ASSIGNMENT 1

Artificial Intelligence.

Carolina Raymond Rodrigues.

Student ID: 210607000

ec21209@qmul.ac.uk

INDEX

AGENDA-BASED SEARCH. 3

Implement DFS, BFS, and UCS 3

DFS 3

BFS 3

UCS 3

Path functions. 3

Compare DFS, BFS, and UCS. 4

Extending the cost function. 5

Heuristic search. 7

ADVERSARIAL SEARCH. 8

Play optimally (MINIMAX algorithm). 8

Play optimally, quicker thinking! (α-β pruning) 8

Entry Free to play. 9

What are the ranges of values for x that will be still worth playing the game for the MAX player (to enter the game at all)? 9

For a fixed value of x, do you prefer to be the first player (MAXimiser) or the second player (MINimiser)? 9

APPENDIX. 10

INDEX OF FIGURES.

Figure 1Example of a non-optimal BFS path 4

Figure 2 Westminster to Piccadilly Circus search tree 5

Figure 3 Possible node station paths. 6

Figure 4 Path from Canada Water to Stratford. The blue route is for vanilla UCS and the red for improved UCS 6

Figure 5 Two interesting routes for the heuristic explanation. 7

Figure 6 Solution for the provided tree using Minimax. 8

Figure 7 Solution for the provided tree using Minimax. 8

Figure 8 Entry Fee to play tree 9

Figure 9 Summary of the experimental results. 10

AGENDA-BASED SEARCH.

Implement DFS, BFS, and UCS.

This section aims to implement DFS, BFS, and UCS algorithms to find the path from an initial to a goal station using the London Tube. For that, a CSV that contains useful information in terms of a logical relation between stations (average time between stations, tube line, zones…) has been given.

Python will be used to build the agenda-based search mechanisms. Therefore, the first thing to do before implementing each algorithm is to load all the data from the CSV file. Once this has been done, the class Graph from the Python library NetworkNx has been used for modelling the dataset.

After correctly uploading and modelling the data, the next step is to implement each algorithm. For that, a class Node() has been created. This class gathers all the necessary features needed to represent the relation between states, such as the current station, the previous one, the time taken from one station to the other, and the tube line that connects them.

Succeeding the class Node generation, AI route mechanisms will be implemented using classes, that will be defined according to how the frontier (an array that contains the nodes to be expanded) must work for each algorithm. In the following lines, each class will be explained.

DFS

For the implementation of the Depth for Search method, a class that contains four functions will be used:

- *__init__(self):* class initialization.
- *addNode (self, node):* appends node at the end of the frontier.
- *isEmpty(self):* checks if the frontier is empty.
- *removeNode(self)*: extracts the node that is at the end of the frontier following a LIFO structure as it is expected in DFS. This function is key for the algorithm, as it assures that it will expand only one node to its deepest level of the tree.

BFS

This class has been implemented as a child of the DFS class. The reason behind this decision is that the only thing that must be changed is the way the nodes are removed from the frontier.

The *removeNode(self)* function must follow a FIFO structure, therefore, it must extract the node that is at the beginning of the frontier (at index 0). This function will guarantee the correct functionality of the algorithm since all the nodes of depth d will be expanded before the nodes at the next level.

UCS

This algorithm is similar to BFS; however, it will expand first the nodes that have a lower cost. Consequently, it has been implemented as a child of the BFS, and the function *removeNode(self)* will be overwritten.

In this case, the frontier will be sorted in ascending order, taking into account the average time taken to commute from the starting station to the current node, therefore this time value will be the cost of our algorithm.

Once the sorting is completed, the function will return the node that is at the index 0 of the frontier, just as it was done with the BFS class.

Path functions.

Finally, two more functions have been defined. Both will be common for each algorithm and will help to find a path to the provided destination.

The function *findPath* takes as parameters the initial state, the destination, the tubeData graph that contains all the data that was loaded and modelled in the first step, and the algorithm that we want to use.

The function will create the root node, for later storage in the frontier. Besides a set known as explore will save the stations that have already been explored to avoid expanding nodes that have been checked.

