

知能プログラミング演習II 課題1

グループ8

29114003 青山周平

29114060 後藤拓也

29114116 増田大輝

29114142 湯浅範子

29119016 小中祐希

2019年10月15日

提出物 rep1, group08.zip

1 課題の説明

必須課題 1-1 Search.java の状態空間におけるパラメータ（コストや評価値）を様々なに変化させて実行し、各探索手法の違いを説明せよ。

具体的には、変化させたパラメータと探索結果（最短パス探索の成否、解を返すまでのステップ数、etc.）の関係を、探索手法毎に表やグラフ等にまとめよ。それらの結果を参照して考察を行い、各探索手法の違いを説明せよ。

必須課題 1-2 グループでの進捗管理や成果物共有などについて、工夫した点や使ったツールについて考察せよ。

発展課題 1-3 Search.java の探索過程や最終的に得られた順路をユーザに視覚的に示す GUI を作成せよ。

2 必須課題 1-1

Search.java の状態空間におけるパラメータ（コストや評価値）を様々なに変化させて実行し、各探索手法の違いを説明せよ。

具体的には、変化させたパラメータと探索結果（最短パス探索の成否、解を返すまでのステップ数、etc.）の関係を、探索手法毎に表やグラフ等にまとめよ。それらの結果を参照して考察を行い、各探索手法の違いを説明せよ。

2.1 手法

グループで役割分担をし、各探索手法について調べた。

▷ 幅優先探索は今いるノードの子ノードの探索が全て終わってから次のノードの子ノードを探索する手法である。またコストやヒューリスティック値のパラメータに依存せず、状態空間の変化に応じてのみ探索順が変化する。そのため今回は状態空間そのものの変更を行った。

▷ 深さ優先探索は、今いるノードの子ノードを次々順に探索していく手法である。この探索手法も幅優先探索同様コストやヒューリスティック値のパラメータに依存せず、状態空間の変化に応じてのみ探索順が変化する。そのため今回は状態空間そのものの変更を行った。

▷ 分岐限定法では最終的に得られる経路は最適解になる。従って、今回はノード間のコストをランダムに生成するようにして様々なパターンを試し、計算時間やステップ数を比較考察する。

▷ 山登り法はヒューリスティック値だけを用いた探索法である。今回は、最初に与えられたヒューリスティック値では無限ループに陥ってしまうため、ヒューリスティック値の一部を変更して適切な経路をたどるようにした。

▷ 最良推定法は各ノードにおけるヒューリスティック値をもとに探索を進めていく方法で、直前のノードから次のヒューリスティック値だけを見て、行き当たりばったりに探索を進める山登り法とは異なり、過去のデータ（これから訪れる可能性をもったノード）のヒューリスティック値を OpenList に保存するので、山登り法よりも最適な探索が可能である。

▷A*アルゴリズムはそのノードまでのコストの合計とそのノードのヒューリスティック値の和をとる。最良優先探索の弱点でさえも初期ノードからのコストを考慮することで、正解にはたどり着ける。

ここから、与えられた Search.java の探索プログラムを以下のように改良した。

1. 状態空間のパラメータや状態空間そのものを変更できるようコマンドライン引数の数を 1 つ増やす。
 2. 分岐限定法で、パラメータ変更を乱数で決定する。
 3. 各探索手法の実行時間を表示する。
 4. 状態空間の関係性を CSV ファイルに出力する。
1. に関しては、パラメータの変化のみであれば(ヒント)のように配列を用いる手法も取れると考えたが、幅優先探索と深さ優先探索はコストや評価値を用いずに探索を行う。これでは与えられた状態空間では常に結果が一定になってしまうため、状態空間そのものを変更出来るよう実装を行った。
 2. に関しては、ノードの生成を各探索ごとに switch 文で場合分けし、コマンドライン引数で指定できるようにした。分岐限定法では、ノードのコストを定義するところでランダム関数を用いることで 0 以上 10 未満の乱数を与えるようにした。
 3. に関しては、(ヒント)にあったように System.currentTimeMillis メソッドを活用して実装した。
 4. に関しては、CSV ファイル出力用に exportCsv メソッドを新しく作成し実装した。

2.2 実装

2.2.1 幅優先探索・深さ優先探索の実装

幅優先探索・深さ優先探索ではパラメータの変更では探索順が変化しないので、手法で示した 1. の実装を行うことで探索順を変更し比較を行えるようにした。

この時ノードの数・開始ノードと終了ノードは変更せず、パラメータの値とノードの接続の関係性のみを変更することとした。

さらに、幅優先探索と深さ優先探索は経路のコストを考えないため、それ以外の探索手法との比較は各々の探索手法のステップ数と、探索開始から終了までにかかる実行時間で行うこととした。

ステップ数は各ノードを探索した回数であるとプログラムから分かったので、これが幅優先探索と深さ優先探索で最小になれば実行時間も最小になると想え、実装は各探索手法のステップ数を考えながら行った。

この時2つ目のコマンドライン引数は、それぞれの探索の場合分けと同じ値を入力すると、それに対応した改良版の状態空間にノードが書き換わるようにした。またそれ以外の数を入力した場合は全て与えられた状態空間になるようにし、エラーが発生しないようにした。

始めに、コマンドライン引数を2つ受け取るmainメソッドの実装をソースコード1に示す。

ソースコード1: main メソッド

```
1 public static void main(String[] args) {
2     if (args.length != 2) {
3         System.out.println("USAGE:");
4         System.out.println("java Search [Number] [
5             NodePattern]");
6         System.out.println("[Number] = 1 : Breadth First
7             Search");
8         System.out.println("[Number] = 2 : Depth First
9             Search");
10        ...
11        System.out.println("[NodePattern] = 1 : Breadth
12            First Search is best");
13        System.out.println("[NodePattern] = 2 : Depth
14            First Search is best");
15        ...
16        System.out.println("[NodePattern] = other :
17            initial value");
18    }
19    ...
20}
```

さらに、受け取った引数を基に各々の状態空間の情報をSearchメソッド

ドと makeStateSpace メソッドに渡すため、引数 whichNode とその操作を以下のように実装した。

ソースコード 2: 引数 whichNode と関連メソッド

```
1 public static void main(String[] args) {
2     if (args.length != 2) {
3         ...
4     } else {
5         int which = Integer.parseInt(args[0]);
6         int whichNode = Integer.parseInt(args[1]);
7         switch (which) {
8             case 1:
9                 // 幅優先探索
10                System.out.println("\nBreadth First Search");
11                (new Search(whichNode)).breadthFirst();
12                ...
13            }
14        Search(int whichNode) {
15            makeStateSpace(whichNode);
16        }
17    private void makeStateSpace(int whichNode) {
18        ...
19        switch (whichNode) {
20            case 1:
21                ...
22        }
23    }
```

これ以降、switch 文を用いた場合分けにより状態空間を変化させる。

始めに、幅優先探索で変更した部分の状態空間のソースコードを示す。

ソースコード 3: 改良した部分の状態空間

```
1 case 1:
2     // 幅優先探索が最小ステップ数となるようノードの関係
3     // 性・コスト・ヒューリスティック値の変更
4     node[2] = new Node("Hollywood", 8);
5     ...
6     node[1].addChild(node[3], 1);
7     node[1].addChild(node[4], 6);
8     node[2].addChild(node[9], 9);
9     node[3].addChild(node[5], 5);
```

```
9     node[4].addChild(node[6], 4);
10    node[4].addChild(node[7], 2);
11    node[5].addChild(node[4], 1);
12    node[6].addChild(node[8], 7);
13    ...
14    break;
```

次に、深さ優先探索で変更した状態空間のソースコードを示す。

ソースコード 4: 改良した部分の状態空間

```
1 case 2:
2     //深さ優先探索が最小ステップ数となるようノードの関
      //係性・コスト・ヒューリスティック値の変更
3     node[1] = new Node("UCLA", 2);
4     ...
5     node[1].addChild(node[3], 6);
6     node[2].addChild(node[3], 4);
7     node[2].addChild(node[4], 6);
8     node[3].addChild(node[5], 5);
9     node[3].addChild(node[6], 2);
10    node[4].addChild(node[6], 2);
11    node[5].addChild(node[7], 1);
12    node[5].addChild(node[8], 1);
13    node[6].addChild(node[7], 7);
14    node[6].addChild(node[8], 2);
15    node[7].addChild(node[9], 1);
16    ...
17    break;
```

2.2.2 分枝限定法の実装

Search.java で改良を行った 2. の乱数発生に関するソースコードを以下に示す。

ソースコード 5: 改良した部分の状態空間

```
1 case 3:
2     //0以上 10未満の乱数を生成し、コストとする
3     Random rand = new Random();
4
5     node[0].addChild(node[1], rand.nextInt(10));
6     node[0].addChild(node[2], rand.nextInt(10));
```

```
7     node[1].addChild(node[2], rand.nextInt(10));
8     node[1].addChild(node[6], rand.nextInt(10));
9     node[2].addChild(node[3], rand.nextInt(10));
10    node[2].addChild(node[6], rand.nextInt(10));
11    node[2].addChild(node[7], rand.nextInt(10));
12    node[3].addChild(node[4], rand.nextInt(10));
13    node[3].addChild(node[7], rand.nextInt(10));
14    node[3].addChild(node[8], rand.nextInt(10));
15    node[4].addChild(node[8], rand.nextInt(10));
16    node[4].addChild(node[9], rand.nextInt(10));
17    node[5].addChild(node[1], rand.nextInt(10));
18    node[6].addChild(node[5], rand.nextInt(10));
19    node[6].addChild(node[7], rand.nextInt(10));
20    node[7].addChild(node[8], rand.nextInt(10));
21    node[7].addChild(node[9], rand.nextInt(10));
22    node[8].addChild(node[9], rand.nextInt(10));
23    break;
```

2.2.3 山登り法の実装

山登り法で変更したパラメータに関するソースコードを以下に示す。

ソースコード 6: 改良した部分の状態空間

```
1 case 4:
2     //node[5] のヒューリスティック値を 2 -> 5 と変更
3     node[5] = new Node("SanDiego", 5);
4     ...
5     break;
```

2.2.4 最良優先探索の実装

最良優先探索で変更したパラメータに関するソースコードを以下に示す。

ソースコード 7: 改良した部分の状態空間

```
1 case 5:
2     //node[7] のヒューリスティック値を 4 -> 2 と変更
3     node[7] = new Node("Pasadena", 2);
4     ...
5     break;
```

