BEST ROUTE RECOMMENDATION

BHALA VIGNESH[1] SWAKSH PATWARI[2] HITESH SHANMUKHA[3]
GOURI PATIDAR[4]

May 2, 2024

Abstract

In this project, we utilize a dataset containing traffic and weather conditions for a city to optimize route planning. We employ three classical graph algorithms - A* (Astar), Dijkstra's, and Bellman-Ford algorithms - to determine the best route, shortest distance, and shortest time between given locations. To adapt to dynamic conditions, such as traffic congestion, road quality, and weather variations, we integrate a neural network model for prediction. Based on predictions, we dynamically update edge weights in the graph representation. Specifically, we leverage attributes such as traffic intensity, road quality classifications, and weather conditions to adjust edge weights. This dynamic updating of weights ensures that the route planning process considers real-time factors, leading to more accurate and efficient navigation recommendations for users.

Keyword : Route optimization, graph algorithms, Astar, Dijkstra's, Bellman-Ford, dynamic edge weights, real-time data, traffic prediction, weather prediction, neural network, osmnx, networkx, pyproj, geopandas, shortest path, shortest distance, shortest time, route planning.

Contents

1 Preprocessing **3**

2 Neural Networks **4**

3 Analysis **6**

4 A* Algorithm **7**

5 Dijkstras Algorithm **8**

6 Bellmann Ford Algorithm **9**

7 Updating Weights **10**

8 Citing **12**

8.1 Dataset 12
8.2 **Pandas** : 12
8.3 **NumPy** : 12
8.4 **Matplotlib** : 12
8.5 **Folium** : 12
8.6 **Plotly Express** : 12
8.7 **OSMnx and NetworkX** : 12
8.8 **Pyproj** : 12
8.9 **GeoPandas** : 12
8.10 **PyTorch** : 12

1 Preprocessing

Optimizing Route Planning in Aarhus, Denmark: Utilizing OpenStreetMap Data : We demonstrated the process of optimizing route planning in Aarhus, Denmark(as we have the dataset of Aarhus itself), by leveraging OpenStreetMap (OSM) data. Here's a breakdown of the steps involved:

1. Coordinates Initialization:

   The latitude and longitude coordinates for Aarhus, Denmark, are initialized (56.1629, 10.2039).

2. Coordinate Projection:

   The coordinates are projected from WGS84 (EPSG:4326) to the appropriate Universal Transverse Mercator (UTM) zone (EPSG:32632) for Aarhus.

3. Street Network Data Download:

   Using the `osmnx` library, street network data for Aarhus is downloaded within a specified distance (10,000 meters) from the given coordinates. The network is simplified and filtered to include only drivable roads.

4. Projection and Network Enhancement:

   The downloaded street network data is projected to the UTM coordinate system. Edge speeds and travel times are added to the projected graph.

5. Plotting the Graph:

   The projected graph representing the street network of Aarhus is visualized using `osmnx`, with nodes represented as black dots.

6. Defining Start and Goal Nodes:

   Start and goal coordinates for route planning are specified. Nearest nodes on the street network graph corresponding to these coordinates are identified.

Overall, this code snippet serves as a foundation for further route optimization and navigation tasks within the city of Aarhus, utilizing accurate and detailed street network data from OpenStreetMap.

