City Navigation and Emergency Route Planning Tool

Group 5

Rohit Bhaskar Uday, Viditi Vartak, Tushar Yadav,

Rohit Chowdary Yadla, Huan Ying, Shiyun Zhou

Department of Computer Science, Cal State Fullerton

CPSC 535

Dr. Syed Hassan Shah

February 23, 2024

ABSTRACT

This project introduces a navigational tool for Orange County, California, using Python, PyQt5 for the UI, Folium for mapping, and the integration of OSMnx with NetworkX for advanced route finding. It allows users to select start and end points from Orange County cities, visualize the shortest routes on a map, and adjust for road closures by recalculating alternative paths. The inclusion of the Floyd-Warshall algorithm enhances the application by providing a comprehensive method for finding the shortest paths between all pairs of nodes in the road network graph, ensuring efficient route planning across the county. This feature is particularly useful for identifying optimal routes in complex scenarios involving multiple blockages. The tool's combination of user-friendly design, dynamic mapping, and sophisticated route optimization techniques makes it a valuable resource for both planning and real-time navigation adjustments in response to road conditions.

TABLE OF CONTENTS

1. INTRODUCTION

The primary objective of City Navigation and Emergency Route Planning Tool is a city navigation tool capable of calculating the shortest path from A spot to B spot based on the weight of the road in Orange County, California. To achieve this, we will implement the Floyd-Warshall algorithm, a powerful algorithm for finding the shortest paths in a graph, to pre-calculate shortest paths between cities in Orange County.

Our project comprises four main components: data collection, algorithm implementation, simulation of blockages, and UI development.

The first step of this project is data collection, which involves gathering data on city roads, intersections, and landmarks from various sources such as city planning websites, open-source mapping platforms, or APIs like OpenStreetMap. This data will be converted into a graph format where intersections serve as nodes and roads as weighted edges. Secondly, algorithm implementation. This process mainly focuses on implementing the Floyd-Warshall algorithm to calculate the shortest path matrix based on the city graph. The team will also develop efficient methods for storing and accessing this matrix. Thirdly, the simulation of blockages aims to create a module capable of simulating road blockages, allowing for the modification of graph weights to simulate blocked roads. After simulating blockages, the Floyd-Warshall algorithm will be rerun to update the shortest path matrix accordingly. Lastly will be the implementation of User Interface (UI) and Visualization. This process focuses on the design and development of a user-friendly interface where users can select start and end points, visualize shortest paths on a city map, and simulate blockages. The team will ensure that the UI facilitates easy interaction with the tool's functionalities. All of the development details will be listed in the section below.

Also, We did another implementation where we explores the application of the Floyd-Warshall algorithm to model and analyze a network of cities connected by various

distances. By simulating a graph where nodes represent cities and edges denote roads with assigned weights, we aim to demonstrate the algorithm's utility in determining the shortest paths between all pairs of nodes. This approach provides valuable insights into efficient route planning and infrastructure development, showcasing the intersection of theoretical computer science with practical urban planning challenges.

