CS5011 – REPORT

220026989

1.Introduction

This report wholesomely summarises the A1Main courswork implementation submitted. It should be noted that all the terminologies, functions and classes are well commented in the javascript file and therefore the same is not repeated in the report.

i. Checklist of implementation.

- ✓ Basic Ferry Agent

  Breadth First Search and Depth First Search implemented, working and completely passing the basic test cases provided. The compilation and running of Basic Ferry Agent is the same as been asked in the requirements.

- ✓ Intermediate Ferry Agent

  Best First Search (Greedy Search) and A Star search implemented, working and passes 98 out of the 105 test cases provided by the supervisor. Apart from the advanced test cases all the results for the basic tests matches accurately with the results provided by the supervisor. The failure in the case of seven test cases will be discussed in detail in the later part of this report.

- ✓ Advanced Ferry Agent

  Two advanced Search methods are implemented:

  - An informed Bidirectional search is implemented with a cost function taking snippets of code/ideas from A Star search and is working as expected.
  - An informed triple agent search with heuristics is implemented and is working as expected.

ii. Compilation instructions.

- After navigating into the src folder :
  - *Compilation :*
    >>javac A1main.java
  - *Running the code:*
    >>java A1main <Algo> <ConfId>
- Apart from running the program function, I have introduced another class where the program prints through all the available configuration ids using all the algorithms implemented. This was made to

easily collect data inorder to assess any configuration with any algorithm or whenever every algorithm has to be run :
For this to work there are some careful commented code that needs to be uncommented. The compilation and running to receive all the data is:

- *Compilation :*
  >>javac A1mainforalgorithmdata.java
- *Running the code:*
  >>java A1mainforalgorithmdata all

2.Design and Implementation

i. Project structure:

- **A1main.java :** This class is the main class of the program. All the search functions are defined in the A1main.java class
- **AlgorithmData.java :** This class has the main properties whose data are collected for evaluation. (For example: number of nodes visited, the time required and the cost for completion by a specific algorithm for a specific configuration.
- **Conf.java :** An enum class containing all the configurations.
- **Coord.java :** A class that has the functions and parameters for a coordinate
- **Map.java :** An enum class that stores all the maps for the program.
- **Node.java :** This class has all the properties required for a node.
- **NodeInformed.java :** A class that extends the Node.java class. This Node class contains more parameters compared to the Node class that is necessary to implement informed search.
- **Search.java :** This class handles the main functions that are used by the A1main functions to reach the goalstate.
- **A1mainForAlgorithmdata.java –** class is similar to the A1main class, but executes all of the algorithms for all the configurations and finally prints an array model of required values. (Not essential but required for data retrieval)

ii. Peas Model:

- **Performance indicator :** The algorithms' objective is to determine the route from a given start state to a particular goal state. The length of the path, which is equal to the number of edges in the path, serves as the performance indicator.
- **Environment:** In this case, the environment is the graph map made into a triangular grid. The agent has to move from the start triangular node to the goal triangular node by various set of constraints :
  - The agent is not supposed to move to the direction where the triangle is pointing
  - Priority of movement is provided as : Right, down, left, up respectively.
  - Agent should not go the islands (islands are marked as 1 and water as 0)

- **Actuators:** Are the actions performed by the agent to move from one node to the other and it will differ according to the algorithm followed.
- **Sensors:** The sensors are used to monitor the environment, record the present condition, and maintain track of the states that have been visited.

iii. Implementation overview:

BFS and DFS

The basic program implementation of BFS and DFS is nearly the same except for the data structure used. To general flow of the program:

I. A new search object for the specific search is created.
II. Along with it, a startNode is created and initiated according to the coordinates of the start.
III. Following this, a specific data structure is created inorder to keep track of the state space(frontier). An arraylist of the type Coord class is also created to keep track of the explored coordinates (the coordinates that have already been added to the frontier).
IV. After the basic steps a node is popped once from the frontier (the node popped depends on the data structure used.) which is checked to ensure if it is or it is not the goal node, so that the operation continues.
V. If the popped node is not the goal node, it is then expanded with the help of nodeCreator function, which is checked with the exploredCoord array list for duplicates and is inserted into the frontier.
VI. The nodes are expanded and added to the data structure according to the priority of direction from the popped/parentNode as requested in the requirements.
VII. This process keeps on happening until a goal is met or until the frontier becomes empty.

The breadth first search is implemented with the help of a Queue and follow FIFO rule, whereas the frontier in Depth First Search is implemented with the help of a Stack and follows the LIFO rule.

Peas (BFS):

- **Actuators:** The BFS algorithm performs actions by visiting nodes in a breadth-first manner, meaning it visits all the nodes at a given depth before moving on to deeper nodes.
- **Sensors:** In BFS, we have a frontier that keep tracks of nodes that are to be popped and explored Coord to keep trace of the nodes that have been popped.

By implementing Queue in breadth First search the program does the search breadthwise first before going any deeper, opposite to the case of depth first search where the search initially goes deeper on one direction.