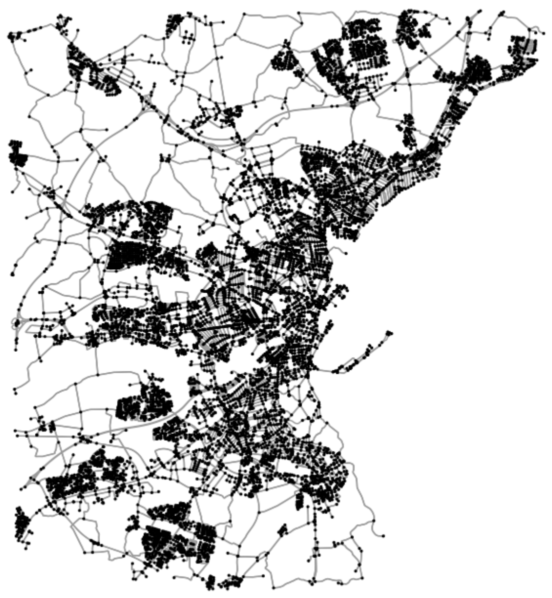

Figure 1: Aarhus, Denmark

2 Neural Networks

Multi-Task Neural Network for Traffic Prediction Using PyTorch : We implemented a multi-task neural network model using PyTorch for traffic prediction based on a dataset we made by combining the parts we found and only necessary columns are filtered, containing hourly traffic data. The dataset includes features such as month, day, hour, minute, and edge identifier, and the target variables consist of average speed, median measured time, and vehicle count.

Code Overview:

1. Data Loading: The dataset is loaded from a CSV file into Pandas DataFrame, and then split into input features (`X_traffic`) and target variables (`y_traffic`).
2. Data Preparation: The input features are converted to NumPy arrays and then PyTorch tensors (`X_traffic_tensor` and `y_traffic_tensor`).
3. Data Loading Utilities: PyTorch `TensorDataset` and `DataLoader` are utilized to create efficient data loading mechanisms for training.
4. Neural Network Architecture: The multi-task neural network (`MultiTaskNet`) is defined, consisting of shared layers and task-specific layers for traffic prediction.
5. Model Training: The model is trained using a defined loss function (MSE loss) and Adam optimizer over a specified number of epochs. Training progresses through batches of data loaded using the DataLoader.
6. Model Persistence: The trained model parameters are saved to a file (`MultiTaskNet_Model.pt`) for future use. If the model file exists, the pretrained model is loaded; otherwise, training starts from scratch.
7. Training Progress: The training loop iterates over epochs, printing the epoch number and total loss after each epoch.
8. Model Saving: In the first part of the code, the parameters of the PyTorch model named `model` are saved to a file named `'MultiTaskNet_Model_100.pt'`. This is accomplished using the `torch.save()` function, which takes two arguments: the model's state dictionary and the file path where the parameters will be saved.
9. Model Loading: In the second part of the code, a new instance of the `MultiTaskNet` model is instantiated. Then, the previously saved model parameters are loaded into this new model using the `load_state_dict()` function. This function loads the state dictionary from the file specified by the `model_path`.

Multi-Task Neural Network Training for Weather Prediction : We demonstrated the process of training a multi-task neural network using PyTorch for weather prediction based on a provided dataset.

1. Data Loading and Preprocessing:
   - The dataset containing weather data is loaded from a CSV file named `'combined_weather_data_new.csv'` using pandas.
   - The dataset is then split into input features (`X_weather`) and target variables (`y_weather`), which represent weather features and corresponding cluster labels, respectively.
2. Data Preparation: The input features and target variables are converted to numpy arrays and then to PyTorch tensors (`X_weather_tensor` and `y_weather_tensor`) for compatibility with PyTorch.
3. Model Architecture:
   - A multi-task neural network architecture (`MultiTaskNet`) is defined.
   - The model consists of shared layers followed by task-specific layers for weather prediction.
4. Model Training:
   - The model is instantiated, and a loss function (Mean Squared Error) and optimizer (Adam) are defined.
   - The training loop runs for a specified number of epochs.
   - Data is loaded in batches using `DataLoader`, and forward and backward passes are performed.
   - The loss is computed and backpropagated to update model parameters.
5. Model Training: The model is trained using a defined loss function (MSE loss) and Adam optimizer over a specified number of epochs. Training progresses through batches of data loaded using the DataLoader.
6. Model Persistence: The trained model parameters are saved to a file (`weather_model.pt`) for future use. If the model file exists, the pretrained model is loaded; otherwise, training starts from scratch.
7. Training Progress: The training loop iterates over epochs, printing the epoch number and total loss after each epoch.
8. Model Saving: In the first part of the code, the parameters of the PyTorch model named `model` are saved to a file named `'weather_model.pt'`. This is accomplished using the `torch.save()` function, which takes two arguments: the model's state dictionary and the file path where the parameters will be saved.
9. Model Loading: In the second part of the code, a new instance of the `MultiTaskNet` model is instantiated. Then, the previously saved model parameters are loaded into this new model using the `load_state_dict()` function. This function loads the state dictionary from the file specified by the `model_path`.

