知能プログラミング演習 II 課題 1

グループ 07
29114007 池口 弘尚
29114031 大原 拓人
29114048 北原 太一
29114086 飛世 裕貴
29114095 野竹 浩二朗

2019 年 10 月 7 日

提出物　rep1 group07.zip

グループ　グループ 07

1　課題の説明

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．

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．

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　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

課題で与えられた探索手法と、我々が考察した点は以下のとおりである。

1. 幅優先探索法
2. 深さ優先探索法
3. 分枝限定法
4. 山登り法
5. 最良優先探索法
6. A*アルゴリズム

各探索法について、状態空間のパラメータを手動で変更して無限ループに陥らせたり、ステップ数の増減を観察した。またパラメータを Random クラスを用いて変動させ試行を繰り返し、最小コストでゴールにたどり着ける手法を探した。

2.2　実装

もともと与えられた Search.java を以下のように変更した。

状態空間のパラメータを Random クラスで変動させられるように、地点名、ヒューリスティック関数、各エッジのコストと、出発元と行先の番号を配列に保存するようにした。また、手法ごとに同じパラメータで探索が行えるように、保存した配列をもとに状態空間をリセットできるようにした。その仕様により、それぞれの状態空間ごとに最小のコストでゴールにたどり着ける探索手法とその時のステップ数を比較できるようになった。

地点名、ヒューリスティック関数、各エッジのコストと、出発元と行先の番号を配列に保存するようにした部分と、それをもとに状態空間を生成する部分を抜粋したコードは以下のとおりである。

ソースコード 1: SearchRand クラスより抜粋

```
1  // ノード名
2        String[] locations = { "L.A.Airport", "UCLA", "
             Hoolywood", "Anaheim", "GrandCanyon", "SanDiego",
              "Downtown",
3                       "Pasadena", "DisneyLand", "Las Vegas"
                             };
4        // 分岐元のインデックス
5        int[] nodeIndex = { 0, 0, 1, 1, 2, 2, 2, 3, 3, 3,
             4, 4, 5, 6, 6, 7, 7, 8 };
6        // 分岐先のインデックス
7        int[] childIndex = { 1, 2, 2, 6, 3, 6, 7, 4, 7, 8,
              8, 9, 1, 5, 7, 8, 9, 9 };
8        int[] nodeRand;
9        int[] costRand;
10       int randSize = 99;
11       // N 回試行の記録(Method,0.step 1.cost)
12 int[][] record = new int[2][6];
13 public int[][] getRecord() {
14   return record;
15 }
16 SearchRand() {
17   // コストとヒューリスティック関数の決定
18   Random rand = new Random();
19   nodeRand = new int[10];
20   costRand = new int[18];
21   for (int i = 0; i < nodeRand.length; i++) {
22     if (i != 0 && i != nodeRand.length - 1)
23       nodeRand[i] = rand.nextInt(randSize) + 1;
24     else
25       nodeRand[i] = 0;
26   }
27   for (int i = 0; i < costRand.length; i++)
28     costRand[i] = rand.nextInt(randSize) + 1;
29   makeStateSpace();
30 }
31
32 private void makeStateSpace() {
33   // 状態空間の生成
34   NodeRand[] nodelist = new NodeRand[10];
35   for (int i = 0; i < locations.length; i++)
36     nodelist[i] = new NodeRand(locations[i], nodeRand[i]);
```

```
37      for (int i = 0; i < nodeIndex.length; i++)
38        nodelist[nodeIndex[i]].addChild(nodelist[childIndex[i
              ]], costRand[i]);
39      node = nodelist.clone();
40      start = node[0];
41      goal = node[9];
42    }
```

