ECS759P: Artificial Intelligence: Coursework 1

Animesh Devendra Chourey

210765551

1 Agenda-based Search

The first thing that we have to do is load the data from tubedata.csv into the file.

```
def csv_data(file):
filename = open(file)
rows = []
csvreader = csv.reader(filename)

for row in csvreader:
    rows.append(row)
return rows

data = csv_data('tubedata.csv')
```

After successfully importing we need to process the data into proper format.

```
data[342][1] = data[342][1]+","+data[342][2]+","+data[342][3]

#Data at the 3rd col on 343rd row will be deleted
del data[342][2]
#Now from the new updated data after deleting that col, we will again delete the
3rd col at 343rd row
del data[342][2]

for row in data:
  row[0] = row[0].strip(' "\'\t\r\n')
  row[1] = row[1].strip(' "\'\t\r\n')
  row[2] = row[2].strip(' "\'\t\r\n')
  row[3] = int(row[3].strip(' "\'\t\r\n'))
  row[4] = row[4].strip(' "\'\t\r\n')
  row[5] = row[5].strip(' "\'\t\r\n')
```

Plotting and visualizing the graph:-

```
def plot_graph(data):
tube_graph = nx.MultiGraph()    # Create empty graph

for row in data:
    tube_graph.add_node(row[0])   #Add 0th column node of the row to the graph
    tube_graph.add_node(row[1])   #Add 1st column node of the row to the graph
    weight = row[3]   #Extract the weight from 4th column
    tube_graph.add_edge(row[0], row[1], weight = weight)

return tube_graph

tube_graph = plot_graph(data)
```

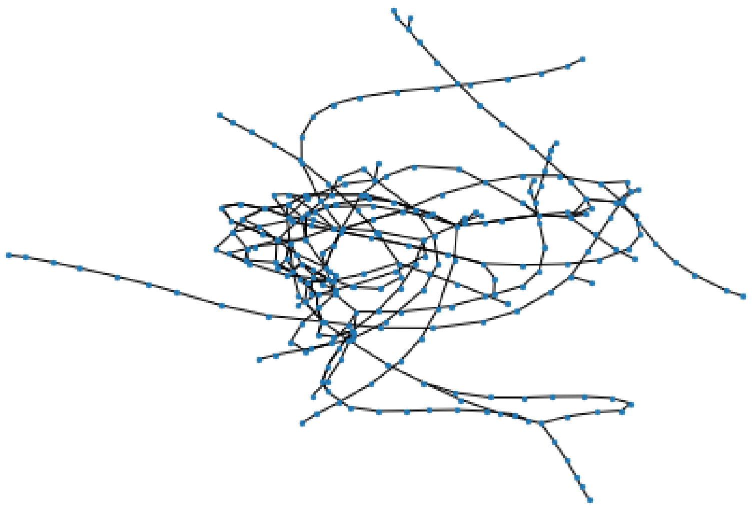

Figure 1

1.1 Implement DFS, BFS, and USC

1.1.1 Depth First Search

In Depth First Search, the algorithm starts from the root node and explores the search space along each branch as deep as it can before backtracking until it finds the solution.

The algorithm start by storing the start node and its parent in a *stack* variable which is a dictionary type variable. Another variable called *explored* is initialized with start node whose task is to ensure while traversing we do not visit the node again. No of nodes explored shows the number of nodes visited by the algorithm while traversing.

```
stack = [{'label':start_node, 'parent':None}]
explored = {start_node}
number_of_explored_nodes = 1
```

The loop keeps on traversing until all the nodes are explored. The algorithm firstly stores the starting point of the graph in the *node* variable. The *node* variable stores the last element appended to the *stack* variable which ultimately denotes the child of the node. Now the algorithm checks whether while iterating is the popped element stored in the stack is the target that the algorithm is tasked with to search. If the condition proves to be true the function returns the *node* variable and the explored nodes count.

```
while stack:
    node = stack.pop()
    number_of_explored_nodes = number_of_explored_nodes + 1
    if node['label'] == end_node:
        return node, number_of_explored_nodes
```