2.2.5 A*アルゴリズムの実装

A*アルゴリズムで変更したパラメータに関するソースコードを以下に示す。

ソースコード 8: 改良した部分の状態空間

```
1 case 6:
2     /*node[1] のヒューリスティック値を 7 -> 3 に変更
3     *node[7]->node[8] のコストを 1 -> 7 に変更
4     *node[7]->node[9] のコストを 7 -> 1 に変更
5     */
6     node[1] = new Node("UCLA", 3);
7     ...
8     node[7].addChild(node[8], 7);
9     node[7].addChild(node[9], 1);
10    ...
11    break;
```

2.2.6 その他の実装

3. の各探索手法の実行時間を表示するプログラムは、以下のように System.currentTimeMillis メソッドを実装した。

ソースコード 9: 実行時間計測機能を加えた main メソッド

```
1 public static void main(String[] args) {
2     ...
3     } else {
4         ...
5         long start = System.currentTimeMillis();
6         switch (which) {
7             ...
8         }
9         long end = System.currentTimeMillis();
10        System.out.println("探索時間: " + (end -
11            start) + "ms");
12    }
```

4. の状態空間の関係性を CSV ファイルに出力するプログラムは、以下のように exportCsv メソッドを実装した。

ソースコード 10: CSV 出力のための exportCsv メソッド (一部抜粋)

```
1 public void exportCsv(List<Node> parentList, List<Node>
2                         childList, String fileName){
3     int i = 0;
4     int j = 0;
5     try {
6         // 出力ファイルの作成
7         FileWriter f = new FileWriter("C:\\\\Users\\\\
8             Owner\\\\Desktop\\\\" + fileName + ".csv",
9             false);
10        PrintWriter p = new PrintWriter(new
11            BufferedWriter(f));
12        // ヘッダーを指定する
13        p.print("ParentNode:n");
14        p.print("ChildrenNode:m");
15        ...
16        // 内容をセットする
17        // 親ノードの表示ループ
18        while(i < parentList.size()) {
19            p.print(parentList.get(i) + ",");
20            // 子ノードの表示ループ
21            while(j < childList.size()) {
22                if(childList.get(j) != null
23                    ) {
24                    p.print(childList.
25                        get(j) + ",");
26                    j++;
27                } else {
28                    j++;
29                    break;
30                }
31            }
32            i++;
33        }
34    }
```

2.3 実行例

各探索手法に対し、1つ目のコマンドライン引数で呼び出される探索手法と同じ数字の2つ目のコマンドライン引数がその探索手法の改良型となるようにプログラムを書き換えた。具体的には以下のように結果が表示される。

-
- 1 Search.java が含まれるディレクトリ > java Search 1 1
 - 2 <幅優先探索に合わせ改良した状態空間を幅優先探索で探索した結果>
 - 3
 - 4 Search.java が含まれるディレクトリ > java Search 4 6
 - 5 <A*アルゴリズムに合わせ改良した状態空間を山登り法で探索した場合の結果>
 - 6
 - 7 Search.java が含まれるディレクトリ > java Search 3 0
 - 8 <与えられた状態空間を分枝限定法で探索した場合の結果>
-

また、与えられた状態空間は次の図1のようになっており、これを改良した実行例を以下に示していく。

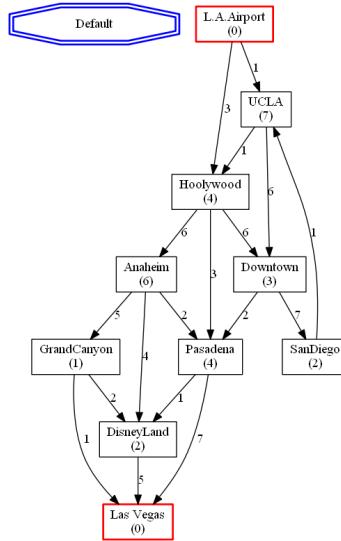


図 1: 与えられた状態空間

2.3.1 幅優先探索の実行例

幅優先探索では状態空間そのものを変更したため、始めにそれを示す(図 2)。

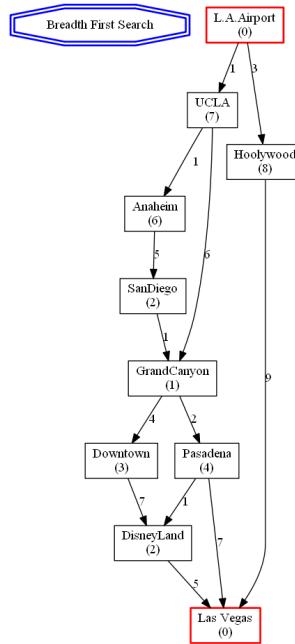


図 2: 幅優先探索が最小ステップ数になる状態空間

実行命令 (java Search 1 1) により出力される結果は以下のようになる。

```

1 Breadth First Search
2 STEP:0
3 OPEN: [L.A.Airport(h:0)]
4 CLOSED: []
5 STEP:1
6 OPEN: [UCLA(h:7), Hollywood(h:8)]
7 CLOSED: [L.A.Airport(h:0)]
8 :
9 :
10 STEP:3
11 OPEN: [Las Vegas(h:0), Anaheim(h:6), GrandCanyon(h:1)]
12 CLOSED: [L.A.Airport(h:0), UCLA(h:7), Hollywood(h:8)]
13 *** Solution ***
14 Las Vegas(h:0) <- Hollywood(h:8) <- L.A.Airport(h:0)
15 探索時間: 80ms
  
```

2.3.2 深さ優先探索の実行例

深さ優先探索でも状態空間そのものを変更したため、まずそれを示す(図3)。

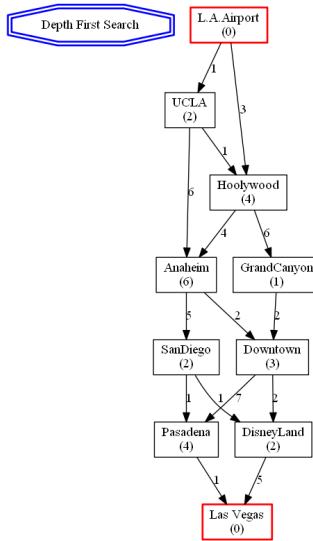


図3: 深さ優先探索が最小ステップ数になる状態空間

実行命令 (java Search 2 2) により出力される結果は以下のようになる。

```

1 Depth First Search
2 STEP:0
3 OPEN: [L.A.Airport(h:0)]
4 CLOSED: []
5 STEP:1
6 OPEN: [UCLA(h:2), Hollywood(h:4)]
7 CLOSED: [L.A.Airport(h:0)]
8 :
9 :
10 STEP:5
11 OPEN: [Las Vegas(h:0), DisneyLand(h:2), Downtown(h:3),
      Hollywood(h:4)]
12 CLOSED: [L.A.Airport(h:0), UCLA(h:2), Anaheim(h:6),
      SanDiego(h:2), Pasadena(h:4)]
13 *** Solution ***
14 Las Vegas(h:0) <- Pasadena(h:4) <- SanDiego(h:2) <-
      Anaheim(h:6) <- UCLA(h:2) <- L.A.Airport(h:0)
15 探索時間: 94ms
  
```

2.3.3 分枝限定法の実行例

分枝限定法は乱数でパラメータを決めるため、実行結果 (java Search 3 3) を二通り示す。

```
1 Branch and Bound Search
2 STEP:0
3 OPEN: [L.A.Airport(h:0)(g:0)]
4 CLOSED: []
5 STEP:1
6 OPEN: [Hollywood(h:4)(g:1), UCLA(h:7)(g:4)]
7 CLOSED: [L.A.Airport(h:0)(g:0)]
8 :
9 :
10 STEP:9
11 OPEN: [Las Vegas(h:0)(g:14)]
12 CLOSED: [L.A.Airport(h:0)(g:0), Hollywood(h:4)(g:1), UCLA
(h:7)(g:4), Downtown(h:3)(g:6), Anaheim(h:6)(g:7),
Pasadena(h:4)(g:8), SanDiego(h:2)(g:8), GrandCanyon(h
:1)(g:8), DisneyLand(h:2)(g:11)]
13 *** Solution ***
14 Las Vegas(h:0)(g:14) <- Pasadena(h:4)(g:8) <- Hollywood(
h:4)(g:1) <- L.A.Airport(h:0)(g:0)
15 探索時間: 139ms
```

```
1 Branch and Bound Search
2 STEP:0
3 OPEN: [L.A.Airport(h:0)(g:0)]
4 CLOSED: []
5 STEP:1
6 OPEN: [Hollywood(h:4)(g:4), UCLA(h:7)(g:9)]
7 CLOSED: [L.A.Airport(h:0)(g:0)]
8 :
9 :
10 STEP:3
11 OPEN: [Las Vegas(h:0)(g:7), UCLA(h:7)(g:9), Anaheim(h:6)(
g:10), DisneyLand(h:2)(g:10), Downtown(h:3)(g:11)]
12 CLOSED: [L.A.Airport(h:0)(g:0), Hollywood(h:4)(g:4),
Pasadena(h:4)(g:6)]
13 *** Solution ***
14 Las Vegas(h:0)(g:7) <- Pasadena(h:4)(g:6) <- Hollywood(h
:4)(g:4) <- L.A.Airport(h:0)(g:0)
15 探索時間: 87ms
```

2.3.4 山登り法の実行例

パラメータを変更した実行結果 (java Search 4 4) を以下に示す.

```
1 Hill Climbing Search
2 [UCLA(h:7), Hoolywood(h:4)]
3 [Anaheim(h:6), Downtown(h:3), Pasadena(h:4)]
4 [SanDiego(h:5), Pasadena(h:4)]
5 [DisneyLand(h:2), Las Vegas(h:0)]
6 *** Solution ***
7 Las Vegas(h:0) <- Pasadena(h:4) <- Downtown(h:3) <-
   Hoolywood(h:4) <- L.A.Airport(h:0)(g:0)
8 探索時間: 72ms
```