各手法で見つけた経路のコストと、それまでにかかったステップ数を配列に保存し、すべての手法の探索が終わった後にその結果を比較する実装は以下のとおりである。

ソースコード 2: 結果を保存し、比較

```
1         public final int hillClimbing = 3;
2   ...
3   if (success) {
4                         // System.out.println("*** Solution
                              ***");
5                         // printSolution(goal, step);
6                         record[STEP][hillClimbing] = step;
7                         record[COST][hillClimbing] = goal.
                              getGValue();
8                 } else {
9                         //System.out.println("faild\nstep:" +
                               step);
10                        record[STEP][hillClimbing] = step;
11                        record[COST][hillClimbing] = step * (
                              randSize + 1);
12                }
13  ...
14  // 同じ状態空間で探索を行った後
15  int[][] record = place.getRecord();
16  int minCost = getMin(record[1]);// 6手法中の最小コスト
17  for (int j = 0; j < record[1].length; j++) {
18    if (minCost == record[1][j]) {
19      //最小コストのときカウント
20      counts[1][j]++;
21      //分枝限定法よりA*アルゴリズムのステップ数が多いとき
22      //分枝限定法よりステップ数が少ないか
23      if(record[0][2]<record[0][5]){
24        if(record[0][j]<=record[0][2])
25          counts[0][j]++;
26      }else{
27        //A*アルゴリズムよりステップ数が少ないか
28        if(record[0][j]<=record[0][5])
```

```
29            counts[0][j]++;
30          }
31        }
32      }
33    if(record[0][3]==100)hillloop++;
```

2.3 実行例

ランダムに状態空間のパラメータを 1 から 9 の間で変更した試行を繰り返し、最短経路を発見した回数と、より少ないステップ数で最短経路を発見した回数は以下のとおりである。入れ替わっているが、counts[0] は後者、counts[1] は前者である。山登り法が無限ループに陥った回数もカウントした。

```
1   counts[0][0]:206679,counts[1][0]:471074
2   counts[0][1]:110876,counts[1][1]:122587
3   counts[0][2]:203326,counts[1][2]:1000000
4   counts[0][3]:106402,counts[1][3]:106436
5   counts[0][4]:213036,counts[1][4]:251926
6   counts[0][5]:802176,counts[1][5]:848311
7   hillloop:298951
8   counts[0][0]:207610,counts[1][0]:471056
9   counts[0][1]:111085,counts[1][1]:122867
10  counts[0][2]:202782,counts[1][2]:1000000
11  counts[0][3]:106586,counts[1][3]:106610
12  counts[0][4]:213300,counts[1][4]:251970
13  counts[0][5]:803118,counts[1][5]:848982
14  hillloop:298163
```

ランダムに状態空間のパラメータを 1 から 99 の間で変更した場合は以下のようになった。

```
1   counts[0][0]:13493,counts[1][0]:37236
2   counts[0][1]:8747,counts[1][1]:10424
3   counts[0][2]:27577,counts[1][2]:100000
4   counts[0][3]:10212,counts[1][3]:10215
5   counts[0][4]:16967,counts[1][4]:20453
6   counts[0][5]:69439,counts[1][5]:77482
7   hillloop:28461
8   counts[0][0]:13318,counts[1][0]:36973
9   counts[0][1]:8827,counts[1][1]:10537
10  counts[0][2]:27631,counts[1][2]:100000
11  counts[0][3]:10231,counts[1][3]:10235
12  counts[0][4]:17010,counts[1][4]:20508
13  counts[0][5]:69210,counts[1][5]:77383
14  hillloop:28604
```

分枝限定法が必ず最短経路を見つけていることがわかる。

2.4 考察

A*アルゴリズムは必ず最短経路を発見できると勘違いしていたが、ヒューリスティック関数が影響して最短経路ではない経路を発見して終了する場合があることがわかった。もともと、状態空間のノードのつなぎ方は変動していないので条件としてはかなり限定されているが、A*アルゴリズムが最短経路を発見したときは分枝限定法より少ないステップ数で発見できている。また、山登り法が無限ループに陥る確率もある程度収束しているように思われる。

