TrekStar Route Planner

General-purpose framework for automatic recreational route planning*

Qingzhuo Aw Young
California Institute of Technology
qingzhuo@caltech.edu

Mary Giambrone
California Institute of Technology
mgiambro@caltech.edu

Timur Kuzhagaliyev
California Institute of Technology
timbokz@caltech.edu

Rachael Morton
California Institute of Technology
rmorton@caltech.edu

Matthew Wu
California Institute of Technology
mewu@caltech.edu

ABSTRACT

Intelligent automatic route planning is a challenging task when the set of customizable constraints extends beyond the standard shortest-distance or least-time-taken requirements. This paper describes our efforts to develop a system for automatic route planning with non-trivial constraints. We cover both frontend and backend architecture of the system and the algorithmic and data ingestion solutions. Our final product is an online route planner with a modular backend. In developing the app, we encountered many challenges in UI, algorithms, and data collection. At the time of writing, our website is available at: https://trekstar.science.

KEYWORDS

Route planning, orienteering, networks

1 INTRODUCTION

Activities such as cycling and running are becoming increasingly popular, with millions of people joining the trend every year [11]. There exist numerous websites and mobile applications used for route planning, ranging from general-purpose services like Google Maps[1] to more athletics-oriented services like Strava [3].

Traditional automatic route planning services often attempt to provide the most efficient route according to some standard metric, e.g. distance covered or time taken [4]. Although some of these services take into account complex factors like road congestion or type of transportation preferred by the user [10], the parameters that users can adjust are often severely limited. When it comes to recreational activities such as cycling, running or simply walking, users often sacrifice the efficiency in favour of routes containing more scenery, more landmarks, tourist attractions, or otherwise more "interesting" routes with respect to some non-trivial metric.

In most general-purpose route planning services, users are only able to specify their starting point, their destination and potentially several extra stops along the way. In more complex and specialized solutions, such as Strava's Route Builder, one can specify a few additional parameters such as preferred elevation. While all of these route planning tools fulfill their purpose, they universally fail to capture the complexity of user preferences in their algorithms. To remedy this flaw, some services rely on route recommendations - these are routes manually constructed, labeled and tested by existing users [2]. Needless to say this approach only works when there are enough active users to make a meaningful contribution to the recommendation pool.

This fundamental limitation of existing route planning services is easy to explain - gathering enough meaningful data and producing an algorithm capable of making sense of this data is generally seen as a challenging, time-consuming task. As a result, more traditional approaches to route planning are used, and users often take whatever route they've been suggested and deviate from it wherever necessary to satisfy their personal requirements.

We believe that this situation can be improved dramatically by leveraging publicly available data sets. By extracting relevant information from road networks, satellite images, geographical data, popular route heatmaps published by Strava, and even Yelp reviews, we can generate a global network that will capture a vast number of possible parameters. This network, along with the parameters encoded in it, will then be used to generate highly-customizable routes based on preferences and constraints specified by the user. It is also possible to learn user preferences and build a collaborative model on top of this data, focusing on social translucence [7].

The structure of the paper is as follows. Section 2 contains the problem statement and our vision of implementation. Section 3 describes the data sources we used, the data ingestion process and it's implementation. Section 4 describes the frontend and backend architecture of the final system. Section 5 talks about how we approached the algorithmic problem behind route planning. Finally, sections 6 and 7 present the end product and our conclusion.

2 PROBLEM STATEMENT

The main goal for our project was to develop a proof-of-concept implementation of a general-purpose framework for automatic route planning. We tried to emphasize flexibility and extensibility, making sure that our solution is not tightly coupled to a particular dataset or a particular path-finding algorithm.

2.1 Overview

There are two tightly-linked yet distinct challenges addressed by our project. The first involves combining data from multiple sources to produce a global network that will capture useful information from said sources in its entirety (or at least a reasonable approximation of it). The second challenge is to find an efficient approach to exploiting this resultant network to generate routes based on user-defined constraints.

*Produced for Caltech's CS 145 Projects in Networking class. Project source code at https://github.com/ariadnes-thread/

This global network links the two challenges together. Clearly, the algorithm we will develop for route generation would have to rely on the network structure. In turn, we might discover that our algorithm might perform better if we tweak the way network is generated in the first place, and vice versa. The next couple of sections will discuss how we planned to address each challenge separately, and then combined our findings to improve interactions between them.

2.2 Product Differentiation

All of the popular route planners today are tailored for a specific purpose like shortest-route planning. Due to this, they only consider a few requirements that are immediately relevant to the metric these route planners are trying to maximize. Although this makes them very efficient at fulfilling their direct purpose, it often makes them unsuitable for generating recreational routes subject to non-trivial constraints. As a result, users tend to stay away from automatic route planners when it comes to recreational route planning, and use recommendations from other users (e.g. manually designed routes from tour guides).

The main difference between our route planner and existing ones is that we're trying to capture as many user preferences and route constraints as possible, generating a route that maximizes user satisfaction. Making this possible requires a very flexible system that can easily ingest new datasets and adjust algorithms to support these data sets. Route planners that are designed for one specific purpose are often optimized to perform that particular efficiently, what often makes the resultant solution very hard to reuse for other purposes. In contrast, we designed every single component of our system (frontend, backend, database, route planner service) with generic interfaces in mind and the ability to swap algorithms and components in and out on demand.