2. DATA COLLECTION

2.1 Data Collection and Geocoding

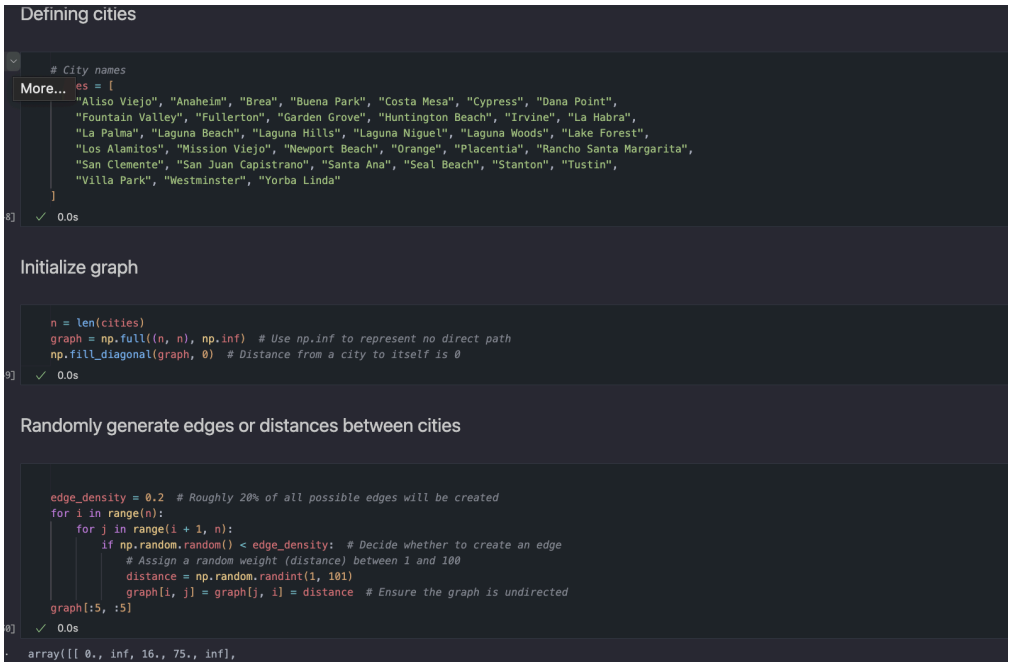
In the project, data collection is a key step, providing the necessary informational foundation for the tool. Initially, the project utilizes the OSMnx library to obtain road network data for specific areas[1]. OSMnx directly interfaces with the OpenStreetMap API to obtain geographical location and network data for Orange County. This involves querying OpenStreetMap to obtain the structure of the road network, including roads, intersections (nodes), and connections between roads (edges). In this process, ox.graph_from_place() is used to specify the name of the location, and by specifying network_type="drive", the road network suitable for driving conditions is filtered, excluding pedestrian paths, bike lanes, and other types of roads. ox.geocode() is used to obtain the latitude and longitude coordinates of the location.

Code for Application Implementation:

```
place_name = "Orange County, California, USA"
G = ox.graph_from_place(place_name, network_type="drive")
map_center = ox.geocode(place_name)
```

Code for Jupyter Script Implementation

For data collection in this implementation we just randomly joined cities and assigned edge weights to simulate a scenario or a system of cities.



2.2 Geocoding and Node Identification

The code converts city names into geographic coordinates (latitude and longitude) through ox.geocode(source_name + ", California, USA") and ox.geocode(destination_name + ", California, USA"), transforming human-readable place names into specific coordinates on the map. Then, using ox.nearest_nodes(G, source_location[1], source_location[0]) and ox.nearest_nodes(G, destination_location[1], destination_location[0]), the code finds the closest nodes in the road network to these coordinates based on the geographic coordinates obtained earlier. These nodes are points in the road network graph, representing actual road intersections or endpoints of road segments. This step relies on the road network data (G) previously collected and constructed via OSMnx, thus locating specific network nodes.

Code:

```
source_location = ox.geocode(source_name + ", California, USA")
destination_location = ox.geocode(destination_name + ", California, USA")
source_node = ox.nearest_nodes(G, source_location[1], source_location[0])
target_node = ox.nearest_nodes(G, destination_location[1], destination_location[0])
```

3. ALGORITHM IMPLEMENTATION

Implementation 1 - Jupyter Notebook

In this project, we demonstrate the application of the Floyd-Warshall algorithm, a classic dynamic programming solution for finding the shortest paths between all pairs of nodes in a weighted graph. This algorithm is particularly useful in scenarios where any pair of nodes might be required to compute the shortest path, not just a specific source and destination. Our simulation involves a network of cities connected by roads of varying distances, aiming to illustrate the algorithm's utility in urban planning and logistics.

Graph Construction

We initiated our simulation by constructing a directed graph where each node represents a city, and each edge represents a road connecting two cities. The cities included in our simulation are part of a predefined list, representing a region of interest. To simulate realistic road networks, we randomly generated edges between cities with randomly assigned weights, representing the distance or travel time between those cities. This approach ensures a diverse set of scenarios to demonstrate the algorithm's capabilities.