Peas (DFS):

- **Actuators:** The DFS algorithm performs actions by visiting nodes in a depth-first manner, meaning it visits all the nodes depth-wise first.
- **Sensors:** In DFS, we have a frontier that keep tracks of nodes that are to be popped and explored Coord to keep trace of the nodes that have been popped.

○ Best First Search

There are a lot of variations from the uninformed search in the case of informed search. In case of best first search we order the frontier with respect to the **Manhattan distance**(heuristics, mentioned as **h_distance** in the program). Eventhough the basic steps of implementation remain the same, there are some extra features:

Actuators:

I. Triangle coordinates of all the nodes are to be calculated at each loop, after which the Manhattan distance for each node is found out. All of these are done by the **calcH_distance()** function, and is stored in the nodeInformed object.

II. The second major difference would be the rearranging of the nodes once all the new popped/expanded nodes are added to the frontier. The frontier is then arranged according to the least Manhattan distance value with direction from the parent node as the tie breaker further extending to arrange according to the least depth if the former values are the same.

- **Sensors:** In best first search, we have a frontier that keep tracks of nodes that are to be popped and explored Coord to keep trace of the nodes that have been popped.

Here, I have used an arraylist instead of a priority queue, since priority queue makes the sorting and rearranging easier. With the arraylist I have to write down every line of code for sorting to be enlightened upon the basic concepts.

○ A* Search

Most of the functions and working of A star is similar to Best first search with some differences. The A star search uses **f_cost function** = heuristics + depth instead of just Manhattan distance. Similar to the Best first search Manhattan distance is calculated and stored in the NodeInformed object using calcH_distance(). The f_cost function is then calculated with the help of calcF_cost().

Actuators:

I. Whenever the new nodes are created by popping/expanding a node the new nodes are then compared to the frontier to check if there are any nodes with the same coordinate(s). If there are any, they are then compared with respect to the f_cost and interchanged if the new node has less cost. If the cost is the same, then they are compared with respect to the depth. Finally the frontier may change accordingly after comparing it with the new nodes.Once the new nodes are added after common processes the frontier is arranged according to the f_cost in ascending order. The process carries on until the frontier becomes empty or the goal node is found.

Sensors: In best first search, we have a frontier that keep tracks of nodes that are to be popped and explored Coord to keep trace of the nodes that have been popped.

○ Bidirectional with A star

Usage : java A1main BidirectionalAStar ConfId

Actuators:
This search method follows the A star algorithm but A star algorithm is implemented from both the start Node and goal Node moving towards each other according to the cost function.

The search method has two A star duplicated methods one from the startNode moving to the goal node, and the other from the goal node moving to the startNode. Every method done in A star algorithm is done twice here. The main difference in the code is that, in bidirectionlAstar the goalCheck is between the currentNode and the reverseCurrentNode that are popped at a time. The goalCheck is also done between the currentNode and reverseCurrentNode with the exploredNodes (exploredNode and exploredNodeReversed) to check if the intersection has already been passed.

Sensors: Here we have two frontiers, one taking nodes from the startNode(frontier) and the other adding nodes from the goalNode(reverseFrontier). Furthermore, inorder to keep track of the explored coordinates from both the directions we keep exploredCoord and exploredCoordReverse.

○ Triple Agents with BestFS

USAGE : java A1main tripleAgentBestFS Confid

Actuators:
This search method follows the Best first Search algorithm but implements 3 agents. One main agent which moves through the frontier. While the other two agents(adjacentAgentOne and adjacentAgentTwo) are always adjacent to the currentNode/popped node is randomly allocated.
This triple agent search is implemented in such a way that it adopts itself to become a 2 agent best first search when the there are less expanded nodes to occupy. Furthermore the frontier can change between adjacentNode moving onto each other when there are no nodes available to explore.

The analogy could be said similar to a 3-legged machine, wherein the middle leg has the controller and the two legs are always adjacent. When there is no path to move front the controller can shift to any one of the legs and then move forward, making it the central leg and the rest its adjacent leg. The program can change from 3 legged agent to a single agent(adaptive) when there is a shortage of nodes available. Furthermore the final result varies each time, since the allocation of the adjacent nodes are done in a random manner.

