

Mazeworld Solution

Junjie Guan < *gjj@cs.dartmouth.edu* >

January 23, 2014

Contents

1	Introduction	1
2	A-star search	2
2.1	Basic Idea	2
2.2	Single robot problem model	2
2.3	Code implementation	2
2.4	Output demonstration	3
2.5	Discussion on cost and heuristic	4
3	Multirobot problem	7
3.1	Problem definition and states	7
3.2	Discussions	7
3.3	Code implementation	8
3.3.1	getSuccessors	8
3.3.2	noCollision	8
3.3.3	Other methods	9
3.4	Output demonstration	9
4	Blind robot planning	14
4.1	Problem definition and states	14
4.2	Discussions Polynomial-time blind robot planning	14
4.3	Code implementation	15
4.3.1	getSuccessors	15
4.3.2	Constructors	15
4.3.3	Other methods	16
4.4	Output demonstration	16
5	Previous work	16
6	Appendix	19

1 Introduction

Solving maze is one of the most classic and popular problems in Artificial Intelligence. This report majorly cover three parts, 1) introducing the A* algorithm as a searching method; 2) Multirobot problem, where we need to take collision into consideration; 3) Blind robot problem, where the robot need to find out its current coordinate in the maze; 4) finally, some further discussion.

2 A-star search

2.1 Basic Idea

A* is a kind of informed search, which is different from traditional uninformed search (such as bfs). One huge difference is, instead of searching while trying to maintain as least cost as possible in bfs, A* also consider another value called heuristic. Figure 1 is a demonstration of A* algorithm. On one hand, the solid line represent the path that we already go through, which is an evaluation of the past. On the other hand, the dash line, it represents the estimated/expected cost from current state to the goal, which is an evaluation of the future.

At every time A* pick the new state with lowest priority from a priority queue. Usually, it considers the past and the future simultaneously. Let's say the priority value is f , value of cost is g , and heuristic is h , then:

$$f = g + h$$

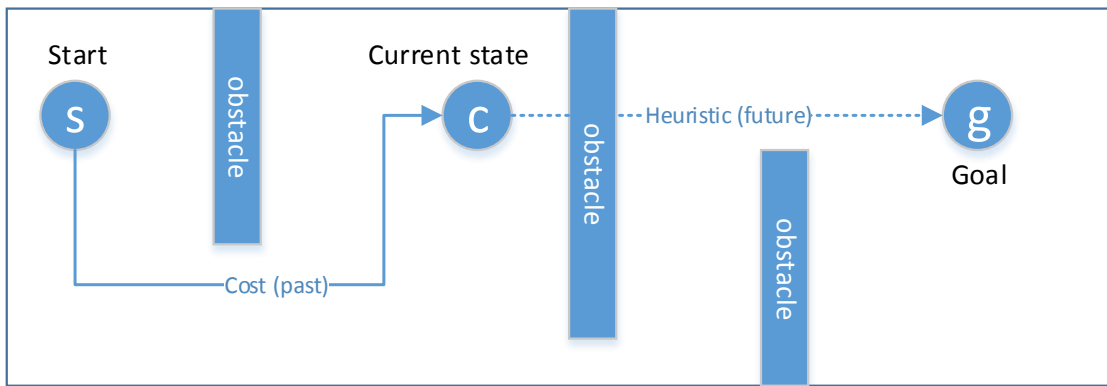


Figure 1: A demonstration of A* algorithm

2.2 Single robot problem model

The basic idea of single robot is to find a path from start position to the goal position. The state here is the coordinate of the robot, (x, y) .

2.3 Code implementation

Line 3-7: There are three major data structure to help me code astar.

- Priority queue: I use a priority queue to store the *frontiers*, sorted by priority. Every search I will pop a node from the head of the queue.
- Hash Map $\times 2$: One hashmap maps from node to node, for creating a backchain at the end. Another hashmap maps from node to priority, for the situation when we re-visit a node, it only worth expanding only if it has higher priority/cost than before.

```
1 public List<SearchNode> astarSearch() {  
    resetStats();  
3    // implementing priority queue for the frontiers  
    PriorityQueue<SearchNode> frontiers = new PriorityQueue<>();
```

```

5 // implementing hashmap for the chain and the visited nodes
HashMap<SearchNode, SearchNode> reachedFrom = new HashMap<>();
7 HashMap<SearchNode, Double> visited = new HashMap<>();

9 // initiate the visited with startnode
reachedFrom.put(startNode, null);
11 // initiate the frontier
frontiers.add(startNode);
13 while (!frontiers.isEmpty()) {
    // keep track of resource
15    updateMemory(frontiers.size() + reachedFrom.size());
    incrementNodeCount();
17    // retrieve from queue
    SearchNode current = frontiers.poll();
19    // discard the node if a shorter one is visited
    if (visited.containsKey(current)
21        && visited.get(current) <= current.priority())
        continue;
23    else
        visited.put(current, current.priority());
25    // mark the goal
    if (current.goalTest())
27        return backchain(current, reachedFrom);
    // keep adding the frontiers and update visited
29    ArrayList<SearchNode> successors = current.getSuccessors();
    for (SearchNode n : successors) {
31        if (!visited.containsKey(n) || visited.get(n) > n.priority()) {
            reachedFrom.put(n, current);
33            frontiers.add(n);
        }
35    }
    }
37 return null;
}

```

Line 20-28: After popping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Another condition is, if the node has been visited before, we don't push it into *frontiers* unless it has shorter cost than before.

Line 29-35: Get the successors of current node, and push those un-visited nodes or node has shorter cost than before, into the *frontiers*.

2.4 Output demonstration

I use Simplex Noise to generate the maze.¹ Figure 2 shows a 40×40 maze, where all three robots try to move from bottom-left to middle-right. I leave the direction of robot at every single node. You can see that bfs and A* perform quite almost equally well, while dfs goes through a lengthy path.

