ヒューリスティック探索入門

陣内 佑

April 9, 2023

目次

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、最寄駅まで歩き、職場へ向かう電車に乗る。

ごはんをよそうためには、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」と頭にあるだけなのに、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。

我々が自覚的にはほとんど頭を使わずにこのような計画を立てることが出来るのはなぜだろうか？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は何故だろうか？それは我々が無数の選択肢から**直感** (ヒューリスティック) 的に必要そうな行動を絞り込めるからである。

本書で扱う**ヒューリスティック探索**は、先に述べたような直感を駆使し、未来を先読みし、知的に行動を計画する能力をコンピュータに実装しようとする、人工知能研究の一分野である。

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、人工知能分野に限らず情報科学のいろいろな分野で使われる。本書では、特に人工知能の要素技術としてのグラフ探索アルゴリズムを解説する。

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 [123] では、人工知能研究の目標として以下の 4 つを掲げている。

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]。探索による**先読み** (lookahead) で、最も合理的な手を選ぶことがここでの目的である。

機械学習による Think Rationally との違いは先読みをするという点である。機械学習は過去の経験を元に合理的な行動を選ぶための技術である。それに対して、探索では未来の経験を先読みし、合理的な行動を選ぶ。

探索技術の大きな課題・欠点はモデルを必要とする点である。モデルがないと未来の経験を先読みできない。例えば将棋で先を読むには、各コマの動き方や、敵の王を詰ますと勝ちであることを知っていなければならない。加えて、ある局面でどちらがどのくらい有利なのかを評価できないと強いエージェントを作れない。 モデルは完璧である必要はないが、ある程度は有用なものでないと先読みがうまく行かなくなる。

[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

以上のような理由で、探索はモデルが容易に得られる問題において使われてきた。例えば経路探索問題では地図から距離を推定し、モデルを作ることが出来る。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]。

より賢い行動をするために、探索と機械学習を組み合わせようとする研究もある。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、探索と機械学習の組み合わせの強力さを体現している [126]。ここではディープラーニングを使って局面の評価値を学習し、それと探索を組み合わせて評価値が良い局面に繋がる手を選んでいる。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アプローチもある。先に述べたように探索にはモデルが必要である。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 [112, 125, 24]。

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。

1.2 本書で扱う内容

本書で主に扱う問題は**状態空間問題** (state-space problem) である。グラフ探索アルゴリズムは様々な場面で使われるが、この本では特に状態空間問題への応用に注目する。状態空間問題は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。

本書では特に、状態空間問題の中でも**完全情報** (perfect information) かつ**決定論的** (deterministic) モデルを取り扱う (2 6 章)。

現実世界をそっくりそのままプログラム上で扱うのは困難である。現実

[2] https://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/

の問題を扱いやすい形式で**モデル化**し、その問題を解くことで現実の問題を解決するのがエンジニアリングである。

世界をどうモデルするべきか判断するのは非常に難しい。モデルが単純であるほどモデル上の問題は解き易くなるが、単純だが正しいモデルをデザインする・自動生成することは難しい。その他にも、モデルの理解しやすさや汎用性の観点からモデルの良し悪しは決まる。

完全情報モデルとは、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。これは神の目線に立った意思決定のモデルである。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の一部だけを**観察** (observation) することで知ることが出来る。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に沿っているが、多くの問題を完全情報モデルで表現できる。

決定論的なモデルではエージェントの行動による世界の状態遷移が一意に (決定論的) に定まる。一方、非決定論的モデルでは、同じ状態から同じアクションを取ったとしても、世界がどう変化するかは一意には定まらない。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 [133]。

本書が扱う完全情報決定論的モデルは、上に挙げた中でもシンプルなモデルである。本書ではこのモデルを対象にしたグラフ探索アルゴリズムを解説する。

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 [113] 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 Schrodl の Heuristic Search Theory and Application [39] 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 [123] 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扱っている。Malik Ghallab, Dana Nau, and Paolo Traverso の Automated Planning and Acting [51] 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、探索をロボット制御等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)。この本では探索をロボットの意思決定に使うための技術的な課題とその解決方法を紹介している。

1.3 Python 実装について

本書では読者の理解の助けとなるべく Python 3 のコードを掲載している。疑似コードと合わせて読むことでアルゴリズムレベルでの処理と実装レベルでの処理の対応関係を理解できるだろう。

言語として Python を選んだ理由は読みやすく、人工知能技術に興味を持つ人にとってよく使われる言語であるためである。コードの目的は理解のためであり、高速な実装やソフトウェア工学的に優れた実装を目的としていない。

一方、探索アルゴリズムは C, C++のような高速な言語による恩恵を強く受けるアルゴリズムである。もし高速な実装が必要な場合は C, C++で実装することをお勧めする。とにかく高速な実装をしたい読者はアルゴリズムレベルの理解だけでなく、実装・コードレベルの理解も必要である。そのような読者は Burns et al. 2012 [20] を参照されたい。

ユニットコスト状態空間問題	$P_u = (S, A, s_0, T)$
状態空間問題	$P = (S, A, s_0, T, w)$
状態空間グラフ	$G_u = (V, E, u_0, T)$
重み付き状態空間グラフ	$G = (V, E, u_0, T, w)$
非明示的状態空間グラフ	$G_i = (\mathcal{E}, u_0, \mathcal{G}, w)$
状態	s, s'
初期状態	s_0
状態集合	S
アクション (行動)	a
アクション (行動) 集合	A
ゴール集合	T
問題グラフ	G
ノード	u, v, n
初期ノード	u_0
ノード集合	V
エッジ	e
エッジ集合	E
解	π
コスト関数	w
実数集合	$\mathbb{R}$
ブーリアン集合	$\mathbb{B}$
分枝度	b
深さ	d
最小経路コスト関数	g
ヒューリスティック関数	h
プライオリティ関数	f
最適解のコスト	c^*
オープンリスト	$Open$
クローズドリスト	$Closed$
状態変数	x
命題変数の集合	AP
適用条件	pre
追加効果	add
削除効果	del

表 1.1: 表記表

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 [123]、[113]、[39] 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 $A = a_1, ..., a_n$、$a_i : S \to S$ 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。

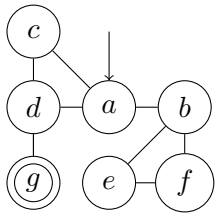


図 2.1: 状態空間問題の例。エージェントはスタート地点 a からゴール地点 g を目指す。

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 $\pi = (a_1, a_2..., a_k)$ 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。

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 $w : A \to \mathbb{R}$ 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。

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 $w : E \to \mathbb{R}$ 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。

定義 5 (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 $(s, Goal, Expand, w)$ 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:* $V \to 2^V$、コスト関数 $w : V \times V \to \mathbb{R}$ 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。

非明示的状態空間グラフも状態空間問題に対して定義できる。明示的状態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 2 次元 (あるいはもっと高次元でもよい) 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

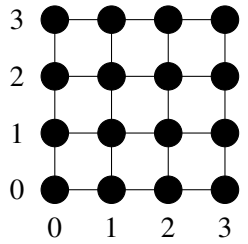


図 2.2: グリッド経路探索問題

問題である [139]。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 [110]。

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 [27, 130]。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。例えば後述する多重整列問題 (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]。この問題では複数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) にたどり着けることが目的となる。各関節の角度をグリッドの各次元で表し、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とでグリッド経路探索問題に帰着することができる。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) [72]。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

[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 [72]。

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** (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 [92]。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 $s=(x_1,x_2,...,x_n)$ 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 $0 \leq x_0 \leq l_0$ である。全てのアライメントが完了した状態 $s=(l_1,l_2,...,l_n)$ がゴール状態である。可能なアクション $a=(b_1,b_2,...,b_n),(b_i=0,1)$ は配列 i に対してそれぞれ欠損を挿入するか否かであり、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 $s' = (x_1 + b_1, x_2 + b_2, ..., x_n + b_n)$ 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 [114]。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 [136, 53, 40] 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				↓

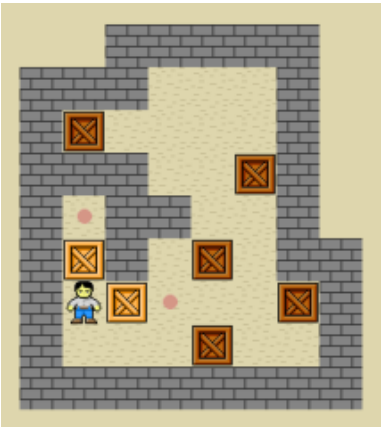


図 2.4: 倉庫番

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 [73, 28]。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 [28]。

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-`$\star$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-`$\star$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。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)

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 個の都市があるとすると (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

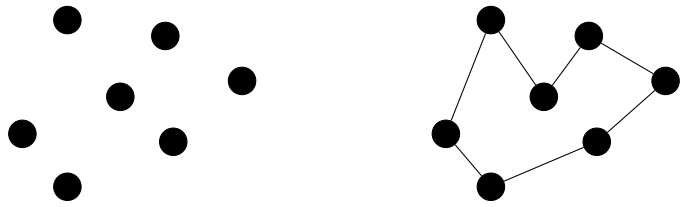


図 2.5: 巡回セールスパーソン問題

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れば理想的である。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, 63]。

[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 Python 実装

状態空間問題 $P = (S, A, s, T, w)$ のインターフェイスを Python で実装すると以下のように書ける。

```
class StateSpaceProblem:
    def __init__(self):
        assert False

    def get_init_state(self):
        assert False

    def is_goal(self, state) -> bool:
        assert False

    def get_available_actions(self, state):
        assert False

    def get_next_state(self, state, action):
        assert False

    def get_action_cost(self, state, action):
        assert False

    def heuristic(self, state):
        assert False
```

状態空間問題の構成要素と Python の実装は以下のように対応している。

- 状態集合 S
  状態集合は明示的にコード中には現れない。
- 行動集合 A: `get_available_actions`
  状態に対してそれに適用できる行動の集合を返す関数 $S \to A$ として表現される。

- 初期状態 s: `get_init_state`
  初期状態を返す関数。

- 遷移関数 T: `get_next_state`
  状態と行動を受け取り、次の状態を返す関数 $S, A \to S$ として表現される。

- コスト関数 w: `get_action_cost`
  行動を受け取り、その行動のコストを返す関数 $A \to \mathbb{R}$ として表現される。

このクラスを親としてグリッド経路探索問題を実装すると、以下のようなコードになる。

```
from state_space_problem import StateSpaceProblem

class GridState:
    def __init__(self, xy):
        self.x = xy[0]
        self.y = xy[1]

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        return (self.x == other.x) and (self.y == other.y)

    def __str__(self):
        return "(" + self.x.__str__() + ", " + self.y.__str__() +
                                                ")"

class GridPathfinding(StateSpaceProblem):
    state_type = GridState

    def __init__(self,
```

```
                width=5,
                height=5,
                init_position=(0, 0),
                goal_position=(4, 4)):

        self.width = width
        self.height = height
        self.init_position = init_position
        self.goal_position = goal_position

        # TODO: make state class?
        self.init_state = GridState(self.init_position)

    def get_init_state(self):
        return self.init_state

    def is_goal(self, state):
        return (state.x == self.goal_position[0]) and (state.y ==
                                                       self.goal_position[1])

    def get_available_actions(self, state):
        actions = []
        if state.x > 0:
            actions.append('l')
        if state.x < self.width - 1:
            actions.append('r')
        if state.y > 0:
            actions.append('u')
        if state.y < self.height - 1:
            actions.append('d')
        return actions

    def get_next_state(self, state, action):
        if action == 'l':
            return GridState((state.x - 1, state.y))
        elif action == 'r':
```

```
            return GridState((state.x + 1, state.y))
        elif action == 'u':
            return GridState((state.x, state.y - 1))
        elif action == 'd':
            return GridState((state.x, state.y + 1))
        else:
            raise Exception("Invalid action: " + action)
```

本書で扱う探索アルゴリズム含めほとんどの状態空間問題の解法は以上の構成要素を含んでいればこのインターフェイスに対応して実装が出来る。

2.5 まとめ

状態空間問題は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プラン (行動の列) を見つける問題である。状態空間問題はグラフ上の経路探索問題に帰着できる。このときのグラフを状態空間グラフと呼ぶ。状態空間グラフを初期ノードとそのノードと隣接するノードを返すノード展開関数によって表現することができる。これを非明示的状態空間グラフと呼ぶ。

状態空間問題ではしばしば行動にコストが定義されていて、プラン全体の行動コストの総和を最小化することが求められる。特に全ての行動のコストが同じである場合はユニットコスト状態空間問題と呼ぶ。

2.6 練習問題

1. ルービックキューブを状態空間問題で表現するとする。このとき、何を状態とするべきか？何を行動とするべきか？

2. スライディングタイル問題でタイルの大きさが 3×3 の時の状態の数はいくつか？4×4 の時は？(ヒント: スライディングタイルには偶奇性があるため、ゴール状態に到達できない盤面がある。)

3. 障害物のない 5x5 の 2 次元空間上で上下左右に移動が出来るグリッド経路探索問題を考える。この状態空間における分枝度はいくつか？

4. (難問) 3x3 のスライディングタイル問題における分枝度はいくつかを正確に計算することはできるか？

5. (難問) 状態空間問題として表現することが難しい問題の例を 2, 3 挙げてみよう。それらの問題を表現するにはどのように状態空間問題を拡張するべきか？

2.7 関連文献

本章で扱うヒューリスティック探索は主にここで定義した状態空間問題への適用を前提として書かれている。そのため状態空間問題よりも広い問題を解くためには少し工夫が必要になることが多い。