Later, an iterative process takes place, where the functions implemented for the chosen algorithms will be used. First, a node from the frontier will be removed, if it corresponds to the destination, the *routeToDestination* function will be used. This function will backtrack the path, from the goal node that it is currently in, to the starting point. The total average time, the number of explored nodes and the route will be returned.

On the other hand, if the removed node does not correspond to the destination, the adjacent stations will be added to the frontier, previously creating child nodes for this cause.

Compare DFS, BFS, and UCS.

Different experimental routes have been tried for evaluating each algorithm. Table 1 in the Appendix summarizes the most relevant information extracted from the analysis.

As it can be seen in the table, DFS does not always give the shortest route to our destiny, since it has only given the shortest path on two occasions (from Ealing Broadway to South Kensington and from Baker Street to Wembley Park). Because of the expansion structure it follows, DFS will choose a node and expand the tree until it finds the destination. Therefore, the order in which nodes are added to the frontier matters. If we are lucky enough, the first node that will be explored will be optimal and consequently, a low number of nodes will be expanded and the shortest path will be found, just as it can be seen in the route from Baker Street to Wembley Park, where DFS is the best method since it not only finds the shortest path, but it also expands a lower number of nodes in comparison to the other two agenda-based mechanisms.

However, there might be a case in which the selected node from the frontier is the worst possible case. For instance, in the route from Ealing Broadway to South Kensington, the algorithm started expanding the possible paths from West Acton. Since it didn't find a route, it then went back and started expanding the path from Ealing Common to South Kensington. Nevertheless, this path is not optimal. It takes 37 minutes more than the other routes, and the number of expanded nodes is elevated, making it computationally expensive.

Better insight into how the order of the frontier matters, can be seen at the end of Table 1. The route from Westminster to Piccadilly Circus has been analysed. However, the last 3 algorithms will add to the frontier the possible paths from the station in reverse order. In the first case, we will take 23 minutes to reach, and 85 nodes have been expanded. But, in the second case, it will take 5 minutes to reach and only 5 nodes will be expanded.

On the other hand, when it comes to BFS, even though it is more consistent than DFS, it will not always find the best path as can be seen in Table 1. In the path from Canada Waters to Stratford, the time taken to reach the destination is higher than in the case of UCS. This is due to its queue structure nature. This mechanism will expand all the nodes to a certain depth d, before expanding the nodes on the next level (d+1). Therefore, again, the order matters. In the following figure, it can be seen how if the child of two nodes of the same level expands to the goal state, the algorithm will choose the one that is first in the frontier. Since node A is saved before B, the former one will be expanded, and the target has been reached. However, the time taken to reach that destination is higher, obtaining consequently the non-optimal path.

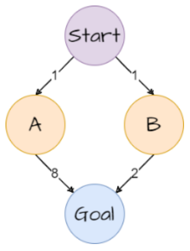

Figure 1Example of a non-optimal BFS path.

Additionally, the order of the nodes in the frontier will also have an influence on the nodes that must be expanded. A good example is a path from Westminster to Piccadilly Circus (Table 1), in both cases, the time taken to reach the destination, and the path are the same. However, when the order of the stations added to the frontier changed, the number of expanded nodes changed too.

As it has been seen in the previous paragraph, order matters, therefore the UCS algorithm has been implemented. In this mechanism, the algorithm will sort the frontier taking into account the time it takes from the starting station to the current station. In the table, it can be seen how this algorithm is the most consistent when it comes to time. No matter how the nodes are added to the frontier, the algorithm will always find the shortest path as it can be seen in the route from Westminster to Piccadilly Circus. Nevertheless, when it comes to the number of expanded nodes, it may be more computationally expensive than BFS. As is observable in the table, the path from Wembley Street to Baker Street will require less expansion from BFS(17 nodes) than from UCS (71 nodes). This is because order still matters in the case in which both nodes have the same cost. The following figure will give a better picture of this problem.

