ヒューリスティック探索

陣内　佑
東京大学　総合文化研究科

March 8, 2017

List of Todos

Contents

まえがき

ヒューリスティック探索はグラフ探索のサブフィールドであり、解こうとしている問題の知識を探索方法に反映させることでより効率的に探索をしよう、という分野である。

ヒューリスティック探索のイントロダクションと現在どのように発展しているのかというのをまとめてみようかと思い、本文を執筆した。以前、さる国内の高名な AI 研究者がご講演で「ヒューリスティック探索は終わった技術であり、Toy Problem しか解けない」とおっしゃった。これは全くの勘違いであるが、思い返すと仕方がないことかと思われる。というのも、日本には探索分野、特にヒューリスティック探索の研究者というのは数えるほどしかいない。大御所の方々は大変忙しく、他分野、ましては自分では終わったと思っている分野の英語論文など読まないだろう。こうなると、その分野の知識は古いままで、なおのこと終わった分野だと思ってしまいがちである、と想像される。そこで、若輩者ながら、数少ない日本語の書けるヒューリスティック探索アルゴリズムの研究者として、日本の AI 研究に微力を添えようと日本語のテキストを書こうと思った次第である。

Chapter 1

イントロダクション

人は様々な問題を探索によって解決している。例えば飛行機で成田からロンドンに行く安い/速い方法などを計画するのは探索の一つである。あるいは囲碁、将棋、チェスなどのゲームも、ある手を選んだ時にどのような局面につながるのかを先読みし、選ぶべき次の一手を探索する。このような様々な問題はグラフ探索問題として統合してモデルすることが出来る。

この章ではまず、1.1 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 1.2 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。

1.1 節における定式化は [21]、[20]、[8] 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。

1.1 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。このテキストでは探索の主な対象として状態空間問題 (State-space problem) を考える。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。状態空間グラフ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。どのような解でもよいのか、経路を最短にする解が良いのか、様々な問題が考えられる。多くの問題では経路のコストの合計を最小に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)* であると言う。

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問題 (ユニットコストドメイン) 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。

2 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、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で表せられる。

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

探索の開始時、エージェントは初期ノードのみを保持する。エージェントは保持しているノードに対して Expand を適用することによって、新しいノードとエッジをグラフに加える。これを求める解を見つけるまで繰り返す。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。

1.2 状態空間問題の例

グラフ探索アルゴリズムによって効率的に解くことが出来ると知られているドメインをいくつか紹介する。

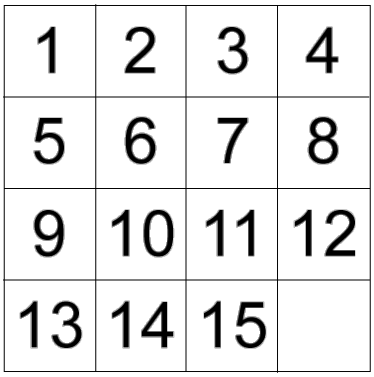

Figure 1.1: 15 パズルのゴール状態の例

1.2.1 グリッド経路探索 (Grid Path-finding)

グリッド経路探索問題は 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。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。

Web 上に簡単に試せるデモがあるので、参照されたい:`http://qiao.github.io/PathFinding.js/visual/`。とてもよくできており、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ゲーム AI などで応用される。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。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 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)。あるいはロボットのモーションプランニングなども経路探索となる。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。

1.2.2 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 1.1)。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 $|V| = (n^2)!/2$[1]であり、n に対

[1]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 [**?**]。

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）を計算する必要がある。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。

1.2.3 Multiple Sequence Alignment (MSA)

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。MSA は複数の遺伝子・アミノ酸配列が与えられた時、それらの配列間の編集距離とその時出来上がった配列を求める問題である。 2 つの配列に対してそれぞれコストの定義された編集操作を繰り返し、同一の配列に並べ替える手続きをアライメントと呼ぶ。 2 つの配列の編集距離はアライメントの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 1.2b 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 があるが、ここでは簡単のため仮のコスト表を用いる [**?**]。図 1.2a 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 1.2b 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。図 1.2c 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 1.2c 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、分枝数は配列の数 n に対して $2^n - 1$ と大きい。

