Marine Navigation Planner Report

1 Introduction

1.1 Meta Information

Assignment: Practical 1

Submission Date: Feb 8th, 2022

ID: 220011642

1.2 Checklist of Implemented Parts

· Basic Agent (BFS & DFS): Completed, fully working

· Intermediate Agent (BestF & AStar): Completed, fully working

· Advanced Agent (BiDirectional Search): Completed, fully working

1.3 Compiling and Running Instructions

The execution of the program is designed as the spec required.

To compile, navigate to the src directory and execute the following command:

```
javac *.java
```

After the programs have been compiled, to run the code using this command:

```
java A1main <BFS|DFS|BestF|AStar|BiDirectional> <ConfID>
```

For example, to run BiDirectional on CONF5, one can type:

```
java A1main BiDirectional CONF5
```

To run BestF on CONF20, one can type:

```
java A1main BestF CONF20
```

2 Design & Implementation

2.1 PEAS Mode

Agent	Performance Measure	Environment	Actuator	Sensor
Marine Navigation Planner	Path length found, Nodes visited to find the path	Configured binary maps designed to simulate a portion of Izaland's sea	Moving onto adjacent coordinates, Detecting land and water	Positional coordinates, Numbers given on the coordinates

2.2 Problem Definition

The problem this practical is trying to tackle is to implement a marine navigation planner that uses different search algorithms within a triangular grid, to find the best route from a given departure port to a destination port. Due to the special nature of triangular grids, the agent can only move around and explore adjacent blocks under certain restrictions. For a triangle facing upward, the legal moves are to go right, down and left, whereas for a downward triangle it can only go right, up and left. The representation of the triangles are differentiated by the coordinates of each node in the given map. The agent shall avoid hitting land which is marked as 1 in the binary map, and only move onto adjacent water areas which are marked as 0 (Figure 1 & Figure 2).

The task is to implement 5 different algorithms to realize the route planning functionality, which will then be compared to each other and evaluated at the end of the report.

Figure1 .Visualization for CONF15

Figure 2. Visualization for CONF20

2.3 System Architecture

All the files are in the same level and the same package, there are no sub-packages involved. The parent class of all required algorithms is called the General Search. It implements only the basic functionalities such as return the extended list, the frontier, etc. It doesn't actually search per se. However, It appointed an abstract method called expand() for all the other algorithms to work in their respective ways to explore the provided map (Figure 3).

There is one exception, which is the BiDirectional Search algorithm. It is kept as a separate individual class, not inheriting from the General Search, simply to distinguish itself as the special advanced agent from the other spec-designated ones. It is an advancement built upon the BFS algorithm.

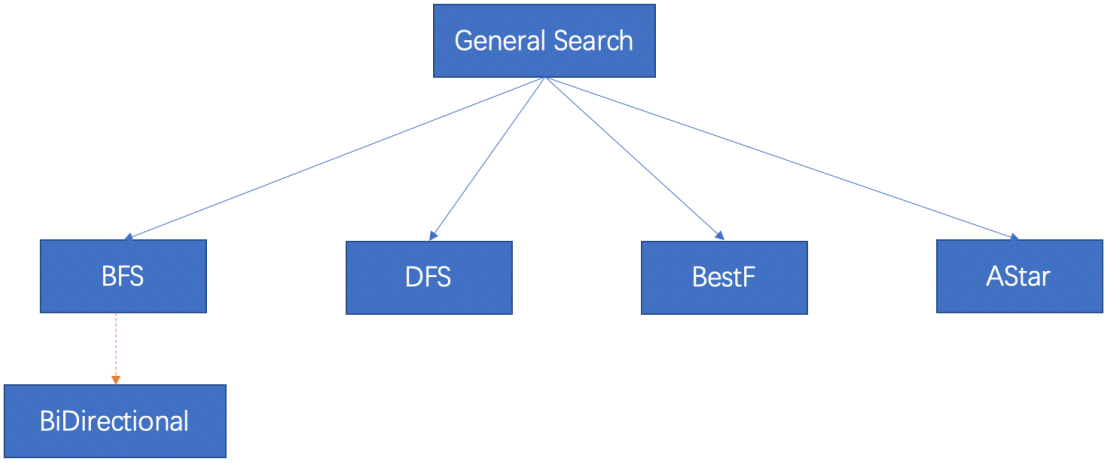

Figure 3. System Architecture of the Navigation Planner

2.4 Key Elements of Implementation

BFS:

The idea of BFS is to always keep exploring the adjacent coordinates level-by-level. To achieve that, a FIFO list is the way to do it because the early adjacent nodes will continue to be popped out first in order to reach nodes that are another step away. Newly explored nodes will be appended to the end of the list and will hence be popped later on. Therefore, the data structure to best realize this effect is the **LinkedList**.

The idea is simple, as long as the head of list popped isn't the goal node, we keep expanding on this popped node together with the information of the direction of the triangle (decided by its coordinate). Since there is a tie-breaking strategy, we also add the adjacent nodes in a particular order that aligns with the strategy:

Right —> Down(if applicable) —> Left —> Up(if applicable)

This way, the right node will always be popped before the other ones. While adding the node, an action index (1,2,3,4) is granted in order to keep track of how it got to this node from its parent. This makes it possible to not only backtrack from the found goal node all the way to the start node using `Node.getParent()` to form a path, but also to print out the action taken every step of the way. Also, an extended list is utilized to avoid redundant exploration.

DFS:

The idea of DFS is only different from BFS in one characteristic, that is to keep exploring the last reached node to further dig deeper. Therefore, an opposite LIFO list is applied because the new node is added to the end, and it will be popped out first. A **Stack** satisfies this desired behavior.

Similarly, the nodes in the stack keep being popped until the goal node shows up, and since a Stack behaves differently from a LinkedList, the adjacent nodes

are added to the stack in a different order to ensure correct sequence of being popped:

`Up(if applicable) -> Left -> Down(if applicable) -> Right`

In this case, the right node will be popped first since it's at the top of the stack. The path is formed by backtracking the goal node's parents as well.

BestF:

This is where the heuristics come in. The heuristics used for BestF is the Manhattan Distance to the goal, and it is computed by the `computeDistance()` method that calculates that using the formula suggested in Red Blog Games[1].

Since BestF cares more about the heuristic than tie-breaking strategy, a **PriorityQueue** is the best data structure to achieve directly the effect, since in a PriorityQueue, the smaller the number, the higher the priority, which aligns with the idea of BestF. However, to still ensure the tie-breaking strategy when priorities are equal, we have to change the way we compare two Nodes.

The solution is to make `Node implements Comparable<Node>`, and modify the `compareTo()` method to compare action index when heuristics are tied, and also to compare depth when actions are tied. Therefore, for each round of exploring neighbor nodes, a computed heuristic value and depth is additionally granted to the node. Eventually, each time the frontier will pop out the node with the lowest heuristic, lowest action index and lowest depth.

AStar:

AStar differs from BestF in only two behaviors. First is the heuristics used, all we have to do is to add up the path taken with the Manhattan Distance. Path cost so far is easy, as each path only cost 1, which essentially makes the depth the path cost so far. Therefore, modifying the `fcost` to `fcost+depth` when creating a node should suffice.

---

[1] https://simblob.blogspot.com/2007/06/distances-on-triangular-grid.html

The second is to replace the old node in the frontier with a new one with higher priority. To achieve this, every time when exploring new nodes, if a node is already in the frontier, the `handleIfInFrontier()` method goes through the frontier to find the same node and compare the priority and swap when needed. Everything else works the same as BestF.

BiDirectional:

The BiDirectional search uses BFS as the base algorithm, and starts two individual BFS at the same time. It keeps two separate frontiers and extended lists for one starting from the `start` and one starting from the `goal` respectively. The idea is to not look for goal, but paths instead. The two BFS makes one exploration in every iteration as long as either of the frontier is not empty yet. While searching, if a node popped is found in the opponent's frontier, a path is found because they both contain the information that helps to trace back to the start and goal node. At this point, attaching the two paths will construct the full path we want. To separate the two BFS's operation, key methods such as `expand()` will take a parameter `number`(1 or 2) to distinguish which BFS called it and operate accordingly.

