University of
St Andrews

School of Computer Science

CS5011
Artificial Intelligence Practice
Practical 1

Parts attempted:

Implemented Breadth-first and Depth-first search

Implemented Best-first and A* search algorithms using both the Euclidean and Chebyshev distance measures

Implemented Bi-directional A* search using the Euclidean distance measure

ID: 170018405

Word Count: 3785 Time spent: ≈55hrs

Table of Contents

Table of Figures

1.0 Introduction

Search algorithms were developed to enable computers to find solutions with specified properties in a large search space. Early applications of search algorithms were focussed on allowing computers to compute a set of moves in a game - like chess. The program Deep-blue in fact utilised a search algorithm to beat a chess world champion in the mid-90s (Hsu, Campbell, & Hoane, 1995). Since then, search algorithms have been put to use in tasks like indexing web pages (Ueno, Suzumura, Maruyama, Fujisawa, & Matsuoka, 2017) (Google, 2018), finding connections between friends on social media (Spirin, 2014), garbage collection by programming languages, calculating paths in navigation, and even theorem proving (Kowalski, 1970). The purpose of this assignment is to use search algorithms to compute paths between two locations on a map allowing a robot to lead survivors outside unsafe building during earthquakes.

2.0 Problem decomposition

For an agent to be able to compute its location, and the possible moves it can take to reach a goal state, several functionalities are required. The agent must be able to:

1. Represent its current and goal states.
2. Find a way to compute a series of legal changes of states to reach the goal.
3. Enact the computed path computed.

2.1 State representation

Par 1

For this search problem adjacency Matrices were provided representing the environment that the agent needs to navigate. The Matrices represent this environment as set of all the possible locations which an agent can navigate between. A value of 10 indicates that there is no connection between the points. For-example, in the map represented in table 1, there are no connections from a-c nor from c-a. A value of 5 on the other hand indicates the contrary. For undirected graphs, it is therefore expected that the Matrix is mirrored along the diagonal which is the case for all maps tested in this assignment. The diagonal (in green) therefore indicates each location that the agent could move to on the map (given a connection exists). Values of 0 on the diagonal indicate that there are no connections (loops) between each point and itself, while, values of 2 or 8 indicate the starting and goal states respectively.

Using this information, the state of the agent at any point in time can therefore be represented as a single integer value i (from 0 to $n - 1$) in a $n \times n$ Matrix. The initial and goal states are also represented by an integer value i where $M(i,i) = 2$, and $M(i,i) = 8$.

Label	a	b	c	d	e	f	g
a	2	5	10	10	5	10	10
b	5	0	5	10	10	10	10
c	10	5	0	5	10	10	10
d	10	10	5	8	10	10	5
e	5	10	10	10	0	5	10
f	10	10	10	10	5	0	5
g	10	10	10	5	10	5	0

Table 1 - Adjacency Matrix for map 6

Part 2, 3

While a single index value is sufficient to keep track of the agent state in the case where all path lengths are considered equal, this representation of state is not enough when using search algorithms that make use of path length and/or heuristics to traverse a graph in order to find the shortest path.

In the case where search algorithms like A-star are used, the state of the agent needs to also include the path length so far, and the future predicted path length.

2.2 Successor and action functions

To compute a series of legal changes of states, a function is required to list these moves. In the case where the connections from any location are listed in a symmetric adjacency Matrix, the connections to any point can be found by iterating over the row i, where i is the index recorded as the state of the agent. A for-loop iterating from $[i, 0] - [i, n]$, is useful in this case as it provides a natural tie-breaking strategy, which, ensures that states with a lower index are explored first even though the two states may be tied cost-wise.

One more functionality that the successor function needs to perform is to determine which states have already been visited to avoid loops and repetitions of explored states. A possible way to implement this is simply to keep track of all the states explored using a Boolean array of size n. When a state becomes visited, then the visited array should be updated to reflect this. The successor function should also ensure that duplicate states are not added to the queue.

Following the successor function, an action function is typically required to enact the results of the search process. In the case of this exercise, such a function was not required as once a path is computed, there are no physical actions to be taken. If there was any function within this practical that can be considered an action function, it would be that of printing the path to the command prompt.