2.3.5 最良優先探索の実行例

パラメータを変更した実行結果 (java Search 5 5) を以下に示す.

```
1 Best First Search
2 STEP:0
3 OPEN: [L.A.Airport(h:0)(g:0)]
4 CLOSED: []
5 STEP:1
6 OPEN: [Hoolywood(h:4), UCLA(h:7)]
7 CLOSED: [L.A.Airport(h:0)(g:0)]
8 :
9 :
10 STEP:3
11 OPEN: [Las Vegas(h:0), DisneyLand(h:2), Downtown(h:3),
   Anaheim(h:6), UCLA(h:7)]
12 CLOSED: [L.A.Airport(h:0)(g:0), Hoolywood(h:4), Pasadena(h
   :2)]
13 *** Solution ***
14 Las Vegas(h:0) <- Pasadena(h:2) <- Hoolywood(h:4) <- L.A
   .Airport(h:0)(g:0)
15 探索時間: 99ms
```

2.3.6 A*アルゴリズムの実行例

パラメータを変更した実行結果 (java Search 6 6) を以下に示す.

```

1 A star Algorithm
2 STEP:0
3 OPEN: [L.A.Airport(h:0)(g:0)(f:0)]
4 CLOSED: []
5 STEP:1
6 OPEN: [UCLA(h:3)(g:1)(f:4), Hoolywood(h:4)(g:3)(f:7)]
7 CLOSED: [L.A.Airport(h:0)(g:0)(f:0)]
8 :
9 :
10 STEP:4
11 OPEN: [Las Vegas(h:0)(g:6)(f:6), Downtown(h:3)(g:7)(f
   :10), Anaheim(h:6)(g:8)(f:14), DisneyLand(h:2)(g
   :12)(f:14)]
12 CLOSED: [L.A.Airport(h:0)(g:0)(f:0), UCLA(h:3)(g:1)(f:4),
   Hoolywood(h:4)(g:2)(f:6), Pasadena(h:4)(g:5)(f:9)]
13 *** Solution ***
14 Las Vegas(h:0)(g:6)(f:6) <- Pasadena(h:4)(g:5)(f:9) <-
   Hoolywood(h:4)(g:2)(f:6) <- UCLA(h:3)(g:1)(f:4) <- L
   .A.Airport(h:0)(g:0)(f:0)
15 探索時間: 96ms

```

2.3.7 CSV ファイル出力の実行例

各探索ファイル出力を行う状態でプログラムを実行すると、先に記述した実行例の*** Solution ***の前で以下のように出力される。

```

1 ParentList.size = 8(各探索により異なる)
2 ChildList.size = 22(各探索により異なる)
3 ファイル出力完了!

```

また、A*アルゴリズムではこれに加え、csv 出力用 親ノード格納リスト parentList, csv 出力用 子ノード格納リスト childrenList, プログラム中で活用される変数 gmn,hmn,fmn もステップの出力中に表示されるようになっている。

2.4 考察

与えられた状態空間で得られた結果とパラメータや状態空間を変化させ得られた結果をもとに、各探索手法について考察していく。

2.4.1 幅優先探索・深さ優先探索の考察

与えられた状態空間と、幅優先探索・深さ優先探索のために作成した状態空間の3つで各探索手法のステップ数と実行時間を表にまとめる(表1, 表2, 表3)。ここでまとめた実行時間は、各探索手法を10回計測した平均時間とした。

実行環境はWindowsで、統合開発環境Eclipseに入っているjavaコンパイラとeclipseを使用しないjavaコンパイラでは実行時間が大きく変化することが分かった。そのため、今回はどちらの場合も実行時間の計測を行った。実行時間の単位は全てmsである。

表1: 各探索手法のステップ数(STEP)

探索手法	初期状態	幅優先探索最良	深さ優先探索最良
幅優先探索	6	3	8
深さ優先探索	5	7	5
分枝限定法	7	9	9
山登り法	∞	(5)	(6)
最良優先探索	5	5	6
A*アルゴリズム	7	6	8

表2: Eclipseを使用する時の実行時間(ms)

探索手法	初期状態	幅優先探索最良	深さ優先探索最良
幅優先探索	4.8	2.4	4.0
深さ優先探索	3.3	5.6	3.5
分枝限定法	5.0	2.9	4.2
山登り法	∞	1.5	2.6
最良優先探索	3.7	3.1	3.0
A*アルゴリズム	5.1	2.8	3.4

表 3: Eclipse を使用しない時の実行時間 (ms)

探索手法	初期状態	幅優先探索最良	深さ優先探索最良
幅優先探索	140.8	104.6	139.5
深さ優先探索	116.0	126.7	112.8
分枝限定法	141.9	140.1	157.2
山登り法	∞	87.4	92.5
最良優先探索	115.2	116.5	117.6
A*アルゴリズム	135.7	142.7	140.2

ここから、幅優先探索と深さ優先探索の違いとその特徴の面と、それぞれの探索手法とその他の探索手法との違いと特徴を考察した。

幅優先探索は今いるノードの子ノードの探索が全て終わってから次のノードの子ノードを探索するため、終了ノードが探索木の浅い位置にあると探索が早く終了する。対して深さ優先探索は、今いるノードの子ノードを次々に順に探索するため、終了ノードが探索する優先度の高い経路(先に探索される経路)にあると探索が早く終了する。このことをふまえ、各探索手法がより効率よくなる状態空間を考えたものが、図2、図3の状態空間である。

また本題とは異なるが、Eclipse 利用時とそうでないときの実行時間が大きく異なった原因も考えた。それぞれの Java のバージョンを調べたところ、以下のようなことが分かった。

環境	バージョン
Eclipse	JavaSE-11
Java	9.0.1

このことから、Eclipse で利用している Java とコマンドプロンプトで利用する Java ではバージョンが異なっていたため、性能に大きな違いが出たのではないかと考えた。Eclipse は今回の課題で利用するためにインストールしたため、Java のバージョンが新しくなっていたのではないかと考えられた。

まず初めに、幅優先探索と深さ優先探索を比較する。

与えられた状態空間では、パラメータを変化させてもステップ数は常に一定である。これは2つの探索手法がパラメータに依存しないことによる。これでは常に深さ優先探索の方がステップ数が少ないとため、実行時間もパラメータに関わらず深さ優先探索の方が短いことが確認できた。そのため状態空間そのものを変更することで、幅優先探索の方がより少ないステップ数で探索が行える場合を考えた。

幅優先探索と深さ優先探索は対照的な探索方法であるため、状態空間の関係性も対照的になるのではないかと考えた。そこでまず開始ノードの終了ノード以外の8つのノードを縦長と横長に並べる状態空間を考えた(図4、図5)。

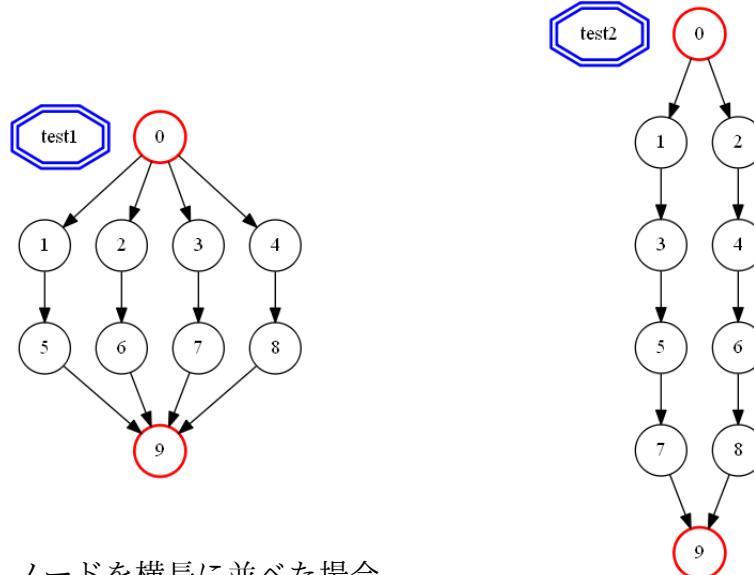


図4: ノードを横長に並べた場合

図5: ノードを縦長に並べた場合

しかし今回の状態空間では、各経路の終端は必ず終了ノードに接続されているため、図4の場合は深さ優先探索の方がステップ数が少なくなってしまう。そのため、対照的なノードの関係性だけでは、それぞれの探索のステップ数が必ずしも対照的になることはないと分かった。

図5を実行すると想定通りに深さ優先探索の方がステップ数が少なくなったため、ノードの位置関係はこのままに矢印のみ増やすこととした

(図3).

深さ優先探索は、先に探索した子ノードから優先的に探索を行う。今回のノードの条件では必ずどの経路も終了ノードにたどり着けるため、深さ優先探索は親ノードに戻って探索をやり直す必要がない。そのため最も効率が悪くなる場合は、先に探索する子ノードがより多くのノードを通して終了ノードにたどり着けば良いことが考えられた。そこからノードの構成を考え直した状態空間が、実行例で示した図2である。

実行結果からステップ数を確認すると、図2の場合は幅優先探索が深さ優先探索より少なく、図3の場合は深さ優先探索の方が少なくなっていることが確認できる。同時に表2・表3から、実行時間も図2の状態空間では幅優先探索が、図3の状態空間では深さ優先探索が短くなっていることが確認できた。

これらのことから全経路が終了ノードに収束する場合は、深さ優先探索は幅優先探索より少ないステップ数で探索できる場合が多いと考えられる。しかし、深さ優先探索の探索順を、初めに探索する経路が長くなるように状態空間を変化させることで、幅優先探索の方が効率の良い状態空間を作成することが出来ると分かった。

次に、幅優先探索・深さ優先探索とその他の探索手法との比較により、これらの特徴や違いを考察する。

幅優先探索・深さ優先探索と異なり、その他の探索手法はコストとヒューリスティック値をふまえて探索を行う。これらの探索手法は、幅優先探索・深さ優先探索より効率的な探索方法として考えられたものであるため、コストの概念をふまえた探索結果により出力される最短経路は、幅優先探索や深さ優先探索よりもその他の探索の方が良い経路となることがほとんどだと考えられる。ただし、ヒューリスティック値のみを用いる場合は、ヒューリスティック値の精度の高さによって得られる結果が変わり、良い結果になるとは限らないとも考えられる。

出力結果はコストやヒューリスティック値により変化してしまうため、幅優先探索や深さ優先探索との比較は難しいと考えた。そこで、先と同様に比較にはステップ数と実行時間を用いることとした。

先に作成した図2、図3の状態空間にコストとヒューリスティック値を与えることで、他の探索手法と比較を行った。

図2, 図3のようにコストとヒューリスティック値を与えた場合の結果が実行例に示した表1から表3である。表1から分かるように、図2の状態空間では幅優先探索が最小ステップ数となり、図3の状態空間では深さ優先探索が最小ステップ数となることが確認できる。山登り法はステップ数が実行結果に出力されなかったが、ヒューリスティック値を用いた深さ優先探索であるので、深さ優先探索と同様にしてステップ数を数えた。このようにステップ数だけで比較をするならば、幅優先探索や深さ優先探索もその他の探索手法より効率の良い探索が行えることが分かった。

