AI 1 report

160020220

October 2019

Contents

1 Introduction

In this practical, I was asked to implement and evaluate a number of AI search algorithms applied to the task of a flight route planner. The planner operates on an air space composed of N-Parallels and 8 angles, the agent can only travel between parallels and angles intersections.
The agent aims to find the best route from the departure 's' to the goal 'g'. Some occasional obstacles would be marked as 'X' as well.
The goal is to implement a basic agent that uses depth-first search (DFS), and breadth-first search (BFS) following the general algorithm for search.
Then, I had to Implement best-first search (BestF) and A*(AStar) search using a Heuristic approach.
Finally, Some additional functionalities were possible to implement. I chose to implement the obstacle handling functionality.

2 Design

2.1 Running Environment

Main method and Argument

To run the program, First compile it in the source folder with the following command:
javac Main.java
Then run it using :
java Main followed by the arguments according to this usage:
Algorithm — Parallels — Start — Goal

The Main method takes these arguments to first create the map where the flight planner will take place. We then run the appropriate algorithm according to the first argument args[0].
We use a try catch block in case we have wrong inputs.

Program's running logic

Running the search algorithm consist of creating an object of the appropriate class. There, the object constructor runs a method to start the search. Following this, the search begins by the Path generation. This part is done differently depending on the algorithms. When all paths are generated, The best path has to be generated and the final output is produced.
This program logic allows easy debugging and readability but also allows the code to scale if more implementations were to be added.

2.2 Nodes and Paths

Node

To represent a position and state on the map we created earlier, a node logic is used. Indeed, the methods used to traverse, discover and search the map are implemented using Node objects. This node object oriented programming allows the logic to be very versatile. This is shown by using these nodes in all our algorithm's implementations.

Path

The logic behind Path Generation splits into two main parts:

- Generate all the possible paths according to the algorithm.
- Select the best path from Set.

Since the algorithms are well encapsulated, we use this principle for all the paths generation.

2.3 Algorithms' design

depth-first search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
Here, when moving forward and no more nodes are along the current path, we go back on the same path to find nodes to traverse.

This recursive nature of DFS can be implemented using stacks, and this is the direction I chose.
The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

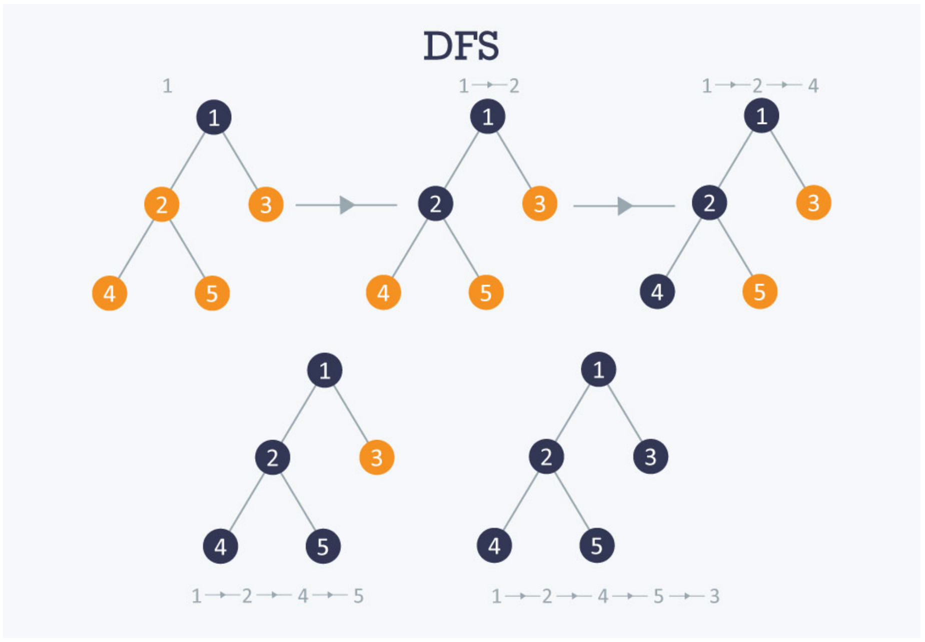


Figure 1: reference: https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Repeat this process until the stack is empty. However, we must ensure that the nodes that are visited are marked. This will prevent the search from visiting the same node more than once. Not marking Nodes as visited will result in an infinite for loop.

breadth-first search (BFS)

BFS is a traversing algorithm where we start traversing from the source node and traverse the graph layer wise thus exploring the neighbour nodes (called adjacent positions in the implementation). Then, A move towards the next-level neighbour nodes is done.
Traversing the graph is done as follows:

- First move horizontally and visit all the nodes of the current layer
- Move to the next layer

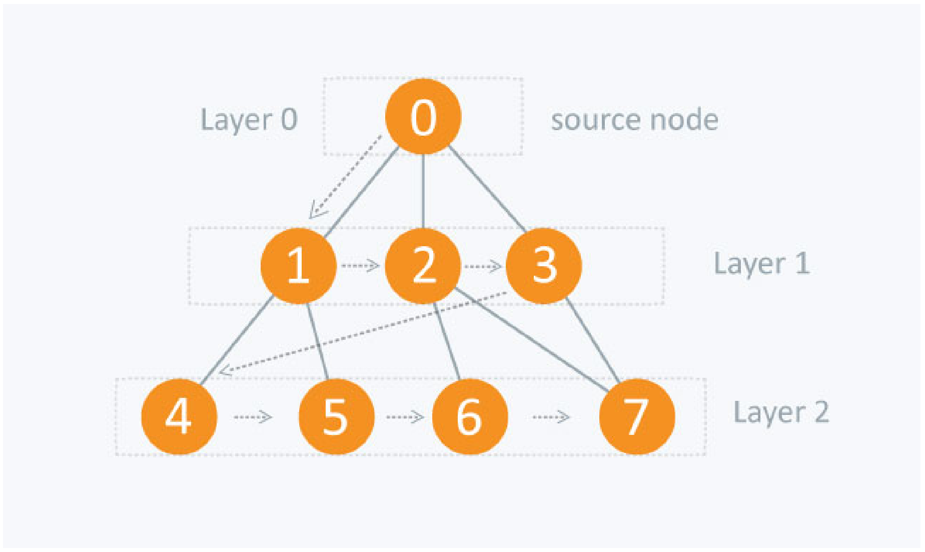


Figure 2: reference: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS, we must traverse all the nodes in layer 1 before moving to the nodes in layer 2.

best-first search (BestF)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

A*(AStar)

What A* Search Algorithm does is that at each step it picks the node according to:
a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination.

This Is referred as a Heuristic approach to the search and it makes this algorithm one of the best and popular used in path-finding and graph traversals.

3 Implementation

3.1 Classes and Project Structure

Map

This Map Class creates the grid map representation. we use here a char 2D array [x][y]. * [x] represents the parallels and * [y] represents the angle. This map representation follows the specification sheet requirement:
"A polar coordinate is a tuple (d, angle), where d is the distance from the pole within [0, N 1] and angle is the angle of direction, one of 0, 45, 90, 135, 180, 225, 270, 315"

We know that the pole can't be used as a path point, but instead of considering it as an obstacle, we simply don't bother representing it at all. Thus having the first parallel at index x[0].

PositionNode

The PositionNode class deals with the node handling explained above. It allows us to store and use useful properties such as Heuristic distance and the parent node if it's part of a path.
A node's main purpose is to hold coordinates, link to a parent and generate children. We do so by using the retrieveAdjacentNodes() method. Here, we add all 4 Children nodes reachable from parent node by movement of one unit on one coordinate axis. This is done by simply checking the surrounding nodes and adding the to the children list as seen fit.

Algorithm

All algorithms extend the Abstract Class Algorithm!
The Search components and inputs are declared here, as we as the main method for search and path reconstruction.

The searchPath() method is very similar throughout all the algorithms implemented:

- We run through a while loop that gets broken when the return is made.
- In this loop, we select the currentNode and it's adjacent positions (called frontier).
- We add this new path to the explored path set.
- We loop and the paths are generated depending on the algorithm.

depth-first search (DFS):
There is an important thing to note concerning this Class. I have attempted both ways of implementing DFS, the recursive way has been commented out while the stack implementation is the one running.
However, The algorithm behaves in a strange way and is not properly working. Thus will not be feature in the testing.
breadth-first search (BFS):
In BFS, we use a Linked list to create a First-in-first-out queue and implement the search.
best-first search (BestF):
We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.
A*(AStar):
In A*, the Heuristic distance to goal is used to generate the paths. Indeed, A property of PositionNode is holding this distance:
this.gcost = Math.abs(this.x - goalCoords[0]) + Math.abs(this.y - goalCoords[1]) + this.x;

3.2 What makes everything work?

Circular Motion

I wanted my search to be able to go from angle 0 to angle 315 and 0 to 315. However, using a 2-D array doesn't allow this looping outside of the array's bounds. To go around that, When looking for adjacent nodes, we look at -1 of the angle index as index 7 and vice-versa. This feature is implemented with the used of conditionals to handle bound cases.