If the goal state is not attained, the algorithm finds all the neighbours of the popped element stored in the *node* variable and assigns those neighbours to the new variable *neighbours.* Now the algorithm iterates over every neighbour and one by one assigns each neighbour to a *child* variable along with its parent. This parent in the *node* variable from where we got all the neighbours. If the child is not already present in *explored* variable, we append this *child* node to the stack. This step represents that algorithm only goes down the search space available to it not to all of its children first. Also, that child is added to the *explored* variable in order to prevent algorithm from going down the same path again while traversing.

```
neighbours = tube_graph.neighbors(node['label'])
for children in neighbours:
    child = {'label':children, 'parent':node}
    if children not in explored:
        stack.append(child)
        explored.add(children)
```

Another function is defined to print out the path from the start node to the end node. A *path* variable is created to store the path taken by the algorithm. While iterating over the node variable, as the node is in the form of dictionary, with every iteration the value of the key is stores in the path and its value if again assigned to the node variable. This loop goes on until it iterates over every key and then return the path extracted form there.

```
def path_from_root(node):

path = [node['label']]
while node['parent']:
    node = node['parent']
```

```
        path = [node['label']] + path
    return path


    solution_dfs, nodes_visited_dfs = depth_first_search(tube_graph, 'Euston','Victoria')
    path_dfs = path_from_root(solution_dfs)
```

In order to calculate the cost of the depth first search, we iterate over the path found by the breath first search. We extract the row from the data with help of the networkx inbuilt function. From the row we extract the weight between the two said edges, and the add it to the overall cost i.e. "cost-dfs" of the algorithm.

```
    cost_dfs = 0
    for i in range(1,len(path_dfs)):
        edge_weight = tube_graph.get_edge_data(path_dfs[i-1], path_dfs[i])
        cost = edge_weight[0]['weight']
        cost_dfs = cost_dfs + cost

    print("BFS Path from Euston to Victoria :-> {}".format(path_bfs))
    print("Cost of BFS :-> {}".format(cost_bfs))
    print("Nodes Visited by BFS :-> {}".format(nodes_visited_bfs))
```

Output : -

DFS Path from Euston to Victoria :- ['Euston', "King's Cross St. Pancras", 'Russell Square', 'Holborn', 'Covent Garden', 'Leicester Square', 'Piccadilly Circus', 'Green Park', 'Victoria']

Cost of DFS :- 13

Nodes Visited by DFS :- 26

1.1.2 Breadth First Search

In Breadth First Search, the algorithm starts from the root node and explores the search space layer wise. During the traversal it explores all the child nodes of the current node and then only goes down one layer to search along their children.

The algorithm starts by checking whether the start node is equal to end node just in case as we are checking the children this time while traversing. It then assigns start node and its parent to *stack* variable. Another variable called *explored* is initialized with start node whose task is to ensure while traversing we do not visit the node again. No of nodes explored shows the number of nodes visited by the algorithm while traversing.

```
    if start_node == end_node:
        return None

    number_of_explored_nodes = 1
    frontier = [{'label':start_node, 'parent':None}]
```

```
explored = {start_node}
```

The loop keeps on traversing until all the nodes are explored. The algorithm firstly stores the starting point of the graph in the *node* variable. The algorithm finds all the neighbours of the popped element stored in the *node* variable and assigns those neighbours to the new variable *neighbours*. Now the algorithm iterates over every neighbour and one by one assigns each neighbour to a *child* variable along with its parent. This parent is the *node* variable from where we got all the neighbours. If any of the child node is the goal node that the algorithm is tasked with to search for, the algorithm returns the *child* variable and the explored nodes count. If the children is not already present in *explored* variable, we append this *child* node to the beginning of the *frontier* variable. This functionality serves as queue data structure .Also, that *child label* variable is added to the *explored* variable in order to prevent algorithm from traversing through the same node again.

```
while frontier:
    node = frontier.pop() # pop from the right of the list

    neighbours = tube_graph.neighbors(node[’label’])

    for child_label in neighbours:
        child = {’label’:child_label, ’parent’:node}
        if child_label == end_node:
            return child, number_of_explored_nodes

        if child_label not in explored:
            frontier = [child] + frontier # added to the left of the list, so a FIFO!
            number_of_explored_nodes += 1
            explored.add(child_label)
```