またこの時の実行時間も確認したところ、山登り法を除き、ほとんどの場合でステップ数が多いほど実行時間も長くなっていることが確認できた。表2の幅優先探索最良時の深さ優先探索と分枝限定法のように、ステップ数が多いにもかかわらず実行時間が短くなっている場合も見受けられたが、これはステップ数の変化があまり大きくないために実行時の別の環境要因が影響しているのではないかと考えられた。

しかし、山登り法はステップ数が多い場合も実行時間が明らかに短くなっていることが確認された。この原因として、ヒューリスティック値はコストよりも値へのアクセスが早く行えることや、出力時に表示する内容が他の探索手法よりも少ないことなどが影響しているのではないかと考えた。

これらの結果から、探索するために訪れるノードの数は、コストやヒューリスティック値により幅優先探索や深さ優先探索の方が少なくなる場合が多いことが分かった。また、実行時間のほとんどがステップ数に影響することが確認できたので、幅優先探索や深さ優先探索の実行時間は他の探索手法より短くなることが多いことも分かった。しかし、今回は結果として得られた経路についてはふまえていないため、これを考慮すると幅優先探索や深さ優先探索はコスト面から効率の悪い探索手法になってしまふことも改めて確認できた。

2.4.2 分枝限定法の考察

分岐限定法はヒューリスティック値に頼らず、そこまでのノード間コストの合計が最も小さくなるようにノードをたどっていく探索方法である。また、openリストとcloseリストを用いて探索済みのノードを記録するため無限ループに陥る場合もない。

これにより最後に得られる解は最適の経路になるが、余計なノードを開く可能性が高く計算時間が長くなってしまうことがある。

実際、今回の実行例の1つ目と2つ目では計算時間やステップ数に大きな差が出た。

これらの実験結果やアルゴリズムの性質から、分岐限定法は最適解を求めたいときに適しているが、ノードの数が多くなるとステップ数が増え、計算時間が莫大になってしまう可能性があることが分かった。

2.4.3 山登り法の考察

山登り法はヒューリスティック値だけを用いた探索法である。

従って、与えられているヒューリスティック値が正確でないと間違った経路を選択してしまい、場合によっては目的のノードにたどり着けないことがある。

また、分岐限定法と異なり一度通ったノードを記憶しておくこともないので、今回のような探索木の場合は無限ループに陥ってしまうこともある。

2.4.4 最良優先探索の考察

最良推定法で初期のパラメータの探索を行うと図6のような木構造を持つ探索となる。

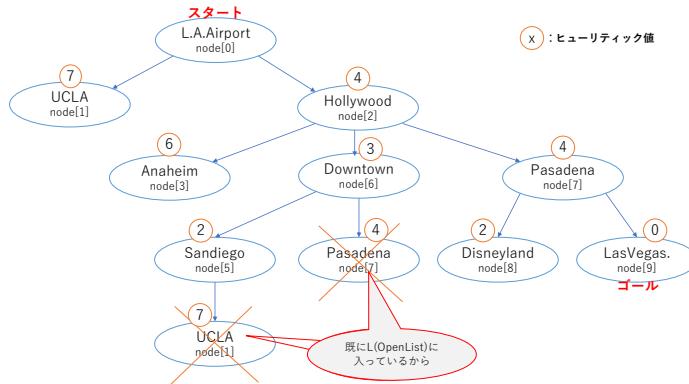


図 6: 初期パラメータにおける木構造モデル

探索経路の解としては,

[L.A.Airport → Hollywood → Pasadena → Las Vegas]

と 4STEP で進めるはずが, 探索ノード (親ノードの推移) としては,

[L.A.Airport → Hollywood → DownTown → Sandiego → Pasadena → LasVegas]

と, 7STEP を踏んでいる. Hollywood から次に進む際に Pasadena へ進むのではなく, 1 度 Downtown へ進んでいるのである. そのため, この改善策として, 「Pasadena:node[7] のヒューリスティック値:h(7) を Downtown:node[6]:h(6) よりも小さくする」方法を取る. そのパラメータ調整後の木構造が図 7 のようになる.

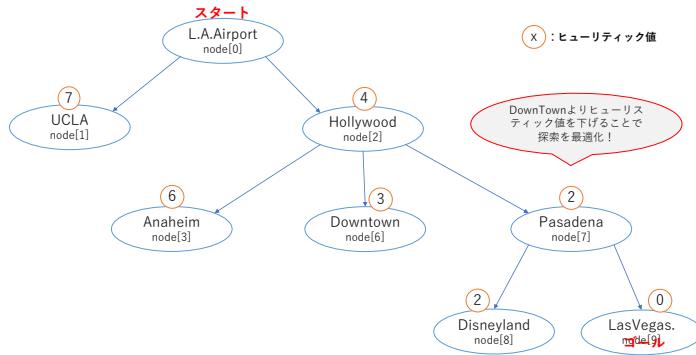


図 7: パラメータ調整後の木構造モデル

探索ノード (親ノードの推移) として,

[L.A.Airport → Hollywood → Pasadena → Las Vegas]

と最適パスとなった.

欠点としては, そのノードに固有なヒューリスティックな値を使って探索を進めていくので, すべてのノードのうちで最小のヒューリスティックの値をもつノードが無限に生成される場合は, 目標ノードに到達できない.

2.4.5 A*アルゴリズムの考察

初期のパラメータを用いて探索を行うと図 8 のような木構造を持つ探索となる。

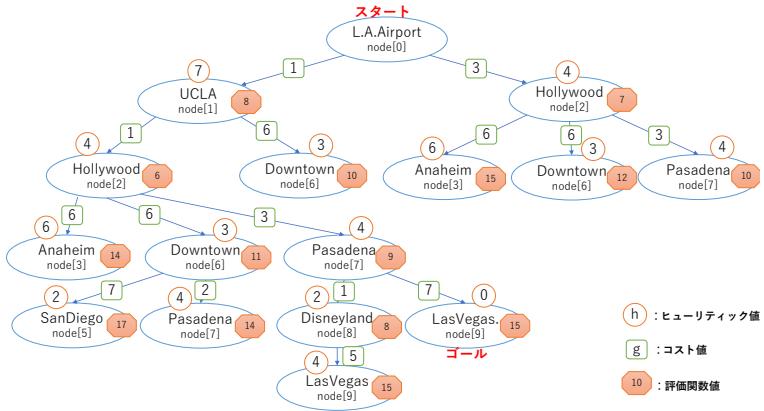


図 8: 初期パラメータにおける木構造モデル

探索経路の解としては,

[L.A.Airport → UCLA → Hollywood → Pasadena → Las Vegas]

と 5STEP で進めるはずが、探索ノード（親ノードの推移）としては、

[L.A.Airport → Hollywood → UCLA → Hollywood → Pasadena → DisneyLand → Dawntown → LasVegas]

と、8STEP を踏んでいる。余分な探索として、2箇所が上げられる。

1. 始めに Hollywood:node[2] に進まずに、UCLA:node[1] に進んでほしい。
2. 最後に Pasadena:node[7] から DisneyLand:node[8] に進まず、LasVegas:node[8] に進んでほしい。

よって、これらを改善するために以下の 2 つを変更してみる。

1. UCLA:node[1] のヒューリスティック値:h(1)=7 を Hollywood:node[2] のヒューリスティック値:h(2)=4 よりも小さくする。（既にコスト g の値は小さいので。）

2. Pasadena:node[7] から DisneyLand:node[8] のコストを Pasadena:node[7] から DisneyLand:node[8] のコストよりも小さくする. (既にヒューリスティック値は $h(9)$ よりも小さいので.)
 その結果, 下図 9 のようになった.

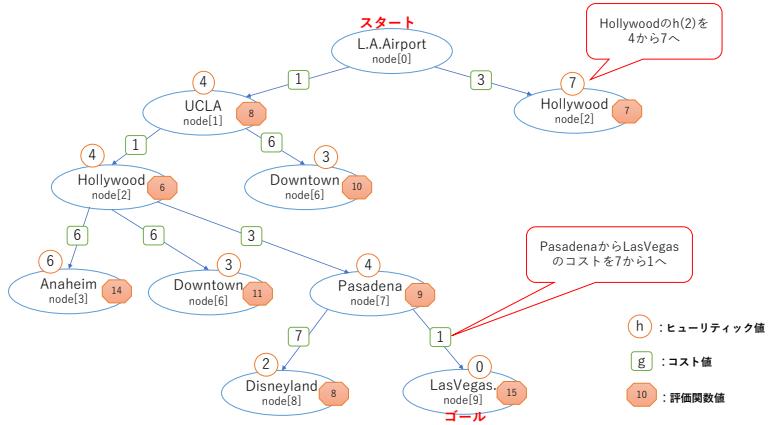


図 9: 2つのパラメータ変更後の木構造モデル

これによって, 探索経路の解と同じ 5STEP の探索ノード (親ノード) の推移を作ることができた.

なお, 始めの分岐で UCLA:node[1] を選ばず, Hollywood:node[2] を選ぶようにパラメータを調整することで, 4STEP 探索も可能となる.(木構造は最良優先探索と同様になるので省略.)