手動で状態空間のパラメータを変更した際の考察については、グループレポートを参考にされたい。

2.5 実装

与えられた Search.java において、2. 4. 5. の手法を用いると、無限ループに陥ってしまう状態空間を作成した。

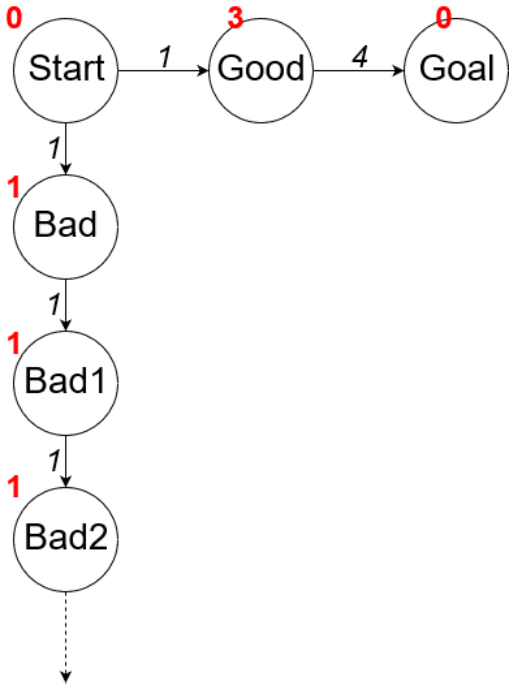

図 1: 無限ループに陥る状態空間

深さ優先探索法に関して、反復深化深さ優先探索とすることにより、先述の状態空間においても解を求めることができるようにした。

2.5.1 BadNode

まず、このクラスは、与えられた Node クラスのサブクラスである。オーバーライドしたメソッドは以下の 1 つ。

- getChildren: Node クラスでは内部メンバ children を返すが、BadNode では新しい BadNode インスタンスを addChild してから children を返す。

BadNode クラスの実装を以下に示す。

ソースコード 3: BadNode クラス

```
1       //無限ループを形成するノード
2       class BadNode extends Node {
3
4           static int id = 0; //名付け用ID
5           BadNode(String theName, int theHValue) {
6               super(theName, theHValue);
7           }
8
9           @Override
10          public ArrayList<Node> getChildren() {
11              addChild(new BadNode("Bad" + ++id, hValue), 1);
                    //新しいBadNode を children に追加
12              return super.getChildren(); //children を返す
13          }
14      }
```

2.5.2 深さ優先探索

深さ優先探索をする depthFirst メソッドを、以下のように改良した。

ソースコード 4: depthFirst メソッド

```
1   /***
2     * 反復深化深さ優先探索
3     */
4   public void depthFirst() {
5     ArrayList<Node> open = new ArrayList<Node>();
6     open.add(start);
7     ArrayList<Node> newOpen = new ArrayList<>(); //深さが 1レ
          ベル深いオープンリスト
```

```
8     ArrayList<Node> closed = new ArrayList<Node>();
9     boolean success = false;
10    int step = 0;
11
12    for (;;) {
13      System.out.println("STEP:" + (step++));
14      System.out.println("OPEN:" + open.toString());
15      System.out.println("NEWOPEN:" + newOpen.toString());
16      System.out.println("CLOSED:" + closed.toString());
17      // open は空か？
18      if (open.size() == 0) {
19        //newOpen も空か?
20        if (newOpen.size() == 0) {
21            success = false;
22            break;
23        }
24        //探索を 1レベル深くし、newOpen をリセット
25        open = newOpen;
26        newOpen = new ArrayList<>();
27      } else {
28        // open の先頭を取り出し node とする.
29        Node node = open.get(0);
30        open.remove(0);
31        // node は ゴールか？
32        if (node == goal) {
33            success = true;
34            break;
35        } else {
36        // node を展開して子節点をすべて求める.
37        ArrayList<Node> children = node.getChildren();
38        // node を closed に入れる.
39        closed.add(node);
40        // 子節点 m が open にも newOpen にも closed にも含ま
              れていなければ,
41        // 以下を実行.幅優先探索と異なるのはこの部分である.
42        // j は複数の子節点を適切に
              newOpen の先頭に置くために位置
43        // を調整する変数であり,一般には展開したときの子節点
44        // の位置は任意でかまわない.
45        int j = 0;
46        for (int i = 0; i < children.size(); i++) {
47          Node m = children.get(i);
48          if (!open.contains(m) && !newOpen.contains(m) && !
                closed.contains(m)) {
49            // m から node へのポインタを付ける
50            m.setPointer(node);
```

```
51              //newOpen に追加
52              if (m == goal) {
53                  newOpen.add(0, m);
54              } else {
55                  newOpen.add(j, m);
56              }
57              j++;
58            }
59          }
60          }
61        }
62      }
63      if (success) {
64          System.out.println("*** Solution ***");
65          printSolution(goal);
66      }
67    }
```

2.6 実行例

BadNode を用いた状態空間に対し、深さ優先探索、反復深化深さ優先探索、山登り法を実行した結果を以下に示す。

ソースコード 5: 深さ優先探索

```
1  (前略)
2  STEP:848
3  OPEN:[Bad846(h:1), Good(h:3)]
4  CLOSED:[Start(h:0), Bad(h:1), Bad0(h:1), Bad1(h:1), Bad2(h
       :1), (中略), Bad843(h:1), Bad844(h:1), Bad845(h:1)]
5  STEP:849
6  OPEN:[Bad847(h:1), Good(h:3)]
7  CLOSED:[Start(h:0), Bad(h:1), Bad0(h:1), Bad1(h:1), Bad2(h
       :1), (中略), Bad844(h:1), Bad845(h:1), Bad846(h:1)]
8  STEP:850
9  (以下略)
```

