Artificial Intelligence Assignment – 1

Palak Dhanadia (210748141)

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

ec21276@qmul.ac.uk

1 Introduction

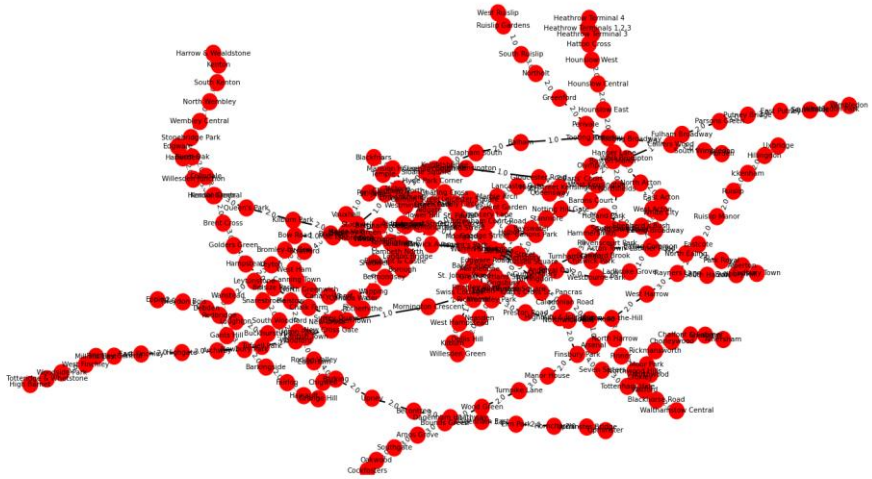
In this assignment, we have to do two parts:

- Agent Based Search
- Adversarial Search

2 Agent Based Search

In Agent Based Search, we will implement and compare Depth First Search(DFS), Breadth First Search(BFS) and Uniform Cost Search(UCS) as well as improving cost function of London Tube Data. Secondly, implementing Heuristic Search.

Making network of Tube Data which has 271 nodes with following network shown below:



2.1 Implementation of DFS, BFS and USC

- Depth First Search(DFS): It is used to traverse or explore data structures such as trees and graphs. The algorithm starts at the root node and explores as far as possible down each branch before backtracking.

  Pseudo Code of DFS:

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

- Breadth First Search(BFS): It is used to search a tree data structure for a node that meets a set of criteria. It begins at the root of the tree and investigates all nodes at the current depth level before moving on to nodes at the next depth level.

  Pseudo Code of BFS:

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

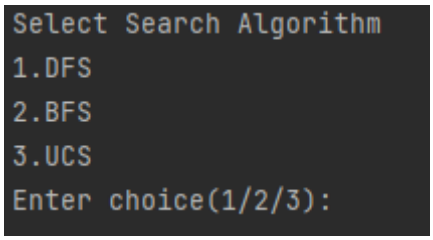
- Uniform Cost Search(UCS): It is a search algorithm that finds a path from the source to the destination by calculating the lowest cumulative cost. A priority queue is used to implement the uniform cost search.

