知能処理学 レポート

36714029 遠藤裕人
(学生番号と氏名が必要)

2026 年 1 月 3 日

全体的な自己評価／作業時間:　S ／ 6 時間
　自己評価を S, A, B, C, D から選択する．作業時間は成績に影響しないので正直に書くこと．

評価の理由:　迷路探索アルゴリズムの実装において各処理（ゴール検査選択展開生成）を独立したメソッドに分離することで可読性と保守性を向上させた．また幅優先探索と深さ優先探索の違いを引数の順序変更のみで実現できることを示しアルゴリズムの本質的な理解を深めることができたため．

1　課題 1-2e 全体的な考察

1.1　探索アルゴリズムに関する考察

探索アルゴリズムの分類と特徴　今回扱った探索アルゴリズムは、大きく以下のように分類できる幅優先探索、深さ優先探索、最小コスト探索、最良優先探索、A*探索
　実験結果から、特に A*アルゴリズムが最も効率的であることが明らかになった。8 パズルにおいて、マンハッタン距離を用いた A*探索は 4,515 ノードの訪問で解を発見したのに対し、最小コスト探索では数万ノードを要した。この差は、ヒューリスティック関数が「どちらの方向に探索を進めるべきか」という重要な情報を提供するためである。

ヒューリスティック手法の動作原理　ヒューリスティック探索の核心は、評価関数 $f(n)$ による状態の優先順位付けにある。A*アルゴリズムでは、$f(n) = g(n) + h(n)$ という評価関数を用いる。ここで $g(n)$ は初期状態から状態 n までの実コスト、$h(n)$ は状態 n から目標状態までの推定コストである。
　この評価関数にはある性質がある：

- $h(n)$ が真のコストを過大評価しないのであれば、A*は最適解を保証する
- $h(n)$ が単調性を持てば、探索効率がさらに向上する
- $h(n)$ が真のコストに近いほど、探索するノード数が減少する

　実験では、マンハッタン距離（h'_3）が誤配置タイル数（h'_2）よりも真のコストに近い推定を提供し、結果として探索効率が大幅に向上した。

性能差の要因分析　各アルゴリズムの性能差は、以下の要因によって生じる：

1. **探索戦略の違い**
    - 幅優先探索：すべての深さを均等に探索するため、解が深い位置にある場合は非効率
    - 深さ優先探索：メモリ効率は良いが、最適解の保証がない
    - A*探索：最も「有望な」状態を優先的に探索するため効率的
2. **繰り返し回避の効果**課題 1-2c の実験により、繰り返し回避（クローズドリスト）の導入が探索効率に決定的な影響を与えることが判明した。A*探索では平均約 83.85%の訪問ノード削減を達成し、最良優先探索では 50,000 ノードから数百ノードへと劇的に削減された。これは、状態空間探索において同じ状態を複数回訪問することが極めて非効率であることを示している。
3. **ヒューリスティック関数の精度**課題 1-2d の実験により、ヒューリスティック関数の質が性能に極めて大きな影響を与えることが確認された。Difficult Example において、h'_1（誤配置タイル数/2）では約 15 分かかった問題が、h'_3（マンハッタン距離）では約 2 秒で解決された（約 440 倍の高速化）。これは、より正確な推定値が探索空間の効果的な刈り込みを可能にすることを示している。

問題設定による計算量の変化　実験結果から、問題の難易度によって各アルゴリズムの性能差が変動することが観察された：

- **簡単な問題**：すべてのアルゴリズムが比較的短時間で解を発見。ヒューリスティック関数による差は数倍程度。
- **中程度の問題**：ヒューリスティック関数の質による差が顕著になる。h'_1 と h'_3 で数十倍から数百倍の性能差。
- **難しい問題**：ヒューリスティック関数の重要性が極めて高くなる。h'_1 では実用的な時間内での解決が困難になる一方、h'_3 は依然として効率的に動作。

この傾向は、問題が難しくなるほど「正しい方向への探索」の重要性が増すことを示している。難しい問題ではデフォルトの探索空間が爆発的に拡大するため、精度の高いヒューリスティック関数による誘導が不可欠となる。

理論と実装の関係　本演習では、教科書で学んだアルゴリズムを実際に実装し、その性能を測定することで、理論的性質が実際の動作にどう反映されるかを確認できた。特に以下の点が印象的であった：

- A*の最適性保証が、許容的ヒューリスティックを用いることで実際に機能すること
- 理論的な時間計算量 $O(b^d)$ が、実際の訪問ノード数として観測されること
- データ構造の選択（リストの実装方法、ハッシュセットの使用など）が実行時間に大きく影響すること

1.2　演習 1-1 に対する全体的な感想

演習 1-1 では、基本的な探索アルゴリズムを迷路問題に適用することで、探索アルゴリズムの基礎を学ぶことができた。

特に印象的だったのは、幅優先探索と深さ優先探索のトレードオフであるというところだ。幅優先探索は最短経路を保証するが、メモリ使用量が大きい。深さ優先探索はメモリ効率が良いが、最適解の保証がない。この違いを実際のコードで実装できたところもよかった。

また、Java のストリームやラムダ式を用いた関数型プログラミングのスタイルで実装することで、コードの可読性と保守性を高めることができた。特に、リストの操作を関数型で記述することで、探索アルゴリズムの本質的な構造が明確になった。