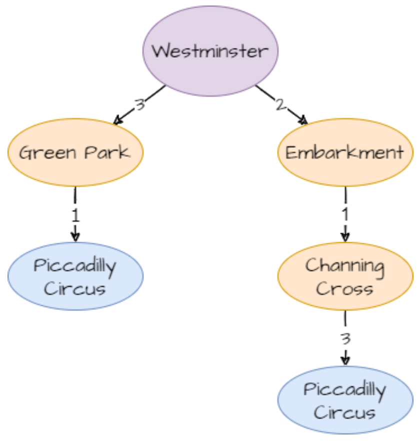

Figure 2 Westminster to Piccadilly Circus search tree.

First, the frontier will add Green Park and Embarkment. Since the former one has the lowest cost, it will be explored first. Then, the frontier will store Channing Cross. When sorting the frontier, since Green Park and Channing Cross have the same cost (3), it might be possible that the former one will be at index 0 of the array, being the first one to be explored. However, when Channing cross is expanded, Piccadilly Circus will have a cost of 6 while Green Park will have a cost of 3. Consequently, the algorithm will go back and start expanding Green Park, obtaining then the optimal solution. In this case, we will expand 3 more nodes, than in the case of expanding Green Park before Channing Cross.

However, even though it might be computationally expensive in some cases, UCS will always find the best path, being, therefore, the best method out of the other two. Nevertheless, some improvements can be made to reduce the number of expanded nodes. These improvements will be discussed in the following sections.

Extending the cost function.

In this section, an enhancement of the cost function has been carried out. Although we are given with the time it takes to commute from one station to another, it is important to take into account the tube line changes. Walking to another platform will add some additional time to the journey and in some cases, it will not be worth it to change the tube line.

The implemented cost function will try to take into account this, and therefore will penalise tube line changes, incrementing the value of the cost function. This augmentation can alter the order of the nodes in the frontier and therefore an optimal path that takes into account the tube line changes will be obtained.

For that, a new class has been implemented. As before, this class is a child of the BFS class and the *addNode* function will be overwritten. In this particular case the function will accept 2 parameters, the node that has to be added to the frontier and the number of platforms the previous station had (this value will be equal to the number of stations that can be reached directly from the parent station).

Inside the *addNode(self, node, platforms)* two parts can be differentiated. The first one is to calculate the cost function as it was done in the UCS class. The second part is to add to that cost, a new value that will penalise the current cost if the tube line has been changed. The following image will give a better insight into how the algorithm works:

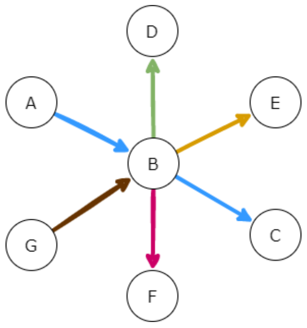

Figure 3 Possible node station paths.

According to the previous figure, if the current station is C, the algorithm will check if the previous station (B) comes from the same tube line, then the one it is currently adding. If the path we want to follow is from A-B-C, no penalisation will be added since no line change has occurred. However, if the path we followed is G-B-C, a penalty will be added since we have changed from the brown line to the blue line.

Since the tube change occurred on station B, the walking time taken to travel from one platform to another is taken into account. The minimum time it takes to change from one tube to another is 2 minutes, however the bigger the station, longer will be the time taken to reach some platforms. To bias this, if the number of platforms is between 4 and 5, an extra minute will be added per platform. If there are even more platforms, 30 seconds per platform will be added instead.

After implementing the algorithm, the same experimental routes analysed in the previous question are evaluated again. As it can be seen in Table 1, if the new cost function is compared to the old one, the number of explored nodes tends to be higher. Now, the explore set will also take into account the tube line, not only stations that have been explored, augmenting, therefore, the number of nodes that must be expanded.

Moreover, in some cases like the path from Canada Water to Stratford, it is observable that the time taken to reach the destination is 1 minute higher than in the vanilla UCS. However, we must highlight that this time does not take into consideration the extra time needed to walk through the station when changing the tube line. If we make a further analysis of this route, the following image will prove that the vanilla UCS, represented with different shades of blue, will change two times the tube (each shade of blue is a different tube line). In contrast, the improved UCS, will not change the tube line at all. Proving, therefore, that even if vanilla UCS takes 14 minutes to reach its destination, the time will end up being higher than the needed in the other route.