The following output shows that, A* explored significantly less node than bfs, while the path length (cost) almost as less as bfs, while comparing to dfs.

```

BFS:
2 Nodes explored during search: 1064

```

¹I use authorized code from <http://webstaff.itn.liu.se/~stegu/simplexnoise/SimplexNoise.javatohelpmewiththenoise>

```

Maximum space usage during search 1099
4 path length: 60
DFS:
6 Nodes explored during search: 399
Maximum space usage during search 242
8 path length: 242
A*:
10 Nodes explored during search: 79
Maximum space usage during search 224
12 path length: 72

```

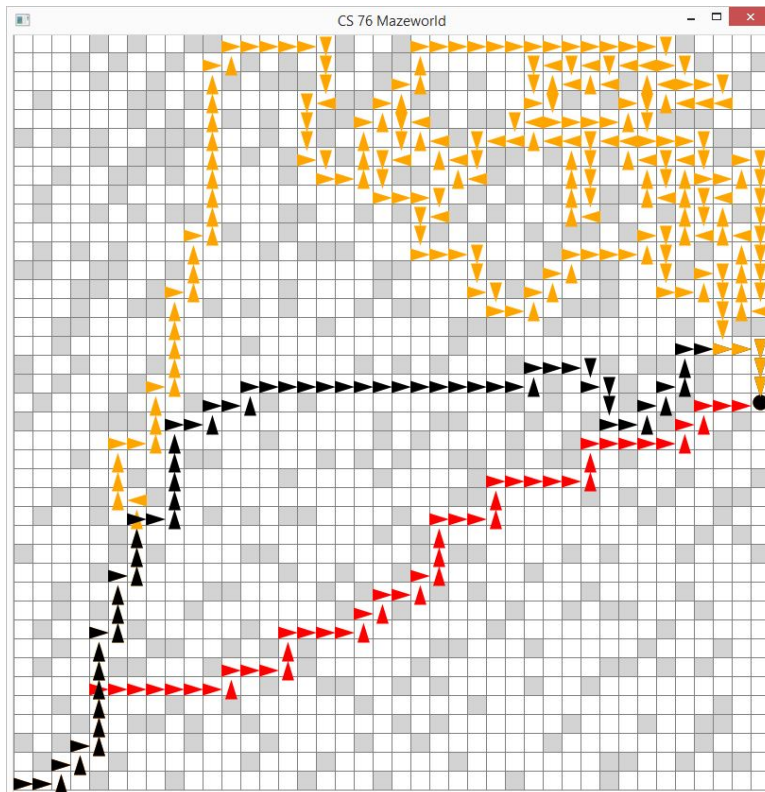


Figure 2: Searching path of dfs(orange), bfs(red) and A*(black)

2.5 Discussion on cost and heuristic

A* leverage both the cost (penalty) and the heuristic (search speed). Currently we use $f = h + g$ for priority, which seems quite a balanced solution. What happens if we go to extrem where the priority only related to g or h ? Or, is 50:50 the best choice for f ?

Before I begin to discuss, I want to introduce a new kind of maze, where the obstacles can be crossed, while there is a certain amount of penalty. While I am first use Simplex Noise generate a maze, i replace the wall with number ranging from 1 to 10 to represent the weights. I use the following functions to differentiate nodes with different weights.

$$Grayscale = 255 - 255 \times weight/20$$

Here I modify the expression of priority as following:

$$f = \alpha \cdot h + (1 - \alpha) \cdot g$$

Then I vary α from 0 to 1, and observe their path length and cost. Figure 4 shows the visual path with different configuration.

- Red ($\alpha = 0.0$) is an extreme case when A* only considers the cost, and becomes uninformed search. (I think it act like Dijkstra Algorithm). Since we are using Manhattan distance and the goal is at top-right, every action of moving south or west is a compromise of cost, and neglecting of path length (search speed).
- Brown ($\alpha = 1.0$) is another extreme case when A* only considers the heuristic, and becomes best-first search. We can observe that it totally neglect the cost/penalty, rushing to the goal in the simplest path.

Figure 3 shows the change of path length and cost while varying α . It depends on the tradeoff on searching speed and cost. It also greatly related to the problem definition, which affects the apperance of state space. For example, if path length and cost are on the scale $a : b$, then we should find α , s.t. $\min(a \cdot \text{cost} + b \cdot \text{length})$.