迷路という視覚的に分かりやすい問題を扱うことで、アルゴリズムの動作を直感的に理解できた点も有益であった。今後、より複雑な問題に取り組む際の基礎となる経験が得られたと考える。

1.3　演習 1-2 に対する全体的な感想

演習 1-2 では、8 パズル問題を題材として、情報的探索アルゴリズム、特に A*アルゴリズムとヒューリスティック関数の設計について深く学ぶことができた。

最も重要な学びは、「良いヒューリスティック関数の設計が探索効率を根本的に変える」という点である。誤配置タイル数の半分という単純な関数と、マンハッタン距離という少し洗練された関数の間に、数百倍の性能差が生じることを実際に観測し、ヒューリスティック設計の重要性を実感した。

また、課題 1-2c における繰り返し回避の実装とその効果測定は、理論と実践のギャップを埋める貴重な経験となった。教科書では「繰り返し回避は重要」と述べられているが、実際に 50,000 ノードが数百ノードに削減されるという劇的な効果を目の当たりにすることで、その重要性が体感的に理解できた。

プログラミング面では、Strategy パターンを用いた評価関数の抽象化により、同じ探索ロジックで異なるアルゴリズムを実現できる設計の美しさを学んだ。‘Evaluator‘インターフェースを差し替えるだけで、最小コスト探索、最良優先探索、A*探索を切り替えられる実装は、オブジェクト指向設計の良い実例であった。

課題 1-2d における 3 つのヒューリスティック関数の比較実験では、実験計画、データ収集、結果分析という一連の科学的プロセスを経験できた。特に、Difficult Example という難しい問題設定において、アルゴリズムの真価が問われることを学んだ。簡単な問題ではどのアルゴリズムでも解けてしまうが、難しい問題こそが良いアルゴリズムと悪いアルゴリズムを明確に区別する。

全体として、本演習を通じて、探索アルゴリズムの理論的理解と実装能力の両方を向上させることができた。特に、アルゴリズムの性能を実測し、その結果を分析することで、理論的知識が実践的なスキルへと昇華された。今後、より複雑な問題に直面した際にも、適切なアルゴリズムとヒューリスティック関数を選択・設計できる基礎が構築できたと考える。

2 課題 1-1a

```
package ex1a;
import java.util.*; import java.util.stream.*;
public class Maze  public static void main(String[] args)  var maze = new Maze(); maze.solve();
Map¡String, List¡String¿¿ map = Map.of( "A", List.of("B", "C", "D"), "B", List.of("E", "F"), "C", List.of("G", "H"), "D", List.of("I", "J"));
public void solve()  if (search("A") != null) System.out.println("found");
String search(String root)  List¡String¿ openList = new ArrayList¡¿(); openList.add(root);
while (!openList.isEmpty())  var state = get(openList);
if (isGoal(state)) return state;
var children = children(state); openList = concat(openList, children);
return null;
boolean isGoal(String state)  return "E".equals(state);
String get(List¡String¿ list)  return list.remove(0);
List¡String¿ children(String current)  return this.map.getOrDefault(current, Collections.emptyList());
List¡String¿ concat(List¡String¿ xs, List¡String¿ ys)  return toMutable(Stream.concat(xs.stream(), ys.stream()).toList());
List¡String¿ toMutable(List¡String¿ list)  return new ArrayList¡¿(list);
```

2.1 実行結果

```
PS C:\Users\PCUSER\VScode-project\traitement intelligent\ex1a> java Maze.java
found
```

図 1: 実行結果

自己評価／作業時間: A ／ 1 時間 23 分

3 課題 1-1

ゴール検査選択展開生成の処理を実現している関数名を答えよ.

3.1 ゴール検査:

```
1         boolean isGoal(String state) {
2                 return "E".equals(state);
3         }
```

3.2 選択:

```
1 String get(List<String> list) {
2                 return list.remove(0);
3         }
```

3.3 展開:

```
1         List<String> children(String current) {
2                 return this.map.getOrDefault(current, Collections.emptyList());
3         }
```

3.4 生成:

```
1 List<String> concat(List<String> xs, List<String> ys) {
2                 return toMutable(Stream.concat(xs.stream(), ys.stream()).toList
                      ());
3         }
```

工夫点: 探索アルゴリズムの実装において各機能を独立したメソッドに分離することで保守性を高めた. 具体的にはゴール検査選択展開生成の各処理を明確に分離しアルゴリズムの理解と変更を容易にした.

