Origin to Destination Route Planning in a Road Network - Project Report

Mingxiao Wei, Zheyuan Zhang

April 2024

1 Introduction

Road transportation plays a prominent role in the present-day transportation system. Freight transported by trucks accounts for 62% of the total tonnage of freight movement in the US in 2020 [1], and motor vehicles traveled 3.17 trillion vehicle miles on US roads in 2022 [2]. Due to the dominant role of road transportation, it arises as a significant objective for motor vehicle users to determine an optimal route to travel from a given origin to a given destination via a system of roads that minimizes travel costs, such as distance or time. A variety of algorithms for planning routes in a network of roads have been proposed and applied.

In this project, we consider the network of roads and cities as a weighted graph, in which a number, namely weight, is associated with each edge to represent a quantity such as distance or cost [3]. In the graph we are building for road transportation, each vertex represents a junction between segments of roads, each edge represents a segment of road between 2 junctions, and the weight associated with an edge represents the length or cost of the road segment represented by the edge. Due to the time complexity of shortest path algorithms, usually $\Omega(V \log_E)$, we also added pruning operations so that the running time will be reduced significantly. Specifically, we "deleted" various roads inside a city and used Dijkstra [4] to find the sub-optimal origin-to-destination route in a road network amounts to minimizing the total time of a path in a weighted graph between two given vertices.

2 Instructions for Running

The datasets `ORN_Road_Elements.geojson.zip` and `ORN_Road_Segments.geojson.zip`, as well as the `distance_weighted_graph.txt.zip` and `travel_time_weighted_graph.txt.zip`, were sent to csc111-2023-01@cs.toronto.edu via UTSend.
The `ORN_Road_Elements.geojson.zip` and `ORN_Road_Segments.geojson.zip` are intended to be placed in a `data` directory.

3 Computational Overview

3.1 Dataset Description

The dataset we used for this project is a dataset containing information on the complete road network of the Ontario province. This is different from the tentative decision of using the US highway network mentioned in the proposal. The dataset is downloaded from the website Ontario GeoHub, a platform containing a variety of authoritative geospatial datasets. The road network is represented by 2 files, namely `ORN_Road_Elements.geojson` and `ORN_Segments.geojson`, where the `geojson` format is a Javascript object notation-based format for representing simple geographical structures.

The `ORN_Road_Elements.geojson` file represents the whole Ontario road network as a collection of road elements. Approximately 5.8×10^5 road elements covering a total distance of over $2.5 \times 10^5 km$ are stored in this file. It is guaranteed that no intersection exists in the interior of any road element, and the endpoints of a road element are defined as junctions. Note that different lanes of the same road may be represented as distinct road elements. Two road elements intersect if and only if they have a shared junction. Every road element or junction is assigned a unique `int` provincial identifier `ogf_id`. The geometric information of a road element is digitalized as a list of longitude-latitude coordinates of points on the road element in a specific order. The first and the last element in the list of coordinates are the coordinates of the 2 junctions that act as the 2 endpoints of the road element. The allowed direction of traffic flow is given by one of the strings `"Both"`, `"Positive"`, or `"Negative"`, where `"Both"` indicates that traffic flows in both directions are allowed, `"Positive"` indicates that only traffic flow in the same direction as the direction of digitalization is allowed, and `"Negative"` indicates that only traffic flow in the direction opposite to the direction of digitalization is allowed.

The `ORN_Segments.geojson` file stores a further segmentation of the road network as a collection of road segments. Every road segment is part of a road element, and every road element corresponds to one or more road segments. In other words, let S be the set of all road segments, and let E be the set of all road elements, then the function $\phi : S \to E, s \mapsto (e|s \text{ is geometrically part of } e)$ is a well-defined, surjective, but not necessarily injective function. There are a total of approximately 6.2×10^5 road segments in the file, which means most of the road elements correspond to a single road segment while a minority of them are subdivided into multiple road segments. In the `ORN_Segments.geojson` file, every road segment has a property `ROAD_NET_ELEMENT_ID` indicating the `ogf_id` of its corresponding road element. The length, road class, road name, and speed limit of every road segment are also given in the file. The road class of a road segment is one of the following 12 classes: `Local / Street`, `Local / Strata`, `Arterial`, `Collector`, `Resource / Recreation`, `Expressway / Highway`, `Alleyway / Laneway`, `Ramp`, `Freeway`, `Local / Unknown`, `Service`, `Rapid Transit`, and `Winter`. The representation of the geometric information of a road segment is in the same way as that of a road element, except that the first and the last elements in the list of coordinates are not necessarily the coordinates of junctions. This is because a segment can exist completely in the interior of a road element that is subdivided into multiple segments.