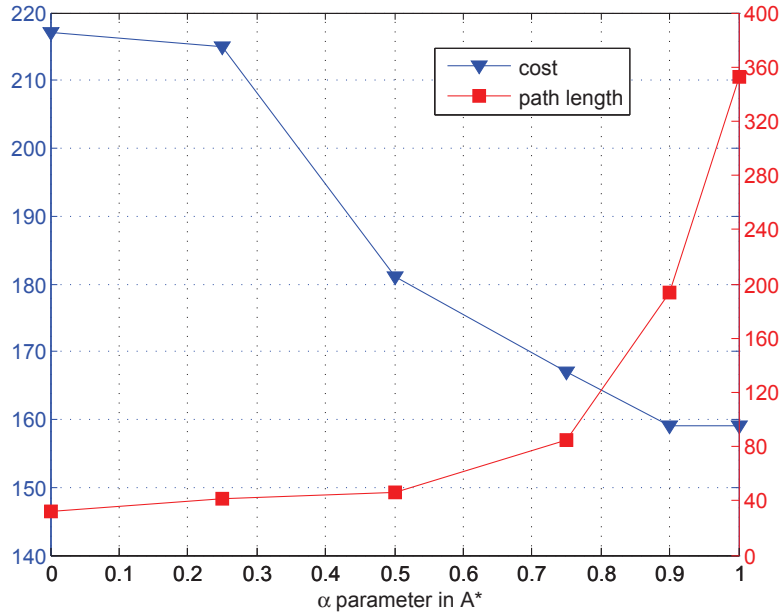


Figure 3: Plot describes the relationship between cost, path length, and α

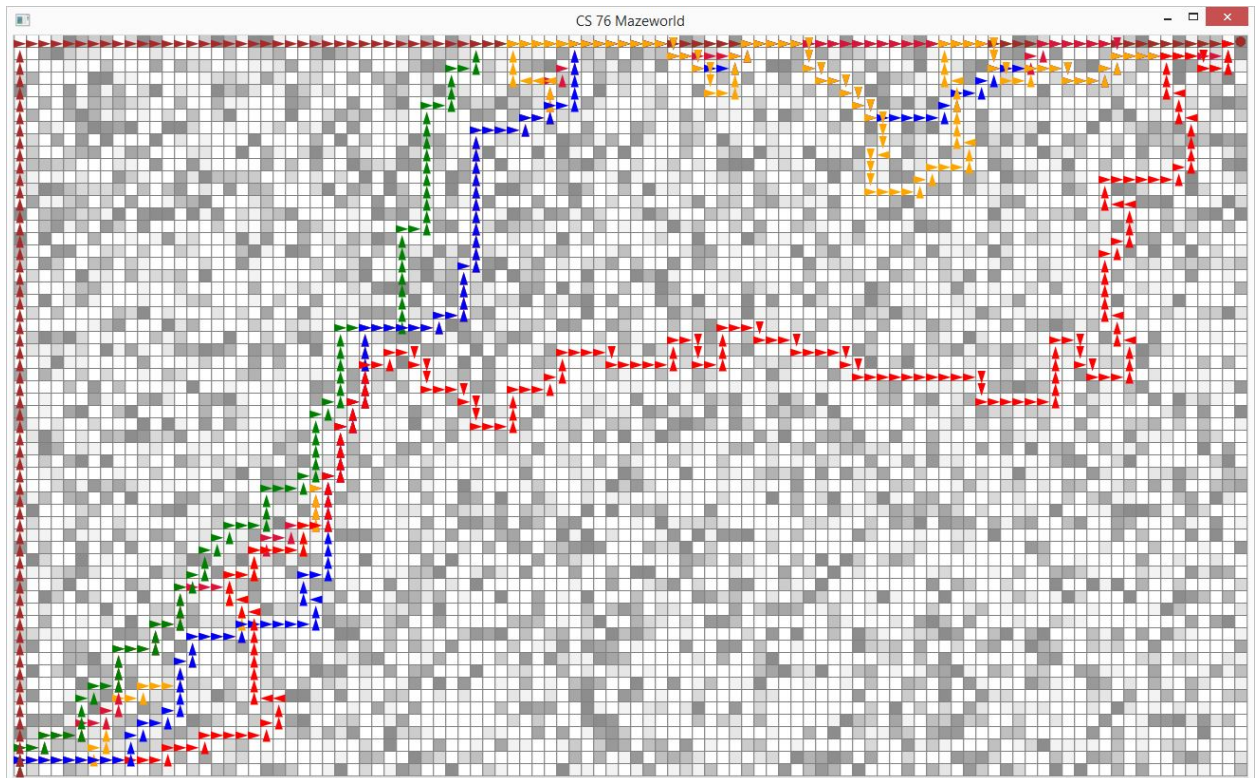


Figure 4: Varying the α in A* as 0.0 (red), 0.25 (orange), 0.5 (blue), 0.75 (crimson), 0.9 (green), 1.0 (brown)

3 Multirobot problem

3.1 Problem definition and states

The different between single robot problem and multi robot problem is, every robot take turns to make actions, and it needs collision detection. In this case, we decide the a whole state for all the k robots, like this:

$$\begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_{k-1} & y_{k-1} \end{pmatrix}$$

However, this is not enough for the states. We still one parameter, which the turn of the robot. This parameter is not necessary, which means you can still solve the problem without it in the state. However, returning all the possible states of k can be very redundant and time consuming for latter searching. If I have k robot, and five actions (4 directions plus not moving), the upper bound of states would be 5^k . It is like giving too much options more next step, while I am not making any decision.

3.2 Discussions

ith $n \times n$ size of maze and k robots the upper bound of this problem, meaning to neglect the legal problem, is n^{2k} . Because each robot has n possible position.

If number of wall square is w , then the number of collisions would be total state minus legal states, $(n^2 - w)^k - C_{n^2-w}^k$.

As for 100×100 , with a few walls and several robot discussion. States number grows exponentially with k , and bfs tends to search all the state starting from start node. I think this problem still depends on whether the goal is close to the starting point.

As for the design of heuristic, I will use the following function, $h = \sum_{i=0}^{k-1} h_i$, where h_i represents the Manhattan distance from robot i to the goal. A function is monotonic as long as²:

$$\begin{aligned} h(N) &\leq c(N, P) + h(P) \\ h(G) &= 0 \end{aligned}$$

where

- h is the consistent heuristic function,
- N is any node in the graph,
- P is any descendant of N ,
- G is any goal node,
- $c(N, P)$ is the cost of reaching node P from N .

It is obvious that for single robot, the single heuristic satisfy this conditions. An extreme case is the empty maze, where $h(N) = c(N, P) + h(P)$. In any other kind of maze, due to the obstacles, the h usually larger than $cost$, because Manhattan distance is the shortest distance.

Under the condition that single heuristic satisfy this conditions. Adding them together should not affect the monotonicity.

The 8-puzzle problem is the case of multi-robot where there is no walls, and there is only one free space to move, and the goal is letting each robots move to its corresponding positions.

²from [wiki:http://en.wikipedia.org/wiki/Consistent_heuristic](http://en.wikipedia.org/wiki/Consistent_heuristic)

Whether 8-puzzle can be divided into two disjoint? This is similar to graph connectivity problem, or disjoint set problem. There is no better way but to traverse through all the possible states in 8 puzzle.

