Maze World

Niranjan Ramanand 09/29/16

1 Introduction

1.1 Problems

There are several path-finding problems that we discuss in this report. In each, we have to navigate to a certain goal position in a maze, but each also involves different conditions. One problem involves multiple robots trying to find an optimal cooperative path to a goal and another involves a robot that lacks any sensors except those for direction. While the former is a challenge to implement, the latter has more theoretical challenges that will be addressed very much like a formal mathematical proof. Due to space, many of the bigger maze examples will only be in the code or in auxiliary files.

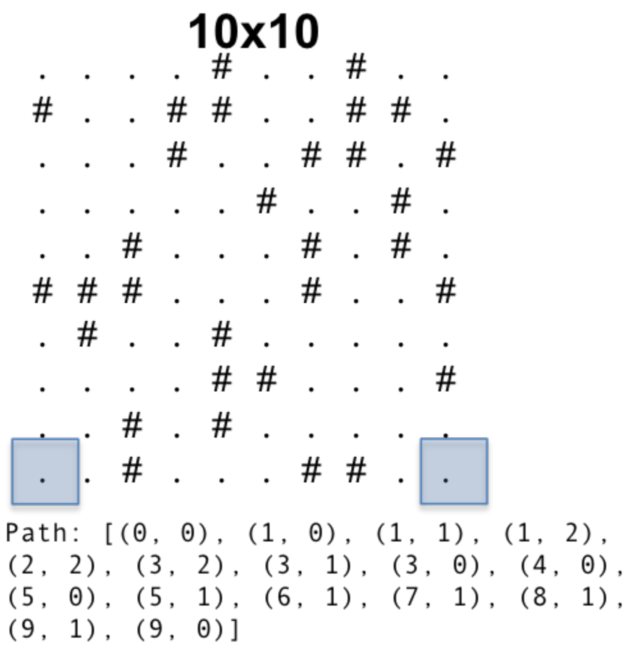
1.2 Previous Work

Work on problems similar to the multi-robot problem, cooperative path-finding problems, was done by Standley, T. and Korf, R. and illustrated in their "Finding optimal solutions to cooperative path-finding problems." In this paper, the authors improve upon their previous Independence Detection - Operation Decomposition (ID-OD) algorithm that had the two major disadvantages of being costly time-wise due to being optimal and not providing coherent results when terminated prematurely. The algorithm used to impose a cost on exploring alternate paths that were not optimal, but, by removing the constraint, an "approximation algorithm" was constructed. The algorithm is complete, but this modification allows the algorithm to come up with alternative, less-expensive paths if terminated prematurely, a stark contrast to its prior form. A very neat idea is that of the anytime algorithms that result, where solutions become "more" optimal the longer the algorithm is run. We do not implement any such approximations in the report, but they are helpful to know because, as mentioned in the paper, many cooperative path-finding problems are PSPACE-hard.

2 Searches

2.1 Breadth-First Search Warm-up

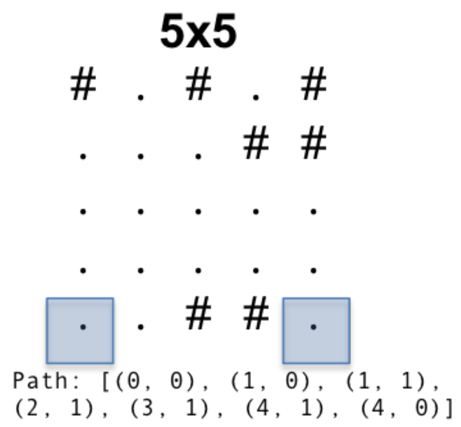
We make use of the same BFS implementation from the previous assignment by extending UUSearch-Problem. We create several mazes randomly, although we could load a maze from a file as well (some maze files provided). We begin our paths from the bottom left corner of the maze. Although instructions asked to make the goal node the right corner of the maze, the code allows any goal node to be specified. In our implementations, we will often translate the maze into a 2D array of 0's and 1's (the latter being in-maze walls).