2.4.6 CSV 出力の考察

実際に探索手順を追っていくために,”知能処理学”で学んだような親ノード, 子ノードと展開されるリストとそれぞれのコストやヒューリスティックの値を表を実際に手で書いた. それをプログラムで出力させてみよう とまずは自分の担当だった A*アルゴリズムを用いてプログラムを書いた. Search クラスの中の aStar メソッドの中で, 該当する親, 子ノードを取得し, リストに格納していく. 探索がすべて終わったところで, 用意した exportCSV メソッドを呼び出し, 出力させる. プログラムのポイントは, 全てのノードデータを集めてから, 1枚の csv ファイルを出力させなければ ならないが, 各 STEP ごとで子ノードの数も違い, またどこまでが 1STEP

の情報なのか判別しないといけないことである。これは、1STEP が終了するごとに子ノードリストに null を入れることで解決した。exportCSV メソッド内でループを回して順にリストの中身を取る際に、子ノードリストに含まれている null の有無で条件判別ができ、STEP ごとに csv ファイルを改行させることに成功した。

試して分かったことだが、Node クラスでノードインスタンスを作る際に、そのノードにコスト:g、ヒューリスティック値:h、評価関数値:f がすでに含まれているので、わざわざ別個でそれらのデータを取る必要はなかった。それらの結果、ただ親ノードとそれに続く子ノードを出力し、csv ファイルに保存するだけのお粗末な仕上がりとなってしまった。ただそのおかげで、コストやヒューリスティック値の使用有無によらず、どの探索でも使用できる汎用性の高いプログラムとなった。

3 必須課題 1-2

グループでの進捗管理や成果物共有などについて、工夫した点や使ったツールについて考察せよ。

必須課題 1-2 は実装を伴わない課題であるため、実装と実行例は記述せずに手法と考察のみを記す。

3.1 手法(工夫した点)

今回のチーム開発を行うに当たって、チーム 8 では主に GitHub を活用することによって、進捗管理と成果物共有を行った。

まず、はじめに本講義用の Organization を作成した。Organization 名は NITech-ProgrammingTeam8 とした。

Organization にチームメンバー全員を招待し、第一回課題用のリポジトリ Work1 を作成した(図 10)。

(Organization の URL:<https://github.com/NITechProgrammingTeam8>)
(Repository の URL:<https://github.com/NITechProgrammingTeam8/Work1>)

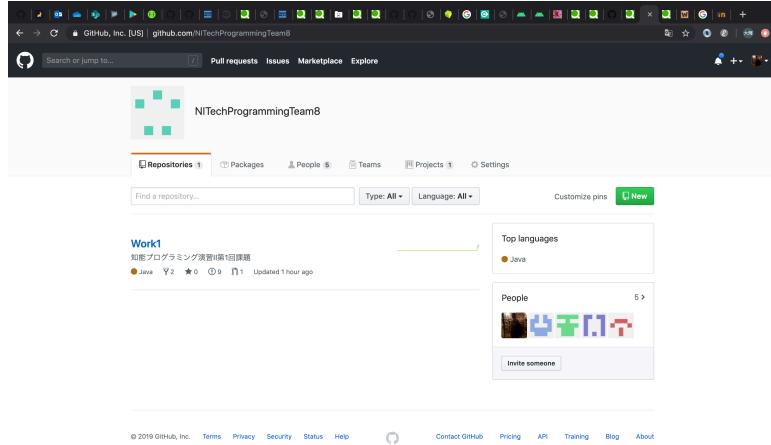


図 10: 作成した Organization

また、コミュニケーションやソースコードの共有を目的として,Slack を導入した.

3.2 考察

次に、導入したこれらのツールについてそれぞれ細かく分類して考察を行う.

3.2.1 GitHub の導入

GitHub で主に利用した機能を以下に列挙する.

Project 機能 Work1 用のプロジェクトを作成し、進捗管理を行うためのカンバンとする.

Issues 機能 課題をいくつかのタスクに分解することにより、それぞれに担当者を割り当てる.

MileStones 機能 個々の Issues を完了するまでの期間を定める.

PullRequest 機能 各 Issues と結びつけることによって、タスクと実際の作業を結びつける.

ここから、各機能に対する運用と考察を行う.

3.2.2 Project 機能の運用と考察

今回は Work1 というリポジトリに対応した Project を作成した。この Project はカンバンと呼ばれ、プロジェクト全体の進行度を視覚的に把握するのに最適である。

今回はタスクの進行度を以下の 3 種類に分類し、Column に登録した。さらに、PRへの割当や Merge といったイベントを完了することによって、タスクの進行度が変化するように自動化を行った。

To do これから着手する予定のタスク。PR 発行時に Developing に移動する。

Developing 開発中のタスク。Merge 後に Done に移動する。

Done 完了後のタスク。

発行した Issues をタスクとしてカンバンにおけるカードとした。以下に、Project の画像を添付する（図 11）。

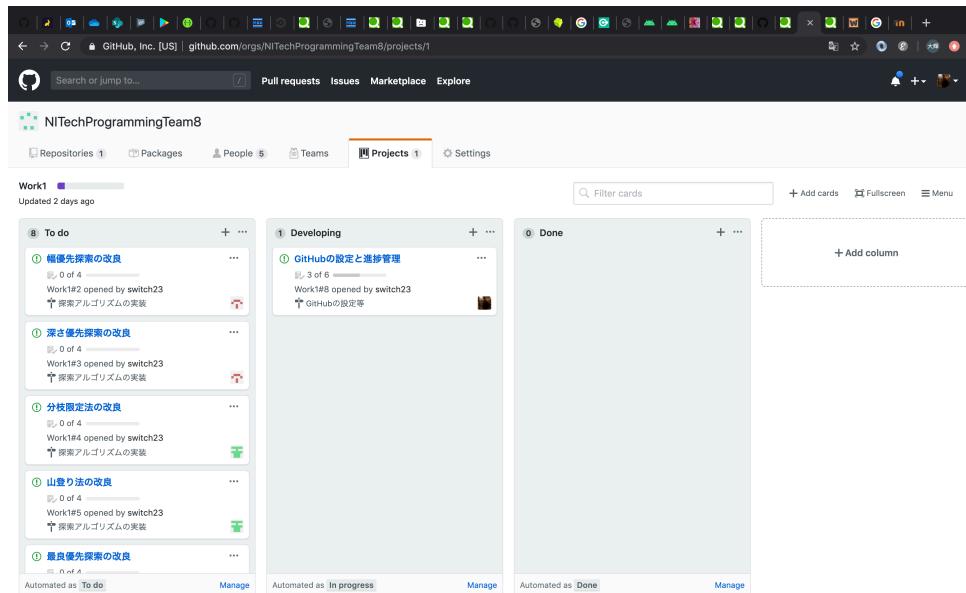


図 11: Project 機能を用いたタスク管理

3.2.3 Issues 機能の運用と考察

今回はタスクに対応した Issues を発行した。それぞれの Issues には最低1人の担当者を割り当て、MileStones 機能により期日を設けた。次のようにタスクを分解し、Issues を発行した。

幅優先探索

深さ優先探索

最良優先探索

A*アルゴリズム

分枝限定法

山登り法

GitHub 管理

GUI 設計

また、個々のタスクはさらに粒度の細かい作業に分解され、それぞれに対してチェック欄を設け、メンバー全体が各々の進捗状況を細かく確認できる仕様にした。以下に、Issues の画像を添付する(図 12,13)。

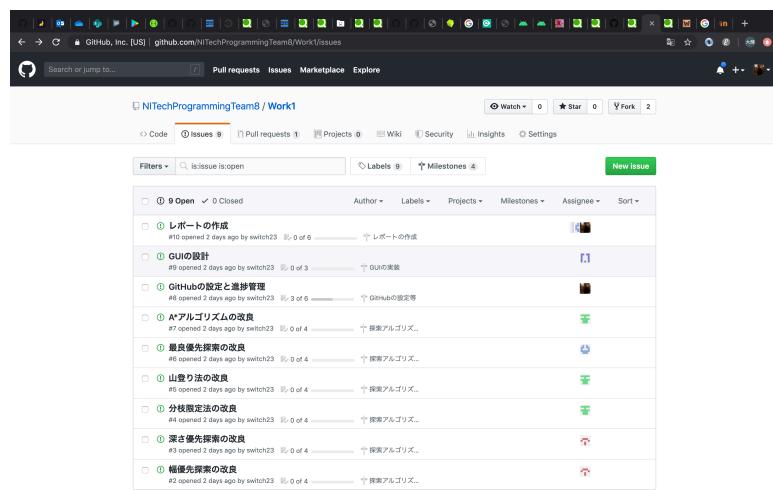


図 12: Issues 一覧

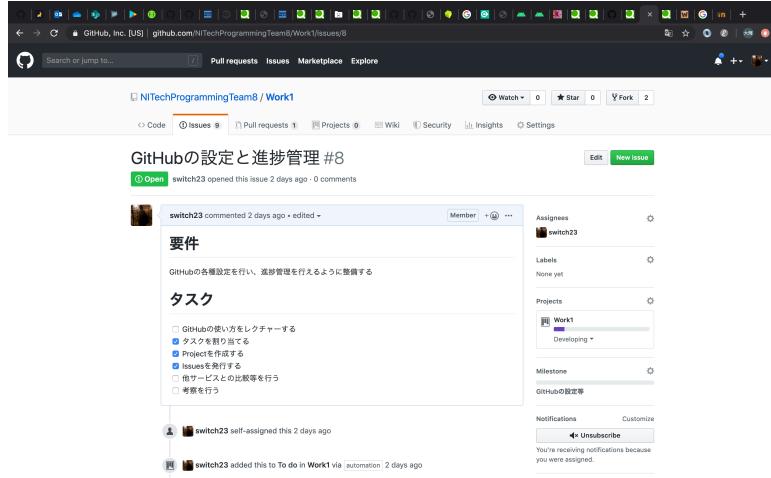


図 13: Issues 機能を用いたタスク割り当てとタスク分解

3.2.4 MileStones 機能の運用と考察

今回は個々のタスクを終了すべき期日を MileStones 機能により設定した。設定した MileStones は以下の通りで、その画像が次ページ図 14, 図 15 である。

探索アルゴリズムの実装 10/10(木)

GitHub の設定等 10/10(木)

GUI の実装 10/13(日)

レポートの作成 10/13(日)

