Artificial Intelligence Coursework 1 report

Claudiu Andrei

Queen Mary University of London, UK

1 Introduction

This report describes the methods and techniques implemented for the Artificial Intelligence module's coursework. In the first part "Agenda-based Search" search strategies and algorithms are discussed, such as BFS, DFS, UCS and heuristic approaches. The second part of this coursework explores Adversarial search algorithms, specifically MINIMAX and α-β pruning.

2 Agenda-based Search

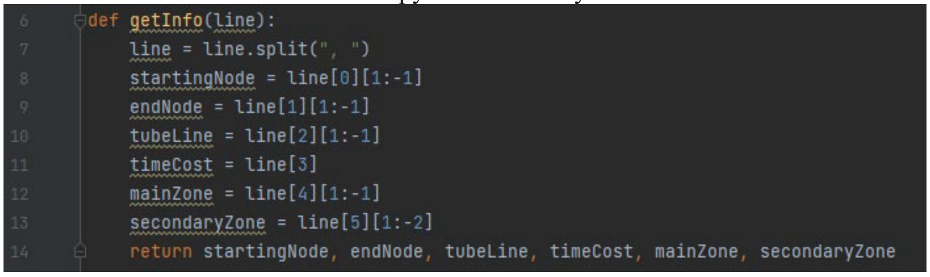
The way various route finder mechanisms are implemented on a dataset defining London Tube stations is outlined in this section. If opened with a text editor, it is possible to observe the manner in which the data is structured, depicted below:

```
"Harrow & Wealdstone", "Kenton", "Bakerloo", 3, "5", "0"
"Kenton", "South Kenton", "Bakerloo", 2, "4", "0"
…
…
"Bank/Monument", "Waterloo", "Waterloo & City", 4, "1", "0"
```

These correspond to the following details:

`[StartingStation], [EndingStation], [TubeLine], [AverageTimeTaken], [MainZone], [SecondaryZone]`

To extract data from the document a python dictionary is used



```python
def getInfo(line):
    line = line.split(", ")
    startingNode = line[0][1:-1]
    endNode = line[1][1:-1]
    tubeLine = line[2][1:-1]
    timeCost = line[3]
    mainZone = line[4][1:-1]
    secondaryZone = line[5][1:-2]
    return startingNode, endNode, tubeLine, timeCost, mainZone, secondaryZone
```

The dictionary has structure: name[(neighbor, cost, tubeLine), (neighbour2, cost2, tubeline2)].