10x10

```
. . . . # . . # . .
# . . # # . . # # .
. . . # . . # # . #
. . . . . # . . # .
. . # . . . # . # .
# # # . . . # . . #
. # . . # . . . . .
. . . . # # . . . #
. . # . # . . . . .
. . # . . . # # . .
```

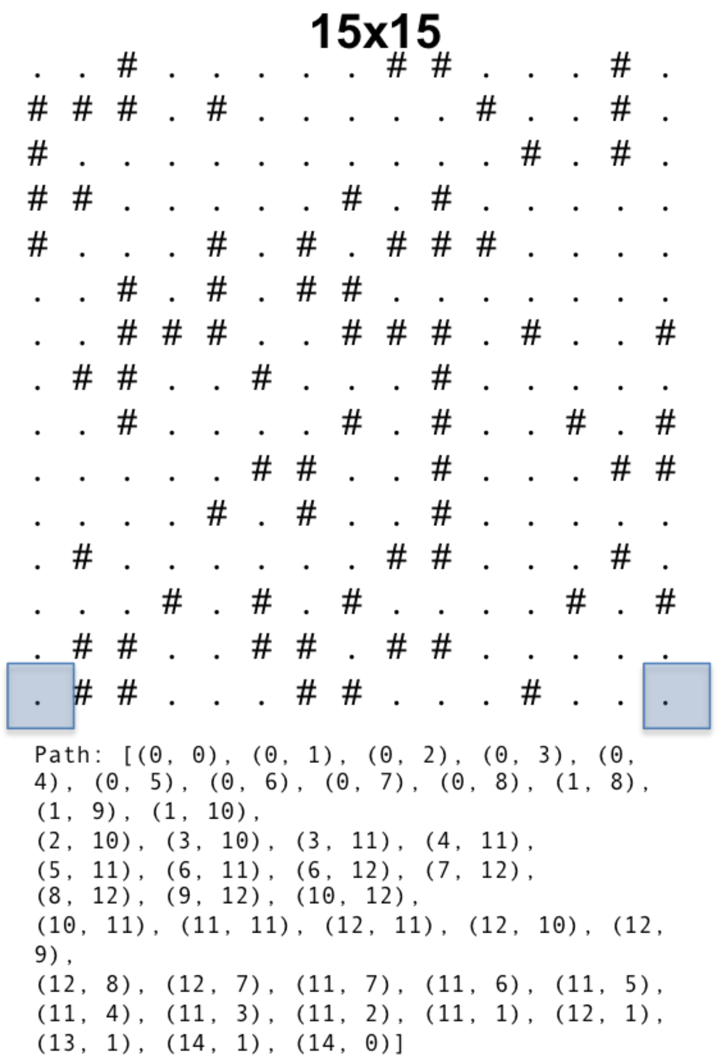
```
Path: [(0, 0), (1, 0), (1, 1), (1, 2),
(2, 2), (3, 2), (3, 1), (3, 0), (4, 0),
(5, 0), (5, 1), (6, 1), (7, 1), (8, 1),
(9, 1), (9, 0)]
```



5x5

```
# . # . #
. . . # #
. . . . .
. . . . .
. . # # .
```

```
Path: [(0, 0), (1, 0), (1, 1),
(2, 1), (3, 1), (4, 1), (4, 0)]
```



15x15

```
. . # . . . . . # # . . . # .
# # # . # . . . . . # . . # .
# . . . . . . . . . . # . # .
# # . . . . . # . # . . . . .
# . . . # . # . # # # . . . .
. . # . # . # # . . . . . . .
. . # # # . . # # # . # . . #
. # # . . # . . . # . . . . .
. . # . . . . # . # . . # . #
. . . . . # # . . # . . . # #
. . . . # . # . . # . . . . .
. # . . . . . . # # . . . # .
. . . # . # . # . . . . # . #
. # # . . # # . # # . . . . .
. # # . . . # # . . . # . . .
```

```
Path: [(0, 0), (0, 1), (0, 2), (0, 3), (0,
4), (0, 5), (0, 6), (0, 7), (0, 8), (1, 8),
(1, 9), (1, 10),
(2, 10), (3, 10), (3, 11), (4, 11),
(5, 11), (6, 11), (6, 12), (7, 12),
(8, 12), (9, 12), (10, 12),
(10, 11), (11, 11), (12, 11), (12, 10), (12,
9),
(12, 8), (12, 7), (11, 7), (11, 6), (11, 5),
(11, 4), (11, 3), (11, 2), (11, 1), (12, 1),
(13, 1), (14, 1), (14, 0)]
```

