Java のヒューリスティック探索を使用して問題の解決を迅速化する

人工知能によく使われている探索アルゴリズムの Java 実装について学ぶ

Matthew Hatem
Senior Software Engineer
IBM

Ethan Burns
Software Engineer
Google

Wheeler Ruml
Associate Professor
University of New Hampshire

2013年 8月 22日

ヒューリスティック探索と、その人工知能への応用について学んでください。この記事では、著者たちが最も広く使用されているヒューリスティック探索アルゴリズムの Java 実装を成功に導いた方法を説明します。そのソリューションでは、JCF (Java Collections Framework) の代わりとなるフレームワークを利用し、ガーベッジ・コレクションが過度に行われるのを回避するためのベスト・プラクティスを使用しています。

人工知能における基本的な手法は、考えられる解の空間を探索して問題を解決することです。この手法は、「状態空間探索」と呼ばれています。ヒューリスティック探索とは、問題に関する情報を活用して解をより効率的に検出する、状態空間探索の 1 つの形です。ヒューリスティック探索は、さまざまなドメインで大きな成功を収めてきました。この記事では、ヒューリスティック探索について概説し、最も広く使用されているヒューリスティック探索アルゴリズムである A* の Java プログラミング言語での実装を紹介します。ヒューリスティック探索アルゴリズムは、計算リソースとメモリーに対して高い要求を課しますが、私たちがこれに対処するために、コストの高いガーベッジ・コレクションを回避し、JCF (Java Collections Framework) に代わるハイパフォーマンスのフレームワークを利用して Java 実装を改善した方法についても説明します。この記事のすべてのコードは、ダウンロードすることができます。

ヒューリスティック探索

コンピューター・サイエンスにおける多くの問題は、グラフ・データ構造によって表現することができます。グラフ・データ構造では、グラフ内のパスで潜在的な解が表され、最適解を見つけ

るには、最短経路を見つける必要があります。例として、自律型テレビ・ゲームのキャラクターを想像してください。キャラクターが取ることができる動きのそれぞれがグラフ内の枝に対応し、キャラクターは、対立するキャラクターと対戦するための最短経路を見つけることを目標とします。

グラフ探索アルゴリズムとしてよく使用されているのは、深さ優先探索や幅優先探索などのアルゴリズムです。けれども、これらのアルゴリズムは十分な知識に基づくものではないと見なされており、通常は、そのアルゴリズムで解決可能な問題の大きさが厳しく制限されます。しかも、深さ優先探索で常に最適解が見つかるとは限りません (場合によっては、解を 1 つも見つけられないこともあります)。幅優先探索にしても、確実に最適解を見つけられるのは特殊な場合に限られます。それとは対照的に、ヒューリスティック探索は知識に基づいた探索であり、問題に関する情報がヒューリスティックの中にエンコードされているのを活用して問題をより効率的に解決します。ヒューリスティック探索では、知識に基づかないアルゴリズムでは解決できないさまざまな難しい問題を解決することができます。

ヒューリスティック探索がよく使用されている分野は、テレビ・ゲームでの経路探索ですが、ヒューリスティック探索ではもっと複雑な問題でも解決することができます。2007年の「DARPA Urban Challenge」無人ロボット・カー・レースの優勝者は、平坦かつ直線的な運転ルートを計画するためにヒューリスティック探索を適用しました (「参考文献」を参照)。ヒューリスティック探索は、自然言語処理の分野でも成功しており、音声認識におけるテキスト構文解析およびスタック・デコーディングに使用されています。また、ロボティクス (ロボット工学)、バイオインフォマティクス (生物情報科学) の分野でも適用されてきました。情報科学で盛んに研究されている多重配列アラインメント (Multiple Sequence Alignment: MSA) の問題は、ヒューリスティック探索を使用すると、従来の動的プログラミング手法よりも、短時間かつ少ないメモリー使用量で解決することができます。

Java でのヒューリスティック探索

Java プログラミング言語は、メモリーと計算リソースに対して高い要求を課すことから、これまでヒューリスティック探索を実装するための言語としてはあまり使用されていませんでした。実装言語としては、通常はパフォーマンス上の理由から C/C++ が選ばれていますが、私たちはこの記事で、Java がヒューリスティック探索の実装に適したプログラミング言語であることを実証します。まず初めに、A* の標準的な実装は処理に非常に時間がかかり、一般的なベンチマーク問題のセットを解決する際に使用可能なメモリーを使い尽くしてしまうことを明らかにします。これらのパフォーマンス問題に対処するために、私たちは重要な実装の詳細を再検討し、JCF の代わりとなる手段を活用します。

この作業の大部分は、この記事の著者たちが共同で執筆した学術論文に公開されている研究の延長線上にあります (「参考文献」を参照)。元の研究では C/C++ プログラミングに重点を置いていますが、この記事では同じ概念の多くが Java にも適用されることを明らかにします。

幅優先探索

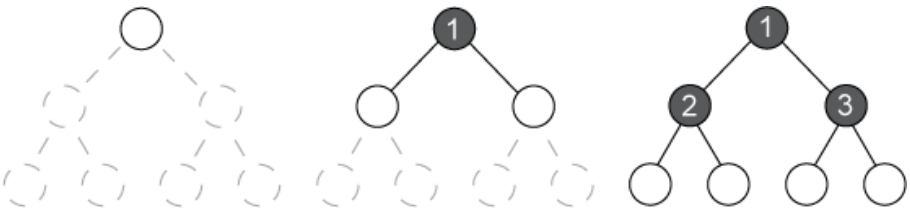
幅優先探索 (ヒューリスティック探索と、同じ概念および同じ用語を数多く共有する、より単純なアルゴリズム) の実装を十分に理解しておくと、ヒューリスティック探索の実装の詳細を理解するのに役立ちます。ここでは、幅優先探索のエージェント中心のビューを取り上げます。エージェ

ント中心のビューでは、エージェントはある特定の状態にあるとされ、その状態から実行可能な一連のアクションを適用することができます。アクションを適用すると、エージェントは現在の状態から新しい後続の状態に遷移します。このビューは、さまざまなタイプの問題に対して適切に一般化されます。

幅優先探索の目標は、エージェントをその初期状態から目標状態にまで導く一連のアクションを編み出すことです。幅優先探索は、初期状態を出発点として、最も新しく生成された状態から順に訪問することで、探索を進めていきます。適用可能なすべてのアクションを訪問先の各状態に適用することで新しい状態を生成し、その新しい状態をまだ訪問していない状態のリスト (探索の「先端」とも呼ばれます) に追加します。ある状態を訪問してその後続の状態のすべてを生成するプロセスは、一般に状態の「展開」と呼ばれます。

この探索手順は、木を生成することであると見なすことができます。つまり、木の根節点 (ルート・ノード) は初期状態を表し、子節点 (子ノード) 間を結ぶ枝が、それぞれの子節点を生成するために使用されたアクションを表します。図 1 に、この探索木の図を示します。白い円は探索の先端にある節点を表し、グレーの円は展開済みの節点を表します。

