CS5011 Artificial Intelligence Practice
Practical 1 Report

Student ID: 150014151

11 October, 2019

1 Introduction

The Advanced Agent was attempted for this Practical. The following features were implemented:

- Basic Agent:
  - Breadth-First Search
  - Depth-First Search
- Intermediate Agent:
  - Best-First Search
  - A* Search
- Advanced Agent (1 extension):
  - Weather obstacles

1.1 Usage

1.1.1 Compilation

To compile the program, navigate to the *A1src* directory and run the following command:

javac src/A1Main.java.

1.1.2 Program Execution

Once the program has been compiled, it can be executed using the following command:

java A1Main <search_type><world_size><start_goal><end_goal>[<obstacles>],

where:

- *search_type* is the type of search algorithm used to find a route. It can take the following values: *BFS*, *DFS*, *BestF*, *AStar*. It is written as a String.
- *world_size* is the size of the world, specified by the number of parallels. It is written as a positive integer.
- *start_goal* is the starting point of the flight. It is written as a tuple of positive integers, e.g. *(2,45)*.
- *end_goal* is the goal point that the flight must reach. It is also written as a tuple of positive integers.
- *obstacles* is a number of locations in the world that the flight cannot take when looking for a route. They are also written as a tuple of positive integers. There can be any number of obstacles, ranging from 0 to the limit set by the Java Virtual Machine [4].

1.1.3 Examples

Here are a few examples that can be used to run the program:

- Running BFS with no obstacles: *"java A1Main BFS 5 2,45 3,225"*
- Running DFS with no obstacles: *"java A1Main DFS 8 1,315 5,270"*
- Running BestF with 1 obstacle: *"java A1Main BestF 4 1,45 3,225 1,90"*
- Running A* with 2 obstacles: *"java A1Main AStar 4 1,45 3,225 1,90 1,0"*
- Running BFS with no possible solution: *"java A1Main BFS 4 1,45 3,225 1,90 1,0 2,45"*

2 Design, Implementation & Evaluation

2.1 Design & Implementation

2.1.1 PEAS Model

Performance measure The path length from initial to goal state, the path cost, the number of states explored, the depth of the tree, the number of nodes created, and the runtime.

Environment A circular world of size N, which represents the number of parallels. Each parallel is divided in 8 meridians, ranging from 0 to 360 degrees, as depicted in Figure 1.

Actuators Moving in of the four following directions, as per Figure 1: East (H90), South (H180), West (H270) and North (H360).

Sensors The flight can see in any of the four aforementioned directions from a state in the world. It also knows if the move is valid based on a set of rules specified in Section 2.1.2.

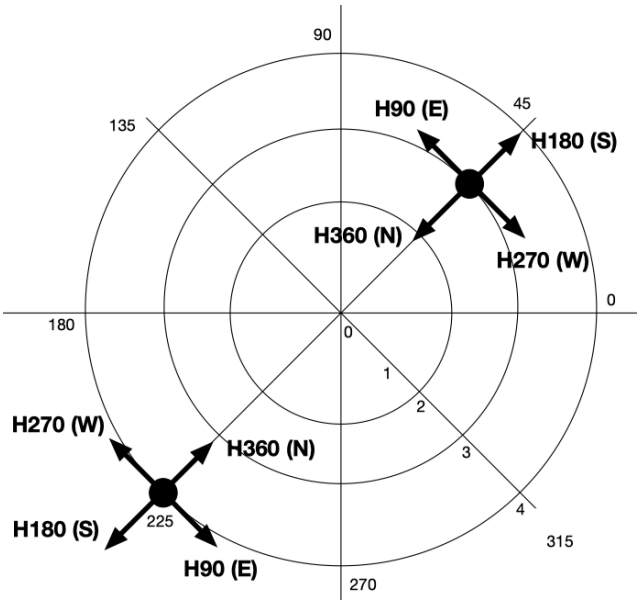


Figure 1: A visualisation of a state space and its valid moves where $N = 5$.

2.1.2 Problem Definition

State space Corresponds to each position in the map. Each position is represented by a State object, as shown in the UML Class diagram in Appendix A. The state space as a whole is implemented with a *LinkedList* of *LinkesLists* of *Nodes*: *"LinkedList¡LinkedList¡State¿¿ world"*, as shown in Figure 2.

```
[
    [(0, 0)=P]
    [(1, 0)=E, (1, 45)=E, (1, 90)=E, (1, 135)=E, (1, 180)=E, (1, 225)=E, (1, 270)=E, (1, 315)=E]
    [(2, 0)=S, (2, 45)=E, (2, 90)=E, (2, 135)=G, (2, 180)=X, (2, 225)=E, (2, 270)=E, (2, 315)=E]
    [(3, 0)=E, (3, 45)=E, (3, 90)=E, (3, 135)=E, (3, 180)=E, (3, 225)=E, (3, 270)=E, (3, 315)=E]
    [(4, 0)=E, (4, 45)=E, (4, 90)=E, (4, 135)=E, (4, 180)=E, (4, 225)=E, (4, 270)=E, (4, 315)=E]
]
```

Figure 2: An example of the implementation of a state space using *LinkedLists* where $N = 5$. The initial state S can be seen at *2,0*, the goal state G at *(2,135)*, an obstacle X at *(2,180)* and the pole P at *(0,0)*.

Initial State Corresponds to the starting point defined by the user input, which is used to create the root node of the search tree. For example, in the implementation in Figure 2, the initial state can be found at *(2, 0)=S*. The goal state is stored in the Problem object, as shown in Appendix A. It is impossible to start from the pole (0,0).

Goal To find the route from the initial state to the goal state while avoiding obstacles. For informed search algorithms, the goal includes finding the route with the minimal path cost. It is impossible for a goal state to be in the pole (0,0).

Successor Function The successor function (see listing below) generates a set of all the valid nodes that are adjacent to the current node, which are added to an *ArrayList* before being added to the frontier. Additionally, it checks that no invalid moves are added to the *ArrayList*.

```java
public ArrayList<State> successor(State state, Problem p, ArrayList<State> o) {
    ArrayList<State> childrenNodes = new ArrayList<>();

    // Move East (H90). If angle = 315 degrees, loop back to 0 degrees.
    if (state.getAngle() == 315) {...}
    else {...}

    // Move West (H270). If angle = 0 degrees, loop back to 315 degrees.
    if (state.getAngle() == 0) {...}
    else {...}

    // Move North (H360), but cannot go to pole (N=0).
    if ( !(state.getD() == 0) ) {...}

    // Move South (H180), but cannot go beyond last parallel N.
    if ( !(state.getD() == problem.getN() - 1) ) {...}

    // Cannot go to pole (0, 0).
    for (State s: childrenNodes) {...}

    // Cannot go to obstacles.
    for (State a: o) {...}

    return childrenNodes; // Return valid moves.
}
```

Actions The flight can move in one of four directions at a time (see Figure 1). It can move East (H90) by adding 45°to its angle, move West (H270) by subtracting 45°from its angle, move North (H360) towards the pole (0,0) by subtracting 1 to its current parallel, or move South (H180) towards the extremity of the world by adding 1 to its current parallel. It cannot move to/through the pole *(0,0)*, to a parallel bigger than N, or to an obstacle.