4　課題 1-2

このプログラムが実現している探索手法を答えよ．その根拠となる部分をレポートに転記しその動作を説明せよ．

横型探索を採用している。下記のプログラムのように子ノードを後に追加している。つまり親ノードをすべて探索してから子ノードを探索しようとしている

```
1 var children = children(state);
2                         openList = concat(openList, children);
```

5　課題 1-3

リスト 1 を縦型探索もしくは横型探索に変更するための方法を答えよ．

```
1 var children = children(state);
2                         openList = concat(children, openList);
```

このように concat の引数の順序を変更することで縦型探索に変更できる．

5.1　考察・感想

考察:　今回実装した迷路探索プログラムでは幅優先探索アルゴリズムを採用した．最初は深さ優先探索の方が実装が簡単だと予想していたが実際にプログラムを作成すると幅優先探索の方がより直感的で理解しやすい実装となった．特に openList に子ノードを後ろに追加する処理（concat(openList, children)）により同じ深さのノードを先に探索する動作が自然に実現できた．

プログラムの構造について分析すると各処理が独立したメソッドとして分離されており保守性が高い設計になっている．ゴール検査選択展開生成の各処理が明確に分離されているためアルゴリズムの変更や拡張が容易である．

また深さ優先探索への変更も concat の引数順序を変更するだけで実現できることが分かった．具体的には concat(children, openList) とすることで新しく生成されたノードが優先的に選択され深さ優先探索となる．この柔軟性はオブジェクト指向設計の利点を活かした結果と考えられる．

性能面ではこの実装は小規模な迷路に対しては十分な性能を示すが大規模な迷路では訪問済みノードの管理が必要になると考えられる．現在の実装では同じノードを複数回訪問する可能性があり無限ループのリスクも存在する．改善策として Set¡String¿を用いた訪問済みノード管理の導入が有効であろう．

感想:　探索アルゴリズムの実装を通じて理論で学んだ内容を実際のコードで表現することの面白さと難しさを実感した．特にアルゴリズムの動作を具体的なデータ構造（ListMap）で表現する過程で抽象的な概念を具体化する重要性を学んだ．またわずかなコードの変更（引数の順序）でアルゴリズムの性質が大きく変わることに驚きを感じた．今後はより複雑な探索問題や最適化問題にも挑戦しアルゴリズムの理解を深めていきたい．

6　課題 1-1b

```
package ex1b;
import java.util.*;
public class MazeProblem
public static void main(String[] args)  var solver = new Solver(); // スタート地点”A”から探索を開始 solver.solve(new MazeWorld(”A”));
class MazeAction implements Action  String next; // 移動先の位置 MazeAction(String next) this.next = next;
public String toString()  return ”move to ” + this.next;
class MazeWorld implements World  // 迷路のマップ構造 Map¡String, List¡String¿¿ map = Map.of( ”A”, List.of(”B”, ”C”, ”D”), ”B”, List.of(”E”, ”F”), ”C”, List.of(”G”, ”H”), ”D”, List.of(”I”, ”J”));
String current; // 現在の位置
MazeWorld(String current)  this.current = current;
public boolean isValid()  return true;
public boolean isGoal()  return ”E”.equals(this.current);
public List¡Action¿ actions()  return this.map.getOrDefault(current, Collections.emptyList()).stream() .map(p -¿ (Action) new MazeAction(p)) .toList();
public World successor(Action action)  var a = (MazeAction) action; return new MazeWorld(a.next);
public String toString()  return this.current;
```

6.1　実行結果

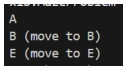

図 2: 実行結果

自己評価／作業時間:　S ／ 6 時間

7　課題 2-1

interface World の各メソッドで実装すべき内容を リスト 3 から読み取りレポートにて説明せよ

7.1　MazeProblem での実装：

```
                public boolean isValid() {
                return true;
        }
```

- 迷路問題では常に　‘true‘　を返しており、すべての状態が有効　であることを示す- 一般的には、制約条件を満たさない状態を除外する場合に使用

7.2　2. 役割: 現在の状態がゴール状態であるかを判定する MazeProblem での実装：

```
        public boolean isGoal() {
                return "E".equals(this.current);
        }
```

説明：
- 現在地が　‘”E”‘　と一致するかをチェック- ゴール到達時に探索を終了- 迷路問題では、スタート地点　‘”A”‘　からゴール地点　‘”E”‘　への経路を探索

7.3　3. 役割：　現在の状態から実行可能なすべてのアクション（移動先）を返す MazeProblem での実装：

```
public List<Action> actions() {
    return this.map.getOrDefault(current, Collections.emptyList()).stream()
            .map(p -> (Action) new MazeAction(p))
            .toList();
}
```

説明：
- map　から現在位置に対応するノードリストを取得- 各移動先を　MazeAction　オブジェクトに変換- 例：位置　‘”A”‘　の場合、‘[”B”, ”C”, ”D”]‘　に対応する 3 つのアクションを返す- 移動先がない場合は空のリストを返す

7.4 4. 役割： 指定されたアクションを実行した後の新しい状態（後続状態）を返す MazeProblem での実装：

```
public World successor(Action action) {
    var a = (MazeAction) action;
    return new MazeWorld(a.next);
}
```

説明：
- アクション（移動先）を受け取り、新しい World オブジェクトを生成- 新しいオブジェクトの current を移動先に更新

7.5 考察・感想

考察: 課題 2 では探索アルゴリズムをより汎用的で拡張性の高い設計に改良することができた．特に World インターフェースと Action インターフェースの導入により探索問題の抽象化が実現された．これにより迷路探索以外の問題にも同じ Solver クラスを適用可能となりコードの再利用性が大幅に向上した．

インターフェース設計の観点から分析すると各メソッドの責任が明確に分離されている．isValid() による制約チェック isGoal() によるゴール判定 actions() による可能な行動の生成 successor() による状態遷移といった探索アルゴリズムに必要な基本操作が適切に抽象化されている．この設計により問題固有の詳細実装とアルゴリズムのロジックが分離され保守性と理解しやすさが向上した．