Testing

My implementation achieved the following score during testing:

Folder Name	Total Tests	Passed Tests
/Tests	23	23
/OtherMaps	105	100

Example showcase (AStar run on CONF20):

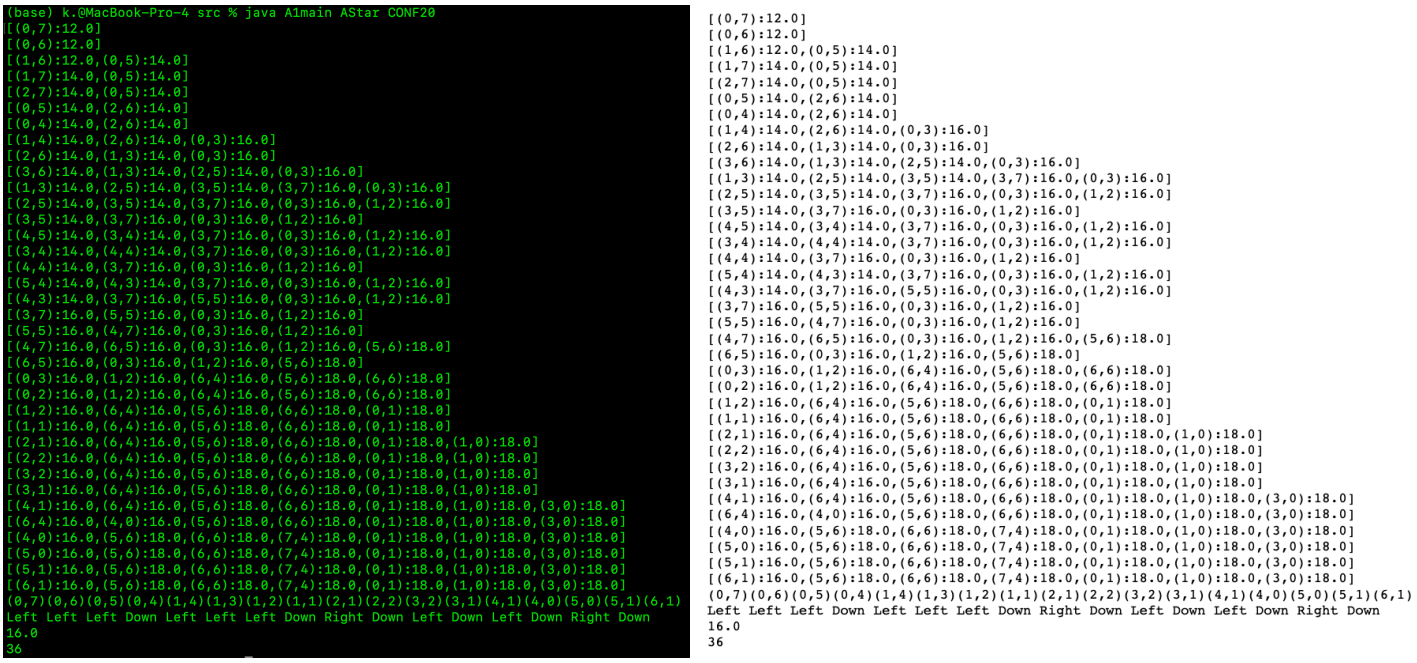

Figure 4. Example run: my output (left) vs required output (right)

Why 100/105 on OtherMaps?

To learn why I didn't pass all tests, I will demonstrate one example failure to explain the cause.

When running AStar on CONF3, even though the path result is correct, by comparing the output line by line, one can find that the following two lines are different (upper one being test output, below being mine):

```
[(3,1):8.0,(2,2):8.0,(2,4):8.0,(5,3):10.0,(4,4):10.0,
(3,5):10.0,(5,1):10.0,(4,0):10.0]
```

```
[(3,1):8.0,(2,4):8.0,(2,2):8.0,(5,3):10.0,(4,4):10.0,
(3,5):10.0,(5,1):10.0,(4,0):10.0]
```

The only difference is the order of (2,2) and (2,4) despite having the same priority. By tracing back, it is found that they are both reached via the same action by going upward, which equates their tie-breaking value. Even further, both their parents (3,2) and (3,4) have the same depth, which makes their depths equal as well. Ergo, in cases like this, the priority queue randomly selected a sequence, which is what caused the difference between the two outputs. This

caused the remaining 4 test failures as well despite having the exact same result print.

