Mazeworld Assignment

Chua Zheng Fu Edrei

January 18, 2016

1 Introduction

In this report, I will implement A* search for robots trying to navigate from a starting location to a goal location in Mazeworld. I will demonstrate the feasibility and optimality of the algorithm by comparing it with breadth-first search (BFS) and depth-first search (DFS) for the problem with one robot. I will also expand the problem to include multiple robots and compare the performance of A* search with other search algorithms. Finally, I will implement a model of a blind robot who doesn't know its original position in the maze and whose goal is to figure out his location in the maze. Additional features will also be discussed.

2 A-star search

I implemented A-star search using the algorithm in the textbook (I modified it to discard before the for loop nodes with states that have been visited and have higher priority value than a previous node with the same state). I made use of three data structures: $frontier$ which is Java's implementation of priority queue to keep track of states to visit, $frontierMap$ which is a hash map to store the priority number of the states in frontier, and $reachedFrom$ which is a hash map that maps the current node to the previous node for back chaining. The relevant code is shown in Listing 1.

Listing 1: Java code for A-star search

```
1  while(!frontier.isEmpty()){
2    SearchNode currentNode = frontier.poll();
3    if(currentNode.goalTest())
4      return backchain(currentNode,reachedFrom);
5    else if(frontierMap.containsKey(currentNode) &&   currentNode.priority()>frontierMap.get(
         currentNode))
6      continue; // discard the node
7
8    for(SearchNode child: currentNode.getSuccessors()){
9      if(!frontierMap.containsKey(child) || child.priority() < frontierMap.get(child)){
10       reachedFrom.put(child, currentNode);
11       frontier.add(child);
12       frontierMap.put(child, child.priority());
13     }
14   }
15 }
16 return null;
```

I used the suggestion on the canvas assignment instructions of marking items if they have already been visited using a hash map and discarding the node if it has a higher cost than already appears in visited for that state. I also tested the algorithm with 5 different maze layout - simple.maz (7x7), empty.maz (7x6), rearrange.maz (5x5), tricky.maz (7x5) and large.maz (40x40). The layout for the different type of mazes will be shown in Section 3. As shown in Table 1, A* is optimal (it produces the shortest path, same as BFS).

In terms of run time (measured by the number of nodes explored), A* is superior to both BFS and DFS. In particular, for larger maze (see the last column), A* is a lot more time and space efficient than BFS (A* explored roughly 10 times less nodes and requires half the space of BFS). Note that for large.maz, DFS is not computationally feasible (more on that in Section 3.3).

	simple.maz			empty.maz			rearrange.maz			tricky.maz			large.maz		
	L	T	S	L	T	S	L	T	S	L	T	S	L	T	S
BFS	21	24	21	7	27	38	11	16	17	21	24	25	16	136	169
DFS	23	23	23	17	17	17	13	13	13	23	23	23	NA	NA	NA
A*	21	23	46	7	6	27	11	10	35	21	23	46	16	15	72

Table 1: A*, DFS and BFS for simple maze robot problem. L denotes path length (number of turns needed), T denotes run time (number of nodes explored) and S denotes space requirement (maximum space usage). NA denotes that search exceeds the given computational time

3 Multi-Robot maze

3.1 Previous Work (non-graduate student)

I refer to the work by Standley, T. 2010 (Finding optimal solutions to cooperative pathfinding problems. In The Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI10), 173178). In this paper, Standley presented "the first practical, admissible, and complete algorithm" for solving the multi-robot maze. His implementation uses A* search to find an initial path for each agent independently at the beginning by assigning each agent to a group. The basic idea is that sometimes the optimal paths of two agents might conflict and when that happen, an algorithm will perform tie breaking and try to find another optimal path. If alternate optimal paths cannot be found for either agents, the algorithm will collapse both groups into one and will solve for the merged problem. While his algorithm has similar performance to the standard algorithm for small number of agents, the improvement in run time is significant as the number of agents increases and an exponential regression yield $t = 0.465e^{1.14s}$ where t is the run time in ms to solve the problem using his algorithm and s is the number of agents. His approach is interesting because he decomposes the original problem into subproblems (solving for the path of each agent independently), and only merge the subproblems together if there is a conflict, resulting in a more practical algorithm that is at the same time, complete and admissible.

