CSC111 Project 2 Proposal: Origin to Destination Route Planning in a Road Network

Mingxiao Wei, Zheyuan Zhang

April 1, 2024

Problem Description and Research Question

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.

Road Networks as Weighted Graphs

A weighted graph is a graph in which a number, namely weight, is associated with each edge to represent a quantity such as distance or cost [3]. Formally, a weighted graph can be defined as a tuple $G = (V, E, w)$, where V is the set of all vertices, E is the set of all edges, and $w : E \to \mathbb{R}$ is a function such that $\forall e \in E, w(e)$ is the weight of the edge e. Networks of roads in the real world can be naturally abstracted as weighted graphs, in which 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. Consequently, finding the optimal origin-to-destination route in a road network amounts to minimizing the total weight of a path in a weighted graph between two given vertices.

Related Work

A classical algorithm for finding the shortest paths from a given vertex to every other vertex in a weighted graph was proposed by Dijkstra [4]. The pseudocode Algorithm 1 illustrates the general design of Dijkstra algorithm. The mapping `dist` returned by the algorithm maps each vertex in the graph to the total weight of the edges in the optimal path from the origin vertex. The mapping `prev` returned by the algorithm maps each vertex u to the previous vertex on the optimal path from the origin vertex to u. By tracing from the destination back to the origin using the mapping *prev*, one can determine the optimal route to travel from the origin to the destination. In order to efficiently determine the unvisited vertex with the minimum distance from the origin, data structures such as priority queues, heaps, and self-balancing binary search trees are often used to store the set of unvisited vertices [5]. When binary heaps or self-balancing binary search trees are used to store the set of unvisited vertices, the worst-case running time of the Dijkstra algorithm is $\Theta((|V| + |E|) \log(|V|))$ where V and E are the vertices and edges of a graph [5].

Project Goal

The goal of this project is to develop a route planning program that can determine a route in a road network from a given origin to a given destination that minimizes the expected travel time. The expected travel time of each road segment in a road network can be calculated based on the length of the road segment and historical data on average traffic speed. Dijkstra algorithm will play a central role in the route planning algorithm in this project.

Algorithm 1 Dijkstra algorithm

```
Require: $G = (V, E, w)$ is a weighted graph defined the same as previously described and $v \in V$.
  procedure DIJKSTRA($G$, $v$)
      $V \leftarrow G[0]$
      $E \leftarrow G[1]$
      $w \leftarrow G[2]$
      $dist \leftarrow \{\}$
      $prev \leftarrow \{\}$
      $dist[v] \leftarrow 0$
      $prev[v] \leftarrow$ undefined
      for all $u \in V \backslash \{v\}$ do
          $dist[u] \leftarrow \infty$
          $prev[u] \leftarrow$ undefined
      end for
      $S \leftarrow \{u : u \in V\}$                                    ▷ A set of unvisited vertices.
      while $S \neq \emptyset$ do
          $cur\_vertex \leftarrow \arg\min_{u \in S} dist[u]$
          $S \leftarrow S \backslash \{cur\_vertex\}$
          for all $neighbour \in \{u \in S : \{u, cur\_vertex \in E\}\}$ do
              if $dist[cur\_vertex] + w(\{cur\_vertex, neighbour\}) < dist[neighbour]$ then
                  $dist[neighbour] \leftarrow dist[cur\_vertex] + w(\{cur\_vertex, neighbour\})$
                  $prev[neighbour] \leftarrow cur\_vertex$
              end if
          end for
      end while
      return $dist, prev$
  end procedure
```

Computational Plan

Dataset

Ideally, a real-world dataset that reflects the information on both the lengths and historical average traffic speeds of roads will be used for the project. But such a dataset hasn't been found yet. We tentatively plan to use a csv file containing information about roads in North America obtained from the US Bureau of Transportation of Statistics as the dataset [6]. In this csv file, each road is subdivided into road segments and each road segment has a unique id. The length and speed limit of each road segment is also present. Using the API of the US Bureau of Transportation of Statistics [7], the latitudes and longitudes of the start and end points of each road segment can be obtained, which makes it possible to determine the connectivity of the roads. We tentatively use the speed limits as the estimated travel speeds of the road segments to calculate the expected travel time the road segments.

