示せばよい。これを示すことは比較的シンプルである。

これらのアルゴリズムの性質は後述する A*探索において非常に有用である。

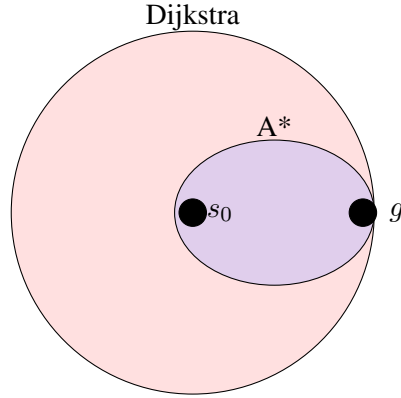
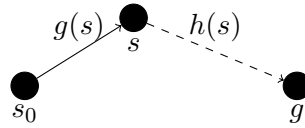


図 4.2: A* (ヒューリスティック探索) とダイクストラ法 (情報なし探索) の探索範囲の比較

4.3 A*探索 (A* Search)

ダイクストラ法は初期状態からそのノードまでのコストである g 値が最小のノードを展開していく。これは間違った方針ではないだろうが、理想的にはゴール状態に向かっていくノードを展開していきたい。図 4.2 の大きい方の円はダイクストラ法 (情報なし探索) による状態空間の探索を図示したものである。ダイクストラ法はゴールがどこにあるかということを見捨てて探索を進めているため、初期位置からどの方向へも均等に探索を行う。人間の目で見れば一目で右に探索していけばよいというのは分かる。そのような人間の持っている知識を利用して探索を効率化出来ないだろうか？ A*探索 (A* search) はゴールまでの距離を見積もるヒューリスティック関数 (heuristic function) を用いることでゴールに向かって探索していくことを目指した手法である。

A*探索はヒューリスティック探索の代名詞である、最も広く知られている手法である [31]。A*探索は以下の f 値が最小となるノードを優先したグラフ探索アルゴリズムである。

図 4.3: f_{A^*} 値の意味

$$f_{A^*}(s) = g(s) + h(s) \quad (4.10)$$

ノード s の f 値は、初期状態 s_0 から s を通過してゴール状態 g に辿り着くためのコストの見積もりである (図 4.3)。 g 値は初期状態からノード s までの既知の最短経路コストである。一方 h 値はヒューリスティック関数による s からゴール状態までの最短経路の見積もりである。A*探索は非明示的グラフ探索アルゴリズム (アルゴリズム 2) の一つであり、 f 値が最小ノードから優先して探索を行う 4.10。

定理 4: グラフに経路コストが 0 以下のサイクルが存在しない場合、A* は完全である。

証明: グラフが有限である場合は、やがてすべてのノードを生成・展開し解を発見するので、完全である。グラフが無限である場合もやがて解を発見できる。最適解のコストを c^* とすると、A*が展開するノードは $f(s) \leq c^*$ のノードのみである。本書ではグラフは局所有限であると仮定している (すべてのノードの分枝数は有限である) ので、 $f(s) \leq c^*$ を満たすノードは有限個である。A*探索はやがてこれらのノードをすべて生成・展開し解を発見するので、完全である。

なお、解が存在しない場合、グラフが無限である場合に A*は停止しないので停止性を満たさない。

A*に用いるヒューリスティック関数は正確であるほど良いが、それに加えて許容的、無矛盾であるという性質も有用である。

定理 5: ヒューリスティックが許容的である時、A*は最適解を返す。

証明:

許容的なヒューリスティック $h(n)$ は n からゴールへの経路の下界である。よって、ゴール状態の h 値は 0 である。つまりゴール状態の f 値は g 値と同じである。この解の $g(n')$ 値を f^* と置く (解のコストに相当)。A*のノードの展開順に従うと、 f^* のノードを展開する前に全ての $f < f^*$ のノードが展開される。これらのノードがいずれもゴール状態でなければ、 $g(n) \leq f(n)$ より、 $g(n) < f^*$ となるゴール状態がない。すなわち、 f^* が最適解のコストとなり、 n' がその時のゴール状態である。

A*探索はノードの再展開 (reexpansion) が生じる可能性がある。アルゴリズム 2 にあるように、すでに訪れた状態に再び訪れたとき、前に訪れたときよりも小さな g 値で訪れた場合、ノードを再展開する。このとき、その状態から先の部分木もより小さな g 値で訪れることになるので、部分木をすべて更新しなければならない。そのため、ノードの再展開によってかなり性能が落ちてしまうことがある。無矛盾なヒューリスティックの魅力は A*で再展開が生じないことにある。

定理 6: 無矛盾なヒューリスティックを用いた A*探索はノードの再展開が生じない。

無矛盾なヒューリスティックである場合、A*探索は全てのノード $n \in S$ に最初に最短経路コストで訪れることが証明できる。そのため再展開が生じない。

効率的な A*探索のためには良いヒューリスティック関数をデザインしたい。許容的で無矛盾なヒューリスティックであると解析的に良い性質を満たしていると言える。一方、経験的に速いと知られているヒューリスティック関数の中にはこれらの性質を満たしていないものも多い。

A*に用いるヒューリスティック関数は正確であるほどより探索範囲を狭めることができる。

定義 10 (ヒューリスティックの情報): ヒューリスティック h_1 とヒューリスティック h_2 が両方許容的であり、かつすべてのゴールでない $s \in S$ に対して

$$h_2(s) > h_1(s) \quad (4.11)$$

であるとき、 h_2 は h_1 よりも情報がある (*more informed*) と呼ぶ。

より情報があるヒューリスティックを使う方が A*は効率的である。

定理 7: (ノード展開の必要条件): A*探索で展開されるノードの f 値はすべて最適解のコスト以下である ($f(s) \leq c^*$)。

定理 8: (ノード展開の十分条件): f 値が最適解のコスト未満であるノードは必ずすべて A*探索で展開される ($f(s) < c^*$)。

A*探索は f 値が一番小さいノードから展開していくので、これらの定理が満たされる。

定理 9: h_2 が h_1 よりも情報がある場合、 h_2 を使った A*探索によって展開されるノードはすべて h_1 を使った A*探索でも展開される。

証明: h_2 による A*で展開されるノードの集合はすべて $c^* \geq g(s) + h_2(s)$ である。ゴールを除き $g(s) + h_2(s) > g(s) + h_1(s)$ である。よって、 $c^* > g(s) + h_1(s)$ なのでこれらのノードはすべて h_1 を使った A*探索でも展開される。

よって、A*に使うヒューリスティックは情報があるほど良い。

4.4 ヒューリスティック関数の例

4.2 章にあるように、なるべく正確であり、許容的、無矛盾なヒューリスティックが望ましい。一般に、許容的なヒューリスティックを得る方法としては、元問題の緩和問題を解き、その最適解コストをヒューリスティック値とすることである。ある問題の緩和問題とは、解集合に元の問題の解を含む問題を指す。要するに元の問題より簡単な問題である²。グラフ探索アルゴリズムにおいて緩和問題を作る方法は様々あるが、一つはグラフのエッジを増やすことで緩和が出来る。グラフのエッジを増やすには、問題の可能なアクションを増やすなどの方法がある。

4.4.1 グリッド経路探索：マンハッタン距離

4 方向グリッド経路探索問題の元問題は障害物のあるグリッドに移動することは出来ない。グリッド経路探索で有効なヒューリスティックの一つはマンハッタン距離ヒューリスティックである。これは現在位置とゴール位置のマンハッタン距離を h 値とする。マンハッタン距離の意味としては、障害物を無視した最短経路の距離であるので、グラフのエッジを増やした緩和問題である。このように、問題の性質を理解していれば許容的なヒューリスティック関数を設計することが出来る。8 方向グリッドにおいても斜め方向を加えた距離を考えることで許容的なヒューリスティックとすることが出来る。Any angle グリッドならば直線距離が許容的なヒューリスティックである。

4.4.2 スライディングタイル:マンハッタン距離

スライディングタイルにおけるマンハッタン距離ヒューリスティックは各タイルの現在の位置とゴール状態の位置のマンハッタン距離の総和を h 値とする。スライディングタイル問題において一度に動かせるタイルは 1 つであり、

²解が多いほど簡単であるとは一概には言えないが

その距離は1つである。そのため、マンハッタン距離ヒューリスティックは許容的なヒューリスティックである。

4.4.3 巡回セールスパーソン問題：最小全域木

TSP の解の下界としては最小全域木 (minimum spanning tree) のコストがよく用いられる (図 4.4)。グラフの全域木 (spanning tree) は全てのノードを含むループを含まない部分グラフである。最小全域木は全域木のうち最もエッジコストの総和が小さいものである。未訪問の都市によるグラフの最小全域木は TSP の下界となることが知られている。ヒューリスティック探索ではまだ訪問していない都市をカバーする最小全域木のコストをヒューリスティック関数に用いる。最小全域木は素早く計算ができるので探索には使いやすい。

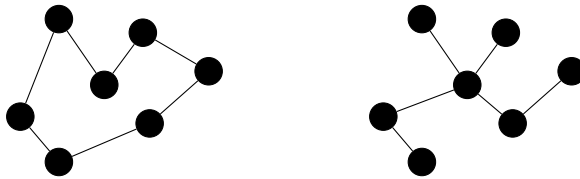


図 4.4: 巡回セールスパーソンにおける TSP 解 (左) と最小全域木 (右)

4.5 非最適解の探索

許容的なヒューリスティックを用いた A*探索は最適解が得られるが、必ずしも最適解がほしいわけではない場合もある。解のクオリティよりもとにかく解が何か欲しい、という場合もある。最適解ではない解を非最適解 (suboptimal solution) と呼び、最適解に限らず解を発見するアルゴリズムを非最適探索 (suboptimal search) と呼ぶ。

4.5.1 重み付き A*探索 (Weighted A*)

重み付き A*探索 (weighted A*) (wA*) は解のクオリティが落ちる代わりにより素早く解にたどり着くための手法である [89]。wA*は重み付き f 値、 f_w が最小のノードを優先して探索する。

$$f_w(n) = g(n) + wh(n) \quad (4.12)$$

定理 10: 許容的なヒューリスティックを用いた重み付き A*探索によって発見される解は最適解のコスト f^* の w 倍以下である。

wA*の利点はA*よりもはるかに高速であることである。多くの場合、 w の大きさに対して指数的な高速化が期待できる。これは深さ d のノードの個数は d に対して指数的 (分枝度を b とすると b^d 個) であることに対応する。

wA*などの非最適探索を使う場合はいずれにせよ最適解が得られないので、許容的でないヒューリスティックと組み合わせて使われることが多い。許容的でないヒューリスティックは許容的なヒューリスティックよりも高速であることが多い。

wA*の解は最適解のコストの上界になるので、A*探索の枝刈りに用いることが出来る。A*探索を実行する前に wA*を走らせ、解の上界 c^* を得、A*探索実行時にノード n に対して f 値が $f(n) \geq c^*$ である場合、そのノードを枝刈りすることができる。このテクニックは多重配列アライメントなどに使われる [50]。

図 4.5a、4.5b はグリッド経路探索問題での A*と wA* ($w = 5$) の比較である。緑のグリッドが初期位置、赤のグリッドがゴール、黒のグリッドは進行不可能の壁である。青のグリッドが探索終了時点で各アルゴリズムによって展開されたノード、薄緑のグリッドが各アルゴリズムに生成され、まだ展開されていないノード (オープンリストにあるノード) である。

wA*の展開・生成ノード数がA*よりも少ない。ただし wA*で発見された解はA*で発見された解よりも遠回りなことがわかる。A*探索は最適解を発

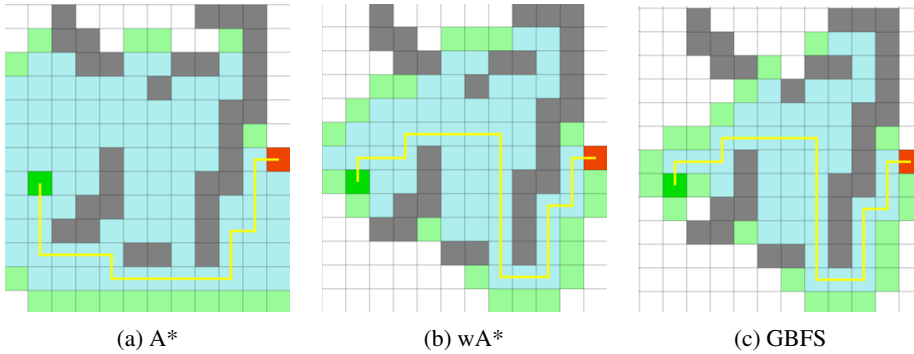


図 4.5: グリッド経路探索問題における A*、wA*、GBFS の比較

見するが、wA*では最初に発見した解が最適とは限らない。

4.5.2 貪欲最良優先探索 (Greedy Best-First Search)

wA*の例で見たように、 g 値に対して h 値に重きを置くことによって解のクオリティを犠牲により高速に解を得ることができる。問題によっては解のクオリティはあまり重要でなかったり、そもそもクオリティという概念がないことがある。このようにとにかく解があればよいという場合は貪欲最良優先探索 (Greedy Best-First Search) が使われることが多い [89]。

$$f_{\text{gbfs}}(s) = h(s) \quad (4.13)$$

貪欲最良優先探索は g 値を無視し、 h 値のみで展開順を決定する。つまり wA*の w を無限大にしたものである。

貪欲最良優先探索は解のクオリティに保証がない。しかし多くの問題で高速に解を発見できるとも強力な手法である。図 4.5c は貪欲最良優先探索である。wA*よりもさらに生成ノード数が少ないことが分かるだろう。

4.5.3 山登り法 (Hill Climbing)

A*探索はクローズドリストに探索済みのノードを記憶しておくことでグラフ全体を見渡して探索を進める。しかし探索済みのノードをメモリにおいておくことは時間がかかるし、メモリも消費する。局所探索 (local search) は現在見つかった最良のノードのみを記憶しておくことで効率的に (最適とは限らない) 解を発見する方法である。

山登り法 (hill climbing) は局所探索アルゴリズムであり、特に組み合わせ最適化問題のためのアルゴリズムとして使われる。山登り法のアルゴリズムは非常に単純である。子ノードのうち最も h 値が小さいノードを選ぶことを繰り返す。

アルゴリズム 4: 山登り法 (Hill Climbing)

Input

: 非明示的状態空間グラフ $(s, Goal, Expand, w)$, 評価関数 h

Output

: s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

1

while not $Goal(s)$ do

2

$u \leftarrow \arg \min_{s' \in Expand(s)} h(s')$;

3

$parentu \leftarrow s$;

4

$s \leftarrow u$;

5

return $Path(s)$

この手法は組み合わせ最適化・連続最適化のためによく使われる。しかしグラフ探索アルゴリズムでこのまま使おうとすると評価関数の極小ではまっ

てしまったり、デッドエンドに落ちてしまう可能性がある。

4.5.4 強制山登り法 (Enforced Hill Climbing)