3.0 Implementation

3.1 State classes

Two separate State classes were developed to store the state of the agent. For part 1 a State class was essentially unnecessary since only 1 variable is being stored within the class – the index of the intersection point. However, in the spirit of building an adaptable object oriented model which can be built-upon, a decision to build a basic State class (named State) was made. Within the State class some basic methods to compare, check for equality, convert to String, and convert the index to its corresponding label were implemented. As for part 2, this basic State class was extended to develop a CostState class. In the CostState class the two additional variables (and the necessary accompanying methods) required to represent the state of the agent (namely the path and heuristic costs) when utilising informed search algorithms were implemented.

3.2 Node classes

Search algorithms typically utilise node-based tree data structures to search for solutions. Starting from an initial node, a successor function is utilised to find the next possible states, through which nodes are created and traversed until a goal node is reached. Once a goal is reached, each node should be able to link back to its direct parent all the way to the initial node providing a path of states from start to finish. Nodes hence are necessary to provide a way of storing the agent state as well as link to the agent's previous states. A Node class was thus developed to enable this functionality.

In the Node class it was decided that in addition to the aforementioned fields, fields to store the node-depth and list of possible connections (edges) were desirable. The depth field in particular is useful as it allows the user to printout the node object and see how far down the search tree the node object is. As for the edges field, this is technically unnecessary as it adds additional complexity to the solution. To explain why this is, a closer look must be payed to the constructor of the Node class. When a node object is created, the constructor uses an internal method named findEdges to loop over the respective row within the adjacency Matrix. When a value of 5 is found, the index of this value is recorded in the edges field within the node object. The problem with this is, the search algorithm then loops over the edges of each node checking if this edge has been explored or not. What results is a complexity of $O(n + e)$ instead of $O(n)$ – where n is the length of the matrix and e is the number of edges between a node and its neighbours. A better solution would be to just remove this extra feature from the node object, and just loop once looking for possible edges which have not been visited within the search algorithm itself. However, for the purposes of printing which connections a node has, given that this

practical is focussed on testing that the correct implementation of the search algorithms was achieved, this extra complexity was left in. Figure 1 shows the output received when a node is printed to the console. Notice how each node prints its state, depth, and edges within curly braces followed by any parent nodes.

A second node class which extends Node (named CostNode) was also created for part 2 of the assignment. This class implements comparable and uses the CostState compareTo method to allow a PriorityQueue to order the nodes in the frontier based on path and/or heuristic costs.

```
Node: {
    State: [Label: n]    Depth: 4    Edges: [10, 12]
} Parent-Node: {
    State: [Label: m]    Depth: 3    Edges: [9, 13]
} Parent-Node: {
    State: [Label: j]    Depth: 2    Edges: [2, 7, 8, 11, 12]
} Parent-Node: {
    State: [Label: c]    Depth: 1    Edges: [0, 1, 3, 4, 9, 11]
} Parent-Node: {
    State: [Label: d]    Depth: 0    Edges: [2]
}
```

Figure 1 - output of Node toString method

3.3 The Breadth-first (BFS) and Depth-first (DFS) search algorithms

Given the necessary foundations were complete, the DFS and BFS algorithms were implemented using the general-search algorithm. These algorithms can be found in the UninformedSearch class.

Breadth-first search (BFS)

BFS is an algorithm that searches by expanding and checking each node at a particular depth before moving to the nodes at the next depth and expanding those. The algorithm can be visualised as working out radially from the initial state until it reaches the goal. Consider the graph shown in Figure 2. A BFS approach would check if a is the goal before expanding and checking if b, c or, d are the goals and expanding those to check e, and finally f. To perform this search operation systematically, the algorithm makes use of two lists. The first is often referred to as the frontier, and the second as the explored list. The purpose of the former is to contain the list of nodes yet to explore, while, the second's purpose is to simply keep track of which nodes have already been explored. A step by step view of the state of the frontier and visited list (until the goal is reached) is shown below. Note that the order by which nodes are added, is the same as how they are removed from the frontier.

$$Frontier: 1.\{\cancel{a}\}\ \ 2.\{\cancel{b}, c, d\}\ \ 3.\{\cancel{c}, d, e\}\ \ 4.\{\cancel{d}, e, f\}\ \ 5.\{\cancel{e}, f, g\}\ \ 6.\{\cancel{f}, g\}\ \ 7.\{g\}$$

$$Visited: 1.\{a\}\ \ 2.\{a, b\}\ \ 3.\{a, b, c\}\ \ 4.\{a, b, c, d\}\ \ 5.\{a, b, c, d, e\}\ \ 6.\{a, b, c, d, e, f\}$$

