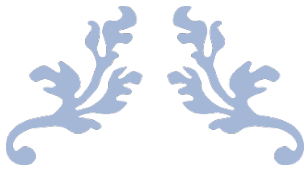

CS5011 A1

200010781

FEBRUARY 9, 2022

Contents

1 Introduction

This report describes the first assignment for CS5011 module. General information can be seen in the Table 1 below.

Table 1 General information of the assignment

Assignment Number	Assignment 1
Date of Submission	9 February 2022
Student ID	200010781
Word Count	1989 (excluding tables)

There are 3 ferry agents in this assignment, including basic, intermediate, and advanced ferry agents. The completion of each agent is described as Table 2 below.

Table 2 Completion of each part

Basic Agent	BFS	attempted, fully working
	DFS	attempted, fully working
Intermediate Agent	Best-first Search	attempted, fully working
	A*	attempted, fully working
Advanced Agent	Bidirectional	attempted, fully working

About running the source code, several steps are needed.

1. Enter the terminal.
2. Change to the CS5011-A1 directory.
3. Compile all source code

   *javac *.java help/*.java informed/*.java startCode/*.java uninformed/*.java*
4. Run the source code

   java A1main <DFS|BFS|AStar|BestF|BiD> <ConfID>

   For example, if I want to use A* to find a path in JCONF00 problem, I should use

   java A1main AStar JCONF00

2 Design and Implementation

2.1 Problem Overview

This assignment addresses a marine navigation plan problem, which aims to find a route from departure port to destination port in efficient ways. In AI agents, we use PEAS model which stands for performance measure, environment, actuator, and sensor to categorize similar agents together. In this assignment, the PEAS model is described as Table 3 below.

Table 3 PEAS model for assignment 1

Agent	**Performance Measure**	**Environment**	**Actuator**	**Sensor**
Marine Navigation Planner System	Path cost, States visited	Sea, Land	Moving Left, Down, Right and Up	Eyes

2.2 System Architecture

All source code can be found in the src directory. The main function is addressed in A1main.java file which locates in src. There are five packages in the src directory.

- **startCode**
  All start code provided by the lecturer can be found at startCode package.
- **uninformed**
  BFS and DFS algorithms can be seen in the BFS class and DFS class respectively.
- **informed**
  Best first search and A* can be seen in the BestFS class and AStar class respectively.
- **advanced**
  The BiD class is addressed for bidirectional search algorithm
- **help**
  Node class is used to represent each node in the searching problem. The State class is used to represent the state of each node. mSharedMethods class includes some shared methods for all searching algorithms. Therefore, all five searching algorithms inherit the SharedMethods class.

  The structure of submitted src directory is shown as Picture below.

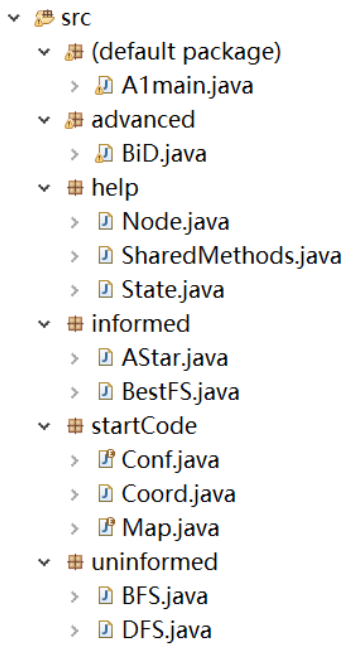

Figure 1 The structure of the Src directory

2.3 Implementation

According to the problem requirements, an important thing that needs to consider for the agent is how to know which type of triangle the agent is on. According to the specification, I find that there is a relationship between the cell coordinate and the triangle type. If the sum of the row and column number of a cell is an even number, the cell locates in an upwards triangle. Otherwise, it locates in a downwards triangle.

As for the four directions allowed, moving left can be represent as c+1 and moving right can be represent as c-1, while moving up can be represent as r-1 and moving downwards can be represent as r+1.

Using objected oriented programming, I used several classes in implementation.

2.3.1 Auxiliary Classes

All auxiliary classes can be found in help package.

- **Node class**

There are six attributes in a node, including state, single step cost, path cost, H cost, F cost, and parent node.

- **State class**

A state records row and column number.

- **SharedMethods class**

SharedMethods includes the methods that can be used for all search

algorithms. Therefore, all search algorithms inherit this class. The details of methods are described as below.

