Assignment 3

12112124 尹晨

Since this operation involves the conversion of multiple algorithms, heuristic functions, etc., I only implemented the A* algorithm in the.py file. Other algorithmic codes mentioned in this presentation are presented in an **appendix**.

All of the following results are run code:

```
python find_shortest_path.py  "Acton Town"  "Bermondsey"
```

Part 1 A star algorithm:

1.1 Algorithm idea:

The **core idea** of the algorithm is as follows:

The **get_path** function takes three arguments:

start_station_name: indicates the name of the starting station
end_station_name: indicates the name of the target subway station
map: A dictionary that maps subway Station names to corresponding station objects

Get the Station objects of the origin and destination stations using *map[start_station_name]* and *map[end_station_name]*.

Priority queues are implemented using a **heap**, where the elements are a binary group *(current_cost, current_station)* representing the current total cost of reaching current_station.

The open_set is initialized as the priority queue, closed_set is used to record the sites that have been processed, and came_from records the site through which each site is reached.

Put the cost of the origin station and the site into the heap, and initialize *came_from[start_station] = None*.

Enter the main loop, continuously eject the currently least expensive site from the heap, and then add it to closed_set.

Iterate over the neighbor of the current site (the adjacent subway station), calculate the cost to that neighbor, and if the neighbor is not in closed_set, update the cost and add it to the heap.

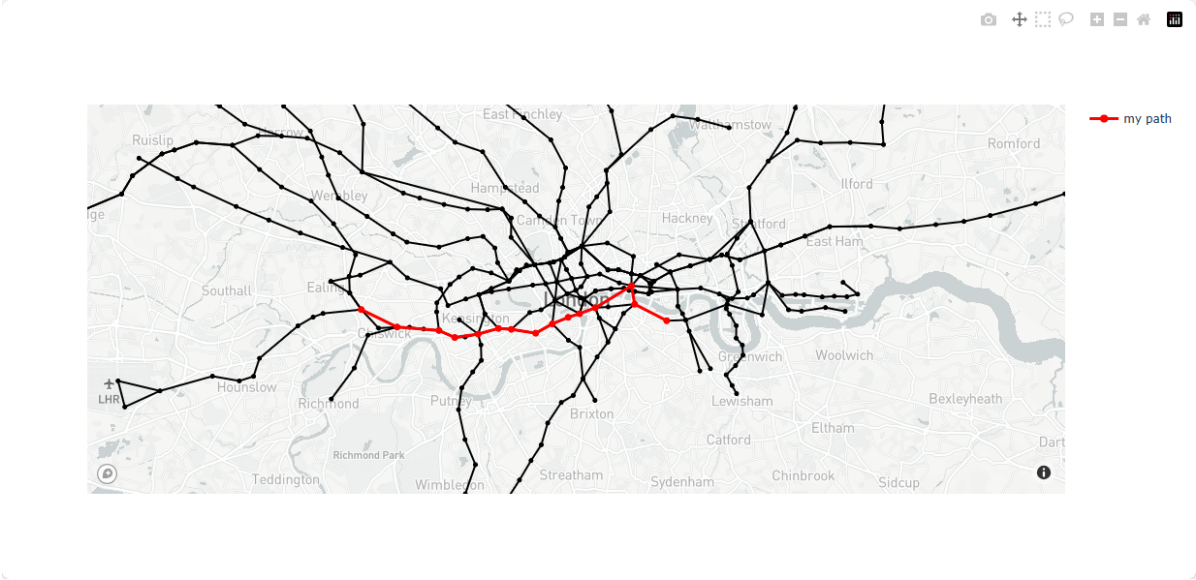
If the current site is the target site, retrace the build path and return.

If the heap is empty and no path is found, an empty list is returned.

