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

January 29, 2017

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教科書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。状態空間問題は様々な応用問題を含む。次に 1.2 節でグラフ探索で解くことの出来る問題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探索研究界隈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 状態空間問題 (State-Space Problem)

探索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経路を返すのが探索問題である。このテキストでは探索問題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。すなわち、アクションに対して

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。

ただし、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陽状態空間グラフ (explicit state-space graph) と呼ぶとする[1]。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陰状態空間グラフで表せられる。

Definition 4 (陰状態空間グラフ、Implicit state-space graph) 陰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。大まかには、情報なし探索による陰グラフは陽グラフよりも指数的に小さく、ヒューリスティック探索による陰グラフは情報なし探索による陰グラフよりも更に指数的に小さいことが多い。

1.2 探索問題の例

グラフ探索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。

[1] 他に対訳を見つけることが出来なかったため、陽・陰状態空間グラフという訳は筆者がつけた。

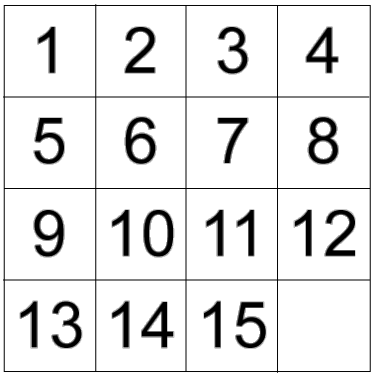

Figure 1.1: 15 パズルのゴール状態の例

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 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。

グリッド経路探索はゲーム AI などで問題となる。

Starcraft 1 では A*探索が使われていた。マルチエージェント経路探索の場合は flocking / swarm AI が使われている。Starcraft 2 では

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)[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 $|V| = (n^2)!/2$[3]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）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。

[2]機械学習・画像処理におけるところの MNIST だろうか

[3]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 [**?**]。

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
	X	X	
B			X	
C				X
B				X

(c) グリッド経路探索への帰着

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)。

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。

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。

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

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。

Chapter 2

情報なし探索 (Blind Search)

最もシンプルなグラフ探索は問題（ドメイン）の知識を利用しない探索である。すなわち、何も情報を見ずに探索を行うという意味で Blind Search と言われる。Blind search の例としては幅優先探索・深さ優先探索などがあり、問題を選べばこれらの手法によって十二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]。4 章はグラフ探索の高速化の紹介をするので、特に競技プログラミングに興味がある場合はそちらも参照されたい。

2.1 木探索アルゴリズム (Tree Search Algorithm)

木は木探索はグラフに対して適用することが出来る。図**??**は木探索の例である。陽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陰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無駄にするだけなのでなんとかして避けたいものであり、重複の効率的な検出方法は探索研究の大きなヤマの一つである。

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。
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。

木探索ベースのアルゴリズムの問題は、解が存在しない場合に停止性を満たさないことである。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。

[1] TODO: 典拠
[2] TODO: 典拠

Algorithm 1: Implicit Tree Search

Input : Implicit problem tree 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  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 ∅;
```

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

木探索がノードの重複を無視して探索を行うのに対して、グラフ探索アルゴリズムはノードの重複を確認して探索を行う。ノードの重複の確認にはいくつかメリットがある。一つは、停止性を満たすことである。すなわち、最悪グラフのノードをすべて展開して停止する。もう一つは、重複して同じ状態を展開せずにすむことである。

Algorithm 2: 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 3: $Improve(u, v)$

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

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

2.3 幅優先探索 (Breadth-First Search)

Open list は Priority queue であり、何らかの基準によってセットから pop するノードを決めている。幅優先探索は探索の「幅」を最大化するようにノードを選択する。

初期状態から現在状態にたどり着くまでのコストをノードの g 値と定義する。

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

Output : Node u

```
return arg min_n g(n)
```

2.4 深さ優先探索 (Depth-First Search)

ゴールがある程度深い所にあり、浅い場所にはないと事前に分かっている場合に上手く行く。

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

Output : Node u

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

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、最もドミナントな手法である。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 $\arg\min_n f(n)$

Shakey the Robot Optimality

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

return $\arg\min_n h(n)$

Chapter 4

探索の高速化

工事中

4.1 Open

4.2 Closed

4.3 キャッシュ効率

Chapter 5

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。

5.1 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 Closed に保存していることに起因する。

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

反復深化 A*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。反復深化 A*は最適解を返すことを保証する。

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 : 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

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

Algorithm 10: CLA*: Cost Limited A*

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 ← CLA*(child, cost − 1);
    if found ≠ ∅ then
        return sequence (s, found);
return ∅;
```