Let's say the total I calculate that the total states amount of this problem is N . My basic idea is, first I pick an arbitrary state, and use bfs to traverse all the connected states, push every one of them into a **hash set**. At this point the number of the set should be less than N . (Otherwise we turn out to prove there is no disjoint set.) Then I will pick another state that is not belong to the first set, and also use bfs to create another **hash set**. Finally, the sum of this two set should equal to N .

3.3 Code implementation

3.3.1 getSuccessors

getSuccessors is used to expand new states from the current state.

Line 4: Iterate through all the 5 possible actions (4 directions plus not moving). **Line 6-7:** Initiate the coordinates for the successor's state, noted that only the robot in turn can take action here. **Line 11-13:** Construct the successor with new coordinates, and new cost based on whether it moves, and also keep looping the turn through R robots.

```

public ArrayList<SearchNode> getSuccessors() {
2   ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
   Integer[] xNew = new Integer[R], yNew = new Integer[R];
4   // take actions
   for (int[] action : actions) {
6       for (int r = 0; r < R; r++) {
           xNew[r] = robots[r][0] + action[0] * (r == turn ? 1 : 0);
8           yNew[r] = robots[r][1] + action[1] * (r == turn ? 1 : 0);
       }
10      if (maze.isLegal(xNew[turn], yNew[turn])
          && noCollision(xNew, yNew)) {
12          SearchNode succ = new MultirobotNode(xNew, yNew, getCost()
            + Math.abs(action[0]) + Math.abs(action[1]),
14              (turn + 1) % R);
            successors.add(succ);
16      }
   }
18   return successors;
}

```

3.3.2 noCollision

noCollision is used to determine whether there is a collision of robots. I set it as private seems it is not used for outer class.

The basic idea is, I hash the position of each robot, and check if the hash code already exists. If so, means there is another robot at this coordinate (x, y) ; if not, push it into the hash set.

If I implement **noCollision** in simple iteration, the time complexity would be $O(k^2)$, while k is the number of robots. By doing so, I reduce time complexity to $O(k)$.

```

1 private boolean noCollision(Integer[] xNew, Integer[] yNew) {
   HashSet<Integer> existed = new HashSet<>();
3   for (int r = 0; r < R; r++) {
       Integer tmpHash = oneHash(xNew[r], yNew[r]);
5       if (!existed.contains(tmpHash)) {

```



```

        existed.add(tmpHash);
7    } else {
        return false;
9    }
    }
11    return true;
}

```

3.3.3 Other methods

Node constructor, it initiates the positions of robots iteratively.

```

public MultirobotNode(Integer[] x, Integer[] y, double c, int t) {
2    robots = new int[R][2];
    for (int i = 0; i < R; i++) {
4        this.robots[i][0] = x[i];
        this.robots[i][1] = y[i];
6    }
    turn = t;
8    cost = c;
}

```

Change the heuristic method to iterative way.

```

1 public double heuristic() {
    // manhattan distance metric for simple maze with one agent:
3    double hValue = 0;
    for (int i = 0; i < R; i++)
5        hValue += Math.abs(xGoal[i] - robots[i][0])
            + Math.abs(yGoal[i] - robots[i][1]);
7    return hValue;
}

```

Line 20-28: After popping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Another condition is, if the node has been visited before, we don't push it into *frontiers* unless it has shorter cost than before.

Line 29-35: Get the successors of current node, and push those un-visited nodes or node has shorter cost than before, into the *frontiers*.

3.4 Output demonstration

Output of shifting 3 robots If we subtract start and end, we know averagely every robot take 4 actions, that is acceptable in a wall-free space. (Figure 5):

```

A*:
2 path length: 14
  Nodes explored during search: 167
4 Maximum space usage during search 577

```

Output of reordering robot in narrow corridor The path length is much longer than I expected. I guess it is because for an amount of time the robot just stay still, waiting for others. It is not easy to move in a narrow space. (Figure 6):

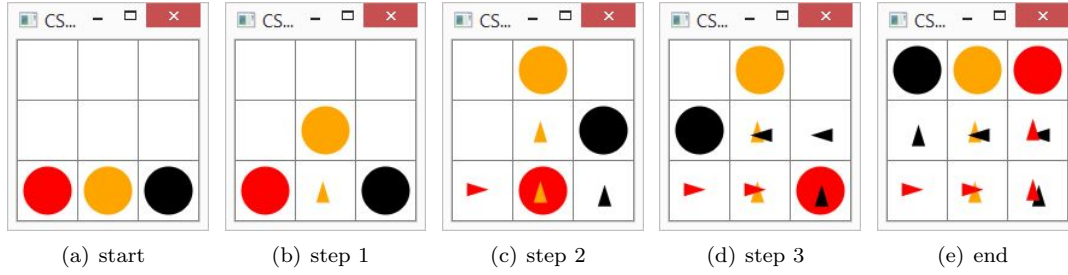


Figure 5: Demo of shifting 3 robots

```
A*:
path length: 52
Nodes explored during search: 953
Maximum space usage during search 861
```

Output of cross road conflict. The average number of movements is 9, which is also reasonable. (Figure 7):

```
A*:
path length: 20
Nodes explored during search: 105
Maximum space usage during search 177
```

Output of moving relatively large number of robots. (Figure 8):

```
A*:
path length: 115
Nodes explored during search: 5070107
Maximum space usage during search 9081343
```

Output of moving relatively large maze. Noted that, although path length is much longer than the previous one, the explored nodes is actually much less. This is because states space size grow exponentially with number of robots. (Figure 9):

```
path length: 233
Nodes explored during search: 109132
Maximum space usage during search 184020
```

