Mazeworld Solution

Lexin Tang

September 29, 2016

1 Introduction

In this report, I will introduce solutions for three robot path planning problems: single robot, multi-robot and blind robot. Firstly, I will introduce my implementation of representing the mazeworld map. Then I will demonstrate feasible solutions for each problem and compare different search algorithms used in them.

2 Maze Initialization

To represent the mazeworld map, I write a class `Maze` in `Maze.java`, which provides functions such as `readFromFile` and `generateRandomMaze`. We could either read a maze from file or generate a random $m * n$ maze by calling `Maze.generateRandomMaze(outputFileName, m, n)`. Inside the `Maze` class, `boolean[][] cells` is the map we need to check when exploring around the maze. If `cells[i][j]` is false, there is a wall at position (i, j), otherwise, (i, j) is an open floor tile. The coordinates of bottom left corner is $(0, 0)$ and the the bottom right corner is $(map.cols - 1, 0)$, just as defined on the assignment webpage.

I craft 5 examples of maze layout and they are shown below.

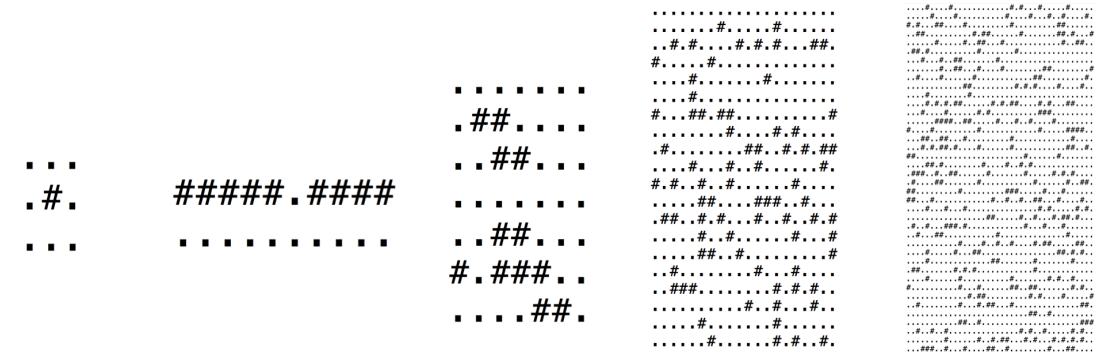

Figure 1: five examples of maze

3 Single Robot Path Planning

3.1 Problem Description

The single robot path planning problem is defined in `SingleRobotProblem.java`. The state of this problem is the location of the robot: (x, y). I used an int array with size of 2 to represent the state of search node.

To enable A* search, each search node must be assigned with cost. I added attributes `heuristic` and `cost` to each search node. `heuristic` is computed based on Manhattan distance. `cost` is the sum of the depth of BFS and heuristic.

$$h(i,j) = |i - goalx| + |j - goaly|$$

In `getSuccessors()`, I searched four directions {West, South, East, North} and check whether the new position is a legal state.

```
int x_max = maze.cols - 1;
int y_max = maze.rows - 1;
int[] dir_x = {-1, 0, 1, 0};
int[] dir_y = {0, -1, 0, 1};

for (int i = 0; i < 4; i++){
int x_new = this.state[0] + dir_x[i];
int y_new = this.state[1] + dir_y[i];

if (x_new >= 0 && x_new <= x_max && y_new >=0 && y_new <= y_max && maze.cells[x_new][y_new])
    {
  reachableStates.add(new MazeNode(x_new, y_new, depth + 1, this));
}
```

I override the `compareTo` interface in class `SingleRobotProblem` so that `PriorityQueue<UUSeartchNode>` in A* search can work correctly.

```
@Override
public int compareTo(MazeNode other){
  return this.cost - other.getCost();
}
```

