2 Shortest Path Finding for a Given Map

2.1.0 Introduction

2.1.1 Overview of problem

Finding the shortest path between two points on a map is a computational problem in graph theory that focuses on finding the shortest path or routes between two nodes in a given graph, with the path's "shortness" often determined by the sum of weights of its constituent edges. This a problem that exists in various forms in everyday life, such as the one encountered while using GPS navigation systems to identify the shortest route between two geographical points.

In a more applied context, consider an example of a city map where the map is abstracted into a graph, with locations as nodes, and paths between these locations as edges. The weight of an edge could represent various attributes such as distance or time. The objective is to find the shortest path from one source location to another destination location. The shortest path between two given locations would be the path that results in the least total cost in terms of distance or travel time.

In order to solve this problem, we implement Dijkstra's algorithm in our program and apply it as a GPS navigation system. Dijkstra's algorithm finds the shortest routes between nodes in a weighted graph such as road networks or similar structures (Dijkstra’s algorithm, 2023). Dijkstra's algorithm works by constructing a tree of shortest paths from the starting vertex (the source) to all other graph nodes (Abiy et al., n.d). This tree then provides the shortest possible path from the source to any given node.

The first stage in Dijkstra's algorithm is initialization. The algorithm starts at the source node. The initial distances are set to zero for the source node and to infinity for all the other nodes (Nagaraj, 2023). Additionally, an unvisited nodes set is established, which initially includes all the nodes in the graph.

The algorithm then repeatedly visits the unvisited node that currently has the smallest known distance from the source. Suppose this node is 'A'. Once at 'A', the algorithm checks all of 'A's adjacent nodes. For example, its adjacent node 'B', the algorithm determines if the path cost through 'A' to 'B' and record it to a table (cheapest path cost table). If there is no previous B record, then this path cost is directly recorded in the table. If the new path is shorter, the shortest distance to 'B' is updated in the table. Since B is reached through A, A is recorded into the previous_cheapest_stopover_node table. Once all of 'A's adjacent nodes have been checked, 'A' is marked as visited. This means that 'A's distance from the source is final and will not change. It is worth noting that a node is only visited once by the algorithm. Once marked as visited, a node is not revisited.

These steps are repeated, selecting the unvisited node with the smallest known distance, visiting that node, checking its neighbours, updating the tables and marking the node as visited until all nodes in the graph have been visited. If the target node is found, and the algorithm is sure there is no possibility of a shorter path (because all the remaining unvisited nodes have greater distances), it can terminate early.

By the end, Dijkstra's algorithm provides the shortest distance from the source node to every other node in the graph.

2.1.2 Choice of Data Structures

The data structures for this problem include classes, dictionaries, heaps, and lists. They represent the map, and store and manipulate information about routes and distances.

Lists are used in our program. They store a sequence of values in a specific order. In Python, lists are dynamic, meaning they can be resized as needed. Lists is used in our problem to store the state space to maintain a list of visited and unvisited nodes. Dynamic lists are suited since these list of values are constantly altered.

A heap is a special type of binary tree data structure where the parent node is compared with its children and arranged accordingly ("Heap data structure", n.d.). A heap could be used to keep track of the unvisited node with the shortest distance from the source node at each step in the algorithm. This data structure allows us to efficiently get the next node to visit, which is the one with the smallest distance.

Dictionaries are data structures that store data in key-value pairs (Python dictionary, 2023). For example, a dictionary is used to store a node's adjacent node, where the keys represents the adjacent nodes and the values represent the pathcost of the edge between the current node and the adjacent node. Dictionaries provide a quick way to access data based on a key, such as looking up the current shortest distance to each node.

Class allows us to define data types with their own methods and attributes. In our case, we could define a class for node. A 'Node' class could contain information about the node such as name, connections, and weights.

2.1.3 Tools and Programming Language Features

To effectively carry out this task, we need to make use of appropriate tools, libraries, and language features that is provided. Here's an introduction to the libraries and tools that we had used:

OSMnx is a Python package that provides spatial data from OpenStreetMap and models, projects, visualizes, and analyzes real-world street networks ("Geoff Boeing", n.d.). We will use OSMnx to obtain the real-world map data required for our simulation. This data will then be represented as a graph and used as input for our implementation of Dijkstra's algorithm.

Geopy is another Python library that simplifies the task of locating the coordinates of addresses, cities, countries, and landmarks worldwide using third-party geocoders and other data sources (GeoPy Contributors, 2018). Nominatim module from geopy, which uses the OpenStreetMap's API to find the latitude and longitude of given location names. This will allow us to accurately find and plot points on our map.

