HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MATHEMATICAL FOUNDATION FOR COMPUTER SCIENCE

Assignment 2

THE K SHORTEST PATH ROUTING PROBLEM

Lecturer: **TS. Trần Tuấn Anh**

Students: **2070423 - Nguyễn Thị Ngọc Phim**
1910006 - Nguyễn Ngọc Thái An
2170554 - Bạch Mai Tuyết Như
1911530 - Nguyễn Kim Lộc
2170551 - Vũ Thành Nhân

HO CHI MINH CITY - MAY 2023

Table of contents

Table of images

I. Introduction

Finding the shortest path is a classic problem in the field of Computer Science.Currently, there are numerous methods and algorithms that can help us solve this problem, such as the Bellman-Ford algorithm, Dijkstra's algorithm, the Floyd-Warshall algorithm, and more.

One famous tool for finding directions that all of us have used is Google Maps. Google Maps is a digital mapping service provided by Google, aiming to replace traditional paper maps in the context of the rapidly developing Internet.

Today, Google Maps is widely used and has become an indispensable utility in our daily lives. Through Google Maps, users can easily search for addresses, look up directions, measure distances between locations, or make easy customizations to save favorite places, pinpoint locations, and more.

For the project, the team will utilize Google Maps as a tool to determine the distances between various locations. This data will be used as input to solve the problem of finding the shortest path.

II. Algorithms

1. Bellman Algorithm

1.1 Introduction

The Bellman algorithm, also known as the Bellman-Ford algorithm, is a popular method used to find the shortest path in a weighted graph. It is named after mathematicians Richard Bellman and Lester Ford. The algorithm is capable of handling graphs with negative edge weights, unlike some other algorithms like Dijkstra's algorithm. It is commonly used in network routing protocols and can also be applied to other scenarios, such as finding the shortest path in a transportation network.

1.2 Pseudo Code

Algorithm 1 Bellman-Ford algorithm

Graph G, Source vertex s distance and predecessor arrays

foreach vertex v in G **do**
  if $v \neq s$ **then**
    distance $[v] \leftarrow$ infinity
  end
  predecessor $[v] \leftarrow$ undefined
end

for $i \leftarrow 1$ **to** $|V| - 1$ **do**
  foreach edge (u, v) with weight w in G **do**
    if distance $[u] + w <$ distance $[v]$ **then**
      distance $[v] \leftarrow$ distance $[u] + w$ predecessor $[v] \leftarrow u$
    end
  end
end

foreach edge (u, v) with weight w in G **do**
  if distance $[u] + w <$ distance $[v]$ **then**
    return "Graph contains a negative-weight cycle"
  end
end

return distance, predecessor

1.3 Pros - Cons

The Bellman algorithm, also known as the Bellman-Ford algorithm, has several advantages. Firstly, it is capable of handling graphs with negative edge weights, which sets it apart from certain other algorithms like Dijkstra's algorithm. This makes it particularly useful in scenarios where negative weights are involved.

Additionally, the Bellman algorithm guarantees to find the shortest path in a graph if it exists. This property makes it a reliable choice for finding optimal routes or paths in various applications. Furthermore, it can be applied to different types of graphs, including directed and undirected graphs, making it versatile for a wide range of problems.

However, the Bellman algorithm has some drawbacks. Its time complexity is $O(|V| * |E|)$, where $|V|$ represents the number of vertices and $|E|$ represents the number of edges in the graph. This time complexity can be a disadvantage when dealing with large graphs, as it may not be as efficient as other algorithms such as Dijkstra's algorithm, which has a better time complexity for certain graph structures.

Another limitation of the Bellman algorithm is that if there is a negative cycle in the graph, the algorithm may not terminate or produce correct results. However, it is still able to identify the presence of negative cycles, which can be valuable information for further analysis or adjustments in the graph.

In summary, the Bellman algorithm offers advantages such as handling negative edge weights, guaranteeing the shortest path, and being applicable to various graph types. However, its time complexity and vulnerability to negative cycles should be considered when choosing an algorithm for specific graph-related problems.

2. The Floyd algorithm

2.1 Introduction

The Floyd or Floyd-Warshall algorithm is the shortest path algorithm for graphs. The Floyd algorithm computes the shortest distances between every pair of vertices in the input graph. It is an example of dynamic programming approach.

The Floyd algorithm is best suited for dense graphs, its complexity depends only on the number of vertices in the given graph.

Imagine that you have a graph which has 5 vertices A, B, C, D, E. You know a few roads that connect some of vertices and you know the lenghts of those roads. The Floyd algorithm will give you the optimal route between each pair of vertices.

Floyd-Warshall is a valuable tool in networking, akin to solving the shortest path problem. It is particularly adept at handling multiple stops on a route, as it can compute the shortest paths between all pertinent nodes. Essentially, one iteration of Floyd can provide all the necessary information about a static network to optimize most types of paths. Additionally, it is helpful for computing matrix inversions.

2.2 Pseudo code

Algorithm 2 Floyd algorithm

```
let dist be a |V| × |V| array of minimum distances initialized to ∞
    (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

2.3 Pros - Cons

Pros:

- Can handle multiple stops on a route and compute shortest paths between all pertinent nodes (all-pairs shortest path problem).
- Provides all necessary information about a static network to optimize most types of paths.
- Helpful for computing matrix inversions.

Cons:

- Time complexity is $O(n^3)$, can be slow for large networks.
- Not suitable for dynamic networks where the topology changes frequently.
- Not suitable for networks with negative edge weights.

3. Yen's algorithm

3.1 Introduction

Yen's algorithm, also known as Yen's K-shortest paths algorithm, is an algorithm used to find the K shortest paths between two nodes in a graph. It was developed by Jin Y. Yen in 1971 and is commonly used in the field of graph theory and network analysis.

The algorithm is an extension of Dijkstra's algorithm, which finds the shortest path between two nodes in a graph. Yen's algorithm builds upon this idea by iteratively finding the next shortest path after removing the previous shortest path from consideration. It repeatedly calculates the shortest path and removes it until K shortest paths are found.

3.2 Pseudo code

Algorithm 3 Yen's Algorithm