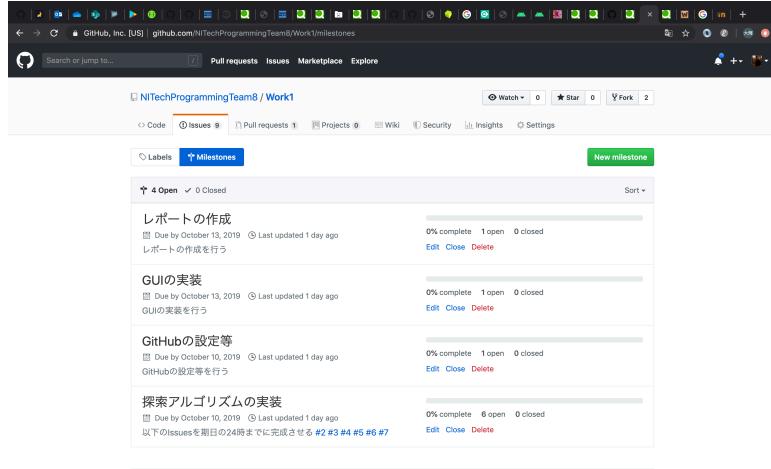


図 14: MileStones 一覧

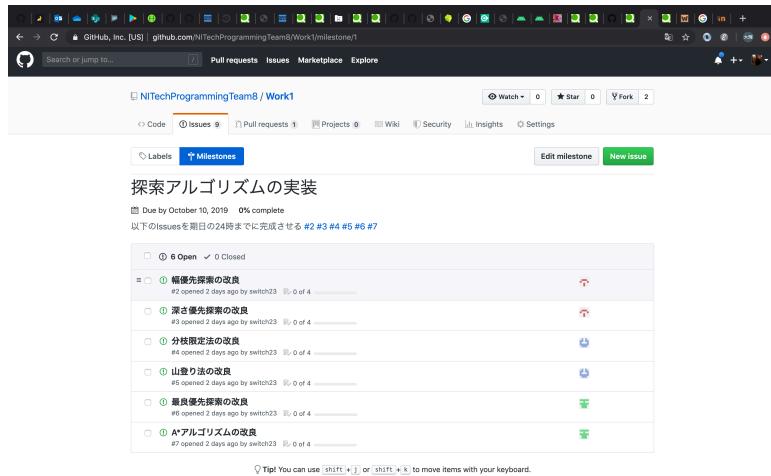


図 15: MileStones 機能を用いた Issues の期日管理

3.2.5 PullRequest 機能の運用と考察

今回は Issues 単位で PR を発行し、紐づけるものとする。
PR を行うことによって、変更を master ブランチに Merge する前にレビューを行い、事前に問題点などを吟味することができる。
また、master との差分によって変更点を視覚的に示すことが可能なため、メンバーの詳細な作業内容を把握しやすく、ソースコードの共有が容易と

なる。

さらに、何か議論がある場合には、PR 上で当該コードを指定してコメントを残すことができるため、具体性が増すメリットがある。

以上のように PR 機能を用いることによって、ソースコードの品質を維持するだけでなく、円滑なコミュニティを形成することが可能となる。以下にその画像を示す(図 16,17)。

The screenshot shows a GitHub pull request (PR) interface. The top bar indicates the repository is 'NI Tech Programming Team' and the PR number is '#14'. The main area displays the diff between two commits. The left side shows the commit history, and the right side shows the detailed code changes. A green box highlights a specific commit message: '• 2011-04-11 [switch23] 増田 倍人 - report/29114116.tex'.

Review comments are overlaid on the code. One comment at line 93 reads: '• 2011-04-11 [switch23] 増田 倍人 - report/29114116.tex' followed by a detailed description of the code's purpose and how it relates to project management. Another comment at line 94 reads: '• 2011-04-11 [switch23] 增田 倍人 - report/29114116.tex'.

At the bottom of the code view, there is a 'Comment for lines #92 - 99' section where a user has typed a message about the code's purpose and how it relates to project management.

図 16: PR の差分詳細

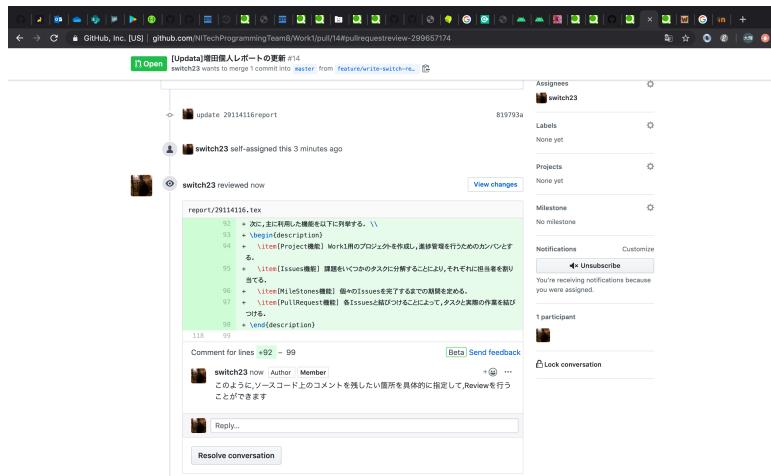


図 17: PR のレビュー機能

3.2.6 チームメンバーへのレクチャー

チーム開発においては必須と言える GitHub だが、概念的な理解が必要かつ多機能であるため、学習コストも非常に大きいというデメリットも存在する。しかし、今後の課題を想定すると、早い段階で GitHub を導入することにより、メンバー全員に Git 管理の習慣を定着させることが最善であると判断した。したがって、学習コストというデメリットを克服する必要があると考えられる。

私は、そのために基本的な Git コマンドの解説や手順を概念的な理解と共に学習できるように README.md を作成した。以下に、実際に作成したものを見せる。



図 18: README.md その 1



図 19: README.md その 2



図 20: README.md その 3

3.2.7 Slack の導入

Slack で専用のワークスペースを作成し、複数のチャネルを作成した。具体的には、今回の課題用のチャネル #work1 と、GitHub 運用についてのトピックスを扱うチャネル #github を作成した。

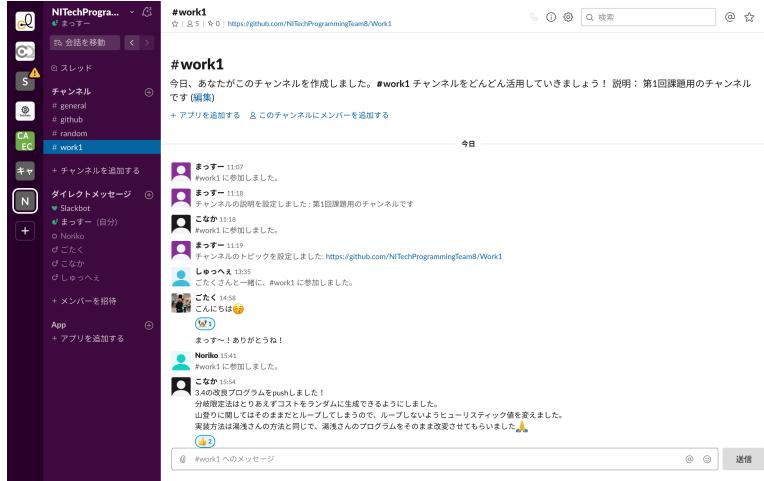


図 21: Slack での専用ワークスペース

4 発展課題 1-3

Search.java の探索過程や最終的に得られた順路をユーザに視覚的に示す GUI を作成せよ。

この課題では、GUI全般の Swing を用いた実装を行った。

4.1 手法

GUIを実装するにあたり、以下のような方針を立てた。

1. 探索空間における各要素（ノードや経路）を表示する。
 2. 要素に付随する値を入力するための入力ボックスを表示する。
 3. 探索を選択するためのボタンと、実行ボタンを表示する。
 4. 入力ボックスやボタンで入力した値を Search.java に反映し、実行する。
 5. 得られた経路を GUI に反映する。

1. に関して、ユーザーがなるべく直感的に理解できるようにすべく、表は用いずに 1 つの図の形式で完結するような仕様とした。
2. に関して、各要素とセットにして図中に入力ボックスを埋め込むことで、より直感的な入力を可能とした。また、入力ボックスには数値の入力に適した JSpinner クラスを用いた。
3. に関して、探索の選択には、選択肢の表示に適した JRadioButton クラスを用いた。値の反映には ActionListener インターフェースを用いて、Search.java の方も値を受け取って実行できるように改良した。
4. に関して、Search.java から探索結果を渡すように改良し、受け取った結果を経路に再描写することで、ユーザーが視覚的に経路を得ることができる仕様とした。

4.2 実装

実装にあたり、主に下記のサイトを参考にした。

TATSUO IKURA : 『Swing を使ってみよう - Java GUI プログラミング』 <https://www.javadrive.jp/tutorial/> (2019/10/15 アクセス)

SearchGUI.java には以下のクラスが含まれる。

- SearchGUI: メソッド main, searchButtons, actionPerformed を実装したクラス
- NodePanel: メソッド update と各種ゲッターを実装したノードに関するパネルを操作するためのクラス
- PathPanel: メソッド paintComponent, paintArrows, setShortestDistance, getMidPoints, execHor, execVer, update, forRepaint を実装した経路に関するパネルを操作するためのクラス

Search.java は以下のように改良した。

- Search クラスに、SearchGUI.java 用の実行メソッド exec を追加。

- Node クラスに、値の更新のためのメソッド setHValue, remakeChild を追加.
- Node クラスに値初期化のためのメソッド reset を追加.

ここから、各々の実装方法について詳細を述べる.

4.2.1 探索空間における各要素（ノードや経路）を表示するまで

SearchGUI クラスではまず、フレームの中に mainPanel と subPanel の 2 つのパネルを生成する.

mainPanel には経路やノードを格納するためのパネルを挿入した. ノードのパネルは JButton クラスと JSpinner クラスを用いて簡単にできるが、経路のパネルは線の描写が必要であるため、やや複雑な paintComponent メソッドを拡張して用いる必要があった.

paintComponent の複雑な点は、このメソッドによって行われる描写は実行時に 1 度限りであるという点である. 描写をパネルごとに分けて行っていたかったので、その仕様は解決すべき課題となった. そこで経路のパネル用のクラス PathPanel を用意し、その中で paintComponent メソッドを実装した. これによりインスタンスごとに描写呼び出しが行われて、経路を各パネルに分けることができた.

次に、表示についての課題が発生した. 教科書通りのような経路図を表示するためには、各パネルを自由な位置に指定する必要があった. Swing には表を表示するのに適したレイアウトは数多くあるが、図を表示するためのレイアウトをあまり見つけることはできず、選択肢としては以下の 2 つの方法があった.

1. レイアウトマネージャーを無効にしてコンポーネントを座標指定で配置する.
2. SpringLayout クラスを用いてコンポーネントを他のコンポーネントとの相対位置で指定して配置する.

1. の方法のほうが簡単ではあるが、実行環境に依存するという問題があるため、2. の方法を用いて実装した. そうすると、経路の表示について、出発地と到着地のノードの座標から、その相対座標を計算する必要があった. そこで、getMidPoints メソッドではノードのパネル 4 面

の各座標を計算し, setShortestDistance メソッドを用いてノード間の最短距離を計算することで実装した.

getMidPoints メソッドをソースコード 11 に, setShortestDistance メソッドをソースコード 12 に示す.