In summary, the `ORN_Road_Elements.geojson` file completely represents the connectivity of the road network, since no junction can exist in the interior of a road element, while the `ORN_Segments.geojson` file completely represents the road conditions such as road class and speed limit, since these conditions do not vary within each segment. While the subdivision of road elements into road segments might appear to be seemingly redundant, it has the crucial effect of capturing fine and local variations in the conditions of the roads.

3.2 Graph-Theoretic Representation of Road Network and Data Preprocessing

Since the dataset imposes restrictions on the direction of traffic flow of a road, then a **directed** weighted graph is needed to represent the road network. Formally, a directed weighted graph can be defined as a tuple $G := (V, E, w)$ where V is the set of all vertices, $E \subset V^2$ is the set of **ordered** pairs (u, v), where $u, v \in V$, such that there exists a directed edge from u to v if and only if $(u, v) \in E$, and $w : E \to \mathbb{R}$ is the function that assigns a real-valued weight to each edge. This is different from the idea of using an undirected weighted graph mentioned in the proposal. Every junction of the road network can be represented as a vertex and every road element can be represented as an edge in a weighted directed graph. For 2 junctions u and v in the road network, there exists an edge (u, v) in the graph if and only if a road element has u and v as its endpoints and traveling from u to v is an allowed direction of traffic flow.

The classes and methods for the representation and computation of the road network is stored in a module `graph_utils.py`. A `Graph` class is used to implement the **directed weighted** graph representing the road network. The classes `_Vertex`, `_Edge`, and `_Segment` are used to implement a vertex, an edge, and a representation of a road segment respectively. The `Graph` class has the following 2 instance attributes: `_vertices: dict[int, _Vertex]` that maps the `ogf_id` of a vertex to the corresponding vertex object, and `_edges: dict[tuple[int, int], _Edge]` that maps a tuple `(start_id, end_id)` to the corresponding `_Edge` representing the directed edge from the vertex with ogf_id `start_id` to the vertex with ogf_id `end_id`. The `_Vertex` class has the following 5 instance attributes: `junc_id: int` that stores the `ogf_id` of the vertex, `upstream: set[_Vertex]` that stores the set of vertices $\{v \in V : (v, \mathit{self}) \in E\}$, `downstream: set[_Vertex]` that stores the set of vertices $\{v \in V : (\mathit{self}, v) \in E\}$, `coordinates: list[int | float]` that stores the coordinates of this vertex, and `message: str` that stores a message that appears with this vertex for future visualization. The `_Edge` class has the following 5 instance attributes: `start_id: int` and `end_id: int` to represent an edge (u, v) for which `u.junc_id == start_id` and `v.junc_id == end_id`, `ogf_ids: set[int]` to represent the set of the `ogf_id` of all road elements in this edge, `segments: set[_Segment]` to represent the set of all segments corresponding to a road element in this edge, and `info: dict[str, float]` that maps a `str` key of weight type to the weight of that type. In this project, the only possible weight types are `"distance"` and `"travel_time"`. The set of `ogf_id` of all the road elements in an edge is stored as a set rather than a single `int` value because in the future steps of pruning and removing redundant edges, certain edges might be merged together to form a longer edge that represents a continuous long road element consisting of multiple original road elements. The `_Segment` class has the following 6 instance attributes: `corr_ogfid: int`, `name: str`, `seg_length: float`, `road_class: str`, `speed_limit: int`, and `coordinates: list[int | float]`. `corr_ogfid` stores the `ogf_id` of the corresponding road element of this road segment; `name` stores the road name of the road segment; `seg_length` stores the length of the road segment; `road_class` stores the road class of the road segment; `speed_limit` stores the speed limit of the road segment; and `coordinates` stores the digitalization of the geometric information of the road segment as a list of coordinates of points on the road segment in the way previously described.