ソースコード 6: 反復深化深さ優先探索

```
1  Depth First Search
2  STEP:0
3  OPEN:[Start(h:0)]
4  NEWOPEN:[]
5  CLOSED:[]
6  STEP:1
```

```
7   OPEN:[]
8   NEWOPEN:[Bad(h:1), Good(h:3)]
9   CLOSED:[Start(h:0)]
10  STEP:2
11  OPEN:[Bad(h:1), Good(h:3)]
12  NEWOPEN:[]
13  CLOSED:[Start(h:0)]
14  STEP:3
15  OPEN:[Good(h:3)]
16  NEWOPEN:[Bad0(h:1)]
17  CLOSED:[Start(h:0), Bad(h:1)]
18  STEP:4
19  OPEN:[]
20  NEWOPEN:[Goal(h:0), Bad0(h:1)]
21  CLOSED:[Start(h:0), Bad(h:1), Good(h:3)]
22  STEP:5
23  OPEN:[Goal(h:0), Bad0(h:1)]
24  NEWOPEN:[]
25  CLOSED:[Start(h:0), Bad(h:1), Good(h:3)]
26  *** Solution ***
27  Goal(h:0) <- Good(h:3) <- Start(h:0)
```

ソースコード 7: 山登り法

```
1   (前略)
2   [Bad199906(h:1)]
3   [Bad199907(h:1)]
4   [Bad199908(h:1)]
5   [Bad199909(h:1)]
6   (以下略)
```

2.7 考察

深さ優先探索、山登り法の場合、解 (Goal) のほうに進む Good ノードには進まず、解が存在しないがヒューリスティクス関数がより良い Bad ノードに進んでしまう。しかしながら、反復進化深さ優先探索により、探索するノードの深さを制限することにより、深さ優先探索で探索できない状態空間でも解を求めることができる。

2.8 実装

今回の課題ではプログラム自体は変えずに、状態空間のパラメータのみを変化させた。

2.9 実行例

まず、パラメータに変更を加える前の各探索手法における STEP 数と探索結果を以下に示す。

探索手法	STEP 数	探索結果
幅優先探索	7	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport
深さ優先探索	6	LasVegas <- Pasadena <- Downtown <- UCLA <- L.A.Airport
分岐限定法	8	LasVegas <- DisneyLand <- Pasadena <- Hoolywood <- UCLA <- L.A.A
山登り法	-	探索失敗
最良優先探索	6	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport
A*アルゴリズム	8	LasVegas <- DisneyLand <- Pasadena <- Hoolywood <- UCLA <- L.A.A

この結果より山登り法において探索が失敗していることがわかる。これはあるノードにおいて次のノードのヒューリスティックスの値のみで探索をしていく山登り法では UCLA、Downtown、Sandiego において無限ループが生じるためである。以下では、分枝限定法に関しては STEP 数をより小さく、山登り法に関しては探索が成功するようにパラメータを変更していく。なお、今回のプログラムにおいて幅優先探索と深さ優先探索ではコストやヒューリスティックスの値は考慮しておらず、パラメータの変更が影響を及ぼさないため、記述は省略する。

まず、分枝限定法における探索 STEP 数を小さくするために、UCLA から Downtown へのコストを 11、Hoolywood から Anaheim・Downtown へのコストを 10 とした。この時の結果を以下に示す。

探索手法	STEP 数	探索結果
分岐限定法	6	LasVegas <- DisneyLand<- Pasadena <- Hoolywood <- UCLA <- L.A.Ai
山登り法	-	探索失敗
最良優先探索	6	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport
A*アルゴリズム	7	LasVegas <- DisneyLand <- Pasadena <- Hoolywood <- UCLA <- L.A.A

この変更において A*アルゴリズムが最短経路を発見することは保障されているため、分枝限定法によって最短経路を最小 STEP 数で探索されていることがわかる。

次に山登り法による探索を成功させるために初期パラメータから、Sandiego のヒューリスティックスの値を 5 に変更した。この時の結果を以下に示す。

この結果から変更によって山登り法による探索が成功することが確認できた。しかしこの変更においても A*アルゴリズムが最短経路を発見することは保証されているため、山登り法により最短経路が探索されていないことがわかる。

