DSA Final Report

Sam Daitzman and Dhara Patel
05 May, 2020

Abstract

We create and analyze a local multimodal transit routing algorithm tailored to students at Olin College, and optimized to provide an optimal pathway between the most commonly traveled locations using standard modes of transit. Our algorithm is modular and can support unidirectional or bidirectional transit on modes such as Lyft, Uber, walking, the MBTA Red Line and Green Line, the Commuter Rail, and bicycles. We implemented a custom map format that concisely stores a set of routing nodes that could be generated programmatically or manually; for our application, we generated a map of common locations and transit modes that students at Olin College often prefer.

Background and Motivation

At Olin College, students regularly take brief day trips with groups of friends into Boston. While we're a short drive from the city, most students don't have a car on campus, so the question of how to travel can be complicated depending on the desired destination(s).

Existing Scholarly Work

A variety of scholars have previously considered the multimodal transit routing problem, so we considered some of their approaches.

- The *metaheuristic* approach considered by Fan, Lang, and Christine L. Mumford [1] uses a modified version of Dijkstra's algorithm, which we were ultimately inspired by. By adding some modifications to Dijkstra's algorithm, the authors tailored it to the specific constraints of transit routing.
- Another approach implemented by Dibbelt, Julian, et al. [3] explicitly accounts for transit timetables using CSA (Connection Scan Algorithm), which inspired our datastructure representation. While we haven't implemented CSA/timetable scanning, we've set up the datastructures needed to add support for this and fall back on our current approach.

Existing Commercial Solutions

Some existing solutions attempt to address the multimodal transit routing problem, with varied success. We catalogued existing solutions to decide on a useful set of features.

- Google Maps provides routing on a variety of methods, but struggles with accurately predicting timing for transfers and is missing many forms of transit. Additionally, it doesn't account for any kind of user preference or constraint.

  The patented Google Maps transit routing implementation [2] uses a graph representation constructed from a collection of locations and transit modes to convert the transit routing problem into a solvable graph theory problem.

- CityMapper offers local routing within the metropolitan area, but is closely focused on travel within dense city environments and struggles to find creative routes into or out of the city. It also fails to allow for control over constraints in routing, beyond arrival/departure time and whether to include dock-less transit options.

Goals

- Provide optimal or near-optimal routing within a limited set of locations
- Adapt easily to custom maps
- Support common modes of transit as a proof of concept
- User-controllable constraints

Final Algorithm

Approaches Considered

We designed several algorithms and compared them to find the best fit. We compared tailored algorithms based on Dijkstra's Algorithm, the A* algorithm, and the Bellman-Ford algorithm. The A* algorithm is commonly used in transit routing, and offers some useful affordances like not cutting off pathways we may later want in future iterations of the same algorithm. We decided to start by implementing Dijkstra's Algorithm.

Selected Implementation

Given a graph of location nodes, the algorithm finds the shortest path from the starting location node to the desired destination node by making locally optimal decisions. The `visited` list is used to track whether or not a given node has been visited. The `queue` list initially contains all nodes and is used to tell the algorithm when to stop. The `dist` dictionary contains the distance between each node and the starting destination node. While `queue` is not empty, the algorithm will go to node `u`, the node with the smallest value in `dist`, and update all of `u`'s neighbors in `dist`. For every adjacent vertex `v`, if the sum of `dist[u]` and the weight of edge `u`-`v`, is less than `dist[v]`, then update the `dist[v]` to equal the sum of `dist[u]` and the weight of edge `u`-`v`. Once `queue` is empty we are left with a dictionary that contains accurate distances between each node and the starting destination node.

The algorithm accounts for the user by assigning the weight of every edge according to preset preferences. It takes in a list of weights that determine the importance of time, number of transportation modes, cost, and calories. The weighted sum is then set as the weight of every edge.

Implementation

We successfully implemented a map format that stores the data our algorithm needs to function, and a routing algorithm that can choose routes intelligently based on the most efficient timing. Our datastructure is tailored to the multimodal transit routing problem, both in its static filestore and in-memory graph form.