3 Analysis

Road Type Analysis in Graph G : The analysis focuses on extracting various attributes associated with each road segment, such as length, road type, speed limit, and OpenStreetMap (OSM) identifier (osmid).

1. Road Type Extraction: The code iterates over each edge in the graph `G` to extract information about the road type, length, speed limit, and OSM identifier.
2. Attributes Extraction: For each edge, it retrieves the length, road type (retrieved from the 'highway' attribute), speed limit (retrieved from the 'maxspeed' attribute), and OSM identifier ('osmid').
3. Data Collection: The extracted attributes are collected into lists for further analysis. Here, a list named `road_types` is initialized to store the road types encountered in the graph.
4. Data Analysis: After collecting road types, the code constructs a Pandas Series (`road_types`) and performs a value count analysis to understand the distribution of different road types in the graph `G`.
5. Output: The output of the analysis provides insights into the frequency of each road type present in the road network represented by graph `G`. This information can be valuable for various applications, such as route planning, traffic management, and urban infrastructure development.

By executing this code, one can gain a comprehensive understanding of the road types comprising the road network, facilitating informed decision-making processes in urban planning and transportation management.

We utilized predictive models trained on historical data to forecast these attributes for a given location and time. The predicted attributes are then classified into categories of "GOOD," "MODERATE," or "BAD" based on predefined thresholds, enabling efficient decision-making in route selection.

Road Quality Classification: The `roads_condition_to_hml_values` function categorizes road types into three classes: "GOOD," "MODERATE," or "BAD." Roads such as motorways, primary, and trunk roads are considered "GOOD," while secondary and tertiary roads fall under the "MODERATE" category. Residential, unclassified, and disused roads are classified as "BAD."

Traffic Intensity Classification: The `traffic_to_hml_values` function assigns traffic intensity into three categories: "GOOD," "MODERATE," or "BAD." Low traffic conditions are labeled as "BAD" if predicted traffic volume is less than or equal to 6, and "GOOD" for values greater than 15. Otherwise, it is classified as "MODERATE."

Weather Condition Classification: The `weather_to_hml_values` function categorizes weather conditions as "GOOD," "MODERATE," or "BAD." Predicted weather attributes falling below 0.5 are considered "GOOD," while those exceeding 1.5 are classified as "BAD." Conditions between 0.5 and 1.5 are labeled as "MODERATE."

4 A* Algorithm

A* Algorithm with Edge Information for Route Planning : The A* algorithm augmented with edge information is used to facilitate route planning in transportation networks. The algorithm utilizes graph representations of road networks, where nodes represent locations, and edges represent road segments connecting these locations. Key functionalities and components of the code include:

1. Haversine Distance Calculation: The `haversine_distance` function computes the great-circle distance between two points on the Earth's surface using the Haversine formula. This distance metric is commonly used for calculating distances between latitude and longitude coordinates.

2. *A Algorithm Implementation**: The `astar_with_edge_info` function implements the A* algorithm to find the shortest path from a start node to a goal node in the graph. The algorithm employs a priority queue (implemented using heapq) to efficiently explore the graph by prioritizing nodes based on a combination of the current cost and the estimated remaining cost (heuristic).

3. Edge Information Retrieval: During the A* search, the algorithm keeps track of additional edge information, including edge IDs (OSMIDs), edge weights (lengths), and road quality (e.g., highway classification). This information is extracted while reconstructing the path from the goal node back to the start node after the search is completed.