Visual Studio Code (VS Code) is used as the Integrated Development Environment (IDE) for our program. VS Code supports Python and provides a rich set of features like syntax highlighting, debugging support, and an integrated terminal, all of which make the process of writing, testing, and debugging Python code much simpler and more efficient.

This project implements **programming language features** using **Python** as a programming language.

Classes and objects are created using **object-oriented programming**. For example, the ‘Node’ class represents nodes on the map, and the Dijkstra function implements Dijkstra's algorithm using objects of the ‘Node’ class.

Various **data structures** are used to organize and manage data efficiently. Examples include dictionaries to store graph information, priority queues (heap) to implement the priority-based exploration in Dijkstra's algorithm, and lists to maintain the visited and unvisited nodes as mentioned in the previous section.

Self-defined functions are used extensively throughout the project to encapsulate specific blocks of code with specific tasks. Functions like ‘dijkstra()’, ‘get_state_space()’, ‘create_graph_from_list()’, and others help modularize the code for better organization and readability.

The project includes **visualisation** features using libraries like ‘osmnx’ to plot and display the map, shortest path, or other relevant data.

By employing these programming language features, the project is organized, efficient, and capable of achieving the goal of finding the shortest path between two points on a map using Dijkstra's algorithm while considering various map data and constraints.

2.2.0 Implementation

Before delving into the actual code, we would like to provide you with a brief overview of how we developed the program to find the shortest path between two nodes in a real-world street network. For Dijkstra's Algorithm to determine the shortest path, it needs the provision of a state_space. In our application, we employed the OSMnx library to give us the network (state_space) of every node within a certain boundary on the map. Having obtained the state_space, we then deployed our Dijkstra Algorithm to identify the most efficient route between the defined starting and ending nodes. The program allows users to input the source, destination, type of network (road, walkways, bicycle lanes), and the optimiser (prioritizing distance or travel time). The resulting shortest route is displayed as a visualization using the geopy library, which returns a folium Map object. Let’s begin understanding the implementation of the code.

```python
import heapq
#find graph edge travel time. download geospatila data from OpenStreetMap and model, project, visualize and analysie real-world street networks
import osmnx as ox
#geopy is used to find coordinates(langitude, longtitude) given the location names
from geopy.geocoders import Nominatim
```

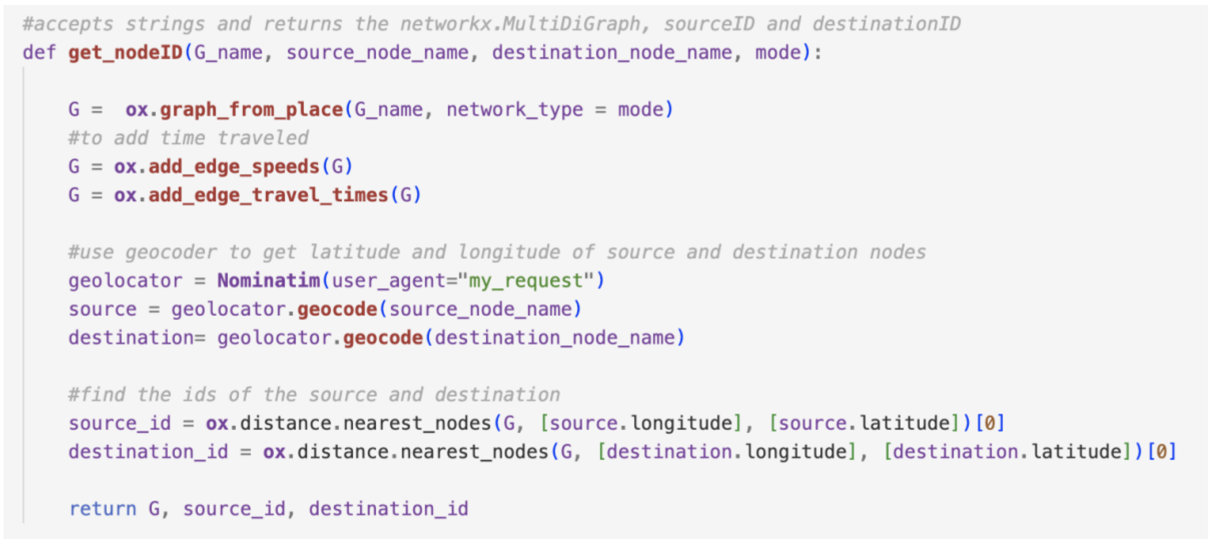
Our program uses libraries as shown above.

```python
#node to represent each node on the graph
class Node:
    def __init__(self, name, wsf = 0):
        self.name = name
        self.adjacentNodes = {} #stores pathcost to adjacent nodes
        self.wsf = wsf #calculates the path cost from the source node

    def addAdjacentNodes(self, node, pathCost):
        self.adjacentNodes[node] = pathCost

    def __lt__(self, node2): #heap will organise the queue based on wsf
        return self.wsf < node2.wsf
```

Node class. The node class represents every node on the map. Each node contains a name, a dictionary of its adjacent nodes and their path costs, and a "wsf" attribute. This attribute represents weight so far and tracks the source-to-node path with the lowest cost.

