

Mazeworld

Mengjia Kong

January 23, 2014

1 Introduction

MazeWorld is a typical problem in Artificial Intelligence, as well as Cannibals and missionaries. The description is that there is a maze with some walls and a robot. The robot is initially in a location in the maze, which we regarded as the initial state. Then the robot wants to go to another location in the maze, which we defined as a goal state. But there are lots of walls in the maze. What we need to do is to help the robot to reach the destination.

Here is a simple maze, drawn in the venerable tradition of ASCII art:

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

The periods represent open floor tiles (places that can be walked on or over), and the number signs represent walls (place where a simple robot cannot go).

The robot can move in any of four directions: North (towards the top of the screen), East, South, and West. There are coordinates on the maze. (0, 0) is the bottom left corner of the maze. (6, 0) is the bottom right corner of the maze.

This is a simple maze problem, which we will using as a problem description to introduce *A-star search algorithm* in section 2. After that we will introduce something more difficultsection 3, multiple robots coordination in a maze; section 4, blind robot with pacman physics in a maze.

2 Previous work

Note: I'm a graduate student

Many scholars have explored the problems in A-star, which need to be improved. I will introduce some problems and improvements have been introduce to the world below:

There are pitfalls of modern incomplete and inadmissible algorithms for the cooperative pathfinding problem. For this problem, two improvements are introduced to the standard admissible algorithm: Operator decomposition, which reduce the branching factor of the problem, allowed the algorithm to consider only a small faction of the successor of each node efficiently; Independence detection, which allowed the paths of various agents to be computed independently, still with optimality. Details are in [1].

In optimal path finding for multiple agents, agents must collide and their travel cost should be minimized. In [2], there is a search tree called the increasing cost tree and a corresponding search algorithm that optimal solutions. The ICTS algorithm can solve optimally MAPF and a pairwise pruning enhancement further sppeds up ICTS.

3 A-star search Algorithm

The most widely known form of best-first search is called A-star search. Best-first search is the general approach of informed search, which is opposite to uninformed search which is introduced in last assignment.

Best-first search is an instance of the general tree-search or graph-search algorithm in which a node is selected for expansion based on **evaluation function**, $f(n)$. The evaluation function is construed as a cost estimate. Most best-first algorithms include as a component of **evaluation function** $f(n)$ a **heuristic function** $h(n)$:

$h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

A star search is the most widely known form of best-first search. It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$f(n)$ = estimated cost of the cheapest solution through n

Provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to *Uniform-cost search* except that A star uses $g + h$ instead of g . Moreover, in computer science, *A star* is a computer algorithm that is widely used in pathfinding and graph traversal.

3.1 Implementation

The algorithm is implemented in `InformedSearchProblem.java`.

Here's my code for `astarSearch`:

```
public List<SearchNode> astarSearch() {

    resetStats();

    // visited = explored + frontier
    // use visited to get priority of some nodes in frontier
    Queue<SearchNode> frontier = new PriorityQueue<SearchNode>();
    HashMap<SearchNode, Double> visited = new HashMap<SearchNode, Double>();

    frontier.add(startNode);
    visited.put(startNode, startNode.priority());

    while (!frontier.isEmpty()) {
        incrementNodeCount();

        updateMemory(frontier.size() + visited.size());

        SearchNode currentNode = frontier.remove();

        // node has been explored or there is a node in frontier has less priority
        if (visited.containsKey(currentNode) && currentNode.priority()
            > visited.get(currentNode)) {
            continue;
        }
    }
}
```

```

    }

    if (currentNode.goalTest()) {
        return backchain(currentNode);
    }

    ArrayList<SearchNode> successors = currentNode.getSuccessors();

    // System.out.println("\ncurrent " + currentNode + "successors " + successors);

    for (SearchNode node : successors) {
        // if not visited
        if (!visited.containsKey(node)) {
            frontier.add(node);
            visited.put(node, node.priority());
        }
        // if in frontier with less priority, replace it with node
        else if (frontier.contains(node)) {
            double priority = visited.get(node);
            if (node.priority() < priority) {
                frontier.remove(node);
                frontier.add(node);
                visited.remove(node);
                visited.put(node, node.priority());
            }
        }
    }
}

return null;
}

// backchain should only be used by bfs, not the recursive dfs
protected List<SearchNode> backchain(SearchNode node) {

    LinkedList<SearchNode> solution = new LinkedList<SearchNode>();

    // chain through the visited hashmap to find each previous node,
    // add to the solution
    while (node != null) {
        solution.addFirst(node);
        node = node.getParent();
    }

    return solution;
}

```

The basic idea of my implementation is to explore the node in frontier with the least $f = g + h$. At first, I add `startNode` to frontier, then if frontier is not empty, I always get the node with least f and add its successors to frontier. But before I explore the node, I will check whether the node has been explored and whether it is what I want, the goal. If the node has been explored, the current node does not need

to be explored because its f cannot be less than that in explored set. If the node is the goal, I have got the solution! When I get successors, I need to check whether the successor has been explored or has been ready to be explored. If the successor has not been explored and is not in frontier, I put it in frontier. If the successor has not been explored but is in frontier, replace the nodes in frontier with higher f with the current node. If the successor has been explored but not in frontier, it means that the node explored must have less f than current node. So just ignore the repeated node.

To implement a-star, for convenience, I chose some special data structures. Here comes reasons.

A-star chooses the node with least $f = g + h$ to be explored. In my implementation, I define f as *priority* in `SimpleMazeNode`. Because we need to choose the least f in frontier, I used a `PriorityQueue` to store the frontier, in which the nodes will be explored in recent future. In the `frontier`, the node with the least priority will be popped out to be explored first, just like the definition of A-star. In `SimpleMazeProblem.java`, there is a function in class `SimpleMazeNode`, `compareTo`, which defines that if the nodes in `PriorityQueue` are type of `SimpleMazeNode`, they will be ordered by priority. Here is the code of `compareTo`.

```
@Override
public int compareTo(SearchNode o) {
    return (int) Math.signum(priority() - o.priority());
}
```

Both nodes which have been explored and nodes which are in the frontier are stored in *visited* as a type, `HashMap<SearchNode, Double>`, which stores a map from `searchNode` to priority of the node. It is convenient to get the priority of nodes in frontier by using this data structure. A-star needs to check whether the current node has a less priority than the same node has been in frontier. Using this `HashMap`, I can get the priority in frontier very quickly and replace it with current node quickly. By deleting the previous node and adding the new one in *frontier* and *visited* when the conditions are satisfied, I always keep nodes in *frontier* and *visited* are not repeated and with the least priority at that time.