To print out the path from the start node to the end node the same function is called when we need to find the path for depth first search.

```
solution_bfs, nodes_visited_bfs = breadth_first_graph_search(tube_graph,’Euston’,’Victoria’)
path_bfs = path_from_root(solution_bfs)
```

In order to calculate the cost of the breadth first search, we iterate over the path found by the breath first search. We extract the row from the data with help of the networkx inbuilt function. From the row we extract the weight between the two said edges, and the add it to the overall cost i.e. ”cost-bfs” of the algorithm.

```
cost_bfs = 0
for i in range(1,len(path_bfs)):
   edge_weight = tube_graph.get_edge_data(path_bfs[i-1], path_bfs[i])
   cost = edge_weight[0][’weight’]
   cost_bfs = cost_bfs + cost
```

```
print("BFS Path from Euston to Victoria :- {}".format(path_bfs))
print("Cost of BFS :- {}".format(cost_bfs))
print("Nodes Visited by BFS :- {}".format(nodes_visited_bfs))
```

Output : -

BFS Path from Euston to Victoria :- ['Euston', 'Warren Street', 'Oxford Circus', 'Green Park', 'Victoria']

Cost of BFS :- 7

Nodes Visited by BFS :- 34

1.1.3 Uniform Cost Search

Uniform Cost Search is a type of uninformed search algorithm which factors in the lowest weight or cost while traversing to find the path from start node to the end node. The nodes with the lowest weight or cost are expanded first.

The algorithm is defined in the uniform cost search() function. The algorithm starts by creating list called as *stack* and an empty dictionary called as *stack index*. Also an empty variable called *explored* is created as a set so that it does not store duplicate values. This comes in handy when we do not want multiple nodes stored while we are keeping track of nodes visited is not getting repeated. A *node* variable is created in the form of tuple to keep the ordered track of nodes as part of a queue. *stack index* keeps a track of the node and its cost. The algorithm then appends the node to the stack variable.

```
stack = []
stack_index = {}
explored = set()
node = (0, start_node, [start_node])
stack_index[node[1]] = [node[0], node[2]]
heapq.heappush(stack, node)
```

Then the algorithm gets into a loop for traversing until all the nodes are not explored. Firstly it checks if the stack is empty of not. Then it pops the last element in the stack variable in the node. This defines the node the algorithm is currently on. Then it deletes the node that has been visited from the stack index to keep the track of remaining nodes in the queue. After this the algorithm checks whether the removed element is the goal node we were searching for. If the goal state is achieved the algorithm return the path, cost and the number of nodes visited by the algorithm. Otherwise, it just add them to the list of explored nodes. Then the algorithm just calculates the neighbours of the currently visited nodes and assigns them to the *neighbours* variable in the form of a list while also extracting the path from the *node* variable.

```
while stack:
    if len(stack) == 0:
        return None
    node = heapq.heappop(stack)
```

```
        del stack_index[node[1]]
        if node[1] == end_node:
            return node[2],node[0], nodes_explored + 1
        explored.add(node[1])
        neighbours = list(tube_graph.neighbors(node[1]))
        path = node[2]
```

Another inner loop is created which iterates over all the neighbours. The algorithm firstly adds the path of the current neighbour explored. Then it extracts the weight of the current node with help of the inbuilt function that networkx provides and assigns it to *data* variable. Then the already found cost value is added with *data* variable along with the child and the path to a new *childNode* variable. The algorithm then checks whether the childNode has already been explored or not.If not the child node is appended to the stack and the number of explored nodes count is incremented by one. If the child node is present in the stack index, we compare the cost just to ensure if the cost present is lower or not. If it is lower than already present cost then we delete that node and add the new lower cost found for that node to the stack.

```
    for child in neighbours:
        path.append(child)
        data = tube_graph.get_edge_data(node[1], child)[0]['weight']
        childNode = (node[0] + data, child, path)
        if child not in explored and child not in stack_index:
            heapq.heappush(stack, childNode)
            stack_index[child] = [childNode[0], childNode[2]]
            nodes_explored = nodes_explored + 1
        elif child in stack_index:
            if childNode[0] < stack_index[child][0]:
                delete_node = (stack_index[child][0], child, stack_index[child][1])
                stack.remove(delete_node)
                heapq.heapify(stack)
                del stack_index[child]

                heapq.heappush(stack, childNode)
                stack_index[child] = [childNode[0], childNode[2]]
        path = path[:-1]
```

