ヒューリスティック探索

陣内　佑
理化学研究所 革新知能統合研究センター *yuu.jinnai@riken.jp*

March 27, 2017

List of Todos

Contents

Chapter 1

イントロダクション

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

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

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

何故、我々は殆ど頭を (自覚的に) 使わずにこのような計画を立てることが出来るのだろうか？何故ごはんをよそうためにお湯を沸かしたり、最寄り駅まで歩いたりする必要はないと分かるのだろうか？

それは我々が**直感** (ヒューリスティック) によって正しそうな行動を絞り込むことが出来るからである。

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

我々の **Grand Scheme** は、人間のように高度な先読みをし、高度な行動計画を行い実行の出来る人工知能を作ることにある！

1.1 何故人工知能に探索が必要なのか

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

人工知能とは何か、と考えることは本書の主眼ではない。人工知能の教科書として有名な Artificial Intelligence: Modern Approach [33] では人工知能と呼ばれる研究は主に以下の４つの目標を目指していると説明している。

1. Think Rationally
2. Think Humanly
3. Act Rationally
4. Act Humanly

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

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

探索は特に学習データを得ることが難しいエージェントに用いられてきた。例えば火星探査車などは宇宙のデータなどとても得られるものではないので、探索に基づくエージェントが用いられている。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だろう。

もちろん両手法は組み合わせて用いることでより賢い行動が出来るようになる。AlphaGo などはまさに探索と機械学習を組み合わせたエージェントの強力さを体現しているといえるだろう。

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

1.2 本書の射程

世界を正確に表現することは不可能である。よって、殆どの問題はより解きやすい問題に**モデル化**され、モデル化された問題を解くことによって解きたい問題を

[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

解決するというのが情報科学である。

どのように世界をモデルするかは非常に難しい問題である。モデルを簡単なものにすればするほど解きやすくなるが、簡単で正しいモデルをデザインする・自動生成することは非常に難しい。

本書が主に扱うモデルは**完全情報** (perfect information) かつ**決定論的** (deterministic) モデルである (2 7 章)。

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

決定論的とはエージェントの行動によって世界の状態がどのように遷移するかが一意に (決定論的) に定まることである。非決定論的モデルでは遷移が一意に定まらない。同じ状態から同じアクションを取ったとしても、世界がどのように変化するかは一意に定まらない。非決定論的モデルにおける探索問題は**??**章で扱う。

本書が扱う完全情報決定論的モデルは最もシンプルなモデルである。これを不完全情報、非決定論的モデルとすることでより元の問題も表現しやすくなることがあるが、一方でモデルのシンプルさを失うことになる。

Chapter 2

状態空間問題 (State-Space Problem)

この章ではまず、2.1 節ではグラフ探索手法が用いられる問題として状態空間問題を定義する。次に 2.3 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 [33]、[31]、[9] 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 7 章までで扱う**状態空間問題** (state-space problem) である。状態空間問題 $P = (S, A, s, T)$ は状態の集合 S、初期状態 $s \in S$、ゴール集合 $T \in S$、アクション集合 $A = a_1, ..., a_n$、$a_i : S \to S$ がある。アクションはある状態を次の状態に遷移させる関数である。状態空間問題の解は初期状態からゴール状態へ遷移させるアクションの列を求めることである。

よって、状態空間問題はグラフにモデルすることで考えやすくなる。状態空間問題を表す**状態空間グラフ** (state-space graph) は以下のように定義される。

Definition 1 (状態空間グラフ、State-space graph)**.** 問題グラフ $G = (V, E, s, T)$ は

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

状態空間問題の**解** (solution) は以下の定義である。

Definition 2 (解、Solution)**.** 解 $\pi = (a_1, a_2..., a_k)$ はアクション $a_i \in A$ の *(*順序付*)* 配列であり、初期状態 s からゴール状態 $t \in T$ へ遷移させる。すなわち、$u_i \in S$,$i \in \{0, 1, ..., k\}$, $u_0 = s, u_k = t$ が存在し、$u_i = a_i(u_{i-1})$ となる。

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

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

本書ではコスト付き状態空間問題をメインの問題として考える。

コストの定義されていない状態空間問題を特に区別して**ユニットコスト** (unit-cost) 問題 (ドメイン) と呼ぶ。コスト付き状態空間問題は重み付き (コスト付き) グラフとしてモデルすることが出来る。すなわち、$G = (V, E, s, T, w)$ は状態空間グラフの定義に加え、エッジの重み $w : E \leftarrow \mathbb{R}$ を持つ。

3 章で詳解するが、探索アルゴリズムは状態空間グラフのノード・エッジ全てを保持する必要はない。全てのノード・エッジを保持した状態空間グラフを特に**明示的状態空間グラフ** (explicit state-space graph) と呼ぶとする。このようなグラフは、例えば隣接行列を用いて表すことが出来る。隣接行列 M は行と列の大きさが $|V|$ である正方行列であり、エッジ (v_i, v_j) が存在するならば $M_{i,j} = 1$、なければ $M_{i,j} = 0$ とする行列である。このような表現方法の問題点は行列の大きさが $|V|^2$ であるため、大きな状態空間を保持することが出来ないことである。例えば、2.3 節で紹介する 15-puzzle は状態の数が $|V| = 15!/2$ であるため、隣接行列を保持することは現在のコンピュータでは非常に困難である。

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

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

Table 2.1: 状態空間問題における問題の定式化。もっとも情報が多く与えられる定式化がドメイン依存エージェントであり、ヒューリスティック関数なども与えられる。ドメイン非依存エージェントは PDDL などのモデル言語で書かれた入力が与えられ、その情報から用いるヒューリスティック関数や効率化手法を自動的に選択する必要がある。ブラックボックスエージェントは事前に何も与えられない、最も挑戦的な定式化である。

問題	状態遷移関数	ヒューリスティック関数	効率化
ドメイン依存	Fully Available	hard-code	hard-code
ドメイン非依存 (PDDL)	Fully Available	自動生成する必要がある	ドメイン非依存の最適化・ポートフォリオ戦略
ブラックボックス	Unavailable (simulator)	Unavailable	非許容的なノード・エッジの枝刈り

探索の開始時、エージェントは初期ノードのみを保持する。エージェントは保持しているノードに対して Expand を適用することによって、新しいノードとエッジをグラフに加える。これを求める解を見つけるまで繰り返す。Expand はある状態からの次の状態の集合を返す関数である。Expand 関数は明示的に与えられるのではなく、ルールによって与えられることが多い。例えば将棋であれば、将棋のルールによって定められる合法手によって得られる次の状態の集合が Expand 関数によって得られる。これによって、エージェントは解を見つけるまでのノード・エッジだけ保持して必要な解を見つけることが出来る。

