Mazeworld

James Jia

September 28, 2016

1 Introduction

In this report, I will discuss my approach to the mazeworld problem. First, I will cover my implementation of an A Star search algorithm on a simple maze with a single robot, and then expand the game state to incorporate multiple robots. In the multiple robots version, I will create a model for finding non-overlapping routes for the robots to take in order to reach the goal. I will then use the A Star search algorithm to find the optimal path for the robots to take in order to reach their respective goal states.

2 A Star Search

I implemented a model of a single robot problem in MazeProblem.java and tested the A Star Search algorithm on the simple model in MazeDriver.java. I used the 7x7 maze from the assignment sheet for this test. The A Star Search uses a priority queue called exploreQueue to arrange MazeNode objects by their relative costs, a function of both their distance away from the start node as well as their manhattan distance away from their goal node. Additionally, similar to BFS, it uses a HashMap to keep track of where each node is reached from in order to backchain each node to its previous node in order to return the final path. Finally, the algorithm uses a hashmap called costMap in order to keep track of the costs of visited states - if a state is reached again with a cheaper cost, then the costMap is updated with the new cheaper node. The algorithm is implemented in SearchProblem.java and is shown below.

```java

  //Search with uniform cost elements as well as heuristic elements
  //cost of move is sum of node depth and manhattan distance
  public List<UUSearchNode> aStarSearch(){

    resetStats();
    //Priority Queue
    PriorityQueue<UUSearchNode> exploreQueue = new PriorityQueue<UUSearchNode>();

    //visited for backchain, costMap to keep track of costs
    HashMap<UUSearchNode, UUSearchNode> visited = new HashMap<>();
    HashMap<UUSearchNode, Integer> costMap = new HashMap<>();

    List<UUSearchNode> successors;
    List<UUSearchNode> finalPath;
    UUSearchNode node = startNode;

    //put startnode into the queue and the visited set and the cost map
    exploreQueue.add(node);
    visited.put(node, null);
    costMap.put(node, node.getCost());

    boolean empty = exploreQueue.isEmpty();

    //until there are no more nodes to explore in the queue
    while (!empty){

      //remove the first value in the queue and store it in currentNode
      incrementNodeCount();
      UUSearchNode currentNode = exploreQueue.poll();
      successors = currentNode.getSuccessors();
      updateMemory(exploreQueue.size() + visited.size() + costMap.size());

      if(currentNode.goalTest()){
        //call backchain to return path from end to startnode when goal is reached
        finalPath = backchain(currentNode, visited);
        return finalPath;
      }

      //if not goal, then look at successors
      for (int j=0; j<successors.size(); j++){
        UUSearchNode successor = successors.get(j);

        //if successor hasn't been visited or is currently mapped to higher cost, put it in visited and
    put it in the queue
        if (!costMap.containsKey(successor)|| successor.getCost()<costMap.get(successor)){
          visited.put(successor, currentNode);
          costMap.put(successor, successor.getCost());
          exploreQueue.add(successor);
        }
      }
    }
    //if goal isn't found, return null;
    return null;
  }
```

The output of A Star Search yielded the following results for the simple maze problem, which I compared to the results I got from using BFS. The maze used in this example was the one from the top of the assignment sheet and the startnode was 0,0 and the goal node was 6,6.

```
    BFS Path:
    [(00), (01), (02), (03), (04), (05), (06), (16), (26), (36), (46), (56), (66)]
Path length: 13
Nodes explored during last search 77
Maximum memory usage during last search 88

    Astar Path:
    [(00), (01), (11), (21), (31), (41), (51), (61), (62), (63), (64), (65), (66)]
Path length: 13
Nodes explored during last search 35
Maximum memory usage during last search 125
```

As expected, A Star search returns the optimal path, which is the same length as the one returned by BFS. However, it explores around half as many nodes due to a heuristic that underestimates the cost to the goal for each node. Additionally, it is worth pointing out that the maximum memory usage is significantly higher for A Star Search, as not only does it have to keep track of a queue full of nodes to explore and a hashmap for backchaining, it also has to maintain an additional hashmap full of successor nodes' costs.