Storing Road Network

A class `WeightedGraph` will be used to represent the road network obtained from the dataset, in which each vertex represents a junction between road segments and each edge represents a road segment between 2 junctions. A class `Vertex` will be used to represent a vertex. The `Vertex` class will have an instance attribute `id` of type `int` as a unique identifier of the junction. It will also contain an instance attribute `weights` of type `dict[Vertex, float]`. For an instance `v` of `Vertex` class, for any `u in v.weights`, `u` is a neighbor of `v` and `v.weights[u]` denotes the weight of the edge {`v, u`}. The edges will be weighted by the expected time needed for a vehicle to travel from `v` to `w` via the road segment between `v` and `w` represented by the edge. The `WeightedGraph` class will have an instance attribute `vertices` of type `dict[int, Vertex]`. For an instance `g` of the `WeightedGraph` class, for every `i in g.vertices`, `i` is the unique identifier of the vertex `g.vertices[i]` (i.e. `g.vertices[i].id = i`). In this way, the vertices of a `WeightedGraph` are explicitly kept track of and it is possible to get a vertex of a `WeightedGraph` by id in $\Theta(1)$ running time, while the edges and weights in a `WeightedGraph` are implicitly kept track of as instance attributes of the `Vertex` class.

Implementation of Dijkstra Algorithm

Algorithm 2 illustrates the pseudocode of the function that will be used to compute the optimal route from the origin vertex with the `start_id` to the destination vertex with the id `end_id`. A class `MinHeap` that implements a binary min heap ADT will be used as a helper to efficiently keep track of the unvisited vertices with the smallest distances to the origin vertex. A binary min heap is a balanced binary tree in which each value is smaller than or equal to its left child and right child. The `MinHeap` class can be initialized by taking an argument vertices of type `list[Vertex]` and an argument `vertices_to_float` of type `dict[Vertex, float]` that maps each vertex in `vertices` to a `float` value. Each vertex in the list `vertices` will be stored as a value in the `MinHeap` in a way such that if `u1` and `u2` are the left and right child of the `u`, then `vertices_to_float[u1]` $\geq$ `vertices_to_float[u]` and `vertices_to_float[u2]`$\geq$ `vertices_to_float[u]`. The `MinHeap` class will have a `extract_min(self, vertices_to_float: dict[Vertex, float])` method that removes and returns the root of the `MinHeap` and restores the heap property with respect to the `vertices_to_float` dictionary. The `MinHeap` class also has a method `sift_up(self, v: Vertex, vertices_to_float: dict[Vertex, float])` that restores the heap property after `dist[v]` is lowered.

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/`
- Dijkstra, E.W. A note on two problems in connexion with graphs. Numer. Math. 1, 269–271 (1959). `https://doi.org/10.1007/BF01386390`
- Wikimedia Foundation. (2024, February 23). Dijkstra's algorithm. Wikipedia. `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`
- North American roads. Geospatial at the Bureau of Transportation Statistics. (n.d.). `https://geodata.bts.gov/datasets/usdot::north-american-roads/about`
- North American roads. Geospatial at the Bureau of Transportation Statistics. (n.d.-a). `https://geodata.bts.gov/datasets/usdot::north-american-roads/api`

Algorithm 2 The route planning algorithm for this project

```
Require: $self$ is an instance of WeightedGraph class and $start\_id, end\_id \in self.vertices$
Ensure:
  If $self.vertices[start\_id]$ is connected to $self.vertices[end\_id]$, then
  self.find_optimal_path(start_id, end_id) = (weight, path)
  where $weight$ is the total weight of the edges in the optimal path and
  $path$ is a list of the unique identifiers of the vertices in the path in the order from start_id to end_id.
  Else, None is returned.
  procedure FIND_OPTIMAL_PATH($self$, $start\_id$, $end\_id$)
    if start_id == end_id then
      return 0, [start_id]
    else
      v1 ← self.vertices[start_id]
      v2 ← self.vertices[end_id]
      dist ← {}
      prev ← {}
      dist[v1] ← 0
      dist[v1] ← None
      for all i in self.vertices do
        v ← self.vertices[i]
        dist[v] ← $\infty$
        prev[v] ← None
      end for
      hp ← MinHeap(self.vertices.values(), dist)
      while not hp.is_empty() do
        cur_vertex ← hp.extract_min(dist)
        for all neighbour in cur_vertex.weights do
          alt_dist ← dist[cur_vertex] + cur_vertex.weights[neighbour]
          if alt_dist < dist[neighbour] then
            dist[neighbour] ← alt_dist
            prev[neighbour] ← cur_vertex
            hp.sift_up(cur_vertex, dist)
          end if
        end for
      end while
      if dist[v2] == $\infty$ then
        return None
      else
        path ← [v2.id]
        u ← v2
        while u != v1 do
          u ← prev[u]
          path.append(u.id)
        end while
        path.reverse()
        return dist[v2], path
      end if
    end if
  end procedure
```