Filestore

Our filestore datastructure uses serialized JSON to represent many modes of transit concisely in a way that can be accessed and generated programmatically. As a proof of concept, we currently generate all longitude/latitude pairs using a trivial Mathematica script—it's very simple to work with this format in any language that support writing to JSON.

```
{
    "places": [
        {
            "name": "olin",
            "coords": [-71.2644, 42.2916],
            "transit": [
                ["tofrom", "babson", "walk"],
                ["tofrom", "eliot", "walk", "lyft"],
                ["tofrom", "reservoir", "lyft"],
                ["to", "babson", "bike"],
                ["to", "eliot", "bike"]
            ]
        }
    ]
}
```

Graph

We use the popular NetworkX graph library implementation, and we selected the `MultiDiGraph` graph class to track our nodes and edges. This allows us to store all common types of transit edges, but it's useful to store additional metadata along the edges and at the nodes, so we bind pointers to additional `LocationNode` class instances to each graph node, and pointers to additional `TransitEdge` class instances to each graph edge. This allows us to embed all of the important data. Below are code excerpts of the important information about these datastructures.

`LocationNode`

We designed our `LocationNode` implementation to hold additional metadata in a standard format at each location on the map. Currently, we use it to store the latitude and longitude of locations. In our NetworkX implementation, the `name` property is redundant since each node's key in the `MultiDiGraph` is already the name of the location. To integrate this implementation with a user-facing interface, we'd want to set the human-readable name of each location.

```
class LocationNode:
    '''
    LocationNode class
    Represents a node along the map, a location with some TransitEdges in/out
    self.name: [string] Name of this node -- optional, unneeded in NetworkX
    self.coords: [enum: (lon, lat)] Location
    '''
    def __init__(self, name=None, coords=None):
        self.name = name     # name of this node
        self.coords = coords # (lon, lat)
```

`TransitEdge`

We designed our `TransitEdge` implementation to support tracking metadata so a future version of this algorithm could provide even more intelligent decision-making between many different potential paths. In our current implementation, we only use simple timing, but we have support for advanced timing types to support light rail scheduling and variable travel time.

```
class TransitEdge:
    '''
    TransitEdge class
    Stores data about the time a transit mode takes

    self.start: [LocationNode] TransitNode at beginning of pathway
    self.to: [LocationNode] TransitNode at end of pathway
    self.cost: [number] Cost in USD to follow this path
    self.type: [string] Type of transit, e.g. redline, greenline, car, walk,
        bike, bus, commuterrail, lyft
    self.calories: [number] Number of calories burned in traversing this mode
    self.timing: [Timing || number] Time to traverse this edge. If a number is
        passed, it will be converted to a constant time traversal Timing object
        with a traversal time in minutes.
    '''
    def __init__(self, start=None, to=None, cost=0, type=None, calories=0, timing=None):
        self.start = start
        self.to = to
        self.cost = cost
        self.type = type
        self.calories = calories
        self.timing = timing
```

Results

Implementation

This project serves as a proof-of-concept that shows computational and algorithmic feasibility of implementing a tailored transit solution specifically for students at Olin College, and making that transit solution easily modified to support any transit system or local environment. Our approach allows for simple implementation of user-input data routing, and is extensible to enable future improvements.

Optimality

The multimodal transit path optimization problem is considered NP-hard [1], so we did not attempt to implement a complete solution on real-world data. By constraining our data to a tailored sample, we were able to implement an algorithm that should always find some solution where one is possible, and will always find at least one route that is close to time-optimal.

Runtime

$$\text{runtime} = O(V^2)$$

Because the algorithm considers a total of *V* nodes, performing one *O(1)* operation at each node, and repeats that pattern at most *V* times, its total runtime is *O(V^2)*. Of course, this disregards our data preprocessing, which is considerably more time-intensive. Since our parsing and graph creation must only happen once, when the graph is created and stored in memory from the on-disk serialized datastore representation, we consider that *O(1)* and disregard it in this consideration of routing runtime.

Next Steps

Our next steps begin with implementing a user interface that allows someone to input their own data and test the algorithm. This would allow us to test the algorithm in practice and confirm that its results are reasonable.

Another important consideration is user-friendliness of routing. We'd like to consider implementing custom heuristics to account for individual users' routing preferences. For example, some people might avoid a particular mode of transit at some times because they find it too busy—ideally, we'd like our transit routing to learn from those patterns and behave intelligently.

Video

Our video can be found at https://www.youtube.com/watch?v=0J1g8YXu_CE.

Works Cited

1. Fan, Lang, and Christine L. Mumford. "A metaheuristic approach to the urban transit routing problem." *Journal of Heuristics* 16.3 (2010): 353-372.
2. Bast, Hannah, et al. "Transit routing system for public transportation trip planning." U.S. Patent No. 8,417,409. 9 Apr. 2013.
3. Dibbelt, Julian, et al. "Intriguingly simple and fast transit routing." *International Symposium on Experimental Algorithms*. Springer, Berlin, Heidelberg, 2013.