  Pseudo Code of UCS:

```
function UNIFORM-COST-SEARCH( problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?( frontier) then return failure
    node ← POP( frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE( problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

When running the program we have to choose from which type of search algorithm we have to perform:

1. DFS
2. BFS
3. USC



After this it prompts us to enter starting and destination station. The route from Euston to Victoria is tested on and the result are shown below:

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	13	9	26
BFS	7	5	34
USC	7	5	29

2.2 Comparison of DFS, BFS and USC

Route: Canada Water to Stratford

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	15	6	7
BFS	15	6	39
USC	14	8	55

Route: New Cross Gate to Stepney Green

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	27	10	33
BFS	14	8	25
USC	14	8	18

Route: Ealing Broadway to South Kensington

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	57	21	180
BFS	20	9	49
USC	20	9	52

Route: Baker Street to Wembley Park

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	13	3	4
BFS	13	3	15
USC	13	3	77

Route: Barking to Manor House

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	66	31	71
BFS	39	16	120
USC	36	17	120

Route: Brent Cross to Blackfriars

Search Algorithm	Cost	Number of Nodes	Number of Explorations
DFS	52	28	58
BFS	28	15	98
USC	28	16	76

As seen from the above observation,

- BFS and UCS search algorithm has slight changes in terms of cost and number of nodes as if cost decreases then number of nodes can increase as well as cost increases number of nodes decreases.
- UCS focus on cost as every action has different cost whereas for BFS and DFS every action has same cost meaning there is no change in cost for different action.
- One of the example above i.e. Baker Street to Wembley Park, Cost and Number of nodes are same but just the Number of explorations are different with minimum exploration to DFS and maximum exploration to UCS.
- In most of the example above, we can see that BFS and UCS will have similar path, cost and number of nodes whereas DFS will differ from them.
- Since the map is large and we avoid graph loops, these techniques require much less time to run. However, this isn't always the case.
- The preceding statement also applies to space complexity. The space complexity is directly proportional to the number of nodes investigated.
- As also seen from above observation that UCS cost will always be less or equal to BFS and DFS but it will never be more than BFS and DFS.

2.3 Extending the Cost Function

A cost function is a mathematical formula that shows how the costs change when different search algorithm gives different path and cost is calculated by average time taken from going one station to another station. To put it another way, it calculates the overall cost of based on the average time taken from starting station to destination station. We tried improving the cost function based on tube line rather than average time taken.

2.4 Heuristic Search

A heuristic is a technique for solving a problem faster than traditional approaches or for estimating a solution when traditional methods fail. We frequently exchange one of optimality, completeness, accuracy, or precision for speed, therefore this is a form of shortcut. A heuristic (also known as a heuristic function) examines search strategies. It analyses the available information at each branching phase and decides which branch to take. It accomplishes this by ranking several options. Any device that is frequently effective but does not guarantee success in every situation is referred to as a heuristic. In this implementation, I tried to calculate the weights by getting primary and secondary zone based on heuristics.

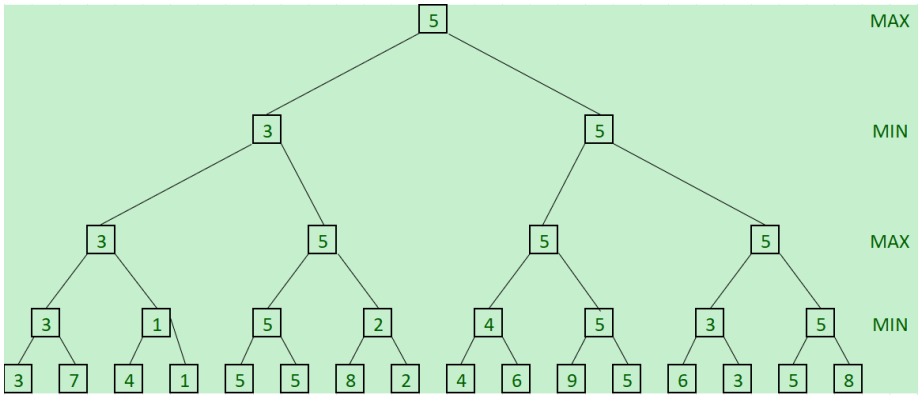
A* algorithm: Due to its completeness, optimality, and optimal efficiency, A* (pronounced "A-star") is a graph traversal and path search method. As a result, it is often outperformed in actual travel-routing systems by algorithms that can pre-process the graph to achieve better performance, as well as memory-bounded techniques; yet, in many circumstances, A* is still the best answer.

Pseudo Code of A* :

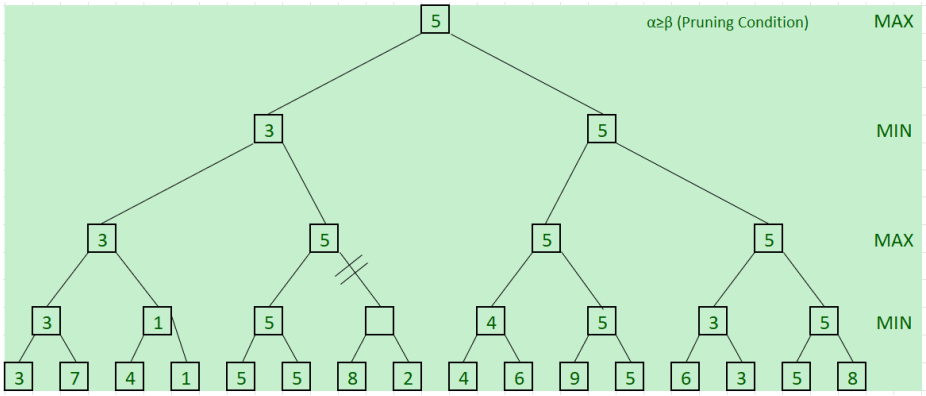
```
agenda = initial state;
while agenda not empty do
   take node from agenda such that
     f(node) = min { f(n) | n in agenda}
     where f(n) = g(n) + h(n)
   if node is goal state then
     return solution;
   new nodes = apply operations to node;
   add new nodes to the agenda;
```

3 Adversarial Search

3.1 MINIMAX Algorithm



3.2 α - β Pruning



The MAX (depth 3) player will choose a node larger than or equal to 5, therefor the node with cut is pruned. As the MAX player sees $\alpha \geq \beta$ where $\alpha = 5$ and $\beta = 3 > 2$ which is β cut off.

3.3 Entry free to play

3.3.1 The values of x lies in range {1,4}, which is when the game is worth playing. If the value of $x < 1$ or $x > 5$ then it will lose the game.

3.3.2 To be the first player i.e MAXimiser or the second player i.e MINimiser depends on the fixed value of x.

- $x > 5$ → Min = Win ; Max = Loss
- $x < 5$ → Max = Win ; Min = Loss
- $x = 5$ → No Win or Loss

References

1. ECS759P – Artificial Intelligence Lab Work