ヒューリスティック探索

陣内　佑
ブラウン大学 (ddyuudd@gmail.com)

April 7, 2018

Contents

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 7
- 1.2 本書の射程 8

2 状態空間問題 (State-Space Problem) **11**
- 2.1 状態空間問題 (State-Space Problem) 11
- 2.2 状態空間問題の例 13
  - 2.2.1 グリッド経路探索 (Grid Path-finding) 14
  - 2.2.2 スライディングタイル (Sliding-tile Puzzle) 14
  - 2.2.3 多重整列問題 (Multiple Sequence Alignment) 15
  - 2.2.4 倉庫番 (Sokoban) 16
  - 2.2.5 巡回セールスパーソン問題 (Traveling Salesperson Problem, TSP) 18
- 2.3 問題の性質・難しさ 19
- 2.4 関連文献 20

3 情報なし探索 (Blind Search) **21**
- 3.1 木探索アルゴリズム (Tree Search Algorithm) 21
- 3.2 グラフ探索アルゴリズム (Graph Search Algorithm) 23
- 3.3 幅優先探索 (Breadth-First Search) 25
- 3.4 深さ優先探索 (Depth-First Search) 26
  - 3.4.1 再帰による深さ優先探索 27
- 3.5 ダイクストラ法 (Dijkstra Algorithm) 27
- 3.6 関連文献 28
  - 3.6.1 No Free Lunch とヒューリスティック 28

4 ヒューリスティック探索 (Heuristic Search) **29**
- 4.1 ヒューリスティックとは？ 29
- 4.2 ヒューリスティック関数 (Heuristic Function) 29
- 4.3 A*探索 (A* Search) 31
  - 4.3.1 重み付き A*探索 (Weighted A*) 33
  - 4.3.2 貪欲最良優先探索 (Greedy Best-First Search) 34
- 4.4 ヒューリスティック関数の例 35
  - 4.4.1 グリッド経路探索：マンハッタン距離 35

4.4.2 スライディングタイル:マンハッタン距離 35
4.4.3 巡回セールスパーソン問題：最小全域木 35
4.5 上手く行かない場合 35
4.6 関連文献 36

5 自動行動計画問題 (Automated Planning Problem) 37
5.1 定義 37
5.2 Planning Domain Definition Language 38
5.3 古典的プランニングモデルの例 39
5.4 ヒューリスティック関数の自動生成 42
5.4.1 ゴールカウントヒューリスティック (Goal Count Heuristic) 42
5.4.2 パターンデータベースヒューリスティック (Pattern Database Heuristic) 42
5.5 関連文献 43

6 グラフ探索のためのデータ構造 45
6.1 オープンリスト (Open List) 45
6.1.1 プライオリティキュー (Priority Queue) 46
6.1.2 タイブレーキング (Tiebreaking) 46
6.2 クローズドリスト (Closed List) 47
6.2.1 ハッシュテーブル (Hash Table) 47
6.2.2 遅延重複検出 (Delayed Duplicate Detection) 47
6.2.3 ハッシュ関数 (Hash Function) 48
6.3 関連文献 48

7 局所探索 (Local Search) 49
7.1 山登り法 (Hill Climbing) 49
7.1.1 強制山登り法 (Enforced Hill Climbing) 49
7.2 関連文献 50

8 ヒューリスティック探索の派生 51
8.1 分枝限定法 (Branch-and-Bound) 52
8.2 反復深化深さ優先探索 (Depth First Iterative Deepening) 53
8.2.1 反復深化 A* (Iterative Deepening A*) 53
8.2.2 Transposition Table 54
8.3 両方向探索 (Bidirectional Search) 55
8.4 Symbolic Search 55
8.4.1 特徴表現 (Symbolic Representation) 56
8.4.2 Binary Decision Diagram 57
8.4.3 特徴関数による状態空間の探索 57
8.4.4 関連文献 59
8.5 新奇性に基づく枝刈り (Novelty-based Pruning) 59
8.5.1 状態の新奇性 (Novelty) 59
8.5.2 Width-Based Search (幅制限探索) 59
8.5.3 Iterative Width Search (反復幅制限探索) 60
8.5.4 関連文献 60

8.6 外部メモリ探索 (External Search) 60
8.6.1 関連文献 61
8.7 並列探索 (Parallel Search) 61
8.7.1 並列化オーバーヘッド (Parallel Overheads) 62
8.7.2 ハッシュ分配 A* (Hash Distributed A*) 63
8.7.3 関連文献 64
8.8 モンテカルロ木探索 (Monte Carlo Tree Search) 65
8.8.1 UCT (UCB1 Applied to Tree) 65
8.9 その他の派生アルゴリズム 66

Chapter 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**直感** (ヒューリスティック) 的に正しそうな行動を絞り込むことが出来るからである。

かつて人工知能の研究者らの一部は人間が直感と呼ぶものをコンピュータに実装することによってこれが実現できるのではないか、と考えた。まだ現在、「ごはんをよそう」という指令だけを受け、先にどういうことが起きるかを予想して、上記の行動を計画し、そして実際に実行するような高度なシステムは実装されていない。このようなエージェントを実装する方法は様々あるかもしれないが、その一つに探索アルゴリズム (を一要素に含む技術) があるだろう。

人工知能という言葉は、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を作るという野望が言葉になったものである！

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 [58] 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ィアである。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 `https://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/`。

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 [61]。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 [55, 60, 12]。

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。

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

[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

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。

本書が主に扱うの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 8 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、簡単で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 infomation) 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**??**章で扱う。

本書が扱う完全情報決定論的モデルは最も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。

Chapter 2

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 [58]、[56]、[18] 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**??**章までで扱う**状態空間問題** (state-space problem) である。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 $A = a_1, ..., a_n$、$a_i : S \to S$ がある。アクションはある状態を次の状態に遷移させる関数である。状態空間問題の解は初期状態からゴール状態へ遷移させるアクションの列を求めることである。

よって、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。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。

Definition 1 (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。

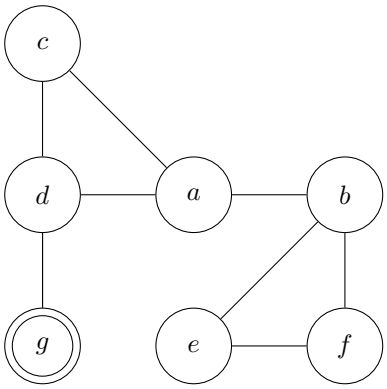


Figure 2.1: 状態空間問題の例。エージェントはスタート地点 a からゴール地点 g を目指す。
問題は [18] より。

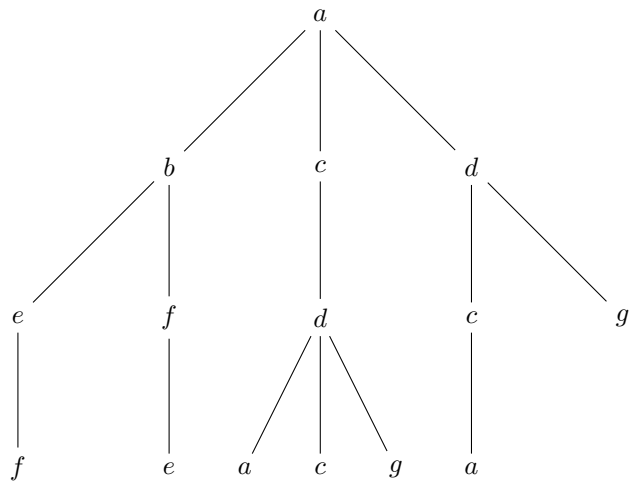


Figure 2.2: 状態空間問題の経路木。エージェントはスタート地点 a からゴール地点 g を目指す。

Definition 2 (解、Solution)**.** 解 $\pi = (a_1, a_2..., a_k)$ 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、$u_i \in S, i \in \{0, 1, ..., k\}$, $u_0 = s, u_k = t$ 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。

Definition 3 (コスト付き状態空間問題、Weighted state-space problem)**.** コスト付き状態空間問題 $P = (S, A, s, T, w)$ は状態空間問題の定義に加え、コスト関数 $w : A \to \mathbb{R}$ がある。経路 $(a_1, ..., a_k)$ のコストは $\sum_{i=1}^{k} w(a_i)$ と定義される。ある解が可能なすべての解の中でコストが最小である場合、その解を最適解 *(optimal cost solution)* であると言う。