1.2 Compare DFS, BFS, and USC

The algorithm that is performing most efficiently when taken in account both the relative cost and the nodes visited, is the Breadth First Search algorithm. Even though Uniform Cost Search is also finding the optimum cost it is doing so on the expense of nodes visited, sometimes UCS is taking considerably more nodes while traversing to find the goal state. While in the case of Depth First Search, the algorithm is just not performing optimally. The path found by the DFS has huge costs as well as the nodes visited by it are vast in number.

	Depth First Search		Breath First Search		Uniform Cost Search	
	Cost	Nodes Visited	Cost	Nodes Visited	Cost	Nodes Visited
Canada Water to Stratford	15	7	15	39	14	63
New Cross Gate to Stepney Green	27	33	14	25	14	28
Ealing Broadway to South Kensington	57	180	20	49	20	59
Baker Street to Wembley Park	13	4	13	15	13	102
Woodford to Whitechapel	71	78	20	30	20	40

Therefore, consistently the algorithm that is performing optimally is *Breadth First Search.*

1.2.1 Depth First Search

Advantages-

- DFS is more preferable when the goal state is far away from the source.
- Less memory is exhausted during implementation.

Disadvantages-

- It is not an optimal algorithm.
- There is no guarantee that solution will be found.

1.2.2 Breadth First Search

Advantages-

- It is guaranteed that the algorithm will find the solution if there is one.
- It is an optimal algorithm and if there are more than one solution it will find the optimal one.

Disadvantages-

- Consumes more memory when compared to others.
- Consumes more time while searching if the goal is far away from the starting node.

1.2.3 Uniform Cost Search

Advantages-

- The algorithm finds the lowest overall cost while traversing.
- It is also an optimal algorithm as from every node the next one selected is the one with lowest weight.

Disadvantages-

- It does not account for the number of steps taken during traversal because of which more memory is consumed.
- The queue in which explored nodes are stored needs to be kept in sorted manner which can become complex.

1.3 Extending the cost function

To improve the cost function what I have done is add more features to the edges of the graph. Adding factors such as tube line, main zone and secondary zone add more functionality to be considered while comparing.

```
for row in data:
tube_graph.add_node(row[0])
tube_graph.add_node(row[1])
tubeline = row[2]
weight = row[3]
main_zone = row[4]
secondary_zone = row[5]
tube_graph.add_edge(row[0], row[1], tubeline=tubeline, weight=weight,
main_zone=main_zone,secondary_zone=secondary_zone)
```

1.4 Heuristic search

The heuristic is designed in such a way that it represents the real world situation. While travelling from one station to another there should be least amount of tube line changes as much as possible. In order to guide the condition while travelling, lets say our starting position is in zone 3 and our destination position is in zone 5, we need an approach that will determine the zone difference to decrease as we go in the right direction. In this manner we can select a line that only goes from the current zone to the desired zone and perform optimally.

