Network Model Generation

ADVANCED SIMULATION

EPA133a

Authors:

Bettin Lorenzo (6132928)
Dell'Orto Alessandro (6129161)
Le Grand Tangui (6172075)
Precup Ada (5240719)
van Engelen Ralph (4964748)

March 21, 2025

TU Delft Delft University of Technology

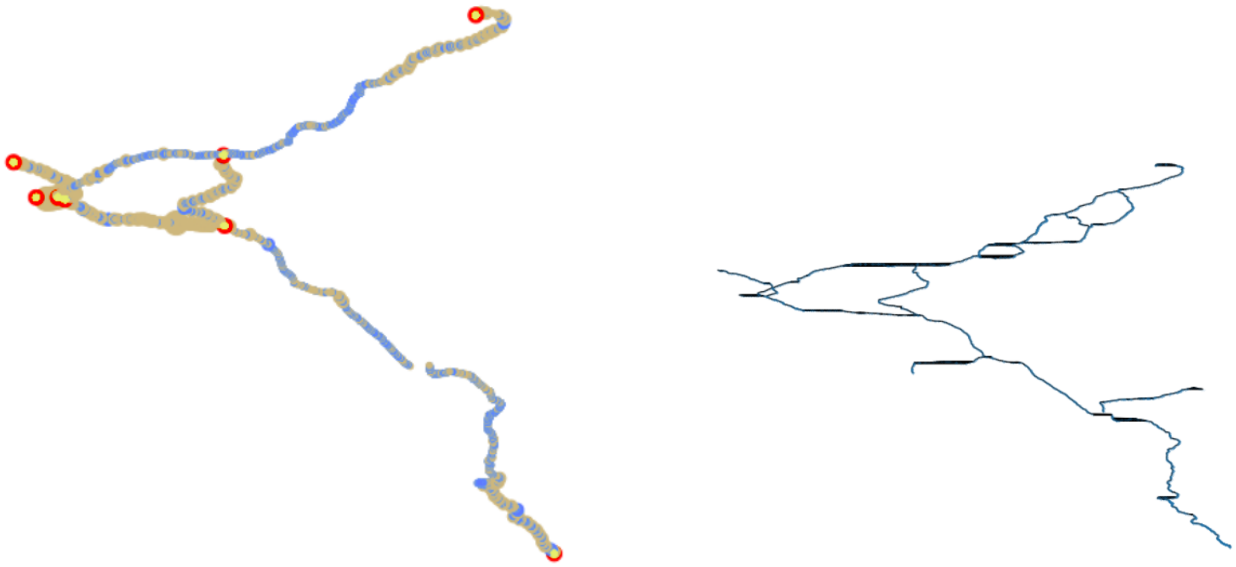
Contents

Introduction

Assignment 2 focused on creating a Mesa network for Bangladesh's N1 road between Dhaka and Chittagong, incorporating components like bridges and vehicles in the process. This assignment expands on this foundation. New roads and intersections are added, and new routing logics are elaborated.

This is done through a multi-modelling approach, using the NetworkX package, designed for complex network design. We create a NetworkX model, which allows us to calculate the shortest path between points, which can then be used in our Agent-Based simulation for routing.

This report details how the model was updated to work with multiple roads and with NetworkX. Then, the model is experimented in for different scenarios, to study what parts of the network might be most vulnerable to traffic disruption. Finally, these results are discussed and expanded on in relation with relevant literature.



(a) The road network as generated with mesa.

(b) The road network as generated with Networkx.

Figure 1: Relationship between bridge breakdown and traffic.

1 Methodology

1.1 Data Cleaning

The code processes a road dataset (roads3.csv) to ensure that only roads meeting specific criteria, namely, roads N1, N2, and any side road longer than 25 km are included in the final output.

The code begins by reading a CSV file containing road data and ensuring that the “chainage” column is treated as a floating-point number for accurate distance calculations. It then groups the data by each road and computes the total length of each road by subtracting the minimum chainage value from the maximum. This calculated length is added back to the dataset as a new “length” column.

Next, the code filters the dataset to retain only those roads with a total length exceeding 25 km.
This filtering step meets the requirement of including major roads (N1 and N2) and side roads that are longer than 25 km. When applied to the dataset, the filtering results in the inclusion of roads such as ’N1’, ’N102’, ’N104’, ’N105’, ’N106’, ’N2’, ’N204’, ’N207’, and ’N208’.

