Minimum-Stop Routing in the Hong Kong MTR Graph: A Breadth-First Search Approach

Mahesa Fadhillah Andre – 13523140
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: mahesa0208@gmail.com, 13523140@std.stei.itb.ac.id

***Abstract*—Minimum-stop routing in transportation networks is a crucial problem in planning efficient journeys. This paper discusses the application of the Breadth-First Search (BFS) algorithm to find routes with the minimum number of stops between two stations in the Hong Kong MTR network, which is modeled as an unweighted, undirected graph. BFS is selected because it guarantees the discovery of the path with the minimum number of nodes traversed in such graphs. The program is implemented in Python, taking station and connection data in JSON format as input. Experimental results show that BFS can efficiently find minimum-stop routes for various station pairs in the Hong Kong MTR network, with relatively fast execution time. This paper also presents an analysis of the algorithm's complexity, testing results using real data, and a discussion of the advantages and limitations of this approach in the context of urban transportation.**

Keywords—Breadth-First Search, minimum-stop routing, graph, Hong Kong MTR, pathfinding algorithm Introduction (*Heading 1*)

I. Introduction

Urban transportation plays a key role in supporting mobility in large cities. Systems like buses, taxis, trains, and ride-hailing services help reduce congestion and provide efficient ways for people to travel within urban areas.

Urban transportation refers to the movement of people and goods within an urban or metropolitan area. It serves as the backbone of a city's mobility, providing accessible, affordable, and efficient means for commuting. Beyond individual convenience, urban transportation systems contribute significantly to a city's economic growth, social connectivity, and environmental sustainability. For many residents, these systems offer a cost-effective and time-saving alternative to private vehicles, helping to alleviate parking issues and reduce carbon emissions.

Among the various forms of urban transportation, one of the most effective and widely adopted solutions is the subway or Mass Transit Railway (MTR) system. MTR systems are designed to integrate seamlessly into the urban environment, providing rapid and frequent services that can transport large numbers of passengers across the city efficiently. Trains typically arrive at stations at short intervals, often just minutes apart, and operate for extended hours, from early morning to late at night. This makes MTR systems highly dependable and convenient for daily commuters, tourists, and businesses alike. Moreover, by minimizing travel time and increasing transport capacity, these systems play a crucial role in enhancing overall productivity and supporting urban economies.

Despite their advantages, MTR systems in large cities present unique challenges, particularly in route planning and navigation. As cities expand and MTR networks grow more complex, with dozens or even hundreds of stations interconnected by multiple lines, navigating these networks can become overwhelming, especially for new users or tourists. Identifying the most efficient route between two stations is not always straightforward. A common objective for travellers is to find the path that minimizes the number of stops between their starting point and destination, thereby reducing travel time and unnecessary transfers.

This paper explores the *Breadth-First Search* (BFS) algorithm as a solution to finding the minimum stops needed in order to get from one station to another in an MTR system. Hong Kong's MTR network is used as the case study for this solution. Hong Kong's MTR system is well known for it's dense and extensive network, so it provides an excellent study case to demonstrate this approach. Graph theory serves as the foundation for representing this model. Nodes represent stations in the network and the edges represent the connection between each neighbouring stations in the network.

By modelling MTR as an unweighted, undirected graph and applying the BFS algorithm, the paper will demonstrate an effective method to find the minimum stops or routes to help passengers navigate the complex network with ease.

II. Theoritical Basis

A. Mass Transit Railway (MTR) as a Network

A Mass Transit Railway (MTR) system is one of the most common forms of urban public transportation. This type of system consists of multiple train lines that connect various stations across a city or region. The stations serve as points where passengers can board or leave the train, and the lines provide direct connections between these stations. The structure of an MTR system forms a complex network that is designed to help people move efficiently from one location to another.

To study and analyze the MTR network more clearly, the system can be represented using a mathematical model. In this model, each station is treated as a point or node, and each direct connection between two stations is treated as a link or line that joins the points. This way of representing the MTR system helps to simplify the network into a structure that can be examined using various techniques. This structure is what is later referred to as a graph.