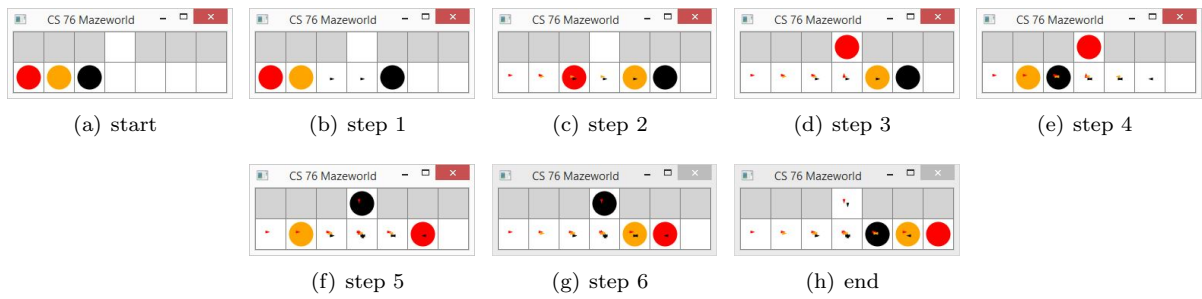


Figure 6: Demo of reordering robot in narrow corridor

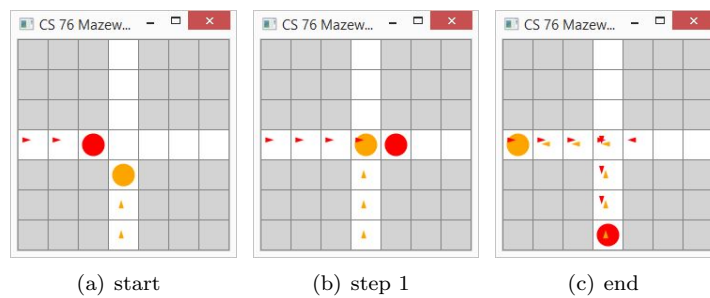
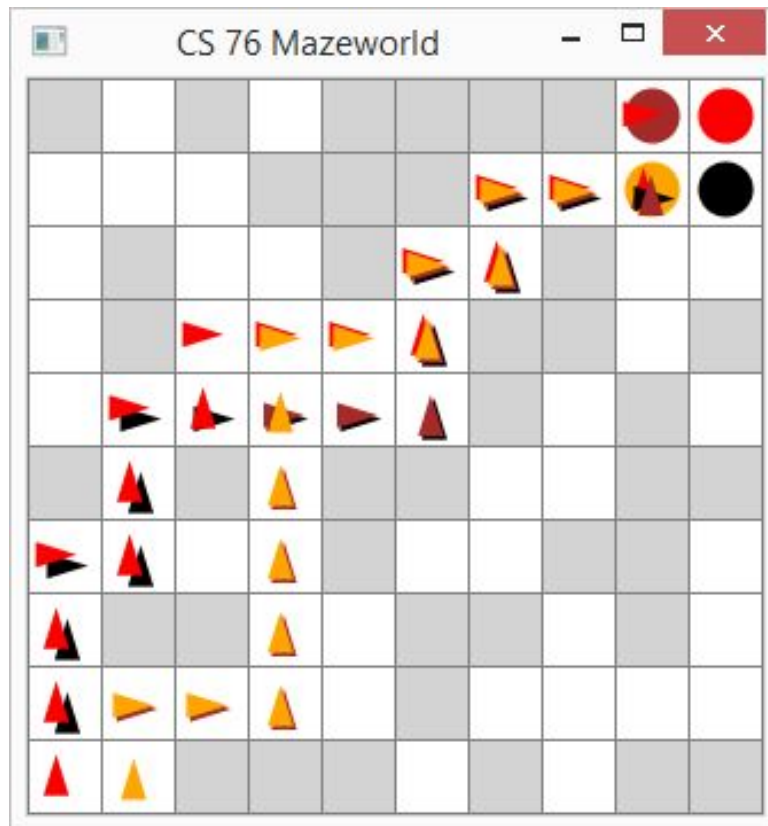
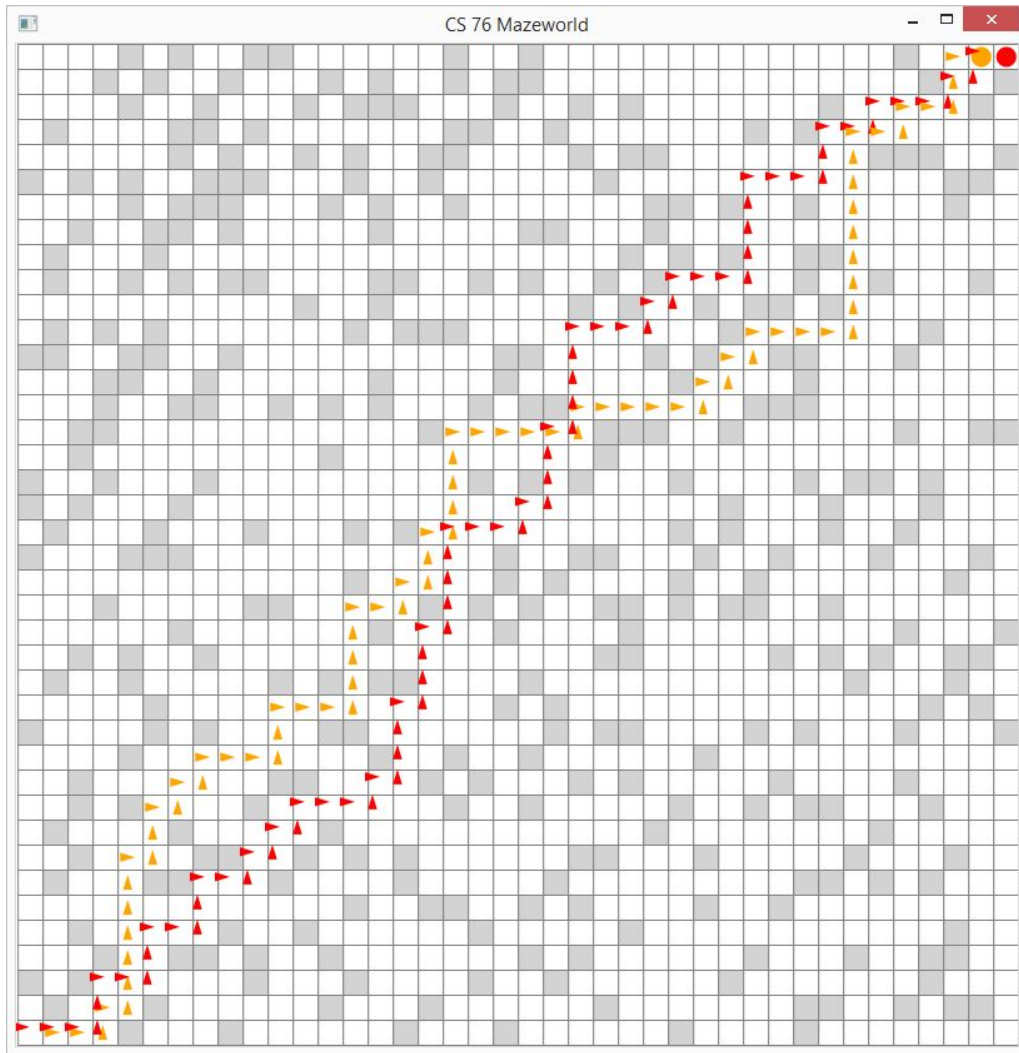


