Mazeworld Solution

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

January 22, 2014

Contents

1 Introduction

Sovling maze is one of the most classic and popular problems in Artificial Intelligence. This report majorly cover three parts, 1) indroducing 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 anohter value called heuristic. Figure 1 is a demostration of A* algorithm. On one hand, the solid line represent the path that we alredy 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:

$$f = g + h$$

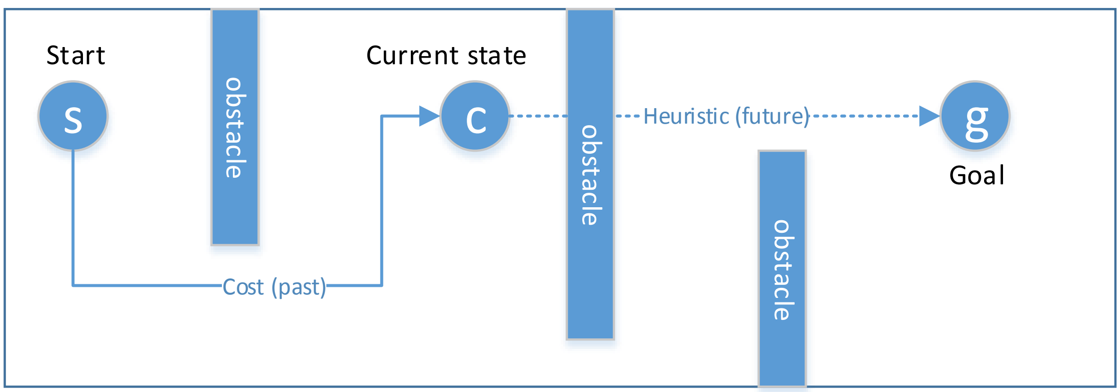


Figure 1: A demostration of A* algorithm

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 prioirty. 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<>();
```

```java
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 poping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Antoher 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 currrent 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:
Nodes explored during search: 1064
```

[1] I use authoried code from `http://webstaff.itn.liu.se/~stegu/simplexnoise/SimplexNoise.javatohelpmewiththenoise`

