Mazeworld Solution

Boying Shi

January 23, 2014

1 Introduction

In Mazeworld problem, we are asked to solve problems of planning single or multiple robots to get to goal nodes and search a practical path from start nodes in a $n * n$ grid. The general formulation of this problem can be described as each robot occupies a single tile of the grid map, and each one has a unique destination node. During a single move from pre-state to post-state, each robot can move north, east, west and south, if there are more than two robots, a action 'wait' can be added.A tile on the grid is legal if it does not contain a wall or more than two robots standing on the same tile at the same time.

Firstly, the assignment as me to implement a well-known searching algorithm: `A* Searching Algorithm`. Not only did I just simply implement this algorithm to apply on a single robot moving problem, but also it was a begin for me to understand its feature of choosing the least cost to the goal at each single time step and the magic of choosing heuristic.

Secondly, I tried to solve multi robots moving problem by finishing answering each question of this part and with enough understanding of this question, I then began design the data structure to represent those robots' states in my program and I found the thinking was the most precious step for me of solving this question. After finishing my design, I implement it and test with different data and got animation view for each test.

Thirdly, I came to the most difficult part of this assignment, to solve blind robot problem. As the robot has no idea of where it starts, it was not the same way of thinking like I solved the two previous questions. It needs a new way of thinking to design the solution. After asking TA about the way of thinking this problem, I then understood to apply `A* Searching Algorithm` could not only solve searching path problem, but also help to solve choosing a best choice and record the path. After understanding this, I finally implemented it.

Finally, as a graduate student, I read a paper named *Finding Optimal Solutions to Cooperative Pathfinding Problems* written by Trevor Standley. And it did help me to understand much more deeply about the multi robots question. Moreover, I will write a review at the end of this report.

2 A* Search

As the main project was given by Professor Devin Balkcom, the structure of the entire program was very well designed and object-oriented. There are 7 files for this project. `SearchProblem.java` contains method of searching algorithms like BFS, DFS and the other two we have done at last assignment. InformedSearch-Problem.java needs me to write `A* Searching Algorithm`. `SimpleMazeProblem.java` contains details of this specific problem to solve and `SimpleMazeDriver.java` is the beginner of the entire program and set the initial values. And `Maze.java` and `MazeView.java` contains codes relating to the maze.

The reason why I list all these everybody known files in my report is that I really appreciate the **object-oriented design** of this problem. It perfectly abstract each possible detailed and certain problem into general problem. A good design is efficient and clear as well as very helpful for further use. And this design helped me a lot to clearly figure out what I should do and where I should modify in Multi Robot problem and Blind Robot problem. What I want to insist here is, I really need to learn how to design software or project in this way and this design model means a lot to me.

Now back to my implementation of `A* Searching Problem`, the implementation of it is alike the implementation of BFS and I referred to the solution code of BFS to write this code:

Listing 1: InformedSearchProblem.java

```java
public List<SearchNode> astarSearch() {
  resetStats();
  //OpenList is used to store possible successors
  PriorityQueue<SearchNode> OpenList = new PriorityQueue<SearchNode>();
  //visited is the close list to to detect duplicate nodes
  HashMap<SearchNode, SearchNode> visited = new HashMap<SearchNode,
     SearchNode>();

  //deal with the initial state outside the loop
  OpenList.add(startNode);
  visited.put(startNode,null);

  while(!OpenList.isEmpty()){
      incrementNodeCount();
      updateMemory(OpenList.size() + visited.size());

    SearchNode currentNode = OpenList.poll();

    //the exit of the search and using back chain to order the path
    if (currentNode.goalTest()) {
        return backchain(currentNode, visited);
      }
      ArrayList<SearchNode> successors = currentNode.getSuccessors();
      for (SearchNode child : successors){
        if(!visited.containsKey(child)){
            OpenList.add(child);
            visited.put(child,currentNode);
            }else{
              //check the cost of current parent and previous parent of the
                 child
              if(!child.equals(startNode) &&
                  visited.get(child).getCost() > child.getCost()){
                  visited.remove(child);
                  visited.put(child,currentNode);
                   }
                }
             }
           }
              return null;
            }
```

2.1 Analysis

`A* Search` is a searching algorithm to the cost of current node to next possible node plus the cost of the next possible node to the goal and as we know the formula: priority() = getCost() + heuristic(). Since getCost() is the cost of going to the next node, it is clearly known since each step can only move one cell and the cost is 1. Therefore, the main point of determining how precise of calculating the cost of moving next possible node to the goal node. Then, it comes to find a good and proper **heuristic()**.

In this project, Professor Devin Balkcom has already helped writing the heuristic() function and it used manhattan distance metric for simple maze with single robot. And it is practically good and proper function of calculating the distance. And I simply used a priority queue which Java provided to help store current possible successor. Also, like what I did in BFS, a HashMap is used to store the visited node and the parent of the current node. If the a successor has already visited, then it means it has a parent. Then, comparing the cost of this parent of current cost, and leave the less one to help choose the less cost.

2.2 Test Result

The result of moving from (0, 0) to (6, 0) in a simple 7*7 maze is shown below:

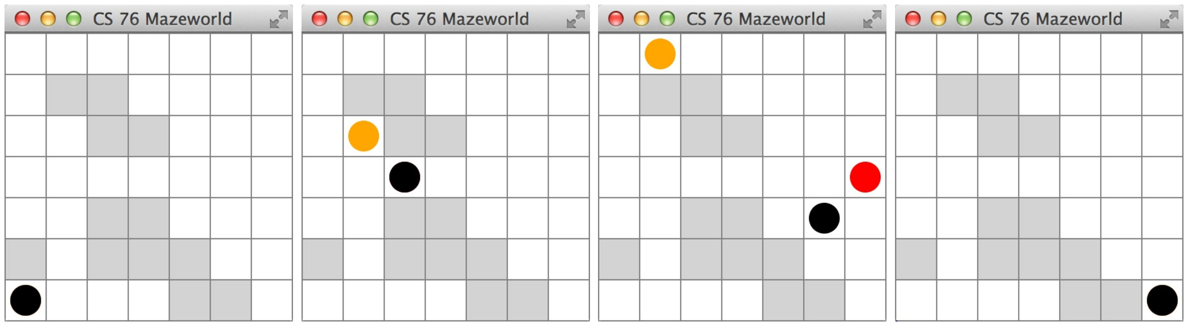

Figure 1: optimal choice of A*

Listing 2: Compare of three searches

```
BFS:
bfs path length:  13 [[0, 0], [1, 0], [1, 1], [1, 2], [1, 3], [2, 3], [3, 3],
    [4, 3], [5, 3], [6, 3], [6, 2], [6, 1], [6, 0]]
  Nodes explored during search:  37
  Maximum space usage during search 41
DFS:
Dfs path length:  21 [[0, 0], [1, 0], [1, 1], [1, 2], [1, 3], [1, 4], [0, 4],
    [0, 5], [0, 6], [1, 6], [2, 6], [3, 6], [4, 6], [5, 6], [6, 6], [6, 5],
    [6, 4], [6, 3], [6, 2], [6, 1], [6, 0]]
  Nodes explored during search:  21
  Maximum space usage during search 21
A*:
a* path length:  13 [[0, 0], [1, 0], [1, 1], [1, 2], [1, 3], [2, 3], [3, 3],
    [4, 3], [5, 3], [5, 2], [5, 1], [6, 1], [6, 0]]
  Nodes explored during search:  19
  Maximum space usage during search 30
```