ヒューリスティック探索でよく使われる局所探索は強制山登り法 (enforced hill-climbing) (EHC) である [46]。EHC は幅優先探索を繰り返し、現在のノードよりも h 値が小さいノードを発見する。発見できればそのノードから再び幅優先を行い、ゴールを発見するまで続ける。もし h 値が小さいノードが見つからなければ極小に陥ってしまったということであるので、探索を打ち切り、失敗 (Fail) を返す。

アルゴリズム 5: 強制山登り法 (Enforced Hill Climbing)

Input : 非明示的状態空間グラフ $(s, Goal, Expand, w)$, 評価関数 h

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1 while not  $Goal(s)$  do
2    $T \leftarrow \{v \in Expand(s)\};$ 
3    $T' \leftarrow \{v \in T | h(v) < h(s)\};$ 
4   while  $T' = \emptyset$  do
5      $T \leftarrow \{v \in Expand(u) | u \in T\};$ 
6      $T' \leftarrow \{v \in T | h(v) < h(s)\};$ 
7    $u \leftarrow \arg \min_{s' \in Expand(s)} h(s');$ 
8    $parent(u) \leftarrow s;$ 
9    $s \leftarrow u;$ 
10 return  $Path(s)$ 

```

ほとんどの局所探索アルゴリズムは完全性を満たさない。なので解が発見できなければ (失敗を返す場合) 続けて A*探索などを走らせないと解が発見できない場合がある。山登り法などの局所探索の利点はなんといってもアルゴリズムはシンプルであり、実装が簡単であることである。また、オープンリストなどのデータ構造を保持する必要がないので消費メモリ量が非常に

少ない。

山登り法が記憶しなければならないノードは現在のノードとその子ノードらだけである。毎回次のノードを選択したら、そのノード以外のノードは捨ててしまう。貪欲最良優先探索 (4.5.2 節) も同様に h 値のみしか見ていないが、すべての生成済みノードをオープンリストに保存して評価値を確認していることと比較すると、山登り法がいかにアグレッシブな手法であるかがうかがえるだろう。

4.6 上手く行かない場合

ヒューリスティック探索が思ったより遅い場合考えるべきことはいくつかある。

まず考えるべきは、そもそもヒューリスティック探索によって解くべき問題なのかという点である。制約プログラミングや他の組合せ最適化手法なら効率的な問題かもしれない。似たような問題が他の手法によって解かれていないかを調べてみると良い。

ヒューリスティック探索が良さそうと思ったら、試しに wA^* にしてみても、weight を大きく取ってみよう。おおまかに言うと wA^* の実行速度は w の大きさに対して指数的に速くなる。 $w = 1$ だと全く解けそうにない問題でも $w = 2$ にすると一瞬で解けることがある。もし w を 10 くらいにしてもすぐに解が発見できなければ、おそらくその問題で最適解を発見するのは非常に難しい。このような問題に対しては最適解を見つけることは諦め、非最適解探索 (節 4.5) を使ったほうが良いだろう。 wA^* でも解が見つけれない場合はそもそも解がない問題かもしれない。解がない場合は制約プログラミングのソルバーを使うと素早く検出できることが多い。

wA^* で非最適解を見つけられる場合なら、工夫すれば A^* で最適解を見つけることができるかもしれない。その場合まず、おおまかな探索空間の大きさを見積もり、そして 1 秒間に何ノードが展開されているかを確認すると良い。ヒューリスティック探索の実行時間はおおまかには (展開するノード数) と (1 ノードの展開にかかる時間) の積である。

もし秒間に展開されているノードの数が少ない場合、プロファイラを使ってどのオペレーションに時間がかかっているかを確認するべきだろう。多くの場合実行時間の大半をオープンリスト、クローズドリスト、ノードの展開関数(現在の状態とアクションから状態を計算する関数)のいずれかが占めている、ということがわかるだろう。

オープンリスト、クローズドリストの処理に時間がかかっている場合はデータ構造の改良によって問題が解決するかもしれない。効率的なデータ構造の設計については5章を参照されたい。特に探索が進みデータ構造が大きくなるほどアクセスするためにかかる時間がどんどん増えていく場合はデータ構造の工夫によって解決できる可能性がある。あとはプロセスのメモリ消費量を確認してみよう。A*はメモリ消費も激しい。実行時間が遅いのは、メモリを使いすぎていてメモリスワップが起きているからかもしれない。より大きなメモリを使うか、IDA*などのメモリの消費量の少ない手法を試してみると良いかもしれない。

ノードの展開関数に時間がかかる場合はその実装を工夫するか、ノードの展開回数を最小限に抑えるしかない。ノードの展開回数を減らす方法はないかなかなか難しいが、例えばヒューリスティック関数の改良によって可能である。ほんの少しのヒューリスティック関数の改善によって探索の効率は劇的に上がりうる。

これらの方法でもまだ解けない場合、扱っている問題は純粋なA*で解くには結構難しい問題かもしれない。1968年にA*が初めて提案されてから、様々な発展手法が考案されてきた。本書の後半で紹介する最新の手法を試してみれば解けるかもしれない。

4.7 関連文献

最良優先探索 (best-first search) は2種類の定義があることに注意したい。本書では最良優先探索をA*を含む、何らかの有望さの指標に従ってグラフを探索するアルゴリズムを指す。一方、ゴールに最も近いと考えられるノード

を常に最優先して展開するアルゴリズムを特に最良優先探索と呼ぶことも多い。本書ではこの方法は貪欲最良優先探索と呼んでいる。

同じ問題に対して重み付き A* を繰り返し w 値をだんだんと小さくしていくことでだんだんと解のクオリティを向上させることができる。この方法を Anytime A* と呼ぶ [64]³。

本書で扱わない重要な探索アルゴリズムとして遺伝的アルゴリズムがある [38]。遺伝的アルゴリズムはさまざまな最適化問題に適用できる。グラフ探索問題で使われることは少ないが、グラフ探索問題ではなかなか解けない難しい問題が遺伝的アルゴリズムによって簡単に解けることがある。例えば N-Queen 問題はチェスの盤面上にクイーンを互いに移動可能範囲がぶつからないように配置する問題であるが、これはグラフ探索の手法だとなかなか解くのが難しいが遺伝的アルゴリズムだと非常に効率的に解けることが知られている。

³一般にいつ停止しても現時点で見つけた中で一番良い (最適とは限らない) 解を返し、時間をかけるほど解のクオリティを改善していくアルゴリズムを anytime アルゴリズムと呼ぶ。

グラフ探索のためのデータ構造

ヒューリスティック探索の効率は探索効率、つまり展開したノードの数によって測られる場合が多い。本書の多くの節は探索効率を上げるためのアルゴリズムについて解説している。しかしヒューリスティック探索の実行時間とメモリ量はデータ構造の実装にも大きく左右される。

ヒューリスティック探索ではオープンリストとクローズドリストの2つのデータ構造を保持する。オープンリストに必要なインターフェイスは必要な操作はエンキュー (enqueue, pop) とデキュー (dequeue, push) である。クローズドリストに必要なインターフェイスは挿入 (insert) と重複検知のための検索 (find, lookup) である。

これらのデータ構造をどのように実装するかは探索の効率に大きな影響を与える。歴史的な経緯からオープン・クローズドリストと呼ばれているが、リストとして実装するのは非効率的である。

これらを実装するための効率的なデータ構造はアルゴリズムと問題ドメインに依存する。この章ではどのようなシチュエーションでどのようなデータ構造を使われるかを説明する。この章は実践的に非常に重要な章である。残念ながらヒューリスティック探索の研究論文のほとんどはこの章で扱われ

る内容について自明のものとして扱わない。あるいはこれらの内容を「コードの最適化」として論文中には明示しない。が、その実自明ではないので初学者の多くはここで苦労することになる。データ構造について議論を行っている論文としては [16] がある。

5.1 オープンリスト (Open List)

オープンリストのプライオリティキューの実装方法は様々ある。 f 値が連続値 (e.g. 実数) である場合はヒープを使うことが多い。 f 値の取りうる値が有限個であり、 f 値が同じノードが沢山ある場合はバケットで実装することが出来る。また、そのような場合は f 値が同じノードのうちどのノードを選ぶかのタイブレーキング (tiebreaking) も重要になってくる。

5.1.1 データ構造の選択

オープンリストは探索の中で沢山エンキューとデキューを行うため、これらの操作の計算時間が速いものを選択したい。オープンリストのシンプルな実装方法としては二分ヒープがあげられる。二分ヒープはほとんどの言語の標準ライブラリにあるため実装が簡単である。探索アルゴリズムはノードの数が膨大になることが多い。計算量のオーダーで見ると単純な二分ヒープよりもフィボナッチヒープ (Fibonacci heap) が効率が良いが、定数項が大きいいため多くの場合二分ヒープが採用される。二分ヒープよりも効率が良いデータ構造として基数ヒープ (radix heap) がある [2]。基数ヒープはエンキューできる数値は最後に取り出した数値以上のものであるという制約がある。A*探索はヒューリスティックが単調である場合この制約を満たす。これらのヒープの実装や性質については Introduction to Algorithms を参照されたい [19]。

プライオリティ値が取る値が有限個である場合はバケットで実装をすることが出来る [16]。プライオリティ値が取る値が $C_{min} \leq i \leq C_{max}$ の整数値であるとする。バケットは長さ $C_{max} - C_{min} + 1$ の配列からなる。プライ

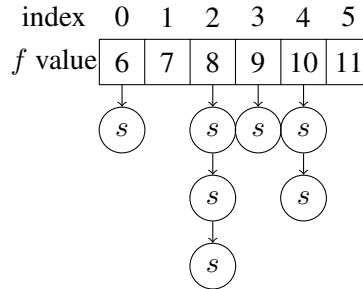


図 5.1: バケットによるオープンリストの実装 ($C_{min} = 6, C_{max} = 11$)

オリティ値が i の要素は配列の $i - C_{min}$ 番目の位置にあるリストに追加される。バケットはエンキューもデキューも $O(C_{max} - C_{min})$ で計算でき、保持しているノードの数に依存しない。そのため二分ヒープよりも高速であることが多い。後述するタイブレーキングを行う場合はバケットの中に更に二段階目のバケットを用意することもできる [16]。

5.1.2 タイブレーキング (Tiebreaking)

3 章、4 章ではオープンリストでどのノードを最初に展開するかによってアルゴリズムの性能が大きく変わることを示してきた。例えば A* 探索では f 値が最小のノードを優先して展開する (アルゴリズム 4.10)。だが、 f 値が最小のノードは複数ある場合がある。コスト関数が離散値である場合は f 値が同じノードが大量にあることが多い。このような場合、同じ f 値のノードの中からどのノードを選ぶかを決定することをタイブレーキング (tie-breaking) と呼ぶ。

A* 探索で広く使われるタイブレーキング戦略は h 値が小さいノードを優先させる方法である。 $\arg \min_{u' \in Open} f(u')$ を選択する代わりに以下の条件を満たす u を選択する。

$$u = \arg \min_{v \in B} h(v)$$

$$s.t. \ B = \{v | v \in Open, f(v) = \min_{v' \in Open} f(v')\}$$

h 値が小さいノードを優先させる理由としては、 h 値がゴールへの距離の推定だからである。なのでゴールに近いノードから展開したほうがゴールにたどり着くのが早いはずだ、というヒューリスティックである。

もう一つは Last-in-first-out (LIFO) タイブレーキングがよく使われる。LIFO は最後にオープンリストに入ったノードから優先して展開する。LIFO を使うメリットはオープンリストをバケット実装している場合に配列の一番後ろのノードが LIFO で得られるノードであることである。5.1.1 節にあるようにバケットは配列で実装されることが多いが、配列の末尾のノードを取り出すには定数時間しかかからない。なので自然にバケットを実装すると LIFO になる。

タイブレーキングは長い間ヒューリスティック探索研究の中であまり重要視されておらず、よいヒューリスティック関数をデザインすることと比較してアルゴリズムの効率に対してあまり影響を及ぼさないと考えられてきた。近年の研究でタイブレーキングが重要なファクターになる場合があるということが実験的に示された [8]。

5.2 クローズドリスト (Closed List)

重複検出は無駄な展開を防ぐために不可欠な操作である。非明示的状態空間グラフではあらかじめすべての状態を知ることが出来ないので、探索中に発見された状態の集合を保持しておくための動的な辞書が必要になる。

クローズドリストは生成済み・展開済みノードを保持し、新しくノードが生成されたときにすでにその中にあるかどうかを確かめる。一般的に辞書

のデータ構造は挿入 (insert)、検索 (lookup)、削除 (delete) をサポートするが、探索アルゴリズムに必要なのは挿入と検索である。

クローズドリストの実装にはハッシュテーブル (hash table) が用いられることが多い。ハッシュテーブルは探索で一番頻繁に使われる検索が定数時間で出来るためである。

5.2.1 ハッシュテーブル (Hash Table)

集合 $U = \{0, 1, \dots, N - 1\}$ をキーの候補とする辞書 (dictionary) は、保持されたキーの集合 $R \subseteq U$ から連想情報の集合 I への部分写像である。グラフ探索において状態 $s \in S$ を一意に表すためのキーを $k(s)$ とする ($k : S \rightarrow U$)。キーは更にハッシュテーブルと呼ばれる配列 $T[0, \dots, m - 1]$ に写される。この写像 $h : U \rightarrow \{0, \dots, m - 1\}$ をハッシュ関数 (hash function) と呼ぶ。簡単のためここでは状態から配列のインデックスまでの関数 $S \rightarrow \{0, \dots, m - 1\}$ を h を使って表す。

多くの場合 $N > m$ なので、このハッシュ関数は単射ではない。よって異なるキーがハッシュテーブル上の同じ位置に配置されることになる。これをハッシュの衝突 (hash collision) と呼ぶ。ハッシュテーブルへのアクセスにかかる時間は 1. ハッシュ値の計算、2. 衝突時の処理方法、3. 保持されたキーの数とハッシュテーブルの大きさの比に依存する。

ハッシュ関数の選択はハッシュテーブルの性能に大きく影響を与える。最悪の場合、すべてのキーがテーブルの同じ位置に移される。最良の場合、一度も衝突が起こらず、アクセス時間は定数になる。最悪の場合の計算時間は理論的には興味のある話題だが、多くの場合正しくハッシュを計算することで避けられる。

衝突したハッシュの処理方法としては連鎖法 (chaining) と開番地法 (open addressing) がある。連鎖法はハッシュテーブルの各位置にリストが置かれ、その位置に挿入された連想情報はそのリストに順次追加されていく。同じ位置に n 個の状態が挿入された場合リストの長さは n になる。このとき、この位置への検索の速度は $O(n)$ となる。開番地法は衝突した場合テーブル中

の空いている別の位置を探し、そちらに入れる方法である。探索アルゴリズムにおいては連鎖法が使われることが多い。

5.2.2 ハッシュ関数 (Hash Function)

良いハッシュ関数に求められる要件は主に 2 つである。一つはハッシュ値が値域内なるべく均等に分散してほしい。もう一つはハッシュの計算に時間がかからない方がよい。

グラフ探索のためのハッシュ関数をデザインするために考えなければならないのは、探索中に現れる状態集合、探索空間は状態空間全体のほんの一部であり、かつ偏りのある集合であるということである。ハッシュ関数はその探索空間内でハッシュ値が十分に均等であってほしい。

5.2.2.1 剰余法 (Remainder Method)

$k(s)$ を整数と考えることが出来るならば、それを m で割った余りをハッシュ値として使える。

$$h(s) := k(s) \bmod m \quad (5.1)$$

このハッシュ関数は S から T への写像になる ($h : S \rightarrow \{0, \dots, m-1\}$)。 T へ均等に分配するためには法 m の選択が重要になる。例えば m に 2 の倍数を選んでしまうと、 $h(s)$ が偶数であるのは $k(s)$ が偶数であるときであり、そのときのみである (iff)。 m が任意の数の倍数であると同様の不均一が起きってしまうので、 m は素数を選ぶと良い。

5.2.2.2 積算ハッシュ法 (Multiplicative Hashing)

積算ハッシュ法はキー $k(s)$ と無理数 ϕ の積の小数部分を使ったハッシュである。

$$h(s) = \lfloor m(k(s)\phi - \lfloor k(s)\phi \rfloor) \rfloor \quad (5.2)$$

ϕ はパラメータであり、 $(\sqrt{5} - 1)/2$ の黄金比を使うと良いとされている。

5.2.2.3 ハッシュの性質

関連して、クローズドリストにとって良い性質のハッシュがいくつかある。

グラフ探索では状態遷移は状態の一部分のみに作用し、状態変数のほとんどは変わらないということが多い。このとき、新しい状態のハッシュ値を一から計算するのではなく、元の状態のハッシュ値と状態遷移による差分を利用して新しい状態のハッシュ値を求めることが出来る場合がある。この方法を差分ハッシュ (incremental hashing) と呼ぶ。差分ハッシュはハッシュ値の計算時間を短くできることがある。

どのような操作があっても衝突が起らないようなハッシュを完全ハッシュ (perfect hash function) と呼ぶ。もし $n = m$ であれば最小完全ハッシュと呼ぶ。完全ハッシュがあればどのような操作も $O(1)$ で行うことができる。問題によって完全ハッシュは簡単に計算することが出来る。例えばスライディングタイルパズルであれば状態 $s = (v_0, v_1, \dots, v_8)$ に対して $h(s) = \sum_{i=0..8} v_i \cdot 9^i$ と単純に状態変数の列を使って計算することが出来る。ただし長さ 9^9 の配列は大きすぎるので工夫が必要になる。完全ハッシュを作るのは簡単だが、小さい完全ハッシュを作るのは難しい。スライディングタイルパズルは辞書順 (lexicographical order) によって最小完全ハッシュを作ることが出来る [61]。

5.2.3 トランスポジションテーブル (Transposition Table)

木探索の問題点は重複検出を行わないため、同じ状態を複数回展開し、時間がかかってしまうことである。一方グラフ探索の問題点は重複検出を行うため、生成したノードの数だけメモリを消費することにある。グラフが大きい場合はやがてメモリを使い果たし、探索が続けられなくなってしまう。しかしながら、実はグラフ探索において重複検出は必ずしも正しくなければならないものではない。重複してノードを生成しても問題なくアルゴリズムは動作する。仮にクローズドリストの一部を捨ててもアルゴリズムの正しさを損

なうものではない。

トランスポジションテーブル (transposition table) はこのアイデアに基づき、展開したノードの一部のみを保存することで重複検出をしつつメモリの消費を抑える手法である。トランスポジションテーブルはグラフ探索におけるクローズドリストと同様、辞書である。クローズドリストがすべての生成済みノードを保持していることを保証するのに対して、トランスポジションテーブルはそのような保証をしない。トランスポジションテーブルはどのノードを保持し、どのノードを捨てるかを賢く選択しなければならない。どのノードを保持するべきかという戦略はいくつか提案されている [13, 3]。

単純なものとしては、テーブルが一杯になるまではすべてのノードをテーブルに追加し、テーブルが一杯になったらすべてのノードを捨てる戦略がある。Stochastic node caching [70] は、新しく追加されたノードを確率 $1 - p$ で捨てるというものである。この戦略だと n 回訪れたノードは確率 $1 - (1 - p)^n$ でテーブルに追加されるため、たくさん訪れたノードほど重複検出されやすいことになるという点で合理的である。2 人ゲームでよく使われる戦略は、2 つのノードのハッシュ値が衝突したらどちらか一方のみを残すという戦略である。より d 値が大きい (深い) ノードを残す戦略が一般的である。この戦略はハッシュテーブルの衝突を減らせるという点で一挙両得になる。

トランスポジションテーブルは A* だけでなく後述する IDA* (節 6.2) でも使われる [76]。IDA* の空間量は最適解の深さに対して線形なので消費メモリが非常に少ない。あまったメモリを効率よく使うために IDA* にトランスポジションテーブルを用いるという研究が多い [76, 3, 58]。

トランスポジションテーブルは元々 2 人ゲームの分野から発展した技術である。同じ手を異なる順番で行うことで同じ状態に到達することを検出するために使われていたので transposition という名前になっている。

5.2.4 遅延重複検出 (Delayed Duplicate Detection)

3.2 節ではノードを生成したタイミングで重複検出を行うアルゴリズムを説明した。この方法だとノードが生成されたその瞬間に検出を行うという意味で

即時重複検出 (immediate duplicate detection) と呼ぶことがある。それに対して、ノードを展開するタイミングで検出を行う遅延重複検出 (delayed duplicate detection) という方法もある [?]. ノードが生成された瞬間に重複検出を行わない場合、オープンリスト内に同じ状態を持ったノードが重複して存在する場合がある。しかしノードを展開するときに重複検出を行うので、クローズドリストにノードの重複はない。アルゴリズム 6 は遅延重複検出を用いる場合のグラフ探索アルゴリズムの疑似コードである。

遅延重複検出は展開ノード毎に重複検出の回数を減らすことができるというメリットがある。一方、オープンリスト内に重複が存在する可能性があるため、プライオリティの選択次第ではノードの再展開が増えてしまうデメリットがある。そのため遅延重複検出はノードの重複が少ない問題や、重複検出に時間がかかる場合に使われる。特にクローズドリストをハードディスクなどの外部メモリに置く外部メモリ探索に相性が良い。

5.2.5 関連文献

TODO: Rabin and Karp hashing

TODO: Universal hash function

アルゴリズム 6: 遅延重複検出を用いたグラフ探索

Input : 非明示的状态空間グラフ $(s, Goal, Expand, w)$, プライオリティ関数 f

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $Closed \leftarrow \emptyset, Open \leftarrow \{s\}, d(s) \leftarrow 0, g(s) \leftarrow 0;$ 
2 while  $Open \neq \emptyset$  do
3    $u \leftarrow \arg \min_{u' \in Open} f(u');$ 
4    $Open \leftarrow Open \setminus \{u\};$ 
5   if  $Goal(u)$  then
6     return  $Path(u);$ 
7    $s \leftarrow parent(u);$ 
8   if  $u \notin Closed$  or  $g(s) + w(s, u) < g(u)$  then
9      $d(u) \leftarrow d(s) + 1;$ 
10     $d(u) \leftarrow g(s) + w(s, u);$ 
11     $parent(u) \leftarrow s;$ 
12    for each  $v \in Expand(u)$  do
13       $Open \leftarrow Open \cup \{v\};$ 
14       $parent(v) \leftarrow u;$ 
15  if  $u \notin Closed$  then
16     $Closed \leftarrow Closed \cup \{u\};$ 
17 return  $\emptyset;$ 