3.2 Test in a simple maze problem

The implementation of the simple maze problem has been implemented by teacher in `SimpleMazeProblem.java`. I have made some small changes to it. There will be `state` which is coordination of location actually, `cost` and `parent` in a search node. `parent` which I added is to help get the path when backchaining.

I will exhibit A-star search by comparing the results with BFS and DFS.

3.2.1 An example of 7*7 maze

Given an example of 7*7 maze:

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

The initial location is (0, 0) and the goal location is (6, 0).

BFS, DFS, A-star choose different paths. Showing all pictures to see the path consumes too much space in this paper. I only show a picture, in which we can see a state in maze when three algorithms start at the same time, as shown in Figure 1.

The red circle goes with BFS: $(0, 0) \rightarrow (1, 0) \rightarrow (1, 3) \rightarrow (6, 3) \rightarrow (6, 0)$;

The orange circle goes with DFS: $(0, 0) \rightarrow (1, 0) \rightarrow (1, 4) \rightarrow (0, 4) \rightarrow (0, 6) \rightarrow (6, 6) \rightarrow (6, 0)$;

The black circle goes with A-star: $(0, 0) \rightarrow (1, 0) \rightarrow (1, 3) \rightarrow (5, 3) \rightarrow (5, 1) \rightarrow (6, 1) \rightarrow (6, 0)$.

The experimental data about number of nodes explored and space usage is shown in Figure 2.

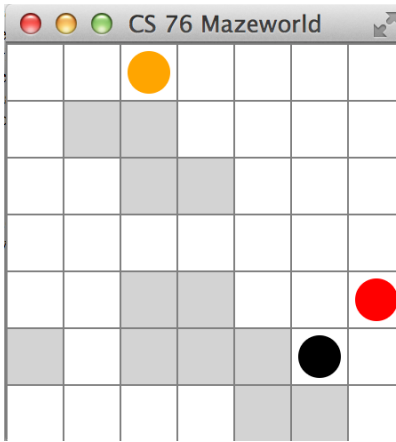


Figure 1: Maze 7*7: paths starting together, one state with cost=10

```

BFS:
  Nodes explored during search: 37
  Maximum space usage during search 41
DFS:
  Nodes explored during search: 21
  Maximum space usage during search 21
A*:
  Nodes explored during search: 19
  Maximum space usage during search 30

```

Figure 2: Maze 7*7: experimental results

3.2.2 An example of 20*20 maze

Given an example of 20*20 maze, the initial location is (0, 0) and the goal location is (6, 0).

BFS, DFS, A-star choose different paths. I don't waste space to give the details of the paths. You can see they have different paths by showing a state in maze in Figure 3.

The experimental data about number of nodes explored and space usage is shown in Figure 4.

3.2.3 Analysis of experimental results

BFS, DFS, A-star may get different paths from the same input. BFS and A-star get the shortest paths because they are optimal. All of three can find a path due to the completeness. According to the experimental outputs, A-star always explores fewer nodes than other two. But DFS always can get a better space usage. Moreover, in my implementation of A-star, heuristic function $h(n)$ is consistent, so A-star is optimal. $h(n)$ is below:

```

public double heuristic() {
    // manhattan distance metric for simple maze with one agent:
    double dx = xGoal - state[0];
    double dy = yGoal - state[1];
    return Math.abs(dx) + Math.abs(dy);
}

```

Obviously, for each node n and each successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching

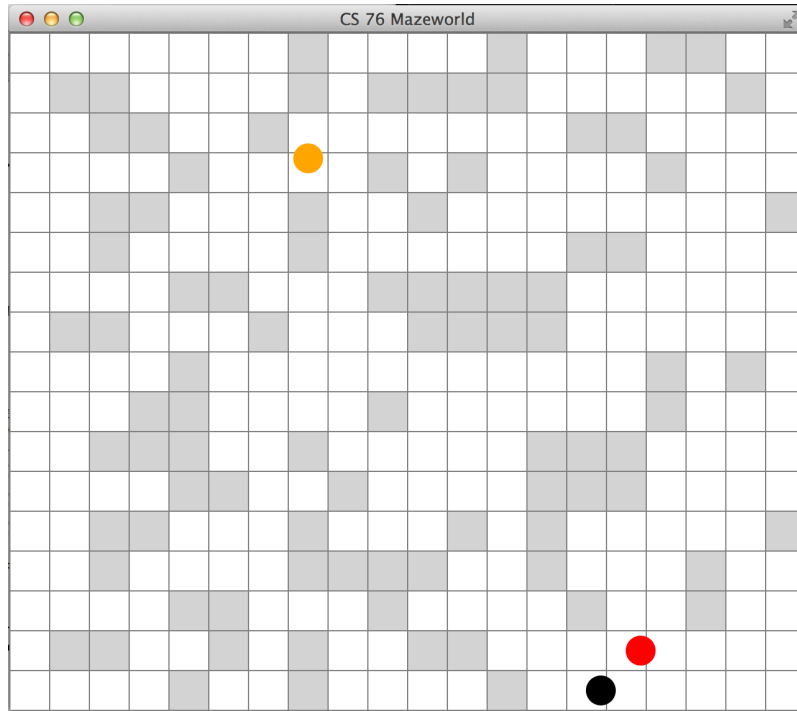


Figure 3: Maze 20*20: paths starting together, 3 locations at that time

the goal from n' :

$$h(n) \leq c(n, a, n') + h(n')$$

Because the method of $h(n)$ is manhattan distance, which satisfies the *triangle inequality* above. So $h(n)$ is proved as consistent. Then I can get a optimal A-star.

4 Multi-robot coordination

Multi-robot maze problem can be described like that: k robots live in an $n * n$ rectangular maze. The coordinates of each robot are (x_i, y_i) , and each coordinate is an integer in the range $0..n-1$. For example, maybe the maze and initials location of robots look like Figure 5:

```

BFS:
  Nodes explored during search: 245
  Maximum space usage during search 253
DFS:
  Nodes explored during search: 115
  Maximum space usage during search 106
Maze state 2, 2 depth 4.0
A*:
  Nodes explored during search: 80
  Maximum space usage during search 115

```

Figure 4: Maze 20*20: experimental results

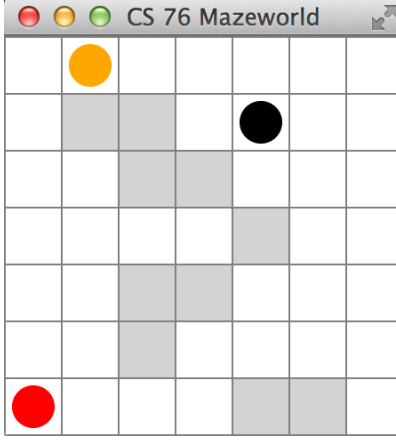


Figure 5: Initial State of 3 robots in 7*7 maze

In this maze, initial coordination of robot A represented by red circle is (0, 0); initial coordination of robot B represented by orange circle is (1, 6); initial coordination of robot C represented by black circle is (4, 5).