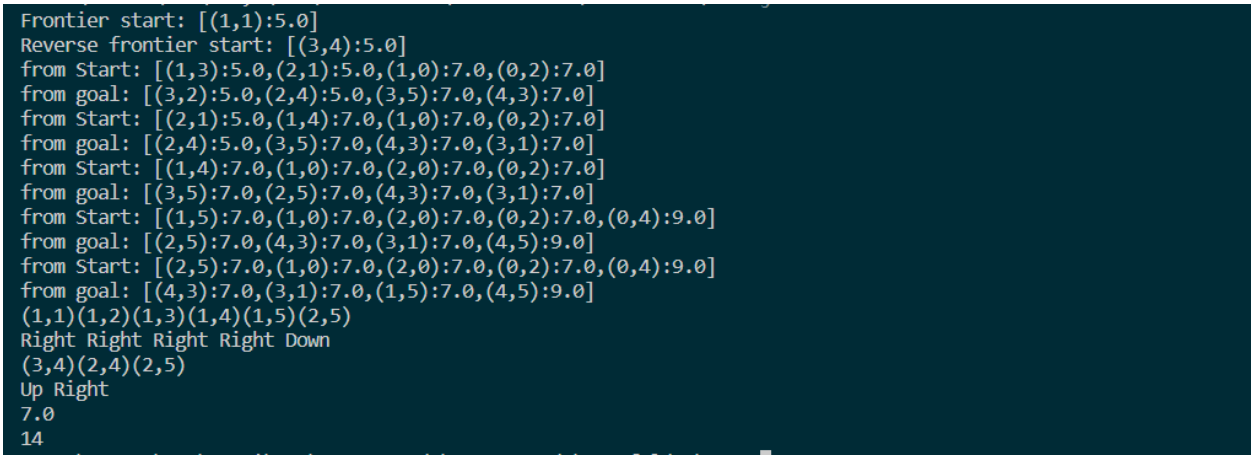
Sensors: Here we have two frontiers to keep track of the nodes visited. Frontier and frontiertrackerCoord. While frontier covers the nodes popped/expanded by the main agent, frontierTracker covers those coordinates through which the main agent solely moves. This is done because the main agent can move freely in this algorithm. To keep check on these movements and to avoid infinite loops we need frontierTrackerCoord.

Working:

Whenever the new nodes are expanded the adjacentNodes are made to adapt according to the number of nodes available. Since the adjacent nodes are made to allocate randomly without any specific priority each time the result received may or may not be slightly different. The program basically follows the BestFirst search algorithm and the goal state is reached if either the main agent reaches the goal state or either of the two agents.

It is to be noted that although the depth may increase depending on the adjacent random selection, the cost is optimum. In this case, the cost is only calculated when the main agent moves. Since the adjacent agents only follow the main agent, the cost is not added for the adjacent agents occupying the nodes.

- **Bidirectional AStar sample output (JCONF00) – Advanced Ferry Agent**
  >> java A1main BidirectionalAStar JCONF00



- **TripleAgentBestFS sample output (JCONF00) – Advanced Ferry Agent**

>> java A1main tripleAgentBestFS JCONF00

```
StartNode(1,1)
Agent1 :(2,1)
Agent2 :(1,0)
Agent1 :(1,3)
Agent2 :(0,2)
currentNode  :(1,2)
Agent1 :(1,4)
Agent2 :(1,2)
currentNode  :(1,3)
Agent1 :(1,5)
Agent2 :(0,4)
currentNode  :(1,4)
[(2,5):2.0]
Agent1 :(1,5)
Agent2 :(1,5)
currentNode  :(1,5)
[(2,4):1.0]
Agent1 :(2,5)
Agent2 :(2,5)
currentNode  :(2,5)
[(3,4):0.0]
Agent1 :(2,4)
Agent2 :(2,4)
currentNode  :(2,4)
(1,1)(1,2)(1,3)(1,4)(1,5)(2,5)(2,4)(3,4)
Right Right Right Right Down Left Down
7.0
7
```

3.Test summary

Out of the 23 and 105 separate tests, my program was able to pass 23 of the basic tests and 98 of the other tests. The failed test cases are as follows:

- AStarCONF18
- AStarConf19
- AStarConf21
- AStarConf24
- AStarConf23
- AStarJCONF11
- BESTFJCONF11

The reason for the failed test can be seen as:
Whenever 2 nodes/coordinates of the same f_cost and the same depth and the same direction from the parentNode is present in the frontier, they appear to switch places in the next frontier print. I presume it has something to do with the usage of queue. Since I am using arraylist as my primary data structure for frontier, it is not seen, which could be the reason of failure, even though everything else appears to be correct.