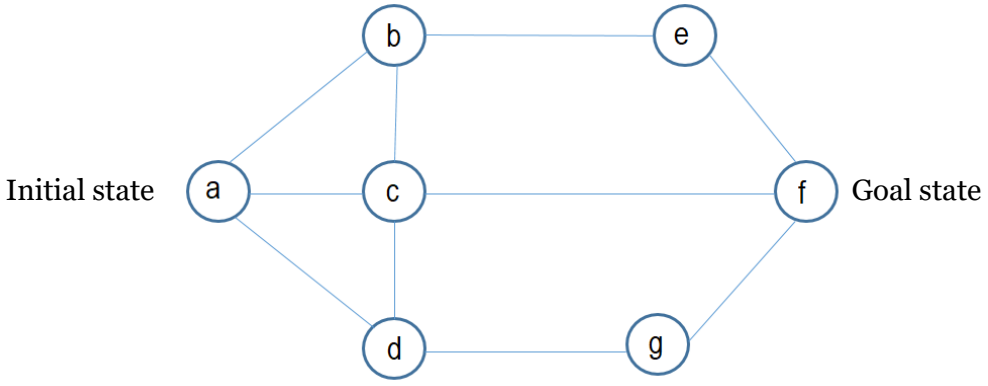

Figure 2 - example node graph

Due to the fact that this algorithm checks all possible paths, it is said to be optimal (as long as the path cost is a none-decreasing function of the depth of the node). The algorithm is also said to be complete as long as there aren't infinite states that can be expanded from a single node (Russell & Norvig, 2016). This means that the algorithm will always return a solution that is of the shortest path. The pseudo-code for the implementation of this algorithm used in this report is given below.

1. Initial and goal states initialised
2. A linked-list (a type of queue used to store the frontier) is instantiated and the initial node is added to this list.
3. A Boolean array of size n (for an adjacency Matrix of size $n \times n$) is instantiated. This Boolean array is required to keep track of both the nodes explored, as well as those in the frontier.
4. The element at the index of the initial state in the Boolean array is marked as true. Indicating that this node (the initial node) is either explored or in the frontier.
5. Begin while loop which loops until the frontier becomes empty.
6. The first node in the frontier is removed and referenced with a variable called currentNode.
7. If the state of currentNode is equivalent to the goal state, then the path taken by this node is returned, else, begin for-loop to loop over each of the node's edges (connections).
8. If the current edge (begin looped over) is already explored or in the frontier, then skip to the next iteration.
9. Else, create a state + node from the vertex connected to this edge and add it to the frontier.
10. Mark the edge as added to frontier in the Boolean array.
11. Once the frontier becomes empty return null (no result).

Note that in the implementation, the Boolean array described in step 3 is used to keep track of both the nodes visited and the nodes in the frontier. This makes it possible to reduce the complexity of needing to check if a node exists in the linked-list (which could involve the traversal of the whole linked-list for every edge looped over at step 7). Using a Boolean array makes it possible to reduce this complexity to a single operation.

Depth-first search (DFS)

DFS searches by iterative deepening. Instead of expanding all the children of a particular node, and all of the children of those expanded, DFS adds the nodes to the frontier so that the last node added is the first node explored. In the case of the node graph in Figure 2, a DFS algorithm will return the following frontier and visited lists:

$$Frontier: 1.\{a\}\ \ 2.\{b,a\}\ \ 3.\{c,b,a\}\ \ 4.\{d,c,b,a\}\ \ 5.\{g,d,c,b,a\}\ \ 6.\{f,g,d,c,b,a\}$$

$$Visited: 1.\{a\}\ \ 2.\{a,b\}\ \ 3.\{a,b,c\}\ \ 4.\{a,b,c,d\}\ \ 5.\{a,b,c,d,g\}\ \ 6.\{a,b,c,d,g,f\}$$

The difference between DFS and BFS are as follows. In DFS a stack is used instead of a linked-list as the frontier. At step 6, the node at the end of the stack is not removed, but only explored. After step 9, the for-loop is broken if an unvisited edge which is not already in the frontier is found. If no edges were found, then the last element added to the frontier is removed.

Note how this search algorithm essentially went down a path until it reached a solution. BFS on the other hand reached the optimal solution going from a to c then f, DFS found a solution that wasn't optimal as it gives a path of a, b, c, d, g, then f.