Path Cost Because the state space is a circular map, there are two different costs. The cost to move across parallels ($N \rightarrow N+1$ or $N \rightarrow N-1$) is 1, while the cost to move across meridians ($+45°$ or $-45°$) is $(2 * \pi * d)/8$. The cost is stored in the *pathCost* of the *Node* class.

2.1.3 System Architecture

Class Design The system is designed with scalability and simplicity in mind. Therefore, the foundation of the system was built around the general search algorithm (see Figure 3) provided in the lecture slides [6].

```
function TREE-SEARCH(problem,frontier) returns a solution, or failure
initial node ← MAKE-NODE(null,initial state)
frontier ← INSERT( initial node, frontier)
explored ← empty set
    loop do
    if frontier is empty return failure
    nd ← REMOVE(index, frontier)
    Add nd to explored
    if GOAL-TEST(STATE[nd], goal)
        return nd
    else
        frontier ← INSERT-ALL (EXPAND(nd, problem, frontier, explored ))
    end loop
end
```

Figure 3: The general search algorithm [6]

Because the search algorithms implemented share many similarities with the general search algorithm, abstract classes are used to avoid unnecessary code duplication by grouping common methods between the different search algorithms. An overview of the class design can be found in the UML Class Diagram in Appendix A. A top-level abstract class called *GeneralSearch* organises all the general search methods such as *successor()*, *goalTest()*, *isNodeInExploredSet()* or *findSolutionPathCost()* into a single class. Abstract methods, such as *makeNode*, are defined in this class and act as a template to implement in classes that extend *GeneralSearch*.

Next, two abstract classes, *InformedSearch* and *UninformedSearch*, extend *GeneralSearch* and its methods. In a similar fashion, they contain methods common to all concrete search algorithms, such as *treeSearch*, *expand* and *removeFrontierNode*. They also override the *makeNode* method declared in the *GeneralSearch* abstract class, and declare their own abstract meth-

ods that will be extended by the concrete classes.

Finally, concrete classes extending the abstract classes are created. The *BFS* and *DFS* classes extend the *UninformedSearch* abstract class, while the *BestF* and *AStar* classes extend the *InformedSearch* abstract class.

Data Structures Java offers a wide variety of data structures to use. The pros and the cons were weighed to choose which data structure to use for each aspect of the program. Concerning the frontier, two data structures are used. For uninformed search, a *LinkedList* is used, while for informed search, a *PriorityQueue* is used. *LinkedLists* are used because manipulation operations for adding/removing Nodes to/from the data structure are much cheaper than *ArrayLists* manipulations [3].

Additionally, custom classes are created to easily store and access different aspects of the data. A *Node* object holds crucial information such as:

- A pointer to a parent Node, which is used to retrace the route found from the current node (once the goal state is reached) all the way back to the root node. It is also used to calculate the route's cost.
- An action (H90, H180, H270, H360).
- A path cost.
- A depth.
- A *State* (custom class).

States represent a location on the state space, including a parallel, an angle, an index to easily access it in the world, and a status to represent if the location is the initial state (S), the goal state (G), the pole (P) or an obstacle (X) (see Figure 1).

2.1.4 Uninformed Search

The two uninformed search algorithms implemented are Breadth-First Search and Depth-First Search. Observing their concrete implementations, they both extend the *UninformedSearch* abstract class, which in turn extends the *GeneralSearch* abstract class. This is due to the fact that both algorithms are almost identical. Indeed, they only differ in terms of order in which the successor nodes are added to the frontier, as shown in the code listing below. BFS implements a FIFO-type[1] of queue, where the nodes are inserted at the end of the frontier, whereas DFS implements a LIFO-type[2] of queue, where the nodes are inserted at the front of the frontier.

```
1 public class BFS extends UninformedSearch {
2     @Override
3     public LinkedList<Node> insertFrontierNodes(LinkedList<Node> frontier,
    ArrayList<Node> nodes) {
```

[1]First-In First-Out
[2]Last-In First-Out

```
        frontier.addAll(nodes); // Add nodes at the end of the queue (FIFO).
        return frontier;
    }
}

public class DFS extends UninformedSearch {
    @Override
    public LinkedList<Node> insertFrontierNodes(LinkedList<Node> frontier,
    ArrayList<Node> nodes) {
        frontier.addAll(0, nodes); // Add nodes at the end of the queue (LIFO).
        return frontier;
    }
}
```

Because uninformed search algorithms can be classified as "blind searches" as they do not calculate cost paths and heuristics to determine the next move, new states are added in the following default order: East/West/North/South. Following this default order ensures consistency across all evaluations of the algorithms in Section 2.2.

2.1.5 Informed Search

The two uninformed search algorithms implemented are Best-First Search and A* Search. In a similar fashion to the informed search algorithms, there is very little difference between the two implementations. Both use *PriorityQueues* as a frontier, and retrieve the next highest-priority item from the frontier using the *frontier.poll()* method, and both extend the *InformedSearch* abstract class, which in turn extends the *GeneralSearch* abstract class. To calculate the distance from the current node to the goal node, Euclidian distance in polar coordinates is used [5]:

$$\sqrt{d_A^2 + d_B^2 - 2 \cdot d_A \cdot d_B \cdot cos(\alpha_A - \alpha_B)} \tag{1}$$

The two algorithms only differ in their heuristic and their implementation of the *extend()* method. Indeed, A*'s heuristic assigns the sum $f(n) = g(n) + h(n)$ of the path cost so far $g(n)$ and the Euclidian distance in polar coordinates from the current note to the goal $h(n)$, whereas Best-First Search only uses the path cost so far $f(n) = g(n)$. The difference can be seen in the listing below:

```
double heuristicH = estimateCostFromNodeToGoal(curNode.getState(), problem.
    getEndPoint());
if (problem.getSearchType().equals("AStar")) { // A* Search.
    double heuristicG = findSolutionPathCost(curNode);
    double heuristicF = heuristicG + heuristicH;
    node.setPathCost(heuristicF);
}
else { // BestF Search.
    node.setPathCost(heuristicH);
}

/* Estimates the Euclidian distance in polar coordinates from the current State
    to the goal State. */
double estimateCostFromNodeToGoal(State curState, State goalState) {
    double pow = Math.pow(curState.getD(), 2) + Math.pow(goalState.getD(), 2);
```

```
    double cos = 2 * curState.getD() * goalState.getD() * Math.cos(goalState.
    getAngle() - curState.getAngle());
    return Math.sqrt(pow - cos);
}
```