3 DATA INGESTION AND PROCESSING

Our project drew data from several sources. We called into the Google Places API and the Yelp Fusion API to get information about Points of Interest (POIs). We obtained data about road popularity from Strava. The National Agriculture Imagery Program data was used to set the greenery ratings for the edges in our graph. We used the United States Geological Survey data for information about elevation. We used Open Street Maps data for our underlying graph of the world.

3.1 Identifying relevant data sets and extracting information

As is often the case with big data, arbitrary data sets can provide valuable insight into a seemingly unrelated subject. Due to this, our long term vision is to continuously look for new potential sources of data and also keep on eye on relevant research to see if someone has identified a data set that might be relevant to our project.

That said, the first iteration of our project was limited to a single term. To account for this time limitation, we began building our proof-of-concept prototype with a only few data sets. Some data sets presented quantitative information, like the Open Street Maps representation of possible routes, while other data sets provided qualitative information, such as the elevation, greenery, and popularity data sets. The significance of the qualitative vs. quantitative separation is discussed in section 3.2.

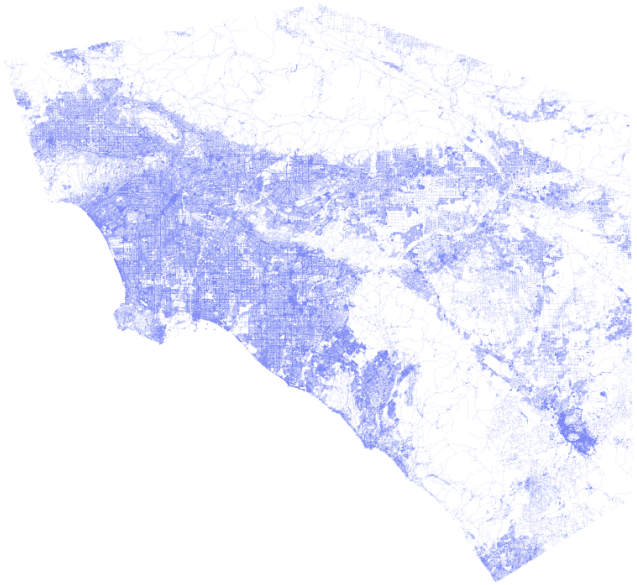

Figure 1: Road network that is supported by the latest version of the routing service.

For the initial build of our application, we chose the area around Pasadena, California. Then, we expanded to cover a bigger part of the Los Angeles County area (Figure 1). Currently, there are over 960,000 road segments and 750,000 junctions in our database.

We used the route data we obtained from Open Street Maps, and then refined this graph by extracting information from the "qualitative" data sets and augmenting relevant nodes and edges[6].

3.2 Producing a global network

Our general approach to producing the final network is outlined above, but there are many challenges we had to tackle along the way. One of the most important issues was keeping the network up-to-date: we had to find a way to update the network, including adding or removing nodes, without corrupting any existing data accumulated by our users. This can include saved routes, routes created manually, or other cached routes that relied on the data we might have deleted. We anticipated that such updates will occur infrequently and on small scale, so performing relative large data manipulations wouldn't cause any major decrease in performance. Of course, this is not the case with real-time data - anything real-time, such as information about road congestion, will be delegated to the route planning algorithm and is discussed in detail in section 5.

In our first attempt to capture all of this data in a single network, we planned to store multiple weights for every edge (extracted from quantitative parameters) and multiple normalized values for every node (extracted from qualitative parameters). This way, during the route planning phase, we were able to adjust the weights of edges

and the values of nodes to better represent the preferences of a particular user. The main idea here is to use different representations for "objective" and "subjective" values. This way we can prioritize yielding an accurate result based on distance, elevation and time constraints, while yielding a somewhat less accurate result for users' preference on amount of greenery and tourist landmarks.

3.3 Network metadata

There are both structured and unstructured datasources from which we can extract network metadata. One source of structured data is OpenStreetMap [9], where the metadata are represented as tags on objects (lines and polygons). Such structured datasources have well defined schemas, allowing us to easily extract relevant information, (e.g. road types, speed limit, junctions, etc).

Most datasources, however, are unstructured. Examples include satellite imagery and heatmaps. For such datasources, we require special tooling for data extraction. Modern web based maps are served using tiled web maps, also known as slippy tiles, which are square raster tiles indexed by 3 coordinates, X, Y, Z, which represent the longitude/azimuth, latitude/inclination, and zoom level respectively.

We have developed a tool intended for metadata extraction from any tiled web map source, with support for authentication. A data source URL template `https://example.map/Z/X/Y.png` needs to be provided, along with a transform function that maps a raster section to a real valued number. The data ingestion process works in a mapreduce manner, the list of sample points are mapped into rasters (fetched from the web), which are then masked and reduced to a real valued number for each sample point.

Due to efficient caching, bandwidth is the limiting factor, and runtime scales linearly with the area processed (i.e. number of tiles that are fetched). Increasing the density of sample points within an area has a negligible impact on runtime, which means that we can afford to have many sample points with small intervals for high resolution data capture.

An example of extracting greenery data is shown in Figure 2. A pooling filter of radius $\sim 10m$ is applied at points sampled at equal intervals from the road network. The pooling function compares the intensity of the green channel against the blue and red channels of pixels in the radius of the filter, and outputs a value in $[0, 1]$ that captures the amount of greenery surrounding the sample point.

