Experimental report for the 2021 COM1005 Assignment: The Rambler's Problem*

Jordan Pownall

November 21, 2021

1 Introduction

The Ramblers Problem is to determine the best walking route from a start point to a goal point, given a terrain map for the walking area. The best route for the Rambler is a route that requires the least changes in altitude. Therefore the aim of any search algorithm for the Ramblers Problem is to find a route from a start point to a goal with the lowest cost possible, the cost being worked out by the change in altitude.
In this assignment I will be using 2 different algorithms to find the optimal route for the Rambler and testing each algorithms efficiency in doing so and whether they can be adapted to perform better.

2 Solving The Ramblers Problem using my Branch-and-Bound

2.1 Branch-and-Bound implementation - The Code

Branch-and-Bound (BB) works by finding all of the possible paths from the current open nodes to the end nodes. After the first exploration, a maximum global cost will be set and therefore any routes that cost past this will be ignored to increase efficiency.
To implement my BB algorithm I made the following classes:

*https://gitlab.com/jord$_p$$ownall/com1005 - assignment$

2.1.1 RamblersSearch

A class that inherits methods from the Search class.
This class is for detecting and drawing what map is being used, and what the goal state is.

```
public class RamblersSearch extends Search {

    private TerrainMap map; // map we're searching
    private Coords goal; // goal coordinates

    public TerrainMap getMap() {
      return map;
    }

    public Coords getGoal() {
      return goal;
    }

    public RamblersSearch(TerrainMap m, Coords g) {
      map = m;
      goal = g;
    }
```

Figure 1: The RamblersSearch code

This is done by implementing an object of RamblersSearch with the parameters being the goal and the map. These parameters can then be called using the methods getGoal and getMap in the other classes to find out what the goal and map is.

2.1.2 RamblersState

A class that inherits methods from the SearchState class.
The purpose of this class is to find the successor states and also to test if the current coordinates are equal to the goal state. The RamblersState class is the most important of the 3 classes.

```java
public class RamblersState extends SearchState {

    // coordinates for this state
    private Coords coords;

    // constructor
    public RamblersState(Coords c, int lc) {
      coords = c;
      localCost = lc;
    }

    // accessor
    public Coords getCoordinates() {
      return coords;
    }

    // goalPredicate
    public boolean goalPredicate(Search searcher) {
      RamblersSearch msearcher = (RamblersSearch) searcher;
      Coords tar = msearcher.getGoal(); // get target coordinates
      return (coords.getx() == tar.getx() && coords.gety() == tar.gety());
    }
    private int getLocalCost(int y, int x, int nY, int nX, TerrainMap map) {
        int fHeight = map.getTmap()[y][x];
        int sHeight = map.getTmap()[nY][nX];
        if (sHeight <= fHeight) {
            localCost = 1;
        } else {
            int absolute = Math.abs(sHeight - fHeight);
            localCost = 1 + absolute;
        }
        return localCost;
    }

    public boolean sameState(SearchState s2) {
      RamblersState ms2 = (RamblersState) s2;
      Coords ms2C = ms2.getCoordinates();
      return (coords.getx() == ms2C.getx() && coords.gety() == ms2C.gety());
    }
```

Figure 2: The RamblersState tester methods

These are methods in the RamblersState class that can do numerous operations for the certain states, meaning checking if it is equal to another state or if it is a goal, and retrieving its coordinates and its cost to another node. The getLocalCost is neccessary for the calculation of the route in the getSuccessors.

```java
  public ArrayList<SearchState> getSuccessors(Search searcher) {
    RamblersSearch msearcher = (RamblersSearch) searcher;
    TerrainMap map = msearcher.getMap();
    ArrayList<SearchState> succs = new ArrayList<SearchState>();
    int y = coords.gety();
    int x = coords.getx();

    if (y > 0) {
        int y1 = y - 1;
        int lc = getLocalCost(y,x,y1,x, map);
        Coords c = new Coords(y1,x);
        succs.add((SearchState) new RamblersState(c,lc));
    }
    if (x > 0) {
        int x1 = x - 1;
        int lc = getLocalCost(y,x,y,x1, map);
        Coords c = new Coords(y,x1);
        succs.add((SearchState) new RamblersState(c,lc));
    }
    if (y < map.getDepth()-1) {
        int y1 = y + 1;
        int lc = getLocalCost(y,x,y1,x, map);
        Coords c = new Coords(y1,x);
        succs.add((SearchState) new RamblersState(c,lc));
    }
    if (x < map.getWidth()-1) {
        int x1 = x + 1;
        int lc = getLocalCost(y,x,y,x1, map);
        Coords c = new Coords(y,x1);
        succs.add((SearchState) new RamblersState(c,lc));
    }
    return succs;
  }
```

Figure 3: The RamblersState getSuccessors method

The most important part of this class is where it finds the successer nodes, by finding out if the node is at a boundary (the edge of a map), finding where it can go from there. If the current node is at a boundary, for example west, it will only find nodes to the north, south and east and create a list to be returned from these nodes (to be added to the open list). Otherwise, if not at a boundary, it will submit to the open list nodes from all directions.

