CS5011 Assignment 1: Search

Student ID: 210034146
(Dated: February 13, 2022)

1: INTRODUCTION

1.1: Sections implemented

The following list describes the sections implemented in this assignment:

- Part 1 (Basic ferry search): Attempted and fully working BFS and DFS agents.
- Part 2 (Intermediate ferry agent): Attempted and fully working Best first and A* agents.
- Part 3 (Advanced Ferry Agent): not attempted.

1.2: Compilation instructions

Steps to compile the algorithm:

- Navigate to the src folder using CDM/terminal/command line.
- Type the following: javac A1main.java
- Compilation is complete if no errors are thrown.

1.3: Running instructions

Steps to run the algorithm:

- Navigate to the src folder using CDM/terminal/command line.
- Type the following: java A1main <Agent> <ConfID>
- For example; to run the BFS agent on CONF11, type the following command: java A1main BFS CONF11.

1.4: Instructions: JUnit tests

In addition to the basic ferry agents and Intermediate search agents, this assignment also contains 23 JUnit tests which test a variety of functionality, ranging from testing how the algorithm handles incorrect/invalid inputs to testing five additional search problems (configurations); this involves comparing the output generated by each agent with manually computed outputs. Steps to run the JUnit tests (macOS/linux):

- Using CDM/terminal/command line navigate to the directory which contains the src and the tests folders.
- Type the following to compile the test: javac -cp junit.jar:hamcrest.jar:./tests/:. tests/*.java
- Type the following to run the test : java -cp junit.jar:hamcrest.jar:./tests/:. org.junit.runner.JUnitCore ModelTest
- To run the JUnit test on windows replace '/' with '\' and replace ':' with ';' in the above code.

2: DESIGN AND IMPLEMENTATION

2.1: Brief introduction to search

Problem-solving agents in Artificial intelligence use search techniques to solve problems efficiently. Agents such as these are goal-based, anticipating their past and future actions and desired outcomes. Search techniques operate within a search space where they are tasked with finding a route to the goal (desired) state from the start state. Search techniques take on an iterative approach towards problem-solving; for example, they work by exploring the start state, then exploring the start state's successors (states directly accessible from the start states), followed by exploring the successor states of the current successor states. This process is repeated until the goal state is found or until there are no more states to explore, in which case the search fails. Search techniques employ a key data structure known as a frontier, a container of successors states yet to be explored. Search techniques work by iteratively expanding the frontier (also expanding the tree) at each step by removing the first element from the frontier and adding the said element's successor states to the frontier. This is repeated until the goal state is the first element in the frontier or the frontier is empty (no more states to explore).

Search algorithms mainly fall under two categories: uninformed and informed search. Uninformed search algorithms such as breadth-first search (BFS) and depth-first search (DFS) don't involve the use domain knowledge (i.e. distance, path cost, closeness) in their decision making process (selecting which nodes to expand first in the frontier). Uninformed search algorithms are considered as brute-force methods.

Informed search, on the other hand, uses domain knowledge such as distance, path cost, closeness in their decision-making process to select which nodes to expand first in the frontier. This allows informed search algorithms to find solutions efficiently (less than or equal to the number of nodes visited compared to uninformed search).

2.2: Main Design

The problem at hand involves navigating a ferry through a 2D lattice of NxN (N is length of the map in one dimension) equilateral triangles alternating in their orientation (facing up and facing down). A cell (triangle) is defined by its coordinates (row, column), for example, the S (start) state and the G (goal) state have coordinates of (1,1) and (3,4) respectively. In terms of the data structure, this map is represented by a 2D (NxN) array of integers. Cells with the value of 0 (white triangles) and 1 (black triangles) are available (water) and unavailable (land) cells respectively (see figure 1).

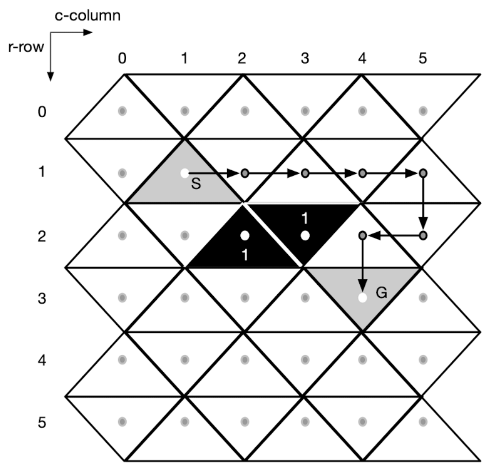


FIG. 1: Example map. Figure taken from [1]

Successor cells are defined as triangles adjacent to the current triangle with the following condition: triangles facing up can only have the following successors: left (row, column-1), right (row, column+1), and down (row+1, column). Whereas triangles facing down have the following successors: left (row, column-1), right (row, column+1), and up (row-1, column).