4. Output and Visualization: The code outputs the found path, edge IDs, edge weights, and road types if a valid path is found. Otherwise, it indicates that no path exists between the specified start and goal nodes.

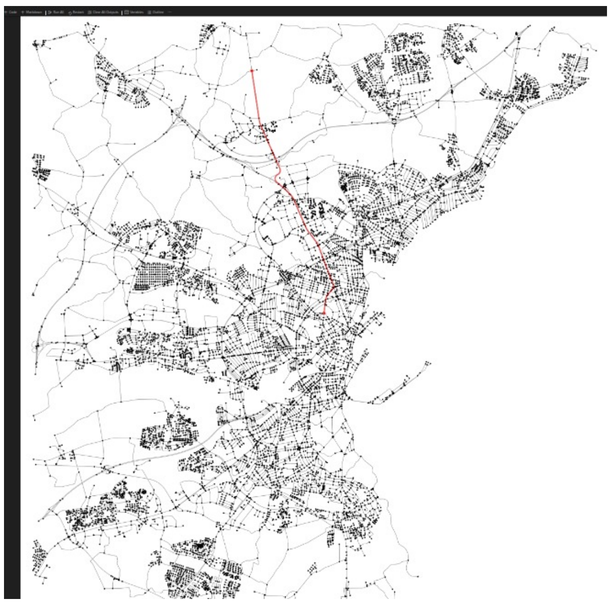

Figure 2: A* Route

5 Dijkstras Algorithm

Dijkstra Algorithm with Edge Information for Route Planning : This Algorithm generates the shortest path in a given graph (`G`). It calculates various attributes of the edges along the path, such as edge weights (length), edge IDs, and road quality.

Code Breakdown:

- `path_dijkstra`: Computes the shortest path using Dijkstra's algorithm from a specified start node to a goal node in the graph `G`, considering edge lengths as weights.
- `edge_weights_dijkstra`: Creates a list of edge lengths for each edge in the shortest path obtained from Dijkstra's algorithm.
- `edges_id_dijkstra`: Gathers the unique identifiers (OSM IDs) for each edge in the shortest path.
- `roads_quality_dijkstra`: Collects the road quality information (if available) for each edge in the shortest path.

Usage: This code segment can be used in applications that require detailed analysis of the shortest path generated by Dijkstra's algorithm. It provides insights into the characteristics of the edges along the shortest path, such as their lengths, identifiers, and road quality, which can be valuable for route planning and optimization tasks.

Dependencies: NetworkX: For graph manipulation and shortest path algorithms.

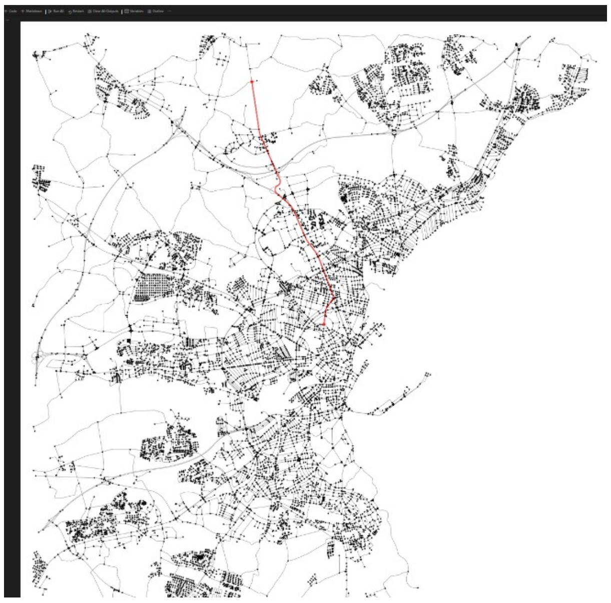

Figure 3: Djikstra Route

6 Bellmann Ford Algorithm

Bellmann Ford Algorithm with Edge Information for Route Planning :

Here we analyzed the route computed using the Bellman-Ford algorithm in order to gain insights into the path's characteristics and the quality of roads traversed.