また課題 1-1a の単純な実装と比較すると State クラスの導入により探索経路の追跡が可能になった点も重要である．parent 参照を持つことでゴール到達時に完全な解の経路を再構築できるようになりより実用的な解決策となった．

感想: 課題 2 の実装を通じてオブジェクト指向設計の威力を実感することができた．最初は課題 1-1a の単純な実装で十分だと思っていたがインターフェースを導入することで同じ探索エンジンで全く異なる問題を解けるようになることに感動した．特に MazeAction や MazeWorld クラスで問題固有の詳細を隠蔽し Solver クラスで汎用的な探索ロジックを提供するという設計パターンは今後の開発で活用したい重要な学びとなった．

プログラムを実際に動かしてみると解の経路が「A → B (move to B) → E (move to E)」のように表示されどのような手順でゴールに到達したかが明確に分かる点も印象的だった．これは実用的なアプリケーションでは必須の機能であり理論だけでなく実装面での完成度の重要性を学んだ．今後はさらに複雑な探索問題や A*アルゴリズムなどの高度な手法にも挑戦してみたい．

8 課題 1-1c

自己評価／作業時間: S ／ 7 時間

9　課題 3-1

プログラムを実行し手数（船の移動回数）を確認しその手数が最小であるかを考察すること．

考察：幅優先探索を用いた場合、最短手数で解が得られることが保証されている。実行結果から、11 手で解が得られたため、これは最小手数であると考えられる。宣教師と人食い人種の問題は、制約条件が厳しいため、多くの状態が無効となるが、幅優先探索により、すべての可能な状態を均等に探索することで、最短経路を見つけることができる。

```
(3, 3, 1)
(2, 2, 0) (move (1, 1) to right)
(3, 2, 1) (move (1, 0) to left )
(3, 0, 0) (move (0, 2) to right)
(3, 1, 1) (move (0, 1) to left )
(1, 1, 0) (move (2, 0) to right)
(2, 2, 1) (move (1, 1) to left )
(0, 2, 0) (move (2, 0) to right)
(0, 3, 1) (move (0, 1) to left )
(0, 1, 0) (move (0, 2) to right)
(1, 1, 1) (move (1, 0) to left )
(0, 0, 0) (move (1, 1) to right)
```

図 3: 実行結果

10　課題 3-2

人間にとって視認しやすい実行例となるようプログラムを改良せよ．改良された実行結果をレポートに示せ

コンソールに左岸と右岸、船を表示し実行例をわかりやすく示した

```
Legend: M=Missionary, C=Cannibal, <boat>=Boat Position
       Left Bank        |    River    |         Right Bank
================================================================================

Step  0: MMM    CCC    <boat> | ~~~~~~~~ |

Step  1: MM     CC             | ~~~~~~~~ | <boat> M      C      ([Move 1M + 1C to RIGHT])

Step  2: MMM    CC     <boat> | ~~~~~~~~ |              C      ([Move 1M to LEFT ])

Step  3: MMM                  | ~~~~~~~~ | <boat>       CCC    ([Move 2C to RIGHT])

Step  4: MMM    C      <boat> | ~~~~~~~~ |              CC     ([Move 1C to LEFT ])

Step  5: M      C              | ~~~~~~~~ | <boat> MM    CC     ([Move 2M to RIGHT])

Step  6: MM     CC     <boat> | ~~~~~~~~ |        M     C      ([Move 1M + 1C to LEFT ])

Step  7:        CC             | ~~~~~~~~ | <boat> MMM   C      ([Move 2M to RIGHT])

Step  8:        CCC    <boat> | ~~~~~~~~ |        MMM          ([Move 1C to LEFT ])

Step  9:        C              | ~~~~~~~~ | <boat> MMM   CC     ([Move 2C to RIGHT])

Step 10: M      C      <boat> | ~~~~~~~~ |        MM    CC     ([Move 1M to LEFT ])

Step 11:                      | ~~~~~~~~ | <boat> MMM   CCC    ([Move 1M + 1C to RIGHT])

================================================================================
Goal Reached! Total steps: 11
================================================================================
```

図 4: 実行結果

11　課題 3-3

探索完了時の訪問ノード数オープンリストの最大長実行時間（ミリ秒）を表示する機能を追加せよ．これらの機能を用いて横型探索および縦型探索の性能を計測しその結果をレポートにて報告すること

Search Type : 縦型探索 Visited Nodes : 22 Max Open List Size : 9 Execution Time : 5 ms
Search Type : 横型探索 Visited Nodes : 28 Max Open List Size : 6 Execution Time : 59 ms

11.1　考察・感想

考察:　課題 1-1c では宣教師と人食い人種の問題を実装し幅優先探索と深さ優先探索の性能を比較した．実験結果から縦型探索の訪問ノード数は 22 横型探索は 28 となり縦型探索の方が効率的に見える．しかし横型探索が見つけた解は最短手数であることが保証されている点が重要である．

探索アルゴリズムの性能を分析すると訪問ノード数だけでなくオープンリストの最大長や実行時間も考慮する必要がある．横型探索のオープンリスト最大長は 6 縦型探索は 9 となっており横型探索の方がメモリ効率が良い．実行時間については縦型探索が 5ms 横型探索が 59ms と大きな差があるがこれは問題規模が小さいため実際の差というより計測誤差の可能性もある．