The functions for data preprocessing and the construction of the graph from datasets are stored in a module `preprocessing.py`. The `data_to_graph` function is responsible for constructing a graph from the 2 datasets. The `data_to_graph` function loads data from the json files using the `json.load` function in the `json` python package, which converts a `json` file into a python `dict`. The `data_to_graph` function first reads the `ORN_Segments.geojson` file to build a `dict` variable `id_to_segment_info` that maps the `ogf_id` of each road element to a list of tuples containing the length, road class, speed limit, coordinates, and road name information of corresponding road segments. Then the function initializes a graph and reads the `ORN_Road_Elements.geojson` file to add vertices and edges to the graph based on the connectivity information given by the road elements data. A method `add_vertex` is used to add vertices to the graph and a method `add_edge_with_segments` is used to add an edge that contains certain segments. The properties of the segments corresponding to an edge with `ogf_id` can be looked up in the `id_to_segment_info` variable stored previously and passed as a parameter into the `add_edge_with_segments` method. It is possible that multiple road elements share a common pair of endpoints. This phenomenon can be detected when an edge to be added has a `(start_id, end_id)` pair that already exists in the `_edges` instance attribute of the graph. When this happens, the weights of the new edge and the existing edge are compared and the edge with a smaller weight is kept.

The graph describing the full road network has approximately 4.2×10^5 vertices and 1.0×10^6 edges. Due to the immense size of the graph, only dozens of selected vertices of interest given by the `SELECTED_DESTINATIONS` global variable in the main block will serve as the destinations available for the user to choose from. Given the immensity of the graph size, pruning the graph is necessary for reducing the running time of route planning to the order of a few seconds. The pruning algorithm is perhaps the most challenging part of this project. The objective of pruning is to remove the roads that are not significant for the overall connectivity of the road network or are rarely used for long-distance traveling, such as roads of the `Local / Street` class, and keep the roads crucial for the overall connectivity of the road network, such as roads of the `Highway / Expressway` or `Arterial` classes. Fortunately, the road segments belonging to the `Local / Street` class account for about 50% of the total number of road segments in the road network, so a desirable effect of pruning can be achieved by removing edges belonging to relatively trivial classes. The road classes to be pruned are stored in the `PRUNED_CLASSES` global variable in the `main` module and it currently includes the following 3 classes: `Local / Street`, `Local / Strata`, and `Local / Unknown`. However, there are some cases when the roads belonging to the classes to be pruned actually play an indispensable role in connecting 2 otherwise completely isolated networks of roads in the classes to be kept. In this case, some of the selected destinations are rendered unreachable by the pruning operation if road class is the sole consideration. Therefore, the edges to be pruned are road elements that both belong to a road class to be pruned and are not necessary for connecting 2 isolated networks of roads in the classes to be kept. To achieve this, we developed an algorithm that utilizes the concept of equivalence classes. The `get_preserved_equiv_classes` method and the `get_pruned_equiv_classes` methods are responsible for this. The pruning of the graph is computationally intensive, so a pruned graph is stored as a `txt` file by the `write_graph` method in the `Graph` class and the pre-pruned graph can be loaded directly by the `read_prebuilt_graph` function in the `preprocessing` module.

For a directed weighted graph $G = (V, E, w)$, we define boolean-valued functions $bidirectional\_connected : V^2 \to \{True, False\}$ and $undirectional\_connected : V^2 \to \{True, False\}$ such that

$$\begin{aligned}\forall u, v \in V, bidirctional\_connected(u, v; G) = True \Leftrightarrow &\\ &(u = v \text{ or } (\exists v_0, v_1, \ldots, v_n \in V : v_0 = u, v_n = v,\\ &\text{ and } \forall k \in \mathbb{Z}^+, k \leq n, (v_{k-1}, v_k) \in E)\\ &\text{ and } (\exists w_0, w_1, \ldots, w_m \in V : w_0 = v, w_m = u,\\ &\text{ and } \forall k \in \mathbb{Z}^+, k \leq m, (w_{k-1}, w_k) \in E))\end{aligned}$$

and

$$\begin{aligned}\forall u, v \in V, undirectional\_connected(u, v; G) = True \Leftrightarrow u = v \text{ or}&\\ &(\exists u_0, u_1, \ldots, u_\ell \in V : u_0 = u, u_\ell = v, \text{ and}\\ &\forall k \in \mathbb{Z}^+, k \leq \ell, ((u_{k-1}, u_k) \in E \text{ or } (u_k, u_{k-1}) \in E))\end{aligned}$$