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

	A	B	C
B	→	↘	
C		↘	↓
B			↘

(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

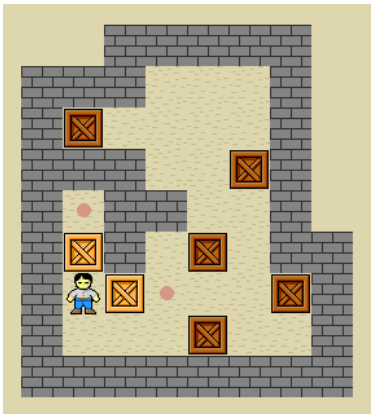

Figure 1.2: Sokoban: 画像は wikipedia より。

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。例えば動的計画法が使われている。

1.2.4 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。

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

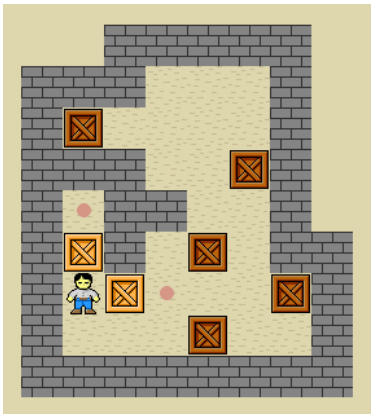

Figure 1.3: 巡回セールスパーソン問題: 画像は wikipedia より。

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。

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

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

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。TSP は NP 完全であることが知られている。TSP 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。

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 [2]。この問題ドメイン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。

Chapter 2

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。情報なし探索で重要なのは 1. 重複検知を行うかと 2. ノードの展開順序の二点である。重複検知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 (らしい)[1]。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（らしい）[2]。

2.1 木探索アルゴリズム (Tree Search Algorithm)

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

アルゴリズム 1 は木探索の疑似コードである。

以下、(k) と書いて疑似コードの k 行目を指すことにする。木探索はオープン

[1] TODO: 典拠
[2] TODO: 典拠

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 ∅;
```

リスト[3]と呼ばれるノードの集合を保持する。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。

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: $Expand(u)$ 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: $Open.insert(v)$ 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。$Open$ は Priority queue であり、何らかの基準によって $Open$ から取り出してくるノードを決める。

木探索アルゴリズムは名前の通り木を探索する場合により効率的であることが多いが (後述するが、重複検出をしないため)、グラフ一般に対して適用することが出来る。

[3]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。

2.2 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。最も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。またの名を幅優先探索と言う。

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 p 値と定義する。すると、幅優先探索の $Open.pop()$ はアルゴリズム 2 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、p 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 2: Breadth-First Search: $Open.pop()$

Output : Node u
1 **return** $\arg\min_n p(n)$

2.3 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。

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

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

2.4 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、重複の効率的な検出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 4: 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 5: $Improve(u, v)$

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;
```

Chapter 3

ヒューリスティック探索

2 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。

3.1 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。

3.2 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。$h : V \to \mathbb{R}_{\geq 0}$

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 $h(u) \leq \delta(u, T)$ が成り立つ。

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。

3.3 A*探索

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 [**?**]。A*探索は以下の f 値が最小となるノードを優先して探索を行う。

$$f(n) = g(n) + h(n) \tag{3.1}$$

Algorithm 6: A*: $Open.pop()$

Output : Node u

return $\arg\min_n f(n)$

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。

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。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限である。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。

3.3.1 重み付き A*探索

許容的なヒューリスティックを用いた場合、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。

$$f_w(n) = g(n) + wh(n) \tag{3.2}$$

Algorithm 7: w A*: $Open.pop()$

Output : Node u

return $\arg\min_n f_w(n)$

3.4 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 8: Greedy Best-First Search: $Open.pop()$

Output : Node u

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

Chapter 4

ヒューリスティック探索 variants

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 Closed 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 [**?**]。

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。

4.1 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。

2.4 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 [17]。アルゴリズム 9 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 [17]。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 [17]。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 9: 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;
```

4.1.1 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

Algorithm 10: 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 $\leq cost$