B. Graph Theory

A graph is a non-linear data structure that is used to represent pairwise connections or relationships between objects. In formal mathematical terms, a graph is written as **G = (V, E)**, where **V** is the set of vertices, or nodes, and **E** is the set of edges that connect pairs of vertices. The vertices usually stand for objects, entities, or points in a system, and are commonly drawn as circles or dots in diagrams to make them easy to identify. The edges represent the links or relationships between these vertices and are drawn as lines connecting two vertices. Graphs have become widely used because they can model so many types of systems and networks found in real life. For example, graphs are often used to model friendships in social networks, stations in a railway system, cities connected by roads, computers in a network, or websites linked by hyperlinks. The flexibility of graphs makes them useful in solving problems involving connectivity, reachability, flow, or the shortest path between points. Graphs are applied in many fields, including computer science, engineering, biology, logistics, and many others where relationships or interactions between parts of a system need to be studied.

C. Simple Graph

A simple graph is one of the most basic forms of a graph. This type of graph follows specific rules to keep its structure clean and easy to work with. In a simple graph, no vertex connects to itself, which means that self-loops are not allowed. In addition, no two vertices are connected by more than one edge, so between any pair of vertices, there is at most one edge.

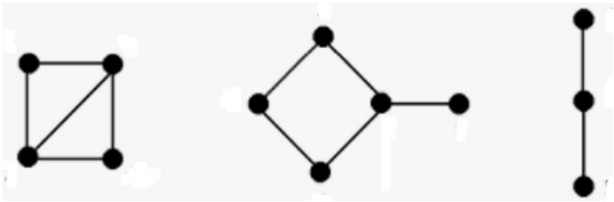

Fig 2.1 Simple Graphs
(Source: Graph (Part 1) Slide by Dr. Rinaldi Munir)

The edges in a simple graph are undirected, so the relationship between two connected vertices is mutual. if vertex A is connected to vertex B, then vertex B is connected to vertex A in the same way. The edges also do not carry any weight or numerical value; they only indicate that a connection exists. Simple graphs are useful when the system being modeled only requires information about whether a connection exists or not, without needing to know anything about the strength, cost, or direction of the connection. Simple graphs are often used in cases like basic social networks, where all friendships are considered equal, or in unweighted transportation networks where the existence of a route matters but not its distance or cost.

D. Unsimple Graph

An unsimple graph is a graph that does not follow the rules of a simple graph. This type of graph allows features that simple graphs do not, such as self-loops or multiple edges between the same pair of vertices. These additional features make it possible to model more complex or layered relationships. An unsimple graph is useful in situations where two objects can be connected in more than one way, or where an object can connect to itself. There are two common types of unsimple graphs.

1. Multi Graph

The first is called a multigraph, where multiple edges between the same pair of vertices are allowed, but self-loops are not. This kind of graph can represent systems like public transportation networks where two cities might have several different bus or train lines connecting them.

2. Psudo-Graph

The second type is called a pseudograph, which allows both multiple edges and self-loops. Pseudographs are used in cases where self-connections are meaningful, such as in feedback loops in a control system or in situations where an entity might interact with itself, like a machine that sends data back to itself.

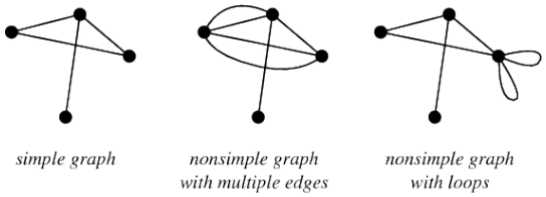

Fig 2.2 Unsimple Graphs
(Source: Graph (Part 1) Slide by Dr. Rinaldi Munir)

In addition to classifying graphs as simple or unsimple, graphs can also be categorized in other ways based on different properties of their edges. One common classification is based on whether the edges have direction. A directed graph, or digraph, is a graph in which each edge has a specified direction. This means that connections between pairs of vertices do not necessarily work the same in both directions. In a directed graph, edges are represented as arrows pointing from one vertex to another, clearly indicating where the connection starts and where it ends. Directed graphs are especially useful for modeling systems where relationships between entities are not always mutual.

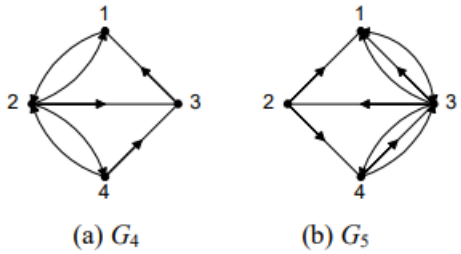

Fig 2.3 Directed Graphs
(Source: Graph (Part 1) Slide by Dr. Rinaldi Munir)

Another way graphs can be classified is by considering whether edges carry additional information such as numerical values. A weighted graph is a graph where each edge has an associated number, known as a weight.