本書ではコスト付き状態空間問題をメインの問題として考える。

コストの定義されていない状態空間問題を特に区別して**ユニットコスト** (unit-cost) 問題 (ドメイン) と呼ぶ。コスト付き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 $w : E \leftarrow \mathbb{R}$ 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**明示的状態空間グラフ** (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**非明示的状態空間グラフ** (implicit state-space graph) で表せられる。

Definition 4 (非明示的状態空間グラフ、Implicit state-space graph)**.** 非明示的状態空間グラフは初期状態 $s \in V$、ゴール条件 *Goal:* $V \to B = \{false, true\}$、ノード展開関数 *Expand:* $V \to 2^V$ によって与えられる。

探索の開始時、エージェントは初期ノードのみを保持する。エージェントは保持しているノードに対して Expand を適用することによって、新しいノードとエッジをグラフに加える。これを求める解を見つけるまで繰り返す。Expand はある状態からの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。これによって、エージェントは解を見つけるまでのノード・エッジだけ保持して必要な解を見つけることが出来る。

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。

Figure 2.3: グリッド経路探索問題

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 [68]。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 [54]。

Web 上に簡単に試せるデモがあるので、参照されたい[1]。この本の画像の一部はこのデモをもとに作成している。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 [2]。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 [14]。よく使われるベンチマーク問題集にも Starcraft のゲームのマップが含まれている [64]. 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 (後述)。ロボットのモーションプランニングも経路探索に帰着することが出来る [7]。すなわち、k 個の関節の角度を変えて、現在状態からゴール状態へ遷移させたい。各関節の角度がグリッドの各次元に相当する。ロボットの物理的な構造により、関節のある角度の組み合わせは不可能である。不可能な組み合わせが、障害物の置かれたグリッドに相当する。よって、障害物を避けた経路というのが関節の動かし方ということになる。

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

[1] http://qiao.github.io/PathFinding.js/visual/

Figure 2.4: 15 パズルのゴール状態の例

ク問題でもある (図 2.4) [41]。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 $|V| = (n^2)!/2^2$[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ので、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) を比較することでニ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

[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 [41]。

は様々考えられるが、ここでは全ての配列のペアの編集距離の総和を用いる。

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 (ABC, BCB, CB) の 3 つの配列のアライメントを考える。図 2.6b は置換と編集に対するコストの例である。-は欠損、すなわち挿入操作が行われたことを示す。アミノ酸配列における有名なコスト表として PAM250[57] があるが、ここでは簡単のため仮のコスト表を用いる。図 2.6a 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 ABC-と-BCB の編集距離は (A,-)、 (B,B)、 (C,C)、 (-,B) のコストの総和であるので、図 2.6b を参照し、$5+0+1+5=11$ である。(-BCB, –CB) の距離は 6, (–CB, ABC-) の距離は 16 であるので、 3 配列の編集距離は $11+6+16=33$ 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 [47]。図 2.6c は (ABC) と (BCB) の 2 つの配列による問題を表す。状態 s は 2 つの変数によって表現される:(x_0, x_1)。x_0 は配列 0 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 $0 \leq x_0 \leq l_0$ である。全てのアライメントが完了した状態 $s=(l_0, l_1)$ がゴール状態である。可能なアクションは $a=(b_0, b_1), (b_i=0, 1)$ の形を取り、これは配列 i に対して欠損を挿入する場合に $b_i=0$ 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 $s'=(x_0+b_0, x_1+b_1)$ となる。例えば図 2.6c 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,B), (-,B) という状態に至る。

このようにして、MSA はグリッド経路探索問題に帰着し、グラフ探索アルゴリズムよって解くことが出来る。状態空間問題として考えた場合に MSA の難しさはアクションのコストが幅広いことにある。また、可能なアクションの数も配列の数 n に対して 2^n-1 と大きい。

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 [66, 28, 19] を参照されたい。

2.2.4 倉庫番 (Sokoban)

倉庫番 (Sokoban) は Atari などでパズルゲームであり、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 [42, 15]。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 [15]。

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。

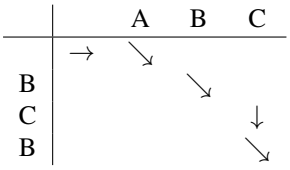
Figure 2.5: 多重整列問題: 画像は wikipedia より。

A	B	C	-
-	B	C	B
-	-	C	B

(a) MSA の解の例

	A	B	C	-
A	0	1	2	5
B		0	3	5
C			1	5
-				0

(b) 操作コスト表の例



(c) グリッド経路探索への帰着

A	B	C	-
-	B	C	B

(d) グリッド経路探索への帰着

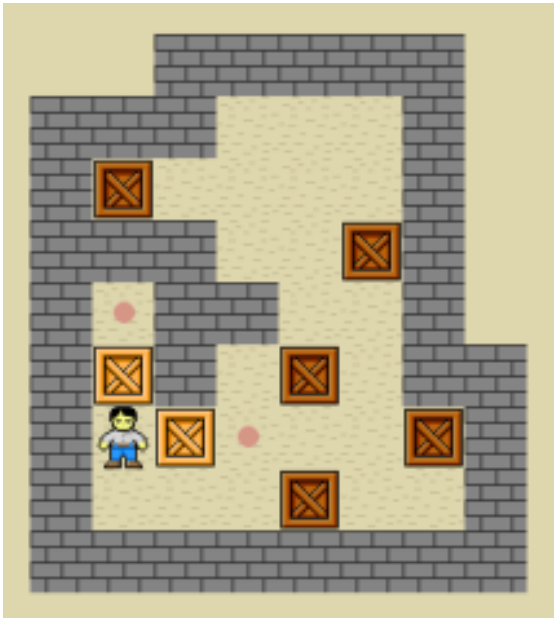


Figure 2.6: Sokoban: 画像は wikipedia より。

グラフ探索問題として倉庫番を考えるときに重要であるのは、倉庫番は不可逆なアクション (irreversible) があることである。グリッド経路探索やスライディングタイルは可逆な (reversible) 問題である。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、問題は可逆であると言う。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出来、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、詰み (trap) 状態に陥る可能性がある (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。

もうひとつ重要な問題は**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。ヘタ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)

セールスパーソンはいくつかの都市に回って営業を行わなければならない。都市間の距離 (=コスト) は事前に与えられている。TSP は全ての都市を最短距離で回っ

Figure 2.7: 巡回セールスパーソン問題: 画像は wikipedia より。

てはじめの都市に戻る経路を求める、という問題である [3]。

n 個の都市があるとすると (最適・非最適含む) 解の数は $(n-1)!/2$ 個である。可能なアクションは「都市 $i \in \{1..n\}$ 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、n 回どれか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。一方、最適解を得る問題は NP 完全であることが知られている。

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 [48, 30]。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。

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問題ドメインである [3]。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 4.3.2 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。そのような場合、アルゴリズムは解が存在しないと示せれば理想的である。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 [6, 35]。

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 [**?**]。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 [**?**, **?**] 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) ($R : (S, A) \to \mathbb{R}$) 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。

Chapter 3

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。特に人工知能の文脈において、なるべく問題の知識を必要とせず、最小限の仮定のみを必要とする解法が求められる。

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 [62]。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 [52]。情報なし探索は [13] 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。

以下、(k) と書いて疑似コードの k 行目を指すことにする。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、

[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 6 章で紹介する。

Table 3.1: 木探索とグラフ探索

Table 3.2: 木探索とグラフ探索の違い。状態の重複検出を行わない手法を木探索と呼ぶ。探索する状態空間が木であるか否かは関係しない。グラフ探索は重複検出を行うことで同じ状態を複数回無駄に探索することを防ぐ。重複検出のためには生成済みノードをすべて保存するデータ構造クローズドリストが必要になる。

	重複検出	保存するノード	完全性
木探索	重複検出しない	オープンリストのみ	ループを含むグラフである場合停
グラフ探索	重複検出する	オープンリストとクローズドリスト	(状態空間が有限なら

Table 3.3: ノードの展開順序。

展開順序	プライオリティ	性質
幅優先	$\arg\min_n d(n)$	ユニットコストドメインだと最初に発見した解が最適解である
深さ優先	$\arg\max_n d(n)$	メモリ消費が少ない場合がある
最良優先	$\arg\min_n g(n)$	非負コストドメインで最適解が得られる

Algorithm 1: 木探索 (Implicit Tree Search)