探索手法	STEP 数	探索結果
分岐限定法	8	LasVegas <- DisneyLand <- Pasadena <- Hoolywood <- UCLA <- L.A.A
山登り法	5	LasVegas <- Pasadena <- Downtown <- Hoolywood <- L.A.Airport
最良優先探索	5	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport
A*アルゴリズム	8	LasVegas <- DisneyLand <- Pasadena <- Hoolywood <- UCLA <- L.A.A

2.10 考察

本課題において分枝限定法に関して上記のように、変更を行った結果探索 STEP 数を小さくすることができた。その理由として、初期値においては最短経路には含まれない Downtown への探索を行っており、変更によってその余分な処理を省くことができたからだと考えられる。

また、分枝限定法と同様に A*アルゴリズムでも最短経路の探索に成功しているが、STEP 数に関しては分枝限定法の方が小さくなっている。今回のように経路のパラメータによっては、無駄な処理を省略する分枝限定法の方が探索を早く終えることができる。

そして山登り法に関しては、今回のプログラムにおいて繰り返し回避のための工夫がなされていないため、初期値のように正しくパラメータを設定していなければ無限ループを起こしてしまう。またゴールノードを子ノードに持つノードに到達すると、最もヒューリスティックスの値が小さいノードがゴールノードとなってしまい、ゴールまでのコストに関係なくゴールへ探索を進めてしまう。今回の変更後の探索において、最短経路探索のためにはゴールノードを子ノードに持つ Pasadena を経由しなければならないため、正しく最短経路が探索できなかったと考えられる。そのため、今回の経路探索において山登り法により正しい最短経路を求めるには Pasadena から LasVegas へのコストを小さくする必要があったと考えられる。また今回はプログラム自体への変更を行わずにパラメータを変更することで無限ループに陥ることを防いだが、繰り返し探索を防止することで無限ループを防ぐことができると考える。

2.11 実装

プログラムのコード本体は変えず、状態空間のパラメータのみを変化させた。

2.12 実行例

幅優先探索、深さ優先探索については、パラメータを変化させても結果は変わらないため、省略する。また、実行結果すべてを載せてしまうと無駄に長くなってしまうため、STEP 数と採取的にどのルートになったかのみを記す。

まず、最良優先が成功する場合として、コスト、評価値を変化させた。分

探索手法	STEP 数	探索結果
分岐限定法	7	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport
山登り法	-	LasVegas <- Pasadena <- Hoolywood <- UCLA <- L.A.Airport
最良優先探索	4	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport
A*アルゴリズム	5	LasVegas <- Pasadena <- Hoolywood <- L.A.Airport

岐限定法、最良優先探索、A*アルゴリズムが成功していることが分かる。
ステップ数を比較すると、最良優先探索が少なく、分岐限定法が多くなっている。最良優先探索が上手くいくようにパラメータを調整したので、そのステップ数が少なくなることは当然である。

次に、A*アルゴリズムが失敗するように Pasadena の評価値を 10 とした。
分岐限定法は成功しているが、山登り法は無限ループ、最良優先探索と A*ア

探索手法	STEP 数	探索結果
分岐限定法	7	LasVegas <- DisneyLand<- Pasadena <- Hoolywood <- UCLA <- L.A.Ai
山登り法	-	-
最良優先探索	6	LasVegas <- GrandCanyon <- Anaheim <- Hoolywood <- L.A.Airport
A*アルゴリズム	5	LasVegas <- GrandCanyon <- Anaheim <- Hoolywood <- UCLA <- L.A.

ルゴリズムはゴールに到達しているものの、最短のルートではなかった。

2.13 考察

最良優先探索の場合、ゴールノードを子ノードに持つノードにたどり着くと、そこからゴールまでのコストに関係なくゴールへと行ってしまう。今回の場合、初期のパラメータだと Pasadena からは一度 DisneyLand を経由しなければならないが、最良優先探索ではそれができないので、成功する例として、コストを無理やり減らすということをしなければならなかった。実際は、コストを変えることは難しいため、ヒューリスティックスのみでは不十分であることが分かる。
A*アルゴリズムは、ヒューリスティックスに加えて、コストも考慮することができるため、最良優先探索では成功できない場合でもしっかりと探索することができる。しかし、ヒューリスティックスに加えて、コストも見なければならないのでステップ数は増えてしまっている。

