

ANAV - A* Navigation for Cyclists with Tired Legs

Chuanpu Luo, Blake Mongeon, Terrance Peters,
Omar Sanchez, Joshua Sennett, Run Zhu

Introduction

ANAV is a web-based routing application that provides users with routes customized for each user's preferences. It provides efficient route optimization based on dynamic preferences for road type, incline, and the presence of bike lanes. Through various data storage and algorithm optimizations, we support routing queries for the entire state of Massachusetts (~9 million road segments), and provide results to the user within a reasonable timeframe (seconds) depending on the route distance and urbanicity. Last, we provide a familiar and intuitive user-interface that displays results in an embedded map, with features such as auto-complete, form validation, and helpful error messages for invalid input.

Data

OpenStreetMap (OSM) is an open-source community that maintains worldwide mapping data. We use OSM data because it is the highest quality free mapping data, and because it contained road characteristics such as highway type and bike-designation that we could incorporate into providing routes optimized for user preferences. After parsing, transforming, and ingesting map data into a PostgreSQL database hosted on a Google Cloud instance, we integrate data for each node using Google's Elevation API. Finally, we include several optimizations to efficiently query this data based on user queries.

Source Data

We downloaded OpenStreetMap data for Massachusetts from Geofabrik¹, which hosts bulk downloads of regional and global OSM data in XML format. The data relevant for a routing application are nodes and ways.

A node represents a point on a map. Each node has a latitude and longitude coordinate associated with it that uses the SRID 4326 coordinate system. A node may optionally contain tags (attributes) denoting, for example, that the point is a bus stop, bicycle rack, or other point of interest. We do not use these tags to distinguish POIs from roads, but instead filter out nodes that do not belong to a way.

¹ <https://download.geofabrik.de>

Each way element denotes many types of lines and segments such as boundaries, fences, train tracks, and footpaths, but for our application we specifically consider roads and cycleways (bike paths).

Data Ingestion

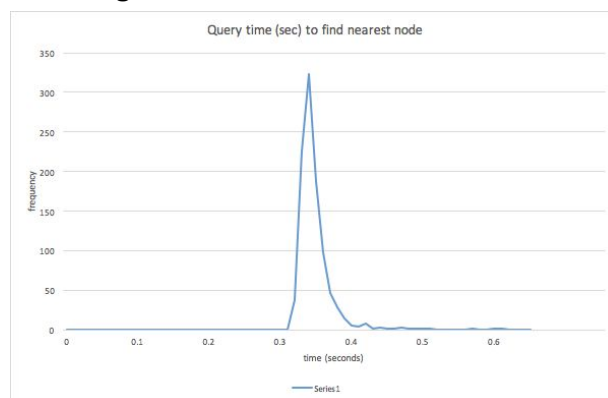
In order to query the Massachusetts OSM data efficiently, we pre-process OSM data for efficient storage, and ingest within a PostgreSQL database that uses PostGIS geospatial objects with spatial indexing. First, we use Python's ElementTree module to stream-parse elements from the Massachusetts OSM XML file. To convert semi-structured XML to a relational schema, we extract <node> elements and parse each <way> element to extract edges. An edge is a connection between two adjacent nodes; by iterating over the list of nodes in each <way>, we generate all edges in the graph. In total, we parse 4.9 million nodes and 8.9 million edges.

We use Google's Elevation API to find the altitude of each node, performing API calls for batches of nodes. The cost to use Google's API (approximately \$17 per million nodes²) was the primary reason we limited our application to support routing within Massachusetts only.

Data Optimization

We do further pre-processing of edges to improve performance of the routing component of the application. For each edge, we calculate elevation gain, incline, 3d distance, and group related highway and bicycle tags together (for example, "motorway" and "motorway_link", "lane" and "shared_lane"). The most significant optimization was to create spatial (R-Tree) indices, which significantly outperform standard B-Tree indices for spatial queries since R-Trees. With spatial indices, we found that we could query the nearest node (among 5 million nodes) in 0.35 seconds (Figure 1), as opposed to over 7 seconds using B-Tree indices, representing 20x speedup.