```

時間・空間制限下のヒューリスティック探索

A*探索などのヒューリスティック探索は時間と空間の両方がボトルネックとなりうる。すなわち、A*はノードを一つずつ展開していかなければならないので、その数だけ Expand を実行しなければならない。また、A*は重複検出のために展開済みノードをすべてクローズドリストに保存する。なので、必要な空間も展開ノード数に応じて増えていく。

残念ながら、ほぼ正しいコストを返すヒューリスティック関数を使っても、A*が展開するノードの数は指数的に増加することが知られている [45]。

そのため、ヒューリスティックの改善のみならず、アルゴリズム自体の工夫をしなければならない。この章では時間・空間制約がある場合の A*の代わりとなるヒューリスティック探索の発展を紹介する。これらのアルゴリズムはメリット・デメリットがあり、問題・計算機環境によって有効な手法が異なる。よって、A*を完全に取って代わるものは一つもないと言える。

分枝限定法 (branch-and-bound) は非最適解を見つけやすい問題で特に有効な手法であり、メモリ消費が小さいというメリットがある。反復深化 A* (iterative deepening A*) は線形メモリアルゴリズムであり、メモリが足りない問題で使われる。両方向探索 (bidirectional search) は初期状態とゴール状態の両方から探索をはじめ、二方向の探索が交わる場所を探す方法である。

ヒューリスティック関数の精度が悪いときに有用である場合が多いことが知られている。シンボリック探索 (symbolic search) は二分決定グラフ (Binary Decision Diagram) を用いて状態集合を表すことによってノードの集合に対して一度に演算を行う手法である。新奇性に基づく枝刈りは目新しくない状態は枝刈りする。並列探索 (parallel search) は複数のコアや計算機を使うことでメモリと計算時間の両方の問題を解決する。

6.1 分枝限定法 (Branch-and-Bound)

分枝限定法 (branch-and-bound) は非最適解が簡単に見つけられるが最適解を発見するのは難しい問題、例えば巡回セールスパーソン問題などに使われることが多い。分枝限定法は広くコンピュータサイエンスで使われる汎用的な考え方である。アイデアとしては問題を複数のサブ問題に分割 (branch) し、これまでに得られた解よりも悪い解しか得られないサブ問題を枝刈りする (bound)、というアイデアである。特にメモリ効率の良い深さ優先分枝限定法 (Depth-First Branch-and-Bound) が探索分野ではよく使われる。分枝限定法の処理は一般的な木探索に加えて枝刈りが行う (アルゴリズム 7)。

アルゴリズム 7: 深さ優先分枝限定法 (Branch-and-Bound)	
Input	: 非明示的状态空間グラフ $(s, Goal, Expand, w)$ 、ヒューリスティック関数 h
Output	: s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset
1 $U \leftarrow \infty$;	
2 $path \leftarrow \emptyset$;	
3 $(U, path) \leftarrow DFB\&B(s, 0, U, path)$;	
4 return $path$;	

一般に、アルゴリズムの実行に従って最適解とは限らない解を次々と発見

アルゴリズム 8: DFB&B(s, g, U, path): 分枝限定法の再帰計算

```

1 if  $\text{Goal}(s)$  then
2   if  $g < U$  then
3      $U \leftarrow g;$ 
4      $\text{path} \leftarrow \text{Path}(s);$ 
5 else
6    $\text{Succ}(s) \leftarrow \text{Expand}(s)$ , sorted according to  $h$ ;
7   for each  $s' \in \text{Succ}(s)$  do
8     if  $g + h(s') < U$  then
9        $(U, \text{path}) \leftarrow \text{DFB\&B}(v, g + w(s, s'), U, \text{path});$ 
10 return  $(U, \text{path})$ 

```

する手法において分枝限定法の考え方が採用できる。すなわち現在発見された中でもっとも良い解 (incumbent solution) を用いて探索範囲を限定していくアイデアである。A*探索と比較して分枝限定法の利点は、任意時間アルゴリズム (anytime algorithm) であることである。任意時間アルゴリズムとはプログラムを適当なタイミングで停止しても解を返すアルゴリズムを指す。A*探索は最適解を発見するまで他の非最適解を発見することはない。一方、分枝限定法はすぐに何かしらの解を見つけることができる。そして探索の過程で現在の解よりも良い解を発見し、だんだんと解のクオリティを上げていき、最終的に最適解を発見する。そのため、どのくらい長い間探索に時間をかけてよいかが分からない場合、任意時間アルゴリズムは理想的である。関連して、A*のそのような問題を解決した任意時間A* (Anytime A*) というアルゴリズムもある [64, 40, 77]。例えば単純に wA^* において w の値を大きなものからだんだんと1に近づける方法がシンプルで効率的である。A*や任意時間A*と比較した分枝限定法のもう一つの利点はオープンリストなどのデータ構造を必要としない深さ優先探索であることである。そのためメモリ・キャッシュ効率が良い。一方、重複検出を行わない木探索であるため、ノードの重

複が多いドメインでは性能が悪いことが多い。

6.2 反復深化深さ優先探索 (Depth First Iterative Deepening)

A*探索は時間・空間の両方がボトルネックになるが、現代の計算機環境では多くの場合空間制約がよりネックになる。これは A*が重複検出のために展開済みノードをすべてクローズドリストに保存していることに起因する。

3.2 節で述べたように、重複検出は正しい解を返すためには必須ではない。グラフに対して木探索を行うことも出来る。しかしながら、単純な幅優先木探索・深さ優先木探索はパフォーマンスの問題がある。

反復深化深さ優先 (depth first iterative deepening) (DFID) は深さ優先探索もメモリ効率と幅優先探索の効率性を併せ持った賢いアルゴリズムである [59, 78]。アイデアとしては、閾値 *cost* を 1 ずつ大きくしながら、繰り返しコスト制限付き深さ優先 (CLDFS) を実行する (アルゴリズム 9)。CLDFS が解を見つければその解を返して停止し、見つけれなければ *cost* を 1 つ大きくしてもう一度 CLDFS を実行する。

DFID は閾値を大きくする度に一つ前のイテレーションで展開・生成したノードをすべて展開・生成しなおさなければならない。各イテレーション内でもクローズドリストを保持していないために重複検出が出来ない。なので、アルゴリズム全体を通して大量の重複ノードが出る可能性がある。一見非常に効率が悪いように思えるかもしれないが、実はあまり損をしないことが多い。分枝数を b 、最適解のコストを c^* とする。DFID は c^* 回目のイテレーションで生成するノードの数はおよそ $1 + b + b^2 + \dots + b^{c^*} = O(b^{c^*})$ である。 $c^* - 1$ 回目のイテレーションで生成されるノードの数はその数のおよそ $1/b$ 倍である。 $c^* - 2$ 回目のイテレーションではその更に $1/b$ 倍と、指数的に生成ノード数は減っていく。そのため、DFID 全体でノードを生成する回数は、 c^* 回目のイテレーションでノードを生成する回数とあまり変わらない。

DFID のメリットはいくつかある。まず、コスト w が 0 となるアクションが存在しない場合、必要なメモリ量が最適解のコストに対して線形である ($O(bc^*)$)。そのため、幅優先ではメモリが足りなくなって解けないような難しい問題でも DFID なら解ける可能性がある。