3.2 Theory and Discussion

For the multi robot problem, we have k robots in a maze of $n \times n$ located at their respective starting position. The aim is for each robot to move to its goal position. It is possible to represent the state of the system by the location of each robot (its x and y position) and the turn (if we assume that only one robot can move at a time in a given sequence). Therefore, we need $2k + 1$ numbers to represent each state.

If we assume no walls inside the maze and ignore collisions, the upperbound for the total number of arrangement of robots can be given as n^2 permutates k since we are choosing k locations from n^2 possible positions and the ordering matters. In addition, there are k possible turns. Thus, $O(\#state) = \frac{k(n^2)!}{(n^2-k)!}$.

To find the number of states that represent collision, we note that if there are w walls, the number of possible robot location without collision will be $(n^2 - w)$ permutates k instead. Thus $O(\#collide_states) = \frac{k(n^2)!}{(n^2-k)!} - \frac{k(n^2-w)!}{(n^2-w-k)!}$. Alternatively, if we assume that $k << w$, we can just introduce a scaling factor of $\frac{w}{n^2}$ to find the ratio of state space with collision to the total state space, thus $O(\#collide_states) = \frac{w}{n^2}\frac{k(n^2)!}{(n^2-k)!}$.

If there are not many walls and n is large (say 100×100) and several robots (say 10), I will expect that a straightforward BFS to be computationally infeasible for most start and goal pairs (except for those in which the start and goal pairs are located near each other). This is because if the start and goal pairs are far from each other, BFS will have to go through almost all the states, and we noted previously that the number of states increases with the factorial of n^2 (which is even faster than exponential increase!). For $n = 100$ and $k = 10$, that amounts to 9.96×10^{40} number of states!

A useful and monotonic heuristic function for this search space will be the summation of the manhattan distance of each robot to its goal state. This heuristic is an underestimate because the shortest distance for each robot to its goal state will be the manhattan distance, and often if there are obstacles along the way, the actual distance will be longer. In addition, due to the triangle inequality, the heuristic will be monotonic too (consider $h(n) \leq c(n, a, n') + h(n')$, note that the manhattan distance will always be less than or equal to the actual distance i.e. $h(n) - h(n') \leq c(n, a, n')$).

Finally, the 8 puzzle in the book is a special case of the multi robot function because the aim is also for each tile to move from its starting position to the goal position. Our previous heuristic will still be monotonic because if anything, it will be a greater underestimate of the cost. Movement is now restricted to the neighbouring (4,3, or 2 depending on if empty slot is at the center, side or corner) tiles of the empty slot. Each tile needs to move at least its manhattan distance to reach the goal.

The state space of the 8-puzzle is made of two disjoint set (one that can be solved to obtain a chronological order of number, and one that could not and each set contains $\frac{9!}{2}$ states) and it is possible to prove using induction. We can also modify our program to prove this by having the start state as the original chronological ordering and by calculating the number of distinct states possible using BFS or DFS until there are no more distinct states left and compare the size of the set with the total number of combination 9!.

3.3 Implementation of model

I implemented the states as a 1D array $int[]state = \{x_1, y_1, ...x_k, y_k, t\}$, where x_i denotes the x-coordinate of the i^{th} robot, y_i denotes the y-coordinate of the i^{th} robot, and t denotes the turn ($t = i$ denotes that it is the i^{th} robot's turn). Listing 2 shows the implementation of `getSuccessors`. Note that $2 * turnIndex$ in line 6 and $2 * turnIndex + 1$ in line 7 are used to extract the cooridinate of the robot with the current turn.

Listing 2: Java code for getSuccessors for multi robot maze problem

```
1  public ArrayList<SearchNode> getSuccessors() {
2    ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
3    for (int[] action: actions) {
4      int[] stateNew = deepCopy(state);
5      int turnIndex = state[state.length-1];
6      stateNew[2*turnIndex] = state[2*turnIndex] + action[0];
7      stateNew[2*turnIndex+1] = state[2*turnIndex+1] + action[1];
8      stateNew[state.length-1] = (turnIndex+1)\%robotNum;
9
10     if(maze.isLegal(stateNew[2*turnIndex], stateNew[2*turnIndex+1]) &&
11                       !robotCollide(stateNew, stateNew[2*turnIndex], stateNew[2*turnIndex+1])){
12       SearchNode succ = new MultiMazeNode(stateNew, getCost() + 1.0);
13       successors.add(succ);
14     }
15   }
16   return successors;
17 }
```

The method `deepCopy` creates a deep copy of the original states and the method `robotCollide` checks if the state will result in two robots having the same location. Next, I will present the multi robot A* search

for different mazes. Figure 1 shows the movement of 3 robots in simple.maz. This configuration is interesting because we note that the orange piece needs to ensure that it does not stand in the way of the red piece.