図 1. 二分木での幅優先探索の順序



探索木では、すべての節点が何らかの状態を表します。ただし、2 つの固有の節点が同じ状態を表すこともあります。例えば、探索木のある深さにある節点に関連付けられた状態が、探索木でそれよりも高いところにある別の節点に関連付けられた状態と同じである場合もあります。これらの重複する節点は、探索問題で同じ状態に至る 2 つの異なる方法を表します。重複は問題の原因になりがちなため、訪問したすべての節点を覚えておく必要があります。

リスト 1 に、幅優先探索の擬似コードを記載します。

リスト 1. 幅優先探索の擬似コード

```
function: BREADTH-FIRST-SEARCH(initial)
open ← {initial}
closed ← 0
loop do:
     if EMPTY(open) then return failure
     node ← SHALLOWEST(open)
     closed ← ADD(closed, node)
     for each action in ACTIONS(node)
          successor ← APPLY(action, node)
          if successor in closed then continue
          if GOAL(successor) then return SOLUTION(node)
          open ← INSERT(open, successor)
```

リスト 1 では、探索の先端を、「オープン・リスト」と呼ばれるリストに保持し (2 行目)、訪問した節点を「クローズド・リスト」と呼ばれるリストに保持します (3 行目)。クローズド・リストにより、ある節点を複数回訪問して探索作業を繰り返すことがなくなります。節点が先端に追加さ

れるのは、その節点がクローズド・リストに含まれていない場合のみです。探索ループは、オープン・リストが空になるか、目標が見つかるまで継続します。

図 1 でお気付きかもしれませんが、幅優先探索は、探索木の各深さの層にあるすべての節点を訪問してから、次の層に移るという方法で進められます。すべてのアクションのコストが等しい問題では、探索木のすべての枝に同じ重みが付けられます。この場合、幅優先探索で最適解を見つけられることが保証されます。つまり、最初に生成される目標が、初期状態からの最短経路です。

ドメインによっては、アクションごとにコストが異なる場合があります。このようなドメインでは、探索木の枝に付けられる重みは一様ではありません。この場合の解のコストは、探索木の根から目標までを結ぶ経路に沿ったすべての枝の重みを合計したものとなり、幅優先検索で最適解を見つけられるとは限りません。さらに幅優先探索では、目標が生成されるまで、探索木のすべての深さの層のすべての節点を展開する必要があります。これらの層を保管するために必要なメモリーは、最近のコンピューターで使用可能なメモリー量をすぐに上回ってしまうのが通常です。このことから、幅優先探索を使用できるのは、サイズの小さい問題に限定されます。

幅優先探索の拡張であるダイクストラ法では、探索の先端にある節点に対して、初期状態からその節点に到達するためのコストに応じて順序を付けます (コストを基準にオープン・リストをソートします)。したがって、アクションのコストが一様であるか一様でないかに関わらず (ただし、コストが負の値でないことが前提です)、最適解が見つかることが保証されます。その一方、ダイクストラ法では、最適解よりも少ないコストの節点をすべて訪問しなければならないため、サイズの小さい問題にしか適用できません。次のセクションでは、最適解を見つけるために訪問する必要のある節点の数を大幅に減らすことで、サイズの大きな問題の解決にも対処できるアルゴリズムについて説明します。

A* 探索アルゴリズム

A* アルゴリズム (あるいは、そのバリエーション) は、最も広く使用されているヒューリスティック探索アルゴリズムの 1 つです。ダイクストラ法の拡張であると見なすことができる A* アルゴリズムは、問題に関する知識を利用することで、解を見つけるために必要な計算の数を減らしながらも、最適解が見つかることを保証します。A* アルゴリズムとダイクストラ法は、最良優先グラフ探索アルゴリズムの典型例です。この 2 つのアルゴリズムが最良優先探索アルゴリズムである理由は、解が見つかるまで、最有力候補の節点 (目標への最短経路にあると思われる節点) を訪問するためです。多くの問題では、最適解を見つけることが必須であることから、A* のようなアルゴリズムが重要となってきます。

A* アルゴリズムを他のグラフ探索アルゴリズムと分け隔てているのは、これがヒューリスティックを使用するという点です。ヒューリスティックとは、より適切な決定を行うことを可能にする、問題に関する知識 (経験則) です。探索アルゴリズムのコンテキストでは、ヒューリスティックは特定の意味を持ちます。それは、特定の節点から目標に到達するまでの残りのコストを推定する関数という意味です。A* はヒューリスティックを利用して、最も訪問する見込みがあると思われる節点を決定することにより、不要な計算を回避します。A* は、グラフ内で最適解に至るとは思われない節点への訪問を回避しようとすることから、知識が少ないアルゴリズムに比べ、より少ないメモリーを使用して短時間で解を見つけられることが多いです。

A* が最も訪問する見込みがあると思われる節点を判別する方法は、各節点の値 (f 値と呼びます) を計算して、この値を基準にオープン・リストをソートするというものです。f 値の計算には、節点の他の 2 つの値である g 値と h 値が使用されます。節点の g 値は、初期状態から節点に到達するために必要なすべてのアクションの総コストです。h 値は、節点から目標に到達するための推定コストです。この推定が、ヒューリスティック探索でのヒューリスティックとなります。最も訪問する見込みがあると思われる節点になるのは、最も低い f 値を持つ節点です。

図 2 に、この探索手順を示します。

図 2. f 値に基づく A* の探索順序

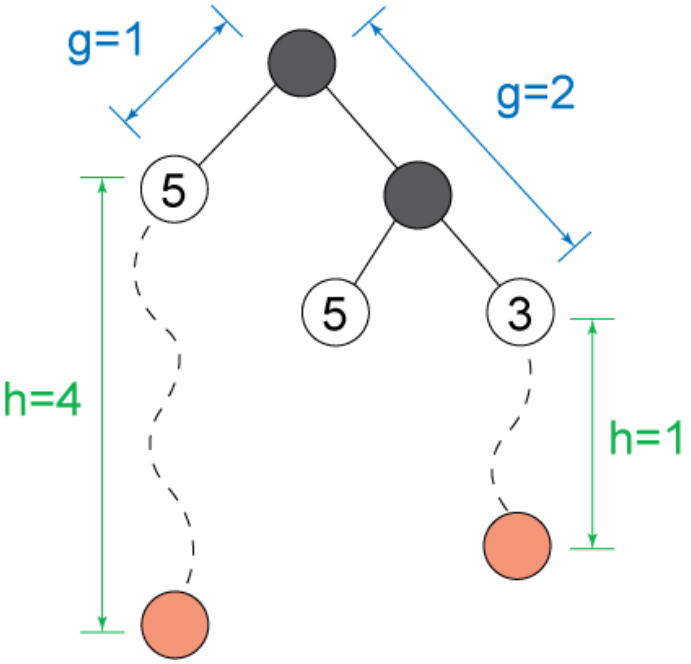
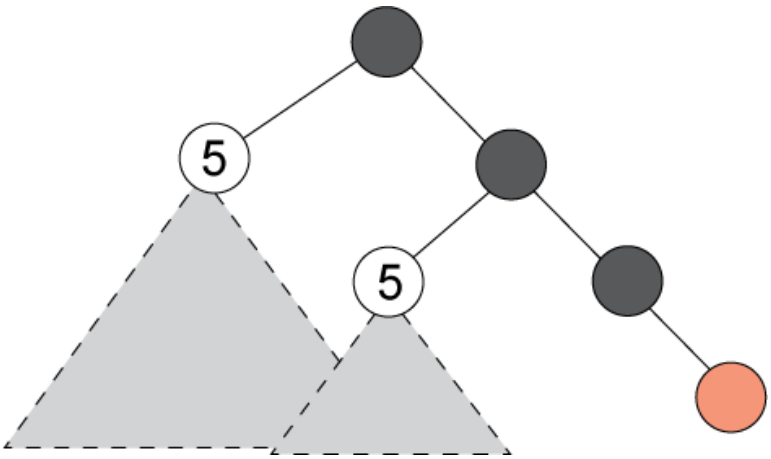


