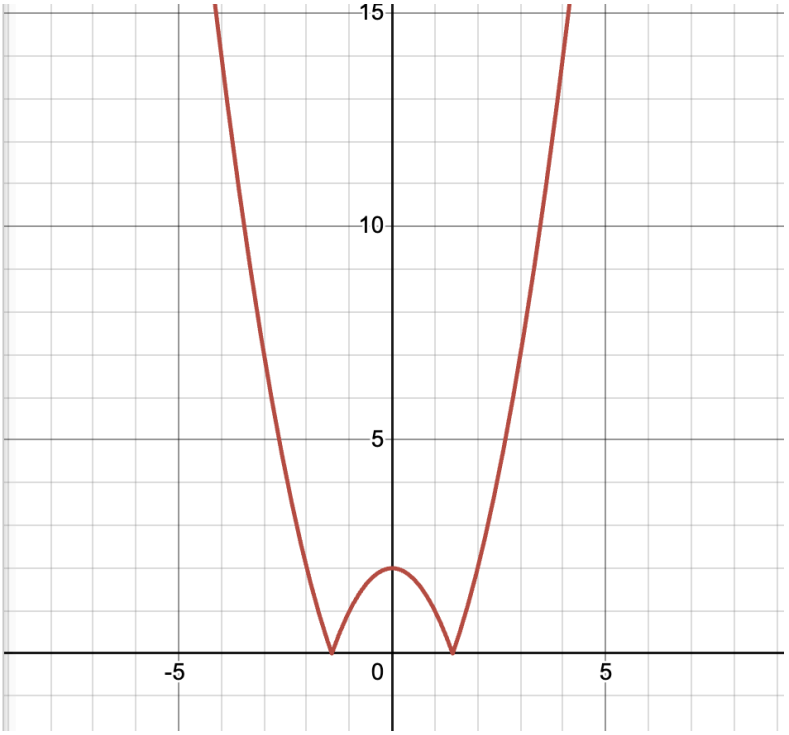
Artificial Intelligence
Coursework 1

1 Square root Agent

1.1 Theory Questions

The cost function $J = \left|y_n^2 - x\right|$, the plot depends on x which impacts a vertical shift in the plot, this plot has been drawn with $x = 2$



$$y_{n+1} = y_n - \frac{J}{\frac{\partial J}{\partial y_n}}$$

$$y_{n+1} = y_n - \frac{\left|y_n^2 - x\right|}{\frac{\partial(\left|y_n^2 - x\right|)}{\partial y_n}}$$

$$y_{n+1} = y_n - \frac{\left|y_n^2 - x\right|}{sign(y_n^2 - x) * 2y_n} \quad \rightarrow y_{n+1} = y_n - \frac{y_n^2 - x}{2y_n}$$

Pseudocode:

function cost(x):