ソースコード 11: getMidPoints メソッド

```
1  Point[] getMidPoints(Rectangle r) {
2      Point[] midPoints = new Point[4];
3      for (int i = 0; i < midPoints.length; i++) {
4          midPoints[i] = new Point();
5      }
6      midPoints[0].setLocation(r.x + r.width / 2.0, r.y
7          ); // 上の中点
8      midPoints[1].setLocation(r.x + r.width, r.y + r.
9          height / 2.0); // 右の中点
10     midPoints[2].setLocation(r.x + r.width / 2.0, r.y
11         + r.height / 2.0); // 下の中点
12     midPoints[3].setLocation(r.x, r.y + r.height /
13         2.0); // 左の中点
14     return midPoints;
15 }
```

ソースコード 12: setShortestDistance メソッド

```
1  void setShortestDistance(Rectangle source, Rectangle
2      distance) {
3      Point[] fromMidPoints = getMidPoints(source);
4      Point[] toMidPoints = getMidPoints(distance);
5
6      double min = Double.MAX_VALUE;
7      for (int i = 0; i < 4; i++) {
8          Point from = fromMidPoints[i].getLocation();
9          for (int j = 0; j < 4; j++) {
10              Point to = toMidPoints[j].getLocation();
11              double value = (from.getX() - to.getX())
12                  * (from.getX() - to.getX())
13                  + (from.getY() - to.getY()) * (
14                      from.getY() - to.getY());
15              if (value < min) {
16                  min = value;
17                  start = from;
18                  end = to;
19              }
20          }
21      }
22  }
```

```

16         }
17     }
18 }
19 }
```

次に、経路のパネルに、コストを格納するためのパネルを埋め込むためには、親となるパネルを大きめにする必要がある。そこで、int型のフィールド graceを作り、相対座標にこの値を考慮することで、パネルの大きさに余裕を持たせることができた。

4.2.2 要素に付随する値を入力するための入力ボックスを表示するまで

まず、経路の初期値を得るために Search.java を実行し、得られたノードを以下のような Map で管理した。

ソースコード 13: main メソッドの一部

```

1   for (int i = 0; i < 10; i++) {
2       map.put(node[i], new NodePanel(node[i]));
3 }
```

NodePanel クラスにおいて、このインスタンスを親のパネルとし、ノードのラベルと値を格納するための入力ボックスを配置した。レイアウトには 2 行 1 列の GridLayout クラスを利用した。NodePanel のコンストラクタをソースコード 14 に示す。

ソースコード 14: NodePanel のコンストラクタ

```

1 NodePanel(Node node) {
2     id = counter++;
3     this.node = node;
4     setLayout(new GridLayout(2, 1));
5     setBackground(Color.ORANGE);
6     setBorder(new BevelBorder(BevelBorder.RAISED));
7
8     JLabel label = new JLabel(id + ": " + node.
9         getName());
10    model = new SpinnerNumberModel(node.getHValue(),
11        0, 9999, 1);
12    spinner = new JSpinner(model);
13    spinner.setPreferredSize(new Dimension(50, 25));
```

```
13         add(label);
14         add(spinner);
15     }
```

4.2.3 探索を選択するためのボタンと、実行ボタンを表示するまで

探索を選択するためのボタンには JRadioButton クラスを用いた。複数選択を許可しないために、ButtonGroup クラスを用いた。これらと実行ボタンを含んだパネルを生成する searchButtons メソッドをソースコード 15 に示す。

ソースコード 15: searchButtons メソッド

```
1 JPanel searchButtons() {
2     JPanel p = new JPanel();
3     p.setLayout(new BoxLayout(p, BoxLayout.PAGE_AXIS
4     ));
5
6     radio = new JRadioButton[6];
7     radio[0] = new JRadioButton("幅優先探索");
8     radio[1] = new JRadioButton("深さ優先探索");
9     radio[2] = new JRadioButton("分岐限定法");
10    radio[3] = new JRadioButton("山登り法");
11    radio[4] = new JRadioButton("最良優先探索");
12    radio[5] = new JRadioButton("A*アルゴリズム");
13    ButtonGroup group = new ButtonGroup();
14    for (int i = 0; i < radio.length; i++) {
15        group.add(radio[i]);
16        p.add(radio[i]);
17    }
18
19    JButton button = new JButton("実行");
20    button.addActionListener(this);
21    p.add(button);
22
23    return p;
}
```

4.2.4 入力ボックスやボタンで入力した値を Search.java に反映し、実行するまで

まず、search.java を反映した値を用いて再実行するために、再実行用のメソッド exec を実装した。

次に、実行ボタンを押したときに入力した値を反映するためには、ActionListener インターフェースの actionPerformed メソッドを実装する必要がある。ノードの値、探索の種類、コストの値を全て反映し実行する、actionPerformed メソッドの前半をソースコード 16 に示す。

ソースコード 16: actionPerformed メソッドの前半

```
1  public void actionPerformed(ActionEvent e) {
2      for (NodePanel p : map.values()) {
3          p.getNode().reset();
4          p.update((Integer)p.getModel().getValue());
4          // ヒューリスティック値の変更を反映
5      }
6
7      for (int i = 0 ; i < radio.length; i++){ // 探索
8          // の選択
9          if (radio[i].isSelected()){
10             which = i + 1;
11         }
12
13     for(int i = 0; i < paths.size(); i++) {
14         PathPanel p = paths.get(i);
15         p.update((Integer)p.getModel().getValue());
16         // コストの変更を反映
17     }
18     ArrayList<Node> route = search.exec(which); // 再
实行
```

4.2.5 得られた経路を GUI に反映するまで

再描写には repaint メソッドを用いる必要がある。これにより、paintComponent メソッドの再度呼び出しが行われるため、描写の条件による分岐を paintComponent 側で用意する必要がある。そこで PathPanel クラス内に boolean 型のフィールド pass を用意し、forRepaint メソッド内で状態を確認、更新することで必要な経路のみ色を変えるような仕様にすることができた。また、Search.java の exec メソッドの戻り値を、通るノードのみ格納したリストし、それを用いることで以上の操作を容易にした。

得られた経路を反映する、actionPerformed メソッドの後半をソースコード 17 に示す。

ソースコード 17: actionPerformed メソッドの後半

```
1      for(int i = 0; i < paths.size(); i++) {
2          PathPanel p = paths.get(i);
3          for(int j = 0; j < route.size() - 1; j++) {
4              if(p.forRepaint(route.get(j), route.get(j
+ 1))) {
5                  break;
6              }
7          }
8          p.repaint();
9      }
10 }
```