3.4 The Best-first and A-star search algorithms

The Best-first and A-star search algorithms are essentially implementations of BFS with heuristics. Best-first uses the future predicted path cost between each of the current node's children and the goal state to rank which node to visit next. The frontier must therefore be sorted in terms of nodes that have the least cost to be removed first. To implement this, a priority-queue was utilised in conjunction with CostNode objects which implement the comparable interface. This allows the priority-queue to sort the nodes in the frontier. Note that the tie-breaking strategy remained the same as with the

BFS/DFS implementation. Given two nodes with the same cost, their indexes are compared so that the node that is lower on the alphabetical series takes precedence.

The A-star search algorithm like Best-first uses a heuristic to calculate the future predicted path cost, however, in addition to this, the algorithm also keeps track of the path cost so far.

In terms of heuristics, both algorithms require admissible distance measures for the predicted path cost (Russell & Norvig, 2016). Two different admissible heuristics were explored. The first is the Euclidian distance measure which is essentially an implementation of the Pythagorean Theorem. A function was written to calculate the Euclidian distance which can be found in the InformedSearch class with the respective Best-first and A-star search algorithms. Also in this class is another function that calculates the Chebyshev distance between two points. The respective A-star and Best-first search algorithms that use the latter distance measure, are also found in the InformedSearch class.

Please note. While the A-star algorithm that uses the Chebyshev distance measure uses this measure to calculate the future path cost, the actual path cost (so far) is calculated using the Euclidian distance measure. This is because Chebyshev will not return the actual path cost unless the path is either vertical or horizontal.

3.6 Bidirectional Search

A bidirectional search was also implemented and can be found in the ExtendedSearch class. This bidirectional search algorithm is essentially two A-star search algorithms working towards each other one starting from the initial node, and the other working backwards from the goal node. The goal state for this algorithm was defined as when the Boolean explored arrays (from both sides) contain a state visited by both. For example, if both searches for the graph in figure 2 visit c, then the goal state is reached. The path from both searches is then combined and returned.

A function to "rewind" one of the nodes (the node that reaches the meeting state first) back to the meeting state was also written. This was needed as often one of the search algorithms reaches a specific state a step or two before the other. Once both the nodes have the same end state, a path can be compiled.

3.7 User interface classes

Three console driven user-interface programs were implemented to facilitate easy interaction between the user and the programs. Figures 3 to 5 show the user-interface programs for each jar file generated. The first user interface allows the user to toggle between the BFS and DFS algorithms as well as one of the 9 maps given for this assignment. Entering the number 3 will then launch the simulation, which, will print out the frontier, explored list, and the removed node from the frontier at each step, followed by the path at the end.

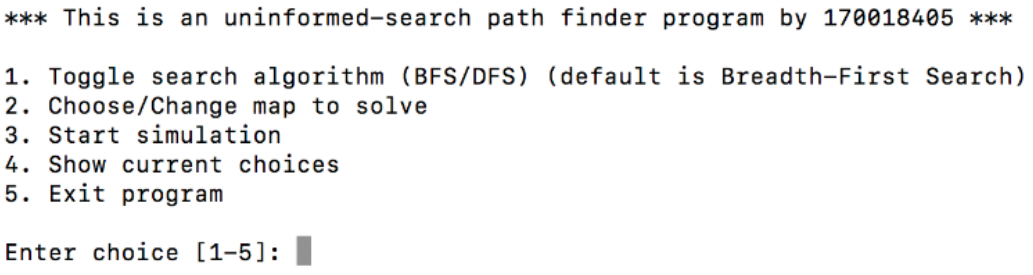

Figure 3 - search1 user-interface

The search2 jar contains the Best-first and A-star search algorithms as well as an option to change the heuristic method used by either of the search algorithms selected.

```
*** This is an Informed-search path finder program by 170018405 ***

1. Toggle search algorithm (BestFS/A*S)  default is Best-First Search
2. Toggle heuristic (Euclidean/Chebyshev)  default is Euclidean
3. Choose/Change map to solve
4. Start simulation
5. Show current choices
6. Exit program

Enter choice [1-6]:
```

Figure 4 - search2 user-interface

The search3 jar contains just the bidirectional search algorithm which can be run on any of the 9 maps given.

```
*** This is the Bidirectional search path finder program by 170018405 ***

1. Choose/Change map to solve
2. Start simulation
3. Exit program

Enter choice [1-3]:
```

Figure 5 - search3 user-interface

4.0 Testing

To test the implementation, the State and Node objects were tested first. Testing included checking to see if the correct data is returned without errors when using getter methods. Additionally, it was vital to check that the compareTo methods worked as expected. To test for this dummy Node objects with and without cost values were compared to see if the tie-breaking strategies worked as expected. Once this was verified, a priority queue was populated with dummy nodes to check if the expected order of nodes removed.