N1 and N2 are the primary roads along which traffic circulates. The rest are side roads considered sufficiently long for analysis, and which intersect with N1 and N2.

In a subsequent phase of the analysis, certain road segments were identified as lacking intersections, which introduced computational errors during model execution. To ensure the integrity and reliability of the simulation, these segments were excluded from the analysis. Some bridge segments were also found to have null values, in which case the average bridge length in the network was taken as a replacement value. These changes ensure the model functions as expected.

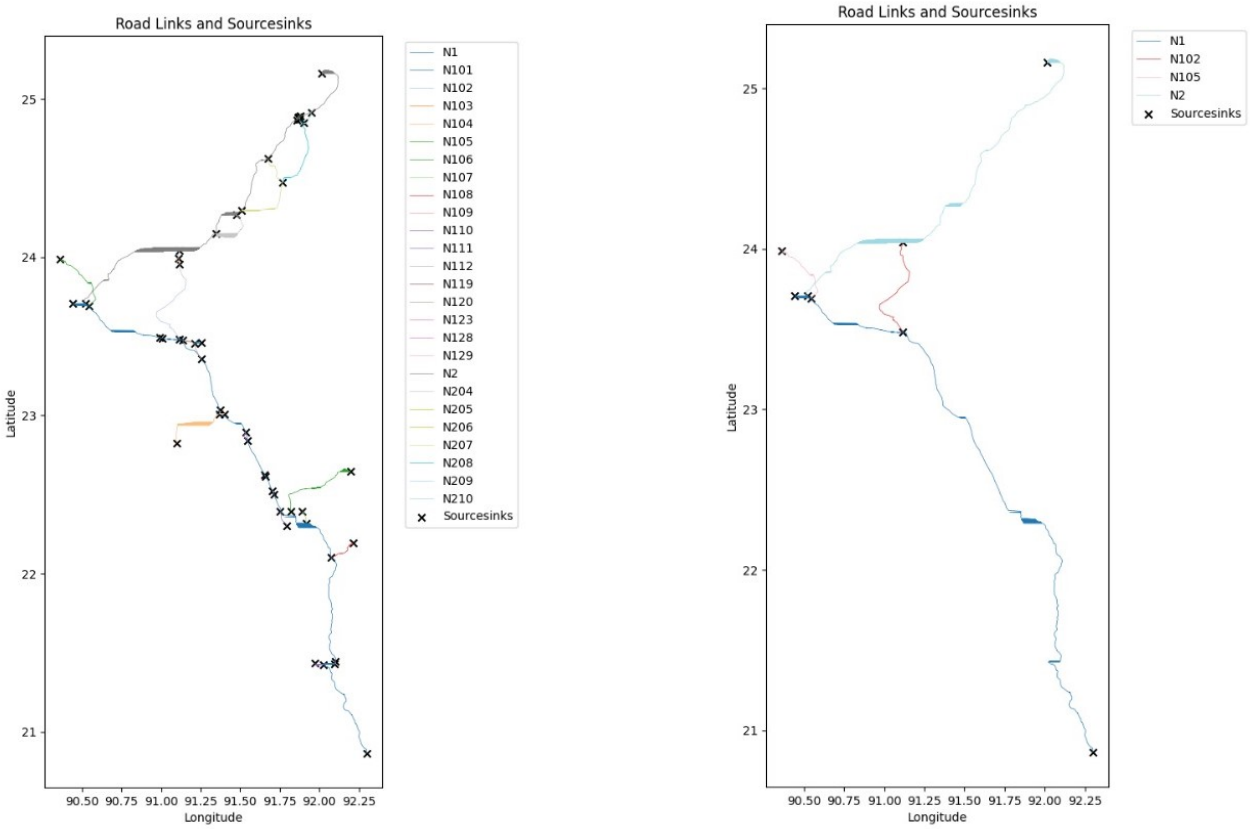


Figure 2: The road network pre and post additional cleaning

1.2 Intersection handling

A script was made that puts the intersections of the roads at the right locations. This is vital to connect the multiple roads to eachother, in contrast to previous assignment where only one road was used. The data preparation builds upon the data preparation script (data_pull) from assignment 2.