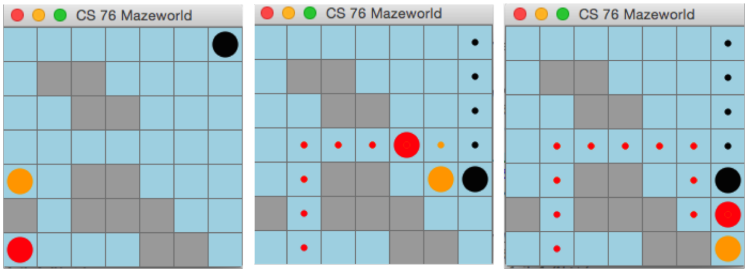

Figure 1: 3 robots in simple.maz. Start state: (0,0), (0,2), (5,5) and goal state is (6,1), (6,0), (6,2)

Figure 2 shows the movement of 3 robots in empty.maz. This configuration is interesting because there is so many degree of freedom, which result in BFS taking a long time (refer to Table 2 for detailed comparison).

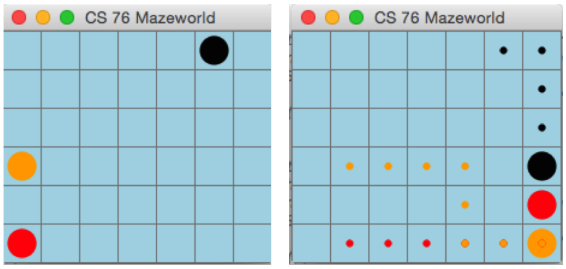

Figure 2: 3 robots in empty.maz. Start state: (0,0), (0,2), (6,6) and goal state is (6,1), (6,0), (6,2)

Figure 3 shows the movement of 3 robots in rearrange.maz. This configuration is interesting because we note that the three pieces need to rearrange among themselves to get to the goal state. This problem cannot be solved by solving for each piece independently since there needs to be some form of coordination.

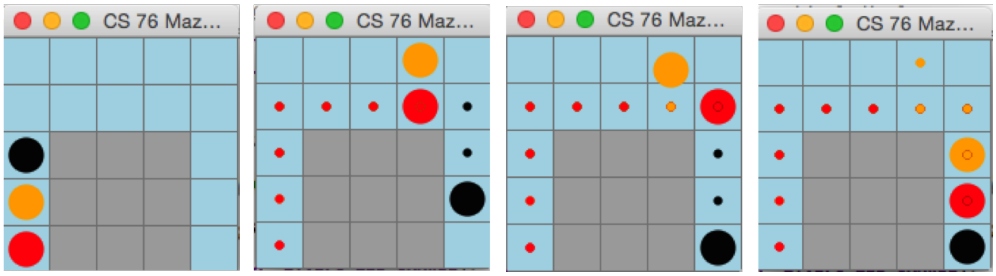

Figure 3: 3 robots in rearrange.maz. Start state: (0,0), (0,1), (0,2) and goal state is (4,1), (4,2), (4,0)

Figure 4 shows the movement of 3 robots in tricky.maz. This configuration is the most tricky of all because the narrow alley requires careful coordination between the 3 robots in order to get to the goal state. The figure shows the path of the 3 robots as they try to rearrange themselves with the least number of turns.

Figure 5 shows the movement of 2 robots in large.maz. This maze has dimensions 40×40 (it is being truncated to fit into the page) which means it has a huge state space, making it computationally expensive for BFS to solve (as noted in section 3.2).