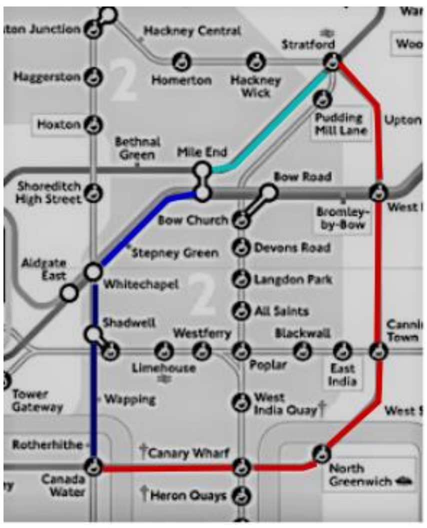

Figure 4 Path from Canada Water to Stratford. The blue route is for vanilla UCS and the red for improved UCS.

Heuristic search.

As it has been seen, the previous algorithm can be computationally expensive, therefore the next step will be increasing the efficiency. the A* algorithm will be used to improve the performance. This new mechanism will take into account the past using the previous cost function, but will also take into consideration the expected future cost.

The future cost will be estimated with help of the zone of the current station. The algorithm will use this knowledge for searching in the right direction.

For that, the node class will be expanded and two additional parameters will be stored: the main and the secondary zone of the station. Moreover, the new UCS_heuristic class will update the UCS_im class, since the heuristic algorithm will be added.

To provide a good explanation of the heuristic, the following image will be used.

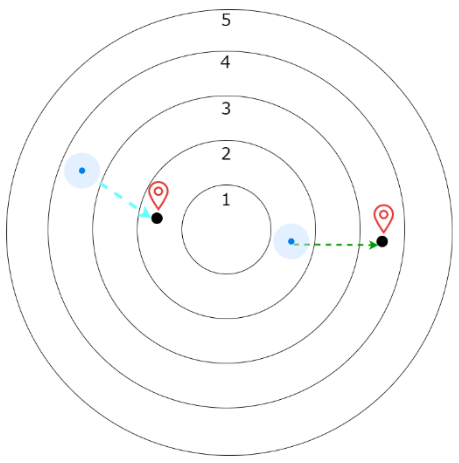

Figure 5 Two interesting routes for the heuristic explanation.

As it can be seen the distribution of the zones is concentric. Therefore, in the case of the green path, we are first in zone 2 and want to go to zone 3. If the route the algorithm is exploring, takes it to a zone lower than 2 (1 for this case), it will be penalised with 2 extra minutes. In the case of staying in the same zone, the penalisation will be of 1 minute and 30 seconds. If it goes to a zone higher than 2 and a zone lower than the end node zone (3 for this case), it means that is going in the correct direction, consequently, the penalisation will be of 1 minute. However, if the node keeps expanding to the station that is far from the end zone (4, 5, 6, a, b, c and d, for this case) it will be again penalised with 2 more minutes, being then ordered in a further position on the frontier.

The same will happen with the blue route, the importance of the step taken to get closer to the end zone will influence the heuristic. If we stay in zone 4, the penalisation will be 30 seconds higher than if we move to zone 3. If we end up in zone 1, the same algorithm used in the green route will be used, creating therefore the need to explore again outer zones instead of the inner ones.

After implementing the algorithm, the experimental routes of the previous sections have been evaluated again. As it can be seen in Table 1, the number of expanded nodes for the UCS with the heuristic has decreased in comparison to the improved UCS. These results were expected because thanks to the heuristic the branching factor has decreased, the nodes that move in the correct zone will be put first in the frontier and consequently, the number of expanded nodes will decrease.

ADVERSARIAL SEARCH.

Play optimally (MINIMAX algorithm).

Minimax is a decision rule algorithm commonly used in Artificial Intelligence for two-player zero-sum games with perfect information. These two players are known as the MIN and the MAX player. It is assumed that both of them play optimally, therefore they will come up with the best strategy to beat their opponent.

In the provided tree, we are the MAX player and consequently, we will try to maximise the worst-case outcome supposing that the MIN player will try to minimise the utility of our moves.

In this section the terminal state values were given. Thus, as it can be seen in the following figure, the MAX player will choose the action that leads to the right of the node since it is the one that will give the highest reward.