As is shown in *Figure 1: optimal choice of A**, the black ball represents the robot using A* search to find the goal position, the red one is using BFS and the yellow one is using DFS. It is clear that the BFS and A* can both find the best solution path (BFS searches for the shortest path) while DFS takes a long way to find the destination.

Also, I have the output result in *Listing 2: Compare of three searches*, and it shows all the nodes on the solution path of each searching method. Clearly, BFS and A* compute the shortest path length and it is 13. While DFS takes 21 steps. But for nodes explored and memory usage, BFS takes the most since it need to traverse each neighbor node of the current node. A* uses the least nodes explored since its path shorter than DFS and it uses priority() to find the node and move to the node. While DFS has the least memory usage, since DFS will follow its way down to the deepest but A* need opened list and closed list to record the possible next move node.

3 Multi-robot coordination

Muti-robot coordination problem talks about more than 2 robots move to their goal cells one by one one the maze. The isLegal() is different from single robot problem because two robots cannot occupy the same space at the same time. Also, how to represent those robots in the program is also a need of thinking. In my design, I learn from what TA said, to represent those robots' states together in one array which helps a lot to simplify my coding work. Moreover, since the project is very well object-oriented, there is no need to modify InformedSearchProblem.java any more, and just apply the A* algorithm.

In order to show my work clearly, I will present my works in the order of each file I modified.

3.1 Maze.java

Firstly, in order to randomly output a maze to test my results, I wrote a randomGenerator function. The code is here:

Listing 3: randomGenerator()

```java
public static Maze randomGenerator(int mazeheightSize, int mazewidth){
Maze m = new Maze();
//mazeSize is a static integer
m.height = mazeheightSize;
m.width = mazewidthSize;
m.grid = new char[m.height][m.width];
int[] draw = new int [mazeheightSize*mazewidthSize];

//Randomly produce the wall on the maze, wallSize is a static integer
for(int i = 0; i < Maze.wallSize;i++){
  Random rand = new Random();
  int temp = rand.nextInt(mazeheightSize*mazewidthSize);
  draw[temp] = 1;
      }
      int count = -1;
      for(int i = 0 ; i < mazeSize; i++) {
        for (int j = 0 ; j < mazeSize; j++){
          count ++;
          //draw the entire maze
          if (draw[count] == 1 ){
             m.grid[j][i] = '#';
             }else{
               m.grid[j][i] = '.';
               }
             }
           }
             return m;
           }
```

Therefore, with the randomGenerator, I can produce random maze with random walls setting. Then in the MultiMazeDriver.java(which I modify the name from SimpleMazeDriver.java to it in order to show this certain problem), I do not need to read file from the simple.maz but use my function to produce maze.

As I mentioned before, the isLegal() function is different from single robot problem, and my change of this part is as below:

Listing 4: isLegal()

```
public boolean isLegal(int[] xnew, int[] ynew) {
  //Maze.k is a static variable to store the number of robots
  for(int i=0; i<Maze.k;i++){
    //if there is one robot is not on the maze then is illegal
    if(xnew[i] < 0 || xnew[i] >= width ||
       ynew[i] < 0 || ynew[i] >= height) {
          return false;
          }else{
             for(int j= i+1;j<Maze.k;j++) {
                  //if there is a conflict between robots then return false
                  if(xnew[j]==xnew[i]&&ynew[j]==ynew[i])
                     return false;
                     }
             //If it is a wall, then return false
             if(getChar(xnew[i],ynew[i])=='#')
                 return false;
                 }
               }
                return true;
             }
```

3.2 MultiMazeProblem.java

In this file, it contains how to real solve this particular problem. As it comes from SingleMazeProblem.java, the basic work for me was to change everything related to single into multiple. Like what I did here:

Listing 5: multiple robots in program

```
    //k robots will have k start position and k goal position
    private int[] xGoal = new int[Maze.k];
    private int[] yGoal = new int[Maze.k];
    private int[] xStart = new int[Maze.k];
    private int[] yStart = new int[Maze.k];

    //store all x of start node in to one array and all y of start node in
       one array
     //store all x of goal node in to one array and all y of goal node in one
        array
    public MultiMazeProblem(Maze m, int[] sx, int[] sy, int[] gx, int[] gy) {
      for(int i = 0; i<Maze.k;i++){
         xStart[i] = sx[i];
         yStart[i] = sy[i];
         xGoal[i] = gx[i];
         yGoal[i] = gy[i];
        }
        startNode = new MultiMazeNode(sx, sy, 0);
        maze = m;
    }
```

As I shown, start nodes and goal nodes need to multiple since there are k robots and I use arrays to solve it. And in the constructor, I initial them by transforming arrays into startNode.

Listing 6: multiMazeNode()

```
  public MultiMazeNode(int x[], int y[], double c) {
      //store all robots' state in one array
      //each even number position store x coordinate and odd number record y
         coordinate
      state = new int[2*Maze.k];

      for(int i= 0; i<Maze.k;i++){
        this.state[2*i] = x[i];
        this.state[2*i+1] = y[i];
        }
          cost = c;
          }

    //get the kth robot x coordinate
    public int getX(int k) {
       return state[2*k];
        }

    //get the kth robot y coordinate
    public int getY(int k) {
       return state[2*k+1];
        }
```

In multiMazeNode(), the node will be represented all k robots' positions and I treat this entire node as a multimazenode rather than I did in single maze node that a node is only for one robot's position. Therefore, there are k robots, then there are 2*k element in this state. And every even number position is for x coordinate of each robot and old number is for y coordination.

Also, I paste two other function getX() and getY(). The change of these two are to get the certain robots position, just simply pass the index of the robot.