2.2 状態空間問題の定式化

2.3 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。グリッド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。

2.3.1 グリッド経路探索 (Grid Path-finding)

グリッド経路探索問題 (grid path-finding problem) は k(多くの場合 $k = 2$) 次元のグリッド上で初期配置からゴール位置までの経路を求める問題である [38]。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 [30]。

Web 上に簡単に試せるデモがあるので、参照されたい[1]。とてもよくできてお

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

Figure 2.1: グリッド経路探索問題

り、この文章で説明する様々なグラフ探索手法をグリッド経路探索に試すことが出来る。

グリッド経路探索はロボットのモーションプランニングやゲーム AI などで応用される [2]。ストラテジーゲームなどでユニット（エージェント）を動かすために使われる。よく使われるベンチマーク問題集にも Starcraft のゲームのマップが含まれている [37]. またグリッドは様々な問題を経路探索に帰着して解くことができるという意味でも重要である。例えば多重整列問題 (Multiple Sequence Alignment) はグリッド経路探索に帰着して解くことが出来る (後述)。ロボットのモーションプランニングも経路探索に帰着することが出来る。すなわち、k 個の関節の角度を変えて、現在状態からゴール状態へ遷移させたい。各関節の角度がグリッドの各次元に相当する。ロボットの物理的な構造により、関節のある角度の組み合わせは不可能である。不可能な組み合わせが、障害物の置かれたグリッドに相当する。よって、障害物を避けた経路というのが関節の動かし方ということになる。

2.3.2 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 2.2)。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

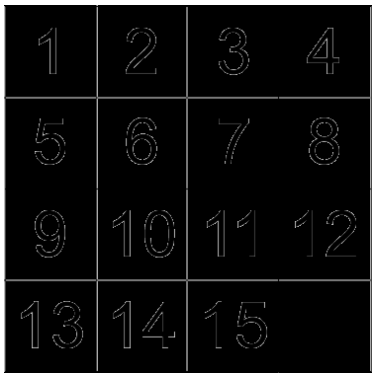


Figure 2.2: 15 パズルのゴール状態の例

位置があり、四方に隣り合うタイルのいずれかをその位置に移動する (スライドする) ことが出来る。スライディングタイル問題は、与えられた初期状態からスライドを繰り返し、ゴール状態にたどり着く経路を求める問題である。

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

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

2.3.3 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 [10]。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

[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 [21]。

返し、同一の配列に並べ替える手続きをアライメントと呼ぶ。２つの配列の編集距離は編集操作の合計コストの最小値である。３つ以上の配列における距離の定義は様々考えられるが、ここでは全ての配列のペアの編集距離の総和を用いる。

MSA における可能な編集操作は置換と挿入である。置換は配列のある要素 (DNA かアミノ酸) を別の要素に入れ替える操作であり、挿入は配列のある位置に要素を挿入する操作である。例えば (ABC, BCB, CB) の３つの配列のアライメントを考える。図 2.3b は置換と編集に対するコストの例である。-は欠損、すなわち挿入操作が行われたことを示す。アミノ酸配列における有名なコスト表として PAM250[32] があるが、ここでは簡単のため仮のコスト表を用いる。図 2.3a はこのコスト表を用いたアライメントの例である。このとき、例えば配列 ABC-と-BCB の編集距離は (A,-)、 (B,B)、 (C,C)、 (-,B) のコストの総和であるので、図 2.3b を参照し、$5+0+1+5=11$ である。(-BCB, –CB) の距離は 6, (–CB, ABC-) の距離は 16 であるので、３配列の編集距離は $11+6+16=33$ である。

n 配列の MSA は n 次元のグリッドの経路探索問題に帰着することが出来る [25]。図 2.3c は (ABC) と (BCB) の２つの配列による問題を表す。状態 s は２つの変数によって表現される:(x_0, x_1)。x_0 は配列 0 のどの位置までアライメントを完了したかを表す変数であり、配列 i の長さを l_i とすると定義域は $0 \leq x_0 \leq l_0$ である。全てのアライメントが完了した状態 $s=(l_0, l_1)$ がゴール状態である。可能なアクションは $a=(b_0, b_1), (b_i=0,1)$ の形を取り、これは配列 i に対して欠損を挿入する場合に $b_i=0$ となる。状態 s に対してアクション a を適用した後の状態 s' は $s'=(x_0+b_0, x_1+b_1)$ となる。例えば図 2.3c は初期状態 $s=(0,0)$ に対して $a=(1,0)$ を適用している。これは (A), (-) までアライメントを進めた状態に対応する。次に $a=(1,1)$ が適用され、アライメントは (A,B), (-,B) という状態に至る。

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

MSA は生物学研究に役立つというモチベーションから非常に熱心に研究されており、様々な定式化による解法が知られている [10]。

2.3.4 倉庫番 (Sokoban)

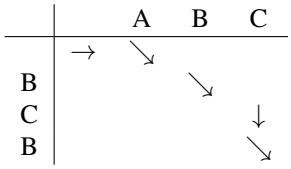
倉庫番 (Sokoban) は日本発のパズルゲームであり、倉庫の荷物を押していくことで指定された位置に置くというゲームである。現在でも様々なゲームでゲーム内ミニゲームとして親しまれている。プレイヤーは「荷物の後ろに回って押す」ことしか出来ず、引っ張ったり、横から動かしたりすることが出来ない。また、荷物の上を通ることも出来ない。PSPACE-complete であることが知られている [7]。

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

(a) MSA の解の例

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

(b) 操作コスト表の例



(c) グリッド経路探索への帰着

A	B	C	-
-	B	C	B

(d) グリッド経路探索への帰着

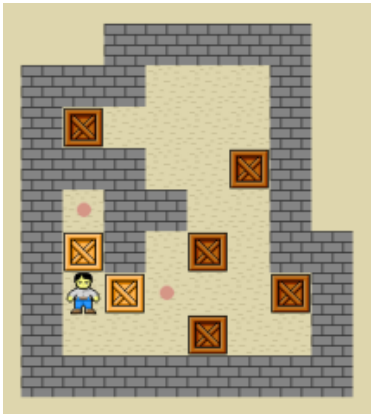


Figure 2.3: Sokoban: 画像は wikipedia より。

状態の表現方法は２通りあり、一つはグリッドの各位置に何が置いてあるかを変数とする方法である。もうひとつはプレイヤー、各荷物の位置に対してそれぞれ変数を割り当てる方法である。可能なアクションは $\{move-up, move-left, move-down, move-right, push-up, push-left, push-down, push-right\}$ の８通りである。$move-*$ はプレイヤーが動くアクションに対応し、コストは 0 である。$push-*$ は荷物を押すアクションであり、正のアクションコストが割当てられている。よって、倉庫番はなるべく荷物を押す回数を少なくして荷物を目的の位置に動かすことが目的となる。