The algorithm starts by defining cells into states; this is done by labelling each cell in the map with a number n. For a NxN there are a total of N^2 states. Each cell is assigned a number between $0 \leq n < N^2$ based on its position. For example in a 6x6 map, cells (0,0), (1,0), and (2,2) are mapped into the states 0, 6, and 14 respectively, whereas in a 3x3 map, cells (0,0), (1,0), and (2,2) are mapped into the states 0, 3, and 8 respectively.

A successor function is defined such that given an input state, the function returns a list of successor states as per the above-defined conditions (separate conditions upward facing and downward facing triangles). A HashMap (see listing 1) pairs all the states $0 \leq n < N^2$ with their respective successors (using the successor function). The use of a HashMap allows the algorithm to look up the successors of any given state n by using n itself as a key (reference). Once the HashMap is generated, the algorithm branches out into different sequences depending on what *agent* is provided as an input to the system.

```
HashMap<int, int[]> map = new HashMap<int, int[]>()
for(int i = 0; i < maplength**2; i++):
    map.put(i, successor-FN(i))
```

Listing 1: Pseudo-code for a Hashmap which pairs every state with its successor states

Data structure known as a node is used to build the search tree at every step. Each node contains a single state n, along with its path cost, distance to the goal state (heuristic), and the path transversed so far. The function make-node (see listing 2) returns a node given an input state. The make-node function is used to create a root node using the start state s, and this root node is added to the frontier.

```
function makeNode(Node n, int state, int goal_state
    ) returns node
    node = new Node(state)
    node.parent = n
    node.path_cost = n.path_cost + 1
    node.depth = n.depth + 1
    node.h_cost = manhattanDistance(int state, int
    goal)
    node.f_cost = node.h_cost + 1*node.path_cost
    return node
```

Listing 2: Make node function

2.3: Uninformed search

The BFS and the DFS algorithms implement a data structure known as a Deque (double-ended queue) as its frontier, allowing the algorithm to add data to the frontier from both ends (works both as a queue and a stack). The tie-breaking strategy for the successor states is *Right* > *Down* > *Left* > *Up* (exploring states in clockwise order).

The BFS algorithm works by transversing the tree from the root node and exploring all the neighbouring states (creating new nodes from the neighbouring states and adding them to the frontier) and then exploring the successors of the said neighbouring nodes (see figure 3). Java Deque provides an add() method, which allows the Deque to effectively work as a queue data structure (first-in first-out) see figure 2.

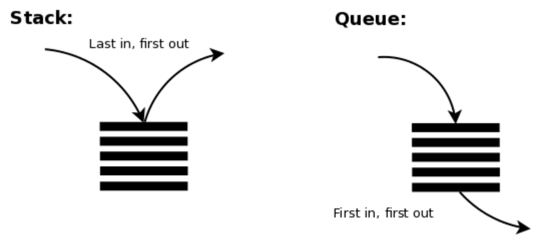


FIG. 2: Stack vs Queue data structure. Figure taken from [2]

The DFS algorithm, on the other hand, works by transversing the tree from the root node and then goes deeper until it finds the required node (containing the goal state) or a node with no successors. Once it reaches a node with no successors, it moves up in-depth and explores the next available node in the frontier (see figure 3). Java Deque provides an .addFirst method which allows the Deque to effectively work as a stack data structure (last-in first-out) see figure 2. Listing 3 describes the pseudo-code for the uninformed search agents.

