MazeWorld

Vivek Sah

January 21, 2016

1 Introduction

In this paper, we will discuss different approaches to finding a path from a starting point to goal point in a maze. We have a robot which can move in any of four directions: North (towards the top of the screen), East, South, and West. There are coordinates on the maze. (0, 0) is the bottom left corner of the maze. (6, 0) is the bottom right corner of the maze. This paper is divided into three different sections. First, we will implement a search algorithm to find optimal path in the maze. Then, we will try to model and implement the scenario when there are more than one robots and they have to coordinate between themselves to reach the goal state. The final part of this paper will be dedicated to discussing the case of blind kidnapped robot in a maze which does not have any sensors.

Model

Before going further, we should discuss the organization of this project. We have an abstract class called SearchProblem.java . It has an interface called SearchNode. Searchproblem.java has some uninformed search algorithms implemented (BFS, Path-checking DFS). To implement A-star search algorithm, we will create a new class "InformedSearchProblem" which will extend SearchProblem.java. Furthermore, to implement MultiRobot scenario and blind robot scenario, we will create a class called *MultiRobotRobot.java* and *blindRobot.java* which will extend InformedSearchProblem.

2 A-star search

A-star search algorithm is an informed search where we make a decision from each successor state using heuristic and the cost of path(let's say priority). The priority tells us which path to choose. In this problem, we will use manhattan distance between currentState and goalState as heuristic and uniform cost design. The implementation is similar to BFS algorithm in that we maintain a frontier and check all the nodes in the frontier before going further. However, we will use PriorityQueue as a datastructure for A-start algorithm. That is because, everything we want to remove a node which has the highest priority based on its cost and heuristic. A snippet with comments about further detail is included beneath.

```
while (!frontier.isEmpty()) {
incrementNodeCount();
updateMemory(frontier.size() + reachedFrom.size());
SearchNode currentNode = frontier.remove(); //pop the highest priority
    element from priorityQueue
if (reachedFrom.containsKey(currentNode) && reachedFrom.get(
   currentNode) != null){ //if this node is already visited
        if (currentNode.getCost() > reachedFrom.get(currentNode).
           getCost() + 1 ) {//if current node's cost is greater than
           the one in the hashmap
                System.out.println("found duplicate, skipping");
                continue; //skip the rest
```

```
                }
        }
        reachedFrom.put(, currentNode);
        if (currentNode.goalTest()) {
                return  backchain(currentNode, reachedFrom);
        }
        ArrayList<SearchNode> successors = currentNode.getSuccessors(); //get
            successors
        for (SearchNode node : successors) { //go thorugh the successors
                // if not visited
                if (!reachedFrom.containsKey(node) ) { //if not visited, add
                    to the priorityQueue
                        reachedFrom.put(node, currentNode);
                        frontier.add(node);
                } else {
                        if ( reachedFrom.get(node) != null && node.getCost() <
                             reachedFrom.get(node).getCost()+1 ) { //if
                            visited heck its cist
                                reachedFrom.put(node, currentNode);
                                frontier.add(node);
                        }
                }
        }
}
return null;
}
```

2.1 Multi-Robot Coordination

In this case, instead of one robot in the maze, we have multiple robots, say k. The problem is stated as such. Robots can either move in one of four directions or stay stationary. They also take turns to move.

Model We will be using a new class called MultiRobotProblem. To represent the class, we will use an array of size $2k+1$. The first k indices will hold the x and y cooordinates of each robot. The last index will hold the turn value(which robot's turn is next?).

Let's look at the implementation of getSuccessors for this class:

```
 ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
 int turn = this.state[this.state.length-1] ; //which robot turn is it, the
     last index of the state array
 // index of x position of current robot is (turn-1)*2
for (int[] action: actions) {
        int xNew = state[(turn-1)*2] + action[0]; //move the robot whose turn
            it is
        int yNew = state[(turn-1)*2+1] + action[1];
      //if its legal move in the maze check to see if other robots are not in
          the same location
      if(maze.isLegal(xNew, yNew)) {
          boolean noCollison = true;
          int[] newSucc = new int[this.state.length];
          for (int i = 0; i < state.length-1 ; i++) {
              newSucc[i] = state[i];
              if (i % 2 == 0 && i != (turn - 1) * 2) { //go to every x
                  coordinate of other robot
                   if (state[i] == xNew && state[i + 1] == yNew) {
```

```
                    noCollison = false;
                } }
            if (noCollison) {
                newSucc[(turn - 1) * 2] = xNew;
                newSucc[(turn - 1) * 2 + 1] = yNew;
                newSucc[this.state.length - 1] = (turn + 1 > this.numRobots ?
                    1 : turn + 1);
                SearchNode succ = new MultiRobotMazeNode(newSucc, getCost() +
                    1.0);
                successors.add(succ);
            }
        }
      }
}
return successors;
```