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

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

もうひとつ重要な問題はゼロコストアクションの存在である。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。

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

セールスパーソンはいくつかの都市に回って営業を行わなければならない。都市間の距離 (=コスト) は事前に与えられている。TSP は全ての都市を最短距離で回ってはじめの都市に戻る経路を求める、という問題である [3]。

n 個の都市があるとすると (最適・非最適含む) 解の数は $(n-1)!/2$ 個である。可能なアクションは「都市 $i \in \{1..n\}$ を訪れる」であり、一度訪れた都市には行けない。TSP のゴール条件はすべての都市を訪れることである。よって、n 回どれかアクションを実行すれば、とりあえず解を得ることが出来る。一方、最適解を得る問題は NP 完全であることが知られている。

TSP の解の下界としては最小全域木 (minimum spanning tree) のコストがよく用いられる [14]。グラフの全域木 (spanning tree) は全てのノードを含むループを

Figure 2.4: 巡回セールスパーソン問題: 画像は wikipedia より。

含まない部分グラフである。最小全域木は全域木のうち最もエッジコストの総和が小さいものである。未訪問の都市によるグラフの最小全域木は TSP の下界となることが知られている。

TSP はヒューリスティック探索に限らず、様々なアプローチで研究されている問題ドメインである [3]。TSP について特に詳しく知りたい方はそちらの教科書を参照されたい。

2.4 問題の難しさ

状態数あるいは状態空間の大きさ**分枝度** (branching factor) Trap vs. reversible

Chapter 3

情報なし探索 (Blind Search)

1 章では様々な状態空間問題を紹介したが、それぞれの問題の解法はどれも沢山研究されている。一つの指針としては、ある問題に特化した解法を研究することでその問題をより高速に解くというモチベーションがある。これは例えば MSA のように重要なアプリケーションがある問題の場合に特に熱心に研究されることが多い。一方、なるべく広い範囲の問題に対して適用可能な手法を研究するというモチベーションもある。特に人工知能の文脈において、なるべく問題の知識を必要とせず、最小限の仮定のみを必要とする解法が求められる。

1 章で紹介した状態空間問題を広く扱うことの出来る手法としてグラフ探索アルゴリズムがある。本章では最もシンプルな問題（ドメイン）の知識を利用しない探索を紹介する。情報なし探索 (Blind Search) は状態空間グラフのみに注目し、背景にある問題に関する知識を一切使わないアルゴリズムである。このような探索を設計に重要なことは **1. 重複検知を行うか 2. ノードの展開順序**の二点である。重複検出は訪問済みのノードを保存しておくことで同じノード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 [35]。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 [28]。

Table 3.1: 木探索とグラフ探索

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

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

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。

Algorithm 1: Implicit Tree Search

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

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

以下、(k) と書いて疑似コードの k 行目を指すことにする。木探索はオープンリスト[1]と呼ばれるノードの集合を Priority queue に保持する。探索の開始時には、初期状態のみがオープンリストに入っている (1)。木探索は、このオープンリストから一つノード u を選び (3)、ゴール条件を満たしているかを確認する (4)。満た

[1]歴史的な経緯でリストと呼ばれているが、データ構造がリストで実装されるという意味ではない。効率的なデータ構造は 6 章で紹介する。

Algorithm 2: Expand

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

for $a \in A(s)$ **do**
  $s' \leftarrow a(s)$;
  $d(s') \leftarrow d(s) + 1$;
  $d(s') \leftarrow d(s) + w(s, s')$;
  $S = S \cup s'$;
return S;

していれば初期状態から u への経路を返す。満たしていなければ、そのノードを展開する (6-)。展開とは、そのノードの子ノードを列挙し、オープンリストに入れる (8) ことを指す。

アルゴリズム 2 は展開関数の動作を表している。初期状態からノード n への既知の最小経路コストを g 値と呼ぶ。状態を更新すると同時に g 値を更新する。これによって解を発見した時に解ノードの g 値が解のコストとなる。なお、状態 s に対して適用可能なアクションの集合 $A(s)$ は与えられていると仮定する。

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

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

非明示的グラフ木探索の強みは、生成済みノードのうち展開済みではないもののみを $Open$ に保持すればよいことにある。未生成ノード、展開済みノードはメモリ上に保持する必要がない。一方、これの問題は、一度展開したノードが再び現れた場合、**再展開 (reexpansion)** をすることになる。よって、グラフがより木から遠いほど (複数の経路で到達可能なノードがあるほど) 同じノードを何度も再展開することになり、効率が悪くなってしまう。もっと言えば、木探索アルゴリズムは状態数が有限であっても停止しない場合がある。これらが問題になるような問題ドメインである場合は後述する重複検出を使うグラフ探索 3.4 を使うと良いだろう。

紛らわしいが、木探索アルゴリズムはグラフを探索するアルゴリズムである。グラフ探索アルゴリズムのうち、後述する重複検出を行わない手法を木探索アルゴリズムと呼ぶ。

3.2 幅優先探索 (Breadth-First Search)

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

幅優先探索の手順は非常に単純であり、FIFO の順に $Open$ から取り出せばいいだけである。これをもう少し大きな視点で、どのようなノードを優先して探索しているのかを考えてみたい。初期状態から現在状態にたどり着くまでの経路の長さをノードの d 値と定義する。すると、幅優先探索の $Open.pop()$ はアルゴリズム 3 のように書くことが出来る。ユニットコスト問題である場合、d 値は g 値と一致する。

幅優先探索のメリットは初めに発見した解が最短経路長であることである。問題がユニットコストドメインであれば、最短経路が最小コスト経路であるので、最適解が得られる。

Algorithm 3: Breadth-First Search: $Open.pop()$

Output : Node n

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

3.3 深さ優先探索 (Depth-First Search)

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

深さ優先探索は解がある一定の深さにあることが既知である場合に有効である。例えば TSP は全ての街を回ったときのみが解であるので、街の数が n であれば全ての解の経路長が n である。このような問題を幅優先探索で解こうとすると、解は最も深いところにしかないので、最後の最後まで解が一つも得られないということになる。一方、深さ優先探索なら n 回目の展開で一つ目の解を見つけることが出来る。