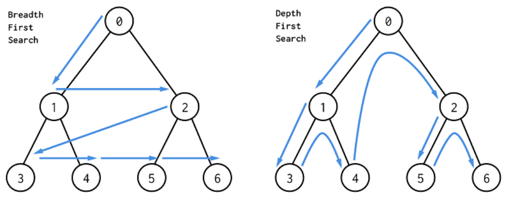


FIG. 3: BFS vs DFS. Figure taken from [3]

```
Node root = makeNode(Node null, int start_state, int goal_state)

Deque frontier = new ArrayDeque()
Deque explored = new ArrayDeque()
frontier.add(root)
boolean achived_goal_or_failed = false

while(!achived_goal_or_failed):
  if(frontier.size() == 0):
    achived_goal_or_failed = true
    print("fail")
    break

  if(frontier.size() > 0):
    Node temp_node = frontier.peek()
    if (temp_node.state == goal):
      achived_goal_or_failed = true
      print(temp_node.path)
      print(temp_node.path_cost)
      break

    explored.add(temp_node)
    frontier.remove()
    // array of successor states
    Node[] successor = expand(temp_node)
    for(int i = 0; i < successor.length; i++):
        int match = 0
        // avoiding redundant nodes (frontier)
        for(Node n: frontier):
            if(n.state == successor[i].state):
                match = 1
        // avoiding redundant nodes (explored list)
        for(Node e : explored):
            if(e.state == successor[i].state):
                match = 1

        // adding non-redundant nodes to the frontier
        if(match == 0):
            // adding nodes: first-in first-out
            if(algorithm.equals("BFS")):
                frontier.add(successor[i])
            // adding nodes: last-in first-out
            if(algorithm.equals("DFS")):
                frontier.addFirst(successor[i])
```

Listing 3: Pseudo-code: Uninformed search agents (BFS/DFS)

2.4: Informed search

2.4.1: Best first search

The Best first search algorithm implements a data structure known as a priorityQueue. This allows the Best first search algorithm to compare nodes in the frontier at each step and choose the node with the smallest distance (heuristic) to the goal state/cell. Eq (1) describes the Manhattan distance heuristic h (referred to as the h cost). The best first search algorithm chooses the best node at every step, gradually moving deeper within the search tree.

$$H = |a_1 - a_2| + |b_1 - b_2| + |c_1 - c_2| \tag{1}$$

Here a, b, and c are coordinates of a cell (triangle) in a triangular coordinate system. These triangular coordinates can be derived using the standard lattice coordinates (row, column) (see equations 2 - 4). The implementation of the Best first search algorithm is described in listing 4. The non-redundant nodes are added to the frontier via the add() method of the priorityQueue.

$$a = -row \tag{2}$$

$$b = (row + column - orientation)/2 \tag{3}$$

$$c = (row + column - orientation)/2 - row + orientation \tag{4}$$

$$orientation = \begin{cases} 0, & \text{if: upwards triangle} \\ 1, & \text{if: downwards triangle} \end{cases} \tag{5}$$

```
Node root = makeNode(Node null, int start_state, int goal_state)

// priority queue's first element is the node with the smallest h_cost
PriorityQueue frontier = new PriorityQueue<>(h_costComparator)
Deque explored = new ArrayDeque()
frontier.add(root)
boolean achived_goal_or_failed = false

while(!achived_goal_or_failed):
  if(frontier.size() == 0):
    achived_goal_or_failed = true
    print("fail")
    break
  if(frontier.size() > 0):
    // node with the lowest heuristic is picked
    Node temp_node = frontier.peek();
    if (temp_node.state == goal):
      achived_goal_or_failed = true
      print(temp_node.path)
      print(temp_node.path_cost)
      break

    explored.add(temp_node)
    frontier.remove()
    // array of successor states
    Node[] successor = expand(temp_node)
    for(int i = 0; i < successor.length; i++):
        int match = 0
        // avoiding redundant nodes (frontier)
        for(Node n: frontier):
            if(n.state == successor[i].state):
                match = 1
        // avoiding redundant nodes (explored list)
        for(Node e : explored):
            if(e.state == successor[i].state):
                match = 1

        // adding non-redundant nodes to the frontier
        if(match == 0):
            frontier.add(successor[i])
```

Listing 4: Pseudo-code: Best first search algorithm

2.4.2: A search*

The A* search agent also implements the priorityQueue data structure for its frontier, allowing it to compare nodes in the frontier at each step and choose the node with the smallest total cost f (referred to as the f cost), defined as the sum of the Manhattan distance to the goal state and the path cost (see equation 6).

$$\begin{aligned} f &= |a_1 - a_2| + |b_1 - b_2| + |c_1 - c_2| + path.cost \\ &= H + path.cost \end{aligned} \tag{6}$$