In this method, we will add actions to the state of the robot who has the current turn. Next we will check if the new state for the current robots doesn't clash with walls or other robots. Then, we increase the cost of the successor and increment the turn.

To define the hashcode, we will just convert the array to an integer such as 1,0,1,1,1 =¿ 10111.

For heuristic, we will use sum of the manhattan distances for each robots and define priority to be sum of this heuristic and cost of the current state.

Discussion Questions

- We have total n^2 location on maze and we have k robots, so total possible states will be less than $\binom{n^2}{k}$
- For a nXn maze , when there are k robots, we will need $2k+1$ numbers to represent the state as apart from the location of each robot, we will need to know whose turn it is.
- If there are w walls, there will $n^2 - w$ free spaces, so total collisions will be total states-free states

$$\binom{n^2}{k} - \binom{n^2 - w}{k}$$

- If there are not many walls, we should not use straightforward BFS as we will get a lot of valid successors. With a branching factor of 5, the the number of nodes will rise exponentially.
- For heuristic, we can use the sum of manhattan distances of each robot. Since Manhattan distance are positive and monotic, the sum of manhattan distances should also be monotonic

Interesting scenarios

- A narrow corridor all around the maze. Their heuristic will not change a lot.
- A bottleneck situation where all robots have to go through a narrow pass
- One robot reaches its goal and the others have to go through it. So, it has to move away from its goal. What if the it has to move away very far from its goal state to allow others to pass through.
- What if the maze is spiral? (spiral in a rectangular way?) and they have to reach in reverse order while moving.
- What if the maze has long corridors towards the goal state but which haveget blocked right before the goal state and the robot has to reach all the way to the end before realizing that it has to turn back?

8 puzzle The heuristic will work in this case because our heuristic will return a optimal path. The problem is also similar to multirobot problem where each of the queen has to be at a certain x and y thoughout the chess board.

Here, we have certain values of x, y which need to be filled. From one perspective, the x -coordinate can be thought of as independent from y-coordinates. In my program, I will change the getSuccessors method, so that instead of just looking for x,y collisions, it should look for all robots which have same x or same y (so two robots will be considered to collide even if they only have one coordinate same)

2.2 Blind robot

In this case, we have a blind robot which is blind, and doesn't know where it starts! The robot does know the map of the maze.

Model and Implementation

We will implement the *blindRobot.java* class to implement Blind robot. First, we will initialize the blind Robot and invoke a method to place at a certain place on the maze. Then, once it is at a certain place, it can just act like a normal SearchNode and use all the algorithms to get to the goal state. This is a slight diversion from the task but seems more straightforward.

To initialize the blindrobot problem, we will just need to pass a maze as an argument. Then in its initialization step, it will create all the possible places that it could be in. We will store all these possibilities in a linkedHashSet which denotes our state. We will use LinkedHashSet so that when we add searchNodes to our set, they can iterated in order. That will be easier for debugging purposes. Next, we call a function called solveBlindRobot which ascertains its starting position.