1.2.1 Cleaning and Ordering Road Data

The function `remove_misplaced_objects(input_data)` was made to ensure that all entries are ordered properly and that the sourcesink objects (which represent the entry and exit points for vehicles on a road) are correctly positioned at the beginning and end of each road segment in the input data.

First, the function sorts the dataset by road name, latitude, and longitude. This is done to make sure that the data is ordered geographically for each road. Once sorted, each road is isolated and analyzed independently.

Within each road segment, the function identifies all "sourcesink" objects. If only one "sourcesink" exists, it is retained at the beginning of the road. If multiple "sourcesink" objects are found, the first and last ones in the sequence are preserved, while any additional "sourcesink" objects are removed (there should be none anyways). The remaining road elements, such as links and bridges, are placed between these boundary points in their original order.

This step is essential to prevent errors in vehicle routing. A misplacement of "sourcesink" objects could cause roads to be disconnected from the simulation, preventing vehicles from entering or exiting the network at the correct points.

1.2.2 Assigning Unique Identifiers to Intersections

The function `assign_intersection_ids(input_data)` was made to ensure that intersections with the same last word in the name and road get the same ID. This is necessary to correctly represent the points where roads connect, allowing vehicles to transition between roads without inconsistencies.

Each intersection has a name that contains both road names involved in the intersection. The function extracts the last word in the name string and compares it with other intersections to determine whether two intersection points are describing the same location.

Once matching intersections are identified, the function ensures they share the same ID. If two intersections are found to represent the same physical crossing, the first-encountered ID is assigned to the second occurrence. This step avoids duplicate intersection entries in the dataset, preventing errors where the same location is treated as two separate intersections.

This process is crucial because mislabeling intersections would cause roads to appear disconnected, disrupting the shortest path calculations used in vehicle routing.

1.2.3 Detecting and Inserting Intersections

The function `find_and_insert_intersections(input_data)` is made to identify locations where roads intersect and for inserting intersection points at appropriate positions in the dataset.
First, the function extracts the latitude and longitude of all road elements and stores them in arrays for distance calculations. Then each point is checked against every other point in the dataset using a pairwise comparison. If two points belong to different roads and are within a predefined distance threshold, an intersection is detected. The threshold was put very high because N106 was intersecting N1 in Chittagong and there is a big empty space there).

For each detected intersection, the function assigns a unique intersection ID and determines the precise location of the intersection. To approximate its position, the function calculates the midpoint between the two closest road points. This ensures that intersections are inserted at a realistic location without distorting the original road structure.

To prevent duplicate intersections, a set of previously detected intersections is made. This avoids duplicate entries, which would otherwise create inconsistencies in the network.

Once intersections are inserted, the dataset is further updated by updating IDs to maintain sequential numbering. This ensures that intersections appear in the correct order when roads are processed.

This step is essential because intersections enable vehicle movement between different roads. Without correctly identified intersections, the road network would be fragmented, preventing vehicles from reaching their destinations efficiently

1.3 Model Changes

Several things needed to be changed and added into `model.py` accordingly to this assignment. In assignment 2, vehicles always took the same path from one source to one sink. In this assignment, the vehicles can leave from any random source or sink and go to any random source or sink. Moreover, the intersections on the roads enable the vehicles to go from one road to another.
To reach their destination they take the shortest path. This was added to the model using NetworkX.

1.3.1 Construction of NetworkX Graph

The NetworkX graph was computed in the init of `class BangladeshModel(Model):` as `self.G = nx.Graph()` to ensure that the graph generates when the model is generated and so it is accesible by both the shortest path function and the generate model function.

The function `generate_networkx_model()` first reads the csv file containing the input data and makes a dataframe. It then iterates over the rows of the dataframe and adds a node to the graph, using the NetworkX command `add_node()` where the node_id is the id part of the row, and the attributes contain the lon and lat part of the row.

Then, the function iterates through the dataframe again to add edges between nodes that belong to the same road. This is achieved using the `add_edge()` function in NetworkX, which establishes a connection between two consecutive nodes if they share the same road identifier.

The weight of the edge is set to the length of the road. This weight is later needed to compute the shortest path between nodes. A visual representation of the network is showcased in figure 3

After adding all nodes and edges, the function extracts the positions of all nodes using the `get_node_attributes()` function. These positions are later used for visualization and further processing.

