CS5011 P1

160004864

11 October 2019

1 Introduction

For this practical we were required to implement a simplified flight route planner to navigate the imaginary constellation of Oedipus. Each of these planets is a circular grid, consisting of 8 meridians, one every 45 degrees, and a fixed number of parallels indicating the size of the planet. An intelligent agent is given a starting point and a goal point on this grid and it makes use of search algorithms to identify a route for an aircraft. Each of these algorithms follows the general search algorithm presented in the lectures[1] with minor changes depending on their particular nature.

1.1 Implementation Checklist

For this assignment, I've implemented the following features:

1. I've implemented a basic agent which makes use of uniformed search algorithms, namely depth-first search (DFS) and breadth-first search (BFS), following the general algorithm for search.

2. I've implemented an intermediate agent which makes use of informed search algorithms, namely best-first search (BestF) and A* (AStar) search, following the general algorithm for search.

3. I've implemented an advanced agent which makes use of a breadth-first bidirectional search algorithm, a variant of the aforementioned BFS algorithm.

4. I have implemented a feature giving the flight route planner the ability to account for the effect of wind on the aircraft. These winds blow in a certain direction, either North, South, West or East. This wind helps an aircraft moving in its direction but it makes it harder for the aircraft to move against it.

5. I have written bash scripts automating the process of invoking the route planner. The output is redirected into text files, depending on the algorithm used and whether there are winds to be accounted for. Each

algorithm is given the same number (20) of runs with the same starting and goal points so that we are able to make a fair comparison.

6. I have written a Python script, making use of the PlotLy module[2], which reads the data produced by the invocations of the route planner program, and uses it to create an interactive visualisation in the form of a scatter plot of nodes expanded vs path cost for each search algorithm. This makes the process of comparing the different search algorithms easier.

1.2 Compiling and running instructions

The base directory is the P1 directory. The source code (.java files) can be found in the P1/A1src directory while the class files (.class files) may be found in the P1/out/production/P1 directory. To run the flight route planner with your own parameters navigate to the base directory and run the command:

java -cp out/production/P1 A1main <DFS|BFS|AStar|BestF|BiDir>
<N><d_s,angle_s><d_g,angle_g>[N|S|E|W]

However one may also navigate to the source code directory, P1/A1src, compile the .java files using: *javac *.java* and then run the program as required by the specification using:

java A1main <DFS|BFS|AStar|BestF|BiDir><N><d_s,angle_s><d_g,angle_g>[N|S|E|W]

Example: *java A1main BFS 5 2,45 3,225 N*

To execute the scripts automating the run, navigate to the base directory /P1 and run the command: ./collect_data_linux.sh for running the program 20 times for each search algorithm without the wind feature, and run ./collect_data_wind_linux.sh for running the program 20 times for each search algorithm with the wind feature. The above scripts will redirect their output to text files so one may carefully examine and compare how different search algorithms carry out search. For example the route planner using the BFS algorithm, without the wind feature will redirect its output in a file called bfs_run.txt and the route planner using the BFS algorithm, with the wind feature will redirect its output in a file called bfs_wind _wind.txt. Running the script also automatically runs the Python script which reads its data from the data.txt file which contains the results of each search algorithm's performance. In turn, this python script will produce an interactive visulisation as well as a text file called results.txt or results_wind.txt containing the average path cost and average of nodes expanded for each search algorithm. Note that the bash scripts and the Python file may need to be given execution privileges using *chmod +x file*.

2 Design Implementation and Evaluation

2.1 Design and implementation

2.1.1 PEAS Model

- Performance measure: This will be the cost of the path taken from start to reach the goal. The number of nodes expanded by the search algorithm to reach a goal can be thought of as a secondary measure.
- Environment:The agent is acting in is the 2-d circular plane. The environment can also be characterised as deterministic, fully observable, static and known. This is because the next state is completely determined by the previous state and the next action, the agent is able to access a complete state of the environment which does not change while the agent is deciding what to do and the agent knows what the outcome of its actions will be.[1]
- Actuators: These are the actions the agent can make which is moving in one of either, North, South, West or East.
- Sensors: The agent can perceive where it currently is on the grid, what the possible moves are and whether the position it is in the goal position.