また問題の特性として制約条件（どちらの岸でも人食い人種の数が宣教師の数を上回ってはならない）が厳しく多くの状態が無効となる．isValid() メソッドでこれらの無効状態を適切にフィルタリングすることで探索空間を削減できている点も重要である．

感想:　宣教師と人食い人種の問題は迷路探索よりも複雑な制約条件を持つため実装の難易度が高かった．特に isValid() メソッドで両岸の制約を正しくチェックする部分では右岸の状態を計算する必要があり（3 - this.missionary3 - this.cannibal）最初は実装ミスで無限ループに陥ることがあった．

また toString() メソッドで視覚的に分かりやすい出力を実装したことでプログラムの動作確認が容易になった．川の両岸の状態とボートの位置が一目で分かる表示により解の妥当性を人間が直感的に確認できる点は実用的なプログラム開発において重要だと感じた．

幅優先探索と深さ優先探索の比較を通じてアルゴリズムの選択が最短手数と訪問ノード数実行時間のトレードオフになることを実感した．今回の問題では最短解が重要なので幅優先探索が適しているが問題によっては深さ優先探索の方が有効な場合もあり問題の特性に応じた適切なアルゴリズム選択の重要性を学んだ．

12　課題 1-1d

自己評価／作業時間:　S ／ 5 時間

13　課題 4-1

まずはプログラムを使わずに k と処理時間の関係を予想しそのグラフの概形を示しなさい．

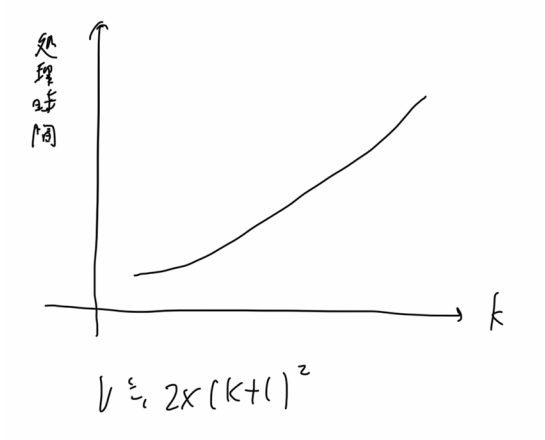

図 5: 実行結果

14 課題 4-2

k の変化に伴う訪問ノード数オープンリストの最大長実行時間（ミリ秒）を調べなさい.

BFS の場合
k=3
Visited Nodes : 29 Max Open List Size : 6 Execution Time : 30 ms
k=4
Visited Nodes : 52 Max Open List Size : 12 Execution Time : 29 ms
k=5
Visited Nodes : 62 Max Open List Size : 13 Execution Time : 37 ms
k=6
Visited Nodes : 101 Max Open List Size : 25 Execution Time : 35 ms
k=7
Visited Nodes : 121 Max Open List Size : 28 Execution Time : 35 ms
k=8
Visited Nodes : 162 Max Open List Size : 40 Execution Time : 60 ms
k=9
Visited Nodes : 186 Max Open List Size : 47 Execution Time : 56 ms
k=10
Visited Nodes : 247 Max Open List Size : 66 Execution Time : 68 ms
DFS の場合
k=3
Visited Nodes : 17 Max Open List Size : 10 Execution Time : 0 ms
k=4
Visited Nodes : 19 Max Open List Size : 17 Execution Time : 7 ms
k=5
Visited Nodes : 23 Max Open List Size : 27 Execution Time : 8 ms
k=6
Visited Nodes : 35 Max Open List Size : 51 Execution Time : 9 ms
k=7
Visited Nodes : 33 Max Open List Size : 49 Execution Time : 7 ms

k=8
Visited Nodes : 39 Max Open List Size : 84 Execution Time : 8 ms
k=9
Visited Nodes : 37 Max Open List Size : 80 Execution Time : 0 ms
k=10
Visited Nodes : 48 Max Open List Size : 117 Execution Time : 15 ms

15 課題 4-3

最後に k の変化に伴う実行時間等の変化について予想を踏まえて考察しなさい.

BFS も DFS も k の変化によって実行時間は比例して変化すると思う

16 課題 4-4, 4-5

クローズドリストを導入しその効果を検証したうえでその効果について考察すること.

すでにクローズドテストを導入していたためベンチマークは上記と変わらない。k が 4 以上は実行時間がかなり長くなったため k=3,4 でのみ比較を行った。クローズドリストの効果比較は— k — With Closed List — Without Closed List — Visited Nodes Ratio — Time Ratio — ———— ——————————————————————— — — Visited — Time(ms) — Visited — Time(ms) — (Without/With) — (W/o / W) — — 3 — 29 — 0 — 10,964 — 1,104 — 378.07 — 1,104.00 — — 4 — 52 — 0 — 16,806 — 1,864 — 323.19 — 1,864.00

16.1 考察・感想

考察: 課題 1-1d では k 人の宣教師と k 人の人食い人種という一般化された問題を実装し k の変化に伴う性能の変化を測定した．またクローズドリストの効果を検証することで探索アルゴリズムの最適化手法について深く理解することができた.