```
function yenAlgorithm(graph, source, destination, K):
    // Initialization
    shortestPaths = []  // List to store the K shortest paths
    candidates = []  // List to store potential candidate paths
    // Step 1: Calculate the shortest path using any shortest path algorithm
    shortestPath = calculateShortestPath(graph, source, destination)
    shortestPaths.append(shortestPath)
    // Step 2: Iteratively find the next K-1 shortest paths
    for k = 2 to K:
        // Step 3: Generate potential candidate paths
        for i = 1 to length(shortestPaths[k-1]) - 1:
            spurNode = shortestPaths[k-1][i]
            rootPath = shortestPaths[k-1][1:i]
            // Step 4: Remove edges
            for each path in shortestPaths:
                if path starts with rootPath:
                    remove edge(path[i], path[i+1]) from graph
            // Step 5: Calculate spur path
            spurPath = calculateShortestPath(graph, spurNode, destination)
            if spurPath exists:
                // Step 6: Combine root and spur paths
                totalPath = concatenate(rootPath, spurPath)
                candidates.append(totalPath)
            // Step 7: Restore removed edges
            for each path in shortestPaths:
                if path starts with rootPath:
                    restore edge(path[i], path[i+1]) to graph
        if candidates is empty:
            break
        // Step 8: Select the best candidate path
        candidates.sort()  // Sort candidate paths in ascending order
        shortestPaths.append(candidates[0])
        candidates.remove(candidates[0])
    return shortestPaths
```

3.3 Pros - Cons

Pros:

- K Shortest Paths: Yen's algorithm is designed to find K shortest paths between two nodes in a graph, providing a way to explore multiple alternative routes or paths.
- Path Diversity: The algorithm aims to discover diverse paths by removing edges and recalculating paths, allowing for a variety of options with different characteristics.
- Flexibility: Yen's algorithm can work with various existing shortest path algorithms, making it adaptable to different graph structures and leveraging efficient algorithms.

Cons:

- Computational Complexity: Yen's algorithm can be computationally expensive, particularly for large graphs or when searching for many shortest paths, due to repeated path calculations and graph manipulations.
- Graph Modification: The algorithm involves modifying the graph by removing and restoring edges, which can be complex to implement and introduce additional computational overhead.
- Dependency on Underlying Algorithm: The effectiveness of Yen's algorithm relies on the efficiency and accuracy of the chosen shortest path algorithm, affecting overall performance.
- Potential Path Duplication: Yen's algorithm may generate duplicate paths if multiple equally optimal paths exist in the graph, requiring additional steps to eliminate duplicates.

III. Implementation

1. Data collection and processing

1.1 Data collection tools