Figure 7: Demo of cross road conflict



(a) start

Figure 8: Demo of moving relatively large number of robots



(a) start

Figure 9: Demo of moving relatively large large maze

4 Blind robot planning

4.1 Problem definition and states

The basic idea is fill the maze with robots, give them instruction simultaneously, while gradually merging them to one point, which is the goal point.

The initial state looks similar to multi robot problem, which is also a 2-D matrix. k equals to number of cell that is not wall.

$$\begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_{k-1} & y_{k-1} \end{pmatrix}$$

The key of the problem is to design a consistent heuristic function. Which satisfy the formula in previous section. My intuition is approaching the goal while shrinking the range of robot positions. I think calculating the average and standard deviation of all the coordinates is a reasonable approach. The average describe the center of the group, the deviation describes the convergence.

So, the state of this problem is as following, noted that $[x_i, y_i]$ and \mathbf{c} are 1×2 matrixs :

$$\mathbf{c}_{x,y} = \sum_{i=0}^{k-1} [x_i, y_i] / k$$

$$d_{x,y} = \|\sqrt{\mathbf{c}_{x^2,y^2} - (\mathbf{c}_{x,y})^2}\|$$

Then, the heuristic function would be:

$$h = \mathbf{max}(\mathbf{manhattan}(\mathbf{c}_{x,y}, g_{x,y}), d_{x,y})$$

How do I prove that the heuristic function is consistent? First, it apparently satisfy $h(G) = 0$, because at that point, all robot will stay at one point, which makes $d_{x,y} = 0$. Since the point is the goal, which makes, $\mathbf{manhattan}(c_{x,y}) = 0$. As for $h(N) \leq c(N, P) + h(P)$, it is similar to the way that I used to prove monotonic previously. First let's assume an extreme case where there is an empty $N \times N$ maze, and the goal is at the top-right. I do a little math on it, it seems that if $N > 1$ or $N < -0.4$, the *cost* is always larger than h . It is obvious always satisfied. (Since I must say I am not a mathematician, I'm not sure if I do it right, feel free to discuss with me if you doubt it).

4.2 Discussions Polynomial-time blind robot planning

If we take it as k-robot problem (without collision issue), where k equals to number of cell. Then upper bound of the sates should be $(N)^k$, where N is the free space in maze. Because each of the k robots can occur in any location.

Assuming the spaces in maze can be desribed by a connected graph. Though the upper bound of space is huge, one big difference from k-robot problem is, the belief state is decreasing, because the robots is merging together.

The problem now is too prove that, if there are 2 robots and moving them simultaneously, can we merge them in polynomial time? If so, We can always pick 2 robot from the k robots, and keep merging the whole states until there is only one left (Actually sometime we merge more than 2 robots at a time).

We can try take actions that trying to shrink their Manhattan distance to 0. How? We can plan a path from A to B, and simply move A to $\langle x_B^{t=0}, y_B^{t=0} \rangle$. Although when A arrive $\langle x_B^{t=0}, y_B^{t=0} \rangle$, B is very likely not there any more, but the Manhattan distance must less or equal than before. The worse case is equal, where B move and A does. But since the map is finite, the Manhattan distance will eventually shorten til 0. It is obvious in a empty maze, while a little tricky in a normal maze.

Because Manhattan distance is shrinking at each approaching, the possible state should not be N^2 (N is total empty square number, 2 means there are nodes here), but actually less than N .

4.3 Code implementation

4.3.1 getSuccessors

Basic idea of **getSuccessors** is to move all robots with one action simultaneously. If there is obstacle for a robot, simply not moves. Note that at **line 14**, the cost is the amount of robots that moves.

```

1 public ArrayList<SearchNode> getSuccessors() {
    ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
3   for (int[] action : actions) {
        HashSet<Coordinate> newCoords = new HashSet<>();
5       // move all the node in the belief set to direction dx dy
        Coordinate dx dy = new Coordinate(action[0], action[1]);
7       for (Coordinate xy : this.allCoord) {
            if (maze.isLegal(xy.x + dx dy.x, xy.y + dx dy.y)) {
9                 newCoords.add(new Coordinate(xy.x + dx dy.x, xy.y
                    + dx dy.y));
11            } else {
                // if not legal, simply not moving
13                newCoords.add(new Coordinate(xy.x, xy.y));
            }
15        }
        successors.add(new BlindRobotNode(newCoords, getCost() + 1.0
17            * newCoords.size()));
    }
19   return successors;
}

```

4.3.2 Constructors

The constructor is fairly simple. Note that, **line 14** is a very important methods, which is used to calculate the average center and deviation of all the coordinates. Some one might ask me why don't I compute the average and deviation while adding each new coordinate in the **getSuccessors**, so that everything can be done in one loop. I got two reasons here. One is the time complexity is the same. The second is I want to keep the code simple, clean and modular. It is not just more easy to read and maintain, but also beneficial to the computing in the low level, since CPU don't have to access different address frequently.