y ← 5 //*initial guess*

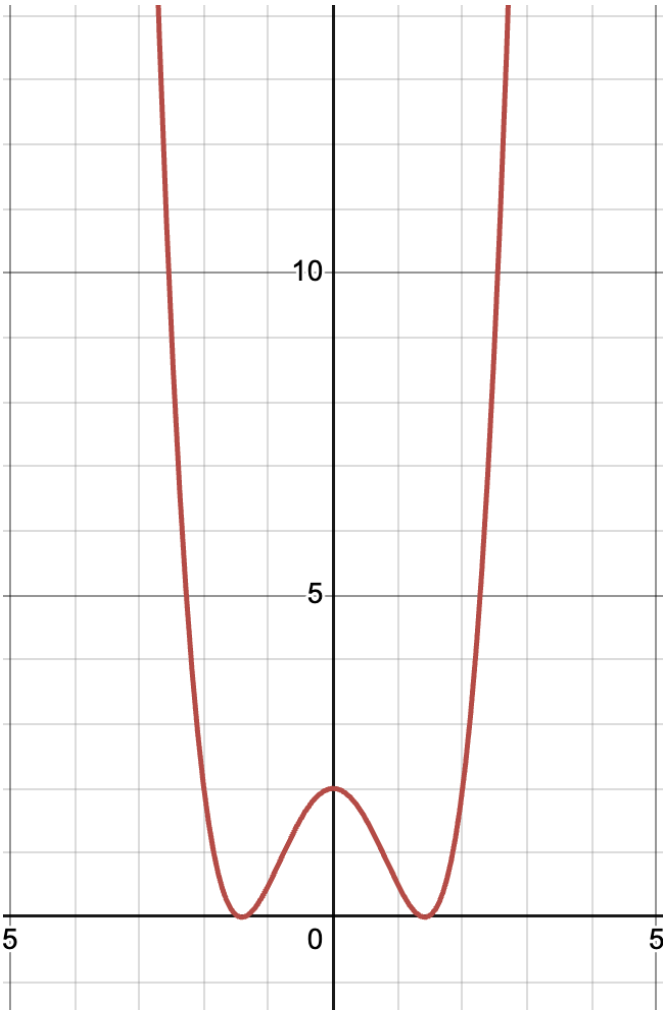
repeat until $|y^2 - x| < 10^{-6}$

$y \leftarrow y - \frac{y^2 - x}{2y}$

end

return y

The cost function $J = \frac{1}{2}(y_n^2 - x)^2$, the plot depends on x which impacts a vertical shift in the plot, this plot has been drawn with $x = 2$



$$y_{n+1} = y_n - \frac{J}{\frac{\partial J}{\partial y_n}}$$

$$y_{n+1} = y_n - \frac{(y_n^2 - x)^2}{\frac{\partial((y_n^2 - x)^2)}{\partial y_n}}$$

$$y_{n+1} = y_n - \frac{(y_n^2 - x)^2}{\frac{\partial(y_n^4 - 2y_n^2 x + x^2)}{\partial y_n}}$$

$$y_{n+1} = y_n - \frac{(y_n^2 - x)^2}{4y_n^3 - 4xy_n}$$

$$y_{n+1} = y_n - \frac{(y_n^2 - x)^2}{4y_n(y_n^2 - x)} \quad (simplifying\ the\ fraction\ by\ (y_n^2 - x))$$

$$y_{n+1} = y_n - \frac{y_n^2 - x}{4y_n}$$

Pseudocode:
function cost(x):

y ← 5 //*initial guess*

repeat until $(y^2 - x)^2 < 10^{-6}$

$y \leftarrow y - \frac{y^2 - x}{4y}$

end

return y

2. Path finding algorithm

2.2 Comparing search algorithms:

- Is any path finding method consistently better?

  Heuristic BFS has the best results with the lowest number of nodes and shortest paths in most cases which makes sense because it's the only informed search algorithm we implemented. In general, the main difference between the search algorithms can be summed up in the different criterions for expanding and choosing the next node. A comparison will be made based on the methods for expanding and choosing the next node and the reason why heuristic BFS has shown better results will be clearer:

  Heuristic BFS: when expanding a node, the cost that gets appended to the priority list, is a result we get from the heuristic, so it is a calculation made from the neighbor to the target node instead of a calculation based on the path from start node to neighbor:

```
priority_list.append((zone_heuristic(zone_dict[neighbor],zone_dict[end
_station]),current_path+[neighbor],current_cost+travel_cost))
```

  This additional information can be thought of as a function that gives us information about what we will face, in case we decide to expand that specific neighbor first and will prevent nodes with high costs of travel to the target node (or no route) to get expanded as the priority queue is sorted based on the heuristic value. In this specific case the heuristic is

defined as the minimum difference between the zones of the stations. A heuristic must be admissible therefore it is important that it does not overestimate the cost of getting from a node to the target node. In this case we need to rely on the travel made between different zones. Since all stations were in one or two zones and since based on the distribution of zones, the values in zone_dict have a format of {i,i+1} or {i}, the heuristic is calculated as the minimum difference between zone values of a starting point and zone values of an ending point. So even if we have a station with zones {a,...,k} and a station with zones {l,...,z}, we would still estimate |l-k| as the heuristic cost of getting from the first station to the second one.