```
Maximum space usage during search 1099
path length: 60
DFS:
Nodes explored during search: 399
Maximum space usage during search 242
path length: 242
A*:
Nodes explored during search: 79
Maximum space usage during search 224
path length: 72
```

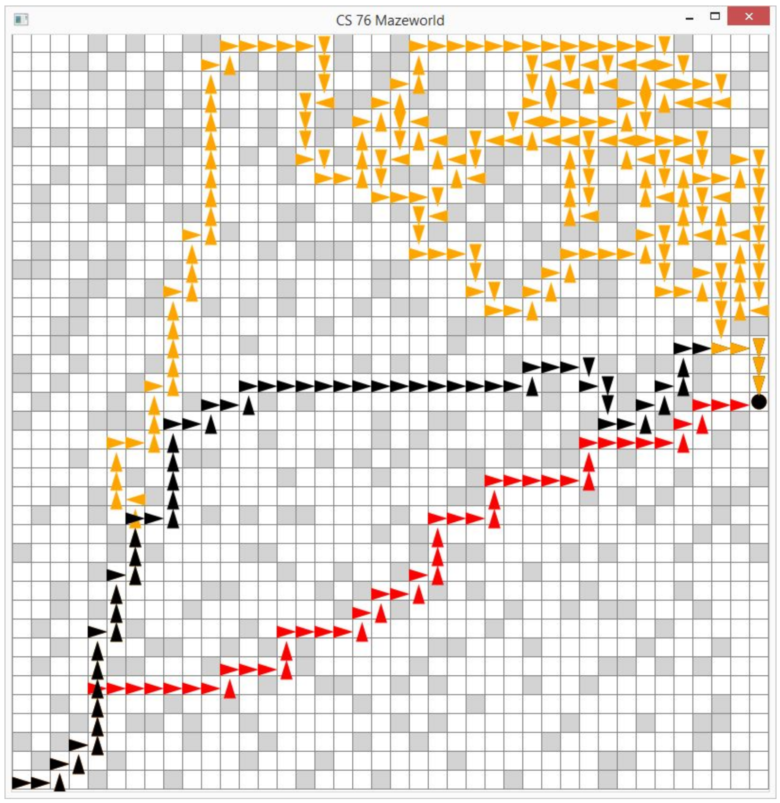


Figure 2: Searhing path of dfs(orange), bfs(red) and A*(black)

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. $min(a \cdot cost + b \cdot length)$.

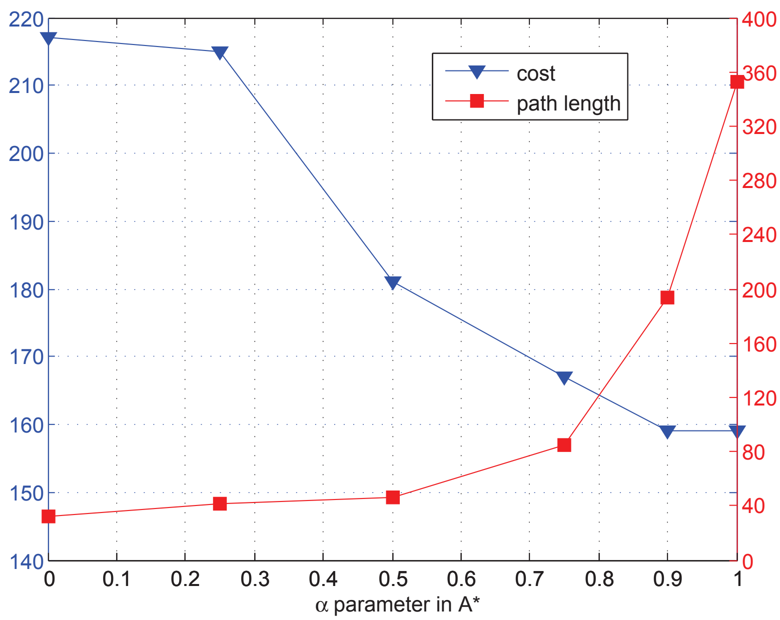


Figure 3: Plot describes the relationship between cost, path length, and α

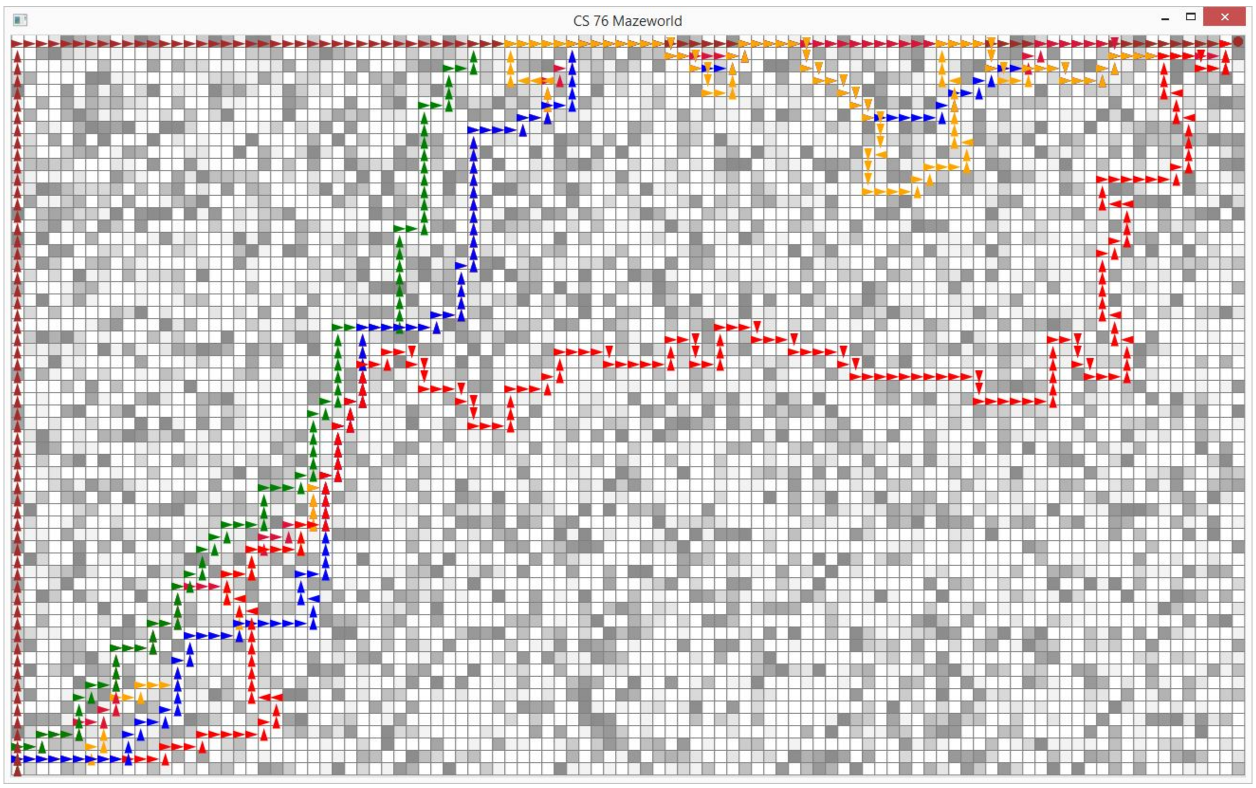


Figure 4: Varying the α in A* as 0.0 (red), 0.25 (orange), 0.5 (blue), 0.75 (crimson), 0.9 (green), 1.0 (brown)

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 thiscase, 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 consumming 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^k. 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, $(n-w)^k - C_{n-w}^k$.

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 as[2]:

$$h(N) \leq c(N,P) + h(P)$$
$$h(G) = 0$$

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 condtions. 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.

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.

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

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 then N. (Otherwise we turn out to prove there is no disjoint set.) Then I will pick antoher state that is not blong 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 for (int[] action : actions) {
   for (int r = 0; r < R; r++) {
 6 xNew[r] = robots[r][0] + action[0] * (r == turn ? 1 : 0);
   yNew[r] = robots[r][1] + action[1] * (r == turn ? 1 : 0);
 8 }
   if (maze.isLegal(xNew[turn], yNew[turn])
10 && noCollision(xNew, yNew)) {
   SearchNode succ = new MultirobotNode(xNew, yNew, getCost()
12 + Math.abs(action[0]) + Math.abs(action[1]),
   (turn + 1) % R);
14 successors.add(succ);
   }
16 }
   return successors;
18 }
```