Floyd-Warshall Algorithm

The core of our project is the implementation of the Floyd-Warshall algorithm. This algorithm iterates through all possible pairs of nodes, systematically updating the shortest path between each pair based on the transitive relationships identified through intermediate nodes. The mathematical formulation of the algorithm updates the distance matrix D as follows:

For each node k, for each pair of nodes (i, j), if the sum of the distances from i to k and from k to j is less than the current distance from i to j, then update the distance from i to j to this sum. Mathematically, this can be represented as:

$$D[i,j]=\min(D[i,j],D[i,k]+D[k,j])$$

$$D[i,j]=\min(D[i,j],D[i,k]+D[k,j])$$

Additionally, we maintained a matrix to track the next node on the shortest path from any given node to another. This data structure enables the reconstruction of the shortest path itself, not just the computation of its length.

Implementation 2 - Shortest Path Application

The shortest path finding algorithm in this project is based on Dijkstra's algorithm, which is a fundamental algorithm in graph theory used to find the shortest paths between nodes in a graph. The algorithm works by iteratively selecting the node with the shortest distance from a set of unvisited nodes, updating the distances of its neighbors, and marking the selected node as visited.

In this project, we use the NetworkX library in Python to implement Dijkstra's algorithm for finding the shortest path between two cities in California, USA. We represent the road network as a graph, where each node represents a location and each edge represents a road between two locations. The weight of each edge is the distance between the two locations, and the algorithm calculates the shortest path based on these weights.

Additionally, we have implemented functionality to block roads between two cities, which modifies the graph by removing nodes along the shortest path. This allows us to find alternate routes that bypass the blocked roads.

The algorithm implementation includes the use of the OSMnx library to retrieve the road network data and the Folium library to visualize the map with the shortest path and blocked roads highlighted.

Overall, the algorithm implementation in this project provides a practical and efficient way to find shortest paths and handle road blockages in a road network.