- checkCorrect(): This method is used for checking input validity
- makeNode(): This method is used for making new node.
- successorFN(): This method is used to search successor nodes of a visited node. It is obvious that both moving right and moving left are allowed in all triangles, and the only difference for upwards triangle and downwards triangle is whether moving down or moving up is allowed. This means I need to check the triangle type before searching for successor nodes above and below. Besides, I checked the boundary issue before searching for successor. When the agent moves up or left, it is necessary to check x-1 (row number -1), or y-1 (column number -1) is greater than -1. Similarly, when the agent moves down or right, checking x+1 (row number + 1), or y+1 (column number + 1) is smaller than N-1. All successor nodes are recorded in the nextStates list.
- checkNotExist(): After finding all successor nodes, it is important to check whether these nodes are already in the frontier or the exploredStates lists. This method focuses on this functionality. If a successor node does not exist in the two lists, this method returns Boolean True.
- orderedSuccessors(): Before adding all successor nodes into the frontier, it is necessary to order the successor nodes to follow tie-breaking strategy. This method focuses on this issue, finding nodes in clockwise order. It created a new list named orderedSuccessors, and add successor nodes in right, down, left and up order. At the end, this method returns orderedSuccessors list.
- printStates(): This method is used to print the frontier in each step.
- printResult(): This method is used to print full path after finding a solution.

2.3.2 Basic Agent

This agent locates in the uninformed package.

◆ **BFS**

Following the pseudocode from slides, BFS searches a path using breadth first strategy. Therefore, a Queue is needed for representing the frontier. However, to minimize the changes between DFS and BFS, I use a List<Node> to represent frontier, but it works like a queue. In order to follow the first in first out (FIFO) rule, I get node whose index is 0 in the frontier when visiting and add new nodes at the end of the list.

To avoid loops and redundant paths, another list exploredStates is used to record the states visited in the searching process. If a success node is already in the exploredState list or the frontier list, the node cannot be

added into frontier.

BFS uses treeSearch() method to control the searching process. Firstly, In expand() method, we use successorFN() to find successor nodes. Secondly, before adding nodes into frontier, we use checkNotExist() method to check whether successor nodes are already visited and in the frontier and orderedSuccessors() to order the successor nodes following tie-breaking strategy. If a solution is found, BFS uses printResult() to print full path. Otherwise, BFS prints 'fail'.

◆ **DFS**

The searching process of DFS is similar as BFS. The difference between these two searching algorithms is that DFS follows depth first strategy. Therefore, a Stack is needed for representing the frontier, instead of a Queue. As mentioned before, to minimize the changes between DFS and BFS, I use the same data structure List<Node> to represent frontier, but it works like a stack in DFS. In this implementation, the beginning of the list is the bottom of the stack, and the end of the list is the top of the stack. Following the first in last out (FILO) rule in stack, I get node whose index is frontier.size()-1 in the frontier when visiting and add new nodes at the end of the list.

Similar as BFS, exploredStates list is also used in DFS.

The searching process of DFS is also similar as the process of BFS, except the orderedSuccessors() process. This is because DFS uses stack to record nodes to be visited. When ordering successor nodes into clockwise order and add into a stack, the order will change into anticlockwise, which breaks tie-breaking strategy. Therefore, DFS pass an argument "DFS" when calling orderedSuccessors(), which will revise the successor nodes into anticlockwise order so that it can be added into a stack in a clockwise order.

2.3.3 Intermediate Agent

The code for this part can be found at informed package. Heuristic knowledge used in these informed search algorithms are Manhattan distance. Since there is no requirements on which Manhattan distance we should use, I implement two methods for computing the distance, estimateCost() and estimateCostTri(). The first one computes the ordinary Manhattan distance, while the other one computes the distance on triangles. By default, I use the ordinary one for evaluation.

◆ **Best First Search**

This algorithm uses heuristic cost as searching strategy. It uses similar data structure as uninformed search algorithms described before.

The general searching process is also similar as BFS, but best first

search chooses the node with the lowest F cost. If there are multiple nodes have the same lowest F cost, the algorithm will choose the first one among them. Because of this characteristic, this searching algorithm does not need to order the successor nodes after finding all successor nodes. Another approach is insert into all successors and sort the frontier. However, the first approach can save some time, so I choose the first approach.

The printing process is slightly different from the two uninformed search algorithms since the F cost of each node need to be printed. Therefore, I override the printStates() method from SharedMethod, and print H cost of each node in the frontier since H cost is F cost in best first search approach.