Where a,b, and c are the coordinates of a cell in a triangular coordinate system as before. Furthermore, unlike the best first search agent which only chooses the best node at every step and moves deeper within the tree, the A* agent can also replace a node within the frontier if it comes across a new node which contains the same state as one of the nodes in the frontier but with a lower total cost. This allows the agent to always return an optimal solution. Listing 5 describes the pseudo-code for the A* search agent.

```
Node root = makeNode(Node null, int start_state, int goal_state)

// priority queue's first element is the node with the smallest f_cost
PriorityQueue frontier = new PriorityQueue<>(f_costComparator)
Deque explored = new ArrayDeque()
frontier.add(root)
boolean achived_goal_or_failed = false

while(!achived_goal_or_failed):
  if(frontier.size() == 0):
    achived_goal_or_failed = true
    print("fail")
    break

  if(frontier.size() > 0):
    // node with the lowest f_cost is picked
    Node temp_node = frontier.peek();
    if (temp_node.state == goal):
      achived_goal_or_failed = true
      print(temp_node.path)
      print(temp_node.path_cost)
      break

    explored.add(temp_node)
    frontier.remove()
    // array of successor states
    Node[] successor = expand(temp_node)
    for(int i = 0; i < successor.length; i++):
        int match = 0
        int match_with_lower_f_cost = 0;
        // avoiding redundant nodes (frontier)
        for(Node n: frontier):
            if(n.state == successor[i].state && n.f_cost < successor[i].f_cost):
                match = 1
        // avoiding redundant nodes (explored list)
        for(Node e : explored):
            if(e.state == successor[i].state):
                match = 1

        // dummy node
        Node node_to_be_replaced = makeNode(null, null, null, null)

        // checking if the successor state already exists in the frontier with a lower f_cost
        for(Node n: frontier):
            if(n.state == successor[i].state && n.f_cost > successor[i].f_cost):
                match_with_lower_f_cost = 1
                node_to_be_replaced = n

        // adding non-redundant nodes to the frontier
        if(match == 0):
            frontier.add(successor[i])

        // replacing a node in the frontier with a successor node contaning the same state with a lower f_cost
        if(match_with_lower_f_cost == 1):
            frontier.remove(placeholder)
            frontier.add(successor[i])
```

Listing 5: Pseudo-code: A* search algorithm

3: TEST SUMMARY

The system passes all stacscheck test cases provided on studres. However, the stacscheck only validates the BFS implementation of the system. For this reason, the four agents were tested on five new configurations (CONF25 - CONF29), resulting in a total of 20 manual hardcoded JUnit tests to assess the correctness of the system. The inputs to the system were the agent type (BFS/DFS/BestF/AStar) and the configurationID. The system prints the states in the frontier at each step, followed by the path, the path cost, and the number of nodes visited. The following five tables depict the test results for the five new configurations. The table contains the path cost, the number of nodes visited, and whether or not the system output matches the expected output (expected outputs are derived manually).

ConfID: CONF25				
Agent Type	BFS	DFS	BestF	A*
Path cost	4	4	4	4
Nodes visited	6	5	5	5
Manual test	Pass	Pass	Pass	Pass

TABLE I: Test results for CONF25

ConfID: CONF26				
Agent Type	BFS	DFS	BestF	A*
Path cost	4	22	4	4
Nodes visited	8	26	5	5
Manual test	Pass	Pass	Pass	Pass

TABLE II: Test results for CONF26

ConfID: CONF27				
Agent Type	BFS	DFS	BestF	A*
Path cost	3	11	3	3
Nodes visited	9	16	4	4
Manual test	Pass	Pass	Pass	Pass

TABLE III: Test results for CONF27

ConfID: CONF28				
Agent Type	BFS	DFS	BestF	A*
Path cost	3	3	3	3
Nodes visited	7	4	4	4
Manual test	Pass	Pass	Pass	Pass

TABLE IV: Test results for CONF28

ConfID: CONF29				
Agent Type	BFS	DFS	BestF	A*
Path cost	4	4	4	4
Nodes visited	11	6	5	7
Manual test	Pass	Pass	Pass	Pass

TABLE V: Test results for CONF29

3.1: Working example

A working example of all agents on configuration 25 is demonstrated in the following subsections. Listing 6 depicts the 2d map belonging to configuration 25. The coordinates of start and end cells are (0,0) and (1,3) respectively.

```
   0 1 2 3 4 5
0|/S\1/1\1/1\1/
1|\./.\./G\./.\
2|/.\./.\./.\./
3|\./.\./.\./.\
4|/.\./.\./.\./
5|\./.\./.\./.\
```

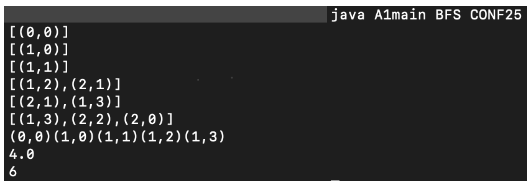
Listing 6: 2d map

3.1.2: BFS

Manual search (frontier at each step, followed by the path, path cost and number of nodes visited).