The algorithm starts by creating a variable admissible heuristic which assigns value of h. Then the calculate heuristic() function is called whose main task is to return the zone difference upon which our main principle is based. This function return a integer value. Algorithm then creates a new variable called visited nodes which simply stores the travel cost + heuristic cost and the path taken. Then a priority queue is created in the variable path to explore , whose task is to automatically sort the path. A loop is then created which iterates until there are no paths left to explore. The algorithm then finds the neighbours of the current node being explored. Amongst the neighbours, edge data is extracted and their weight is calculated as cost from each of the neighbour. Then new cost is calculated. The algorithm then checks whether the neighbour is already been added to the visited nodes and if not then the new cost added with the heuristic cost along with path is appended to the visited node.

```
    def A_star(tube_graph, start_node, end_node):
        admissible_heuristics = {}
        h = calculate_heuristic(start_node,end_node)
        admissible_heuristics[start_node] = h
        visited_nodes = {}
        visited_nodes[start_node] = (h, [start_node])

        paths_to_explore = PriorityQueue()
        paths_to_explore.put((h, [start_node], 0))

        while not paths_to_explore.empty():
            _, path, total_cost = paths_to_explore.get()
            current_node = path[-1]
            neighbors = tube_graph.neighbors(current_node)

        for neighbor in neighbors:
            edge_data = tube_graph.get_edge_data(path[-1], neighbor)
            if "weight" in edge_data:
                cost_to_neighbor = edge_data["weight"]
            else:
                cost_to_neighbor = 1

            if neighbor in admissible_heuristics:
                h = admissible_heuristics[neighbor]
            else:
                h = calculate_heuristic(neighbor,end_node)
                admissible_heuristics[neighbor] = h

            new_cost = total_cost + cost_to_neighbor
            new_cost_plus_h = new_cost + h
            if(neighbor not in visited_nodes)or(visited_nodes[neighbor][0]>new_cost_plus_h):
                next_node = (new_cost_plus_h, path+[neighbor], new_cost)
                visited_nodes[neighbor] = next_node
                paths_to_explore.put(next_node)


def calculate_heuristic(a,b):
    current_zone = 0
    end_zone=0
    zone_difference = 0
    for item in tube_graph:
        if(item[0]==a):
            current_zone=item[4]
            break;
    for items in tube_graph:
        if(items[0]==b):
            end_zone=items[4]
            break;
    if str(current_zone).isdigit() and str(end_zone).isdigit():
        current_zone = int(current_zone)
        end_zone = int(end_zone)
        zone_difference = abs(end_zone - current_zone)

    return zone_difference


goal = A_star(tube_graph, 'Woodford', 'Whitechapel')
print("Path found by heuristic from Woodford to Whitechapel:- ",goal[1])
print("Cost of the heuristic :- ",goal[0])
```

Output : -

Path found by heuristic from Woodford to Whitechapel:- ['Woodford', 'South Woodford', 'Snaresbrook', 'Leytonstone', 'Leyton', 'Stratford', 'Mile End', 'Stepney Green', 'Whitechapel']

Cost of the heuristic :- 8

2 Adversarial Search

Adversarial Search is technique where agents are pitted against one another and they compete while trying to defeat the other opponents. Each agent is designed to consider the actions of other agent and make decisions according to their decision making.

2.1 Play optimally (MINIMAX algorithm)

Given that we start at top and that we are playing MAX player, the last level would be MIN level.

The bottom most MIN level will take the value from its child whoever has minimum one:-

1. In the first node, amongst 3 and 7, minimum value will be selected i.e. 3.
2. In the second node, amongst 4 and 1, minimum value will be selected i.e. 1.
3. In the third node, amongst 5 and 5, since it is a tie any one can be selected.
4. In the fourth node, amongst 8 and 2, minimum value will be selected i.e. 2
5. In the fifth node, amongst 4 and 6, minimum value will be selected i.e. 4
6. In the sixth node, amongst 9 and 5, minimum value will be selected i.e. 5

7. In the seventh node, amongst 6 and 3, minimum value will be selected i.e. 3

8. In the eighth node, amongst 5 and 8, minimum value will be selected i.e. 5

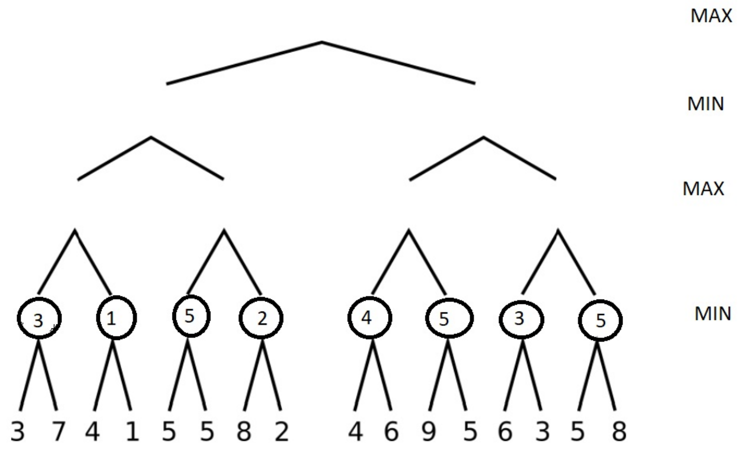

Figure 2

After the last MIN level has done its calculations, now its parents will do the calculations. Note here that the parents are performing MAX calculation and will take the max values amongst their children.