図 2 の例では、先端に 3 つの節点があります。そのうちの 2 つの節点の f 値は 5 で、もう 1 つの節点の f 値は 3 です。この図では、f 値が最も小さい節点が次に展開されて、すぐに目標に到達しています。したがって、A* は他の 2 つの節点の下にある副木を一切訪問する必要がありません (図 3 を参照)。このように、A* は、幅優先探索などのアルゴリズムよりも遥かに効率的となっています。

図 3. 大きい f 値を持つ節点の下にある副木を訪問する必要がない A*



A* が使用するヒューリスティックが「許容的」である場合、A* は最適解を見つけるために必要な節点しか訪問しません。これが、A* がよく使用されている理由です。許容的ヒューリスティックを使用した A* よりも少ない節点の訪問数で、最適解を見つけることを保証するアルゴリズムは他にありません。ヒューリスティックの推定を許容的にするには、ヒューリスティックが下限値、つまり目標に到達するためのコスト以下の値である必要があります。ヒューリスティックが「一貫性」という追加プロパティーを満たす場合には、最適経路を通じて初めてすべての状態が生成されるため、重複の処理に関してより効率的なアルゴリズムになります。

前のセクションで説明した幅優先探索の場合と同じく、A* は 2 つのデータ構造を保持します。生成された後にまだ訪問されていない節点は「オープン・リスト」に保管され、訪問済みのすべての標準節点は「クローズド・リスト」に保管されます。この 2 つのデータ構造の実装および使用法は、パフォーマンスを大きく左右します。この点については後のセクションで詳しく説明するとして、リスト 2 に、標準的な A* 探索の完全な擬似コードを記載します。

リスト 2. A* 探索の擬似コード

```
function: A*-SEARCH(initial)
open ← {initial}
closed ← 0
loop do:
     if EMPTY(open) then return failure
     node ← BEST(open)
     if GOAL(node) then return SOLUTION(node)
     closed ← ADD(closed, node)
     for each action in ACTIONS(node)
          successor ← APPLY(action, node)
          if  successor in open or successor in closed
              IMPROVE(successor)
          else
              open ← INSERT(open, successor)
```

リスト 2 では、A* はオープン・リスト内の初期状態の節点から探索を開始します。ループの繰り返し処理ごとに、オープン・リストから最良節点が取り除かれます。次にその (リスト 2 では `open` である) 最良節点に対して適用可能なすべてのアクションが適用され、考えられるすべての後続節点が生成されます。これらの後続節点のそれぞれについて、その節点が表す状態がすでに訪問されているかどうかを調べます。まだ訪問されていない場合は、その節点をオープン・リストに追加します。訪問済みであれば、前に到達したときよりも良い経路でこの状態に到達したか

どうかを判断する必要があります。良い経路で到達したのであれば、この節点をオープン・リストに追加して、準最良となった節点を除去します。

上記の擬似コードは、解決対象の問題に関する 2 つの前提によって、単純化することができます。それは、すべてのアクションのコストは同じであるという前提、そしてヒューリスティックが許容的で一貫しているとい前提です。ヒューリスティックが一貫していて、ドメイン内のすべてのアクションのコストが同じであることから、この前提の下では、ある状態をより良い経路で再び訪問することは決してありません。また、一部のドメインでは、新しい節点を生成するたびにオープン・リストに重複する節点があるかどうかを調べるよりも、オープン・リストで重複した節点を許容するほうが効率的であることが明らかになっています。したがって、その節点を訪問済みであるかどうかに関わらず、すべての後続節点をオープン・リストに追加することで、実装を単純化することができるというわけです。この擬似コードを単純化するには、リスト 2 の最後の 4 行を 1 つの行に結合します。一方、循環を避けなければならないことに変わりはないため、節点を展開する前に、重複があるかどうかをチェックする必要があります。`IMPROVE` 関数は単純化したバージョンでは不要になるため、この関数の詳細は省略することができます。リスト 3 に、単純化した後の擬似コードを記載します。

リスト 3. A* 探索の単純化された擬似コード

```
function: A*-SEARCH(initial)
open ← {initial}
closed ← 0
loop do:
     if EMPTY(open) then return failure
     node ← BEST(open)
     if node in closed continue
     if GOAL(node) then return SOLUTION(node)
     closed ← ADD(closed, node)
     for each action in ACTIONS(node)
          successor ← APPLY(action, node)
          open ← INSERT(open, successor)
```

Java での A* の標準的な実装

このセクションでは、リスト 3 の単純化した擬似コードを基にした、Java での A* の標準的な実装について解説します。この後わかるように、この実装では、30GB のメモリー制約内で標準的なヒューリスティック探索ベンチマークを解決することはできません。

この実装はできる限り汎用的なものにしたいので、まずは、A* の解決対象とする問題を抽象化するインターフェースをいくつか定義するところから始めます。A* を使用して解決する問題は、いずれも `Domain` インターフェースを実装する必要があります。`Domain` インターフェースは、以下の処理を行うメソッドを提供します。

- 初期状態についての問い合わせ
- ある状態で適用可能なアクションについての問い合わせ
- ある状態のヒューリスティック値の計算
- 後続の状態の生成

`Domain` インターフェースの完全なコードは、リスト 4 のとおりです。

リスト 4. `Domain` インターフェースの Java ソース

```
public interface Domain<T> {
  public T initial();
  public int h(T state);
  public boolean isGoal(T state);
  public int numActions(T state);
  public int nthAction(T state, int nth);
  public Edge<T> apply (T state, int op);
  public T copy(T state);
}
```

A* 探索では、探索木の枝オブジェクトと節点オブジェクトが生成されるため、`Edge` クラスと `Node` クラスが必要です。各節点には 4 つのフィールドが含まれます。これらのフィールドの内容は、節点が表す状態、親節点への参照、節点の `g` 値、節点の `h` 値です。リスト 5 に、`Node` クラスの完全なコードを記載します。

リスト 5. `Node` クラスの Java ソース

```
class Node<T> {
  final int f, g, pop;
  final Node parent;
  final T state;
  private Node (T state, Node parent, int cost, int pop) {
    this.g = (parent != null) ? parent.g+cost : cost;
    this.f = g + domain.h(state);
    this.pop = pop;
    this.parent = parent;
    this.state = state;
  }
}
```

各枝には、3 つのフィールドがあります。これらのフィールドの内容は、枝のコストまたは重み、枝の後続節点を生成するために使用するアクション、そして枝の親節点を生成するために使用するアクションです。リスト 6 に、`Edge` クラスの完全なコードを記載します。