Listing 7: getSuccessors()

```
  public ArrayList<SearchNode> getSuccessors() {
     ArrayList<SearchNode> successors = new ArrayList<SearchNode>();

     //record new position after each action
     int[] xNew = new int[Maze.k];
     int[] yNew = new int[Maze.k];

     for(int i=0; i<Maze.k;i++){
       xNew[i] = state[2*i];
       yNew[i] = state[2*i+1];
       }
```

```
        //using flag to show it is which robot's to move since only one robot
           can move at one time step
        for (int[] action: actions) {
          xNew[flag] = this.state[2*flag] + action[0];
          yNew[flag] = this.state[2*flag+1] + action[1];

          if(maze.isLegal(xNew, yNew)) {
            SearchNode succ = new MultiMazeNode(xNew, yNew, getCost() + 1.0);
            successors.add(succ);
               }
              }
              flag++;
              //reset the flag

              if(flag > Maze.k-1){
                flag = 0;
                }
                return successors;
               }


@Override
public boolean goalTest() {
  for(int i = 0; i<Maze.k;i++){
    if(state[2*i]!=xGoal[i]||state[2*i+1]!=yGoal[i])
      return false;
      }
       return true;
     }
```

Since I treat the entire big state as a state in this problem, so I need to get successors of each state change. And as is mentioned in the assignment, each time will be only one robot to move. Then, for each action out of the four, it should test only one particular robot and get its successors and then move to the next robot. To clearly represent it, I use a flag to point it out, and after getting one robot's successors then the flag will plus one. When it comes to the final robot, the flag will reset to 0 and begin next round.

Additionally, the goalTest() should also be modified to test k robots' goal node. So I just simply as a for loop to check if all robots reach them personal goal nodes.

3.3 heristic()

In my design of heristic() function, I calculation each robot's manhattan distance to their goal state and then sum them up since I treat the entire robots as one single state. Although it is not the most optimal heuristic(), I think it really works and help my robots to found their goal nodes in admissible time period.

Listing 8: heuristic1()

```
  @Override
   public double heuristic() {
   // manhattan distance
     double[] dx = new double[Maze.k];
     double[] dy = new double[Maze.k];
     double sumx = 0.0;
     double sumy = 0.0;
```

```
    //record each robot's manhattan distance
    for(int i=0; i<Maze.k;i++){
       dx[i]= xGoal[i]-state[2*i];
       dy[i] = yGoal[i]-state[2*i+1];
       }

    //sum them up
    for(int i=0;i<Maze.k;i++){
      sumx = sumx + dx[i];
      sumy = sumy + dy[i];
      }

      return Math.abs(sumx) + Math.abs(sumy);
    }
```

Also this is another way of compute heuristic(), I will record the smallest `dx[i]` and `dx[i]` related to my previous one. And then get the manhattan distance. The code is like this:

Listing 9: heuristic2()

```
  @Override
   public double heuristic() {
   // manhattan distance
    double[] dx = new double[Maze.k];
    double[] dy = new double[Maze.k];
    double sumx = 0.0;
    double sumy = 0.0;

    //record each robot's manhattan distance
    for(int i=0; i<Maze.k;i++){
       dx[i]= xGoal[i]-state[2*i];
       dy[i] = yGoal[i]-state[2*i+1];
       }

    double minx = dx[0];
    double miny = dy[0];
    for(int i=1;i<Maze.k;i++){
      if(minx > dx[i]){
         minx = dx[i];
      }
      if(ming >dy[i]){
         miny = d[i];
      }y
      }
      return Math.abs(minx) + Math.abs(miny);
    }
```

After testing for several times, I got the result of the two heuristic() can have almost the same number for path length(the same path). While, the nodes explored and memory usage, the latter one is much larger than the former one. Therefore, I finally choose the former one for my mainly test heuristic().

These are those main changes and I also have some changes in MutiMazeDriver.java but they are basically related to draw animation graph. So I do not put the detailed code here.

3.4 Test Result

3.5 result of the example

This is the test of the example in the assignment, the robot B smartly wait A to get its goal and then get the goal. The path is shown below:

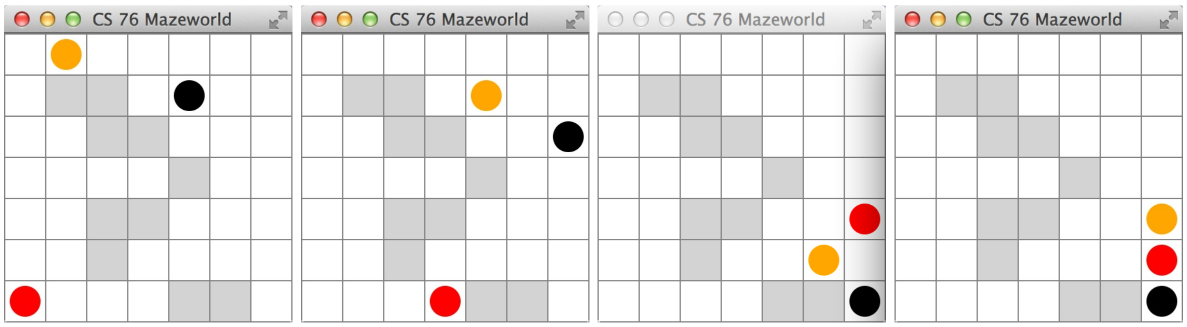

Figure 2: test result of example

Listing 10: test of example

```
A*:
A* path length:  28 [[0, 0, 1, 6, 4, 5], [1, 0, 1, 6, 4, 5], [1, 0, 2, 6, 4,
    5], [1, 0, 2, 6, 4, 4], [1, 0, 3, 6, 4, 4], [2, 0, 3, 6, 4, 4], [3, 0, 3,
    6, 4, 4], [3, 0, 3, 6, 5, 4], [3, 0, 3, 5, 5, 4], [3, 0, 4, 5, 5, 4], [3,
    0, 4, 5, 6, 4], [3, 0, 4, 5, 6, 3], [3, 0, 4, 4, 6, 3], [3, 0, 5, 4, 6,
    3], [3, 1, 5, 4, 6, 3], [4, 1, 5, 4, 6, 3], [4, 1, 5, 4, 6, 2], [4, 1, 5,
    4, 6, 1], [4, 1, 5, 3, 6, 1], [4, 1, 5, 2, 6, 1], [4, 1, 5, 1, 6, 1], [4,
    1, 5, 1, 6, 0], [4, 2, 5, 1, 6, 0], [5, 2, 5, 1, 6, 0], [6, 2, 5, 1, 6,
    0], [6, 1, 5, 1, 6, 0], [6, 1, 5, 2, 6, 0], [6, 1, 6, 2, 6, 0]]
```

3.6 Test on 5*5 Maze

Here is the first test on 5*5, the starting position of these there robots are: A(0,0), B(0,1), C(0,2). And the goal position are: A(4,2), B(4,1), B(4,0). There are two walls in the middle and the robots should find its right order of goal states.