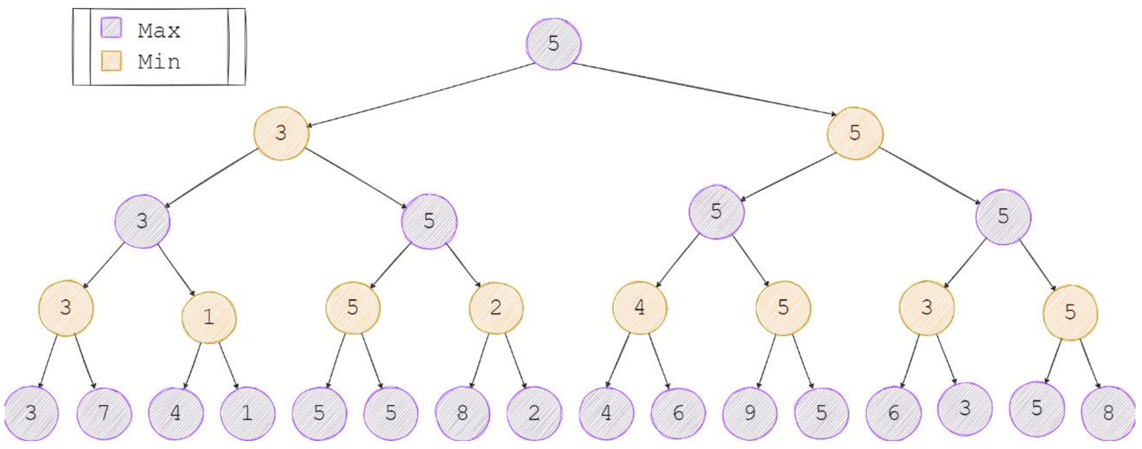

Figure 6 Solution for the provided tree using Minimax.

Play optimally, quicker thinking! (α-β pruning)

$\alpha - \beta$ pruning will be applied to reduce the number of nodes to be evaluated while using the MiniMax algorithm. Ranges will be considered instead of the values.

In the node highlighted with pink, β pruning is applied. As it can be seen in the pink node, the MAX player will choose a value higher or equal than 5, however, its parent (blue node) is the MIN player, consequently taking into account that the other child (green node) is 3, MIN player will always prefer a 3.

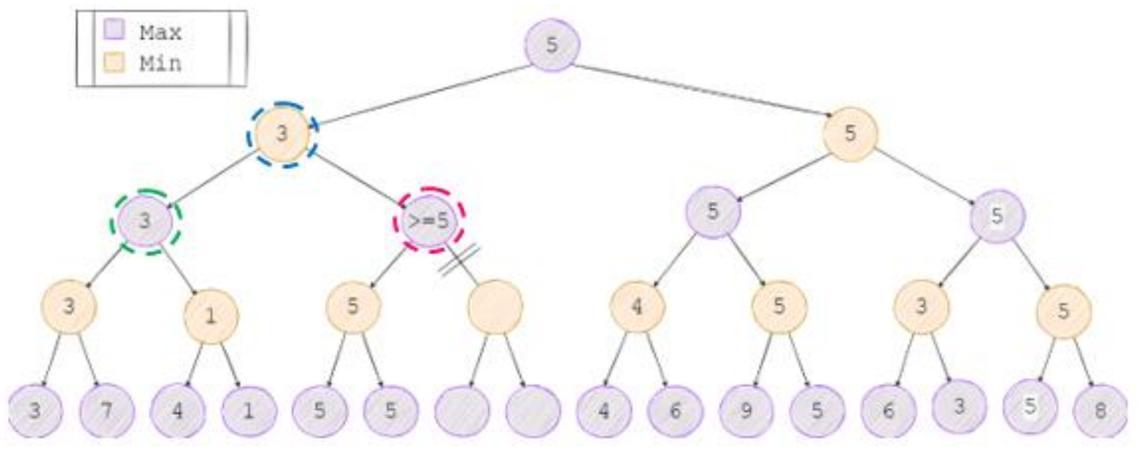

Figure 7 Solution for the provided tree using $\alpha - \beta$ pruning.