2.2 A* Search

Unlike breadth-first search, where all nodes are regarded equally, A* search assigns priority to certain nodes. These are determined via the formula $S = g(n) + h(n)$, where $g(n)$ is the distance travelled from the start node and $h(n)$ is a heuristic that underestimates the distance to the goal node. This, coupled with $h(n)$ equalling zero at the goal node, means that $h(n)$ is an "admissible" heuristic. We use a min-priority queue provided by Java implement this, so the node with the smallest S value is pursued. We make the heuristic a function that every node implementing SearchNode must have a value for. Note that simply making this function always return zero will make the algorithm a normal Uniform-Cost Search

```
public List<SearchNode> aStarSearch(){
        PriorityQueue<SearchNode> pq = new PriorityQueue<>(20, new SearchNodeComparator());

        //backchain to get the path
        HashMap<SearchNode, SearchNode> backchainMap = new HashMap<>();
        //allows shortest distances to be constantly updated
        HashMap<SearchNode, Integer> distanceFromStart = new HashMap<>(); //Also functions as visited

        pq.add(startNode);
        distanceFromStart.put(startNode, 0);

        SearchNode goal = null;

        frontierLoop:
        while(!pq.isEmpty()){
                SearchNode current = pq.poll();
                if(current.goalTest()){
                        goal = current;
                        break frontierLoop;
                } else {
                for(SearchNode successor : current.getSuccessors()){
                                int newDist = distanceFromStart.get(current) + 1; //successor is one away from current

                                if(backchainMap.get(successor) == null ||
                                                distanceFromStart.get(successor) > newDist){

                                        distanceFromStart.put(successor, newDist);
                                        successor.updateDistance(newDist);
                                        backchainMap.put(successor, current);
                                        pq.add((SearchNode)successor);
                        }
                }

        }

        }
                return goal == null ? null : backchain(goal, backchainMap);
}
```

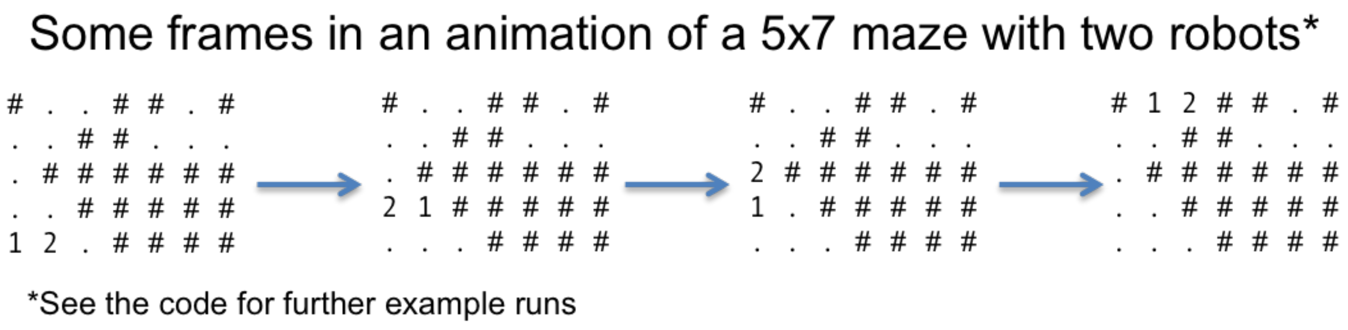
Notice that the distance from the start to a SearchNode is not a property of the SearchNode, as it should be updated whenever a better path to it is found.

3 Problem Implementations

We have three classes for our three problems: BFSMazeProblem, MultiRobotProblem, and BlindRobotProblem. These classes house each problem's respective nodes. Each of these nodes implements SearchNode, a class analogous to UUSearchNode in our previous assignment. It contains the main A* algorithm. We then actually load in/construct the mazes, which we converted to 2D arrays of integers, in the driver classes.