A* differs from Best-First Search as well by removing Nodes in the frontier that have a higher cost than new Nodes, as betrayed in the listing below. This way, it ensure to find cheaper paths.

```
@Override
public ArrayList<Node> expand(Node node, Problem problem, PriorityQueue<Node>
    frontier, ArrayList<Node> exploredSet, ArrayList<State> obstacles) {
    ArrayList<Node> successorsSet = new ArrayList<>();
    ArrayList<State> nextStates = successor(node.getState(), problem, obstacles
    );

    for (State state: nextStates) {
        Node newNode = makeNode(node, state.getD(), state.getAngle(), problem);
        if (isNodeInQueueFrontier(frontier, state)) {
            for (Node n: frontier) {
                if (newNode.getPathCost() < n.getPathCost()) {
                    frontier.remove(n);
                    break;
                }
            }
        }
        if (!(isNodeInQueueFrontier(frontier, state)) && !(isNodeInExploredSet(
    exploredSet, state))) {
            successorsSet.add(newNode);
        }
    }
    return successorsSet;
}
```

2.1.6 Additional Features

Invalid Input Security Robust checks are carried out to ensure that correct arguments are passed when running the program. These include grounding the flight if the initial state S = (0,0) or if the goal state G = (0,0)[3].

Javadocs The entire system is covered with Javadoc [1] comments. They can be compiled using the command below and opening the *javadoc/index.html* file in a web browser:

javadoc -d javadoc A1src/.java*

2.2 Evaluation

Starting with the evaluation of uninformed search algorithms, the major difference lies within the length of the path found, as seen in Figure 4.

[3]This includes any meridian located on parallel $d = 0$ e.g. (0,45) is an invalid location.

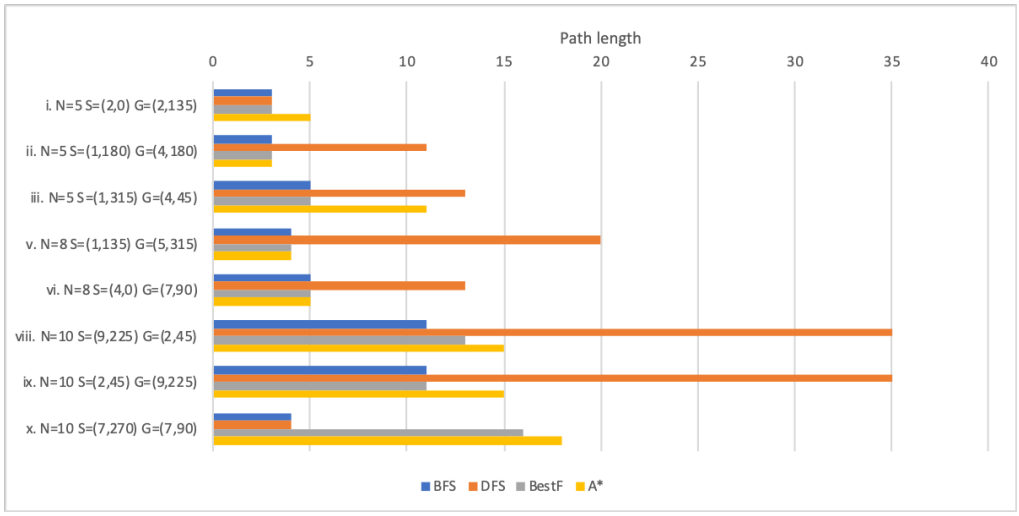


Figure 4: The number of nodes created for different algorithms. See Appendix B for detailed results.

Indeed, DFS often finds the longest path possible of all the algorithms, which leads to it having the longest path cost as well (see Figure 5). This is due to the fact that DFS favours depth by exploring each child node exhaustively until it reaches a leaf node: when a goal state is found, it may not be the optimal state. On the other hand, BFS explores each node on the current depth of the search tree before going one level deeper. Therefore, it is more likely to find an optimal path because once a goal state is found, all the shallower nodes have already been explored.

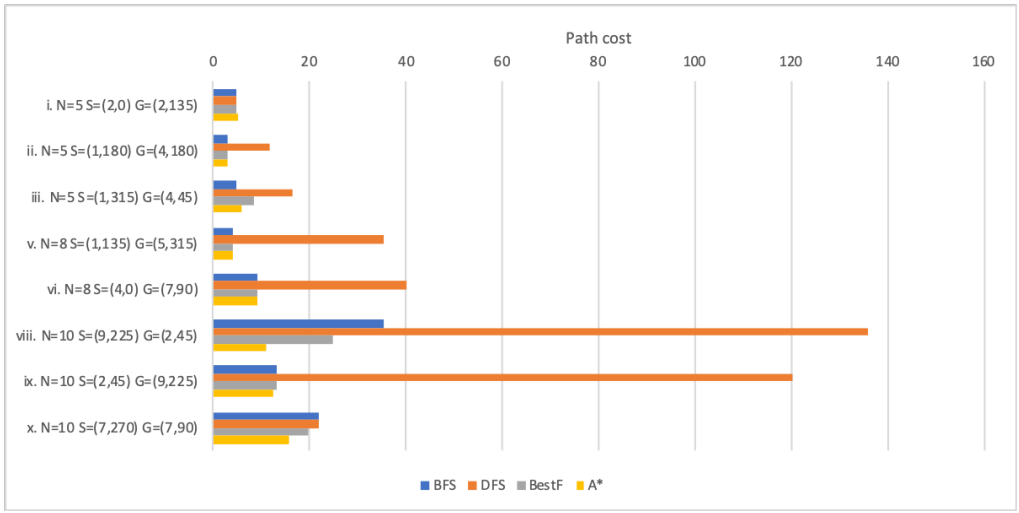


Figure 5: The path costs of different algorithms. See Appendix C for detailed results.

As a result of this, DFS often expands less nodes than BFS (see Figure 6) and the goal node is much deeper than BFS's goal node (see Figure 7).