リスト 6. `Edge` クラスの Java ソース

```
public class Edge<T> {
  public int cost;
  public int action;
  public int parentAction;
  public Edge(int cost, int action, int parentAction) {
    this.cost = cost;
    this.action = action;
    this.parentAction = parentAction;
  }
}
```

A* アルゴリズム自体は `SearchAlgorithm` インターフェースを実装することになり、`Domain` インターフェースと `Edge` インターフェースのみを必要とします。`SearchAlgorithm` インターフェースには、指定された初期状態で探索を行うためのメソッドが 1 つあるだけです。この `search()` メソッドは、`SearchResult` のインスタンスを返します。`SearchResult` クラスが提供するのは、探索に関する統計です。リスト 7 に、`SearchAlgorithm` インターフェースの定義を記載します。

リスト 7. `SearchAlgorithm` インターフェースの Java ソース

```
public interface SearchAlgorithm<T> {
  public SearchResult<T> search(T state);
}
```

オープン・リストとクローズド・リストに使用するデータ構造の選択は、実装の重要な詳細です。ここでは、Java の `PriorityQueue` を使用してオープン・リストを実装します。`PriorityQueue` は平衡二分ヒープの実装であり、要素のエンキューとデキューには O(log n) 時間を要し、要素がキューに入れられているかどうかの判断には線形時間を要し、キューの先頭にアクセスするには一定時間でを要します。二分ヒープは、オープン・リストの実装によく使用されるデータ構造です。後で、一部のドメインでは、「バケット優先度付きキュー (bucket priority queue)」と呼ばれる、より効率的なデータ構造を使用してオープン・リストを実装できることを説明します。

`PriorityQueue` が節点を適切にソートできるようにするには、`Comparator` インターフェースを実装する必要があります。A* アルゴリズムでは、各節点をその f 値を基準にソートしなければなりません。多数の節点が同じ f 値を持つドメインの場合、1 つの単純な最適化として、大きい g 値を持つ節点を優先することで順序付けを行うという方法があります。ここで、少し時間を割いて、このようにして順序付けを行うと A* のパフォーマンスが向上する理由を十分に理解してください (ヒント: `h` は推定値ですが、`g` はそうではありません)。リスト 8 に、`Comparator` 実装の完全なコードを記載します。

リスト 8. `NodeComparator` クラスの Java ソース

```
class NodeComparator implements Comparator<Node> {
  public int compare(Node a, Node b) {
    if (a.f == b.f) {
      return b.g - a.g;
    }
    else {
      return a.f - b.f;
    }
  }
}
```

実装する必要があるもう 1 つのデータ構造はクローズド・リストですが、これを実装するのにうってつけなのは、Java の `HashMap` クラスです。`HashMap` クラスはハッシュ・テーブルの実装であり、適切なハッシュ関数を使用すれば、期待される一定の時間で要素を取得および追加します。ドメインの状態を実装するクラスの `hashcode()` メソッドと `equals()` メソッドはオーバーライドする必要があります。この実装については、次のセクションで見ていきます。

実装する必要がある最後のインターフェースは、`SearchAlgorithm` です。それには、リスト 3 の擬似コードを使用して `search()` メソッドを実装します。リスト 9 に、A* の `search()` メソッドの完全なコードを記載します。

リスト 9. A* の `search()` メソッドの Java ソース

```
public SearchResult<T> search(T init) {
 Node initNode = new Node(init, null, 0, 0 -1);
 open.add(initNode);
 while (!open.isEmpty() && path.isEmpty()) {
   Node n = open.poll();
   if (closed.containsKey(n.state)) continue;
   if (domain.isGoal(n.state)) {
```

```
      for (Node p = n; p != null; p = p.parent)
        path.add(p.state);
      break;
    }
    closed.put(n.state, n);
    for (int i = 0; i < domain.numActions(n.state); i++) {
      int op = domain.nthAction(n.state, i);
      if (op == n.pop) continue;
      T successor = domain.copy(n.state);
      Edge<T> edge = domain.apply(successor, op);
      Node node = new Node(successor, n, edge.cost, edge.pop);
      open.add(node);
    }
  }
  return new SearchResult<T>(path, expanded, generated);
}
```

この A* 実装を評価するには、この実装の実行対象となる問題が必要です。次のセクションでは、ヒューリスティック探索アルゴリズムを評価するのによく使用されているドメインについて説明します。このドメインでは、すべてのアクションに伴うコストは同じであり、使用するヒューリスティックは許容的であるため、この単純化した実装で十分に必要を満たします。

15 パズルのベンチマーク

この記事の目的のために焦点を当てるのは、15 パズルというゲームのドメインです。この単純なドメインには十分に解明されている性質があり、ヒューリスティック探索アルゴリズムを評価するための標準的なベンチマークとなります (これらのパズルを AI 研究の「ミバエ」と呼んでいる人もいます (訳注: 「ミバエ」は遺伝研究の発展に大きく寄与したハエ目ミバエ科に属するハエの総称であり、ここでは AI 研究の発展に大きく寄与したパズルをミバエになぞらえています))。15 パズルはタイルをスライドさせるタイプのパズルで、4x4 のグリッド上に 15 枚のタイルが配置されます。タイルを配置できる場所は 16 箇所あり、そのうち 1 箇所が常に空き状態となります。空き状態となっている場所の隣にあるタイルは、スライドさせることができます。目指すところは、パズルが目標の配置になるまで、タイルをスライドさせることです。図 4 に、タイルがランダムに配置された状態のパズルを示します。

図 4. ランダムに配置された 15 パズル

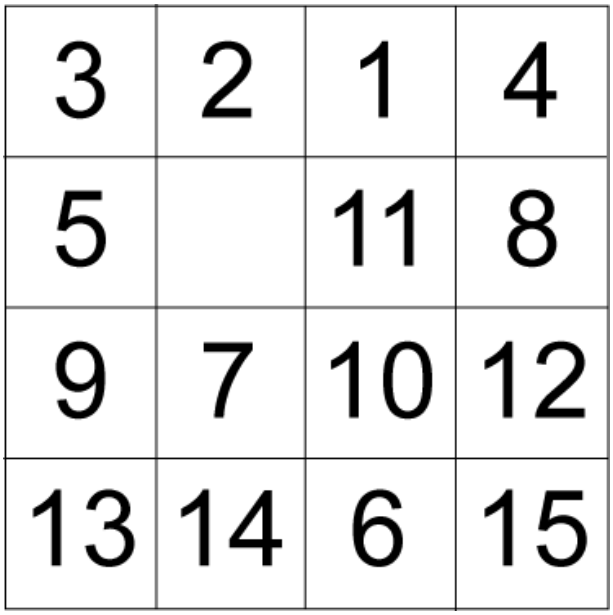
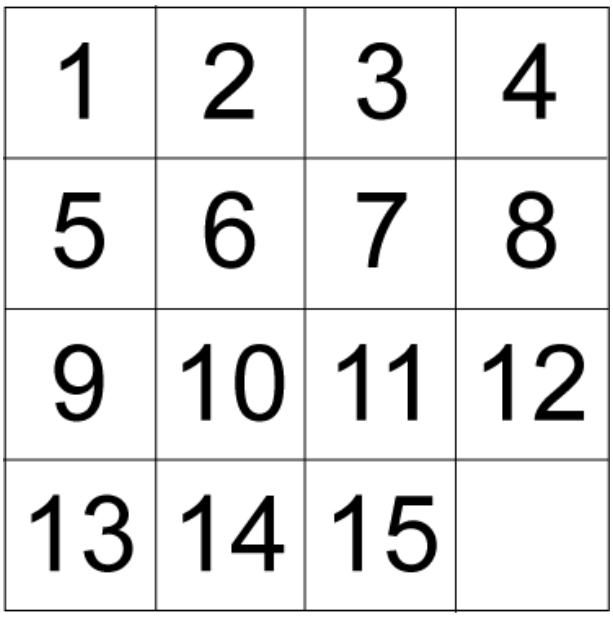