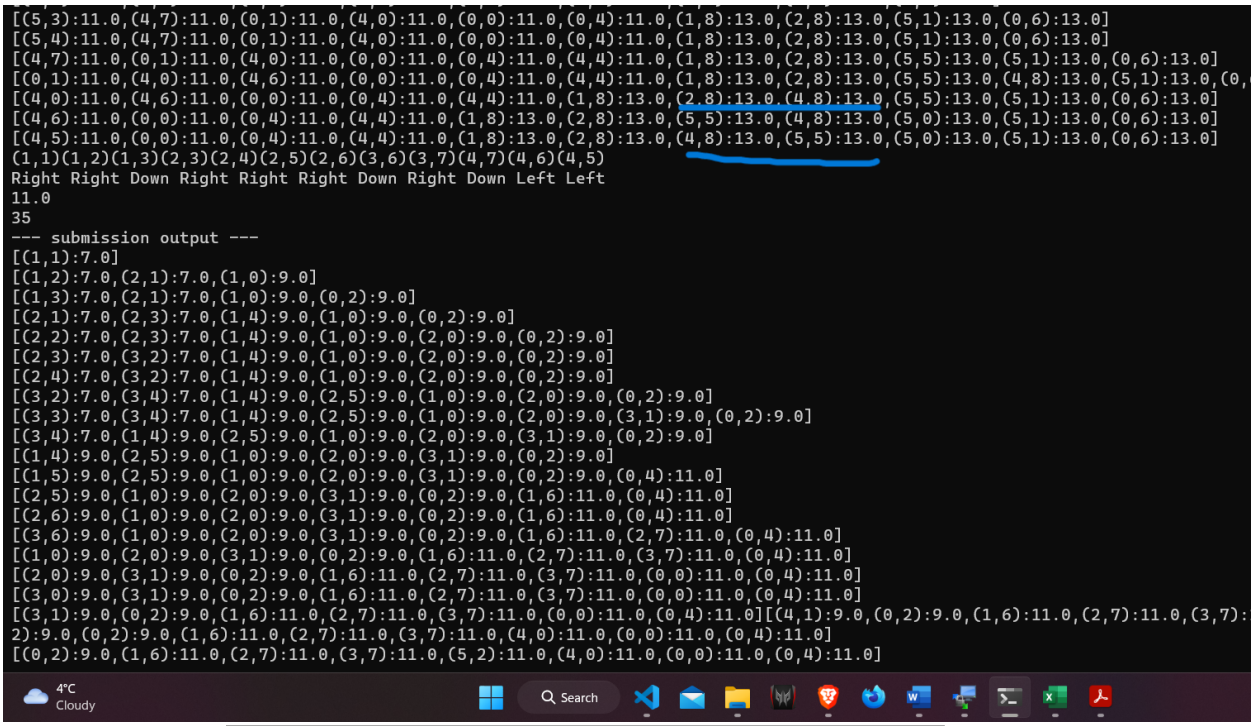


fig: expected result for AStarConf18

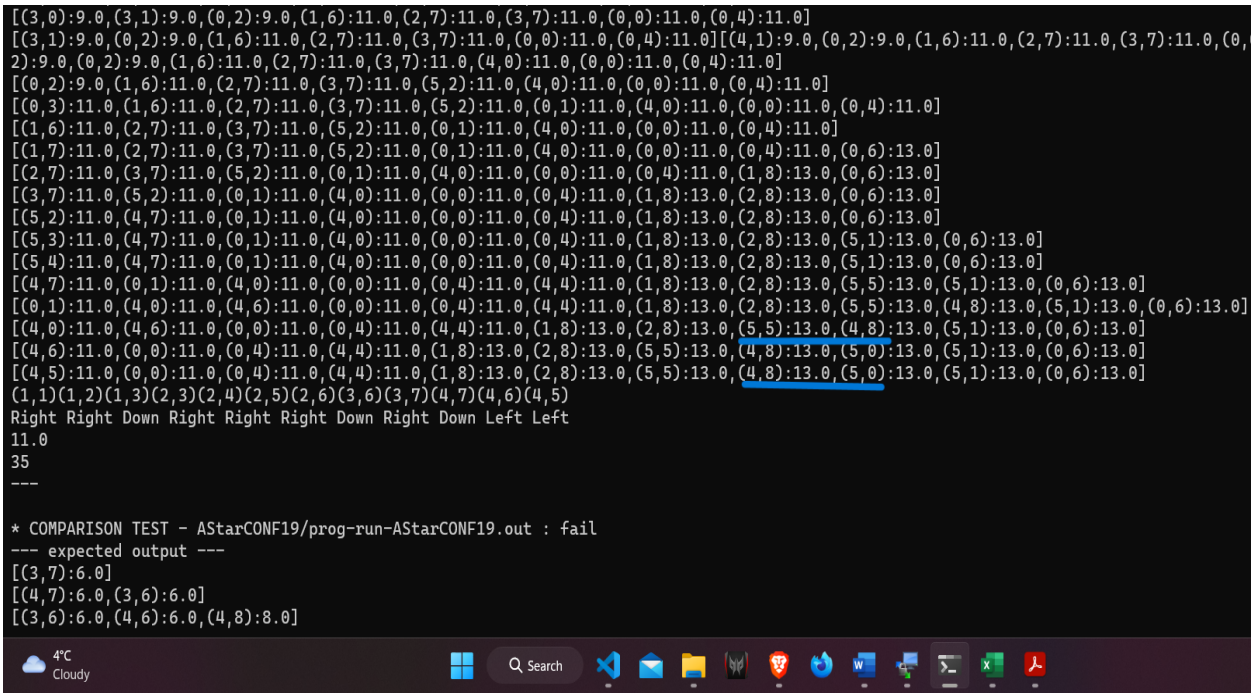


Fig: submitted output

The same case has been seen appearing for all the failed test cases.