Noted that the number of existing coordinate is decreasing, since some robots may merge together.

```

public BlindRobotNode(HashSet<Coordinate> coords, double c) {
2   // initiate the positions
    allCoord = coords;
4   cost = c;
    newCenDev();
6 }

8 // dev = E(x^2) - (Ex)^2
private void newCenDev() {
10    center = new Coordinate(0, 0);
    sDeviation = new Coordinate(0, 0);
}

```

```

12   for (Coordinate xy : allCoord) {
13       center.x += xy.x;
14       center.y += xy.y;
15       sDeviation.x += Math.pow(xy.x, 2);
16       sDeviation.y += Math.pow(xy.y, 2);
17   }
18   center.x /= allCoord.size();
19   center.y /= allCoord.size();
20   sDeviation.x = Math.pow(sDeviation.x - center.x * center.x, 0.5);
21   sDeviation.y = Math.pow(sDeviation.y - center.y * center.y, 0.5);
22 }

```

4.3.3 Other methods

The **heuristic** method.

```

@Override
2 public double heuristic() {
3     double dx = coordGoal.x - center.x;
4     double dy = coordGoal.y - center.y;
5     return Math.max((Math.abs(dx) + Math.abs(dy)),
6         (sDeviation.x + sDeviation.y));
7 }

```

4.4 Output demonstration

Output of blind robot in 3x3, with the goal at (1,1). (Figure 10):

```

1 A*:
2   Nodes explored during search: 15
3   Maximum space usage during search 100
4   path length: 7

```

Output of blind robot in 7x7, with the goal at (3,5). (Figure 12):

```

1 A*:
2   Nodes explored during search: 88981
3   Maximum space usage during search 622862
4   path length: 20

```

Output of blind robot in 8x8, with the goal at (0, 0). (Figure 11):

```

1 A*:
2   Nodes explored during search: 133773
3   Maximum space usage during search 936406
4   path length: 23

```

5 Previous work

For the k-robot problem, one biggest obstacle here is that the state space grow exponentially with the number of robots, which is N^k . This article put forward a very efficient method to find the optimal solution quick

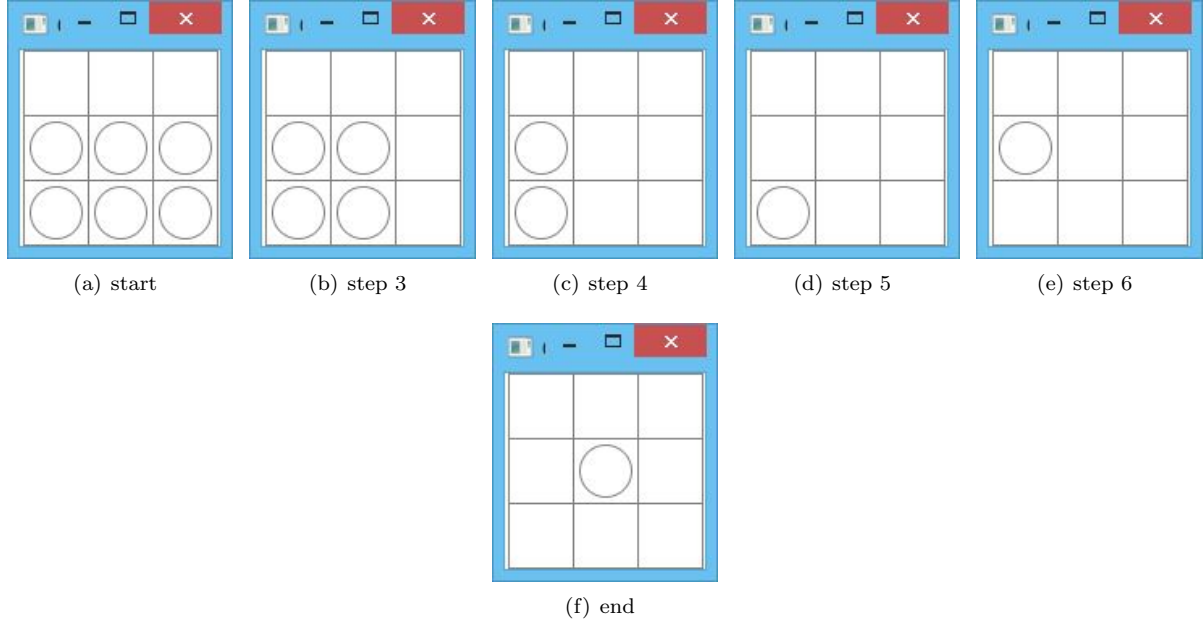


Figure 10: Demo of blind robot in 3x3

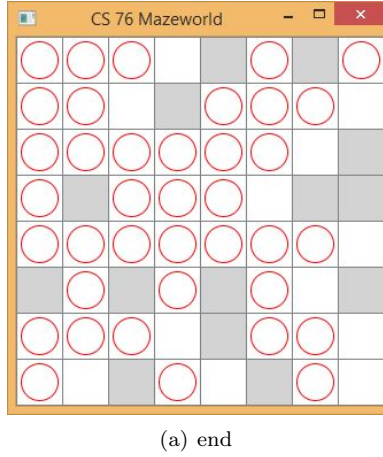


Figure 11: Demo of blind robot in 8x8, this time I'm not showing all the snap shot, just want to tell the readers what kind of maze it is.

against the huge states space³. The philosophy here is, they first A* to plan the shortest path individually. Then they resolve the conflict that occurs in in the k paths. If it cannot be resolved, they merge them in to one group and solve this path as m-robot problem individually. Notice that $m \leq k$.