図 5 に、目標の配置になったタイルを示します。

図 5. 目標の配置になった 15 パズル



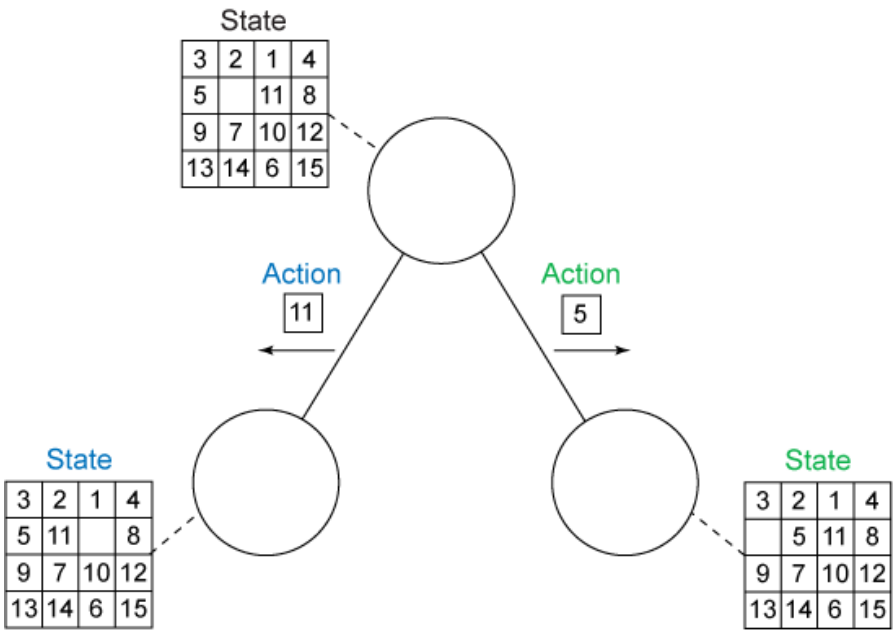
ヒューリスティック探索ベンチマークとして、私たちはこのパズルを何らかの初期の配置から可能な限り少ない移動数で目標の配置にします。

このドメインに使用するヒューリスティックは、「マンハッタン距離」ヒューリスティックと呼ばれます。タイルのマンハッタン距離は、そのタイルがその目標位置に到達するまでに必要な縦および横の移動の数です。状態のヒューリスティックを計算するために、パズル内のすべてのタイル (空き状態の部分は含みません) のマンハッタン距離の合計を出します。任意の状態に関するすべてのマンハッタン距離の合計は、パズルの目標の状態に到達するコストの下限となります。

それは、それよりも少ない移動数で、それぞれのタイルをその目標の配置に動かすことは不可能だからです。

最初は直感的ではないように思えるかもしれませんが、15 パズルは、タイルに可能な配置を節点として表すグラフによってモデル化することができます。ある配置を別の配置に変換する単一のアクションがある場合、その 2 つの節点を枝で結びます。このドメインでのアクションは、タイルを空き状態の部分にスライドさせることです。図 4 に、この探索グラフを示します。

図 6. 15 パズルの状態空間探索グラフ



グリッド上に 15 のタイルを配置する方法は 16! 通り考えられますが、15 パズルで実現可能な配置または状態は、実際には「わずか」 16!/2 = 10,461,394,944,000 だけです。これは、パズルの物理的な制約事項により、実現可能な配置は、考えられるすべての配置のちょうど半分となるからです。この状態空間の大きさを理解するには、ある状態を 1 バイトで表現できると想定すると (実際には、これは不可能です)、状態空間全体を格納するには、10TB を超えるメモリーが必要になります。この値は、最近のほとんどのコンピューターで扱えるメモリーの上限を超えています。そこでこの後、ヒューリスティック探索ではこの状態空間のほんの一部にアクセスするだけで、パズルを最適に解決できる仕組みを説明します。

実験の実行

私たちが行う実験では、Korf 100 セットと呼ばれる、15 パズル開始時の配置として有名な一連の配置を使用します。Korf 100 セットは、IDA* と呼ばれるA* の反復深化バージョンを使用すると、15 パズルのランダムな配置を解決できることを示す初めての実験結果を公開した Richard E. Korf 氏の名前にちなんで名付けられています。この結果が公開されてからは、その後のヒューリスティック探索での数え切れないほどの実験では、Korf 氏が彼の実験で使用した 100 のランダム・インスタンスが再使用されています。私たちは、反復深化手法が必要にならないようにするためにも、この実装を最適化します。

開始時の配置は、一度に 1 つずつ解決します。各開始構成は、それぞれに別個のプレーンテキスト・ファイルに格納します。このファイルの場所をコマンドライン引数に指定して、実験を開始します。コマンドライン引数を処理して問題インスタンスを生成し、A* 探索を実行する Java プログラムへのエントリー・ポイントが必要です。このエントリー・ポイント・クラスには `TileSolver` という名前を付けます。

探索のパフォーマンスに関する統計は、各実行の終了時に、標準出力に印刷します。この統計のなかで最も関心があるのは、実測時間です。すべての実行の実測時間を合計して、このベンチベークの合計実行時間を出します。

この記事のソース·コードに、実験を自動化するための Ant タスクが含まれています。以下のコマンドで、完全な実験を実行することができます。

```
ant TileSolver.korf100 -Dalgorithm="efficient"
```

`algorithm` (アルゴリズム) には、`efficient` (効率的) または `textbook` (標準的) のいずれかを指定することができます。

単一のインスタンスを実行するための Ant ターゲットも用意してあります。以下はその一例です。

```
ant TileSolver -Dinstance="12" -Dalgorithm="textbook"
```

さらに、3 つの最も難しいインスタンスを除外したベンチマークのサブセットを実行するための Ant ターゲットもあります。

```
ant TileSolver.korf97 -Dalgorithm="textbook"
```

お使いのコンピューターには、標準的な実装を使用して完全な実験を完了するだけのメモリーがない可能性が考えられます。スワッピングを回避するには、Java プロセスに使用できるメモリー量を慎重に制限してください。この実験を Linux で実行する場合は、`ulimit` などのシェル・コマンドを使用して、アクティブ·シェルのメモリー制限を設定することができます。

最初は成功しません

表 1 に、私たちが使用したすべての手法の結果を記載します。標準的な A* 実装の結果は、最初の行に示されています (圧縮状態と HPPC、およびそれぞれの結果については、以降のセクションで説明します)。

