CS76 Assignment 1: Missionaries and Cannibals

Benjamin Hannam

Fall 2016

NB: I collaborated with Brian Keare

1 Warm Up Task

Implementation

I implemented my maze by using a 2D array of integers within which -1 represented a wall. The states were represented by two integers, the first one held the x location, the second one the y location and these were kept within the MazeNode class as well as the depth for that node.

The set up code was:

```
1  public class MazeworldWarmUp extends SearchProblem{
2
3    //hold the maze layout
4    private int maze[][];
5    private int height;
6    private int width;
7    private int start_x;
8    private int start_y;
9
10   //holds the goal location
11   private int goal_x;
12   private int goal_y;
13
14   //constructor
15   public MazeworldWarmUp(int maze_layout[][], int init_height, int init_width, int gx, int
       gy, int init_x, int init_y){
16
17     //initialize values
18     maze = maze_layout;
19     goal_x = gx;
20     goal_y = gy;
21     start_x = init_x;
22     start_y = init_y;
23     height = init_height;
24     width = init_width;
25     startNode = new MazeNode(init_x, init_y, 0);
```

```
26  }
27
28  //a node class for each state of the game
29  private class MazeNode implements UUSearchNode{
30
31    //hold the current location
32    private int x;
33    private int y;
34    private int depth;
35
36    public MazeNode(int start_x, int start_y, int depth){
37
38      this.x = start_x;
39      this.y = start_y;
40      this.depth = depth;
41    }
```

src/mazeworld/MazeworldWarmUp.java

Here we can clearly see the variables for both the MazeworldWarmUp class and the MazeNode class as well as look at the constructors and see what they take in as parameters. Perhaps the most noticeable thing is that the MazeworldWarmUp constructor takes in a pre made maze which consists of only walls and empty spaces. The tracking of the robot location and the goal location is done outside of the 2D array.

getSuccessors(). goalTest() and isSafeState()

The getSuccessors() method was fairly straight forward and involved generating child states for each cardinal direction then calling the isSafeState() function on them to ensure that they were not wall locations or out of bounds for the maze. The goalTest() method was also very straightforward and was just a check between the $goal_x, goal_y$ coordinates and the x, y state coordinates for that node.The code is as follows:

```
1     //gets all the child nodes of a state
2     public ArrayList<UUSearchNode> getSuccessors(){
3
4       //the array to hold the successors
5       ArrayList<UUSearchNode> children = new ArrayList<UUSearchNode>();
6
7       //move up x
8       MazeNode up_x = new MazeNode(this.x + 1, this.y, depth + 1);
9       //check it
10      if(up_x.isSafeState()){
11        children.add(up_x);
12      }
13
14      //move down x
15      MazeNode down_x = new MazeNode(this.x - 1, this.y, depth + 1);
16      //check it
17      if(down_x.isSafeState()){
```

```
18          children.add(down_x);
19        }
20
21        //move up y
22        MazeNode up_y= new MazeNode(this.x, this.y + 1, depth + 1);
23        //check it
24        if(up_y.isSafeState()){
25          children.add(up_y);
26        }
27
28        //move down y
29        MazeNode down_y = new MazeNode(this.x, y - 1, depth + 1);
30        //check it
31        if(down_y.isSafeState()){
32          children.add(down_y);
33        }
34        return children;
35      }
36
37      //checks if a state is valid
38      public boolean isSafeState(){
39
40        //check if the location is within bounds
41        if( !(this.x >= 0 && this.x < width && this.y >= 0 && this.y < height) ){
42          return false;
43        }
44
45        //check if the location is a wall, walls are marked in the maze as -1 at maze[x][y]
46        if( maze[this.x][this.y] == -1 ){
47          return false;
48        }
49        else{
50          return true;
51        }
52      }
53
54      //check if a state is the goal state
55      public boolean goalTest(){
56        // check if the coordinates match
57        return( this.x == goal_x && this.y == goal_y);
58      }
```

src/mazeworld/MazeworldWarmUp.java

Example mazes using BFS

I ran 4 different example mazes during the testing phase which are shown below along with their paths to the goal node, in this case the bottom left corner (0,9), in the driver (WarmUpDriver.java) the path is animated using ASCII graphics but there only the starting positions are shown. The mazes can be run using the configurations detailed in the README:

0. An empty maze:

```
bfs path length:  18 [[0,9], [0,8], [0,7], [0,6], [0,5], [0,4], [0,3], [0,2], [0,1], [0,0], [1,0], [2,0], [3,0], [4,0], [5,0], [6,0], [7,0], [8,0]]

.........R
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
---------------
```

1. A maze with a path across and down the side

```
bfs path length:  18 [[0,9], [0,8], [0,7], [0,6], [0,5], [0,4], [0,3], [0,2], [0,1], [0,0], [1,0], [2,0], [3,0], [4,0], [5,0], [6,0], [7,0], [8,0]]

.........R
..........
..########
..########
..########
..########
..########
..########
..########
..########
```

2. A maze with a column with an opening in the middle

```
bfs path length:  18 [[0,9], [0,8], [0,7], [0,6], [0,5], [1,5], [2,5], [3,5], [4,5], [5,5], [6,5], [6,4], [6,3], [6,2], [6,1], [6,0], [7,0], [8,0]]

.....#...R
.....#....
.....#....
.....#....
.....#....
..........
.....#....
.....#....
.....#....
.....#....
```

3. A maze that forces the robot to snake around

```
bfs path length:  36 [[0,9], [1,9], [2,9], [3,9], [4,9], [5,9], [6,9], [7,9], [8,9], [9,9], [9,8], [9,7], [9,6], [9,5], [9,4], [9,3], [9,2], [8,2], [7,2],

.........R
.#########
..........
..........
..........
..........
..........
..........
#########.
..........
```

2 A-Star Search

The A* search method is very similar to that of the BFS with the main exception being that we use a Priority Queue instead of a standard Queue which attaches a value to each object inserted into it based on the heuristic provided. This means that we can effectively "weight" our edges in our graph as well.

Implementation

As said above the set up of the A* search is similar to that of BFS but here I used the built in java Priority Queue structure and defined a custom comparator to determine the priority for the nodes we insert into it:

```
1
2     //set up the priority queue
3     PriorityQueue<UUSearchNode> frontier = new PriorityQueue<UUSearchNode>(10000, new
      Comparator<UUSearchNode>(){
4       public int compare(UUSearchNode node1, UUSearchNode node2){
5         int pri_one = node1.getPriority();
6         int pri_two = node2.getPriority();
7
8         if(pri_one < pri_two){
9           return -1;
10        }
11        else if(pri_one > pri_two){
12          return 1;
13        }
14        else{
15          return 0;
16        }
17      }
```

src/mazeworld/SearchProblem.java

Here we can see that the compare function calls another function called getPriority() which uses the priority for the node depending on the heuristic provided for the type of node that is being implemented. For example the heuristic for the WarmUp could be the manhattan distance (the sum of absolute difference in the x and y positions between the goal and the robot) plus the current depth.

Another difference in the implementation is that a node is only added to the explored set once it has been taken off of the queue, not when it is added. This means that we can have multiple instances of the same node but with different priority in the queue. On top of this we have to check the node that is removed from the front of the queue to see if it has been explored as we only want to consider the version of that node with the lowest priority: We have already found a shorter path to that node, there is no point considering the version of it with a longer path.

This is handled in the following lines:

```
1
2     //go through each new layer
```

```
3       while(!frontier.isEmpty()){
4         UUSearchNode current = frontier.poll();
5         //if we haven't already explored the current node
6         if(!explored.contains(current)){
7           //add the current node to explored
8           explored.add(current);
9           //check if its the goal node
10          if(current.goalTest()){
11            path = backchain(current, visited);
12
13            // update stats
14            updateMemory(visited.size() + frontier.size());
15            incrementNodeCount();
16            return path;
17          }
18
19          //for each child
20          for(UUSearchNode child : current.getSuccessors()){
21            //check if they are in explored
22            if(!explored.contains(child)){
23              //add the child to the HashMap pointing back to the current node
24              visited.put(child, current);
25              //add the child to the frontier
26              frontier.add(child);
27              //increment the node count
28              incrementNodeCount();
29            }
30          }
31        }
```

src/mazeworld/SearchProblem.java

Explored is a HashSet keeping track of all the nodes that we have visited, whilst visited is a HashMap that keeps track of which nodes back-point to which.

