CS5011 Artificial Intelligence Practice: Assessment 1 Search

170004680

University of St. Andrews

February 2021

Contents

A Introduction **1**
A.1 Checklist . . . 1
A.2 Compilation & Execution Instructions . . . 1

B Design, Implementation and Evaluation (Word Count: 1524) **2**
B.1 Design and Implementation . . . 2
B.1.1 PEAS Model & Search Components Description . . . 2
B.1.2 Implementation Of The Search Algorithms . . . 2
B.1.3 Successor Function Details . . . 3
B.1.4 Advanced Agent: Alternate/Extension Heuristic . . . 3
B.2 Evaluation . . . 4

C Test Summary **11**
C.1 Basic Agent Testing . . . 11
C.2 Intermediate Agent Testing . . . 15
C.3 Advanced Agent (Bidirectional Search) Testing . . . 18
C.4 Advanced Agent (Alternate/Extension Heuristic) Testing . . . 19

D Bibliography **23**

A Introduction

In this assessment, several search algorithms have been implemented to find solutions for the described *Coastguard Rescue Simulation*. In this simulation, a robot is required to search for a path from a person who requires rescuing to a position of safety amongst the Giant's Causeway.

Uninformed (depth-first and breadth-first) and informed (best-first and A*) search algorithms have been implemented to find solutions to various configurations of this problem. Additionally, the bidirectional search (BDS) algorithm has been implemented for comparison. An alternate/extension heuristic has also been implemented for heuristic comparisons with the informed search algorithms.

The implementations can be executed as instructed in section A.2. The design and implementation of these algorithms is documented in section B.1. The implemented search algorithms are empirically evaluated in section B.2. This evaluation aims to determine the most appropriate of the search algorithms in terms of which produces the shortest paths and with greatest efficiency. Correctness testing of the implementations is documented in section C.

A.1 Checklist

Table 1: Table indicating the completeness of this submission with respect to the specification requirements.

Agent	**Section**	**Status**
Basic	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
Intermediate	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
Advanced (Bidirectional Search)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
Advanced (Additional Heuristic)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.

A.2 Compilation & Execution Instructions

From within the `A1src/` directory, the `A1main` program can be compiled using (see figure 1):

javac A1main.java

After compiling the program in the `A1src/` directory, `A1main` can be executed using (see figure 2):

java A1main <DFS|BFS|AStar|BestF|BDS> <ConfID> [-aH]

<ConfID> is the name of a configuration in the `Conf` class, such as *TCONF00*. Providing '-aH' (optionally) as an argument specifies to use the alternate heuristic for informed search as part of the advanced agent.

Figure 1: The following screenshot shows the program can be compiled within the St. Andrews School of Computer Science lab environment without error. All of the `.java` files have been compiled into `.class` files usable for program execution.

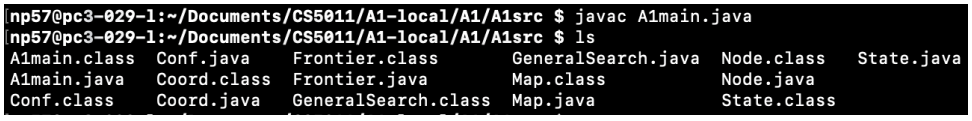


Figure 2: The following screenshot shows the program can be executed within the St. Andrews School of Computer Science lab environment without error. As seen, the selected search algorithm runs on a test configuration.

```
np57@pc3-029-l:~/Documents/CS5011/A1-local/A1/A1src $ java A1main
Usage: 'java A1main <DFS|BFS|AStar|BestF|...> <ConfID> [<any other param>]'
np57@pc3-029-l:~/Documents/CS5011/A1-local/A1/A1src $ java A1main DFS TCONF00
[(1,1)]
[(1,2), (0,1), (0,0), (1,0), (2,1)]
```

B Design, Implementation and Evaluation (Word Count: 1524)

B.1 Design and Implementation

B.1.1 PEAS Model & Search Components Description

The performance of the search algorithms is measured according to the path cost (length) of solutions.

The search environment models the Giant's Causeway as a 2-dimensional, hexagonal grid of uniformly distanced coordinates. The grid is fully observable, static, and deterministic. The laws of the environment are completely understood by the robot.

The robot can perceive any location on the grid to determine the extent the location is traversable. Some locations are barriers and cannot be traversed (out-of-bounds and coordinates marked with '1'), whilst other locations are traversable only in certain directions (coordinates marked with '2' must be traversed eastwards/westwards).

Actions are limited to moving between adjacent coordinates, transitioning the robot from one state to the next. Thus, the state space is all reachable coordinates. The initial state is the coordinate of the person. The goal state is the coordinate of a safe position. The cost of an action is an addition of 1 distance unit to the path cost.

B.1.2 Implementation Of The Search Algorithms

The implementation of the algorithms follows a general search algorithm (GSA). The GSA performs a search by creating a 'frontier' of nodes to be explored. At first, this frontier contains only the initial node (starting state). Then, the following instructions are repeated until the goal state is reached (success) or the frontier is emptied (failure):

- Remove node X from the frontier.
- If X is the goal state, a solution has been found.
- Otherwise, find all states reachable from X (via a successor function) and create a set of new nodes for them.
- Then, merge the set of new nodes into the frontier if the states of the new nodes are not already in the frontier and have not been explored.

Implementations of the search algorithms differ only by how the frontier is managed (how nodes are added, removed, and updated).

The frontier is implemented as a `Collection` of node objects. The frontier behaviour for each algorithm is achieved by instantiating the frontier with a suitable data structure (inheriting the `Collection` class). Using standard data structures is beneficial as they are heavily optimised and more efficient for problems at scale.

The depth-first (DFS) and breadth-first (BFS) search implementations differ only in that BFS instantiates the frontier as a queue (nodes are explored at the same level first), whilst DFS instantiates the frontier as a stack (nodes are explored as far as possible down a branch first).

The best-first (BestF) implementation instantiates the frontier as a priority queue (prioritising by `fCost`). Nodes are created with an `fCost` (cost estimated via a heuristic). The default heuristic used for this *Coastguard Rescue* problem is Manhattan distance over a hexagonal grid.

The A* implementation is almost identical to BestF. The `fCost` of an A* node is given by the heuristic cost for the node added to the path cost up to the node. Also, A* adds to the `expand()` function, which allows for path costs to be updated as shorter paths are found.

The frontier implementation design is beneficial for several reasons. Implementations are simpler and more understandable; a GSA is used for all algorithms with only the frontier behaviour changing. This makes differences between the search algorithms clearer. Also, by minimising implementation differences, an evaluation focused only upon the algorithms can occur (see section B.2).

The BDS implementation also uses the GSA. However, BDS instantiates two search trees: one searching from the start to the goal, the other searching from the goal to the start. Here, BDS uses BFS in both search directions, which is complete and optimal. A solution is found when one search tree finds a solution or a node is present in both search trees. The intersecting node can be used to join both search trees into a solution, as a path through this node is a path from the start to the goal. Search fails when either of the search tree frontiers is emptied.

B.1.3 Successor Function Details

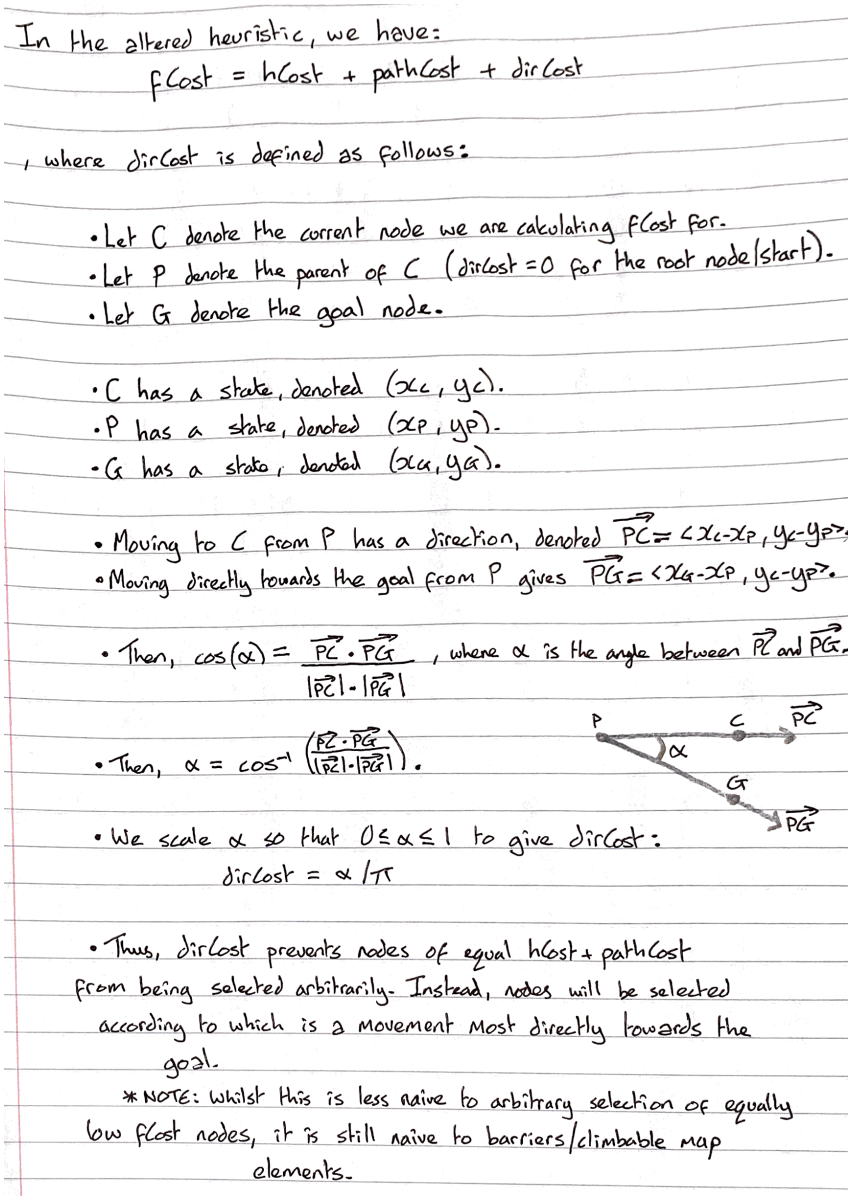
The successor function returns the set of coordinates legally accessible from the current coordinate. If the current coordinate is free-space, then neighbouring coordinates are successor states if they are also free space or climbable. If the current coordinate is marked as climbable, then the robot must traverse east or west. Provided going east or west is not blocked or out-of-bounds, the resulting coordinates are successor states.