2.1.2 Search Components

- State space: The possible states are the positions on the grid.
- Actions: The aircraft can either move, North, South, East and West but it may not move past the last parallel of the grid and it may not reach or fly past the pole.
- Initial State: This is a point on the circular grid, passed as a parameter in the program e.g. 2,225
- Goal: This is a state on the grid passed as a parameter in the program e.g. 4,45.
- Search: An algorithm searches through the state space until a goal is found. This is done differently depending on the search algorithm used.

2.1.3 Input validation

The program carries out input validation by checking if the right number or arguments are passed to it. If not, an InvalidArgumentsException is thrown. It also checks whether the goal state is reachable. If unreachable, the program throws an 'ImpossibleFlightException'.

2.1.4 States and Nodes

States are positions on the grid, represented by objects with two properties, the level i.e. the parallel on which they lie and the 'bearing' which for this practical it will be used interchangeably with 'meridian'. Nodes are objects used by the algorithms in their search through the state space. Each node has a state, a predecessor node, a depth (depth in the search tree), a path cost i.e. the cost of reaching that node and a heading holding a label representing the action that was taken to get there.

2.1.5 Grid

The circular grid is represented using a 2d array of State of N-1 rows and 8 columns (one for each meridian), with the additional complexity that a move to the East at meridian 315, for example, will lead to the agent being in meridian 0. The rows 'wrap around' and by doing so we can represent the ability of the aircraft to move East or West, going in circles if done for sufficient steps. Since the aircraft can't reach the pole and can't move past the last parallel, the columns do not wrap around in the same way.

2.1.6 Setting up

After processing the input, the program instantiates the grid and populates it with states. Furthermore it instantiates the frontier, represented as a Linked List, into which the starting node is added, kick-starting the search process. A list is also instantiated with the purpose of holding the nodes already expanded, to prevent a search algorithm from expanding the same nodes over and over i.e. loops.

2.2 Search Strategy

The search strategy carried out is described by the general search algorithm[1]. All the algorithms use the same principles, in that they begin by creating the initial node, adding it to the frontier and then enter a loop in which a node, the 'current node', is taken from the frontier and it is expanded. A successor function finds the possible moves from the current node's position and the nodes representing these moves are added to the frontier. The loop continues until the goal is found or until the frontier is empty which means that the path from the start to the goal could not be found. The subtle difference between the search algorithms, mainly have to do with the way a node is chosen from the frontier.

2.2.1 Successor function

This function is implemented in the 'getPossibleMoves' method, which takes in the node as a parameter and it returns a list of the moves that are reachable from its state. The possible moves are the ones which do not take the agent to the pole, or beyond the last parallel.

2.2.2 Breadth First Search

This search algorithm carries out the strategy described above and its nature stems from the fact that the nodes are expanded 'shallowest first'. It considers the frontier as a FIFO queue, that is the nodes are expanded in the order they were added in the frontier. As a result the nodes are expanded in the order of increasing depth.

2.2.3 Depth First Search

Like BFS this algorithm carries out the general search strategy but it differs in that the nodes are expanded 'deepest first'. It considers the frontier as a LIFO stack meaning it expands the node, most recently added to the stack. As a result the algorithm will go on to expand nodes in a path until no new possible moves can be made, before moving to another branch.

2.2.4 Best First

This is an informed search algorithm which while it also carries out the general search strategy, it chooses the node to be expanded using a heuristic f(n). This heuristic is the Euclidian distance in polar coordinates from the node to the goal. This is $\sqrt{d_a^2 + d_b^2 - 2d_a d_b cos(a_b - a_a)}$. The node with the lowest distance to the goal is chosen to be expanded. This distance is an estimate which we shall call h(n). As a result the heuristic is f(n) = h(n).

2.2.5 A*

Similar to best first, this is another informed search strategy using a heuristic f(n). However this algorithm uses another component in addition to h(n), namely the path cost to reach the node, which we shall denote by g(n). The node with the lowest f(n) = g(n) + h(n) will be chosen to be expanded.

2.2.6 Tie breaking strategy

Scoring the nodes using a heuristic used means that often we will have nodes with the same value. The tie-breaking strategy I implemented is that in the case of a tie the frontier is treated as a FIFO queue. Between the tied nodes, the node that was added first is chosen.