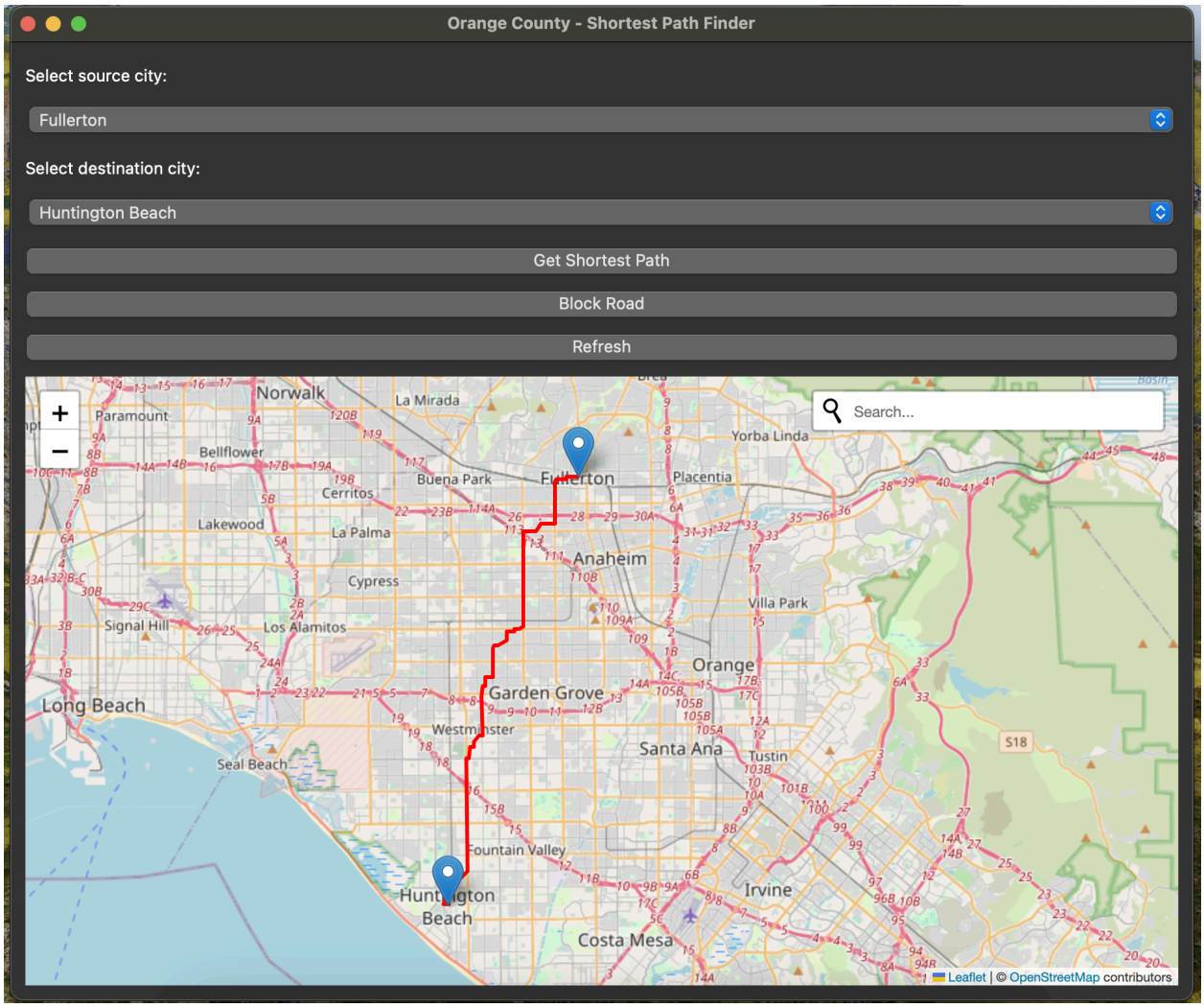
4. SIMULATION OF BLOCKAGES

In this project, we simulate road blockages between two cities in the road network to demonstrate the algorithm's ability to find alternate routes. The simulation involves dynamically blocking roads along the shortest path between the selected cities and then recalculating the shortest path to find an alternative route.

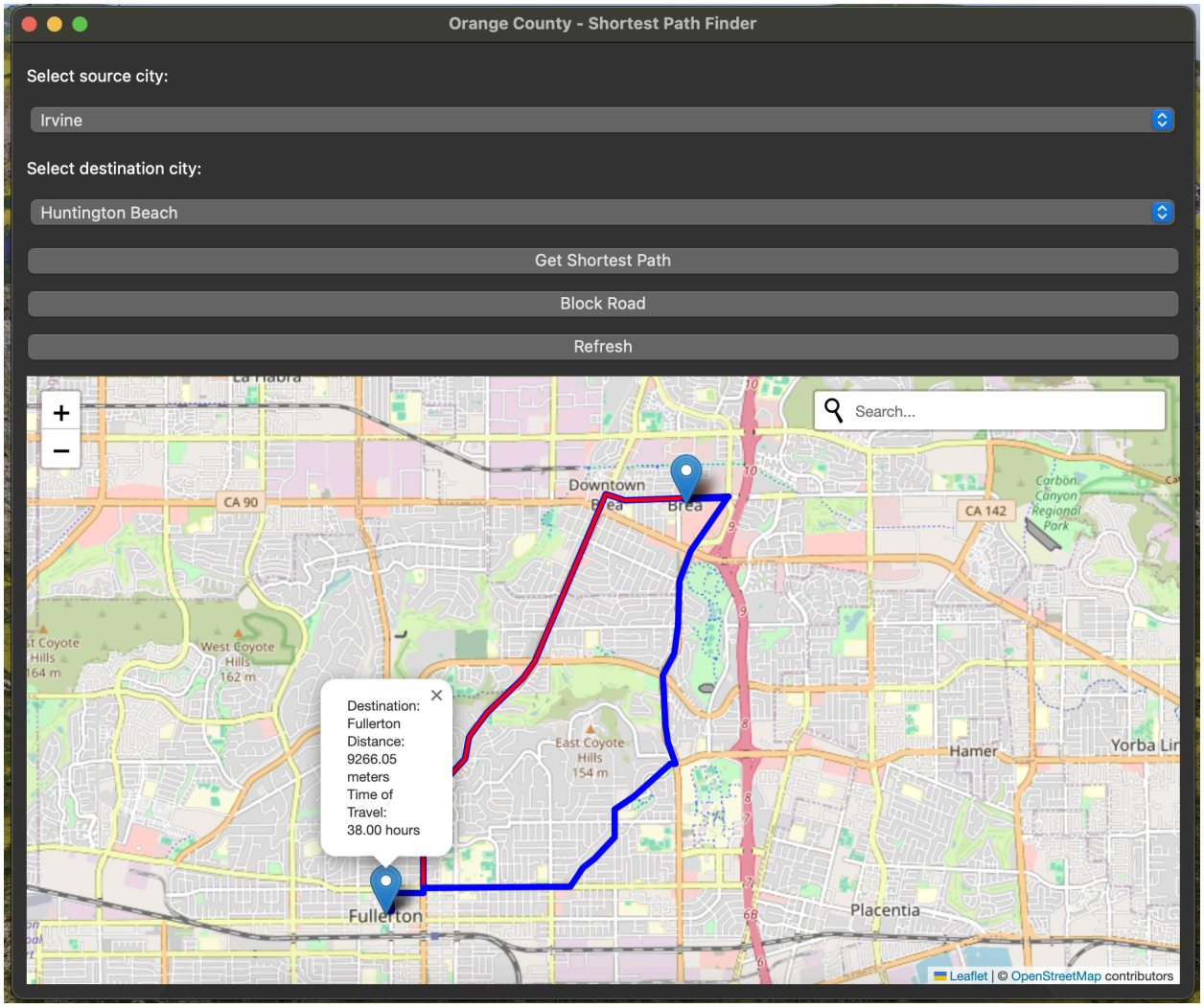
To simulate a road blockage, we identify a node along the shortest path and temporarily increase the weight of the corresponding edge in the graph representation of the road network. This simulates the closure or obstruction of a road segment, forcing the algorithm to find an alternate route that avoids the blocked segment.