3 Multi-robot Coordination

In this problem we have k robots that are each at their own start location within a maze and each has their own goal location. They may not pass through one another and they do know where they are and the layout of the maze. The robots take it in turns to move and can move in each of the four cardinal directions (north, south, east and west) or stay where they are.

State Representation

I represented the state of the system by using a integer array of length $2k + 1$ inside of the MultiNode class which also kept track of the depth we were at. I numbered the robots i $= 0, 2...k - 1$ and stored their

x,y locations in the spots in the array corresponding to $state[2i]$ and $state[2i+1]$. I then filled in the last spot in the array ($state[2k]$) with the number of the robot whose turn to move it was. The layout of the maze was stored within a 2D array similar to the one from the WarmUp in the wider MultiRobot class. The code is as follows:

```
1
2
3  public class MultiRobot extends SearchProblem{
4
5    //hold the maze layout
6    private int maze[][];
7    private int goal_locations[];
8    private int height;
9    private int width;
10   private int num_bots;
11
12
13   //constructor
14   public MultiRobot(int maze_layout[][], int start_locs[], int goal_locs[], int init_height,
       int init_width, int k){
15
16     //initialise the variables
17     height = init_height;
18     width = init_width;
19     maze = maze_layout;
20     num_bots = k;
21     goal_locations = goal_locs;
22     startNode = new MultiNode(start_locs, 0);
23
24   }
25
26   private class MultiNode implements UUSearchNode{
27     //variables to hold the state and depth
28     private int[] state;
29     private int depth;
30
31     public MultiNode(int robot_locations[], int new_depth){
32       //initialise the variables
33       depth = new_depth;
34       state = new int[2*num_bots + 1];
35       //input the locations and whose turn it is
36       for(int i = 0; i < 2*num_bots + 1; i++){
37         state[i] = robot_locations[i];
38       }
39     }
```

src/mazeworld/MultiRobot.java

The upper bound on the states depends on the side length(n) of the maze as well as the number of

robots(k) in the system. Each x location can be in a maximum of n different values and each n location can be in a maximum of h different values. There are k different x, y pairs and there are also k different values for the integer representing whose turn it is.Therefore:

Upper bound $= n \times n \times n \times n ... \times k = n^{2k} \times k$ possible states, not all of which are legal.

Now if have w walls then how many of these states can we rule out due to collisions? Well if there are n^2 total spots in the maze that means that there are $n^2 - w$ different spots that a robot can be and w that it cannot be. Only one robot can occupy one space so there are $k \times w$ states that are now ruled out.

A BFS Search would not be feasible in a large maze with with few walls and several robots. This is because we will be searching through a huge amount of nodes as there will be a lot of safe states and so even though there is most probably a pretty open direct path to the goal and there is a low chance of colliding with a wall or a robot we will still be searching outwards in all directions and wasting lots of memory and computation of searching paths that are not in the right direction.

Heuristic

The heuristic that I used for this search space was to take the current depth and add to it the total manhattan distance for all of the robots in the maze. This heuristic is monotonic because the manhattan distance is always the smallest cost path from one spot in the maze to another. If we move to a neighbor node that is the opposite way from the goal node then not only are we increasing the cost of getting to that node we are also increasing the manhattan distance from the goal and so the estimated cost is also going up. Meanwhile if we move in the direction towards the goal then we may be increasing our cost thus far of reaching that goal by 1 but we we are also reducing the manhattan distance by 1 as well so our total cost stays the same as before. This means that no matter which direction we go the estimated cost of reaching the goal is the same or better than that of our successor nodes. The priority is calculated here:

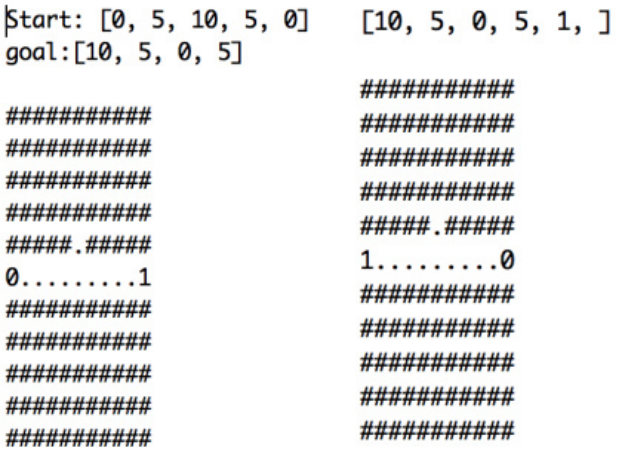
```
1     @Override
2     public int getPriority() {
3       int total = depth;
4
5       for(int i = 0; i < num_bots; i++){
6         // get the current and goal coords
7         int goal_x = goal_locations[2*i];
8         int goal_y = goal_locations[2*i + 1];
9         int bot_x = state[2*i];
10        int bot_y = state[2*i + 1];
11
12        //get the manhattan distance and add it to the total for each robot
13        int man_x = Math.abs(bot_x - goal_x);
14        int man_y = Math.abs(bot_y - goal_y);
15        total += man_x + man_y;
16      }
```

```
17        return total;
```

src/mazeworld/MultiRobot.java

Example Mazes Using A* Search

1. The first maze is a corridor with two robots that start at the opposite ends of the corridor with their goals at the opposite end and there is a gap in the middle to allow one to pass. On the animation you can quite clearly see one of them move into the recess to allow the other to pass through. The left is the start and the right is the end.



2. The second maze is very similar to the first maze but instead this time I added a third robot and made it so that the two that were on one side had to switch before they could get their goals. It was interesting to see the two on one side allow the lone robot through before switching themselves around. In this case the start is on the right and the end is on the left but you can clearly see how robots 2 and 0 switch.

```
[10, 5, 0, 5, 9, 5, 0, ]

##########
##########
##########
##########
#####.#####
1........20
##########
##########
##########
##########
##########
```

```
##########
##########
##########
##########
#####.#####
02.......G1
##########
##########
##########
##########
##########
```

3. This maze is a scaled up version of the one from earlier with the column down the middle and an opening in the center.

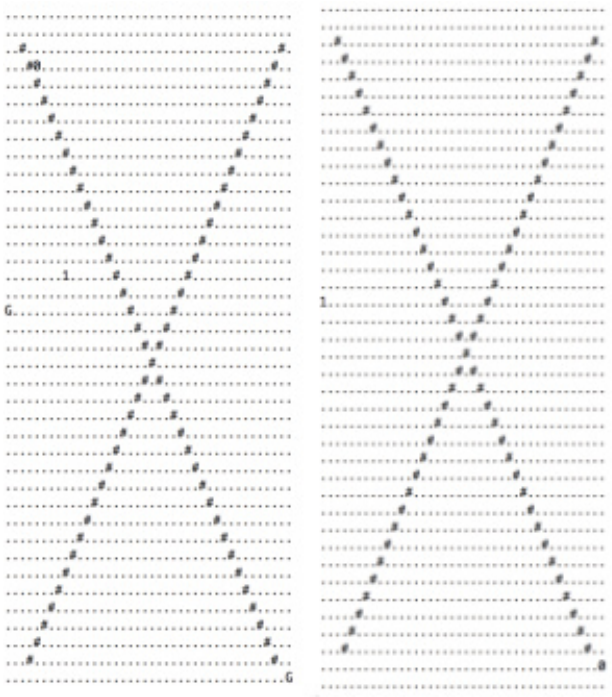
```
Start: [4, 3, 8, 15, 0]
goal:[19, 18, 0, 17]

..........#.........
..........#.........
..........#.........
....0.....#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
.......1.#.........
..........#.........
G.........#.........
..........#........G
..........#.........
```

```
[19, 18, 0, 17, 1, ]

..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
..........#.........
1.........#.........
..........#........0
..........#.........
```

4. In this maze there are only a few openings that a robot can go through to get to from one side to another so despite the fact that 1 is very close to it's goal, 0 is really far away and it was interesting to see it decide to go up and around the top right hand corner before heading down to its goal.