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。詳しくは**??**章で解説する。

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

Output : Node n

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

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

明示的グラフのあるノードが初期状態から複数の経路でたどり着ける場合、同じ状態を表すノードが木探索による非明示的グラフに複数現れるということが生じる。このようなノードを**重複** (duplicate) と呼ぶ。ノードの重複は計算資源を消費してしまうので、効率的な**重複検出** (duplicate detection) の方法は重要な研究分野である。

本書ではノードの重複検出を行う探索アルゴリズムを狭義にグラフ探索アルゴリズムと呼び、重複検出を行わない探索を木探索と区別する。

Algorithm 5: Implicit Graph Search

Input : Implicit problem graph with initial node s, weight function w, successor generation function $Expand$, goal function $Goal$

Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path exists

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

Algorithm 6: $Improve(u, v)$

Input : Node u and its successor v
Side effect: Update parent of v, $Open$, and $Closed$

```
1 if v ∉ Closed ∪ Open then
2 |   Open.insert(v);
3 |   parent(v) ← u;
```

3.5 ダイクストラ法 (Dijkstra Algorithm)

ダイクストラ法 (Dijkstra's Algorithm) は非負コストグラフにおいて最短経路を返すアルゴリズムである [8]。ダイクストラ法は g 値が最も小さいノードを優先して展開するグラフ探索アルゴリズムである。

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

Output : Node u

```
1 return arg max_n g(n)
```

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。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 [29]。

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 [**?**]。実用するにあたっては、後述するヒューリスティック関数が得られない (難しい) 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。

Chapter 4

ヒューリスティック探索 (Heuristic Search)

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

4.1 ヒューリスティックとは？

経路探索問題を幅優先探索で解くことを考えよう。図**??**の初期状態からゴールへの最短経路の長さは **X** である。このとき、幅優先探索は図**??**の領域を探索する。しかし人間が経路探索を行うときにこんなに広い領域を探索しないだろう。なぜか。それは人間が問題の特徴を利用して、このノードを探索したほうがよいだろう、という推論を働かせているからである。問題の特徴を利用してノードの**有望さ**をヒューリスティック関数として定量化し、ヒューリスティック関数を利用した探索アルゴリズムをヒューリスティック探索と呼ぶ。ヒューリスティック関数は人間が自分の知識を利用してコーディングする場合もあるが、特にプランニング問題などでは自動的にヒューリスティックを生成する手法も広く使われている。

4.2 ヒューリスティック関数

ヒューリスティック関数はある状態からゴールまでの最短距離の見積もりである。

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

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

ヒューリスティック関数の望ましい性質として、まず正確である方が望ましい。すなわち、h 値が実際のゴールまでの最短距離に近いほど、有用な情報であると言える。もう一つ望ましい性質は h 値が最適解コストの下界である場合である。4.3 章で解説するが、h 値が最短距離の下界である場合、それを用いた効率的な探索アルゴリズム (A*探索、重み付き A*探索) において解コストに理論的保証が得られることが広く知られている。h 値が常に最適解コストの下界であるヒューリスティック関数を許容的なヒューリスティックと呼ぶ。

Definition 6 (許容的なヒューリスティック)**.** ヒューリスティック関数 h は最適解のコストの下界である場合、許容的である。すなわち、全てのノード $u \in V$ に対して $h(u) \leq h^*(u)$ が成り立つ。

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

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

もう一つ重要な性質は無矛盾性である。

Definition 7 (無矛盾なヒューリスティック)**.** ヒューリスティック関数 h は全てのエッジ $e = (u, v) \in E$ に対して $h(u) \leq h(v) + w(u, v)$ が成り立つ場合、無矛盾である。

無矛盾性は特に 4.3 章で後述する A*探索において探索の効率性に重要な性質である。

また、無矛盾なヒューリスティックのうちゴールノードの h 値が 0 となるヒューリスティックは許容的である。

Theorem 1. 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。

[1] 解が多いほど簡単であるとは一概には言えないが

Proof. あるノード n_0 からゴールノード $n_k \in T$ への最短経路 (ノードの列) を $(n_0, n_1, ...)$ と置く。無矛盾なヒューリスティック $h(n)$ は

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

よって $h(n_0) \leq h^*(n_0)$ より許容的である。

□

4.3 A*探索

ダイクストラ法は初期状態からそのノードまでのコストである g 値が最小のノードを展開していく。これは間違った方針ではないだろうが、理想的にはゴール状態に向かっていくノードを展開していきたい。図**??**はダイクストラ法による状態空間の探索を図示したものである。ダイクストラ法はゴールがどこにあるかということを無視して探索を進めているため、図のように探索空間が広がっていく。

A*探索 (A* search) はゴールまでの距離を見積もる**ヒューリスティック関数** (heuristic function) を用いることで図**??**のようにゴールに向かって探索していくことを目指した手法である。

A*探索はヒューリスティック探索の代名詞である、最も広く知られている手法である [12]。A*探索は以下の f 値が最小となるノードを優先したグラフ探索アルゴリズムである。

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

ノード n の f 値は、初期状態から n を通過してゴール状態に辿り着くためのコストの見積もりである。g 値は初期状態からノード n までの既知の最短経路コストである。一方 h 値はヒューリスティック関数による n からゴール状態までの最短経路の見積もりである。A*探索は非明示的グラフ探索アルゴリズム (アルゴリズム 5) の一つであり、*Open.pop*() を f 値最小ノードを返すようにしただけである 8。

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

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

Output : Node n

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

図**??**はマンハッタン距離ヒューリスティック (Manhattan distance heuristic) による A*探索である。

A*に用いるヒューリスティック関数に求められる要件とは何であるか。一つは正確であるほど良い。もう一つはヒューリスティック関数がゴールまでのコストの下界であると実用的・理論的に良い性質があることが知られている。

Theorem 2. ヒューリスティックが許容的である時、*A**は最適解を返す。

Proof. 許容的なヒューリスティック h(n) は n からゴールへの経路の下界である。よって、ゴール状態の h 値は 0 である。つまりゴール状態の f 値は g 値と同じである。この解の g(n') 値を f*と置こう (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)¡=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。

□

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

Theorem 3. 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。

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

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

Theorem 4. 許容的なヒューリスティックを用いた重み付き *A**探索は最適解のコスト f^* に対して、発見される解のコストが wf^* 以下であることを保証する。

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

Output : Node u

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

wA*の利点はそこそこの計算時間で解のクオリティに保証がついた解を見つけることが出来ることにある。wA*の解は最適解のコストの上界になるので、A*探索の枝刈りに用いることが出来る。A*探索を実行する前に wA*を走らせ、解の上界を得、A*探索実行時にその値を超す f 値を持ったノードを