```
1 if Goal(s) then
2 |   return s;
3 for each child ∈ Expand(s) do
4 |   found ← CLDFS(child, cost − 1);
5 |   if found ≠ ∅ then
6 |   |   return sequence (s, found);
7 return ∅;
```

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

4.2 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。

4.3 External Search

A*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。

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。`http://norvig.com/21-days.html#answers` 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[1]。

[1]似たような理由で、HDD を用いない RAM ベースの探索を効率化するためにはキャッシュ効率を工夫しなければならない。詳しくは [7] を参照されたい。

4.3.1 External A*

4.4 Symbolic Search

Binary Decision Diagram (BDD) は二分木によってブーリアン vector からブーリアンへの関数 $(x_0, x_1, ..., x_n) \rightarrow \{0, 1\}$ を効率良く表すグラフ構造である。Symbolic Search 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。最新の International Planning Competition (2014) の Sequential Optimal 部門 (最適解を見つけるパフォーマンスを競う部門) の一位から三位までを Symbolic Search が総なめした。現在 (2017 年) の state-of-the-art の手法であるといえるだろう。

4.4.1 Binary Decision Diagram

BDD を使う準備として、状態をブーリアン vector に変換する。状態空間問題の状態 s が定数長の vector であるとすると、例えばそのビット vector を BDD に使うブーリアン vector として使うことが出来る。

Definition 8 (特徴関数) 特徴関数 $\phi : s \rightarrow \{0, 1\}$ は状態の集合を表すために用いられ、状態 s が集合に含まれれば *1* を返し、なければ *0* を返す。

例: Sliding-token puzzle

図**??**は Sliding-token puzzle という説明のために作られた問題である。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。

4.5 Exploration Based Search

Iterative Width search [18].
Prioritized iterative width search [22]
Invasion percolation [4]

4.6 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[2]。

4.6.1 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。[15] 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：

$$CO := \frac{\text{\# messages sent to other threads}}{\text{\# nodes generated}}. \tag{4.1}$$

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：

$$CO := \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}. \tag{4.2}$$

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

[2]個人の感想である

ンにおいて重要なオーバーヘッドになる。一般に、プロセス数が多いほど CO は大きくなる。

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

$$SO := \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1. \tag{4.3}$$

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。

$$LB := \frac{\text{Maximum number of nodes assigned to a thread}}{\text{Average number of nodes assigned to a thread}}. \tag{4.4}$$

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, 16].

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。

4.6.2 Hash Distributed A*

ハッシュ分配 A* (Hash Distributed A*, HDA*) [16] 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 [16] では Zobrist hashing [24] 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：

$$Z(s) := R_0[x_0] \ xor \ R_1[x_1] \ xor \ \cdots \ xor \ R_n[x_n] \tag{4.5}$$

Zobrist hashing は初めにランダムテーブル R を初期化する 12。これを用いてハッシュ値を計算する。

Algorithm 11: 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 12: 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 [13]。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：

$$Z(s) := R_0[A_0(x_0)] \; xor \; R_1[A_1(x_1)] \; xor \; \cdots \; xor \; R_n[A_n(x_n)] \tag{4.6}$$

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 [14] 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 [15] 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。

4.6.3 GPU-based Parallelization

Chapter 5

古典的プランニング問題

この章では古典的プランニング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 [10]。

ロジスティック [12, 23]、セルアセンブリ [3]、遺伝子距離計算 [9]、ビデオゲーム [19] 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 [21]。

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 [**?**]。

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 [10]。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 [8] に従う。

Definition 9 (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 [10] 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 ブラックボックスプランニング

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 [**?**] や、General Video Game Playing [11] 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]。

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、3 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)[**?**] 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。

[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。

[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 [18]。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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

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 寰杯 he planning domain definition language version 1.2 (1998)

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

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

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

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

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

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

[10] 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)

[11] 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)

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

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

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

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

[16] 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

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

[18] 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

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

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

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

[22] Shleyfman, A., Tuisov, A., Domshlak, C.: Blind search for Atari-like online planning revisited. In: Proc. IJCAI, pp. 3251–3257 (2016). URL http://www.ijcai.org/Abstract/16/460

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

[24] 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