5. This was my favorite maze to watch. In this maze there is a T shaped corridor with a robot at each end and they then have to go and switch positions. It was interesting how one of the robots stayed put whilst the other two moved out of the way.

```
Start: [0, 5, 9, 5, 5, 0, 0] [5, 0, 0, 5, 9, 5, 2, ]
goal:[5, 0, 0, 5, 9, 5]

                                #####0####
#####2####                      #####.####
#####.####                      #####.####
#####.####                      #####.####
#####.####                      #####.####
#####.####                      #####.####
0........1                      1........2
##########                      ##########
##########                      ##########
##########                      ##########
##########                      ##########
```

8-Puzzle

I do not think that this heuristic would be particularly good for dealing with the 8-puzzle as it is not particularly good at dealing with scenarios where robots have to move around each other which in the 8-puzzle is highly important given that each move involves another robot.

I would set up my program to be 8-robots in a 3x3 grid first. Then I would set up the program to start looking for impossible configurations instead of possible once and then all the states that can be made from

that starting one go into one set and all those that cannot go into another.

4 Blind robot

In this problem we have a robot that has absolutely no sensors, it cannot even tell if it has hit a wall in the maze. However we do know the layout of the maze but not where the robot is, how do we go about creating a plan that will tell us where the robot is?

Implementation

In the set up here the maze layout is stored in the overarching BlindRobot class. The state is represented by a Set of all possible locations which are themselves represented by a class called Pair that keeps track of two ints: the x and y locations. The constructor sets up the startNode by inputting every spot that does not have a wall as the starting location.

```
1
2  public class BlindRobot extends SearchProblem{
3
4    //hold the maze layout
5    private int maze[][];
6    private int height;
7    private int width;
8
9    //holds the goal location
10   private int goal_x;
11   private int goal_y;
12
13   //constructor
14   public BlindRobot(int maze_layout[][], int init_height, int init_width, int gx, int gy){
15
16     //set for the start state
17     Set<Pair> start_set = new HashSet<Pair>();
18
19     //initialize values
20     maze = maze_layout;
21     goal_x = gx;
22     goal_y = gy;
23     height = init_height;
24     width = init_width;
25
26     //set up the start state
27     //loop through all the spots in the maze
28     for(int x = 0; x < width; x++){
29       for(int y = 0; y < height; y++){
30         Pair possible_location = new Pair(x, y);
31         //if its not a wall add it to the start set
```

```
32          if(maze[x][y] != -1){
33            start_set.add(possible_location);
34          }
35        }
36      }
37      startNode = new BlindNode(start_set, 0, 0);
38
39    }
40
41    // a class to hold a x,y pair
42    private class Pair{
43      private int x;
44      private int y;
45
46      public Pair(int nx, int ny){
47        x = nx;
48        y = ny;
49      }
```

src/mazeworld/BlindRobot.java

The priority heuristic for this search problem is the total of the depth, the size of the state and also the total manhattan distance from each possible location to the goal. I tried various constants to place more or less emphasis on parts of the heuristic and decided to place more emphasis on reducing the size of the state. Whilst this might not be an optimistic heuristic initially I choose it because the quicker we reduce down to a single belief state, the sooner we will know where we are and from that point we can definitely make an optimal path.

