Queen Mary, University of London
School of Electronic Engineering and Computer Science

ECS759P: Artificial Intelligence

Coursework 1

Mughees Asif | 180288337

Due: 22 November 2021

Agenda-based Search

Search methods

This report contains a comparison between three search methods that were used to calculate distances from one London Underground station to another. The provided dataset contained all the relevant information including the average time taken to reach from one station to another, and the associated tube line. The starting and ending stations were represented as nodes on a weighted undirected graph and the vertices/weights represented the average time taken to travel between the stations.

Depth-First Search (DFS)

Depth-First Search starts at the root node and expands each corresponding node till the end (maximum depth) of the branch. Once the required node is found, the algorithm backtracks to the root node. However, since each node leads to another node the algorithm can spiral into an unending sequence of possibilities. The time complexity of DFS is $O(b^m)$, where b is the branching factor and m is the maximum depth of the tree, however the space complexity is `O(bm)`.

Breadth-First Search (BFS)

Breadth-First Search starts at the root node and searches along each node in the first level till the end (maximum width) of the branch. If no solution is found, the algorithm backtracks to the left-most node and starts searching on the second level from left to right of the branch. This pattern continues at each new level. Both, the time and space complexity of BFS equate to $O(b^d)$ highlighting the extra memory space needed to store the pointers to keep track of the child nodes. d is the depth of shallowest solution.

Uniform Cost Search (UCS)

Uniform Cost Search is an extension of BFS that uses the lowest-cost node and works optimally if the costs increase monotonically (always increasing). The time and space complexity is $O(b^{1+C/\epsilon})$, where C is the optimal cost and ϵ is the cost of passing through each node.

Comparison

Table 1: Explored routes for each search method

Route	**Cost**			**Total average time taken**
	DFS	*BFS*	*UCS*	
Euston ⟶ Victoria	6	23	7	7
Canada Water ⟶ Stratford	8	10	15	14
Ealing Broadway ⟶ South Kensington	24	25	20	16
Baker Street ⟶ Wembley Park	8	8	13	13
Stonebridge Park ⟶ Maida Vale	8	11	14	14
Stratford ⟶ Tower Hill	19	28	17	16

Table 1 highlights the number of explorations that were executed by the search algorithms to reach the final destination. Evidently, DFS outperforms BFS and UCS. Since, the input graph was densely connected with a high branching factor i.e., the number of edges is equal to or near to the maximal number of edges, DFS will outperform BFS as an exponential increase in the number of expanded nodes is instigated due to an increase in the branching factor. In addition, UCS outperforms vanilla BFS by a small margin as the lowest-cost nodes (lowest average time taken) were always positive. This guaranteed that there is no path in the graph that would result in infinite expansion due to zero/negative-cost edges.

Moreover, UCS performs optimally for the longest journeys (Ealing Broadway → South Kensington & Stratford → Tower Hill) as the algorithm is destined to choose the lowest-cost nodes, thereby optimising the search for longer journeys, where DFS and BFS would explore every possible route whilst accumulating unnecessary cost.

Cost function

The UCS implementation for this research only considered the average time taken to find the path with the lowest-cost. However, this is not a practical application as in the real-world many other factors can cause an increase in the total cost of a route. Therefore, the cost function was improved by *discouraging* the search to continue using different underground lines. The search was constrained to use the same line as a penalty was applied at each instance where the underground line was changed. This is one example of improvement in the cost function but can be extended to incorporate more parameters relevant to the application.

Heuristic

A* Search

A* Search is a best-first search algorithm that uses a heuristic function to guide the search towards the goal node via the shortest path with the smallest cost. The algorithm chooses to expand towards a path, using the cost of the path and an estimate of the cost required to extend the path towards the goal node. Mathematically, the algorithm aims to minimise Eq. 1:

$$f(n) = g(n) + h(n) \tag{1}$$

where:

n = the proceeding node
$g(n)$ = cost of the path for the current node
$h(n)$ = heuristic function to find the lowest-cost path

The time complexity of A* depends on $h(n)$, where if $h(n) = 0$ (Dijkstra's algorithm), the algorithm does zero pruning of the paths that results in unnecessary exploration. The space complexity of the algorithm is exponential, $O(b^d)$, due to storage of pointers to keep track of the child nodes.

Table 2: Explored routes for the A* search method

Route	**Cost**	**Total average time taken**
	*A**	
Euston ⟶ Victoria	7	7
Canada Water ⟶ Stratford	15	14
Ealing Broadway ⟶ South Kensington	20	16
Baker Street ⟶ Wembley Park	13	13
Stonebridge Park ⟶ Maida Vale	14	14
Stratford ⟶ Tower Hill	17	16

For the purposes of this research, a custom heuristic function was implemented that generated a random number from 0 to 5 to mimic the workings of Manhattan or Euclidean Distance heuristics that return an optimal path from the target node. Due to the authors lack of understanding on how to represent the ending station (a node) as a co-ordinate point, the traditional heuristic functions could not be utilised. Table 2 displays the results of using the A* search algorithm on the same routes as the other algorithms. The results indicate that the random heuristic function *did not* work as intended as the cost of the journey is very similar to the vanilla UCS function which is surprising as a constant value for the heuristic would effectively transform A* search into UCS. However, the optimal path was found for all routes indicating the validity of the search algorithm.

