Mazeworld

Jing Ge

January 20, 2016

1 Introduction

This assignment has four main parts. The first part shows the implementation of A* search and tested it by a single robot in a simple maze. The second part discusses multi-robot coordination and the third part has a blind robot that knows the map of maze but doesn't know its location and doesn't have any sensor, both of them use A* search. The last part introduces paper related to multi-robot problem.

2 Implementation of A* search

A* search is similar with breadthfirst search, but it is an informed search algorithm that uses heuristics to guide its search path. Every node will be given a priority in this algorithm. A* search need a ordered data structure to determine which node will be visited in next step. Thus we use a priority queue (which is implemented by heap) to sort and track all nodes. Node with the highest priority (smallest estimate distance to the goal) will be popped and visited first. If the node has been visited and its cost is more than the previous one recorded in `visitied`, we simply discard it.

There are two methods of recording the path from start to the goal. We can add a parent node in SimpleMazeNode and add two functions setParentNode() and getParentNode() in SearchNode interface, then implement these two functions in SimpleMazeNode. Every time we add a new successor node in priorityqueue and hashmap, set its parent node as current node. Then write a newbackchain(), find the search path by using parent node. Or we can use another hashmap to record and update node's parent node. I implemented both and used first one in my code.

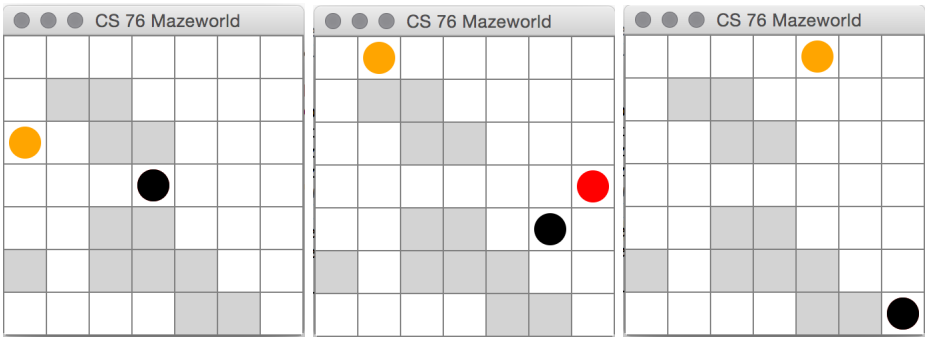
A* search is implemented in `InformedSearchProblem.java`. Here's my code:

```
public List<SearchNode> astarSearch() {
     // insert node based on their priority
     PriorityQueue<SearchNode> fringe = new PriorityQueue<SearchNode>();
     // record nodes we have visited and their priority
     HashMap<SearchNode, Double> visited = new HashMap<SearchNode, Double>();
     fringe.add(startNode);
     visited.put(startNode, null);
     while (!fringe.isEmpty()){
        // pull the highest priority node
        SearchNode current = fringe.remove();
        // if current node's priority larger than the previous, discard
        if(current.priority() > visited.get(current))
           continue;
        // return path if reach the goal
        if (current.goalTest())
           return newbackchain(current);
```

```
        // search next states
        else{
           List<SearchNode> successorsList = current.getSuccessors();
           for (SearchNode node : successorsList) {
              // if we have not visited this node
              if (!visited.containsKey(node)) {
                // set this node's parent
                 node.setParent(current);
                 fringe.add(node);
                 visited.put(node, node.priority());
              }
           }
        }
     }
     return null;
  }
```

The test code for A* is in `SimpleMazeDriver` and the result for this problem is:

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



As we can see, A* search is better than both bfs and dfs.

3 Multi-robot coordination

I will answer those questions and introduce my implementation in Q.6:

1. We would have to know the location (x and y) of every robot and which robot moves this turn. Thus we need $(2 * k + 1)$ variables to record all information we need.

2. Since the size of maze is $n * n$ and number of robot is k, every robot has different location and there's a index indicates which robot moves this turn, the upper bound of states is $\prod_{i=0}^{k-1}(n * n - i) * k$.