Another heuristic may not consider the size of the state at all and argue that this would be reflected in the total manhattan distance as the fewer states there are then the smaller the total distance.

```
1     public int getPriority() {
2       int total = depth + state_set.size() * 10;
3       for(Pair coords: state_set){
4         //get the manhattan distance and add it to the total for each robot
5         int man_x = Math.abs(coords.x - goal_x);
6         int man_y = Math.abs(coords.y - goal_y);
7         total += man_x + man_y;
8       }
9       return total;
10    }
```

src/mazeworld/BlindRobot.java

getSuccessors()

My getSuccessors() function looped through every Pair in the current state, moved them all north, checked if they were valid and then put them into a new state. It then did the same thing for east, west

and south. This meant that I then had four states, in which every location inside had all been moved in the same direction.

```
    //get the child nodes
    public ArrayList<UUSearchNode> getSuccessors(){

      //set up the array
      ArrayList<UUSearchNode> children = new ArrayList<UUSearchNode>();

      //loop through set and go north in each node
      Set<Pair> north_set = new HashSet<Pair>();
      for(Pair child: state_set){
        Pair north = go_direction(1, child);
        if(isSafePair(north)){
          north_set.add(north);
        }
      }
      BlindNode north_child = new BlindNode(north_set, depth + 1, 1);

      //loop through set and go east in each node
      Set<Pair> east_set = new HashSet<Pair>();
      for(Pair child: state_set){
        Pair east = go_direction(2, child);
        if(isSafePair(east)){
          east_set.add(east);
        }
      }
      BlindNode east_child = new BlindNode(east_set, depth + 1, 2);

      //loop through set and go south in each node
      Set<Pair> south_set = new HashSet<Pair>();
      for(Pair child: state_set){
        Pair south = go_direction(3, child);
        if(isSafePair(south)){
          south_set.add(south);
        }
      }
      BlindNode south_child = new BlindNode(south_set, depth + 1, 3);

      //loop through set and go west in each node
      Set<Pair> west_set = new HashSet<Pair>();
      for(Pair child: state_set){
        Pair west = go_direction(4, child);
        if(isSafePair(west)){
          west_set.add(west);
        }
      }
      BlindNode west_child = new BlindNode(west_set, depth + 1, 4);

      //add the new states
      children.add(north_child);
```

```
49        children.add(east_child);
50        children.add(south_child);
51        children.add(west_child);
52
53        return children;
54      }
```

src/mazeworld/BlindRobot.java

I used a function called godirection() to move a Pair in a certain direction within which I either moved them one spot in the specified direction or did not move them if that direction was a wall or out of bounds. This meant that because I was using a set as well that I got no repeated Pairs in each set (which is what a set does anyway).

```
1     //go a direction
2     public Pair go_direction(int direction, Pair child){
3
4       //go north one spot
5       if(direction == 1){
6         if(child.y > 0 && maze[child.x][child.y - 1] != -1){
7           Pair north_spot = new Pair(child.x, child.y - 1);
8           return north_spot;
9         }
10        else{
11          return child;
12        }
13      }
14      //go east one spot
15      else if(direction == 2){
16        if(child.x < width - 1 && maze[child.x + 1][child.y] != -1){
17          Pair east_spot = new Pair(child.x + 1, child.y);
18          return east_spot;
19        }
20        else{
21          return child;
22        }
23      }
24      //go south one spot
25      else if(direction == 3){
26        if(child.y < height - 1 && maze[child.x][child.y + 1] != -1){
27          Pair south_spot = new Pair(child.x, child.y + 1);
28          return south_spot;
29        }
30        else{
31          return child;
32        }
33      }
34      //go west one spot
35      else{
36        if(child.x > 0 && maze[child.x - 1][child.y] != -1){
```