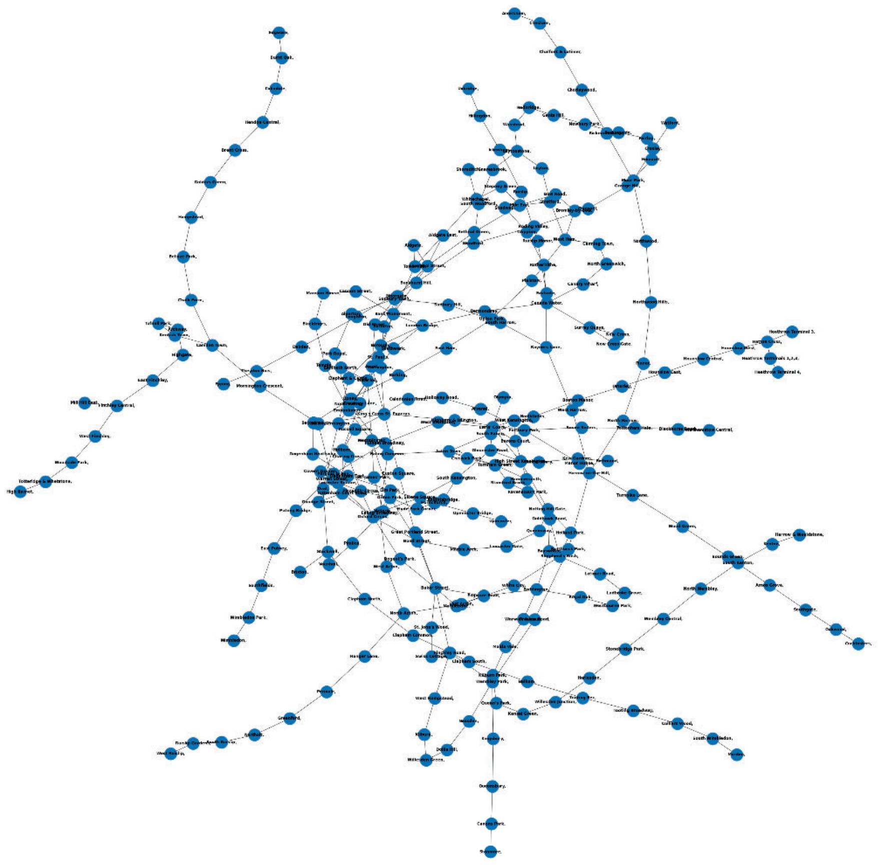
```
In [137]: import pandas as pd
          data = pd.read_csv('tubedata.txt', sep=" ", header=None)
          pd.set_option("display.max_rows", 30)
          pd.set_option("expand_frame_repr", False)
```

```
In [138]: print (data)
```

```
                        0                  1                 2   3   4  5
0    Harrow & Wealdstone,            Kenton,         Bakerloo,  3,  5,  0
1                 Kenton,      South Kenton,         Bakerloo,  2,  4,  0
2           South Kenton,     North Wembley,         Bakerloo,  2,  4,  0
3          North Wembley,   Wembley Central,         Bakerloo,  2,  4,  0
4        Wembley Central,  Stonebridge Park,         Bakerloo,  3,  4,  0
..                    ...                ...               ...  ..  .. ..
369             Victoria,           Pimlico,         Victoria,  3,  1,  0
370              Pimlico,          Vauxhall,         Victoria,  1,  1,  0
371             Vauxhall,         Stockwell,         Victoria,  3,  1,  2
372            Stockwell,           Brixton,         Victoria,  2,  2,  0
373        Bank/Monument,          Waterloo,  Waterloo & City,  4,  1,  0

[374 rows x 6 columns]
```

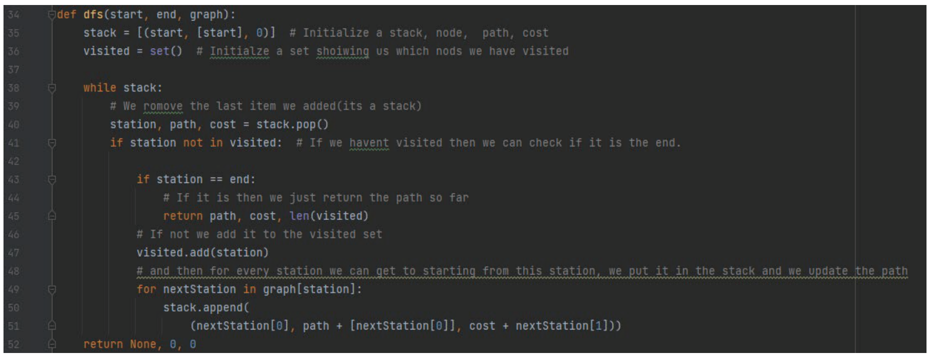
```
In [139]: G=nx.from_pandas_edgelist(data, 0, 1, edge_attr=3)
          [e for e in G.edges]
          G.number_of_nodes()

Out[139]: 271
```

```
In [140]: pos = nx.spring_layout(g, k=0.3*1/np.sqrt(len(g.nodes())), iterations=20)
          plt.figure(3, figsize=(40, 40))
          nx.draw(G, with_labels=True, node_size=1500, font_weight='bold')
          plt.savefig("data.png")
```