Code Breakdown:

- `path_bellman_ford = nx.bellman_ford_path(G, source=start_node, target=goal_node, weight='length`
  This line computes the shortest path using the Bellman-Ford algorithm between the specified `start_node` and `goal_node`, considering the weight attribute as 'length', which typically represents the distance of edges.

- `edge_weights_bellman_ford = [G[path_bellman_ford[i]][path_bellman_ford[i+1]][0].get('length', 1`
  Here, we extract the length (weight) of each edge in the computed path. This is achieved by iterating over each pair of consecutive nodes in the path and retrieving the 'length' attribute of the corresponding edge in the graph. If the 'length' attribute is not available for an edge, a default value of 1 is used.

- `edges_id_bellman_ford = [G[path_bellman_ford[i]][path_bellman_ford[i+1]][0]['osmid'] for i in r`
  This line retrieves the unique OpenStreetMap IDs (osmids) of the edges along the computed path. Similar to the previous step, it iterates over each pair of consecutive nodes in the path and extracts the 'osmid' attribute of the corresponding edge.

- `roads_quality_bellman_ford = [G[path_bellman_ford[i]][path_bellman_ford[i+1]][0].get('highway',`
  Here, we determine the quality of roads traversed in the computed path. It extracts the 'highway' attribute of each edge along the path, which typically represents the road type or quality. If the 'highway' attribute is not available for an edge, it defaults to 'Unknown'.

Dependencies: NetworkX: For graph manipulation and shortest path algorithms.

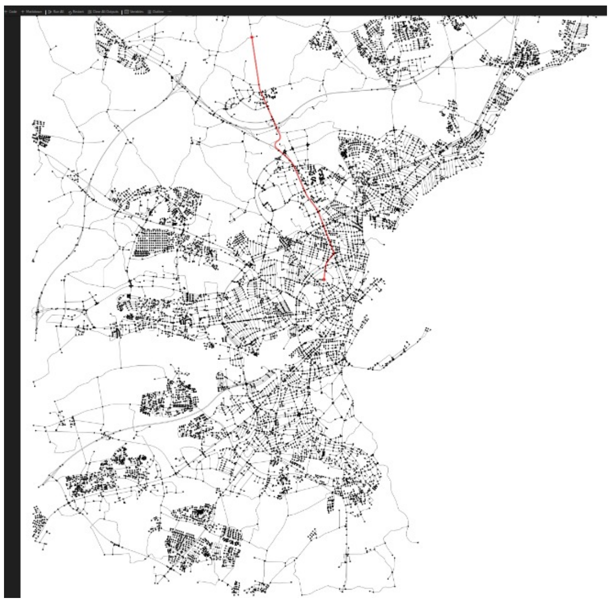

Figure 4: Bellmann Ford Route

7 Updating Weights

The algorithm consists of three main cases, each addressing different combinations of traffic and weather conditions:

1. High Traffic (GOOD) with Bad Weather: In this scenario, where traffic is high and weather conditions are bad, the algorithm decreases the speed by a certain factor to reflect the impact of both factors on travel time.
2. Moderate Traffic (MODERATE) with Various Weather Conditions: When traffic is moderate, the algorithm considers both road conditions and weather. Depending on the road condition (GOOD, MODERATE, or BAD) and the weather condition (GOOD or BAD), the speed is adjusted accordingly to account for their combined effect on travel time.
3. Low Traffic (BAD) with Various Weather Conditions: In the case of low traffic, road conditions and weather are still taken into consideration. Similar to the previous case, adjustments are made to the speed based on the road condition and weather condition to estimate travel time accurately.

The algorithm computes the new travel time based on the adjusted speed and returns it as the output. This updated travel time can then be used in route optimization algorithms such as A* (Astar), Dijkstra's, or Bellman-Ford to find the most efficient route considering the dynamic traffic and weather conditions.