3.1 Test Examples

	BFS	DFS	BestF	AStar	BidirectionalBFS
JCONF00	(1,1)(1,2)(1,3)(1,4)(1,5)(2,5)(2,4)(3,4) Right Right Right Right Down Left Down 7.0 24	(1,1)(1,2)(1,3)(1,4)(1,5)(2,5)(2,4)(3,4) Right Right Right Right Down Left Down 7.0 8	(1,1)(1,2)(1,3)(1,4)(1,5)(2,5)(2,4)(3,4) Right Right Right Right Down Left Down 7.0 8	(1,1)(2,1)(2,0)(3,0)(3,1)(3,2)(3,3)(3,4) Down Left Down Right Right Right Right 7.0 14	from start : (1,1)(1,2)(1,3)(1,4)(1,5)(2,5) Right Right Right Right Down from goal : (3,4)(2,4)(2,5) Up Right 7.0 14
JCONF11	Fail 34	Fail 34	Fail 34	Fail 34	Frontier start: [(1,1):3.0] Reverse frontier start: [(2,3):3.0] fail 0

JCONF01	(1,1)(1,2) Right 1.0 2	(1,1)(1,2) Right 1.0 2	(1,1)(1,2) Right 1.0 2	(1,1)(1,2) Right 1.0 2	from start : (1,1)(1,2) Right from goal : (1,2) 1.0 4

➢ Data Analysis

The number of nodes traversed, cost and time taken for algorithm completion for each algorithm for each configuration is compared. (Data retrieved with **A1mainForAlgorithmdata.java**)

Table 1:

			Nodes traversed					
Conf	BFS	DFS	BestFS	Astar	Bidirectional	Triple agent 1	TripleAgent2	TripleAgent3
1	24	8	8	14	14	8	8	12
2	34	34	34	34	34	33	33	33
3	2	2	2	2	2	1	1	1
4	2	2	2	2	2	2	1	1
5	8	7	5	5	5	7	5	7
6	5	5	3	3	3	3	3	3
7	6	6	6	6	6	6	6	6
8	34	16	11	11	11	13	13	13
9	33	25	11	11	11	10	10	10
10	33	16	11	11	11	10	10	10
11	28	21	9	14	14	11	11	9
12	32	28	9	15	15	9	14	9
13	22	13	11	13	13	11	11	11
14	20	12	13	16	16	12	12	12
15	19	13	11	12	12	13	13	13
16	17	16	9	9	9	8	8	8
17	16	10	14	14	14	2	8	14
18	1	1	1	1	1	0	0	0
19	1	1	1	1	1	0	0	0
20	11	8	8	8	8	7	8	8

21	21	21	13	20	20	7	7	13
22	23	20	20	19	19	12	10	12
23	74	83	26	44	44	26	29	29
24	14	20	6	7	7	5	5	5
25	64	13	13	19	19	13	13	13
26	55	62	14	35	35	14	14	16
27	51	46	9	16	16	11	9	11
28	50	28	22	36	36	21	21	21
29	52	49	21	33	33	21	21	26
30	42	46	27	35	35	20	20	27
31	49	46	22	33	33	17	17	17
32	46	23	23	36	36	22	23	22