3 Multi-robot Coordination

Next, I created a model that simulated multiple robots taking turns moving. This required some changes to the MazeProblem class in order to implement. Instead of taking ints for the starting x/y and the goal x/y, the problem would take arrays of ints. For example, the x coordinate of the first robot's starting position would correspond to the first element in the startX array that is passed to the maze problem. Additionally, I created a 2d int array variable inside of MultiMazeNode that would keep track of the states of all the robots and an int variable that would keep track of which robot's turn it is. In this model, move selection is a little bit different - instead of simply moving one robot, one first has to identify the robot whose turn it is to move. In order to find the successor states, I first copied all the values in the current state into a temporary 2d array called newstate. Then, I would change the part of newstate corresponding to the robot whose turn it was to move. Then, I would check the updated overall state with an isSafe() method to determine whether the state was valid. If it was valid, I would wrap it in a MultiMazeNode and add it to a list of successorNodes. The getSuccessors() function is shown below.

```
    public ArrayList<UUSearchNode> getSuccessors() {

      ArrayList<UUSearchNode> successorList = new ArrayList<UUSearchNode>();
      int[][] newstate = new int[robotNumber][2];

      //first copy old state
      for(int i=0; i<robotNumber; i++){
        for(int j=0; j<2; j++){
          newstate[i][j]=state[i][j];
        }
      }

      //for robot whose turn it is, check all possible moves, then check if overall state is safe
      for(int x=0;x<moveset.length; x++){
        newstate[turn][0]=state[turn][0]+moveset[x][0];
        newstate[turn][1]=state[turn][1]+moveset[x][1];

        //check if safe
        if(isSafe(newstate)){
          int[] xList = new int[robotNumber];
          int[] yList = new int[robotNumber];
          for(int a=0; a<robotNumber; a++){
            xList[a]=newstate[a][0];
            yList[a]=newstate[a][1];
          }

          //greedy algorithm - set distance to 0 to solely rely on heuristic
          //int distance = 0;
          int distance=Math.abs(newstate[turn][0]-state[turn][0])+Math.abs(newstate[turn][1]-state[turn
    ][1]);
          //wrap new state in a node and add it to the list
          MultiMazeNode successor = new MultiMazeNode(xList, yList, depth+distance, (turn+1)%robotNumber);
          successorList.add(successor);
        }
      }
      return successorList;
    }
```

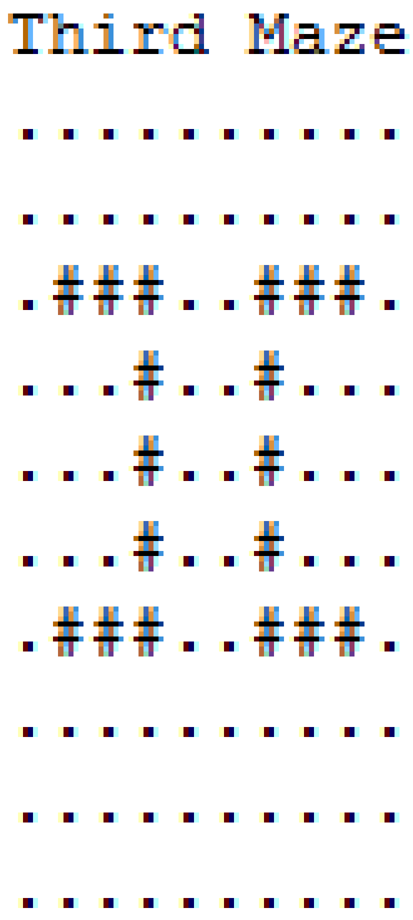
The isSafe() method differed from that of the single robot problem in that not only did the method have to check for collisions, it also had to check for whether or not paths ran into each other. To do this, I used a hashmap called ”occupied” to check whether all the robots’ xy coordinates were unique. This method is shown below.

```
  public boolean isSafe(int[][] newstate){

    //for each node, check if within bounds and not negative
    for (int i=0; i<robotNumber; i++){

        if (newstate[i][0]>=maze.width || newstate[i][0]<0){
          return false;
        }
        if (newstate[i][1]>=maze.height || newstate[i][1]<0){
          return false;
        }

        //check if wall
        if (maze.grid[newstate[i][0]][newstate[i][1]] == 1){
          return false;
        }
    }

    //check if there are any collisions, i.e. robots can't be in same coordinate
    LinkedList<String> occupied = new LinkedList<String>();
    for(int j=0; j<robotNumber; j++){

      //convert coordinates to string to check for equality
      String state = "";
      state=Integer.toString(newstate[j][0])+Integer.toString(newstate[j][1]);
      if (occupied.contains(state)){
        return false;
      }
      occupied.add(state);
    }
    return true;
  }
```

4 Sample Output

Here is some sample output taken from the output of MultiMazeDriver.java. This is from the maze marked MAZE 3.