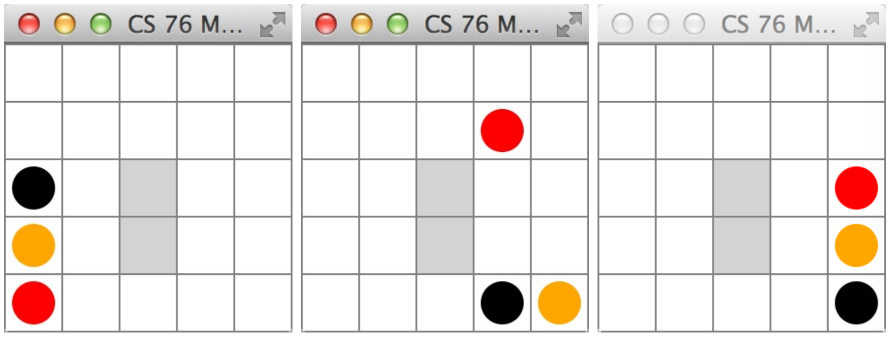

Figure 3: test result of 5*5

Listing 11: test of 5*5

```
A*:
A* path length:  23 [[0, 0, 0, 1, 0, 2], [1, 0, 0, 1, 0, 2], [1, 0, 0, 0, 0,
    2], [1, 1, 0, 0, 0, 2], [1, 1, 1, 0, 0, 2], [1, 2, 1, 0, 0, 2], [1, 3, 1,
    0, 0, 2], [1, 3, 2, 0, 0, 2], [1, 3, 2, 0, 1, 2], [1, 3, 2, 0, 1, 1], [1,
    3, 3, 0, 1, 1], [2, 3, 3, 0, 1, 1], [2, 3, 3, 0, 1, 0], [2, 3, 3, 0, 2,
    0], [2, 3, 4, 0, 2, 0], [3, 3, 4, 0, 2, 0], [3, 3, 4, 0, 3, 0], [3, 3, 4,
    1, 3, 0], [3, 3, 3, 1, 3, 0], [3, 3, 3, 1, 4, 0], [3, 3, 4, 1, 4, 0], [4,
    3, 4, 1, 4, 0], [4, 2, 4, 1, 4, 0]]
```

The second test on 5*5 is shown here, it is funny here because I set all walls on them around and there are two robots whose start positions are (1,1), (1,2) and destinations are (2,2), (2,1). In my test although they reach their destinations but not the optimal solution. Since the islegal() test is not perfectly check all possibilities. At this situation, if robotA firstly take 'wait' action and robotB take 'north' action and then robotA should take 'north' action which will step on the previous node of robotB. And follow this pattern, the path of them is more close to realize and more intelligent.

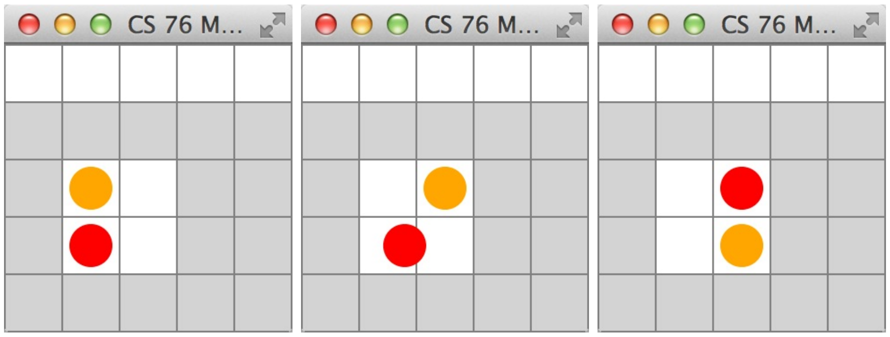

Figure 4: test result of 5*5

Listing 12: test of 5*5

```
A*:
A* path length:  7 [[1, 1, 1, 2], [2, 1, 1, 2], [2, 1, 2, 2], [1, 1, 2, 2],
    [1, 2, 2, 2], [1, 2, 2, 1], [2, 2, 2, 1]]
```

3.7 Test on 40*40 Maze

Since 40*40 maze will be really big maze and if I run 2 or 3 robots on this maze will need to wait at least 5 minutes, I used more easy one robot to test on it. I give one exit on the first layer of wall and two exits on the second layer of wall and the start node is (0,0) and the goal node is (39, 39). Clearly, the robot found the closer exit to reach its goal. The test result is as shown below in figure 5.

Listing 13: test of 40*40

```
A*:
A* path length:  79 [[0, 0], [0, 1], [1, 1], [2, 1], [3, 1], [4, 1], [5, 1],
    [6, 1], [6, 2], [6, 3], [7, 3], [8, 3], [9, 3], [10, 3], [11, 3], [12,
    3], [12, 4], [13, 4], [13, 5], [14, 5], [15, 5], [16, 5], [17, 5], [18,
    5], [19, 5], [20, 5], [20, 6], [20, 7], [20, 8], [20, 9], [20, 10], [20,
    11], [20, 12], [20, 13], [20, 14], [20, 15], [20, 16], [20, 17], [20,
    18], [20, 19], [21, 19], [22, 19], [23, 19], [24, 19], [25, 19], [26,
    19], [27, 19], [27, 20], [28, 20], [29, 20], [29, 21], [30, 21], [31,
    21], [32, 21], [33, 21], [33, 22], [34, 22], [34, 23], [35, 23], [36,
    23], [36, 24], [36, 25], [36, 26], [36, 27], [36, 28], [36, 29], [36,
    30], [36, 31], [36, 32], [36, 33], [36, 34], [36, 35], [36, 36], [36,
    37], [36, 38], [36, 39], [37, 39], [38, 39], [39, 39]]
```

Also, I have another test of two robots with starting position (0,0), (0,1) and destination position of (39,39) and (39, 38). The robots can find the right exit to get to the goal node. The test result is as shown below at figure 6.