3. Since n is much larger than k, we don't need to consider collision of different robots. So we only need to calculate the total number of states that some robots may collide into wall. If there's one robot collide into wall, the number is: $w*k*\prod_{i=0}^{k-2}(n*n-i)*k$. If there are 2, the number is: $(w*(w-1))*(k*(k-1)/2)*\prod_{i=0}^{k-3}(n*n-i)*k...$ So the total number of collision is: $\sum_{i=1}^{k}\{\{k!/(k-i)!\}*\{\prod_{j=1}^{i}(w-j+1)\}*\{\prod_{m=0}^{k-i-1}(n*n-m)*k\}\}$.

4. I don't think bfs is feasible for this problem because the number of nodes we need to record may be too large. The total number of nodes we may record could be $\prod_{i=0}^{9}(10000-i)*10$ and it will definitely crash on the computer.

5. The text book said that heuristic p(n) is monotonic if p(n) never decrease. In this program, we use $p(n) = h(n) + g(n)$ to represent the heuristic, where $h(n)$ is manhattan distance between current node and the goal, $g(n)$ is current cost. It's easy to prove that $g(n)$ will increase more as $h(n)$ decreases. The robot takes at least $h(b) - h(a)$ steps from a to b, while the real cost of this move may be larger since there are walls in the maze and robot will make long detour if it hit a wall. So $g(b) - g(a) \geq h(b) - h(a)$, thus p(n) will never decrease.

6. This model is implemented in `MultiRobotProblem`, the main difference between this problem and the single robot problem is the state representation. We use an int array `state` and a int `index` here to record the current node's state. The `state` array records the current locations of every robot and `index` records which robot will move this turn.

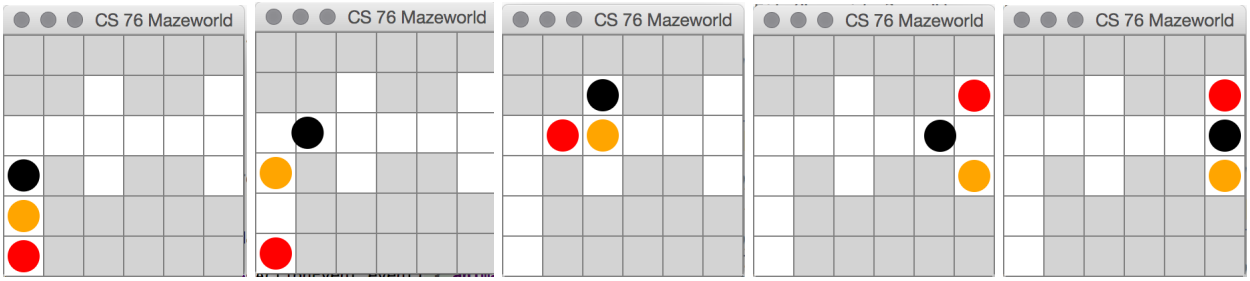
```
int[][] goal;
// location of the agents in the maze
int[][] state;
// which robot will move this round
int index;
public ArrayList<SearchNode> getSuccessors() {
     for (int[] action: actions) {
       int xNew = state[index][0] + action[0];
       int yNew = state[index][1] + action[1];

       // check if robot is on a legal position
       // or two robots are on the same position
       if(maze.isLegal(xNew, yNew) && isSafe(xNew, yNew)) {
         int succState[][] = new int[num][2];
         for(int i = 0; i < num; i++){
           succState[i][0] = state[i][0];
           succState[i][1] = state[i][1];
         }
         succState[index][0] = xNew;
         succState[index][1] = yNew;

         // index may larger than num, set it to 0
         MultiRobotNode succ = new MultiRobotNode(succState, getCost() + 1, (index + 1) % num);

         successors.add(succ);
       }
     }
     return successors;
   }
```

```
// check if two robots are on the same position
private boolean isSafe(int x, int y) {
  for(int i = 0; i < num; i++)
    if(i != this.index)
      if(state[i][0] == x && state[i][1] == y
              return false;
    return true;
  }
```

The test code for this problem is in `MultiRobotDriver` and the test cases for this problem is:

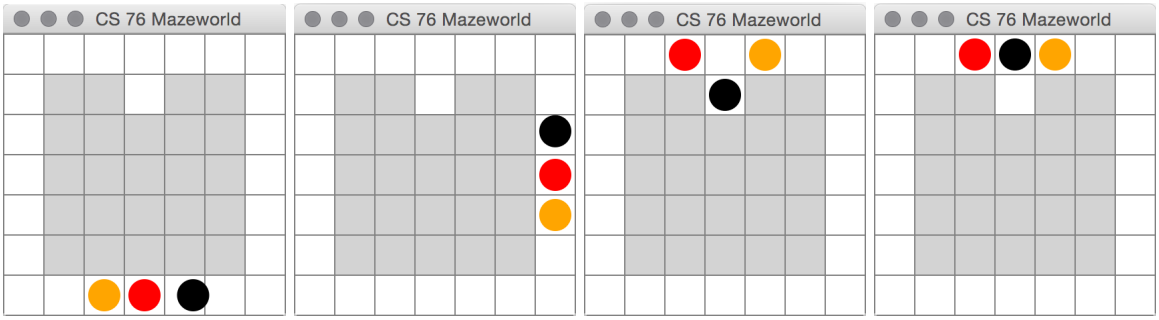
1. Robots need to change their order to reach the end. The map is `MultiTest1.maz`:

```
A*:
  Nodes explored during search: 900
  Maximum space usage during search 1480
```



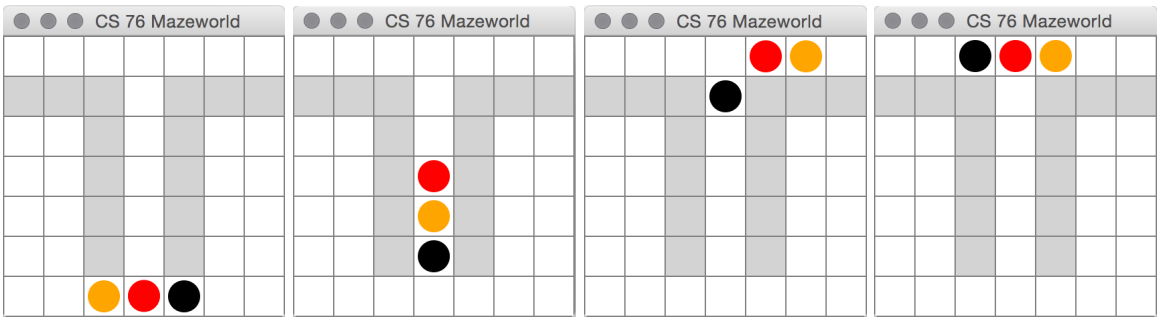
2. Robots also need to change their order, but it's more tricky, they only have one crack to swap their locations. The map is `MultiTest2.maz`:

```
A*:
  Nodes explored during search: 3830
  Maximum space usage during search 5798
```



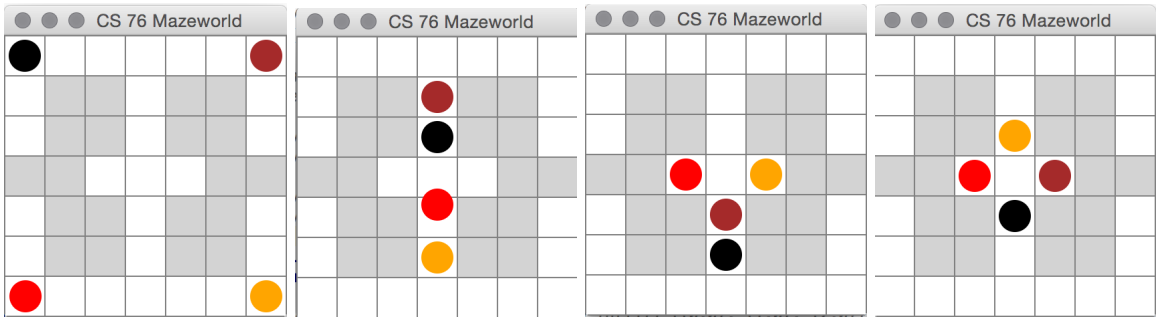
3. When they reach the top line, the first two robots must move to another direction first. There are blanks in left and right, which may mislead the search, and we could see it from the result. The map is `MultiTest3.maz`:

```
A*:
  Nodes explored during search: 18820
  Maximum space usage during search 36011
```