メモリ量と関連してもう一つの重要なメリットはキャッシュ効率である。上述のように DFID は必要なメモリ量が非常にすくない。また、メモリアクセスパターンもかなりリニアである。そのため、ほぼキャッシュミスなく探索を行えるドメインも多い。例えば、スライディングタイルパズルなどの状態が少ないビット数で表せられるドメインでは特にキャッシュ効率が良く、1 ノードの展開速度の差は圧倒的に速い [59]。

DFID は解を返す場合、得られた解が最適解であることを保証する。DFID をはじめとする重複検出のないアルゴリズムを用いる際の問題は、解がない場合に停止性を満たさないことである。問題に解がなく、グラフにループがある場合、単純な木探索は停止しない。よって、この手法は解が間違いなく存在することが分かっている問題に対して適用される。あるいは、解が存在することを判定してから用いる。例えばスライディングタイプパズルは解が存在するか非常に高速に判定することが出来る。

アルゴリズム 9: 反復深化深さ優先 (Depth First Iterative Deepening)

Input : 非明示的状態空間グラフ $(s, Goal, Expand, w)$

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $U' \leftarrow 0$ ;
2  $path \leftarrow \emptyset$ ;
3 while  $path = \emptyset$  and  $U' \neq \infty$  do
4    $U \leftarrow U'$ ;
5    $(U', path) \leftarrow CLDFS - DFID(s, 0, U)$ ;
6 return  $path$ ;
```

アルゴリズム 10: CLDFS-DFID: DFID のためのコスト制限付き深さ優先

Input : 状態 s 、経路コスト g 、閾値 U

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $U' \leftarrow \infty$ ;
2 if  $Goal(s)$  then
3   | return ( $, Path(s)$ );
4 for each  $u \in Expand(s)$  do
5   | if  $g + w(s, u) \leq U$  then
6     |  $p \leftarrow CLDFS(u, g + w(s, u), U)$ ;
7     | if  $p \neq \emptyset$  then
8       | | return ( $s, p$ );
9   | else if  $g + w(s, u) < U'$  then
10    | |  $U' \leftarrow g + w(s, u)$ ;
11 return ( $U', \emptyset$ );

```

6.2.1 反復深化 A* (Iterative Deepening A*)

反復深化 A* (IDA*) は木探索に対してヒューリスティックを用いた、非常にメモリ効率の良いアルゴリズムである [59]。DFID と同様、コストを大きくしながら繰り返し CLDFS を呼ぶ。ただし、DFID では g 値によってコストを制限していたのに対して、IDA* では f 値によってコストを制限する。 f 値によって制限することによってヒューリスティック関数を用いることができる。アルゴリズム 12 は IDA* での CLDFS である。コストが f 値で制限されていること以外アルゴリズム 10 と同一である。

IDA* は深さ優先探索を繰り返すので消費メモリが非常に少ない。なので A* ではメモリが足りなくなって解けないような難しい問題でも IDA* なら解ける可能性がある。DFID と同様にキャッシュ効率も非常に良い場合がある。例えばスライディングタイルパズルでは IDA* のほうが A* よりも速く解を見つけることができることが知られている [59]。何度も何度も重複して同じノードを展開しているのにも関わらずである。

DFID 同様、IDA* は解を返す場合、得られた解が最適解であることを保証する。IDA* も解がない場合に停止性を満たさない。

アルゴリズム 11: 反復深化 A* (Iterative Deepening A*)

Input : 非明示的状态空間グラフ $(s, Goal, Expand, w)$ 、ヒューリスティック h

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $U' \leftarrow h(s);$ 
2  $path \leftarrow \emptyset;$ 
3 while  $path = \emptyset$  and  $U' \neq \infty$  do
4    $U \leftarrow U';$ 
5    $(U', path) \leftarrow CLDFS - IDA(s, 0, U);$ 
6 return  $path;$ 
```

アルゴリズム 12: CLDFS-IDA: IDA*のためのコスト制限付き深さ優先

Input : 状態 s 、経路コスト g 、閾値 U
Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $U' \leftarrow \infty$ ;
2 if  $Goal(s)$  then
3   | return ( $Path(s)$ );
4 for each  $u \in Expand(s)$  do
5   | if  $g + w(s, u) + h(u) > U$  then
6   |   | if  $g + w(s, u) + h(u) < U$  then
7   |     |  $U' \leftarrow g + w(s, u) + h(u)$ ;
8   | else
9   |   |  $p \leftarrow CLDFS - IDA(u, g + w(s, u), U)$ ;
10  |     | if  $p \neq \emptyset$  then
11  |       | return ( $s, p$ );
12 return ( $U', \emptyset$ );

```

6.3 両方向探索 (Bidirectional Search)

両方向探索 (bidirectional search) は初期状態とゴール状態の両方から探索をする手法である。初期状態からゴールを目指して探索する方向を正方向探索、ゴール状態から初期状態を目指して探索する方向を逆方向探索と呼ぶ。

両方向探索のメリットは探索の深さが正方向探索のみの手法と比較して半分になることである。最適解の深さが d であると仮定すると、分枝数を b とすると正方向探索のみでは $O(b^d)$ 個程度のノードを展開しなければならない。一方、両方向探索を行った場合、正・逆方向の探索の深さが $d/2$ になった時点で二つの探索がつながり、ゴールへの経路が発見できる。このとき展開しなければならないノードの数は $O(b^{d/2}) * 2$ であるので、正方向探索よりも非常に小さい。直感としては、両方向探索をすると半径が半分の円が2つできるので、面積としては正方向探索よりも小さくなると考えると良い (図 6.1)。

そのため、両方向探索はヒューリスティック関数が不正確な時には正方向探索よりも効率的であることが多いこと [10]。一方、ヒューリスティック関数が正確である場合は正方向探索のみの方が効率的な場合が多い。

両方向探索の一番のボトルネックは重複検出である。すなわち、正方向探索で生成されたノードの集合と逆方向探索で生成されたノードの集合に共通部分があるかどうかを確認しなければならない。このため生成ノード数としては両方向探索は少なくとも、find 命令の実行回数が片方向探索よりもかなり多くなる。

両方向探索は長らく最適解の保証が難しかったが、正方向探索と逆方向探索がちょうど真ん中で重なり (meet-in-the-middle)、最初に発見された解が最適解であることを保証する方法が 2016 年に提案された [48]。

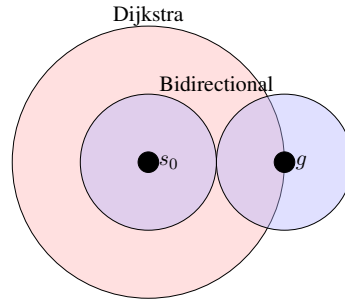


図 6.1: 両方向探索

6.4 外部メモリ探索 (External Search)

グラフ探索は重複検出のために今までに展開したノードをすべて保持しなければならない。よって、保持できるノードの量によって解ける問題が決まってくる。探索空間があまりに大きすぎると、ノードが多すぎてメモリに乗り切らないということが起きる。

外部メモリ探索 (External Search) は外部記憶、HDD や SSD を用いることでこの問題を解決する [17]。すなわち、オープンリストとクローズドリストの一部を外部記憶に保持し、必要に応じて参照しメモリに持ってくる、ということをする。外部メモリ探索のミソは、外部記憶へのアクセス回数をどのように減らすかにある。表 6.1 は一般的なコンピュータのキャッシュ・メモリ・ハードディスクへのアクセスレイテンシーを比較した表である¹。メモリから 1MB 逐次に読みだすオペレーションは 250,000 nanosec かかるが、ハードディスクからの読出しは 20,000,000 nanosec もかかる。更にハードディスクにランダムアクセスする場合 (Disk seek) は 8,000,000 nanosec もかかる。よって、HDD は工夫して使わなければ実行時間が非常に遅くなってしまう。

¹表は (<https://gist.github.com/jboner/2841832>) より。

表 6.1: 一般的なハードウェアのアクセス速度。メモリへのアクセス速度に対して外部記憶のアクセスは遅い。加えて、ランダムアクセスは seek の時間がかかるためさらに遅くなる。

	nano sec
命令実行	1
L1 キャッシュから Fetch	0.5
分枝予測ミス	5
L2 キャッシュから Fetch	7
mutex ロック・アンロック	25
メインメモリから Fetch	100
SSD から 4KB をランダムに Read	150,000
メモリから 1MB の連続した領域を Read	250,000
ディスクから新しい領域を Fetch	8,000,000
SSD から 1MB の連続した領域を Read	1,000,000
ディスクから 1MB の連続した領域を Read	20,000,000

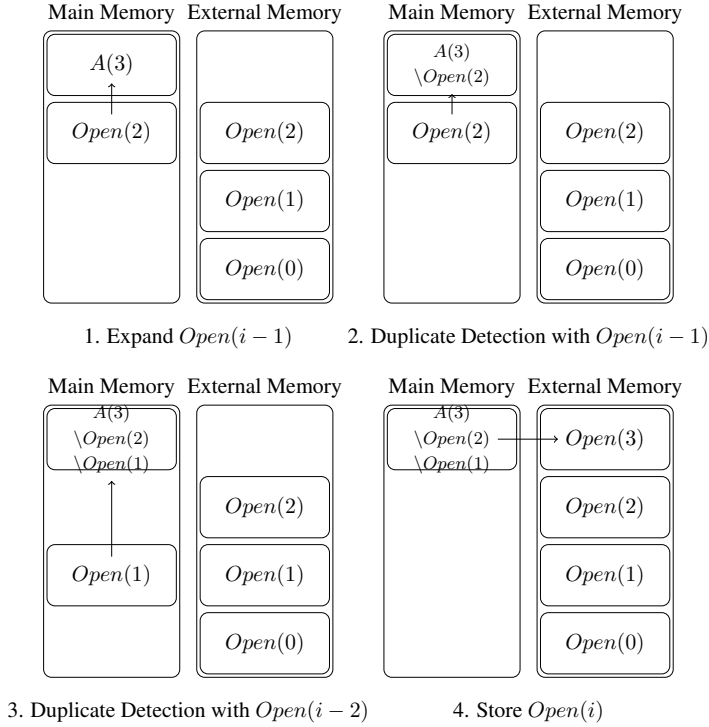


図 6.2: 外部メモリ幅優先探索の操作

6.4.1 外部メモリ 幅優先探索 (External BrFS)

一般に外部メモリが効率的であるためには問題に何らかの制約が必要になる。例えばアルゴリズム 13 にある外部メモリ幅優先探索 (external breadth-first search) (External BrFS) はグラフが無向グラフの時にしかうまくいかない [69]。

外部メモリ幅優先探索は深さ i のノードを $Open(i)$ に保持する (図 6.2)。 $Open(i)$ にあるノードを全て展開し、生成されたノードを $A(i)$ に保存する。 $A(i)$ にあるノードから $Open(i-1)$, $Open(i-2)$ と重複したノードを取り除

アルゴリズム 13: 外部メモリ幅優先探索 (External Breadth-first search)