Currently, there are various tools available for collecting map data, such as OpenStreetMap, Apify, etc. However, the team chooses to use Google My Maps due to its simplicity and convenience. This tool allows for the creation and customization of maps based on specific research requirements. One significant advantage of Google My Maps is its ability to import data from spreadsheets and CSV files, enabling the convenient and automated creation of maps.

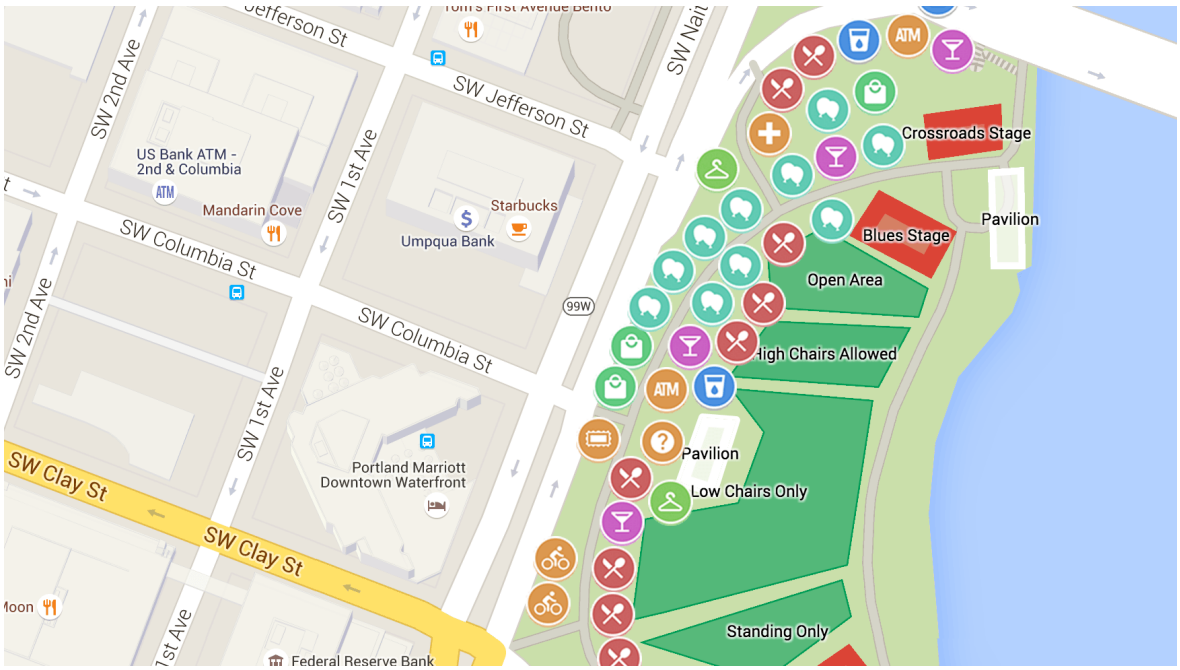

Image 1: Get subcategories

To collect and process data using Google My Maps, you can follow the following steps:

1. Click on **"Create a new map."**.
2. Add points on the map by using the "Add a marker" function at intersections, crossroads, roundabouts, or curves.
3. Connect the points added in Step 2 with straight lines, representing the graph's direction, from left to right. the data as a CSV file containing the points and lines created at step 3 and step 4.

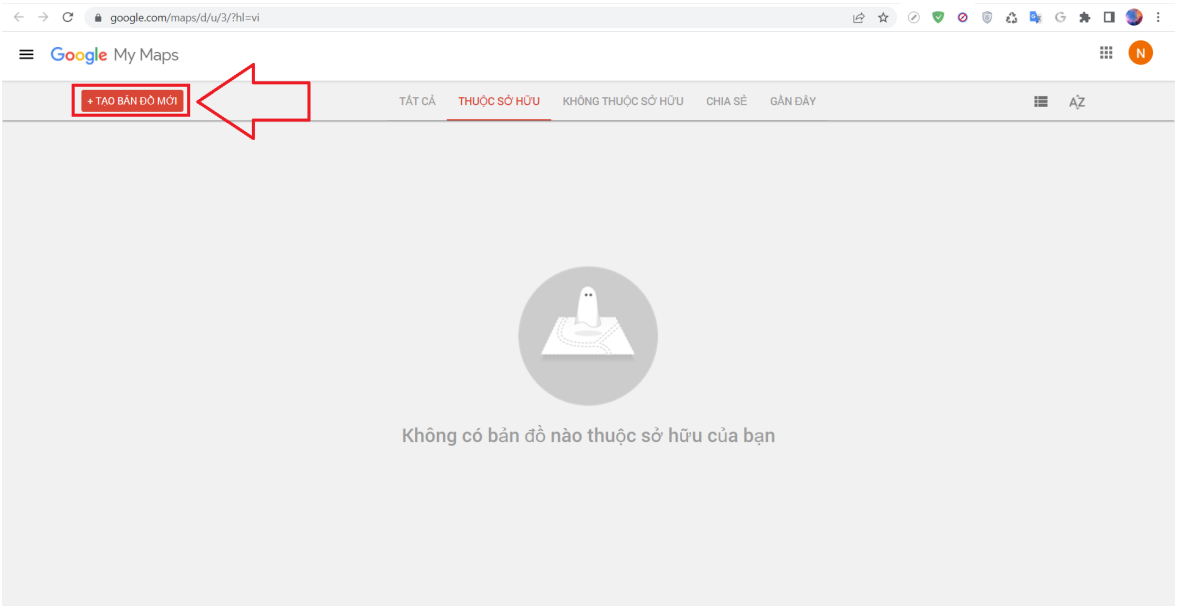

Image 2: Create a new map

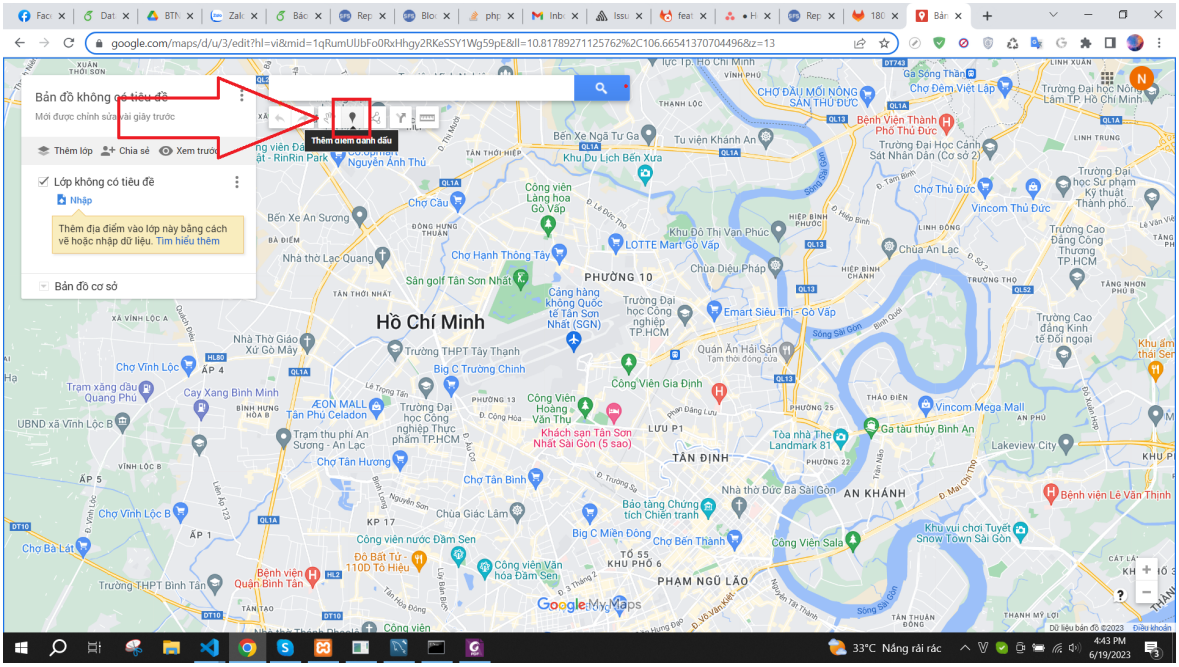

Image 3: Add a marker

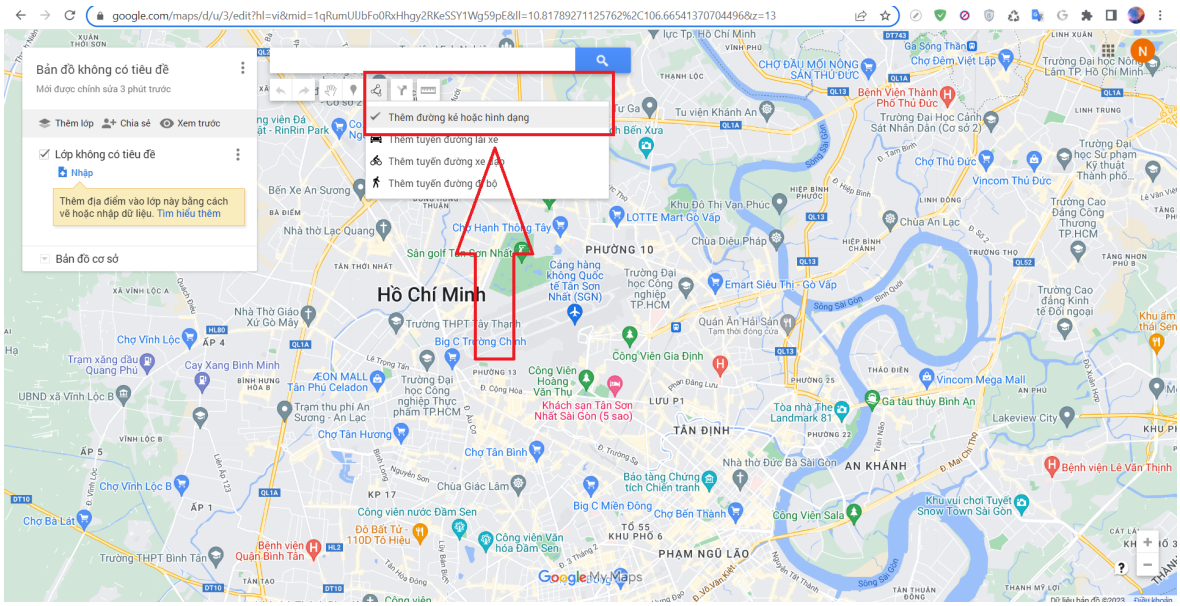

Image 4: Add line or shape

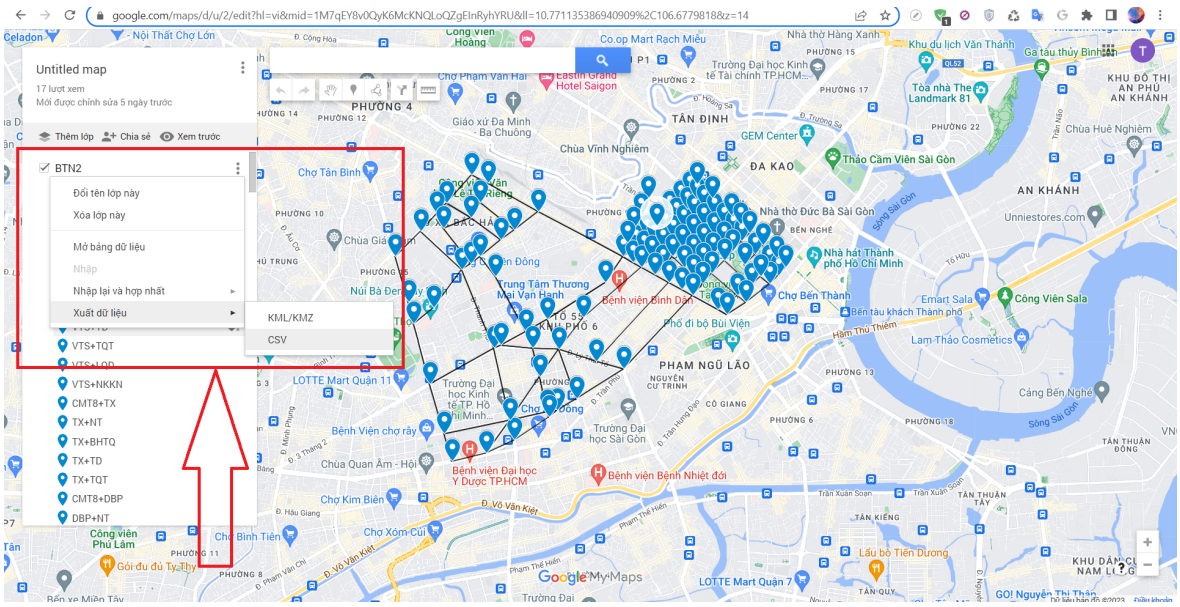

Image 5: Export data in csv type

1.2 Data collection result

Using Google My Maps, we have collected a total of 109 points with the Well-Known Text (WKT) representation of the geometry in the format "POINT (longitude, latitude)". The second column contains the description of each point. Here's an example entry:

"POINT (106.6581284 10.7705307)", LTK+THT

In this format, the longitude is specified first, followed by the latitude. Each point is enclosed in quotation marks, and the description is provided after a

comma (LTK+THT) represent cross point of Ly Thuong Kiet street and To Hien Thanh street.

And in this exercise, we find the shortest path from DHBK point (Bach Khoa University) at location "POINT (106.6574749 10.7728391)",DHBK, to DLL point (Independence Palace) at the location "POINT (106.6952984 10.7770133)",DDL

Additionally, we have 64 edges representing the directions of multiple streets. Each edge is represented in the Well-Known Text (WKT) format as "LINESTRING (longitude1 latitude1, longitude2 latitude2, ...)", followed by a description. Here's an example:

"LINESTRING (106.6560496 10.7779309, 106.6574749 10.7728391, 106.6581284 10.7705307, 106.6599147 10.7638529, 106.6615095 10.7583902, 106.6624214 10.7553358)", LKT

In this format, the longitude and latitude pairs form a continuous line representing the directed of the specific street. Each point is separated by a comma. The description, in this case, denotes "LKT," which stands for Ly Thuong Kiet street. We utilized the Haversine distance formula to calculate the weight of each node in the graph, determining the distance between geographical coordinates. The Haversine formula is expressed as follows:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

In the formula, d represents the distance between two points on the Earth's surface, r is the radius of the Earth (typically 6,371 kilometers)

1.3 Pseudocode

