Maze World Solution

Nicholas Golini

September 29, 2016

1 Introduction

In this problem, we have a maze of walls, and open spots. The open coordinates are the only places the robots can be in the maze, and the walls represent coordinates in the maze that the robot cannot go. The walls are represented by the '#' character, and open spots are represented by a period. Given n robots, a maze, starting coordinate(s) for the robot(s), and goal coordinate(s), we want to bring all of the robots to their goal coordinates. To represent the system, we will use a state that holds the coordinates of the r robots. For example if there were two robots, the state would be: (x_1, y_1, x_2, y_2), where (x_1, y_1) represents the coordinates of the first robot, and (x_2, y_2) represents the coordinates of the second robot. Given an $n \times n$ maze, there will be n^{r*2} possible states in the maze, since each part of the state can be a number between 0 and $n - 1$.

1.1 General Hierarchy

The way my problem is set up is that I have a MazeProblem class, a MultiRobot class, and a SearchProblem class. The MazeProblem class takes in a char[][] array of # and periods, and converts them into a variable called world, which holds a 1 if a robot can move into the position in the maze, a 0 if it cannot, or the name of a robot in that coordinate (robots named A, B, C, etc.). The SearchProblem class contains the SearchNode subclass, the A^* algorithm, and the driver algorithm that starts with the robots in their starting coordinates, and brings them to their goal coordinates (this implementation will be talked about later). Finally, the MultiRobot class contains the subclass, RobotNode, which extends SearchNode, and represents a possible state of the robots in the maze. This is where the getNeighboringSpaces() function is called, which finds all of the possible locations a robot can go to, as well as functions to get specific information of the states, such as getX/Y(), heuristic(), and updateWorld(), where if a robot moves, the world is updated to represent the moved robot.

2 Implementation of the model

Since the state is represented as an array of integers, where each pairing represents a coordinate of a robot, it is vital to know which robot is being considered for movement. To take this into account, I set up a global variable called robotTurn in the MultiRobot class which starts at 0, being robot A, gets incremented by 1 to be 1, being robot B, and finally incremented to 2 if there is a third robot. To get the second robot's coordinates, for instance, I get it by calling $state[robotTurn * 2]$ for the x value, and $state[robotTurn * 2 + 1]$ for the y value. By keeping track of which robot's turn it is, I could generalize functions such as getX() to return $state[robotTurn * 2]$ and I would receive the coordinates of the current bot trying to move. This was especially helpful in my getNeighboringSpaces() function. This function looked at the coordinates to the North, East, South, and West of the current bot to see if the coordinates were open, and would return a list of the open spots adjacent to the bot. The function NoWall() simply returns a boolean true if there is no wall in the given x, y coordinate, and false if there is a wall. The function doRobotsCollide() also returns a boolean true if that move will cause a collision between robots, and false if it doesn't. Thus a neighbor is only created if there will be no collision with a wall or another robot. The following segment of code shows how I would check if the North spot was open:

```
// if north is open
currY++;
RobotNode safeNode = new RobotNode(stateTemp);
if(currY < rows && !doRobotsCollide(currX, currY, safeNode) && noWall(currX, currY)) {
  stateTemp[robotTurn*2+1] = currY;
  safeNode = new RobotNode(stateTemp);
  neighbors.add(safeNode);
}
currY--;
 stateTemp[robotTurn*2+1] = currY;
```

The function doRobotsCollid() takes in the coordinate the robot is trying to move to, and returns true if a robot is in that position, and false if a robot is not in that position. So a coordinate is only added to the list of neighbors if it does not contain a wall or another bot. With a function that returns a list of all the nodes a robot can move to, we can now implement A^*.