Let's look at the solveBlindRobot function:

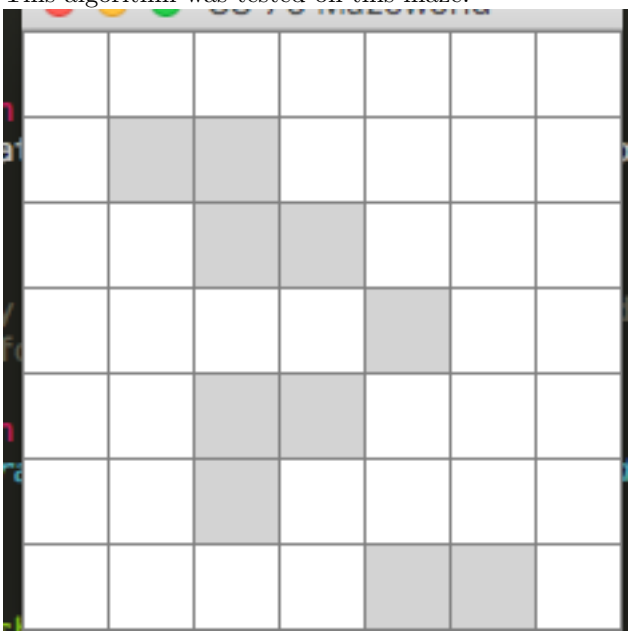
```
public void solveBlindRobot( ){ //figures out a starting point for the blind
    robot
    if (possibles.size() == 1){ //base case
        return;
    }
    LinkedHashSet<blindNode> newPossiblePlaces= new LinkedHashSet<blindNode>()
        ; //store new state
    int[] minStuckAction = null;
    int minStuckNumber = 999999999;
    int movNum = 0; //count for movement number for an action=> how many
        places move
    for (int[] action: actions) {
        if (!(Arrays.equals(action, getReverse(prevMove))) && !(Arrays.equals(
            action, restrictedMove) )) {
            int newStuckNumber = getStuckNumber(action); //for each action
                determine how many p;aces are going to be stuck
            //prioritize the actions which result in minum stuck positions and
                then the ations which result in maximum movement
            if (minStuckNumber > newStuckNumber) {
                minStuckNumber = newStuckNumber;
                minStuckAction = action; //action resulting in minimum stuck
                    positions
            } else if (minStuckNumber == newStuckNumber) {
                int tempMov = moveNumber(action);
                if (movNum < tempMov) {
                    minStuckAction = action;
                    minStuckNumber = newStuckNumber;
                    movNum = tempMov;
                }}}}
    prevMove = minStuckAction;
    solution.add(minStuckAction); //add  this action to solution

    for (blindNode node: possibles) {
        int xNew = node.state[0] + minStuckAction[0];
        int yNew = node.state[1] + minStuckAction[1];
        if (maze.isLegal(xNew, yNew)) {
            blindNode newNode = new blindNode(xNew, yNew, 0);
            newPossiblePlaces.add(newNode);
        }else{
```

```
                newPossiblePlaces.add(node);
            }
        }
        if(newPossiblePlaces.size() == possibles.size()) {
            restrictedMove = minStuckAction;
        }
        possibles = newPossiblePlaces; //set new state to the real state of the
            system
        }
```

Here's how the algorithm works. We will try the action which results in minimum number of occurences where we will get stuck between walls. Being stuck is defined as being in the position such that the action which led to this state, its reverse will get us out of there. In our blindRobot function, we have some helper functions for solveBlindRobot such as getStuckNumber() , isStuck() and moveNumber(). We will always apply the action which results in minimum stuck position. If we have two minimums, we will prioritize the one which results in maximum movements. Using this heuristic, we will be able to reach a certain location which can the startNode of the problem.

This algorithm was tested on this maze:



The output was:

Action: -1:0———Action: -1:0———Action: -1:0———Action: -1:0———Action: -1:0———Action: -1:0———Action: 0:1———Action: 1:0———Action: 0:1———Action: -1:0———Action: 0:-1———Action: -1:0———Action: -1:0———Action: 0:1———Action: -1:0———Action: 0:-1———Action: -1:0———Action: 0:-1———Action: -1:0———Action: 0:-1———Action: 0:-1———Action: -1:0———Action: -1:0———Action: 0:1———Action: -1:0———Action: 0:-1———Action: -1:0———Action: 0:-1———Action: 0:-1———Action: -1:0———Action: 0:-1———Action: 0:-1———Action: 1:0———Action: 0:-1———Action: -1:0———Action: 0:-1———Action: 1:0———Action: 0:-1———Action: -1:0———Action: 0:-1———

StartNode is Maze state 0, 0 depth 0.0