Directions are considered starting with the south-east direction. Then, all other directions are considered in an anti-clockwise rotation (tie-breaking strategy). The set of successor states returned is a linked hash set, which preserves the order in which the directions are considered. Note that this tie-breaking strategy is arbitrary and could be improved by considering directions that point towards the goal first.

B.1.4 Advanced Agent: Alternate/Extension Heuristic

The Manhattan distance heuristic is altered for the advanced agent. When using Manhattan distance, it is common for multiple nodes in the frontier to have equal `fCost`. For equally smallest `fCost` nodes, one is explored arbitrarily. The alternate heuristic aims to improve on this by exploring nodes closest to the direction of the goal first. This alternate heuristic is defined more formally in figure 3.

Figure 3: More rigorous definition of the 'direction cost' being added to the `fCost` of nodes. This addition only affects the order in which equal `fCost` nodes will be explored; the node that results in movement most directly towards the goal node is explored first. This extension to the heuristic is less naive to arbitrary exploration of equal `fCost` nodes, but is still naive to map elements such as barriers and climbable positions.



B.2 Evaluation

The path costs (performance) and the number of explored nodes (efficiency) of each implemented search algorithm for each of the provided problem configurations are shown in figure 4.

For uninformed search, figure 5 shows that BFS finds solutions with equal or lower path costs than DFS. This is expected because BFS is optimal whilst DFS is not. DFS only provides optimal solutions when the first solution found is optimal, which commonly occurs because there is just one solution (trivially optimal). Although BFS is preferable for performance, it is less efficient than DFS on average (figure 6).

For informed search, figure 7 shows that A* finds solutions of equal or lower path costs than BestF, which is expected as A* is optimal (admissible heuristic). Figure 8 illustrates there is a trade-off in efficiency for performance; A* requires equal or more explored nodes than BestF for all configurations. Notably, the informed, optimal (A*) and non-optimal (BestF) algorithms improve on the uninformed,

optimal (BFS) and non-optimal (DFS) algorithms respectively for both performance and efficiency on average.

Figure 9 shows that BFS and A* produce the smallest path costs (both are optimal). Figure 10 shows that A* is significantly more efficient on average than BFS, so A* is favourable. BFS actually has the worst efficiency for all configurations. For the non-optimal algorithms, BestF outperforms DFS and is more efficient; BestF produces the best results in terms of efficiency out of all algorithms on average.

BDS (using BFS in both search directions) is optimal, so it is equally the best in terms of performance with BFS and A* (see figure 4). BDS improves on efficiency compared to BFS on average (figure 11). Thus, the trade-off between performance and efficiency experienced by BFS can be reduced on average by adapting BFS into bidirectional search. Despite this, BDS explores more nodes than A* on average, so A* is still favourable. Bidirectional search is advantaged when solutions do not exist. By searching from the goal, less nodes need to be explored on average to reach a state where a solution cannot exist. This is seen with *CONF10*, which has no solutions, as 13.5x fewer nodes was required by BDS than all other algorithms.

In summary, BFS, BDS, and A* give the best performance, though A* is preferable since it is more efficient on average. BestF is the most efficient on average. Interestingly, BestF produced solutions that are just $\sim$7% longer than A* but used $\sim$28% fewer explored nodes on average. Thus, BestF may be a suitable alternative to A* if efficiency is a concern. In practice, it is usually favourable to forgo optimal solutions for efficiency, especially for problems at a larger scale.

The results of using the altered heuristic compared to the default heuristic for informed search is given in figure 12. The figure shows the altered heuristic does not improve performance for BestF. In fact, the altered heuristic slightly decreased efficiency for BestF (figure 13). This is because prioritising nodes in the direction of the goal has a weakness; barriers towards the goal have to be manoeuvred around by moving indirectly to the goal. This is the case with *CONF18*, *CONF21*, *CONF22*, and *CONF24* where the number of nodes required increased. Despite this, the average number of explored nodes decreased for A* by $\sim$14%. A* with the altered heuristic is the best in terms of path cost and second-best in terms of efficiency. Future work should investigate heuristics that are less naive to map obstructions such as barriers.

Several improvements can be made to this evaluation. More configurations should be tested to capture the scope of the problem class; conclusions about the *Coastguard Rescue* problem may be supported only by the limited set of configurations evaluated. Also, larger configurations should be used to capture the behaviour of the algorithms at scale, which represents more practical scenarios. Further work should quantify the differences in performance and efficiency between the algorithms. This will inform on the extent to which one algorithm can be considered as an alternative to another for this search problem.

Figure 4: Table of results indicating the path cost and number of nodes explored as a result of executing each of the implemented search algorithms on all of the provided configurations of the problem.

	Path Cost					Number Of Nodes Explored				
Configuration	DFS	BFS	Astar	BestF	BDS	DFS	BFS	Astar	BestF	BDS
TCONF00	5.0	5.0	5.0	5.0	5.0	6	22	10	6	13
TCONF01	3.0	3.0	3.0	3.0	3.0	4	10	4	4	6
CONF00	5.0	5.0	5.0	5.0	5.0	6	26	6	6	14
CONF01	9.0	5.0	5.0	5.0	5.0	11	31	6	6	13
CONF02	14.0	10.0	10.0	10.0	10.0	25	36	20	11	31
CONF03	7.0	6.0	6.0	6.0	6.0	29	32	14	7	27
CONF04	7.0	4.0	4.0	4.0	4.0	10	27	9	5	12
CONF05	6.0	6.0	6.0	6.0	6.0	7	23	15	7	15
CONF06	7.0	6.0	6.0	6.0	6.0	9	23	12	7	16
CONF07	6.0	4.0	4.0	4.0	4.0	7	18	5	5	15
CONF08	6.0	5.0	5.0	5.0	5.0	7	20	10	7	13
CONF09	8.0	4.0	4.0	5.0	4.0	19	19	9	6	13
CONF10	fail	fail	fail	fail	fail	27	27	27	27	2
CONF11	fail	fail	fail	fail	fail	1	1	1	1	2
CONF12	5.0	5.0	5.0	5.0	5.0	6	13	6	6	13
CONF13	8.0	8.0	8.0	8.0	8.0	13	27	13	9	22
CONF14	8.0	7.0	7.0	8.0	7.0	10	23	16	9	21
CONF15	13.0	12.0	12.0	13.0	12.0	15	27	26	19	29
CONF16	18.0	14.0	14.0	14.0	14.0	21	29	18	15	28
CONF17	10.0	10.0	10.0	10.0	10.0	23	28	27	25	22
CONF18	17.0	14.0	14.0	17.0	14.0	19	29	28	24	27
CONF19	9.0	9.0	9.0	9.0	9.0	20	23	20	14	28
CONF20	10.0	7.0	7.0	10.0	7.0	12	34	16	12	26
CONF21	17.0	16.0	16.0	17.0	16.0	19	40	39	24	34
CONF22	18.0	7.0	7.0	9.0	7.0	21	33	20	14	22
CONF23	10.0	9.0	9.0	10.0	9.0	12	35	23	11	26
CONF24	21.0	10	10.0	12.0	10.0	29	36	28	23	30
	Average	Average	Average	Average	Average	Average	Average	Average	Average	Average
	9.9	7.6	7.6	8.2	7.6	14.4	25.6	15.9	11.5	19.3
Solutions	No Solutions									

Figure 5: Bar chart showing how the path cost of solutions found by the DFS and BFS algorithms compare between the provided configurations of the problem. BFS finds solutions, when they exist, that have an equal or lower path cost than DFS.

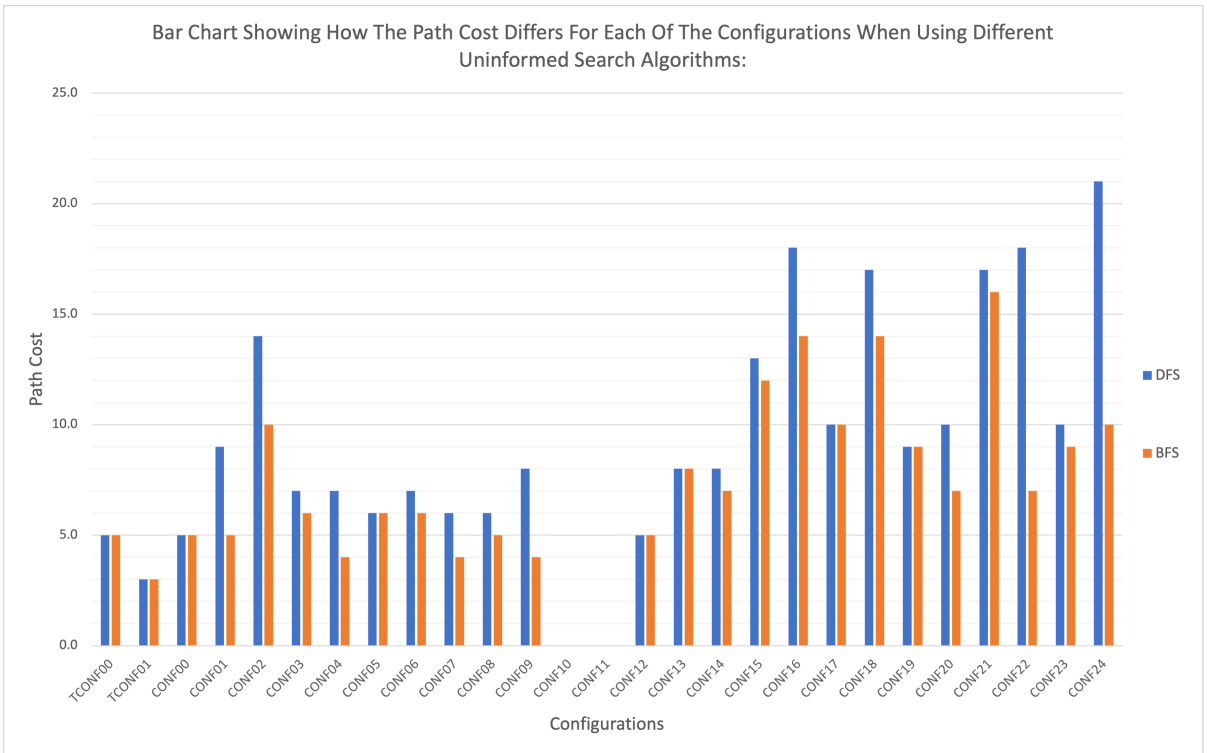


Figure 6: Bar chart showing how the number of nodes explored by the DFS and BFS algorithms compares across the provided configurations of the problem. On average, BFS requires that more nodes are explored in comparison to DFS.