3.1 Multi-Robot Coordination

In this problem, we represent the coordinates of k different robots with a k-by-2 2D array of integers. This, together with an integer representing the current robot turn, makes up the state. We see that an upper bound on the number of states is therefore the number of turns multiplied by the number of ways the robots can be arranged in the maze, or $B_1 = k * (n^2)^k$. Note that this does not take into account that robots cannot collide with each other. We see that there are $B_2 = k * ((n^2 - w) * (n^2 - w - 1) * ... * (n^2 - w - k + 1)$ permutations of robot arrangement and turns. This is because $n^2 - w$ is the number of places a robot can be, and we decrease each term by one in the product to prevent two robots from being placed in the same spot. Thus, when $k << n$, we can approximate that $B_1 - B_2 = k * (n^{2k} - (n^2 - w)^k)$ collisions exist in our former estimate of B_1. When n is large and there are not too many walls, we see going from two robots to ten robots involves an order-increase of going from approximately $2 * n^{2*2} = 2 * n^4$ to $10 * n^{2*10} = 10 * n^{20}$. For large n, we see this latter upper bound is insanely high!! Clearly, BFS, being an algorithm that gradually checks all surrounding nodes, will not be able to process certain start-goal pairs.


Some frames in an animation of a 5x7 maze with two robots*

```
# . . # # . #        # . . # # . #        # . . # # . #        # 1 2 # # . #
. . # # . . .        . . # # . . .        . . # # . . .        . . # # . . .
. # # # # # #   →    . # # # # # #   →    2 # # # # # #   →    . # # # # # #
. . # # # # #        2 1 # # # # #        1 . # # # # #        . . # # # # #
1 2 . # # # #        . . . # # # #        . . . # # # #        . . . # # # #
```

*See the code for further example runs

The heuristic for the system of robots is the sum of all the robots' Manhattan distances. Note that A* works optimally with this heuristic because it underestimates (or equals) the actual distance to the goal. This heuristic is also consistent because, regardless of what node you pick, the heuristic of the node's successor is going to differ by one (one robot moving will change the Manhattan distance by just one). If the node has heuristic value H, therefore, its successor has heuristic value at least $H - 1$. However, the two nodes differ by one, and so we see that $H \leq (H - 1) + 1 = H$ (a.k.a. the triangle equality at equality.) In this algorithm, getSuccessors() looks at the adjacent positions available to the current robot and also updates robot turn:

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

    ArrayList<SearchNode> successors = new ArrayList<>();
        int nextTurn = (currTurn + 1) % currCoords.length;

        int x = currCoords[currTurn][0];
        int y = currCoords[currTurn][1];
        int [][] arr = null;

        if(x < maxX && maze[y][x + 1] != 1){ //right node
                if(!hasRobot(x + 1, y)){
                        arr = arrayCopy(currCoords);

                        arr[currTurn][0] = x + 1;
                        arr[currTurn][1] = y;
                        successors.add(new MultiRobotNode(arr, nextTurn));
            }
    }
```

```
        //right, bottom, and top nodes...
        successors.add(new MultiRobotNode(arrayCopy(currCoords), nextTurn)); //make no move