In order to face the real time demand, such as gaming system, they further design this method, by incorporating the idea of iterative deepening. While increasing the grouping number from one to some large number, they keep record of groups with lower bound, which is quite helpful in the latter iteration.

³Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In The Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI10), 173178.

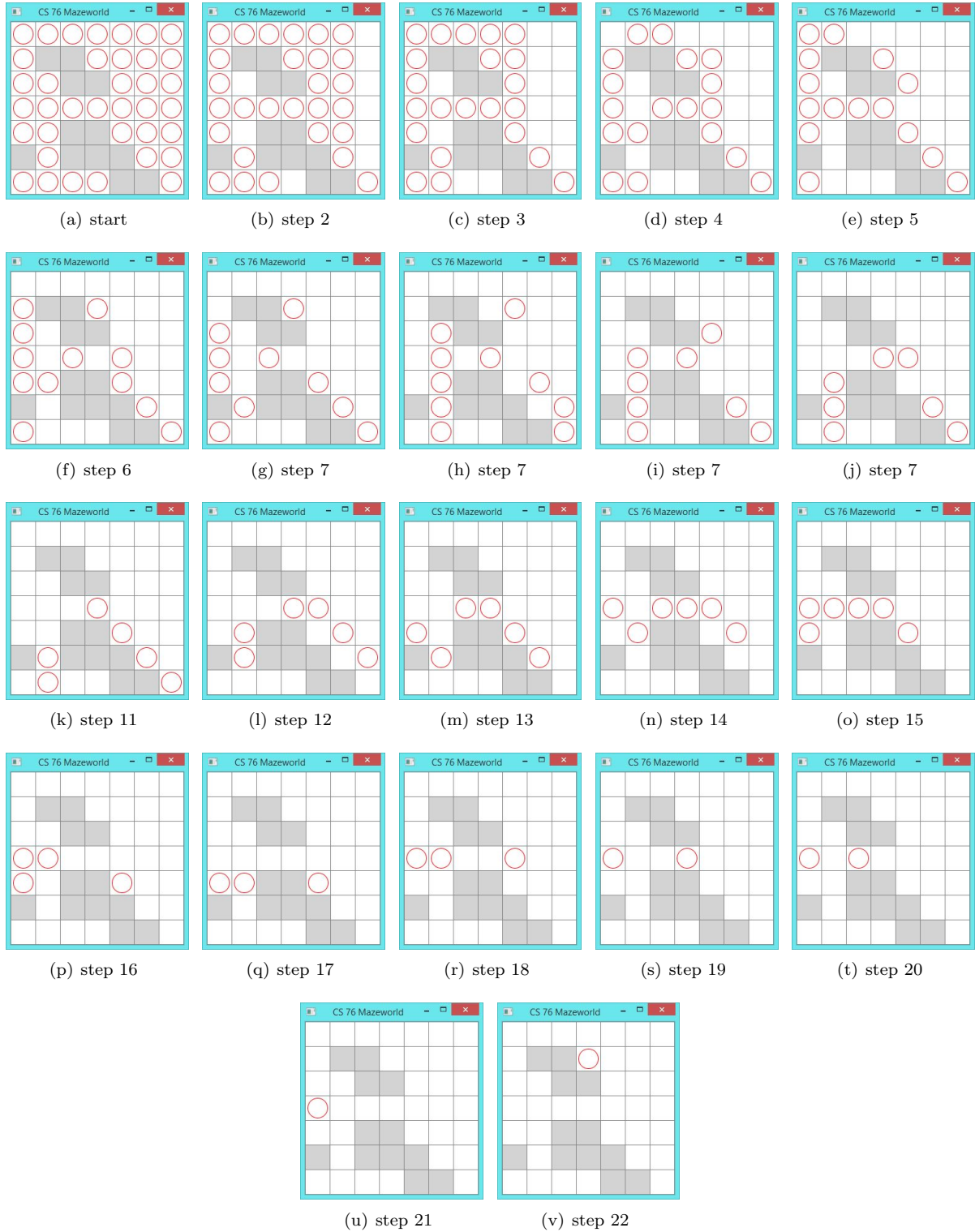


Figure 12: Demo of blind robot in 7x7

6 Appendix

This is the auxiliary class I used to simplify my codes.

```
public class Coordinate {
2   public Double x;
   public Double y;
4
   void add2center(Coordinate other, int newN) {
6       x = (x * (newN - 1) + other.x) / newN;
       y = (y * (newN - 1) + other.y) / newN;
8   }

   Coordinate(int a, int b) {
10      x = (double) a;
12      y = (double) b;
   }

14   Coordinate(double a, double b) {
16      x = a;
       y = b;
18   }

20   Coordinate(Coordinate other) {
       x = other.x;
22      y = other.y;
   }

24   public int getX() {
26      return x.intValue();
   }

28   public int getY() {
30      return y.intValue();
   }

32   private boolean equalWithCast(Object other) {
34      return Math.abs((x - ((Coordinate) other).x)) < 0.01
          && (Math.abs(y - ((Coordinate) other).y) < 0.01);
36   }

38   private boolean isZero() {
       return (x < 0.01) && (y < 0.01);
40   }

42   @Override
   public boolean equals(Object other) {
44      return hashCode() == ((Coordinate) other).hashCode();
   }

46   @Override
48   public int hashCode() {
```

```
    return (int) (x + 1000 * y);
50 }

52 @Override
public String toString() {
54     DecimalFormat df = new DecimalFormat("0.0");
    return "(" + df.format(x) + "," + df.format(y) + ")";
56 }
}
```