Next, tests were performed to ensure that the BFS and DFS algorithms worked as expected. Example maps which were solved by hand were converted to adjacency matrices and simulated. Consider the example map in Figure 2. When tested using the system, the results shown in Figure 6 and 7 were obtained which matched the hand-written solutions (section 3.3). Moreover, to test other functionalities, such as the backtracking mechanism of DFS, the connections to the goal state were removed. The results for this simulation also matched the hand-written solution (showing the agent moves from a-b-c-d-g, then backtracking to b, then to e, and then returning to a as no solution is found). The computer solutions can be found in Figures 8 and 9 for both BFS and DFS respectively.

Testing for part2 of the assignment involved ensuring that the tie-breaking strategy worked correctly. Once this was verified, a hand-written solution was written for some of map 9 to ensure that the differences between Best-first and A-star search were verified. This involved calculating the future costs from the child nodes to the goal for Best-first to confirm the path calculated by the computational model is the path expected.

5.0 Results

The following tables present the results obtained from the 9 maps tested. Table 1 presents the results for both the BFS and DFS algorithms. Table 2 presents the results for the Best-first and A-star search algorithms using both Euclidean and Chebyshev distance measures. Finally, table 3 presents the results obtained from the bidirectional search algorithm.

Map	Path		Number of nodes in route		States visited		Max number of nodes in frontier	
	DFS	BFS	DFS	BFS	DFS	BFS	DFS	BFS
1	d c e f g k h i j m n	d c j m n	11	5	18	14	11	5
2	d c j h f		5	5	10	11	5	5
3					26	13	11	5
4	a b c d l i h e f g k n m	a c j m	13	4	12	13	13	7
5	d c e f h i j m n	d c e f h k n	9	7	14	13	9	4
6	a b c d		4	4	3	5	4	2
7					10	5	4	2
8	o k g c b		5	5	4	12	5	6
9	h d c b a e f j	h d c b f j	8	6	7	13	8	3
Averages			7.9	5.1	11.5	11	7.7	4.3

Table 2- results for DFS and BFS

Map	Path		Length of route		States visited		Max number of nodes in frontier	
	Best	A*	Best	A*	Best	A*	Best	A*
1 E	d c j m n		12.32	12.32	6	4	6	7
1 C					4		7	
2 E	d c j h f		12.32	12.32	6	5	6	7
2 C								
3 E					13	13	6	6
3 C								
4 E	a c i n m	a c j m	20.74	9.96	4	3	10	8
4 C								
5 E	d c e f h i j m n	d c e f h k n	23.56	20.54	10	11	5	5
5 C					11		4	4
6 E	a b c d		13.55	13.55	3	4	2	2
6 C						6		
7 E					5	5	2	2
7 C								
8 E	o k g c b		12.00	12.00	4	4	8	8
8 C								
9 E	h g k o n m i j	h g c b f j	21.00	15.00	7	10	6	4
9 C								
Averages E:			16.5	13.7	6.4	6.5	5.7	5.4
Averages C:					6.3	6.7	5.7	5.3

Table 3 - results for Best-first and A-star search using both Euclidean (E) and Chebyshev (C) distance measures

Map	Path	Length of route	States visited	Max number of nodes in frontiers
1	d c j m n	12.32	5	13
2	d c j h f	12.32	5	11
3				
4	a c j m	9.96	3	14
5	d c e f h k n	20.54	11	6
6	a b c d	13.55	7	4
7				
8	o k g c b	12.00	7	16
9	h g c b f j	15.00	9	11
Averages:		13.7	6.7	10.7

Table 4 - results for bidirectional search algorithm

6.0 Evaluation

6. 1 Breadth-first vs Depth-first search

In terms of BFS vs DFS, it was clear that BFS always generated the shortest path in terms of the number of nodes in each path. On average, BFS returned solutions that were 3 nodes shorter. As for the number of visited states, both algorithms visited around the same number of states. Of course, where there is direct route that follows alphabetical ordering, DFS visited less states. However, in situations where a solution didn't exist or even if the goal node is connected directly to the initial node, but the goal node was the third option (for example), then DFS was way less efficient as it explored all other possible paths first before returning to the initial node. It was hence clear that BFS was the better algorithm when the goal node is within a limited depth, or, when the shortest path was required.