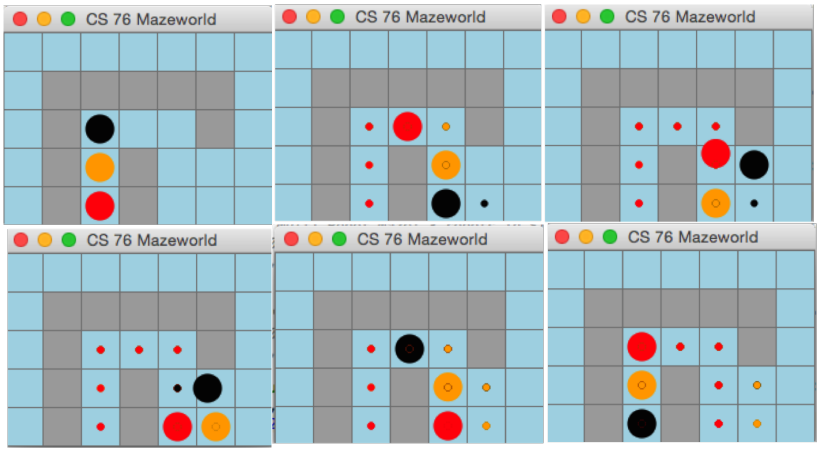

Figure 4: 3 robots in tricky.maz. Start state: (2,0), (2,1), (2,2) and goal state is (2,2), (2,1), (2,0)

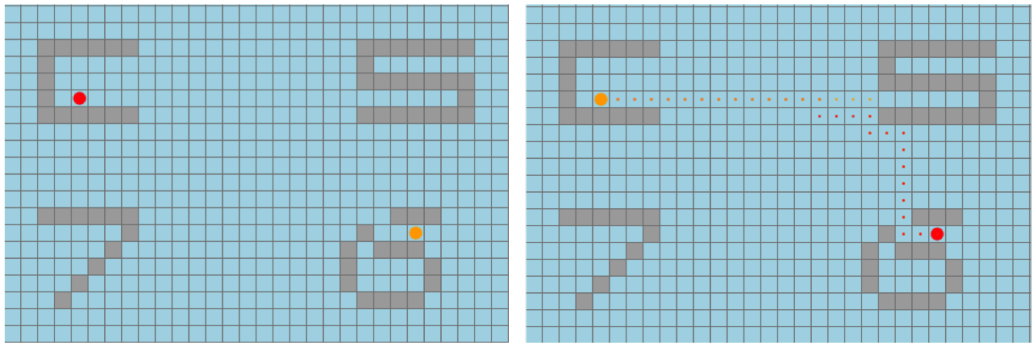

Figure 5: 2 robots in large.maz. Start state: (7,32), (27,24) and goal state is (27,24), (7,32)

Table 2 shows the comparison between BFS, DFS and A* for the different mazes. Again, we note that A* is optimal and complete. It is also faster and requires less memory than BFS (the difference gets more significant for larger mazes: more than 1639 times the space requirement and 17755 times the time requirement). Again, DFS is computationally infeasible for tricky and large mazes.

	simple.maz			empty.maz			rearrange.maz			tricky.maz			large.maz		
	L	T	S	L	T	S	L	T	S	L	T	S	L	T	S
BFS	32	97164	113810	24	135819	161459	33	8978	9599	44	14644	16430	57	1367188	1511281
DFS	5000	76583522	5001	5001	4737183	5001	NA	NA	NA	NA	NA	NA	NA	NA	NA
A*	32	4960	18224	24	2517	13661	33	1205	4468	44	3698	10155	57	77	922

Table 2: A*, DFS and BFS for multi maze robot problem. Description of parameters similar to Table 1.

4 Blind Robot with Pacman physics

4.1 Model and Implementation

The next implementation considers a blind robot in a maze trying to figure out its original position. Note that this problem is very similar to vacuum world described in the book, except that it is deterministic. The basic idea is as follows: if the robot moves North from its orginal position, it knows that its start state cannot possibly be the northern most row. By carefully planning the transition, the number of possible locations

will decrease since some locations collapse into each other. The goal of A* is to discover the path that makes all the possible locations collapse.

I represent the states in a two dimensional boolean array that is the same dimension as the maze (it is memory expensive but the code looks a lot cleaner and I think the tradeoff is worth it. A more memory efficient data structure will be a bitset). The implementation for `getSuccessors` for the blind robot maze problem is given in Listing 3.