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 outter class.

The basic idea is, I hash the position of each robot, and 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 $O(k)$.

```
private boolean noCollision(Integer[] xNew, Integer[] yNew) {
HashSet<Integer> existed = new HashSet<>();
for (int r = 0; r < R; r++) {
Integer tmpHash = oneHash(xNew[r], yNew[r]);
if (!existed.contains(tmpHash)) {
existed.add(tmpHash);
} else {
return false;
```

```
}
}
return true;
}
```

3.3.3 Ohter methods

Node constructor, it initiates the positions of robots iteratively.

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

Change the heuristic method to interative way.

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

Line 20-28: After poping the node, we check for two condition. One is if it is the goal, we simply return the solution path and terminate the search. Antoher 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 currrent node, and push those un-visited nodes or node has shorter cost than before, into the *frontiers*.

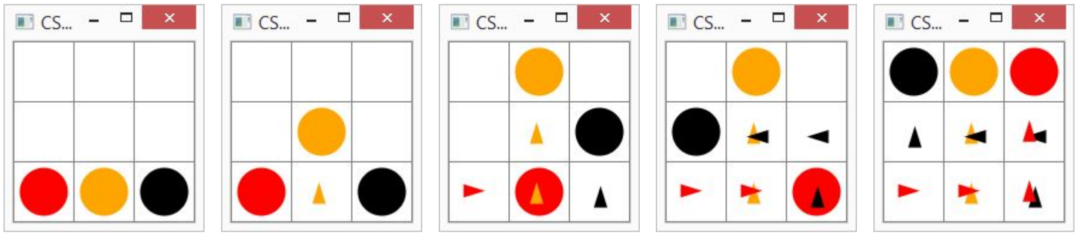
3.4 Output demonstration

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

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

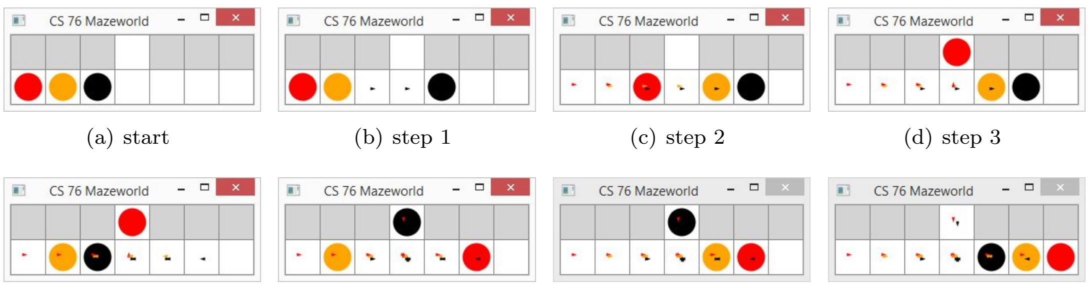
Output of reodering robot in narrow corridor The path length is much longer that I expected. I guess it is because for an amount of time the robot just stay still, waiting for others. It is not easy to move in a narrow space. (Figure 6):

```
A*:
path length: 52
Nodes explored during search: 953
```