- [(0,0)]
- [(1,0)]
- [(1,1)]
- [(1,2),(2,1)] //tie breaking strategy (right > down)
- [(2,1),(1,3)]
- [(1,3),(2,2),(2,0)] //reached goal state
- path: (0,0)(1,0)(1,1)(1,2)(1,3)
- path cost: 4.0
- nodes visited: 6

System output:



```
java A1main BFS CONF25
[(0,0)]
[(1,0)]
[(1,1)]
[(1,2),(2,1)]
[(2,1),(1,3)]
[(1,3),(2,2),(2,0)]
(0,0)(1,0)(1,1)(1,2)(1,3)
4.0
6
```

FIG. 4: System output: BFS

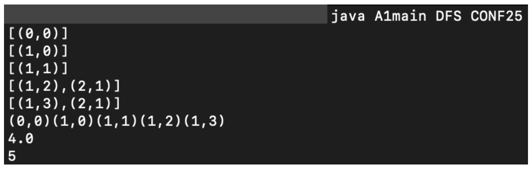
3.1.3: DFS

Manual search (frontier at each step, followed by the path, path cost and number of nodes visited).

- [(0,0)]
- [(1,0)]
- [(1,1)]
- [(1,2),(2,1)] //tie breaking strategy (right > down)
- [(1,3),(2,1)] //reached goal state

- path: (0,0)(1,0)(1,1)(1,2)(1,3)
- path cost: 4.0
- nodes visited: 5

System output:



```
java A1main DFS CONF25
[(0,0)]
[(1,0)]
[(1,1)]
[(1,2),(2,1)]
[(1,3),(2,1)]
(0,0)(1,0)(1,1)(1,2)(1,3)
4.0
5
```

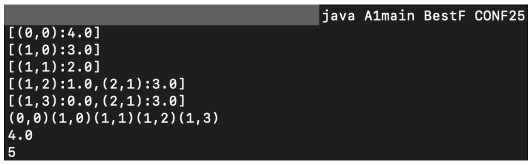
FIG. 5: System output: DFS

3.1.4: Best first search

Manual search (frontier at each step, followed by the path, path cost and number of nodes visited). The number after ":" represents the Manhattan distance to the goal state.

- [(0,0):4.0]
- [(1,0):3.0]
- [(1,1):2.0]
- [(1,2):1.0,(2,1):3.0] //exploring the node with the lowest distance
- [(1,3):0.0,(2,1):3.0] //reached goal state
- path: (0,0)(1,0)(1,1)(1,2)(1,3)
- path cost: 4.0
- nodes visited: 5

System output:



```
java A1main BestF CONF25
[(0,0):4.0]
[(1,0):3.0]
[(1,1):2.0]
[(1,2):1.0,(2,1):3.0]
[(1,3):0.0,(2,1):3.0]
(0,0)(1,0)(1,1)(1,2)(1,3)
4.0
5
```

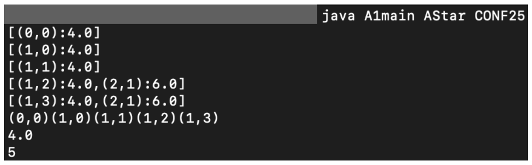
FIG. 6: System output: Best first search

3.1.5: A search*

Manual search (frontier at each step, followed by the path, path cost and number of nodes visited). The number after ":" represents the total cost: Manhattan distance + path cost.