Parallel Priority

Since we have a circle as our flight space, A distance from A to B with [AB] being 2 points with and angle difference of 45 would be bigger on a higher parallel. For this reason, the search algorithms give priority to paths that use lower parallels. The path length is still the most important criteria. However, for 2 paths the same length, the search will return the one that uses lower parallels. Thus having a smaller distance.

Path Generation

To generate the best Path, We use the Explored Set to have a list of all the possible paths. Then, starting from the goal, we backtrack until reaching the source node. the output of this operation is the best paths. Indeed, the key value system of Explored Set allows a valid path generation that has been sorted by the algorithm before hand.
The last thing to do then is to reverse the collection and have our path from source to goal.

Heuristic implementation

The heuristic implementation is held in the PositionNode class as explained above:
this.gcost = Math.abs(this.x - goalCoords[0]) + Math.abs(this.y - goalCoords[1]) + this.x;
Adding this.x at the end of this line allows the search to give priority to the lower parallels.

Obstacles

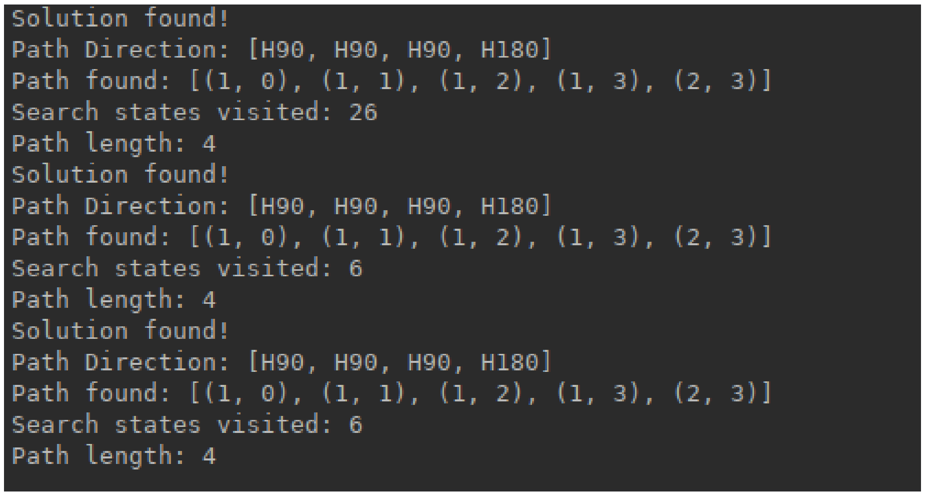
Obstacles are markes as an 'X' character on the map. Then before adding a node to a possible path, We check that this node is valid by calling isValid(). This method returns the node state and checks if the node exists and if the node is not and obstacle.

4 Testing

To have the full data of the algorithms running. Un-comment the print statements.

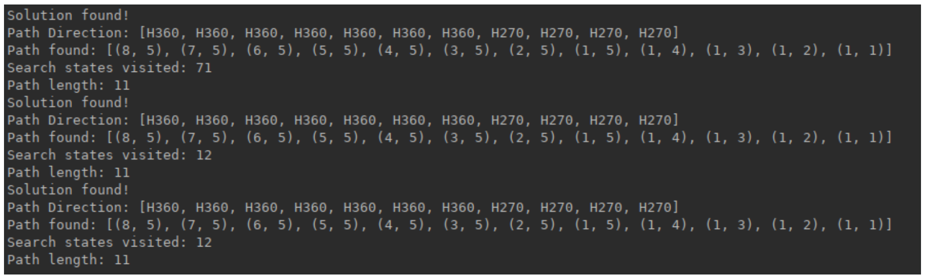
given inputs

I have tested all inputs given in the specification sheet. However, it would be too bulky to have all results printed here. Thus, here are the tests for input 1,8 and 10 selected randomly.