The goal may like Figure 6.

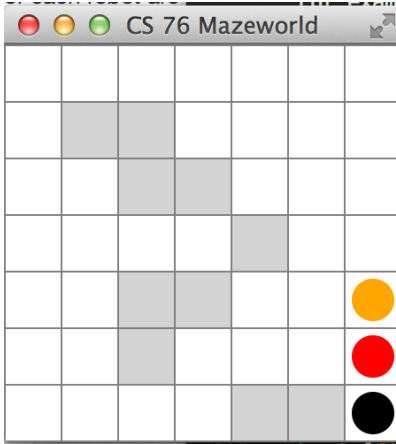


Figure 6: Goal State of 3 robots in 7*7 maze

In this maze, goal coordination of robot A represented by red circle is (6, 1); goal coordination of robot B represented by orange circle is (6, 2); goal coordination of robot C represented by black circle is (6, 0).

There are some rules. The robots only move in four directions, north, south, east, and west. The robots move one at a time. First robot A moves, then robot B, then robot C, then D, E, and eventually, A gets another turn. Any robot may decide to give up its turn and not move. So there are five possible actions from any state. Only one robot may occupy one square at a time. If more than one robots are in a square at a time, they will be crashed. The map is given ahead of time, and it will not change during the course of the game.

4.1 Implementation

The model of multi-robot maze problem is implemented in `MultiRobotsMazeProblem.java`.

For the problem, there is a class named `MultiRobotsMazeProblem`, which extends from `InformedSearchProblem`. In this class, I record the number of robots, initial and goal coordinations of robots, maze and `startNode` inherited from `InformedSearchProblem`.

The `searchNode` is defined as class `MultiRobotsMazeNode`. In this node, there are:

state is represented by a two dimensional int array, `int[number of robots][2]`: an array of coordinations which is represented by two numbers, `int[2]`.

cost is the length from start state to current state, that is depth from `startNode` to this node.

parent points to the node which leads to this node. **parent** is for backchain to get the path.

currentTurn represents who will do the next action, such as robot 0,1,..., number of robots -1.

Here is code of `MultiRobotsMazeProblem.java`:

```
package assignment_mazeworld;

import java.util.ArrayList;
import java.util.Arrays;

public class MultiRobotsMazeProblem extends InformedSearchProblem {
    // 5 actions NOTMOVE:{0,0}
    private static int actions [][] = {{0,0}, Maze.NORTH, Maze.EAST, Maze.SOUTH, Maze.WEST};

    private int numofRobots; //number of Robots

    private int [][] startState; //initial state
    private int [][] goalState; //goal state

    private Maze maze;

    public MultiRobotsMazeProblem(Maze m, int [][] ss, int [][] gs, int num) {
        numofRobots = num;
        startNode = new MultiRobotsMazeNode(ss, 0, null, 0);
        startState = ss;
        goalState = gs;

        maze = m;
    }

    // node class used by searches. Searches themselves are implemented
    // in SearchProblem and InformedSearchProblem.
    public class MultiRobotsMazeNode implements SearchNode {

        // coordinates of robots in the maze
        protected int [][] state;

        // how far the current node is from the start. =depth
        // in multirobots nodes with same state dif cost are dif, due to NOTMOVE action
        private double cost;
    }
}
```



```

// for backchain
private SearchNode parent;

// who's turn [0:numofRobots-1]
private int currentTurn;

public MultiRobotsMazeNode(int [][] s, double c, SearchNode pa, int turn) {
    state = new int[numofRobots][2];
    for (int i = 0; i < numofRobots; i++)
        System.arraycopy(s[i], 0, state[i], 0, 2);

    cost = c;
    parent = pa;
    currentTurn = turn;
}

public int[] getaState(int num) {
    return state[num];
}

public ArrayList<SearchNode> getSuccessors() {

    ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
    int nextTurn = (currentTurn + 1) % numofRobots;

    for (int[] action: actions) {
        int xNew = state[currentTurn][0] + action[0];
        int yNew = state[currentTurn][1] + action[1];

        //System.out.println("testing successor " + xNew + " " + yNew);

        //if not knock a wall or edge and not crash another robot
        if(maze.isLegal(xNew, yNew) && isSafeState(xNew, yNew, currentTurn)) {
            //System.out.println("legal successor found " + " " + xNew + " " + yNew);
            SearchNode succ = new MultiRobotsMazeNode(this.state, getCost()
                + 1.0, this, nextTurn);
            ((MultiRobotsMazeNode)succ).state[currentTurn][0] = xNew;
            ((MultiRobotsMazeNode)succ).state[currentTurn][1] = yNew;
            successors.add(succ);
        }
    }

    return successors;
}

// if crash another robot return false; not return true
private boolean isSafeState(int x, int y, int turn) {
    for (int i = 0; i < numofRobots; i++)
        if (i != turn && Arrays.equals(state[i], new int[]{x, y}))

```

```

        return false;
    return true;
}

// every robot achieves its goal
@Override
public boolean goalTest() {
    for (int i = 0; i < numofRobots; i++)
        if (!Arrays.equals(state[i], goalState[i]))
            return false;
    return true;
}

// an equality test is required so that visited sets in searches
// can check for containment of states
@Override
public boolean equals(Object other) {
    for (int i = 0; i < numofRobots; i++)
        if (!Arrays.equals(state[i], ((MultiRobotsMazeNode) other).state[i]))
            return false;
    if (currentTurn != ((MultiRobotsMazeNode) other).currentTurn)
        return false;
    return true;
}

@Override
public int hashCode() {
    int code = 0;

    for (int i = 0; i < numofRobots; i++)
        code = code * 10000 + (state[i][0] * 100 + state[i][1]);

    // because of action:NotMove, nodes may have same state dif turn, nodes still dif.
    // code = code * 10 + currentTurn;
    return code;
}

@Override
public String toString() {
    String str = new String("Maze_state_");
    for (int i = 0; i < numofRobots; i++)
        str += state[i][0] + "," + state[i][1] + "_";
    str += "depth_" + getCost() + "_priority_" + priority() + "_";
    return str;
}

@Override
public double getCost() {
    return cost;
}