As for the space complexity of both algorithms, it seemed like BFS takes up less space as there were fewer nodes in the frontier at any one time. However, this number can be misleading as the DFS frontier typically contained nodes that linked to each other (i.e. within the same branch on the tree). A better measure for space complexity would have been to measure the number of nodes from different branches still in the frontier. In any case, DFS is expected to still take up less space as whole branches are removed from memory once a path is not found on the respective branch. This is of course true only if the depth of the problem was limited, given than DFS can continue searching a branch for infinity if there were no stopping conditions.

BFS: Optimal, complete. DFS: non-optimal, non-complete.

6.2 A-star vs Best-first search

The difference between A-star and Best-first search were less pronounced than the former algorithms as both of the latter are implementations of BFS. Given this fact, both algorithms visited around the same number of states and kept track of around the same number of nodes throughout the search process. Where the two algorithms differed was the path length (and the number of connections) traversed by the agent. A-star clearly always returned the shortest paths. In fact, those paths – with the exception of map 9 – were identical to BFS. The reason A-star visits g instead of d, is because the Euclidean distance between g and the goal state is closer than that from d. Best-first on the other hand didn't return the shortest paths. The reason for this is that the algorithm doesn't take into account the path length already traversed, and, only cares about the predicted future path length. This meant, that just like DFS, this algorithm tends to stay with a path until this path proves to either not lead to the goal state, or the predicted distance grows larger than one of the other paths in the frontier. For this reason, A-star is the better overall algorithm as it is both complete and optimal as opposed to just complete. Both the algorithms are better than DFS and BFS though given that almost half the states were visited to reach a solution.

6.3 Euclidian vs Chebyshev

Given the fact that both the distance measures were admissible, there were no differences in the paths computed for both informed algorithms. There were only minor differences in the number of states visited, and the number of nodes kept in the frontier. However, the differences were too minute for a conclusion to be formed.

6.4 Implementation improvements

Given more time, the following steps were identified as sources for further improvements:

1. The path compilation functionality for bidirectional search needs to be fixed so that an exception is not returned when no path exists.
2. Node objects should not keep track of their edges as this leads to extra space and time complexities.
3. A file reading system should be implemented to allow users to test their own maps.
4. The code could be adapted to allow users to change which states are goal and which are initial states.

7.0 Bibliography

Google. (2018, October 14). *How Search organises information*. Retrieved from Google.com: https://www.google.com/intl/en_uk/search/howsearchworks/crawling-indexing/

Hsu, F.-h., Campbell, M. S., & Hoane, J. A. (1995). Deep Blue System Overview. *9th international conference on Supercomputing* (pp. 240 - 244). Barcelona: ACM.

Kowalski, R. (1970). *Studies in the completeness and efficiency of theorem-proving by resolution*. Edinburgh: University of Edinburgh.

Russell, S., & Norvig, P. (2016). *Artificial Intelligence A Modern Approach* (3rd ed.). Upper Saddle River, NJ, USA: Pearson.

Spirin, N. V. (2014). People Search within an Online Social Network: Large Scan Analysis of Facebook Graph Search Query Logs. *ACM International Conference on Information and Knowledge Management* (pp. 1009-1018). Shanghai, China: ACM.

Ueno, K., Suzumura, T., Maruyama, N., Fujisawa, K., & Matsuoka, S. (2017). Efficient Breadth-First Search on Massively Parallel and Distributed-Memory Machines. *Data Sci. Eng.* , 2-22.

8.0 Appendix

8.1 Compilation instructions

The compiled programs (the jar files) for parts 1-3 can be found in the folder named jars. The class and manifest files used to make these jar files are found in the classes folder.