After the road blockage is simulated by increasing the edge weight, the algorithm recalculates the shortest path between the cities, considering the blocked roads. This demonstrates the algorithm's adaptability to changing road conditions and its ability to find efficient routes even when faced with obstacles.

The simulation of blockages provides insights into the algorithm's robustness and effectiveness in real-world scenarios where road closures or blockages may occur. By showcasing the algorithm's ability to find alternate routes, even in the presence of road blockages, we demonstrate its practical utility in route planning and navigation applications.


Orange County - Shortest Path Finder
Select source city:
Fullerton
Select destination city:
Huntington Beach
Get Shortest Path
Block Road
Refresh
Search...
Norwalk
La Mirada
Paramount
Bellflower
Buena Park
Fullerton
Placentia
Yorba Linda
Cerritos
Lakewood
La Palma
Anaheim
Cypress
Villa Park
Signal Hill
Los Alamitos
Orange
Garden Grove
Long Beach
Westminster
Santa Ana
Tustin
Seal Beach
S18
Fountain Valley
Irvine
Huntington Beach
Costa Mesa
Leaflet | © OpenStreetMap contributors

5. USER INTERFACE



1. Initializing the Map and GUI:

   In addition to setting up the main window and PyQt5 GUI elements, such as labels, combo boxes for choosing cities, action buttons, and a map widget for showing Folium maps, the script initializes the map of Orange County.