UCS with cost for changing lines: This method has the second best output in most cases compared to heuristic BFS and given the fact that it only relies on current cost + path cost it is understandable that it won't be as efficient as an informed search algorithm since it does not choose based on the path from neighbor to destination but based on the path from start to neighbor. But including the cost for changing lines doesn't allow neighbors far from the current node (irrelevant subtrees) to be explored.

```
                if line == tube_line.get(current_station):
                    priority_queue.append((current_cost + path_cost,
neighbor, current_path + [neighbor]))
                else:
                    priority_queue.append((current_cost + path_cost+10,
neighbor, current_path + [neighbor]))
                tube_line[neighbor]=line
```

So a neighbor from a different tube line will have to be popped from the priority queue if the others have a very significant cost and therefore that neighbor will only get chosen if it is cheaper to go there by 2 to 5 nodes. (given that costs are more than 1 and less than 5, a cost of 10 for changing the line would be equivalent to traversing 2 to 5 nodes). This is the reason why UCS without the cost for changing lines traverses more nodes if the weights are not expanding critically while trying to find the target and UCS with the additional cost does not. However depending on the situation, if there is a path change with a lower cost route, it would not be considered as an option if the difference it makes isn't bigger than 10. Therefore the connectivity of nodes in between the path, their weights and the number of changes needed determines that UCS or UCS with an additional cost would be better.

UCS: The result has been compared to the UCS with the train line change, but if we were to compare it to BFS and DFS, it does usually show better results than the BFS because it uses a priority queue to choose a node for expansion but the results cannot be compared with

DFS as DFS isn't a complete search method but it will perform better if we have a shallow result (result in a limited depth in a large graph) or a high branching factor like the tube problem where the graph is very connected.

```
            if neighbor not in visited:

    # STEP 3.5.2: Calculate the new cumulative cost and add the new
path to the priority queue.
    # WHY DO WE NEED IT? To explore new paths from the current station
and prioritize them by cost.
                priority_queue.append((current_cost + path_cost,
neighbor, current_path + [neighbor]))
```

DFS: As mentioned, the result we get with df cannot be fully compared with the BFS and UCS results as it depends on the positioning of the start node and the end node but given that it does not use a heuristic and goes with an in depth search, it is evident that BFS with a heuristic and the UCS with cost that prohibits the exploration of irrelevant nodes, would have better performance than DFS.
BFS: Given that we are facing a problem that has weights, it performs poorly compared to UCS but again, its performance compared to dfs would rely on the positioning of nodes. We can clearly conclude that UCS with costs and a heuristic with bfs would have better performance compared to regular bfs which uses a dequeue and expands all nodes in a breath to reach to goal.

- Report the count of visited nodes by each algorithm for the most interesting routes you tested and explain your result based on the knowledge of how each algorithm works.
    - Example 1: Testing routes from Charing Cross to Holborn:
      This path particularly interested me because it passes nodes with high branching factors so many paths can be taken and I mainly relied on it to test and compare my results with different algorithms.

      Method: BFS
      Path: ['Charing Cross', 'Leicester Square', 'Covent Garden', 'Holborn']
      N Visited Nodes: 33
      Total Travel Cost: 4

      Since BFS explores all nodes at the same level, it does find an optimal solution with the travel cost of 4 however the number of nodes explored are 33 which is a lot more compared to UCS that takes edge weights into consideration before exploring a node.

      Method: DFS

Path: ['Charing Cross', 'Embankment', 'Temple', 'Blackfriars', 'Mansion House', 'Cannon Street', 'Bank/Monument', 'Liverpool Street', 'Aldgate', 'Tower Hill', 'Aldgate East', 'Whitechapel', 'Shadwell', 'Wapping', 'Rotherhithe', 'Canada Water', 'Bermondsey', 'London Bridge', 'Borough', 'Elephant & Castle', 'Kennington', 'Oval', 'Stockwell', 'Vauxhall', 'Pimlico', 'Victoria', 'Green Park', 'Bond Street', 'Baker Street', 'Great Portland Street', 'Euston Square', "King's Cross St. Pancras", 'Euston', 'Warren Street', 'Goodge Street', 'Tottenham Court Road', 'Holborn']
N Visited Nodes: 212
Total Travel Cost: 68

DFS does not prioritise shortest path or cost; it explores each path full in depth before backtracking, which is the reason why the number of nodes visited is so high, and it also does not guarantee an optimal path which is the reason why it takes a path with the travel cost of 68 which is much more costly compared to UCS and BFS.