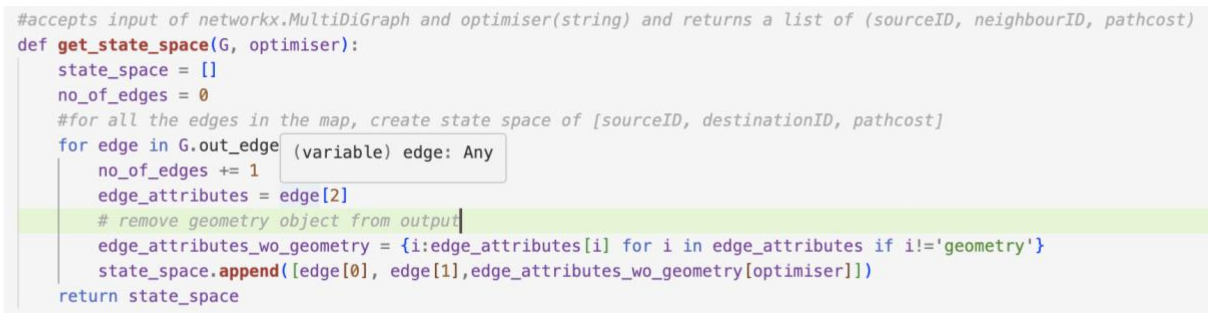
```python
#function to get the user input and print out instructions
def get_user_input():
    G_name = input("Boundary of search(e.g., 'Subang Jaya, Malaysia'): ")
    source_node_name = input("Source location(e.g., 'Sunway University'): ")
    destination_node_name = input("Destination location(e.g., 'Sunway Pyramid'): ")
    mode = input("Travel mode('drive', 'bike', 'walk'): ")
    optimiser = input("Optimiser('length', 'travel_time'): ")
    print("")
    return G_name, source_node_name, destination_node_name, mode, optimiser
```

get_user_input() function. This function prompts the user to input the boundary of search, source location, destination location, mode of transportation such as 'walk', 'bike', 'drive', and the optimiser(length or travel time). It returns these values.



```
#accepts strings and returns the networkx.MultiDiGraph, sourceID and destinationID
def get_nodeID(G_name, source_node_name, destination_node_name, mode):

    G =  ox.graph_from_place(G_name, network_type = mode)
    #to add time traveled
    G = ox.add_edge_speeds(G)
    G = ox.add_edge_travel_times(G)

    #use geocoder to get latitude and longitude of source and destination nodes
    geolocator = Nominatim(user_agent="my_request")
    source = geolocator.geocode(source_node_name)
    destination= geolocator.geocode(destination_node_name)

    #find the ids of the source and destination
    source_id = ox.distance.nearest_nodes(G, [source.longitude], [source.latitude])[0]
    destination_id = ox.distance.nearest_nodes(G, [destination.longitude], [destination.latitude])[0]

    return G, source_id, destination_id
```

get_nodeID() function. This function receives the boundary name, source and destination, and method of transportation. The OSMnx package produces a street network graph for the given boundary and mode using the OSMnx library. Then it uses the geopy library to locate the latitude and longitude of the source and destination location (these may not necessarily be nodes in OSMnx). Finally, it identifies the closest nodes to these coordinates(these are the official nodes on the map) in the graph and returns their IDs. OSMnx identifies nodes through distinct IDs thus rather than using names like how we usually do in labs, we use ID(integers) throughout this program instead. This function returns a networkx multigraph of the boundary, the sourceID and the destinationID.



```
#accepts input of networkx.MultiDiGraph and optimiser(string) and returns a list of (sourceID, neighbourID, pathcost)
def get_state_space(G, optimiser):
    state_space = []
    no_of_edges = 0
    #for all the edges in the map, create state space of [sourceID, destinationID, pathcost]
    for edge in G.out_edge
        no_of_edges += 1
        edge_attributes = edge[2]
        # remove geometry object from output
        edge_attributes_wo_geometry = {i:edge_attributes[i] for i in edge_attributes if i!='geometry'}
        state_space.append([edge[0], edge[1],edge_attributes_wo_geometry[optimiser]])
    return state_space
```

get_state_space() function. The function accepts two inputs: the network multigraph and an optimizer, which determines the attribute used to assess the path cost(distance/time). It creates a list of all edges in the network graph. Each edge is denoted as a tuple consisting of the source node ID, the adjacent node ID, and the path cost which is how our state space is represent [sourceID, destinationID, pathCost]. This function returns the state space in a useful form that can be used by our algorithm.

```python
#accepts a list of [sourceID, nodeID, pathcost] and returns a map that maps nodeID to the nodeObject
def create_graph_from_list(state_space):
    node_map = {} #list of nodes
    for source_id, destination_id, path_cost in state_space:
        if source_id not in node_map:
            node_map[source_id] = Node(source_id)
        if destination_id not in node_map:
            node_map[destination_id] = Node(destination_id)
        node_map[source_id].addAdjacentNodes(node_map[destination_id], path_cost)
    return node_map
```