2.1.3 RunRamblersBB

BB meaning Branch and Bound, this class is what uses both the RamblersSearch and RamblersState together to make a map traversal algorithm.

```
public class RunRamblersBB {


    public static void main(String[] args) {
    TerrainMap tm = new TerrainMap("tmc.pgm");

    RamblersSearch searcher = new RamblersSearch(tm, new Coords(5,8));
    RamblersState initialState = new RamblersState(new Coords(5,9), 0);

    String result = searcher.runSearch(initialState, "B&B");
    System.out.println(result);

    }
```

Figure 4: The RunRamblersBB code

2.2 The Results

When running the algorithm from the car park (coordinates (7,0)) to the col (coordinates (5,8)), the BB search completes within 80 iterations with the following results:

```
Search Succeeds
Efficiency 0.1375
Solution Path

node with state (Ramblers State: (7,0)) parent state (null) local cost (0) global cost (0)
node with state (Ramblers State: (7,1)) parent state (Ramblers State: (7,0)) local cost (1) global cost (1)
node with state (Ramblers State: (6,1)) parent state (Ramblers State: (7,1)) local cost (31) global cost (32)
node with state (Ramblers State: (5,1)) parent state (Ramblers State: (6,1)) local cost (11) global cost (43)
node with state (Ramblers State: (5,2)) parent state (Ramblers State: (5,1)) local cost (11) global cost (54)
node with state (Ramblers State: (5,3)) parent state (Ramblers State: (5,2)) local cost (1) global cost (55)
node with state (Ramblers State: (5,4)) parent state (Ramblers State: (5,3)) local cost (21) global cost (76)
node with state (Ramblers State: (5,5)) parent state (Ramblers State: (5,4)) local cost (1) global cost (77)
node with state (Ramblers State: (5,6)) parent state (Ramblers State: (5,5)) local cost (1) global cost (78)
node with state (Ramblers State: (5,7)) parent state (Ramblers State: (5,6)) local cost (11) global cost (89)
node with state (Ramblers State: (5,8)) parent state (Ramblers State: (5,7)) local cost (11) global cost (100)
```

Figure 5: The results from BB search from C(7,0) to C(5,8)

2.3 Assessing the efficiency of my branch-and-bound search algorithm

The figure for efficiency, as shown, is 0.1375. This is quite low which means this algorithm is not very good for finding the best route quickly. This may be because there is nothing for the search to indicate which of the successor nodes may be the next to explore, so explores them at random.

3 Solving Ramblers Problem using my A* implementation

3.1 Overall A* implementation

The difference between an A* algorithm and a BB algorithm is that A* is a combination of BB and another algorithm called Best First (better known as Greedy). This means that A* takes into account heuristics, which are estimates of the global cost from the current node to the goal node. It then finds out which of the next open nodes to select by finding which one has the lowest figure for the estimated remaining cost. I have used two different calculations for the estimated remaining cost, these are:

1. The absolute height difference to the goal node from the successor nodes.

2. The estimated closeness to the goal node, better known as the 'as the crow flys' method (so not taking into account the change in altitude).

This gives the following changes to the above implemented classes:

3.1.1 RamblersState

Very similar to Figure 2, there is now an added constructor for the RamblersState being the estimatedRemainingCost method that calculates the heurustics from the successor states to the goal state. Figures 6 and 7 show the two different methods used to calculate the estimatedRemainingCost.

The acquired figure is then used in the Search class to select the next node from the successer node list that has the lowest estimated remaining cost.

```
private int getEstRemCost(int y, int x, int gY, int gX, TerrainMap map) {
    int height = map.getTmap()[y][x];
    int gHeight = map.getTmap()[gY][gX];
    if (gHeight <= height) {
        estRemCost = 1;
    } else {
        int absolute = Math.abs(gHeight - height);
        estRemCost = 1 + absolute;
    }
    return estRemCost;
}
```

Figure 6: estimatedRemainingCost method for absolute height difference

```
private int getEstRemCost(int y, int x, int gY, int gX, TerrainMap map) {
    int yDiff = Math.abs(gY-y);
    int xDiff = Math.abs(gX-x);
    estRemCost = xDiff + yDiff;
    return estRemCost;
}
```

Figure 7: estimatedRemainingCost method for goal closeness

3.1.2 RunRamblersA

A meaning A*, this class uses both the RamblersSearch and the updated RamblersState together to make a map traversal algorithm. The initialisation of the algorithm now has the added parameter of the initial ramblers state with a 0 estimated global cost.

```
public class RunRamblersA {

    public static void main(String[] args) {
    TerrainMap tm = new TerrainMap("tmc.pgm");

    RamblersSearch searcher = new RamblersSearch(tm, new Coords(5,8));
    RamblersState initialState = new RamblersState(new Coords(5,9), 0, 0);

    String result = searcher.runSearch(initialState, "A*");
    System.out.println(result);

    }
```