Input : initial node s, weight function w, successor generation function $Expand$, goal function $Goal$
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

```
1  Open ← {s};
2  while Open ≠ ∅ do
3  |  u ← Open.pop();
4  |  if Goal(u) then
5  |  |  return Path(u);
6  |  Succ(u) ← Expand(u);
7  |  for each v ∈ Succ(u) do
8  |  |  Open.insert(v);
9  |  |  parent(v) ← u;
10 return ∅;
```

Algorithm 2: Expand

Input : Parent node s, a set of actions applicable to the state $A(s)$
Output : A set of child nodes S

```
1  for a ∈ A(s) do
2  |  s' ← a(s);
3  |  d(s') ← d(s) + 1;
4  |  g(s') ← g(s) + w(s, s');
5  |  S = S ∪ s';
6  return S;
```

初期状態のみがオープンリストに入っている (1)。木探索は、このオープンリストから一つノード u を選び (3)、ゴール条件を満たしているかを確認する (4)。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 (6-)。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 (8) ことを指す。

アルゴリズム 2 は展開関数の動作を表している。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。

探索の進行によってエージェントが保持する情報は変化していく。ここでは探索がどのように進行するかを記述するため、以下の 3 つの言葉を定義する：

1. 展開済みノード: $Expand$ によって子ノードが参照されたノードを指す。$Open$ からは取り除かれたノードである。
2. 生成済みノード: $Open.insert$ によって $Open$ に一度でも入れられたノードを指す。
3. 未生成ノード: 状態空間内のまだ生成されていないノード。よって、非明示的グラフに保持されていない。

非明示的グラフ木探索の強みは、生成済みノードのうち展開済みではないもののみを $Open$ 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、これの問題は、一度展開したノードが再び現れた場合、**再展開 (reexpansion)** をすることになる。よって、グラフがより木から遠いほど (複数の経路で到達可能なノードがあるほど) 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。もっと言えば、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 (6.2.2 節) を使うと良いだろう。

紛らわしいが、木探索アルゴリズムはグラフを探索するアルゴリズムである。グラフ探索アルゴリズムのうち、重複検出を行わない手法を木探索アルゴリズムと呼ぶ。疑似コードでは明示されていないが、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 ($Open.pop$ の実装にあたる)。探索アルゴリズムの性能はこのプライオリティの設定方法に強く依存する。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木探索と区別する。

Algorithm 3: グラフ探索 (Implicit Graph Search)

Input : Implicit problem graph with initial node s, weight function w, successor generation function $Expand$, goal function $Goal$
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

```
1  Closed ← ∅;
2  Open ← {s};
3  while Open ≠ ∅ do
4  |  u ← Open.pop();
5  |  if Goal(u) then
6  |  |  return Path(u);
7  |  Succ(u) ← Expand(u);
8  |  for each v ∈ Succ(u) do
9  |  |  if Closed.find(v.state)isnull then
10 |  |  |  Open.insert(v);
11 |  |  |  parent(v) ← u;
12 |  |  |  Closed.insert(u);
13 |  |  else
14 |  |  |  v' ← Closed.find(v.state);
15 |  |  |  if v.g < v'.g then
16 |  |  |  |  Open.insert(v);
17 |  |  |  |  parent(v) ← u;
18 |  |  |  |  Closed.insert(v);
19 |  |  |  |  Open.remove(v');
20 |  |  |  |  Closed.remove(v');
21 return ∅;
```

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ができる場合も存在する。まず、解の最適性が必要でない場合 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ができる。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 6.2 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とき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、**??**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 ($Open.pop()$)。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 $Open.pop()$ はアルゴリズム 4 のように書くことが出来る。ユニットコスト問題である場合、d 値は g 値と一致する。

幅優先探索のメリットは初めに発見した解が最短経路長であることである。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)。

Algorithm 4: Breadth-First Search: $Open.pop()$

Output : Node n
return $\arg\min_n d(n)$

Table 3.4: 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 ([18] 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	
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}	ゴールを発見

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 3.4 のように遷移する。

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。

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 3.5 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 2.2 と合わせてノードが展開される順序を確認すると良い。

良い解、最適解を見つけた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 $Open.insert(v)$ をスキップする。詳しくは**??**章で解説する。

Algorithm 5: Depth-First Search: $Open.pop()$

Output : Node n
return $\arg\max_n g(n)$

Table 3.5: 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 ([18] 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	
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}	ゴールを発見

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。一般的にアルゴリズム 1、7 に従った方法の実装は効率的ではない [2]。深さ優先探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 6)。ここでこのアルゴリズムにはオープンリストがないことに注目したい。これは実装が簡単というだけではなく、消費メモリが少なく、実行時間が少ないというメリットがある。

Algorithm 6: Depth-First Search (DFS)

Input : Node s, weight function w, successor generation function $Expand$, goal function $Goal$
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

```
1 if Goal(s) then
2 |   return s;
3 Succ(u) ← Expand(u);
4 for each v ∈ Succ(u) do
5 |   if DFS(v) ≠ ∅ then
6 |   |   return (s, DFS(v))
7 return ∅;
```

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 [16]。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 [53]。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。つまりグラフ探索の疑似コード 3 の $Open.pop$ を以下のように実装すればダイクストラ法である。

[2] 深さ優先探索を並列化する場合はオープンリストを用いる場合がある

Algorithm 7: Best-First Search: $Open.pop()$

Output : Node u

1 **return** $\arg\max_n g(n)$

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 [25] を用いてオープンリストを実装したダイクストラ法は $O(|E| + |V|log|V|)$ 時間でであることが知られている [25]。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 [8, 23]。

ダイクストラ法などの情報なし探索は人工知能の文脈よりも**組み合わせ最適化** (combinatorial optimization) のための手法として使われることが多い [65]。

3.6.1 No Free Lunch とヒューリスティック

No Free Lunch 定理 [**?**] 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 定理の主張は以下である。

Theorem 1. すべての可能なコスト関数による状態空間問題の集合を考える。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。

問題に対して知っている知識を利用して効率化をすることができる。すなわち、ある問題集合への性能を犠牲に、ある問題集合への性能を向上させることができる。与えられている問題の特徴を知っていれば、その問題に対する性能の良いアルゴリズムを選ぶことができる。それが次章で扱うヒューリスティック探索の行うことである。

Chapter 4

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？

経路探索問題を幅優先探索で解くことを考えよう。図**??**の初期状態からゴールへの最短経路の長さは X である。このとき、幅優先探索は図**??**の領域を探索する。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。なぜか。それは人間が問題の特徴を利用して、このノードを探索したほうがよいだろう、という推論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、ヒューリスティック関数を利用した探索アルゴリズムをヒューリスティック探索と呼ぶ。ヒューリスティック関数は人間が自分の知識を利用してコーディングする場合もあるが、自動行動計画問題などでは自動的にヒューリスティックを生成する手法も広く使われている。

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ある状態からゴールまでの最短距離の見積もりである [29]。

Definition 5 (ヒューリスティック関数)**.** ヒューリスティック関数 h はノードの評価関数である。$h : V \to \mathbb{R}_{\geq 0}$

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 値と呼ぶことが多い。

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。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。このようなヒューリスティックをブラインドヒューリスティックと呼ばれる。ブラインドヒューリスティックを使った探索は、3 章で扱った情報なし探索の一種といえる。

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、多くの場合これに近いほど必要な展開ノード数が小さいことが知られている [34]。

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と呼ぶ。

Definition 6 (許容的なヒューリスティック)**.** 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**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]。

もう一つ重要な性質は無矛盾性である。

Definition 7 (無矛盾なヒューリスティック)**.** 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。

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。

Theorem 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. あるノード n_0 からゴールノード $n_k \in T$ への最短経路 (ノードの列) を

[1] 解が多いほど簡単であるとは一概には言えないが

$(n_0, n_1, ...)$ と置く。無矛盾なヒューリスティック $h(n)$ は

$$\begin{aligned} h(n_0) &\leq h(n_1) + w(n_0, n_1) && (4.1)\\ &\leq h(n_2) + w(n_0, n_1) + w(n_1, n_2) && (4.2)\\ &... && (4.3)\\ &\leq h(n_k) + \sum_{i=0..k-1} (w(n_i, n_{i+1})) && (4.4)\\ &= h^*(n_0) && (4.5) \end{aligned}$$

よって $h(n_0) \leq h^*(n_0)$ より許容的である。

□

後述する A*探索においてこれら二つの性質は非常に有用である。

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**??**はダイクストラ法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無視して探索を進めているため、図のように探索空間が広がっていく。