- [(0,0):4.0]
- [(1,0):4.0]
- [(1,1):4.0]
- [(1,2):4.0,(2,1):6.0] //exploring the node with the lowest total cost
- [(1,3):4.0,(2,1):6.0] //reached goal state
- path: (0,0)(1,0)(1,1)(1,2)(1,3)
- path cost: 4.0
- nodes visited: 5

System output:



```
java A1main AStar CONF25
[(0,0):4.0]
[(1,0):4.0]
[(1,1):4.0]
[(1,2):4.0,(2,1):6.0]
[(1,3):4.0,(2,1):6.0]
(0,0)(1,0)(1,1)(1,2)(1,3)
4.0
5
```

FIG. 7: System output: A* search

4: EVALUATION AND CONCLUSION

Two metrics are used to evaluate which agent works the best for Marine navigation planning:

- Frequency of best routes (lowest path cost/optimal route)
- Frequency of routes with the lowest number of search states visited (computationally efficient)

Assuming the correctness of the system holds, the above-mentioned metrics are measured by running the system on all 37 configurations defined in the Conf.java file. Out of these 37 configurations, only 33 have a valid/possible route from the start and goal states. Figures 7 and 8 graphically represent the frequency of best routes and the frequency of routes with the lowest number of search states visited for each agent type. The BFS and the A* search algorithms always return the best route (lowest path cost/optimal solution). However, due to its informed searched approach, the A* search algorithm does find the best optimal route by visiting significantly fewer number of nodes (computationally efficient). On the other hand, the best first search agent finds a valid path by visiting significantly fewer nodes relative to other agents. However, unlike the BFS and A* agents, the route obtained by the best first search agent is not guaranteed to be optimal (best route). In the light of these facts, the A* search agent would be the best agent for marine navigation planning since it is guaranteed to find the most optimal solution while making use of the domain knowledge to do so in a computationally efficient manner (visiting significantly fewer number of nodes compared to BFS & DFS).

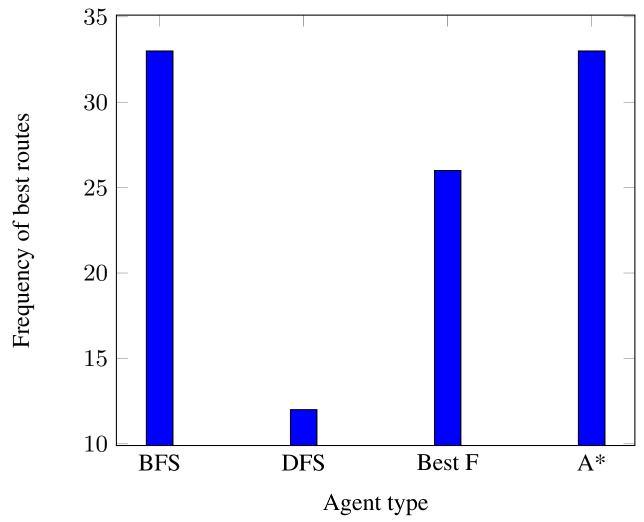


FIG. 8: Frequency of routes with the lowest path cost (best route) found by each agent.

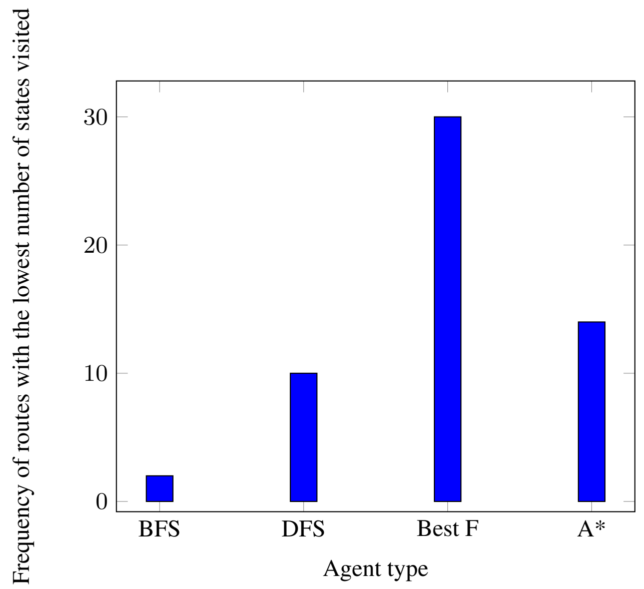


FIG. 9: Frequency of routes with the lowest number of search states visited by each agent.

REFERENCES

[1] *CS5011: A1 - Search - Marine Navigation Planner*, School of Computer Science, University of St Andrews, (2022)

[2] *Computer science: Stacks and Queues*, Highbrow.com, Available at: https://gohighbrow.com/stacks-and-queues/ [Accessed 11 February 2022].

[3] *What is the Difference Between BFS and DFS Algorithms*, K. Gupta, Available at: ¡https://www.freelancinggig.com/blog/2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms/¿ [Accessed 11 February 2022].