4.3 実行例

SearchGUI を実行したところ、以下のような画面が得られる。

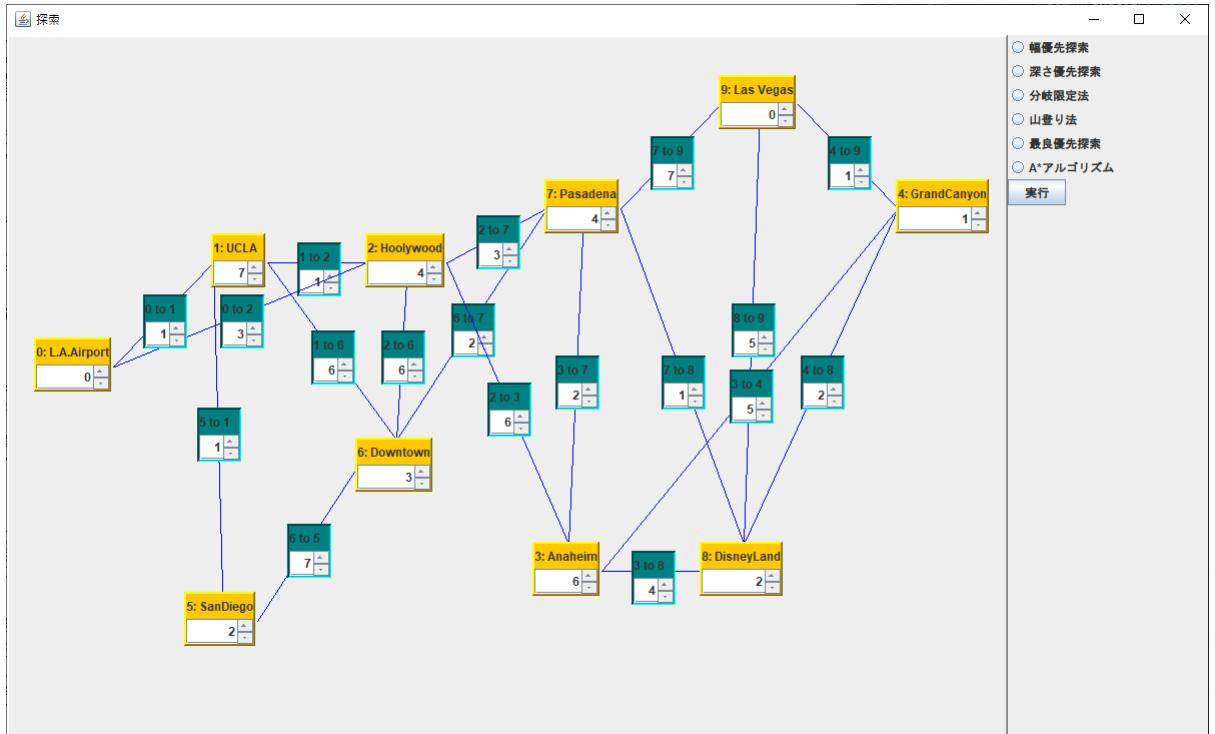


図 22: 初期状態

幅優先探索を選択し、実行したところ、以下のような画面が得られる。

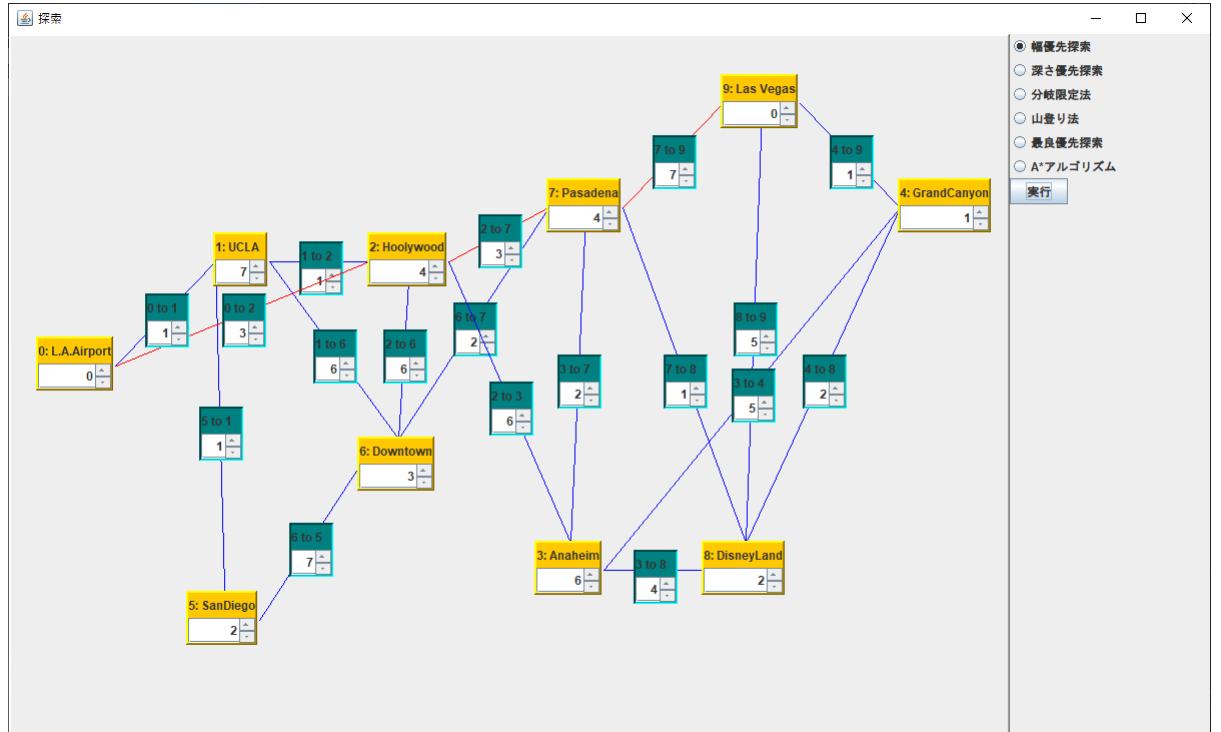


図 23: 幅優先探索の実行

同様にして A*アルゴリズムを実行したところ、以下のような画面が得られる。

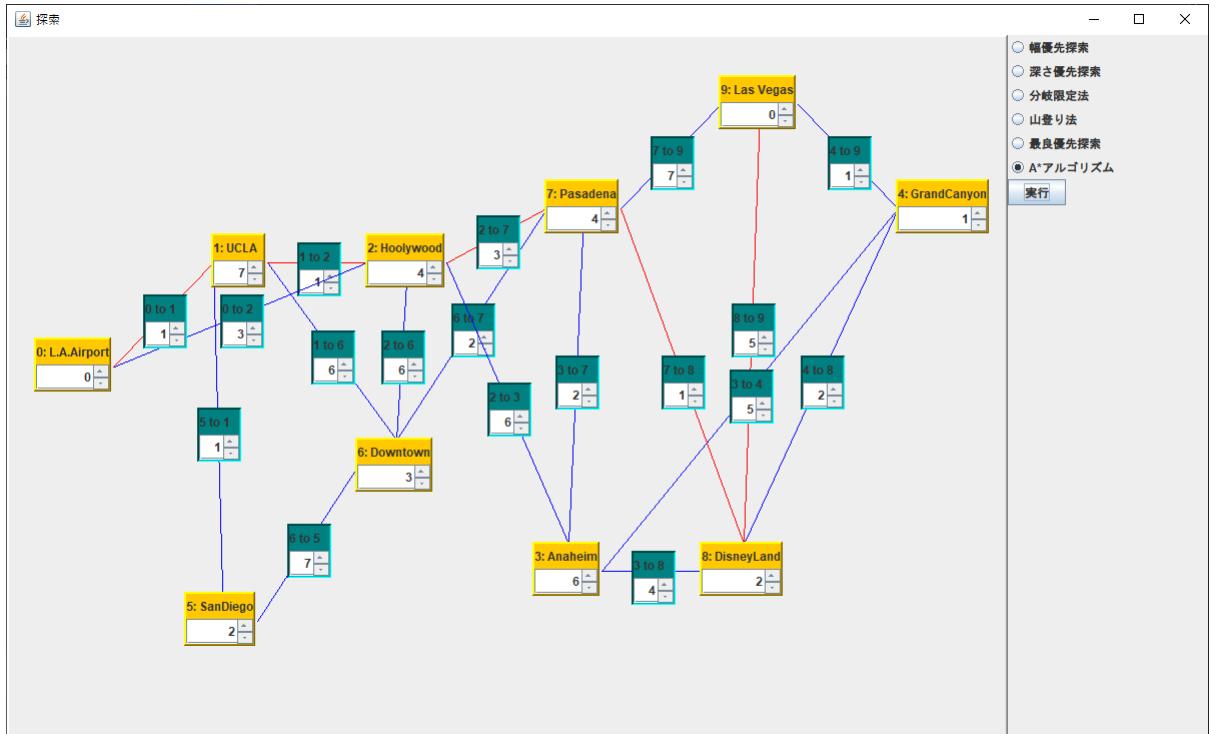


図 24: A*アルゴリズムの実行

4.4 考察

初めて Swing を使ってみて、ボタンや入力ボックスを用いた表形式の GUI は作りやすそうだと感じた一方で、今回実装したような図形式の GUI は大変であると感じた。そもそも、SpringLayout クラスによる相対座標の指定となると、実行と表示を繰り返しながら試行的に作っていかねばならず、また、相対座標というものの自体、パネルを階層的に作っている中で、正しく相対すべき相手との座標になっているかなどの点で間違いやさしいと感じた。

一方で、マウスを使ったパネルの操作から、座標の取得もできるようなので、あらかじめそういういったプログラムを作り、それによる座標の取得と保持を上手に実装できれば、Swing における自由な配置での GUI 制作も、より簡単なものになると考えられる。

また、表示面では、paintComponent メソッドの扱いも難しかった。関連する情報の載っているサイトがあまり見つからず、どのように扱えばよいのか理解するのに時間を要した。このメソッドは実行時に自動的に 1 度だけ呼び出されることが難点だと感じたが、インスタンスごとにこのメソッドを持たせることで、より自由で部分的な実行を可能にすることができた。このように、制約のあるものは、大元から変えてあげることや、java であるならばオブジェクト指向を用いた分解ができるいかを第一に考えることこそが、より迅速な問題解決につながると考えられる。

また、経路の表示には矢印を用いるよう試みたが、先述した相対座標の難しさ等もあり、上手に表示できなかつたため断念した。また、矢印では先端部分の重なり等の問題等もあるため、今回のように”0 to 2”のようなラベルで表現した方が、誤解を減らす上でも役立ったと考えられる。しかし、経路同士の重なりを防ぐことはできないため、より誤解を減らすための方法として、線を曲線にすることや、線を縁取ることでより誤解の少ない表示ができるのではないかと考えられる。

また、今回 NodePanel や PathPanel といったクラスに分解したこと、SearchGUI クラスのコードの複雑さを軽減することができた。これらのクラスは、実装中に分解できるんじゃないかなと思い、後で分解したものであったので、今後はコードを書く前からクラスやメソッドとして取り出せないかを考えるようにすれば、より効率的なオブジェクト指向におけるプログラミングが見込めると考えられる。

Swing におけるパネルについても、どこを一纏めにするかを予め考慮しておくことで、混乱せずに GUI の実装が見込めると考えられる。

5 感想

グループレポートを個人レポートから集約する際に、言葉が重ならないようにしたり、見やすく並び替えたりしたため、思っていた以上に時間がかかってしまった。もっと効率よくレポートをまとめられるようになたいと感じた。

各課題に対する感想は、それぞれの個人レポートを参考にされたい。

参考文献

- [1] 新谷虎松『Javaによる知能プログラミング入門』コロナ社, 2002年.
- [2] 新谷虎松, 大園忠親, 白松俊『知識システムの実装基礎』コロナ社, 2012年.
- [3] Samurai Blog Java のバージョンを確認する方法(バージョンの切り替えも解説)
<https://www.sejuku.net/blog/62105> (2019年10月13日アクセス) .
- [4] Eclipseで使用する JDK を確認します
itdoc.hitachi.co.jp/manuals/link/cosmi_v0870/APFG/EU020018.HTM (2019年10月13日アクセス) .
- [5] Graphviz をインストールする
ruby.kyoto-wu.ac.jp/info-com/Softwares/Graphviz/ (2019年10月13日アクセス) .
- [6] Graphviz の使い方を例題で覚える
<https://www.write-ahead-log.net/entry/2016/03/04/220933> (2019年10月13日アクセス) .
- [7] データのビジュアル化を最小の労力で
www.showa-corp.jp/special/graphtools/graphviz.html (2019年10月13日アクセス) .
- [8] Pynote Graphviz-ノードの属性
pynote.hatenablog.com/entry/pygraphviz-node-attributes#ノードの形状 (2019年10月13日アクセス) .

- [9] Latex コマンド集 表組みの基本
www.latex-cmd.com/fig_tab/table01.html (2019年10月13日アクセス) .
- [10] LaTex 箇条書き -著：LaTex コマンド集
<http://www.latex-cmd.com/struct/list.html> (2019年10月13日アクセス) .
- [11] java CSV 出力 -著：TECH Pin
[https://tech.pjin.jp/blog/2017/10/17/\[java\] csv 出力のサンプルコード](https://tech.pjin.jp/blog/2017/10/17/[java] csv 出力のサンプルコード) (2019年10月13日アクセス) .
- [12] java List 配列処理と変換 -著：Samurai Blog
<https://www.sejuku.net/blog/16155> (2019年10月13日アクセス) .
- [13] Git でブランチを作成する方法 -著：ProEngineer
<https://proengineer.internous.co.jp/content/columnfeature/7633> (2019年10月13日アクセス) .
- [14] Git レポジトリの変更と取得 -著：GitHub ヘルプ
<https://help.github.com/ja/articles/getting-changes-from-a-remote-repository> (2019年10月13日アクセス) .
- [15] 新入社員におくる GitHub でのプロジェクト管理の初歩 -著：hayato ki
<https://qiita.com/gumimin/items/63dcb36d4730213bd63a> (2019年10月7日アクセス) .
- [16] TATSUO IKURA : 『Swing を使ってみよう - Java GUI プログラミング』
<https://www.javadrive.jp/tutorial/> (2019年10月15日アクセス).