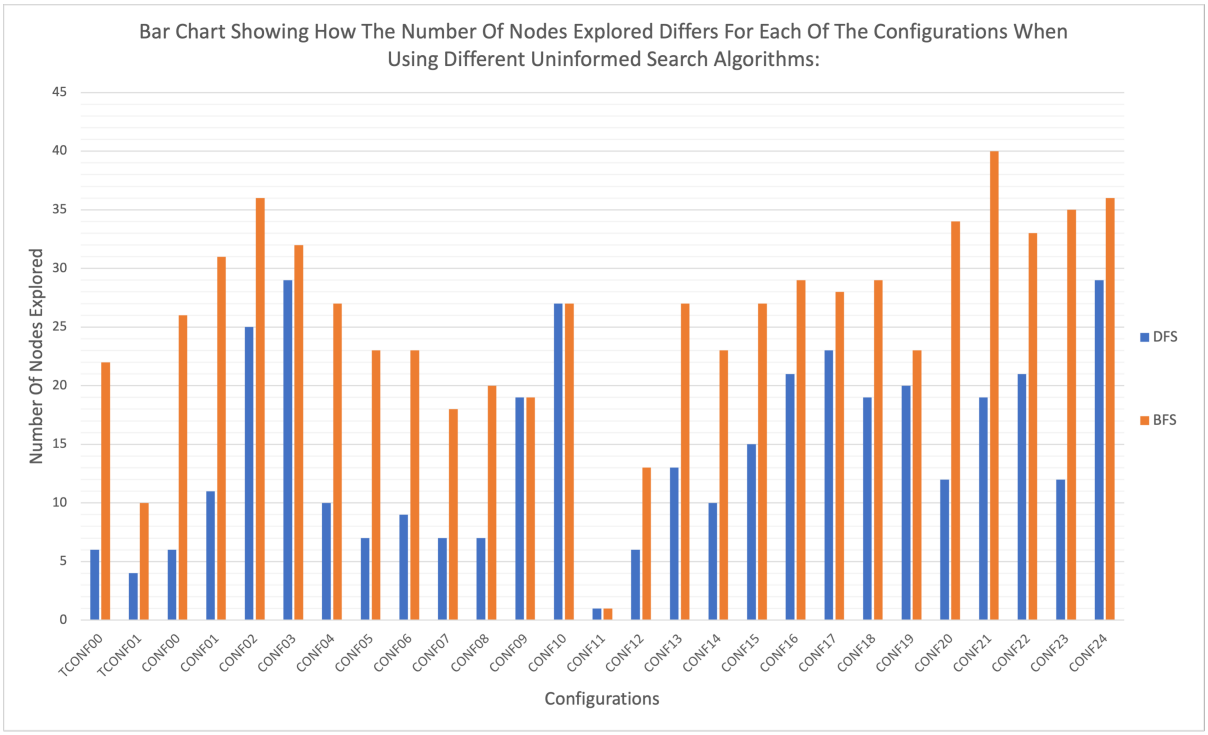


Figure 7: Bar chart showing how the path cost of solutions found by the best-first and A* search algorithms compare between the provided configurations of the problem. A* search finds solutions, when they exist, that have an equal or lower path cost than best-first search.

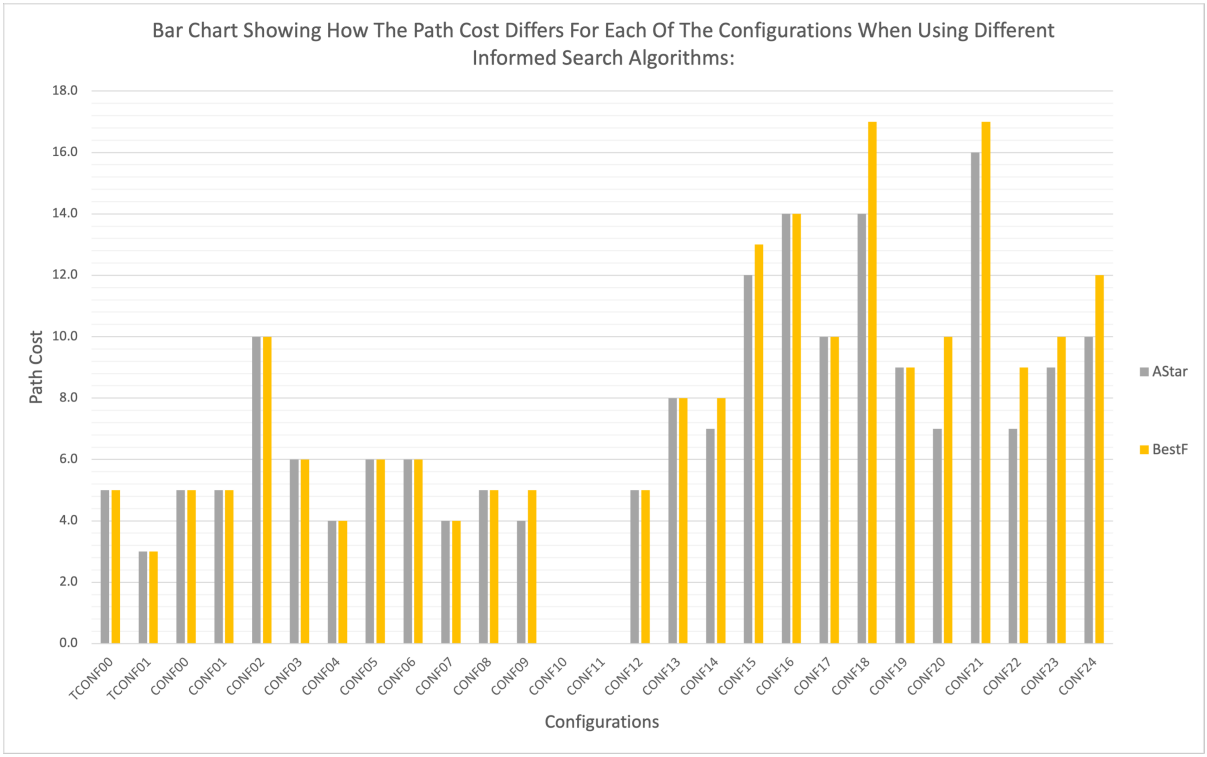


Figure 8: Bar chart showing how the number of nodes explored by the best-first and A* search algorithms compares across the provided configurations of the problem. On average, A* search requires that more nodes are explored in comparison to best-first search.

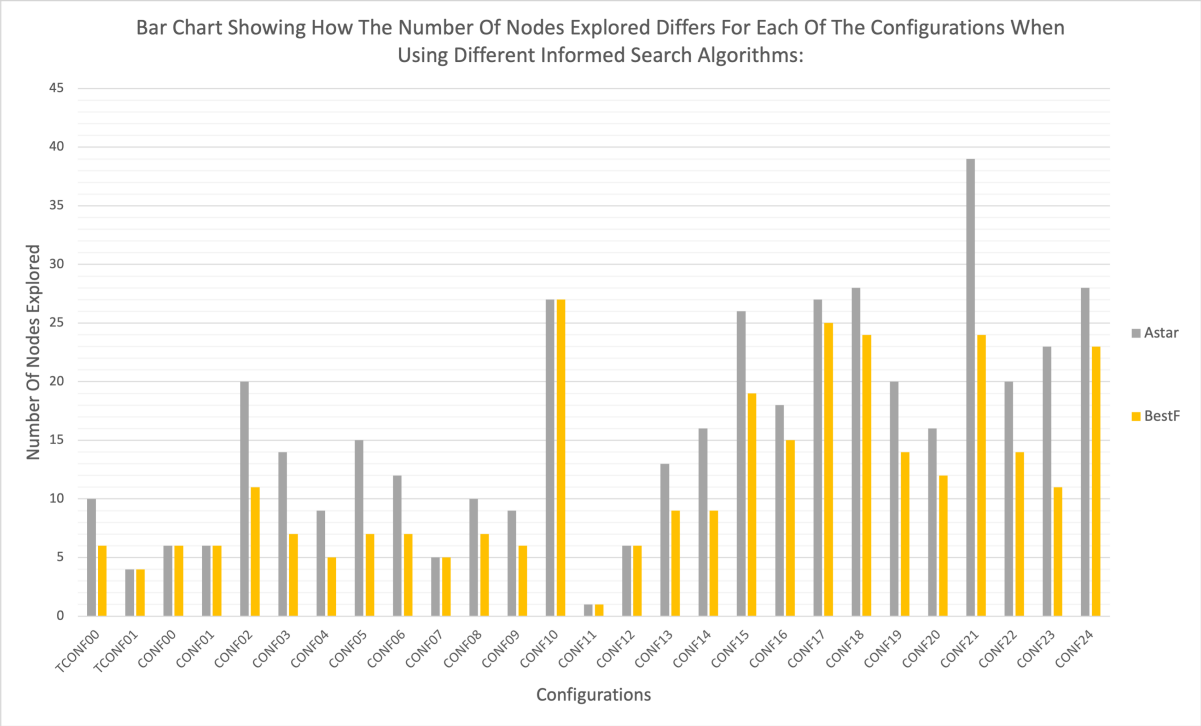


Figure 9: Bar chart comparing the path cost of solutions found by all of the search algorithms implemented. BFS and A* solutions are equivalent, which is expected as these algorithms provide optimal solutions. DFS and best-first search provide sub-optimal solutions for most configurations, with DFS being the worst of the algorithms for finding shortest paths. DFS provides the longest solutions on average.