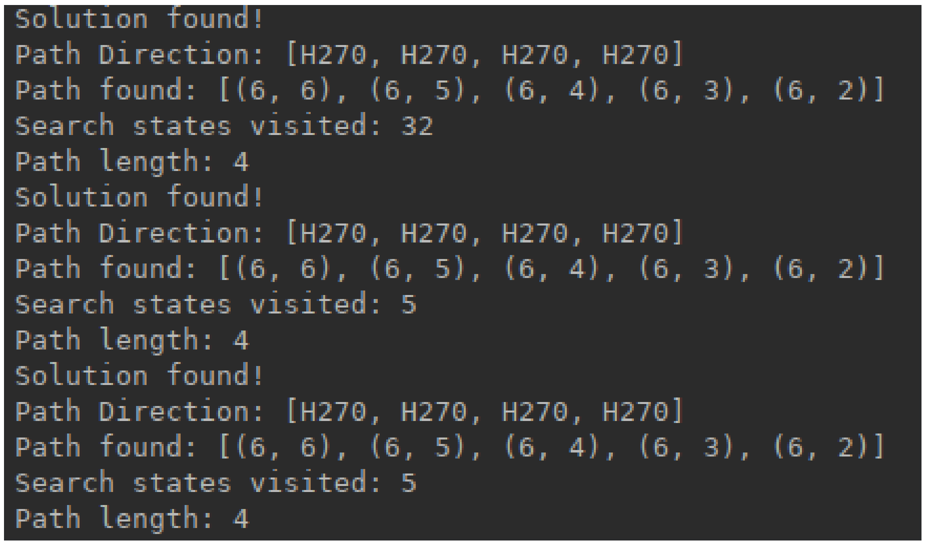
```
Solution found!
Path Direction: [H90, H90, H90, H180]
Path found: [(1, 0), (1, 1), (1, 2), (1, 3), (2, 3)]
Search states visited: 26
Path length: 4
Solution found!
Path Direction: [H90, H90, H90, H180]
Path found: [(1, 0), (1, 1), (1, 2), (1, 3), (2, 3)]
Search states visited: 6
Path length: 4
Solution found!
Path Direction: [H90, H90, H90, H180]
Path found: [(1, 0), (1, 1), (1, 2), (1, 3), (2, 3)]
Search states visited: 6
Path length: 4
```

Figure 3: input 1: N=5 S=(2,0) G=(2,135)



```
Solution found!
Path Direction: [H360, H360, H360, H360, H360, H360, H360, H270, H270, H270, H270]
Path found: [(8, 5), (7, 5), (6, 5), (5, 5), (4, 5), (3, 5), (2, 5), (1, 5), (1, 4), (1, 3), (1, 2), (1, 1)]
Search states visited: 71
Path length: 11
Solution found!
Path Direction: [H360, H360, H360, H360, H360, H360, H360, H270, H270, H270, H270]
Path found: [(8, 5), (7, 5), (6, 5), (5, 5), (4, 5), (3, 5), (2, 5), (1, 5), (1, 4), (1, 3), (1, 2), (1, 1)]
Search states visited: 12
Path length: 11
Solution found!
Path Direction: [H360, H360, H360, H360, H360, H360, H360, H270, H270, H270, H270]
Path found: [(8, 5), (7, 5), (6, 5), (5, 5), (4, 5), (3, 5), (2, 5), (1, 5), (1, 4), (1, 3), (1, 2), (1, 1)]
Search states visited: 12
Path length: 11
```

Figure 4: input 8: viii. N=10 S=(9,225) G=(2,45)



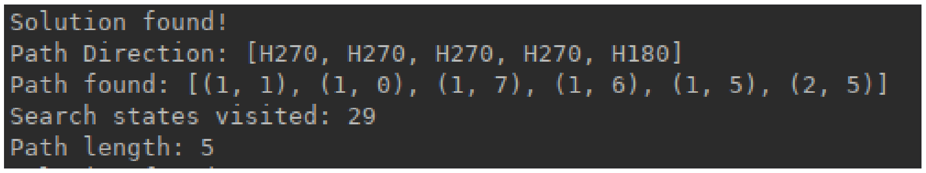
```
Solution found!
Path Direction: [H270, H270, H270, H270]
Path found: [(6, 6), (6, 5), (6, 4), (6, 3), (6, 2)]
Search states visited: 32
Path length: 4
Solution found!
Path Direction: [H270, H270, H270, H270]
Path found: [(6, 6), (6, 5), (6, 4), (6, 3), (6, 2)]
Search states visited: 5
Path length: 4
Solution found!
Path Direction: [H270, H270, H270, H270]
Path found: [(6, 6), (6, 5), (6, 4), (6, 3), (6, 2)]
Search states visited: 5
Path length: 4
```

Figure 5: input 10: N=10 S=(7,270) G=(7,90)

The first Solution represents the BFS algorithm while the following 2 represent BestF and A* respectively. As we can clearly see and as expected, A* and BestF perform way better than DFS and BFS. We output the valid path for all 3 algorithms here and can consider this a success.

circular motion

For this test, we want to check that if we have a node that is at the angle 0 and wants to head WEST H270, it is able to do so and it should transition to a node on the angle 315 represented by 7 in my code:



```
Solution found!
Path Direction: [H270, H270, H270, H270, H180]
Path found: [(1, 1), (1, 0), (1, 7), (1, 6), (1, 5), (2, 5)]
Search states visited: 29
Path length: 5
```

