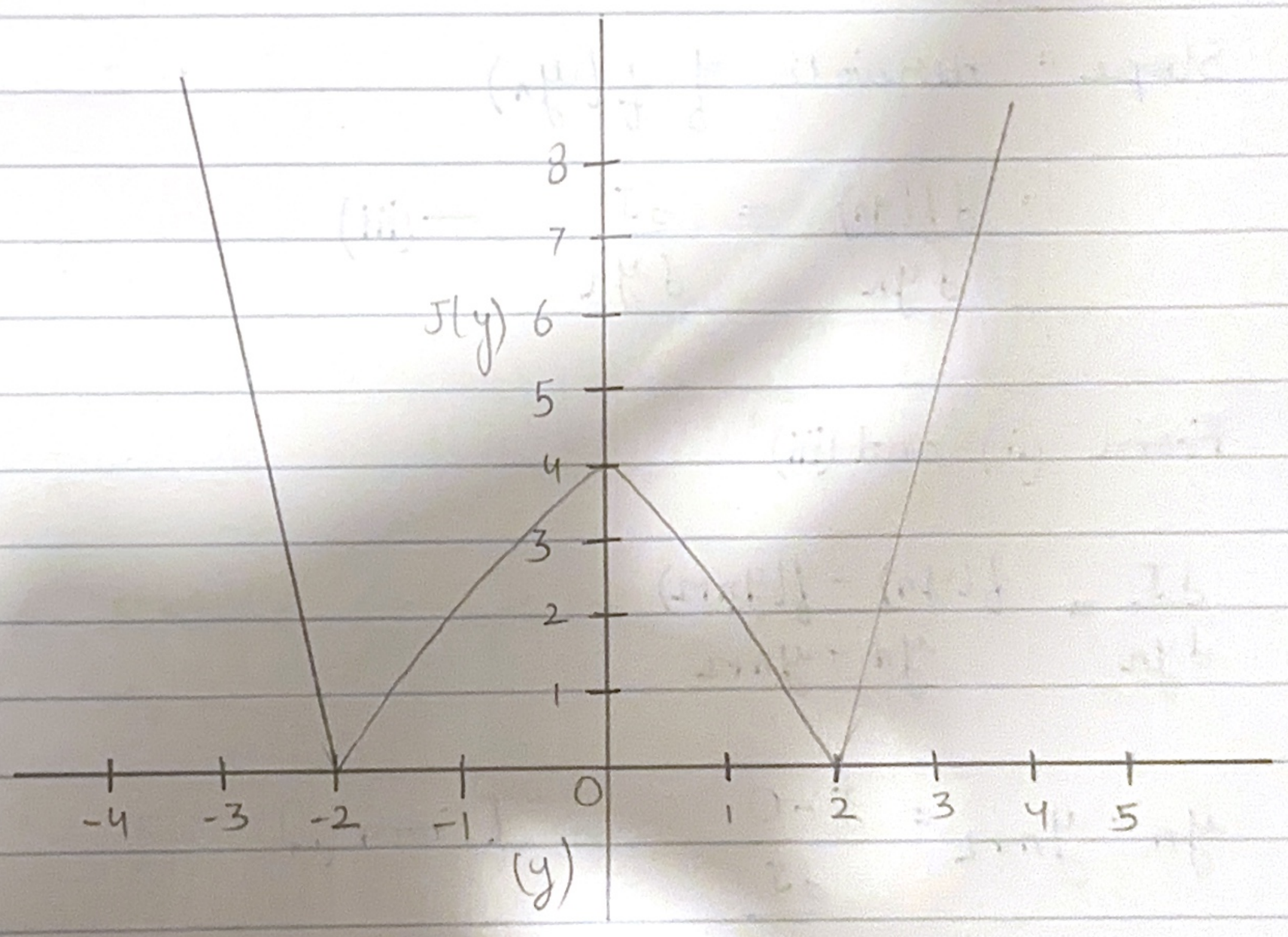
Q(1) Drow the cost function $J = |y_n^2 - x|$

The plot of the cost function $J = |y_n^2 - x|$ where $x = 4$
The cost function becomes significantly small at $y = 2, -2$, optimal y values.

