MazeWorld

William Brophy Tyree

December 14, 2016

1 Introduction

This lab report required that we spent some time living in "mazeworld." This world is enclosed in a finite width and length. Living in this world is a robot (or multiple robots). Said robot can move in any of four directions: north, east, south, west. At multiple coordinates, there are barriers that the robot cannot collide into. The goal is to get the robot from the start point to the end point. After solving a simple maze variation, we moved on to multiple robot mazes and blind robot mazes.

2 A-star Search

The first step in the process was to implement the A* Search method. In order to do this, I first had to decide which data structures to use. After much experimented, I settled on using multiple HashMaps and a HashSet. While this did cost space, I felt that it was the cleanest way to keep track of all the necessary information.

The HashSet kept track of the visited nodes. Contrary to the name of the structure, this did not hold every node that had been accessed, but only the nodes that had been fully explored. At first, I used a HashMap for this so that I could keep track of the preceding node for the purpose of backtracking. Unfortunately, since the nodes are only added to the table when explored, rather than when they are found as successors, there was no way to know at the time of adding what the preceding node was or what its distance was. For this reason, I kept track of the preceding node of each successor on the optimal path in another HashMap "backtracker." As I found successors for each node, I would add them to backtracker if they were not in visited or frontier with the node they were succeeding as the hashed value to be accessed later. I did the same with the distance (cost of the successor). If the node was already in the frontier, I would check to see if its cost was lower than that already in the frontier. If it was, I would update the distanceMap and the backtracker map. Lastly, I implemented the frontier with a Priority Queue. If the goal was reached, I backtracked to return the path.

Below is the implementation of A* Search...

```
1  public List<UUSearchNode> AStarSearch() {
2    System.out.println("\nFinding a path using AStar...");
3    resetStats();
4    ArrayList<UUSearchNode> returnList;
5    HashSet<UUSearchNode> visited = new HashSet<UUSearchNode>();
6    HashMap<UUSearchNode, UUSearchNode> backtracker = new HashMap<UUSearchNode, UUSearchNode
       >();
7    HashMap<UUSearchNode, Integer> distanceMap = new HashMap<UUSearchNode, Integer>();
8    Comparator<UUSearchNode> comparator = new AStarComparator();
9    PriorityQueue<UUSearchNode> frontier = new PriorityQueue<UUSearchNode>(comparator);
10
11   frontier.add(startNode);
```

```
12
13    while (frontier.size() > 0) {
14      UUSearchNode current = frontier.poll();
15      if (current.goalTest()) {
16        System.out.println("Path Found!");
17        returnList = backchain(current, backtracker);
18        return returnList;
19      }
20
21      visited.add(current);
22      ArrayList<UUSearchNode> successors = current.getSuccessorsWithHeuristic();
23      if (successors != null) {
24        for (int i = 0; i < successors.size(); i++) {
25          UUSearchNode successor = successors.get(i);
26          if (!visited.contains(successor) && !frontier.contains(successor)) {
27            frontier.add(successor);
28            backtracker.put(successor, current);
29            distanceMap.put(successor, successor.getDepth());
30          } else if (frontier.contains(successor)) {
31            if (distanceMap.get(successor) != null &&
32                successor.getDepth() < distanceMap.get(successor)) {
33              frontier.add(successor);
34              backtracker.put(successor, current);
35              distanceMap.put(successor, successor.getDepth());
36            }
37          }
38        }
39      }
40    }
41    return null;
42  }
```

3 Multi-robot coordination

Things began to get complicated when we started to incorporate multiple robots. Here, we had k (labeled A, B, C...) robots living in an n x n rectangular maze. The rules were the same as above, but only one robot could move at a time, specifically in the order of A, B, C... The first challenge came in representing the state of the game. With k robots in an n x n maze, that gave us an upper bound of $(n*n)^k$ states. For one robot, there are $(n*n)$ states. For two robots, there are $(n*n)$ possible locations for the second robot for every $(n*n)$ locations for the first robot. This continues to apply as more robots are added. If there are w wall squares and n is much larger than k, the number of states that represent collisions would be around $(w*k)+k(n*n)$. The way that I got this number is the following: The first product is all of the states where a robot is in the same coordinate as a wall. The second product is the number of robot collisions with eachother. k robots can collide in any coordinate, so the product is the number of coordinates times the number of robots.