Method: UCS
Path: ['Charing Cross', 'Leicester Square', 'Covent Garden', 'Holborn']
N Visited Nodes: 14
Total Travel Cost: 4

UCS finds the cheapest path by expanding nodes based on travel cost. In this case, UCS has the fewest visited nodes which is 14 and finds the optimal path that BFS found with minimal exploration.

Method: UCS_COST
Path: ['Charing Cross', 'Leicester Square', 'Covent Garden', 'Holborn']
N Visited Nodes: 32
Total Travel Cost: 24

UCS_COST adds a cost for line changes, and in this case, more nodes need to be explored for this reason compared to the standard UCS. The reason for this is that it needs to recalculate the added line-change cost for the nodes that come with this option and make a comparison. This approach makes more sense in an actual scene when a person would want to use the tube but it has more exploration compared to a no cost UCS. the 20 added to the optimal 4 indicates 2 station changes but we can see that even with the change, the path remained the same and this was truly the best path even other algorithms could find.

Method: Heuristic BFS
Path: ['Charing Cross', 'Leicester Square', 'Covent Garden', 'Holborn']
N Visited Nodes: 17
Total Travel Cost: 4

Heuristic BFS uses a heuristic based on the zone of each station and the oath it has to take to reach the goal to guide the search, therefore it improves node efficiency over BFS by not exploring all nodes and only exploring the

ones with promising paths to the destination. This algorithm finds the optimal path with a cost of 4 while reducing the number of visited nodes compared to standard BFS.

- Example 2: Testing routes from Holborn to Charing Cross

  This path is a return trip from the previous example and I was curious to know how the number of visits would change even in cases that have already found the optimal path.

  Method: BFS
  Path: ['Holborn', 'Covent Garden', 'Leicester Square', 'Charing Cross']
  N Visited Nodes: 29
  Total Travel Cost: 4

  BFS finds the shortest path again but again requires exploring a larger number of nodes compared to other algorithms. However it has traversed slightly less when the start and end nodes changed which is probably due to the decisions made on the node to be explored by the final state which is different.

  Method: DFS
  Path: ['Holborn', 'Chancery Lane', "St. Paul's", 'Bank/Monument', 'Cannon Street', 'Mansion House', 'Blackfriars', 'Temple', 'Embankment', 'Charing Cross']
  N Visited Nodes: 10
  Total Travel Cost: 13

  DFS explores a longer path than BFS or UCS, but only with changing the direction of our trip, it visits fewer nodes compared to BFS due to its depth-first nature. Surely the end goal in a search algorithm is finding the least travel cost but this was a nice illustration of how dfs's performance can vary based on the depth of the result.

  Method: UCS
  Path: ['Holborn', 'Covent Garden', 'Leicester Square', 'Charing Cross']
  N Visited Nodes: 10
  Total Travel Cost: 4

  UCS finds the cheapest path for the return with even less node exploration, demonstrating that it consistently finds the optimal path for weighted graphs without excess exploration. The priority queue manages the nodes to be explored very well.

  Method: UCS_COST
  Path: ['Holborn', 'Covent Garden', 'Leicester Square', 'Charing Cross']
  N Visited Nodes: 9
  Total Travel Cost: 14

  UCS_COST achieves the optimal number of visited nodes but with a more costly path. The added cost is because of a line change that also appeared in

the previous algorithms but didn't count as a cost so we can conclude that it is the same path with even less exploration, making it the optimal algorithm for this case.

Method: Heuristic BFS
Path: ['Holborn', 'Covent Garden', 'Leicester Square', 'Charing Cross']
N Visited Nodes: 11
Total Travel Cost: 4

Heuristic BFS finds the optimal path and less nodes were explored for the return path compared to the previous example. In general total number of nodes explored are quite similar to UCS_COST which was the optimal solution.