The figure shown below is a maze that was constructed to test the whether the heuristic would work when the goal would have to be reached by first moving in a way that would actually increase the node’s manhattan distance from the goal. Each of the two C shaped sections would have a node start inside them and they would have to back out of the walls before



```
Third Maze

..........

..........

.###..###.

...#..#...

...#..#...

...#..#...

.###..###.

..........

..........

..........
```

proceeding toward their goal. The heuristic was able to find the path to the goal due to the algorithm using A Star search, which was not completely dependent upon the Manhattan distance and also considered a uniform cost element as well. The two nodes started at (2,4) and (7,4), and were able to find their goal nodes of (9,4) and (0,4) respectively by following this path:

[(2,4)(7,4), (1,4)(7,4), (1,4)(7,4), (0,4)(7,4), (0,4)(7,4), (0,3)(7,4), (0,3)(7,4), (0,2)(7,4), (0,2)(7,4), (1,2)(7,4), (1,2)(7,4), (2,2)(7,4), (2,2)(7,4), (3,2)(7,4), (3,2)(7,4), (4,2)(7,4), (4,2)(7,4), (5,2)(7,4), (5,2)(7,4), (6,2)(7,4), (6,2)(7,4), (7,2)(7,4), (7,2)(8,4), (7,2)(8,4), (7,2)(9,4), (7,2)(9,4), (7,2)(9,3), (8,2)(9,3), (8,2)(9,2), (8,2)(9,2), (8,2)(9,1), (8,2)(9,1), (8,2)(8,1), (8,2)(8,1), (8,2)(7,1), (8,2)(7,1), (8,2)(6,1), (8,2)(6,1), (8,2)(5,1), (8,2)(5,1), (8,2)(4,1), (8,2)(4,1), (8,2)(3,1), (8,2)(3,1), (8,2)(2,1), (8,2)(2,1), (8,2)(1,1), (8,2)(1,1), (8,2)(0,1), (8,2)(0,1), (8,2)(0,2), (9,2)(0,2), (9,2)(0,3), (9,3)(0,3), (9,3)(0,4), (9,4)(0,4)]

The rest of the MultiMazeDriver.java file tests other mazes of varying sizes with 2-3 robots. Note: The last 40x40 maze needs to run for around a minute before returning the path.

5 Discussion 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?

We need 2*the number of robots to represent the xy coordinates of all the robots in the problem. Additionally, we need one more number to represent whose turn it currently is, so the total is (2*k + 1).

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

Assumming collisions are valid states, each robot has n squared possible positions, as it has n possible x coordinates * n possible y coordinates. Each additional robot, therefore, increases the number of states by a factor of n squared. Thus, the overall number of possible xy coordinates for the robots = n to the power of 2k. However, there are also k possible turn positions for each set of xy coordinates, so the final number of possible states is k(*n to the power of 2k).

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 maximum number of potential xy coordinates is k*(n to the power of 2k). However, with w walls, this number becomes k*(n to the power of 2k) -1. Assuming all combinations of x and y occurr equally frequently, the possibility of k robots having the same x coordinates is 1/(nsquared). (1/(nsquared)/(k*(n to the power of 2k)-w) would be a rough estimate of how many collissions there would be.

4. If there are not many walls, n is large (say 100x100), and several robots (say 10), do you expect a straightforwards breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not?

I would assume this to be computationally infeasible. From question 2, the number of states possible would be 10*(100 to the 20th power), or (10 to the 41st power), which is a massive number of states that would require an extremely long time to run a breadth first search on, due to the fact that BFS weighs all paths equally.

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.

A monotonic heuristic function, or one that underestimates the true distance to the goal for each state, is the manhattan distance of the state to the goal. The reason for this is due to the fact that the actual distance will almost certainly be greater if there are a significant number of walls between the state and the goal. Another monotonic heuristic would be just the manhattan x coordinate distance from the state to the goal; the reason for this would be the aforementioned walls as well as the fact that there may be additional y distance that the state

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 5x5 to 40x40. (You might want to randomly generate the 40x40 mazes.) Ill 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?)

See sample output above and MultiMazeRobot.java's output.

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?
The 8-puzzle is a special case of this problem because it's the model is exactly the same, except that for each state there is only one acceptable move that will lead to a safe state. The heuristic function will still be good for the 8-puzzle because it will still underestimate the true cost of each state to the goal. Each tile will likely have to move more than the manhattan distance because it will probably have to back out at some point to let other tiles move.

6 Blind Robot

Blind robot was not implemented due to time constraints.