Listing 14: test of 40*40

```
A*:
A* path length:  155 [[0, 0, 0, 1], [1, 0, 0, 1], [1, 0, 0, 2], [2, 0, 0, 2],
    [3, 0, 0, 2], [3, 1, 0, 2], [3, 2, 0, 2], [3, 2, 1, 2], [3, 3, 1, 2], [3,
    3, 2, 2], [3, 4, 2, 2], [3, 4, 3, 2], [4, 4, 3, 2], [4, 4, 4, 2], [4, 4,
    5, 2], [4, 4, 6, 2], [4, 4, 7, 2], [5, 4, 7, 2], [6, 4, 7, 2], [6, 4, 8,
    2], [6, 4, 9, 2], [6, 4, 10, 2], [7, 4, 10, 2], [8, 4, 10, 2], [9, 4, 10,
    2], [9, 5, 10, 2], [10, 5, 10, 2], [11, 5, 10, 2], [11, 5, 11, 2], [12,
    5, 11, 2], [13, 5, 11, 2], [14, 5, 11, 2], [15, 5, 11, 2], [15, 5, 12,
    2], [15, 5, 13, 2], [16, 5, 13, 2], [17, 5, 13, 2], [17, 5, 14, 2], [18,
    5, 14, 2], [19, 5, 14, 2], [20, 5, 14, 2], [20, 6, 14, 2], [20, 7, 14,
    2], [20, 7, 14, 3], [20, 7, 15, 3], [20, 8, 15, 3], [20, 8, 16, 3], [20,
    9, 16, 3], [20, 10, 16, 3], [20, 10, 17, 3], [20, 10, 18, 3], [20, 10,
    18, 4], [20, 10, 18, 5], [20, 10, 19, 5], [21, 10, 19, 5], [21, 10, 20,
    5], [22, 10, 20, 5], [22, 10, 20, 6], [22, 11, 20, 6], [22, 11, 20, 7],
    [22, 11, 20, 8], [22, 11, 20, 9], [22, 11, 21, 9], [22, 11, 21, 10], [22,
    11, 22, 10], [23, 11, 22, 10], [24, 11, 22, 10], [24, 11, 22, 11], [25,
    11, 22, 11], [26, 11, 22, 11], [26, 12, 22, 11], [26, 12, 22, 12], [26,
    13, 22, 12], [26, 13, 23, 12], [26, 13, 23, 13], [26, 14, 23, 13], [26,
    15, 23, 13], [26, 16, 23, 13], [26, 16, 23, 14], [26, 16, 23, 15], [26,
    16, 23, 16], [26, 17, 23, 16], [27, 17, 23, 16], [27, 18, 23, 16], [27,
    19, 23, 16], [28, 19, 23, 16], [28, 20, 23, 16], [28, 21, 23, 16], [28,
    22, 23, 16], [28, 22, 23, 17], [28, 23, 23, 17], [28, 24, 23, 17], [28,
    24, 24, 17], [28, 24, 25, 17], [28, 24, 26, 17], [29, 24, 26, 17], [30,
    24, 26, 17], [30, 24, 27, 17], [30, 24, 27, 18], [30, 24, 27, 19], [30,
    24, 27, 20], [30, 24, 27, 21], [30, 24, 27, 22], [30, 24, 27, 23], [30,
    24, 27, 24], [31, 24, 27, 24], [32, 24, 27, 24], [33, 24, 27, 24], [34,
    24, 27, 24], [34, 24, 28, 24], [34, 24, 28, 25], [34, 24, 28, 26], [34,
    24, 28, 27], [34, 24, 28, 28], [34, 25, 28, 28], [34, 25, 28, 29], [34,
    25, 28, 30], [34, 25, 28, 31], [34, 25, 29, 31], [34, 25, 29, 32], [35,
    25, 29, 32], [35, 25, 30, 32], [35, 25, 30, 33], [35, 25, 30, 34], [35,
    25, 30, 35], [35, 25, 30, 36], [35, 25, 31, 36], [35, 25, 31, 37], [35,
    25, 32, 37], [35, 25, 33, 37], [36, 25, 33, 37], [36, 26, 33, 37], [36,
    27, 33, 37], [36, 28, 33, 37], [36, 29, 33, 37], [36, 30, 33, 37], [36,
    30, 34, 37], [36, 31, 34, 37], [37, 31, 34, 37], [37, 31, 35, 37], [37,
    32, 35, 37], [37, 33, 35, 37], [37, 33, 36, 37], [37, 34, 36, 37], [37,
    35, 36, 37], [37, 35, 37, 37], [37, 35, 38, 37], [37, 35, 39, 37], [37,
    36, 39, 37], [37, 37, 39, 37], [37, 38, 39, 37], [37, 38, 39, 38], [38,
    38, 39, 38], [38, 39, 39, 38], [39, 39, 39, 38]]
```

3.8 Question

3.8.1 Question 1

Q: If there are k robots, how would you represent the state of the system?
A: I represent them as a array with 2*k elements and treat them as a single state.

3.8.2 Question 2

Q: Give an upper bound on the number of states in the system, in terms of n and k.
A: I think the answer is 5*k*n. Since each step can only have one robot to move, the branching factor becomes 5 for each robot(north, west, east, south, wait) and there are k robots and will go to depth of n.

3.8.3 Question 3

Q: Give a rough estimate on how many of these states represent collisions if the number of wall squares is w, and n is much larger than k.
A: I think the answer is k^w. Each robot will have k possibilities to have collision with wall and then I think the answer is k^w.

3.8.4 Question 4

Q: If there are not many walls, n is large (say 100x100), and several robots (say 10), do you expect a straightforwards breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not?
A: I don't think so. And I also think it depends on different situations . Although the questions says that there are not many walls and n is large, but it does not show how those walls established in the Maze. Different setting of walls on the maze will affect the result of BFS.

3.8.5 Question 5

Q: Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic.
A: I use manhattan distance and I have already described early in the report and it is obvious monotonic since the destination is certain and calculation steps are all monotonic.

3.8.6 Question 6

Q: Implement a model of the system and use A* search to find some paths. Test your program on mazes with between one and three robots, of sizes varying from 5x5 to 40x40.
A: I have done that and report it previously.

3.8.7 Question 7

Q: Describe why the 8-puzzle in the book is a special case of this problem. Is the heuristic function you chose a good one for the 8-puzzle?
A: 8-puzzle problem and be treated as there are 8 robots on a 3*3 maze and only one robot can move at each time and there is only one available cell on the maze, thus each time can only two robots have the possibility to move and the final state should be placing those 8 robots in a clockwise order.
I do not think my heuristic function would be a very good one for this special problem. And I think there should be some improvements to add those robots' order number onto the heuristic function and then it will be more proper for it.