1.3.2 Construction of the Shortest Path Function

The function `get_shortest_path()` is responsible for computing the shortest route between a given source and sink node within the network.

When the function is called, checks the inputted source and sink against the `path_ids_dict` dictionary to determine if the shortest path for the given pair has already been computed. If the path exists in the dictionary, this path is returned and used. This makes sure that the shortest path does not need to be calculated every time a vehicle takes the same route.

If the path is not found in the dictionary, the function computes the shortest path using the `networkx.shortest_path()` method. This method takes the graph, source, and sink as inputs and uses the edge weights to determine the optimal path. Once the shortest path is obtained, it is stored in `path_ids_dict` as a `pandas.Series`. The function then returns the dictionary.

The function checks if no paths exist with the `NetworkXNoPath` exception. Instead of causing an error, it prints a message that no valid path is available and returns `None`. This prevents the simulation from crashing.

1.3.3 Changing the get_route function

The `get_route()` function, which uses a source as input to get a route from that source, first used the `get_straight_route()` function. This function used a sink to compute a straight path. This is changed to

use the `get_shortest_path()` function. A while loop is computed in which a random sink is choosen. If the sink is the same as the input source, the loop is broken, if not the funtion returns `get_shortest_path()`, which returns the dictionary in which the shortest route between the source and sink is stored.

Figure 3: The road network as generated with NetworkX of the relevant roads.

2 Results

Table 1 presents five simulation scenarios, each characterized by its average travel time, in minutes (Avg travel time), the total number of trucks processed (N. of trucks), and the number of broken bridges (N. of broken bridges).
All the shown values are averaged over 10 iterations with different seeds

Scenario	Avg travel time (minutes)	N. of trucks	N. of broken bridges	Avg delay time (minutes)
Scenario 0	312	1378	0	0
Scenario 1	315	1377	0.1	0
Scenario 2	315	1377	0.1	0
Scenario 3	362	1360	2.1	0
Scenario 4	386	1353	3.0	0

Table 1: Summary of Scenarios

Table will be updated as soon as simulation is completed.

3 Bonus Assignement

3.1 Methods

For the Bonus Assignment, intersections are computed using two distinct approaches.
The approximate method, which is represented by blue “X” markers, uses a threshold-based approach that checks whether two road points are within a specified distance; if they are, it inserts an intersection at the midpoint between these points. As a result, we can see fewer blue markers, and in some cases, they may appear slightly off from where the roads actually intersect, especially if the true crossing is located between surveyed points or on a curve.
In contrast, the accurate method leverages GeoPandas and Shapely to analyze the full geometry of each road segment; when two LineStrings intersect, the precise crossing point is extracted, resulting in a dense cluster of red dots that align closely with the actual visual intersections, even on curved or angled roads.