3　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．

課題 1-2 は実装を伴わない課題であるため、考察のみ記す。

3.1　考察

今回は、扱っているデータがそれほど機密性の高いものではなかったため、連絡の手段としてはより気軽に使えることを重視して、LINE を使用した。普段から使っているツールを使用することによって、より円滑なコミュニケーションを図った。しかし、それぞれのメンバーが進捗を報告しているわけではなかったので、どれだけ進捗しているのかわからないという状況になってしまった。そのため、進捗管理にはどれだけ進んでいるのかなどがわかる Trello なども活用していくことが重要だと思う。成果物共有に関しては、git を使用した。git を使ったことがないメンバーが多かったので、今回はプルとプッシュのみで作業を行った。そのためにそれぞれの作業をフォルダごとに分けて管理した。今回の課題では、あまりソースコードを変える必要がなかったためできたことだが、次回の課題からはソースコードを書くことになると思うので、ブランチを分けて作業していく必要があると考える。

3.2　考察

今回は課題の分量が少なかったので、演習時間内に終わらなかった分は後日全員で集まって課題を進めた。連絡手段として LINE を用いた。今後の予定としては、GitHub を用いて進捗状況の共有をできるようにしたい。

3.3　考察

演習時間内に終わらなかった分は後日全員で集まって課題を進めた。連絡手段として LINE を、ファイル管理に GitHub を用いた。

3.4　考察

事務連絡については LINE を用いた。ファイルは GitHub を用いて共有した。LINE に関しては、ずっと使っているものなので特に不便を感じることは無かった。GitHub はファイルの共有するため、初めて使った。まだまだ機能についてあまり分かっていないので少しずつ使いこなせるようにしていきたい。

4　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.

4.1　実装

まず、新たに追加したクラスは以下の

- DrawArrow クラス: Path2D.Float を継承。向きがある矢印の描画をするためのクラスであり、参考文献のコードを利用した。
- FrameBase クラス: JFrame を継承。コンストラクタのみ実装しており、JFrame の各種設定が 1 行でできるようにしてある。
- GraphPanel クラス: JPanel を継承。各種グラフを描画するためのメソッドが実装されている。
- MakeGUI クラス: 各種 GUI を作成するためのクラス。

DrawArrow クラスについては参考文献 [?] をそのまま用いた。

FrameBase クラスでは、JFrame のサイズやウィンドウを閉じたときにプログラムを終了することなどを一括して設定するためのコンストラクタを以下のように実装した。

ソースコード 8: FrameBase のコンストラクタ

```
1        //位置を指定
2    public FrameBase(String title,int x,int y,int width,int
         height) {
3                setTitle(title);
4                setBounds(x, y, width, height);
5                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
                     );
6        }
7        //中央に表示
8        public FrameBase(String title,int width,int height) {
9                setTitle(title);
10               setSize(width, height);
11               setLocationRelativeTo(null);
12               setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
                     );
13       }
```

GraphPanel クラスでは、探索のグラフを表示するための様々なメソッドが実装してある。探索木を表示することに関連する makeTreeList,addTreeList,selectTreeNode, removeNodeFromTree,addLeaf,addGoal は以下のように実装した。

ソースコード 9: 探索木関連のメソッド

```
1         //ルートノードを作成
2         public void makeTreeList(Node root) {
3                 nodeMap = new HashMap<>();
4                 //clear();
5                 DefaultMutableTreeNode rootNode = new
                      DefaultMutableTreeNode(root);
6                 nodeMap.put(root.name, rootNode);
7                 tree = new JTree(rootNode);
8                 add(tree);
9                 setVisible(false);
10                setVisible(true);
11        }
12        //p を c の親にする
13        public void addTreeList(Node p, Node c) {
14                DefaultMutableTreeNode parent = nodeMap.get(p.
                      name);
15                if (parent == null) {
16                        System.out.println("error");
17                        return;
18                }
19                DefaultMutableTreeNode child = new
                      DefaultMutableTreeNode(c);
20                parent.add(child);
21                nodeMap.put(c.name, child);
22                DefaultTreeModel m = (DefaultTreeModel) tree.
                      getModel();
23                m.reload();
24                for (int i = 0; i < tree.getRowCount(); i++)
                       {
25                        tree.expandRow(i);
26                }
27        }
28
29    //node を選択状態にする
30        public void selectTreeNode(Node node) {
31                DefaultMutableTreeNode p = nodeMap.get(node.
                      name);
32                tree.setSelectionPath(new TreePath(p.getPath
                      ()));
33        }
34
```

```
35          //c をルートとする木を親ノードから除く
36          public void removeNodeFromTree(Node c) {
37                  DefaultMutableTreeNode child = nodeMap.get(c.
                        name);
38                  if (child != null) {
39                          child.removeFromParent();
40                  }
41          }
42
43          //p にリーフノードをつける
44          public void addLeaf(Node p) {
45                  DefaultMutableTreeNode parent = nodeMap.get(p.
                        name);
46                  if (parent == null) {
47                          System.out.println("error");
48                          return;
49                  }
50                  parent.add(new DefaultMutableTreeNode("leaf
                        "));
51                  DefaultTreeModel m = (DefaultTreeModel) tree.
                        getModel();
52                  m.reload();
53                  for (int i = 0; i < tree.getRowCount(); i++)
                         {
54                     tree.expandRow(i);
55                  }
56          }
57
58          //p に goal ノードをつける
59          public void addGoal(Node p) {
60                  DefaultMutableTreeNode parent = nodeMap.get(p.
                        name);
61                  if (parent == null) {
62                          System.out.println("error");
63                          return;
64                  }
65                  parent.add(new DefaultMutableTreeNode("Goal
                        "));
66                  DefaultTreeModel m = (DefaultTreeModel) tree.
                        getModel();
67                  m.reload();
68                  for (int i = 0; i < tree.getRowCount(); i++)
                         {
69                     tree.expandRow(i);
70                  }
71          }
```