1.2 Application of heuristic functions

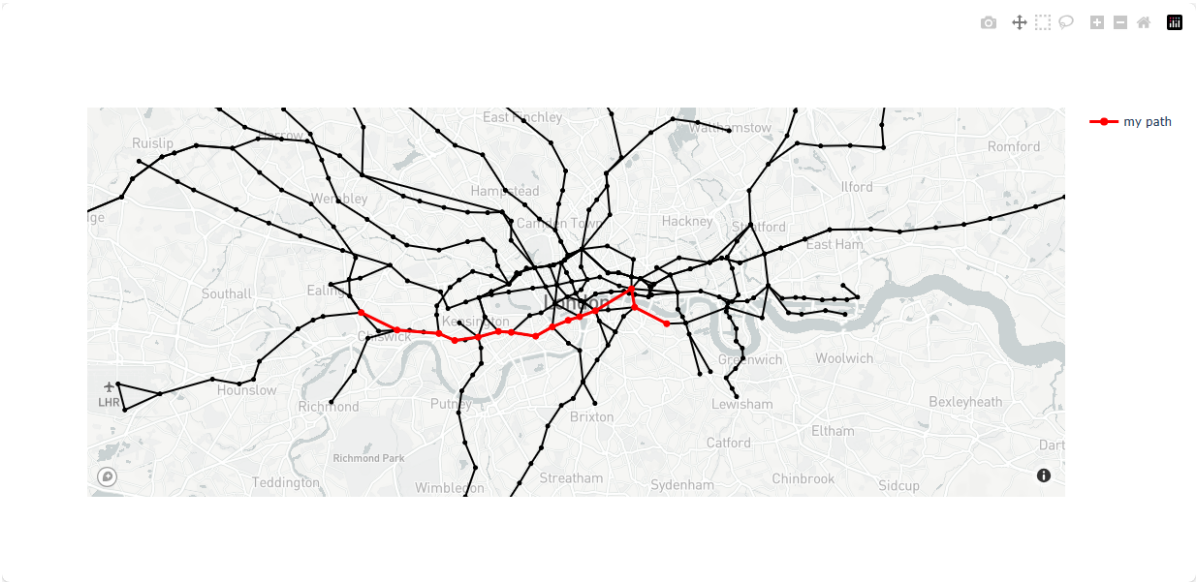
1.2.1 Euler distance:

We use Euler distance (Appendix 1) as a heuristic estimation function and customize the implementation of the heuristic function. The following image results are obtained after running:



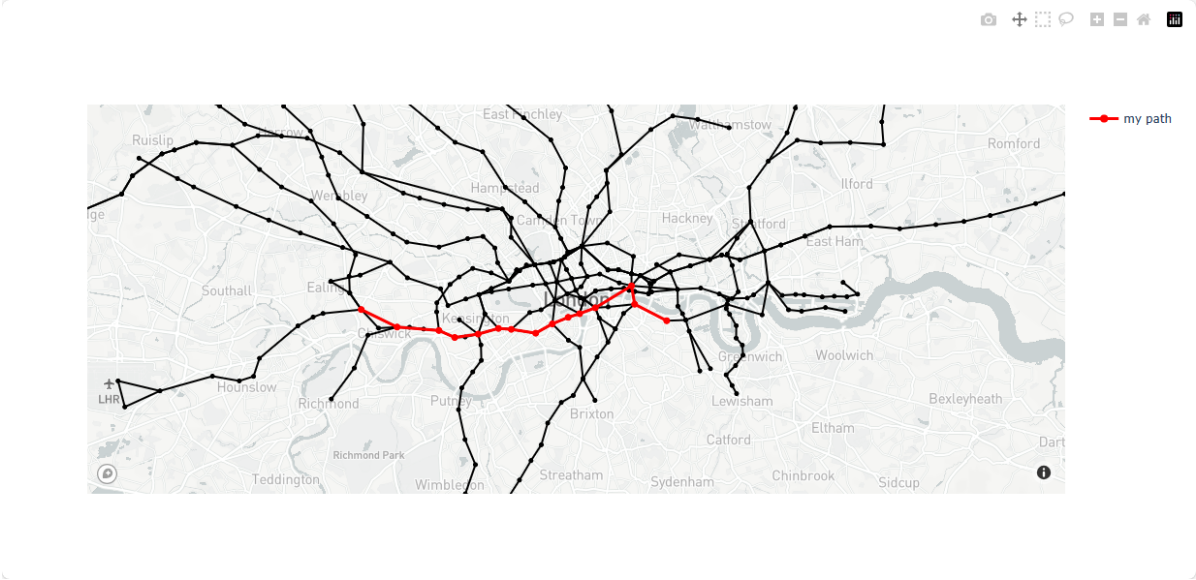
1.2.2 Manhattan Distance

Next, we replace the heuristic function with the Manhattan distance and get the following result:



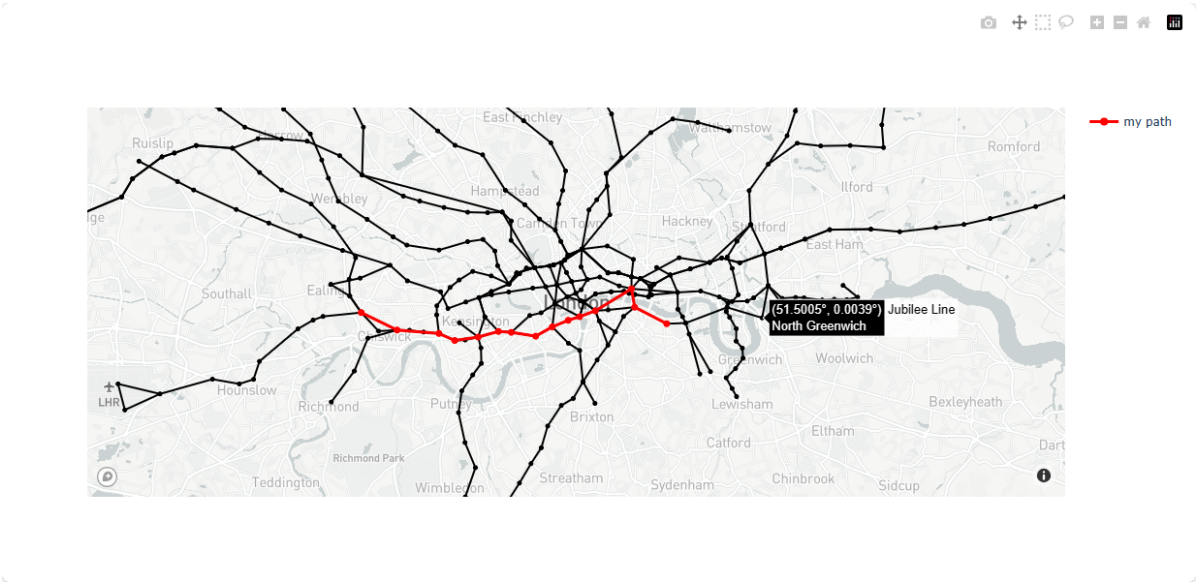
1.2.3 Chebyshev Distance

Similarly, we replace the heuristic function with Chebyshev distance:



1.2.4 Diagonal Distance

Next we replace the heuristic function with Diagonal Distance:



1.3 Conclusions and questions

We can see that it seems that changing the different heuristic function does not have much effect on the result of the problem.

Question 1 : Why do we change the heuristic function but get the same result?

We switched to a different heuristic function, but got the same result, for several possible reasons:

Small problem scale: If the number of subway stations is small, it is possible that no matter which heuristic function is used, the A* algorithm will be able to find the optimal path in a relatively short time, so the result will be the same.

Heuristic functions have similar effects: Sometimes different heuristic functions may have similar effects on certain problems, especially if there is not much difference in the structure of the problem.

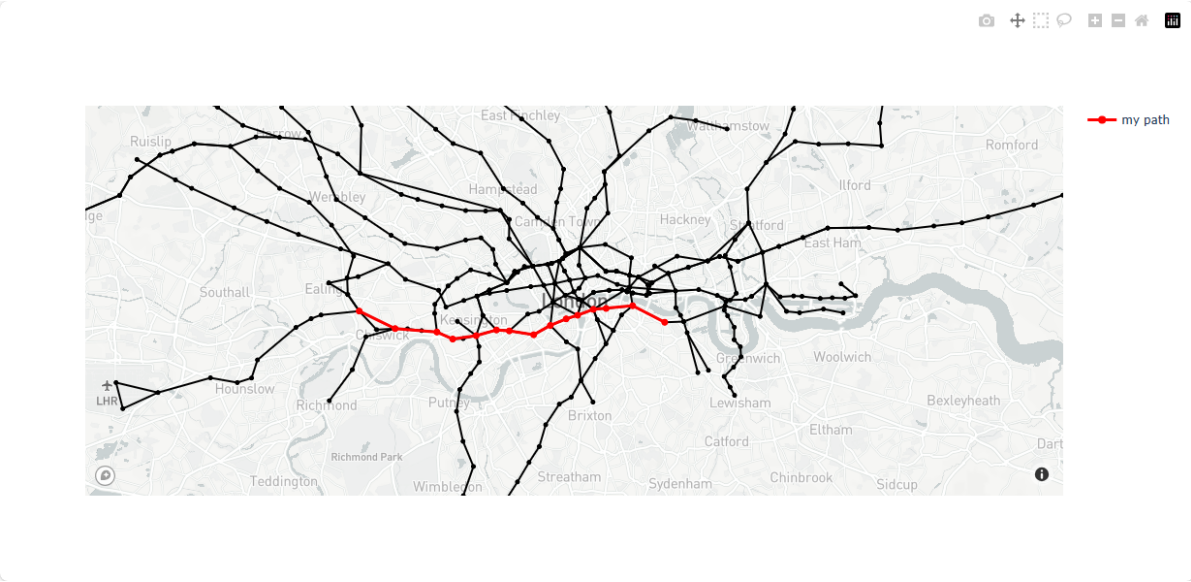
The heuristic function does not have A significant effect on the algorithm: In some cases, the nature of the problem may cause the effect of the heuristic function on the A* algorithm to be less obvious, so choosing a different heuristic function does not make a significant difference.