- **A***

A* uses same data structure and similar searching process as best first search algorithm. The difference between best first search and A* is the definition of F cost. As mentioned before, best first search uses H cost as F cost. Different from it, A* uses H cost plus path cost as F cost. Therefore, I override the printStates() method from SharedMethod, and print F cost of each node in the frontier.

2.3.4 Advanced Agent

This locates in the advanced package.

- **Bidirectional Search Algorithm**

If knows the start node and goal node, bidirectional search algorithm can be used. Unlike BFS and DFS, the search begins simultaneously from start point and goal point and ends when the two searches meet.

In order to improve the efficiency, I decide to use heuristic bidirectional search, which select node with the lowest F cost in the frontier. I use isIntersecting() method to check if there is a intersecting node in the two searching process. If there is such node, the agent prints the full path. I use two list to record the nodes that will be visited. List frontier is used for search from start to the goal, while list endier is used for search from the opposite direction. When printing these lists, I use ”Front” and “End” to distinguish the two different process.

2.3.5 Special Situations

There are some special situations are considered in this assignment. All these situations are considered in this assignment, and relevant test cases are shown in part 3.

- Departure port is the destination port, such as start coordinate (1,1) and goal coordinate (1,1) are the same.
- Departure port or destination port is land. For example, a cell (2,3) is a land, which marks ‘1’, but start coordinate is the same as this cell.

- Departure port or destination port is out of the boundary. For example, there is a 3*3 map, but start coordinate is (4,4).

3 Test Summary

In this assignment, I tested all 32 tests and selected 3 cases to show in this report.

Table 4 Test results for JCONF05

Tests on JCONF05

Strategy	Input and Output
BFS	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BFS JCONF05 [(0,0)] [(0,1),(1,0)] [(1,0),(0,2)] [(0,2),(1,1)] [(1,1),(1,2)] [(1,2)] fail 6
DFS	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main DFS JCONF05 [(0,0)] [(1,0),(0,1)] [(1,0),(0,2)] [(1,0),(1,2)] [(1,0),(1,1)] [(1,0)] fail 6
BestF	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BestF JCONF05 [(0,0):4.0] [(0,1):3.0,(1,0):3.0] [(1,0):3.0,(0,2):2.0] [(1,0):3.0,(1,2):1.0] [(1,0):3.0,(1,1):2.0] [(1,0):3.0] fail 6
A*	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main AStar JCONF05 [(0,0):4.0] [(0,1):4.0,(1,0):4.0] [(1,0):4.0,(0,2):4.0] [(0,2):4.0,(1,1):4.0] [(1,1):4.0,(1,2):4.0] [(1,2):4.0] fail 6
BiD	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BiD JCONF05 Front[(0,0):4.0] End[(2,2):4.0] fail 2