3.8.8 Question 8

Q: The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this.

A: The two disjoint sets are such that any state is reachable from any other state in the same state while no state is reachable from the other set. Therefore, I should modify the **isLegal()** function to check if after a legal move an odd N remains odd while an even N keeps even (N is let the sum of the total number of inversions). Therefore, when N = 0, it can only be reached from starting states with even N not with odd N.

4 Blind robot with Pacman physics

Blind robot is a special case that the robot can move 4 directions but does not know where it starts. So the problem solving becomes to help the robot find its position at the same time he is finding the path to the goal. Then I can print out the result of planning for him to show how to choose to move each time.

4.1 Imeplementation

The mainly part of implementation is to understand this problem and design new data structure to represent it. As is described in the question, at the beginning, the robot does not know where he is. Then, all the cells on the Maze except those walls are all possible starting position for it. In order to remove some possibilities of the starting position, the robot can move one step to know the possibilities position of its second step. For example, if the possibilities of starting position are 10, after the robot moves 1 step to north, the possibilities of the position of second step becomes 8. While, if the robot takes a step to east, the possibilities of the position for second step may be 6. And or if move west, the number becomes 5 and move south the number becomes 2. Therefore, the problem becomes how to choose the best and proper direction for the next move. Then, I applied A* Search to help choose each step. And obviously, each time the successors are all four and the A* will use heuristic() to help choose one move.

In order to solve this question, I simply build a set(HashSet) to store all possible states at each time. And then, there will be a blindMazeNode implements SearchNode to have structure of the possible states node and the cost. And I added a factor named direction to record which direction to take at each move and then I modified toString() to print out those directions.

Here is the code of my new data structure and it contains a function allpossibleStates() to help get all possible states at the beginning for the start node.

Listing 15: Data structure

```
//SimpleMazeNode is just a structure to store the position of robot
//and the cost. (x, y, cost).
  public class SimpleMazeNode{
        //state[0] is x coordinate state[1] is y coordinate
        protected int[] state;
        private double cost;

        public SimpleMazeNode(int x, int y, double c) {
          state = new int[2];
          this.state[0] = x;
          this.state[1] = y;
          cost = c;
          }
        }
```

```
//BlindSearchNode is used to store the possible states set + cost + direction
    for each node
public class BlindSearchNode implements SearchNode{
  protected HashSet<SimpleMazeNode> possible_states;
  private double cost;
  public String direction;

  public BlindSearchNode(HashSet<SimpleMazeNode> s, double c, String d) {
     possible_states = new HashSet<SimpleMazeNode>();
     possible_states = s;
     cost = c;
     direction = d;
     }
     }
```

Therefore, with these two data structures and I can then clearly form the particular problem, the BlindRobotProblem:

Listing 16: Blind Robot Problem

```
public BlindRobotProblem(Maze m,int sx, int sy, int gx, int gy) {
                xStart = sx;
                yStart = sy;
                xGoal = gx;
                yGoal = gy;
                maze = m;
                startPos = new SimpleMazeNode(sx, sy, 0);
                //get all possible positions of the starting cells
                startStates = allpossibleStates();
                startNode = new BlindSearchNode(startStates,0,"null");
        }
```

In the constructor of BlindRobotProblem, I should initially have all possible states for the starting position and therefore I wrote a function to get all nodes with "." represent to be a legal cell on the maze. In other words, all legal cells on the maze are all possible starting node for this robot. As is a discussion on Piazza, a **HashSet** data structure is better used here to both store those simple nodes and to search in the set. The code of allpossibleStates() is shown below:

Listing 17: Get all possible states

```
public HashSet<SimpleMazeNode> allpossibleStates(){
  //Using a hash set to store all possible beginning positions
  HashSet<SimpleMazeNode> apStates = new HashSet<SimpleMazeNode>();
    for (int x = 0; y < maze.width; x++) {
       for (int y = 0; x < maze.height; y++) {
            //if the cell is legal on the maze, then it must be one
                possibility
            if (maze.isLegal(x, y)){
               SimpleMazeNode cell = new SimpleMazeNode(x,y,0);
               apStates.add(cell);
               }
             }
          }
        return apStates;
        }
```

And the getSuccessors() is modified to help found the most proper direction of move for the next step regarding to the current position. For the current position, it will check all four direction and they are all successors. Each successor will be a node to store if taking this direction, what are the possible positions for the next step. The code is shown here:

Listing 18: getSuccessors()

```
public ArrayList<SearchNode> getSuccessors() {
  ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
  for (int[] action: actions) {
   //for each action to store the possible states for it
    HashSet<SimpleMazeNode> newPossibleStates = new HashSet<SimpleMazeNode>();
    String direction = "";
    if(action[0] == 0&&action[1] == 1){
      direction = "North";
      }else if(action[0] == 0&&action[1] == -1){
        direction = "South";
      }else if(action[0] == 1&&action[1] == 0){
        direction = "East";
      }else{
        direction = "West";
        }

    for(SimpleMazeNode pState: this.states){
      int xNew = pState.state[0] + action[0];
      int yNew = pState.state[1] + action[1];
      int xOld = pState.state[0];
      int yOld = pState.state[1];

      //is the new move is legal means it can move to a new cell for the next
          step
      //if is not legal, then it will hit the wall and back to it current
          cell for the next step
      if(maze.isLegal(xNew, yNew)) {
          //if newPossibleState is empty or newpMove has not appeared in
              newPossibleStates
          //then add newpMove into newPossibleStates.
          SimpleMazeNode newpMove = new SimpleMazeNode(xNew, yNew, getCost()
              + 1.0);

          if(!newPossibleStates.contains(newpMove) ||
              newPossibleStates.size()==0) {
                  newPossibleStates.add(newpMove);
                  }
            }else{
              //if newPossibleState is empty or oldpMove has not appeared
                  in newPossibleStates
              //then add oldpMove into newPossibleStates.
              SimpleMazeNode oldpMove = new SimpleMazeNode(xOld,yOld,
                  getCost() + 1.0);
```

```
                    if(!newPossibleStates.contains(oldpMove) ||
                        newPossibleStates.size()==0) {
                         newPossibleStates.add(oldpMove);
                      }
                    }
                  }
                 //a successor has been gotten from the newPossibleStates
                SearchNode succ =
                new BlindSearchNode(newPossibleStates, getCost() +
                    1.0,direction);
                successors.add(succ);
                }
                //after the iteration of actions, return successors
                return successors;
            }
```

Since the searching still need A* algorithm, the heuristic() function of this problem is shown below and I still used manhattan distance(Also, there will be another way to calculate it, like I have discussed on section **3.3 heuristic()**, to calculate the minimum d[x] and d[y] and then get the Manhattan distance.) :