また、局所的な解法に関連するメソッド makeLocalGraph は以下のように実装した。

ソースコード 10: makeLocalGraph

```
1         public void makeLocalGraph(Node root) {
2                 setLayout(null);
3                 removeAll();
4                 JLabel rootlab = makeNodeLabel(root,Color.RED,
                       150, 200);
5                 int num = root.getChildren().size();
6                 if (num == 1) {
7                         JLabel child = makeNodeLabel(root.
                              getChildren().get(0),Color.BLUE,
                              700, 200);
8                         add(child);
9                 } else {
10                        int t = 400/(num-1);
11                        for (int i = 0; i < num; i++) {
12                                JLabel child = makeNodeLabel(
                                      root.getChildren().get(i),
                                      Color.BLUE, 700, t*i);
13                                add(child);
14                        }
15                }
16                add(rootlab);
17                setVisible(false);
18                setVisible(true);
19        }
```

上記の GraphPanel を利用して、必要な GUI を作成するクラス MakeGUI は以下のように実装した。

ソースコード 11: MakeGUI

```
1 public class MakeGUI {
2
3         public static void MakeChooseSearchGUI(ActionListener
              listener, String[] searchNames) {
4                 FrameBase frame = new FrameBase("test", 500,
                      500);
5                 JPanel panel = new JPanel();
6                 for (int i = 0; i < searchNames.length; i++)
                       {
7                         JButton but = new JButton(searchNames[
                              i]);
8                         but.addActionListener(listener);
9                         but.setActionCommand(searchNames[i]);
```

```
10                      panel.add(but);
11              }
12              panel.setLayout(new GridLayout(searchNames.
                    length, 1));
13              frame.getContentPane().add(panel);
14              frame.setVisible(true);
15      }
16
17      public static GraphPanel MakeSearchGUI(ActionListener
            listener) {
18              FrameBase frame = new FrameBase("test", 1000,
                     500);
19              GraphPanel gPanel = new GraphPanel(1000,
                    500);
20              JPanel p = new JPanel();
21              JButton btn1 = new JButton("閉じる");
22              btn1.addActionListener(listener);
23              btn1.setActionCommand("close");
24              JButton btn2 = new JButton("１ステップ");
25              btn2.addActionListener(listener);
26              btn2.setActionCommand("step");
27              JButton btn3 = new JButton("最後まで");
28              btn3.addActionListener(listener);
29              btn3.setActionCommand("last");
30
31              p.add(btn1);
32              p.add(btn2);
33              p.add(btn3);
34
35              JPanel p2 = new JPanel();
36              JLabel label1 = new JLabel("Step :");
37              JLabel label2 = new JLabel();
38              p2.add(label1);
39              p2.add(label2);
40              gPanel.SetStepLabel(label2);
41
42              frame.getContentPane().add(p2, BorderLayout.
                    NORTH);
43              frame.getContentPane().add(p, BorderLayout.
                    SOUTH);
44              frame.getContentPane().add(gPanel,
                    BorderLayout.CENTER);
45              frame.pack();
46              frame.setVisible(true);
47
48              return gPanel;
```

```
49            }
50 }
```