状態空間問題は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 [117]。MDP(S, A, P, R) は状態集合 S、アクション集合 A、状態遷移確率 P、報酬 R からなる。状態空間問題は状態遷移が決定論的であるのに対して MDP のそれは確率的である。また、MDP はコストを最小化するのではなく報酬の総和の期待値を最大化する問題として定義されることが多い。状態空間問題は MDP のうち状態遷移が常に決定的であるものである。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 [132]。状態遷移が決定論的であれば、状態遷移は一つの状態から一つの状態へのエッジによって表すことができる。状態遷移が確率的である場合は可能な次状態を列挙する AND-OR 木 [102] でモデルすることが出来る。MDP を解くには動的計画法 [117, 12] やモンテカルロ木探索 [17, 84] が使われることが多い。

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

[74]。POMDP におけるプランニング問題の厳密解は Belief space [74] 上を探索することで求められるが、多くの場合厳密計算は困難なのでモンテカルロ木探索などの近似手法が用いられる。POMDP は非常に計算が難しいモデルであるので、仮に POMDP が与えられた問題をもっとも正確に表せられるモデルであったとしても近似的に MDP を使った方がうまくいくかもしれない。

囲碁やチェスなどの敵対するエージェントがいる問題も探索が活躍するドメインであり、特に二人零和ゲームでの研究が盛んである。このような問題では敵プレイヤーの取るアクションが事前に分からないので AND-OR 木でモデルすることが多い。敵対ゲームを解くための手法としては MiniMax 木、α-β 木 [113] やモンテカルロ木探索などがある。近年では強化学習によってヒューリスティック関数を学習するアプローチが盛んに研究されている [132]。

ヒューリスティック探索はゲームやロボティクスの応用の中で現れる経路探索問題を解くために特によく使われる。ゲームに探索を使う場合問題になるのは計算にかかる時間である。たとえば次の一手を計算するために一日以上かかるチェス AI とプレイしたい人は少ないだろう。このような応用では解のコストだけでなく、解を得るために使う時間も評価しなければならない。このような問題をリアルタイム探索と呼ぶ [88, 12]。リアルタイム探索には Learning real-time A* (LRTA*) [88, 85] や Real-time adaptive A* (RTAA*) [86] などが使われる。

本書ではノードの次数が有限 (局所有限グラフ) である問題を対象としたアルゴリズムを紹介している。一見自然な仮定だが、この仮定ではアクションや状態空間の大きさが無限である連続空間問題を解くことができない。そのためロボティクスで現れる連続空間問題を解くためには工夫が必要になる。例えばグリッド経路探索問題で任意の角度を取ることが出来る Any-angle path planning 問題は本書では扱わないが、ヒューリスティック探索の派生アルゴリズムで解くことができる [94, 95, 30, 111]。

状態空間問題は制約充足問題の一種であり、Boolean satisfiability problem (SAT) として書くことができる [49]。それを利用して SAT によって状態空間問題を解くアプローチも広く研究されている [79, 78, 121]。状態空間問題で

は表現することが難しい制約や目的関数がある場合は制約充足問題や離散最適化問題としてモデルすることもできる。

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。**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 [128]。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 [104]。情報なし探索は [26] 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。

アルゴリズム 1: 木探索 (Implicit Tree Search)

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

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

```
1  Open ← {s}, d(s) ← 0, g(s) ← 0;
2  while Open ≠ ∅ do
3      u ← arg min_{u'∈Open} f(u') ;
4      Open ← Open \ {u};
5      if Goal(u) then
6          return Path(u);
7      for each v ∈ Expand(u) do
8          Open ← Open ∪ {u};
9          d(v) ← d(u) + 1;
10         g(v) ← g(u) + w(u, v);
11         parent(v) ← u;
12 return ∅;
```

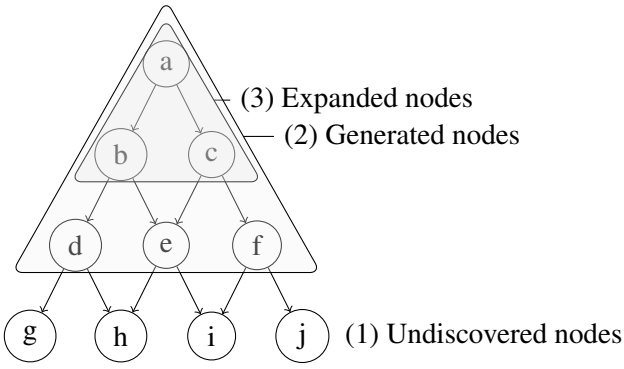


図 3.1: 未生成・生成済み・展開済みノードの例。木探索は生成済みノードのうち展開済みでないものをひとつ取り出し、そのノードを展開し子ノードを得る。新しく得られた子ノードは未生成ノードから生成済みノードとなり、展開したノードは展開済みノードになる。

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。

[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。

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。$(u, parent(u), parent(parent(u)), ..., s)$ 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。一方これの問題は、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

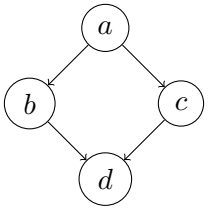


図 3.2: ノードの再展開の例

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。**

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

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

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

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

```
1  Open ← {s}, Closed ← {s}, d(s) ← 0, g(s) ← 0;
2  while Open ≠ ∅ do
3      u ← arg min_{u'∈Open} f(u') ;
4      Open ← Open \ {u};
5      if Goal(u) then
6          return Path(u);
7      for each v ∈ Expand(u) do
8          if v ∉ Closed or g(u) + w(u, v) < g(v) then
9              Open ← Open ∪ {v};
10             d(v) ← d(u) + 1;
11             g(v) ← g(u) + w(u, v);
12             parent(v) ← u;
13         if v ∉ Closed then
14             Closed ← Closed ∪ {v};
15 return ∅;
```

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 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、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。

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

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)。

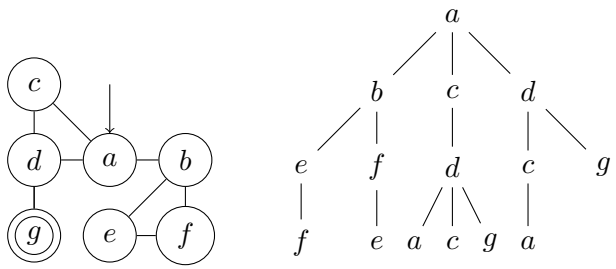


図 3.3: 探索木

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みてほしい。

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 ([39] 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。

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

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 $Open \leftarrow Open \cup \{v\}$ 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 ([39] 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 [32]。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 [105]。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 : 非明示的状態空間グラフ $(s, Goal, Expand, w)$、前状態 s'
Output : s からゴール状態への経路、経路が存在しなければ $\emptyset$

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

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

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 $O(|E| + |V|log|V|)$ 時間でであることが知られている [47]。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 情報なし探索の比較

探索アルゴリズムの評価指標としては以下の四点が重要である。

1. 完全性: 解が存在するとき、有限時間内に解を返すか。
2. 最適性: 最初に発見された解が最適解か。

3. 時間: アルゴリズムの実行にかかる時間

4. 空間: アルゴリズムの実行にかかる空間

完全なアルゴリズムであることは有用だが、時間・空間とのトレードオフにあることが多い。完全であっても実現可能な時間・空間で実行することが出来なければ意味がないので、完全でない高速なアルゴリズムを選択する方が良い場合もある。最適性も同様に時間・空間とのトレードオフにあることが多いので、解きたい問題に最適な解が必要かどうかを考える必要がある。また、後述する wA* (節 4.5.1) は発見された解が最適解のコスト c^* の定数倍以下であることを保証する近似アルゴリズムである。

表 3.3 は情報なし探索をこれら四点で比較している。反復深化深さ優先は節 6.3、両方向探索は節 6.4 で紹介する。b は分枝数、d は解の深さ、m は可能な探索木の深さの最大値である。重複検出を行うグラフ探索の場合、深さ優先探索は有限グラフならば完全であり、時間・空間は状態空間の大きさ $|S|$ でバウンドされる。この表にある時間・空間量は理論的な最悪の場合の比較である。解きたい問題にかかる平均的な性能は必ずしもこれに相関しない。最終的には実験的なベンチマークを行い良いアルゴリズムを選択する必要がある。

表 3.3: 木探索アルゴリズムの比較 ([123] の Figure 3.21 より)

性能	幅優先探索	深さ優先探索	反復深化深さ優先	両方向幅優先探索
完全性	局所有限グラフなら Yes	No	局所有限グラフなら Yes	局所有限グラフなら Yes
最適性	ユニットコストなら Yes	No	ユニットコストなら Yes	ユニットコストなら Yes
時間	$O(b^d)$	$O(b^m)$	$O(b^d)$	$O(b^{d/2})$
空間	$O(b^d)$	$O(bm)$	$O(bd)$	$O(b^{d/2})$

3.7 Python 実装

情報なし探索は様々な言語でライブラリが開発されているため、参考に出来る実装例が多い。本書では理解の助けとなるために紹介することが目的であるため、既存のライブラリは用いない実装を示す。

Python による木探索は例えば以下のように実装できる。

```
class SearchNode:
    def __init__(self, state):
        self.state = state

    def set_g(self, g):
        self.g = g

    def set_d(self, d):
        self.d = d

    def set_prev_n(self, prev_n):
        self.prev_n = prev_n

    def __str__(self):
        return self.state.__str__() + ": g=" + str(self.g) + ", d
                                        =" + str(self.d)

    def get_path(self):
        cur_node = self
        path = []

        while (cur_node is not None and hasattr(cur_node, 'prev_n
                                        ')):
            path.append(cur_node)
            cur_node = cur_node.prev_n

        return path
```

```
from search_node import SearchNode
from util import SearchLogger

def TreeSearch(problem, priority_f=None):
    open = []

    init_state = problem.get_init_state()

    init_node = SearchNode(init_state)
    init_node.set_g(0)
    init_node.set_d(0)

    logger = SearchLogger()
    logger.start()

    open.append(init_node)

    while (len(open) > 0):
        open.sort(key=lambda node: priority_f(node), reverse=True
                                                )

        node = open.pop()
        logger.expanded += 1

        if problem.is_goal(node.state):
            logger.end()
            logger.print()
            return node.get_path()
        else:
            actions = problem.get_available_actions(node.state)

            for a in actions:
                next_state = problem.get_next_state(node.state, a
                                                    )
```

```
                next_node = SearchNode(next_state)
                next_node.set_g(node.g + problem.get_action_cost(
                                                    node.state, a))
                next_node.set_d(node.d + 1)
                next_node.set_prev_n(node)
                open.append(next_node)
                logger.generated += 1

    logger.end()
    logger.print()
    return None
```

TreeSearch は状態空間問題とプライオリティ関数 priority_f を引数に取る。

このコードに重複検出を加えたグラフ探索は以下のように実装できる。

```
from search_node import SearchNode
from util import SearchLogger

def is_explored(node, closed_list):
    for n in closed_list:
        if (n.state == node.state) and (n.g <= node.g):
            return True
    return False

def GraphSearch(problem, priority_f=None):
    open = []
    closed = []

    init_state = problem.get_init_state()

    init_node = SearchNode(init_state)
    init_node.set_g(0)
    init_node.set_d(0)

    logger = SearchLogger()
```

```
    logger.start()

    open.append(init_node)
    closed.append(init_node)


    while (len(open) > 0):
        open.sort(key=lambda node: priority_f(node), reverse=True
                                            )

        node = open.pop()
        logger.expanded += 1

        if problem.is_goal(node.state):
            logger.end()
            logger.print()
            return node.get_path()
        else:
            # Expand the node
            actions = problem.get_available_actions(node.state)

            for a in actions:
                next_state = problem.get_next_state(node.state, a
                                                    )

                next_node = SearchNode(next_state)
                next_node.set_g(node.g + problem.get_action_cost(
                                                    node.state, a))
                next_node.set_d(node.d + 1)
                if not is_explored(next_node, closed):
                    next_node.set_prev_n(node)
                    open.append(next_node)
                    closed.append(next_node)
                    logger.generated += 1
                else:
                    logger.pruned += 1
```

```
    logger.end()
    logger.print()
    return None
```

priority_f を適当に渡すことで本章で紹介した幅優先探索、深さ優先探索、ダイクストラ法を実行することができる。

```
from graph_search import GraphSearch

def BreadthFristSearch(problem):
    return GraphSearch(problem, lambda node: node.d)
```

```
from graph_search import GraphSearch

def DepthFristSearch(problem):
    return GraphSearch(problem, lambda node: -node.d)
```

```
from graph_search import GraphSearch

def DijkstraSearch(problem):
    return GraphSearch(problem, lambda node: node.g)
```