Looking at informed search algorithms, Best-First seems to expand less nodes than A*,

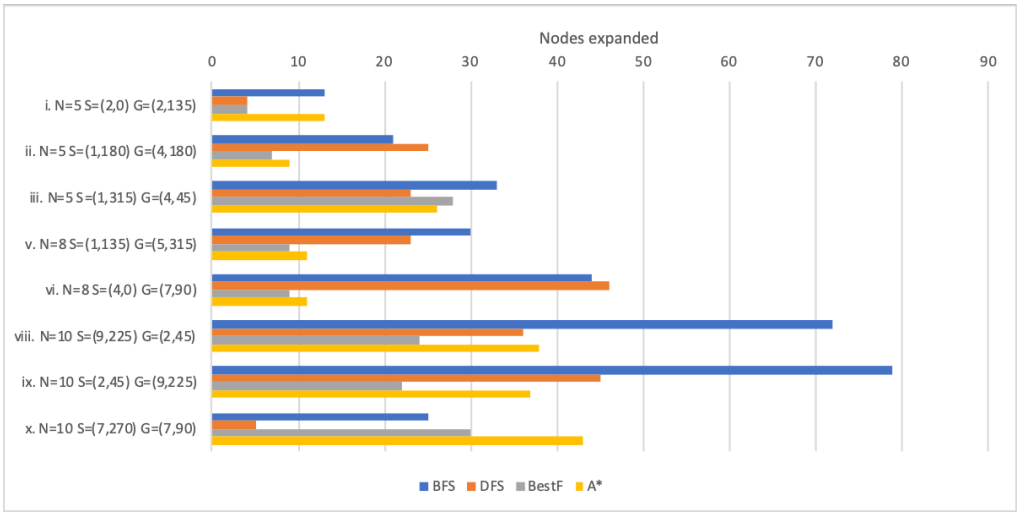


Figure 6: The number of nodes created for different algorithms. See Appendix D for detailed results.

as seen in Figure 6. This is due to the fact that Best-First's objective is to reach the goal state as quickly as possible without considering the cost of the current path. Therefore, A* may expand more nodes, but it is the algorithm that finds the cheapest path from initial to goal state of all algorithms, as seen in Figure 5. In large maps, path length does not matter as much as path cost, since A*'s path is longer than most but always cheaper (recall that crossing parallels is cheaper than crossing meridians, especially when d is big). Perhaps other distance functions could be used to improve the path cost, such as the Manhattan distance, or a combination of multiple distance metrics.

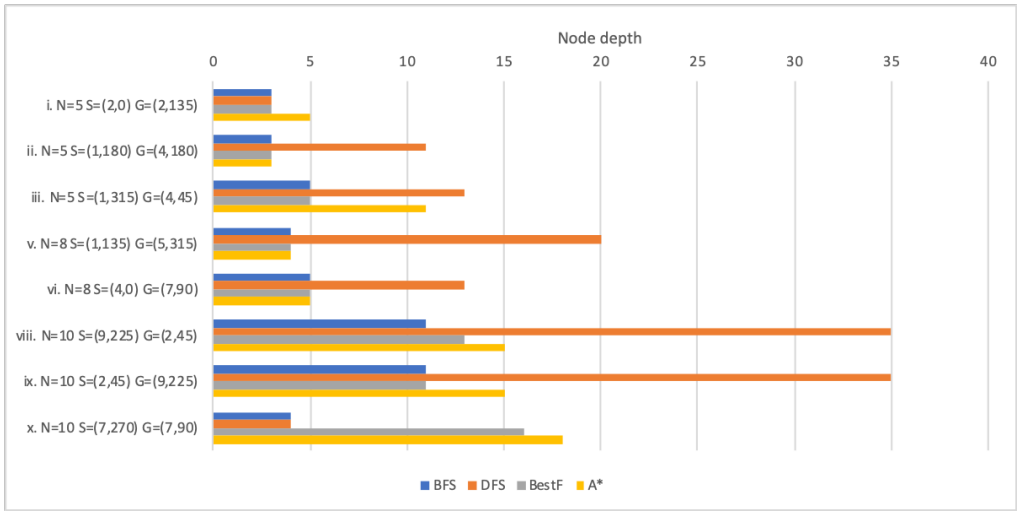


Figure 7: The node depth of different algorithms. See Appendix E for detailed results.

Run configurations *iv* and *vii* were excluded from the evaluation graphs since no path can be found, as the goal states are located at the pole ($d = 0$).

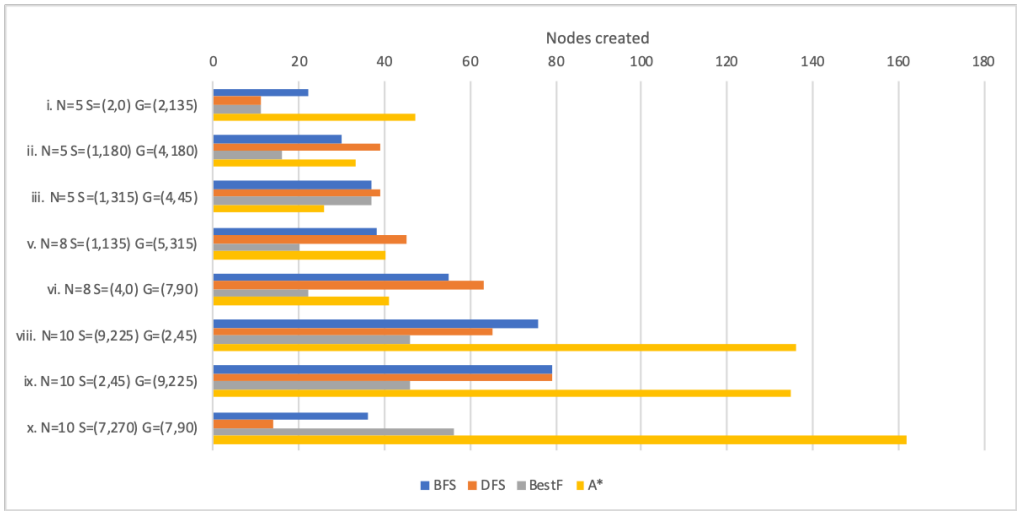


Figure 8: The number of nodes created for different algorithms. See Appendix F for detailed results.

Runtime was also calculated, but didn't prove to be relevant due to the small size of the problems used to evaluate the search algorithms. They can be found in Appendix G.