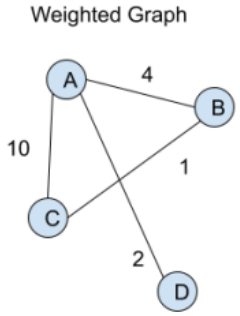

Fig 2.4 Directed Graphs
(Source: Graph (Part 1) Slide by Dr. Rinaldi Munir)

These weights can represent various attributes depending on the system being modeled. In transportation networks, for example, weights might indicate the distance or travel time between locations. Weighted graphs are essential in optimization tasks, such as finding the shortest path, the fastest route, or the most cost-effective way to connect different points in a network.

E. Graph Terminologies

There are several basic terms that are often used when describing or working with graphs.

1. Adjacency (Neighbours)

One of the most important is neighbours. Two vertices are considered neighbours if they are directly connected by an edge. In an undirected graph, if vertex A is a neighbour of vertex B, then vertex B is also a neighbour of vertex A. In directed graphs, a neighbour could be incoming or outgoing, depending on whether the edge points to or away from the vertex. Understanding which vertices are neighbours is essential in algorithms that explore a graph, like breadth-first search (BFS) or depth-first search (DFS).

2. Adjacency List

There are also different ways to represent a graph in a computer so that it can be stored and processed efficiently. One common method is the adjacency list. In this method, each vertex keeps a list of all the other vertices it is connected to. This method is good for graphs that don't have too many edges, because it only stores what actually exists and does not waste space on connections that aren't there.

3. Path

A path in a graph is a sequence of vertices where each consecutive pair of vertices is connected by an edge. A path describes a way to move through the graph from one vertex to another by following the edges that link them. The length of a path is measured by the number of edges it contains. A path is called simple if no vertex appears more than once along the path, except possibly the first and last vertex if the path forms a cycle. Paths are important in graph theory because they are used to study how vertices are connected and how to travel between them within the network.

Understanding these basic terms and representations is essential because they provide the foundation for applying various algorithms to graphs. The concepts of neighbours, paths, and adjacency help in describing how vertices are linked and how information or movement can pass through a network. With these definitions in place, it becomes possible to study algorithms that explore graphs, search for specific vertices, or find optimal paths.

F. Breadth First Search

Breadth-First Search (BFS) is one of the basic algorithms used to explore or traverse graphs. This algorithm works by visiting vertices level by level, starting from a chosen vertex and visiting all of its neighbours first before moving on to the neighbours of those neighbours. The idea is to explore all vertices that are closest to the starting point before going deeper into the graph.

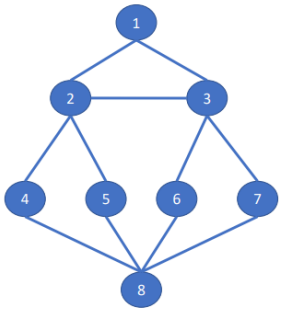

Fig 2.5 BFS Vertex Traversal Order

(Source: Breadth First Search (BFS) and Depth First Search (DFS) 2025 (Part 1) Slide by Dr. Rinaldi Munir)

In a BFS, vertices that are discovered but whose neighbours have not been fully explored are kept in a queue. This queue makes sure that vertices are visited in the right order, first those that are closest to the starting point, then those that are one step further, and so on. Each vertex is marked once it has been visited, so the algorithm does not visit the same vertex more than once.

The process of BFS can be visualized as forming a tree, often called a BFS tree. The root of this tree is the starting vertex, and the tree shows how other vertices are reached step by step. This tree helps to show the layers in the traversal and the order in which the vertices are visited.

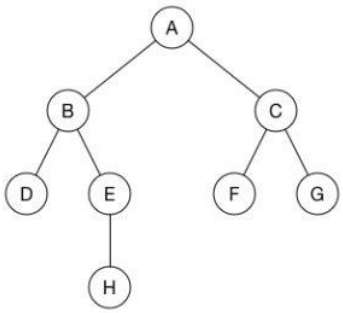

Fig 2.6 BFS Tree
((Source: Breadth First Search (BFS) and Depth First Search (DFS) 2025 (Part 1) Slide by Dr. Rinaldi Munir)

For example the traversal order for the BFS algorithm in Figure 2.6 is as follows: A, B, C, D, E, F, G, H. Each step of the algorithm can be recorded in a table that shows which vertex is being visited, the contents of the queue, and which vertices have been marked as visited. This makes it easier to follow the process.