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) を用いることで図**??**のように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 [22]。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。

$$f(n) = g(n) + h(n) \tag{4.6}$$

ノード n の f 値は、初期状態から n を通過してゴール状態に辿り着くためのコストの見積もりである。g 値は初期状態からノード n 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 n 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 3) の一つであり、$Open.pop()$ を f 値最小ノードを返すようにしただけである 8。

Algorithm 8: A* Search: $Open.pop()$

Output : Node n

1 **return** $\arg\min_n g(n) + h(n)$

$g(n)$ のみでノードを選択するダイクストラ法 (3.5 章) と比較すると、A*探索はゴール状態までのコストの見積もりを考慮して次に展開するノードを決めている。

Figure 4.1: A*探索

Figure 4.2: ダイクストラ法

図 4.1 は**マンハッタン距離ヒューリスティック** (Manhattan distance heuristic) による A*探索である。青いノードは展開済みノード、緑のノードはオープンリストに入れられた未展開ノードである。ダイクストラ法による図 4.2 と比較すると、展開済み・未展開ノードの数が少なく済んでいることがわかるだろう。

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。

Theorem 3. 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。

Proof. 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。**TODO**kobayashi et al のような図を

Table 4.1: 15 パズルにおけるヒューリスティックの性能の比較

無矛盾なヒューリスティックである場合、全てのノード n は展開時までに $g(n)$ が n に辿り着くための最短経路コストの値になる。

Theorem 4. 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。

Theorem 5. ゴールノードが存在するとする。ゴールノード $n \in T$ の深さ $d(n)$ の中でもっとも小さいものを d とする *($d = min_{n \in T}(d(n))$)*。完璧なヒューリスティックを用いた *A**探索は d 回のノードの展開でゴールを発見し停止する。

Proof. □

効率的な A*探索を実装するために考えるべきことはいくつかある。まずヒューリスティック関数の正確さはパフォーマンスに大きな影響を与える。テーブル 4.1 は 15 パズルにおいてブラインドヒューリスティック、マンハッタン距離ヒューリスティック、パターンデータベースヒューリスティック (後述) を用いた A*の性能比較である。

4.3.1 重み付き A*探索 (Weighted A*)

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)、あるいは**局所探索** (local search) と呼ぶ。**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買いにたどり着くための手法である [67]。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。

$$f_w(n) = g(n) + wh(n) \tag{4.7}$$

Algorithm 9: Weighted A*: $Open.pop()$

Output : Node u

1 **return** $\arg\min_n f_w(n)$

Theorem 6. 許容的なヒューリスティックを用いた重み付き *A**探索は最適解のコスト f^* に対して、発見される解のコストが wf^* 以下であることを保証する。

TODOwA*: 解コストの上界の証明

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。

Figure 4.3: A*, wA*, GBFS 比較

wA*などの非最適探索を使う場合は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。許容的でないアルゴリズムはその代わりに高速であることがある。

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。このテクニックは多重配列アライメントなどに使われる [37]。

図 4.3 は A*と wA*の比較である (ここではエージェントは 4 方向に動けるとする)。wA*の重み w は 5 としている。wA*の展開・生成ノード数が A*よりも少ないことがわかるだろう。その一方 wA*で発見された解が A*で発見された解よりも遠回りなことがわかる。

4.3.2 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 [67]。

Algorithm 10: Greedy Best-First Search: $Open.pop()$

Output : Node u

1 **return** $\arg\min_n h(n)$

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ても強力な手法である。

図 4.3 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。

4.4 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、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。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。

4.5 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。

　データ構造の実装は効率的か？オープンリスト、クローズドリスト、両方重要である。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。例えば A*探索は C++でスライディングタイル問題だと秒間 XXX ノードが展開できる。もし

[2]解が多いほど簡単であるとは一概には言えないが

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、ヒューリスティック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。

メモリを使いすぎていてメモリスワップが起きていないか？もう一つの可能性はメモリスワップである。

より良いヒューリスティック関数は作れないか？ヒューリスティック関数の改善は実行時間を指数的に改善する可能性がある。

最適解は必要か？試しに wA*にして、weight を大きく取ってみよう。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 weight 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局所探索 (7 章) を使ったほうが良い場合が多い。

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。8 章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。

4.6 関連文献

最良優先探索 (best-first search) という言葉は 2 種類の定義があることに注意したい。[**?**] は最良優先探索を A*を含むアルゴリズムの集合を定義した [**?**]。一方 [**?**] はゴールに最も近い、つまり h 値が一番小さいノードを常に最優先して展開するアルゴリズムを最良優先探索と呼んだ [**?**]。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 []。

Chapter 5

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 [27]。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 [22]。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 [22]。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 [33, 63]、セルアセンブリ [4]、遺伝子距離計算 [20]、ビデオゲーム [51] 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 [58]。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 [21, 17]。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 [59]。

5.1 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 [22]。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

[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いう風に定義される。例えば問題に全く関係ない冗長な変数が含まれる場合、その情報がエージェントに与えられない場合を考えることができる。このような問題も冗長な変数を無視し古典的プランニング問題で扱うことができる。

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 [18] に従う。

Definition 8 (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、$A : S \to S$ 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 [22] 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 $s' = suc(s, a)$ は

$$s' = (s \cup add(a)) \setminus del(a) \tag{5.1}$$

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。例えば各ドメインに対して特化した手法を用いることも考えられるが、そのような人間の知識を必要とする手法を考えることは人工知能技術の目的とは離れてしまう。グラフ探索によって古典的プランニング問題を解く場合、問題の定義のみが必要である。解法についての知識は必要ではない。つまりプランナー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、ということだ。

fast-downward [31] 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多くが fast-downward に組み込まれている。

5.2 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 5.2 と図 5.3 はブロックス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) 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**

Figure 5.1: Blocks world ドメイン

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。例えばブロックス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。

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, 24]。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。

5.3 古典的プランニングモデルの例

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されているかを紹介する。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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)
                  (not (clear ?x))
                  (not (handempty))
                  (not (on ?x ?y)))))
```

Figure 5.2: blocks-world の domain ファイル

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

Figure 5.3: blocks-world の instance ファイル

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 (sattellite) 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。The domain is inspired from a NASA application, where satellites have to take images of spatial phenomena and send them back to Earth. In an extended setting, timeframes for sending messages to the ground stations are imposed

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る。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。[**?**]

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。[**?**]

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。

5.4 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とされている分野の一つである。

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。

5.4.1 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。

5.4.2 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。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。加えてパターンデータベース

5.4.2.1 15 パズル

パターンデータベースは 15 パズルのヒューリスティック関数としてよく使われる。

5.5 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) [36]。任意の許容的なヒューリスティックの組 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を使ってまた、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) [46]。

パターンデータベースはアブストラクト問題の解を陽に列挙する。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 [32]。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の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。

Declarative heuristic

Chapter 6

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。オープンリストはインターフェイスとしては Priority queue であり、必要な操作は pop と push である。クローズドリストは Hash Table であり insert と重複検知のための find 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 [10] がある。

6.1 オープンリスト (Open List)

オープンリストの Priority queue の実装方法は様々ある。まず、f 値の定義域が連続値か離散値かは重要である。連続値 (e.g. 実数) である場合は二分木のような一般的な Priority queue を使うことが多い。離散値である場合は *bucket* 実装をすることが出来る。

次に、f 値が同じノードが複数ある場合のタイブレーキング (tiebreaking) もパフォーマンスに影響を与える。h 値が最も小さいノードを優先することが多い。FIFO, LIFO のどちらが良いかという問題もある。

Table 6.1: オープンリストのデータ構造の比較

pests are used to support farmer decisions. Such maps are costly to obtain since they require 実装	*pop* 計算量
二分木	$O(log(n))$
bucket	XXX

6.1.1 プライオリティキュー (Priority Queue)

オープンリストは Priority queue である。ほとんどのケースで使える実装方法は二分木である。離散値である場合は *bucket* 実装をすることが出来る。許容的なヒューリスティックである場合は f 値の上界も分かる。

6.1.2 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 8)。だが、f 値が最小のノードは複数ある場合がある。特にユニットコストドメインにおいては f 値が同じノードが大量にあることがほとんどである。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 (アルゴリズム **??**)。

Algorithm 11: f, h tiebreaking: $Open.pop()$

Output : Node n

1 $N \leftarrow \arg\min_n(f(n))$ **return** $\arg\min_{n \in N} h(n)$

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、という直観である。

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。6.1.1 節にあるようにバケットは配列で実装されるが、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いなかった。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タイブレーキングは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。しかし特にゼロコストアクションのあるドメインではタイブレーキングこそがアルゴリズムの実行効率に影響を与えることが実験的に示された [5]。

タイブレーキングに関する詳細な議論と実験は [5] にある。

6.2 クローズドリスト (Closed List)

TODOclosed list クローズドリストはハッシュテーブルであり、ハッシュキーには状態が使われ、状態と g 値のペアが取り出せる必要がある。クローズドリストの実装はグラフ探索アルゴリズムの性能に大きな影響を与える。

ノードが生成されるたびに find が実行される。ノードが重複でなければ新たにクローズドリストに追加 (insert) される。これらの命令を効率的に実行できる実装をしたい。

6.2.1 ハッシュテーブル (Hash Table)

ハッシュテーブルの効率的な実装は

チェインリスト実装

6.2.2 遅延重複検出 (Delayed Duplicate Detection)

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 [45]。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 12 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。

Algorithm 12: Implicit Graph Search with delayed duplicate detection

Input : Implicit problem graph with initial node s, weight function w, successor generation function $Expand$, goal function $Goal$
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