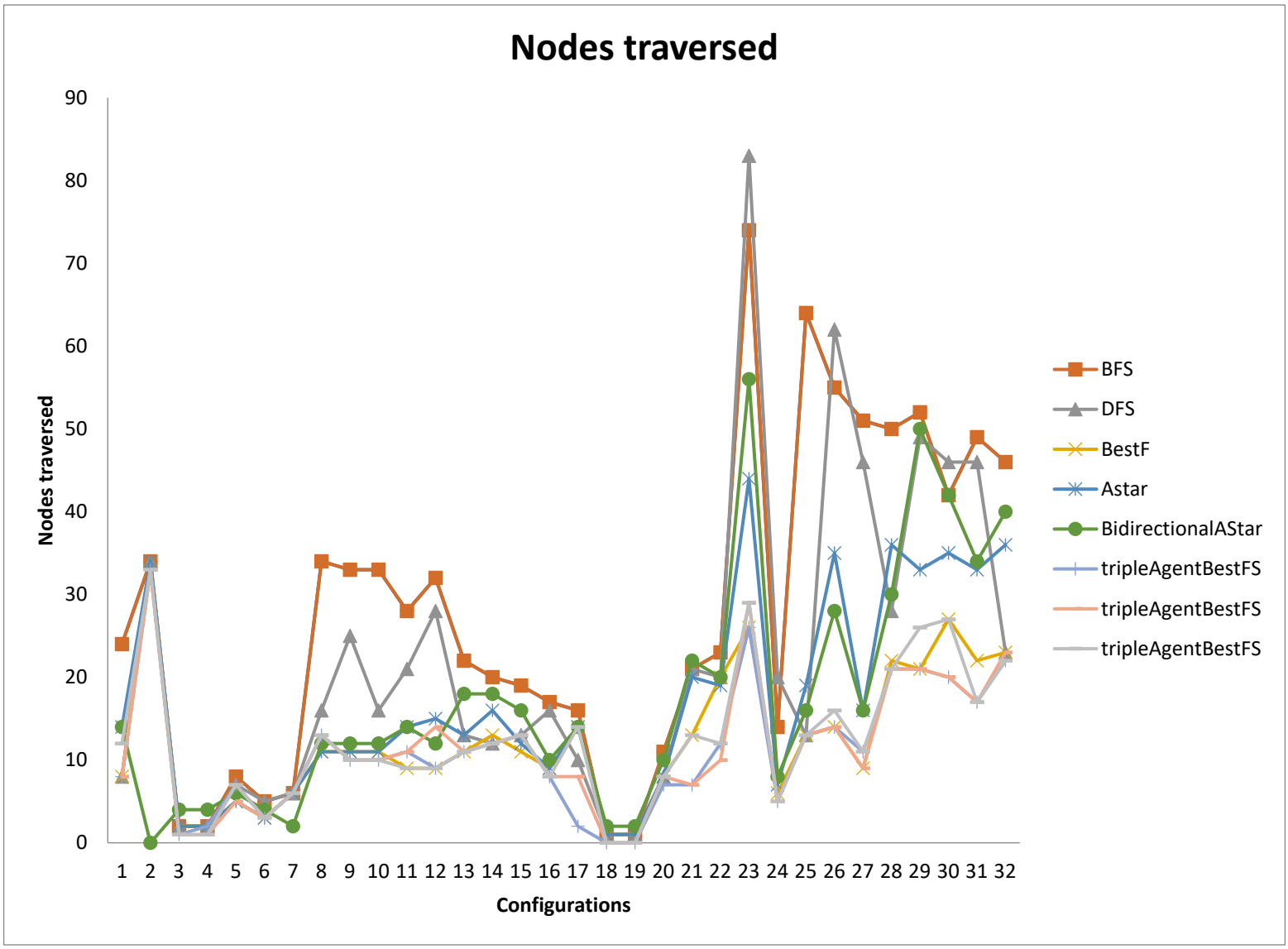


Table 2:

Cost

			Cost					
Conf	BFS	DFS	BestFS	Astar	Bidirectional	Triple agent1	tripleAgentBestFS 2	tripleAgentBestFS 3
1	7	7	7	7	7	7	7	11
2	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
5	4	6	4	4	4	6	4	6
6	2	4	2	2	2	2	2	2
7	0	0	0	0	0	0	0	0
8	10	14	10	10	10	12	12	12
9	10	20	10	10	10	10	10	10

10	10	14	10	10	10	10	10	10
11	8	16	8	8	8	10	10	8
12	8	12	8	8	8	8	12	8
13	10	12	10	10	10	10	10	10
14	10	10	12	10	10	12	12	12
15	8	12	8	8	8	12	12	12
16	8	14	8	8	8	8	8	8
17	7	7	13	7	7	0	7	13
18	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0
20	6	6	6	6	6	6	6	6
21	12	12	12	12	12	0	0	12
22	11	11	11	11	11	11	0	11
23	18	24	22	18	18	22	26	26
24	5	19	5	5	5	5	5	5
25	11	11	11	11	11	11	11	11
26	11	41	11	11	11	11	11	15
27	8	28	8	8	8	10	8	10
28	16	24	20	16	16	20	20	20
29	17	23	17	17	17	17	17	21
30	16	16	22	16	16	18	18	22
31	15	21	21	15	15	15	15	15
32	17	19	21	17	17	21	21	21