2.2.7 Bidirectional Breadth First

This uninformed search algorithm is a variant of the BFS algorithm and while it follows the general search strategy as well, it searches for the path in both directions, from start to goal and from goal to start at the same time. In essence it performs two breadth first searches and it terminates when the two frontiers, intersect, indicating that a path from start to goal has been found.

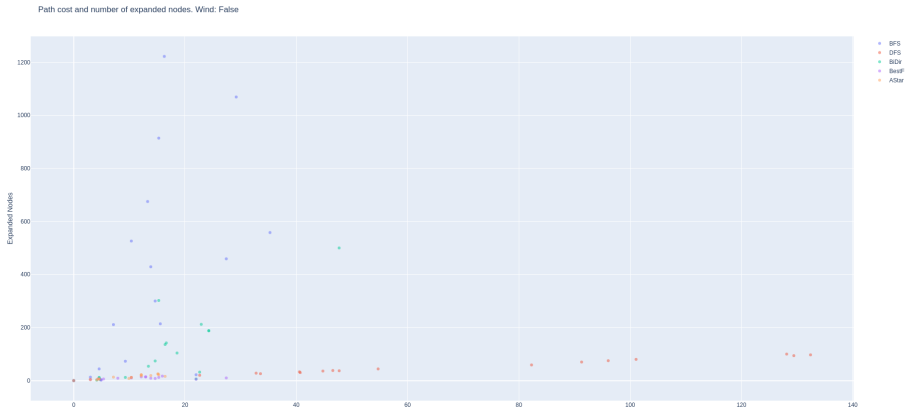
2.3 Wind

The wind direction affects the path cost of each move. If a move is in the direction of the move, the cost of that move is cut in half, while if it is against the wind's direction it is doubled. As a result, moves moving in the direction of the wind as much as possible would be favourable.

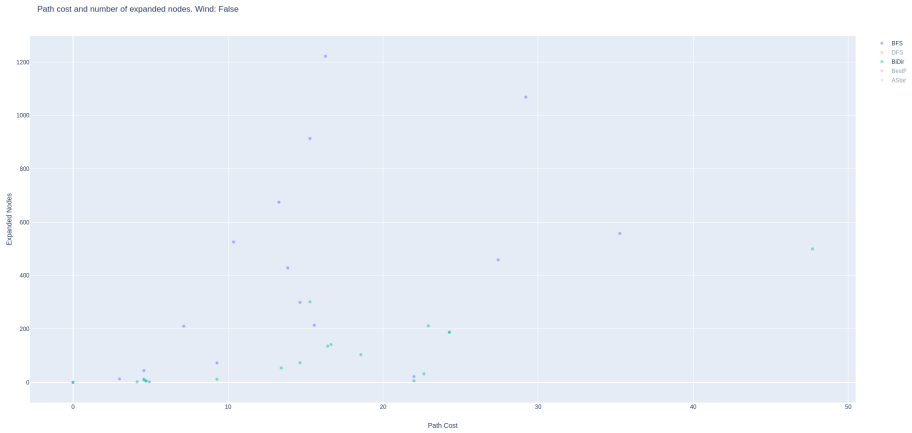
2.4 Evaluation

The comparison of the different search algorithms can be seen from the scatter plot (images can be found in P1/images) generated by the automation scripts:

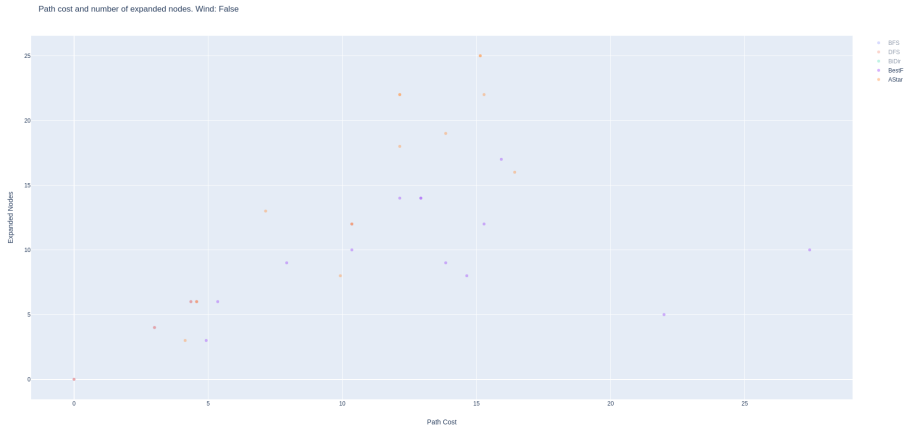
2.4.1 Evaluation on search without wind



It is obvious to see that there are some significant differences in behaviour. BFS expands the most nodes, while DFS will find paths with high cost but it won't expand as many nodes. This makes sense as BFS will find the shallowest node in the search tree, but it has to expand all the nodes that have a depth lower that the goal's.
The following image, shows that bidirectional search solutions have a similar path cost to BFS's but the number of expanded nodes is significantly lower. This is expected at each depth BFS, for a branching factor b, the number of nodes at depth d will be b^d. However in bidirectional search the number of nodes expanded will be $b^{\frac{d}{2}}$ for each part of the search.



When compared to the other search algorithms Best First and A* are clearly superior, having solutions with far fewer nodes expanded and a much lower path cost. However they have some differences between themselves:



The A* search algorithm which uses a better heuristic provides solutions with a lower path cost, at the cost of having expanded a few more nodes.
The graph observations are confirmed by the results in the results.txt text-file which are as follows:

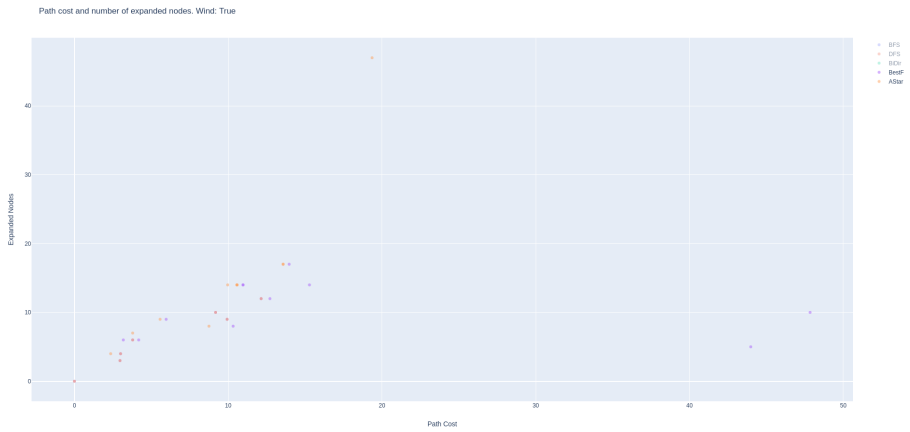
- BFS Cost Average: 13.71
- DFS Cost Average: 62.65
- BiDir Cost Average: 15.89
- BestF Cost Average: 11.0
- AStar Cost Average: 9.48
- BFS Nodes Average: 374.28

- DFS Nodes Average: 48.39
- BiDir Nodes Average: 109.89
- BestF Nodes Average: 8.83
- AStar Nodes Average: 13.28

Note on results: The reason why the cost Bidirectional Search solutions are higher that BFS's is that there might be other intersections in the frontier, but I haven't implemented a way of finding the intersection node with the lowest path cost.

2.4.2 Evaluation of search with wind

Since only A* uses the path cost (affected by the wind direction) as part of the heuristic, this is the only algorithm that will be able to use it to its benefit. To confirm this, I will be comparing A* with Best First on runs with an Eastward wind.



The averages in this case, found in the results_wind.txt text-file are as follows:

- BestF Cost Average: 12.24
- AStar Cost Average: 8.31
- BestF Nodes Average: 8.83
- AStar Nodes Average: 11.61

As we can see, with the effect of the wind, A*'s paths have a decreased average cost than without it, while Best First's paths have an increased cost. Furthermore, A*'s number of expanded nodes have decreased while Best First's have remained the same.