```
1  Closed ← ∅;
2  Open ← {s};
3  while Open ≠ ∅ do
4  |  u ← Open.pop();
5  |  if Closed.find(u.state) then
6  |  |  u' ← Closed Continue;
7  |  Closed.insert(u);
8  |  if Goal(u) then
9  |  |  return Path(u);
10 |  Succ(u) ← Expand(u);
11 |  for each v ∈ Succ(u) do
12 |  |  Open.insert(v);
13 |  |  parent(v) ← u;
14 return ∅;
```

遅延重複検出のメリットは

6.2.3 ハッシュ関数 (Hash Function)

ハッシュ関数のキーは状態である。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で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良い。

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。

6.2.3.1 剰余法 (Remainder Method)

6.2.3.2 積算ハッシュ法 (Multiplicative Hashing)

積算ハッシュ法は状態 $s = \{v_0, v_1, ..., v_n\}$ 以下のように定義される。

$$H(s) = \lfloor m(x\phi \lfloor x\phi \rfloor) \rfloor \tag{6.1}$$

k_i と d はハッシュの変数

6.2.3.3 ハッシュの性質

incremental hashing perfect hashing universal hashing

6.3 関連文献

オープンリスト Two-layered radix heap Fibonacci heap
クローズドリスト
外部ストレージ

Chapter 7

局所探索 (Local Search)

7.1 山登り法 (Hill Climbing)

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。

Algorithm 13: Hill Climbing

Input : Initial node s_0
Output : Path from s_0 to a goal node $t \in T$

$s \leftarrow \{s_0\}$;
while $s \notin Goal$ **do**
  $s \leftarrow \arg\min_{s' \in Expand(s)} h(s)$
return 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 h 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。

7.1.1 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 []。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。

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。

Algorithm 14: Enforced Hill Climbing

Input : Initial node s_0
Output : Path from s_0 to a goal node $t \in T$

$s \leftarrow \{s_0\}$;
while $s \notin Goal$ **do**
  Let $T = \{v \in S | h(v) < h(s)\}$ $s' \leftarrow BreadthFirstSearch(s, T)$
return s

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 (4.3.2 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入れ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。

7.2 関連文献

焼きなまし法 (Simulated Annealing) ビームサーチ (Beam Search)。遺伝的アルゴリズム (Genetic Algorithm)。８クイーン問題。ポピュレーション

Chapter 8

ヒューリスティック探索の派生

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 [34]。

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。

反復深化 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。山登り法は解が得られる保証はないが、非常に高速に解を発見することがある局所探索手法である。また、消費メモリも非常に少ない。Symbolic Search は 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。外部メモリ探索は状態空間が RAM に収まりきらない問題に対して外部記憶を使うことで解決する手法である。並列探索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。

Table 8.1: 派生アルゴリズムの比較。

	空間計算量	時間計算量
反復深化 A*	$O(n)$	展開ノード数が大きくなる。キャッシュ効率が良く
両方向探索	A*よりも小さい	重複検出の効率による
External Search	外部記憶を用いる	I/O を必要とするため遅い
Symbolic Search	BDD によって効率的になる	複数のノードを同時に展開できる
並列探索	使用する計算ノードのメモリの総計が使える	線形スピードアップならコアの数だけ高

8.1 分枝限定法 (Branch-and-Bound)

分枝限定法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。アイディ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)、というアイディアである。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 15)。

Algorithm 15: Branch-and-Bound

Input : node s, weight function w, successor generation function $Expand$, goal function $Goal$, upper bound function U
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

$C \leftarrow \infty\ path \leftarrow \emptyset\ (C, path) = DFB\&B(s, 0, C, path)$ **return** $path$;

Algorithm 16: DFB&B(s, g, C, path): Depth-First Branch-and-Bound-Recursion

```
if Goal(s) then
    if g < C then
        C = g;
        path = Path(s);
Succ(s) ← Expand(s);
for each s' ∈ Succ(s) do
    if g + h(s') < C then
        (C, path) = DFB&B(v, g + w(s, s'), C, 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ィアである。A*探索と比較して分枝限定法の利点は、Anytime アルゴリズムであることである。Anytime 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、Anytime 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 Anytime A*というアルゴリズムもある []。A*や Anytime 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。

8.2 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。

6.2.2 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) アルゴリズム 17 は DFID の概要を示している。アイディアとしては、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木探索 (CLDFS) を実行する。CLDFS が解を見つければその解を返して停止し、見つけられなければ $cost$ 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。これは非常に効率が悪いように思えるかもしれないが、様々な状況において A*よりも有用な手法である。

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。深さ優先木探索は可能な最長経路だけのノードを保持する必要がある。木探索はクローズドリストは保持しない。コスト制限付きの場合、最長経路は $cost$ 以下である。$0 < w < 1$ となる実数コストがある場合、最小の w が 1 となるようにリスケールすることが出来る。 反復深化は $cost$ が最適解のコストになった時に停止するので、必要なメモリ量は最適解のコストに対して線形である。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、15-puzzle 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 [44]。

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 15-puzzle は解が存在するか非常に高速に判定することが出来る。

8.2.1 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 [44]。DFID と同様、コストを大きくしながら

Algorithm 17: Depth First Iterative Deepening

Input : Initial node s, weight function w, successor generation function $Expand$, goal function $Goal$
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

```
for c from 0 to ∞ do
    found ← CLDFS*(s, c);
    if found ≠ ∅ then
        return found;
```

Algorithm 18: CLDFS: Cost Limited Depth First Search for DFID

Input : Initial node s, cost c
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path with cost $\leq c$

```
if Goal(s) then
    return s;
for each child ∈ Expand(s) do
    if g(child) ≤ c then
        s' ← CLDFS(child, c) if s' ≠ ∅ then
            return s'
return ∅;
```

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 19 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 18 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 15-puzzle 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 [44]。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。

8.2.2 Transposition Table

IDA*で必要な空間は最適解のコストに対して線形である。そうすると、むしろかなりの量のメモリが余ることになる。そこで、メモリの余った分だけを使って重複検出をするという Transposition Table という手法がある。A*で用いられる Closed と異なり、このテーブルはすべての生成済みノードを保持しない。

Algorithm 19: CLDFS: Cost Limited Depth First Search for IDA*

Input : Initial node s, cost c
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path with cost $\leq c$

```
1 if Goal(s) then
2 |   return s;
3 for each child ∈ Expand(s) do
4 |   if f(child) ≤ c then
5 |   |   s' ← CLDFS(child, c) if s' ≠ ∅ then
6 |   |   |   return s'
7 return ∅;
```

8.3 両方向探索 (Bidirectional Search)

状態空間グラフの特徴を理解するための重要な指標として枝分数 (Branching factor) がある。枝分数は Expand 関数によって得られる子ノードの数の平均である。すなわち、重複検出をしないとすると、枝分数が b であるグラフにおいて深さ d のノードの数はおおよそ b^{d-1} である。例えば 15-puzzle は X であり、２次元４方向グリッド経路探索問題は４である。幅優先探索において最も浅い解の深さが C^* であると仮定すると、少なくとも b^{C^*-2} 個のノードを Expand しなければならない。