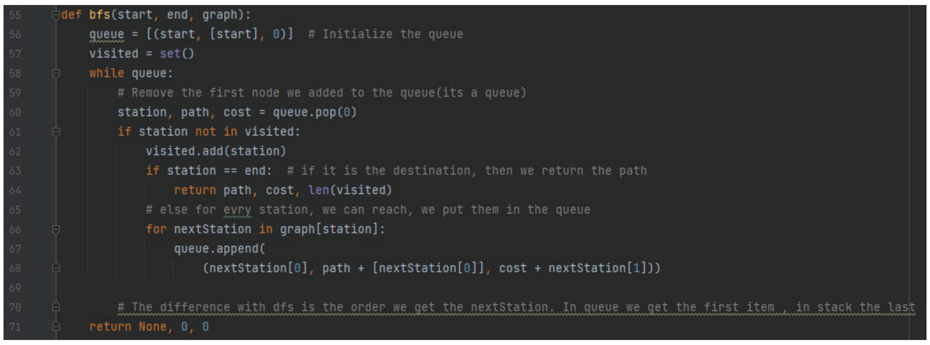
2.1 Implementation of DFS, BFS and UCS

The python code which implements the search algorithms is present in below.



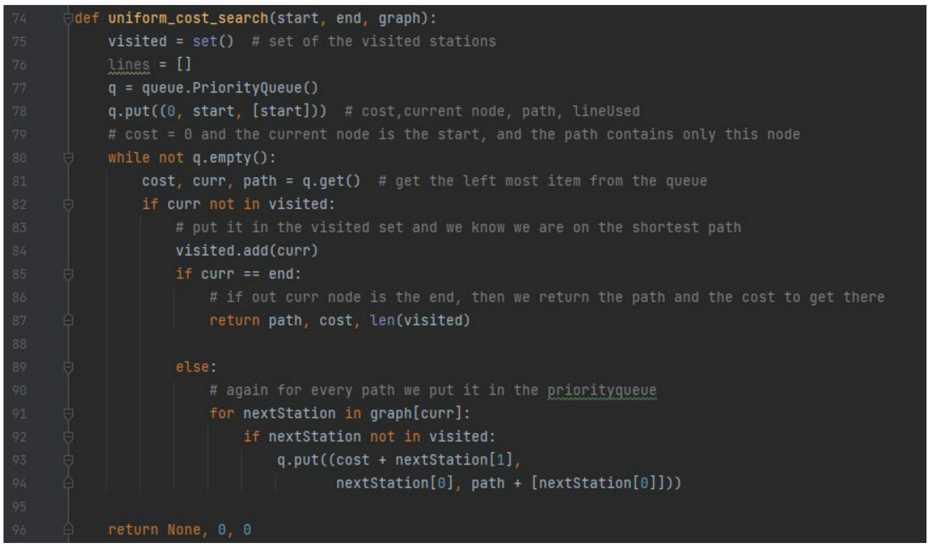
```python
def dfs(start, end, graph):
    stack = [(start, [start], 0)]  # Initialize a stack, node,  path, cost
    visited = set()  # Initialze a set shoiwing us which nods we have visited

    while stack:
        # We romove the last item we added(its a stack)
        station, path, cost = stack.pop()
        if station not in visited:  # If we havent visited then we can check if it is the end.

            if station == end:
                # If it is then we just return the path so far
                return path, cost, len(visited)
            # If not we add it to the visited set
            visited.add(station)
            # and then for every station we can get to starting from this station, we put it in the stack and we update the path
            for nextStation in graph[station]:
                stack.append(
                    (nextStation[0], path + [nextStation[0]], cost + nextStation[1]))
    return None, 0, 0
```

DFS



```python
def bfs(start, end, graph):
    queue = [(start, [start], 0)]  # Initialize the queue
    visited = set()
    while queue:
        # Remove the first node we added to the queue(its a queue)
        station, path, cost = queue.pop(0)
        if station not in visited:
            visited.add(station)
            if station == end:  # if it is the destination, then we return the path
                return path, cost, len(visited)
            # else for evry station, we can reach, we put them in the queue
            for nextStation in graph[station]:
                queue.append(
                    (nextStation[0], path + [nextStation[0]], cost + nextStation[1]))

        # The difference with dfs is the order we get the nextStation. In queue we get the first item , in stack the last
    return None, 0, 0
```