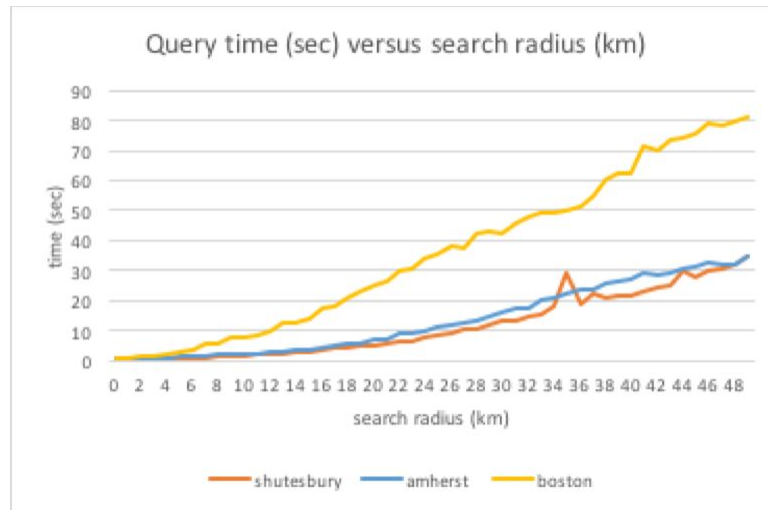
Figure 1 Time to find nearest node



² <https://developers.google.com/maps/documentation/elevation/start>

Figure 2 shows the relation between query time versus search radius for queries in Shutesbury, Amherst, and Boston (which are sparse, middle, and dense jurisdictions respectively). In Boston, our application can fetch edges within a radius of 12 km in under 10 seconds; for Amherst and Shutesbury, which have a lower density of roads, our application can fetch edges within a 24 km and 28 km radius respectively.

Figure 2: Query time versus search radius



Graph Generation

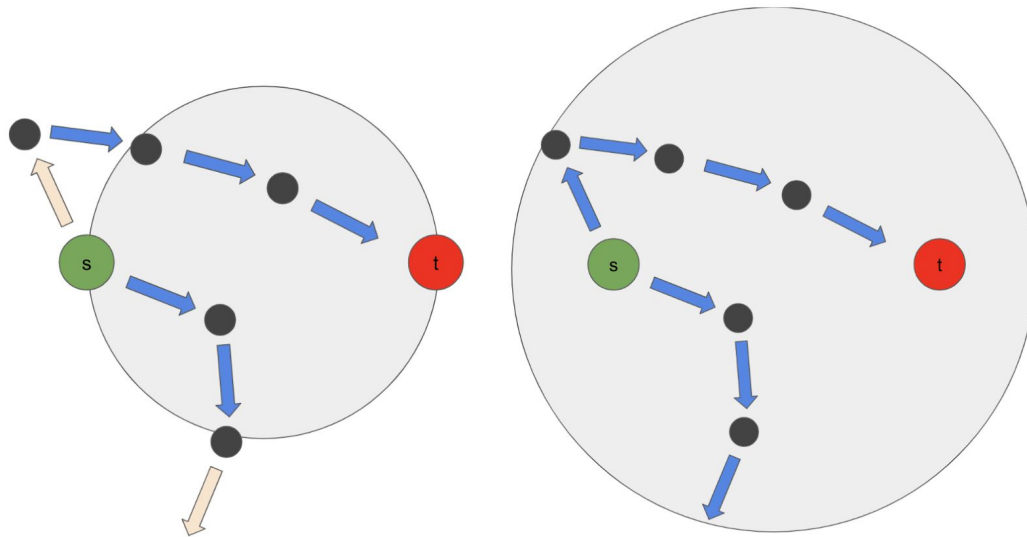
To implement graph algorithms, we must create a graph data structure consisting of nodes and edges. Creating a graph involves three parts:

1. Calculate a search radius relevant for the user query
2. Query the database for all edges in the search radius
3. For each edge:
 - a. add its endpoint nodes to the unique set of all nodes
 - b. add the edge relation to an adjacency list data structure

Since Massachusetts's road data contains almost 5 million nodes and 9 million edges, it is not feasible to run an efficient graph algorithm on the full dataset. Instead, we develop a procedure to determine a relevant search radius around the user's start and end points to create a smaller graph suitable for route optimization.

To determine this search area, we calculate the midpoint of the user's start and end positions, and calculate a radius that encompasses both start and end points. We set the radius to be slightly larger than the distance from midpoint to each point to reduce the chance that the optimal route is not included in the search area. However, this presents a tradeoff, illustrated in Figure 3: a larger search radius increases the chance of including the optimal path, but requires fetching more data and routing over a larger graph.

Figure 3: Tradeoff of buffering the search radius



Having no buffer reduces the number of edges, but cuts off the only valid path.