図**??**は幅優先探索

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。

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ても、find 命令の実行回数が片方向探索よりもかなり多くなる。両方向探索はヒューリスティック関数が不正確な時には片方向探索よりも効率的であることが知られている。Meet-in-the-Middle (MM) は

8.4 Symbolic Search

Binary Decision Diagram (BDD) は二分木によってブーリアン vector からブーリアンへの関数 $(x_0, x_1, ..., x_n) \rightarrow \{0, 1\}$ を効率良く表すグラフ構造である []。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 []。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。International Planning Competition (2014) 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 Symbolic Search が総なめした []。現在 (2017 年) の state-of-the-art の手法であるといえるだろう。

Figure 8.1: Sliding-token puzzle とそのバイナリ表現

8.4.1 特徴表現 (Symbolic Representation)

説明のためにシンプルな Sliding-token puzzle を用いる (図 8.1)。初期状態でタイルは位置 0 にある。タイルは右か左に動かすことが出来る。ゴール状態はタイルを位置 3 に置いた状態である。タイルの位置 $x(dom(x) = \{0, 1, 2, 3\})$ はバイナリ (x_0, x_1) に変換されている。状態および状態の集合は特徴関数 $\phi : S \to \{0, 1\}$ によって記述される。

例えば $S = \{0, 1\}$ とすると、$\phi_S(x)$ は $x \in S$ の場合に (かつその場合のみに) 真を返す特徴関数は $\neg x_0$ である。面白いことに、1 つの状態のみを含む状態集合 $S' = \{0\}$ を表す特徴関数よりも要素 2 つの S を表す特徴関数の方が表現がコンパクトになる。このように、特徴表現は明示的に列挙するよりも状態の集合をコンパクトに表現出来る場合がある。

アクションによる状態遷移も特徴関数 $Trans : S \times S \to \{0, 1\}$ 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)。

Sliding-token puzzle で可能なアクションは $(00) \to (01), (01) \to (00), (01) \to (10), (10) \to (01), (10) \to (11), (11) \to (10)$ の 6 つである。これらを表す遷移関数は

$$\begin{aligned} Trans(x, x') = &(\neg x_0 \neg x_1 \neg x'_0 x'_1) \\ &\vee (\neg x_0 x_1 \neg x'_0 \neg x'_1) \\ &\vee (\neg x_0 x_1 x'_0 \neg x'_1) \\ &\vee (x_0 \neg x_1 \neg x'_0 x'_1) \\ &\vee (x_0 \neg x_1 x'_0 x'_1) \\ &\vee (x_0 x_1 x'_0 \neg x'_1) \end{aligned} \tag{8.1}$$

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)。

Table 8.2: Sliding-token puzzle のエンコーディング

State ID	State Role	Binary Code	Boolean Formula
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	$x_0 x_1$

8.4.2 Binary Decision Diagram

状態集合およびアクション集合は **Binary Decision Diagram** (BDD) でコンパクトに表現することが出来る。

Definition 9. *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 $x_i (i \in \{1, ..., n\})$ 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。BDD を用いて集合演算を行うことが出来る。

BDD を使って状態やアクションの特徴関数を表現することが出来る。

8.4.3 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。

$$Image_S(x') = \exists x (Trans(x, x') \wedge \phi_S(x)) \tag{8.2}$$

8.4.3.1 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：

$$\phi_{S_i}(x') = \exists x (\phi_{S_{i-1}}(x) \wedge Trans(x, x')) \tag{8.3}$$

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**??**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。

Algorithm 20: BDD Breadth-first Tree Search

Input : Initial node s_0
Output : Path from s_0 to a goal node $t \in T$

$S_0 \leftarrow \{s_0\}$;
for $i \leftarrow 1...$ **do**
  $\phi_{S_i}(x) \leftarrow \exists x(\phi_{S_{i-1}}(x) \wedge Trans(x, x'))[x'/x]$;
  if $\phi_{S_i}(x) \wedge \phi_T \neq false$ **then**
    return $Construct(\phi_{S_i} \wedge \phi_T(x), i)$;

$Construct$ 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。

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。

8.4.3.2 BDD-幅優先探索

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 21 は BDD-幅優先探索のコードである。アルゴリズム 20 と異なり特徴関数 $Closed$ を用いて重複検出を行っている。

Algorithm 21: BDD Breadth-first search

Input : Initial node s_0
Output : Path from s_0 to a goal node $t \in T$

$S_0 \leftarrow \{s_0\}$;
$Closed \leftarrow \{s_0\}$;
for $i \leftarrow 1...$ **do**
  $Succ(x) \leftarrow \exists x(\phi_{S_{i-1}}(x) \wedge Trans(x, x'))[x'/x]$;
  $\phi_{S_i}(x) \leftarrow Succ(x) \wedge \neg Closed(x)$;
  $Closed(x) \leftarrow Closed(x) \vee Succ(x)$;
  if $\phi_{S_i}(x') \wedge \phi_T \neq false$ **then**
    return $Construct(\phi_{S_i} \wedge \phi_T(x), i)$;

Figure 8.2: マルバツゲームにおける新奇性の例

8.4.3.3 BDD-最適コスト探索

8.4.4 関連文献

8.5 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。

8.5.1 状態の新奇性 (Novelty)

状態に対して新奇性を定義する試みは人工知能研究において様々な場面で現れる [**?**]。なので新奇性の定義も様々であるが、本書では [26] の定義に従い、新奇性を以下のように定義する。

Definition 10. *Novelty* m 個の特徴関数の集合 $h_1, h_2, ..., h_m$ 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 $\{h_{i_1}, h_{i_2}, .., h_{i_n}\}$ が存在し、$h_{i_1}(s) = h_{i_1}(s')$, $h_{i_2}(s) = h_{i_2}(s')$,...,$h_{i_n}(s) = h_{i_n}(s')$ 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を返すものが使われることが多い [**?**, **?**, **?**, **?**]。つまり状態 $s = \{v_1, v_2, ..., v_m\}$ 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るということなので、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。

例えばマルバツゲームの新奇性を考える。特徴関数 h_i は i の位置にあるマーク (空、マル、バツ) を返す。初期状態 s_0 の新奇性は生成済み状態が存在しないので 1 である。

8.5.2 Width-Based Search (幅制限探索)

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 [50]。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。

Definition 11. 幅制限探索 *IW(i)* は新奇性が i よりも大きい状態を枝刈りする幅優先探索である。

i が特徴関数の数と同じである場合、幅制限探索はただの (重複検知を行う) 幅優先探索になる。

幅を制限することのメリットは 2 つある。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。

生成済みタプルの集合は真偽値のテーブルによって実装することができる。状態 s のある変数 X の値が x であることを s は**アトム** (atom) $(X = x)$ を持つと言う。例えば IW(1) であれば保存しなければならない情報は可能なアトムの数の真偽値のみである。

例えば IW(1) であれば真偽値の各値は h_i

すべての大きさ i 以下のタプルに対して

$i = m$ の場合この真偽値テーブルはクローズドリストそのものになる。

ここでよりも小さい

幅制限探索のパフォーマンスは特徴関数の選択によって大きく左右される。

[26] は新奇性の定義域を有理数に定義しなおした幅制限探索を提案した [26]。

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

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 [50]。IW(i) は幅 i が大きくなるほど大きな状態空間を探索することになる。幅を大きくしていくことでだんだんと探索する状態空間を大きくしていき、解を探すという手法である。

Algorithm 22: Iterative Width Search

```
1 n ← 1 while solisnull do
2 |   sol ← IW(n);
3 |   n ← n + 1;
4 return sol
```

8.5.4 関連文献

Best-First Width Search (最良優先幅制限探索) [26] Novelty Heuristics (新奇性に基づくヒューリスティック) [26]

8.6 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 SDD を用いることでこの問題を解決する [11]。すなわち、Open、Closed の一部を外部記憶に保持し、必要に応じて参照し RAM に持ってくる、ということをする。外部メモリ探索のミ

Table 8.3: 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。 (https://gist.github.com/jboner/2841832)

	nano sec
命令実行	1
fetch from L1 cache memory	0.5
branch misprediction	5
fetch from L2 cache memory	7
mutex lock/unlock	25
fetch from main memory	100
Read 4K randomly from SSD	150,000
read 1MB sequentially from memory	250,000
fetch from new disk location (seek)	8,000,000
Read 1 MB sequentially from SSD	1,000,000
read 1MB sequentially from disk	20,000,000