- Example 3: Testing routes from Mile End to Whitechapel:
This route was particularly interesting to me as I wanted to inspect the number of nodes explored with BFS in a situation where the answer is quite straightforward and first handedly experiment the effect of exploring all nodes.

Method: BFS
Path: ['Mile End', 'Stepney Green', 'Whitechapel']
N Visited Nodes: 14
Total Travel Cost: 5

BFS explores a lot of nodes given the fact that only the destination is only one node away. However it does find the optimal path which is because it is a complete algorithm.

Method: DFS
Path: ['Mile End', 'Bethnal Green', 'Liverpool Street', 'Aldgate', 'Tower Hill', 'Aldgate East', 'Whitechapel']
N Visited Nodes: 7
Total Travel Cost: 15

Given the fact that DFS explores a depth completely, it is showing good results for a shallow node and finding the target node in optimal number of steps but not the optimal travel cost.

Method: UCS
Path: ['Mile End', 'Stepney Green', 'Whitechapel']
N Visited Nodes: 7
Total Travel Cost: 5

UCS finds the optimal path in optimal number of node visits and this is mainly because the one node that needs to be traversed has the lowest cost and gets chosen instantly.
Method: UCS_COST
Path: ['Mile End', 'Stepney Green', 'Whitechapel']
N Visited Nodes: 7
Total Travel Cost: 15

Similar to the normal UCS, just adds a 10 to the cost because of the line change that didn't get accounted in other methods.

Method: Heuristic BFS
Path: ['Mile End', 'Stepney Green', 'Whitechapel']
N Visited Nodes: 7
Total Travel Cost: 5

Since the heuristic does choose based on the zone cost between mile end and stepney green and the stepney green and whitechapel, it was expected that heuristic bfs would also be suitable for this example and find the optimal path in optimal number of node visits.

- Example 4: Testing routes from Oxford Circus to Charing Cross:
This case was interesting to test because it passes nodes with a high branching factor and connectivity so different neighbor exploration techniques will have different results.

Method: BFS
Path: ['Oxford Circus', 'Piccadilly Circus', 'Charing Cross']
N Visited Nodes: 29
Total Travel Cost: 4

Exploring a node with a high branching factor did lead to more node visited but the optimal path was found.

Method: DFS
Path: ['Oxford Circus', 'Bond Street', 'Baker Street', 'Edgware Road', 'Paddington', 'Bayswater', 'Notting Hill Gate', 'High Street Kensington', "Earls' Court", 'Gloucester Road', 'South Kensington', 'Knightsbridge', 'Hyde Park Corner', 'Green Park', 'Piccadilly Circus', 'Charing Cross']
N Visited Nodes: 101
Total Travel Cost: 32

Going in depth while searching for the target led to an intense amount of node visits and the optimal travel cost wasn;t even found, showing us that the algorithm was looking for the target in wrong subtrees.

Method: UCS
Path: ['Oxford Circus', 'Piccadilly Circus', 'Charing Cross']
N Visited Nodes: 14
Total Travel Cost: 4

Given the fact that choices are made based on edge weights, optimal path is found through search in a number of nodes that were close to optimal.

Method: UCS_COST
Path: ['Oxford Circus', 'Piccadilly Circus', 'Leicester Square', 'Charing Cross']
N Visited Nodes: 17
Total Travel Cost: 16

The added cost makes the algorithms take a completely different path which is slightly higher in cost and goes through more nodes but technically, involves less line change.

Method: Heuristic BFS
Path: ['Oxford Circus', 'Piccadilly Circus', 'Charing Cross']
N Visited Nodes: 12
Total Travel Cost: 4

This method finds the path with the least number of node visits and least cost showing that heuristic was a good estimation of the travel price in this specific example.