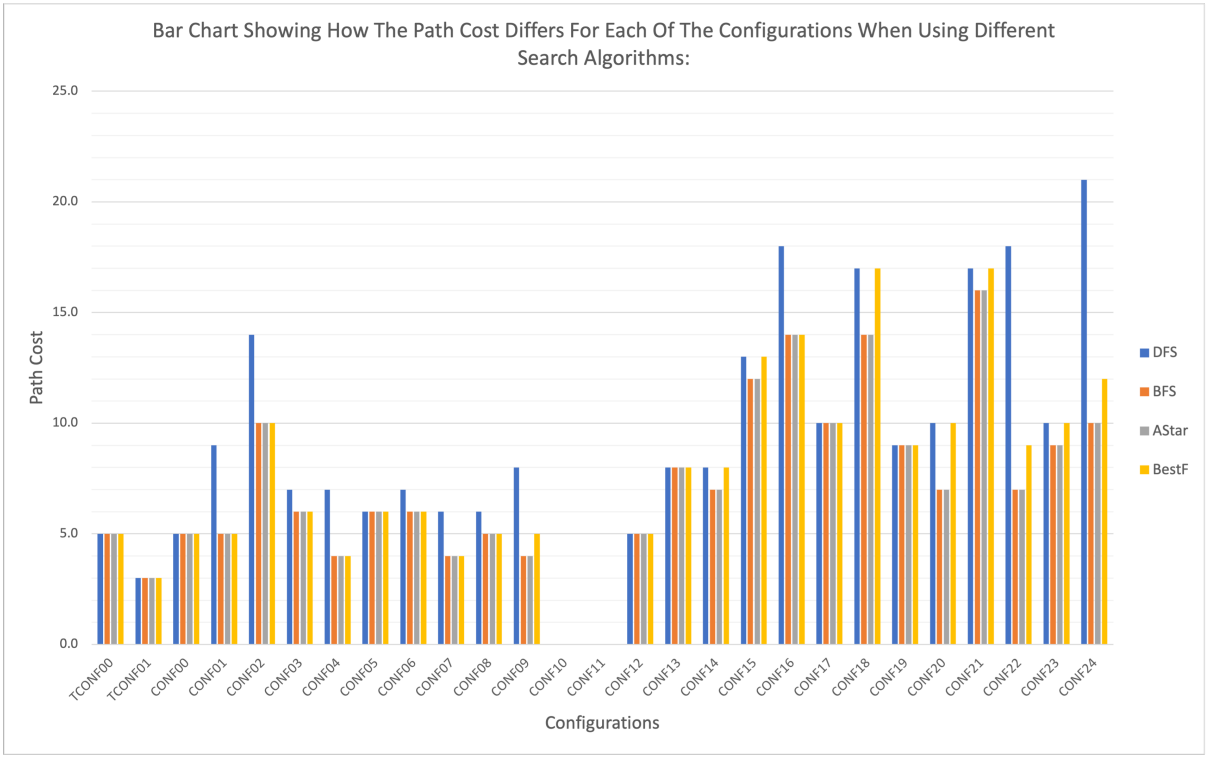


Figure 10: Bar chart showing how the number of nodes explored by all of the search algorithms implemented compares across the provided configurations of the problem. On average, the optimal algorithms (BFS and A* search) require more explored nodes than the sub-optimal search algorithms, though A* consistently requires less explored nodes than BFS. Best-first search requires fewest explored nodes of all the implemented search algorithms, on average.

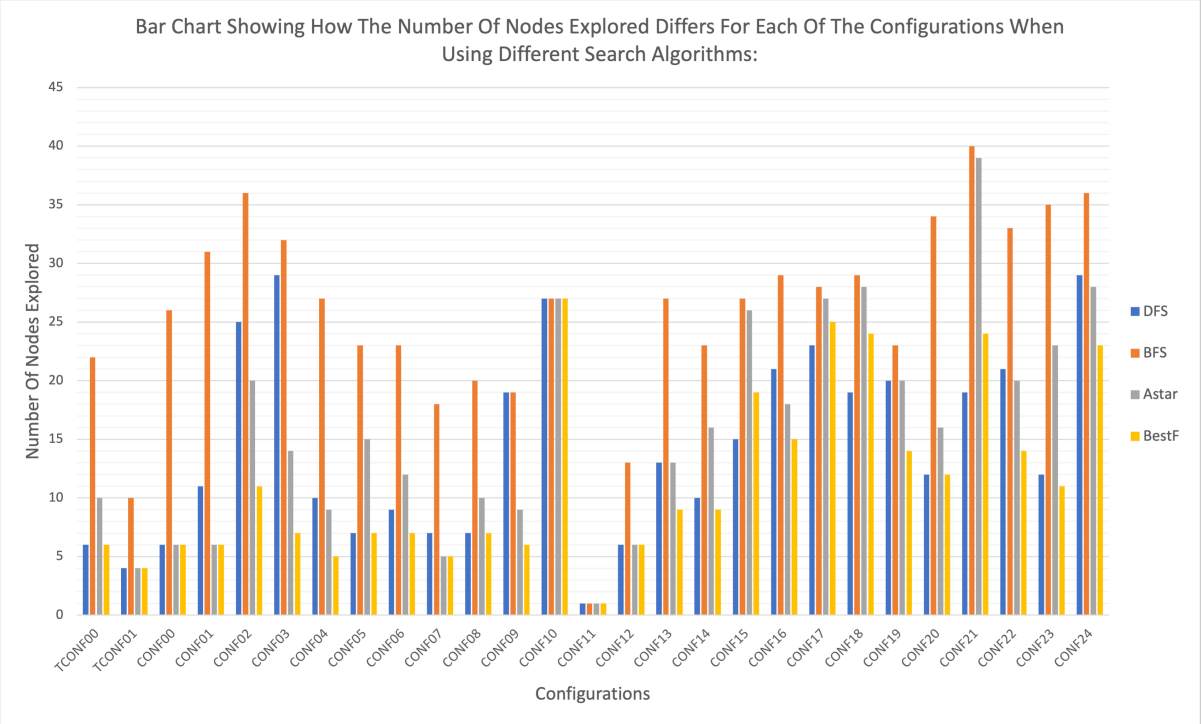


Figure 11: Bar chart showing how the number of nodes explored by A*, BFS, and BDS (which uses BFS in both search directions) compares across the provided configurations of the problem. On average, BDS explores fewer nodes than BFS. The improvement is significant for many configurations (*TCONF01* to *CONF10*). After *CONF10*, improvements to the number of explored nodes are less significant. In *CONF15* and *CONF19*, BDS actually required more nodes than BFS, so the improvement is not always the case. A* is still the most efficient optimal search implementation on average.

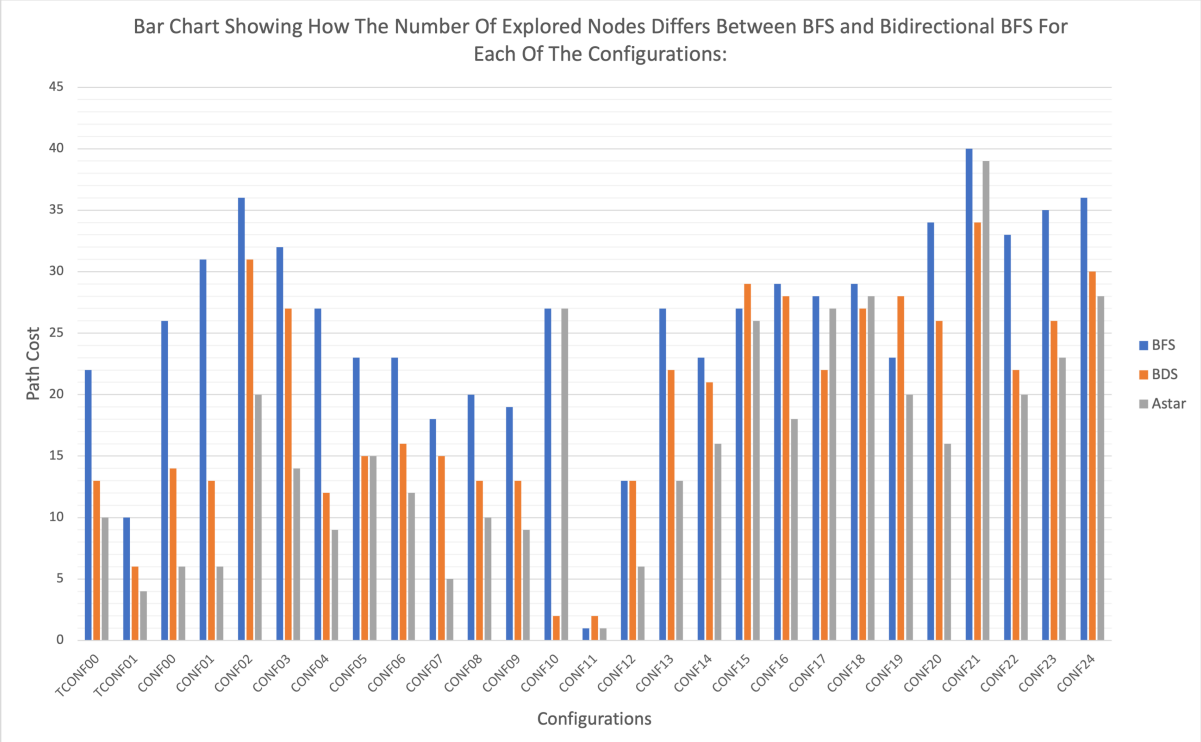
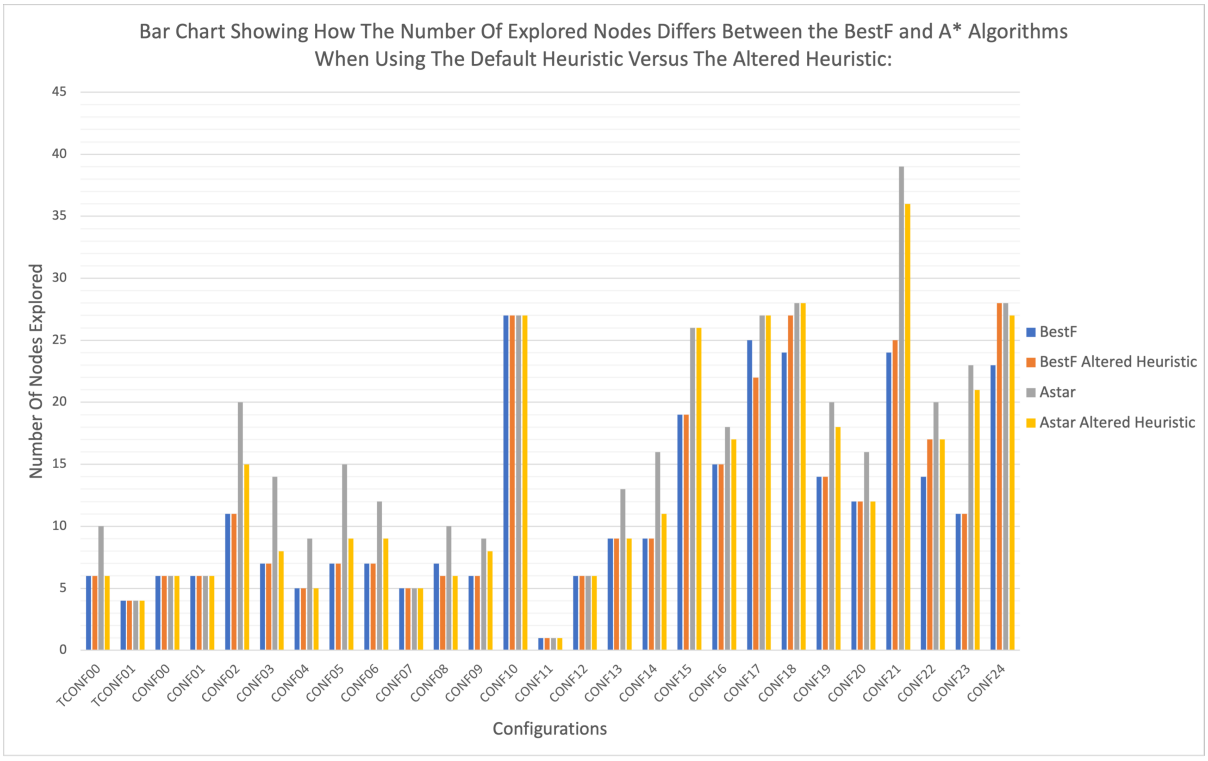