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 8.3 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。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。

8.6.1 関連文献

Structured duplicate detection [] Multiple Sequence Alignment PA* hash-based parallel external

8.7 並列探索 (Parallel Search)

近年コンピューター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 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。加えてハードウェアは刻々と変化を続けており、数年後にどのような環境がメジャーとなるのかはなかなか想像をすることが出来ないだろう。本書では CPU を用いた分散メモリ並列アルゴリズムと GPU 一台を用いた並列アルゴリズムについて説明する。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

ムワークが現在の state-of-the-art である。

より詳細な議論は**??**を参照されたい。

8.7.1 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 *perfect linear speedup* と呼ぶ。しかしながら、殆どの場合 perfect linear speedup 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。[40] 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。

通信オーベーヘッド (Communication overhead, CO): 通信オーバーヘッドはプロセス間で情報交換を行うことにかかるオーバーヘッドである。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：

$$CO := \frac{\text{\# messages sent to other threads}}{\text{\# nodes generated}}. \tag{8.4}$$

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：

$$CO := \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}. \tag{8.5}$$

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。

探索オーバーヘッド (Search Overhead, SO): 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：

$$SO := \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1. \tag{8.6}$$

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。

$$LB := \frac{\text{Maximum number of nodes assigned to a thread}}{\text{Average number of nodes assigned to a thread}}. \tag{8.7}$$

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。

同期オーバーヘッド (Coordination Overhead) 同期オーバーヘッドは他のスレッドの処理を待つためにアイドル状態にならなければならない時に生じるオーバーヘッドを指す。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 [9, 43].

これらのオーバーヘッドは独立ではなく、むしろ相互に関係しており、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。

8.7.2 ハッシュ分配 A* (Hash Distributed A*)

ハッシュ分配 A* (Hash Distributed A*, HDA*) [43] は CPU を用いた 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：

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 [43] では Zobrist hashing [69] がハッシュ関数として用いられていた。状態 $s = (x_1, x_2, ..., x_n)$ 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：

$$Z(s) := R_0[x_0] \; xor \; R_1[x_1] \; xor \; \cdots \; xor \; R_n[x_n] \tag{8.8}$$

Zobrist hashing は初めにランダムテーブル R を初期化する 24。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

Algorithm 23: ZHDA*

Input : $s = (x_0, x_1, ..., x_n)$
$hash \leftarrow 0$;
for *each* $x_i \in s$ **do**
  $hash \leftarrow hash\ xor\ R[x_i]$;
Return $hash$;

Algorithm 24: Initialize ZHDA*

Input : $V = (dom(x_0), dom(x_1), ...)$
for *each* x_i **do**
  for *each* $t \in dom(x_i)$ **do**
    $R_i[t] \leftarrow random()$;
Return $R = (R_1, R_2, ..., R_n)$

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 [9]。State abstraction は状態 $s = (x_1, x_2, ..., x_n)$ に対して簡約化状態 (abstract state) $s' = (x'_1, x'_2, ..., x'_m), where m < n, x'_i = x_j (1 \leq j \leq n)$. 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 congrugent 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 [38]。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：

$$Z(s) := R_0[A_0(x_0)]\ xor\ R_1[A_1(x_1)]\ xor\ \cdots\ xor\ R_n[A_n(x_n)] \tag{8.9}$$

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。

AZH はパラメータとして abstract feature を設定しなければならない。Abstract feature を自動的に生成する手法は複数提案されており、最もシンプルなものは Greedy abstract feature generation [39] である。

Domain transition graph

Abstract feature の生成方法として state-of-the-art の手法は Graph partitioning-based [40] であり、各 DTG を与えられた最適化指標下で分割することで abstract feature を生成する。

8.7.3 関連文献

並列深さ優先探索 [49]. Transposition Table Driven Work Distribution (TDS) []. Parallel Structured Duplicated Detection (PSDD) []. Parallel Best-NBlock First (PBNF) [].

一方、執筆時現在、GPU を用いたアルゴリズムはあまり研究が進んでいない。原因としては、既存の CPU を用いた探索アルゴリズムにはない様々な難しさがあ

るだろう。たとえば、GPU はスレッド当りのメモリ量が非常に少ない。A*探索はメモリが大きなボトルネックであり、メモリ量が少ないとそのまま解ける問題の大きさが制限されてしまう。この問題を解決する方法は提示されていない。もうひとつの難しさは、GPU は複数のスレッドが同じ命令を実行する Single instruction multiple thread (SIMT) という計算モデルであることである。そのため、既知の有力なヒューリスティック関数を GPU 環境において効率的に実装する方法が自明ではない。パターンデータベースなどのシンプルな命令によるヒューリスティックも考えることが出来るが、このようなヒューリスティックは今度はメモリを沢山消費するという問題点がある。効率的な GPU 並列化アルゴリズムの開発は大きな成果が期待されるブルーオーシャンであるといえる[1]。もし CPU/GPU を利用した効率的なグラフ探索アルゴリズムが開発出来れば、非常に大きなインパクトになるかもしれない。

8.8 モンテカルロ木探索 (Monte Carlo Tree Search)

モンテカルロ木探索 (Monte Carlo Tree Search, MCTS) は非常に多様な問題で応用されている手法である。その長所はたくさんある。まず、MCTS は状態空間が非決定論的でも扱える。不完全情報でも扱える。２人プレイヤーゲームなどにも使える。特に碁で使われている。Alpha Go でも使われた。

MCTS のエージェントは木データ構造を保持する。木の根は初期状態（２人プレイヤーゲームなどでは現在の盤面を初期状態とする）である。MCTS は初期状態から木を辿り、木に新しいノードを追加する、というイテレーションを繰り返す。一つのイテレーションは４つの処理のループからなる。

1. Select a next node following a tree policy
2. Add a new node to the tree
3. Simulate following a default policy
4. Back propagate

Algorithm 25: Monte Carlo Planning(state)

```
1 while True do
2 |   search(state, 0)
3 return bestAction(state, 0)
```

8.8.1 UCT (UCB1 Applied to Tree)

MCTS の tree policy の選択は状態空間をどのように探索するかに大きな影響を与える。最も広く使われている tree policy は UCB1 による [**?**, **?**]。

[1] 個人の感想である

Algorithm 26: search(state, depth)

```
1 if Terminal(state) then
2 |   return 0
3 if Leaf(state, d) then
4 |   return Evaluate(state)
5 action ← selectAction(state, depth);
6 (nextstate, reward) ← simulateAction(state, action);
7 q ← reward + γ search(nextstate, depth + 1);
8 UpdateValue(state, action, q, depth)
```

$$UCB1(s, s') = \bar{X_{s'}} + 2C_p\sqrt{\frac{2\ln n(s)}{\ln n(s')}} \tag{8.10}$$

where $\bar{X_{s'}}$ is the average reward from visiting s', $n(s), n(s')$ is a number of visits to state s, s', respectively. UCB1 を使う MCTS は Upper bound applied to tree (UCT) と呼ばれる。

8.9 その他の派生アルゴリズム

ここで紹介した以外にもヒューリスティック探索の派生アルゴリズムはたくさんある。

再帰的最良優先探索 (Recursive best-first search) (RBFS) は IDA*の拡張である [**?**]。RBFS はノードを最良優先の順で展開し、ヒューリスティック値をバックアップする。余ったメモリを使ってノードの展開数を抑える拡張が提案されている (Memory-aware RBFS) [**?**]。

Partial Expansion A* []。多重整列問題。

K-Best First Search

K-ビームサーチ Partial A* Partial IDA* MA* Frontier Search

ポートフォリオ戦略どれか一つのアルゴリズムに絞らなくてもよいいろいろ試してみるのが有効なことが多い

Index

A* search, 25
A*探索, 25

branching factor, 16

deterministic, 7
Dijkstra's Algorithm, 22
duplicate, 21
duplicate detection, 21

explicit state-space graph, 9

grid path-finding problem, 10

heuristic function, 25

implicit state-space graph, 9

lookahead, 6

Manhattan distance heuristic, 26
Multiple Sequence Alignment, 12

observation, 7

partial infomation, 7
path cost, 9
perfect information, 7

solution, 9
state-space graph, 8
state-space problem, 8

unit-cost, 9

グリッド経路探索問題, 10
ダイクストラ法, 22
ヒューリスティック関数, 25
マンハッタン距離ヒューリスティック, 26
ユニットコスト, 9

不完全情報, 7

先読み, 6
分枝度, 16
多重整列問題, 12
完全情報, 7

明示的状態空間グラフ, 9
決定論的, 7

状態空間グラフ, 8
状態空間問題, 8
経路コスト, 9

観察, 7
解, 9

重複, 21
重複検出, 21
非明示的状態空間グラフ, 9

Bibliography

[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] 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 **2015**, 7 (2015)