```
for each linestring in lines_in_csv:
    coordinates = list(linestring.coords)

    for each pair of adjacent coordinates in coordinates:
        query_start_point = create_point(start_coordinate)
        query_end_point = create_point(end_coordinate)

        found_start_point = find_point(query_start_point,
   points_in_csv)
```

```
9           if found_start_point is not found or multiple points are
    found:
10              continue
11
12          found_end_point = find_point(query_end_point, points_in_csv
    )
13          if found_end_point is not found or multiple points are
    found:
14              continue
15
16          add_weighted_edge(found_start_point, found_end_point,
    calculate_haversine_distance(start_coordinate, end_coordinate))
```

2. Bellman Algorithm

2.1 Implementation

```
def bellman_ford(graph, source):
    distances = {node: float('inf') for node in graph.nodes}
    distances[source] = 0
    predecessors = {node: None for node in graph.nodes}

    for _ in range(len(graph.nodes) - 1):
        for u, v, weight in graph.edges.data('weight'):
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight
                predecessors[v] = u

    # Check for negative cycles
    for u, v, weight in graph.edges.data('weight'):
        if distances[u] + weight < distances[v]:
            raise ValueError("Graph contains a negative-weight
    cycle")

    paths = {}
    for node in graph.nodes:
        path = []
        current_node = node
        while current_node is not None:
            path.insert(0, current_node)
            current_node = predecessors[current_node]
        paths[node] = path
```

```
    return distances, paths
```

2.2 Result

Execution time of Bellman Ford: 0.015509843826293945 seconds

```
[5.8049764098281695, ['DHBK', 'LTK+THT', 'THT', 'TT+THT', 'DN+THT',
    'THT+SVH', 'THT+NGT', 'THT+HBK', 'THT+CMT8', 'CMT8+VTS+LCT+3/2'
   , 'CMT8+TX', 'CMT8+DBP', 'CMT8+NTN', 'NDC+CMT8', 'CMT8+VVT', '
   NTMK+CMT8', 'NTMK+BHTQ', 'NTMK+TD', 'NTMK+HTCC', 'NTMK+LQD', '
   NTMK+NKKN', 'NKKN+LD', 'DDL']]
```

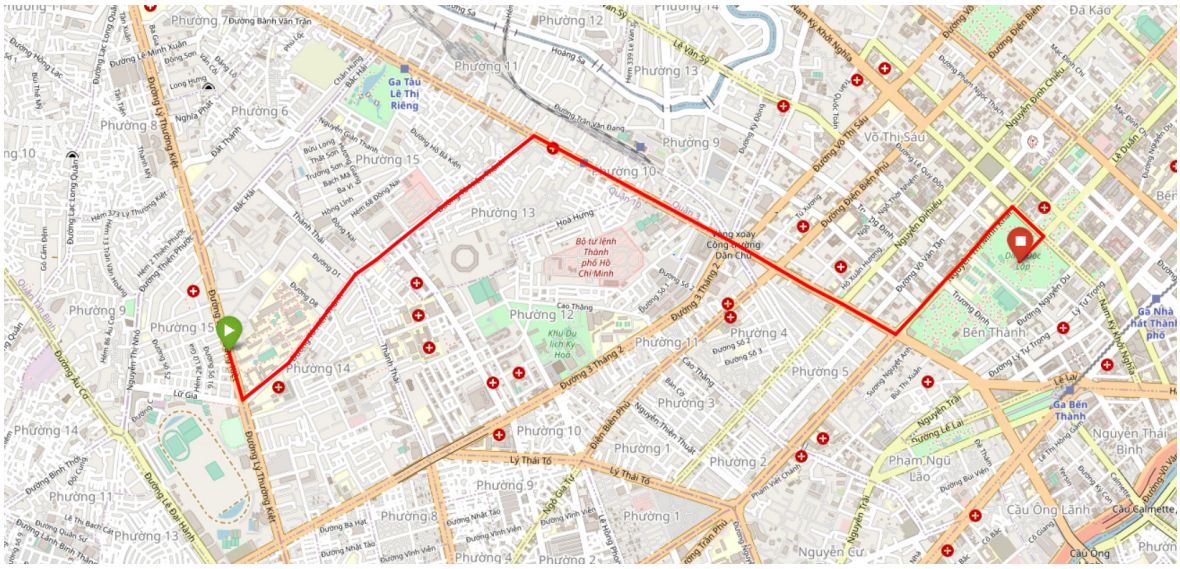

Image 6: Shortest path from HCMUT to The Independence Palace

2.3 Compare with Google result and comment

Google result:

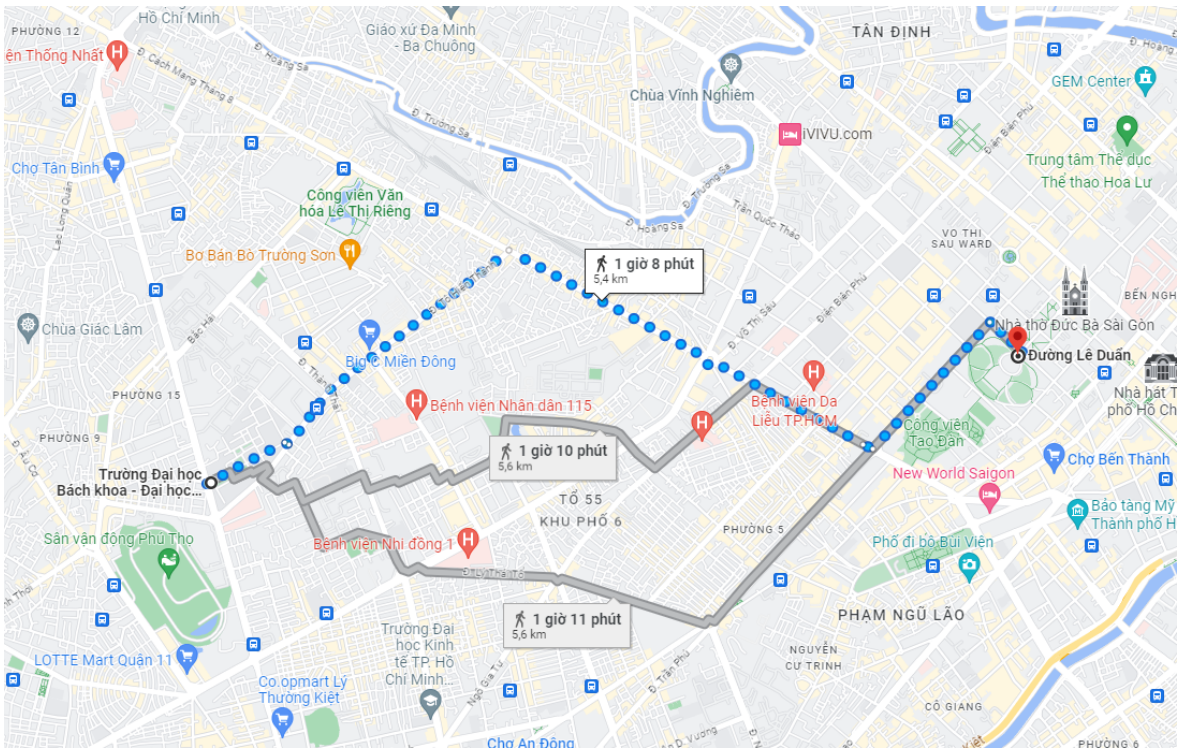

Image 7: Google route result

Compare with Google result, we can see the recommended path from Google is not same with Bellman's shortest path. The shortest path that Bellman algorithm is showing here is 90% same with the google.

3. The Floyd algorithm

3.1 Implementation