4 ARCHITECTURE

This section describes the setup we used to get the system running (with the exception of algorithmic solutions, which are described in section 5). We start by providing links to the source, then describing user-facing components, and slowly move towards backend and database descriptions.

All of the source code for the systems is publicly available under Ariadne's Thread organization on GitHub, which can be found by following this link: https://github.com/ariadnes-thread. The name of the repositories roughly correspond to the components described below, and each individual repository contains a README file which can be used for further guidance.

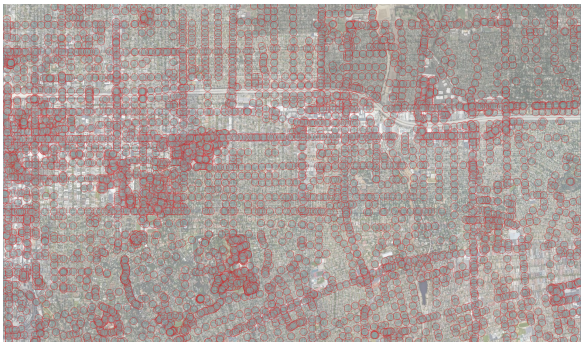

Figure 2: Example of filter used to extract greenery metadata from satellite imagery. Red circles represent the radii of points sampled along roads. Background consists of stitched satellite imagery, desaturated to emphasize the foreground.

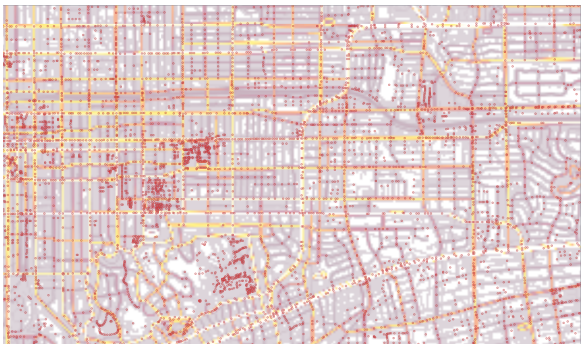

Figure 3: Example of filter used to extract popularity metadata from heatmap imagery. Background consists of stitched heatmap rasters. Same geographic area as in Figure 2, note the difference in filter size (red circles are smaller).

The system for our route planning service consisted of four major components (*layers*) which are listed below. Figure 4 gives a high level overview of how these components interact with each other.

(1) User-facing web application (*frontend*),
(2) User-facing Node.js HTTP API (*backend*),
(3) Internal Python API (*route planner*), and
(4) Instance of PostgreSQL with two databases (*database*).

Each layer had a clearly established interface, making it possible to apply major changes to one layer while only requiring minor adjustments (if any) to other layers. Due to the experimental nature of our project, this level of separation of concerns proved to be very convenient - for example, we could swap completely different implementations of the Python route planner in and out without compromising the functionality of the whole system.

Communication between different layers was handled by appropriate protocols and software: frontend to backend communication was done via HTTP (`GET`, `PUT`, `POST`, `DELETE`) requests; backend to route planner communication was done using gRPC protocols; backend/route planner to database communication was handled by

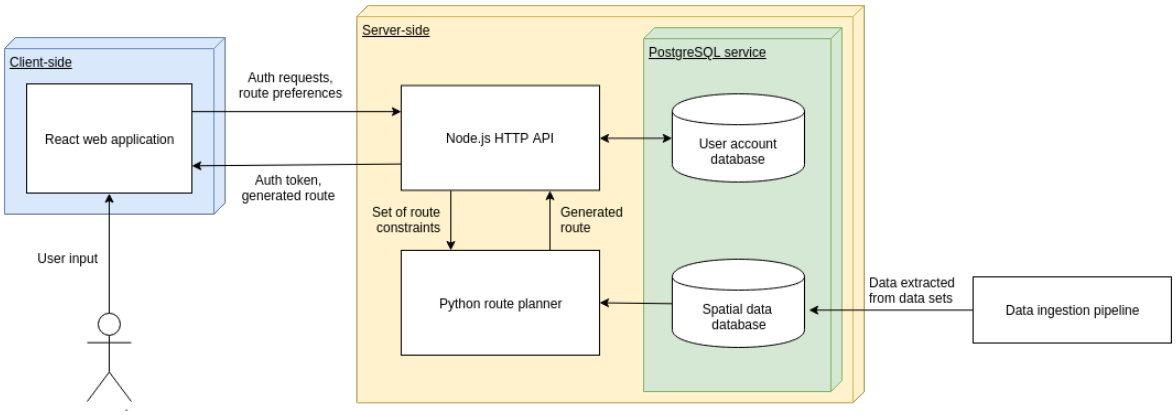

Figure 4: High-level overview of the architecture of our system.

appropriate PostgreSQL drivers. Refer to sections below for detailed descriptions.

4.1 Frontend

The frontend for our system is a standalone web application (web app) written in JavaScript. The only point of contact between the frontend and other parts of the system was the user-facing HTTP API, so consistent interface in said API was the only requirement for the frontend to function correctly.

4.1.1 User input. Our backend was designed to be frontend-agnostic - that is, it only takes an array of constraints as the input. As a result, web app developers had complete freedom in how they gather user input, as long as they produce an array of constraints as the output. This allowed us to experiment with different user interface (UI) and user experience (UX) solutions to see which ones work best.