1. In the first node, amongst 3 and 1, maximum value will be selected i.e. 3.

2. In the second node, amongst 5 and 2, maximum value will be selected i.e. 5.

3. In the third node, amongst 4 and 5, maximum value will be selected i.e. 5.

4. In the fourth node, amongst 3 and 5, minimum value will be selected i.e. 5

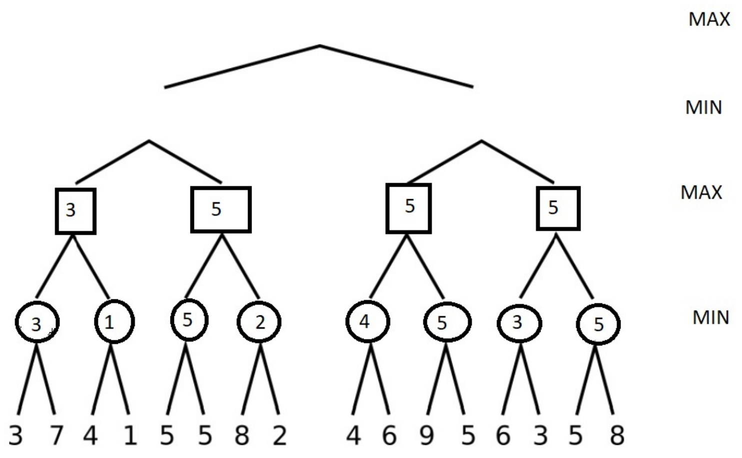

Figure 3

After the last MAX level has done its calculations, now its parents will do the calculations. Note here that the parents are performing MIN calculation and will take the min values amongst their children.

1. In the first node, amongst 3 and 5, minimum value will be selected i.e. 3.
2. In the second node, amongst 5 and 2, since it is a tie any one can be selected.

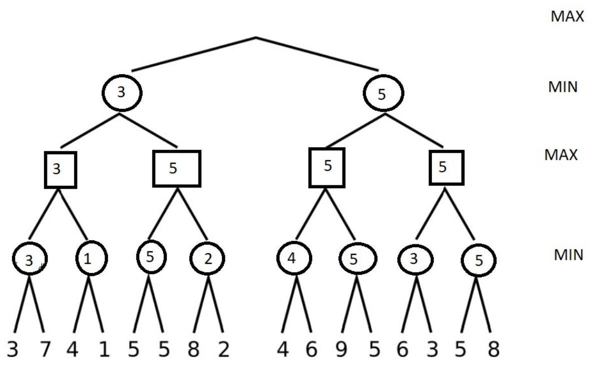

Figure 4

After the last MIN level has done its calculations, now its parents will do the calculations. Here we are at the root node and it is performing MAX calculation and will take the max values amongst their children.

1. At the root node, amongst 3 and 5, maximum value will be selected i.e. 5.

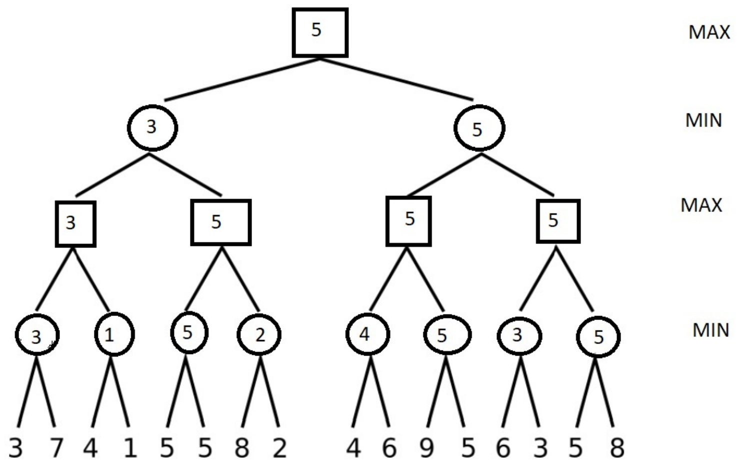

Figure 5

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

We have to perform α -β pruning on the above figure 5 starting from left to right.