3 Test Summary

To test the algorithms, a simple problem is used, where:

- $N = 4$
- $S = (1,45)$
- $G = (2,315)$

3.1 Breadth-First Search

The expected flight path is *[H270,H270,H180]*. Current nodes being expanded should be added to the explored set, while new nodes should be added at the end of the frontier. The console output below meets the requirements, which shows that Breadth-First Search works as expected.

```
Search type: BFS
Size of world N: 4
Start point: (1, 45)
End point: (2, 315)
Obstacles located at: []

Starting Breadth-First Search...
Iteration #1 ------------------------
Current node: (1, 45)
Frontier: [(1, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45)]
```

```
Iteration #2 ----------
Current node: (1, 90)
Frontier: [(1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90)]
Explored States: [(1, 45), (1, 90)]
Iteration #3 ----------
Current node: (1, 0)
Frontier: [(0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0)]
Explored States: [(1, 45), (1, 90), (1, 0)]
Iteration #4 ----------
Current node: (0, 45)
Frontier: [(2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45)]
Iteration #5 ----------
Current node: (2, 45)
Frontier: [(1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45)]
Iteration #6 ----------
Current node: (1, 135)
Frontier: [(0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135)]
Iteration #7 ----------
Current node: (0, 90)
Frontier: [(2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90)]
Iteration #8 ----------
Current node: (2, 90)
Frontier: [(1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90)]
Iteration #9 ----------
Current node: (1, 315)
Frontier: [(2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90), (1, 270), (0, 315), (2, 315)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315)]
Iteration #10 ----------
Current node: (2, 0)
Frontier: [(3, 45), (1, 180), (0, 135), (2, 135), (3, 90), (1, 270), (0, 315), (2, 315), (3, 0)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0)]
Iteration #11 ----------
Current node: (3, 45)
Frontier: [(1, 180), (0, 135), (2, 135), (3, 90), (1, 270), (0, 315), (2, 315), (3, 0)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45)]
Iteration #12 ----------
Current node: (1, 180)
Frontier: [(0, 135), (2, 135), (3, 90), (1, 270), (0, 315), (2, 315), (3, 0), (1, 225), (0, 180), (2, 180)]
```

```
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180)]
Iteration #13 ------------------------
Current node: (0, 135)
Frontier: [(2, 135), (3, 90), (1, 270), (0, 315), (2, 315), (3, 0), (1, 225), (0, 180), (2, 180)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135)]
Iteration #14 ------------------------
Current node: (2, 135)
Frontier: [(3, 90), (1, 270), (0, 315), (2, 315), (3, 0), (1, 225), (0, 180), (2, 180), (3, 135)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135)]
Iteration #15 ------------------------
Current node: (3, 90)
Frontier: [(1, 270), (0, 315), (2, 315), (3, 0), (1, 225), (0, 180), (2, 180), (3, 135)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90)]
Iteration #16 ------------------------
Current node: (1, 270)
Frontier: [(0, 315), (2, 315), (3, 0), (1, 225), (0, 180), (2, 180), (3, 135), (0, 270), (2, 270)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90), (1, 270)]
Iteration #17 ------------------------
Current node: (0, 315)
Frontier: [(2, 315), (3, 0), (1, 225), (0, 180), (2, 180), (3, 135), (0, 270), (2, 270)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90), (1, 270), (0, 315)]
Iteration #18 ------------------------
Current node: (2, 315)
Frontier: [(3, 0), (1, 225), (0, 180), (2, 180), (3, 135), (0, 270), (2, 270)]
Explored States: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90), (1, 270), (0, 315), (2, 315)]

Path found using BFS in 4.776 ms!
Flight instructions (3) : [H270, H270, H180]
Path followed (4 states): [(1, 45), (1, 0), (1, 315), (2, 315)]
Solution path cost: 2.571

Current node depth: 3
Nodes created: 25
18 states expanded: [(1, 45), (1, 90), (1, 0), (0, 45), (2, 45), (1, 135), (0, 90), (2, 90), (1, 315), (2, 0), (3, 45), (1, 180), (0, 135), (2, 135), (3, 90), (1, 270), (0, 315), (2, 315)]
```

3.2 Depth-First Search

The expected flight path is different to *[H270,H270,H180]* since DFS is not optimal. Current nodes being expanded should be added to the explored set, while new nodes should be added at the front of the frontier. The console output below meets the requirements, which shows that Depth-First Search works as expected.

```
Search type: DFS
Size of world N: 4
Start point: (1, 45)
End point: (2, 315)
Obstacles located at: []

Starting Depth-First Search...
Iteration #1 ------------------------
Current node: (1, 45)
Frontier: [(1, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45)]
Iteration #2 ------------------------
Current node: (1, 90)
Frontier: [(1, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90)]
Iteration #3 ------------------------
Current node: (1, 135)
Frontier: [(1, 180), (0, 135), (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135)]
Iteration #4 ------------------------
Current node: (1, 180)
Frontier: [(1, 225), (0, 180), (2, 180), (0, 135), (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135), (1, 180)]
Iteration #5 ------------------------
Current node: (1, 225)
Frontier: [(1, 270), (0, 225), (2, 225), (0, 180), (2, 180), (0, 135), (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225)]
Iteration #6 ------------------------
Current node: (1, 270)
Frontier: [(1, 315), (0, 270), (2, 270), (0, 225), (2, 225), (0, 180), (2, 180), (0, 135), (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225), (1, 270)]
Iteration #7 ------------------------
Current node: (1, 315)
Frontier: [(0, 315), (2, 315), (0, 270), (2, 270), (0, 225), (2, 225), (0, 180), (2, 180), (0, 135), (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225), (1, 270), (1, 315)]
Iteration #8 ------------------------
Current node: (0, 315)
Frontier: [(2, 315), (0, 270), (2, 270), (0, 225), (2, 225), (0, 180), (2, 180), (0, 135), (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225), (1, 270), (1, 315), (0, 315)]
```

```
Iteration #9 ————————————————————
Current node: (2, 315)
Frontier: [(0, 270), (2, 270), (0, 225), (2, 225), (0, 180), (2, 180), (0, 135)
    , (2, 135), (0, 90), (2, 90), (1, 0), (0, 45), (2, 45)]
Explored States: [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225), (1, 270), (1,
     315), (0, 315), (2, 315)]

Path found using DFS in 3.335 ms!
Flight instructions (7) : [H90, H90, H90, H90, H90, H90, H180]
Path followed (8 states): [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225), (1,
    270), (1, 315), (2, 315)]
Solution path cost: 5.712

Current node depth: 7
Nodes created: 22
9 states expanded: [(1, 45), (1, 90), (1, 135), (1, 180), (1, 225), (1, 270),
    (1, 315), (0, 315), (2, 315)]
```