Figure 12: Table of results indicating the path cost and number of nodes explored as a result of executing the informed search algorithms on all of the provided configurations with the original heuristic and the altered heuristic.

	Path Cost				Number Of Explored Nodes			
Configuration	BestF	BestF Altered Heuristic	Astar	Astar Altered Heuristic	BestF	BestF Altered Heuristic	Astar	Astar Altered Heuristic
TCONF00	5.0	5.0	5.0	5.0	6	6	10	6
TCONF01	3.0	3.0	3.0	3.0	4	4	4	4
CONF00	5.0	5.0	5.0	5.0	6	6	6	6
CONF01	5.0	5.0	5.0	5.0	6	6	6	6
CONF02	10.0	10.0	10.0	10.0	11	11	20	15
CONF03	6.0	6.0	6.0	6.0	7	7	14	8
CONF04	4.0	4.0	4.0	4.0	5	5	9	5
CONF05	6.0	6.0	6.0	6.0	7	7	15	9
CONF06	6.0	6.0	6.0	6.0	7	7	12	9
CONF07	4.0	4.0	4.0	4.0	5	5	5	5
CONF08	5.0	5.0	5.0	5.0	7	6	10	6
CONF09	5.0	5.0	4.0	4.0	6	6	9	8
CONF10	fail	fail	fail	fail	27	27	27	27
CONF11	fail	fail	fail	fail	1	1	1	1
CONF12	5.0	5.0	5.0	5.0	6	6	6	6
CONF13	8.0	8.0	8.0	8.0	9	9	13	9
CONF14	8.0	8.0	7.0	7.0	9	9	16	11
CONF15	13.0	13.0	12.0	12.0	19	19	26	26
CONF16	14.0	14.0	14.0	14.0	15	15	18	17
CONF17	10.0	10.0	10.0	10.0	25	22	27	27
CONF18	17.0	17.0	14.0	14.0	24	27	28	28
CONF19	9.0	9.0	9.0	9.0	14	14	20	18
CONF20	10.0	10.0	7.0	7.0	12	12	16	12
CONF21	17.0	17.0	16.0	16.0	24	25	39	36
CONF22	9.0	9.0	7.0	7.0	14	17	20	17
CONF23	10.0	10.0	9.0	9.0	11	11	23	21
CONF24	12.0	12	10.0	10	23	28	28	27
	Average	Average	Average	Average	Average	Average	Average	Average
	8.2	8.2	7.6	7.6	11.5	11.8	15.9	13.7
Solutions	No Solutions							

Figure 13: Bar chart showing how the number of nodes explored by the informed search algorithms compares when using the default heuristic versus the altered heuristic. The path costs of all solutions remain unchanged when using either heuristic (solutions are identical). Using the altered heuristic slightly reduces efficiency on average for BestF, but increases efficiency on average for A*.



C Test Summary

This section outlines core testing performed on the implementations of the search agents to ensure correctness and conformity with the specification. The provided test cases are not an exhaustive list of all testing carried out.

Manual, method-by-method testing was performed to ensure the correctness of the fundamental components of the system. These tests included normal, edge, and extreme cases to guarantee the robustness of the system for the input space. The provided test cases in the following sections are specifically selected to demonstrate the correctness of the system at a high-level without having to individually provide all test cases for all methods; the correctness of high-level functionality is dependant on the correctness of low-level functionality. To this end, screenshots of program behaviour and worked examples are given as evidence.

C.1 Basic Agent Testing

Table 2: Table outlining the high-level, core correctness testing carried out for the Basic Agent.

Test Description	**Passed?**	**Proof**
Search algorithm argument must be provided.	Yes	Figure 14
Search algorithm argument must be valid.	Yes	Figure 15
Configuration argument must be provided.	Yes	Figure 16
Configuration argument must be valid.	Yes	Figure 17
Program output is correct format for successes.	Yes	Figure 18
Program output is correct format for failures.	Yes	Figure 19
Queue behaviour of frontier is correct.	Yes	Figure 20
Stack behaviour of frontier is correct.	Yes	Figure 21
Correctness of BFS implementation via working example.	Yes	Figure 22
Correctness of DFS implementation via working example.	Yes	Figure 23
Program outputs failure when a barrier is specified as the start/goal.	Yes	Figure 30
Program outputs failure when specified start/goal is out-of-bounds.	Yes	Figure 31

Figure 14: Test showing that a usage message is displayed when a search algorithm is not specified.

```
nmpoole@Nathans-MacBook-Pro A1src % java A1main TCONF00
Usage: 'java A1main <DFS|BFS|AStar|BestF|...> <ConfID> [<any other param>]'
```

Figure 15: Test showing that providing an unimplemented search algorithm results in a failure and no nodes are explored.

```
nmpoole@Nathans-MacBook-Pro A1src % java A1main NotAnAlg TCONF00
fail
0
```

Figure 16: Test showing that a usage message is displayed when a problem configuration is not specified.

```
nmpoole@Nathans-MacBook-Pro A1src % java A1main DFS
Usage: 'java A1main <DFS|BFS|AStar|BestF|...> <ConfID> [<any other param>]'
```

Figure 17: Test showing that an error/usage message is displayed when an invalid problem configuration is specified.

```
nmpoole@Nathans-MacBook-Pro A1src % java A1main BFS NotAConf
Not A Configuration.
Usage: 'java A1main <DFS|BFS|AStar|BestF|...> <ConfID> [<any other param>]'
```

Figure 18: Test showing the output of the program is correct when solutions are found, as described by the specification. As seen, the frontier at each step is presented as a comma (and space) separated list of coordinates within brackets. Also, the final three lines are the solution path as a list of coordinates without a delimiter or spaces, the path cost as a double, and the number of nodes explored as an integer.

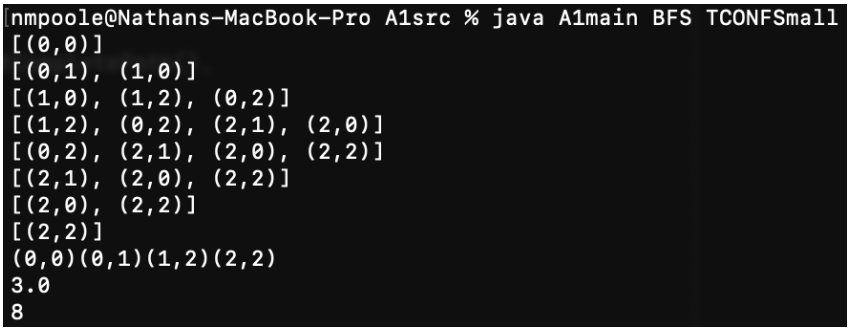


Figure 19: Test showing the output is correct when no solution is found, as described by the specification. As seen, the frontier at each step is presented as a comma (and space) separated list of coordinates within brackets. Then, the next two lines are: 'fail', and the number of nodes explored as an integer.

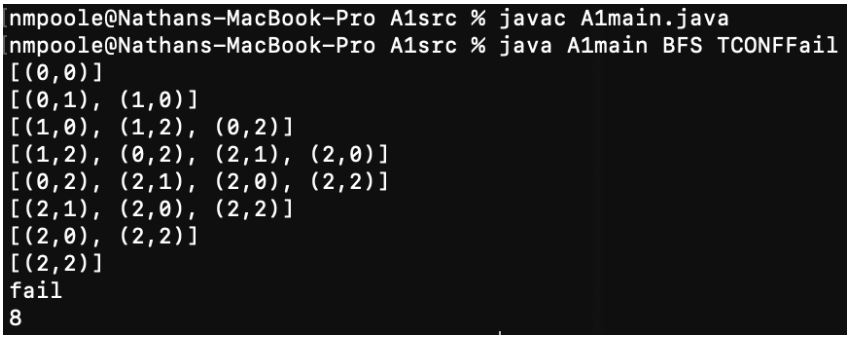


Figure 20: Test showing that the first-in, first-out behaviour of the frontier for breadth-first search is correct. As expected, nodes are added to the frontier at the end, whilst nodes are removed from the front.