BFS

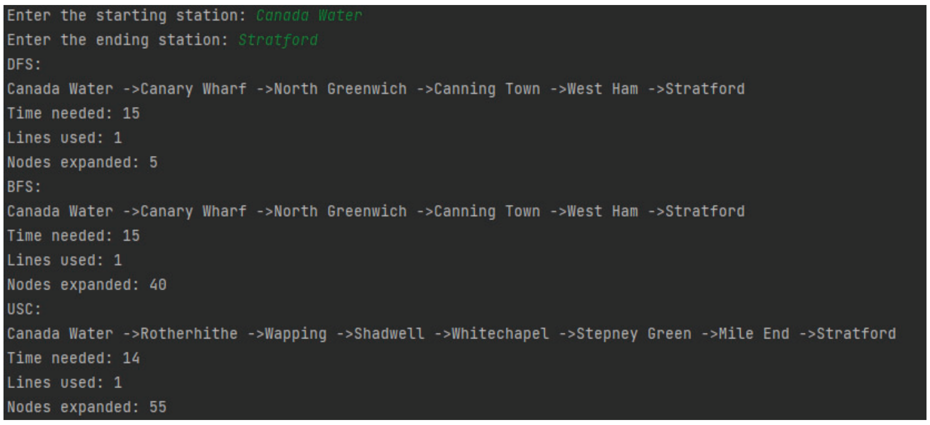


```python
def uniform_cost_search(start, end, graph):
    visited = set()  # set of the visited stations
    lines = []
    q = queue.PriorityQueue()
    q.put((0, start, [start]))  # cost,current node, path, lineUsed
    # cost = 0 and the current node is the start, and the path contains only this node
    while not q.empty():
        cost, curr, path = q.get()  # get the left most item from the queue
        if curr not in visited:
            # put it in the visited set and we know we are on the shortest path
            visited.add(curr)
            if curr == end:
                # if out curr node is the end, then we return the path and the cost to get there
                return path, cost, len(visited)

            else:
                # again for every path we put it in the priorityqueue
                for nextStation in graph[curr]:
                    if nextStation not in visited:
                        q.put((cost + nextStation[1],
                               nextStation[0], path + [nextStation[0]]))

    return None, 0, 0
```

UCS

2.2 Comparisons amongst DFS, BFS and UCS

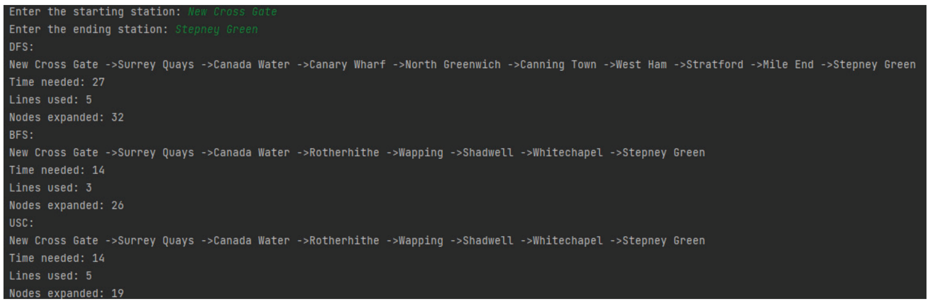
Some comparisons for the performance of each algorithm are carried out below, using the test routes specified in the assignment document.



```
Enter the starting station: Canada Water
Enter the ending station: Stratford
DFS:
Canada Water ->Canary Wharf ->North Greenwich ->Canning Town ->West Ham ->Stratford
Time needed: 15
Lines used: 1
Nodes expanded: 5
BFS:
Canada Water ->Canary Wharf ->North Greenwich ->Canning Town ->West Ham ->Stratford
Time needed: 15
Lines used: 1
Nodes expanded: 40
USC:
Canada Water ->Rotherhithe ->Wapping ->Shadwell ->Whitechapel ->Stepney Green ->Mile End ->Stratford
Time needed: 14
Lines used: 1
Nodes expanded: 55
```