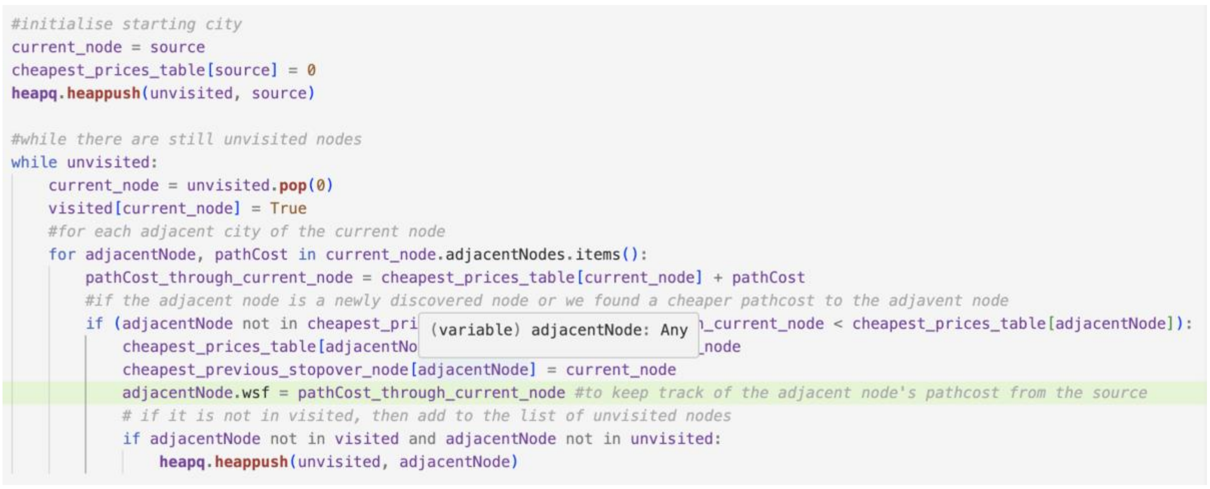
create_graph_from_list() function. Using the state space list, it generates a map by creating node objects from the nodeIDs and connects all nodes in the network by initialising neighbour nodes based on the state_spaces. This function returns a map containing all nodeIDs pointing towards to their nodeObjects.

```python
#takes in user's input and process it for suitable use and call the Dijkstra Algorithm, this returns the n
def run_djikstra(G_name, source_node_name, destination_node_name, mode, optimiser):
    G, source_id, destination_id = get_nodeID(G_name, source_node_name, destination_node_name, mode)
    #from the map, generate the state space of all connecting nodes on the given map
    state_space = get_state_space(G, optimiser)
    #initialise all nodes and link them
    node_map = create_graph_from_list(state_space)
    shortest_path = dijkstra(node_map[source_id], node_map[destination_id])
    return G, shortest_path
```

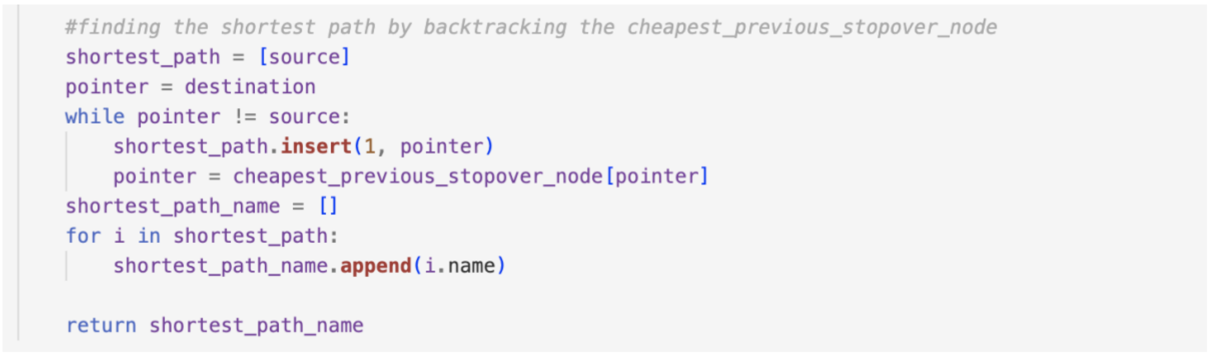
run_djikstra() function. The above function is a more abstract level of running the program, so that user can call this function right away without needing to know the specifics. It first calls get_nodeID() to obtain the graph, sourceID and destinationID, produce the state space(get_state_space), create nodes and linking them to produce a useful graph(create_graph_from_list) for our algorithm and run our Dijkstra's algorithm for execution by feeding in the sourceNode and the destinationNode(dijkstra). The function returns the multidigraph and the shortest path.

```python
#Dijkstra algorithm
def dijkstra(source, destination):
    #stores cheapest known price from starting to all other known destinations(uses hashtables)
    cheapest_prices_table = {}
    #stores cheapeast previous stopover city to reach the cheapest price (hashtable)
    cheapest_previous_stopover_node = {}
    #list of visited nodes
    visited = {}
    #list of unvisited nodes
    unvisited = []
    heapq.heapify(unvisited)
```