```

```

@Override
public double heuristic() {
    //sum of manhattan distances of robots
    double d = 0;
    for (int i = 0; i < numofRobots; i++) {
        // manhattan distance metric for simple maze with one agent:
        double dx = goalState[i][0] - state[i][0];
        double dy = goalState[i][1] - state[i][1];
        d += Math.abs(dx) + Math.abs(dy);
    }
    return d;
}

@Override
public int compareTo(SearchNode o) {
    return (int) Math.signum(priority() - o.priority());
}

@Override
public double priority() {
    return heuristic() + getCost();
}

@Override
public SearchNode getParent() {
    return parent;
}
}
}

```

The basic idea of **getSuccessors** is to use a for loop to compute the next coordination of the robot which should act in this step. The coordination needs to be tested whether it is a wall or edge by **maze.isLegal** and whether the robot will crash into another robot by **isSafeState**. If the next coordination of the robot is legal and safe, add the coordination and the coordinations of other robots which does not move as a state of a new node to successors.

isSafeState is to find whether the robot which acts this step will crash into another robot.

goalTest needs to make sure each robot achieves its goal.

equals which is used in HashMap needs to check every coordination of each robot is equal. Because the state is an array of many coordinations, a two dimensional array of int, I have debugged much time for this part. I have made a mistake to regard the array as a one dimensional array to use a **Arrays.equal**. Actually, I need to use many **Arrays.equal** to check every coordination. Besides, nodes with same state but different **currentTurn** are still not equal. Because of the **NOTMOVE** action, nodes may have same state but different **currentTurn**.

HashCode which is also used in HashMap is to compute the hashcode for a key. I used a simple 100 based system for every number. For example, if there are 3 robots, the coordinations are $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, the hashcode is $x_1 * 1000000000000 + y_1 * 10000000000 + x_2 * 100000000 + y_2 * 1000000 + x_3 * 10000 + y_3 * 100 + currentTurn$. But the integer is range from -2 147 483 648 to 2 147 483 647. So the hashcode may exceed the max integer to a negative integer, even exceed 0xffffffff. So different states may get a same hashcode, which means the hashmap gets a crash.

`heuristic()` here is different from that in `SimpleMazeNode` because there are more than one coordinations. I add all manhattan distances of every coordinations of robots together to get the h.

4.2 Test

4.2.1 7 * 7 maze

A robot in 7*7 maze is shown in section 2.2.1. The black circle goes according to A-star.

There is a 7*7 maze and 3 robots test. The initial state is like Figure 5

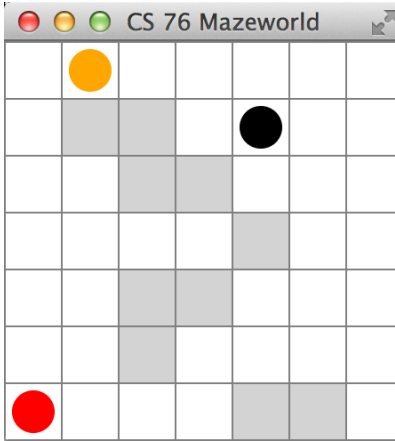


Figure 7: 7*7 maze Initial State

The goal state is like Figure 8:

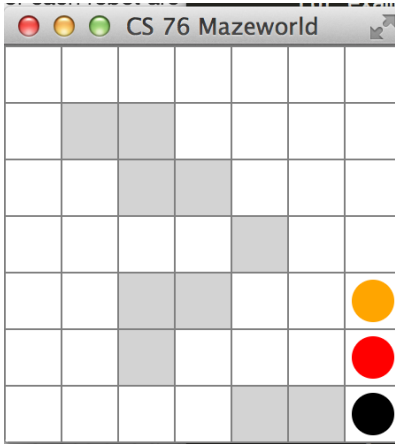


Figure 8: 7*7 maze Goal State

The experimental output is in Figure 9:

There is a very interesting state in the path. The red circle wants to move east to reach the goal but black one is there. So the red one is blocked and give up its action to wait for the black one. Actually, the red one can move north, west as well. But giving up an action is really a good choice to get a shortest path for red circle. Moreover, the orange one still has 3 more steps to reach the goal. If red one chooses move north or west, it still can reach the goal before the orange one reaches the goal. The state is in Figure 10:

A*:
 Nodes explored during search: 1098
 Maximum space usage during search 3595

Figure 9: 7*7 maze experimental results

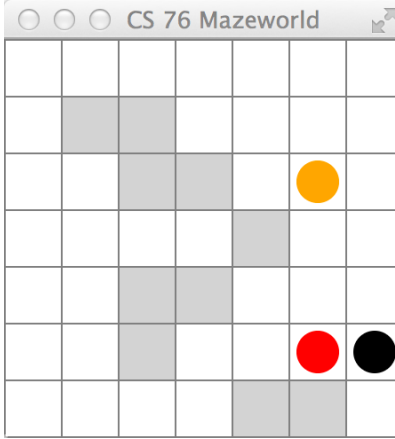


Figure 10: 7*7 maze red circle blocked and waiting for black circle

4.2.2 9 * 9 maze

A robot in 9*9 maze is shown in section 2.2.2. The black circle goes according to A-star.

There is a 9*9 maze and 3 robots test. The initial state is like Figure 11.

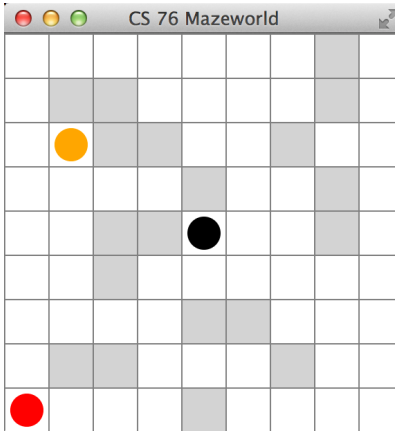


Figure 11: 9*9 maze Initial State

The goal state is like Figure 12:

The experimental output is in Figure 13:

There are some very interesting locations: (3,1) and (3,2), where the robot needs to choose to go north way or south way. Both of two ways can reach the goal. In Figure 14, the red one through (3,1) chooses the south way to the goal (8,1). Then the orange one makes the choice to go the north way to the goal (8, 2) in Figure 15.