- Explain how you overcame all the issues you had while implementing all path finding algorithms, such as infinite loops, etc.
  - I had a problem in the UCS route finder where I was trying to implement this part using a break statement:

```
    # STEP 3.3: Check if the current station has already been visited.
    # WHY DO WE NEED IT? To skip processing if the station has already been visited.
        if current_station in visited:
            #problem 2
            continue
```

    The code would technically jump out of the loop when it encountered a visited node so the results were wrong. Changing it to a continue statement where the loop continued iterations after a visited node fixed my problem.
  - I had a problem in the UCS with cost algorithms where I was trying to store the line (to have a sense of the current lin) in a list but initialising it was being problematic since it is not possible to pop from n empty list and I didn't want to set the initial line as the line corresponding to the starting node since that would be problematic for stations that have access to many lines. Later I figured out how to use a dictionary and this solved the issue because I used station as the key and the line as the value.
  - I had a problem with the zone heuristic function because the dictionary values are sets and I initially thought i had to unpack them one by one and write different cases fo when stations have different length of zones (1 or 2) but I tested using a for loop on a set and saw it worked so it all summed up in two for loops instead of multiple if statements.
  - I had a problem trying to iterate through the values of neighbors, cost and tube line since they are dictionaries of sets. I provided an example

to chatgpt and asked how I could iterate it and it suggested the dictionary.get() method and this fixed my problem

- Report the longest route(s) possible without repeated nodes in the state space of this tube map. Describe how you found such routes.

  DFS returns the longest routes without revisiting nodes because a set for visited nodes was kept and as it can be seen in all examples it returns the longest routes because it explores each path in full depth before backtracking, this will help evaluate all possible paths from any starting station.
  I wrote a code to calculate all paths from all stations to all stations with dfs

```
def find_longest_path(station_dict):
   longest_path = []
   max_cost = 0

   # Running DFS from eall stations to all stations in order to store the
longest path
   for start_station in station_dict.keys():
       for end_station in station_dict.keys():
           if start_station != end_station:
               path, _, total_cost = dfs_route_finder(start_station,
end_station, station_dict)
               if len(path) > len(longest_path) or (len(path) ==
len(longest_path) and total_cost > max_cost):
                   longest_path = path
                   max_cost = total_cost
   return longest_path, max_cost

longest_path, max_cost = find_longest_path(station_dict)
print(f"Longest path: {longest_path}")
print(f"Total travel cost: {max_cost}")
```

  This was the result:

```
Longest path: ['Harrow & Wealdstone', 'Kenton', 'South Kenton', 'North
Wembley', 'Wembley Central', 'Stonebridge Park', 'Harlesden', 'Willesden
Junction', 'Kensal Green', "Queen's Park", 'Kilburn Park', 'Maida Vale',
'Warwick Avenue', 'Paddington', 'Bayswater', 'Notting Hill Gate', 'High
Street Kensington', "Earls' Court", 'Barons Court', 'Hammersmith', 'Goldhawk
Road', "Shepherd's Bush", 'White City', 'East Acton', 'North Acton', 'West
Acton', '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', 'Bond Street', 'Green
Park', 'Hyde Park Corner', 'Knightsbridge', 'South Kensington', 'Sloane
Square', 'Victoria', 'Pimlico', 'Vauxhall', 'Stockwell', 'Oval',
'Kennington', 'Elephant & Castle', 'Borough', 'London Bridge',
'Bank/Monument', 'Cannon Street', 'Mansion House', 'Blackfriars', 'Temple',
'Embankment', 'Charing Cross', 'Leicester Square', 'Covent Garden',
'Holborn', 'Russell Square', "King's Cross St. Pancras", 'Angel', 'Old
Street', 'Moorgate', 'Liverpool Street', 'Aldgate', 'Tower Hill', 'Aldgate
```

```
East', 'Whitechapel', 'Shadwell', 'Wapping', 'Rotherhithe', 'Canada Water',
'Canary Wharf', 'North Greenwich', 'Canning Town', 'West Ham',
'Bromley-by-Bow', 'Bow Road', 'Mile End', 'Stratford', 'Leyton',
'Leytonstone', 'Snaresbrook', 'South Woodford', 'Woodford', 'Roding Valley',
'Chigwell', 'Grange Hill', 'Hainault', 'Fairlop', 'Barkingside', 'Newbury
Park', 'Gants Hill', 'Redbridge', 'Wanstead']
Total travel cost: 233
```