3 A-Star Search

First I will go through the data structures I used for A^* and give a brief explanation as to why I chose such a data structure. First for the priority queue, I implemented a Java Priority Queue named 'heap' (I used heap as the name for my own purpose of visualizing a priority queue, not because it is a heap). In order to input a node and the node's priority, I created a custom class called a heapNode, which took in a SearchNode, and a priority (int), which was calculated during the search. I also had to override the comparator of the Java Priotiry Queue, which only is used for integers. This override was to check if the priority of a node was smaller than the priority of a second node. I used a HashMap called predMap to map nodes to their predecessor nodes in order to backchain after the search was done. To keep a list of all of the nodes that were processed in A^*, I used a HashSet called visited. I used a HashMap because they have very fast run times for visited.contains calls. To keep track of the priorities of each node, I used HashMaps called f_g and f_h, which kept track of each node's distance from the start, and each node's heuristic value respectively. These were useful when dealing with nodes that were already in the PQ and a faster path through that node was found. Lastly, I used a HashSet to hold an ArrayList of integers (to represent a state) called obsInHeap, which kept track of all of the states that were in the PQ. This was necessary because I was using my custom class for the PQ, and the PQ contains call for the PQ was unable to effectively see if a node was already in the PQ. The following segment of code is the while loop for my A^* function:

```
while(!heap.isEmpty()){
  SearchNode currNode = heap.peek().node;
  if(currNode.robotAtGoal()) {
    List<SearchNode> path;
    path = backchain(currNode, predMap);
    return path;
  }
  // If the node has already eben visited, do not process it
  if(visited.contains(currNode) && currNode != searchStartNode) {
    continue;
  }
  heap.remove();
  visited.add(currNode);
  for(SearchNode neighbor : currNode.getNeighboringSpaces()){
    if(visited.contains(neighbor))
17           continue;
18       // all neighbors are 1 away from currentNode, so g(nbor) is g(curr) + 1
19       int tempG = f_g.get(currNode.getState())+1 + neighbor.heuristic(neighbor);
20       if(!obsInHeap.contains(neighbor.getState())){
21         hn = new heapNode(tempG + neighbor.heuristic(neighbor), neighbor);
22         obsInHeap.add(neighbor.getState());
23         heap.add(hn);
24         // if the node has been visited, see if new priority from heuristic is better
25       } else{
26         if(tempG >= f_g.get(neighbor.getState())+f_h.get(neighbor.getState())) {
27                 continue;
28                  }
29       }
30       predMap.put(neighbor, currNode);
31       f_g.put(neighbor.getState(), tempG);
32       f_h.put(neighbor.getState(), tempG + neighbor.heuristic(neighbor));
33       }
34}
```

Starting at line 3, it is first necessary to check to see if the current node represents the robot at its goal. If it is at its goal, backtrack to get the path, and return the path. If not at the goal, check to see if the current node has already been visited. If it has been visited, then it has already been processed, thus does not need to be processed again. After these initial checks, we then want to get all of the neighboring spaces for the robot, and we do that by removing the top of the PQ, adding the current node to visited (because it is currently being processed thus should not be processed in the future), and getting into a for loop for all of the neighbors returned by the getNeighboringSpaces function on the current node. If the neighbor has been visited before, there is no need to add it to the PQ to be processed later. Now, calculate the estimated path length of the node, which is the depth of the current node + 1 (since all neighbors are 1 away from their predecessors), + the heuristic of the neighboring node. The heuristic used in this problem is the average of the manhattan distances of the robots, which will always be optimistic. Once the estimated path is calculated, we then want to check if the node is already in the PQ. If it is not in the PQ, we have found a new node, and myst create a new node and add it to the PQ with the estimated path length as the priority of the node. If the node is already in the PQ, we want to replace it if the new estimated path to the neighbor is better than the old estimated path. I do this by checking if the new estimated cost is larger than the old estimated cost. If it is, then there is no need to change anything, because the old path is the better path. If this is untrue, however, we want to update the neighbor node's important information, which includes predecessor, f_g value, and f_h value. By making the predecessor of the neighbor node be the current node, we are updating the old path to the new, better path, and by adding in the new *f_g* and *f_h* values to those hash maps, we are making sure that if this node is found again as a neighbor node in the search, the better path cost estimates are being used in comparison if a better path is found. We do not have to delete the node with the worse estimated cost from the PQ because when we pop it off of the PQ, it will already be in the visited data structure, and it will be skipped and not processed before it is removed from the PQ. Outside the while loop there is a return null statement, so if a path is not found to the goal, the function will return null.

4 Multi-Robot Coordination

4.1 Discussion Questions

1. The discussion for the state representation of the multi robot maze world problem is in the introduction. To represent all of the robot locations if you forgot where the robots were supposed to go, all you need is the x and y coordinate of each bot. In order to know which actions are available form this state, you

also need to know if surrounding areas have walls or are open, but the walls in the maze are constant, so are not needed in the state of the system.

2. This is also discussed in the introduction. There are n^{2k} possible states, not counting states where the bots are in the same place. There are $k! * n^2$ possible states where one or more robots are in the same place, since the $k!$ represents any number of robots in the same place, and there are n^2 possible locations of this happening. So a more accurate number of possible states would be $n^{2k} - k! * n^2$.