2. Creating PyQt5 GUI Components:

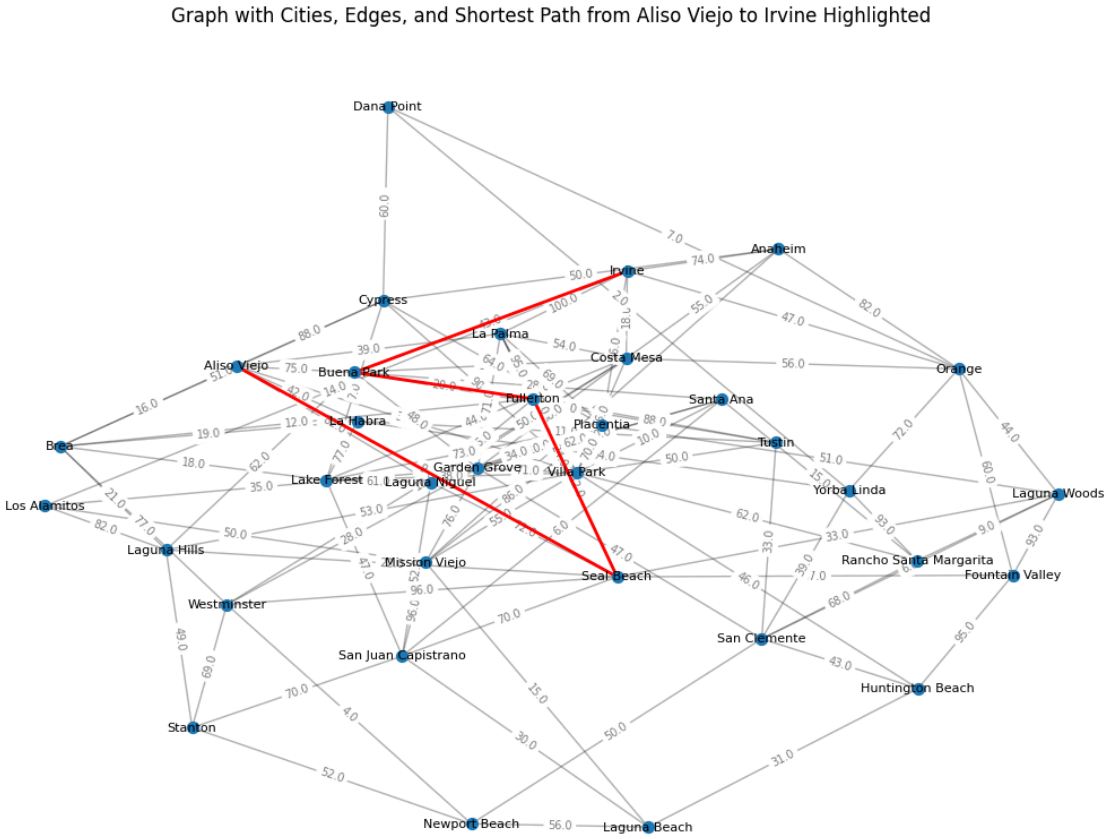
   A number of PyQt5 widgets are made: labels, combo boxes for choosing a city, buttons for functions like obstructing traffic and finding the shortest way, and a map widget for showing Folium maps.

3. Displaying the GUI:

   In order to guarantee that the user interface is visible and interactive, the script ends by displaying the main window and initiating the PyQt5 application event loop.