Listing 3: Java code for getSuccessors blind robot maze problem

```
1  public ArrayList<SearchNode> getSuccessors() {
2    ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
3    for (int[] action: actions) {
4      int xNew = x + action[0];
5      int yNew = y + action[1];
6
7      if(maze.isLegal(xNew, yNew)) {
8        boolean[][] prevstate = deepCopy(state);
9        int xdiff = Math.abs(xNew - xoriginal);
10       int ydiff = Math.abs(yNew - yoriginal);
11       int decrement = 0;
12       SearchNode succ;
13
14       if(Arrays.equals(action,Maze.EAST)){
15          // previous location can't be at the eastern most state
16         if(xNew - xoriginal > 0 ) // only make changes if we are not backtracking
17             decrement = nullifyCol(mostEast - xdiff + 1, prevstate);
18          succ = new BlindMazeNode(prevstate,xNew, yNew, getCost() + 1.0,numPossible-decrement, "E");
19       }else if(Arrays.equals(action,Maze.WEST)){
20         //...previous location can't be at the western most state (refer to BlindMazeProblem.java)
21       }else if(Arrays.equals(action,Maze.NORTH)){
22          //...previous location can't be at the northern most state (refer to BlindMazeProblem.java)
23       }else{
24         //...previous location can't be at the southern most state (refer to BlindMazeProblem.java)
25       }
26       successors.add(succ);
27     }
28   }
29    return successors;
30 }
```

I then performed A* search for the blind robot model on 5 different mazes. The output is given in Listing 4 with the directions taken. Note that the output for large.maz is not included because the maze is too large and it ran out of memory (due to data structure used).

The heuristic used for A* search is the number of possible locations divided by n i.e. $h = numPossible/n$. This heuristic will certainly be an underestimate of the number of moves required to achieve the goal state. Note that in an empty maze of $n \times n$ starting at (0,0), the total number of moves required to collapse all the possible locations to just one location is $2n$. Each time we arrive at a x-coordinate or y-coordinate for the first time, we eliminate a column or row from the possible locations. Therefore, the maximum number of states that can be reduced at each move is n.

Another possible heuristic that can be used is the total number of unique x-coordinate and y-coordinate left in the set of possible locations (so that at the beginning, the heuristic will yield $2n$). We can keep track of the number of unique x and y values using a hashset. This heuristic will be a better estimate of the

number of moves required and thus, will result in a more optimal A*-search. However, the tradeoff is that we have to keep another data structure and this might result in more memory required.

Listing 4: Output for A* search of blind robot model for tricky.maz

```
Blind Robot Maze: Empty Maze
    Start -> W -> W -> E -> E -> N -> N -> E -> E -> E -> N -> E -> N -> N -> End
    Number of turns: 14; Nodes explored during search: 1210508; Maximum space usage: 11654236
Blind Robot Maze: Simple Maze
    Start -> W -> N -> N -> N -> W -> E -> E -> E -> E -> E -> N -> E -> N -> N -> End
    Number of turns: 15; Nodes explored during search: 15927; Maximum space usage: 113868
Blind Robot Maze: Tricky Maze
    Start -> N -> N -> E -> E -> S -> E -> E -> N -> N -> N -> W -> W -> W -> W -> W -> W -> End
    Number of turns: 17; Nodes explored during search: 21155; Maximum space usage: 130927
Blind Robot Maze: Rearrange Maze
    Start -> N -> N -> N -> N -> E -> E -> E -> E -> End
    Number of turns: 9; Nodes explored during search: 94; Maximum space usage: 644
```

Figure 6 shows the animation for A* search for the blind robot in tricky.maz and the direction plan is included in Listing 4. Note that whenever the robot encounters a new y-coordinate, a row is eliminated and whenever it encounters a new x-coordinate, a column is eliminated. The possible locations at each stage is denoted by the opaque tiles, which becomes translucent when it is removed from the set of possible locations. Also note that the number of possible locations never increases at each stage (it can remain the same, and this happen when it explore a x-coordinate or a y-coordinate that has already been explored).