The buffer includes the valid path, but increases the number of edges from four to seven.

Our solution balances these tradeoffs. For longer routes, we use a smaller buffer (as a percent of the search radius) to improve the application's speed; for shorter routes, we use a relatively larger buffer, since we do not face bottlenecks with respect to the graph size.

Algorithm

A* is a widely used algorithm in pathfinding and graph traversal. We use A* with the Euclidean distance heuristic because it provides efficient performance and near-optimal accuracy. The graph optimization component takes in user preferences (sent from the user interface) and a set of edges (road segments); it calculates edge costs as a function of the edge characteristics and user preferences, and then creates a graph data structure with nodes and edges. Last, it runs the A* algorithm to find a single route that minimizes total cost, and returns the optimal route if one can be found.

Graph Creation

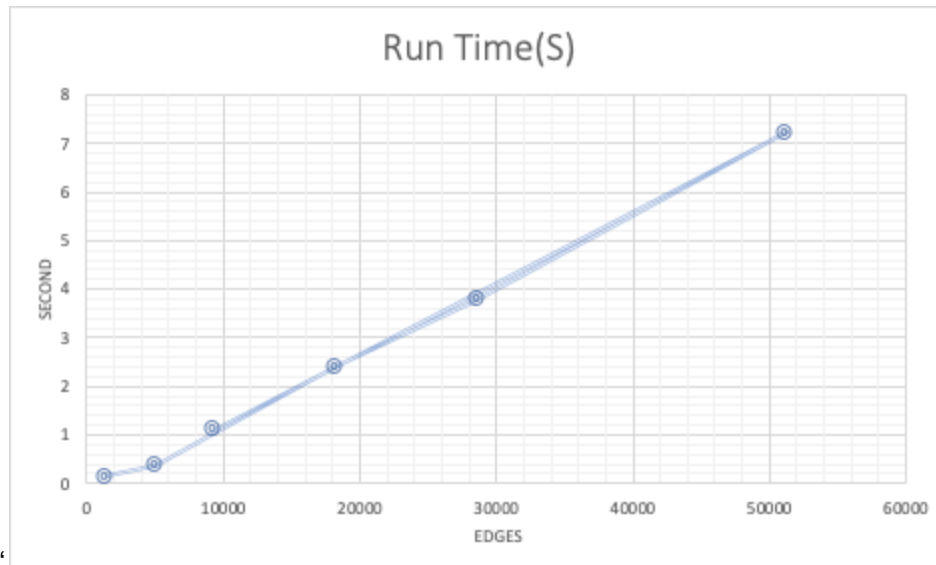
Edges contain start node and end node id, latitude and longitude, distance, elevation changes, and road type. With these information, the optimizer calculates the edge cost based on distance, incline, road type, and user preference. The optimizer will iterate over each edge and store nodes in a hash map and edges in an adjacency list.

Path Search Algorithm

The A* algorithm is a variation of Dijkstra's shortest path algorithm. The key feature of A* algorithm is its heuristic function to prioritize particular edges based on a given heuristic that tells which edge is more likely to be part of the shortest path. In our case, our heuristic function is to minimize the Euclidean distance from the end destination. In other words, our algorithm prioritizes edges that tend towards the destination.

Typical graphs have tens to hundreds of thousands of edges, so we implemented several optimizations to speed up the performance of the search algorithm. We use binary heaps to set up the priority queues for the nodes that need to be traversed and those already traversed; and, we use a hashmap adjacency list for constant-time lookup of a node's neighbors. As a result, we achieve search time that is nearly linear to the number of edges in the graph (Figure 4).