Entry Free to play.

What are the ranges of values for x that will be still worth playing the game for the MAX player (to enter the game at all)?

If the MIN player does not play optimally in any of the cases, we could have the following tree instead:

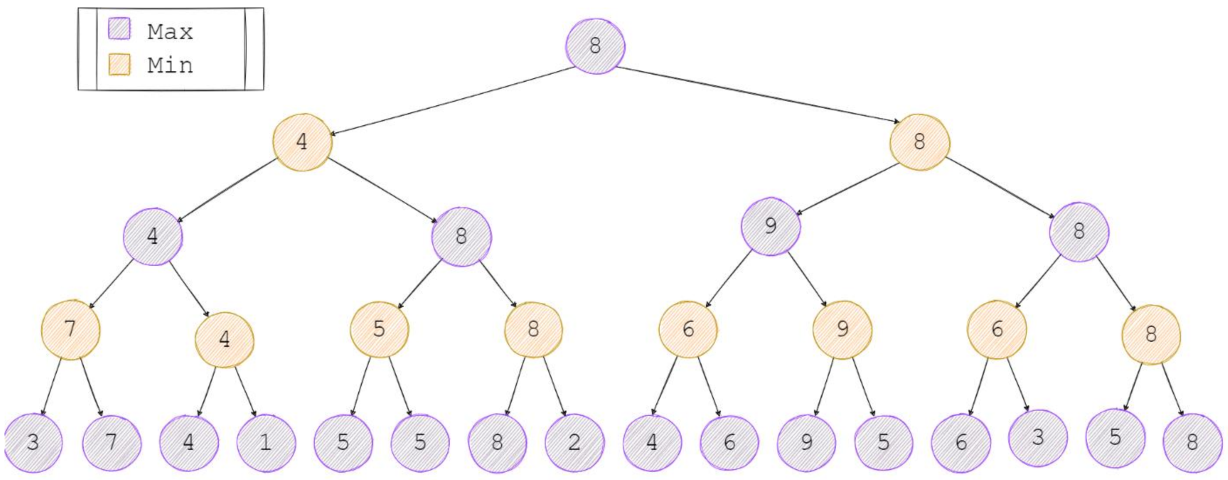

Figure 8 Entry Fee to play tree

In this scenario, if we choose the left node instead of the right one at the beginning of the game, we will get the minimum reward, 4, therefore since we don't want to have any lost and the function for the worst possible scenario will be $4 - x \geq 0,$ it will be advisable not to pay more than 4 units.

For a fixed value of x, do you prefer to be the first player (MAXimiser) or the second player (MINimiser)?

As it has been proved before, it will depend on the terminal states. If those are known, as well as the deepness and all the possible states of the tree (perfect information game), then we will be able to decide if it is worth it to play first just as we did in the previous case.

APPENDIX.

Algorithm	Starting station	Destination	Total time	Number of expanded nodes.
DFS	Euston	Victoria	13	26
BFS			7	36
UCS			7	31
UCS_IM			7	18
UCS_H			7	18
DFS	Canada Water	Stratford	15	7
BFS			15	41
UCS			14	53
UCS_IM			15	63
UCS_H			15	41
DFS	New Cross Gate	Stepney Green	27	33
BFS			14	27
UCS			14	19
UCS_IM			14	28
UCS_H			14	21
DFS	Ealing Broadway	South Kensington	57	180
BFS			20	51
UCS			20	54
UCS_IM			20	59
UCS_H			20	55
DFS	Baker Street	Wembley Park	13	4
BFS			13	17
UCS			13	71
UCS_IM			13	80
UCS_H			13	78
DFS	Epping	Wanstead	20	11
BFS			20	16
UCS			20	14
UCS_IM			20	15
UCS_H			20	14
DFS	Westminster	Piccadilly Circus	23	85
BFS			4	11
UCS			4	12
UCS_IM			5	21
UCS_H			5	21
DFS*	Westminster	Piccadilly Circus	5	5
BFS*			4	14
UCS*			4	12
UCS_IM*			5	21
UCS_H*			5	21

Figure 9 Summary of the experimental results.

*The neighbors list has been reversed.