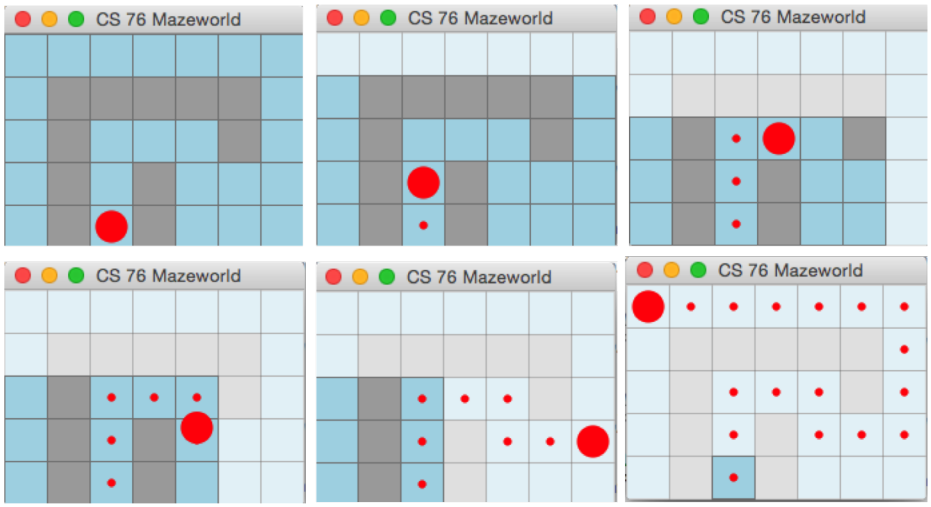

Figure 6: Blind robot in tricky.maz

5 Extra credit

5.1 Polynomial-time blind robot planning

Figure 7 shows the optimal strategy for various mazes. Note that as long as the robot is able to access the set of all unique x-values and y-values (there are $2n - 2$ of those for a $n \times n$ maze since we exclude the starting position) i.e. the maze is finite and the goal is in the same connected component of the maze as the start, a solution exists. The proof is as such: whenever the robot encounters a new x-value, it eliminates a new column and whenever it encounters a new y-value, it eliminates a new row. Since we have

to exclude the starting position of the robot as the robot can only sense difference in x and y coordinate from it starting position and not absolute coordinates (its starting position is set as potential zero for all other relative measurement), if the robot has access to $n-1$ possible x-values and $n-1$ possible y-values, it will necessarily eliminate $2n-2$ rows and columns, leaving only one possible location.

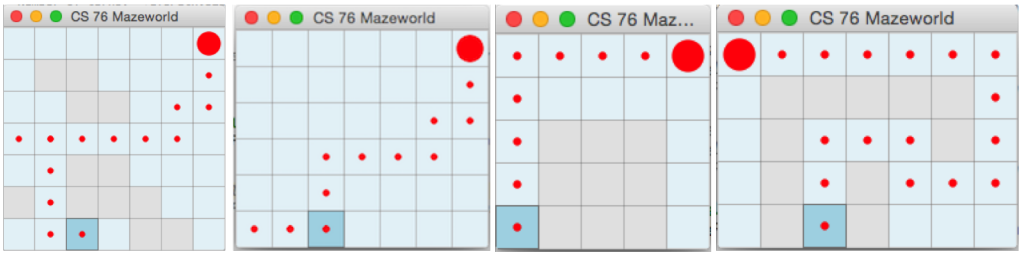

Figure 7: Blind robot in simple.maz, empty.maz, rearrange.maz and tricky.maz (left to right)

A polynomial algorithm is possible: the robot should follow the wall or the outer boundary of the maze on the right side (if the robot has not encounter a wall or the boundary, it will just keep moving straight). This is similar to the right-hand rule for maze search. By doing this, the robot must be able to access all $2n-2$ x-values and y-values if the goal is in the same connected component of the maze as the start. This has quadratic run time since in the worse case, the robot will visit all n^2 tiles.

5.2 Optimizing A* search by tie breaker

I attempted to optimize A* search by following the suggestion of the TA on piazza to implement a tie-breaking rule: when there are two search nodes with the same priority, A* search would choose the node with the higher heuristic, since that state is closer to the goal. The code is given in Listing 5. I tested the tie-breaking rule with tricky.maz for the multi robot and blind robot problem. Without tie breaking, the multi robot explored 4392 nodes and requires 11886 memory usage; with tie-breaking, the values are 3698 and 10155 respectively. Similarly for the blind robot, the results are 21164 vs 21155 nodes explored and 130963 vs 13097 memory usage, showing that the tie breaking rule is more optimal.

Listing 5: Output for A* search of blind robot model for tricky.maz

```
1  public int compareTo(SearchNode o) {
2    if(priority() > o.priority()){
3      return 1;
4    }else if (priority() < o.priority()){
5      return -1;
6    }else{ // tie breaker
7      if(heuristic() > o.heuristic()) return 1;
8      else if(heuristic() < o.heuristic()) return -1;
9      else return 0;
10   }
11 }
```

5.3 Additional graphics

I implemented the method `addFootPrint` in MazeView.java to show the movement of the robot and the method `updateColor` to make the eliminated location tiles for the blind robot translucent.

5.4 Literature review

I did the optional literature review on work by Standley, T. 2010 (refer to section 3.1).