Question 2 : How do we verify the effect of the heuristic function?

We can try to test on a larger scale problem, or try to introduce more complexity into the structure of the problem. In addition, it is also possible to further understand the behavior of the algorithm by output some intermediate results during the operation of the algorithm, such as generation value, search path, etc.

Part 2 Dijkstra's algorithm

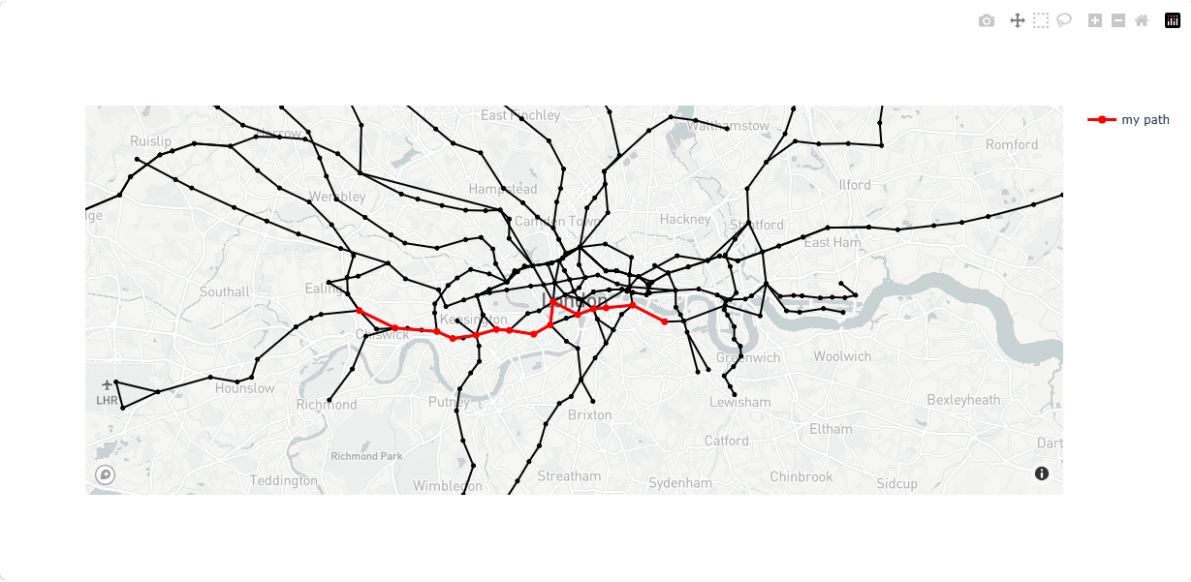
We replaced it with Dijkstra's algorithm (Appendix 2) and got the following result:



We can see that after changing to Dijkstra's algorithm, the shortest path given by the algorithm has changed.

Part 3 :Bellman-Ford Algorithm:

We change the Algorithm again to the Bellman-Ford Algorithm (Appendix 3), and we can see that this time the route also changes:



Part 4 : Freude-voschel algorithm

We replace the algorithm with the Freude-Voschel algorithm and add a new function **extract_shortest_path**, which extracts the shortest path based on the results of the Floyd-Warshall algorithm.

This time we found that the code ran surprisingly slow, indicating that the time complexity of this algorithm is large.

There was nothing wrong with the logic of our code, but the code ran for 12 hours without any output. So I finally chose to give up the implementation of the algorithm on this problem.

Let's turn to **why the algorithm runs for such a long time**.