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

解のクオリティに保証がない。

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

Output : Node u

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

4.5 ドメイン固有のヒューリスティック関数

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

4.5.1 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

[2]解が多いほど簡単であるとは一概には言えないが

離を h 値とする。マンハッタン距離の意味としては、障害物を無視した最短経路の距離であるので、グラフのエッジを増やした緩和問題である。このように、問題の性質を理解していれば許容的なヒューリスティック関数を設計することが出来る。８方向グリッドにおいても斜め方向を加えた距離を考えることで許容的なヒューリスティックとすることが出来る。Any angle グリッドならば直線距離が許容的なヒューリスティックである。

4.5.2 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。

4.5.3 巡回セールスパーソン問題：最小全域木

TSP の解の下界としては最小全域木 (minimum spanning tree) のコストがよく用いられる。グラフの全域木 (spanning tree) は全てのノードを含むループを含まない部分グラフである。最小全域木は全域木のうち最もエッジコストの総和が小さいものである。未訪問の都市によるグラフの最小全域木は TSP の下界となることが知られている。

Chapter 5

自動行動計画問題 (Automated Planning Problem)

この章では自動行動計画問題について説明する。古典的プランニング問題はエージェントの自動行動計画を行うための問題の一つであり、状態空間問題の一つである [12]。

ロジスティック [15, 36]、セルアセンブリ [4]、遺伝子距離計算 [11]、ビデオゲーム [27] など、様々な応用問題を含むフレームワークである。

環境が決定的であり、完全情報を仮定する。これらの仮定を緩和した問題（確率的モデルや不完全情報モデル）もグラフ探索によって解かれることが多いが、本文の範囲外とする。詳細は AI の教科書を参照されたい [33]。

なお、プランニング問題は A*などの状態空間探索アルゴリズム以外にも、SAT や CSP などの制約充足問題に変換して解く方法もあるがこれも本書の範囲外とする [22]。

5.1 定義

古典的プランニングは述語論理によって世界が記述される [12]。Proposition AP は世界の状態において何が真・偽であるかを記述する。世界の状態はエージェントがアクションを行うことによって遷移し、遷移後の状態は遷移前の状態と異なる proposition が真・偽でありうる。古典的プランニングの目的は与えられた初期状態からゴール条件を満たすまでのアクションの列を求めることにある。以下、定義は [9] に従う。

Definition 8 (古典的プランニング問題、Classical Planning Problem)**.** 古典的プランニング問題は有限状態空間問題 $P = (S, A, s_0, T)$ の一つである。$S \subseteq 2^{AP}$ は状態の集合であり、$s_0 \in S$ は初期状態、$T \subseteq S$ はゴール状態の集合、$A : S \to S$ は可能なアクションの集合である。

古典的プランニング問題の最も基本となる STRIPS モデル [12] の場合、ゴールは proposition のリストで表せられる $Goal \subseteq AP$。ゴール状態の集合 T は $p \in Goal$ となるすべての p が真である状態の集合である。アクション $a \in A$ は条件 $pre(a)$、効果 $(add(a), del(a))$ で表せられる。条件 $pre(a) \subseteq AP$ はアクション a を実行するために状態が満たすべき proposition の集合である。効果 $add(a)$ はアクション a を適用後に真になる proposition の集合であり、$del(a)$ は偽になる集合である。従って、アクション a を状態 s に適用後の状態 $s' = suc(s, a)$ は

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

である。このようにして、古典的プランニング問題は後述のグラフ探索問題に帰着することが出来る。

As such, a classical planning problem can be solved by an A* search $(G(V', E', w'), s'_0, T')$; $V' = S$, $e(v_i, v_j) \in E'$ exists if there exists a such that $v_j = succ(v_i, a)$, $s'_0 = s_0$, $T' = T$.

5.2 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) [1] はプランニング問題を記述されるために用いられる言語の一つである。PDDL は domain ファイルと instance ファイルの 2 つのファイルによって一つの入力となる。domain ファイルは predicate とアクションが定義され、instance ファイルは初期状態、ゴール状態とオブジェクトが定義される。図 5.1 は blocks-world の domain ファイルである。図 5.2 は blocks-world の instance ファイルである。

5.3 古典的プランニング問題の例

ソリティアエレベーター空港最適化グリッド経路探索: 2 章で紹介した問題が PDDL で表せられることを示すためグリッパーサテライト

5.3.1 ブロックスワールド (blocks world)

説明用候補

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 4 Op-blocks world
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
               (ontable ?x)
               (clear ?x)
               (handempty)
               (holding ?x)
               )

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

  (:action put-down
             :parameters (?x)
             :precondition (holding ?x)
             :effect
             (and (not (holding ?x))
                  (clear ?x)
                  (handempty)
                  (ontable ?x)))
  (:action stack
             :parameters (?x ?y)
             :precondition (and (holding ?x) (clear ?y))
             :effect
             (and (not (holding ?x))
                  (not (clear ?y))
                  (clear ?x)
                  (handempty)
                  (on ?x ?y)))
  (:action unstack
             :parameters (?x ?y)
             :precondition (and (on ?x ?y) (clear ?x) (handempty))
             :effect
             (and (holding ?x)
                  (clear ?y)
                  (not (clear ?x))
                  (not (handempty))
                  (not (on ?x ?y)))))
```


Figure 5.1: blocks-world の domain ファイル