Then I exchanged the goals of the red one and orange one. The paths were changed too. Both of orange

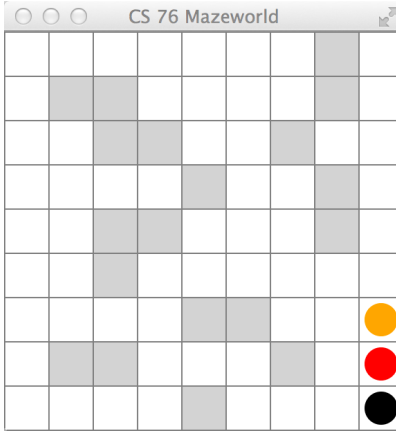


Figure 12: 9*9 maze Goal State

A*:
 Nodes explored during search: 20775
 Maximum space usage during search 41644

Figure 13: 9*9 maze experimental results

one and red one choose the north way! When they reached (3,2) in Figure 16, they all choose to go north, in Figure 17. Another change is that the first action of the red circle is to go north(in Figure 18) not east(in Figure 19).

The experimental output changes too much by exchanging the goal of red and orange! See the output now in Figure 20. Just change red one's goal one move and orange one's goal one move. The nodes explored are twice of before! It's amazing!

4.2.3 20 * 20 maze

A robot in 20*20 maze is shown in section 2.2.2. The black circle goes according to A-star.

There is a 20 * 20 maze and 3 robots. The initial state is like Figure 21:

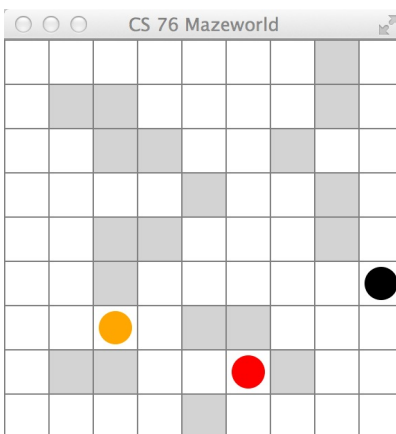


Figure 14: 9*9 maze red one chooses to go south

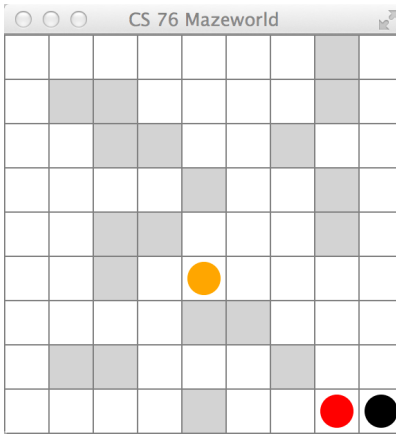


Figure 15: 9*9 maze orange one chooses to go north

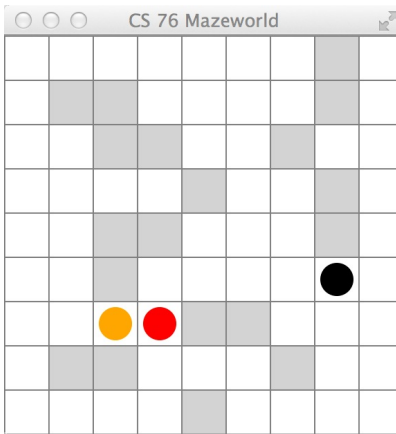


Figure 16: 9*9 maze red one needs to choose a way

The goal state is like Figure 22:

The experimental output is in Figure 23:

The 20*20 maze and 3 robots has made 542531 nodes explored. Moreover, it consumes me no less than half an hour to get the result! The paths of 3 robots are very long. I won't show these here. According to the results of A-star, BFS and DFS must make more nodes explored and consume more time to run.

4.3 Discuss Questions

1. If there are k robots, how would you represent the state of the system? Hint – how many numbers are needed to exactly reconstruct the locations of all the robots, if we somehow forgot where all of the robots were? Further hint. Do you need to know anything else to determine exactly what actions are available from this state?

I need to represent all locations of robots, two number for a location as a coordination. For k robots, it is $k * 2$. Besides, *currentTurn* recording who's turn in this state is also required. It is only one number. So actually, it needs $k * 2 + 1$.

2. Give an upper bound on the number of states in the system, in terms of n and k .

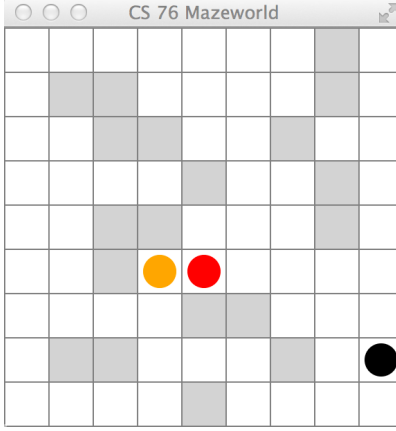


Figure 17: 9*9 maze both choose to go north

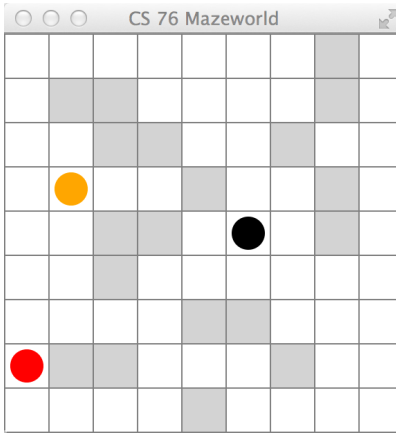


Figure 18: 9*9 maze red one choose to go north now

$$\prod_{i=0}^{k-1} (n * n - i)$$

3. 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 .

The number of states in last question has excluded the collision between robots. But considering walls, it is

$$(n * n - w)^k - \prod_{i=0}^{k-1} (n * n - w - i)$$

Collision of walls of state in last question:

$$\prod_{i=0}^{k-1} (n * n - i) - \prod_{i=0}^{k-1} (n * n - w - i)$$

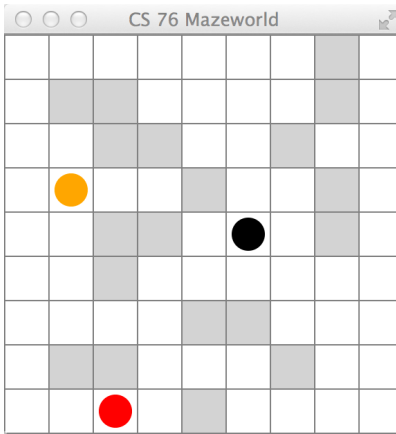


Figure 19: 9*9 maze red one previous choice

A*:
 Nodes explored during search: 42658
 Maximum space usage during search 79437

Figure 20: 9*9 maze experimental results

4. If there are not many walls, n is large (say 100×100), and several robots (say 10), do you expect a straightforward breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not?