3 Test Summary

In order for one to assess the correctness of the system, they must follow the output which the search algorithms produce, assess whether the right node is chosen from the frontier and to verify that the path given in the solution is valid. The automation scripts produce a text file for each algorithm, containing the output at each step of the algorithm which includes the current node, the list of states expanded, the frontier at each step in the search as well as the final solution details, as required. The first 8 out of the 20 runs the script invoked are the ones found in the Appendix of the specification and if followed it can be seen that the program indeed carries out the search as expected. Note however that, for the runs whose goal parameter is set at the pole, an exception is thrown as follows:

```
am425@pc7-118-l:~/CS5011/P1 $ java -cp out/production/P1 A1main DFS  8 6,270 0,45
Inputs are the following

Search algorithm used: DFS
Number of parallels: 8
Starting state: 6,270
Goal state: 0,45

Exception in thread "main" ImpossibleFlightException: Impossible flight: Can't reach pole.
        at A1main.main(A1main.java:629)
am425@pc7-118-l:~/CS5011/P1 $
```

Furthermore, if the wrong number of arguments is passed to the program, an InvalidArguments exception is also thrown:

```
am425@pc7-118-l:~/CS5011/P1 $ java -cp out/production/P1 A1main DFS  8 6,270
Exception in thread "main" InvalidArgumentsException: Invalid number of arguments
        at A1main.main(A1main.java:601)
am425@pc7-118-l:~/CS5011/P1 $
```

```
am425@pc7-118-l:~/CS5011/P1 $ java -cp out/production/P1 A1main DFS  8 6,270 2,225 N E
Exception in thread "main" InvalidArgumentsException: Invalid number of arguments
        at A1main.main(A1main.java:601)
am425@pc7-118-l:~/CS5011/P1 $
```

3.1 Testing Heuristics

In order to convince the reader of the correctness of the heuristics used in the informed search algorithm without them having to calculate the Euclidian distance on the polar plane (a time consuming process), I modified the code to print the value of the heuristics used and show that the program chooses the next node to be expanded correctly.
In the case of the BestF algorithm:

```
From: 2,270 Distance to goal: 5.60
From: 3,225 Distance to goal: 5.00
From: 1,225 Distance to goal: 4.12
From: 3,180 Distance to goal: 2.83
From: 1,180 Distance to goal: 3.37
From: 2,180 Distance to goal: 2.95
From: 2,90 Distance to goal: 2.95
From: 3,135 Distance to goal: 1.00
From: 1,135 Distance to goal: 3.00
Current node to be expanded: State: 3,135 Predecessor state: 2,135 Depth: 3 Path Cost: 4.14
```

As you can see, the node with the smallest distance to the goal (4,135 in this case) is chosen to be expanded next.
In the case of the AStar algorithm:

```
Current node: 2,270 path cost: 1.57 total heuristic distance: 7.17
Current node: 3,225 path cost: 1.00 total heuristic distance: 6.00
Current node: 3,180 path cost: 2.57 total heuristic distance: 5.40
Current node: 1,180 path cost: 2.57 total heuristic distance: 5.94
Current node: 1,270 path cost: 1.79 total heuristic distance: 6.55
Current node: 1,180 path cost: 1.79 total heuristic distance: 5.15
Current node: 2,180 path cost: 4.71 total heuristic distance: 7.66
Current node: 2,90 path cost: 4.71 total heuristic distance: 7.66
Current node: 3,135 path cost: 4.14 total heuristic distance: 5.14
Current node: 1,135 path cost: 4.14 total heuristic distance: 7.14
Current node to be expanded: State: 3,135 Predecessor state: 2,135 Depth: 3 Path Cost: 4.14
```

As one can see the node chosen from the frontier to be expanded is the one with the lowest heuristic distance. Note that the goal is 4,135 as above.

4 Conclusion

Overall, I'm very happy with the result of my work as I've managed to implement the basic, intermediate and parts of the advanced agent and I have also managed to write scripts which automate the mundane process of running the program with different parameters and gathering data. The automation also allowed me to gather enough data to produce an insightful visualisation which I generate using a Python script. This practical helped me understand the different search algorithms and it gave me an insight to how they are different in terms of their implementation and their performance. The most challenging part was the implementation of the bidirectional search as I found it tricky to form a path in 'reverse', from the goal to the point of intersection of the two frontiers. If I had more time, I'd attempt the suggested advanced requirement which requires the scheduling of the two wing-man airplanes on either side of the main one.

References

[1] Toniolo, A. (2019). CS5011(L2 W1): Introduction, Introduction to Search.

[2] PlotLy https://plot.ly/

[3] Directions and Bearings. (2019). Retrieved 11 October 2019, from https://www.mathsteacher.com.au/year10/ch15_trigonometry/