There are several possible reasons why Bellman-Ford, A*, and Dijkstra all run fast on the same data set while Floyd-Warshall's algorithm is slower:

Time complexity:
The time complexity of Floyd-Warshall's algorithm is O(V^3), where V is the number of sites. In contrast, Dijkstra's algorithm and the A* algorithm can reach O(V^2) or O(E + V log V) in some cases, where E is the number of edges. In large-scale networks, the time complexity of Floyd-Warshall may result in long runtimes.

Algorithm design:
Floyd-Warshall algorithm is a global shortest path algorithm that calculates the shortest path between each pair of nodes. Dijkstra algorithm and A* algorithm are single-source shortest path algorithms, they only calculate the shortest path from one source node to other nodes. In some cases, global algorithms may become slower due to redundant calculations.

Memory access:
The triple loop of the Floyd-Warshall algorithm can result in an unfriendly memory access pattern, which can cause cache misses in large networks, affecting performance. In contrast, Dijkstra's algorithm and the A* algorithm generally have better cache locality.

Optimization problem:
The Floyd-Warshall algorithm generally performs better on dense graphs and may be less efficient on sparse graphs. If your subway network map is sparse, other algorithms may be more suitable.

Part 5 : Time complexity analysis of different algorithms

5.1 A* Algorithm:

Time complexity: Depends on the underlying search algorithm used.
In the worst case, exponential complexity can be achieved, but in practice it is usually much lower because of heuristic functions.

5.2 Dijkstra's Algorithm:

Time complexity: O((V + E) * log(V)), where V is the number of nodes and E is the number of edges. The time complexity of Dijkstra's algorithm using priority queue (minimum heap) is mainly affected by heap operations.

5.3 Bellman-Ford Algorithm:

Time complexity: O(V * E), where V is the number of nodes and E is the number of edges. Because of the use of a loop that relaxes all edges, it may perform poorly on larger graphs.

5.4 Floyd-Warshall Algorithm:

Time complexity: O(V^3) where V is the number of nodes.
Iterate over all pairs of nodes through a three-tier nested loop, so performance on large-scale graphs can be poor.

Part 6 : Analyze the advantages and disadvantages of different algorithms

6.1 A* algorithm :

6.1.1 Advantages:
A* is a heuristic search algorithm that uses heuristics to efficiently find the path with the lowest cost.
It is generally faster than algorithms without information, such as Dijkstra's algorithm, because it uses heuristics to guide the search.
It is suitable for situations with good heuristics and can be used to estimate residual costs.

6.1.2 Limitations:
Performance depends largely on the quality of the heuristic.

6.2 Dijkstra's algorithm:

6.2.1 Advantages:
Ensure that the shortest path is found.
Applies to graphs without negative weighted edges.

6.2.2 Limitations:
When the graph is large, the time complexity is O((V + E) * log(V)), where V is the number of nodes and E is the number of edges.
Does not apply to graphs with negative weighted edges.

6.3 Bellman-ford algorithm:

6.3.1 Advantages:
Graphs with negative weighted edges can be handled.

6.3.2 Limitations:
The time complexity is O(V * E), which is high.
In some cases, it may not be as fast as Dijkstra's algorithm.

6.4 Freude-voschel algorithm:

6.4.1 Advantages:
The shortest path between all pairs of nodes can be calculated.
Applicable to graphs with negative weighted edges.

6.4.2 Limitations:
The time complexity is O(V^3), which may not be efficient enough for large graphs.
It is not suitable for dynamic graphs because all paths need to be recalculated every time there is a change.

Part 7 : conclusion

After comparing the effects of different algorithms and the effects of different heuristic functions inside A* algorithm, we can draw the following conclusions:

7.1 If the graph is dense (with a large number of nodes) and has no negative weighted edges:

Dijkstra's algorithm is recommended:
Dijkstra's algorithm usually performs better in this case, especially when the heuristic function is difficult to define in the A* algorithm.

7.2 If the graph is dense and has negative weighted edges:

The **Floyd-Warshall algorithm** is recommended:
The Floyd-Warshall algorithm is suitable for graphs with negative weighted edges and can calculate the shortest path between all pairs of nodes.
Although the time complexity is higher, it performs better for small-scale graphs.

7.3 If the graph is sparse:

Algorithm A* is recommended:
By introducing A heuristic function, A* algorithm can find the shortest path more efficiently in sparse graphs.
Given A suitable heuristic, the A* algorithm is generally faster than Dijkstra's algorithm.