I don't expect a straightforward breadth-first search on the state space to be computationally feasible for all start and goal pairs. According to the Question 2 and few walls in maze, the upper bound on states in state space is nearly $(k = 10, n = 100)$

$$\prod_{i=0}^9 (100 * 100 - i)$$

The result is nearly 10^{40} . The state space is too large to compute. So it is not feasible.

5. Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic. See the textbook for a formal definition of monotonic.

In the textbook, the definition of monotonic is that: A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

$$h(n) \leq c(n, a, n') + h(n')$$

The Manhattan distance obviously satisfies the requirement due to it is the length of possible shortest path. Whatever the action a is chosen, $c(n, a, n') + h(n')$ is not less than the length of shortest path, which is $h(n)$.

Moreover, each robot acts on turn, one finishes its action then another can do. According to robots can give up their actions, the number of all actions cannot be less than the sum of numbers of robots' actions. So in my heuristic function, it satisfies the requirement. It is monotonic.

6. 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 5×5 to 40×40 . (You might want to randomly

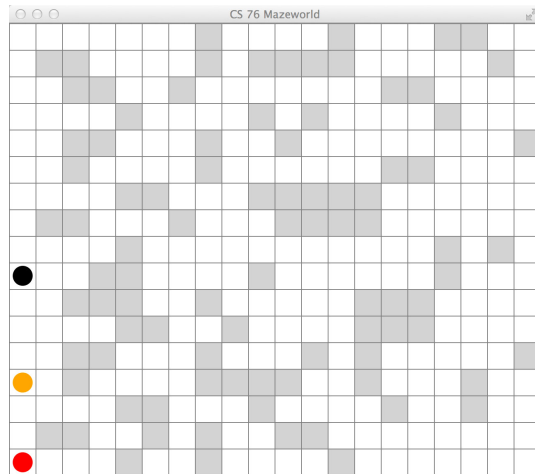


Figure 21: 20*20 maze Initial State

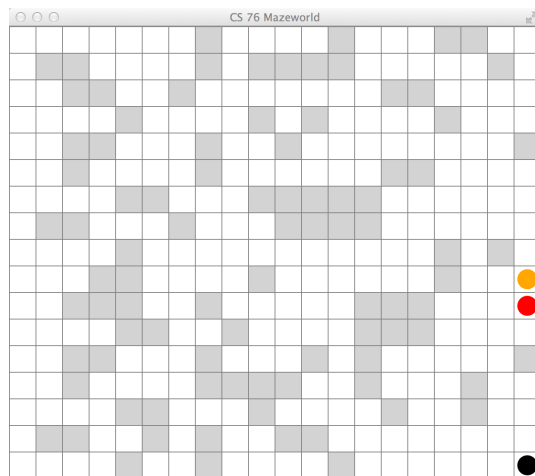


Figure 22: 20*20 maze Goal State

*generate the 40*40 mazes.) I'll leave it up to you to devise some cool examples – but give me at least five and describe why they are interesting. (For example, what if the robots were in some sort of corridor, in the wrong order, and had to do something tricky to reverse their order?)*

The implementation is in Section 3.1 and test is in Section 3.2 .

7. *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?*

8-puzzle may not have any solution. Although A-star is complete, but in some cases, 8-puzzle doesn't have a solution, which is why it is a special case. Some states cannot be reached by all possible states, which means that all possible states in state space are not in a disjoint set. For example, the initial state is :

1	2	3
4	5	6
8	7	0

```

A*:
Nodes explored during search: 542531
Maximum space usage during search 938580

```

Figure 23: 20*20 maze experimental results

0 means blank in the puzzle. The goal state is:

```

1 2 3
4 5 6
7 8 0

```

Yes. My heuristic function is a good one for the 8-puzzle. In the textbook, page 103, the h_2 is exactly like my heuristic function.

8. *The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this. (You do not have to implement this.)*

First, use an array of int to represent the state, of which the length is 9. Use the goal state I have mentioned above as the startNode. So the state of the startNode is {1,2,3,4,5,6,7,8,0}. Change the `goalTest` to `if explored.size() == 9*8*7*6*5*4*3*2 return true; else return false;`. If the program cannot find a path to achieve the goal, all states in state space cannot be linked together in a graph. There are two disjoint set in the state space of 8-puzzle. If you want to make sure whether the left states are in a disjoint set, choose {1,2,3,4,5,6,8,7,0} as startNode to see whether the sum of size of previous explored and this explored is 9.

5 Blind robot with Pacman physics

There is a blind robot in a maze, which doesn't know where it starts. But the robot does know the map of the maze and has a sensor of direction. What we need to do is to plan a path which can lead robot to reach the goal, no matter where it starts.

5.1 Implementation

The key point of the model is to use state to store all possible locations in maze. When the robot does an action, the possible locations will be cut down. Finally, there is only one possible location, the goal location. The path is what we need. Blind robot only needs to follow the path to reach the goal.

For the state, there are many types I can choose. For example, HashMap or HashSet, which is not easy to go through all elements, but it is convenient to know whether an element is in it; LinkedList, ArrayList or Arrays, which is easy for a loop, but we need a loop to know whether an element is in it; Boolean array, BitSet or BitMap, which is not easy to go through all possible locations, but it is convenient to know whether an element is in it, moreover, it consumes much unused space. To make a tradeoff, I choose ArrayList as the type of state.

For locations in the state, I create a new class named `Location` to store the coordination. I can use `SimpleMazeNode` instead of this. But `SimpleMazeNode` has many unused elements and functions. A new class can be clearer.