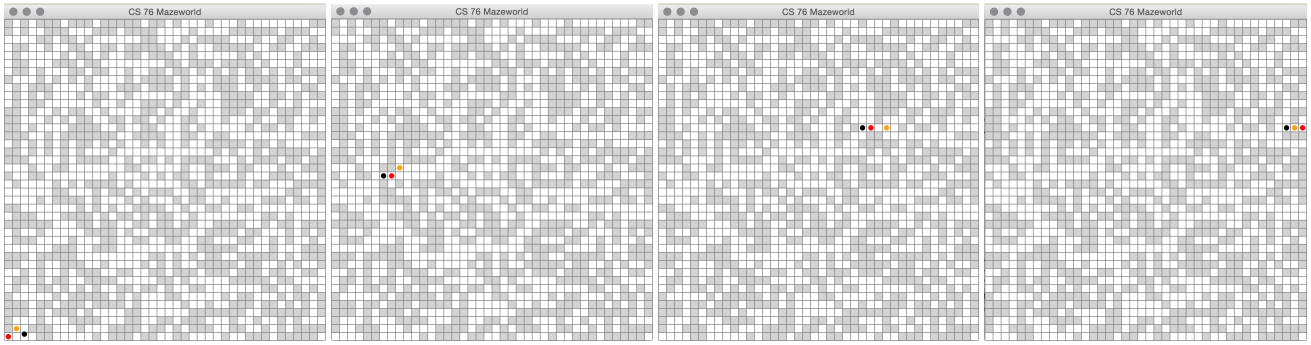
4. This is the most interesting map, the robot must pass over its destination and then back to it. We could see that the A* search uses lots of memory in this case. The map is `MultiTest4.maz`:

```
A*:
  Nodes explored during search: 245805
  Maximum space usage during search 322196
```



5. A very large map and need to record lots of information. The map is `MultiTestLargeMap.maz`:

```
A*:
  Nodes explored during search: 2334523
  Maximum space usage during search 4217786
```



I also tried this algorithm in 8-puzzle but it could only work in several cases.

7. Theoretically, I think the 8 puzzle problem could be solved by multi-robot coordination algorithm since it is 8 robots in a $3 * 3$ maze which only 2 nodes could move. But the heuristic function in this program has no help (the manhattan distance never larger than 4) since the order of these nodes are more important than their distance to the goal.

8. In order to prove this assumption. I would like to generate a states set S which contains all kinds of states (without index now, only locations of 8 nodes). Then I will start from a node a, generate a set A which contains all states that node a could reach. Then I will choose a node b from S but not in A, generate a set B which contains all states that node b could reach. If A and B do not have intersection and $A\&B = S$, then we can prove that.

4 Blind robot with Pacman physics

4.1 Model and implementation

In this problem, we use a set to record all possible locations of the blind robot. First, we search and add all legal locations in the start set. Then we move this robot to four directions. In each direction, move all nodes and check if them are legal, add the legal nodes in the next state set. Repeat it until the set is equals to the goal set. The goal set of this problem is a set contains the final location.

This model is implemented in `BlindRobotProblem`:

```
Set<List<Integer>> state;
goal = new HashSet<List<Integer>>();
public ArrayList<SearchNode> getSuccessors() {
  ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
  // iterate four directions
  for (int[] action: actions) {
    Set<List<Integer>> nextStates = new HashSet<List<Integer>>();
    for (List<Integer> s : state){
      int newX = s.get(0) + action[0];
      int newY = s.get(1) + action[1];
      // check if it is a legal location
      if (maze.isLegal(newX, newY)) {
        // add it to the next state
        List<Integer> possibleState = new ArrayList<Integer>();
        possibleState.add(newX);
        possibleState.add(newY);
        nextStates.add(possibleState);
      }
      else
        nextStates.add(s);
    }
    SearchNode succ = new BlindRobotNode(nextStates, getCost() + 1);
    successors.add(succ);
  }
  return successors;
}
```