Most existing route planning tools, such as Google Maps, aim for instant route generation with minimal user configuration. In the beginning, we aimed for a different philosophy - our first UI/UX prototype let user define a large number of parameters (e.g. start and destination points, types of points of interests, loop vs. point-to-point route generation). The problem with this approach was that new users were overwhelmed with the sheer number of parameters they had to specify, which made our route planner look very intimidating. In further iterations of UI design, we tried to define user values in as many places as possible, so that a user could generate a route after having specified only one or two parameters. Additionally, we try to hide some non-essential parameters by default, letting more advanced users discover them as they get more familiar with the web app.

4.1.2 Implementation notes. We used Create React App (CRA) tool from Facebook to aid frontend development. It supports a local web server which lets developers preview the changes they make in real time and considerably speeds up the development process. To serve the web app to our end users, we compiled the source code into a static HTML+CSS+JS website with no external dependencies and exposed it to the world as a static Apache 2 site.

We used the Single Page Application (SPA) approach when developing our web app. In SPAs, the browser only ever loads a single HTML page, and new content is presented to the user by dynamically updating that page using JavaScript (JS) and Asynchronous JavaScript And XML (AJAX). AJAX requests are essentially dynamic HTTP requests, and in our implementation all of the communication with the API was handled using AJAX requests. To make rapid prototyping possible while keeping a professional overall look, we used the Bulma CSS framework.

We used Facebook's React library as our primary SPA framework due to its maturity, large existing knowledge base and previous experience our team had with it. React encourages breaking the UI down into small, reusable components. This philosophy was useful for the preference editor in our web app. We designed a UI framework on top of Bulma and React that allowed us to easily add new preference fields, while only requiring few changes to be made.

The prototype we developed targets desktop users and requires reasonable screen size to get the best experience with the web app. That said, the UI is still responsive and should be usable on mobile devices too, but the UX hasn't been specifically optimized for mobile usage. Additionally, we tried to add visual guides and tooltips where possible, explaining what certain preferences are for, showing the active state of map point selection and visualizing the user's progress along their route when they hover the elevation profile plot.

4.2 Backend

Our user-facing HTTP API is a server-side application written in JavaScript and powered by Node.js. This backend setup is responsible for receiving requests from the frontend, either handling them locally or routing them to appropriate handler. The API is intentionally exposed to the world through an Apache 2 reverse-proxy to allow different clients (not only our frontend) to use our route planning service. The primary functions of the backend are to handle authorization, provide user profile data to the frontend and generate routes.

Route generation does not happen locally. When a new route generation request arrives from the frontend (or otherwise), the backend parses the HTTP message, extracts constraints for the route, converts everything into JSON and only then passes the data to our Python route planner service. The communication between different processes (Node.js and Python) is handled by an open source Remote Procedure Call (RPC) framework called gRPC. In combination with JSON, which is a widely supported data representation format, relying on gRPC for inter-process communication meant we had complete freedom in the tools and programming languages that we use, as gRPC has drivers for almost all mainstream languages.

4.2.1 Implementation notes. We followed the fairly standard Node.js API development practice and used Express.js to power our API. To handle authorization, we used JSON Web Tokens (JWTs) and the standard Bearer authentication scheme. In this scheme, the client (e.g. frontend) submits some identification information (email and password in our case) to receive a secure token, which must be included in the `Authorization` header of all further requests. Since we never got to implementing advanced user profile functionality in our system, our frontend automatically authorizes all users under the demo account. The Node.js API directly accesses one of the two database in our system. The database it uses holds all of the user profiles, account information and other meta data that is not directly relevant to the route generation.

For developer convenience and to make the system easier to debug, we documented all of the endpoints and made the documentation available online under the domain https://api.trekstar.science/. We also tried to validate and sanitize all requests from the frontend using a Node.js package called Joi. Initially, we would enforce strict validation on the constraints that frontend would provide for the route planner service. However, halfway through the development process, we realized that every change to the way constraints are defined would require changes to be made in both the frontend and backend, so we removed strict validation. Now, the backend doesn't do any extra validation on input constraints, and relies on the Python backend service to handle its own validation.

4.3 Route planner

The route planner service for our system is written in Python 3.6. It accepts route generation requests over gRPC, parses the constraint JSON data and returns a route that best matches the specified constraint. The route planner is not meant to be accessed directly by clients but rather through the Node.js HTTP API.

To ensure that we generate routes that always satisfy the user-specified constraints, we decided to implement and use several different route planning algorithms. When a new route generation request arrives at our route planner, it checks if the set of constraints fits certain requirements and picks the most suitable algorithm based on that. To make this functionality possible, we defined a generic router interface which made our route planner independent of the algorithm used to generate the route. See section 5 for more information about all of the route generation algorithms we implemented.

4.4 Database

We used PostgreSQL as our primary database software. We defined two distinct databases - one was meant to permanently hold records relevant to standard non-route-planning functionality such as user profile and account information; the second database holds the data described in section 3, including actual spatial data about the road network and data from different data sets such as greenery or elevation.

On its own, PostgreSQL is a very powerful and a well-established database system. It contains all standard database features out of the box, except for the spatial data representation. To get the relevant functionality working we used a PostgreSQL extension called PostGIS, which exposes all of the data types necessary to represent geographic data. Additionally, we made extensive use of the pgRouting library, which extends PostgreSQL by adding implementation of a large number of search, pathfinding and graph traversal algorithms.