Here is my code in `BlindRobotMazeProblem.java`.

```

package assignment_mazeworld;

import java.util.ArrayList;

```

```

import java.util.Arrays;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;

import assignment_mazeworld.BlindRobotMazeProblem.Location;

public class BlindRobotMazeProblem extends InformedSearchProblem {

    private static int actions[][] = {Maze.NORTH, Maze.EAST, Maze.SOUTH, Maze.WEST};

    private int xGoal, yGoal;

    private Maze maze;

    public BlindRobotMazeProblem(Maze m, int gx, int gy) {
        // System.out.println("Blind Robot begin!");
        maze = m;

        startNode = new BlindRobotMazeNode(0, null);
        // System.out.println("Blind Robot begin!" + maze.width + maze.height);

        for (int i = 0; i < maze.width; i++)
            for (int j = 0; j < maze.height; j++) {
                // System.out.println(i + " " + j);
                if (maze.isLegal(i, j))
                    ((BlindRobotMazeNode)startNode).addLocation(i, j, null);
            }
        // System.out.println("Blind Robot begin!");

        xGoal = gx;
        yGoal = gy;
    }

    public List<List<Location>> findPaths(List<SearchNode> astarPath) {
        // System.out.println("findPaths begin" + astarPath.size());
        // get all trying paths
        List<List<Location>> paths = new LinkedList<List<Location>>();

        // reverse searching astarPath
        for (int i = astarPath.size(); i > 0; i--) {
            // current search node
            SearchNode node = astarPath.get(i-1);

            // all possible locations in this node
            ArrayList<Location> locs = ((BlindRobotMazeNode)node).state;

            // delete locations in the paths which we have found
            for (List<Location> path : paths) {

```

```

        Location loc = path.get(0); //locations in next state
        loc = loc.getPrev(); //locations in this state
        ((LinkedList)path).push(loc); //add into path
        locs.remove(loc); //delete from locations in this state
    }

    // locations left which will disappear in next state
    for (Location loc: locs) {
        LinkedList<Location> newPath = new LinkedList<Location>();
//        System.out.println(loc);
        newPath.add(loc);
        paths.add(newPath);
    }
//    SearchNode node = path.get(path.size() - 1);

/*    for (Location loc: ((BlindRobotMazeNode)node).state) {
        Location locc = loc;
        System.out.println(loc);

        LinkedList<Location> p = new LinkedList<Location>();
        while (locc.getPrev() != null) {
            p.push(locc);
            locc = locc.getPrev();
            System.out.println(locc);
        }
        paths.add(p);
    }*/
}
for (Location loc: paths.get(1))
    System.out.println(loc);
return paths;
}

// node class used by searches. Searches themselves are implemented
// in SearchProblem.
public class BlindRobotMazeNode implements SearchNode {

    // possible locations in the maze
    protected ArrayList<Location> state;

    // how far the current node is from the start.
    private double cost;

    // for backchain
    private SearchNode parent;

    public BlindRobotMazeNode(double c, SearchNode pa) {

```

```

state = new ArrayList<Location>();

cost = c;

parent = pa;

action = null;
}

public boolean addLocation(int x, int y, Location prev) {
    for (Location loc: state) {
        if (loc.getX() == x && loc.getY() == y) {
            return false;
        }
    }
    Location loc = new Location(x, y, prev);
    return state.add(loc);
}

public ArrayList<SearchNode> getSuccessors() {

    ArrayList<SearchNode> successors = new ArrayList<SearchNode>();

    // according to currentLocation, robot knows the blocked directions
    // if blocked, new successor not move, only keep loc will be blocked too

    for (int[] action: actions) {
        SearchNode succ = new BlindRobotMazeNode(getCost() + 1.0, this);
        for (Location loc: state) {
            int xNew = loc.getX() + action[0];
            int yNew = loc.getY() + action[1];

            //if legal add new location
            if (maze.isLegal(xNew, yNew)) {
                ((BlindRobotMazeNode)succ).addLocation(xNew, yNew, loc);
            }
            //if not move add current location
            else {
                ((BlindRobotMazeNode)succ).addLocation(loc.getX(), loc.getY(), loc);
            }
        }

        if (((BlindRobotMazeNode)succ).state.size() != 0)
            successors.add(succ);
    }
    return successors;
}

```

```

@Override
public boolean goalTest() {
    if (state.size() == 1 && state.get(0).getX() == xGoal && state.get(0).getY() == yGoal)
        return true;
    return false;
}

// an equality test is required so that visited sets in searches
// can check for containment of states
@Override
public boolean equals(Object other) {
    ArrayList<Location> state2 = ((BlindRobotMazeNode) other).state;
    if (state.size() == state2.size()) {
        boolean flag = false;
        for (Location loc: state) {
            flag = false;
            for (Location loc2: state2)
                if (loc.getX() == loc2.getX())
                    flag = true;
            if (flag == false)
                return false;
        }
        return true;
    }
    return false;
}

@Override
public int hashCode() {
    int hash = 0;
    for (Location loc: state)
        hash = hash * 100 + loc.hashCode();
    return hash;
}

@Override
public String toString() {
    String str = new String("Maze_state_");
    for (Location loc: state)
        str += loc;
    str += "_cost_" + cost + "_prior_" + priority() + '\n';
    return str;
}

@Override
public double getCost() {
    return cost;
}

```

```

@Override
public double heuristic() {
    // h1: tell size, but when size = 1 manhattan
    // prior (size = 1) > prior (size >1) slow!
    /*      double h = state.size();
        double dx = xGoal - state.get(0).getX();
        double dy = yGoal - state.get(0).getY();
        if (h == 1) {
            return (Math.abs(dx)+Math.abs(dy));
        }
        else {
            return h/(maze.width * maze.height);
        }
        // h2: h = longest distance to goal    cannot tell size
        double h = state.get(0).distance();
        for (Location loc: state) {
            double d = loc.distance();
            if (d > h)
                h = d;
        }
        return h;
    */
    //h3: h = sum of manhattan distance
    double h = 0;
    for (Location loc: state) {
        double d = loc.distance();
        h += d;
    }
    return h;
}

@Override
public int compareTo(SearchNode o) {
    return (int) Math.signum(priority() - o.priority());
}

@Override
public double priority() {
    return heuristic() + getCost();
}

@Override
public SearchNode getParent() {
    return parent;
}
}

// Now state in BlindRobotMazeNode needs to store many locations,
// So this class represents location (x, y)

```



```

public class Location extends Object {
    private int x;
    private int y;

    private Location prev;

    public Location(int m, int n, Location pa) {
        x = m;
        y = n;
        prev = pa;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public Location getPrev() {
        return prev;
    }

    public double distance() {
        double dx = xGoal - x;
        double dy = yGoal - y;
        return Math.abs(dx) + Math.abs(dy);
    }

    @Override
    public int hashCode() {
        return x*10 + y;
    }

    @Override
    public String toString() {
        return new String("Loc_" + x + "," + y + "_");
    }
}

```

The basic idea of `getSuccessors` is to use a for loop to compute the next coordination of all possible locations. The coordination needs to be tested whether it is a wall or edge by `maze.isLegal`. If the next coordination of the possible location is legal, add the next coordination to the successor like the robot does an action; else, add the current coordination like the robot cannot move. Successors are possible coordinations of do all four actions.

`goalTest` needs to make sure there is only one location in the state and this location is the goal.

`equals` which checks whether the two nodes have all same possible locations in state.

`HashCode` which is just like the heuristic function in `MultiRobotsMazeProblem`. So different states may get a same hashcode, which means the hashmap gets a crash.