```
(define (problem BLOCKS-4-0)
(:domain BLOCKS)
(:objects D B A C )
(:INIT (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D) (ONTABLE C) (ONTABLE A)
 (ONTABLE B) (ONTABLE D) (HANDEMPTY))
(:goal (AND (ON D C) (ON C B) (ON B A)))
)
```

Figure 5.2: blocks-world の instance ファイル

5.3.2 ロジスティック (logistics)

説明用候補

5.3.3 プリンタースケジュリング (printer scheduling)

説明用候補

5.3.4 セルアセンブリ (cell assembly)

コラム候補

5.3.5 クエリ最適化 (query optimization)

コラム候補

5.3.6 宇宙探査車 (rovers)

コラム候補

5.3.7 サイバーセキュリティ (cyber security)

コラム候補

5.3.8 ゲノムリアレンジメント (genome rearrangement)

コラム候補

5.3.9 グリッドコンピューティング

コラム候補

5.4 ヒューリスティック関数の自動生成

ドメイン非依存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。

5.4.1 ゴールカウントヒューリスティック

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。スライディングタイルであれば 1 つのアクションで動かせるタイルの数は 1 つなので、許容的である。

Chapter 6

実験評価

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。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。オープンリストは Priority queue であり、必要な操作は pop と push である。クローズドリストは insert と find である。

6.1 オープンリスト

Priority queue の実装方法は様々ある。まず、f 値の定義域が実数か、整数かは重要である。実数である場合は二分木のような一般的な Priority queue を使うことが多い。整数である場合は *bucket* 実装をすることが出来る。

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

6.1.1 データ構造

6.1.2 タイブレーキング

bucket open list h tiebreaking FIFO, LIFO tiebreaking

Table 6.1: オープンリストのデータ構造の比較

実装	pop 計算量	$push$ 計算量
二分木	$O(log(n))$	XX
bucket	XXX	XXX

6.2 クローズドリスト

クローズドリストはハッシュテーブル incremental hashing perfect hashing universal hashing

6.2.1 Delayed Duplicate Detection

重複検出のタイミングは２通りある。

Chapter 7

ヒューリスティック探索の派生

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

残念ながら、ほぼ正しいコストを返すヒューリスティック関数を使っても、A*が展開するノードの数は指数的に増加することが知られている [16]。

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。

7.1 Branch-and-Bound

admissible pruning Branch and Bound wA* pruning (MSA)

Table 7.1: 派生アルゴリズムの比較。

	空間計算量	時間計算量	有効な場面
反復深化 A*	$O(n)$	展開ノード数が大きくなる。キャッシュ効率が良くなる場合がある	キャッシュ効率が良いと A*よりも高速な場合がある
両方向探索	A*よりも小さい	重複検出の効率による	ヒューリスティック関数の性能が低い場合に有効
External Search	外部記憶を用いる	I/O を必要とするため遅い	探索空間がメモリに乗り切らない場合
Symbolic Search	BDD によって効率的になる	複数のノードを同時に展開できる	**TODO**
並列探索	使用する計算ノードのメモリの総計が使える	線形スピードアップならコアの数だけ高速化	問題が難しい

7.2 反復深化 A* (Iterative Deepening A*)

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。

3.4 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 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 [24]。アルゴリズム 11 は反復深化 A*の概要を示している。アイディアとしては、閾値 $cost$ を 1 ずつ大きくしながら、繰り返しコスト制限付き深さ優先木探索 (CLDFS) を実行する。コスト制限付き深さ優先探索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コスト制限付き深さ優先探索を実行する。

反復深化 A*は閾値を大きくする度に一つ前のイテレーションで展開・生成したノードをすべて展開・生成しなおさなければならない。各イテレーション内でもクローズドリストを保持していないために重複検出が出来ない。なので、アルゴリズム全体を通して大量の重複ノードが出る可能性がある。これは非常に効率が悪いように思えるかもしれないが、様々な状況において A*よりも有用な手法である。

反復深化 A*のメリットはいくつかある。まず、コスト w が 0 となるアクションが存在しない場合、必要なメモリ量が最適解のコストに対して線形である。深さ優先木探索は可能な最長経路だけのノードを保持する必要がある。木探索はクローズドリストは保持しない。コスト制限付きの場合、最長経路は $cost$ 以下である。$0 < w < 1$ となる実数コストがある場合、最小の w が 1 となるようにリスケールすることが出来る。 反復深化は $cost$ が最適解のコストになった時に停止するので、必要なメモリ量は最適解のコストに対して線形である。そのため、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反復深化 A*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。上述のように反復深化 A*は必要なメモリ量が非常にすくない。また、メモリアクセスパターンもかなりリニアである。そのため、ほぼキャッシュミスなく探索を行えるドメインも多い。例えば、15-puzzle などの状態が少ないビット数で表せられるドメインでは特にキャッシュ効率が良く、1 ノードの展開速度の差は **TODO** 倍という実験結果もある [24]。実際、15-puzzle では IDA*のほうが A*よりも速く解を見つけることが出来る [24]。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。

反復深化 A*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。反復深化 A*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、停

止性を満たさないことである。すなわち、問題に解がなく、グラフにループがある場合、単純な木探索は停止しない。よって、この手法は解が間違いなく存在することが分かっている問題に対して適用される。あるいは、解が存在することを判定してから用いる。例えば 15-puzzle は解が存在するか非常に高速に判定することが出来る。

Algorithm 11: Iterative Deepening A*

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

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

Algorithm 12: CLDFS: Cost Limited Depth First Search

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