まず k の増加に伴う性能の変化について分析する．BFS の結果から k=3 では訪問ノード数 29k=10 では 247 と k が増加するにつれて訪問ノード数が増加している．これは状態空間が k の増加とともに拡大するためである．しかし増加率は指数関数的ではなく比較的緩やかな増加に留まっている．これは制約条件により多くの無効な状態が排除されるためと考えられる.

DFS の場合訪問ノード数は BFS よりも少ない傾向にあるがこれは運良く早期に解を発見できた場合の結果である．ただし DFS は最短解を保証しないため実用的には BFS の方が信頼性が高い.

最も重要な発見はクローズドリストの効果である．実験結果から k=3 でクローズドリストなしの場合訪問ノード数が 10,964（約 378 倍）実行時間が 1,104ms（約 1,100 倍）となった．k=4 では訪問ノード数が 16,806（約 323 倍）とクローズドリストの有無で差が生じた．これは同じ状態に到達する経路が複数存在するこの問題の性質によりクローズドリストなしでは同じ状態を何度も訪問してしまうためである.

またボート容量を k/2+1 とすることで k が大きい場合でも解が存在することを確認できた．この設定により効率的な移動が可能となり問題の拡張性が向上している．

感想:　課題 1-1d の実装を通じてアルゴリズムの最適化がいかに重要かを実感した．クローズドリストという単純な改良により訪問ノード数が約 300 倍以上削減されるという結果は驚きであった．特に k=5 以上ではクローズドリストなしでは現実的な時間で解を得ることが困難になる点が印象的だった．

また k=3 から k=10 まで段階的に性能を測定することで問題規模の増加に伴うアルゴリズムの振る舞いを可視化できた点も有益だった．グラフや表形式で結果を整理することで訪問ノード数やオープンリストの最大長が k に対してほぼ線形に増加することが明確になった．

BFS と DFS の比較からは最短解を求める必要がある場合には BFS が適していることを再確認できた．DFS は訪問ノード数が少なく見えるがこれは偶然早期に解を発見できた場合であり常に効率的とは限らない．問題の特性に応じた適切なアルゴリズム選択の重要性を実データをもとに学ぶことができた．

17　課題 1-2a

演習 1-1 のプログラムにコストを導入するための拡張点について説明せよ, 特に経路コスト g やヒューリスティック関数 h ′ の実現方法を答えよ

船の左右の移動で消費コストが変わったり、船で運ぶ人数でコストが変わったりしてもコストを導入することで対応可能である。この点で拡張性がある。経路コスト g はアクションのコストと状態の累計コストで構成されている。

ヒューリスティック関数 h ’ は

```
class MisCanHeuristic implements Heuristic  public float eval(State s)  var w = (MisCanWorld) s.world(); return w.missionary;
```

実際に左岸にいる宣教師の数をそのまま返している。

どのように探索プログラムに評価関数を与えているのかを説明せよ

```
public interface Evaluator  public static Evaluator minCost()  return new MinCostEvaluator();  public static Evaluator bestFirst(Heuristic h)  return new BestFirstEvaluator(h);
public static Evaluator aStar(Heuristic h)  return new AStarEvaluator(h);
float f(State s);
default float g(State s)  return s.cost;
```

この関数によりかかったコストを評価として扱っている。

18　課題 1-2b

最小コスト優先探索

visited: 4674, max length: 2677 Execution time: 234 ms

最良優先探索

visited: 404, max length: 252 Execution time: 37 ms

A*アルゴリズム

visited: 236, max length: 160 Execution time: 15 ms

19　課題 1-2c

繰り返し回避の導入に伴う訪問ノード数の削減率を報告せよ

19.1　実行結果

19.1.1　初期状態: Example (2 3 5 / 7 1 6 / 4 8 0)

最小コスト優先探索:

- 繰り返し回避なし: 計測省略（処理が遅すぎるため）
- 繰り返し回避あり: 訪問ノード数 = 4,674

最良優先探索:

- 繰り返し回避なし: 訪問ノード数 = 50,000（上限到達）
- 繰り返し回避あり: 訪問ノード数 = 404
- 削減率: 計算不可（上限到達のため）。参考値として 50,000 → 404 に削減

A*アルゴリズム:

- 繰り返し回避なし: 訪問ノード数 = 3,135
- 繰り返し回避あり: 訪問ノード数 = 236
- **削減率 = 1 - 236 / 3,135 = 0.9247 (92.47%)**

19.1.2 初期状態: Random n=100 (4 1 3 / 2 5 6 / 0 7 8)

最小コスト優先探索:

- 繰り返し回避あり: 訪問ノード数 = 230
- 繰り返し回避なし: 計測省略

最良優先探索:

- 繰り返し回避なし: 訪問ノード数 = 50,000（上限到達）
- 繰り返し回避あり: 訪問ノード数 = 42
- 削減率: 計算不可（上限到達のため）

A*アルゴリズム:

- 繰り返し回避なし: 訪問ノード数 = 45
- 繰り返し回避あり: 訪問ノード数 = 18
- **削減率 = 1 - 18 / 45 = 0.6000 (60.00%)**

19.1.3 初期状態: Random n=200 (2 4 3 / 6 7 8 / 1 5 0)

最小コスト優先探索:

- 繰り返し回避あり: 訪問ノード数 = 4,278
- 繰り返し回避なし: 計測省略

最良優先探索:

- 繰り返し回避なし: 訪問ノード数 = 50,000（上限到達）
- 繰り返し回避あり: 訪問ノード数 = 366
- 削減率: 計算不可（上限到達のため）

A*アルゴリズム:

- 繰り返し回避なし: 訪問ノード数 = 2,334
- 繰り返し回避あり: 訪問ノード数 = 206
- **削減率 = 1 - 206 / 2,334 = 0.9117 (91.17%)**

19.1.4 初期状態: Random n=300 (4 3 1 / 5 8 0 / 7 6 2)

最小コスト優先探索:

- 繰り返し回避あり: 訪問ノード数 = 2,467
- 繰り返し回避なし: 計測省略

最良優先探索:

- 繰り返し回避なし: 訪問ノード数 = 50,000（上限到達）
- 繰り返し回避あり: 訪問ノード数 = 672
- 削減率: 計算不可（上限到達のため）

A*アルゴリズム:

- 繰り返し回避なし: 訪問ノード数 = 1,918
- 繰り返し回避あり: 訪問ノード数 = 158
- **削減率 = 1 - 158 / 1,918 = 0.9176 (91.76%)**

19.2 削減率のまとめ

表 1: 繰り返し回避による訪問ノード数の削減率

手法	Example	n=100	n=200	n=300
最小コスト優先探索	-	-	-	-
最良優先探索	上限到達	上限到達	上限到達	上限到達
A*アルゴリズム	92.47%	60.00%	91.17%	91.76%

19.3 考察

A*アルゴリズムの削減率について　A*アルゴリズムでは、4 つの初期状態すべてで削減率を計算することができた。削減率は 60.00%から 92.47%と非常に高く、特に Example と n=200、n=300 の問題では約 90%以上の削減を達成している。n=100 の問題では削減率が 60.00%と比較的低いが、これは元々の問題が簡単で訪問ノード数が少なかったためと考えられる。

最良優先探索について　最良優先探索では、繰り返し回避なしの場合、すべての初期状態で上限（50,000 訪問ノード）に達してしまった。一方、繰り返し回避ありの場合は 404 ノード以下で解を発見できており、繰り返し回避の効果が極めて大きいことが確認できた。最良優先探索はヒューリスティック関数のみを評価するため、同じ状態を何度も訪問してしまう傾向が強いと考えられる。

最小コスト優先探索について　最小コスト優先探索では、繰り返し回避なしの場合は処理が遅すぎるため計測を省略した。繰り返し回避ありの場合でも 230 から 4,674 ノードと比較的多くのノードを訪問しており、他の手法と比較して効率が悪い。これは経路コストのみを評価するため、ゴールへの方向性が考慮されないためである。

結論　繰り返し回避の導入により、訪問ノード数を大幅に削減できることが確認された。特に A* アルゴリズムでは平均約 83.85%の削減率を達成し、効率的な探索が可能となった。最良優先探索では削減率の正確な計算はできなかったが、50,000 ノードから数百ノードへの劇的な削減が見られた。8 パズルのような状態空間探索では、繰り返し回避が必須の最適化手法であると言える。

20　課題 1-2d

異なる 3 種類のヒューリスティック関数 h ′ 1, h ′ 2, h ′ 3 を比較しヒューリスティック関数の違いが性能に与える影響を調べよ

20.1　ヒューリスティック関数の設計

以下の 3 種類のヒューリスティック関数を設計し、$h'_1 \leq h'_2 \leq h'_3 \leq h$ の関係を持つように実装した。

h'_1: 誤配置タイル数の半分

$$h'_1(n) = \frac{1}{2} \times |\{i \mid \text{board}[i] \neq 0 \wedge \text{board}[i] \neq \text{GOAL}[i]\}| \tag{1}$$

最も弱いヒューリスティック関数で、正しい位置にないタイルの数を 2 で割った値を返す。誤配置されたタイルを正しい位置に移動するには、平均的に複数回の移動が必要であるため、誤配置数の半分を推定値とする。この関数は常に真の最短手数 $h(n)$ を過小評価する。

h'_2: 誤配置タイル数

$$h'_2(n) = |\{i \mid \text{board}[i] \neq 0 \wedge \text{board}[i] \neq \text{GOAL}[i]\}| \tag{2}$$

中程度の強さのヒューリスティック関数で、正しい位置にないタイルの総数を返す。各誤配置タイルを正しい位置に移動するには最低 1 回の移動が必要であるという推定に基づく。ただし、実際には複数のタイルが同時に動くことはないため、真の最短手数を過小評価する。

h'_3: マンハッタン距離

$$h'_3(n) = \sum_{i=1}^{8} (|\text{row}_{\text{current}}(i) - \text{row}_{\text{goal}}(i)| + |\text{col}_{\text{current}}(i) - \text{col}_{\text{goal}}(i)|) \tag{3}$$

最も強いヒューリスティック関数で、各タイルが目標位置に到達するまでに必要な最小移動回数（水平距離+垂直距離）の総和を返す。この関数は各タイルが独立に動けると仮定しているため、実際にはタイル同士が干渉するために真の最短手数を過小評価する。