In the graph given below, the function has 2 minima, One positive and one negative.
Both $y = 2$ and $y = -2$ satisfy $y^2 = x$ where $x = 4$




$J = |y_n^2 - x|$ at $x = 4$

[ $y = 2$
$J = |(2)^2 - 4| = 0$

[ $y = -2$
$J = |(-2)^2 - 4| = 0$

(Q1)(2) Derive the update function $y_{n+1} = y_n - \frac{J}{\frac{dJ}{dy_n}}$, where $J = |y_n^2 - x|$

$J = f(y_n) = |y_n^2 - x|$

Let us assume that $y_{n+1}$ is our desired value, such that

$f(y_{n+1}) = 0$ —(i)

$$\text{Slope} = \frac{\text{change in } f}{\text{change in } y} = \frac{f(y_n) - f(y_{n+1})}{y_n - y_{n+1}} \quad \text{—(ii)}$$

Slope = derivate of $f(y_n)$

$$= \frac{df(y_n)}{dy_n} = \frac{dJ}{dy_n} \quad \text{—(iii)}$$

From (ii) and (iii)

$$\frac{dJ}{dy_n} = \frac{f(y_n) - f(y_{n+1})}{y_n - y_{n+1}}$$

$$y_n - y_{n+1} = \frac{J - 0}{\frac{dJ}{dy_n}} \quad \text{[From (i)]}$$

$$-y_{n+1} = -y_n + \frac{J}{dJ/dy_n}$$

$$y_{n+1} = y_n - J/(dJ/dy_n)$$

COST FUNCTION $\longrightarrow$ $J = |y_n^2 - x|$

UPDATE FUNCTION $\longrightarrow$ $y_{n+1} = y_n - \frac{J}{\delta J / \delta y_n}$

let

$u = y_n^2 - x$

Then,

$J = |u|$

$\frac{\delta |u|}{\delta u} = u/|u|$

Applying chain rule:

$$\frac{\delta J}{\delta y_n} = \frac{\delta |u|}{\delta u} \times \frac{\delta u}{\delta y_n}$$

$$= \frac{\delta |y_n^2 - x|}{\delta y_n} \quad \frac{\delta |u|}{2y_n} \quad \frac{u}{|u|} \times 2y_n$$

Substituting this into update function:

$$y_{n+1} = y_n - \frac{J}{\frac{u}{|u|} \times 2y_n}$$

$$= y_n - \frac{|y_n^2 - x|}{\frac{y_n^2 - x}{|y_n^2 - x|} \times 2y_n} = y_n - \frac{|y_n^2 - x|}{\text{sign}(y_n^2 - x) * 2y_n}$$

To simplify:

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

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

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

$$= \frac{y_n}{2} + \frac{x}{2y_n}$$

Q1 B) Write 'pseudo-code' to use this update function and go from $y_0$ to $y_n$

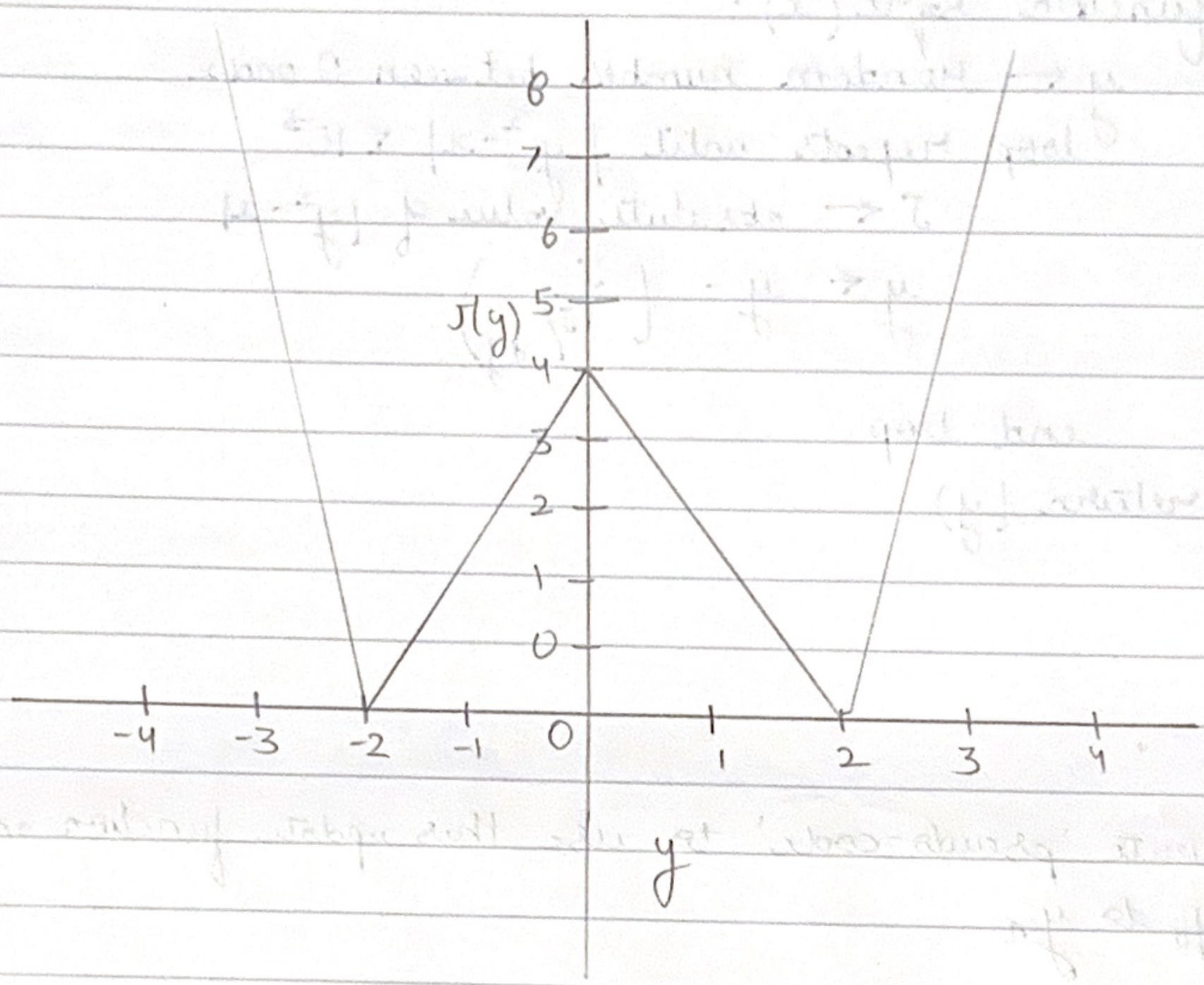
```
function sqrt(x, y_0, epsilon, j):
    y_n = y_0
    for i in range(j):
        J = abs(y_n^2 - x)
        if (J < epsilon):
            return(y_n)

        y_new = (y/2) + (x/2y)
        y_n = y_n - y_new

return(y_n)
```

Q1(4) Draw the cost function $J = (y_n^2 - x)^2$

$J = (y_n^2 - x)^2$




The function reaches minimum at $y=2$ and $y=-2$ where $x=4$, Hence, having two global minima.

Since square root of 4 is 2, selected values of $y \rightarrow -3$ to 3 to capture both positive and negative values.

$J = (y_n^2 - 4)^2$ at $x=4$

[ $y=2$

$J = ((2)^2 - 4)^2 = 0$

[ $y=-2$

$J = ((-2)^2 - 4)^2 = 0$

Q1(5) Derive the update function $y_{n+1} = y_n - J / dJ/dy_n$ where.