Table 5 Test results for CONF7

Tests on CONF7

Strategy	Input and Output
BFS	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BFS CONF7 [(2, 0)] [(2, 1), (3, 0)] [(3, 0), (1, 1)] [(1, 1), (3, 1)] [(3, 1), (1, 0)] [(1, 0), (4, 1)] [(4, 1), (0, 0)] [(0, 0), (4, 2), (4, 0)] [(4, 2), (4, 0), (0, 1)] [(4, 0), (0, 1), (4, 3)] [(0, 1), (4, 3)] [(4, 3), (0, 2)] [(0, 2), (4, 4), (3, 3)] [(4, 4), (3, 3), (0, 3)] [(3, 3), (0, 3)] [(0, 3), (3, 4)] [(3, 4), (0, 4)] [(0, 4), (2, 4)] [(2, 4), (1, 4)] (2, 0) (3, 0) (3, 1) (4, 1) (4, 2) (4, 3) (3, 3) (3, 4) (2, 4) 8.0 19
DFS	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main DFS CONF7 [(2, 0)] [(3, 0), (2, 1)] [(3, 0), (1, 1)] [(3, 0), (1, 0)] [(3, 0), (0, 0)] [(3, 0), (0, 1)] [(3, 0), (0, 2)] [(3, 0), (0, 3)] [(3, 0), (0, 4)] [(3, 0), (1, 4)] [(3, 0), (1, 3)] [(3, 0), (2, 3)] [(3, 0), (2, 4)] (2, 0) (2, 1) (1, 1) (1, 0) (0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (1, 4) (1, 3) (2, 3) (2, 4) 12.0 13
BestF	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BestF CONF7 [(2, 0):4.0] [(2, 1):3.0, (3, 0):5.0] [(3, 0):5.0, (1, 1):4.0] [(3, 0):5.0, (1, 0):5.0] [(1, 0):5.0, (3, 1):4.0] [(1, 0):5.0, (4, 1):5.0] [(4, 1):5.0, (0, 0):6.0] [(0, 0):6.0, (4, 2):4.0, (4, 0):6.0] [(0, 0):6.0, (4, 0):6.0, (4, 3):3.0] [(0, 0):6.0, (4, 0):6.0, (4, 4):2.0, (3, 3):2.0] [(0, 0):6.0, (4, 0):6.0, (3, 3):2.0] [(0, 0):6.0, (4, 0):6.0, (3, 4):1.0] [(0, 0):6.0, (4, 0):6.0, (2, 4):0.0] (2, 0) (3, 0) (3, 1) (4, 1) (4, 2) (4, 3) (3, 3) (3, 4) (2, 4) 8.0 13
A*	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main AStar CONF7 [(2, 0):4.0] [(2, 1):4.0, (3, 0):6.0] [(3, 0):6.0, (1, 1):6.0] [(1, 1):6.0, (3, 1):6.0] [(3, 1):6.0, (1, 0):8.0] [(1, 0):8.0, (4, 1):8.0] [(4, 1):8.0, (0, 0):10.0] [(0, 0):10.0, (4, 2):8.0, (4, 0):10.0] [(0, 0):10.0, (4, 0):10.0, (4, 3):8.0] [(0, 0):10.0, (4, 0):10.0, (4, 4):8.0, (3, 3):8.0] [(0, 0):10.0, (4, 0):10.0, (3, 3):8.0] [(0, 0):10.0, (4, 0):10.0, (3, 4):8.0] [(0, 0):10.0, (4, 0):10.0, (2, 4):8.0] (2, 0) (3, 0) (3, 1) (4, 1) (4, 2) (4, 3) (3, 3) (3, 4) (2, 4) 8.0 13

BiD	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BiD CONF7 Front[(2,0):4.0] End[(2,4):4.0] Front[(2,1):3.0, (3,0):5.0] End[(3,4):5.0, (2,3):3.0] Front[(3,0):5.0, (1,1):4.0] End[(3,4):5.0, (1,3):4.0] Front[(3,0):5.0, (1,0):5.0] End[(3,4):5.0, (1,4):5.0] Front[(1,0):5.0, (3,1):4.0] End[(1,4):5.0, (3,3):4.0] Front[(1,0):5.0, (4,1):5.0] End[(1,4):5.0, (4,3):5.0] Front[(4,1):5.0, (0,0):6.0] End[(4,3):5.0, (0,4):6.0] Front[(0,0):6.0, (4,2):4.0, (4,0):6.0] End[(0,4):6.0, (4,4):6.0, (4,2):4.0] (2,0) (3,0) (3,1) (4,1) (4,2) (4,3) (3,3) (3,4) (2,4) 8.0 16

Table 6 Test results for CONF12

Tests on CONF12

Strategy	Input and Output
BFS	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BFS CONF12 [(5, 0)] [(5, 1), (4, 0)] [(4, 0)] [(4, 1)] [(4, 2), (3, 1)] [(3, 1), (4, 3)] [(4, 3), (3, 0)] [(3, 0), (4, 4)] [(4, 4), (2, 0)] [(2, 0), (4, 5)] [(4, 5), (2, 1)] (5, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) 6.0 11
DFS	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main DFS CONF12 [(5, 0)] [(4, 0), (5, 1)] [(4, 0)] [(4, 1)] [(3, 1), (4, 2)] [(3, 1), (4, 3)] [(3, 1), (4, 4)] [(3, 1), (4, 5)] (5, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) 6.0 8
BestF	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BestF CONF12 [(5, 0):6.0] [(5, 1):5.0, (4, 0):5.0] [(4, 0):5.0] [(4, 1):4.0] [(4, 2):3.0, (3, 1):5.0] [(3, 1):5.0, (4, 3):2.0] [(3, 1):5.0, (4, 4):1.0] [(3, 1):5.0, (4, 5):0.0] (5, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) 6.0 8
A*	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main AStar CONF12 [(5, 0):6.0] [(5, 1):6.0, (4, 0):6.0] [(4, 0):6.0] [(4, 1):6.0] [(4, 2):6.0, (3, 1):8.0] [(3, 1):8.0, (4, 3):6.0] [(3, 1):8.0, (4, 4):6.0] [(3, 1):8.0, (4, 5):6.0] (5, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) 6.0 8
BiD	C:\Users\DaisyDai\Desktop\JAVA-StAndrews\S2\5011_A1\src>java A1main BiD CONF12 Front[(5, 0):6.0] End[(4, 5):6.0] Front[(5, 1):5.0, (4, 0):5.0] End[(4, 4):5.0, (3, 5):7.0] Front[(4, 0):5.0] End[(3, 5):7.0, (4, 3):4.0] Front[(4, 1):4.0] End[(3, 5):7.0, (4, 2):3.0] Front[(4, 2):3.0, (3, 1):5.0] End[(3, 5):7.0, (4, 1):2.0] (5, 0) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) 6.0 10