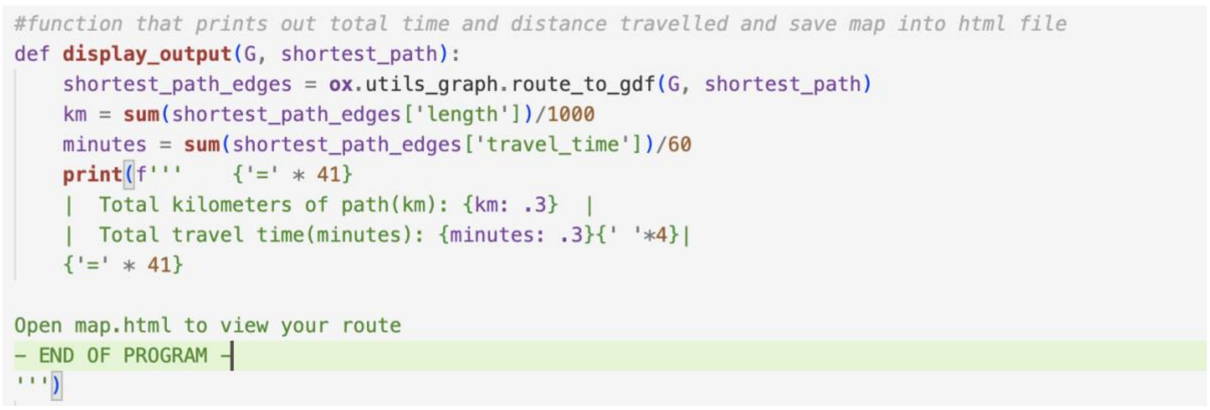
dijkstra() function. The Dijkstra's algorithm calculates the shortest path of every node on the map and generates the shortest path for graph given the source and destination node. Our algorithm consists hash tables to store the the cheapest cost to every node(node -> cost), the cheapest previous stopover node(node -> node), visited nodes(node -> true/false), and finally a heapq of unvisited nodes.


```
#initialise starting city
current_node = source
cheapest_prices_table[source] = 0
heapq.heappush(unvisited, source)

#while there are still unvisited nodes
while unvisited:
    current_node = unvisited.pop(0)
    visited[current_node] = True
    #for each adjacent city of the current node
    for adjacentNode, pathCost in current_node.adjacentNodes.items():
        pathCost_through_current_node = cheapest_prices_table[current_node] + pathCost
        #if the adjacent node is a newly discovered node or we found a cheaper pathcost to the adjavent node
        if (adjacentNode not in cheapest_pri (variable) adjacentNode: Any _current_node < cheapest_prices_table[adjacentNode]):
            cheapest_prices_table[adjacentNo                              _node
            cheapest_previous_stopover_node[adjacentNode] = current_node
            adjacentNode.wsf = pathCost_through_current_node #to keep track of the adjacent node's pathcost from the source
            # if it is not in visited, then add to the list of unvisited nodes
            if adjacentNode not in visited and adjacentNode not in unvisited:
                heapq.heappush(unvisited, adjacentNode)
```