Canada Water – Stratford

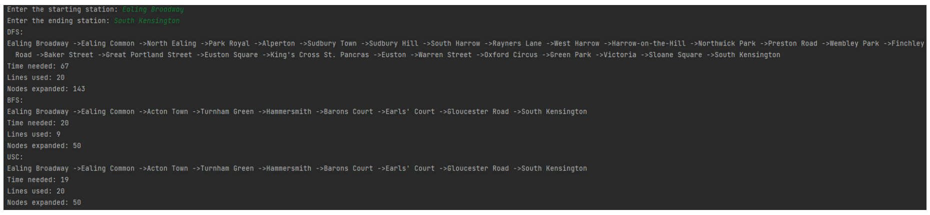
It is possible to observe that in the first example: Canada Water to Stratford, the 3 algorithms perform similarly if the amount of time needed to complete the journey is used as metric: all take 15 minutes apart from the path found by the UCS, which takes one minute less. If the metric used is the amount of time required for the computation of the path to conclude, DFS offers the best solution, as it is able to find a suitable path between the two stations by only expanding 5 nodes, thus taking the least amount of time to compute, compared to the 44 nodes of BFS and the 55 nodes expanded by UCS.



```
Enter the starting station: New Cross Gate
Enter the ending station: Stepney Green
DFS:
New Cross Gate ->Surrey Quays ->Canada Water ->Canary Wharf ->North Greenwich ->Canning Town ->West Ham ->Stratford ->Mile End ->Stepney Green
Time needed: 27
Lines used: 5
Nodes expanded: 32
BFS:
New Cross Gate ->Surrey Quays ->Canada Water ->Rotherhithe ->Wapping ->Shadwell ->Whitechapel ->Stepney Green
Time needed: 14
Lines used: 3
Nodes expanded: 26
USC:
New Cross Gate ->Surrey Quays ->Canada Water ->Rotherhithe ->Wapping ->Shadwell ->Whitechapel ->Stepney Green
Time needed: 14
Lines used: 5
Nodes expanded: 19
```

New Cross Gate – Stepney Green

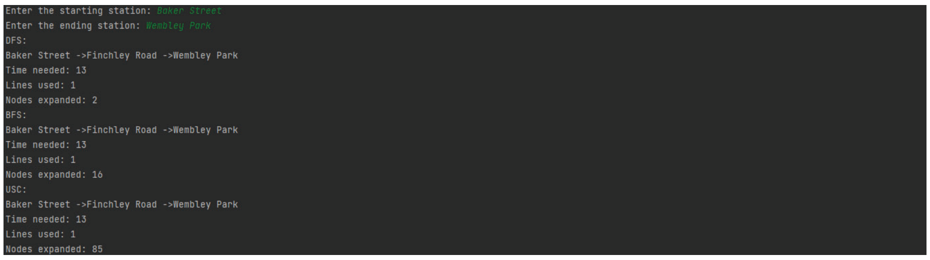
In the second example: New Cross Gate to Stepney Green, BFS and UCS perform similarly, however DFS given its nature takes longer to reach the goal station both in terms of time needed to complete the journey (27 minutes compared to the 14 taken by both BFS and UCS) and time taken to finish calculating the optimal route since 32 nodes are expanded. In this case the better performing algorithm is UCS.



```
Enter the starting station: Ealing Broadway
Enter the ending station: South Kensington
DFS:
Ealing Broadway ->Ealing Common ->North Ealing ->Park Royal ->Alperton ->Sudbury Town ->Sudbury Hill ->South Harrow ->Rayners Lane ->West Harrow ->Harrow-on-the-Hill ->Northwick Park ->Preston Road ->Wembley Park ->Finchley
  Road ->Baker Street ->Great Portland Street ->Euston Square ->King's Cross St. Pancras ->Euston ->Warren Street ->Oxford Circus ->Green Park ->Victoria ->Sloane Square ->South Kensington
Time needed: 67
Lines used: 20
Nodes expanded: 143
BFS:
Ealing Broadway ->Ealing Common ->Acton Town ->Turnham Green ->Hammersmith ->Barons Court ->Earls' Court ->Gloucester Road ->South Kensington
Time needed: 20
Lines used: 9
Nodes expanded: 50
USC:
Ealing Broadway ->Ealing Common ->Acton Town ->Turnham Green ->Hammersmith ->Barons Court ->Earls' Court ->Gloucester Road ->South Kensington
Time needed: 19
Lines used: 20
Nodes expanded: 50
```