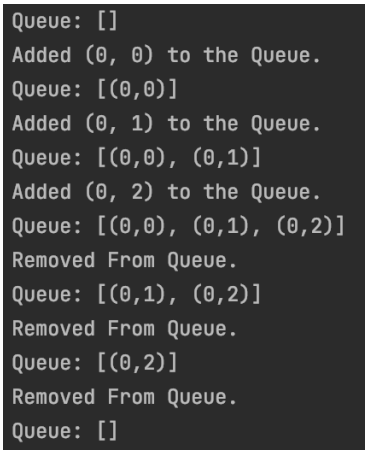
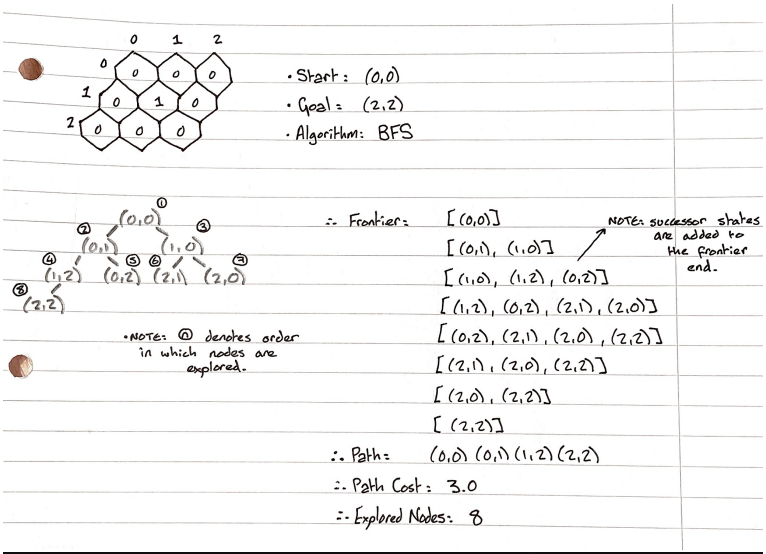


Figure 21: Test showing that the last-in, first-out behaviour of the frontier for DFS is correct. As expected, frontier nodes are added to and removed from the front.

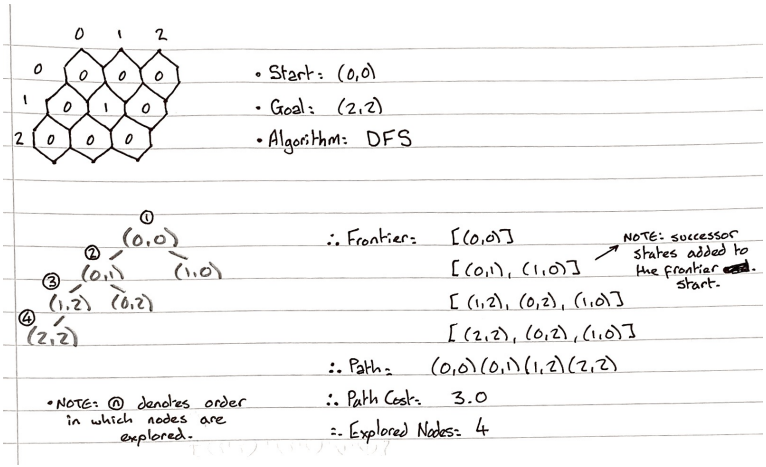
```
Stack: []
Added (0, 0) to the Stack.
Stack: [(0,0)]
Added (0, 1) to the Stack.
Stack: [(0,1), (0,0)]
Added (0, 2) to the Stack.
Stack: [(0,2), (0,1), (0,0)]
Removed From Stack.
Stack: [(0,1), (0,0)]
Removed From Stack.
Stack: [(0,0)]
Removed From Stack.
Stack: []
```

Figure 22: The top image represents a manually calculated execution of the BFS algorithm for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the BFS implementation. Note that this also shows that nodes are correctly explored according to the tie-breaking scheme.




```
nmpoole@Nathans-MacBook-Pro A1src % java A1main BFS TCONFSmall
[(0,0)]
[(0,1), (1,0)]
[(1,0), (1,2), (0,2)]
[(1,2), (0,2), (2,1), (2,0)]
[(0,2), (2,1), (2,0), (2,2)]
[(2,1), (2,0), (2,2)]
[(2,0), (2,2)]
[(2,2)]
(0,0)(0,1)(1,2)(2,2)
3.0
8
```

Figure 23: The top image represents a manually calculated execution of the DFS algorithm for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the DFS implementation. Note that this also shows that nodes are correctly explored according to the tie-breaking scheme.




```
nmpoole@Nathans-MacBook-Pro A1src % java A1main DFS TCONFSmall
[(0,0)]
[(0,1), (1,0)]
[(1,2), (0,2), (1,0)]
[(2,2), (0,2), (1,0)]
(0,0)(0,1)(1,2)(2,2)
3.0
4
```

C.2 Intermediate Agent Testing

Table 3: Table outlining the high-level, core correctness testing carried out for the Intermediate Agent.

Test Description	**Passed?**	**Proof**
Priority queue behaviour of frontier is correct.	Yes	Figure 24
Priority queue update behaviour of frontier is correct.	Yes	Figure 25
Correctness of best-first search implementation via working example.	Yes	Figure 26
Correctness of best-first search heuristic calculation.	Yes	Figure 26
Correctness of best-first search output.	Yes	Figure 26
Correctness of A* search implementation via working example.	Yes	Figure 27
Correctness of A* search `fCost` calculation.	Yes	Figure 27
Correctness of A* search output.	Yes	Figure 27

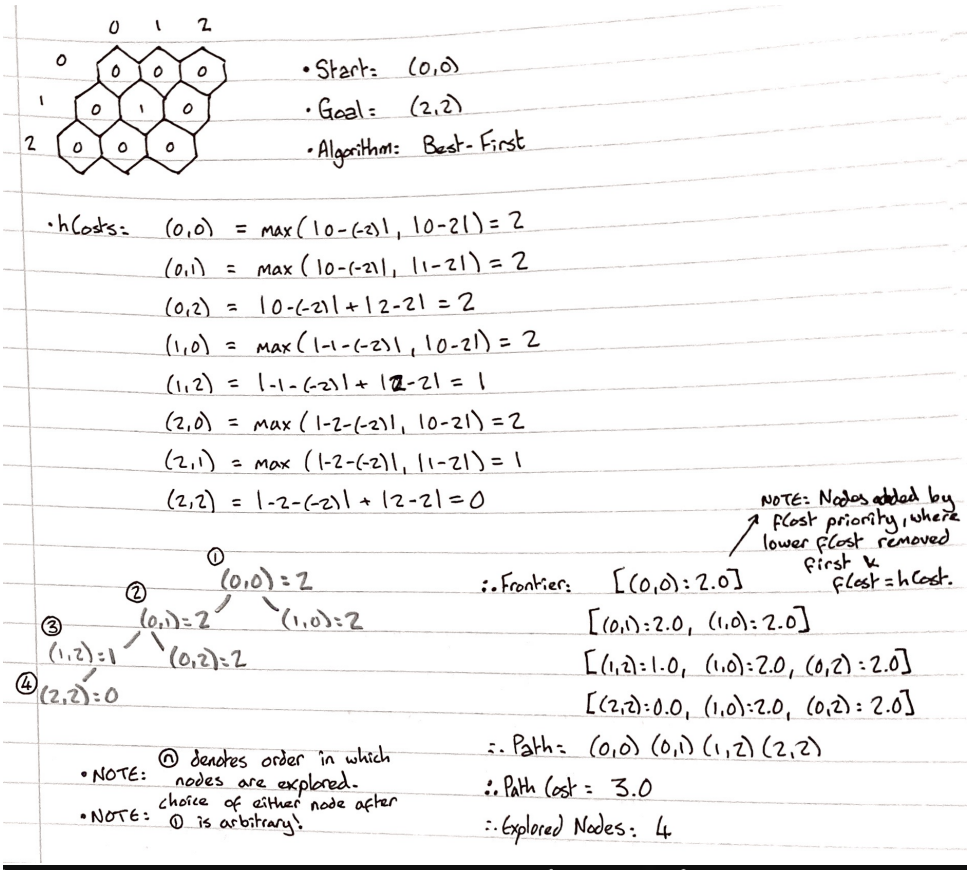
Figure 24: Test showing nodes are added and removed from the priority queue data structure (used for best-first and A* search) correctly. Nodes in the priority queue are successfully added and are removed correctly according to lowest `fCost` first.

```
PriorityQueue: []
Added (0,0) to the Priority Queue with fCost = 1.
PriorityQueue: [(0,0):1.0]
Added (0,1) to the Priority Queue with fCost = 2.
PriorityQueue: [(0,0):1.0, (0,1):2.0]
Added (0,2) to the Priority Queue with fCost = 3.
PriorityQueue: [(0,0):1.0, (0,1):2.0, (0,2):3.0]
Removed (0,0) from Priority Queue.
PriorityQueue: [(0,1):2.0, (0,2):3.0]
Removed (0,1) from Priority Queue.
PriorityQueue: [(0,2):3.0]
Removed (0,2) from Priority Queue.
```

Figure 25: Test showing nodes within the priority queue frontier can be successfully replaced, which is required for A* search when lower path costs are found for nodes already in the frontier.