$J = (y_n^2 - x)^2$

$J = (y_n^2 - x)^2$

$f(y_n) = (y_n^2 - x)^2$

Let us assume $y_{n+1}$ is our desired value i.e $(y_{n+1})^2 = x$

$\therefore f(y_{n+1}) = 0$ —(i)

$$\text{Slope} = \frac{\text{Change in } f}{\text{Change in } y} = \frac{f(y_n) - f(y_{n+1})}{y_n - y_{n+1}} \quad \text{—(ii)}$$

Slope = derivative of $f(y_n)$

$$= \frac{\delta f(y_n)}{\delta y_n} = \frac{dJ}{\delta y_n} \quad \text{—(iii)}$$

From (ii) and (iii)

$$\frac{dJ}{\delta y_n} = \frac{f(y_n) - f(y_{n+1})}{y_n - y_{n+1}}$$

$$\frac{dJ}{\delta y_n} = \frac{J - 0}{y_n - y_{n+1}} \quad \text{from (i)}$$

$$y_n - y_{n+1} = \frac{J}{dJ/dy_n}$$

$$-y_{n+1} = -y_n + \frac{J}{dJ/dy_n}$$

$$y_{n+1} = y_n - J / dJ/dy_n$$

ALSO:

$$J = (y_n^2 - x)^2$$

$$\frac{dI}{dy_n} = 2(y_n^2 - x) \cdot \frac{d}{dy_n}(y_n^2 - x)$$ — Applied chain Rule

$$\frac{d}{dy_n}(y_n^2 - x) = 2y_n \longrightarrow A$$

Substituting (A)

$$\frac{dJ}{dy_n} = 2(y_n^2 - x) \cdot 2y_n = 4y_n(y_n^2 - x) \longrightarrow B$$

Now substituting (B)

$$y_{n+2} = y_n - \frac{J}{dJ/dy_n} = y_n - \frac{(y_n^2 - x)^2}{4y_n(y_n^2 - x)}$$

On Simplifying:

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

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

$$= y_n - \frac{y_n}{4} + \frac{x}{4y_n}$$

$$y_{n+1} = \frac{3}{4} y_n + \frac{x}{4y_n}$$

Q(6) Write pseudo-code to use this update function to go from $y_0$ to $y_n$:

```
function sqrt (x, y0, epsilon, j):
    yn = y0
    for i in range (j):
        J = (yn^2 - x)^2
        if J < epsilon:
            return (yn)
        ynew = (3/4) yn + (x/(4*yn))

        yn = yn - ynew

    return (yn)
```

# AI Report (ec24781)

## Question 1:

Is any path finding method consistently better?

## Answer:

Personally, I think no method consistently outperformed the others, BFS appeared to be superior because it has lower travel costs across all test routes—even going so far as to reduce the cost to 15 (New Cross Gate to Stepney Green) compared to the previous route, Canada Water to Stratford, which had a cost of 18—and the BFS's cost doesn't significantly differ when compared to the costs of other routes. The first total travel expense remains around 18.

## Question 2:

Report the count of the visited nodes by each algorithm for the most interesting routes you tested (minimum 4) and explain your results based on the knowledge of how each algorithm works.

## Answer:

Count of the visited nodes:

| Routes/ Algorithm | BFS | DFS | UCS | UCS_Cost | Heuristic_BFS |
|---|---|---|---|---|---|
| **Canada Water to Stratford** | 52 | 52 | 51 | 10 | 46 |
| **Canada Water to Stratford** | 53 | 33 | 17 | 50 | 50 |
| **Ealing Broadway to South Kensington** | 53 | 22 | 48 | 18 | 58 |
| **Baker Street to Wembley Park** | 26 | 100 | 73 | 18 | 21 |

1. **For BFS**

   The algorithm starts with a single source and proceeds to each vertex that is accessible from the source. We begin with the designated source and proceed level by level through the vertices using a queue data structure.