Evaluations and Conclusion

The two main criteria used to evaluate the algorithms' performance is: optimality (i.e. whether it returns the shortest path) & number of nodes visited to find a path. Completeness is not discussed because it doesn't apply here.

Criterion 1. Optimality:

The table below lists the optimality of each algorithm after comparison of example path length results on CONF20.

	BFS	DFS	BestF	AStar	BiDirectional
Best Path	**16**	**16**	**16**	**16**	**16**
Found Path	**16**	**24**	**20**	**16**	**16**
Optimal?	**Yes**	**No**	**No**	**Yes**	**Yes**

BFS, BiDirectional Search using BFS and AStar showed optimal results running on all configured maps, while DFS and BestF failed to do so. The reason BFS-based algorithms give promising result is because for this scenario, all the path costs are equal to one and therefore by progressing level-by-level, it will always guarantee a path with minimum steps and cost. DFS failed because it could start its journey on the wrong path to begin with, the goal found is simply a working path.

BestF Search however can be stuck in a local optima because it doesn't take past cost into account and a node that's heuristically closer to the goal may not even be on the optimal path. On the contrary, AStar considers the past costs

together with an admissible heuristic, which guarantees an estimate that's always smaller than the optimal path cost.

Criterion 2. Nodes visited:

The table below lists the number of nodes visited for all 5 algorithms on 5 example runs:

Nodes Visited to Reach Goal					
	BFS	DFS	BestF	AStar	BiDirectional (BFS)
JCONF00	24	8	8	14	23
CONF0	34	16	11	11	30
CONF5	22	13	11	13	20
CONF15	74	83	26	44	80
CONF20	50	28	22	36	40
Average	**41**	**30**	**16**	**24**	**39**

The bar chart below visualizes the number of nodes visited on different maps by different algorithms:

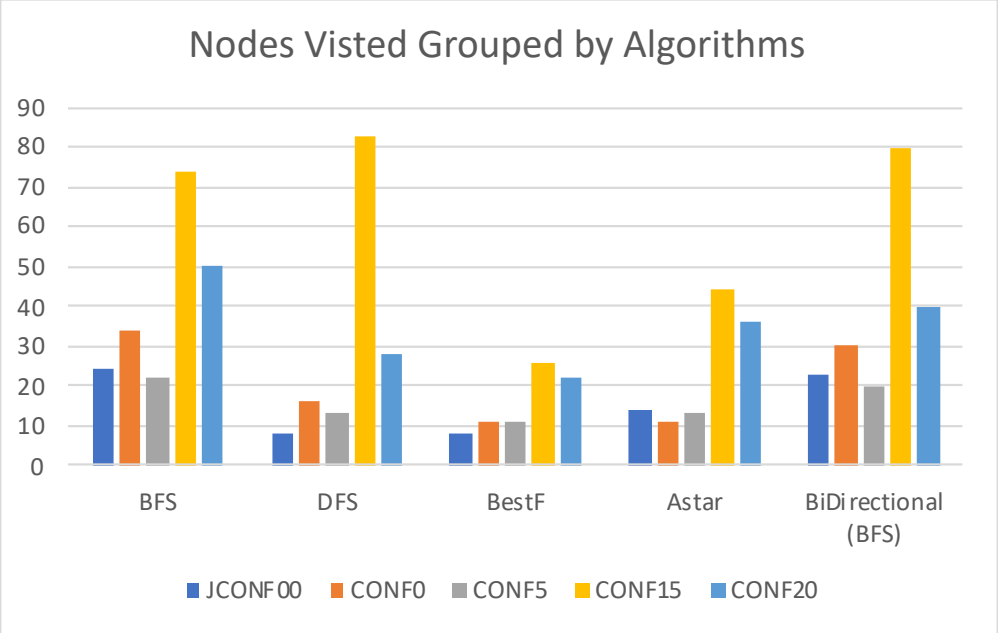

Figure 5. Bar chart of nodes visited on 5 configurations grouped by algorithms

From comparison, informed search algorithms tend to traverse significantly fewer nodes to find a path in most situations and on average. BiDirectional search also demonstrates slight improvement in this regard comparing to the original BFS.