It can be proven that *bidirectional_connected* and *undirectional_connected* satisfies reflexivity, symmetry, and transitivity, then *bidirectional_connected* and *undirectional_connected* are equivalence relations. Let $E' \subset E$ be the set of edges belonging to the road classes to be pruned, and let $E'' \subset E$ be the set of edges belonging to the road classes to be kept, and let $G' := (V, E')$ and $G'' := (V, E'')$. Then $\mathit{undirectional\_connected}(\cdot, \cdot; G)$ and $\mathit{bidirectional\_connected}(\cdot, \cdot; G'')$ are equivalence relations on G' and G'' respectively. Then the equivalence classes

3.3 Computation of Optimal Route

In our project, we used Dijkstra [4] to figure out the optimal route between two positions. The procedures of Dijkstra are initialization of distance of the starting position to be 0.0 and repetition of following operations: 1. Select a position that has the minimal shortest distance to the starting position and add the position to S; 2. Update/relax all the edges/roads going out from the chosen position in operation 1, where S is the set of positions that have been whose optimal route has been sought out and T is the set of positions whose optimal route has yet been sought out/visited. $\forall v \in V, v \in T$ before the algorithm starts running. Proof of correctness of Dijkstra in our project:

1. **Satisfaction of Preconditions**

   There are no negative weights in the graph based on the topic we chose for building the graph.

2. **Proof of Partial Correctness**

   $\forall v \in V$, let $D(v) =$"the actual shortest distance between the starting position s to the current position u"; let $dist(v) =$"the estimated shortest distance between the starting position s to the current position v"; and we have $dist(v) \geq D(v)$. Partial Correctness of Dijkstra stands if and only if $\forall v \in V$, if v is added to set S in operation 1, then $dist(v) = D(v)$.

```
Base Case: $S = \emptyset$ OR $S = \{s\}$, where $s$ is the starting position
    Statement is true because when $S = \emptyset$, no positions has been added; and when $S = \{s\}$, $dist(s) = D(s) = 0.0$.
Induction Step:
    Suppose the statement is false.
    Then by well ordering principle, there exists the first position $v \in V$ such that when adding to $S, dist(v) > D(v)$.
    Let $u \in V$ be the position by instantiation.
    There exists a route between $s$ and $v$ if $v$ is added to $S$ because otherwise $dist(v) = D(v) = None$.
    Assume the path from $s$ to $v$ is $s \to u \to w \to v$, where $w$ is the first position such that $w \in T$ on the route from $s$ to $v$, and $u \in S$ is the position directly ahead of $w$.
    By definition of $v$, we have $dist(u) = D(u)$ and because relaxation function will work on edge/road $(u, w)$, then we also have $dist(w) = D(w)$.
    On route $s \to u \to w \to v$, because all weights on the graph are nonnegative, therefore we have $D(w) \leq D(v)$.
    Thus $dist(w) = D(w) \leq D(v) \leq dist(v)$.
    However, because when $v \in T$ is added to $S$, $w \in T$ by construction of route.
    Then we have $dist(v) = dist(w)$.
    This means that $dist(v) = D(v) = D(w) = dist(w)$, which is a contradiction to our assumption of $v$.
Thus, $\forall v \in V$, if $v$ is added to $S$ in operation 1, then $dist(v) = D(v)$.
```

   The proof has shown that we can find the optimal route for corresponding starting position and ending position by Dijkstra.

3.4 Visualization

The visualization of the destinations for the user to select from and the planned optimal route is achieved by the python package `folium`. A `folium.Map` object represents a visualizable map, and markers can be created as `folium.Marker` objects to mark the position of the vertices on a map. A `folium.PolyLine` object can represent a path on a map. `folium.Marker` and `folium.PolyLine` objects can be added to a map using the `folium.Map.add_child` method. A `folium.Popup` object that stores some `str` messages can be added to a `folium.Marker` or `folium.PolyLine` object. The `folium.Map.save` method saves a map as an HTML file. Using the `webbrowser.open_new_tab` function, the saved HTML file can be opened by the Python interpreter, and the map showing the destination vertices or planned route will appear. When clicking on a marker or polyline in on the displayed map, the messages stored in the `folium.Popup` object associated with the marker or polyline will appear.