5 ALGORITHMS

Creating attractive routes from data requires effective algorithms. Our proof-of-concept routing service contains 4 algorithms and selects one to use based on provided preferences. Since we are doing recreational routing (instead of just shortest-path from point A to point B), we tried to employ more descriptive route preferences than just a "start" and an "end," such as a desired length, preferred points of interest, and preferred road types. Thus, our algorithms attempt to simultaneously satisfy multiple constraints and objectives. Coming up with such algorithms was one of the challenges of this project; however, we were able to devise algorithms for all combinations of preferences.

In this section, we first discuss our theoretical model of the datasets and user input. Then, we describe how the route planner selects between the 4 algorithms. Finally, we give details about the algorithms themselves.

5.1 Data model

The world is a directed graph $G(V, E)$.

While many vertices are just junctures between roads, some vertices are worth visiting on their own right and are called points of interest (POIs). Each POI v has a type, denoted by $T(v)$, and a value, denoted by $V(v)$. The possible types are $1 \dots t$. We assume that each POI has only one type. The value often represents the POI's rating.

Currently, POI information is not actually stored in the database, but is fetched on the fly when making each route. External POI services usually provide the locations of POIs as latitudes and longitudes. To convert these to vertices on our graph, we snap them to the nearest vertex. This works extremely well in practice, resolving external POIs to a vertex less than 100 feet away. This allows multiple external POIs (imagine shops in a mall) to share a vertex, and shifts the burden of maintaining 'fresh' POIs to third parties. We are thus also able to integrate our routing service with any POI provider.

Each edge e has a base cost $C(e)$, which represents its length in the physical world. For advanced routing algorithms, we introduce a dynamic cost function $C'(e)$, which is a function of its length

	No POI preference	POI preference
No desired distance	Shortest path	"POIs-on-the-way"
Desired distance	"Cost-map"	Orienteering

Table 1: Algorithm selection based on provided preferences.

and metadata values. Metadata values of edges are represented by u dimensional vector $1 \dots u$, each representing an aspect of edge "characteristics." $W_j(e)$ is the value of edge e along dimension j. Importantly, we have $0 \le W_j(e) \le 1$ for all j and e; in other words, all characteristics are normalized.

Although our route planner is connected to the specific datasets mentioned before, our data model is general, allowing extension to more datasets on points of interest or edges.

5.2 User input

The user inputs an origin and destination, a desired distance, POI preferences, and edge preferences. Our user input is much more descriptive than the simple "start" and "end" offered by many other route planners.

The origin, denoted by O, and destination, denoted by D, are vertices of the graph and are required.

The desired distance, denoted by Δ, is optional.

POI preferences are a vector of nonnegative numbers $p_1 \dots p_t$. p_i is the user's relative preference for points of interest of type i. POI preferences are optional; if they are not provided, then $p_1 = \dots = p_t = 0$.

Edge preferences are a vector of nonnegative numbers $q_1 \dots q_u$. q_j is the user's relative preference for characteristic j. Edge preferences are also optional; if they are not provided, then $q_1 = \dots = q_u = 0$.

5.3 Algorithm selection

The route planner contains four algorithms:

- Shortest path
- "Cost-map" algorithm
- "POIs-on-the-way" algorithm
- Orienteering algorithm

Selection is based on whether a desired distance and POI preferences are provided, as shown in Table 1. Note that each algorithm handles edge preferences on its own, so edge preferences do not affect algorithm selection.

5.4 Shortest path

If neither a desired distance nor POI preferences is provided, we simply use Dijkstra's algorithm to compute the shortest path from O to D.

Edge discounting. If edge preferences are provided, we make edges that are similar to the user's preferences cheaper, so they are more likely to be chosen by the shortest-path algorithm. We call this process *edge discounting*.

Specifically, we define the new cost function:

$$C'(e) = C(e)\left[1 - 0.7 \cdot \frac{\sum_{j=1}^{u} q_j W_j(e)}{\sum_{j=1}^{u} q_j}\right] \tag{1}$$

Then, we perform Dijkstra's algorithm with C'. Note that $0 \le \left(\sum_{j=1}^{u} q_j W_j(e)\right) / \left(\sum_{j=1}^{u} q_j\right) \le 1$. Thus, an edge can be discounted to a minimum of 0.3x its original cost.

5.5 "Cost-map" algorithm

If a desired distance is provided but POI preferences are not provided, we use Dijkstra's algorithm with heavily modified edge costs. Set the cost of edges to be

$$\text{actual cost} \times (\%\text{ desired dist consumed} - \%\text{ way to the destination})^{12}$$

Then, the shortest path seems to be about the right distance. If edge preferences are provided, we adjust the costs using edge discounting again. Before running Dijkstra's algorithm, we first adjust the weight of the edges to be:

- l_p = distance(source, start)
- s_p = distance(target, source)
- r = distance(target, end)
- r_r = distance(start, source)
- l = edge_length
- multiplier

$$M(e) = 1 - 0.7 \cdot \frac{\sum_{j=1}^{u} q_j W_j(e)}{\sum_{j=1}^{u} q_j}$$

$$C'(e) = C(e)\left(\frac{l_p + l}{\Delta} - \frac{s_p}{s_p + r}\right)^{12} \cdot l \cdot M(e) \tag{2}$$

Then, we use Dijkstra's algorithm on these adjusted edge costs.