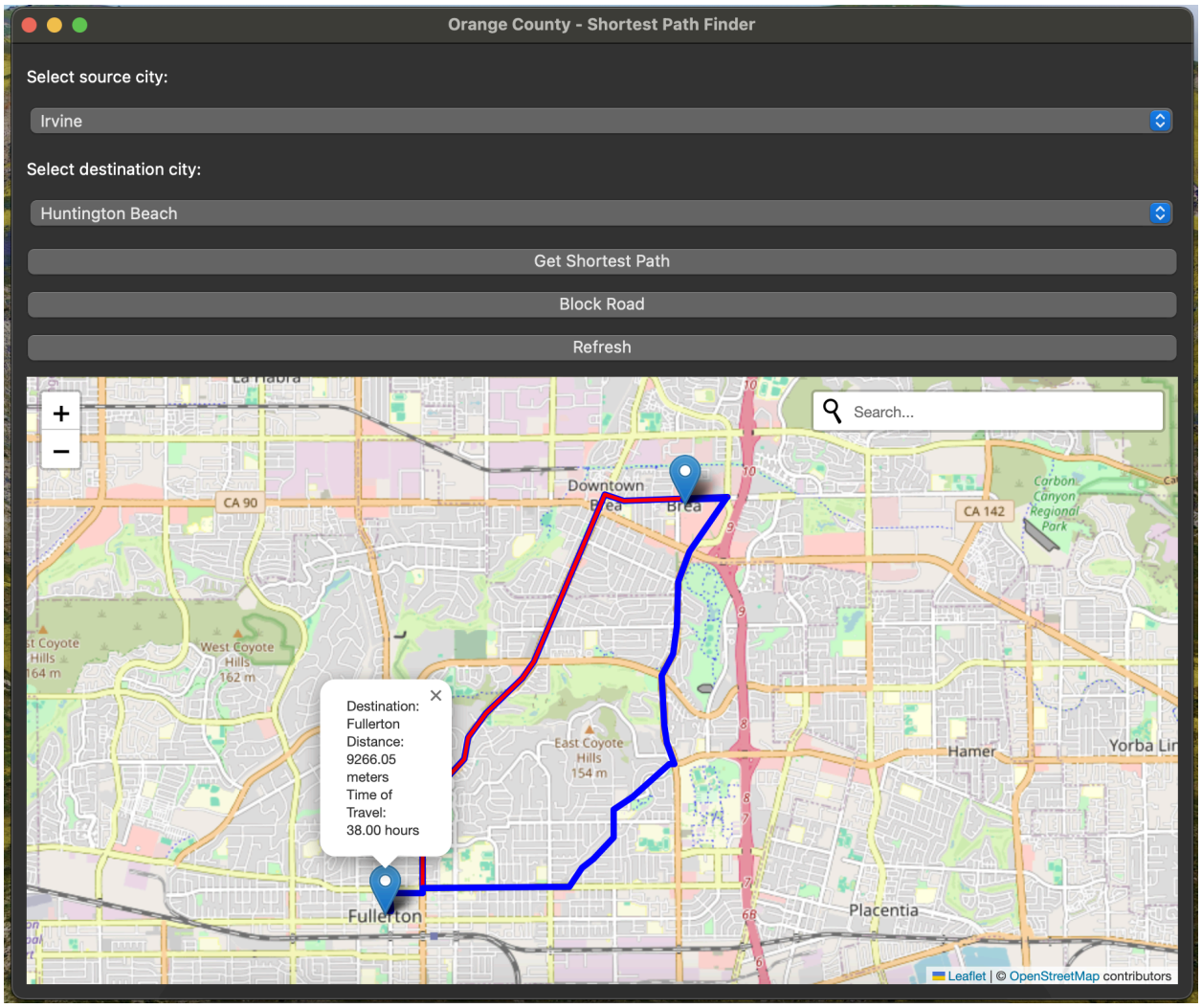
6. VISUALIZATION

Implementation 1: Jupyter Notebook



The visualization of the cities and their connections, along with the shortest paths computed by the Floyd-Warshall algorithm, was achieved using the networkx and matplotlib libraries. This approach enabled us to graphically represent the network of cities as a graph, where nodes represent cities and edges represent the roads between them. We highlighted the shortest path between two specific cities to demonstrate the algorithm's effectiveness. The code snippet below illustrates how we generated the graph and highlighted a path:

Implementation 2: Shortest Path Application



The visualization part of the code is primarily handled using the folium library, which is integrated with the PyQt5 GUI. The key components for visualization include the creation of a Folium map (mymap), adding polylines to represent paths, and updating the map display within the PyQt5 application.

1. Initialization of Folium Map:

A Folium map focused on Orange County, California, is initialized by the script. The road infrastructure in the given location can be represented as a network graph using the ox.graph_from_place function.

2. Highlighting Shortest Path (calculate_shortest_path):

Using the custom weight function (custom_weight), the calculate_shortest_path function determines the shortest path between a given set of source and destination cities. On the Folium map, the resulting path is then shown as a blue polyline.

3. Blocking Road (block_road):

The block_road function eliminates a node—or several nodes—from the shortest path computation in order to block a road. On the Folium map, the obstructed road is shown as a red polyline.

4. Refreshing Map (refresh_map):

The refresh_map function modifies the map display in the PyQt5 application and resets the Folium map, removing any highlighted or blocked roads.