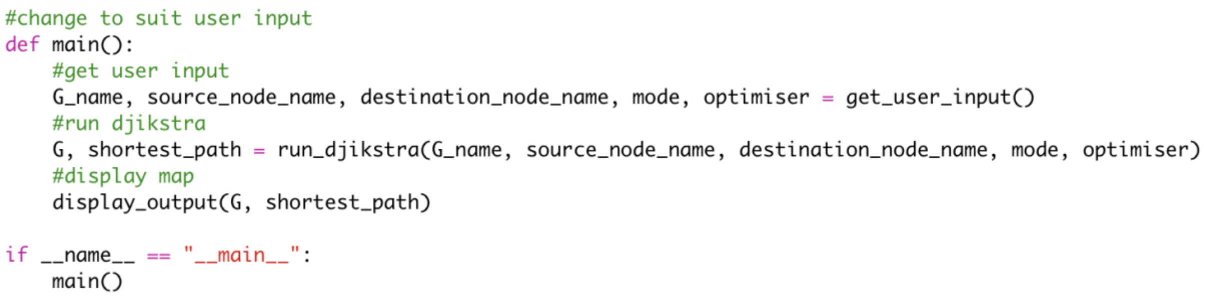
The source node is first added to unvisited. While there are nodes in the unvisited list, the first node in the unvisited list(shortest cost) will be popped off to be visited. For each of its neighbour, it calculates the cost to reach that neighbour from the source node and adds it to the unvisited list (given that it has not been visited and not in already in the unvisited list). The cost recorded for each node will be the shortest, it will only be recorded if the node has not yet been recorded or if the current cost is lower than the cost recorded previously, the cheapest_previous_stopover_node hash table will also be updated accordingly.


```
    #finding the shortest path by backtracking the cheapest_previous_stopover_node
    shortest_path = [source]
    pointer = destination
    while pointer != source:
        shortest_path.insert(1, pointer)
        pointer = cheapest_previous_stopover_node[pointer]
    shortest_path_name = []
    for i in shortest_path:
        shortest_path_name.append(i.name)

    return shortest_path_name
```

When there are no longer any unvisited nodes, it exits the while loop and proceed to backtrack from the destination node, basing off the cheapest previous stopover node list the until it finds the source node, the IDs of the nodes are retrieved and they are returned by the function.


```
#function that prints out total time and distance travelled and save map into html file
def display_output(G, shortest_path):
    shortest_path_edges = ox.utils_graph.route_to_gdf(G, shortest_path)
    km = sum(shortest_path_edges['length'])/1000
    minutes = sum(shortest_path_edges['travel_time'])/60
    print(f'''    {'=' * 41}
    |  Total kilometers of path(km): {km: .3}  |
    |  Total travel time(minutes): {minutes: .3}{' '*4}|
    {'=' * 41}

Open map.html to view your route
- END OF PROGRAM -
''')
```

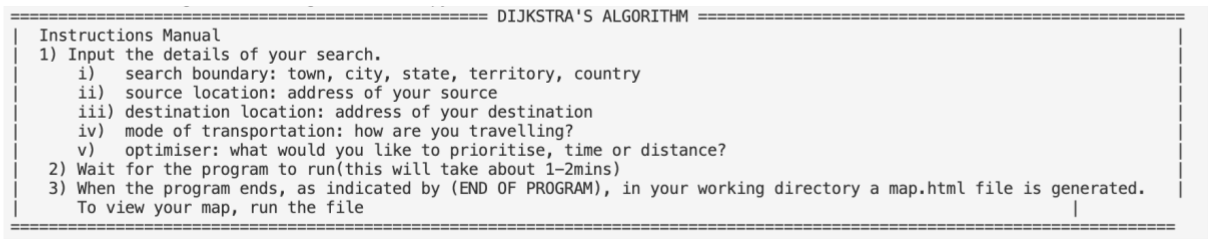
display_output() function. The inputs of the function consist of the graph and the shortest path. It converts the shortest path into a GeoDataFrame, computes the overall distance and travel time of the path, and displays these values. It then uses the GeoDataFrame's explore function to plot out the graph and the shortest route and then saves it into an HTML file. For simplicity in demonstrating our algorithm, we did not include printing out directions from source to destination, however this is definitely implementable using functions by the libraries.



```python
#change to suit user input
def main():
    #get user input
    G_name, source_node_name, destination_node_name, mode, optimiser = get_user_input()
    #run djikstra
    G, shortest_path = run_djikstra(G_name, source_node_name, destination_node_name, mode, optimiser)
    #display map
    display_output(G, shortest_path)

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

main() function.It is the entry point of the program. It gets the user input, runs the Dijkstra's algorithm, and displays the output.

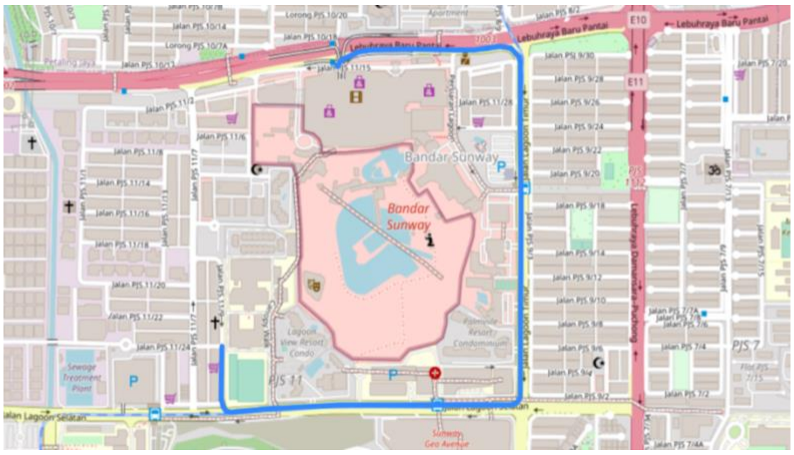
To run this program, there are a few steps. The instruction manual will printed when the program starts to run:



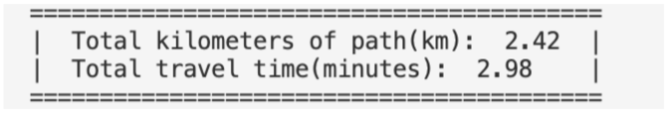
```
================================================= DIJKSTRA'S ALGORITHM =================================================
|  Instructions Manual                                                                                                  |
|  1) Input the details of your search.                                                                                 |
|      i)   search boundary: town, city, state, territory, country                                                      |
|      ii)  source location: address of your source                                                                     |
|      iii) destination location: address of your destination                                                           |
|      iv)  mode of transportation: how are you travelling?                                                             |
|      v)   optimiser: what would you like to prioritise, time or distance?                                             |
|   2) Wait for the program to run(this will take about 1-2mins)                                                        |
|   3) When the program ends, as indicated by (END OF PROGRAM), in your working directory a map.html file is generated.  |
|      To view your map, run the file                                                                         |
========================================================================================================================
```

For example,