The tuple' values in the station_dict are contained in the connected_station variable in the for loop. Every one of these linked stations is iterated over by the for loop.
**connected_station[0]:** The next station's name.
**connected_station[1]**: The price or duration of travel to go to this nearby station.
**linked_station[2]**: The name of the line or route that links the two.

The value of connected_station[0] is saved in next_station, which is the name of the next station, and the value of connected_station[1] is saved in cost, which includes the cost of travelling between stations. After that, the if condition determines whether the next_station has already been visited; if so, it skips it; if not, it adds it to the visited set. Next, we merge the path and next_station to produce a new list(station).

The values of the visited set are then entered into visited_nodes, which is then updated. Its length is also counted to determine how many nodes have been visited overall. In end, we append the queue by passing arguments, next_station, station, cost, so that the BFS can reach the end station by exploring the neighbour vertices

2. **For DFS**

Regarding a graph, DFS,One by one, we look at each neighbouring vertex. When we travel via a nearby vertex, we have traversed all vertices reachable through that vertex. Once we have traversed one neighbouring vertex and all of its reachable vertices, we move on to the next nearby vertex and repeat the process.
The for loop iterates through every station in station_dict that is connected to start_station, just like BFS does. After that, the if condition determines whether the next_station has already been visited; if so, it skips it; if not, it adds it to the visited set.
Then, call back the function in recursive manner so that it can do in depth of the vertices till it finds dead end and then explores the next branch until end station is not found. (The NotDeadEnd variable is used to check whether the path is dead end or not, and the loop continues to next connected station.)

3. **For UCS**

In the Uniform Cost Search algorithm, a directed weighted search space is traversed from a start node to one of the ending nodes with the lowest cumulative cost.

Every station in station_dict that is connected to start_station is iterated through by the for loop. After that, the if condition determines whether the next_station has already been visited; if so, it skips it; if not, it adds it to the

visited set. Using the same for loop, we update the path by adding the next station to it, send the arguments to the queue by appending to it, and then add the trip cost to reach the next station from the current station to min_cost, which contains the cumulative cost to reach the present station.

Now each entry in priority_queue holds, cumulative cost to reach next station, name of next station and complete path to the station.

4. **For UCS added cost.**

   The for loop iterates through all stations present in station_dict, connected to start_station. Then if condition checks if the next_station has already been visited or not, if yes then it skips the station else it adds to the visited set. Again, by applying the same for loop, station_dict returns a list of tuples, where each tuple represents:

   **next_station**: name of the connected station.

   **min_cost**: travel cost to reach that station from current station.

   **next_line**: the line name associated with the route to next station.

   The if condition adds a line switch cost if the next like is different from the current line, line cost is set to 10 else line cost is set to 0.

   Then total_cost_ variable consist the sum of min cost, temp_tuple[1] (cost of the start station) and line cost.

   After that we take out the min value, lowest cumulative cost to reac next station.

   In end, new tuple is added to the priority_queue holding, new_min_cost, next_station, updated_path, next_line.

5. **Heuristic BFS**

   A search algorithm that uses the same heuristic information, to refine the search space while expanding nodes in breadth-first order. It assists in determining the best and quickest route to the objective.

   The for loop, goes through each of the connected stations, each station maps to a list of tuples, represents:

**connection**: the neighbouring station name

**cost**: travel cost to reach neighbour station from current station

**line**: the subway line connecting these two particular stations.

## Question 3:

Report the costs in terms of the time taken for the most interesting routes you tested (minimum 4) and explain your results based on the knowledge of how each algorithm works.

## Answer:

Total travel cost:

| Routes/ Algorithm | BFS | DFS | UCS | UCS_Cost | Heuristic_BFS |
|---|---|---|---|---|---|
| **Canada Water to Stratford** | 18 | 54 | 14 | 14 | 18 |
| **Canada Water to Stratford** | 15 | 54 | 15 | 15 | 15 |
| **Ealing Broadway to South Kensington** | 21 | 45 | 21 | 29 | 21 |
| **Baker Street to Wembley Park** | 19 | 71 | 19 | 34 | 19 |

I used the same logic to determine the trip cost for each of the five functions.