5.6 "POIs on the way" algorithm

If a desired distance is not provided but POI preferences are provided, we use an algorithm that visits points of interest "on the way" from the origin to the destination.

We suppose the user wants to visit a small number, say $N = 3$, POIs. The ideal solution would be to get all relevant POIs and find the shortest path from the origin to the destination that visits N of them, which is a variant of the traveling salesman problem.

We do something a bit simpler:

(1) Get all relevant POIs. We get all POIs within a large radius of the origin and destination where $p_{T(v)} \neq 0$; in other words, these POIs have positive value to the user.
(2) Find the N POIs P that minimize $d(O,P)^2 + d(D,P)^2$. d is not the shortest-path distance in the road network, but the straight-line distance in the lat/lon Cartesian plane. This selects nearby POIs and thus approximates the N best POIs to visit.
(3) Sort the POIs in ascending order of scalar projection onto $\vec{OD}$:
$$\frac{\vec{OP} \cdot \vec{OD}}{\|\vec{OD}\|}$$
Again, the vectors are in the lat/lon plane. This approximates the best order to visit the POIs.
(4) Make the route by using Dijkstra's algorithm to visit the POIs in order.

Edge preferences. If edge preferences are provided, we use the same modified cost function as (1), and perform Dijkstra's with C'.

5.7 Orienteering algorithm

If both a desired distance and POI preferences are provided, we use an orienteering algorithm. The main steps of this router are:

(1) Get nearby POIs.
(2) Compute pairwise distances between POIs.
(3) Use an orienteering algorithm to find a good path.

The first step is to find all relevant POIs. Like in the POIs-on-the-way router, we get all POIs within a large radius of the origin and destination where $p_{T(v)} > 0$. We can cull the set of POIs to only those within the ellipse with foci O and D and major axis Δ, since those are the only POIs that can be reached in time [5]. We assume the user wants to visit as many of these POIs as possible.

The remaining problem is to find the highest-scoring route through the POIs that stays under the desired distance. This is known as the *orienteering problem*. We use an algorithm by Tsiligirides (1984) [12].

The algorithm by Tsiligirides assumes that the pairwise distances between all POIs are known. To compute this, we run Dijkstra's algorithm from the origin, destination, and each POI. Let the cost function C be extended, so that $C(u, v)$ is the shortest-path distance between vertices u and v if both are the origin, destination, or any POI.

Before we present the algorithm, it is helpful to define some terms. When we are at node u and have already traveled a distance δ, it is still possible to visit the node v if $\delta + C(u, v) + C(v, D) \leq \Delta$. We call such nodes *feasible*.

Also, when we are at node u, we define the *desirability* of a node v as $A(u, v) = [p_{T(v)}V(v)/C(u, v)]^4$. This balances the distance to v with the user's value of visiting it.

Then, the following algorithm computes a random path that stays under the desired distance:

(1) Let $u = O$, $\delta = 0$, and let the current path contain only O.
(2) Repeat:
    (a) For each feasible node not yet on the path v, compute its desirability. Of the 4 nodes with the greatest desirability, choose one with probability proportional to its desirability. Let the chosen node be v. Add v to the current path, and update $u = v$ and δ += $C(u, v)$.
        (i) If there are no feasible nodes, add D to the current path and return the path.

For each path, we also compute its score, which is the sum of $p_{T(v)}V(v)$ for all POIs on the path. We generate 1000 random paths and pick the one with the highest score.

With edge preferences. To incorporate edge preferences, we use edge discounting. We use the same edge discounting equation (1) as the shortest-path algorithm.

We find the shortest paths using C', but then compute the *actual costs* of the paths using C. Let this actual cost of the path from u to v be $C''(u, v)$. In other words,

$$C''(u, v) = \sum_{\substack{e \in \{\text{shortest path from } u \text{ to } v, \\ \text{using cost function } C'\}}} C(e)$$

Then, we use C'' instead of C in the algorithm above. Therefore, the resulting path incorporates edge preferences, but it is also under the user's desired distance Δ.

5.8 Implementation notes

Although the user actually provides a latitude and longitude for the origin and destination, we snap them to the nearest vertices in the graph. This is usually reasonable since the density of vertices is high.

We use PgRouting as a framework for routing algorithm implementations within PostgreSQL. In particular, it is used for all calls of Dijkstra's algorithm.

To improve the efficiency of routing, we use bounding boxes. We restrict the considered edges and vertices to a box within some distance of the origin and destination. This leads to up to a 10x improvement in routing speed.

6 END PRODUCT AND RESULTS

This section walks through the resulting web application and analyzes the performance of different path-finding algorithms.

6.1 Data Ingestion Pipeline

The data ingestion pipeline (known internally as `cartograph`) is described in Section 3. It relies on OpenStreetMap (OSM) as the main datasource for road networks and the associated spatial data. As objects are represented using lines and polygons in OSM, we had to convert them to nodes and vertices and create a topology from the spatial data to allow for routing. The bulk of the conversion work was done with `osmconvert`, `osm2pgsql` and `osm2pgrouting` tools.

We built a metadata processing and ingestion tool described in Section 3.3, which is capable of processing and extracting metadata from any tiled web map datasource. Greenery data and road popularity were some of the metrics that we extracted from maps. Runtime varies depending on the density of roads and resolution of data sampling, in our case we extracted and processed metadata for ~ 1 million roads in 5 minutes.