3.3 Best-First Search

The expected flight path is *[H270,H270,H180]*. Current nodes being expanded should be added to the explored set, while new nodes should be added at in the frontier[4] based on their estimate distance to goal. The console output below meets the requirements, which shows that Best-First Search works as expected.

```
Search type: BestF
Size of world N: 4
Start point: (1, 45)
End point: (2, 315)
Obstacles located at: []

Starting Best-First Search...
Iteration #1 ————————————————————
Current node: (1, 45)
Frontier: [(1, 90) 1.031, (1, 0) 1.031, (0, 45) 1.031, (2, 45) 1.031, ]
Explored States: [(1, 45)]
Iteration #2 ————————————————————
Current node: (1, 90)
Frontier: [(2, 45) 1.031, (1, 0) 1.031, (0, 45) 1.031, (1, 135) 1.879, (0, 90)
    1.879, (2, 90) 1.879, ]
Explored States: [(1, 45), (1, 90)]
Iteration #3 ————————————————————
Current node: (2, 45)
Frontier: [(2, 0) 0.353, (2, 90) 1.879, (3, 45) 0.353, (1, 135) 1.879, (0, 90)
    1.879, (0, 45) 1.031, (1, 0) 1.031, ]
Explored States: [(1, 45), (1, 90), (2, 45)]
Iteration #4 ————————————————————
Current node: (2, 0)
Frontier: [(3, 45) 0.353, (3, 0) 1.632, (1, 0) 1.031, (2, 90) 1.879, (0, 90)
    1.879, (0, 45) 1.031, (2, 315) 1.632, (1, 135) 1.879, ]
```

[4]When printing the *PriorityQueue*, the only sorted object is the first one, so the entire frontier should not be taken into account.

```
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0)]
Iteration #5 ————————————————————
Current node: (3, 45)
Frontier: [(1, 0) 1.031, (3, 90) 1.09, (0, 45) 1.031, (3, 0) 1.632, (0, 90) 1.879, (1, 135) 1.879, (2, 315) 1.632, (2, 90) 1.879, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45)]
Iteration #6 ————————————————————
Current node: (1, 0)
Frontier: [(0, 45) 1.031, (3, 90) 1.09, (2, 315) 1.632, (1, 315) 1.527, (0, 90) 1.879, (1, 135) 1.879, (2, 90) 1.879, (3, 0) 1.632, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0)]
Iteration #7 ————————————————————
Current node: (0, 45)
Frontier: [(3, 90) 1.09, (1, 315) 1.527, (2, 315) 1.632, (3, 0) 1.632, (0, 90) 1.879, (1, 135) 1.879, (2, 90) 1.879, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45)]
Iteration #8 ————————————————————
Current node: (3, 90)
Frontier: [(1, 315) 1.527, (3, 0) 1.632, (2, 315) 1.632, (2, 90) 1.879, (0, 90) 1.879, (1, 135) 1.879, (3, 135) 2.931, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45), (3, 90)]
Iteration #9 ————————————————————
Current node: (1, 315)
Frontier: [(1, 270) 1.0, (0, 315) 1.0, (3, 0) 1.632, (2, 90) 1.879, (0, 90) 1.879, (1, 135) 1.879, (2, 315) 1.632, (3, 135) 2.931, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45), (3, 90), (1, 315)]
Iteration #10 ————————————————————
Current node: (1, 270)
Frontier: [(0, 315) 1.0, (1, 225) 1.703, (3, 0) 1.632, (0, 270) 1.703, (2, 270) 1.703, (1, 135) 1.879, (2, 315) 1.632, (3, 135) 2.931, (2, 90) 1.879, (0, 90) 1.879, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45), (3, 90), (1, 315), (1, 270)]
Iteration #11 ————————————————————
Current node: (0, 315)
Frontier: [(3, 0) 1.632, (1, 225) 1.703, (2, 315) 1.632, (0, 270) 1.703, (2, 270) 1.703, (1, 135) 1.879, (0, 90) 1.879, (3, 135) 2.931, (2, 90) 1.879, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45), (3, 90), (1, 315), (1, 270), (0, 315)]
Iteration #12 ————————————————————
Current node: (3, 0)
Frontier: [(2, 315) 1.632, (1, 225) 1.703, (2, 90) 1.879, (0, 270) 1.703, (2, 270) 1.703, (1, 135) 1.879, (0, 90) 1.879, (3, 135) 2.931, (3, 315) 2.235, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45), (3, 90), (1, 315), (1, 270), (0, 315), (3, 0)]
Iteration #13 ————————————————————
Current node: (2, 315)
Frontier: [(1, 225) 1.703, (0, 270) 1.703, (2, 90) 1.879, (3, 315) 2.235, (2, 270) 1.703, (1, 135) 1.879, (0, 90) 1.879, (3, 135) 2.931, ]
Explored States: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0, 45), (3, 90), (1, 315), (1, 270), (0, 315), (3, 0), (2, 315)]

```

```
Path found using BestF in 8.669 ms!
Flight instructions (3) : [H180, H270, H270]
Path followed (4 states): [(1, 45), (2, 45), (2, 0), (2, 315)]
Solution path cost: 4.142

Current node depth: 3
Nodes created: 21
13 states expanded: [(1, 45), (1, 90), (2, 45), (2, 0), (3, 45), (1, 0), (0,
    45), (3, 90), (1, 315), (1, 270), (0, 315), (3, 0), (2, 315)]
```

3.4 A* Search

The expected flight path is *[H270,H270,H180]*. Current nodes being expanded should be added to the explored set, while new nodes should be added at in the frontier[5] based on their estimate distance to goal on top of the path cost so far. The console output below meets the requirements, which shows that Best-First Search works as expected.

```
Search type: AStar
Size of world N: 4
Start point: (1, 45)
End point: (2, 315)
Obstacles located at: []

Starting A* Search...
Iteration #1 ------------------------
Current node: (1, 45)
Frontier: [(1, 90) 1.031, (1, 0) 1.031, (0, 45) 1.031, (2, 45) 1.031, ]
Explored States: [(1, 45)]
Iteration #2 ------------------------
Current node: (1, 90)
Frontier: [(2, 45) 1.031, (1, 0) 1.031, (0, 45) 1.031, (1, 135) 2.664, (0, 90)
    2.664, (2, 90) 2.664, ]
Explored States: [(1, 45), (1, 90)]
Iteration #3 ------------------------
Current node: (2, 45)
Frontier: [(1, 0) 1.031, (2, 90) 1.353, (0, 45) 1.031, (1, 135) 2.664, (0, 90)
    2.664, (2, 0) 1.353, (3, 45) 1.353, ]
Explored States: [(1, 45), (1, 90), (2, 45)]
Iteration #4 ------------------------
Current node: (1, 0)
Frontier: [(0, 45) 1.031, (2, 90) 1.353, (3, 45) 1.353, (2, 0) 1.353, (0, 90)
    2.664, (1, 315) 2.313, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0)]
Iteration #5 ------------------------
Current node: (0, 45)
Frontier: [(2, 90) 1.353, (2, 0) 1.353, (3, 45) 1.353, (1, 315) 2.313, (0, 90)
    2.664, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45)]
Iteration #6 ------------------------
Current node: (2, 90)
```