`heuristic()` I considered 3 heuristic functions.

The first one is that h is the size of possible states / all possible states when the size is more than one; h is the manhattan distance when the size is one. It is monotonic but the priority of node($size = 1$) is greater than that($size > 1$). That means all nodes with size greater than 1 need to be explored.

The second one is that h is the largest distance of all manhattan distance from possible locations to goal. But it cannot tell the different between size of possible locations.

The third one is that h is the sum of manhattan distances of all possible locations.

The forth one is that $h = 0$ when $size > 1$; when $size = 1$, $h = \text{manhattan distance}$. That means when $size > 1$, its a BFS.

finally I chose the third. But wether which one I choose, it runs very slowly.

5.2 Test

5.2.1 5*5 maze

In the initial state, all locations except walls are possible locations in state, in Figure 24. The sequences are below:

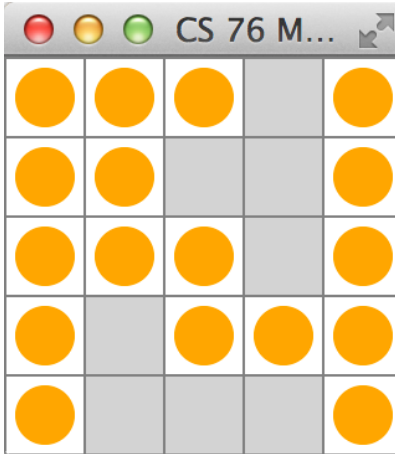


Figure 24: 5*5 maze red one previous choice

Then go north(up) in Figure 25

Then go west(left) in Figure 26

Then go up(north) in Figure 27

Then west, east, south, south, east, south, east, east, south, south.

The experimental ouput of h_2 is like that in Figure 28

The experimental ouput of h_3 is like that in Figure 29

For the 7*7 maze in section 3, the experimental ouput of h_3 is like that in Figure 30. The goal is (6, 0).

6 Polynomial-time blind robot planning

1. *Prove that a sensorless plan of the type you found in problem 3 always exists, as long as the size of the maze is finite and the goal is in the same connected component of the maze as the start.*

In the sentence, the goal is in the same connected component of the maze as the start. So all locations in this connected component as the start can reach the goal. Assume all locations as vertices in a graph. If any action can lead a vertex to another vertex, the two vertices are adjacent in the graph. So all possible locations in the same connected component are in a connected graph. According to graph

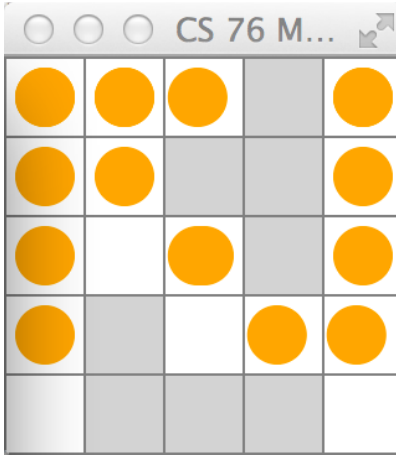


Figure 25: 5*5 maze red one previous choice

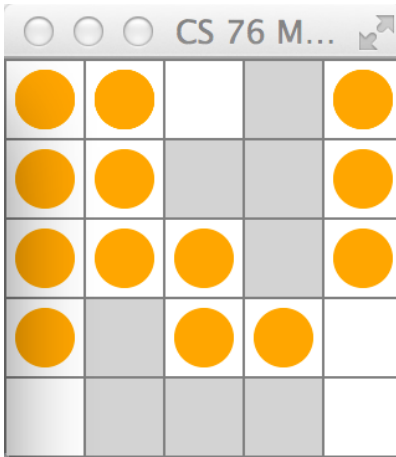


Figure 26: 5*5 maze red one previous choice

theory, obviously, every vertex can connect to any other vertices in the graph. The path we found in problem 3 is to help the blind robot reach the goal wherever it starts. In a conclusion, the plan in problem 3 always exists.

2. *Now describe (but do not implement) a motion planner that runs in time that is linear or polynomial in the number of cells in the maze. (Notice that the size of the belief space is exponential in the number of cells, so this would be a very interesting result!)*

The key of creating a motion planner that runs in time that is linear or polynomial in the number of cells in the maze is to find a good heuristic function for A-star search algorithm. Assumint n as the number of cells, the time complexity of the planner can be $O(n^2)$ with a heuristic function like that:

Try to decrease the number of possible locations (number of states in a belief state) first, if the number of possible locations is greater than one. A robot have 4 actions on any location. So a robot have at most $4 * n$ actions in a maze. The worst case is to get the step can decrease the possible locations after trying every actions, $4 * n$. So the number of steps before decreasing the number of possible locations to 1 is $4n^2$. Next, when the number of possible locations is one, use manhattan distance as heuristic function. The worst case is to reach the goal after going through every cells in the maze, n . Finally,

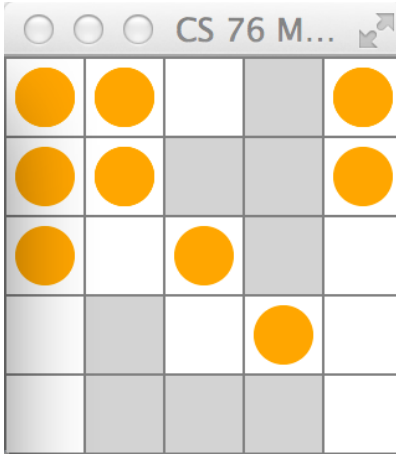


Figure 27: 5*5 maze red one previous choice

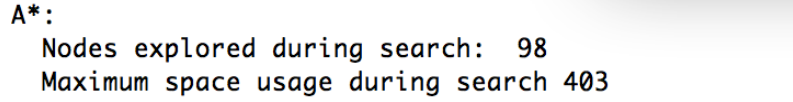
A*:
 Nodes explored during search: 699
 Maximum space usage during search 2802

Figure 28: 5*5 maze experimental results of h2

the sum of the two cases, the time complexity is $O(n^2)$.

A*:
 Nodes explored during search: 238
 Maximum space usage during search 607

Figure 29: 5*5 maze experimental results of h2



A*:
Nodes explored during search: 98
Maximum space usage during search 403

Figure 30: 5*5 maze experimental results of h2

References

- [1] Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In The Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI10), 173178.
- [2] Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011. The increasing cost tree search for optimal multi-agent pathnding. In The International Joint Conference on Artificial Intelligence (IJCAI11), 662667.