表 1. 3 つの A* のバリエーションで、15 パズルの Korf 100 セットの 97 インスタンスを解決したときのベンチマークの結果

アルゴリズム	最大メモリー使用量	合計実行時間
標準的	25GB	1,846 sec
圧縮状態	11GB	1,628 sec
HPPC	7GB	1,084 sec

標準的な実装では、すべてのテスト・インスタンスを解決することはできませんでした。最も難しい 3 つのインスタンスの解決には失敗し、解決できたインスタンスについては実行時間が 1,800 秒を超えています。C/C++ での最適な実装では、これらの 100 のインスタンスのすべてを 600 秒未満で解決できることを考えると、これは最良の結果ではありません。

最も難しい 3 つのインスタンスを解決できなかった理由は、メモリーの制約にあります。探索の繰り返しごとに、オープン・リストから節点を取り除いて、その節点を展開すると、通常は複数の追加節点が生成されることになります。生成される節点の数が増えるにつれ、これらの節点をオープン・リストに格納するために必要なメモリーの量も増加します。ただし、このようなメモリーの需要は、Java 実装に特有のものではありません。C/C++ での同等の実装でも同じく失敗します。

Burns 氏らによる論文 (「参考文献」を参照) では、A* 探索の効率的な C/C++ 実装は、このベンチマークを 30GB 未満のメモリーで解決可能であることを明らかにしています。このことから、私たちは A* の Java 実装を断念するにはまだ至りませんでした。以下のセクションで説明するように、メモリーの使用をもっと効率化するために適用できる手法は他にもあります。それによる結果は、ベンチマーク全体を素早く解決できる A* の効率的な Java 実装です。

状態の圧縮

VisualVM などのプロファイラーを使用して、この A* 探索のメモリー使用状況を調べると、すべてのメモリーは `Node` クラスによって使用されており、より直接的には `TileState` クラスによって使用されていることがわかります。したがって、メモリー使用量を削減するには、これらのクラスの実装を再検討する必要があります。

すべてのタイルの状態には、15 のタイルすべての位置を格納しなければなりません。そのために、15 の整数からなる配列に各タイルの位置を格納しています。これらの位置を 64 ビットの整数 (Java の `long`) に圧縮すれば、位置をより簡潔に表現することができます。オープン・リストに節点を格納する必要がある場合には、状態のこの圧縮表現のみを格納することが可能です。これにより、それぞれの節点につき、52 バイトの節約になります。ベンチマークの最も難しいインスタンスを解決するには、約 5.33 億の節点を格納しなければなりませんが、状態表現を圧縮することで、25GB を超えるメモリーの節約になります。

この実装の全般的特性を維持するには、`SearchDomain` インターフェースを継承して、状態を圧縮および圧縮解除するためのメソッドを持たせる必要があります。オープン・リストに節点を格納する前に、状態の圧縮表現を生成して、`Node` クラスには状態へのポインターの代わりにこの圧縮表現を格納することにします。節点の後続節点を生成する必要があるときには、単に状態を圧縮解除します。リスト 10 に、`pack()` メソッドの実装を記載します。

リスト 10. `pack()` メソッドの Java ソース

```
public long pack(TileState s) {
  long word = 0;
  s.tiles[s.blank] = 0;
  for (int i = 0; i < Ntiles; i++)
    word = (word << 4) | s.tiles[i];
  return word;
}
```

リスト 11 に、`unpack()` メソッドの実装を記載します。

リスト 11. `unpack()` メソッドの Java ソース

```
public void unpack(long packed, TileState state) {
  state.h = 0;
  state.blank = -1;
  for (int i = numTiles - 1; i >= 0; i--) {
    int t = (int) packed & 0xF;
    packed >>= 4;
    state.tiles[i] = t;
    if (t == 0)
      state.blank = i;
    else
      state.h += md[t][i];
  }
}
```

圧縮表現は状態の正準形式であるため、クローズド・リストには圧縮表現を格納することができます。`HashMap` クラスにプリミティブ型をそのまま格納することはできないため、プリミティブ型は `Long` クラスのインスタンスでラップしてから格納する必要があります。

表 1 の 2 行目には、状態の圧縮表現を使用して実験を実行した結果が示されています。圧縮された状態表現を使用することで、メモリー使用量が 55 パーセント削減され、実行時間も多少短縮されていますが、それでもまだ、ベンチマーク全体を解決することはできません。

JCF (Java Collections Framework) に伴う問題

圧縮された状態表現のすべてを `Long` のインスタンスでラップすると、かなりのオーバーヘッドになると考えている場合、それは当たっています。これはメモリーの浪費であり、過度のガーベッジ・コレクションを引き起こす可能性があります。JDK 1.5 で追加している「autoboxing」のサポートは、プリミティブ型の値からオブジェクト表現への変換 (`long` から `Long` への変換) と、その逆の変換を自動的に行います。大規模なコレクションの場合、これらの変換によってメモリー・アクセスの回数や CPU の使用が増えることで、パフォーマンスが低下する可能性があります。

JDK 1.5 では Java Generics も導入しています。これは、C++ のテンプレートとよく比較される機能です。Burns 氏らは、ヒューリスティック探索の実装では、C++ のテンプレート機能がかなりのパフォーマンス上のメリットをもたらすことを明らかにしています。Generics には、そのようなメリットがありません。Generics の実装に使用されている型消去は、コンパイル時にすべての型情報を削除 (消去) します。その結果、実行時に型情報をチェックしなければならないため、大規模なコレクションの場合にはパフォーマンスの問題をもたらす可能性があります。

メモリーに関する詳細

Java でのメモリー使用量について、多くの JCF (Java Collections Framework) クラスに重点を置いて調べるには、Chris Bailey 氏の優れた developerWorks 記事「Java コードから Java ヒープまで」を読むことをお勧めします。

`HashMap` クラスを実装すると、ある程度のメモリー・オーバーヘッドが追加されることが明らかになります。`HashMap` は、内部 `HashMap$Entry` クラスのインスタンスからなる配列を格納します。要素を `HashMap` に追加するたびに、新しいエントリーが作成されて、この配列に追加されます。このエントリー・クラスの実装には、一般に 3 つのオブジェクト参照と 1 つの 32 ビットの

整数参照が含まれます。つまり、エントリーごとの合計は 32 バイトです。クローズド・リストに 5.33 億の節点を格納するとなると、オーバーヘッドは 15GB を超えてしまいます。

そこで、今度は `HashMap` クラスに代わる手段を導入し、この代替手段でプリミティブ型を直接格納することで、さらなるメモリー使用量の削減を可能にします。

HPPC (High Performance Primitive Collections)

現時点でプリミティブ型を格納しているのはクローズド・リストだけなので、HPPC (High Performance Primitive Collections) を利用することができます。HPPCは、プリミティブ型の値を直接格納して JCF のオーバーヘッドを完全に排除できる、代替コレクション・フレームワークです (「参考文献」を参照)。Java の Generics とは対照的に、HPPC は C++ のテンプレートと同様の手法を使用して各コレクション・クラスの実装を分離し、コンパイル時に Java プリミティブ型が生成されるようにします。したがって、プリミティブ値をコレクションに格納する際に、`Long` や `Integer` などのクラスでラップする必要はありません。さらに副次効果として、JCF では必要な多くのキャストを回避することができます。