Ealing Broadway – South Kensington

In example 3 a similar behavior is observed, as DFS has performed poorly, outputting a 67 minutes long journey and expanding 143 nodes. It has performed close to 3 times worse than BFS and UCS.



```
Enter the starting station: Baker Street
Enter the ending station: Wembley Park
DFS:
Baker Street ->Finchley Road ->Wembley Park
Time needed: 13
Lines used: 1
Nodes expanded: 2
BFS:
Baker Street ->Finchley Road ->Wembley Park
Time needed: 13
Lines used: 1
Nodes expanded: 16
USC:
Baker Street ->Finchley Road ->Wembley Park
Time needed: 13
Lines used: 1
Nodes expanded: 85
```

Baker Street – Wembley Park

In the final example the results are in favor of DFS, as this time it has been the better performer by finding the shortest path in terms of journey completion time and also expanding the least nodes. The nature of DFS can be observed here: given its approach to quickly dive downwards in the tree, it is able to reach the end node quickly if it follows the correct directions and the 2 stations are on the same line not very far apart.

DFS
When elaborating on the results of the experiments it is possible to come to conclusions about the advantages and disadvantages of each method. DFS has the advantage of a linear memory requirement, if with respect to the nodes. It also benefits from less time and space complexity than BFS whilst finding a solution without much more search.
DFS however is not guaranteed to output an optimal solution and can expand too many nodes and still give a non-optimal result. Furthermore, in DFS, cut-off depth is smaller so time complexity is more.

BFS has some advantages over DFS, this is because if a solution to the problem is present or exists, it will be found by BFS. This algorithm also has the advantage of not getting trapped in blind alleys and in case of multiple solutions it will find the one with minimal steps and it guarantees the lowest number of stations.
The limits of DFS come in the form of memory constrains as it stores all the nodes on the current level to reach the next, therefore, consuming more time if the solution is further away.

UCS
UCS is the algorithm that will return the optimal solution to find the path from root node to destination node with the lowest cumulative cost. UCS is considered to have an optimal solution because it chooses the path with the least cost at each state. The open list in the UCS needs to be kept sorted as priorities and a priority queue needs to be kept. Its cost in terms of storage requirements is exponentially large. UCS also risks getting stuck in loops while considering every possible path going from the root node to the destination.

2.3 Extending the cost function

In order to try and optimize the UCS algorithm further its cost function can be expanded and modified, therefore the approach explored was to modify the cost function using a version which bases the cost function on the amount of line changes performed to reach the destination, as it could be argued that line changes take time therefore if those could be reduced, the effectiveness of the algorithm could result different. This approach however is not implemented as issues were encountered with the development of a method able of detecting line changes in an accurate manner, resulting often unexpected results.