```
Boundary of search(e.g., 'Subang Jaya, Malaysia'): Subang Jaya
Source location(e.g., 'Sunway University'): Sunway University
Destination location(e.g., 'Sunway Pyramid'): Sunway Pyramid
Travel mode('drive', 'bike', 'walk'): drive
Optimiser('length', 'travel_time'): length
```



The program will then determine the shortest path between the source and destination node based on the specified optimizer, in this example, the ‘length’. It will then display the total distance and travel time of the path, as well as a map illustrating the highlighted route.

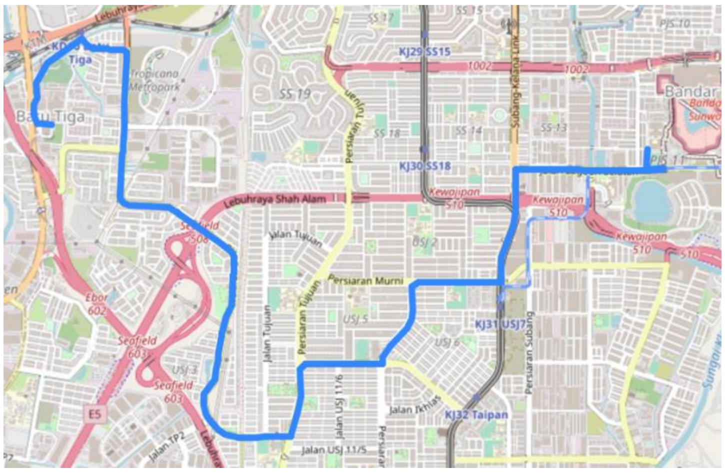