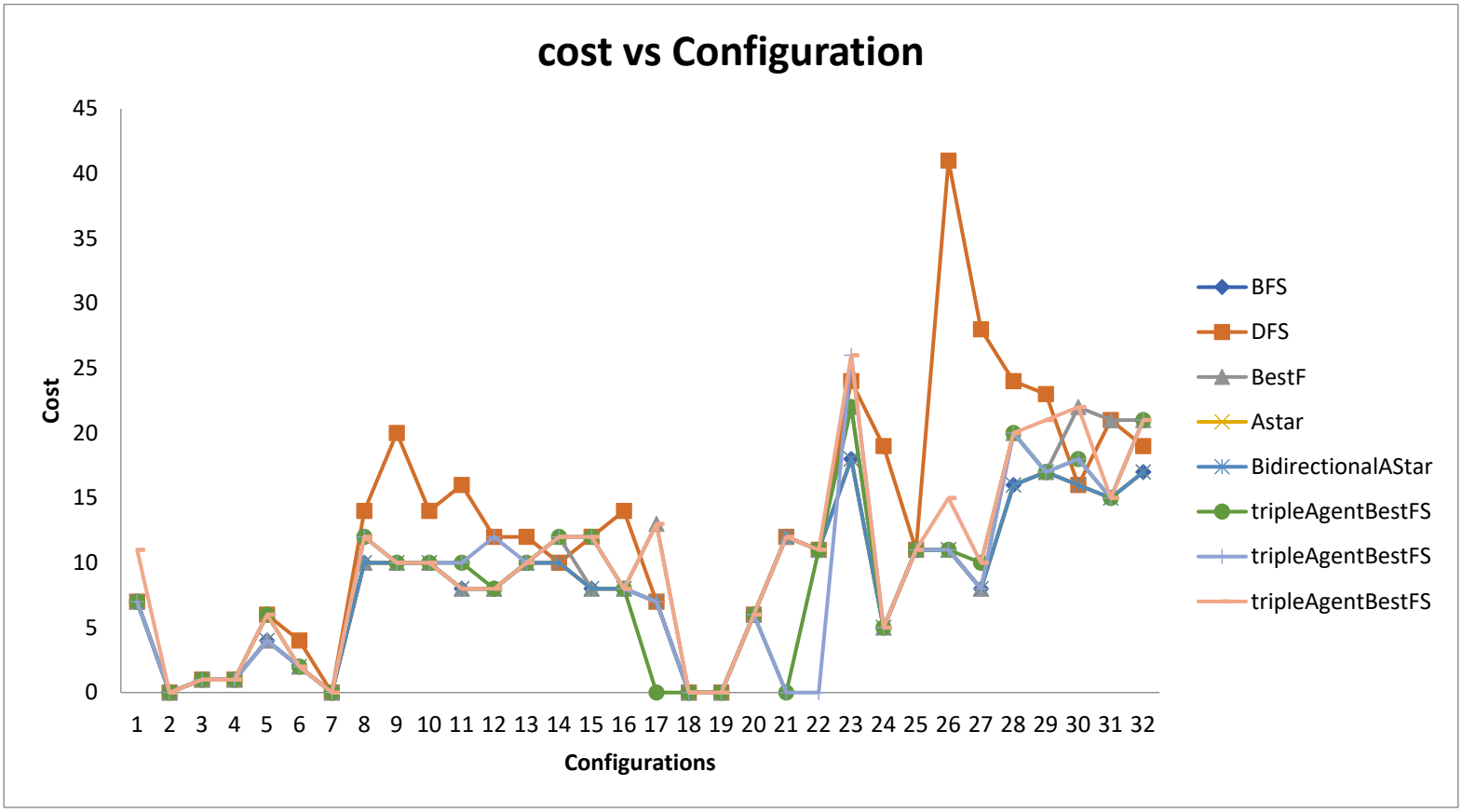
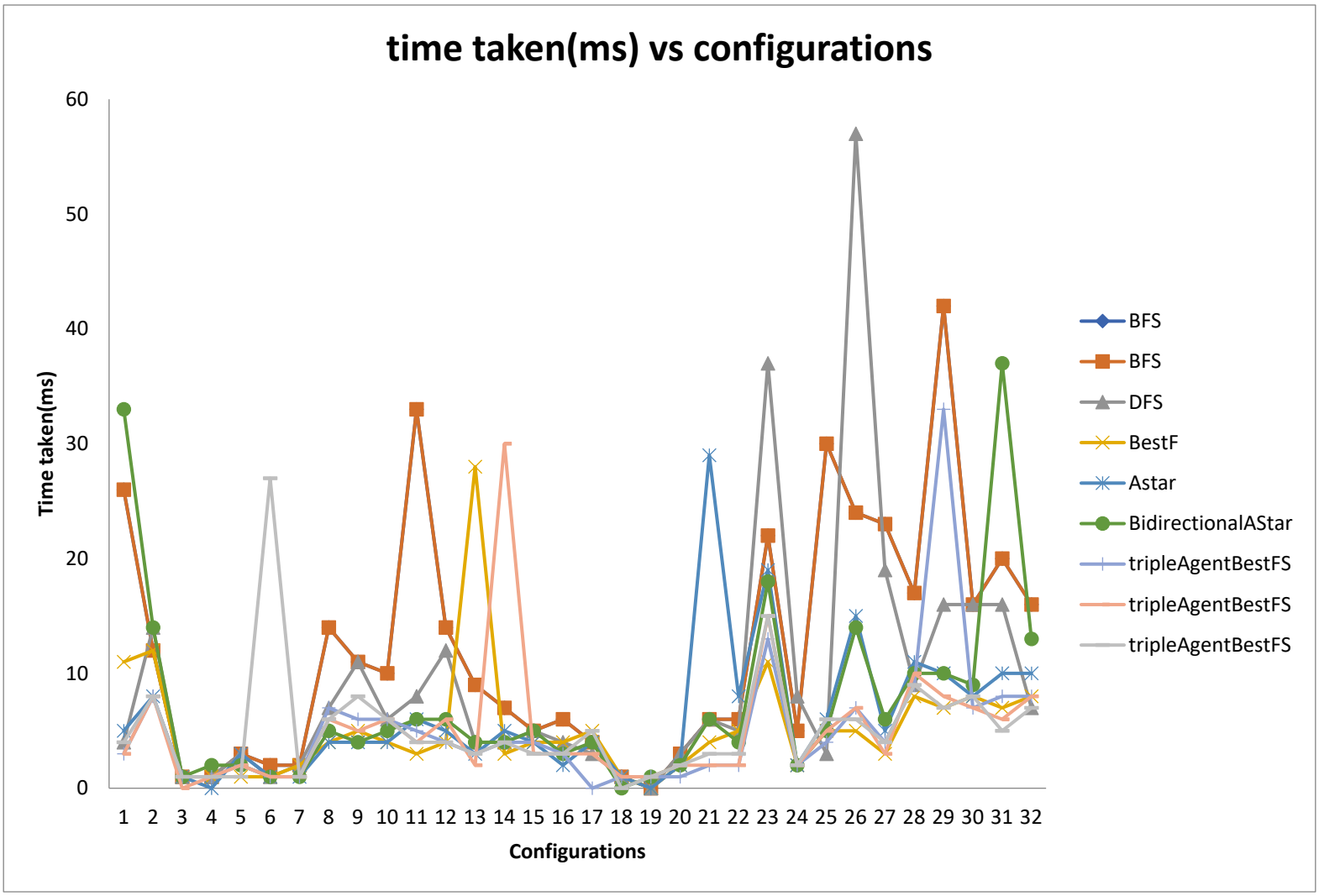