3. If we are assuming that n is much larger than k, we are not worrying about if the robots are in the same location (robot collisions), because that will happen less frequently as n becomes larger and larger than k. The other collisions would be if a robot runs into a wall. If the number of walls is W, then there would be roughly $k!W$ collisions. The k represents any number of robots hitting W walls.

4. I would expect a breadth first search to work well without collisions if the start locations of the robots are roughly evenly distributed. With a relatively low number of walls, we can assume that the manhattan distance of a robot will be very close to the bfs path of the bots. However, if the goals of the robots are very close together, then I would not expect it to be feasible given that there will be many collisions as the robots get close to their goals, and the BFS would not allow for paths to be reconsidered to be able to find paths around the other moving robots.

5. The heuristic I was using was the average of all the manhattan distances of the robots to their goals. This is monotonic, meaning the heuristic of the successors of a node will never be larger than that of the current node. It is monotonic because every action will bring one of the robots closer to its goal, thus the manhattan distance decreases by 1 for the moving robot, thus the average of the manhattan distances will also be smaller for the successor states of each current state.

6. This will be discussed later

7. The 8-puzzle is similar to this problem in the sense that there are open spaces (in the 8 ball puzzle there is only one space) and the tiles can only be moved into the open spaces. It is different because in this case, only tiles adjacent to the open space can be moved. So if a tile is right next to it's goal location, it cannot be moved until the open space is in the tile's goal location, which may take a series of complex moves to reach. While the manhattan distance of each tile is admissable, since a tile may need to make more than 1 moves before the open space is in it's goal space, it does not take into account how many tiles need to be moved in order for one tile to reach its goal state. Instead, a better heuristic would be the number of tiles out of place. This is admissable because each tile that is out of place takes at least one move to get to its goal.

8. The state space of the problem would be the coordinate of each state. To prove that the state space is in two disjoint sets, I would show that state A's neighbors, say state B, does not have a neighbor that is state A. I would implement this in my getSuccessors() function, comparing the neighbors of a node's neighbor, to see if that original node was in the set of its neighbor's neighbors.

4.2 Implementation

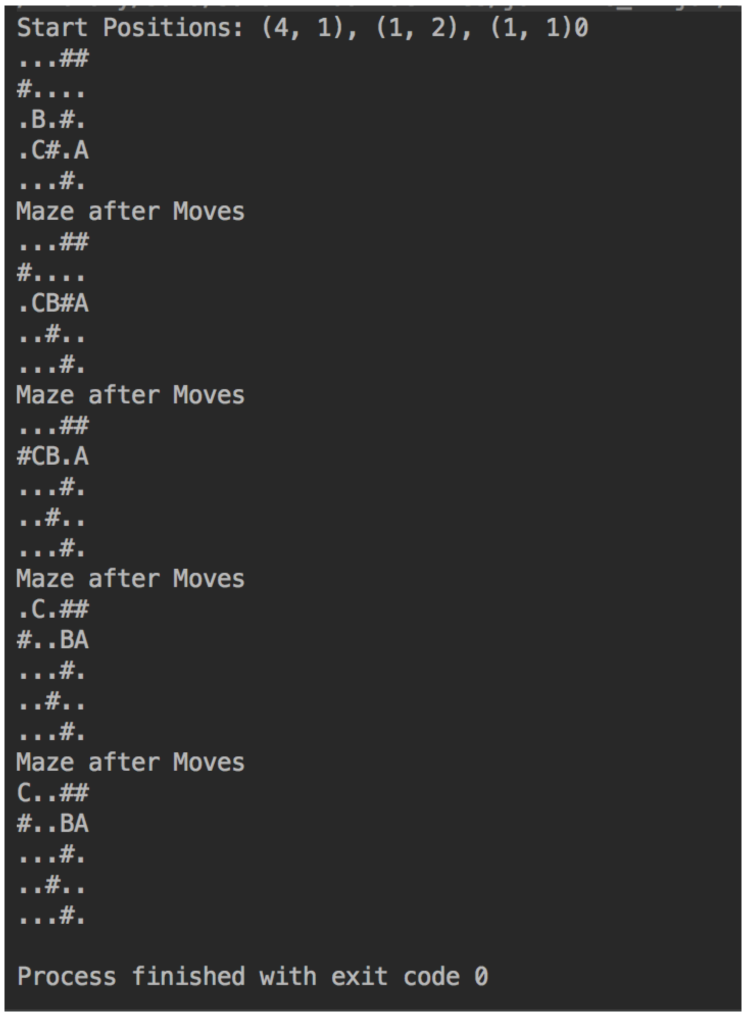
The process of the multi-robot problem is to run A^* search on each robot in order (i.e. A, B, C, back to A, B, C, A, B, C, etc.), and move each robot to the first coordinate in the path found in A^*. This is done by creating a new node after each robot movement with the updated robot's position, where each node represents the movement of only one robot at a time. I needed to add some functionality to my getNeighbors() function, as seen in the first code snippet, in order to make sure a robot movement would not collide with a wall, or another robot. This is vital to the multi-robot problem because if a robot is going to run into another robot, it cannot move to that collision position, and may have to not move for a turn or two. The movement of the robots is driven in my Driver() function in the SearchNode class. The Driver() function is below:

```
1   while(goals){
2     path = aStarSearch(currNode);
3     if(path != null){
4       if(path.size() == 1)
5         currNode = path.get(0);
6       else
7         currNode = path.get(1);
8     } else{
9        System.out.println("Bot␣will␣wait␣till␣its␣next␣turn");
10  }
11  // go to next robot's turn and update the world.
12  currNode.updateTurn(turn);
13  turn = currNode.getTurn();
14  currNode.updateWorld();
15  if(turn%3 == 0) {
16    System.out.println("Maze␣after␣Moves");
17    currNode.mazeToString();
18  }
19  if(currNode.goalCoordinate())
20    goals = false;
21  }
```