5.1.1 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

5.2 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。

5.3 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]。

5.3.1 External A*

5.4 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 $(x_0, x_1, ..., x_n) \to \{0, 1\}$ 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。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。

5.4.1 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 $\phi : s \to \{0, 1\}$ 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。

[1]同様の理由で、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 [**?**] を参照されたい。

5.4.2 Symbolic Blind Search

5.4.3 Symbolic Heuristic Search

5.5 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。残念ながら、探索アルゴリズムの並列化は比較的難しいと考えられる。

5.5.1 Hash Distributed A*

Hash Distributed A* (HDA*) [10] is a parallel A* algorithm which incorporates the idea of hash-based work distribution from PRA* [5] and asynchronous communication from TDS [14]. In HDA*, each processor has its own OPEN and CLOSED. A global hash function assigns a unique owner thread to every search node. Each thread T repeatedly executes the following:

1. T checks its message queue if any new nodes are in. For all new nodes n in T's message queue, if it is not in CLOSED (not a duplicate), put n in OPEN.
2. Expand node n with the highest priority in OPEN. For every generated node c, compute hash value $H(c)$, and send c to the thread that owns $H(c)$.

HDA* has two distinguishing features compared to preceding parallel A* variants. First, there is little coordination overhead because HDA* communicates asynchronously, and locks for an access to shared OPEN/CLOSED are not required because each thread has its own local OPEN/CLOSED. Second, the work distribution mechanism is simple, requiring only a hash function. However, the effect of the hash function was not evaluated empirically, and the importance of the choice of hash function was not fully understood or appreciated – at least one subsequent work which evaluated HDA* used an implementation of HDA* which failed to achieve uniform distribution of the nodes (see Section **??**).

5.5.2 GPU-based Parallelization

5.6 Online Search

Online, planning is a real-time search problem [11], where we are given an initial black-box planning instance B_0, and a resource limit (e.g., time limit, limit on number of node generations, etc.). An agent for online black-box planning behaves as follows:

initialization : I is initialized to I_0.

termination check : If some termination condition has been met, then terminate.

planning episode : The agent applies a planning algorithm P until the resource limit is exhausted, at which point the agent selects an action a to execute.

world update : The agent executes a, resulting in an updated world state $s' = Apply(a, s)$. In black-box domains where the simulator Sim is a perfect model of the actual world inhabited by the agent, then $Apply(a, s) = Sim(a, s)$.

1. Set $I = s$, and go to step 2.

In step 3 (planning episode), after the planning algorithm is terminated, the selection of the action to execute in step 4 can be implemented in many different ways. In a satisficing problem, if a path has been found to a goal (maximal utility) state, then the first step on that path should be selected. However, in most cases, such a path is unavailable, so the action is chosen based on the search space that has been explored so far, e.g., choose the first step in the path with the highest utility frontier node.

5.7 関連研究

Chapter 6

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る []。

ロジスティック [8, 16]、セルアセンブリ [1]、遺伝子距離計算 [4]、ビデオゲーム [13] 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 []。

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 []。

6.1 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 [6]。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 [3] 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 [6] 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

$$s' = (s \cup add(a)) \setminus del(a) \tag{6.1}$$

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 $v_j = succ(v_i, a)$, $s'_0 = s_0$, $T' = T$.

6.2 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) [**?**] 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 6.1 は blocks-world の domain ファイルである。図 6.2 は blocks-world の instance ファイルである。

6.3 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 [**?**] 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 [2]。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 [**?**]。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 6.1: 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 6.2: blocks-world の instance ファイル

6.4 アプリケーション

6.4.1 クエリ最適化

6.4.2 ロジスティック

6.4.3 セルアセンブリ

6.4.4 宇宙探査車

6.4.5 サイバーセキュリティ

6.4.6 ゲノムリアレンジメント

6.4.7 グリッドコンピューティング

6.4.8 プリンタースケジュリング

6.4.9 ビデオゲーム

Atari General Video Game Playing

Bibliography

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

[2] 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

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

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

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

[6] 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)

[7] 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)

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

[9] Jinnai, Y., Fukunaga, A.: Learning to prune dominated action sequences in online black-box domain. In: Proc. AAAI (2017). (to appear)

[10] 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

[11] 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

[12] 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

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

[14] Romein, J.W., Plaat, A., Bal, H.E., Schaeffer, J.: Transposition table driven work scheduling in distributed search. In: Proc. AAAI, pp. 725–731 (1999)

[15] 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

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