プリミティブ値を格納するための JCF に代わる手段は他にもあります。その好例としては Apache Commons Primitive Collections と fastutil の 2 つが挙げられますが、私たちの考えでは、HPPC の設計にはハイパフォーマンス・アルゴリズムを実装する上での大きな利点が 1 つあります。それは、HPPC ではコレクション・クラスごとに内部データ·ストレージを公開することです。このストレージに直接アクセスすることで、さまざまな最適化が可能になります。例えば、オープン・リストまたはクローズド・リストをディスクに保管するとしたら、イテレーターを使用して間接的にデータにアクセスするのではなく、ベースとなるデータ配列に直接アクセスできるようにすることで、より効率化することが可能です。

A* 実装は、クローズド・リストに `LongOpenHashSet` クラスのインスタンスを使用するように変更することができます。そのために必要となる変更は、極めて単純です。プリミティブ値だけを格納することから、状態を実装するクラスの `hashcode` メソッドと `equals` メソッドをオーバーライドする必要はもうありません。クローズド・リストは 1 つのセットであるため (重複する要素は格納しません)、格納する必要があるのは、キーと値のペアではなく、値のみとなります。

表 1 の 3 行目に、JCF の代わりに HPPC を使用して実験を実行した結果が示されています。HPPC を使用した場合、メモリー使用量は 27 パーセント削減され、実行時間は 33 パーセント短縮されました。

メモリー使用量は合わせて 82 パーセント削減されたため、メモリー制約の範囲内でベンチマーク全体を解決できるようになりました。表 2 の最初の行に、結果を記載します。

表 2. 3 つの A* のバリエーションで Korf 100 セットの 100 インスタンスを解決したときのベンチマークの結果

アルゴリズム	最大メモリー使用量	合計実行時間
HPPC	30GB	1,892 sec
ネストされたバケット・キュー	30GB	1,090 sec
ガーベッジ・コレクションの回避	30GB	925 sec

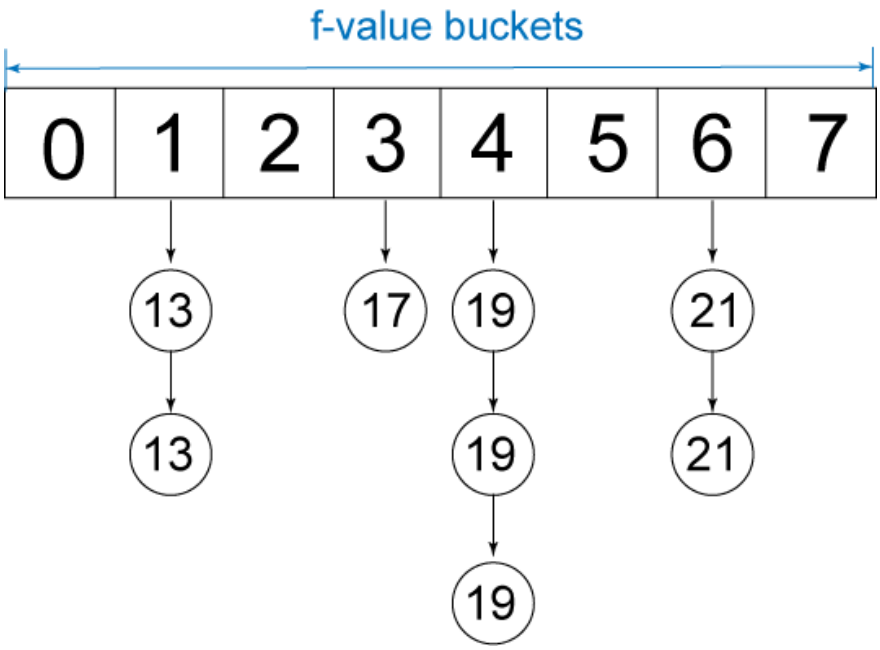
HPPC を使用すると、30GB のメモリーで 100 インスタンスをすべて解決することができますが、それには 1,800 秒以上の時間がかかります。表 2 の他の結果には、もう 1 つの重要なデータ構造であるオープン・リストの改善によって実装を高速化した結果が反映されています。

`PriorityQueue` に伴う問題

要素をオープン・リストに追加するたびに、キューをソートしなおさなければなりません。`PriorityQueue` は、エンキューおよびデキューの処理に O(log(n)) 時間を要します。これは、ソートに関しては効率的ですが、コストがかかることは確かです。そのコストは、特に n の値が大きい場合には顕著になります。最も難しい問題のインスタンスには、5 億を超える節点をオープン・リストに追加することを思い出してください。しかも、このベンチマークの問題におけるすべてのアクションのコストは同じであることから、可能な f 値の範囲はわずかです。したがって、`PriorityQueue` を使用するメリットは、それによるオーバーヘッドを上回るほどのものではありません。

1 つの代替手段は、バケット・ベースの優先度付きキューを使用することです。このドメインのアクションのコストは、狭い値の範囲に収まることから、f 値につき 1 つのバケットという固定した範囲のバケットを定義することができます。節点を生成する際には、その節点に対応する f 値を持つバケットに節点を入れるだけです。キューの先頭にアクセスする必要がある場合は、最小の f 値を持つバケットから順に、節点が見つかるまでバケットを調べます。このタイプのデータ構造 (「1 レベル・バケット優先度付きキュー (1-level bucket priority queue)」と呼ばれます) により、エンキュー処理とデキュー処理を一定時間で行うことが可能になります。図 7 に、このデータ構造を示します。

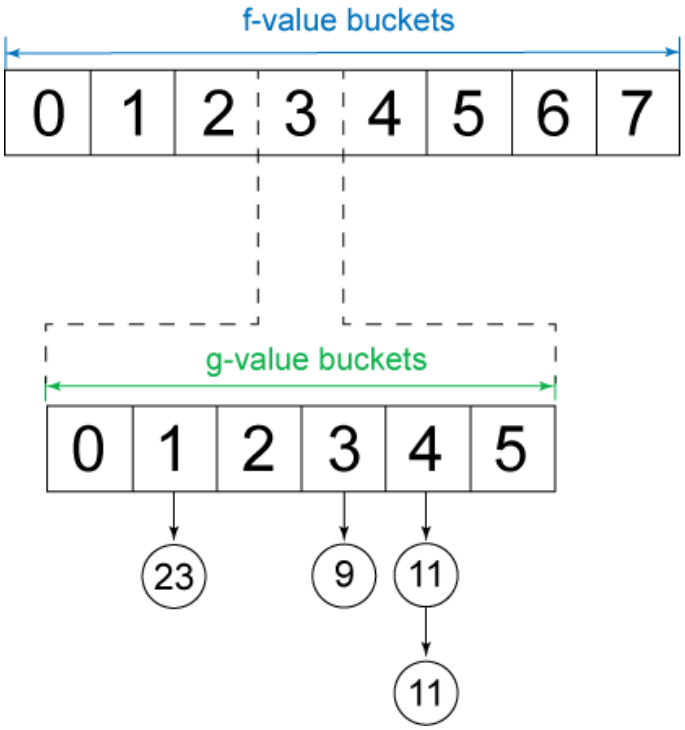
図 7. 1 レベル・バケット優先度付きキュー



鋭い読者は、ここで説明した 1 レベル・バケット優先度付きキューを実装すると、g 値を使用して節点間で順序付けを行うことができなくなることにお気付きでしょう。このようにして順序付けを行うことが価値のある最適化であることは、先ほど理由を十分に理解しているはずです。こ