Figure 6: input for circular: N=5 S=(2,45) and G=(3, 225)

As we can see, the plane is heading towards H270 as expected and we got from node (1, 0) to node (1, 7). This behaviour is exactly what we are looking for!

obstacles

Now placing random obstacles on the map with the same inputs. How will the search adapt? We manually place an obstacle like so: map[1][7] = ’X’;
The map then looks like this:

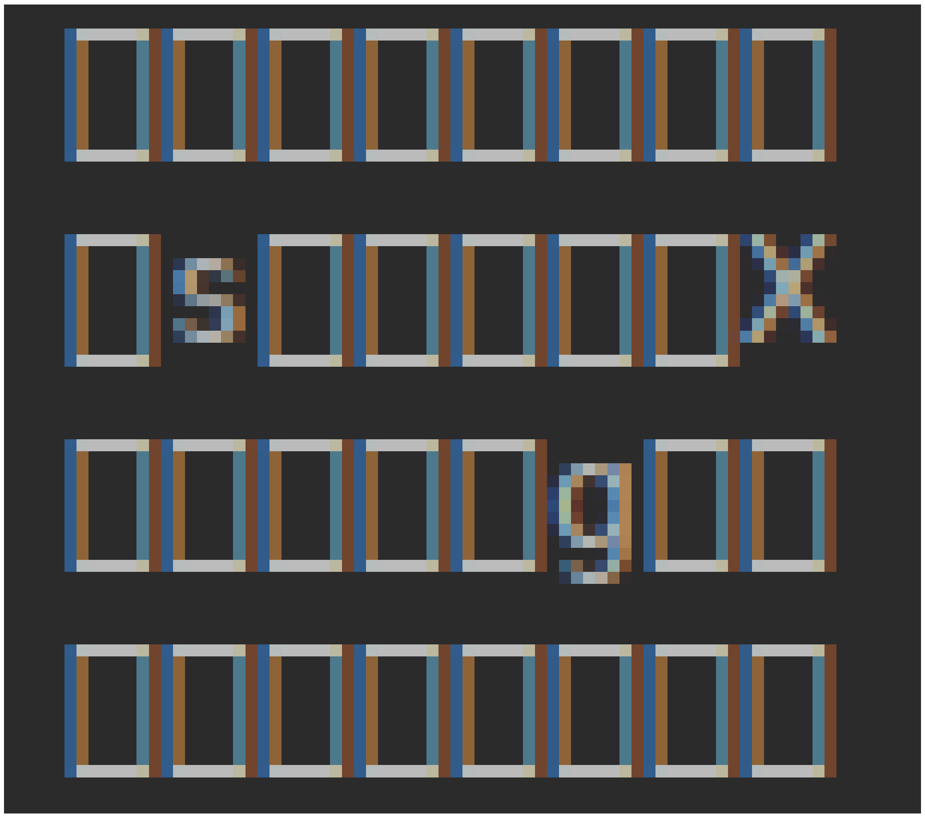
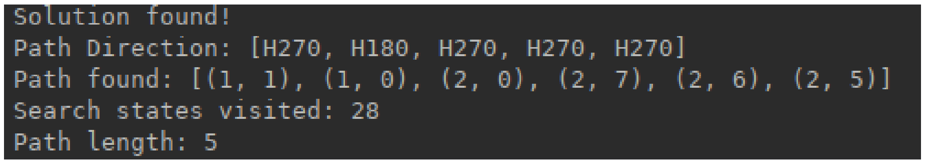


Figure 7: input for circular: N=5 S=(2,45) and G=(3, 225) and obstacle at [1][7] = (2, 315)

The search should allow the plane to no hit the obstacle but rather avoid it!



```
Solution found!
Path Direction: [H270, H180, H270, H270, H270]
Path found: [(1, 1), (1, 0), (2, 0), (2, 7), (2, 6), (2, 5)]
Search states visited: 28
Path length: 5
```

Figure 8: input for circular: N=5 S=(2,45) and G=(3, 225) and obstacle at [1][7] = (2, 315)

As we can see, Avoiding the obstacle is a success! instead of going to (1, 7), it passes to the parallel higher at (2, 0) and continues the normal search.

5 Conclusion

I am more than happy with the Project I submitted. However, I am heavily frustrated with the bug in my DFS algorithm. Indeed, DFS's implementation is faulty and doesn't work as intended even if the logic behind it looks correct. I have tried to explain the ides behind the implementation though and I hope it will not negatively impact this piece of work in a dramatic way!