Figure 8: The RunRamblersA code

3.2 The Results

When running the algorithm, from the car park (coordinates (7,0)) to the goal (coordinates (5,8)), the A* search completes within 54 iterations (height difference) and 78 iterations (closeness) with the following results:

```
Search Succeeds
Efficiency 0.2037037
Solution Path

current Ramblers State: (7,0)
parent null lc 0 gc 0 erc 0 etc 0
current Ramblers State: (6,0)
parent Ramblers State: (7,0) lc 1 gc 1 erc 71 etc 72
current Ramblers State: (5,0)
parent Ramblers State: (6,0) lc 1 gc 2 erc 71 etc 73
current Ramblers State: (5,1)
parent Ramblers State: (5,0) lc 41 gc 43 erc 31 etc 74
current Ramblers State: (5,2)
parent Ramblers State: (5,1) lc 11 gc 54 erc 21 etc 75
current Ramblers State: (5,3)
parent Ramblers State: (5,2) lc 1 gc 55 erc 21 etc 76
current Ramblers State: (5,4)
parent Ramblers State: (5,3) lc 21 gc 76 erc 1 etc 77
current Ramblers State: (5,5)
parent Ramblers State: (5,4) lc 1 gc 77 erc 11 etc 88
current Ramblers State: (5,6)
parent Ramblers State: (5,5) lc 1 gc 78 erc 21 etc 99
current Ramblers State: (5,7)
parent Ramblers State: (5,6) lc 11 gc 89 erc 11 etc 100
current Ramblers State: (5,8)
parent Ramblers State: (5,7) lc 11 gc 100 erc 1 etc 101
```

Figure 9: The results from A* search from C(7,0) to C(5,8) using height difference

3.3 Assessing the efficiency of my A* search algorithm

As shown in figure 9, the A* search completes the route in 10 moves. The figure for efficiency using absolute height difference, as shown, is 0.2037037. When using the closeness heuristic, the efficiency is 0.14102565. This suggests height difference is a better heuristic than closeness.

4 Comparison of A* and BB

Table 1 (shown below) shows the statistics of each of my implemented algorithms with different start points. It shows the start nodes with the corresponding iterations and efficiency of each algorithm.

Search Type	Starting Node	Iterations	Efficiency	Which is the most efficient?
Branch and Bound	(1,0)	151	0.08609272	A* (HD)
A* (height diff)		55	0.23636363	
A* (closeness)		140	0.09285714	
Branch and Bound	(1,5)	136	0.05882353	A* (HD)
A* (height diff)		52	0.15384616	
A* (closeness)		108	0.07407407	
Branch and Bound	(14,10)	175	0.10285714	A* (HD)
A* (height diff)		141	0.12765957	
A* (closeness)		165	0.10909091	
Branch and Bound	(0,10)	144	0.09722224	A* (HD)
A* (height diff)		55	0.23454545	
A* (closeness)		137	0.10218978	
Branch and Bound	(4,8)	2	1.00000000	BB/A*
A* (height diff)		2	1.00000000	
A* (closeness)		2	1.00000000	
Branch and Bound	(5,0)	79	0.11392405	A* (HD)
A* (height diff)		54	0.16666667	
A* (closeness)		78	0.11538461	
Branch and Bound	(3,8)	9	0.33333334	A* (C)
A* (height diff)		7	0.42857143	
A* (closeness)		3	1.00000000	
Branch and Bound	(5,7)	10	0.20000000	A* (HD)
A* (height diff)		2	1.00000000	
A* (closeness)		8	0.25000000	
Branch and Bound	(6,8)	2	1.00000000	BB/A*
A* (height diff)		2	1.00000000	
A* (closeness)		2	1.00000000	
Branch and Bound	(5,9)	56	0.03571429	A* (HD)
A* (height diff)		2	1.00000000	
A* (closeness)		40	0.05000000	

Table 1: Table showing stats for each algorithm from different start nodes to goal node C(5,8)

5 Conclusions

According to my results gathered, it is shown that in every scenario, the A* algorithms take less iterations and therefore has a higher efficiency than BB. In only two cases was the BB algorithm equal to the A*, and that was when the starter coordinates were one place north or west from the destination. This may be because the A* algorithm uses heuristics and therefore is able to gather more information about each successor nodes distance to the final destination, giving indication to whether these nodes are worth exploring, and therefore gains better initial maximum costs. This eliminates many different unneccesary routes, resulting in a much more efficient algorithm. For assessing which heuristic is better for this problem, it seems the height difference is better to use than the closeness - this is because the cost is the elevation and not the total distance travelled. The closeness tests the routes that appear closer first but does not take into account the change in altitude between the points, therefore is unsuitable for the Ramblers Problem. This still gives a better efficiency than the BB however, as it explores closer routes first instead of routes in the opposite direction of the goal.