To investigate further, I have grouped the results by map configurations:

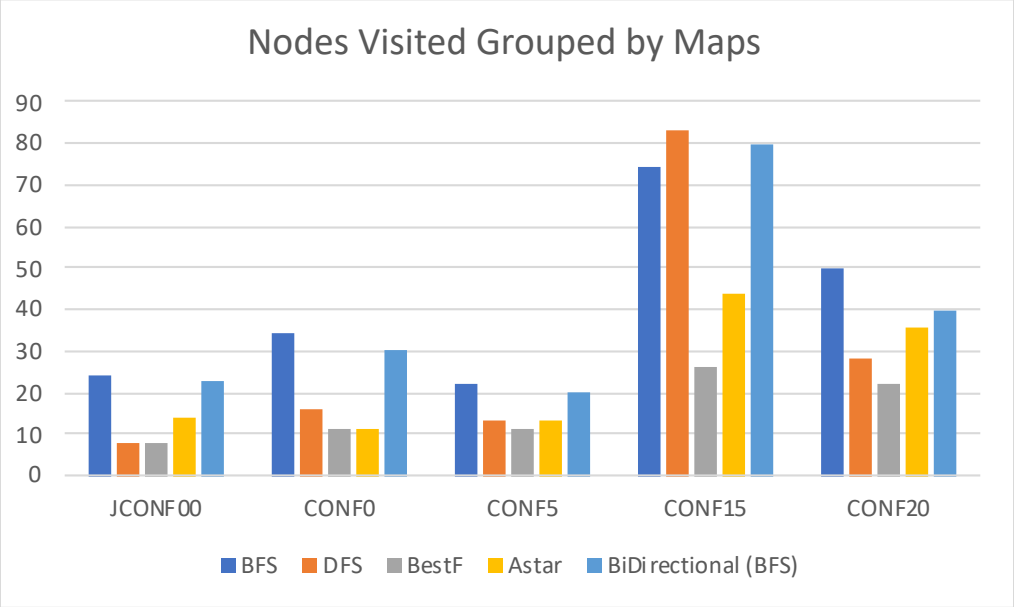

Figure 6. Bar chart of nodes visited by 5 algorithms grouped by configurations

From the visualization, the visited nodes for uninformed search shows stark increase as the map gets more complex. The algorithms react negatively as the complexity of the scenario elevates, whereas informed search generally responds reasonably.

Conclusion:

In conclusion, the implementation of all five algorithms work perfectly fine as requested. The five "test failures" occur only under BestF and AStar runs where priority queue is involved, due to the random order decided by PQ when all three tie-breaking strategies are tied. The five algorithms respond differently to the tasks and while most return optimal solutions, DFS and BestF aren't always able to. Uninformed search (BFS, DFS, BiDirectional) perform poorly in terms of nodes visited in complex scenarios, whereas informed search finds the goal visiting significantly fewer nodes consistently in different situations. While

BFS-based BiDirectional implementation shows slight improvement upon the original algorithm, further studies can be done by investigating its effectiveness on informed search algorithms such as AStar search.

Bibliography

Amit Patel (2007) “Distances on a triangular grid”. *Red blob game’s blog. https://simblob.blogspot.com/2007/06/distances-on-triangular-grid.html*

Mehak Kumar (2023) “PriorityQueue in Java”. *GeeksforGeeks.* https://www.geeksforgeeks.org/priority-queue-class-in-java/

Oliver Charlesworth (2011) “How to iterate over a PriorityQueue”. *StackOverflow.* https://stackoverflow.com/questions/8129122/how-to-iterate-over-a-priorityqueue

Unknown Author (2023) “Stack Class in Java”. *GeeksforGeeks.* https://www.geeksforgeeks.org/stack-class-in-java/

Priya Pedamkar (2021) “Bidirectional Search”. *EDUCBA. https://www.educba.com/bidirectional-search/*