            return successors;
}
```

The 8-puzzle is a specific problem associated with the multi-robot problem. Each piece can be a robot and, because the heuristic is admissible, we will always find an optimal solution. The heuristic, however, incentivizes pieces to stay close to their goals, although this could be a negative when it comes to the 8-puzzle because, to solve the puzzle, pieces will many times need to be moved away from their eventual positions. Like a Rubik's cube, the puzzle has two disjoint sets whose union is the total number of its states. The program as it currently is will return a null path when there is no path to the end goal. Thus, we set the solved puzzle as the goal and try all permutations of starting positions. We can then show that only half of the initial configurations will be solved. By swapping two pieces in our goal state, we get a state that none of these initial configurations can be turned into. The other half, however, could all transform to this new goal state. Furthermore, by tracking which sets are solved in which goal state, we can show that exactly two equal, disjoint sets make up the state space of the 8-puzzle.

3.2 Blind Robot Planning and A Polynomial-Time Solution

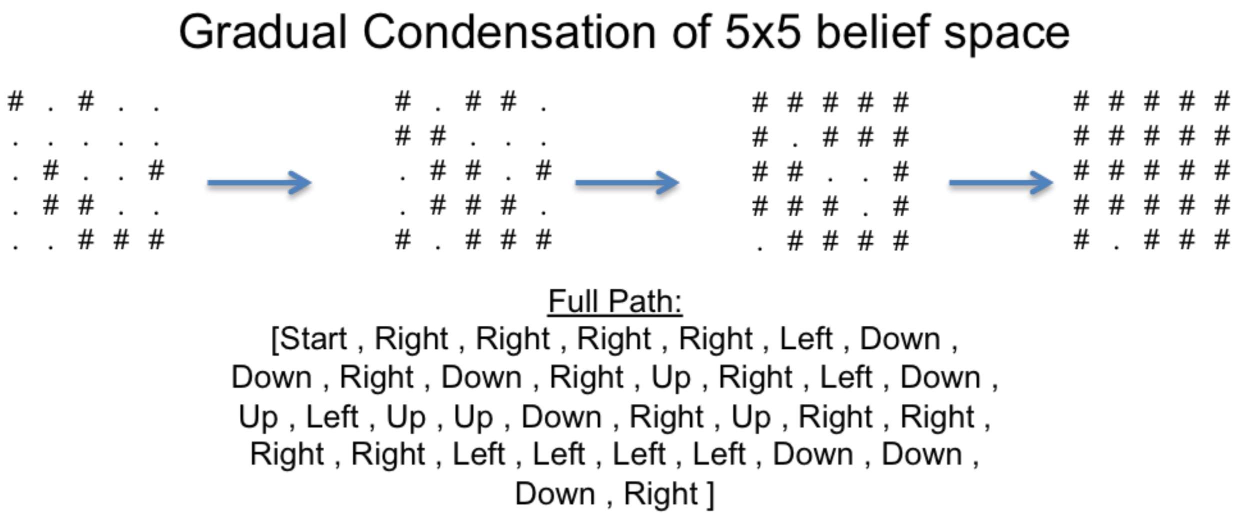
The blind robot described in the instructions can be guided to a single spot by condensing its belief space to that single spot. Thus, each node in this search problem will consist of all the places in the maze where the robot may be. Thus, the state is described only by these potential positions. We use an A* search algorithm to find a path to the node that only has the goal in its set of potential positions. Thus, we need to come up with a heuristic and there are a few that may do the job. The one used in the code makes use of the idea that by going in one of the four directions, the belief state can be shrunk by a certain amount. The heuristic looks at all four directions and tries to find the highest number of potential positions that the robot can rule out as the result of moving somewhere. This results in a higher priority (lower cost) for nodes that give immediately better results, very similar to greedy search.

```
public int heuristic() {
        int max = 0; int sum = 0; int numPossiblePositions = 0;

        FromTheBottom:
        for(int i = 0; i < beliefState.length; i++){
                sum = 0;//
                for(int j = 0; j < beliefState[0].length; j++){
                        if(beliefState[i][j] == 0)
                                sum++;
                }
                if(sum > max || sum > 0){
                        if(sum > max) //going in this direction gets rid of more positions
                                max = sum;
                        break FromTheBottom; //found the bottom-most row with a possible position
                                                                // and so no need to continue through other rows
                }
        }
        FromTheTop:
        ...
        FromTheLeft:
        ...
        FromTheRight:
                ...
        for(int i = 0; i < beliefState.length; i++){
                for(int j = 0; j < beliefState[0].length; j++){
                        if(beliefState[i][j] == 0)
                                numPossiblePositions++;
                }
        }

        return numPossiblePositions - sum;
}
```

Note that an alternative heuristic could be derived from adding all of the Manhattan distances from each of the potential positions to the goal. The implemented method in the code works better in cases where there are not too many walls on the interior, as it focuses only on how the belief space can be narrowed down via reducing the number of possible positions near the boundaries of the maze. The alternate heuristic would, however, be more practical when there are a lot of walls, for it would take into account every potential position's relationship to the goal.



Gradual Condensation of 5x5 belief space

```
# . # . .      # . # # .      # # # # #      # # # # #
. . . . .      # # . . .      # . # # #      # # # # #
. # . . #  ->  . # # . #  ->  # # . . #  ->  # # # # #
. # # . .      . # # # .      # # # . #      # # # # #
. . # # #      # . # # #      . # # # #      # . # # #
```

Full Path:
[Start , Right , Right , Right , Right , Left , Down ,
Down , Right , Down , Right , Up , Right , Left , Down ,
Up , Left , Up , Up , Down , Right , Up , Right , Right ,
Right , Right , Left , Left , Left , Left , Down , Down ,
Down , Right]

(Larger mazes given in code)

This condensation occurs in getSuccessors(). There are a few checks in this method to make sure we do not condense incorrectly. For example, we never condense a point if $x = 0, maxX$ and $y = 0, maxY$ and the direction we are moving in (right in the case of the code snippet) opposes the maze boundary. We do something similar when an obstacle is detected.