Breadth-First Search has been chosen in this study because of its ability to find the shortest path in terms of the number of stops. In the next section, the focus will shift to how this algorithm is implemented to solve the problem of finding the shortest path between two stations in an MTR network. The method of representing the data, the design of the program, and the steps of the algorithm in practice will be explained in detail.

III. Implementation Method

This program was developed to find the shortest path, in terms of stops, between two stations in an MTR network using Breadth-First Search (BFS). The dataset for the MTR system was taken from Kaggle. The original Kaggle data was generated through accessing public data from the official MTR Website and processed into a JSON format where each station includes its name and a list of neighbouring stations. The data structure makes it possible to represent the network as a graph where stations act as nodes and direct connections as edges.

The program is written in Python and consists of three main files: node.py, station_loader.py, and route_finder.py. Each file has a specific role in building and searching the graph.

A. Data

The station data is stored in a JSON file that describes the network as a list of station entries. Each entry represents a station and contains two main attributes:

- "name": A string that represents a station's name.
- "neighbours": A list of neighbouring stations, where each neighbour is represented as a dictionary containing its "name".

```
{
    "name": "Whampoa",
    "neighbours": [
      {
        "name": "Ho Man Tin",
        "distance_km": 0.8509269636,
        "line": "Kwun Tong",
        "color": "#1C934A"
      }
    ]
  },
```

Fig 3.1 Example of one station entry

This format allows the network to be modeled as an undirected, unweighted graph where the stations are nodes and direct connections are edges.

B. Preprocessing

The preprocessing step involves building a graph from the JSON data so that it can be used by the search algorithm. This step uses the Node class and functions in station_loader.py.

The Node class is used to model each station in the network. It contains:

- val: Stores the station's name.
- neighbours: Stores a list of Node objects representing the station's direct connections.

```
class Node:
    def __init__(self, val=0, neighbours=None):
        self.val = val
        self.neighbours = []

    def add_neighbour(self, neighbour_node):
        self.neighbours.append(neighbour_node)

    def remove_neighbour(self, neighbour_node):
        self.neighbours.remove(neighbour_node)

    def __str__(self):
        return f"Value: {self.val}, neighbours: {self.neighbours}"
```

Fig 3.2 Node class structure

In the preprocessing, station_loader.py is responsible for reading the JSON file and building the graph. The main functions in this file include:

- load_json_from_data_folder(json_filename): Reads the JSON file from the data folder. It handles errors such as file not found, invalid format, or failure to parse the data.
- Load_station_nodes(json_filename): Converts the JSON data into a list of Node objects. It first

creates a Node for each station and stores these in a dictionary with station names as keys. Then, It then links the stations by populating their neighbours lists according to the connections specified in the JSON data.

During this process, each station becomes aware of its neighbours, which allows efficient traversal during the search. The use of a dictionary for mapping station names helps ensure that stations can be accessed and linked quickly during preprocessing.

```python
import json
import sys
from pathlib import Path

# Add path directory
current_script_path = Path(__file__).resolve()
project_src_directory = current_script_path.parent.parent  # src folder
sys.path.append(str(project_src_directory))

# import from absolute path
from models.node import Node

def load_json_from_data_folder(json_filename):
    try:
        # Get absolute path of current file
        current_script_path = Path(__file__).resolve()

        # Get main project folder directory
        project_folder_directory = current_script_path.parent.parent.parent

        # Get data folder directory
        json_file_path = project_folder_directory /'data'/ json_filename

        with open(json_file_path, 'r') as f:
            data = json.load(f)

        return data

    except FileNotFoundError:
        print(f"Error: JSON file not found in {json_file_path}")
        return None
    except json.JSONDecodeError:
        print(f"Error: Failed to decode JSON file: {json_file_path}. Make sure JSON format is valid.")
        return None
    except Exception as e:
        print(f"Error: {e}")
        return None

def load_station_nodes(json_filename):
    station_data = load_json_from_data_folder(json_filename)

    if not isinstance(station_data, list):
        print(f"Error: Expected a list of stations from JSON, but got {type(station_data)}.")
        return []

    station_dict = {}

    for raw_station in station_data:
        if isinstance(raw_station, dict) and 'name' in raw_station:
            station_name = raw_station['name']
            new_station = Node()
            new_station.val = station_name
            new_station.neighbours = []
            station_dict[station_name] = new_station


    for raw_station in station_data:
        if isinstance(raw_station, dict):
            current_station_name = raw_station['name']
            current_station = station_dict[current_station_name]

            if 'neighbours' in raw_station and isinstance(raw_station['neighbours'], list):
                for neighbour_station in raw_station['neighbours']:
                    neighbour_station_name = neighbour_station['name']
                    if neighbour_station_name in station_dict:
                        current_station.neighbours.append(station_dict[neighbour_station_name])
    return list(station_dict.values())
```