```
# -*- coding: utf-8 -*-
"""Floyd-Warshall algorithm for shortest paths.

import math
import networkx as nx

def floyd_warshall_paths(g, verbose=False):
    #start_time = timeit.default_timer()
    G_number = nx.convert_node_labels_to_integers(g)

    edges = G_number.edges(data=True)
    vertices = list(G_number.nodes())
    n_vertices = len(vertices)
```

```
    # Init dist matrix
    dist = [[math.inf for j in range(n_vertices)] for i in range(
   n_vertices)]
    path = [[None for j in range(n_vertices)] for i in range(
   n_vertices)]

    for (u, v, w) in edges:
        u = u - 1
        v = v - 1
        w = w['weight']
        dist[u][v] = w
        path[u][v] = v + 1

    for v in range(n_vertices):
        dist[v][v] = 0
        path[v][v] = v + 1

    if verbose:
        print('>> run floyd-warshall algorithm')

    for k in range(n_vertices):
        for i in range(n_vertices):
            for j in range(n_vertices):

                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    path[i][j] = path[i][k]

                    if verbose:
                        print('   update edge (', i, ',', j, '),
   value:', dist[i][j])

    # Elapsed time
    #if verbose:
        #elapsed = (timeit.default_timer() - start_time) * 1000
        #print('>> elapsed time', elapsed, 'ms')

    # Return nodes and dist matrix
    return {'nodes': vertices, 'dist': dist, 'path': path}

def get_shortest_path(paths, u, v):

    # Validation
```

```
    if paths[u - 1][v - 1] is None:
        return []

    # Build the shortest path
    path = [u]
    while u != v:
        u = paths[u - 1][v - 1]
        path.append(u)

    return path
```

3.2 Result

```
[5.800076409800695 , ['DHBK','LTK+THT','THT','TT+THT','DN+THT','THT
    +SVH','THT+NGT','THT+HBK','THT+CMT8','CMT8+VTS+LCT+3/2','CMT8+TX
    ','CMT8+DBP','CMT8+NTN', 'NDC+CMT8', 'CMT8+VVT', 'NTMK+CMT8', '
    NTMK+BHTQ', 'NTMK+TD', 'NTMK+HTCC', 'NTMK+LQD', 'NTMK+NKKN', '
    NKKN+LD', 'DDL']]
```

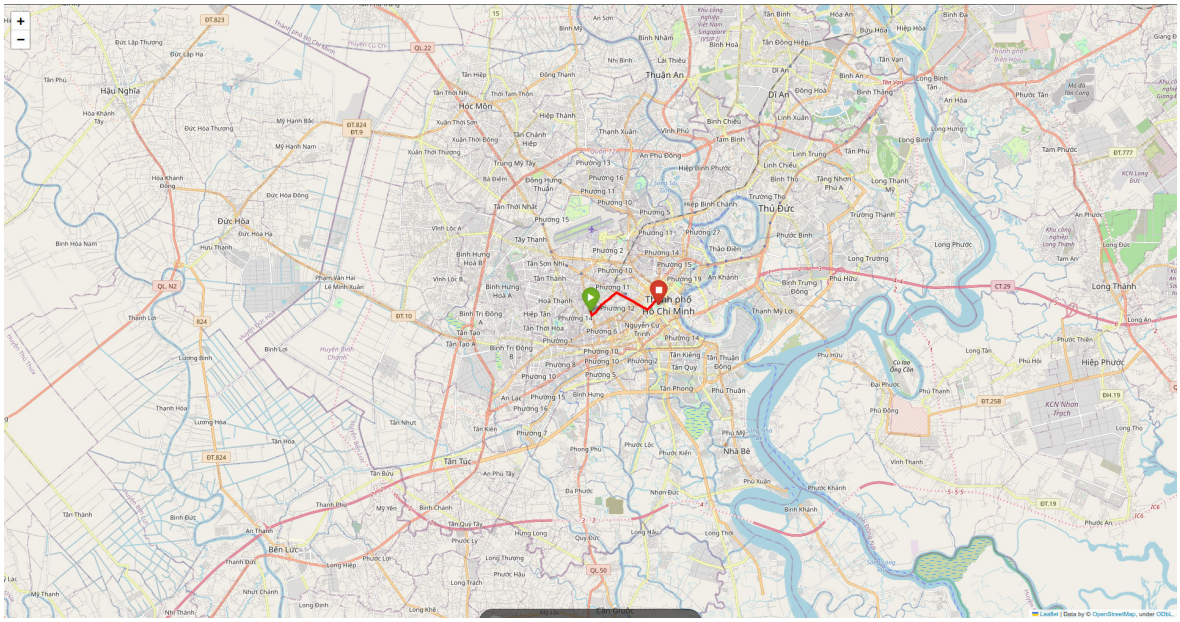

Image 8: FLoyd path

3.3 Compare with Google result and comment

Google result:

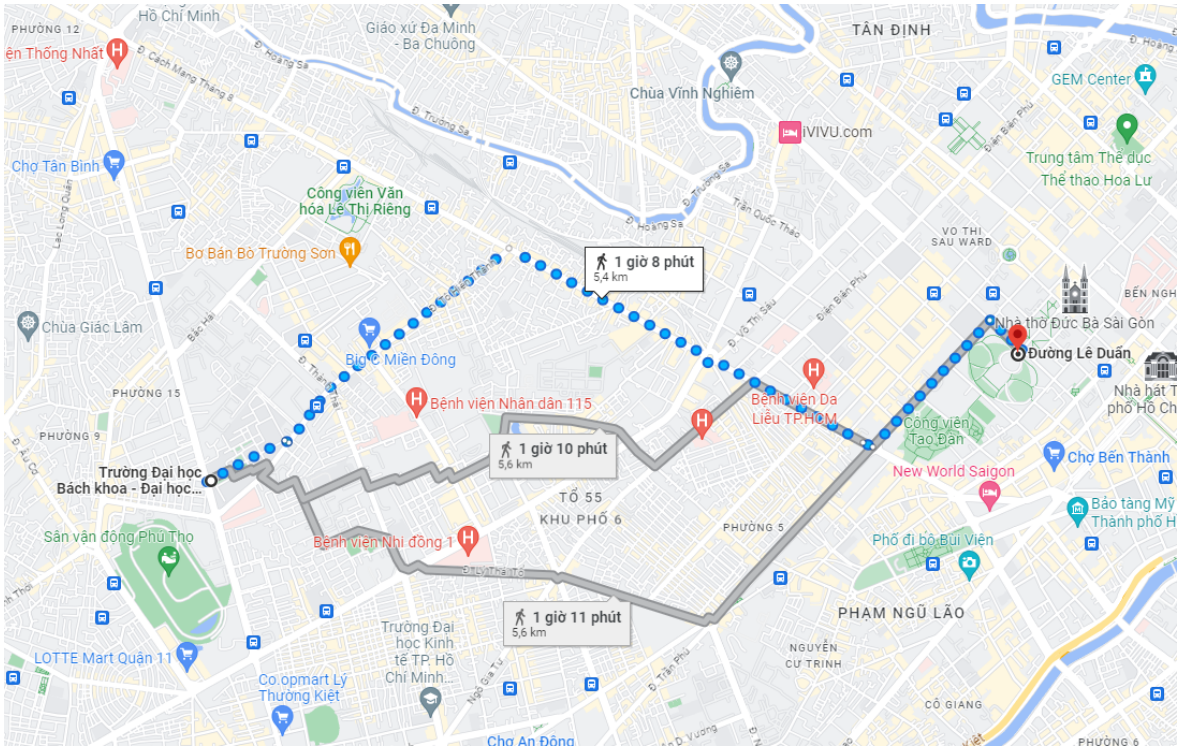

Image 9: Google route result

Compare with Google result, we can see the recommended path from Google is not same with Floyd's shortest path. The shortest path that FLoyd algorithm is showing here is 90% same with the google.

4. Yen' s algorithm

4.1 Implementation

Follow the pseudocode, we implement the steps as below:

Function defination:

```
def k_shortest_paths(G, source, target, k=1, weight='weight'):
```

Step 1: Calculate the shortest path using any shortest path algorithm, this code use Dijkstra

```
length, path = nx.single_source_dijkstra(G, source, target=None,
    weight=weight)
```

Step 2: Iteratively find the next K -1 shortest paths: we will repeat step 3, 4, 5, 6, 7, 8 below for each kth-shotest path.

Step 3: Generate potential candidate paths

```
for j in range(len(paths[-1]) - 1):
    spur_node = paths[-1][j]
    root_path = paths[-1][:j + 1]
```

Step 4: Remove edges

```
edges_removed = []
for c_path in paths:
    if len(c_path) > j and root_path == c_path[:j + 1]:
        u = c_path[j]
        v = c_path[j + 1]
        if G.has_edge(u, v):
            edge_attr = G.adj[u][v]
            G.remove_edge(u, v)
            edges_removed.append((u, v, edge_attr))
```

Step 5: Calculate spur path

```
spur_path_length, spur_path = nx.single_source_dijkstra(G,
   spur_node, target=None, weight=weight)
```

Step 6: Combine root and spur paths

```
if target in spur_path and spur_path[target]:
    total_path = root_path[:-1] + spur_path[target]
    total_path_length = get_path_length(G_original, root_path,
   weight) + spur_path_length[target]
    heappush(B, (total_path_length, next(c), total_path))
```

Step 7: Restore removed edges

```
for e in edges_removed:
    u, v, edge_attr = e
    G.add_edge(u, v, weight = edge_attr[weight])
```

Step 8: Select the best candidate path

```
if B:
    B.sort()
    while True:
        (l, _, p) = heappop(B)
        if p not in paths:
            break
    lengths.append(l)
    paths.append(p)
```

To get the 2nd shortest path, we run below code:

```
distances, paths = k_shortest_paths(G, source_node, destination, 2)
```

4.2 Result

Time for finding 1st and 2nd path

— 0.18931937217712402 seconds —

1st shortest path (found by Dijkstra) and its distance:

```
[5.8049764098281695, ['DHBK', 'LTK+THT', 'THT', 'TT+THT', 'DN+THT',
    'THT+SVH', 'THT+NGT', 'THT+HBK', 'THT+CMT8', 'CMT8+VTS+LCT+3/2'
   , 'CMT8+TX', 'CMT8+DBP', 'CMT8+NTN', 'NDC+CMT8', 'CMT8+VVT', '
   NTMK+CMT8', 'NTMK+BHTQ', 'NTMK+TD', 'NTMK+HTCC', 'NTMK+LQD', '
   NTMK+NKKN', 'NKKN+LD', 'DDL']]
```

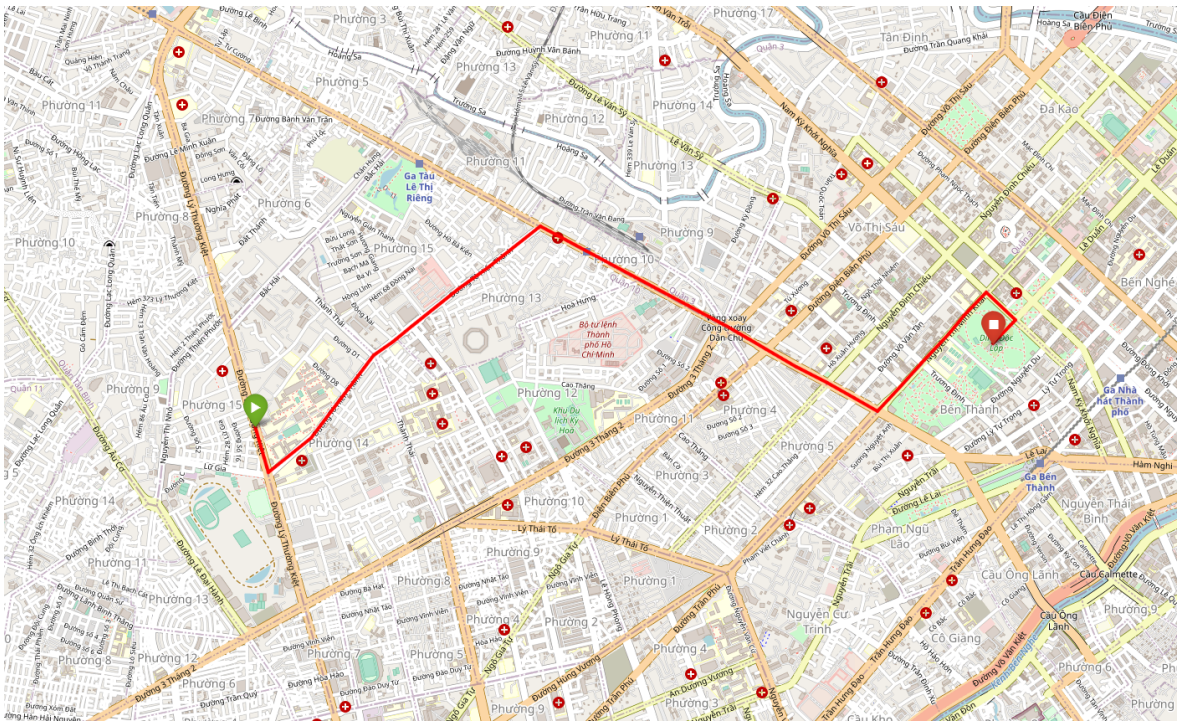

Image 10: Yen's 1st path

2nd shortest path and its distance:

```
[5.88040141289269, ['DHBK', 'LTK+THT', 'THT', 'TT+THT', 'DN+THT', '
   THT+SVH', 'THT+NGT', 'THT+HBK', 'THT+CMT8', 'CMT8+VTS+LCT+3/2',
   'CMT8+TX', 'TX+NT', 'DBP+NT', 'DBP+BHTQ', 'DBP+TD', 'TQT+DBP', '
   LQD+DBP', 'DBP+NKKN', 'NTN+NKKN', 'NDC+NKKN', 'VVT+NKKN', 'NTMK+
   NKKN', 'NKKN+LD', 'DDL']]
```

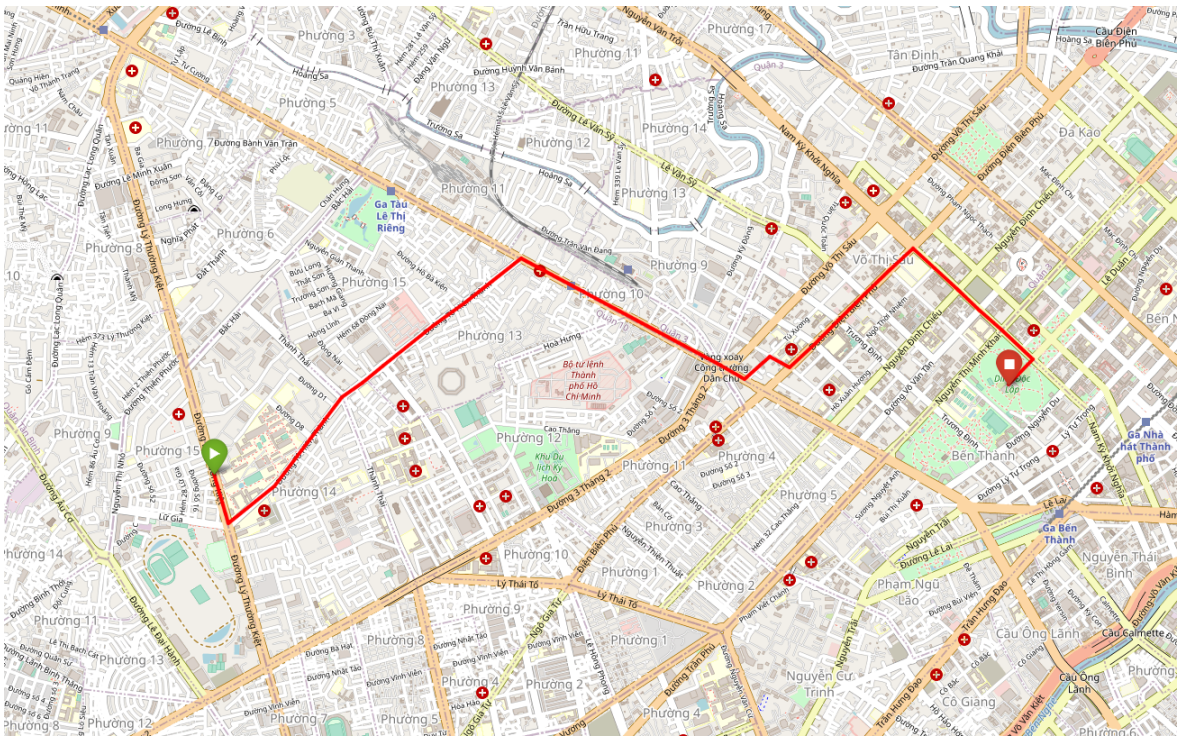

Image 11: Yen's 2nd path

4.3 Compare with Google result and comment

Google result:

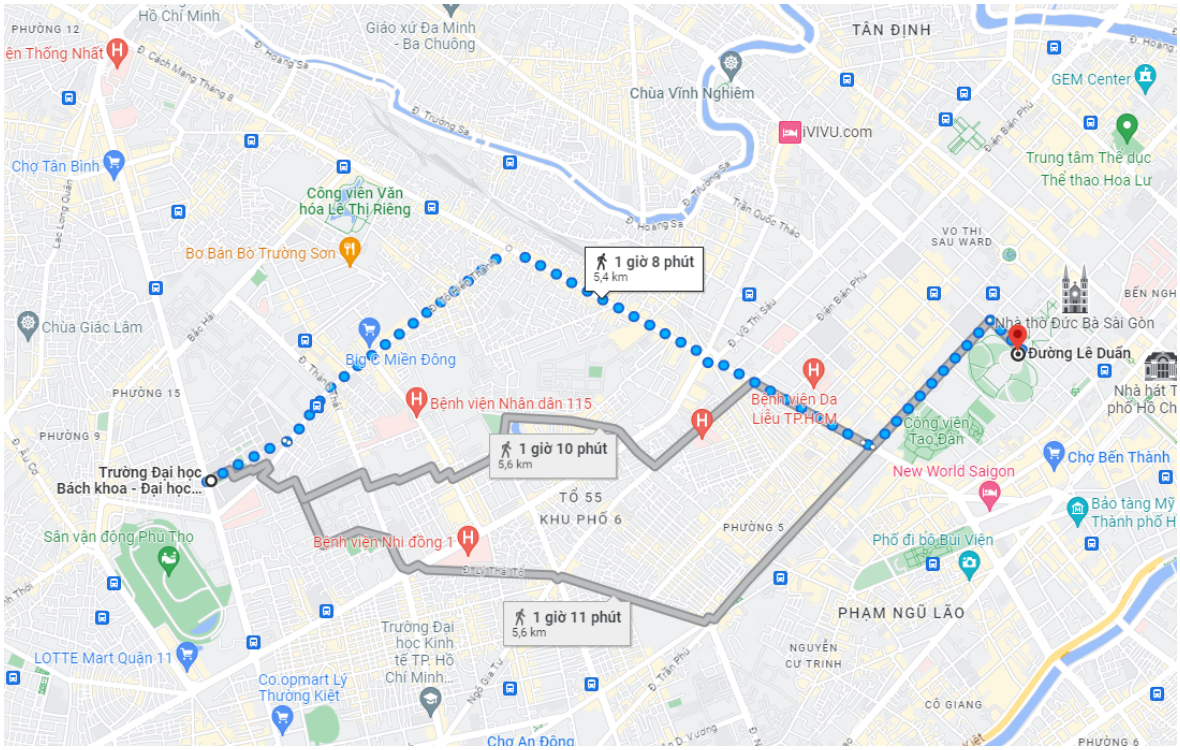

Image 12: Google route result

Compare with Google result, we can see the 2nd recommended path from

Google is not same with Yen's 2nd shortest path. This might due to Google will have many other criteria to recommend route meanwhile Yen algorithm refer only the distance. The 2nd shortest path that Yen algorithm is showing here is 90% same with the 1st one.

So in the real life, we also need to consider traffic situation, topographic, percentage likely with 1st shortest path, etc. to get the path that have high applicability.

IV. Conclusion

1. Compare Bellman-Ford and Floyd in finding shortest path

In this project, we employ an Acer VX5 laptop to conduct a comparative analysis between the Bellman-Ford and Floyd-Warshall algorithms. The laptop possesses the following specifications: 16GB of RAM, a 512GB SSD, and an Intel Core i5 Kabylake 7300HQ CPU clocked at 2.5GHz.

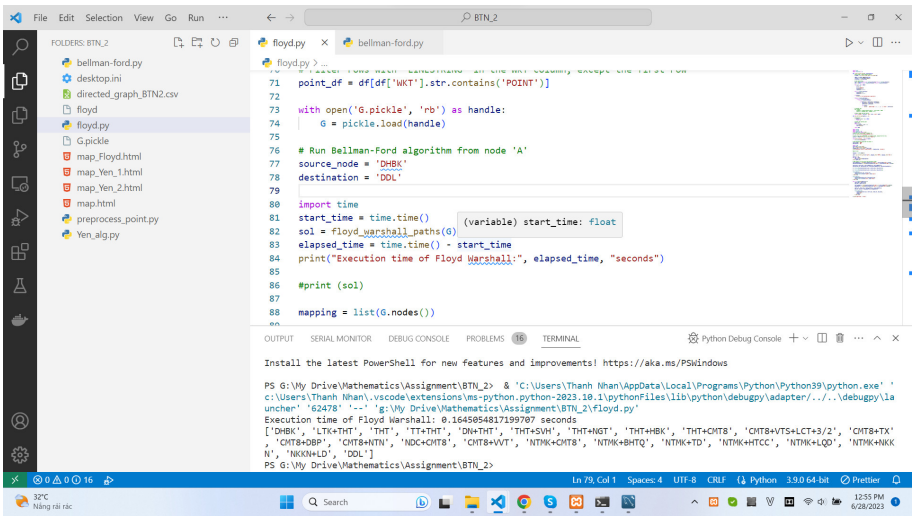

Image 13: Floyd Warshall execution time result

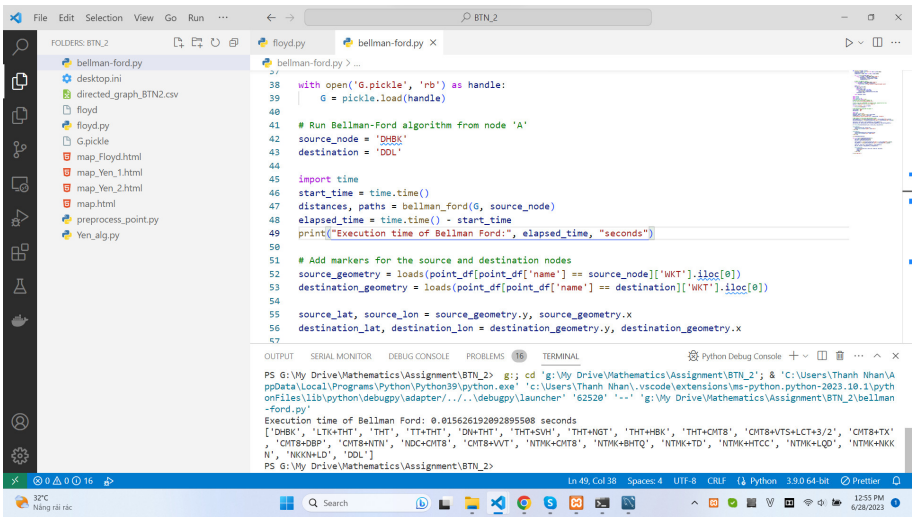

Image 14: Bellman Ford execution time result

	Bellman-Ford	**Floyd-Warshall**
Purpose	The Bellman-Ford algorithm is primarily used to find the shortest path from a single source vertex to all other vertices in a weighted directed graph. It can handle graphs with negative edge weights, but it detects negative cycles and reports their existence.	The Floyd-Warshall algorithm is used to find the shortest path between all pairs of vertices in a weighted directed graph. It is applicable to graphs with both positive and negative edge weights, but it does not handle negative cycles.
Approach	The Bellman-Ford algorithm uses a dynamic programming approach. It iteratively relaxes the edges in the graph, updating the distance estimates until the shortest paths are found. It repeats this process for a maximum of $V - 1$ times, where V is the number of vertices in the graph.	The Floyd-Warshall algorithm uses a matrix-based approach. It builds a distance matrix that represents the shortest distances between all pairs of vertices. It iteratively updates the matrix by considering intermediate vertices in each iteration.
Time Complexity	The time complexity of the Bellman-Ford algorithm is $O(V * E)$, where V is the number of vertices and E is the number of edges in the graph.	The time complexity of the Floyd-Warshall algorithm is $O(V^3)$, where V is the number of vertices in the graph.