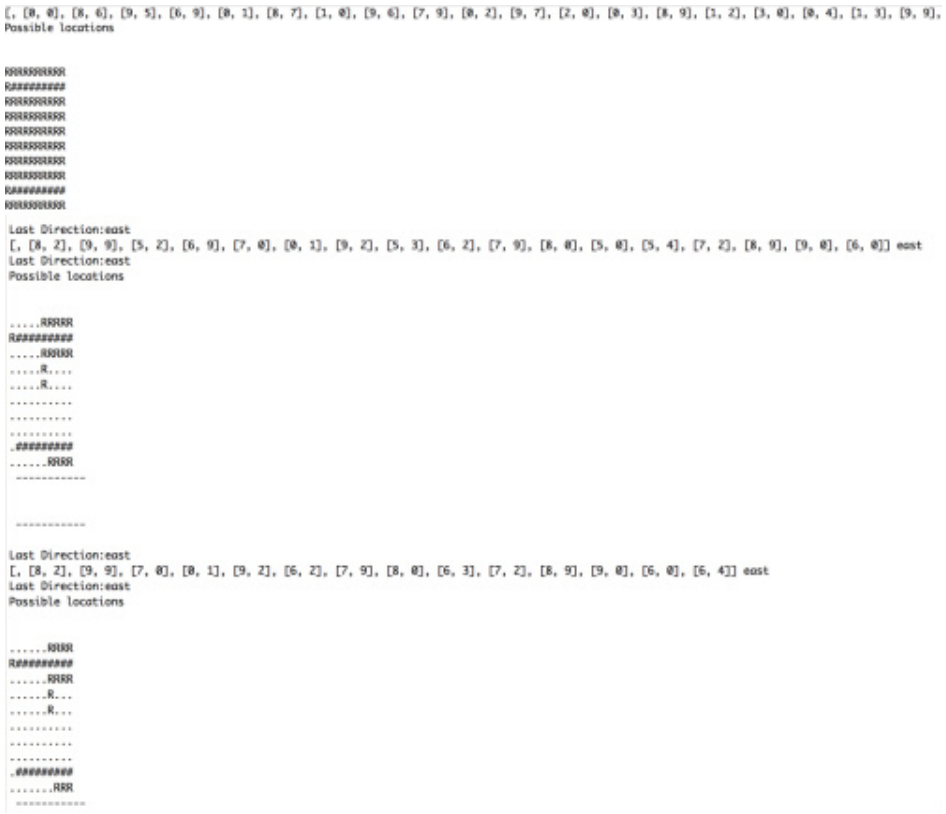
```
37            Pair west_spot = new Pair(child.x - 1, child.y);
38            return west_spot;
39          }
40          else{
41            return child;
42          }
43        }
44      }
```

src/mazeworld/BlindRobot.java

Finally I used isSafeState() to double check that the node was still valid. This may have been unnecessary but I still did it to ensure there were no errors.

Example Mazes

1. In this maze we can see that at the start every spot that does not have a wall is being treated as being possible. However as we proceed along and reduce the number of possible locations we can see from the lower two that the belief state is being compressed. Here we moved everything east and we can see although it is still possible for us to be in more than one location we managed to get rid of 5 more possible locations.



2. Going back to the favorite column down the middle with an opening maze. Here we can see how we remove two whole columns of possible locations when we move east.

```
Last Direction:east
[, [8, 6], [9, 5], [
Last Direction:east
Possible locations


..RRR#..RR
..RRR#..RR
..RRR#..RR
..RRR#..RR
..RRR#..RR
..RRRRRRR
..RRR#..RR
..RRR#..RR
..RRR#..RR
..RRR#..RR
 -----------


 -----------

Last Direction:east
[, [4, 3], [9, 5], [
Last Direction:east
Possible locations


...RR#...R
...RR#...R
...RR#...R
...RR#...R
...RR#...R
...RRRRRR
...RR#...R
...RR#...R
...RR#...R
...RR#...R
 -----------
```

Polynomial-time Robot Planning

With the blind robot problem it is impossible to add more belief states as the robot can not come out of walls so even if we move away from a wall, the space we vacated will not become a possible location. Following on from this if we have 2 or more adjacent possible locations we can always get rid of a possible location by moving in one direction until we hit a wall and then keep moving in that direction until we get rid of all the adjacent possible locations. We can repeat this and reduce down any clumps of possible locations into singleton spots. The next step is proving that it is possible to always bring two of these singletons

together because then we can just keep reducing down until we only have one possible location.

I argue that it is always possible to bring two belief states together because in the maze it is impossible to stop one of them from moving. This would actually work to our advantage as we can the keep one stationary whilst the other moves towards it and if we do a move that moves the stuck one out of its position then we can always just reverse the move later on once the other state has found a way to start heading back towards the first state.