Listing 19: heuristic()

```
  @Override
  public double heuristic() {
    double []dx = new double[this.possible_states.size()];
    double []dy = new double[this.possible_states.size()];
    double sumx=0.0;
    double sumy=0.0;

      for(int i=0;i<this.possible_states.size();i++){
        dx[i] = xGoal - possible_states.get(i).state[0];
        dy[i] = yGoal - possible_states.get(i).state[1];
        }

      for(int i=0;i<this.possible_states.size();i++){
        sumx += dx[i];
        sumy += dy[i];
        }
         return Math.abs(sumx) + Math.abs(sumy);
      }
```

4.2 Test Result

```
......
......
.....#
####..
......
......
```

For maze above and goal position is (5,0). The test result is:

Listing 20: Maze 6*6

```
A*:
a* path:  [South, South, East, East, East, East, South, East]
  Nodes explored during search:  9
  Maximum space usage during search 58
```

```
........
........
.....#..
####....
........
........
....##..
##......
```

For maze above and goal position is (7,0). The test result is:

Listing 21: Maze 8*8

```
A*:
a* path:  [South, South, East, East, East, East, South, South, South, East]
  Nodes explored during search:  11
  Maximum space usage during search 72
```

```
.......
.##....
..##...
.......
..##...
#.###..
....##.
```

For maze above and goal position is (6,6). The test result is:

Listing 22: Maze 7*7

```
A*:
a* path:  [East, East, East, North, North, North, East, North, East, East,
   North, North, East, East, East, West, North, North, West, North, East,
   East, North, North, East, East, East, East, North, North, North, East]
  Nodes explored during search:  248
  Maximum space usage during search 857
```

From my test results, I think my heuristic() function is not bad to give a motion plan for the blind robot. And my heurisitc() still used Mahattan distance to calculate.

4.3 Polynomial-time blind robot planning

In this planning problem, if a robot is sensorless, then it means that the robot not only has no idea of its location but also it cannot tell the directions. Then in this case, if the size of the maze is finite and the goal is in the same connected component of the maze as the start, a sensorless plan will always exists.

If there exist a maze map that no matter which direction to take and the next possible states of this robot will decrease only 1. Then, the robot will finally reach a state set which the number of possible state is 1. Then it will finally satisfy the goal test and produce a plan.

My algorithm is to build a directed graph according to the maze. Each node of this graph will be a set of possible states and each edge will be a sequence of actions. It means, if I have a node v and a node v' and there is an edge between them. Therefore, any state in node v can using the action sequence on this edge go to one of a state in v'. In order to always have a plan, there should not be any "dead state" for the blind and sensorless robot, which means that the robot cannot be at a state that it cannot reach other state. Therefore, I will then traverse this directed graph and find for each node there will be a path that it can reach to the goal state(no dead state and no deadlock between each nodes).

So for example, if there is a special maze map for each step there is only 1 possible state decrease, then the directed graph will be very clear and then there will alway have a plan for the robot to reach the goal.

5 Paper Review

Among those paper lists, I choose one that related to A* search of multi-robots path-finding problem. The reason why I choose this paper is because I think it can help me understand much deeper about the A* Search Algorithm and I am also interested in multi-robots path-finding problem since it contains so many valuable research points such as how to find the most optimal path, or how to reduce the running time as well as how to choose heuristic function.

The paper I read is written by Trevor Standley, named *Finding Optimal Solutions to Cooperative Pathfinding Problems*. The whole paper is clearly structured with firstly talking about the related work of pathfinding problems and secondly discussing the formulation of cooperative pathfinding problem and finally it gives the standard algorithm and two improvements. The paper gives a modified algorithm that is complete and find optimal solutions in a very quick time.

For part of related work, this paper clearly introduce two previous modified A* algorithm, LRA* and HCA*. Not only introduce the advantages of these two algorithms, but also the paper points out the disadvantages of them and in which they are those the paper focus on solving. That is, these methods limit the types of paths that agents can follow and the problem of dead lock.

Then, the paper comes to the problem formulation of cooperative pathfinding problem. The formulation is clearly introduced, and reader could easily follow this formulation to create standard model of robots and mazes and actions. After that, the standard admissible algorithm is discussed in this paper. It gives me a hint to build my multi-robot model which represent each state to be all positions of all robots, known as operator decomposition(OD). Also, actions are in a set expect adding NE,SE, SW, NW to the actions set, which is more complicated than our assignment.

Then, the paper talks about another improvement of the standard algorithm, that is independence detection. The paper introduces a few interesting and special examples to show how important to do independence detection. Given two pieces of pseudocode, the algorithm is clearly shown to the reader. Finally, the test result is simple but strong to show its improvement on the standard algorithm.

All in all, this paper is a good one for me to not only know about A* algorithm but also understand multi-robots path-finding problem. Moreover, it gives me much more deep thinking about how to improve the standard algorithm and what problems will I have when building such models. For my suggestion of improvement, I think this paper could spend more on its own works and improvements of the standard algorithm, also, I think more test results and pictures are welcomed for readers to understand better of the writer's work. Except the suggestion from me, I think the paper is very worth reading.