Negative Cycles	The Bellman-Ford algorithm can detect negative cycles in a graph. If there is a negative cycle reachable from the source vertex, the algorithm will report the presence of a negative cycle.	The Floyd-Warshall algorithm does not handle negative cycles. It assumes that there are no negative cycles in the graph or any negative cycle is not reachable from any pair of vertices.
Graph Types	The Bellman-Ford algorithm can be applied to both directed and undirected graphs, as well as graphs with negative edge weights.	The Floyd-Warshall algorithm can be applied to directed and undirected graphs with positive or negative edge weights, but it does not handle negative cycles.

Table 1: Bellman-Ford and Floyd finding shortest path comparision

In summary, the Bellman-Ford algorithm is primarily used to find the shortest path from a single source to all other vertices, while the Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in a graph. The Bellman-Ford algorithm can handle negative edge weights and detect negative cycles, whereas the Floyd-Warshall algorithm does not handle negative cycles but can handle both positive and negative edge weights.

In finding shortest path problem from Ho Chi Minh University of Technology to The Independence Palace, Bellman-Ford algorithm and Floyd-Warshall algorithm generate same path. However, Floyd-Warshall is more computational efficiency than Bellman-Ford due to the approach and time complexity of 2 algorithms.

2. Choosing K-th shortest path algorithm

When compared to other algorithms for finding the kth shortest path, such as the K-shortest paths algorithm or the A* algorithm, the Yen algorithm stands out due to its ability to handle negative edge weights. This makes it a preferred choice in scenarios where negative weights are present.

Compare with another popular aglorithm - Eppstein, the Yen algorithm provides versatility in handling negative edge weights but may have longer runtimes for larger graphs. On the other hand, the Eppstein algorithm is generally more efficient, especially for sparse graphs, but it assumes non-negative edge weights and is more suitable for directed acyclic graphs.

In conclusion, the choice among algorithms depends on the specific requirements and characteristics of the problem at hand. Besides, as mentioned above, we need to consider many other conditions to give the solution for a realife problem.

REFERENCES

[1] Yen's algorithm. Last access 20/06/2023.
https://en.wikipedia.org/wiki/Yen%27s_algorithm

[2] Yen' s Shortest Path algorithm. Last access 20/06/2023.
https://neo4j.com/docs/graph-data-science/current/algorithms/yens/

[3] - Bellman–Ford Algorithm | DP-23. Last access 24/06/2023.
https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

[4] Bellman Ford's Algorithm. Last access 24/06/2023.
https://www.programiz.com/dsa/bellman-ford-algorithm