The implementation of the entire pipeline is at https://github.com/ariadnes-thread/ariadne-cartograph.

6.2 UI

The final web app consists of two major sections: the home page, which serves as a soft introduction to our project, and the more advanced preference editor for route planning.

A screenshot of the home page can be seen on Figure 5. It gives the user a short introduction to our project and lets them specify a few simple preferences to help them get started. This page was meant to give the user a basic idea of what is possible with our application and also to help them start thinking about what kind of route they want to explore. All the fields on the home page have default values, so user can just proceed to the next step without choosing anything.

Figure 6 shows the route planning page. This is the main page of our web app - it includes a preference editor, route viewer and an interactive map. When the user first opens this page, they only see the preference editor and a map. We tried to make the editor as user-friendly as possible—we provide simple input fields (multi-choice fields, checkboxes, sliders) and, where possible, "help" tooltips to describe the meaning of each preference setting. The map can be used to pick the start and finish locations for the route.

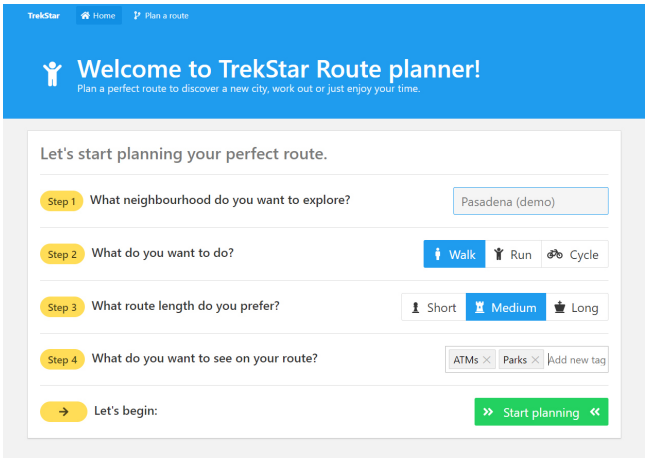

Figure 5: Home page of our web app.

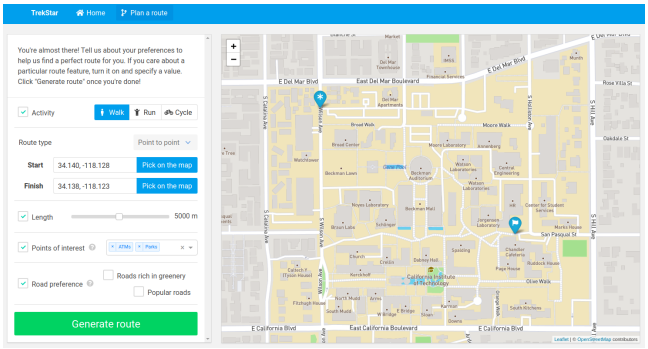

Figure 6: Default preference editor screen for route planning.

Once the user has entered their preferences and clicked on the "Generate route" button, they are presented with a route viewer tab and a visualization of their route. The route viewer tab contains the route's metadata such as its distance, elevation profile, points of interest along the way and a time estimate. Figure 7 shows an example route viewer tab. As can be seen from the figure, users have the option of going back to the preference editor tab to adjust the constraints for the route.

The generated route is also visualized on the map. Examples of route visualizations for different preferences can be seen in Figure 8. The user can visualize their progress along the route by hovering over the elevation profile plot—the portion of the route corresponding to where they are hovering is highlighted in red.

6.3 Algorithms

Our algorithms vary in speed and usability. Shortest-path routing is relatively fast, taking only a couple seconds. Some other routers need to compute all pairs of distances between points of interest, which can take several seconds. Table 2 shows a rough benchmark of round trip time of the API calls for each routing algorithm.

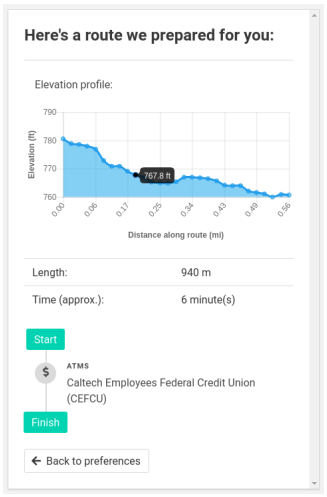

Figure 7: Example of route viewer tab for a route that visits a single ATM.

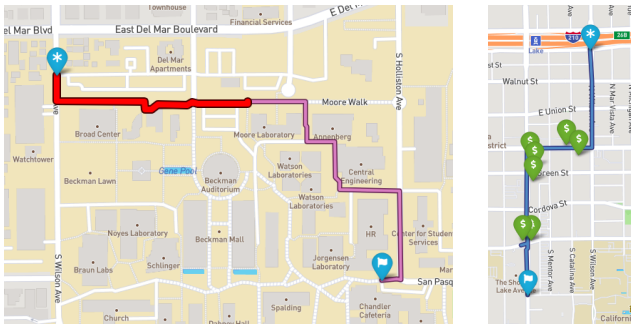

Figure 8: Example of route visualizations on the map: (Left) Route that simply takes the shortest path between two points; (Right) Route that tries to visit ATMs and parks along the way.

Algorithm	Time
Shortest path	1.8s
"POIs-on-the-way"	3.1s
"Cost-map"	2s
Orienteering	7s

Table 2: Time taken for various routing algorithms with same origin/destination and bounding boxes.