7.4 If the graph may have negative weighted edges and is sparse:

The **Bellman-Ford algorithm** is recommended:
Bellman-Ford algorithm can handle negative weighted edges and is suitable for sparse graphs.
Despite the high time complexity, the handling of negative weighted edges makes it a viable option.

Part 8 : Possible future improvements or research directions

Possible future improvements or research directions include the following:

8.1 More efficient heuristic function design:

The performance of A* algorithm largely depends on the quality of the heuristic function. Future research can focus on designing more efficient heuristic functions to improve the effectiveness of A* algorithms in practical applications.

8.2 Optimization of large-scale graphs:

For large-scale metro networks, existing algorithms can be explored and optimized, or new ones developed to address performance and efficiency challenges. Distributed algorithms and parallelization techniques may be one direction.

8.3 Real-time path planning:

Path planning for real-time applications is an interesting direction. Research can focus on how to respond to changes in real time and deal with dynamically changing graph structures in subway networks.

8.4 Multimodal path planning:

Considering the diversity of urban transportation, future research can be extended to multimodal path planning, including the integration of multiple transportation modes such as subway, bus, and walking.

8.5 Application of reinforcement learning in path planning:

Try to introduce reinforcement learning into the field of path planning, so that the algorithm can learn more effective path selection strategies through interaction with the environment.

8.6 Adaptation to dynamic networks:

In view of the dynamic nature of subway network, especially the changes of station switch and line adjustment, this paper studies how to make the path planning algorithm better adapt to these dynamic changes.

In general, the future research direction should be aimed at improving the adaptability, efficiency and real-time performance of path planning algorithms in real urban traffic. This involves improvements to the algorithms themselves, as well as a deeper understanding of the characteristics of urban transport systems.

appendix:

Appendix 1: *A star algorithm*:

Heuristic function:

①Euler Distance

```
def heuristic(station1, station2):
    return math.sqrt((station1.position[0] - station2.position[0])**2 +
(station1.position[1] - station2.position[1])**2)
```

②Manhattan Distance

```
def manhattan_distance(station1, station2):
    return abs(station1.position[0] - station2.position[0]) +
abs(station1.position[1] - station2.position[1])
```

③Chebyshev Distance

```
def chebyshev_distance(station1, station2):
    return max(abs(station1.position[0] - station2.position[0]),
abs(station1.position[1] - station2.position[1]))
```

④Diagonal Distance

```
def diagonal_distance(station1, station2):
    dx = abs(station1.position[0] - station2.position[0])
    dy = abs(station1.position[1] - station2.position[1])
    return max(dx, dy) + (math.sqrt(2) - 1) * min(dx, dy)
```

Algorithm implementation:

```
def get_path(start_station_name: str, end_station_name: str, map: dict[str,
Station]) -> List[str]:
    start_station = map[start_station_name]
    end_station = map[end_station_name]

    open_set = []
    closed_set = set()
    came_from = {}
```

```
    heapq.heappush(open_set, (0, start_station))
    came_from[start_station] = None

    while open_set:
        current_cost, current_station = heapq.heappop(open_set)

        closed_set.add(current_station)

        for neighbor in current_station.links:
            if neighbor in closed_set:
                continue

            tentative_cost = current_cost + 1 + heuristic(neighbor, end_station)

            neighbor_in_open_set = [(cost,s) for cost, s in open_set if s ==
neighbor]
            if not neighbor_in_open_set or tentative_cost <
neighbor_in_open_set[0][0]:
                heapq.heappush(open_set, (tentative_cost, neighbor))
                came_from[neighbor] = current_station

        if current_station == end_station:
            path = []
            while current_station:
                path.insert(0, current_station.name)
                current_station = came_from[current_station]
            return path

    return []
```