Fig 3.3 File loader that converts the data from JSON format into a list of Node objects

C. *Main Algorithm*

The search is carried out using the Breadth-First Search (BFS) algorithm. After the data is loaded and the graph is built, the program prompts the user to enter the starting station and the destination station.

The BFS algorithm proceeds as follows:

- A queue is used to manage stations whose neighbours need to be explored.
- A visited set keeps track of stations that have already been visited to avoid revisiting them.
- A parent dictionary maps each station to the station it was discovered from, which allows the path to be reconstructed after the destination is reached.

The algorithm begins at the starting station and explores all its neighbours before moving on to the neighbours' neighbours, ensuring that the shortest path is found in terms of stops. When the destination is reached, the program uses the parent mapping to trace back the path from the destination to the start.

The output of the program includes:

- The sequence of stations on the shortest path.
- The total number of stops required to reach the destination.

```
Enter JSON file name: hk-metro-nodes.json
Successfully loaded 98 stations.

Input start station: Central

Input end station: Whampoa

Route from Central to Whampoa:
1. Central
2. Admiralty
3. Exhibition Centre
4. Hung Hom
5. Ho Man Tin
6. Whampoa
Total Stations visited: 6
```

Fig 3.4 Output example

This shows the path with the minimum number of stops from the start station to the destination.

```python
from models.node import Node
from collections import deque
from utils.station_loader import load_station_nodes

def bfs_search(start: Node, end: Node):
    visited = set([start])
    queue = deque([start])
    path = []
    stop_ctr = 0
    parent = {start: None}

    while queue:
            current = (queue).popleft()
            if current.val == end.val:
                while current:
                    path.append(current)
                    current = parent.get(current)

                path.reverse()
                stop_ctr = len(path)
                return path, stop_ctr

            else:
                for n in current.neighbours:
                    if n not in visited:
                        visited.add(n)
                        queue.append(n)
                        parent[n] = current

    return path, stop_ctr

def find_station_by_name(stations, name):
    """Find a station in the list by its name."""
    for station in stations:
        if station.val.lower() == name.lower():
            return station
    return None

def main():
    file_name = input("Enter JSON file name: ")
    stations = load_station_nodes(file_name)
    if not stations:
        print("No stations loaded. Please check your JSON file.")
        return

    print(f"Successfully loaded {len(stations)} stations.")

    start_name = input("\nInput start station: ")
    end_name = input("\nInput end station: ")

     # Find station objects by name
    start_station = find_station_by_name(stations, start_name)
    end_station = find_station_by_name(stations, end_name)

    # Validate user input
    if not start_station:
        print(f"Error: Start station '{start_name}' not found.")
        return
    if not end_station:
        print(f"Error: End station '{end_name}' not found.")
        return

    result = bfs_search(start_station, end_station)

    # Display results
    if isinstance(result, tuple) and len(result) == 2:
        path, stop_count = result
        print(f"\nRoute from {start_name} to {end_name}:")
        for i, station in enumerate(path):
            print(f"{i+1}. {station.val}")
        print(f"Total Stations visited: {stop_count}")
    else:
        print("Could not find a route between the specified stations.")



if __name__ == "__main__":
    main()
```

Fig 3.5 route_finder.py, main file with BFS algorithm to find minimum stops between two MTR stations

IV. Testing and Results

To verify the correctness of the program, the shortest paths produced by the implementation were compared with results obtained using the official tool provided by MTR Corporation, the MTR Journey Planner. This online tool provides recommended routes between stations in the MTR system, including the sequence of stations and the total number of stops.