7. TESTING & RESULTS

TestCase_id	Test_Case	TestCase_Description	Expected_Output	Status
TC_01	GUI Layout and map	User should be able to open the application to ensure the GUI layout is visually appealing and components are well-organized.	A GUI layout with map and all the buttons should open.	PASS
TC_02	Source city	User should be able to select city from the drop down.	The city should be selected from the dropdown list.	PASS
TC_03	Destination city	User should be able to select city from the drop down.	The city should be selected from the dropdown list.	PASS
TC_04	Zoom buttons	User should be able to zoom in and zoom out the map.	Zoom in and out should work properly.	PASS
TC_05	Get shortest path button	User should be able to click on the button.	The shortest path should be displayed on map.	PASS
TC_06	Block road button	User should be able to click on the button.	The road should be blocked between selected cities.	PASS
TC_07	Refresh button	User should be able to click on the button.	The layout should be refreshed.	PASS

TC_08	Getting shortest path	User should get the shortest path displayed on the map for the selected cities.	The shortest path should be displayed on map.	PASS
TC_09	Blocking the road	User should be able to see the blocked path on map.	Blocked path is being displayed on map.	PASS
TC_10	Refresh	User should be able to refresh the map.	The map is refreshed.	PASS

We tested all the functionalities and have generated a total of 10 test cases. All the test cases passed successfully.

8. DISCUSSION

The challenges faced by this project primarily include:

1. Data Accuracy and Updates

This project relies on data provided by OpenStreetMap (OSM), meaning the accuracy and real-time updates of the road network are limited by the current state of the OSM database. OSM data is contributed by the community, which could result in missing, outdated, or incorrect information.

2. Performance and Efficiency

Calculating the shortest path, processing geocode queries, and updating map displays may involve significant computation. For complex networks with many nodes and edges, performance and response times can be challenging, especially in environments with limited resources.

3. Road Blocking and Alternative Path Processing

When users choose to block a certain road, the algorithm needs to efficiently calculate an alternative path. This requires not only accurately identifying the blocked node or road but also considering how to optimize the path search algorithm to quickly find an alternative route.

9. CONCLUSION

In conclusion, the Shortest Path Finder project successfully integrates Folium for dynamic map rendering and PyQt5 for a user-friendly interface, providing a strong tool for route planning and analysis in Orange County, California. The application calculates and displays the shortest path between chosen cities using network analysis tools like OSMnx and NetworkX. This enables users to dynamically block roads, investigate other routes, and learn more about the effects of road closures.

The implementation of the Floyd-Warshall algorithm in our city network simulation has underscored its efficacy in solving complex route optimization problems. Through our project, we effectively modeled a practical scenario, enabling the visualization and analysis of the shortest paths between cities. This exploration not only highlights the algorithm's robustness and versatility but also paves the way for its application in real-world logistics and urban planning strategies, offering a foundational tool for enhancing connectivity and efficiency in transportation networks.

While the Floyd-Warshall algorithm is highly effective for finding shortest paths in dense networks, its application faces certain limitations in scenarios with a vast number of nodes. The algorithm's computational complexity is O(n3), where n is the number of nodes in the graph. This cubic relationship implies that as the network size grows, the time required for computation increases dramatically, potentially rendering the algorithm impractical for very large networks, such as those encountered in country-scale logistics or global transportation systems. Additionally, the algorithm's memory requirement, which is O(n2) due to the need to store distance and path information for every pair of nodes, can become a significant constraint on resources. These limitations suggest the necessity for more efficient algorithms or optimization techniques when dealing with large-scale networks.

10. REFERENCES

1. Boeing, Geoff. “OSMnx.” GitHub, 28 Mar. 2022, github.com/gboeing/osmnx.

2. Hagberg, Aric, et al. "NetworkX: High Productivity Software for Complex Networks." GitHub Repository. github.com/networkx/networkx

3. Python Visualization. "Folium Documentation." Folium Documentation. github.com/python-visualization/folium

4. The Qt Company Ltd. "PyQt5 Documentation." PyQt5 Documentation. doc.qt.io/qtforpython/

5. The Qt Company Ltd. "QWebEngineView Class." Qt Documentation. doc.qt.io/qt-5/qwebengineview.html