そして次章で紹介するヒューリスティック探索の手法も実はこのプライオリティ関数を適切に渡すことで実装ができる。

上記これらのアルゴリズムはイテレーション (反復) によって実装されている。

再帰による探索アルゴリズムはコードの構成が異なるため、別途実装を示す。再帰による深さ優先探索は以下のように実装できる。

```
from search_node import SearchNode
from util import SearchLogger

logger = SearchLogger()

def RecursiveSearchEngine(problem, cur_node):
```

```
    logger.expanded += 1

    if problem.is_goal(cur_node.state):
        return [cur_node]
    else:
        actions = problem.get_available_actions(cur_node.state)

        for a in actions:
            next_state = problem.get_next_state(cur_node.state, a
                                                )

            next_node = SearchNode(next_state)
            next_node.set_g(cur_node.g + problem.get_action_cost(
                                                cur_node.state, a))
            next_node.set_d(cur_node.d + 1)
            next_node.set_prev_n(cur_node)

            logger.generated += 1
            path = RecursiveSearchEngine(problem, next_node)
            if len(path) > 0:
                path.append(cur_node)
                return path
    return []

def RecursiveDepthFirstSearch(problem):
    init_state = problem.get_init_state()
    init_node = SearchNode(init_state)
    init_node.set_g(0)
    init_node.set_d(0)

    logger.start()

    path = RecursiveSearchEngine(problem, init_node)

    logger.end()
    logger.print()
```

```
    return path


if __name__ == "__main__":
    from tsp import Tsp
```

探索アルゴリズムの性能評価は様々な指標が考えられる。多くの場合モニターするべきと考えられるのは以下の指標である。

- 探索アルゴリズムの実行時間 (walltime, cputime)
- 展開されたノード数
- 生成されたノード数
- 重複検出・枝刈りされたノード数
- ノードの展開速度 (生成速度)
- 重複検出・枝刈りされたノードの割合

これらをモニターするためのユーティリティとして `SearchLogger` を実装した。

```
import time

class SearchLogger:
    def __init__(self) -> None:
        self.expanded = 0
        self.generated = 0
        self.pruned = 0

    def start(self):
        self.start_perf_time = time.perf_counter()
        self.start_time = time.time()
```

```
    def end(self):
        self.end_perf_time = time.perf_counter()
        self.end_time = time.time()

    def branching_factor(self):
        return self.generated / self.expanded

    def pruned_rate(self):
        return self.pruned / (self.generated + self.expanded)

    def time(self):
        return self.end_time - self.start_time

    def perf_time(self):
        return self.end_perf_time - self.start_perf_time

    def expansion_rate(self):
        return self.expanded / self.time()

    def generation_rate(self):
        return self.generated / self.time()

    def print(self):
        print("Time: ", self.time())
        print("Perf Time: ", self.perf_time())
        print("Expanded: ", self.expanded)
        print("Generated: ", self.generated)
        print("Pruned: ", self.pruned)
        print("Expansion rate: ", self.expansion_rate())
        print("Generation rate: ", self.generation_rate())
        print("Branching factor: ", self.branching_factor())
        print("Pruned rate: ", self.pruned_rate())
```

探索アルゴリズムの性能が思わしくない場合、その原因がどこにあるかを突き止める必要がある。まずこれらの指標を確認することはその特定に役

立つだろう。

3.8 まとめ

状態空間問題を解くための手法としてグラフ探索アルゴリズムがある。グラフ探索アルゴリズムは 1. 展開するノードの順序と 2. 展開したノードの処理の 2 点で特徴づけることが出来る。幅優先探索はオープンリストの中から深さが浅いノードから優先して探索し、深さ優先探索は逆に深いノードから優先して探索する。ダイクストラ法はコストが小さいノードから優先して探索するため、最短経路を最初に見つけることができる。木探索は展開したノードを保持せず同じ状態を複数回重複して展開してしまうリスクがある。グラフ探索は展開済みノードをクローズドリストに保存しておくことで重複展開を防ぐ。一方メモリ消費量が大きくなるというデメリットがある。

探索アルゴリズムの選択において考慮するべきことは主に以下の 4 点である。解が存在するとき、有限時間内に解を返すか (完全性)、最初に発見された解が最適解か (最適性)、アルゴリズムの実行にかかる時間と空間。

3.9 練習問題

1. 幅優先木探索を実装し、10x10 程度のグリッド経路探索問題を解いてみよう。

2. 重複検出を使った幅優先探索を実装し、10x10 程度のグリッド経路探索問題を解いてみよう。重複検出を使わない幅優先木探索と比べて展開したノードの数は？重複検出によって枝刈りされたノードの数は？

3. 重複検出を使った深さ優先探索を実装し、10x10 程度のグリッド経路探索問題を解いてみよう。見つけた解は 1, 2 で見つけた解と比べて長いか？

4. 重複検出を使った幅優先探索で巡回セールスパーソン問題を解いてみよう。この時、見つかった解は最短経路だったか？

5. ダイクストラ法を実装し、巡回セールスパーソン問題を解いてみよう。この時、見つかった解は最短経路だったか？

6. 幅優先探索も再帰によって実装することができる。再帰による幅優先探索を実装してみよう。再帰による深さ優先探索と比べてどのような違いがあるか？

3.10 関連文献

時間・空間制約が厳しい場合は反復深化探索、両方向探索や並列探索を使えば解決できることがある。これらのアルゴリズムについては 6 章で紹介する。

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 [13, 45]。

No Free Lunch 定理 [138] 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。

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。

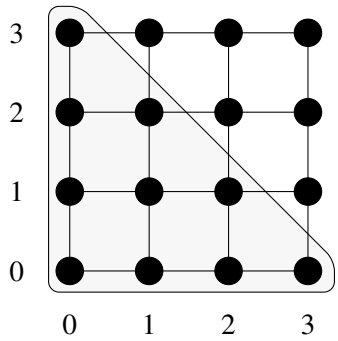


図 4.1: グリッド経路探索におけるダイクストラ法

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 [55]。

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。$h : V \to \mathbb{R}_{\geq 0}$

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。

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 [61]。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

$$h(u) \leq h(v) + k(u, v) \tag{4.1}$$

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。

$$\begin{aligned} h(u) &\leq h(n) + k(u, n) && (4.2)\\ &= k(u, n) && (4.3)\\ &= h^*(u) && (4.4) \end{aligned}$$

よって $h(s) \leq h^*(n)$ より許容的である。 □

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

$$
\begin{aligned}
h(n_0) &\leq h(n_1) + w(n_0, n_1) && (4.5)\\
&\leq h(n_2) + w(n_0, n_1) + w(n_1, n_2) && (4.6)\\
&\cdots && (4.7)\\
&\leq h(n_k) + \sum_{i=0..k-1} (w(n_i, n_{i+1})) && (4.8)\\
& && (4.9)
\end{aligned}
$$

$u = n_0, v = n_k$ 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にシンプルである。

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。

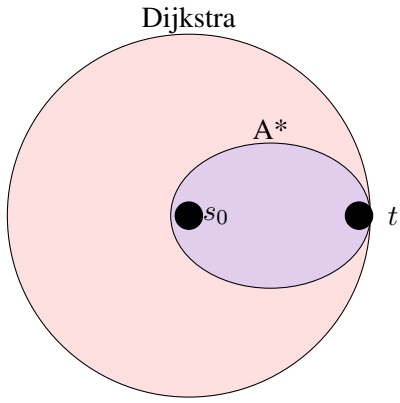


図 4.2: A* (ヒューリスティック探索) とダイクストラ法 (情報なし探索) の探索範囲の比較

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無視し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 [44]。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。

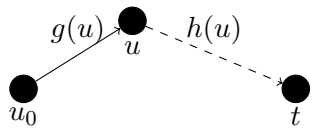


図 4.3: f_{A^*} 値の意味

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

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

$$h_2(s) > h_1(s) \tag{4.11}$$

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。

一方のヒューリスティック関数が他方より情報がある場合はより情報のある方を使えばよい。しかし、ある場所では h_1 よりも h_2 の方が正確であり、他のある場所では h_2 よりも h_1 の方が正確である、となる場合がある。この場合どちらのヒューリスティックを使えば良いだろうか？実は両方を使うことによってより良い正確なヒューリスティックを得ることができる。

定理 10: ヒューリスティック $h_1, ..., h_m$ が許容的であるとき、$h(s) = \max(h_1(s), ..., h_m(s))$ は許容的である。ヒューリスティック $h_1, ..., h_m$ が許容的で無矛盾であるとき、$h(s) = \max(h_1(s), ..., h_m(s))$ は許容的で無矛盾である。

このように複数のヒューリスティックを組み合わせることでより良いヒューリスティックを得ることができる。このアイディアはさまざまなヒューリスティック関数の自動生成に応用されている。

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。

[2]解が多いほど簡単であるとは一概には言えないが

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。マンハッタン距離は障害物を無視した最短経路の距離であるので、元の問題グラフに対してグラフのエッジを増やした緩和問題での解のコストに対応する。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。

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 スライディングタイル: パターンデータベース

パターンデータベースヒューリスティック (Pattern database heuristic) は部分問題を解き、その解コストをヒューリスティック値とするアルゴリズムである [35]。探索を始める前に部分問題を解き、部分問題のすべての状態からゴール状態への最適解のコストをテーブルに保存する。図 4.4 はスライディングタイルにおけるパターンデータベースの例である。図の例は「1, 2, 3, 4 のタイルを正しい位置に動かす」という部分問題を解いている。この部分問題では 5, 6, 7, 8 のタイルの位置はどこでもよい。1, 2, 3, 4 さえゴール位置にあればよい。ゴール条件以外は元の問題と同様である。この部分問題は元の問題の緩和問題になっているので許容的なヒューリスティックが得られる。

図の例では 1, 2, 3, 4 のタイルに絞った部分問題だったが、他のタイルの組み合わせでも良い。例えば 1, 2, 3, 4 のタイルによる部分問題と 5, 6, 7, 8 のタ

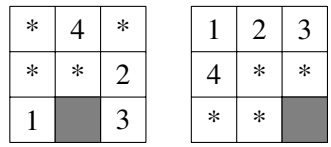


図 4.4: パターンデータベースの初期状態 (左) とゴール条件 (右) の例。1, 2, 3, 4 のタイルだけゴール位置にあればよく、他のタイルの位置は関係ない。

イルによる部分問題からは異なるヒューリスティック値を得ることができる。これらのヒューリスティック値の最大値を取ることでより正確なヒューリスティックにすることができる。このように複数のパターンデータベースを使うヒューリスティックを**複数パターンデータベース** (multiple pattern databse) と呼ぶ。複数パターンデータベースは一つの大きなパターンデータベースを使うよりも正確さに欠けるが、計算時間・空間の面で大きなアドバンテージがある。

1, 2, 3, 4 のタイルによる部分問題と 5, 6, 7, 8 のタイルによる部分問題は一見完全に分割された部分問題なので、これらのヒューリスティック値の和を取ってもよさそうだが、そうすると許容的なヒューリスティックにはならない。なぜなら例えばタイル 5 を動かすアクションによるコストは両方の部分問題のコストに数えられているからである。そこで部分問題のコストを「1, 2, 3, 4 (あるいは 5, 6, 7, 8) のタイルを動かすときのみコストがかかり、他のタイルはコスト 0 で動かせる」とすると、コストを重複して数えることはなくなる。そうするとこれらの部分問題のコストの和は許容的なヒューリスティックになる。このように複数のパターンデータベースで重複してコストが数えられないようにしたものを**素集合パターンデータベース** (disjoint pattern databse) と呼ぶ [90]。

マンハッタン距離ヒューリスティックは、各タイルごとの部分問題 8 つに分けた場合の素集合パターンデータベースである。

4.4.4 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 4.5)。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。

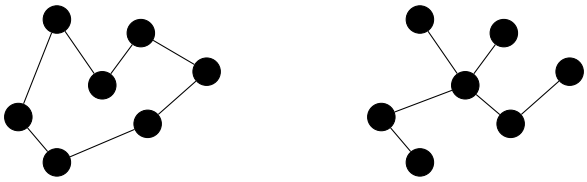


図 4.5: 巡回セールスパーソンにおける TSP 解 (左) と最小全域木 (右)

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 [137]。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。

$$f_w(n) = g(n) + wh(n) \tag{4.12}$$

定理 11: 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。

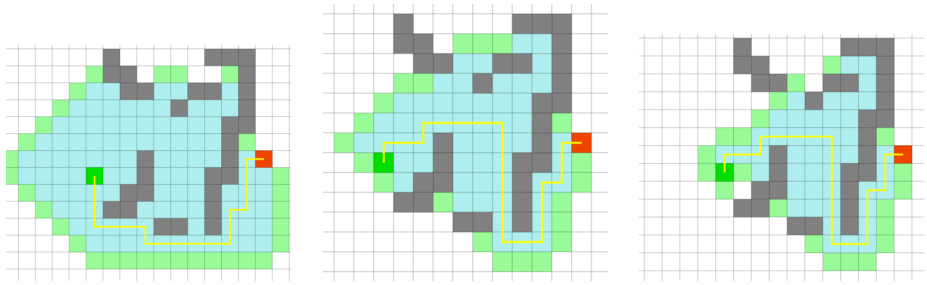
wA*の利点は A*よりもはるかに高速であることである。多くの場合、w の大きさに対して指数的な高速化が期待できる。これは深さ d のノードの個数は d に対して指数的 (分枝度を b とすると b^d 個) であることに対応する。

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 [68]。

図 4.6a、4.6b 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に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*探索は最適解を発見するが、wA*では最初に発見した解が最適とは限らない。



(a) A* (b) wA* (c) GBFS

図 4.6: グリッド経路探索問題における A*、wA*、GBFS の比較

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 [137]。

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

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ても強力な手法である。図 4.6c 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 $\emptyset$

```
while not Goal(s) do
    u ← arg min_{s'∈Expand(s)} h(s');
    parentu ← s;
    s ← u;
return Path(s)
```