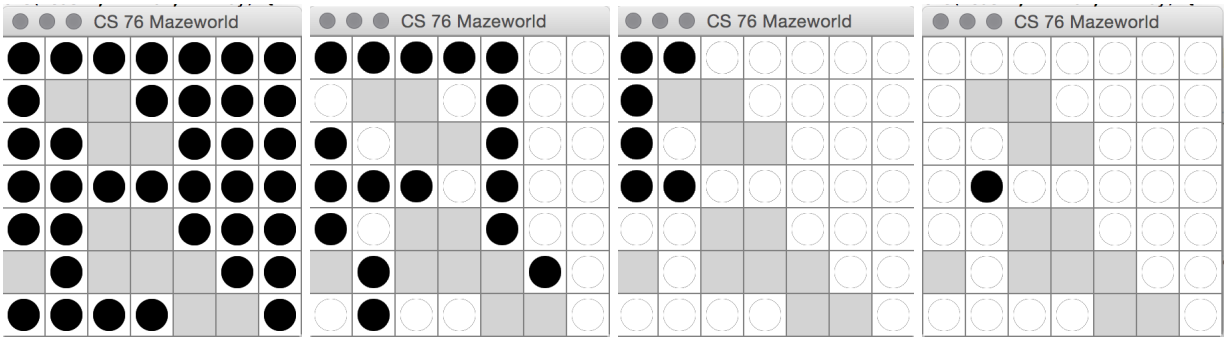
In order to show animation on the screen, I modified `BlindRobotDriver` and `MazeView`. I will overlap the original circle by a white circle (this method is tricky but I don't find any other way) by using remove() function in MazeView.

The test code for this problem is in `BlindRobotDriver` and the test cases for this problem is:

The goal state is $(1, 3)$, the map is `simple.maz`:

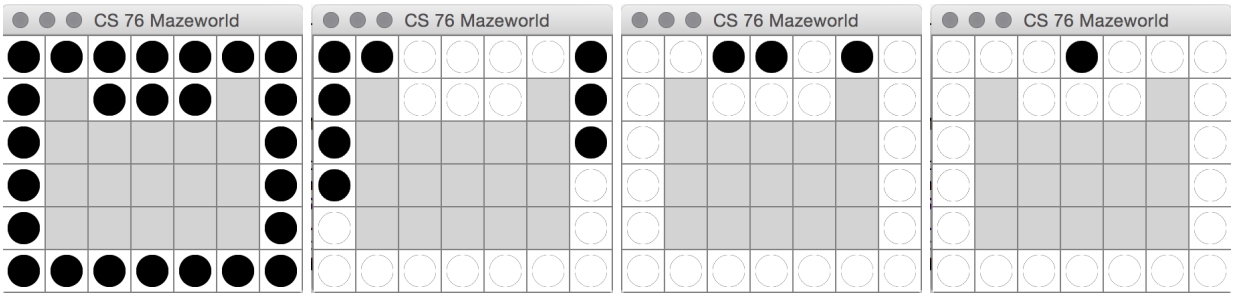
`A*:`

```
  Nodes explored during search: 53
  Maximum space usage during search 276

If we use Euclidean distance:
A*:
  Nodes explored during search: 22473
  Maximum space usage during search 85000
```



The goal state is $(3, 5)$, the map is `BlindTest1.maz`

```
A*:
  Nodes explored during search: 34
  Maximum space usage during search 129

If we use Euclidean distance:
A*:
  Nodes explored during search: 25192
  Maximum space usage during search 86135
```



We can see that manhattan distance is much better than Euclidean distance.

4.2 Polynomial-time blind robot planning

We could prove that there is always a solution since we can move the node to a corner and the map always has edges and walls. If we could confirm a node's location, then we can use SimpleMazeProblem's algorithm to solve it. And since number of the set's elements will continue to decrease, I think we could solve it in polynomial-time.

5 Literature Review

I read Complete Algorithms for Cooperative Pathfinding Problems. This paper is about cooperative pathfinding problems (assignment's multi-robot coordination problem). The author mentioned two ways of solving this problem: global search approaches and decoupled approaches. Global search uses the entire set of agents as a single state and can always find a solution, but cost lot. Decoupled approaches are fast but not complete. In this paper, the authors proposed two techniques to find a complete and optimal solution: Independence Detection (ID) and Operator Decomposition (OD). ID decomposes problem into independent subproblems and OD will work when no independence can be found. This paper coupled OD with ID, proposed an algorithm that ensures the paths for each group are optimal and do not conflict. The algorithm first start by finding a path for each agent and look for collisions in those paths. Then use OD to merge these paths until there are no more conflicts. Then the paper mentioned that the real-time performance of these kinds of algorithms are often not efficient enough. So they extended OD + ID to "offers a tradeoff between solution quality and running time". They embedded constraint in both OD and ID to ensure the solution could be optimal. The paper also discussed optimal anytime algorithms that could stop anytime and return a best solution to where they reach. One of the method to adapt the above algorithm to an anytime algorithm is iterative deepening, it can return the best solution and the lower bound on the cost of the optimal solution.