If you would like to recompile any of the source code, go to the src folder and compile the following classes:

For part 1: use the command javac UninformedSearchUI.java

For part 2: use the command javac InformedSearchUI.java

For part 3: use the command javac ExtendedSearchUI.java

```
Frontier: [a]
Explored and in Frontier: [a, , , , , , , ]
Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [b, c, d]
Explored and in Frontier: [a, b, c, d, , , ]
Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [c, d, e]
Explored and in Frontier: [a, b, c, d, e, , ]
Node: {
    State: [Label: c]    Depth: 1    Edges: [0, 1, 3, 5]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [d, e, f]
Explored and in Frontier: [a, b, c, d, e, f, ]
Node: {
    State: [Label: d]    Depth: 1    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [e, f, g]
Explored and in Frontier: [a, b, c, d, e, f, g]
Node: {
    State: [Label: e]    Depth: 2    Edges: [1, 5]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [f, g]
Explored and in Frontier: [a, b, c, d, e, f, g]
Node: {
    State: [Label: f]    Depth: 2    Edges: [2, 4, 6]
} Parent-Node: {
    State: [Label: c]    Depth: 1    Edges: [0, 1, 3, 5]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Path: [a, c, f]
```

Figure 6 - results of the BFS solution of the example map

```
Frontier: [a]
Explored and in Frontier: [a, , , , , , ]
Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b]
Explored and in Frontier: [a, b, , , , , ]
Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c]
Explored and in Frontier: [a, b, c, , , , ]
Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3, 5]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c, d]
Explored and in Frontier: [a, b, c, d, , , ]
Node: {
    State: [Label: d]    Depth: 3    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3, 5]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c, d, g]
Explored and in Frontier: [a, b, c, d, , , g]
Node: {
    State: [Label: g]    Depth: 4    Edges: [3, 5]
} Parent-Node: {
    State: [Label: d]    Depth: 3    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3, 5]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c, d, g, f]
Explored and in Frontier: [a, b, c, d, , f, g]
```

Figure 7 - results of the DFS solution of the example map

```
Frontier: [a]
Explored and in Frontier: [a, , , , , , ]
Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [b, c, d]
Explored and in Frontier: [a, b, c, d, , , ]
Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [c, d, e]
Explored and in Frontier: [a, b, c, d, e, , ]
Node: {
    State: [Label: c]    Depth: 1    Edges: [0, 1, 3]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [d, e]
Explored and in Frontier: [a, b, c, d, e, , ]
Node: {
    State: [Label: d]    Depth: 1    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [e, g]
Explored and in Frontier: [a, b, c, d, e, , g]
Node: {
    State: [Label: e]    Depth: 2    Edges: [1]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [g]
Explored and in Frontier: [a, b, c, d, e, , g]
Node: {
    State: [Label: g]    Depth: 2    Edges: [3]
} Parent-Node: {
    State: [Label: d]    Depth: 1    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}
```

Figure 8 - result from using BFS on the modified example map

```
Frontier: [a]
Explored and in Frontier: [a, , , , , , ]
Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b]
Explored and in Frontier: [a, b, , , , , ]
Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c]
Explored and in Frontier: [a, b, c, , , , ]
Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c, d]
Explored and in Frontier: [a, b, c, d, , , ]
Node: {
    State: [Label: d]    Depth: 3    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c, d, g]
Explored and in Frontier: [a, b, c, d, , , g]
Node: {
    State: [Label: g]    Depth: 4    Edges: [3]
} Parent-Node: {
    State: [Label: d]    Depth: 3    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c, d]
Explored and in Frontier: [a, b, c, d, , , g]
```

```
Explored and in Frontier: [a, b, c, d, , , g]
Node: {
    State: [Label: d]    Depth: 3    Edges: [0, 2, 6]
} Parent-Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, c]
Explored and in Frontier: [a, b, c, d, , , g]
Node: {
    State: [Label: c]    Depth: 2    Edges: [0, 1, 3]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b]
Explored and in Frontier: [a, b, c, d, , , g]
Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b, e]
Explored and in Frontier: [a, b, c, d, e, , g]
Node: {
    State: [Label: e]    Depth: 2    Edges: [1]
} Parent-Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a, b]
Explored and in Frontier: [a, b, c, d, e, , g]
Node: {
    State: [Label: b]    Depth: 1    Edges: [0, 2, 4]
} Parent-Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}

Frontier: [a]
Explored and in Frontier: [a, b, c, d, e, , g]
Node: {
    State: [Label: a]    Depth: 0    Edges: [1, 2, 3]
}
```

Figure 9 - results from using DFS on the modified example map