Python function `calculate_updated_weights` aims to dynamically update edge weights in a graph based on predicted conditions such as road quality, weather, and traffic. This function integrates machine learning models for predicting weather and traffic conditions and adjusts edge weights accordingly.

1. Function Definition:
   - The function `calculate_updated_weights` is defined to calculate and return updated weights for a list of edges in a graph.
   - Inputs to the function include:
     - `weights_list`: A list containing the original weights of the edges.
     - `rd_qualities`: A list containing the road qualities for each edge.
     - `edges_osmid`: A list containing the OSM IDs of the edges.
     - `year`, `m`, `dy`, `hr`, `min`, `pr`, `t`, `h`, `v`, `win_dir`, `win_s`: Parameters representing the year, month, day, hour, minute, pressure, temperature, humidity, visibility, wind direction, and wind speed respectively.
   - The function returns:
     - `new_weights`: An array containing the updated weights for each edge.
     - `new_time`: A tuple representing the updated date and time after processing all edges.
2. Iterating Over Edges:
   - The function iterates over each edge in the graph, processing one edge at a time.
   - Inside the loop, it extracts information such as distance, road quality, and edge ID for each edge.
3. Predicting Weather Conditions:
   - Weather conditions are predicted using a machine learning model (`model_weather`) trained on weather-related features.
   - Features such as month, day, hour, minute, pressure, temperature, humidity, visibility, wind direction, and wind speed are used to make predictions.
   - The predicted weather condition is then converted to a human-readable format (`weather_cond`).

4. Predicting Traffic Conditions:

    - Traffic conditions are predicted using another machine learning model (`model`) trained on traffic-related features.
    - Features such as month, day, hour, minute, and edge ID are used to make predictions.
    - The predicted traffic condition and average speed are extracted from the predictions.
    - The average speed prediction is converted to meters per minute (`avgSpeed_pred`).

5. Updating Edge Weights:

    - The `update_that_edge_weight` function is called to update the edge weight based on predicted conditions such as road quality, weather, and traffic.
    - The updated weight is then appended to the `new_weights` array.

6. Time Update: After processing each edge, the function updates the date and time based on the calculated travel time (`updated_weight`).
7. Returning Results: Finally, the function returns the array of updated weights (`new_weights`) and the updated date and time (`new_time`).

Route Selection Based on Algorithm Performance The code calculates the total time required for each route generated by the Astar, Dijkstra's, and Bellman-Ford algorithms. These times are computed by summing up the weights of the edges in the respective shortest paths. Then, it selects the route with the shortest total time among the three and visualizes it on a map using the `ox.plot_graph_routes` function.

1. Calculating Total Time: The total time for each route is computed by summing up the weights of the edges in the shortest path obtained from each algorithm.
2. Selecting the Best Route: The code determines the route with the shortest total time among the three algorithms.
3. Visualization: Finally, the shortest route is visualized on a map, highlighting it in red color.

By considering factors such as road conditions, traffic intensity, and weather predictions, we ensure that users are provided with the most suitable route option to reach their destination in the shortest possible time.

8 Citing

8.1 Dataset

Traffic Dataset **Publisher:** University of Surrey **URL:** http://iot.ee.surrey.ac.uk:8080/datasets.html#traffic

Weather Dataset **Publisher:** University of Surrey **URL:** http://iot.ee.surrey.ac.uk:8080/datasets.html#weather

OPENSTREET MAP: https://www.openstreetmap.org/

8.2 Pandas :

https://pandas.pydata.org/

8.3 NumPy :

https://numpy.org/

8.4 Matplotlib :

https://matplotlib.org/

8.5 Folium :

https://pypi.org/project/folium/

8.6 Plotly Express :

https://plotly.com/python/plotly-express/

8.7 OSMnx and NetworkX :

https://osmnx.readthedocs.io/

8.8 Pyproj :

https://pypi.org/project/pyproj/

8.9 GeoPandas :

https://geopandas.org/

8.10 PyTorch :

https://pytorch.org/docs/stable/index.html