```
if Goal(s) then
    return s;
for each child ∈ Expand(s) do
    found ← CLDFS(child, cost − 1);
    if found ≠ ∅ then
        return sequence (s, found);
return ∅;
```

7.2.1 Transposition Table

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

ここでもミソは重複検出は生成済み

7.3 両方向探索 (Bidirectional Search)

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

7.4 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。

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。すなわち、Open、Closed の一部を外部記憶に保持し、必要に応じて参照し RAM に持ってくる、ということをする。External Search 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 7.2 は一般的なコンピュータのキャッシュ・メモリ・ハードディスクへのアクセスレイテンシーを比較した表である。メモリから 1MB 逐次に読みだすオペレーションは 250,000 nanosec かかるが、ハードディスクからの読出しは 20,000,000 nanosec もかかる。更にハードディスクにランダムアクセスする場合 (Disk seek) は 8,000,000 nanosec もかかる。よって、HDD は工夫して使わなければ実行時間が非常に遅くなってしまう。

7.4.1 External 幅優先探索

7.4.2 External A*

7.5 Symbolic Search

Binary Decision Diagram (BDD) は二分木によってブーリアン vector からブーリアンへの関数 $(x_0, x_1, ..., x_n) \to \{0, 1\}$ を効率良く表すグラフ構造である。Symbolic Search では 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。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。International Planning Competition (2014) の Sequential Optimal 部門 (最適解を見つけるパフォーマンスを競う部門) の一位から三位まで

Table 7.2: 一般的なハードウェアのアクセス速度。メモリへのアクセス速度に対して外部記憶のアクセスは遅い。加えて、ランダムアクセスは seek の時間がかかるためさらに遅くなる。 (https://gist.github.com/jboner/2841832)

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

Algorithm 13: External Breadth-first search

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

$Open(-1) \leftarrow \emptyset$;
$Open(0) \leftarrow \{s\}$;
$i \leftarrow 1$;
while $Open(i-1) \neq \emptyset$ **do**
    $A(i) \leftarrow Succ(Open(i-1))$;
    if $Goal(Open(i))$ **then**
        return $Construct(Open(i))$;
    $A'(i) \leftarrow RemoveDuplicates(A(i))$;
    $Open(i) \leftarrow A'(i) \setminus (Open(i-1) \cup Open(i-2))$;
    $i \leftarrow i+1$;
return No solution found

Algorithm 14: External A* search

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

XXX **return** No solution found

を Symbolic Search が総なめした。現在 (2017 年) の state-of-the-art の手法であるといえるだろう。

7.5.1 特徴表現 (Symbolic Representation)

説明のためにシンプルな Sliding-token puzzle を用いる (図 7.1)。初期状態でタイルは位置 0 にある。タイルは右か左に動かすことが出来る。ゴール状態はタイルを位置 3 に置いた状態である。タイルの位置 $x(dom(x) = \{0, 1, 2, 3\})$ はバイナリ (x_0, x_1) に変換されている。状態および状態の集合は特徴関数 $\phi : S \to \{0, 1\}$ によって記述される。

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

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

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

$$\begin{aligned} Trans(x, x') = &(\neg x_0 \neg x_1 \neg x'_0 x'_1) \\ &\vee (\neg x_0 x_1 \neg x'_0 \neg x'_1) \\ &\vee (\neg x_0 x_1 x'_0 \neg x'_1) \\ &\vee (x_0 \neg x_1 \neg x'_0 x'_1) \\ &\vee (x_0 \neg x_1 x'_0 x'_1) \\ &\vee (x_0 x_1 x'_0 \neg x'_1) \end{aligned} \tag{7.1}$$

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)。

Figure 7.1: Sliding-token puzzle とそのバイナリ表現

Table 7.3: Sliding-token puzzle のエンコーディング

State ID	State Role	Binary Code	Boolean Formula
0	初期状態	00	$\neg x_0 \neg x_1$
1	-	01	$\neg x_0 x_1$
2	-	10	$x_0 \neg x_1$
3	ゴール状態	11	$x_0 x_1$

7.5.2 Binary Decision Diagram

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

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

BDD は決定木と同様な処理によって入力 x に対して $\{1, 0\}$ を返す。すなわち、根ノードから始まり、ノードのラベル x_i に対して、入力 x の x_i が 1 であればラベル 1 が付いたエッジをたどり、0 であればラベル 0 をたどる。これを繰り返し、シンクにたどり着いたらシンクのラベルの値を返す。決定木と異なり BDD は木ではなく、途中で合流などがあるため、決定木よりも空間効率が良い場合が多い。BDD を用いて集合演算を行うことが出来る。

BDD を使って状態やアクションの特徴関数を表現することが出来る。

7.5.3 特徴関数による状態空間の探索

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

ある状態集合 S に対して、$s \in S$ となる s の次状態の集合を S の *image* と呼

ぶ。S の image は以下の特徴関数によって表すことが出来る。

$$Image_S(x') = \exists x(Trans(x, x') \wedge \phi_S(x)) \tag{7.2}$$

BDD-幅優先木探索

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

$$\phi_{S_i}(x') = \exists x(\phi_{S_{i-1}}(x) \wedge Trans(x, x')) \tag{7.3}$$

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

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

Algorithm 15: BDD Breadth-first Tree Search

Input : Initial node s_0
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path with cost $\leq cost$