```
==========================================
|  Total kilometers of path(km):  2.42   |
|  Total travel time(minutes):  2.98     |
==========================================
```

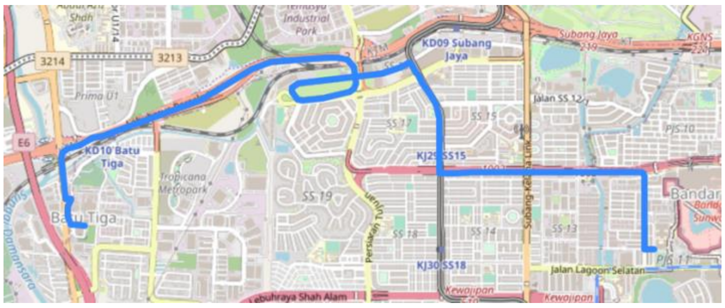
2.3.0 Limitations

The project of implementing shortest path finding using Dijkstra's algorithm for a given map has some limitations that should be taken into account.

One significant limitation we realised is the need to specify a boundary for search due to the consideration of the **graph size and complexity**. Real-world maps can be extensive and complex, resulting in creation of large graphs with many nodes and edges. Handling such large graph structures can lead to increased computational time and memory usage. In our scenario, we had specified a boundary such as Subang Jaya to get the graph edges from OSMnx. While this saves time, it means that our route options might be limited to only the ones within this territory and neglecting the paths that are outside the boundary but shorter. We tested this out by expanding the boundary to Selangor, this process took much longer as the graph size increased tremendously(5+ minutes). Below is an example of running the same source and destination but using different boundaries.



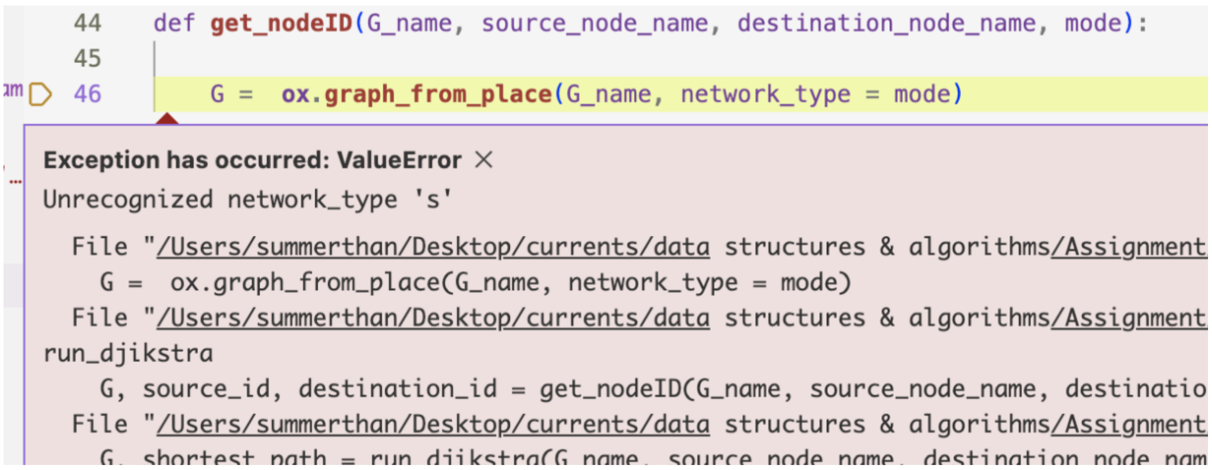
Within the boundary of Subang Jaya



Within the boundary of Selangor

The project also assumes the use of **static map data** during the shortest path finding algorithm. In dynamic environments that need real-time navigation, the map data may vary due to factors such as traffic updates, road closures, or construction. These changes have the potential to affect the accuracy of calculated shortest paths. This is very prominent in calculating the time for our program. The time needed to travel is calculated solely through travelling across the path using the maximum speed possible provided by OSMnx's path features thus it is highly inaccurate as it does not consider enviromental factors. Another similar problem we encountered was the inaccuracy of certain networks. In the program, we are free to choose the type of networks we would like to work with (driving, biking and walking). While testing for biking and walking, we found some inaccuracy especially in areas like Malaysia where there lacks very accessible biking and walking routes. This program also require user to input accurate names, for some of the places we tried, the library was unable to find the names of the input. A good reference point for input names will be this website: https://www.openstreetmap.org/#map=13/3.0734/101.5900, and the user is also suggested to input full addresses during input.

Lack of exception handling. To further elaborate our above point, while the problem above is partially resulted from the library, our program also do not handle exceptions thus the program will not run if the input is not what as expected. The function of exception handling would require string manipulation and a few other function making the code very tedious, thus we decided to omit it and focus entirely on optimising the core of the problem.



```
def get_nodeID(G_name, source_node_name, destination_node_name, mode):

    G =  ox.graph_from_place(G_name, network_type = mode)
```

```
Exception has occurred: ValueError ×
Unrecognized network_type 's'
  File "/Users/summerthan/Desktop/currents/data structures & algorithms/Assignment
    G =  ox.graph_from_place(G_name, network_type = mode)
  File "/Users/summerthan/Desktop/currents/data structures & algorithms/Assignment
run_djikstra
    G, source_id, destination_id = get_nodeID(G_name, source_node_name, destinatio
  File "/Users/summerthan/Desktop/currents/data structures & algorithms/Assignment
    G, shortest_path = run_djikstra(G_name, source_node_name, destination_node_nam
```

Irregular geographies, such as islands or disconnected regions, may pose difficulties in finding optimal route between certain source-destination pairings on the map, since we do not have cross-sea travelling options. In such cases, disconnected graph components may prevent the algorithm from suggesting complete routes / even suggesting routes at all.

Finally, Dijkstra's algorithm operates on **non-negative edge weights**. If the map data includes negative edge weights, the algorithm might be unable to determine the accurate shortest path.