この手法は組合せ最適化・連続最適化のためによく使われる。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 [62]。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 $\emptyset$

```
while not Goal(s) do
    T ← {v ∈ Expand(s)};
    T' ← {v ∈ T | h(v) < h(s)};
    while T' = ∅ do
        T ← {v ∈ Expand(u) | u ∈ T};
        T' ← {v ∈ T | h(v) < h(s)};
    u ← arg min_{s'∈Expand(s)} h(s');
    parent(u) ← s;
    s ← u;
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だと思ったら、試しに wA*にしてみて、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*でも解が見つけら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 Python実装

A*探索の実装は非常にシンプルである。グラフ探索のコードに渡すプライオリティ関数を $f = g + h$ にすればよいだけである。

```
from graph_search import GraphSearch

def AstarSearch(problem):
    f = lambda node: problem.heuristic(node.state) + node.g
    return GraphSearch(problem, f)
```

重み付きA*や貪欲最良優先探索もほぼ同様である。

```
from graph_search import GraphSearch
```

```
def WAstarSearch(problem, w=1.0):
    f = lambda node: problem.heuristic(node.state) * w + node.g
    return GraphSearch(problem, f)
```

```
from graph_search import GraphSearch

def GreedyBestFirstSearch(problem):
    h = lambda node: problem.heuristic(node.state)
    return GraphSearch(problem, h)
```

ヒューリスティック関数の実装は問題によっては難しい場合もある。ここではグリッド経路探索のためのマンハッタン距離ヒューリスティック関数を紹介する。

```
        return 1

    def heuristic(self, state):
```

この関数を GridPathfinding クラスに追加すればよい。スライディングタイル問題のためのマンハッタン距離ヒューリスティック関数は以下のようになる。

```
        # Manhattan distance heuristic
        return self.manhattan_distance(state)

    def manhattan_distance(self, state):
        dist = 0
        for tile_id in range(1, len(state.tile)):
            position = state.tile.index(tile_id)

            pos_x = position % self.width
            pos_y = position // self.width

            goal_x = tile_id % self.width
            goal_y = tile_id // self.width
```

各問題に対してそれぞれのヒューリスティック関数を実装することは面倒に感じられるかもしれない。ある特定の問題に特化して高速化させたい場合は、ヒューリスティック関数はその問題の前提知識を利用した関数であるので、それぞれの問題毎に適切なものを実装する必要があるだろう。一方で汎用的に様々な問題に対して使えるヒューリスティック関数もあるので、7 章で紹介する。

4.8 まとめ

状態空間問題に対して何らかの前提知識がある場合、その知識を利用して探索を効率化させる方法の一つとしてヒューリスティック探索がある。ヒューリスティック探索は状態の「有望さ」を推定できる場合、それをヒューリスティック関数として表すことが出来る。

ヒューリスティック探索で基本となる手法は A*探索である。A*は各ノードの有望さをそのノードに到達するまでのコストとそのノードからゴールに到達するまでのコストの推定値の和として推定し、最も有望なノードから優先して探索する手法である。

ヒューリスティック関数のうち重要な性質は 2 つある。許容的なヒューリスティック関数はゴールまでのコスト未満の推定をしないヒューリスティックであり、これを用いた A*は最適解が見つけられることが保証されている。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。

A*で解を見つけるのに時間がかかりすぎる場合は、最適解を見つけることを諦めて wA*や局所探索を用いることができる。

4.9 練習問題

1. グリッド経路探索問題においてマンハッタン距離ヒューリスティックを実装し、A*探索を実装せよ。幅優先探索と比較して展開するノードの

数は変わったか。

2. グリッド経路探索問題において、グリッドが斜めを含めた隣り合う 8 マスに動ける場合を考える。このとき、現在位置とゴール位置とのマンハッタン距離は許容的である。では、現在位置とゴール位置とのユークリッド距離 (直線距離) は許容的なヒューリスティックになるか？

3. 斜め移動ありのグリッド経路探索問題においてマンハッタン距離よりも推定値が正確でありかつ許容的なヒューリスティック関数はあるだろうか？(ヒント：ある。)

4. 上述のヒューリスティック関数を用いた A*探索で斜め移動ありのグリッド経路探索問題を解いてみよう。マンハッタン距離を用いた場合と比較して展開するノードの数は変わったか。

5. ヒューリスティック h が許容的であるとする。このとき、$h'(s) = h(s)+c$ は許容的であるか？c が何を満たせば許容的であるか？(c は定数とする)

6. ヒューリスティック h が無矛盾であるとする。このとき、$h'(s) = h(s)+c$ は無矛盾であるか？c が何を満たせば無矛盾であるか？(c は定数とする)

7. 「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」ことを証明せよ。(ヒント:「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」定理の証明が参考になる。)

8. 山登り法が解を見つけることが出来ないような状態空間問題の例を一つ考えてみよう。

4.10 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。

制限時間内でできるだけ良いクオリティの解が欲しい場合はAnytime Reparing A* [98] が使える。Anytime Reparing A*は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。一般にいつ停止しても現時点で見つかった中で一番良い (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。Anytime アルゴリズムとしては他にもモンテカルロ木探索などがある [17]。

山登り法は評価値が最も良い状態を決定論的に選ぶので解・最適解を発見する保証はない。一方、すべての次状態を均等な確率でランダムに選ぶランダムウォークはいずれ解を発見するが、時間がかかりすぎる。**焼きなまし法** (simulated annealing) この二つの手法の中間をとったような手法であり、最初は評価値を無視して均等なランダムウォークからはじめ、探索が進行していくにつれて評価値に応じて次状態を選択するようにするという手法である [22, 135]。焼きなまし法は最適化問題で非常に重要なアルゴリズムである [81]。

遺伝的アルゴリズム (genetic algorithm) さまざまな最適化問題に適用できる重要なアルゴリズムである [52]。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。

5

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。

探索にかかる実行時間をシンプルに見積もるとすると、(展開したノードの数) と (1展開あたりにかかる時間の平均値) の積である。1展開あたりにかかる時間とは、大きく分けて以下の4つの部分に分けられる。

1. オープンリストからノードを取り出す時間
2. 次状態の生成にかかる時間
3. ヒューリスティック関数の計算にかかる時間
4. 重複検出を行う時間
5. オープンリスト・クローズドリストにノードを入れる時間

ほとんどの問題では次状態の生成とヒューリスティック関数の計算にかかる時間はノードの数によらず定数である。一方、オープンリスト・クロー

ズドリストへのアクセスにかかる時間はノードの数が大きくなるほど増加する。例えばシンプルなヒープでオープンリストを実装した場合、ノードを入れるためにかかる時間は $O(log(n))$ である (n はノード数)。実際、グリッド経路探索問題やスライディングタイル問題などのシンプルな問題においてはデータ構造へのアクセスにかかる時間が探索のほとんどの時間を占めている。よって、探索を効率化するためにはデータ構造の選択が重要である。

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, push) とデキュー (dequeue, pop) 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ことが多い。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 [20] 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 [8]。

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) [47] が効率が良いが、定数項が大き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 [26]。

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プライオリティキュー [31] で実装をすることが出来る [20]。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 [20]。

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

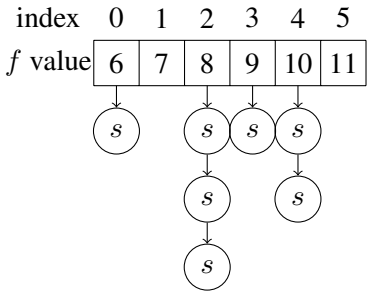


図 5.1: バケットによるオープンリストの実装 ($C_{min} = 6, C_{max} = 11$)

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使われる検索操作が上手くいけばほぼ定数時間で出来るため効率的であることが多い。

5.2.1 ハッシュテーブル (Hash Table)

集合 $U = \{0, 1, ..., N-1\}$ 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 ($k : S \to U$)。キーは更にハッシュテーブルと呼ばれる配列 $T[0, ..., m-1]$ に写される。この写像 $h : U \to \{0, ..., m-1\}$ 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 $S \to \{0, ..., m-1\}$ 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。

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。

$$h(s) := k(s) \bmod m \tag{5.1}$$

このハッシュ関数は S から T への写像になる ($h : S \to \{0, ..., m-1\}$)。$T$ 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。

$$h(s) = \lfloor m(k(s)\phi - \lfloor k(s)\phi \rfloor) \rfloor \tag{5.2}$$

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こ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 $s = (v_0, v_1, ..., v_8)$ 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。スライディングタイルパズルでは**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 [91]。しかしながら完全ハッシュを作るのは簡単であっても、小さく実用的な完全ハッシュを作るのは難しい問題も多い。

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) はこのアイディアに基づき、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 [16, 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 [109] 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 6.3) でも使われる [118]。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 [118, 3, 83]。

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 [89]。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。

アルゴリズム 6: 遅延重複検出を用いたグラフ探索

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$, プライオリティ関数 f

Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1  Closed ← ∅, Open ← {u_0}, d(u_0) ← 0, g(u_0) ← 0;
2  while Open ≠ ∅ do
3      u ← arg min_{u'∈Open} f(u') ;
4      Open ← Open \ {u};
5      if G(u) then
6          return Path(u);
7      s ← parent(u);
8      if u ∉ Closed or g(s) + w(s, u) < g(u) then
9          d(u) ← d(s) + 1;
10         d(u) ← g(s) + w(s, u);
11         parent(u) ← s;
12         for each v ∈ E(u) do
13             Open ← Open ∪ {v};
14             parent(v) ← u;
15     if u ∉ Closed then
16         Closed ← Closed ∪ {u};
17 return ∅;
```

5.3 Python実装

タイブレーキング戦略の実装はシンプルである。プライオリティ関数に複数のプライオリティ値を出力する関数にすればよい。Python の場合タプルを sort で渡せば先頭の値からタイブレーキングをしてくれる。

```
from graph_search import GraphSearch

def TiebreakingAstarSearch(problem):
    f = lambda node: (problem.heuristic(node.state) + node.g,
                                    problem.heuristic(node.state
                                    ))
    return GraphSearch(problem, f)
```

バケットプライオリティキューの実装のためにはオープンリスト・クローズドリストを選べるようにグラフ探索のコードを書き換える必要がある。以下のコードは GraphSearch の引数にオープンリストとクローズドリストを渡せるようにしたものである。

```
from search_node import SearchNode
from util import SearchLogger

def OptimizedGraphSearch(problem, priority_f, open_list,
                                  closed_list):

    init_state = problem.get_init_state()

    init_node = SearchNode(init_state)
    init_node.set_g(0)
    init_node.set_d(0)

    logger = SearchLogger()
    logger.start()

    open_list.push(init_node, priority_f(init_node))
```

```
    closed_list.push(init_node)

    while (len(open_list) > 0):
        node = open_list.pop()
        logger.expanded += 1

        if problem.is_goal(node.state):
            logger.end()
            logger.print()
            return node.get_path()
        else:
            actions = problem.get_available_actions(node.state)

            for a in actions:
                next_state = problem.get_next_state(node.state, a
                                                    )

                next_node = SearchNode(next_state)
                next_node.set_g(node.g + problem.get_action_cost(
                                                    node.state, a))
                next_node.set_d(node.d + 1)

                if not closed_list.is_explored(next_node):
                    next_node.set_prev_n(node)
                    open_list.push(next_node, priority_f(
                                                        next_node))
                    closed_list.push(next_node)
                    logger.generated += 1
                else:
                    logger.pruned += 1

    logger.end()
    logger.print()
    return None
```

バケットオープンリストおよびハッシュテーブルによるクローズドリス

トは以下のように実装できる。

```
class BucketOpenList:
    def __init__(self, C_min, C_max, beam_width=None):
        self.C_min = C_min
        self.C_max = C_max
        self.bucket = [[] for _ in range(C_max - C_min + 1)]
        self.size = 0
        self.beam_width = beam_width

    def __len__(self):
        return self.size

    def pop(self):
        assert self.size > 0
        for i in range(len(self.bucket)):
            if len(self.bucket[i]) > 0:
                self.size -= 1
                return self.bucket[i].pop()
        assert False

    def push(self, item, priority):
        self.bucket[priority - self.C_min].append(item)
        self.size += 1

        if self.beam_width is not None:
            self.shrink()

    def shrink(self, beam_width=None):
        if beam_width is None:
            beam_width = self.beam_width
        if beam_width is not None:
            cur_C = self.C_max - self.C_min
            while self.size > beam_width:
                if len(self.bucket[cur_C]) > 0:
                    self.bucket[cur_C].pop()
                    self.size -= 1
```

```
                else:
                    cur_C -= 1
```

```
class HashClosedList:
    def __init__(self, max_size=10000):
        # This assumes that the hash function is not always a
                                            perfect hasing.
        self.table = [[]] * max_size
        self.table_size = max_size
        self.size = 0

    def __len__(self):
        return self.size

    def push(self, item):
        hash_value = hash(item.state) % self.table_size

        states = [n.state for n in self.table[hash_value]]
        if item.state in states:
            idx = states.index(item.state)

            if item.g < self.table[hash_value][idx].g:
                self.table[hash_value][idx].set_g(item.g)
                self.table[hash_value][idx].set_d(item.d)
                self.table[hash_value][idx].set_prev_n(item.
                                                    prev_n)
        else:
            self.table[hash_value].append(item)
            self.size += 1

    def is_explored(self, item):
        hash_value = hash(item.state) % self.table_size

        for n in self.table[hash_value]:
            if (n.state == item.state) and (n.g <= item.g):
                return True
```

```
        return False
```