Input : 非明示的状態空間グラフ $(s, Goal, Expand, w)$

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $Open(-1) \leftarrow \emptyset;$ 
2  $Open(0) \leftarrow \{s\};$ 
3  $i \leftarrow 1;$ 
4 while  $Open(i-1) \neq \emptyset$  do
5   if  $\exists (s \in Open(i-1)) Goal(s)$  then
6     return  $Path(s);$ 
7    $A(i) \leftarrow \{u \mid u \in Expand(u'), u' \in Open(i-1)\};$ 
8   for each  $s \in A(i)$  do
9      $parent(s) \leftarrow u \text{ s.t. } s \in Expand(u);$ 
10   $Open(i) \leftarrow A(i) \setminus (Open(i-1) \cup Open(i-2));$ 
11   $i \leftarrow i + 1;$ 
12 return  $\emptyset;$ 

```

き、残ったノードを $Open(i+1)$ に入れる。このような操作をすることによって、外部メモリ幅優先探索は深さ i のノードを展開するときは $Open(i)$ のみをメインメモリに保持し、他の展開済みノードはすべて外部メモリに置くことができる。

グラフが無向グラフである場合、状態 s と同じ状態は深さ $d(s)-2, d(s)-1, d(s)$ にしか現れない。よって深さ i のノードの集合 $A(i)$ は深さ $i-1, i-2$ で発見されたノードとだけ重複検出をすれば十分である。そのためオープンリストの全ノードではなく、 $Open(i-1), Open(i-2)$ のみを外部メモリから読み込んでくればよい。

もしグラフが有向グラフである場合、 $Open(0), Open(1), \dots, Open(i-1)$ まですべてと重複検出を行う必要があり、その場合外部メモリに何度もアクセスしなければならない。

6.5 シンボリック探索 (Symbolic Search)

二分決定グラフ (Binary Decision Diagram) (BDD) は二分木によってブーリアンの配列からブーリアンへの関数 $(x_0, x_1, \dots, x_n) \rightarrow \{0, 1\}$ を効率良く表すグラフ構造である [4, 14]。シンボリック探索 (symbolic search) は BDD を使って状態の集合、アクションの集合を表し、BDD 同士の演算によって状態の集合を一気に同時に展開していく手法である [25, 26]。A*探索がノードを一つずつ展開していき、一つずつ生成していく手間と比較して非常に効率的に演算が出来るポテンシャルを秘めている。また、オープン・クローズドリストを BDD で表せられるため、メモリ消費量が少なくなる場合がある。2014 年の International Planning Competition の Sequential Optimal 部門 (最適解を見つけるパフォーマンスを競う部門) の一位から三位までをシンボリック探索が総なめした [87]。

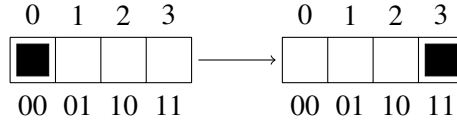


図 6.3: 特徴表現の例

表 6.2: 特徴関数の例

状態	コメント	ブーリアン表現	特徴関数
0	初期状態	00	$\neg x_0 \neg x_1$
1	-	01	$\neg x_0 x_1$
2	-	10	$x_0 \neg x_1$
3	ゴール状態	11	$x_0 x_1$

6.5.1 特徴表現 (Symbolic Representation)

説明のために非常にシンプルなスライディングタイルパズルを考える (図 7.1)。初期状態でタイルは位置 0 にあり、タイルを右か左に動かすことが出来る。タイルを位置 3 に動かせばゴールである。この問題では状態は 4 通り ($S = \{0, 1, 2, 3\}$) しかないが、可能な状態の集合は 2^4 通りある ($2^S = \emptyset, \{0\}, \{1\}, \{2\}, \{3\}, \{0, 1\}, \dots, \{0, 1, 2, 3\}$)。オープン・クローズドリストが保持する状態の集合はこのべき集合の要素である ($Open, Closed \in 2^S$)。

特徴表現 (symbolic representation) はこの状態の集合を効率よく表現するための手法である。状態の集合 $O \in 2^S$ に対して、ある状態 s が O に含まれているかを返す関数 $\phi_O : 2^S \rightarrow \{0, 1\}$ を特徴関数 (characteristic function) と呼ぶ。

ϕ_O は O に含まれる状態を全て明に保持すれば表現することが出来るが、もっと簡潔に表現することもできる。まず、4 通りの状態を 2 つのブーリアン変数 $s = (x_0, x_1)$ で表すと $S = \{00, 01, 10, 11\}$ になる。この表現を用いると、例えば $O = \{0\}$ とすると、 $\phi_O(x) = \neg x_0 \neg x_1$ と表すことが出来る。

$O = \{0, 1\}$ ならば $\phi_O(x) = \neg x_0$ となる。

アクションによる状態遷移も特徴関数 $Trans : S \times S \rightarrow \{0, 1\}$ によって定義される。アクション $a \in A$ によって状態 x から x' に遷移するならば、 $Trans_a(x, x')$ は真を返す (かつその時のみ)。アクション集合 A による遷移は $Trans(x, x')$ によって表現され、 $Trans(x, x')$ は $Trans_a(x, x')$ が真となる $a \in A$ が存在する場合に真を返す (かつその時のみ)。

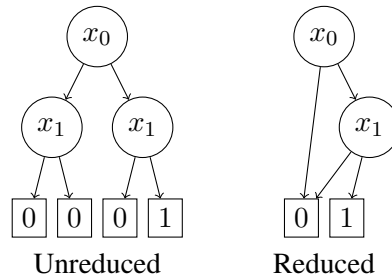
図 7.1 の問題で可能なアクションは $(00) \rightarrow (01), (01) \rightarrow (00), (01) \rightarrow (10), (10) \rightarrow (01), (10) \rightarrow (11), (11) \rightarrow (10)$ の 6 つである。これらを表す遷移関数は

$$\begin{aligned}
 Trans(x, x') = & (\neg x_0 \neg x_1 \neg x'_0 x'_1) \\
 & \vee (\neg x_0 x_1 \neg x'_0 \neg x'_1) \\
 & \vee (\neg x_0 x_1 x'_0 \neg x'_1) \\
 & \vee (x_0 \neg x_1 \neg x'_0 x'_1) \\
 & \vee (x_0 \neg x_1 x'_0 x'_1) \\
 & \vee (x_0 x_1 x'_0 \neg x'_1)
 \end{aligned} \tag{6.1}$$

となる。アクションのコストがある場合は $Trans(w, x, x')$ として表現され、 $Trans(x, x')$ は $Trans_a(x, x')$ が真となる $a \in A$ が存在し、かつそのアクションのコストが w である場合に真を返す (かつその時のみ)。

6.5.2 二分決定グラフ (Binary Decision Diagram)

状態の集合や遷移関数はブーリアンによる特徴関数によって表すことができる。この特徴関数は二分決定グラフ (binary decision diagram) (BDD) というデータ構造によってコンパクトに保持しつつさまざまな集合演算を行うことができる。

図 6.4: x_0x_1 を表す決定木 (左) と BDD (右)

定義 11 (二分決定グラフ、BDD): *BDD* はループのない有向グラフであり、ノードとエッジはラベルが付いている。単一の根ノードと2つのシンクがあり、シンクのラベルは 1 と 0 である。*sink* 以外のノードのラベルは変数 $x_i (i \in \{1, \dots, n\})$ であり、エッジのラベルは 1 か 0 である。

BDD は決定木と同様な処理によって入力 x に対して $\{1, 0\}$ を返す。すなわち、根ノードから始まり、ノードのラベル x_i に対して、入力 x の x_i が 1 であればラベル 1 が付いたエッジをたどり、 0 であればラベル 0 をたどる。これを繰り返し、シンクにたどり着いたらシンクのラベルの値を返す。決定木と異なり BDD は木ではなく、途中で合流などがあるため、決定木よりも空間効率が良い場合が多い(図 6.4)。BDD は集合演算によってたくさんの状態に対して同時に展開・重複検知を行い、探索を進めることができる。

6.5.3 特徴関数による状態空間の探索

状態空間の探索は特徴関数の演算によって表現することが出来、その演算は BDD の演算によって実装することが出来る。

ある状態集合 S に対して、 $s \in S$ となる s の次状態の集合を S の *image* と呼ぶ。 S の *image* は以下の特徴関数によって表すことが出来る。

$$Image_S(x') = \exists x(Trans(x, x') \wedge \phi_S(x)) \quad (6.2)$$

これを使ってグラフ探索アルゴリズムを実装することが出来る。例えば $Image_{Open}(x')$ はオープンリストにあるノードを全て展開して生成されるノードの集合になる。重複検出にはこの中から $Closed(x)$ に入っていないノードを取り出せばよい ($Image_{Open}(x) \wedge \neg Closed(x)$)。

6.5.3.1 BDD-幅優先木探索

$image$ を繰り返し求めていくことで幅優先木探索は簡単に実装することが出来る。まず、初期状態 s_0 だけによる集合 $S_0 = \{s_0\}$ を考える。これに対して ϕ_{S_i} は集合 S_i を表す特徴関数だとする。これを用いることで次状態集合を次々と求めることが出来る：

$$\phi_{S_i}(x') = \exists x(\phi_{S_{i-1}}(x) \wedge Trans(x, x')) \quad (6.3)$$

簡単に言えば、状態 x' は、もし親状態 x が S_{i-1} に含まれていれば、 S_i に含まれる。探索を停止するためには探索した状態にゴール状態が含まれているかをテストしなければならない。ゴールテストも特徴関数を用いて表すことが出来る。ゴール状態集合 T を表す特徴関数を ϕ_T とすると、 $\phi_{S_i}(x') \wedge \phi_T \neq false$ であれば S_i はゴール状態を含む。

アルゴリズム 14 は BDD-幅優先木探索である。 $image$ の計算とゴールテストによって実装することが出来る。

$Construct$ 関数はゴールに至るための経路を計算する関数である。 $\phi_{S_i} \wedge \phi_T(x)$ によってゴール状態、解経路における i ステップ目の状態 (s_i) が得られる。次に $Trans(\phi_{S_{i-1}}, s_i)$ によって $i-1$ ステップ目の状態 s_{i-1} が得られ、 $Trans_a$ を見ていくことで $i-1$ ステップ目のアクションが得られる。これを繰り返すことによって元の解経路を復元することが出来る。ゴール状態は一つ取り出せば十分であるため、 $Construct$ の計算時間は大きくはない。

アルゴリズム 14: BDD 幅優先木探索 (BDD-Breadth-first Tree Search)

Input : 非明示的状态空間グラフ $P = (s, Goal, Expand, w)$ 、前状態 s'

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $S_0 \leftarrow \{s\};$ 
2 for  $i \leftarrow 1, 2, \dots$  do
3    $\phi_{S_i}(x) \leftarrow \exists x(\phi_{S_{i-1}}(x) \wedge Trans(x, x'))[x'/x];$ 
4   if  $\phi_{S_i}(x) \wedge \phi_T \neq false$  then
5     return  $Construct(\phi_{S_i} \wedge \phi_T(x), i);$ 

```

BDD-幅優先木探索は幅優先探索と同様、解の経路長が最短であることを保証する。

6.5.3.2 BDD-幅優先グラフ探索

重複検出を行う場合はクローズドリストに展開済みノードを保存する必要がある。この展開済みノードも特徴関数及び BDD で表すことが出来る。アルゴリズム 15 は BDD-幅優先探索のコードである。アルゴリズム 14 と異なり特徴関数 $Closed$ を用いて重複検出を行っている。

6.6 新奇性に基づく枝刈り (Novelty-based Pruning)

状態空間を広く探索することは局所最適やデッドエンドに陥らないためには必要である。より新奇性 (novelty) のある状態を優先して探索することによって広く状態空間が探索できると考えられる。状態空間が非常に大きい場合は、よりアグレッシブに、すでに生成された状態と似たような状態を枝刈りしていく方法がある。

アルゴリズム 15: BDD 幅優先探索 (BDD Breadth-first search)

Input : 非明示的状态空間グラフ $P = (s, Goal, Expand, w)$ 、前状態 s'

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $S_0 \leftarrow \{s_0\};$ 
2  $Closed \leftarrow \{s_0\};$ 
3 for  $i \leftarrow 1, 2, \dots$  do
4    $Succ(x) \leftarrow \exists x(\phi_{S_{i-1}}(x) \wedge Trans(x, x'))[x'/x];$ 
5    $\phi_{S_i}(x) \leftarrow Succ(x) \wedge \neg Closed(x);$ 
6    $Closed(x) \leftarrow Closed(x) \vee Succ(x);$ 
7   if  $\phi_{S_i}(x') \wedge \phi_T \neq false$  then
8     return  $Construct(\phi_{S_i} \wedge \phi_T(x), i);$ 

```

6.6.1 状態の新奇性 (Novelty)

状態に対して新奇性を定義する試みは古くから人工知能研究にある [63]。なので新奇性の定義も様々であるが、本書では [36] の定義に従い、新奇性を以下のように定義する。

定義 12 (新規性、Novelty): m 個の特徴関数の集合 h_1, h_2, \dots, h_m に対して新たに生成された状態 s の新奇性 $w(s)$ は n であるとは、 n 個の特徴関数によるタプル $\{h_{i_1}, h_{i_2}, \dots, h_{i_n}\}$ が存在し、 $h_{i_1}(s) = h_{i_1}(s')$, $h_{i_2}(s) = h_{i_2}(s'), \dots, h_{i_n}(s) = h_{i_n}(s')$ を満たす生成済みの状態 s' が存在せず、かつ、この条件を満たすそれよりも小さいタプルが存在しない。

特徴関数は単純に状態変数の値を返す関数を使うことができる [36, 66]。つまり状態 $s = \{v_1, v_2, \dots, v_m\}$ に対して特徴関数は $h_i = v_i$ とする。この場合、 $w(s) = m$ であれば状態変数がすべて同じノードがすでに生成済みであ

ということなので、 s は重複したノードである。このため枝刈りのために新奇性を定義する場合はこれが便利である。

6.6.2 幅制限探索 (Width-Based Search)

幅制限探索 (width-based search) は状態の新奇性に基づいてノードを枝刈りする手法である [65]。新奇性によってノードを枝刈りするので、解が存在しても発見される保証はない (完全性を満たさない)。

アルゴリズム 16: 幅制限探索 (Width-based search)

Input : 非明示的状态空間グラフ $(s, Goal, Expand, w)$ 、特徴関数 h_0, h_1, \dots, h_{m-1} 、幅 i

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $Closed \leftarrow \emptyset, Open \leftarrow \{s\};$ 
2 while  $Open \neq \emptyset$  do
3    $u \leftarrow \arg \min_{u' \in Open} f(u');$ 
4    $Open \leftarrow Open \setminus \{u\};$ 
5   if  $Goal(u)$  then
6     return  $Path(u);$ 
7   for each  $v \in Expand(u)$  do
8      $H(v) \leftarrow \{j | j \in \{0, 1, \dots, m-1\}, h_j(v) = 1\};$ 
9      $H \leftarrow \{j | j \in \{0, 1, \dots, m-1\}, h_j(v) \in Closed\};$ 
10    if  $|H \setminus Closed| \geq i$  then
11       $Open \leftarrow Open \cup \{v\};$ 
12       $parent(v) \leftarrow u;$ 
13    for each  $h \in H(v) \setminus Closed$  do
14       $Closed \leftarrow Closed \cup \{h\};$ 
15 return  $\emptyset;$ 

```

$IW(i)$ は新奇性が i よりも大きい状態を枝刈りする探索である。新奇性に基づく枝刈りには 2 つのメリットがある。一つは状態空間が著しく小さくなる。もう一つは幅を制限することで生成済みノードをクローズドリストにすべて保存する必要がなくなる。その代わり保存しなければならない情報は、生成済みの特徴のタプルのうち大きさが i 以下のものの集合である。これがないと新奇性を計算することができない。 IW におけるクローズドリストは過去に真であったことのある特徴の集合である。状態 s において真である特徴の集合を H としたとき、 H に存在し $Closed$ に存在しない特徴の数が i 以上かどうかを確認し、 i 以上であればそのノードを生成する。そうでなければ枝刈りを行う。

幅 i を大きくするほど発見できる解のクオリティが上がりやすいが、一方探索空間は i に対して指数的に大きくなっていく。このトレードオフを調整しやすくするために新奇性の定義域を有理数に拡張した幅制限探索が提案されている [36]。

6.6.3 反復幅制限探索 (Iterative Width Search)

反復幅制限探索 (iterative width search) は幅制限探索を幅を大きくしながら解を発見するまで繰り返すアルゴリズムである [65]。 $IW(i)$ は幅 i が大きくなるほど解のクオリティが良くなるが、より大きな状態空間を探索することになる。反復幅制限探索は小さい幅からはじめ、解が見つからない・良い解でなければ幅を大きくして再度探索を行う手法である。いずれ幅の大きさが特徴の数と同じまでになるので、反復幅制限探索は解があればいずれ必ず発見する。

6.7 並列探索 (Parallel Search)

近年コンピューター一台当たりのコア数は増加を続けており、コンピュータクラスタにも比較的容易にアクセスが出来るようになった。Amazon Web Service

アルゴリズム 17: 反復幅制限探索 (Iterative Width Search)

```
1  $i \leftarrow 1$ ;  
2  $path \leftarrow \emptyset$ ;  
3 while  $path = \emptyset$  do  
4    $path \leftarrow IW(i)$ ;  
5    $i \leftarrow i + 1$ ;  
6 return  $path$ 
```

のようなクラウドの計算資源も普及し、将来的には並列化が当然になると考えられる。並列化の成功例は枚挙にいとまないが、近年のディープラーニングはまさに効率的な並列計算アーキテクチャによって得られたブレイクスルーであるといえる。グラフ探索アルゴリズムの並列化に考えなければならないオーバーヘッドは様々であり、それらの重要性は問題、インスタンス、マシン、さまざまな状況に依存する。CPU 並列ではハッシュによってノードを各プロセスにアサインし、各プロセスはアサインされたノードのみを担当して探索を行うというフレームワークが有効である。より詳細な議論は [35] を参照されたい。

6.7.1 並列化オーバーヘッド (Parallel Overheads)

理想的には n プロセスで並列化したら n 倍速くなってほしい。逐次アルゴリズムと比較して、プロセス数倍の高速化が得られることを完全線形高速化 (perfect linear speedup) と呼ぶ。しかしながら、殆どの場合完全線形高速化は得られない。それは並列化にさいして様々なオーバーヘッドがかかるからである。[53] の記法に従うと、並列化オーバーヘッドは主に以下の 3 つに分けられる。

定義 13 (通信オーバーヘッド、communication overhead): プロセス間で情報交換を行うことにかかる時間を通信オーバーヘッド (*communication overhead*) (*CO*) と呼ぶ。