```
1 S_0 ← {s_0};
2 for i = 1... do
3 |   φ_{S_i}(x) ← ∃x(φ_{S_{i-1}}(x) ∧ Trans(x, x'))[x'/x];
4 |   if φ_{S_i}(x) ∧ φ_T ≠ false then
5 |   |   return Construct(φ_{S_i} ∧ φ_T(x), i);
```

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

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

BDD-幅優先探索

重複検出を行う場合はクローズドリストに展開済みノードを保存する必要がある。この展開済みノードも特徴関数及び BDD で表すことが出来る。アルゴリズム 15 は

Algorithm 16: BDD Breadth-first search

Input : Initial node s_0
Output : Path from s to a goal node $t \in T$, or $\emptyset$ if no such path with cost $\leq cost$

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

7.6 並列探索 (Parallel Search)

近年コンピューター台当たりのコア数は増加を続けており、コンピュータクラスタにも比較的容易にアクセスが出来るようになった。Amazon Web Service のようなクラウドの計算資源も普及し、将来的には並列化が当然になると考えられる。並列化の成功例は枚挙にいとまないが、近年のディープラーニングはまさに効率的な並列計算アーキテクチャによって得られたブレイクスルーであるといえる。もし CPU/GPU を利用した効率的なグラフ探索アルゴリズムが開発出来れば、非常に大きなインパクトになるかもしれない。グラフ探索アルゴリズムの並列化に考えなければならないオーバーヘッドは様々であり、それらの重要性は問題、インスタンス、マシン、さまざまな状況に依存する。加えてハードウェアは刻々と変化を続けており、数年後にどのような環境がメジャーとなるのかはなかなか想像をすることが出来ないだろう。本書では CPU を用いた分散メモリ並列アルゴリズムと GPU 一台を用いた並列アルゴリズムについて説明する。CPU 並列ではハッシュによってノードを各プロセスにアサインし、各プロセスはアサインされたノードのみを担当して探索を行うというフレームワークが現在の state-of-the-art である。一方、執筆時現在、GPU を用いたアルゴリズムはあまり研究が進んでいない。原因としては、既存の CPU を用いた探索アルゴリズムにはない様々な難

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

7.6.1 並列化オーバーヘッド

理想的には n プロセスで並列化したら n 倍速くなってほしい。逐次アルゴリズムと比較して、プロセス数倍の高速化が得られることを *perfect linear speedup* と呼ぶ。しかしながら、殆どの場合 perfect linear speedup は得られない。それは並列化にさいして様々なオーバーヘッドがかかるからである。[20] の記法に従うと、並列化オーバーヘッドは主に以下の 3 つに分けられる。

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

$$CO := \frac{\text{\# messages sent to other threads}}{\text{\# nodes generated}}. \tag{7.4}$$

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

$$CO := \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}. \tag{7.5}$$

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。

[1] 個人の感想である

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

$$SO := \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1. \tag{7.6}$$

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

$$LB := \frac{\text{Maximum number of nodes assigned to a thread}}{\text{Average number of nodes assigned to a thread}}. \tag{7.7}$$

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

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

同期オーバーヘッド (Coordination Overhead) 同期オーバーヘッドは他のスレッドの処理を待つためにアイドル状態にならなければならない時に生じるオーバーヘッドを指す。アルゴリズム自体が同期を必要としないものだとしても、メモリバスのコンテンションによって同期オーバーヘッドが生じることがある [6, 23].

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

7.6.2 Hash Distributed A*

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

1. プロセス T はメッセージキューを確認し、ノードが届いているかを確認する。届いているノードのうち重複でないものをオープンリストに加える (A* 同様、クローズドリストに同じ状態が存在しないか、クローズドリストにある同じ状態のノードよりも f 値が小さい場合に重複でない)。

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

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

HDA*が提案された論文 [23] では Zobrist hashing [39] 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：

$$Z(s) := R_0[x_0] \; xor \; R_1[x_1] \; xor \; \cdots \; xor \; R_n[x_n] \tag{7.8}$$

Zobrist hashing は初めにランダムテーブル R を初期化する 17。これを用いてハッシュ値を計算する。

Algorithm 17: ZHDA*

Input : $s = (x_0, x_1, ..., x_n)$
1 $hash \leftarrow 0$;
2 **for** *each* $x_i \in s$ **do**
3 $\quad hash \leftarrow hash \; xor \; R[x_i]$;
4 **Return** $hash$;

Algorithm 18: Initialize ZHDA*

Input : $V = (dom(x_0), dom(x_1), ...)$
1 **for** *each* x_i **do**
2 $\quad$ **for** *each* $t \in dom(x_i)$ **do**
3 $\quad\quad R_i[t] \leftarrow random()$;
4 **Return** $R = (R_1, R_2, ..., R_n)$

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 [6]。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 [18]。AZH は feature から abstract feature へのマッピングを行い、abstract feature を Zobrist hashing への入力とするという手法である：

$$Z(s) := R_0[A_0(x_0)] \; xor \; R_1[A_1(x_1)] \; xor \; \cdots \; xor \; R_n[A_n(x_n)] \tag{7.9}$$

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

AZH はパラメータとして abstract feature を設定しなければならない。Abstract feature を自動的に生成する手法は複数提案されており、最もシンプルなものは Greedy abstract feature generation [19] である。

Domain transition graph

Abstract feature の生成方法として state-of-the-art の手法は Graph partitioning-based [20] であり、各 DTG を与えられた最適化指標下で分割することで abstract feature を生成する。

Chapter 8

探索問題の派生

この章では 2 章で定式化した状態空間問題と少し定式化の異なる問題を扱う。ブラックボックスプランニングは状態空間問題の一つであるが、今まで扱った問題と異なり、ドメインモデルがブラックボックスで与えられるという違いがある (8.1 章)。よってヒューリスティック関数を生成することが出来ず、情報なし探索が必要となる。

8.1 ブラックボックスプランニング (Blackbox Planning)

プランナーは PDDL を用いることでドメインの知識を吸い出し、それを利用して探索を効率化する。しかしながら、完全なモデルを得るのが難しい問題の場合、PDDL のような記述を得ることが出来ない。例えばビデオゲームのような環境では、ゲームをクラックしない限り、完全なモデルを得ることは出来ない。このような中身を見ることの出来ない環境でのプランニング問題をブラックボックスプランニング問題と呼ぶ。

ブラックボックスプランニングは Atari 2600 や [27] や、General Video Game Playing [13] などのビデオゲームなどの環境に応用されている。

ブラックボックスプランニング問題は状態空間問題である。状態 s は有限長の配列 V で表せられ、$v \in V$ の値域は $D(v)$ とする。ただし、V の各変数がどのような意味を持つのかは未知である。Expand 関数、Goal 関数はブラックボックスとして与えられる。また、ある状態に対して A のうち実行可能なアクションの集合が既知とは限らない[1]。

[1] 厳密にブラックボックスである場合は既知とするべきではないが、多くの研究ではオラクルによって実行可能なアクションが知らされるというモデルを用いている。

このようなドメインではドメインの知識を得ることが出来ないので、4 章で解説したようなヒューリスティック関数を用いることは出来ない。

幅優先探索などによって Brute-force に探索しつくす方法を取ることも出来るが問題のサイズが大きい場合に解くことが出来ない [5]。Iterative Width 探索 (IW search)[27] は幅優先探索に新奇性による枝刈りを加えた手法である[2]。IW(1) は新しく生成された状態は新しい atom を真にしない場合、枝刈りされる。

8.1.1 新奇性に基づく枝刈り

inadmissible pruning Novelty-based pruning Iterative Width search

8.1.2 ビデオゲーム AI: Atari 2600

8.2 二人プレイヤーゲーム

and or tree Alpha beta pruning

8.2.1 $\alpha\beta$ 木

8.2.2 Monte Carlo Tree Search

8.3 オンラインプランニング

[2]Iterative Width 探索はドメインモデルのある場合でも有用であることが知られている [26]。

Chapter 9

機械学習と探索・プランニング (Machine Learning, Search, and Planning)

A review of ML for AP [17]

9.1 機械学習による探索の効率化

YJ DASP

9.2 ドメインモデルの生成

本書を通して扱ってきた状態空間問題の大きな問題点は、問題モデルをどのように獲得するか、である。1.2 章で述べたように、本書はこれまで正しいモデルが与えられていることが前提として話を進めてきた。

多くのアプリケーションではドメインモデル (e.g. PDDL) は人間のエキスパートが手でコーディングする。しかしながら、この方法だと人間のエキスパートがあらかじめ想定した環境にしか適用できない。ダイナミックな環境で活躍できるようなエージェントを実装するためには、エージェントが何らかの方法でドメインモデルを生成する方法が必要である。

LOCM はプランからアクションスキームを生成する。LOCM static な constraint も見つける。

階層的プランニングオプション Kaelbling Konidaris et al.

9.3 探索と機械学習の融合

R. Sutton David Silver RL and simulation-based search

9.3.1 Alpha Go

9.4 参考文献

Predictron DL for Reward design in MCTS Juhn, Satinder, et al.

Index

Bibliography

[1] Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I.D., Ram, A., Veloso, M., Weld, D., SRI, D.W., Barrett, A., Christianson, D., et al.: Pddl—the planning domain definition language version 1.2 (1998)

[2] Algfoor, Z.A., Sunar, M.S., Kolivand, H.: A comprehensive study on pathfinding techniques for robotics and video games. International Journal of Computer Games Technology **2015**, 7 (2015)

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

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

[5] Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research **47**, 253–279 (2013). DOI 10.1613/jair.3912. URL `http://arxiv.org/abs/1207.4708http://dx.doi.org/10.1613/jair.3912`

[6] 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)

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

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

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

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

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

[12] 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)

[13] 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)

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

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

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

[17] Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., Borrajo, D.: A review of machine learning for automated planning. The Knowledge Engineering Review **27**(04), 433–467 (2012)

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

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

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

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

[22] Kautz, H., Selman, B.: Planning as Satisfiability. In: ECAI, pp. 359–363 (1992)

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

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

[25] Korf, R.E., Zhang, W.: Divide-and-conquer frontier search applied to optimal sequence alignment. In: Proceedings of the 17th National Conference on Artificial Intelligence AAAI-00, pp. 910–916 (2000)

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

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

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

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

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

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

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

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

[34] 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)

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

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

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

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

[39] 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