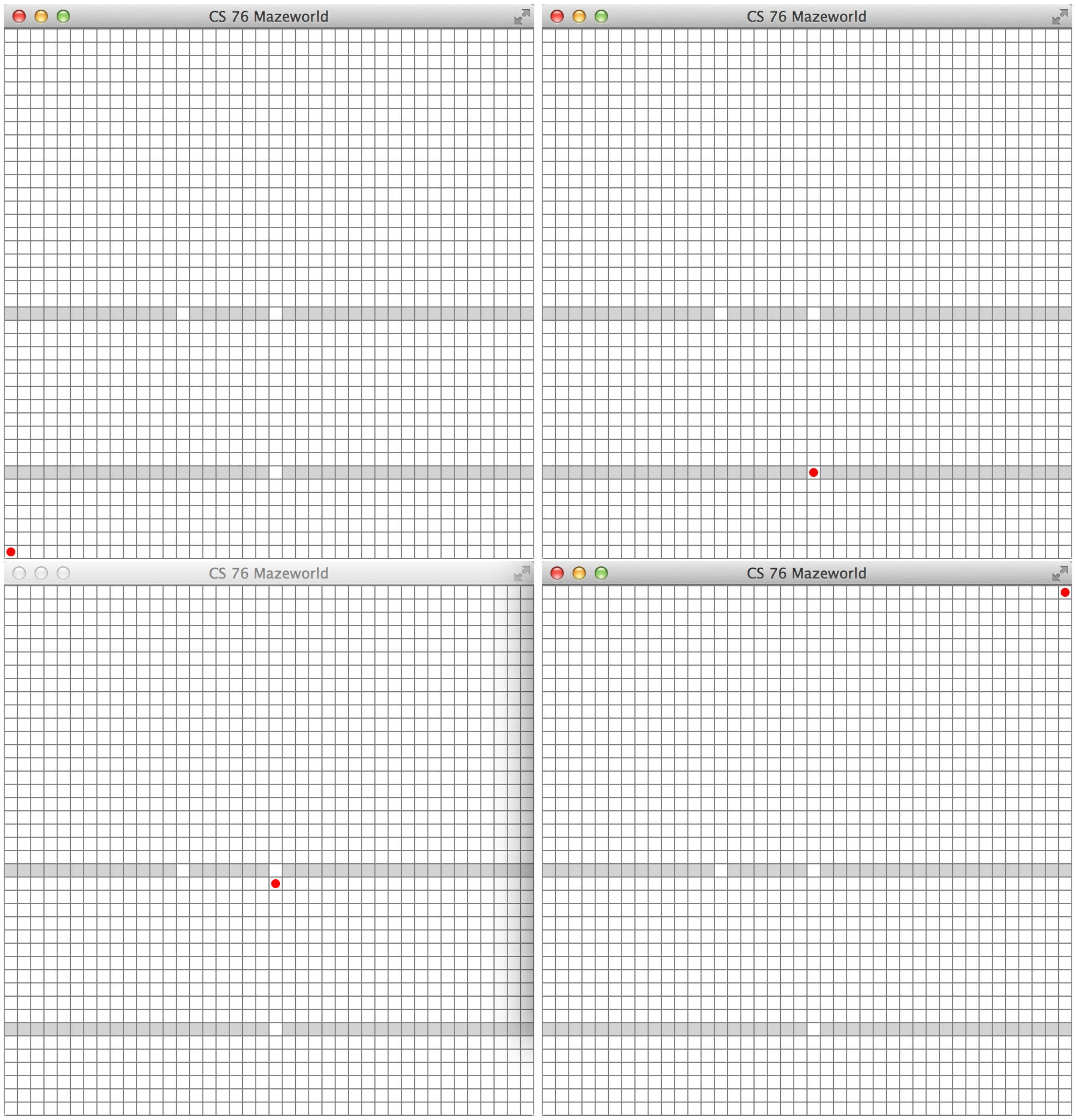

Figure 5: test result of 40*40(1)

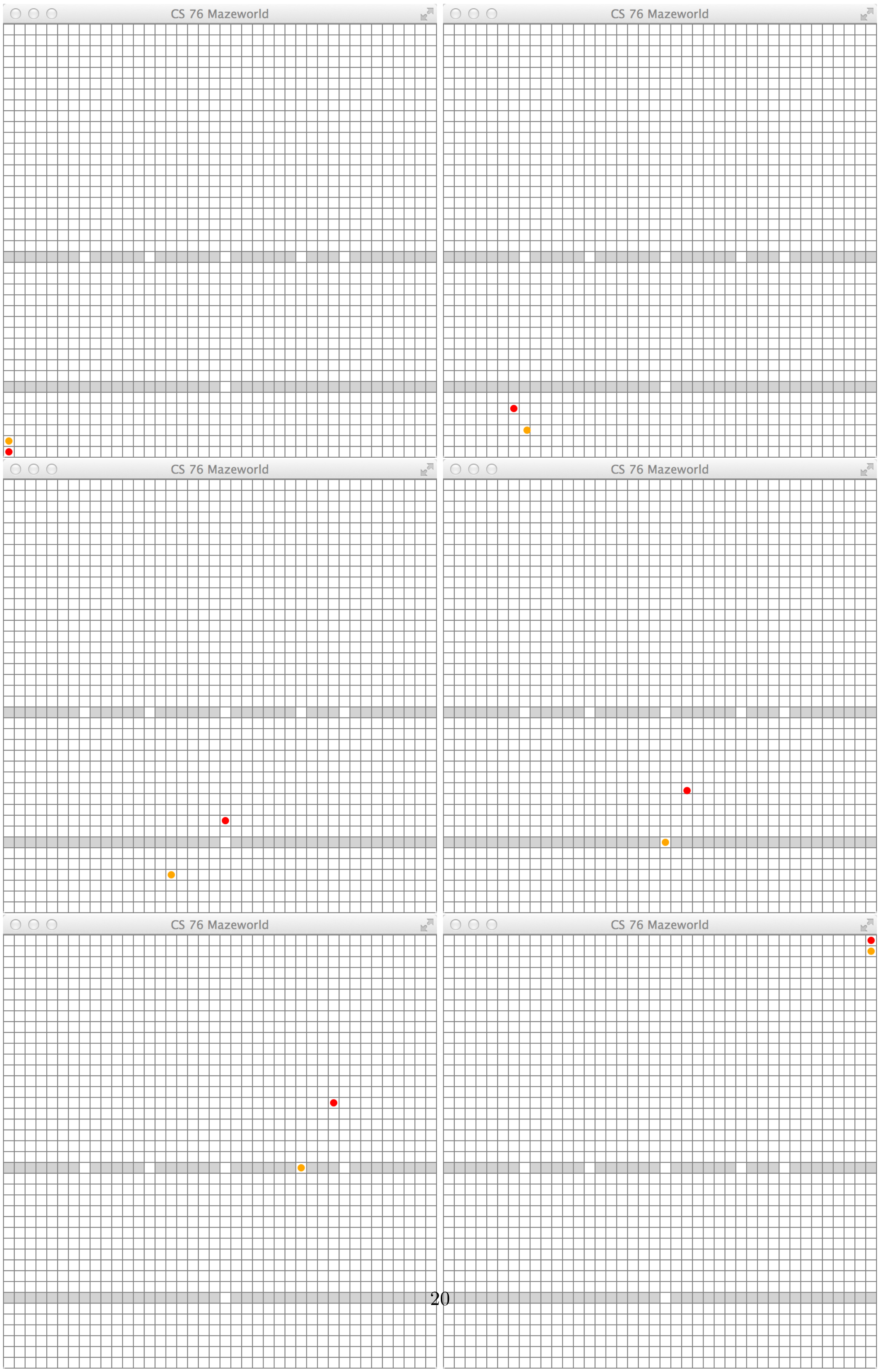

Figure 6: test result of 40*40(2)