Adversarial Search

Play optimally (MINIMAX algorithm)

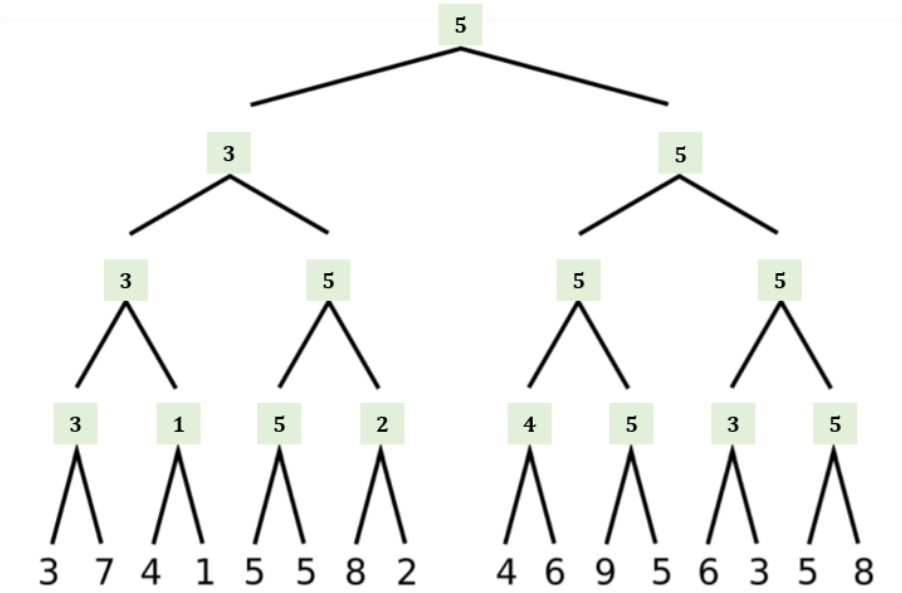

Figure 1: MINIMAX algorithm

Play optimally ($\alpha - \beta$ pruning)

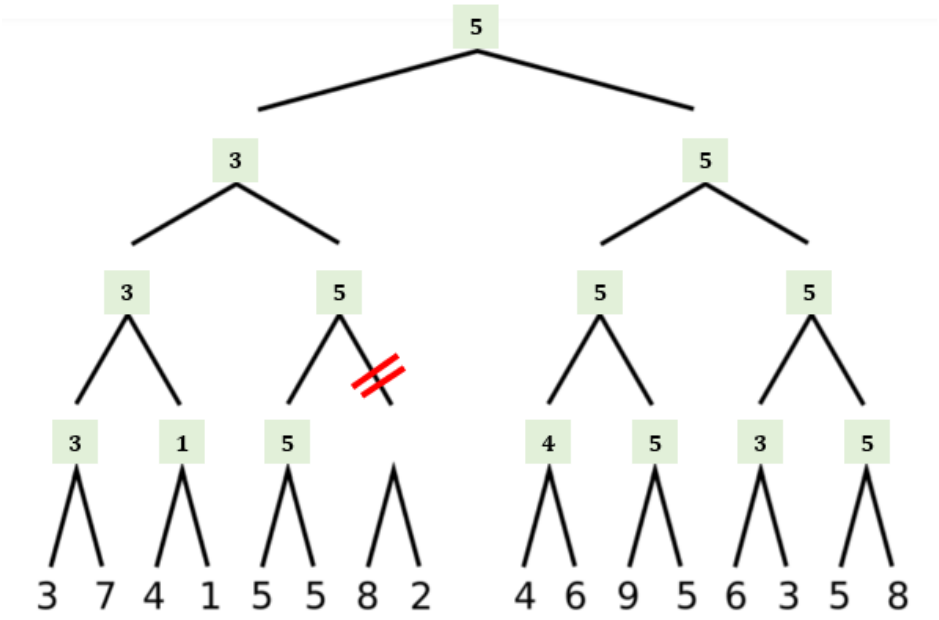

Figure 2: $\alpha - \beta$ pruning

Given $\alpha \geq \beta$, the highlighted node is pruned as the MAX player will always choose a node which is ≥ 5. Therefore, exploring the right child of the MAX node is unnecessary as the path will be never explored.

Entry free to play

1. Ranges of values for x worth playing the game for the MAX player: If $0 \leq x \leq 4$, the game is worth playing for the MAX player. $x > 5$ will incur a loss for the MAX player.
2. Preference: The preference depends on the value of x. If $x < 5$, the MAX player is preferred as $5 - x$ units determine the player's entry fee. Contrarily, if $x > 5$, the MIN player is preferred as $x - 5$ units would determine the entry fee.