[5] When printing the *PriorityQueue*, the only sorted object is the first one, so the entire frontier should not be taken into account.

```
Frontier: [(2, 0) 1.353, (1, 315) 2.313, (3, 45) 1.353, (0, 90) 2.664, (2, 135) 4.821, (3, 90) 4.821, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90)]
Iteration #7 ————————————————————
Current node: (2, 0)
Frontier: [(3, 45) 1.353, (1, 315) 2.313, (2, 315) 4.203, (0, 90) 2.664, (2, 135) 4.821, (3, 90) 4.821, (3, 0) 4.203, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0)]
Iteration #8 ————————————————————
Current node: (3, 45)
Frontier: [(1, 315) 2.313, (0, 90) 2.664, (2, 135) 4.821, (3, 0) 4.203, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0), (3, 45)]
Iteration #9 ————————————————————
Current node: (1, 315)
Frontier: [(1, 270) 2.571, (0, 315) 2.571, (2, 315) 2.571, (3, 0) 4.203, (0, 90) 2.664, (2, 135) 4.821, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0), (3, 45), (1, 315)]
Iteration #10 ————————————————————
Current node: (1, 270)
Frontier: [(0, 315) 2.571, (0, 90) 2.664, (2, 315) 2.571, (2, 270) 4.059, (2, 135) 4.821, (1, 225) 4.059, (0, 270) 4.059, (3, 0) 4.203, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0), (3, 45), (1, 315), (1, 270)]
Iteration #11 ————————————————————
Current node: (0, 315)
Frontier: [(2, 315) 2.571, (0, 90) 2.664, (1, 225) 4.059, (2, 270) 4.059, (0, 270) 4.059, (3, 0) 4.203, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0), (3, 45), (1, 315), (1, 270), (0, 315)]
Iteration #12 ————————————————————
Current node: (2, 315)
Frontier: [(0, 90) 2.664, (2, 270) 4.059, (1, 225) 4.059, (3, 0) 4.203, (0, 270) 4.059, ]
Explored States: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0), (3, 45), (1, 315), (1, 270), (0, 315), (2, 315)]

Path found using AStar in 6.071 ms!
Flight instructions (3) : [H270, H270, H180]
Path followed (4 states): [(1, 45), (1, 0), (1, 315), (2, 315)]
Solution path cost: 2.571

Current node depth: 3
Nodes created: 39
12 states expanded: [(1, 45), (1, 90), (2, 45), (1, 0), (0, 45), (2, 90), (2, 0), (3, 45), (1, 315), (1, 270), (0, 315), (2, 315)]
```

3.5 Obstacles

The same problem is tested using BFS with an obstacle at (1,90). The following console output shows that the BFS search algorithm avoids going going to (1,90), instead taking a longer path.

```
Search type: BFS
Size of world N: 4
Start point: (1, 45)
End point: (3, 225)
Obstacles located at: [(1, 90)]

[...]

Path found using BFS in 5.83 ms!
Flight instructions (6) : [H270, H270, H270, H270, H180, H180]
Path followed (7 states): [(1, 45), (1, 0), (1, 315), (1, 270), (1, 225), (2, 225), (3, 225)]
Solution path cost: 5.142

Current node depth: 6
Nodes created: 31
30 states expanded: [(1, 45), (1, 0), (0, 45), (2, 45), (1, 315), (2, 0), (0, 90), (2, 90), (3, 45), (1, 270), (0, 315), (2, 315), (3, 0), (0, 135), (2, 135), (3, 90), (1, 225), (0, 270), (2, 270), (3, 315), (0, 180), (2, 180), (3, 135), (1, 180), (0, 225), (2, 225), (3, 270), (3, 180), (1, 135), (3, 225)]
```

If the initial state is surrounded by obstacles, then the program should not be able to find a solution, as shown by the console output below.

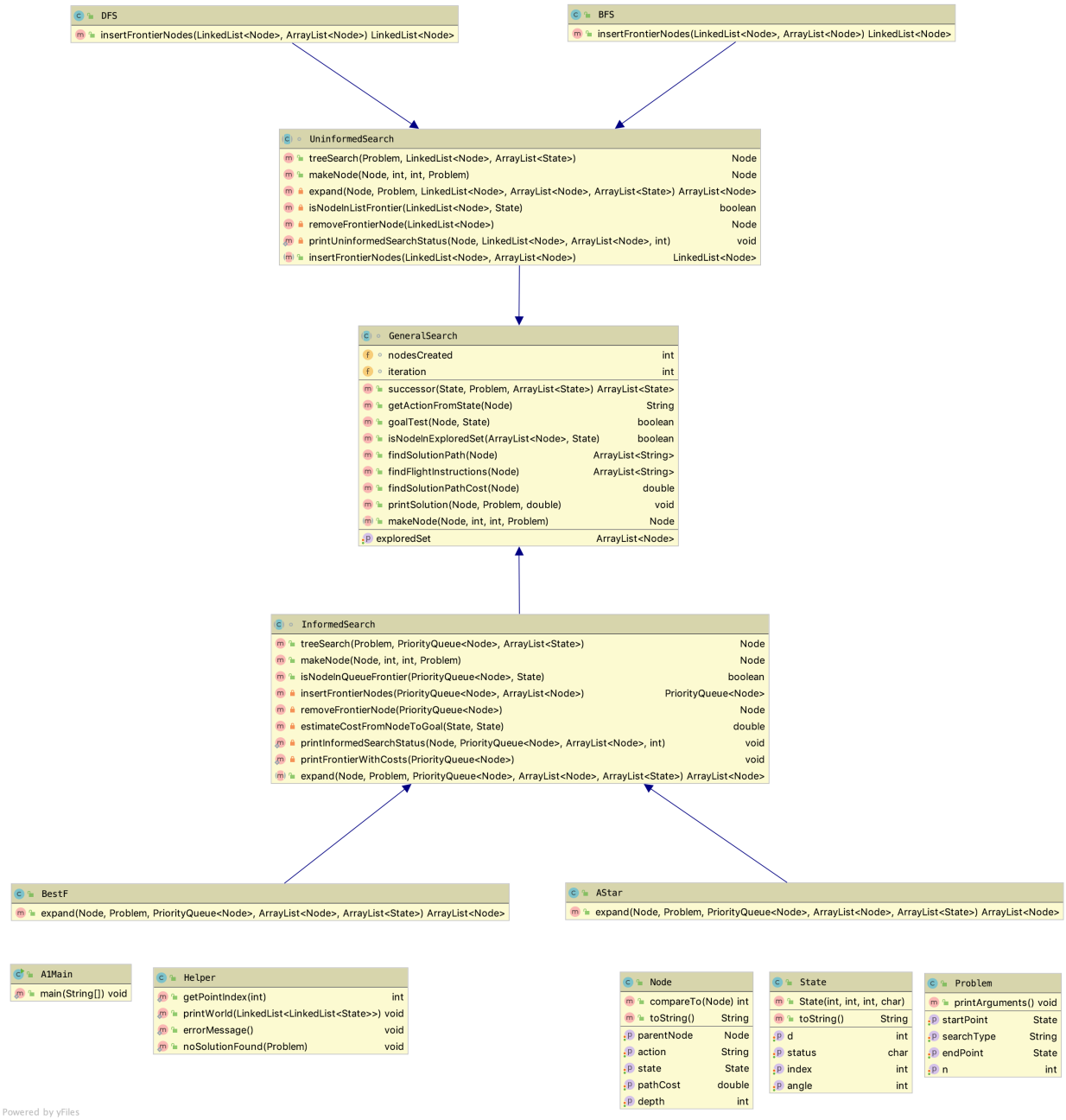
```
Search type: BFS
Size of world N: 4
Start point: (1, 45)
End point: (3, 225)
Obstacles located at: [(1, 90), (1, 0), (2, 45)]

[...]

No path from (1, 45) to (3, 225) found!
Try removing obstacles to allow the algorithm to find a route.
```