4.2 実行例

実行すると次のような GUI が表示される。

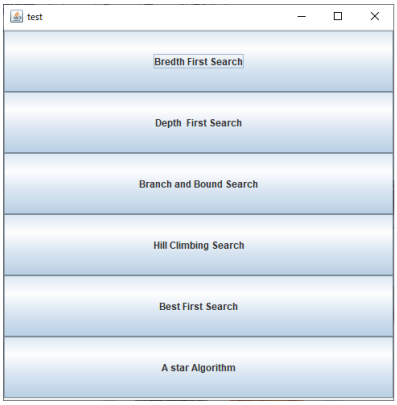

図 2: 探索方法の選択

この中から行いたい探索方法を選んでクリックする。探索が終わると次のようになる。

4.3 考察

今回の実装では swing を使用して GUI を作成した。swing には、ウィンドウや各種のボタンなどは取り揃えてあった。しかし、元々描画をするためのものではないようなので、図形を描画したり、レイアウトの位置を座標で指定したいときには新しいクラスを作った方が良いと考えられる。この実装では、レイアウトの方が間に合わなかったため、探索木は JTree を利用した。

今回作った GUI では、1 ステップごとに状況を確認できるように 1 ステップごとに進む機能が実装されている。この実装ではビジーウェイトなどを使わないようにするため、スレッドを分けて wait と notify で実現している。それらは以下のように実装した。

ソースコード 12: wait 部分

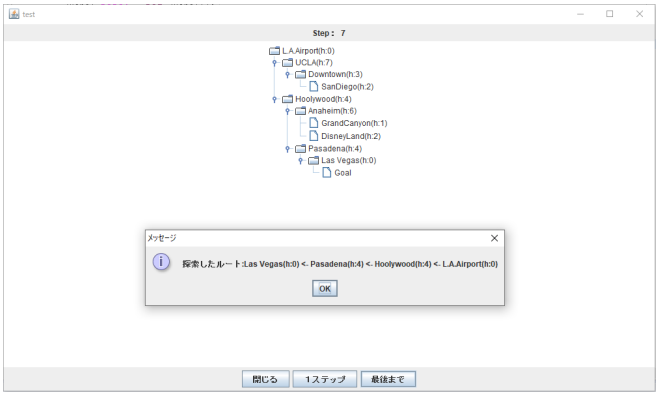

図 3: 探索終了

```
synchronized public void breadthFirst() {
  ...
  for (;;) {
    //1ステップごとに進むかどうか
    if (isStop) {
      try {
        wait();
      } catch (InterruptedException e) {
        // TODO 自動生成された catch ブロック
        e.printStackTrace();
      }
    }
  ...
}
```

ソースコード 13: notify 部分

```
    ActionListener listener = new ActionListener() {

        //ボタンを押したときに呼ばれるメソッド
        @Override
        public void actionPerformed(ActionEvent e) {
            String cmd1 = e.getActionCommand();
            switch (cmd1) {
            //プログラムを終了する
            case "close":
                System.exit(0);
                break;
```

```
12              //1ステップ進む
13              case "step":
14                  synchronized (sh) {
15                      sh.notifyAll();
16                  }
17                  break;
18              //最後まで進む
19              case "last":
20                  sh.isStop = false;
21                  synchronized (sh) {
22                      sh.notifyAll();
23                  }
24                  break;
25              default:
26                  break;
27              }
28          }
29      };
```

wait や notify は Object に対して働くため、Runnnable を作ってそこに Search のメソッドを入れるという方法ではうまくいかなかった。これは Search のインスタンスと Runnnable のインスタンスが別であったために生じたことであると考えられる。これを解決するため、Search に Runnnable を implement し、それを Thread の中に入れることによって、この問題を解決することができた。

5　感想

5.1

自分にとって、締め切りギリギリを攻めるのはいつものことであるが、グループワークとなると甚だ迷惑なので、しっかり時間をとって早めに済ませるようにしたい。グループ間のコミュニケーションについては、うまくアイスブレイク活動を行ったので話しやすい関係を築けた。

5.2

ほぼ初対面の人とのグループワークに慣れておらず、スムーズに課題を進めることができなかった。2 か月という長くない時間だが、グループのみんなと仲良くなりたい。

参考文献

[1] 知能処理学の講義スライド、主に分枝限定法の部分

[2] 矢印を描画 -JAVA で矢印を描画したいのですが、どうしたらいいのかわか- Java — 教えて!goo https://oshiete.goo.ne.jp/qa/4014364.html （2019 年 10 月 7 日アクセス）.

[3] 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 7 日アクセス）.