The verification process was done by manually entering the start and end stations in the MTR Journey Planner and recording the suggested route. The output generated by the program was then compared to this official result to ensure that the path found matches the minimum number of stops as recommended by MTR. This approach provides reliable validation because the Journey Planner reflects the actual design of the MTR network.

The first test case is from Ocean Park to Sheung Wan. This case represents a direct journey along one line without transfers. It checks whether the program correctly identifies the shortest path with no unnecessary detours.

```
Enter JSON file name: hk-metro-nodes.json
Successfully loaded 98 stations.

Input start station: Ocean Park

Input end station: Sheung Wan

Route from Ocean Park to Sheung Wan:
1. Ocean Park
2. Admiralty
3. Central
4. Sheung Wan
Total Stations visited: 4
```

Fig 4.1 Test case 1 BFS program result

The result of the first test case from the program is identical to the result from the official route finder.

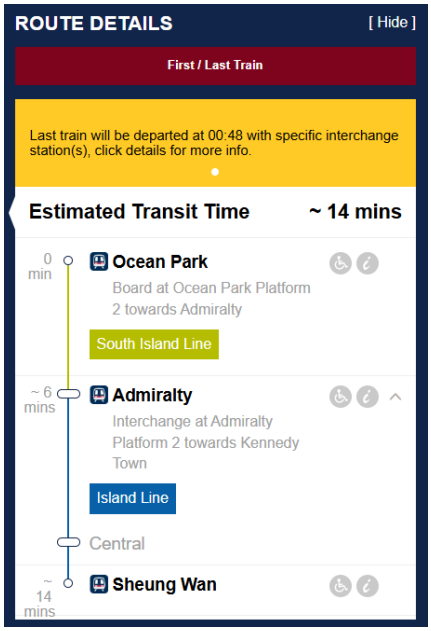

Fig 4.2 Test case 1 official result

The second test case is from Prince Edward to Fortress Hill. This case involves a route that requires changing lines. It tests if the program can handle line transfers and still return the path with the minimum number of stops.

```
Enter JSON file name: hk-metro-nodes.json
Successfully loaded 98 stations.

Input start station: Prince Edward

Input end station: Fortress Hill

Route from Prince Edward to Fortress Hill:
1. Prince Edward
2. Mong Kok
3. Yau Ma Tei
4. Jordan
5. Tsim Sha Tsui
6. Admiralty
7. Wan Chai
8. Causeway Bay
9. Tin Hau
10. Fortress Hill
Total Stations visited: 10
```

Fig 4.3 Test case 2 BFS program result

The result of the second test case from the program is identical to the result from the official route finder.

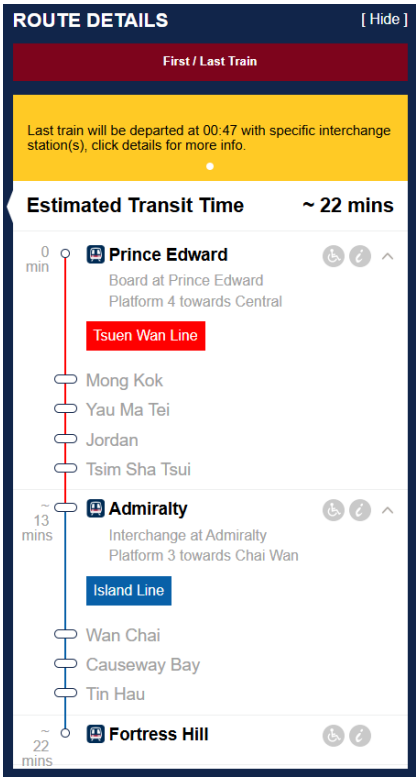

Fig 4.4 Test case 2 official result

The third test case is from Yau Tong to Lai King. This case includes multiple transfers across different lines. It ensures that the program is able to find the correct minimum-stop path even in more complex routing situations.

```
Enter JSON file name: hk-metro-nodes.json
Successfully loaded 98 stations.

Input start station: Yau Tong

Input end station: Lai King

Route from Yau Tong to Lai King:
1. Yau Tong
2. Quarry Bay
3. North Point
4. Fortress Hill
5. Tin Hau
6. Causeway Bay
7. Wan Chai
8. Admiralty
9. Exhibition Centre
10. Hung Hom
11. East Tsim Sha Tsui
12. Austin
13. Nam Cheong
14. Lai King
Total Stations visited: 14
```

Fig 4.5 Test case 3 BFS program result

The result of the third test case from the program differs from the result in the official website.