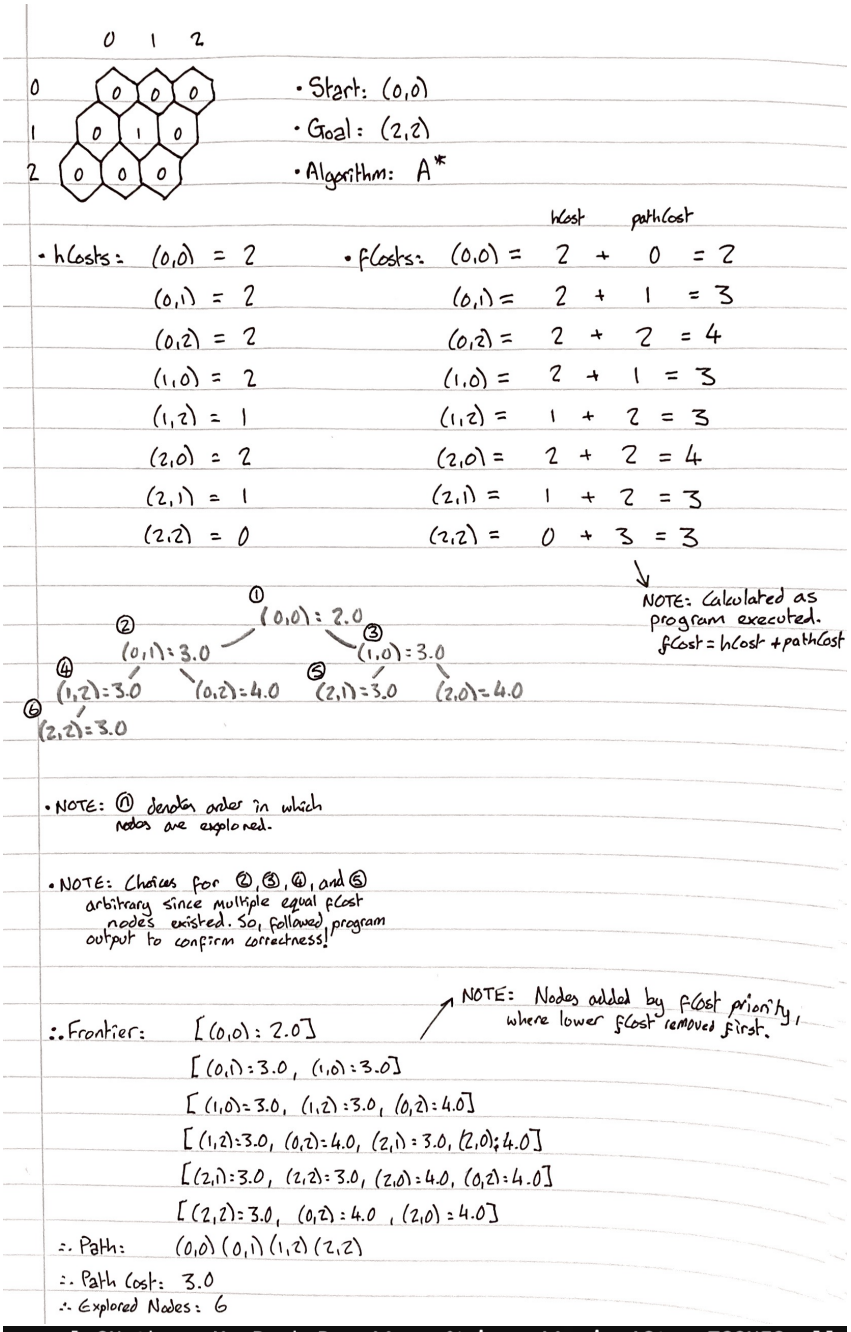
```
PriorityQueue: [(0,0):1.0, (0,1):2.0, (0,2):3.0]
Updated fCost of (0,0) to 0.0
PriorityQueue: [(0,0):0.0, (0,2):3.0, (0,1):2.0]
```

Figure 26: The top image represents a manually calculated execution of the best-first search algorithm for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the best-first search implementation. The correctness of `fCost` calculation is demonstrated, as all `fCost` values in the program frontier equal those from the working example. Also, the bottom image demonstrates the output of the program for best-first search is correct, as described by the specification. The frontier is printed as with the basic agent, but all nodes in the frontier have their `fCost` presented, separated by a colon. The path, path cost, and number of explored nodes is calculated and output correctly.




```
nmpoole@Nathans-MacBook-Pro A1src % java A1main BestF TCONFSmall
[(0,0):2.0]
[(0,1):2.0, (1,0):2.0]
[(1,2):1.0, (1,0):2.0, (0,2):2.0]
[(2,2):0.0, (1,0):2.0, (0,2):2.0]
(0,0)(0,1)(1,2)(2,2)
3.0
4
```

Figure 27: The top image represents a manually calculated execution of the A* search algorithm for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the A* search implementation. The correctness of `fCost` calculation is demonstrated, as all `fCost` values in the program frontier equal those from the working example. Also, the bottom image demonstrates the output of the program for A* search is correct, as described by the specification. The frontier is printed as with the basic agent, but all nodes in the frontier have their `fCost` presented, separated by a colon. The path, path cost, and number of explored nodes is calculated and output correctly.




```
nmpoole@Nathans-MacBook-Pro A1src % java A1main AStar TCONFSmall
[(0,0):2.0]
[(0,1):3.0, (1,0):3.0]
[(1,0):3.0, (1,2):3.0, (0,2):4.0]
[(1,2):3.0, (0,2):4.0, (2,1):3.0, (2,0):4.0]
[(2,1):3.0, (2,2):3.0, (2,0):4.0, (0,2):4.0]
[(2,2):3.0, (0,2):4.0, (2,0):4.0]
(0,0)(0,1)(1,2)(2,2)
3.0
6
```

C.3 Advanced Agent (Bidirectional Search) Testing

Table 4: Table outlining the high-level, core correctness testing carried out for the Advanced Agent (Bidirectional Search).

Test Description	Passed?	Proof
Correctness of bidirectional search algorithm via working example.	Yes	Figure 28
BDS produces failure when no solution.	Yes	Figure 29
BDS produces failure when goal is a barrier.	Yes	Figure 30
BDS produces failure when goal is out-of-bounds.	Yes	Figure 31

Figure 28: The top image represents a manually calculated execution of the BDS search algorithm for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the BDS search implementation.

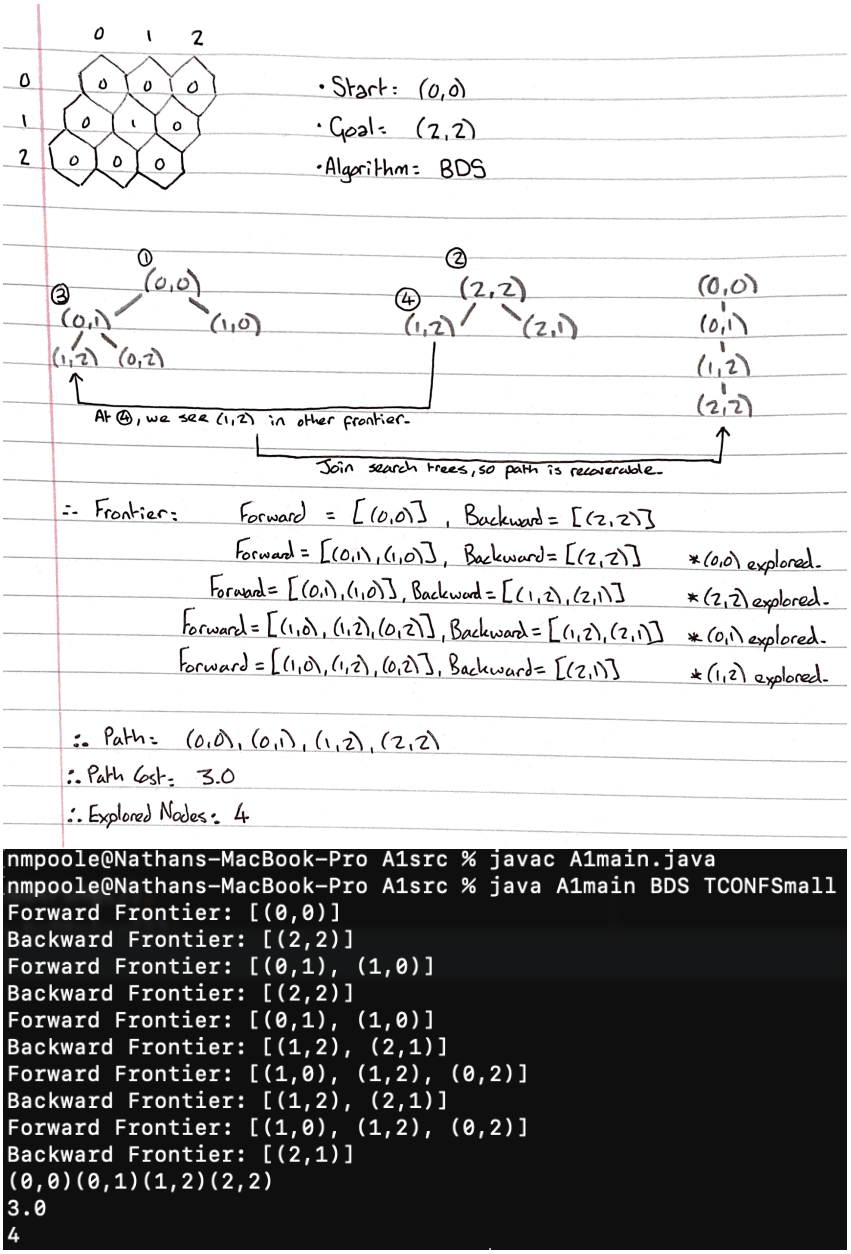


Figure 29: Test showing that the program correctly produces a failure with bidirectional search when a solution does not exist. Here, *CONF10* is given as input and specifies that coordinate (2, 2) is the goal. This cannot lead to a solution as (2, 2) is surrounded by barriers, as found by BDS when the frontier for searching backwards is emptied immediately (no states are accessible from the goal).

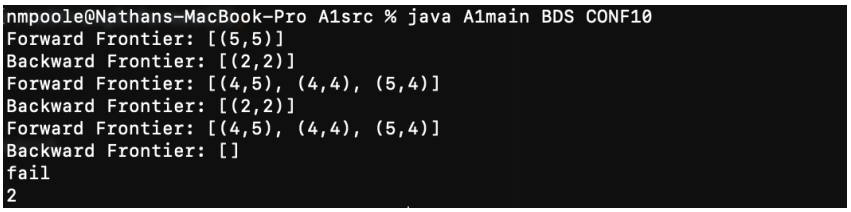


Figure 30: A bug was found showing that BDS would find solutions for configurations where a barrier was selected as the goal. This was due to the implementation assuming that the current state must be valid (because moving to invalid coordinates is disallowed). However, this is not the case when the goal (or start) is specified as a barrier. This issue was rectified, which also fixed the fact that all algorithms would find solutions if the start state was a barrier. Here, (1, 1) was set as a goal, which is also marked as a barrier, and the program shows a failure (so bug is fixed).

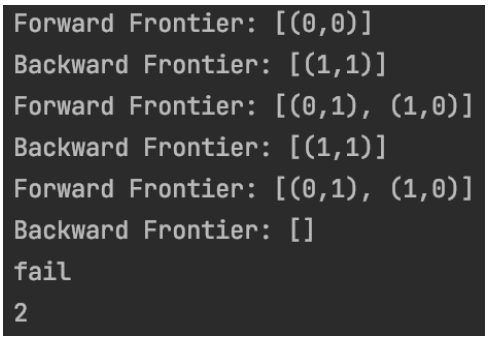
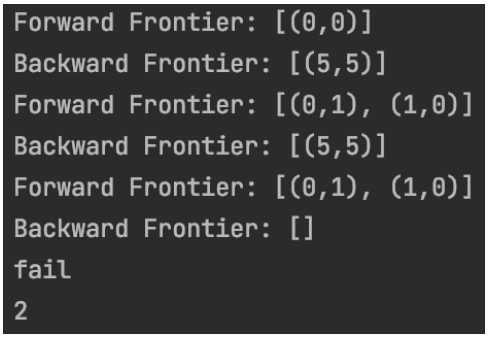


Figure 31: A bug was found for BDS which would cause exceptions when the goal is specified out-of-bounds. This was also experienced by all algorithms when the start was specified out-of-bounds. These issues have been rectified. Here, (5, 5) is the goal, but the grid is a 3x3. The program outputs a failure instead of causing an exception (so bug fixed).