```java
public ArrayList<SearchNode> getSuccessors() {
        ArrayList<SearchNode> successors = new ArrayList<>();

        int maxX = beliefState[0].length − 1;
        int maxY = beliefState.length − 1;

        //moving right
        int [] [] nextBeliefState = new int [beliefState.length][beliefState[0].length];

        for(int i = 0; i <= maxY; i++){
                for(int j = 0; j <= maxX; j++){
                        if(beliefState[i][j] == 0 && (j == maxX || maze[i][j + 1] == 1)){
                        nextBeliefState[i][j] = 0;
                        } else {
                                if(j > 0 && beliefState[i][j − 1] == 0 && maze[i][j] != 1){
                                        nextBeliefState[i][j] = 0;
                                } else if(j == maxX && beliefState[i][j] == 0){
                                        nextBeliefState[i][j] = 0;
                                } else {
                                        nextBeliefState[i][j] = 1;
                                }
                        }
                }
        }

        if(!Arrays.deepEquals(nextBeliefState, beliefState))
                successors.add(new BlindRobotNode(nextBeliefState, "Right"));

        //moving left, up, and down similarly implemented
        return successors;
}
```

Assuming that the blind robot is in the same connected component as its goal, is there always such a way to condense the belief space? Yes, and it can be rigorously proven. If there was not always such a way, then there must exist a set of potential positions within a connected component of some maze that could not be merged into a single position. Then, it follows that there are at least two different positions that cannot be merged. There must be a shortest path between these two positions, however, because they are in the same connected component. Let us call the two positions A and B. We know that we can move A along the shortest path and B must follow correspondingly. We know that while A is traversing this shortest path, it cannot be obstructed by a wall (otherwise this would not be a path). The interesting thing, though, is that B cannot be obstructed either, for if it ran into a wall while A was traversing the path, then the path would get shorter, and, by construction, we know this cannot happen. Thus, A will eventually end up at B's initial position, and B would have taken an identically shaped path to some other spot. We notice, however, that A can once again traverse the shortest path and B would, again, move correspondingly, without obstruction. The path cannot get longer (for A traversing the shortest path reduces the path by one and B can only move one away, resulting in a net zero change in this shortest path) and it cannot get shorter (by construction). This, however, means that these two points can always make some displacement in some direction, implying the maze is infinite. However, we know the maze must be finite, and thus such a method of condensing the belief space always exists. (qed.)

Piggybacking off of this idea that if we have two positions A and B, we can gradually combine them into one by having A traverse the shortest path between them, we begin to form an idea about how to create an algorithm that condenses the belief space into a single spot in polynomial time.

Let us say that we have a set of n possible places that our blind robot could be. We know that regardless of what we do, the number of possible places that our robot could be does not increase (it either decreases or stays the same). Thus, we can keep picking pairs of two locations and combine them, where we do this $n-1$ times in the worst case, a number proportional to the number of maze cells, k^2, in the worst case. Let's now analyze the complexities of combining two potential places the blind robot might be. The shortest path between two spots in the cell is at worst linear in relation to the number of cells in the maze. Let us say, without loss of generality, that the two points differ in their y-coordinates by m units (we could use the same logic for the x-coordinates as well). Using our previous argument, we need only have our position A traverse the shortest path between A and B at most $\lfloor \frac{k}{m} \rfloor$ if the maze is k-by-k, a number proportional to $\sqrt{k^2}$, in order to reduce the path between A and B by 1. At most, we do this on the order of k^2 times. Altogether we derive a worst case time complexity on the order of $(n-1) * \lfloor \frac{k}{m} \rfloor * k^2$, or $O(k^{4.5})$, or $O(N^{2.25})$, where N is the number of cells. This is therefore an algorithm that is polynomial in the number of cells in the maze.