Appendix A UML Class Diagram

The UML Class Diagram of the program, generated using yWorks [2].



Appendix B Path length across different algorithms

	BFS	**DFS**	**BestF**	**A***
i. N=5 S=(2,0) G=(2,135)	3	3	3	5
ii. N=5 S=(1,180) G=(4,180)	3	11	3	3
iii. N=5 S=(1,315) G=(4,45)	5	13	5	11
v. N=8 S=(1,135) G=(5,315)	4	20	4	4
vi. N=8 S=(4,0) G=(7,90)	5	13	5	5
viii. N=10 S=(9,225) G=(2,45)	11	35	13	15
ix. N=10 S=(2,45) G=(9,225)	11	35	11	15
x. N=10 S=(7,270) G=(7,90)	4	4	16	18

Table 1: Path length

Appendix C Path cost across different algorithms

	BFS	**DFS**	**BestF**	**A***
i. N=5 S=(2,0) G=(2,135)	4.712	4.712	4.712	5.142
ii. N=5 S=(1,180) G=(4,180)	3	11.639	3	3
iii. N=5 S=(1,315) G=(4,45)	4.571	16.352	8.498	5.785
v. N=8 S=(1,135) G=(5,315)	4	35.416	4	4
vi. N=8 S=(4,0) G=(7,90)	9.283	39.914	9.283	9.283
viii. N=10 S=(9,225) G=(2,45)	35.274	135.805	24.708	11
ix. N=10 S=(2,45) G=(9,225)	13.283	120.097	13.283	12.571
x. N=10 S=(7,270) G=(7,90)	21.991	21.991	19.854	15.571

Table 2: Path cost

Appendix D Numbers of nodes expanded across different algorithms

	BFS	**DFS**	**BestF**	**A***
i. N=5 S=(2,0) G=(2,135)	13	4	4	13
ii. N=5 S=(1,180) G=(4,180)	21	25	7	9
iii. N=5 S=(1,315) G=(4,45)	33	23	28	26
v. N=8 S=(1,135) G=(5,315)	30	23	9	11
vi. N=8 S=(4,0) G=(7,90)	44	46	9	11
viii. N=10 S=(9,225) G=(2,45)	72	36	24	38
ix. N=10 S=(2,45) G=(9,225)	79	45	22	37
x. N=10 S=(7,270) G=(7,90)	25	5	30	43

Table 3: Nodes/States Expanded

Appendix E Node depth across different algorithms

	BFS	**DFS**	**BestF**	**A***
i. N=5 S=(2,0) G=(2,135)	3	3	3	5
ii. N=5 S=(1,180) G=(4,180)	3	11	3	3
iii. N=5 S=(1,315) G=(4,45)	5	13	5	11
v. N=8 S=(1,135) G=(5,315)	4	20	4	4
vi. N=8 S=(4,0) G=(7,90)	5	13	5	5
viii. N=10 S=(9,225) G=(2,45)	11	35	13	15
ix. N=10 S=(2,45) G=(9,225)	11	35	11	15
x. N=10 S=(7,270) G=(7,90)	4	4	16	18

Table 4: Node depth

Appendix F Number of nodes created across different algorithms

	BFS	**DFS**	**BestF**	**A***
i. N=5 S=(2,0) G=(2,135)	22	11	11	47
ii. N=5 S=(1,180) G=(4,180)	30	39	16	33
iii. N=5 S=(1,315) G=(4,45)	37	39	37	26
v. N=8 S=(1,135) G=(5,315)	38	45	20	40
vi. N=8 S=(4,0) G=(7,90)	55	63	22	41
viii. N=10 S=(9,225) G=(2,45)	76	65	46	136
ix. N=10 S=(2,45) G=(9,225)	79	79	46	135
x. N=10 S=(7,270) G=(7,90)	36	14	56	162

Table 5: Nodes created

Appendix G Runtime across different algorithms

	BFS	**DFS**	**BestF**	**A***
i. N=5 S=(2,0) G=(2,135)	3.684	2.64	7.166	8.07
ii. N=5 S=(1,180) G=(4,180)	5.606	6.977	5.975	6.858
iii. N=5 S=(1,315) G=(4,45)	7.841	5.817	14.596	11.578
v. N=8 S=(1,135) G=(5,315)	8.881	6.08	7.257	7.489
vi. N=8 S=(4,0) G=(7,90)	11.234	10.708	6.744	9.283
viii. N=10 S=(9,225) G=(2,45)	11.658	8.161	14.814	15.223
ix. N=10 S=(2,45) G=(9,225)	13.42	8.852	12.803	14.164
x. N=10 S=(7,270) G=(7,90)	21.991	2.948	19.296	17.119

Table 6: Runtime (ms)

References

[1] Java documentation. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html. [Online] Accessed: 2019-10-05.

[2] yworks, the diagramming expert. https://www.yworks.com/, 2019. [Online] Accessed: 2019-10-10.

[3] Heinz M. Kabutz. Difference between arraylist and linkedlist. https://www.javatpoint.com/difference-between-arraylist-and-linkedlist. [Online] Accessed: 2019-10-10.

[4] Heinz M. Kabutz. When arguments get out of hand. https://www.javaspecialists.eu/archive/Issue059.html, November 2002. [Online] Accessed: 2019-10-10.

[5] A. Toniolo. Cs5011 uninformed search - euclidian distance in polar coordinates heuristic. https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L5_w3.pdf. [Online] Accessed: 2019-10-04.

[6] A. Toniolo. Cs5011 uninformed search - general search algorithm v2. https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L3_w2.pdf. [Online] Accessed: 2019-10-04.