通信する情報は様々なものが考えられるが、オーバーヘッドとなるものはノードの生成回数に比例した回数通信を必要とするものである。すなわち、ノードの生成回数 n に対して $\log(n)$ 回しか通信を行わない場合、その通信によるオーバーヘッドは無視出来るだろう。ここではノードの生成回数に対するメッセージ送信の割合を *CO* と定義する：

$$CO := \frac{\text{他プロセスへ送信されたメッセージの数}}{\text{生成されたノードの数}}. \quad (6.4)$$

例えば、ハッシュなどによってプロセス間でノードの送受信を行いロードバランスを行う手法 (e.g. 後述する HDA*) の場合、通信するメッセージは主にノードである。この場合：

$$CO := \frac{\text{他プロセスへ送信されたノードの数}}{\text{生成されたノードの数}}. \quad (6.5)$$

となる。*CO* は通信にかかるディレイだけでなく、メッセージキューなどのデータ構造の操作も行わなければならないので、特にノードの展開速度が速いドメインにおいて重要なオーバーヘッドになる。一般に、プロセス数が多いほど *CO* は大きくなる。

定義 14 (探索オーバーヘッド、Search Overhead): 並列化によって余分に増えた展開ノード数の割合を探索オーバーヘッドと呼ぶ。

一般に並列探索は逐次探索より多くのノードを展開することになる。このとき、余分に展開したノードは逐次と比較して増えた仕事量だと言える。本書では以下のように探索オーバーヘッドを定義する：

$$SO := \frac{\text{並列探索で展開されるノード数}}{\text{逐次探索で展開されるノード数}} - 1. \quad (6.6)$$

SO はロードバランス (load balance) (LB) が悪い場合に生じることが多い。

$$LB := \frac{\text{各プロセスに割り当てられたノード数の最大値}}{\text{各プロセスに割り当てられたノード数の平均}}. \quad (6.7)$$

ロードバランスが悪いと、ノードが集中しているスレッドがボトルネックとなり、他のスレッドはノードがなくなるか、あるいはより f 値の大きい (有望でない) ノードを展開することになり、探索オーバーヘッドになる。

探索オーバーヘッドは実行時間だけでなく、空間オーバーヘッドでもある。ムダに探索をした分だけ、消費するメモリ量も多くなる。分散メモリ環境においてもコア当りの RAM 量は大きくなるわけではないので、探索オーバーヘッドによるメモリ消費は問題となる。

定義 15 (同期オーバーヘッド、Coordination Overhead): 他のスレッドの処理をアイドル状態で待たなければならない時間の割合を同期オーバーヘッド (*coordination overhead*) と呼ぶ。

アルゴリズム自体が同期を必要としないものだとしても、メモリバスのコンテンションによって同期オーバーヘッドが生じることがある [15, 57].

これらのオーバーヘッドは独立ではなく、トレードオフの関係にある。多くの場合、通信・同期オーバーヘッドと探索オーバーヘッドがトレードオフの関係にあたる。

6.7.2 並列 A* (Parallel A*)

並列 A* (parallel A*) (PA*) は A* 探索のシンプルな並列化である [51]。PA* は一つのオープンリスト・クローズドリストを全てのプロセスで共有してアクセスする。オープンリスト・クローズドリストには複数のプロセスが同時にアクセスできないようにロックがかけられる。PA* の問題点はデータ構造

を一つに集中させているため、分散メモリ環境の場合はアクセスに大きな通信オーバーヘッドが生じることである。共有メモリ環境の場合でも、それぞれのプロセスは他のプロセスがデータ構造にアクセスしている間にはアクセスできないため、同期オーバーヘッドが生じる。この同期オーバーヘッドはプロセスの数が増えるほど問題になる。ロックフリーのデータ構造を使ってもこのオーバーヘッドは解消されない [15]。

6.7.3 ハッシュ分配 A* (Hash Distributed A*)

PA*の問題点はすべてのプロセスが共有された一つのデータ構造へアクセスしなければならない点である。このような手法を集中型手法 (centralized approach) と呼ぶ。共有データ構造へのアクセスは計算のボトルネックとなり、スケーラビリティの限界点になってしまう。そのため、A*の並列化には各プロセスが自分のオープンリスト・クローズドリストを持ち、アクセスを分散させる分散型手法 (decentralized approach) が有力である。分散型手法はデータ構造へのアクセスが分散するため同期オーバーヘッドを解消される。一方、各プロセスがアクセスするデータ構造が異なるため、各プロセスの仕事量を均等に分配しづらいというロードバランシング (load balancing) の問題が生じる。各プロセスに対してどのようにして仕事を均等に分配するかが問題になる。

ハッシュ分配 A* (Hash Distributed A*) (HDA*) [57] は state-of-the-art の並列 A*探索アルゴリズムである。HDA*の各プロセスはそれぞれローカルなオープンリスト、クローズドリストを保持する。ローカルとは、データ構造を保持するプロセスが独占してアクセスを行い、他のプロセスからはアクセスが不可能であるという意味である。グローバルなハッシュ関数によって全ての状態は一意に定まる担当のプロセスが定められる。各プロセス T の動作は以下を繰り返す (アルゴリズム 23) :

1. プロセス T はメッセージキューを確認し、ノードが届いているかを確認する。届いているノードのうち重複でないものをオープンリストに

加える (A*同様、クローズドリストに同じ状態が存在しないか、クローズドリストにある同じ状態のノードよりも f 値が小さい場合に重複でない)。

2. オープンリストにあるノードのうち最もプライオリティ (f 値) の高いノードを展開する。生成されたそれぞれのノード n についてハッシュ値 $H(n)$ を計算し、ハッシュ値 $H(n)$ を担当するプロセスに非同期的に送信される。

HDA*の重要な特徴は2つある。まず、HDA*は非同期通信を行うため、同期オーバーヘッドが非常に小さい。各プロセスがそれぞれローカルにオープン・クローズドリストを保持するため、これらのデータ構造へのアクセスにロックを必要としない。次に、HDA*は手法が非常にシンプルであり、ハッシュ関数 $Hash : S \rightarrow 1..P$ を必要とするだけである (P はプロセス数)。しかしながらハッシュ関数は通信オーバーヘッドとロードバランスの両方を決定する為、その選択はパフォーマンスに非常に大きな影響を与える。

HDA*が提案された論文 [57] では Zobrist hashing [92] がハッシュ関数として用いられていた。状態 $s = (x_1, x_2, \dots, x_n)$ に対して Zobrist hashing のハッシュ値 $Z(s)$ は以下のように計算される：

$$Z(s) := R_0[x_0] \text{ xor } R_1[x_1] \text{ xor } \dots \text{ xor } R_n[x_n] \quad (6.8)$$

Zobrist hashing は初めにランダムテーブル R を初期化する4。これを用いてハッシュ値を計算する。

Zobrist hashing を使うメリットは2つある。一つは計算が非常に速いことである、XOR 命令はCPUの演算で最も速いものの一つである。かつ、状態の差分を参照することでハッシュ値を計算することが出来るので、アクション適用によって値が変化した変数の $R[x]$ のみ参照すれば良い。もうひとつは、状態が非常にバランスよく分配され、ロードバランスが良いことである。一方、この手法の問題点は通信オーバーヘッドが大きくなってしまいうことにある。この問題を解決するために State abstraction という手法が提案された [15]。

アルゴリズム 18: Hash Distributed A*

Input : 非明示的状态空間グラフ $(s, Goal, Expand, w)$ 、ヒューリスティック関数 h 、ハッシュ関数 H

Output : s からゴール状態 $t \in T$ への経路、経路が存在しなければ \emptyset

```

1  $Closed_p \leftarrow \emptyset, Open_p \leftarrow \emptyset, Buffer_p \leftarrow \emptyset$  for each thread  $p$ ;
2  $Open_{H(s)} \leftarrow \{s\}, g(s) \leftarrow 0$ ;
3  $goal \leftarrow \emptyset, c' \leftarrow \infty$ ;
4 for each process  $p$  do
5   while not  $TerminateDetection()$  do
6     for each  $(v, g_v, p_v) \in Buffer_p$  do
7       if  $v \notin Closed$  or  $g_v < g(v)$  then
8          $Open \leftarrow Open \cup \{v\}$ ;
9          $g(v) \leftarrow g_v$ ;
10         $parent(v) \leftarrow p_v v$ ;
11      if  $v \notin Closed$  then
12         $Closed \leftarrow Closed \cup \{v\}$ ;
13       $Buffer_p \leftarrow \emptyset$ ;
14      while  $Open_p \neq \emptyset$  and  $\min_{s \in Open_p} f(s) < c'$  do
15         $u \leftarrow \arg \min_{u' \in Open} f(u')$ ;
16         $Open \leftarrow Open \setminus \{u\}$ ;
17        if  $Goal(u)$  and  $g(u) < c'$  then
18           $goal \leftarrow u$ ;
19           $c' \leftarrow g(u)$ ;
20        for each  $v \in Expand(u)$  do
21           $g_v \leftarrow g(u) + w(u, v)$ ;
22           $Buffer_{H(v)} \leftarrow Buffer_{H(v)} \cup \{(v, g_v, u)\}$ ;
23  return  $Path(goal)$ ;

```


アルゴリズム 19: Zobrist Hashing

Input : $s = (x_0, x_1, \dots, x_n)$

```

1  $hash \leftarrow 0$ ;
2 for each  $x_i \in s$  do
3    $hash \leftarrow hash \text{ xor } R[x_i]$ ;
4 return  $hash$ ;

```

アルゴリズム 20: Zobrist Hashing の初期化

Input : $V = (dom(x_0), dom(x_1), \dots)$

```

1 for each  $x_i$  do
2   for each  $t \in dom(x_i)$  do
3      $R_i[t] \leftarrow random()$ ;
4 return  $R = (R_1, R_2, \dots, R_n)$ 

```

State abstraction は状態 $s = (x_1, x_2, \dots, x_n)$ に対して簡約化状態 (abstract state) $s' = (x'_1, x'_2, \dots, x'_m)$, where $m < n, x'_i = x_j (1 \leq j \leq n)$. State abstraction は簡約化状態からハッシュ値への関数の定義はされておらず、単純な linear congruent hashing が用いられていた。そのため、ロードバランスが悪かった。

Abstract Zobrist hashing (AZH) は Zobrist hashing と Abstraction の良い点を組み合わせた手法である [52]。AZH は feature から abstract feature へのマッピングを行い、abstract feature を Zobrist hashing への入力とするという手法である：

$$Z(s) := R_0[A_0(x_0)] \text{ xor } R_1[A_1(x_1)] \text{ xor } \dots \text{ xor } R_n[A_n(x_n)] \quad (6.9)$$

ここで関数 A は feature から abstract feature へのマッピングであり、 R は abstract feature に対して定義されている。

HDA*のための分解関数の自動生成方法は [53] にまとめられている。

自動行動計画問題 (Automated Planning Problem)

本書の冒頭でグラフ探索アルゴリズムが人工知能のための技術として研究されていると説明した。人工知能とはさまざまな定義で使われる言葉であるが、グラフ探索は自動推論や自動行動計画のために使うことができるために研究されている。この章では人工知能の一分野である自動行動計画問題 (automated planning problem) について説明する [37]。自動行動計画は初期状態から命題集合で表される目的を達成するためのプランを発見する問題である。自動行動計画のためのモデルは様々提案されている。最も基本となるモデルは古典的プランニング問題である [31]。古典的プランニングは完全情報¹、決定的状態遷移を仮定とするモデルであり、状態空間問題に含まれる [31]。古典的プランニングによって様々な実問題を表すことができる。例えばロジスティック [44, 83]、セルアセンブリ [7]、遺伝子距離計算 [29]、ビデオゲーム [66] など、様々な応用問題を含むモデルである。

完全情報、決定的状態遷移の仮定を緩和した問題（確率的モデルや不完

¹正確には完全情報ではなく、アクションの決定のために必要な情報がすべて得られると仮定される。例えば問題に全く関係ない冗長な変数が含まれ、その情報がエージェントに与えられない場合は完全情報ではないが、アクションの決定のためには不要な情報であれば古典的プランニング問題で扱うことができる。

全情報モデル) もグラフ探索によって解かれることが多いが、本文の範囲外とする。詳細は詳しい教科書を参照されたい [78]。なお、プランニング問題は A* などの状態空間探索アルゴリズム以外にも、SAT や CSP などの制約充足問題に変換して解く方法もある [30, 23]。古典的プランニングのための解法としてはグラフ探索アルゴリズムが state-of-the-art であるが、例えばマルチエージェント経路探索問題などでは SAT に変換する手法が効率的であることが知られている [79]。

7.1 定義

古典的プランニングは述語論理によって世界が記述される [31]。Proposition AP は世界の状態において何が真・偽であるかを記述する。世界の状態はエージェントがアクションを行うことによって遷移し、遷移後の状態は遷移前の状態と異なる proposition が真・偽でありうる。古典的プランニングの目的は与えられた初期状態からゴール条件を満たすまでのアクションの列を求めることにある。以下、定義は [27] に従う。

定義 16 (古典的プランニング問題、Classical Planning Problem): 古典的プランニング問題は有限状態空間問題 $P = (S, A, s_0, T)$ の一つである。 $S \subseteq 2^{AP}$ は状態の集合であり、 $s_0 \in S$ は初期状態、 $T \subseteq S$ はゴール状態の集合、 $A : S \rightarrow S$ は可能なアクションの集合である。

古典的プランニング問題の最も基本となる STRIPS モデル [31] の場合、ゴール条件は命題 (proposition) のリストで表せられる $Goal \subseteq AP$ 。ゴール状態の集合 T は $p \in Goal$ となるすべての p が真である状態の集合である。アクション $a \in A$ は適用条件 $pre(a)$ 、効果 ($add(a), del(a)$) で表せられる。適用条件 $pre(a) \subseteq AP$ はアクション a を実行するために状態が満たすべき proposition の集合である。効果 $add(a)$ はアクション a を適用後に真になる proposition の集合であり、 $del(a)$ は偽になる集合である。従って、アクショ

ン a を状態 s に適用後の状態 $s' = \text{suc}(s, a)$ は

$$\text{suc}(s, a) = (s \cup \text{add}(a)) \setminus \text{del}(a) \quad (7.1)$$

である。

このようにして、古典的プランニング問題は状態空間問題に帰着することが出来る。状態空間問題はさらにグラフ探索問題に帰着することができる。グラフ探索による解法の利点は任意の状態空間問題に対して解法となる点である。各ドメインに対して特化した手法を用いることなくさまざまな古典的プランニング問題の形式で書ける問題に対して同じグラフ探索の手法が適用できる。つまりプログラムを使う側に立ってみれば、何かの問題解決を行うにあたって問題の定義のみを与えれば、それをどうやって解くかを考えることなく、問題の解を得ることができるのである。逆に言えば、効率的なグラフ探索アルゴリズムが開発できれば、あらゆる状態空間問題に対して適用できる汎用な手法が高速化できたと言える。