Below is the setup for the Multi Robot Maze Node...

```
1  private class MultiMazeWorldNode implements UUSearchNode {
2    // maintains the locations of all of the robots
3    private int[][] state;
4    // maintains the current cost as well as the index of the robot that is moving
5    private int cost, robotToMove;
6
7    public MultiMazeWorldNode(int d, int robotIndex, int[][] robotPositions) {
8      // wrap around if the last robot just moved
9      if (robotIndex == robotPositions.length) {
10       robotToMove = 0;
11     } else {
```

```
      robotToMove = robotIndex;
    }

    state = robotPositions;
    cost = d;
  }
```

The information stored in the state of this node is the cost of the state, the index of the robot that is queued to move, and a 2-D array of the positions of all of the robots in the maze. If there is a large maze with very few walls, I think that bfs would be a computationally feasible route for all start and goal pairs. Without many walls, the robots would probably not have to move around each other and could easily navigate to their goals. The A* Search and the heuristic are needed primarily in tight quarters when the robots need to move around each other and do tricky things to navigate.

The heuristic that I used is monotonic and quite useful. It's pretty simple - it is a variation of the Manhattan heuristic. It is a combination of the Manhattan heuristics from every node's position to its goal. This goal is monotonic, as each robot will be moving toward its individual goal. Therefore, n1 + the heuristic cannot be greater than n + the heuristic.

Below is the successors method and the heuristic for multiple robots...

```
// There are five possible moves per state - up, down, left, right and skip.
// this method creates a new state for every move (the state is the locations of all
// robots. Then it calculates the heuristic of the new state. Finally it checks to make
  sure the states are safe. If there are no safe or legal moves available, the method
  skips
// to the next robot, who picks a move at the same cost.
public ArrayList<UUSearchNode> getSuccessorsWithHeuristic() {
  // if the current robot is at its goal, skip the turn
  while (this.individualGoalTest()) {
    robotToMove++;
    if (robotToMove == state.length) {
      robotToMove = 0;
    }
  }

  int newDepth = cost + 1;
  while (true) {
    ArrayList<UUSearchNode> successors = new ArrayList<UUSearchNode>();

    int[][] newStateUpMove = createNewState(this.state[robotToMove][0], this.state[
  robotToMove][1] + 1);
    int depth1 = newDepth + calculateHeuristic(this.state[robotToMove][0], this.state[
  robotToMove][1] + 1);
    MultiMazeWorldNode topSuccessor = new MultiMazeWorldNode(depth1, robotToMove + 1,
  newStateUpMove);

    int[][] newStateDownMove = createNewState(this.state[robotToMove][0], this.state[
  robotToMove][1] - 1);
    int depth2 = newDepth + calculateHeuristic(this.state[robotToMove][0], this.state[
  robotToMove][1] - 1);
    MultiMazeWorldNode bottomSuccessor = new MultiMazeWorldNode(depth2, robotToMove + 1,
  newStateDownMove);

    int[][] newStateRightMove = createNewState(this.state[robotToMove][0] + 1, this.state[
  robotToMove][1]);
    int depth3 = newDepth + calculateHeuristic(this.state[robotToMove][0] + 1, this.state[
  robotToMove][1]);
    MultiMazeWorldNode rightSuccessor = new MultiMazeWorldNode(depth3, robotToMove + 1,
  newStateRightMove);

```

```
31        int[][] newStateLeftMove = createNewState(this.state[robotToMove][0] - 1, this.state[
      robotToMove][1]);
32        int depth4 = newDepth + calculateHeuristic(this.state[robotToMove][0] - 1, this.state[
      robotToMove][1]);
33        MultiMazeWorldNode leftSuccessor = new MultiMazeWorldNode(depth4, robotToMove + 1,
      newStateLeftMove);
34
35        if (topSuccessor.isStateSafe()) { successors.add(topSuccessor); }
36        if (bottomSuccessor.isStateSafe()) { successors.add(bottomSuccessor); }
37        if (rightSuccessor.isStateSafe()) { successors.add(rightSuccessor); }
38        if (leftSuccessor.isStateSafe()) { successors.add(leftSuccessor); }
39
40
41        if (successors.size() != 0) {
42          return successors;
43        } else {
44          robotToMove++;
45          if (robotToMove == state.length) {
46            robotToMove = 0;
47          }
48        }
49      }
50    }
51
52
53    public int calculateHeuristic(int x, int y) {
54      int heuristic = 0;
55      int newX = Math.abs(robotsGoals[robotToMove][0] - x);
56      int newY = Math.abs(robotsGoals[robotToMove][1] - y);
57      heuristic += newX + newY;
58      for (int i = 0; i < state.length; i++) {
59        if (i != robotToMove) {
60          int xDist = Math.abs(robotsGoals[i][0] - state[i][0]);
61          int yDist = Math.abs(robotsGoals[i][1] - state[i][1]);
62          heuristic += xDist + yDist;
63        }
64      }
65      return (heuristic);
66    }
```

The 8 puzzle problem is a special case of this problem. It is a 3 x 3 maze in which 8 of the 9 possible coordinates have robots in them. The reason it is unique is that only one robot has an available move at any given time. I think that the heuristic I chose would still be a good heuristic for this problem. A good way to judge the strength of the move is to tell how close every robot is to its goal. Judging a move by this sum would eventually yield the correct path if it is possible. Furthermore, the 8 puzzle belief state is composed of two disjoint sets. One set contains a closed loop of states from which you can get from the start goal to the end goal, while the other set contains all of the states from which it is impossible to get to the goal. There are cases where it is not possible to get to the goal given the arrangement. To prove this, I would run the algorithm on many randomly generated initial arrangements. My guess is that a few of them would return no path, because the state lies in the second set.