This runs by calling aStarSearch on the current node, and since robotTurn is a global variable, we do not have to pass it to A^*, but we do have to update it using the updateTurn function, which oscillates robotTurn between 0, 1, and 2. After A^* is called and finished, if the path is null, then there is not a path for that node in the current maze. If there is no path, then the current robot may have to wait for another bot to move, so it simply waits and lets the next robot move. After a robot is processed in the if statement in lines 3-8, then we update the robot who's turn it is, update the world to represent the movement of the bot, print out the maze after each bot makes a move, and then checks to see if all the bots are in their goal coordinates. The results of the multi Robot solution is show in **Figure 1**.

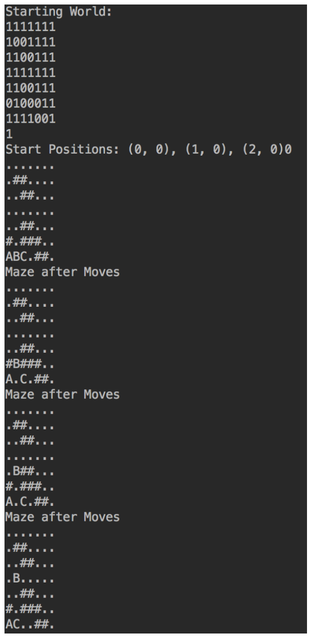
5 Blind Robot

I have no code implementation for the blind robot, as I ran out of time on this assignment. The heuristic to use in this problem is the cardinality of the state space. First, we must discuss what the state space would be in this problem. The state space would be all of the nodes that the bot could possibly be in. Therefore, the beginning state would be a set of all the locations without a wall.



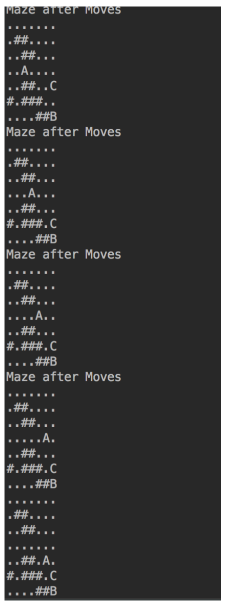
```
Start Positions: (4, 1), (1, 2), (1, 1)0
...##
#....
.B.#.
.C#.A
...#.
Maze after Moves
...##
#....
.CB#A
..#..
...#.
Maze after Moves
...##
#CB.A
...#.
..#..
...#.
Maze after Moves
.C.##
#..BA
...#.
..#..
...#.
Maze after Moves
C..##
#..BA
...#.
..#..
...#.

Process finished with exit code 0
```

Figure 1: The very top maze pictured is the starting state of the maze, with robots A, B, and C. Each maze below the one before represents the maze after each robot has gone through A^* and has moved one space. This is a good test case because it clearly shows the bots moving from coordinate to coordinate without colliding (overwriting) any of the walls or other robots.