Appendix 2: Dijkstra's algorithm:

```
from typing import List
from plot_underground_path import plot_path
from build_data import Station, build_data
import argparse
import math

def dijkstra(start_station_name: str, end_station_name: str, map: dict[str,
Station]) -> List[str]:
    start_station = map[start_station_name]
    end_station = map[end_station_name]

    distances = {station_name: float('inf') for station_name in map}
    distances[start_station_name] = 0
    came_from = {}

    unvisited = set(map.keys())

    while unvisited:
        current_station_name = min(unvisited, key=lambda name: distances[name])
        unvisited.remove(current_station_name)

        current_station = map[current_station_name]
```

```
        for neighbor in current_station.links:
            neighbor_name = neighbor.name
            tentative_distance = distances[current_station_name] + 1

            if tentative_distance < distances[neighbor_name]:
                distances[neighbor_name] = tentative_distance
                came_from[neighbor_name] = current_station_name

    path = []
    current_station_name = end_station_name
    while current_station_name:
        path.insert(0, current_station_name)
        current_station_name = came_from.get(current_station_name)

    return path if path and path[0] == start_station_name else []

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('start_station_name', type=str,
help='start_station_name')
    parser.add_argument('end_station_name', type=str, help='end_station_name')
    args = parser.parse_args()
    start_station_name = args.start_station_name
    end_station_name = args.end_station_name

    stations, underground_lines = build_data()
    path = dijkstra(start_station_name, end_station_name, stations)
    plot_path(path,
'visualization_underground/my_shortest_path_in_London_railway_dijkstra.html',
stations, underground_lines)
```

Appendix 3 : Bellman-Ford Algorithm

```
from typing import List
from plot_underground_path import plot_path
from build_data import Station, build_data
import argparse
import math

def bellman_ford(start_station_name: str, end_station_name: str, map: dict[str,
Station]) -> List[str]:
    start_station = map[start_station_name]
    end_station = map[end_station_name]

    distances = {station_name: float('inf') for station_name in map}
    distances[start_station_name] = 0
    came_from = {}

    for _ in range(len(map) - 1):
        for current_station in map.values():
            for neighbor in current_station.links:
                neighbor_name = neighbor.name
                tentative_distance = distances[current_station.name] + 1

                if tentative_distance < distances[neighbor_name]:
```

```
                    distances[neighbor_name] = tentative_distance
                    came_from[neighbor_name] = current_station.name

    path = []
    current_station_name = end_station_name
    while current_station_name:
        path.insert(0, current_station_name)
        current_station_name = came_from.get(current_station_name)

    return path if path and path[0] == start_station_name else []

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('start_station_name', type=str,
help='start_station_name')
    parser.add_argument('end_station_name', type=str, help='end_station_name')
    args = parser.parse_args()
    start_station_name = args.start_station_name
    end_station_name = args.end_station_name

    stations, underground_lines = build_data()
    path = bellman_ford(start_station_name, end_station_name, stations)
    plot_path(path,
'visualization_underground/my_shortest_path_in_London_railway_bellman_ford.html',
stations, underground_lines)
```

Appendix 4 : Floyd-Warshall Algorithm

```
from typing import List
from plot_underground_path import plot_path
from build_data import Station, build_data
import math

def floyd_warshall(map: dict[str, Station]) -> List[List[int]]:
    stations_list = list(map.keys())
    num_stations = len(stations_list)

    distances = [[float('inf') for _ in range(num_stations)] for _ in
range(num_stations)]

    for i in range(num_stations):
        distances[i][i] = 0

    for station_name, station in map.items():
        for neighbor in station.links:
            j = stations_list.index(station_name)
            k = stations_list.index(neighbor.name)
            distances[j][k] = 1

    for k in range(num_stations):
        for i in range(num_stations):
            for j in range(num_stations):
                distances[i][j] = min(distances[i][j], distances[i][k] +
distances[k][j])
```

```
    return distances, stations_list

def extract_shortest_path(distances, stations_list, start_station_name,
end_station_name):
    start_station_index = stations_list.index(start_station_name)
    end_station_index = stations_list.index(end_station_name)

    if distances[start_station_index][end_station_index] == float('inf'):
        # If there is no path between the two stations
        return []

    # Reconstruct the shortest path
    current_index = start_station_index
    path = [stations_list[current_index]]
    while current_index != end_station_index:
        next_index = min(range(len(stations_list)), key=lambda i:
distances[current_index][i])
        path.append(stations_list[next_index])
        current_index = next_index

    return path

if __name__ == '__main__':
    stations, underground_lines = build_data()
    all_shortest_paths, stations_list = floyd_warshall(stations)

    start_station_name = "Acton Town"
    end_station_name = "Bermondsey"

    # Extract the shortest path
    path = extract_shortest_path(all_shortest_paths, stations_list,
start_station_name, end_station_name)
    print(f"Shortest path: {path}")

    # Visualize the shortest path
    plot_path(path,
'visualization_underground/my_shortest_path_in_London_railway_floyd_warshall.html
', stations, underground_lines)
```