(a) start (b) step 1 (c) step 2 (d) step 3 (e) end

Figure 5: Demo of shifting 3 robots



(a) start (b) step 1 (c) step 2 (d) step 3

(e) step 4 (f) step 5 (g) step 6 (h) end

Figure 6: Demo of reodering robot in narrow corridor

```
Maximum space usage during search 861
```

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

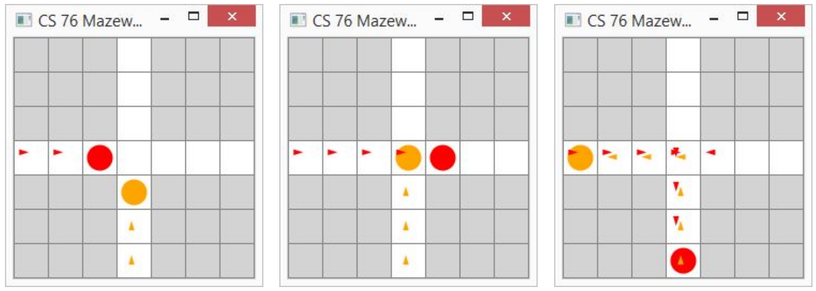
```
A*:
path length: 20
Nodes explored during search: 105
Maximum space usage during search 177
```