Considering the special situations discussed in 2.3.5, I created five new test cases, which are shown in Table 7 below.

Table 7 Test results for special situations

ID	Description	Output		
JCONF06	Departure is the destination	BFS,DFS: [(0, 0)] (0, 0) 0.0 1	BestF, A*: [(0, 0):0.0] (0, 0) 0.0 1	BiD: Front[(0, 0):0.0] End[(0, 0):0.0] (0, 0) 0.0 2
JCONF07	Departure is a land	BFS, DFS, BestF, A*, and BiD: Error! Departure or destinations is a land.		
JCONF08	Destination is a land			
JCONF09	Departure out of bounds	BFS, DFS, BestF, A*, and BiD: Error! Departure or destinations out of bounds.		
JCONF10	Destination out of bounds			

4 Evaluation and Conclusions

I record the path cost of the route and the number of search states explored for evaluation. The data is shown in the Table 8 and Table 9. It is obvious that both BFS and A* can find the optimal solution which has the lowest path cost. This is because the cost for each move is uniform, and we use admissible heuristic strategy. The solution found by DFS is not optimal, but it visits less nodes to find a solution in most cases comparing with BFS. The solution found by best first search is not optimal in most situations either. However, it is directed, which means this algorithm visits the minimum number of nodes to get a solution among all four algorithms. In this assignment, I use best first search in both searching process of bidirectional search. In most cases, the bidirectional can find the optimal solution. Comparing with BFS, bidirectional search visits less nodes to get a solution.

Table 8 Path costs of solutions in different algorithms

CASES	BFS	DFS	BEST-F	A*	BID
JCONF00	7.0	7.0	7.0	7.0	7.0
JCONF01	1.0	1.0	1.0	1.0	1.0
JCONF04	2.0	4.0	2.0	2.0	2.0
JCONF05	-	-	-	-	-
CONF0	10.0	14.0	14.0	10.0	14.0
CONF1	10.0	20.0	14.0	10.0	14.0
CONF2	10.0	14.0	10.0	10.0	10.0
CONF3	8.0	16.0	10.0	8.0	8.0
CONF4	8.0	12.0	8.0	8.0	8.0
CONF9	7.0	7.0	7.0	7.0	7.0
CONF13	12.0	12.0	12.0	12.0	12.0
CONF15	18.0	24.0	20.0	18.0	20.0
CONF16	5.0	19.0	5.0	5.0	5.0
CONF23	15.0	21.0	21.0	15.0	15.0
CONF24	17.0	19.0	17.0	17.0	17.0

Table 9 The number of nodes explored in different algorithms

CASES	BFS	DFS	BEST-F	A*	BID
JCONF00	24	8	8	17	16
JCONF01	2	2	2	2	4
JCONF04	5	5	3	3	4
JCONF05	6	6	6	6	2
CONF0	34	16	16	26	30
CONF1	33	25	15	21	30
CONF2	33	16	11	21	22
CONF3	28	21	11	22	18
CONF4	32	28	10	21	20
CONF9	16	10	8	15	10
CONF13	21	21	18	21	20
CONF15	74	83	24	60	42
CONF16	14	20	6	9	8
CONF23	49	46	22	38	34
CONF24	46	23	21	40	22

In conclusion, informed search algorithms use heuristic strategy to reach goal, so it usually visits less nodes than uninformed search algorithms. In this Marine Navigation Planner System, BFS and A* algorithms produce the best routes on the basis of path cost. Although best first search algorithm cannot find the best path, it is directed and visits the minimum number of search states. Performance of bidirectional search tend to follow the search strategy used in both searching. In this assignment, the bidirectional search is based on best

first search, which is more directed than BFS. I assume that the performance of a bidirectional search based on BFS can achieve completeness and optimal for this problem.