3.2 Observations

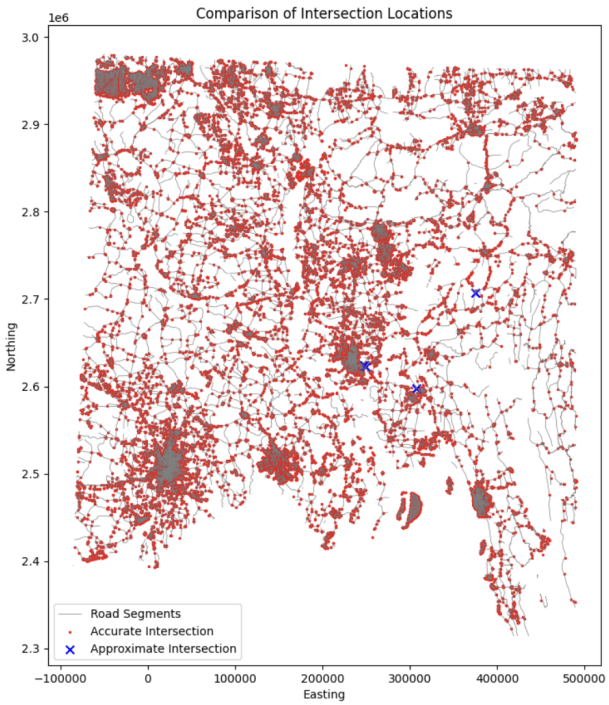
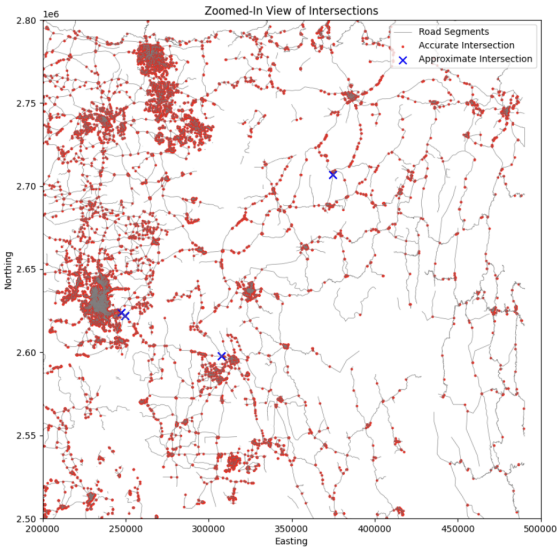
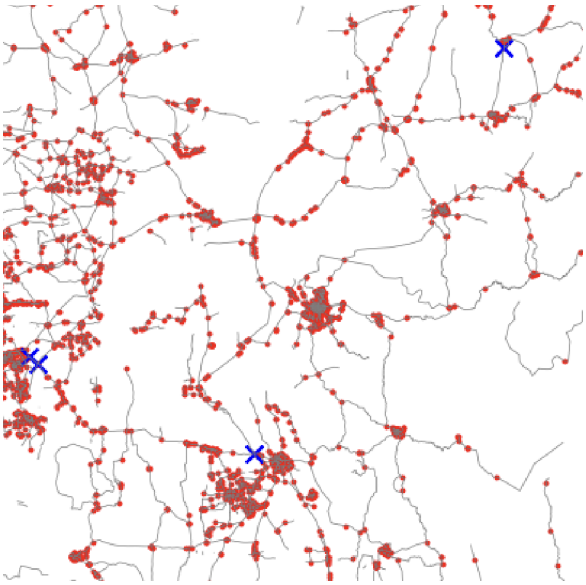


Figure 4: Comparison of Intersection Locations.

Observations from the plots are useful as they reveal that in a full map view of the road network (displayed in gray), the accurate intersections form a high-density cluster wherever roads meet, while the approximate intersections are more sparsely distributed and may miss or slightly misplace some of the subtler junctions.



(a) First zoomed-in view.



(b) Second zoomed-in view.

Figure 5: Zoomed-in Views of location intersections.

As it is possible to notice from Figure 3, in a zoomed-in view, the reduction in road segments makes it easier to discern that the red dots accurately capture the crossing points, whereas the blue markers, as it better noticeable in Figure 3(b), based on the midpoint approximation, sometimes sit off the true intersection line.
This difference in placement becomes particularly evident when the axes are restricted to a specific region, highlighting the discrepancies between the two methods.

3.3 Reflections

From a practical perspective, the approximate method is fast, straightforward, effective, and suitable for obtaining a quick overview of intersections, although it can omit real intersections if no points are sufficiently close and might produce inexact placements in curved or angled situations. Conversely, the accurate method, while computationally more intensive and potentially more complex to manage when handling overlapping segments or multi-geometries, provides precise crossing points that are essential for in-depth spatial analysis, traffic modeling, and reliable route planning.
To conclude, while approximate intersections may be sufficient for preliminary exploration or coarse decision-making, accurate intersections are critical for applications where precise network connectivity is paramount.

4 Discussion and Conclusion

5 Acknowledgments

5.1 Task Division

During the extension of this assignment, the tasks were equally and strategically divided within the team's member as following.

- Data cleaning: Lorenzo
- Data collector: Ada
- Data intersections: Alessandro and Ada
- Model edits: Ralph and Tangui
- Simulation: Alessandro
- Bonus Assingment: Lorenzo
- Report: Lorenzo, Ralph and Tangui

5.2 AI Acknowledgment

ChatGPT, GitHub Copilot and DeepSeek were used to assist in the programming part of this assignment. More specifically to convert flow code into working code and for debugging. ChatGPT was also used for formatting and changing texts in the writing of this report.