3.2 Implementation of A-star Search

I do marking by following the guidelines on assignment webpage. If a new state is explored as a child of the current node, check if it is in visited set already. If not, add to frontier. If so, but the new node has a less expensive path cost, still add it to frontier. Otherwise, do not add the node to the frontier. I made use of data structure `PriorityQueue` and `HashMap` in java. The hash value of `visited` stores the cost from start state to the key. To backtrace the previous state of the current node, I added an attribute `parrent` to search node so that `backchain()` can be done in a very simple way. The code is shown as follows.

```
    HashMap<UUSearchNode, Integer> visited = new HashMap<UUSearchNode, Integer>();
    PriorityQueue<UUSearchNode> queue = new PriorityQueue<UUSearchNode>();
    queue.offer(startNode);

    while (!queue.isEmpty()){
      int size = queue.size();
      for (int i = 0; i < size; i++){
        UUSearchNode node = queue.poll();

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

        ArrayList<UUSearchNode> nextNodeList = node.getSuccessors();
        for (int j = 0; j < nextNodeList.size(); j++){
          UUSearchNode nextNode = nextNodeList.get(j);

          if (!visited.containsKey(nextNode) || visited.get(nextNode) > nextNode.getCost()){
            queue.offer(nextNode);
```

```
20                visited.put(nextNode, nextNode.getCost());
21                incrementNodeCount();
22                updateMemory(visited.size() + queue.size());
23              }
24            }
25          }
26        }
```

I ran tests on 5 different mazes and compare the output results derived from different search algorithms. The results are shown in Figure 2.

	BFS			A*			DFS-mem		
	path length	node explored	memory	path length	node explored	memory	path length	node explored	memory
maze1.txt	3	4	7	3	4	7	3	3	3
maze2.txt	10	11	12	10	11	14	10	10	10
maze3.txt	13	36	41	13	24	30	15	17	17
maze4.txt	28	250	276	28	83	96	100	120	120
maze5.txt	48	792	857	48	190	221	282	311	311

Figure 2: comparison of different search algirthms on 5 mazes

Since the Manhattan distance is an optimistic heuristic, A* search is guaranteed to generate optimal path. In terms of run time and space efficiency, A* is better than both BFS, especially for large mazes. And DFS doesn't always generate optimal solutions.

4 Multi-Robot Path Planning

4.1 Discussion

The state of the system can be represented by the location of each robot in the map and the turn if we assume that k robots take actions in roundrobin fashion. Thus, we need $2k + 1$ numbers to represent each state.
The upper bound on the number of states in the system should be $(n * n)^k * k$.
If the number of wall squares is w, and n is much larger than k, the number of the states without collisions $= A^k_{(n-w)^2} = \frac{k(n^2-w)!}{(n^2-w-k)!}$.
The number of the states represent collisions = total states - non-collision states $= (n * n)^k * k - \frac{k(n^2-w)!}{(n^2-w-k)!}$.
If there are not many walls, n is large (say 100x100), and several robots (say 10), I think a straightforwards breadth-first search on the state space is computationally infeasible for most of start and goal pairs. BFS will need a huge queue to explore almost all the states in the maze. For the case of n = 100 and k = 10, the number of total states is $(n * n)^k * k = 10^{41}$, which is too huge to be computed!
A useful, monotonic heuristic function for this search space would be the sum over Manhattan distances of k locations toward their goals.

$$h(state) = \sum_{i=1}^{k} |x_i - goalx_i| + |y_i - goaly_i|$$

This heuristic is optimistic because Manhattan distance for each robot towards its goal is shorter than or equal to its true cost and the sum over Manhattan distance must not be greater than the total move of k

robots.
The 8 puzzle is a special case of the multi-robot problem because the goal of 8 puzzle is also for each queen to move from its starting position to the goal position. The heuristic used in multi-robot problem would still be a good one for 8 puzzle.