There are few things to keep in mind while performing pruning:

- α contains maximum value
- β contains minimum value
- Whenever the value of α becomes greater than or equal to β we prune its next children node.
- Initialize α = - ∞
- Initialize $\beta = \infty$

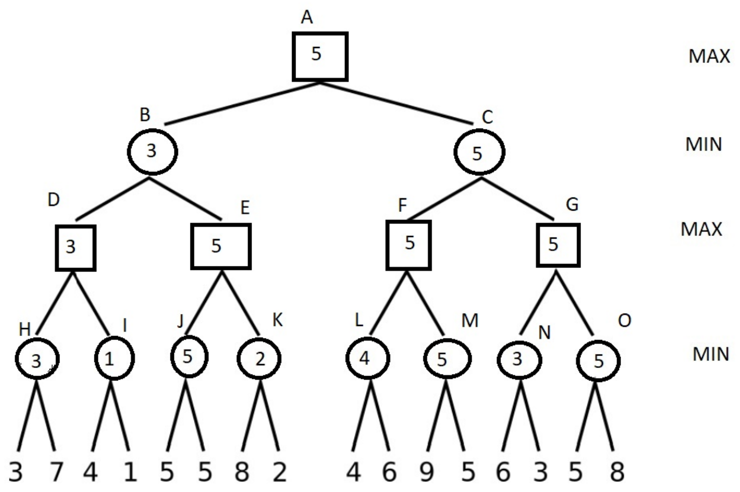

Figure 6

Each node has been assigned a name to understand the pruning better.Nodes A,B,D and H are assigned the initial α and β values i.e. -∞ and ∞ .

For node H, as it is present in MIN level, and β represents MIN it will get updated. Since its child 3 is minimum than ∞, β will be updated to 3. Now after looking at its other child 7, since it is larger than new β value 3,it will remain 3. These new α and β values will travel to node D. As node D is present in MAX level,only α will be updated. Since $\beta = 3$ at node H is greater than α = -∞ at node D, node D α will be updated to 3.

Now these values will be updated to node I i.e. $\alpha = 3$ and $\beta = \infty$. Since one of the child is 4 and smaller than $\beta = \infty$, β will get updated. From its other child which is 1 again β will be updated to one. Now note that value of $\alpha > \beta$ at node I, therefore the child with value 1,its branch will get pruned. The node D will again check its α value for evaluation. Since it is greater than or equal to values of α and β at node D, α will remain as it is i.e. α=3. Furthermore, node B will be update now. As it in MIN level it will only update β value. Since value of $\beta = \infty$ is smaller than node D α =3, node B β will be updated to 3.

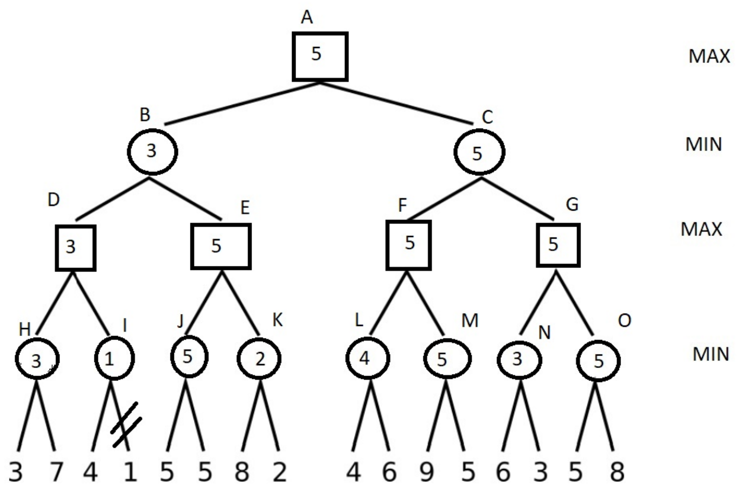

Figure 7

The values of node B will be passed down to node E and then J. In node J, value of β=3 will remain same as both of its children have larger values than 3. Node E will need to be updated. Since value of $\beta = 3$ at node J > α = - ∞ at node E, α will updated to 3. Note that now at node E values of $\alpha = \beta$ i.e. 3, the path down the node K will be pruned. Now the values of β at node B will remain same as values at node E are also equal. Node A will be updated with α value to 3 as β at node B is lower.