2.4 Heuristic search

```
def a_star_algorithm(graph, start, stop):
    not_visited = set([start])
    visited = set([])

    # poo has present distances from start to all other
nodes
    dist = {}
    dist[start] = 0

    # par contains an adjac mapping of all nodes
    par = {}
    par[start] = start


    while len(not_visited) > 0:
        n = None

        # it will find a node with the lowest value of
f() -
        for v in not_visited:
            for neighbor in graph[v]:
                if n == None or dist[v] + h(graph, v,
neighbor[2]) < dist[n] + h(graph, v, neighbor[2]):
```

```
                    n = v


        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop
        # then we start again from start
        if n == stop:
            reconst_path = []
while par[n] != n:
            reconst_path.append(n)
            n = par[n]

        reconst_path.append(start)

        reconst_path.reverse()

        return reconst_path

    # for all the neighbors of the current node do
    for neighbor in graph[n]:
        # if the current node is not presentin both
open_lst and closed_lst
        # add it to open_lst and note n as it's par
        if neighbor[0] not in not_visited and neighbor[0]
not in visited:
            not_visited.add(neighbor[0])
            par[neighbor[0]] = n
            dist[neighbor[0]] = dist[n] + neighbor[1]


        # otherwise, check if it's quicker to first visit
n, then m
        # and if it is, update par data and poo data
        # and if the node was in the visited, move it to
not visited
        else:
            if dist[neighbor[0]] > dist[n] + neighbor[1]:
                dist[neighbor[0]] = dist[n] + neighbor[1]
                par[neighbor[0]] = n

                if neighbor[0] in visited:
                    visited.remove(neighbor[0])
                    not_visited.add(neighbor[0])

    # remove n from the not visited, and add it to
```

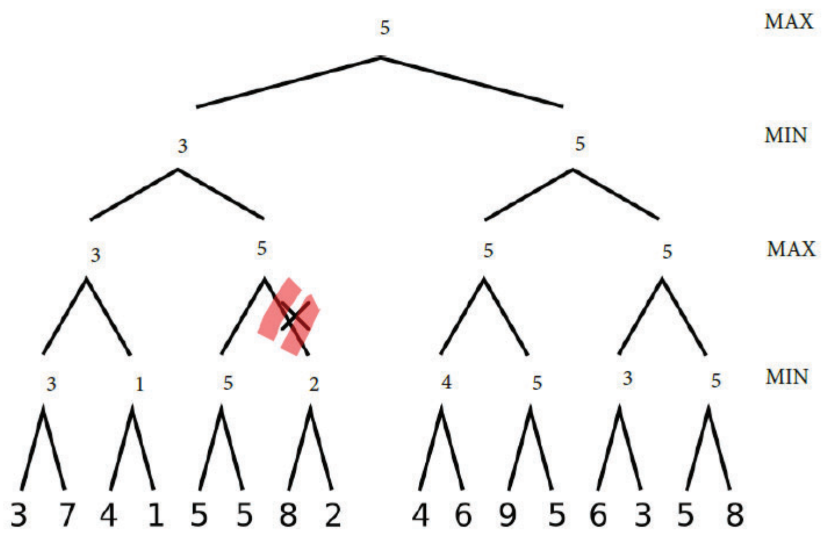
```
visited
    # because all of his neighbors were inspected
    not_visited.remove(n)
    visited.add(n)


print('Path does not exist!')
return None
```

The heuristic approach implemented is displayed in the images above, depicting the code implementing the function a_star_algorithm. This uses a heuristic function in the A* algorithms which gets the distance to the next path and the shortest length of the path from start to end.

3 Adversarial search

This section of the report explores the use of the MINIMAX algorithm and adversarial search.

3.1 Play optimally (MINIMAX algorithm)



The completed tree is found above, the pruned branch is also highlighted. An observation can be made on the pruned branch: as the branch offers a worst minimax value than ones already available. In the position of the node where the MAX player choses between 5 and 2 it can be observed that the MIN player had a better option available prior therefore the branch can be pruned.

3.3 Entry fee to play

Following the MINIMAX algorithm execution, a rule is introduced imposing a fee "$x > 0$" to enter the game, this represents the units which the MAX player pays the MIN player to be allowed to take part in the game.
The values of x for which the MAX player would want to enter the game is chosen to be $0 < x < 5$. This guarantees the MAX player doesn't lose money since the root node at the top of the tree has minimax value "5".
In case of a fixed value of x, if this is unknown before the start of the game there would be no way of choosing whether to play as the MIN or MAX player as there are possibilities of x being smaller than 5 or greater. However for a fixed value of 3 it would be optimal to chose to play as the MAX player while for an x value of 7 it would be profitable to play as the MIN player.