In order to prove the algorithm works, I ran it on 5 different test cases. For the sake of space, I do not display the paths taken to get to the goal state, but you can see them by running the code. These test cases are interesting because they show various scenarios in which the robots have to move around eachother and around walls in order to get to their goal. The example in the top right corner is the most indicative of the necessary cooperation between robots, as one has to exit the corridor to allow the other to reach its goal before it can re-enter the corridor. Similarly, the bottom example has three robots that need to change their order in order to get to the goal state. Robot C gets to its goal first, so robots A and B have to take a different route (down and around) to get to their goal.

Here are the initial states and setups for these test cases:

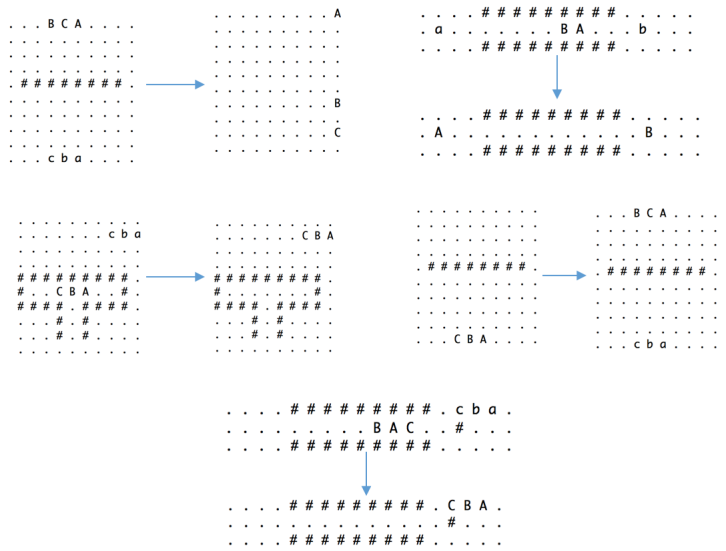

Figure 1: The Multi Robot Examples

4 Blind Robot with Pacman Physics

The final portion of the lab involves a blind robot in a maze with Pacman physics. This robot does not know where it starts, but has a sensor that can tell it what direction is North. Despite this, the robot can't tell when it has hit a wall. The algorithm below allows this blind robot to reach its goal using belief states. At the beginning, the belief state is all possible coordinates of the maze. By moving up, the robot can remove the lower y coordinates from its belief state, shrinking the size of the state. The robot has reached its goal when the only state in the belief state is the goal coordinates. I set the problem up as followed to represent this belief state:

```
1   private ArrayList<Integer> stateX;
2   private ArrayList<Integer> stateY;
3   int cost, topMostOption, bottomMostOption, rightMostOption, leftMostOption, x, y;
4   String direction;
5
6   public BlindSearchNode(ArrayList<Integer> possibleX, ArrayList<Integer> possibleY, int
      startX, int startY, String theDirection) {
7     stateX = possibleX;
8     stateY = possibleY;
9     cost = this.calculateHeuristic();
10    x = startX;
11    y = startY;
12    direction = theDirection;
13
14    if (x < 0) {
15      x = 0;
```

```
    } else if (x >= mazeWidth) {
      x = mazeWidth - 1;
    }

    if (y < 0) {
      y = 0;
    } else if (y >= mazeWidth) {
      y = mazeWidth - 1;
    }

    // variables hold the topMost, bottomMost, leftMost, and rightMost
    // coordinates in the current belief state. These values are used when
    // a robot moves (removed or added)
    topMostOption = 0;
    bottomMostOption = mazeHeight;
    rightMostOption = 0;
    leftMostOption = mazeWidth;
    for (int i = 0; i < stateX.size(); i++) {
      int x = stateX.get(i);
      rightMostOption = Math.max(x, rightMostOption);
      leftMostOption = Math.min(x, leftMostOption);
    }
    for (int i = 0; i < stateY.size(); i++) {
      int y = stateY.get(i);
      topMostOption = Math.max(y, topMostOption);
      bottomMostOption = Math.min(y, bottomMostOption);
    }
  }
```