In order to track each station's contribution to the overall cost along the route, the loop initialises temp_dict.by utilising the list path, which contains the names of all the stations from beginning to end. When temp_dict[connected_station] for each station in the path is set to 0, it indicates that the cost has not yet been determined.

Next, each stop on the discovered route is represented by the outer loop (for connected_station in path), which iterates through each station in path.

Each of its connections is iterated over in the inner loop (for connected, cost, next_line). If the connected station is a component of the current path and the current station has already contributed to the overall cost, the if statement is contained within.

## Question 4:

Explain how you overcame all the issues you had while implementing all path finding algorithms, such as infinite loops, etc. Describe four or five issues you encountered.

## Answer:

I first made an effort to comprehend the theory behind each algorithm as well as the implementation code. I broke down the algorithms into manageable bits to better understand how they worked using Gemini and GeeksForGeeks.

**1)** Second, I struggled to understand the excel sheet during BFS because it lacked headers, leaving me unsure of what to do. In addition, I was attempting to figure out the meaning of the columns as in what column meant what. After a few minutes, I began to understand the concept and attempted coding in an attempt to comprehend the website's reasoning and connect it to the question.

I made the BFS code gradually, but I encountered problems with the tuple indexing. I debugged the code to understand how the flow of the stations is being added to the visited set and appended in the queue list.

Following that, I was able to see how a few logics would flow consistently across all codes, such as adding station names to the visited set, calculating the basic cost of travel, and determining whether the current station is equal to the end station before returning the arguments.

Because DFS and BFS were similar, I was able to code it with ease. The only difference was that I had to call the DFS function recursively until the current station equalled the end station. I had to review the concept and its implementations for the remainder.

**2)** Although I understood the theory, I had no idea how to put the logic into practice or apply it when calculating the travel cost function. After talking with a few friends, I began putting it into practice. To check if a station is in the path list, I first created a temporary dictionary. Then, I took the cost of that station out of the main station dictionary and made sure the value in the temporary dictionary was equal to zero. If it was, the cost would be added. Additionally, the previous station must be connected to the next station's tuple in order to prevent the station from entering the cycle.

After learning about our course, I attempted to modify my code to meet the new requirements. Previously, we had a common function to implement the total cost of travel. I attempted to implement the same function and call it

in every algorithm, but despite numerous attempts, it did not function as intended. After realising that the cost was provided as an argument, I attempted to apply the reasoning for travel costs to each function and succeeded in passing it back.

**3)** I first attempted to return the default visited set, which was provided, in order to return the names of visited nodes. However, I encountered several problems with it, including the fact that it was returning station names in alphabets separated by commas and displaying a type error. To avoid this, I created a new set and updated the visited set's values onto the new one (visited_nodes).

**4)** My heuristic_bfs was not functioning correctly in the updated search.py file; the visited set had taken alphabets from the start station and was unable to unpack the result when it was time to print it. To prevent it from splitting the word into comma-separated letters, I placed square brackets on the start station. Additionally, I was unsure of how to determine the value for the heuristic in the function zone_heuristic. I attempted a couple mathematical methods, but they produced large numbers that led to odd total cost amounts. I then just assigned it a value of two.

## Question 5:

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

## Answer:

The journey from Harrow & Wealdstone to St. John's Wood was the furthest I could find.
I used two functions: one to update the longest route if it was identified, and another to explore neighbours if the depth restriction permits.
The current station is appended to the visited set and the current path for the first function. It continues to explore every nearby station if the depth limit is larger than zero. In order to examine alternative routes, the function ultimately reverses itself by deleting the current station from the visited collection.
The second function determines whether the recently discovered path is longer than the paths that were previously discovered. It will exit the loop and proceed to the next station if no new routes are discovered at the current depth.