バケットオープンリストにある `beam width` は後述するビームサーチのために用いる。ビームサーチを利用する予定がなければ無視してよい。

5.4 まとめ

探索にかかる時間はおよそ (展開したノードの数) と (1 展開あたりにかかる時間の平均値) の積である。1 展開あたりにかかる時間の多くがデータ構造へのアクセスにかかっているのであれば、データ構造を適切に選択することでアルゴリズムの実行時間を向上させることができる可能性がある。

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プライオリティキューが強力である [20]。タイブレーキング戦略はシンプルで強力な戦略であるので、これも積極的に使うことをお勧めする。

5.5 練習問題

1. 3x3 のスライディングタイル問題のオープンリストにタイブレーキングを実装してみよう。f 値が最小のノードが複数ある場合に h 値が最小のノードを優先して展開するタイブレーキング戦略と、逆に h 値が最大のノードを優先する戦略を実装し、性能を比較してみよう。どちらの方が効率的だったか？それは何故か？

2. グリッド経路探索問題において完全ハッシュを作ることはできるか？

3. 3x3 のスライディングタイル問題で遅延重複検出を実装してみよう。展開時に重複検出をする場合と比較し、展開ノード数はどちらが多かったか？生成ノード数はどちらが多かったか？

5.6 関連文献

ヒューリスティック探索の性能がデータ構造の選択によって大きく左右されることは古くから知られていたようであるが、それを学術論文としてまとめたものは多くはない。Burns et al. (2012) [20] は高速なヒューリスティック探索のコードを実装するための方法をまとめた論文である。データ構造の選択によって性能が大きく変わるということを指摘した論文である。オープンリストのタイブレーキングに関しては Asai&Fukunaga (2016) [8] を参照されたい。

6

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、その数だけノード展開関数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 [61]。

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。

ビームサーチ (beam-search) は非最適解を高速に見つけるための手法である。実行時間・メモリ消費の双方が限られた状況においてなるべく良い解を見つけたい場合に使われる。自然言語処理などでも使われている。**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ところを探す方法である。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 ビームサーチ (Beam Search)

ビームサーチ (beam-search) は実行時間・メモリ消費の双方が限られた状況においてなるべく良い解を見つけたい場合に使われる。実装は非常にシンプルである。探索中にオープンリストがビーム幅よりも大きくなった場合、最も有望ではないノードを取り除いていくというものである。プライオリティ関数を g とする場合、すなわち幅優先探索とする場合を特にビームサーチと呼ぶ場合もある。ビームサーチは許容的ではない。有望ではないノードを探索から取り除いているため、最適解を発見する保証がない。ビームサーチのメリットはメモリ効率である。オープンリストの大きさを一定に保つため、メモリの消費量が抑えられる。より正確にはビームサーチのメモリ消費量はビーム幅と探索の深さの積に比例する ($O(kd)$)。特に重複検出を必要としないような問題であればクローズドリストも取り除くことでメモリ消費量を探索ノード数に関わらず一定にすることができる。メモリにハード制約があるような状況下であるときに有効である。ビームサーチはビーム幅を適切に設定することが良い解を発見するために重要である。ビーム幅を動的に設定する方法はいくつか提案されている [97]。

アルゴリズム 7: ビームサーチ (Beam Search)

Input : 非明示的状態空間グラフ $(s, Goal, Expand, w)$、プライオリティ関数 f、ビーム幅 k

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

```
1  Open ← {s}, Closed ← {s}, d(s) ← 0, g(s) ← 0;
2  while Open ≠ ∅ do
3      u ← arg min_{u'∈Open} f(u') ;
4      Open ← Open \ {u};
5      while |Open| > k do
6          u ← arg max_{u'∈Open} f(u') Open ← Open \ {u};
7      if Goal(u) then
8          return Path(u);
9      for each v ∈ Expand(u) do
10         if v ∉ Closed or g(u) + w(u, v) < g(v) then
11             Open ← Open ∪ {v};
12             d(v) ← d(u) + 1;
13             g(v) ← g(u) + w(u, v);
14             parent(v) ← u;
15         if v ∉ Closed then
16             Closed ← Closed ∪ {v};
17 return ∅;
```

6.2 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のような問題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 8)。

アルゴリズム 8: 深さ優先分枝限定法 (Branch-and-Bound)

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$、ヒューリスティック関数 h

Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1 U ← ∞;
2 P ← ∅;
3 (U, P) ← DFB&B((E, u_0, G, w), 0, U, P);
4 return P;
```

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、**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、

アルゴリズム 9: DFB&B$((\mathcal{E}, u, \mathcal{G}, w), g, U, P)$: 分枝限定法の再帰計算

```
1  if G(u) then
2      if g < U then
3          U ← g;
4          P ← Path(u);
5  else
6      Succ(u) ← E(u), sorted according to h;
7      for v ∈ Succ(u) do
8          if g + h(v) < U then
9              (U, P) ← DFB&B((E, v, G, w), g + w(u, v), U, P);
10 return (U, P)
```

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 [98, 54, 120]。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。

6.3 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 [87, 123]。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 (CLDFS) を実行する (アルゴリズム 10)。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。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およそ $1 + b + b^2 + ... + b^{c^*} = O(b^{c^*})$ 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 [87]。

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 10: 反復深化深さ優先 (Depth First Iterative Deepening)

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$
Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1 U' ← 0;
2 P ← ∅;
3 while P = ∅ and U' ≠ ∞ do
4 |   U ← U';
5 |   (U', P) ← CLDFS-DFID((E, u_0, G, w), 0, U);
6 return P;
```

6.3.1 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 [87]。DFID と同様、コストを大きく

アルゴリズム 11: CLDFS-DFID: DFID のためのコスト制限付き深さ優先

Input : 非明示的状態空間グラフ $(\mathcal{E}, u, \mathcal{G}, w)$、経路コスト g、閾値 U

Output : u からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1  U' ← ∞;
2  if G(u) then
3  |  return (U', Path(u));
4  for each v ∈ E(u) do
5  |  if g + w(u, v) ≤ U then
6  |  |  U', P ← CLDFS-DFID((E, v, G, w), g + w(u, v), U);
7  |  |  if P ≠ ∅ then
8  |  |  |  P' ← (u, P);
9  |  |  |  return (U', P');
10 |  else if g + w(u, v) < U' then
11 |  |  U' ← g + w(u, v);
12 return (U', ∅);
```

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 13 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 11 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 [87]。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 12: 反復深化 A* (Iterative Deepening A*)

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$、ヒューリスティック h

Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
U' ← h(u_0);
P ← ∅;
while P = ∅ and U' ≠ ∞ do
    U ← U';
    (U', P) ← CLDFS-IDA((E, u_0, G, w), 0, U);
return P;
```

6.4 両方向探索 (Bidirectional Search)

両方向探索 (bidirectional search) は初期状態とゴールの両方から探索をする

アルゴリズム 13: CLDFS-IDA: IDA*のためのコスト制限付き深さ優先

Input : 非明示的状態空間グラフ $(\mathcal{E}, u, \mathcal{G}, w)$、経路コスト g、閾値 U

Output : コストの上界と u からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1  U' ← ∞;
2  if G(s) then
3  |  return (U', Path(u));
4  for each v ∈ E(u) do
5  |  if g + w(u,v) + h(v) > U then
6  |  |  if g + w(u,v) + h(v) < U then
7  |  |  |  U' ← g + w(u,v) + h(v);
8  |  else
9  |  |  U', P ← CLDFS-IDA(v, g + w(u,v), U);
10 |  |  if P ≠ ∅ then
11 |  |  |  P' ← P||u;
12 |  |  |  return (U', P');
13 return (U', ∅);
```

手法である [115]。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。両方向探索が可能であるためにはゴール条件が具体的にゴール状態あるいはゴール状態の集合として得られることが必要である。

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 $O(b^{d/2}) \cdot 2$ であるので、正方向探索よりも非常に少ない展開ノード数で探索が出来るポテンシャルがある。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)。

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 [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。両方向探索は最悪の場合 A*探索のおよそ倍のノード展開数になるが、単純な両方向ヒューリスティック探索だとノード展開数は A*の倍程度になることが実験的に示されている [10]。

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る (meet-in-the-middle) ことを保証する方法が 2016 年に提案された [66]。

6.5 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。

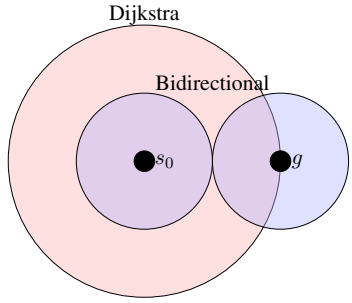


図 6.1: 両方向探索

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 [23]。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。

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 14 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 [106]。

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 $R(i)$ に保存する。$R(i)$ 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

[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

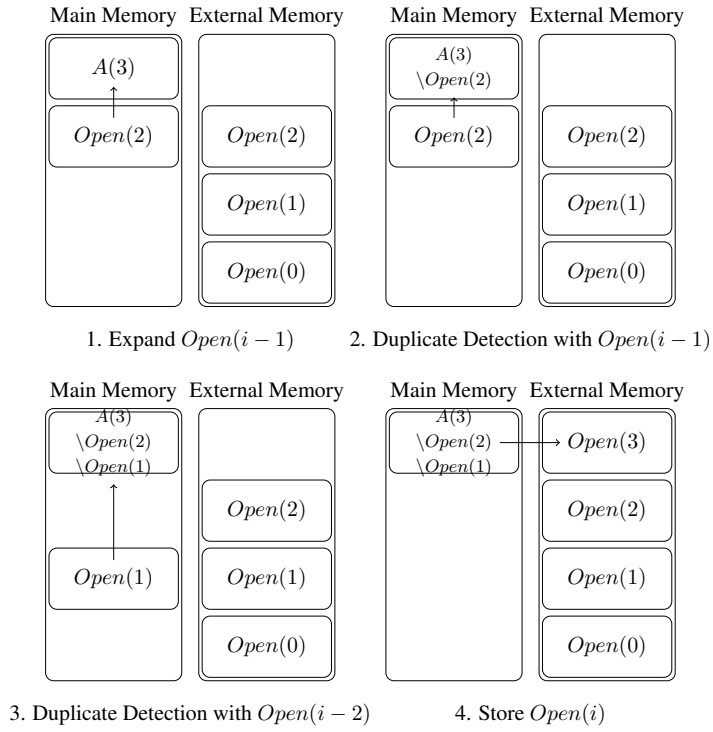


図 6.2: 外部メモリ幅優先探索の操作

アルゴリズム 14: 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 $(\mathcal{E}, u_0, \mathcal{G}, w)$
Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1  Open(−1) ← ∅;
2  Open(0) ← {u_0};
3  i ← 1;
4  while Open(i − 1) ≠ ∅ do
5  |  if ∃(u ∈ Open(i − 1))G(u) then
6  |  |  return Path(u);
7  |  R(i) ← u|u ∈ E(u'), u' ∈ Open(i − 1);
8  |  for each v ∈ R(i) do
9  |  |  parent(v) ← u s.t. v ∈ E(u);
10 |  Open(i) ← R(i) \ (Open(i − 1) ∪ Open(i − 2));
11 |  i ← i + 1;
12 return ∅;
```

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 u と同じ状態は深さ $d(u)-2, d(u)-1, d(s)$ にしか現れない。よって深さ i のノードの集合 $R(i)$ 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、$Open(0), Open(1), ..., Open(i-1)$ 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。

外部メモリを利用した探索は幅優先に限らず、様々な (整数の) プライオリティ関数に対応してデザインすることができる。外部メモリ A*は外部メモリを用いた A*探索である [36]。

6.6 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 $\mathbb{B}^n \to \mathbb{B}$ を効率良く表すグラフ構造である [4, 18]。**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 [37, 38]。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 [134]。

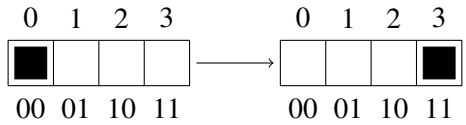


図 6.3: 特徴表現の例

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	$x_0 x_1$

6.6.1 特徴表現 (Symbolic Representation)

説明のために非常なシンプルなスライディングタイルパズルを考える (図 6.3)。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 ($2^S = \emptyset, \{0\}, \{1\}, \{2\}, \{3\}, \{0, 1\}, ..., \{0, 1, 2, 3\}$)。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 $Trans : S \times S \to \{0, 1\}$ 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)。

図 6.3 の問題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

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

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)。

6.6.2 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。

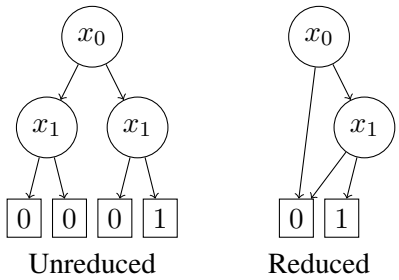


図 6.4: x_0x_1 を表す決定木 (左) と BDD (右)

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 $x_i(i \in \{1, ..., n\})$ 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。

6.6.3 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。

$$Image_S(x') = \exists x(Trans(x, x') \wedge \phi_S(x)) \tag{6.2}$$

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 ($Image_{Open}(x) \wedge \neg Closed(x)$)。

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：

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

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 15 はシンボリック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。

シンボリック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。

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

アルゴリズム 15: シンボリック幅優先木探索 (Symbolic Breadth-first Tree Search)

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$
Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

$S_0 \leftarrow \{u_0\}$;
for $i \leftarrow 1, 2, ...$ **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 $Construct(\phi_{S_i} \wedge \phi_T(x), i)$;