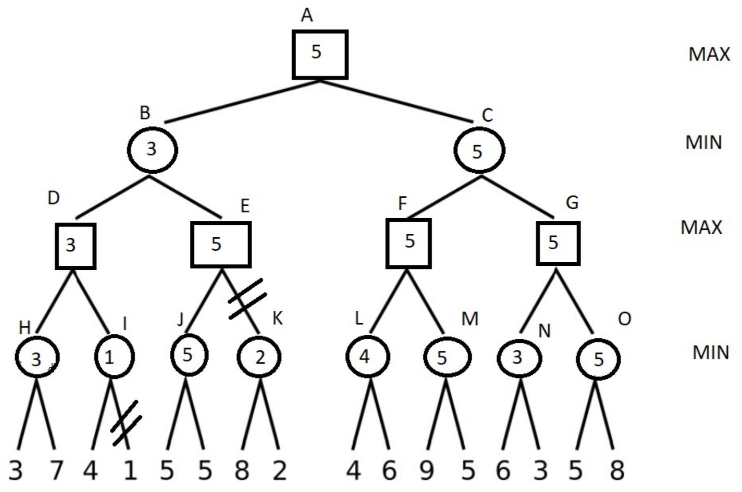

Figure 8

Node C,F, and L will be assigned with $\alpha = 3$ and $\beta = \infty$, same as node A. After traversing to node L, β will be updated to 4 as 4 is smaller than ∞. No traversing back to its parent, node F α will be updated from 3 to 4 as node F is in MAX level. Node M will be assigned $\alpha = 4$ and $\beta = \infty$. β will be updated with 9 and then with 5 as node M children are smaller than ∞. Again traversing back to the parent i.e. node F, its values will be updated. α will be now equal to 5 as it is greater than its previous value 4. Node F $\alpha = 5$ and β will remain same i.e. ∞. Node F parent node C will be updated in the next step. Since it is in MIN level only β will update and α will remain same i.e $\alpha = 5$. β will be updated to 5 as node F $\alpha = 5$. Now at node C $\alpha = \beta$ i.e. 5, and therefore its children from node G will be pruned. Node G,N,O will be pruned.

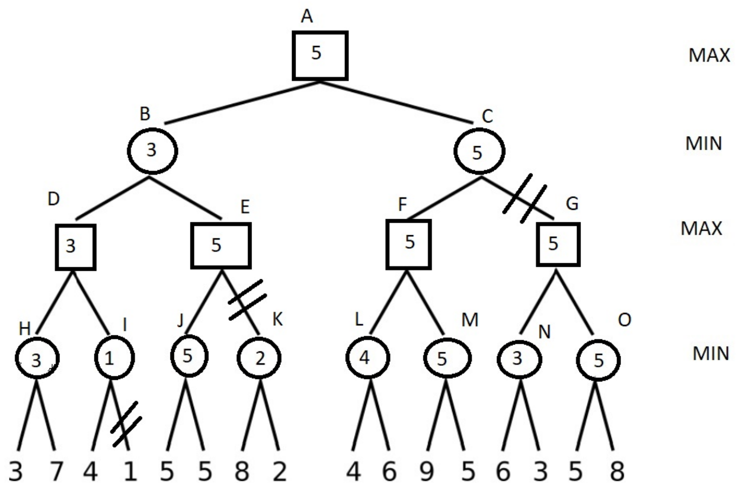

Figure 9

2.3 Entry free to play

1. The range of x should be in between 0-1. This is because the lowest unit that MAX player can win in 1 unit, and if the entry cost goes up than 1 and if the MAX player will be going into loss even if he wins the game. This is taking in the factor that if MAX player only wins with 1 unit, the player can easily pay the entry fees with that.
2. For a fixed value of x, I would prefer to be MAXimiser. If beforehand the entry cost is known all we have to do is guide the traversing in such a way that the cost units are always higher.