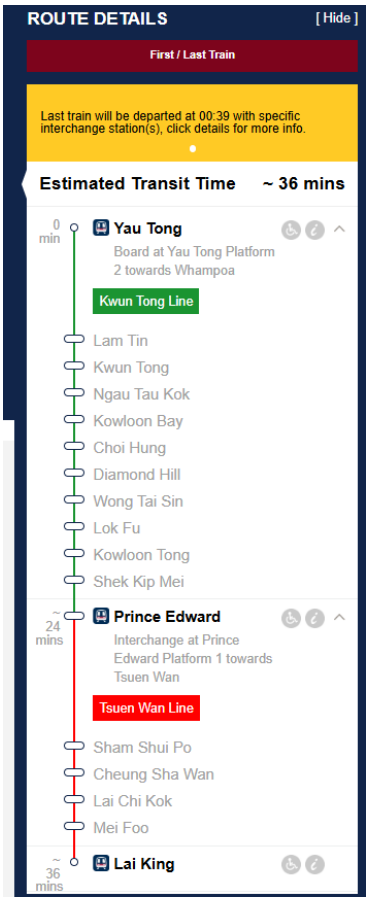

Fig 4.5 Test case 3 official result

The testing shows that the program produces correct results for most cases when compared with the official MTR Journey Planner. For the first test case, from Ocean Park to Sheung Wan, the program correctly identifies the shortest path along one line, with no unnecessary detours. The path found by the

program is identical to the one recommended by the official tool.

In the second test case, from Prince Edward to Fortress Hill, the program is able to handle the route involving a line transfer and still finds the path with the minimum number of stops. The output matches the official result both in terms of the sequence of stations and the total stops.

In the third test case, from Yau Tong to Lai King, the program finds a valid path, but it differs from the official recommendation. The route found by the program still provides a correct connection between the two stations, but it may not be the path with the fewest transfers or may take a different transfer point compared to the MTR Journey Planner. This suggests that while the program guarantees the minimum number of stops using BFS, it does not account for factors such as line preferences or transfer convenience, which might be considered by the official tool.

V. Conclusion

This study demonstrates the use of the Breadth-First Search (BFS) algorithm to solve the problem of minimum-stop routing in an MTR network. By representing the stations and connections as an unweighted, undirected graph, the program is able to find the shortest path between two stations in terms of the number of stops. The testing results confirm that the program performs correctly in most cases, producing paths that match the official recommendations of the MTR Journey Planner. However, in certain complex routes, the program may produce a path that differs slightly due to the nature of BFS, which focuses purely on stop count without considering other practical factors like line preferences or transfer convenience. Overall, the BFS approach provides an effective solution for minimum-stop pathfinding in transportation networks.

VI. Appendix

The complete Hong Kong MTR minimum stop finder using Breadth First Search algorithm can be found below.

https://github.com/mahesa005/HK-MTR-RouteFinder

VII. Acknowledgement

All praise and gratitude are offered to the presence of the Almighty God, Allah Subhanahu wa Ta'ala, who has given the author the opportunity to complete the paper entitled *"Minimum-Stop Routing in the MTR Network Using Breadth-First Search"*. In addition, the author would like to express his deepest gratitude to the lecturer in charge of the Discrete Mathematics course, Rinaldi Munir, M.T., for the lessons and motivation that have been given during the lecture.

References

[1] MTR Corporation, "MTR System Map." [Online]. Available: https://www.mtr.com.hk/en/customer/services/system_map.html. [Accessed: 26 May 2025].

[2] Kaggle, "Hong Kong MTR Network Dataset." [Online]. Available: https://www.kaggle.com/datasets/d1om3d3s/hong-kong-mtr-network. [Accessed: 26 May 2025].

[3] MTR Corporation, "MTR Journey Planner." [Online]. Available: https://www.mtr.com.hk/en/customer/jp/index.php. [Accessed: 20 June 2025].

[4] R. Munir, "Graf Bagian 1" IF1220 Matematika Diskrit. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Accessed: 18 June 2025].

[5] R. Munir, "Breadth First Search (BFS) and Depth First Search (DFS) 2025 Bagian 1" IF1220 Matematika Diskrit. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf. [Accessed: 20 June 2025].

Statement of Originality

I hereby declare that this paper is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, 20 Juni 2025

Mahesa Fadhillah Andre - 13523140