ルゴリズム 16 はシンボリック幅優先グラフ探索のコードである。アルゴリズム 15 と異なり特徴関数 $Closed$ を用いて重複検出を行っている。

ヒューリスティック関数を用いたシンボリック探索としてはシンボリック A*などがある [37]。シンボリック幅優先探索では g 値毎にオープンリストを分けていたがシンボリック A*では f 値と h 値のペア毎にオープンリストを分けることで f 値の小さい状態を優先して探索する。

6.7 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 16: シンボリック幅優先探索 (Symbolic Breadth-first search)

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$
Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1 S_0 ← {u_0};
2 Closed ← {u_0};
3 for i ← 1, 2, ... do
4     Succ(x) ← ∃x(φ_{S_{i-1}}(x) ∧ Trans(x, x'))[x'/x];
5     φ_{S_i}(x) ← Succ(x) ∧ ¬Closed(x);
6     Closed(x) ← Closed(x) ∨ Succ(x);
7     if φ_{S_i}(x') ∧ φ_T ≠ false then
8         return Construct(φ_{S_i} ∧ φ_T(x), i);
```

6.7.1 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 [96]。なので新奇性の定義も様々であるが、本書では [50] 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 $h_1, h_2, ..., h_m$ 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の値を返す関数を使うことができる [50, 101]。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。

6.7.2 幅制限探索 (Width-Based Search)

アルゴリズム 17: 幅制限探索 (Width-based search)

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$、特徴関数 $h_0, h_1, ..., h_{m-1}$、幅 i
Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

$Open \leftarrow \{u_0\}, Closed \leftarrow \emptyset$;
while $Open \neq \emptyset$ **do**
  $u \leftarrow \arg\min_{u' \in Open} f(u')$;
  $Open \leftarrow Open \setminus \{u\}$;
  if $\mathcal{G}(u)$ **then**
    return $Path(u)$;
  for *each* $v \in \mathcal{E}(u)$ **do**
    $H(v) \leftarrow \{j | j \in \{0, 1, ..., m-1\}, h_j(v) = 1\}$;
    $H \leftarrow \{j | j \in \{0, 1, ..., m-1\}, h_j(v) \in Closed\}$;
    if $|H \setminus Closed| \geq i$ **then**
      $Open \leftarrow Open \cup \{v\}$;
      $parent(v) \leftarrow u$;
    for *each* $h \in H(v) \setminus Closed$ **do**
      $Closed \leftarrow Closed \cup \{h\}$;
return $\emptyset$;

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 [100]。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)。

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 [50]。

6.7.3 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 [100]。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。

6.8 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

アルゴリズム 18: 反復幅制限探索 (Iterative Width Search)

```
1 i ← 1;
2 P ← ∅;
3 while P = ∅ do
4 |   P ← IW(i);
5 |   i ← i + 1;
6 return path
```

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 [48] を参照されたい。

6.8.1 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。[71] 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：

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

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：

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

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：

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

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。

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

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 [19, 82].

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。

6.8.2 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 [69]。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 [19]。

6.8.3 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ずらいという**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*) [82] 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 [82] では Zobrist hashing [142] 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：

$$Z(s) := R_0[x_0] \; xor \; R_1[x_1] \; xor \; \cdots \; xor \; R_n[x_n] \tag{6.8}$$

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 [19]。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 [70]。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：

アルゴリズム 19: Hash Distributed A*

Input : 非明示的状態空間グラフ $(\mathcal{E}, u_0, \mathcal{G}, w)$、ヒューリスティック関数 h、ハッシュ関数 H

Output : u_0 からゴール状態への経路、経路が存在しなければ $\emptyset$

```
1  Closed_p ← ∅, Open_p ← ∅, Buffer_p ← ∅ for each thread p;
2  Open_{H(u_0)} ← {u_0}, g(u_0) ← 0;
3  goal ← ∅, c' ← ∞;
4  for each process p do
5      while not TerminateDetection() do
6          for each (v, g_v, p_v) ∈ Buffer_p do
7              if v ∉ Closed or g_v < g(v) then
8                  Open ← Open ∪ {v};
9                  g(v) ← g_v;
10                 parent(v) ← p_v v;
11             if v ∉ Closed then
12                 Closed ← Closed ∪ {v};
13         Buffer_p ← ∅;
14         while Open_p ≠ ∅ and min_{s∈Open_p} f(s) < c' do
15             u ← arg min_{u'∈Open} f(u') ;
16             Open ← Open \ {u};
17             if G(u) and g(u) < c' then
18                 goal ← u;
19                 c' ← g(u);
20             for each v ∈ E(u) do
21                 g_v ← g(u) + w(u, v);
22                 Buffer_{H(v)} ← Buffer_{H(v)} ∪ {(v, g_v, u)};
23     return Path(goal);
```

アルゴリズム 20: Zobrist Hashing

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 $hash$;

アルゴリズム 21: Zobrist Hashing の初期化

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 $R = (R_1, R_2, ..., R_n)$

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

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*のための分配関数の自動生成方法は [71] にまとめられている。

6.9 Python 実装

分枝限定法の実装は再帰による。再帰による深さ優先探索のコードとほぼ同様である。

```
from search_node import SearchNode
from util import SearchLogger
```

```
logger = SearchLogger()

def BnBEngine(problem, cur_node, best_solution, path=None):
    logger.expanded += 1
    if problem.is_goal(cur_node.state):
        if cur_node.g < best_solution:
            best_solution = cur_node.g
            path = cur_node.get_path()
    else:
        actions = problem.get_available_actions(cur_node.state)

        for a in actions:
            next_state = problem.get_next_state(cur_node.state, a
                                                )

            next_node = SearchNode(next_state)
            next_node.set_g(cur_node.g + problem.get_action_cost(
                                                cur_node.state, a))
            next_node.set_d(cur_node.d + 1)
            next_node.set_prev_n(cur_node)

            if next_node.g + problem.heuristic(next_state) <
                                                best_solution:
                logger.generated += 1
                best_solution, path = BnBEngine(problem,
                                                    next_node,
                                                    best_solution,
                                                    path)
            else:
                logger.pruned += 1

    return (best_solution, path)

def Branch_and_Bound(problem, best_solution=None):
    init_state = problem.get_init_state()
```

```
    init_node = SearchNode(init_state)
    init_node.set_g(0)
    init_node.set_d(0)

    if best_solution is None:
        best_solution = float('inf')

    logger.start()
    solution_cost, path = BnBEngine(problem, init_node,
                                    best_solution)

    logger.end()
    logger.print()
    return (solution_cost, path)
```

反復深化深さ優先探索は、再帰による深さ優先探索のコードを変更するだけで実装できる。

```
from search_node import SearchNode
from util import SearchLogger

logger = SearchLogger()

def CLDFS_DFID(problem, cur_node, max_priorty, priority_f):
    logger.expanded += 1
    if problem.is_goal(cur_node.state):
        return [cur_node]
    else:
        actions = problem.get_available_actions(cur_node.state)

        for a in actions:
            next_state = problem.get_next_state(cur_node.state, a
                                                )
            next_node = SearchNode(next_state)
            next_node.set_g(cur_node.g + problem.get_action_cost(
                                                cur_node.state, a))
            next_node.set_d(cur_node.d + 1)
```

```
            next_node.set_prev_n(cur_node)

            if priority_f(next_node) <= max_priorty:
                logger.generated += 1
                path = CLDFS_DFID(problem, next_node, max_priorty
                                                    , priority_f)
                if len(path) > 0:
                    path.append(cur_node)
                    return path
            else:
                logger.pruned += 1
    return []

def IterativeDeepening(problem, priority_f):
    logger.start()
    max_priorty = 1
    path = []

    while len(path) == 0:
        init_state = problem.get_init_state()
        init_node = SearchNode(init_state)
        init_node.set_g(0)
        init_node.set_d(0)
        init_node.set_prev_n = 0

        path = CLDFS_DFID(problem, init_node, max_priorty,
                                        priority_f)

        max_priorty += 1

    logger.end()
    logger.print()
    return path


def DepthFirstIterativeDeepening(problem):
```

```
    return IterativeDeepening(problem, lambda node: node.g)
```

反復深化 A* (IDA*) はこのコードを使えばプライオリティ関数を変えるだけで実装できる。

```
from depth_first_iterative_deepening import IterativeDeepening

def IterativeDeepeningAstar(problem):
    return IterativeDeepening(problem, lambda node: node.g +
                                        problem.heuristic(node.state
                                        ))
```

並列探索の実装は手間がかかる。特に計算機クラスターなどの分散メモリ環境で動くコードを実装するためには MPI ライブラリなどを使い、message passing を行う必要がある。ここでは 1 台のコンピュータで実行される共有メモリ環境で動くコードを実装する。プロセスではなくスレッドを使って並列化を行う。

```
from threading import Thread, Lock
from multiprocessing import Value, Array
import queue

from search_node import SearchNode
from util import SearchLogger

from bucket_openlist import BucketOpenList
from hash_closedlist import HashClosedList

def terminate_detection(has_job, terminating, confirm_terminating
                                    , n_threads, thread_id):
    if not has_job:
        terminating[thread_id] = True

        for i in range(n_threads):
            if not terminating[i]:
                return False
```

```
        confirm_terminating[thread_id] = True

        for i in range(n_threads):
            if not confirm_terminating[i]:
                return False
        return True
    else:
        terminating[thread_id] = False
        confirm_terminating[thread_id] = False
        return False

def search_thread(problem,
                  priority_f,
                  open,
                  closed,
                  buffers,
                  n_threads,
                  thread_id,
                  incumbent_cost,
                  incumbent_goal_node_pos,
                  incumbent_lock,
                  terminating,
                  confirm_terminating,
                  logging):

    terminating[thread_id] = False
    confirm_terminating[thread_id] = False

    while (True):
        while not buffers[thread_id].empty():
            recv_node = buffers[thread_id].get()
            f = priority_f(recv_node)
            if (not closed.is_explored(recv_node)) and (f <
                                                incumbent_cost.value
                                                ):
                open.push(recv_node, f)
```

```
                closed.push(recv_node)
                logging.generated += 1
            else:
                logging.pruned += 1

        if terminate_detection(len(open) > 0, terminating,
                                        confirm_terminating,
                                        n_threads, thread_id):
            break

        if (len(open) == 0):
            continue

        node = open.pop()
        logging.expanded += 1

        if problem.is_goal(node.state) and (node.g <
                                        incumbent_cost.value):
            incumbent_lock.acquire()
            incumbent_cost.value = node.g
            idx = closed.find(node)
            incumbent_goal_node_pos[0] = thread_id
            incumbent_goal_node_pos[1] = idx[0]
            incumbent_goal_node_pos[2] = idx[1]
            print("incumbent solution updated: ", incumbent_cost.
                                            value)
            incumbent_lock.release()
        else:
            # Expand the node
            actions = problem.get_available_actions(node.state)

            for a in actions:
                next_state = problem.get_next_state(node.state, a
                                                    )

                next_node = SearchNode(next_state)
```

```
                next_node.set_g(node.g + problem.get_action_cost(
                                                    node.state, a))
                next_node.set_d(node.d + 1)
                next_node.set_prev_n(node)

                f = priority_f(next_node)
                if  (f < incumbent_cost.value):
                    dst = hash(next_state) % n_threads
                    buffers[dst].put(next_node)

    return

def HashDistributedGraphSearch(problem, priority_f=None,
                                n_threads=2):

    init_state = problem.get_init_state()

    init_node = SearchNode(init_state)
    init_node.set_g(0)
    init_node.set_d(0)
    init_node.set_prev_n = 0

    init_dst = hash(init_state) % n_threads

    init_h_value = priority_f(init_node)
    opens = [BucketOpenList(C_min=init_h_value, C_max=
                                    init_h_value*8)] * n_threads
    closeds = [HashClosedList()] * n_threads

    opens[init_dst].push(init_node, priority_f(init_node))
    closeds[init_dst].push(init_node)

    # Data structure to keep the incumbent solution
    incumbent_lock = Lock()
    incumbent_cost = Value('f', 1000000000000000.0)
    incumbent_goal_node_pos = Array('i', [0, 0, 0])
```

```
# Data structure to keep the termination status
terminating = Array('b', [False] * n_threads)
confirm_terminating = Array('b', [False] * n_threads)