プランニングの問題定義やアルゴリズムの実装を見たい方は fast-downward [42] を見ると良い。fast-downward はプランニング問題を解く state-of-the-art のプランナーである。本書で紹介するアルゴリズムのほとんどが fast-downward に組み込まれている。

7.2 プランニングドメイン記述言語: Planning Domain Definition Language

Planning Domain Definition Language (PDDL) [1] はプランニング問題を記述されるために用いられる言語の一つである。PDDL はプランニング問題を一階述語論理で表現する。PDDL はドメイン記述とインスタンス記述の2つに分けられ、Lisp のような文法で書かれる。図 7.2 と図 7.3 はブロックワールド問題のドメイン記述とインスタンス記述である。ここでドメインとインスタンスは計算理論などで定義される問題 (problem) とインスタンス (instance)

102チャプター 7. 自動行動計画問題 (AUTOMATED PLANNING PROBLEM)

に対応する。ドメイン記述は問いの集合を定義し、インスタンスは一つの問いを指定する。例えば「グリッド経路探索問題」はドメインであり、そのうちの特定の一つのマップがインスタンスに対応する。ドメイン記述には述語 (predicate) (predicates) とアクションスキーム (action scheme) (action) がある。インスタンス記述にはオブジェクト (object) (objects) と初期状態 (init)、ゴール状態 (goal) がある。これら以外にも例えばオブジェクトの型など様々な文法があるが、簡単のためここでは割愛する。これらの記述によって古典的プランニング問題が定義される。

まず、命題集合 AP は述語に含まれる変数にオブジェクトを割り当てることによって得られる。図の例だと例えば $(on\ A\ B)$, $(on\ A\ C)$, $(ontable\ D)$, ... などの命題が AP に含まれる。アクション集合 A はアクションスキームに含まれる変数にオブジェクトを割り当てることによって得られ、アクションの変数は `parameters` に定義される。アクション a の適用条件 $pre(a)$ はアクションスキームの `precondition` にオブジェクトを割り当てることで得られる。`effect` のうち `not` のついていない命題は $add(a)$ に対応し、`not` のついた命題は $del(a)$ に対応する。例えばアクション $(pickup\ A)$ の適用条件は $(clear\ A)$, $(ontable\ A)$, $(handempty)$ 、追加効果は $(holding\ A)$ 、削除効果 $(ontable\ A)$, $(clear\ A)$, $(handempty)$ である。初期状態 s_0 は `init` の命題集合である。この例では $(CLEAR\ C)\ (CLEAR\ A)\ (CLEAR\ B)\ (CLEAR\ D)\ (ONTABLE\ C)\ (ONTABLE\ A)$ である。ゴール条件 $Goal$ は `goal` の命題集合である。つまり、ゴール状態集合 T は $Goal$ を含む状態の集合である。

PDDL のミソは一階述語論理によってプランニング問題を記述する点である。状態空間に含まれる命題を一つ一つ記述するのではなく、述語とオブジェクトの組み合わせによって複数の命題をコンパクトに記述することができる。また、インスタンス記述を変えることで同じドメインの異なるインスタンスをモデルすることができる。例えばブロックワールドのドメイン記述はそのままに、インスタンス記述のオブジェクトや `init` などを変えることで違う問いにすることができる。

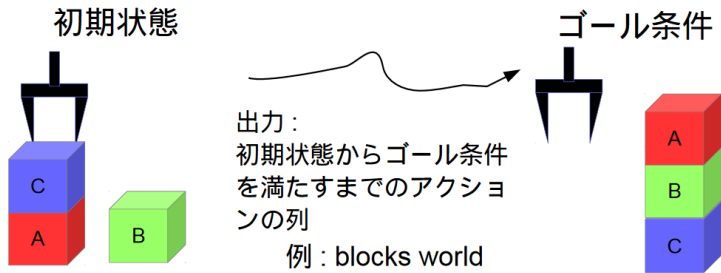


図 7.1: Blocks world ドメイン

PDDL は状態空間問題だけでなくより広く様々な問題を扱うことができる [1, 33]。fast-downward は PDDL の文法の多くをサポートしているので試してみるには便利である。

7.3 古典的プランニング問題の例

プランニングは様々な問題解決に役立てることができる。ここでは簡単にプランニングによってどのような問題がモデルできるかを紹介する。

1. エアポート (airport) 空港の地上の交通管理を行う問題である。飛行機の離着陸の時間が与えられるのに対し、安全かつ飛行機の飛行時間を最小化する交通を求める問題である。
2. サテライト (satellite) 人工衛星は与えられた現象を撮影し、地球にデータを送らなければならない。このドメインは NASA の人工衛星の実際の応用から考案されたものである。
3. ローバー (rovers) ローバーとは惑星探査ロボットのことであり。この問題は惑星探査ロボットのグループを使って惑星を探索する計画を作る問題である。ロボットらは興味深い地形などの写真を取るなどの作業

104チャプター 7. 自動行動計画問題 (AUTOMATED PLANNING PROBLEM)

```
////////////////////////////////////
;;; 4 Op-blocks world
////////////////////////////////////
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
               (ontable ?x)
               (clear ?x)
               (handempty)
               (holding ?x)
               )

  (:action pick-up
    :parameters (?x)
    :precondition (and (clear ?x) (ontable ?x) (
      handempty))
    :effect
    (and (not (ontable ?x))
          (not (clear ?x))
          (not (handempty))
          (holding ?x)))

  (:action put-down
    :parameters (?x)
    :precondition (holding ?x)
    :effect
    (and (not (holding ?x))
          (clear ?x)
          (handempty)
          (ontable ?x)))

  (:action stack
    :parameters (?x ?y)
    :precondition (and (holding ?x) (clear ?y))
    :effect
    (and (not (holding ?x))
          (not (clear ?y))
          (clear ?x)
          (handempty)
          (on ?x ?y)))

  (:action unstack
    :parameters (?x ?y)
    :precondition (and (on ?x ?y) (clear ?x) (handempty)
      )
    :effect
    (and (holding ?x)
          (clear ?y))
  )
)
```



```
(define (problem BLOCKS-3)
  (:domain BLOCKS)
  (:objects A B C)
  (:init (CLEAR C) (CLEAR B) (ONTABLE C) (ONTABLE B)
        (ON C A) (HANDEEMPTY))
  (:goal (AND (ON A B) (ON B C)))
)
```

図 7.3: blocks-world の instance ファイル

を実行する必要がある。このドメインも NASA の応用をもとにしたものである。

4. パイプスワールド (pipesworld) 複数の原油の油田から複数の目的地にパイプを通して送りたい。各目的地に定められた量を送るように調整することが目的である。パイプのネットワークはグラフとしてモデルされており、また同時に原油を通せないパイプの組が存在する。
5. セルアセンブリ (cell assembly) セルアセンブリは狭いセルの中で働き手が複雑な製品を作成する工場システムである。大規模な製造ラインと比較して、セルアセンブリは主に中程度の数 (100 個程度など) の製品を作るために使われる。製品の開発や受注生産などに対応して、今生産しなければならない製品を手早く作成するためのセルの行動プランを考えることが問題の目的である。[7]
6. ゲノムリアレンジメント (Genome rearrangement) ゲノムリアレンジメントは多重整列問題の一つである。ゲノム間の編集距離とは類似性を測るための指標として使われ、生物の進化の歴史をたどるために使われる。編集距離はあるゲノムから操作を繰り返してもう一方のゲノムに変換するためのコストの総和として定義される。プランニングモデルを用いることでより様々な操作を定義することができる。例えば遺

伝子の位置に入れ替えなど、2.2.3 節で説明したように単純にグリッド経路探索に落とし込むことのできない複雑な操作を考えることができる。[29]

7. トラック (trucks) ロジスティクスと呼ばれる問題の一種である。トラックを運転してすべての荷物をそれぞれ定められた運び先に届ける問題である。ただしトラックが運べる荷物の総量は有限であるため、それを考慮して経路を考えなければならない。加えて、届けるまでの期限が存在する荷物が存在する。

8. 工場のセル生産システム

これらの問題はすべて問題に特化した特別なアルゴリズムをそれぞれの問題に対して開発することもできる。多くの場合、特化アルゴリズムの方が汎用プランナーよりも高速である。プランナーの利点は問題に特化した実装をしなくても PDDL を書くだけで問題を解くことが出来るという点にある。

7.4 ヒューリスティック関数の自動生成

PDDL にはヒューリスティック関数は何を使えばよいかなどの情報は書かれていない。よって、PDDL を入力とする状態空間問題を解く場合、エージェントはヒューリスティック関数を自動生成しなければならない。PDDL からヒューリスティック関数を自動生成する手法はプランニング研究の最も重要な研究課題の一つである。

ヒューリスティックの生成方法の一つの指針としては4.4 節で言及した緩和問題によるヒューリスティックが分かりやすい。すなわち、元の問題 P よりも簡単な問題 P' を生成し、 P の状態 s から P' の状態 s' へのふさわしい関数を定義する。そして $h(s)$ の値を P' における s' を初期状態とするゴールへの最適解にする。このようにしてヒューリスティック関数は自動生成するこ

とができる。以下、具体的にどのような手法があるかを説明する。各アルゴリズムの実装は fast-downward²を参照されたい。

7.4.1 ゴールカウントヒューリスティック (Goal Count Heuristic)

多くの問題ではゴールはいくつかの条件を満たした状態の集合として与えられる。ゴールカウントヒューリスティックは満たしていないゴール条件の数をヒューリスティック値とする関数である。例えばスライディングタイルのゴール条件は全てのタイルが所定の位置にあることである。なので所定の位置にないタイルの数を h 値とすることが出来る。

ゴールカウントヒューリスティックは許容的であるとは限らない。コスト 1 のアクションが 2 つのゴールを同時に満たすかもしれないからだ。1 つのアクションで同時に満たせるゴールが 1 つ以下である場合、そしてその時のみ、許容的である。

7.4.2 パターンデータベースヒューリスティック (Pattern Database Heuristic)

パターンデータベースヒューリスティック (Pattern database heuristic) は非常に強力な state-of-the-art のヒューリスティック関数である [24]。問題の命題集合 AP に対して $AP' \subset AP$ を置き、 AP' を命題集合とする緩和問題 P' を考えるというのが大筋の考え方である。シンプルだが、 AP' の選び方によっては非常に正確であり、かつ高速なヒューリスティック関数になることが知られている。

7.5 関連文献

複数のパターンデータベースを使ってヒューリスティック値をより正確にすることができる (multiple pattern databases) [49]。任意の許容的なヒューリス

²<http://www.fast-downward.org>

ティックの組 h_1, h_2 に対して、 $h(s) = \max(h_1(s), h_2(s))$ は許容的である。このことから、複数のパターンデータベースからヒューリスティック値を計算し、それらの最大値を h 値とする方法がある。また、複数のデータベースの h 値の和 $h(s) = h_1(s) + h_2(s)$ が許容的なヒューリスティックになる手法もある (disjoint pattern databases) [60]。

パターンデータベースはアブストラクト問題の解を列挙してメモリに保存する。アブストラクト問題の状態空間の大きさに比例したメモリ量を消費することになる。それに対してマージアンドシュリンク (merge and shrink) は複数のアブストラクト問題のグラフをマージして、出来上がったグラフのノードを縮約してグラフを小さくする (シュリンク) 手法である [43]。この方法はパターンデータベースよりもアブストラクト問題を少ないメモリ量で表すことができる。どのアブストラクト問題をマージするか、どのノードを縮約するか戦略によってアルゴリズムの性能は大きく変わる。

パターンデータベース、マージアンドシュリンクなどのヒューリスティックは fast downward に実装されている。興味があれば様々な問題で性能比較をしてみると面白い。

参考文献

- [1] Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I.D., Ram, A., Veloso, M., Weld, D., SRI, D.W., Barrett, A., Christianson, D., et al.: PDDL the planning domain definition language version 1.2 (1998)
- [2] Ahuja, R.K., Mehlhorn, K., Orlin, J., Tarjan, R.E.: Faster algorithms for the shortest path problem. *Journal of the ACM* **37**(2), 213–223 (1990)
- [3] Akagi, Y., Kishimoto, A., Fukunaga, A.: On transposition tables for single-agent search and planning: Summary of results. In: *Third Annual Symposium on Combinatorial Search* (2010)
- [4] Akers, S.B.: Binary decision diagrams. *IEEE Transactions on computers* (6), 509–516 (1978)
- [5] Algfoor, Z.A., Sunar, M.S., Kolivand, H.: A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology* p. 7 (2015)

- [6] Applegate, D.L.: The traveling salesman problem: a computational study. Princeton University Press (2006)
- [7] Asai, M., Fukunaga, A.: Fully automated cyclic planning for large-scale manufacturing domains. In: Twenty-Fourth International Conference on Automated Planning and Scheduling (2014)
- [8] Asai, M., Fukunaga, A.: Tiebreaking strategies for A* search: How to explore the final frontier. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI) (2016)
- [9] Bäckström, C., Jonsson, P., Ståhlberg, S.: Fast detection of unsolvable planning instances using local consistency. In: Sixth Annual Symposium on Combinatorial Search (2013)
- [10] Barker, J.K., Korf, R.E.: Limitations of front-to-end bidirectional heuristic search. In: Twenty-Ninth AAAI Conference on Artificial Intelligence (2015)
- [11] Barraquand, J., Latombe, J.C.: Robot motion planning: a distributed representation approach. *International Journal of Robotics Research* (1991)
- [12] Bellman, R.: On a routing problem. *Quarterly of applied mathematics* **16**(1), 87–90 (1958)
- [13] Breuker, D.M., Uiterwijk, J.W., van den Herik, H.J.: Replacement schemes for transposition tables. *ICGA Journal* **17**(4), 183–193 (1994)
- [14] Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* pp. 293–318 (1992)
- [15] Burns, E., Lemons, S., Ruml, W., Zhou, R.: Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research (JAIR)* **39**, 689–743 (2010)

- [16] Burns, E.A., Hatem, M., Leighton, M.J., Ruml, W.: Implementing fast heuristic search code. In: Proceedings of the Annual Symposium on Combinatorial Search (2012)
- [17] Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. Society for Industrial and Applied Mathematics (1995)
- [18] Chiappa, S., Racaniere, S., Wierstra, D., Mohamed, S.: Recurrent environment simulators. arXiv preprint arXiv:1704.02254 (2017)
- [19] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press (2001)
- [20] Cui, X., Shi, H.: A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security **11**(1), 125–130 (2011)
- [21] Culberson, J.: SOKOBAN is PSPACE-complete. Tech. rep., Department of Computing Science, University of Alberta (1997)
- [22] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
- [23] Do, M.B., Kambhampati, S.: Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. Artificial Intelligence **132**(2), 151–182 (2001)
- [24] Edelkamp, S.: Planning with pattern databases. In: European Conference on Planning (ECP), pp. 13–24 (2001)
- [25] Edelkamp, S., Reffel, F.: Obdds in heuristic search. In: Annual Conference on Artificial Intelligence, pp. 81–92. Springer (1998)