の最適化を維持するには、ネストされたバケット優先度付きキューを実装するという方法があります。この方法では、バケットの 1 つのレベルを使用して f 値の範囲を表し、ネストされたレベルを使用して g 値の範囲を表します。図 8 に、このデータ構造を示します。

図 8. ネストされたバケット優先度付きキュー



これで、オープン・リストに対してネストされたバケット優先度付きキューを使用するように、A* 実装を更新することができます。ネストされたバケット優先度付きキューの完全な実装は、この記事のソース・コード (「ダウンロード」を参照) に含まれている BucketHeap.java ファイルで確認することができます。

表 2 の 2 行目に、ネストされたバケット優先度付きキューを使用した実験の実行結果が示されています。`PriorityQueue` の代わりにネストされたバケット優先度付きキューを使用することで、実行時間は 58 パーセントほど改善されましたが、それでもまだ、所要時間は 1,000 秒です。実行時間を改善するには、もう 1 つの単純な方法があります。

ガーベッジ・コレクションの回避

Java では多くの場合、ガーベッジ・コレクションがボトルネックとして見なされます。ここでの内容に当てはまる、JVM でのガーベッジ・コレクションのチューニングに関しては、数々の優れた記事が公開されているので (「参考文献」を参照)、それについて詳しく掘り下げることはしません。

A* は通常、多くの短時間存続する状態オブジェクトと枝オブジェクトを生成するため、コストのかかるガーベッジ・コレクションが大量に発生します。必要なガーベッジ・コレクションの量を抑える方法は、オブジェクトを再利用することです。そのためには、いくつかの単純な変更を加えることができます。現状では、A* 探索ループを繰り返すごとに、(最も難しい問題では 5.33 億

の節点に) 新しい枝と新しい状態を割り振っています。このように毎回新しいオブジェクトを割り振る代わりに、ループのすべての繰り返し処理で常に同じ状態オブジェクトおよび枝オブジェクトを再利用することが可能です。

枝オブジェクトと状態オブジェクトを再利用するには、`Domain` インターフェースを変更する必要があります。`apply()` メソッドで `Edge` のインスタンスを返す代わりに、`apply()` 呼び出しによって変更される独自のインスタンスを提供しなければなりません。`edge` に対する変更は漸進的ではないため、`apply()` に渡す前の `edge` に格納される値について懸念する必要はありません。一方、`apply()` が `state` オブジェクトに対して加える変更は、漸進的です。状態をコピーすることなく、存在し得るすべての後続状態を適切に生成するには、行われた変更を取り消す手段が必要になります。そのためには、`Domain` インターフェースを継承して `undo()` メソッドを持たせる必要があります。リスト 12 に、`Domain` インターフェースに加える変更を記載します。

リスト 12. 更新後の `Domain` インターフェース

```
public interface Domain<T> {
  ...
   public void apply(T state, Edge<T> edge, int op);
   public void undo(T state, Edge<T> edge);
  ...
}
```

最終的な実験の結果は、表 2 の 3 行目に記載されています。状態オブジェクトおよび枝オブジェクトを再利用することによって、私たちはコストの高いガーベッジ・コレクションを回避し、実行時間を 15 パーセント以上短縮しています。この極めて効率的な A* の Java 実装では、ちょうど 30GB のメモリーを使用してベンチマーク・セット全体を 925 秒で解決することができます。C/C++ による最適な実装での実行時間は 540 秒で、27GB のメモリーが必要であることを考えると、これは素晴らしい結果です。この Java 実装の場合、速度はわずか 1.7 倍遅くなるだけで、必要なメモリー量はほとんど変わりません。

まとめ

ヒューリスティック探索について紹介したこの記事では、A* アルゴリズムについて概説し、Java でのその標準的な実装についても説明しました。そして、この実装にはパフォーマンス上の問題があり、かなりの時間的制約またはメモリーの制約があることから、標準的なベンチマーク問題を解決できないことを明らかにしました。これらの問題に対処するために取った方法は、HPPC といくつかの手法を用いてメモリー使用量を削減し、コストのかかるガーベッジ・コレクションを回避する方法です。このように改善した後の実装では、わずかな時間およびメモリーの制約でベンチマーク問題を解決できるようになり、ヒューリスティック探索アルゴリズムを実装する上で Java が素晴らしい選択肢であることが実証されました。さらに、この記事で紹介した手法は、多くの実際の Java アプリケーションにも適用することができます。例えば、大量のプリミティブ値を保管する Java アプリケーションでは、HPPC を使用するとパフォーマンスがたちまち改善される場合があります。

謝辞

NSF からの支援 (0812141 および 1150068)、DARPA の支援 (N10AP20029)、およびニューハンプシャー大学の Dissertation Year Fellowship に深く感謝いたします。

ダウンロード

内容	ファイル名	サイズ
Sample code	j-ai-code.zip	58KB

著者について

Matthew Hatem

Matthew Hatem は、ニューハンプシャー大学の博士論文提出資格者です。彼の研究には、並列および外部メモリーを使用したヒューリスティック探索のアルゴリズムおよびアプリケーションがあり、その研究は Proceedings of the Association for the Advancement of Artificial Intelligence and of the Symposium on Combinatorial Search で発表されています。彼は IBM のシニア・ソフトウェア・エンジニアでもあり、現在は Watson Solutions チームの一員となっています。以前は、さまざまな Lotus 製品に取り組み、eclipse.org のコミッターでもありました。

Ethan Burns

Ethan Burns は、Google のソフトウェア・エンジニアであり、ニューハンプシャー大学で博士号を取得しています。彼が取り組んだプロジェクトは、iSCSI Linux カーネル・モジュール、SPIN モデル・チェッカーの並列化、並列ヒューリスティック探索と自動プランニング、ロボティクスのプランニング手法、時間制約下でのヒューリスティック探索のアルゴリズムなど多岐に渡ります。2007年に Richard Lyczak 記念教員賞、2012年に Symposium on Combinatorial Search のプログラム委員会最優秀賞を受賞し、2012年から 2013年はニューハンプシャー大学のDissertation Year Fellowship を受けています。

Wheeler Ruml

Wheeler Ruml は、ニューハンプシャー大学でコンピューター・サイエンスの准教授を務め、同大学で UNH Artificial Intelligence Group の指導をしています。彼は International Symposium on Combinatorial Search の共同設立者であり、2014年の ICAPS Conference では共同議長を務める予定です。NSF CAREER 賞である DARPA Computer Science Study Panel と、UNH Outstanding Assistant Professor Award に選ばれた経験もあります。ニューハンプシャー大学に勤める前は、AI 手法を使用して世界最速のプリンターを作成した Xerox PARC のチーム・リーダーとして活躍しました。彼は 2002年にハーバード大学で博士号を取得しています。