# Buffer for asynchronous message passing
waiting_buffers = [queue.Queue() for i in range(n_threads)]

loggings = [SearchLogger() for i in range(n_threads)]
threads = []
for i in range(n_threads):

    t = Thread(target=search_thread,
               args=(problem,
                     priority_f,
                     opens[i],
                     closeds[i],
                     waiting_buffers,
                     n_threads,
                     i,
                     incumbent_cost,
                     incumbent_goal_node_pos,
                     incumbent_lock,
                     terminating,
                     confirm_terminating,
                     loggings[i]))
    threads.append(t)

global_logger = SearchLogger()

global_logger.start()

for i in range(n_threads):
    threads[i].start()

for i in range(n_threads):
```

```
        threads[i].join()


    pos = incumbent_goal_node_pos.get_obj()
    incumbent_goal_node = closeds[pos[0]].get_by_index(pos[1],
                                        pos[2])
    path = incumbent_goal_node.get_path()

    global_logger.end()
    global_logger.expanded = sum([l.expanded for l in loggings])
    global_logger.generated = sum([l.generated for l in loggings]
                                        )
    global_logger.pruned = sum([l.pruned for l in loggings])
    global_logger.print()
```

この実装は早くなっただろうか？おそらくこの実装で実行した場合、並列化の恩恵を受けることはほとんどないだろう。並列化の恩恵を受けるためには、より複雑な問題を解くか、あるいはより洗練された message passing を行う必要がある。

6.10 まとめ

ヒューリスティック探索の手法は数多く提案されており、それぞれ様々な長所・短所がある。時間・空間制限がありA*探索ではうまくいかない条件であっても、別のヒューリスティック探索手法であれば解決できるかもしれない。すべての状況で有効な手法は存在しないため、状況に応じて適切な手法を選択する必要がある。

これらの手法はA*探索と比較してやや複雑なため、実装するのに苦労するかもしれない。可能であれば、その手法を実装して試す前に事前に「その手法で本当に目の前の問題を解決できるのか？」を見積もることが出来るとよいだろう。

6.11 練習問題

1. あなたはグリッド経路探索問題を解こうとしている。あなたの目的は経路を見つけることであり、その経路の長さは重要ではない。このとき分子限定法を用いることは適切か？

2. (難問) 直感的には両方向探索は有効に思えるかもしれないが、実はほとんどの場合単方向探索の性能を超えることが難しい。何故だろうか？

3. 反復深化 A*探索と A*の秒間のノードの展開速度を比較してみよう。どちらの方が速いだろうか？また、どのような問題に対してどちらの方が適しているだろうか？

4. 今あなたはある状態空間問題を解こうとしている。経験的に、A*探索を使った時解の深さが d 程度である場合まで解けていることが分かっている。それ以上解が深い場合はメモリが足りず解くことが出来ないようである。このとき、外部メモリ A*探索を使うことで、どのくらいの深さまで解くことができるか見積もることはできるだろうか？ただし、その問題の分枝度は 4 であり、外部メモリはおよそ内部メモリの 1000 倍程度であるとする。

5. さらに解が深い問題を解くためにはどのようなアルゴリズムを使うと良いだろうか？このとき実行時間に制限はないものと仮定する。

6. どのような特徴を持った問題であれば並列化によって解の発見を高速化しやすいだろうか？

6.12 関連文献

両方向探索で正方向ではゴール状態までのコストの推定、逆方向では初期状態までのコストの推定を用いたヒューリスティック探索を Front-to-End 戦略

と呼ぶ [115]。Front-to-Front 戦略は反対方向のオープンリストに入っている状態への距離の推定を用いたアルゴリズムである [127]。Front-to-Front 戦略はコストの推定がより正確である代わりに各オープンリストに入っている状態のペアすべてに対してヒューリスティック値を計算し、それを更新し続けなければならない。Perimiter search はこの問題を解決するため、探索の方向を一度だけ変える手法である [33, 103]。最初は逆方向探索である深さまで探索をしてからあとは正方向で探索を行う。

[140] は状態空間をいくつかの部分空間に分割し、その分割内でそれぞれオープン・クローズドリストを持つという Structured Duplicate Detection を用いた外部メモリ探索を提案した。Solid state drive (SSD) は HDD と比べて高速なだけでなく、得意なアクセスパターンが異なる。SSD の特徴を利用した外部メモリ探索アルゴリズムも提案されている [99]。

現在提案されているシンボリック探索はほとんど BDD を用いたものである。BDD 以外のデータ構造、例えば Zero-surpressed Decision Diagram [107] などをヒューリスティック探索に使う例は著者の知る限りない。

[93] は並列深さ優先探索のさまざまな実装を比較した。ハッシュ関数によって状態をプロセスに振り分けるアイディアは Parallel Retracting A* (PRA*) からである [43]。PRA*はプロセス間の同期処理が行っていたのに対して、Transposition Table Driven Work Distribution (TDS) [122] は同期処理を行わないことによって性能を改善した。

執筆時現在、GPU や FPGA などを用いたアルゴリズムはあまり例がない。GPU-Parallel A*はオープンリストをいくつものキューに分けて並列化した [141]。探索に GPU を用いる場合の一番の問題は GPU のメモリが非常に小さいことである。そのため A*ではなく IDA*の並列化の研究もおこなわれている [67]。GPU と CPU の両方を使った探索アルゴリズムも提案されている [131]。

7

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 [51]。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 [44]。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 [44]。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 [60, 129]、セルアセンブリ [7]、遺伝子距離計算 [41]、ビデオゲーム [101] など、様々な応用問題を含むモデルである。

[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。

7.1 定義

古典的プランニング問題の最も基本となる STRIPS モデル [44] に従って定義する。

定義 16 (STRIPS 問題 $P = (AP, A, s_0, Goal)$ は命題変数の集合 AP、アクションの集合 A、初期状態 $s_0 \subseteq AP$、ゴール条件 $Goal \subseteq AP$ からなる。アクション $a \in A$ には適用条件 $pre : A \to 2^{AP}$、追加効果 $add : A \to 2^{AP}$、削除効果 $del : A \to 2^{AP}$ からなる。アクション a は適用条件 $pre(a)$ の命題を全て含む状態にのみ適用可能である。追加効果 $add(a)$ はアクション a を適用すると状態に追加される命題の集合であり、削除効果 $del(a)$ はアクション a を適用すると削除される命題の集合である。古典的プランニングの目的は与えられた初期状態 s_0 からはじめ、ゴール条件に含まれる命題がすべて含まれる状態に到達するまでのアクションの列を求めることである。):

STRIPS 問題は状態空間問題の一種である。状態空間は 2^{AP} であり、アクション a を状態 s に適用後の状態 $s' = a(s)$ は

$$a(s) = (s \cup add(a)) \setminus del(a) \tag{7.1}$$

である。状態空間問題であるので、状態空間グラフを考えることができ、ヒューリスティック探索によって解くことができる問題である。

上の定義では集合の言葉で定義したが、人工知能の言葉、命題論理の言葉で説明することもできる。STRIPS モデルは**命題論理** (propositional logic) で書かれたモデルである。**閉世界仮説** (closed-world assumption) を仮定し、真であると示されていない命題は全て偽であるとする。状態 $s_0 = p_1 \wedge p_2 \wedge ...$ は命題論理で書かれた世界の状態である。アクションの適用条件はアクションを適用するために真であるべき命題である。追加効果はアクションを適用した後に真になる命題、削除効果は偽になる命題である。ゴール条件はゴール状態で真であるべき命題である。

このように古典的プランニングは論理エージェントと非常に密接な関係がある。古典的プランニングに興味がある方は [123] を参照されたい。

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** (first-order predicate logic) 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、インスタンスはその個例に対応する。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`)、ゴール条件 (`goal`) がある。これら以外にも例えばオブジェクトの型などを定義することができるが、簡単のためここではこれら基本の文法のみを紹介する。

PDDL に記述された一階述語論理は命題論理に変換できる。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

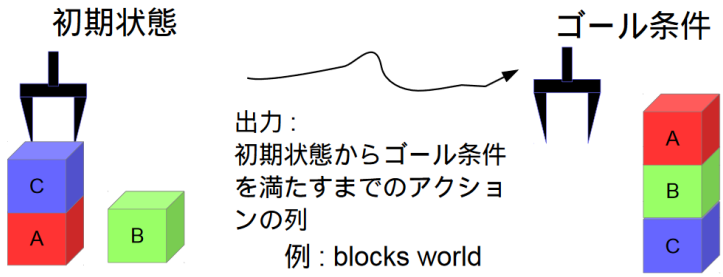


図 7.1: Blocks world ドメイン

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, 46]。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。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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) (HANDEMPTY))
(:goal (AND (ON A B) (ON B C)))
)
```

図 7.3: blocks-world の instance ファイル

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 (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。

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。[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。[41]

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。

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。

各アルゴリズムの実装は fast-downward [2]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。

[2] `http://www.fast-downward.org`

7.4.2 適用条件緩和 (Precondition Relaxation)

適用条件緩和 (precondition relaxation) はアクションの適用条件を無視し、すべてのアクションをすべての状態で適用できる緩和問題を解くヒューリスティックである。すべてのアクションが適用できるようになるので、グラフのエッジの数が増えることになる。このとき、すべてのゴール条件の命題は 1 ステップで満たすことができる。適用条件緩和はゴールカウントヒューリスティックに近いが、少しだけ適用条件緩和の方が正確である。なぜなら適用条件緩和の場合、1. 複数のゴールを同時に満たすアクションがある場合、そのアクションを 1 ステップで実行することができ、2. アクションを適用することによって一旦満たされた命題が削除されることがあるからである。適用条件緩和された問題は元の問題と比べて非常に簡単になっているが、まだまだ難しい。なので更に緩和し、一度満たされた命題が削除されないとすることが多い。こうすると緩和問題は、追加効果 $add(a)$ の和集合がゴール条件を全て満たす $Goal \subseteq \bigcup_{a \in A'} add(s)$ ような最小のアクション集合 A' を求める問題になる。これはまさしく**集合被覆問題** (set cover) である [75]。集合被覆問題でもまだ NP 困難問題であるがシンプルな貪欲で $\log n$ の近似アルゴリズムになる [25]。ただし近似アルゴリズムを使う場合、許容的なヒューリスティックではなくなってしまう。

7.4.3 削除緩和 (Delete Relaxation)

削除緩和 (delete relaxation) はアクションの削除効果を無視する緩和問題を用いたヒューリスティックである [64]。この緩和問題では各アクション $a \in A$ の代わりに a^+ を用いる。a^+ は a^+ と同じ適用条件、追加効果を持っているが削除効果が空集合である ($del(a^+) = \emptyset$)。この緩和問題における最適解のコストをヒューリスティック関数 h^+ として用いる。緩和問題では削除効果がないので状態に含まれる命題変数は単調増加していく。そのため元の問題よりも簡単になるが、これでもまだ NP 困難である [21]。そのため h^+ を非許容的に見積もるヒューリスティック、例えば additive heuristic [15]、FF heuristic

[64]、pairwise max heuristic [108]、set-additive heuristic [80] などがつかわれる。これらを用いた場合得られるヒューリスティックは非許容的になる。

h^+ を許容的に見積もるヒューリスティックとしては max heuristic h^{max} がある [15]。h^{max} はゴール条件の各命題 $t \in Goal$ に対してその命題一つのみをゴール条件と更に緩和した問題 ($del(a^{max}) = \emptyset, Goal^{max} = t$) を解く。このコストを $c(t)$ として、h^{max} はその最大値である ($h^{max} = \max_{t \in Goal} c(t)$)。$h^{max}$ は許容的であるが、非許容的なヒューリスティックよりも探索の効率が悪いこと多いことが実験的に示されている。ちなみに additive heuristic は最大値の代わりに $c(t)$ の総和を取るものである。非許容的である代わりに h^{max} よりも性能が良いことが多い。

7.4.4 最長経路 (Critical Path)

最長経路ヒューリスティック (critical path heuristic) haslum:00 は命題の集合を全て満たすための最小コスト $c(X)$ を X の大きさ m 以下の部分集合 $K \subseteq X$ の最大値で近似する ($c^m(X) = \max_{X' \subseteq X, |X'| \leq m} c^m(X')$) というアイディアに基づいたヒューリスティックである。この近似は下界になるので許容的なヒューリスティックが得られる。max heuristic は最長経路ヒューリスティックのうち $m = 1$ の場合である ($h^{max} = h^1$)。$h^m \geq h^{m-1}$ なので m が大きいほど正確な見積もりが出来るが、同時に計算コストが m に対して指数的に伸びるトレードオフがある。

元々GraphPlan と呼ばれる並行プランナーで使われたアイディアに基づいている [14]。

7.4.5 抽象化 (Abstraction)

抽象化 (abstraction) ヒューリスティックは状態 s を抽象状態 $\alpha(s)$ への写像を用いたヒューリスティックである。ヒューリスティック値 $h^\alpha(s)$ は抽象化された状態空間 $S^\alpha = \{\alpha(s) | s \in S\}$ でのゴールへの距離である。抽象化は元の問題よりも簡単になるので許容的なヒューリスティックが得られる。ヒューリ

スティックの性能は α の選択による。α の選択方法としてはパターンデータベースヒューリスティック [29, 35, 65, 77]、merge-and-shrink [58, 59] などがある。

7.4.6 ランドマーク (Landmark)

プランニング問題の**ランドマーク** (landmark) とは、全ての解の中で一度でも真になることがある命題である [116]。初期状態とゴール条件に含まれる命題はランドマークである。ランドマークの直感的な理解としては、問題を解くために通過する必要がある中間目標地点である。

ランドマークカウント (landmark count) ヒューリスティックはこれから通過しなければならないランドマークの数をヒューリスティック値とする。ランドマークを全て正しく発見する問題はPSPACE困難なので近似をする必要がある。近似の方法によって非許容的なヒューリスティック (e.g. LAMA) [119] を得る手法と許容的なヒューリスティックを得る手法がある [76]。

ランドマークカット (Landmark cut) はランドマークカウントを一般化したヒューリスティックである [57]。「全ての可能な解で真になる命題」はいくつかは発見できるが、十分な数を発見するのはかなり難しい。なのでランドマークカットはその代わりに「全ての可能な解で少なくとも一つが真になる命題の集合」を使う。このような命題の集合は justification graph と呼ばれるランドマーク発見のためのグラフのカットに相当するため、ランドマークカットと呼ばれる。

7.5 Python 実装

PDDL プランナーの多くはC, C++で実装されているが、Python で書かれたプランナーもある。**pyperplan** (pyperplan) は著名なプランニングの研究者らがGPLv3 で公開している Python で実装された STRIPS のプランナーである。基本的な探索アルゴリズムや、ランドマーク、ランドマークカットなどのヒュー