4 Discussion

start_id	end_id	travel_time_predicted_by_Google_maps / hrs	travel_time_predicted_by_this_project / hrs
3576325	7367553	15.6	16.04
6295901	5535850	5.45	5.81
1500202822	4547920	17.37	23.16
1500144236	6070348	6.13	6.53
4181412	6295901	7.57	8.16
7609145	4360362	1.03	0.51
5727850	1500301341	0.35	0.25
1509660794	6043947	0.7	0.39
6039549	6514158	0.22	0.05
4175230	5692111	0.22	0.13

The route planning functionality of this project is generally satisfactory. It is capable of finding a close-to-optimal path. In order to evaluate the route planning performance, we selected 10 sample origin-destination pairs and benchmarked the minimum travel time predicted by this project against the travel time predicted by Google Maps. The first 5 sample origin-destination pairs consist of far-apart pairs of points separated by a distance of hundreds or thousands of kilometers to evaluate the route planning performance for long-distance travel. The last 5 sample origin-destination pairs are separated by distances of the order 10^1 kilometers to evaluate the route planning performance for short-distance travel. For the first 5 origin-destination pairs, the travel time predicted by this project is consistently longer than that predicted by Google Maps, whereas for the last 5 origin-destination pairs, the travel time predicted by this project is consistently shorter than that predicted by Google Maps. Hence, the route planning functionality of this project has a bias correlated with the distance from the origin to the destination. The incomplete information revealed by the dataset, as well as the pruning operations of this route planning program, are likely causes contributing to this pattern of bias.

When the graph gets pruned, certain paths or shortcuts are no longer available for the route planning algorithm to choose, which would lead to sub-optimal solutions. For a pair of origin and destination separated by a long distance, the effect of pruning in the increase of predicted travel time is more manifested since the sub-optimal local solutions are likely to add up cumulatively, causing a relatively significant bias in predicting a route that uses more time than necessary. When the origin and destination are separated by a relatively small distance, the effect of missing more desirable edges is less influential. The fact that the dataset only contains road elements in the Ontario province leads to inaccurate route planning results in the cases when a more desirable route pass through regions outside Ontario. This is manifested in the 3rd origin-destination pair, in which the route from the origin to the destination planned by Google Maps passes through the territory of the United Sates.

The information provided by the dataset only reveals information about the speed limits of roads, but disregards the actual and live traffic conditions, such as any traffic jams or accidents. Google Maps, on the other hand, collects real-time traffic data and predicts the travel time based on the real-time conditions of the roads. As a result, the reduction of traffic speed caused by real-time incidents is not taken into account by this route planning program but is considered by Google Maps. This likely leads to the bias of this route-planning program in predicting a shorter travel time for short-distance travel. The short-distance travel routes are mostly positioned in urban areas, where traffic congestion is frequent, making the assumption that most vehicles travel at speeds close to the speed limit less accurate.

We also made a 2 sample t-test based on the data from Google Map and our project with the ten data above, and ultimately we have p-value = 0.845178, which means that the time consumption predicted by our project highly resembles that of Google Map. Meanwhile, a 2 sample t-interval on 0.95 confidence level interval is made, and it turns out that the mean difference on time is 0.6398 hours (predicted - Google Map), with each standard deviation 7.92098 and 6.43649 hours, while the combined margin of error is 6.80122 hours. In conclusion, our projects gives a reasonable result in predicting path and time consumption in reality.

References

- Placek, M. (2023, December 8). U.S. freight movement mode share by tonnage 2020. Statista. `https://www.statista.com/statistics/184595/us-freight-movement-mode-share-by-tonnage/`
- Carlier, M. (2023, May 4). Road traffic in the United States: Vehicle-miles. Statista. `https://www.statista.com/statistics/185537/us-vehicle-miles-on-highways-from-since-1990/`

- GfG. (2023, June 7). What is weighted graph with applications, advantages and disadvantages. GeeksforGeeks. `https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-weighted-graph/`
- Shortest Path Algorithms. `https://oiwiki.org/graph/shortest-path/`