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
```

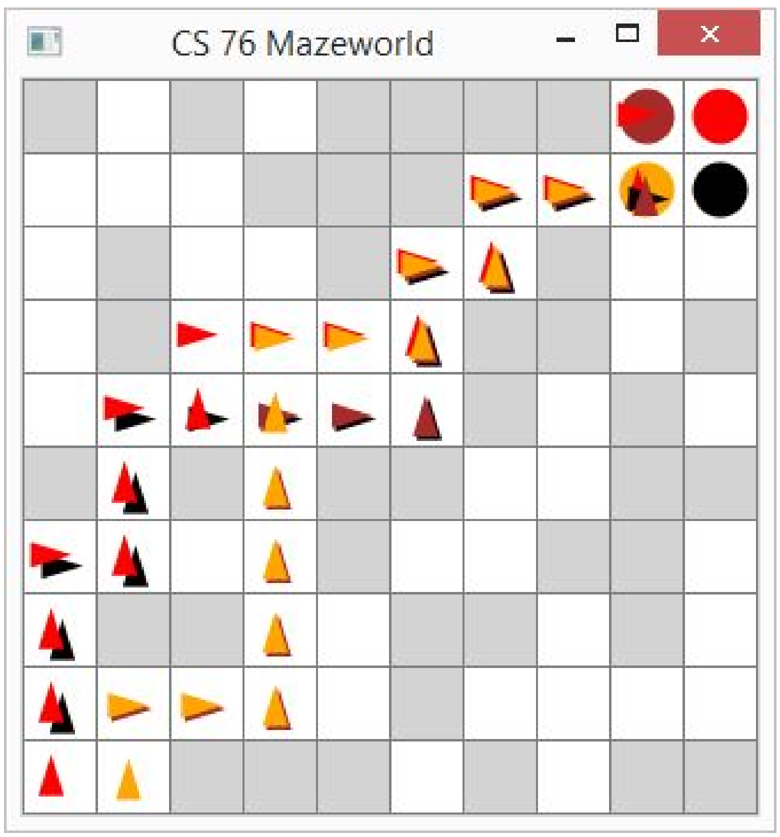
Output of moving relatively large maze. Noted that, although path length is much longer than the previous one, the explored nodes is actually much less. This is because states space size grow expoentially with number of robots. (Figure 9):

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



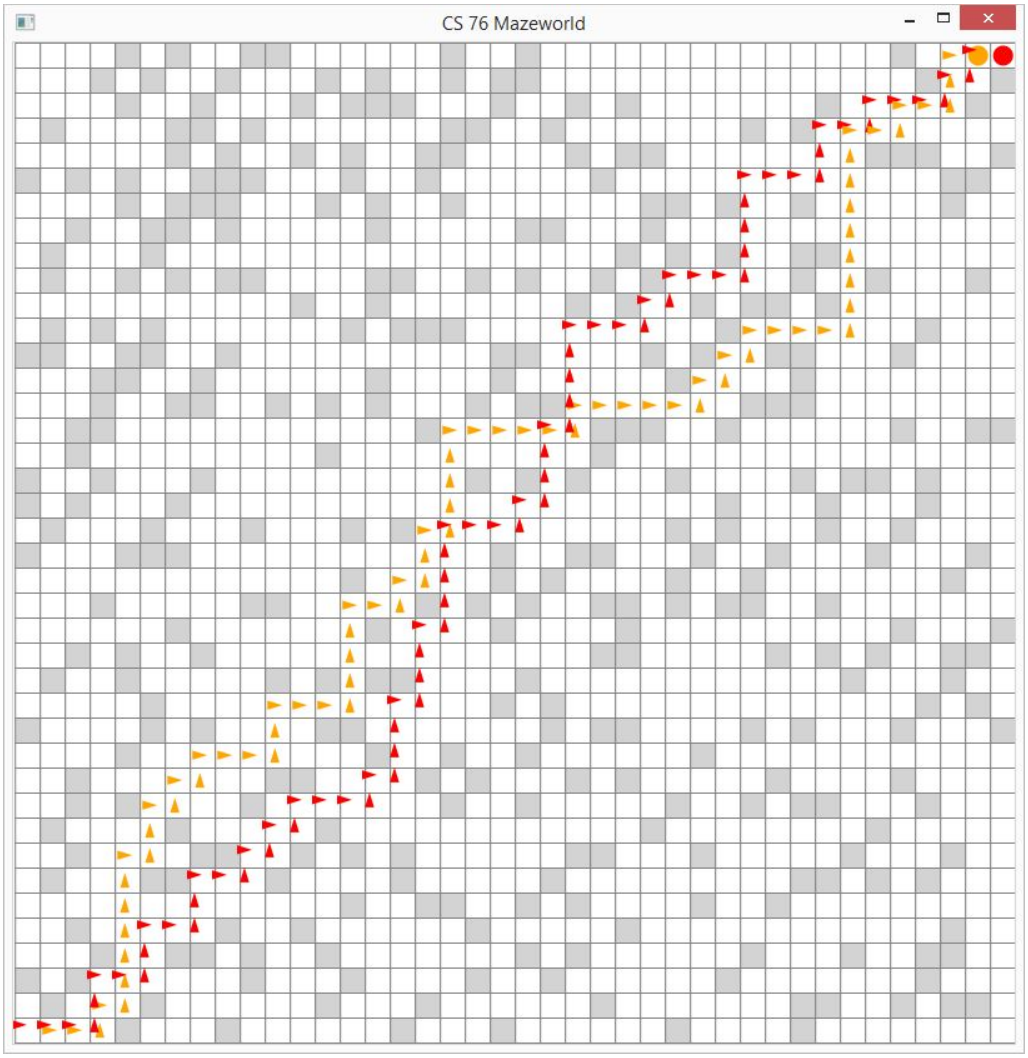
(a) start (b) step 1 (c) end

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 Multirobot problem

4.1 Problem definition and states

The basic idea is put a robot at any abitrary legal position, and give the robot a map. Let the robot try to figure out where it is.

The problem definition is significantly different from the previous two. This time, the state is all the possible current posiiton. At each movement we can eliminate all those positions that does not match the robot's action. Finally, there will be only one position left in the state. The coordinate become nothing but a auxiliary tool this time.

At first sight I think of pattern recognition. For example, let the robot try to go to 4 direction and do the mathcing on the map. Though it can exploit maximum information at each position, it has some drawbacks. Such as no metric to evaluate which direction should go. We probably have no choice but do random search here. What's more, matching four directions looks redundant, which is just a repeat of four checking.

There must a smaller atom in this problem model. So, instead of matching four direction at one position, I try to match the states only when I am moving. And at each move, I can eliminate some candidates.

The initial state looks similar to multi robot problem, which is also a 2-D matrix. Except that the list of the state will shrink during the searching.

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

4.2 Discussions Polynomial-time blind robot planning