I kept track of two ArrayLists, one representing the possible x coordinates in the belief state and one representing the possible y coordinates of the belief state. I chose an ArrayList so that coordinates could easily be added or removed. Every time a robot moves, I generated a new state with a modified belief state. When the robot moved north, for example, the bottom-most y coordinate would be removed from the belief state. Also, the (top-most y coordinate + 1) would be added to the belief state unless the top-most coordinate was already the highest possible y coordinate. I did the same for all other actions. Below is the implementation of the creation of the new belief state for all successors:

```
public ArrayList<Integer> createNewState(String direction) {
    ArrayList<Integer> returnState = new ArrayList<Integer>();

    if (direction.equals("up")) {
      for (int i = 0; i < stateY.size(); i++) {
        if (stateY.get(i) != bottomMostOption) {
          returnState.add(stateY.get(i));
        }
      }
      if (topMostOption < mazeHeight && !returnState.contains(topMostOption)) {
        returnState.add(topMostOption);
      }
    }
    else if (direction.equals("down")) {
      for (int i = 0; i < stateY.size(); i++) {
        if (stateY.get(i) != topMostOption) {
          returnState.add(stateY.get(i));
        }
      }
      if (bottomMostOption >= 0 && !returnState.contains(bottomMostOption)) {
        returnState.add(bottomMostOption);
      }
    }
    else if (direction.equals("left")) {
      for (int i = 0; i < stateX.size(); i++) {
```

```
26          if (stateX.get(i) != rightMostOption) {
27            returnState.add(stateX.get(i));
28          }
29        }
30        if (leftMostOption >= 0 && !returnState.contains(leftMostOption)) {
31          returnState.add(leftMostOption);
32        }
33      }
34      else if (direction.equals("right")) {
35        for (int i = 0; i < stateX.size(); i++) {
36          if (stateX.get(i) != leftMostOption) {
37            returnState.add(stateX.get(i));
38          }
39        }
40        if (rightMostOption < mazeWidth && !returnState.contains(rightMostOption)) {
41          returnState.add(rightMostOption);
42        }
43      }
44      return returnState;
45    }
46
47
48    private boolean isStateSafe() {
49      if (this.stateX.size() != 0 && this.stateY.size() != 0 ) {
50        if (maze.getMazeState(this.getx(), this.gety()) != '#') {
51          return true;
52        }
53      }
54      return false;
55    }
```

The heuristic I used built on the Manhattan Distance I had been using before. The heuristic summed up all of the Manhattan distances to the goal state of all the possible states. In addition, I used a multiplier to make sure the robot was choosing moves that would get it closer (and eventually to) the goal state. I raised the multiplier to the power of the distance of each possible state to the goal state and added that to the heuristic as well. This assured that the lowest cost belief states would be the ones with the least number of possible coordinates that were closest to the goal state, eventually producing a belief state of one coordinate that was the goal state. Below is the heuristic:

```
1   public int calculateHeuristic() {
2     int heuristic = 0;
3     int multiplier = 10;
4     for (int i = 0; i < stateX.size(); i++) {
5       int distanceToGoalX = Math.abs(goalx - stateX.get(i));
6       heuristic++;
7       heuristic += Math.pow(multiplier, distanceToGoalX);
8     }
9     for (int i = 0; i < stateY.size(); i++) {
10      int distanceToGoalY = Math.abs(goalx - stateY.get(i));
11      heuristic++;
12      heuristic += Math.pow(multiplier, distanceToGoalY);
13    }
14    return heuristic;
15  }
```

Below is a snippit of a path that the blind robot is producing followed by the end result of that path:

```
. . . . . .
. . . . g .
. . . . . .
. # . . . .
. s . . . .
. . . . . .

Finding a path using AStar...
Path Found!

. . . . . .
. . . . g .
. . . . . .
. # . . . .
. s > . . .
. . . . . .
The robot just moved east

The current belief state is:
Potential x coordinates: (1, 2, 3, 4, 5)
Potential y coordinates: (0, 1, 2, 3, 4, 5)
. . . . . .
. . . . g .
. . . . . .
. # ^ . . .
. s > . . .
. . . . . .
The robot just moved north

The current belief state is:
Potential x coordinates: (1, 2, 3, 4, 5)
Potential y coordinates: (1, 2, 3, 4, 5)
. . . . . .
. . . . g .
. . ^ . . .
. # ^ . . .
. s > . . .
. . . . . .
The robot just moved north

The current belief state is:
Potential x coordinates: (1, 2, 3, 4, 5)
Potential y coordinates: (2, 3, 4, 5)
. . . . . .
```

Figure 2: The annimation snippet

```
. . . . ^ .
. . . ^ < >
. . ^ > . .
. # ^ . . .
. s > . . .
. . . . . .
The robot just moved west

The current belief state is:
Potential x coordinates: (4)
Potential y coordinates: (4)
The whole path is: east, north, north, east, north, east, north,
south, east, west,
------------------------------
```

Figure 3: The final path