7 DISCUSSION

7.1 Lessons Learned

7.1.1 UI/UX. One of the main issues we encountered during frontend development is optimizing user interface (UI) and user experience (UX). It seemed that regardless of how advanced our backend and algorithms are, bad UI and UX design would always bottleneck the efficiency and usability of the whole system.

Optimizing UX involved decoupling everything on the frontend from the structure and data representation used in the backend, since what works well for code doesn't necessarily work well for user interaction. We solved this issue by introducing a post-processing for the preferences in the frontend - when the user hits "Submit", the web app converts all of the preferences entered through the UI into a format understood by the API. It's worth noting that these UX design challenges stemmed from the nature of our project and general difficulty of getting UX right, and not from our choice of the frontend framework. In fact, React (yet again) proved to be a very capable tool, perhaps one of the best if not the best for frontend development.

7.1.2 Backend/Infrastructure. Algorithms are hard. For our routing algorithms, there were lots of apparent edge cases or features of routes that were easy for a human to look at and evaluate, but more difficult to encapsulate in an algorithm.

Querying for data takes a long time. Obtaining information such as elevation data for a route, or ratings data for a point of interest takes a non-trivial amount of time which is not ideal for this type of application. This was somewhat mitigated by the fact that most data were queried once and stored, and thus did not affect the user experience.

Gathering data is difficult. APIs like Google Places and Yelp Fusion each have their own unique terms and conditions stipulating how the data is used and usage limits. While these issues were not too difficult to sort out for small scale use of the product, larger scale use would require us to examine this problem more closely. https://v2.overleaf.com/5947897216rpfxskwbcccm Finally, DevOps matters and it is important to ensure that development work can be carried out smoothly while maximizing the productivity of the team. Spending time to properly set up dev infrastructure such as code repositories, issue tracking, team communications via Slack, continuous integration & deployment means that we were able to be much more productive later on and move fast.

7.2 Future Work

For algorithms, one aspect of future work is user testing to determine whether the routes produced by our algorithms are attractive. If not, then we can seek feedback from testers on how to make them better, and improve them.

Another potential aspect is speeding up existing algorithms. Our current algorithms can take a few seconds to run, and making them faster would enhance interactivity. Although techniques to speed up shortest-path routing can be pretty advanced, there are still many methods to do so, such as route contraction hierarchies [8].

If we have several algorithms for each combination of preferences, we can provide *route options*, or suggest several routes per user input. Then, the user may have a better chance of receiving a route that captures their preferences.

We also had several ideas for other directions our app could grow in. In order to better capture nuanced user preferences, we could add more data from more sources to our database. Another idea to improve route quality is to incorporate user feedback into our product. The user could rate different aspects of the route, and this information could potentially be used with machine learning to predict what kind of routes users would like in the future. The implementation of this would also require some test users.

To create a more intuitive user interface, one idea was also to add a natural language description of the route to the results page. Also, given more time, we could implement more route post-processing, to smooth out some of the routes our algorithms return. There are also several features that are common to many route planning services, like travel modes and accounting for the time of day, that could be added on to our current application.

8 ACKNOWLEDGMENTS

We would like to thank Adam Wierman for his guidance, and the Caltech CMS Department for allowing this work to be done under CS 145.

REFERENCES

[1] [n. d.]. *Google Maps*. Google. https://www.google.com/maps
[2] [n. d.]. Maps & Routes. *California Bicycle Coalition* ([n. d.]). http://www.calbike.org/maps_routes
[3] [n. d.]. Strava | Run and Cycling Tracking on the Social Network for Athletes. *Strava | Run and Cycling Tracking on the Social Network for Athletes* ([n. d.]). https://www.strava.com/
[4] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. In *Algorithm engineering*. Springer, 19–80.
[5] I-Ming Chao, Bruce L. Golden, and Edward A. Wasil. 1996. A fast and effective heuristic for the orienteering problem. *European Journal of Operational Research* 88, 3 (1996), 475 – 489. https://doi.org/10.1016/0377-2217(95)00035-6
[6] Zaiben Chen, Heng Tao Shen, and Xiaofang Zhou. 2011. Discovering popular routes from trajectories. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 900–911.
[7] Thomas Erickson and Wendy A Kellogg. 2000. Social translucence: an approach to designing systems that support social processes. *ACM transactions on computer-human interaction (TOCHI)* 7, 1 (2000), 59–83.
[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 319–333.
[9] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing* 7, 4 (2008), 12–18.
[10] Jing-Quan Li, Kun Zhou, Liping Zhang, and Wei-Bin Zhang. 2012. A multimodal trip planning system with real-time traffic and transit information. *Journal of Intelligent Transportation Systems* 16, 2 (2012), 60–69.
[11] Nielsen Scarborough. [n. d.]. Number of people who went jogging or running within the last 12 months in the United States from spring 2008 to spring 2017 (in millions). In *Statista - The Statistics Portal*. https://www.statista.com/statistics/227423/number-of-joggers-and-runners-usa/ Retrieved March 16, 2018, from https://www.statista.com/statistics/227423/number-of-joggers-and-runners-usa/.
[12] Theodore Tsiligirides. 1984. Heuristic methods applied to orienteering. *Journal of the Operational Research Society* 35, 9 (1984), 797–809.