- [26] Edelkamp, S., Reffel, F.: Deterministic state space planning with bdds (1999)
- [27] Edelkamp, S., Schroedl, S.: Heuristic Search: Theory and Applications. Morgan Kaufmann Publishers Inc. (2010)
- [28] Edgar, R.C., Batzoglou, S.: Multiple sequence alignment. Current opinion in structural biology **16**(3), 368–373 (2006)
- [29] Erdem, E., Tillier, E.: Genome rearrangement and planning. In: Proceedings of the AAAI Conference on Artificial (AAAI), pp. 1139–1144 (2005)
- [30] Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic SAT-compilation of planning problems. In: Proceedings of the International Joint Conference on Intelligence (IJCAI), vol. 97, pp. 1169–1176 (1997)
- [31] Fikes, R.E., Nilsson, N.: STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence **5**(2), 189–208 (1971)
- [32] Ford Jr, L.R.: Network flow theory. Tech. rep., Rand Corp Santa Monica Ca (1956)
- [33] Fox, M., Long, D.: PDDL2. 1: An extension to PDDL for expressing temporal planning domains. Journal of artificial intelligence research (2003)
- [34] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3), 596–615 (1987)
- [35] Fukunaga, A., Botea, A., Jinnai, Y., Kishimoto, A.: A survey of parallel a* (2017)

- [36] Geffner, T., Geffner, H.: Width-based planning for general video-game playing. In: The IJCAI-15 Workshop on General Game Playing, pp. 15–21 (2015)
- [37] Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Elsevier (2004)
- [38] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc. (1989)
- [39] Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press (1997)
- [40] Hansen, E.A., Zhou, R.: Anytime heuristic search. *Journal of Artificial Intelligence Research* **28**, 267–297 (2007)
- [41] Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics* pp. 100–107 (1968)
- [42] Helmert, M.: The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)* **26**, 191–246 (2006)
- [43] Helmert, M., Haslum, P., Hoffmann, J., Nissim, R.: Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM* **61**(3), 16 (2014)
- [44] Helmert, M., Lasinger, H.: The scanalyzer domain: Greenhouse logistics as a planning problem. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) (2010)
- [45] Helmert, M., Roger, G.: How good is almost perfect? In: Proceedings of the 23rd National Conference on Artificial Intelligence, pp. 944–949 (2008)

- [46] Hoffmann, J.: Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research* **24**, 685–758 (2005)
- [47] Hoffmann, J., Kissmann, P., Torralba, A.: Distance? who cares? tailoring merge-and-shrink heuristics to detect unsolvability. In: *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, pp. 441–446 (2014)
- [48] Holte, R.C., Felner, A., Sharon, G., Sturtevant, N.R.: Bidirectional search that is guaranteed to meet in the middle. In: *Thirtieth AAAI Conference on Artificial Intelligence* (2016)
- [49] Holte, R.C., Newton, J., Felner, A., Meshulam, R., Furcy, D.: Multiple pattern databases. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 122–131 (2004)
- [50] Ikeda, T., Imai, H.: Enhanced algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoretical Computer Science* **210**(2), 341–374 (1999)
- [51] Irani, K., Shih, Y.: Parallel A* and AO* algorithms: An optimality criterion and performance evaluation. In: *International Conference on Parallel Processing*, pp. 274–277 (1986)
- [52] Jinnai, Y., Fukunaga, A.: Abstract Zobrist hash: An efficient work distribution method for parallel best-first search. In: *Proceedings of the AAAI Conference on Artificial (AAAI)*, pp. 717–723 (2016)
- [53] Jinnai, Y., Fukunaga, A.: On work distribution functions for parallel best-first search. *Journal of Artificial Intelligence Research (JAIR)* (2017)
- [54] Johnson, W.W., Story, W.E., et al.: Notes on the 15 puzzle. *American Journal of Mathematics* **2**(4), 397–404 (1879)

- [55] Junghanns, A., Schaeffer, J.: Sokoban: A challenging single-agent search problem. In: In IJCAI Workshop on Using Games as an Experimental Testbed for AI Research (1997)
- [56] Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. *Artificial intelligence* **101**(1-2), 99–134 (1998)
- [57] Kishimoto, A., Fukunaga, A., Botea, A.: Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence* **195**, 222–248 (2013)
- [58] Kishimoto, A., Schaeffer, J.: Distributed game-tree search using transposition table driven work scheduling. In: *Proceedings of the 31st International Conference on Parallel Processing ICPP-02*, pp. 323–330 (2002)
- [59] Korf, R.: Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* **97**, 97–109 (1985)
- [60] Korf, R.E., Felner, A.: Disjoint pattern database heuristics. *Artificial Intelligence* **134**(1), 9–22 (2002)
- [61] Korf, R.E., Schultze, P.: Large-scale parallel breadth-first search. In: *Proceedings of the AAAI Conference on Artificial (AAAI)*, pp. 1380–1385 (2005)
- [62] Korf, R.E., Zhang, W.: Divide-and-conquer frontier search applied to optimal sequence alignment. In: *Proceedings of the 17th National Conference on Artificial Intelligence*, pp. 910–916 (2000)
- [63] Lehman, J., Stanley, K.O.: Exploiting open-endedness to solve problems through the search for novelty. In: *ALIFE*, pp. 329–336 (2008)

- [64] Likhachev, M., Gordon, G.J., Thrun, S.: ARA*: Anytime A* with provable bounds on sub-optimality. In: Advances in neural information processing systems, pp. 767–774 (2004)
- [65] Lipovetzky, N., Geffner, H.: Width and serialization of classical planning problems. In: European Conference on Artificial Intelligence, pp. 540–545 (2012)
- [66] Lipovetzky, N., Ramirez, M., Geffner, H.: Classical planning with simulators: Results on the Atari video games. In: Proceedings of the International Joint Conference on Intelligence (IJCAI), pp. 1610–1616 (2015)
- [67] McDowell, G.L.: Cracking the coding interview. CarrerCup (2011)
- [68] McQuillan, J., Richer, I., Rosen, E.: The new routing algorithm for the arpanet. IEEE Transactions on Communications **28**(5), 711–719 (1980)
- [69] Mehlhorn, K., Meyer, U.: External-memory breadth-first search with sub-linear i/o. Algorithms ESA 2002 pp. 21–26 (2002)
- [70] Miura, T., Ishida, T.: Stochastic node caching for memory-bounded search. In: Proceedings of the AAAI Conference on Artificial (AAAI) (1998)
- [71] Nash, A., Daniel, K., Koenig, S., Felner, A.: Theta: Any-angle path planning on grids. In: Proceedings of the AAAI Conference on Artificial (AAAI), pp. 1177–1183 (2007)
- [72] Oh, J., Guo, X., Lee, H., Lewis, R.L., Singh, S.: Action-conditional video prediction using deep networks in atari games. In: Advances in Neural Information Processing Systems, pp. 2863–2871 (2015)
- [73] Pearl, J.: Heuristics - Intelligent Search Strategies for Computer Problem Solving. Addison–Wesley (1984)

- [74] Pearson, W.R.: Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in enzymology* **183**, 63–98 (1990)
- [75] Puterman, M.L.: *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons (2014)
- [76] Reinefeld, A., Marsland, T.A.: Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **16**(7), 701–710 (1994)
- [77] Richter, S., Thayer, J.T., Ruml, W.: The joy of forgetting: Faster anytime search via restarting. In: *Twentieth International Conference on Automated Planning and Scheduling* (2010)
- [78] Russell, S., Norvig, P.: *Artificial Intelligence: A Modern Approach*, 2nd edition edn. Prentice-Hall, Englewood Cliffs, NJ (2003)
- [79] Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* **219**, 40–66 (2015)
- [80] Silver, D., van Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., Dulac-Arnold, G., Reichert, D., Rabinowitz, N., Barreto, A., et al.: The predictron: End-to-end learning and planning. *arXiv preprint arXiv:1612.08810* (2016)
- [81] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. *Nature* **529**(7587), 484–489 (2016)
- [82] Skiena, S.S., Revilla, M.A.: *Programming challenges: The programming contest training manual*. Springer Science & Business Media (2006)

- [83] Sousa, A., Tavares, J.: Toward automated planning algorithms applied to production and logistics. IFAC Proceedings Volumes **46**(24), 165–170 (2013)
- [84] Sturtevant, N.R.: Benchmarks for grid-based pathfinding. IEEE Transactions on Computational Intelligence and AI in Games **4**(2), 144–148 (2012)
- [85] Sutton, R.S., Barto, A.G., et al.: Introduction to reinforcement learning, vol. 2. MIT press Cambridge (1998)
- [86] Tarjan, R.E.: Data structures and network algorithms. Society for Industrial and Applied Mathematics (1983)
- [87] Vallati, M., Chrpá, L., Grześ, M., McCluskey, T.L., Roberts, M., Sanner, S., et al.: The 2014 international planning competition: Progress and trends. Ai Magazine **36**(3), 90–98 (2015)
- [88] Waterman, M.S.: Introduction to computational biology: maps, sequences and genomes. CRC Press (1995)
- [89] Wilt, C.M., Thayer, J.T., Ruml, W.: A comparison of greedy search algorithms. In: third annual symposium on combinatorial search (2010)
- [90] Wolpert, D.H., Macready, W.G., et al.: No free lunch theorems for optimization. IEEE transactions on evolutionary computation **1**(1), 67–82 (1997)
- [91] Yap, P.: Grid-based path-finding. In: Conference of the Canadian Society for Computational Studies of Intelligence, pp. 44–55. Springer (2002)
- [92] Zobrist, A.L.: A new hashing method with applications for game playing. Tech. rep., Dept of CS, Univ. of Wisconsin, Madison (1970). Reprinted in *International Computer Chess Association Journal*, 13(2):169–173, 1990

A* search, 44
action scheme, 98
admissible heuristic, 41
Anytime A*, 69
anytime algorithm, 69
automated planning problem, 95

best-first search, 54
bidirectional search, 67, 75
Binary Decision Diagram, 68, 80
binary decision diagram, 82
blind search, 41
Branch-and-Bound, 34
branch-and-bound, 67, 68
branching factor, 22

chaining, 61

characteristic function, 81
closed list, 30
combinatorial optimization, 36
communication overhead, 88
consistent heuristic, 41
coordination overhead, 90

delayed duplicate detection, 32, 64
depth first iterative deepening, 70
deterministic, 10
dictionary, 61
Dijkstra's Algorithm, 35
duplicate, 29
duplicate detection, 29

enforced hill-climbing, 52
explicit state-space graph, 13

- external breadth-first search, 78
- External Search, 76
- Fibonacci heap, 58
- Greedy Best-First Search, 50
- grid path-finding problem, 14
- hash collision, 61
- hash function, 61
- hash table, 61
- heuristic function, 39, 44
- heuristic search, 39
- hill climbing, 51
- immediate duplicate detection, 63
- implicit state-space graph, 14
- incremental hashing, 63
- instance, 98
- irreversible action, 20
- iterative deepening A*, 67
- iterative width search, 87
- lexicographical order, 63
- load balance, 89
- local search, 51
- lookahead, 8
- Markov Decision Process Problem, 23
- merge and shrink, 104
- minimum spanning tree, 48
- monotone heuristic, 42
- Multiple Sequence Alignment, 17
- novelty, 86
- object, 98
- observation, 10
- open addressing, 61
- parallel search, 68
- partial information, 10
- partially observable Markov decision process problem, 23
- path cost, 13
- Pattern database heuristic, 103
- perfect hash function, 63
- Perfect Heuristic, 40
- perfect information, 10
- perfect linear speedup, 88
- plan, 10
- predicate, 98
- problem, 97
- pruning, 34
- radix heap, 58
- reexpansion, 28, 31, 46
- reinforcement learning, 23
- relaxed problem, 41
- reversible, 20
- reward, 23
- solution, 12

state-space graph, 12
state-space problem, 10, 11
suboptimal search, 49
suboptimal solution, 49
symbolic representation, 81
symbolic search, 68, 80

tie-breaking, 59
tiebreaking, 58

weighted A^* , 49
width-based search, 86

zero-cost action, 20

アクションスキーム, 98

インスタンス, 98

A^* 探索, 44

枝刈り, 34

オブジェクト, 98

重み付き A^* 探索, 49

解, 12

開番地法, 61

外部メモリ探索, 76

外部メモリ幅優先探索, 78

可逆, 20

観察, 10

完全情報, 10

完全線形高速化, 88

完全ハッシュ, 63

完璧なヒューリスティック, 40

緩和問題, 41

基数ヒープ, 58

強化学習, 23

強制山登り法, 52

局所探索, 51

許容的なヒューリスティック, 41

組み合わせ最適化, 36

グリッド経路探索問題, 14

クローズドリスト, 30

経路コスト, 13

決定論的, 10

最小全域木, 48

再展開, 28, 31, 46

最良優先探索, 54

先読み, 8

差分ハッシュ, 63

辞書, 61

辞書順, 63

自動行動計画問題, 95

述語, 98

状態空間グラフ, 12

状態空間問題, 10, 11

衝突, 61

- 新奇性, 86
- シンボリック探索, 68, 80
- ゼロコストアクション, 20
- 即時重複検出, 63
- ダイクストラ法, 35
- タイブレーキング, 58, 59
- 多重整列問題, 17
- 単調なヒューリスティック, 42
- 遅延重複検出, 32, 64
- 重複, 29
- 重複検出, 29
- 通信オーバーヘッド, 88
- 同期オーバーヘッド, 90
- 特徴関数, 81
- 特徴表現, 81
- 貪欲最良優先探索, 50
- 二分決定グラフ, 68, 80, 82
- 任意時間アルゴリズム, 69
- 任意時間 A^* , 69
- パターンデータベースヒューリスティック, 103
- ハッシュ関数, 61
- ハッシュテーブル, 61
- 幅制限探索, 86
- 反復深化 A^* , 67
- 反復深化深さ優先, 70
- 反復幅制限探索, 87
- 非最適解, 49
- 非最適探索, 49
- 非明示的状态空間グラフ, 14
- ヒューリスティック関数, 39, 44
- ヒューリスティック探索, 39
- フィボナッチヒープ, 58
- 不可逆なアクション, 20
- 不完全情報, 10
- 部分観測マルコフ過程問題, 23
- ブラインド探索, 41
- プラン, 10
- 分枝限定法, 34, 67, 68
- 分枝度, 22
- 並列探索, 68
- 報酬, 23
- マージアンドシュリンク, 104
- マルコフ過程問題, 23
- 無矛盾なヒューリスティック, 41
- 明示的状态空間グラフ, 13
- 問題, 97
- 山登り法, 51

両方向探索, 67, 75

連鎖法, 61

ロードバランス, 89