20.2 大小関係の理論的考察

$h'_1 \leq h'_2$ の成立　任意の状態 n において、$h'_1(n) = h'_2(n)/2$ であるため、$h'_1(n) \leq h'_2(n)$ は常に成立する。例外は発生しない。

$h'_2 \leq h'_3$ の成立　誤配置されているタイル 1 つあたりのマンハッタン距離は最小で 1（隣接位置）であるため、一般に $h'_2(n) \leq h'_3(n)$ が成立する。ただし、すべてのタイルが隣接位置に誤配置されている場合は $h'_2(n) = h'_3(n)$ となるが、これは稀である。実験結果からも、ほぼすべての状態で $h'_2(n) \leq h'_3(n)$ が成立することが確認できた。

$h'_3 \leq h$ の成立　マンハッタン距離は「障害物がない場合の最短移動距離」を示すが、8 パズルでは他のタイルが障害物となるため、実際の最短手数 $h(n)$ はマンハッタン距離以上になることが多い。したがって、一般に $h'_3(n) \leq h(n)$ が成立する（許容的ヒューリスティック）。稀に $h'_3(n) = h(n)$ となる理想的な状態も存在するが、$h'_3(n) > h(n)$ となることはない。

20.3 実験設定

以下の 5 つの初期状態に対して A*アルゴリズムを実行し、3 種類のヒューリスティック関数の性能を比較した。

1. Example: $\{2, 3, 5, 7, 1, 6, 4, 8, 0\}$（授業例）
2. Random n=100: 学生番号を乱数シードとして 100 回ランダム操作
3. Random n=200: 学生番号を乱数シードとして 200 回ランダム操作
4. Random n=300: 学生番号を乱数シードとして 300 回ランダム操作
5. Difficult Example: $\{8, 6, 7, 5, 0, 4, 2, 3, 1\}$（難しい 8 パズルの例）

20.4 実験結果

表 2 に各初期状態・各ヒューリスティック関数における性能指標を示す。

20.5 解法の報告

各パズルの具体的な解法手順は「8-puzzle-ex12d.txt」に出力した。手数は以下の通りである。

- Example: 14 手
- Random n=100: 21 手
- Random n=200: 17 手
- Random n=300: 20 手
- Difficult Example: 28 手

すべてのヒューリスティック関数で同じ手数の最適解が得られており、A*アルゴリズムが正しく機能していることが確認できる。

表 2: ヒューリスティック関数による性能比較

初期状態	関数	手数	訪問数	Open 最大	Closed 最大	時間 (ms)
Example	h'_1	14	902	572	901	88
Example	h'_2	14	236	160	235	18
Example	h'_3	14	85	60	84	6
Random n=100	h'_1	21	21,920	11,399	21,919	13,170
Random n=100	h'_2	21	5,279	3,231	5,278	556
Random n=100	h'_3	21	409	250	408	11
Random n=200	h'_1	17	4,041	2,470	4,040	358
Random n=200	h'_2	17	1,035	668	1,034	20
Random n=200	h'_3	17	264	173	263	0
Random n=300	h'_1	20	16,323	8,803	16,322	7,635
Random n=300	h'_2	20	3,964	2,475	3,963	281
Random n=300	h'_3	20	494	287	493	18
Difficult	h'_1	28	153,442	32,147	153,441	909,515
Difficult	h'_2	28	80,807	29,721	80,806	197,077
Difficult	h'_3	28	4,515	2,472	4,514	2,069

20.6 考察

訪問ノード数の比較　ヒューリスティック関数が強いほど訪問ノード数が劇的に減少している。Difficult Example では、h'_1 が 153,442 ノード、h'_2 が 80,807 ノード（約 47%減）、h'_3 が 4,515 ノード（約 97%減）を訪問している。これは、より正確な残り手数の推定により、無駄な探索が削減されたためである。

Random n=100 の場合、h'_1 から h'_3 への改善により訪問ノード数が約 98%（21,920 → 409）削減されており、ヒューリスティック関数の質が探索効率に極めて大きな影響を与えることが分かる。

オープンリストとクローズドリストの最大長　強いヒューリスティック関数ほど、オープンリストとクローズドリストの最大サイズが小さくなっている。これはメモリ使用量の削減につながり、より大規模な問題にも対応可能になることを示唆している。

実行時間の比較　実行時間も訪問ノード数に比例して改善されている。Difficult Example では、h'_1 が約 15 分（909 秒）かかったのに対し、h'_3 は約 2 秒で解を発見しており、約 440 倍の高速化が達成されている。

ヒューリスティック関数の選択指針　実験結果から、マンハッタン距離（h'_3）が最も優れた性能を示すことが明らかになった。誤配置タイル数（h'_2）も実用的な性能を持つが、難しい問題では h'_3 との性能差が顕著になる。h'_1 は探索効率が著しく低く、実用的ではない。

一般に、ヒューリスティック関数は「許容的（真の距離を過大評価しない）」かつ「できるだけ真の距離に近い」ことが望ましい。マンハッタン距離はこの両方の性質を高いレベルで満たしているため、8 パズルにおいて最適なヒューリスティック関数と言える。

大小関係の実験的検証　すべての実験結果において、3 つのヒューリスティック関数は同じ手数の最適解を発見している。これは、すべてのヒューリスティック関数が許容的（$h' \leq h$）であり、A* アルゴリズムの最適性が保証されていることを示している。