リスティック関数、また本書では扱っていないが充足可能性問題 (SAT) のソルバーを利用したプランナーも実装されている。pyperplan は `pip` からインストールできる。

```
pip install pyperplan
```

例えばランドマークカットヒューリスティックで A*探索を行うプランナーを実行するには以下のようにする。

```
pyperplan -H=lmcut --domain=domain.pddl --problem=problem.pddl
```

7.6 まとめ

ヒューリスティック探索は PDDL で記述された幅広い問題を自動行動計画問題として解くことができる。ヒューリスティック関数は人間がデザインする必要はなく、PDDL から自動抽出した関数を利用して探索を効率化することができる。

7.7 練習問題

1. グリッド経路探索問題を PDDL で記述し、それを pyperplan に入力して解いてみよう。

2. いくつかのヒューリスティック関数を用いてグリッド経路探索問題を解いてみて、どのヒューリスティック関数が最も効果的かを調べてみよう。例えばランドマークカットと FF ヒューリスティックではどちらが効率的だったか？

3. (難問) ヒューリスティック探索は幅広い自動行動計画問題を解くことができる。では、それぞれ個別の問題を解くためのアルゴリズムを考える必要はあるのか？例えば巡回セールスパーソン問題はヒューリスティッ

ク探索以外だとどのような手法が使われているか？ヒューリスティック探索で解こうとすると、どのような問題があるのか？

7.8 関連文献

自動行動計画を状態空間探索アルゴリズムで解くための手法はStefan Edelkamp and Stefan Schrodl の Heuristic Search Theory and Application [39] にまとめられている。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 [42, 34, 124]。

状態遷移が確率的である問題を**確率的プランニング** (probabilistic planning) と呼ぶ。確率的プランニングはマルコフ決定過程でモデルされることが多い。更に不完全情報である場合は**信念空間プランニング** (belief space planning) 問題である。これらの問題は本文の範囲外とする。詳細は教科書を参照されたい [123]。

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 [56] をオススメする。Fast Downward は PDDL で表現された古典的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。

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 pp. 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

[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] Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial intelligence pp. 81–138 (1995)

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

[14] Blum, A., Furst, M.: Fast Planning Through Planning Graph Analysis. Artificial Intelligence pp. 281–300 (1997)

[15] Bonet, B., Geffner, H.: Planning as Heuristic Search. Artificial Intelligence **129**(1–2), 5–33 (2001)

[16] Breuker, D.M., Uiterwijk, J.W., van den Herik, H.J.: Replacement schemes for transposition tables. ICGA Journal **17**(4), 183–193 (1994)

[17] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games pp. 1–43 (2012)

[18] Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys pp. 293–318 (1992)

[19] 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)

[20] 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)

[21] Bylander, T.: The Computational Complexity of Propositional STRIPS Planning. Artificial Intelligence **69**(1–2), 165–204 (1994)

[22] Černỳ, V.: Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of optimization theory and applications pp. 41–51 (1985)

[23] 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)

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

[25] Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of operations research **4**(3), 233–235 (1979)

[26] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press (2001)

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

[28] Culberson, J.: SOKOBAN is PSPACE-complete. Tech. rep., Department of Computing Science, University of Alberta (1997)

[29] Culberson, J.C., Schaeffer, J.: Pattern databases. Computational Intelligence **14**(3), 318–334 (1998)

[30] Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research **39**, 533–579 (2010)

[31] Dial, R.B.: Algorithm 360: Shortest-path forest with topological ordering [h]. Communications of the ACM **12**(11), 632–633 (1969)

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

[33] Dillenburg, J.F., Nelson, P.C.: Perimeter search. Artificial Intelligence **65**(1), 165–178 (1994)

[34] 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)

[35] Edelkamp, S.: Planning with pattern databases. In: European Conference on Planning (ECP), pp. 13–24 (2001)

[36] Edelkamp, S., Jabbar, S., Schrödl, S.: External a*. KI **4**, 226–240 (2004)

[37] Edelkamp, S., Reffel, F.: Obdds in heuristic search. In: Annual Conference on Artificial Intelligence, pp. 81–92. Springer (1998)

[38] Edelkamp, S., Reffel, F.: Deterministic state space planning with bdds (1999)

[39] Edelkamp, S., Schroedl, S.: Heuristic Search: Theory and Applications. Morgan Kaufmann Publishers Inc. (2010)

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

[41] Erdem, E., Tillier, E.: Genome rearrangement and planning. In: Proceedings of the AAAI Conference on Artificial

[42] Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic SAT-compilation of planning problems. In: Proceedings of the International Joint Conference on

[43] Evett, M., Hendler, J., Mahanti, A., Nau, D.: PRA*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing **25**(2), 133–143 (1995)

[44] 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)

[45] Ford Jr, L.R.: Network flow theory. Tech. rep., Rand Corp Santa Monica Ca (1956)

[46] Fox, M., Long, D.: PDDL2. 1: An extension to PDDL for expressing temporal planning domains. Journal of artificial intelligence research (2003)

[47] 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)

[48] Fukunaga, A., Botea, A., Jinnai, Y., Kishimoto, A.: A survey of parallel a* (2017)

[49] Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 174. freeman San Francisco (1979)

[50] 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)

[51] Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Elsevier (2004)

[52] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc. (1989)

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

[54] Hansen, E.A., Zhou, R.: Anytime heuristic search. Journal of Artificial Intelligence Research **28**, 267–297 (2007)

[55] 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)

[56] Helmert, M.: The Fast Downward planning system. Journal of Artificial Intelligence Research (JAIR) **26**, 191–246 (2006)

[57] Helmert, M., Domshlak, C.: Landmarks, critical paths and abstractions: What's the difference anyway? In: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009), pp. 162–169 (2009)

[58] Helmert, M., Haslum, P., Hoffmann, J.: Flexible abstraction heuristics for optimal sequential planning. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 176–183 (2007)

[59] 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)

[60] 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)

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

[62] Hoffmann, J.: Where'ignoring delete lists' works: Local search topology in planning benchmarks. Journal of Artificial Intelligence Research **24**, 685–758 (2005)

[63] 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)

[64] Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research **14**, 253–302 (2001)

[65] Holte, R., Newton, J., Felner, A., Meshulam, R., Furcy, D.: Multiple Pattern Databases. In: Fourteenth International Conference on Automated Planning and Scheduling ICAPS-04, pp. 122–131. AAAI Press, Whistler, Canada (2004)

[66] 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)

[67] Horie, S., Fukunaga, A.: Block-parallel ida* for gpus. In: Tenth Annual Symposium on Combinatorial Search (2017)

[68] 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)

[69] 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)

[70] 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

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

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

[73] 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 (1997)

[74] 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)

[75] Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of computer computations, pp. 85–103. Springer (1972)

[76] Karpas, E., Domshlak, C.: Cost-optimal planning with landmarks. In: Twenty-First International Joint Conference on Artificial Intelligence (2009)

[77] Katz, M., Domshlak, C.: Structural patterns heuristics via fork decomposition. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 182–189 (2008)

[78] Kautz, H., Selman, B., Hoffmann, J.: Satplan: Planning as satisfiability. In: 5th international planning competition, p. 156 (2006)

[79] Kautz, H.A., Selman, B., et al.: Planning as satisfiability. In: ECAI, vol. 92, pp. 359–363. Citeseer (1992)

[80] Keyder, E.R., Geffner, H.: Trees of shortest paths versus steiner trees: Understanding and improving delete relaxation heuristics. In: Twenty-First International Joint Conference on Artificial Intelligence (2009)

[81] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)

[82] Kishimoto, A., Fukunaga, A., Botea, A.: Evaluation of a simple, scalable, parallel best-first search strategy. Artificial Intelligence **195**, 222–248 (2013)

[83] 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)

[84] Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: European conference on machine learning, pp. 282–293. Springer (2006)

[85] Koenig, S.: Minimax real-time heuristic search. Artificial Intelligence **129**(1-2), 165–197 (2001)

[86] Koenig, S., Likhachev, M.: Real-time adaptive a. In: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, pp. 281–288 (2006)

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

[88] Korf, R.E.: Real-time heuristic search. Artif. Intell. **42**(2-3), 189–211 (1990). DOI 10.1016/0004-3702(90)90054-4. URL `http://dx.doi.org/10.1016/0004-3702(90)90054-4`

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

[90] Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Artificial Intelligence **134**(1), 9–22 (2002)

[91] Korf, R.E., Schultze, P.: Large-scale parallel breadth-first search. In: Proceedings of the AAAI Conference on Artificial

[92] 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)

[93] Kumar, V., Ramesh, K., Rao, V.N.: Parallel best-first search of state-space graphs: A summary of results. In: Proceedings of the AAAI Conference on Artificial

[94] LaValle, S.M.: Planning algorithms. Cambridge university press (2006)

[95] LaValle, S.M., Kuffner, J.J., Donald, B., et al.: Rapidly-exploring random trees: Progress and prospects. Algorithmic and computational robotics: new directions pp. 293–308 (2001)

[96] Lehman, J., Stanley, K.O.: Exploiting open-endedness to solve problems through the search for novelty. In: ALIFE, pp. 329–336 (2008)

[97] Lemons, S., López, C.L., Holte, R.C., Ruml, W.: Beam search: Faster and monotonic. In: Proceedings of the International Conference on Automated Planning and Scheduling (2022)

[98] 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)

[99] Lin, S., Fukunaga, A.: Revisiting immediate duplicate detection in external memory search. In: Proceedings of the AAAI Conference on Artificial Intelligence (2018)

[100] Lipovetzky, N., Geffner, H.: Width and serialization of classical planning problems. In: European Conference on Artificial Intelligence, pp. 540–545 (2012)

[101] Lipovetzky, N., Ramirez, M., Geffner, H.: Classical planning with simulators: Results on the Atari video games. In: Proceedings of the International Joint Conference on

[102] Luger, G.F., Stubblefield, W.A.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Addison-Wesley (1997)

[103] Manzini, G.: Bida*: an improved perimeter search algorithm. Artificial Intelligence **75**(2), 347–360 (1995)

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

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

[106] Mehlhorn, K., Meyer, U.: External-memory breadth-first search with sublinear i/o. Algorithms—ESA 2002 pp. 21–26 (2002)

[107] Minato, S.i.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: Proceedings of the 30th international Design Automation Conference, pp. 272–277 (1993)

[108] Mirkis, V., Domshlak, C.: Cost-sharing approximations for h+. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 240–247 (2007)

[109] Miura, T., Ishida, T.: Stochastic node caching for memory-bounded search. In: Proceedings of the AAAI Conference on Artificial

[110] Nash, A., Daniel, K., Koenig, S., Felner, A.: Theta: Any-angle path planning on grids. In: Proceedings of the AAAI Conference on Artificial

[111] Nash, A., Koenig, S.: Any-angle path planning. AI Magazine **34**(4), 85–107 (2013)

[112] 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)

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

[114] Pearson, W.R.: Rapid and sensitive sequence comparison with FASTP and FASTA. Methods in enzymology **183**, 63–98 (1990)

[115] Pohl, I.: Bi-directional search. Machine intelligence **6**, 127–140 (1971)

[116] Porteous, J., Sebastia, L., Hoffmann, J.: On the extraction, ordering, and usage of landmarks in planning. In: Sixth European Conference on Planning (2001)

[117] Puterman, M.L.: Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons (2014)

[118] Reinefeld, A., Marsland, T.A.: Enhanced iterative-deepening search. IEEE Transactions on Pattern Analysis and Machine Intelligence **16**(7), 701–710 (1994)

[119] Richter, S., Helmert, M., Westphal, M.: Landmarks revisited. In: Proceedings of the AAAI Conference on Artificial

[120] 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)

[121] Rintanen, J.: Planning as satisfiability: Heuristics. Artificial intelligence **193**, 45–86 (2012)

[122] Romein, J.W., Plaat, A., Bal, H.E., Schaeffer, J.: Transposition table driven work scheduling in distributed search. In: Proceedings of the AAAI Conference on Artificial

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

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

[125] 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)

[126] 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)

[127] Sint, L., de Champeaux, D.: An improved bidirectional heuristic search algorithm. Journal of the ACM (JACM) **24**(2), 177–191 (1977)

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

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

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

[131] Sulewski, D., Edelkamp, S., Kissmann, P.: Exploiting the computational power of the graphics card: Optimal state space planning on the GPU. In: Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011 (2011). URL http://aaai.org/ocs/index.php/ICAPS/ICAPS11/paper/view/2699

[132] Sutton, R., Precup, D., Singh, S.: Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. Artificial Intelligence **112**, 181–211 (1999)

[133] Sutton, R.S., Barto, A.G., et al.: Introduction to reinforcement learning, vol. 2. MIT press Cambridge (1998)

[134] Vallati, M., Chrpa, 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)

[135] Van Laarhoven, P.J., Aarts, E.H.: Simulated annealing. In: Simulated annealing: Theory and applications, pp. 7–15. Springer (1987)

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

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

[138] Wolpert, D.H., Macready, W.G., et al.: No free lunch theorems for optimization. IEEE transactions on evolutionary computation **1**(1), 67–82 (1997)

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

[140] Zhou, R., Hansen, E.A.: Structured duplicate detection in external-memory graph search. In: Proceedings of the AAAI Conference on Artificial

[141] Zhou, Y., Zeng, J.: Massively parallel A* search on a GPU. In: Proceedings of the AAAI Conference on Artificial URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9620`

[142] 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

索引