4.2 Implementation of Problem Model

The multi-robot path planning problem is defined in `MultiRobotProblem.java`. The state of this problem can be represented by locations of k robot. I used an 2D array to store their locations. Since the k robots plan path in roundrobin fashion, I added an attribute `turn` to search node, specifying it is whose turn to take action. Heuristic used in this problem is the sum over Manhattan distances of k locations.

$$h(state) = \sum_{i=1}^{k} |x_i - goalx_i| + |y_i - goaly_i|$$

In `getSuccessors()`, if it is i's turn, I searched four directions {West, South, East, North} from i's current location and check whether the new position is a legal state. Note that there are multiple robots in the maze, the new location of robot i must not collide with other robots. So I used a separate function to check whether the new state is legal. The successors should also pass the turn to the next robot by adding `turn` by 1. The code is shown below.

```
1     private boolean isSafeState(int x, int y){
2       int x_max = maze.cols - 1;
3       int y_max = maze.rows - 1;
4       if (x >= 0 && x <= x_max && y >=0 && y <= y_max && maze.cells[x][y]){
5         for (int k = 0; k < totalRobots; k++){
6           if (k == turn){
7             continue;
8           }
9           if (x == this.state[k][0] && y == this.state[k][1]){
10            return false;
11          }
12        }
13      }
14      else{
15        return false;
16      }
17
18      return true;
19    }
```

4.3 Results

I ran tests on the 5 mazes provided and set starting positions and goals for each maze. A* search can find optimal solutions for all of them.

1. maze1.txt

```
1 total robots = 2
2 A* search path length:  9
3 Nodes explored during last search:  38
4 Maximum memory usage during last search 58
```

2. maze2.txt

```
total robots = 3
A* search path length:  52
Nodes explored during last search:  2202
Maximum memory usage during last search 2426
```

3. maze3.txt

```
total robots = 3
A* search path length:  32
Nodes explored during last search:  11006
Maximum memory usage during last search 14773
```

4. maze4.txt

```
total robots = 3
A* search path length: 81
Nodes explored during last search:  1092077
Maximum memory usage during last search 1419005
```

5. maze5.txt (may take a few minutes)

```
total robots = 3
A* search path length:  147
Nodes explored during last search:  6702881
Maximum memory usage during last search 8777659
```

5 Blind Robot Problem

5.1 Implementation of Problem Model

The blind robot path planning problem is defined in `BlindRobotProblem.java`. The state of this problem is represented as belief state, which is a HashSet containing all the possible locations the robot might be at. The heuristic used in this problem is defined as the sum over Manhattan distance of each possible location in belief state towards the goal.

$$h(state) = \sum_{pair \in state} |pair.x - goalx| + |pair.y - goaly|$$

Another heuristic could be used in this problem may be the size of belief state. However, it takes much longer to find a solution compared with the sum Manhattan distance heuristic when I was trying it on a 7*7 maze.

In `getSuccessors()`, I searched four actions {West, South, East, North} and generate new belief state from the current state based on maze. Note that I used a self-defined class `Pair` to represent location in the maze, which is quite convenient when checking whether a location is in belief set. The code is shown below.

```
1  private class Pair{
2    public int x;
3    public int y;
4    public Pair(int x, int y){
```

```
5         this.x = x;
6         this.y = y;
7       }
8
9       @Override
10      public boolean equals(Object other) {
11        return (this.x == ((Pair) other).x && this.y == ((Pair) other).y);
12      }
13
14      @Override
15      public int hashCode() {
16        return this.x * 100 + this.y;
17      }
18    }
```

5.2 Results

I ran tests on a 7*7 and a 20*20 maze and set the goal to the bottom right corner. The results are shown below.

1. 7*7 maze

```
A* search path length:  30 [Initial Belief State, East, East, East, South, South, East,
    East, South, South, East, East, South, East, South, East, South, West, West, North
    , North, North, East, East, East, South, East, East, South, South]
Nodes explored during last search:  709
Maximum memory usage during last search 976
```

2. 20*20 maze

```
A* search path length:  103
Nodes explored during last search:  3694
Maximum memory usage during last search 4424
```

6 Previous Work (graduate student)

"Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems"
In this paper, the author presented a efficient algorithm for solving the general cooperative path-finding problem. Firstly, He assigned each agent to a individual group and used A* search to find an optimal path for each agent. If the found paths of two agents conflicts with each other, the algorithm will perform tie breaking and try to find another optimal path. If no other optimal path is found, the algorithm will combine the two groups and then find an optimal path for the merged group. The run time efficiency of the algorithm is improved when the number of agents increases.