Table 3: Time taken for completion

			Cost					
Conf	BFS	DFS	BestFS	Astar	Bidirectional	Triple agent	tripleAgentBestFS	tripleAgentBestFS
1	26	4	11	5	33	3	3	4
2	12	14	12	8	14	8	8	8
3	1	1	1	1	1	1	0	1
4	1	1	1	0	2	1	1	1
5	3	3	1	3	2	2	2	1
6	2	1	1	1	1	1	1	27
7	2	2	2	1	1	1	1	1
8	14	7	4	4	5	7	6	6
9	11	11	5	4	4	6	5	8
10	10	6	4	4	5	6	6	6
11	33	8	3	6	6	5	4	4
12	14	12	4	5	6	4	6	4
13	9	4	28	3	4	3	2	3
14	7	4	3	5	4	4	30	4
15	5	5	4	4	5	4	3	3
16	6	4	4	2	3	3	3	3
17	4	3	5	4	4	0	3	5
18	1	1	1	1	0	1	1	0
19	0	0	0	0	1	1	1	1
20	3	3	2	2	2	1	2	2
21	6	6	4	29	6	2	2	3
22	6	5	5	8	4	2	2	3
23	22	37	11	19	18	13	15	15
24	5	8	2	2	2	2	2	2
25	30	3	5	6	5	4	5	6
26	24	57	5	15	14	7	7	6
27	23	19	3	5	6	4	3	4
28	17	9	8	11	10	9	10	9
29	42	16	7	10	10	33	8	7
30	16	16	8	8	9	7	7	8
31	20	16	7	10	37	8	6	5
32	16	7	8	10	13	8	8	7



- ## Evaluation

➢ **Performance:**

To summarize over how each of the algorithms performed, looking at the nodes Traversed graph, the uninformed search algorithms both BFS and DFS has poor performance compared to other algorithms as expected.

In case of informed search, A star and bidirectional graphs nearly similar, although the performance of A star is better than bidirectional A star as observed from the graph.
All other informed search show better performance than the rest, with Best first search and triple agent going through the lowest number of nodes. There are 3 triple agent graphs as, the adjacent agents are chosen random, and 3 are taken to observe the difference between them.

➢ **Time taken:**

Eventhough the time taken by each algorithm depends on the code, w.r.t the coursework submitted, triple agent Best first, best first are seen to show the least amount of time taken. Bidirectional A star although shows low time taken in the initial stage, spikes up at the end

- **Cost:**
  In case of cost effectiveness it can be said without a doubt that BFS is the most optimal solution staying below every other algorithms quickly followd by bidirectional Astar.

Conclusion

The shortest path can be found using BFS (Breadth-First Search), however it can be slow.because before going to the next level, it is necessary to store and visit each node at a certain depth. DFS (Depth-First Search) is quicker than BFS, but if used improperly, it can result in long pathways and loops. A-Star is an intelligent search algorithm that prioritises nodes to search for and utilises a heuristic function to estimate the cost of achieving the goal. If the heuristic is acceptable and consistent, it can find optimal solutions and is more effective than BFS. Nodes are prioritised using Best-First Search according to the evaluation function. It can be put into practise utilising various evaluation functions, including A*. In contrast to a regular BFS, bidirectional BFS does two simultaneous BFS searches, one from the start and one from the target, which can drastically shorten search time. But Bidirectional was unable to narrow its search space due to the tie-breaking restrictions in the problem statement.