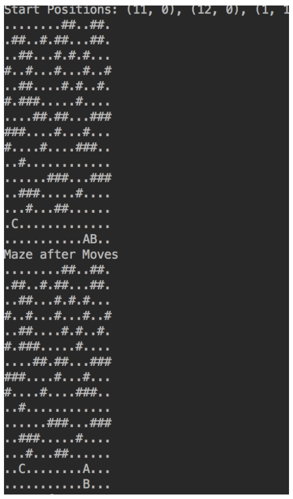
```
Starting World:
1111111
1001111
1100111
1111111
1100111
0100011
1111001
1
Start Positions: (0, 0), (1, 0), (2, 0)0
.......
.##....
..##...
.......
..##...
#.###..
ABC.##.
Maze after Moves
.......
.##....
..##...
.......
..##...
#B###..
A.C.##.
Maze after Moves
.......
.##....
..##...
.......
.B##...
#.###..
A.C.##.
Maze after Moves
.......
.##....
..##...
.B.....
..##...
#.###..
AC..##.
```

Figure 2: This image is a running of the second test maze, a 7 x 7 maze with three bots. This test is interesting because all of the robots start really close together, and all of their goals are close together too, so there is a lot of collision prevention going on in the algorithm. The full movement of the robots is not pictured, but the last few movements are in **Figure 3**.



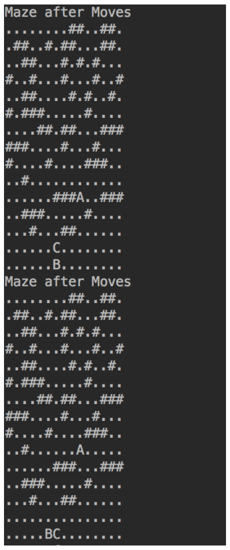
```
Maze after Moves
.......
.##....
..##...
..A....
..##..C
#.###..
....##B
Maze after Moves
.......
.##....
..##...
...A...
..##...
#.###.C
....##B
Maze after Moves
.......
.##....
..##...
....A..
..##...
#.###.C
....##B
Maze after Moves
.......
.##....
..##...
.....A.
..##...
#.###.C
....##B
.......
.##....
..##...
.......
..##.A.
#.###.C
....##B
```

Figure 3: The final movements from Maze 2 to get the robots in their goal coordinates.



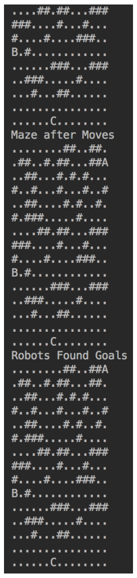
```
Start Positions: (11, 0), (12, 0), (1, 1
........##..##.
.##..#.##...##.
..##...#.#.#...
#..#...#...#..#
..##....#.#..#.
#.###.....#....
....##.##...###
###....#...#...
#....#....###..
..#............
......###...###
..###.....#....
...#...##......
.C.............
...........AB..
Maze after Moves
........##..##.
.##..#.##...##.
..##...#.#.#...
#..#...#...#..#
..##....#.#..#.
#.###.....#....
....##.##...###
###....#...#...
#....#....###..
..#............
......###...###
..###.....#....
...#...##......
..C........A...
...........B...
```

Figure 4: This is a solution for a 15 x 15 maze. This test is interesting given how large the maze is, and how many walls there are. There are tons of collissions in this full solution, and these three images are just samples along the path. The first image is the beginning maze and the maze after all oft the robots make one move.



```
Maze after Moves
........##..##.
.##..#.##...##.
..##...#.#.#...
#..#...#...#..#
..##....#.#..#.
#.###.....#....
....##.##...###
###....#...#...
#....#....###..
..#............
......###A..###
..###.....#....
...#...##......
......C........
......B........
Maze after Moves
........##..##.
.##..#.##...##.
..##...#.#.#...
#..#...#...#..#
..##....#.#..#.
#.###.....#....
....##.##...###
###....#...#...
#....#....###..
..#......A.....
......###...###
..###.....#....
...#...##......
...............
.....BC........
```

Figure 5: These are some of the middle steps in the solution.



```
....##.##...###
###....#...#...
#....#....###..
B.#............
......###...###
..###.....#....
...#...##......
...............
......C........
Maze after Moves
........##..##.
.##..#.##...##A
..##...#.#.#...
#..#...#...#..#
..##....#.#..#.
#.###.....#....
....##.##...###
###....#...#...
#....#....###..
B.#............
......###...###
..###.....#....
...#...##......
...............
......C........
Robots Found Goals
........##..##A
.##..#.##...##.
..##...#.#.#...
#..#...#...#..#
..##....#.#..#.
#.###.....#....
....##.##...###
###....#...#...
#....#....###..
B.#............
......###...###
..###.....#....
...#...##......
...............
......C........
```

Figure 6: Final few steps for the 15 x 15 graph.