[3] Applegate, D.L.: The traveling salesman problem: a computational study. Princeton University Press (2006)

[4] Asai, M., Fukunaga, A.: Fully automated cyclic planning for large-scale manufacturing domains. In: Proc. ICAPS (2014)

[5] Asai, M., Fukunaga, A.: Tiebreaking strategies for a* search: How to explore the final frontier. In: Proc. AAAI (2016)

[6] Bäckström, C., Jonsson, P., Ståhlberg, S.: Fast detection of unsolvable planning instances using local consistency. In: Sixth Annual Symposium on Combinatorial Search (2013)

[7] Barraquand, J., Latombe, J.C.: Robot motion planning: a distributed representation approach. International Journal of Robotics Research (1991)

[8] Bellman, R.: On a routing problem. Quarterly of applied mathematics **16**(1), 87–90 (1958)

[9] 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)

[10] Burns, E.A., Hatem, M., Leighton, M.J., Ruml, W.: Implementing fast heuristic search code. In: Proceedings of the Annual Symposium on Combinatorial Search, pp. 25–32 (2012)

[11] Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: SODA, vol. 95, pp. 139–149 (1995)

[12] Chiappa, S., Racaniere, S., Wierstra, D., Mohamed, S.: Recurrent environment simulators. arXiv preprint arXiv:1704.02254 (2017)

[13] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press (2001). URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20{\&}path=ASIN/0262531968`

[14] Cui, X., Shi, H.: A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security **11**(1), 125–130 (2011)

[15] Culberson, J.: Sokoban is pspace-complete (1997)

[16] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)

[17] 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)

[18] Edelkamp, S., Schroedl, S.: Heuristic Search: Theory and Applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)

[19] Edgar, R.C., Batzoglou, S.: Multiple sequence alignment. Current opinion in structural biology **16**(3), 368–373 (2006)

[20] Erdem, E., Tillier, E.: Genome rearrangement and planning. In: Proc. AAAI, pp. 1139–1144 (2005)

[21] Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic sat-compilation of planning problems. In: IJCAI, vol. 97, pp. 1169–1176 (1997)

[22] 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)

[23] Ford Jr, L.R.: Network flow theory. Tech. rep., RAND CORP SANTA MONICA CA (1956)

[24] Fox, M., Long, D.: Pddl2. 1: An extension to pddl for expressing temporal planning domains. Journal of artificial intelligence research (2003)

[25] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM) **34**(3), 596–615 (1987)

[26] 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)

[27] Ghallab, M., Nau, D., Traverso, P.: Automated Planning Theory and Practice. Elsevier (2004)

[28] Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press (1997)

[29] Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on System Sciences and Cybernetics **SSC-4**(2), 100–107 (1968)

[30] Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees. Operations Research **18**(6), 1138–1162 (1970)

[31] Helmert, M.: The Fast Downward planning system. Journal of Artificial Intelligence Research (JAIR) **26**, 191–246 (2006). DOI 10.1613/jair.1705

[32] 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 (JACM) **61**(3), 16 (2014)

[33] Helmert, M., Lasinger, H.: The scanalyzer domain: Greenhouse logistics as a planning problem. In: Proc. ICAPS (2010)

[34] Helmert, M., Roger, G.: How good is almost perfect? In: Proceedings of the 23rd National Conference on Artificial Intelligence AAAI-08, pp. 944–949 (2008)

[35] 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. IOS Press (2014)

[36] Holte, R.C., Newton, J., Felner, A., Meshulam, R., Furcy, D.: Multiple pattern databases. In: ICAPS, pp. 122–131 (2004)

[37] Ikeda, T., Imai, H.: Enhanced a$*$ 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)

[38] Jinnai, Y., Fukunaga, A.: Abstract Zobrist hash: An efficient work distribution method for parallel best-first search. In: Proc. AAAI, pp. 717–723 (2016)

[39] Jinnai, Y., Fukunaga, A.: Automated creation of efficient work distribution functions for parallel best-first search. In: Proc. ICAPS (2016)

[40] Jinnai, Y., Fukunaga, A.: On work distribution functions for parallel best-first search. Journal of Artificial Intelligence Research (JAIR) (2017). (to appear)

[41] Johnson, W.W., Story, W.E., et al.: Notes on the 15 puzzle. American Journal of Mathematics **2**(4), 397–404 (1879)

[42] Junghanns, A., Schaeffer, J.: Sokoban: A challenging single-agent search problem. In: In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch. Citeseer (1997)

[43] Kishimoto, A., Fukunaga, A., Botea, A.: Evaluation of a simple, scalable, parallel best-first search strategy. Artificial Intelligence **195**, 222–248 (2013). DOI 10.1016/j.artint.2012.10.007. URL `http://linkinghub.elsevier.com/retrieve/pii/S0004370212001294`

[44] Korf, R.: Depth-first iterative deepening: An optimal admissible tree search. Artificial Intelligence **97**, 97–109 (1985)

[45] Korf, R.E.: Delayed duplicate detection. In: International Joint Conference on Artificial Intelligence, vol. 18, pp. 1539–1541. LAWRENCE ERLBAUM ASSOCIATES LTD (2003)

[46] Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Artificial Intelligence **134**(1), 9–22 (2002). DOI 10.1016/S0004-3702(01)00092-3. URL http://linkinghub.elsevier.com/retrieve/pii/S0004370201000923

[47] 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 AAAI-00, pp. 910–916 (2000)

[48] Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society **7**(1), 48–50 (1956)

[49] Kumar, V., Ramesh, K., Rao, V.N.: Parallel best-first search of state-space graphs: A summary of results. In: Proc. AAAI, vol. 88, pp. 122–127 (1988)

[50] Lipovetzky, N., Geffner, H.: Width and serialization of classical planning problems. In: Proc. ECAI, pp. 540–545 (2012). DOI 10.3233/978-1-61499-098-7-540. URL http://dx.doi.org/10.3233/978-1-61499-098-7-540

[51] Lipovetzky, N., Ramirez, M., Geffner, H.: Classical planning with simulators: Results on the Atari video games. In: Proc. IJCAI, pp. 1610–1616 (2015)

[52] McDowell, G.L.: Cracking the coding interview. CarrerCup (2011)

[53] McQuillan, J., Richer, I., Rosen, E.: The new routing algorithm for the arpanet. IEEE Transactions on Communications **28**(5), 711–719 (1980)

[54] Nash, A., Daniel, K., Koenig, S., Felner, A.: Theta^*: Any-angle path planning on grids. In: Proc. AAAI, pp. 1177–1183 (2007)

[55] 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)

[56] Pearl, J.: Heuristics - Intelligent Search Strategies for Computer Problem Solving. Addison–Wesley (1984)

[57] Pearson, W.R.: Rapid and sensitive sequence comparison with FASTP and FASTA. Methods in enzymology **183**, 63–98 (1990). Matrix score is available at http://prowl.rockefeller.edu/aainfo/pam250.htm

[58] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edition edn. Prentice-Hall, Englewood Cliffs, NJ (2003)

[59] Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence **219**, 40–66 (2015)

[60] 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)

[61] 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)

[62] Skiena, S.S., Revilla, M.A.: Programming challenges: The programming contest training manual. Springer Science & Business Media (2006)

[63] Sousa, A., Tavares, J.: Toward automated planning algorithms applied to production and logistics. IFAC Proceedings Volumes **46**(24), 165–170 (2013)

[64] Sturtevant, N.R.: Benchmarks for grid-based pathfinding. IEEE Transactions on Computational Intelligence and AI in Games **4**(2), 144–148 (2012)

[65] Tarjan, R.E.: Data structures and network algorithms. SIAM (1983)

[66] Waterman, M.S.: Introduction to computational biology: maps, sequences and genomes. CRC Press (1995)

[67] Wilt, C.M., Thayer, J.T., Ruml, W.: A comparison of greedy search algorithms. In: third annual symposium on combinatorial search (2010)

[68] Yap, P.: Grid-based path-finding. In: Conference of the Canadian Society for Computational Studies of Intelligence, pp. 44–55. Springer (2002)

[69] 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