C.4 Advanced Agent (Alternate/Extension Heuristic) Testing

Table 5: Table outlining the high-level, core correctness testing carried out for the Advanced Agent (Extended/Altered Heuristic).

Test Description	**Passed?**	**Proof**
Test directional cost is 0 when appropriate.	Yes	Figure 32
Test directional cost is 0.5 when appropriate.	Yes	Figure 33
Test directional cost is 1 when appropriate.	Yes	Figure 34
Correctness of BestF with altered heuristic via working example.	Yes	Figure 35
Correctness of A* with altered heuristic via working example.	Yes	Figure 36
Test `fCost` output to 2 decimal places when using altered heuristic.	Yes	Figure 37

Figure 32: Moving to a state (0, 1) from state (0, 2) is directly towards the goal state (0, 0). Thus, the calculated direction cost should be 0.0 by the direction cost definition, which is seen to be the case in the program output below.

```
P = (0,2), C = (0,1), G = (0,0)
dirCost = 0.0
```

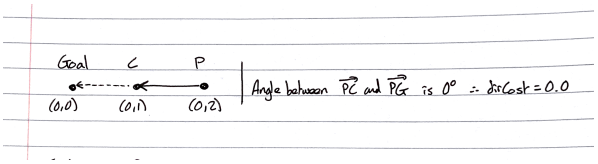


Figure 33: Moving to a state (1, 1) from state (0, 1) is right-angled to the direction of the goal state (0, 0). Thus, the calculated direction cost should be 0.5 by the direction cost definition, which is seen to be the case in the program output below.

```
P = (0,1), C = (1,1), G = (0,0)
dirCost = 0.5
```

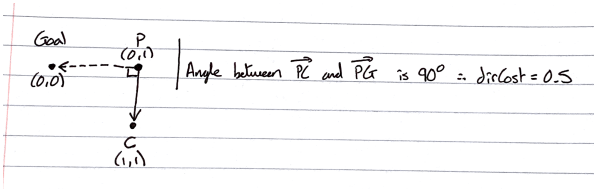


Figure 34: Moving to a state (0, 2) from state (0, 1) is in the opposite direction to the goal state (0, 0). Thus, the calculated direction cost should be 1.0 by the direction cost definition, which is seen to be the case in the program output below.

```
P = (0,1), C = (0,2), G = (0,0)
dirCost = 1.0
```

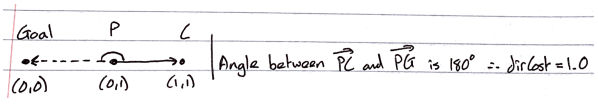
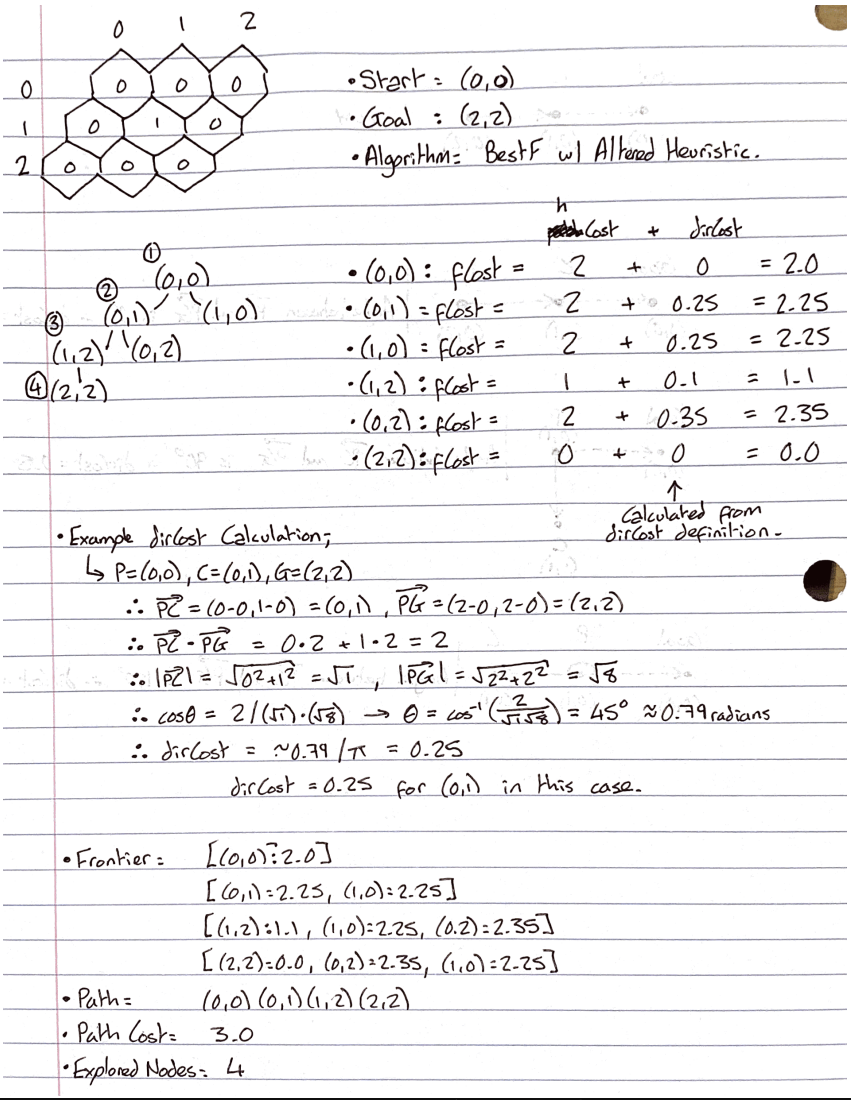
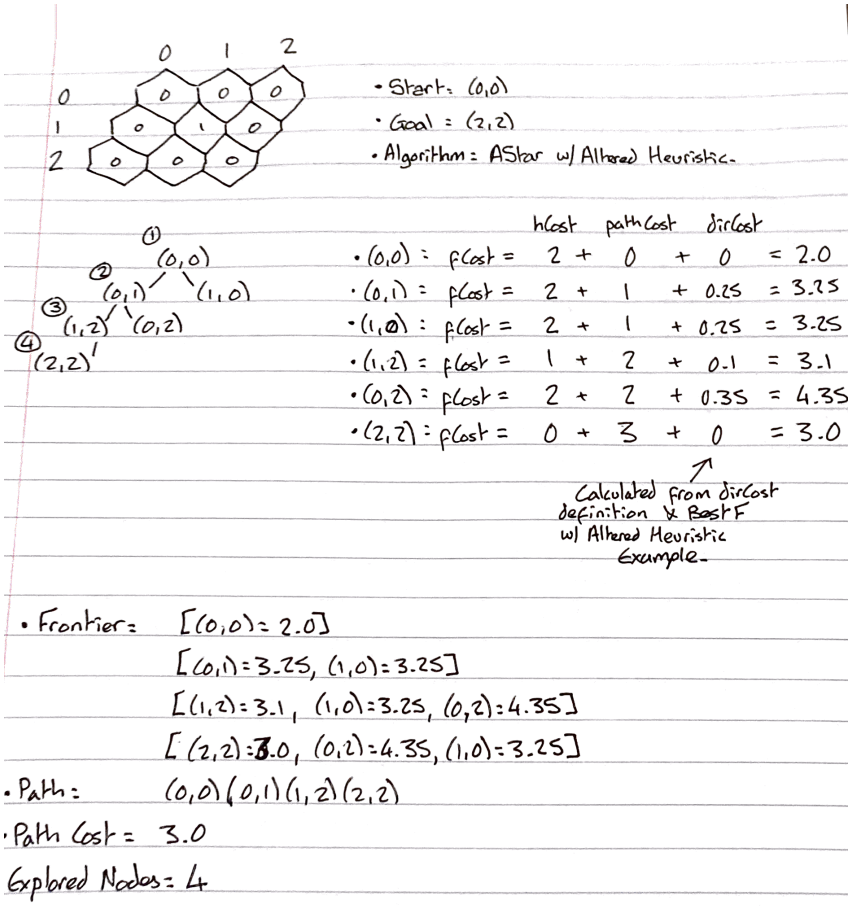


Figure 35: The top image represents a manually calculated execution of BestF with the altered heuristic for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the BestF implementation when using the altered heuristic.




```
nmpoole@Nathans-MacBook-Pro A1src % java A1main BestF TCONFSmall -aH
[(0,0):2.0]
[(0,1):2.25, (1,0):2.25]
[(1,2):1.1, (1,0):2.25, (0,2):2.35]
[(2,2):0.0, (0,2):2.35, (1,0):2.25]
(0,0)(0,1)(1,2)(2,2)
3.0
4
```

Figure 36: The top image represents a manually calculated execution of A* with the altered heuristic for a small configuration, *TCONFSmall*. The bottom image represents the output of the program when solving the same problem configuration. The frontier, path, path cost, and number of explored nodes are identical in the working example and program output. This supports the correctness of the A* implementation when using the altered heuristic.




```
nmpoole@Nathans-MacBook-Pro A1src % java A1main AStar TCONFSmall -aH
[(0,0):2.0]
[(0,1):3.25, (1,0):3.25]
[(1,2):3.1, (1,0):3.25, (0,2):4.35]
[(2,2):3.0, (0,2):4.35, (1,0):3.25]
(0,0)(0,1)(1,2)(2,2)
3.0
4
```

Figure 37: The following screenshot shows that the output of costs in the frontiers when using the altered heuristic is given to 2 decimal places, which is as expected.

```
[(0,0):2.0]
[(0,1):2.25, (1,0):2.25]
[(1,2):1.1, (1,0):2.25, (0,2):2.35]
[(2,2):0.0, (0,2):2.35, (1,0):2.25]
(0,0)(0,1)(1,2)(2,2)
3.0
4
```

D Bibliography

The required content for this assessment was acquired from the specification and module material.