Figure 4: Speed-performance of A* Algorithm



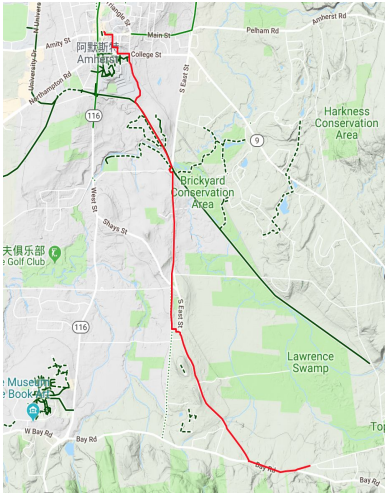
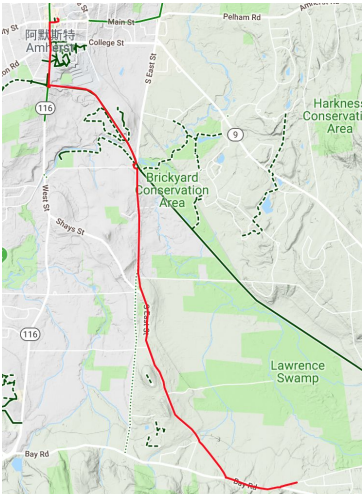
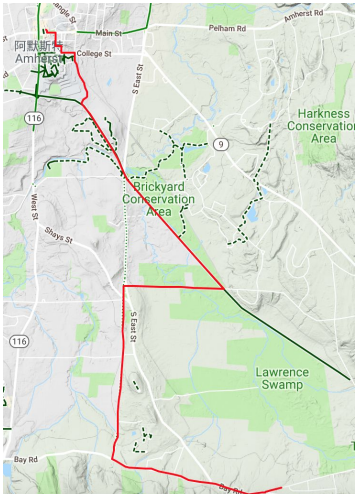
User Preference

Our algorithm changes route based on elevation changes and user preference by adjusting cost assigned to the edge. We have a multiplier as coefficient of the distance that reduce cost based on whether the edge has bike line or is the preferred road type, and elevation change multiplier that increase cost based on the degree of incline of the road.

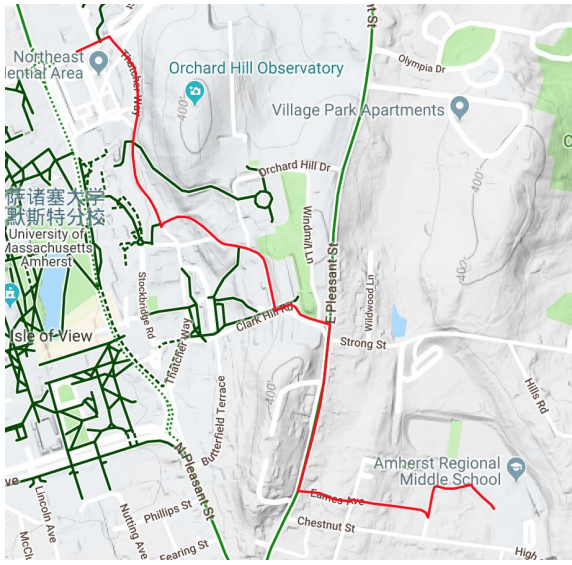
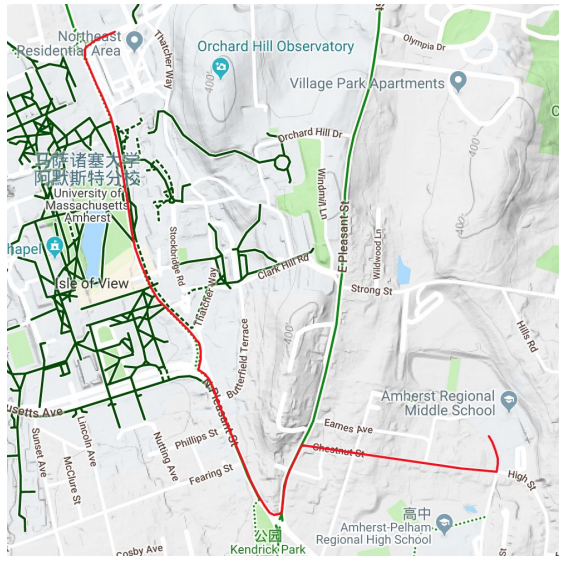
Figure 5. Black-box Tests of User Preferences

Start: 59 Boltwood Walk A, Amherst, MA

End: 24 Hulst Rd, Amherst, MA

Shortest Route	Preference for Bike-Designated Road	Preference for Primary Highway
		

Start: 30 Eastman Ln Amherst, MA
End: 170 Chestnut St Amherst, MA

Shortest Route	Flat Road Preferred
	

User Interface

Input Parameters:

The web application provides users with an intuitive interface for inputting preferences and visualizing the resulting routes. Users must input a starting point and end destination, and then may optionally choose road and terrain settings.

Input Validation:

Start Point/ Destination: A user's starting and ending position is a string address; if null, we remind users to fill out each field before searching for a path (Figure 6). Once a query is sent, we use the Nominatim Geocoder, which converts string addresses to coordinates. If the geocoder is unable to locate the string address, the user interface will display an error message (Figure 7).

Figure 6. Address left blank error message

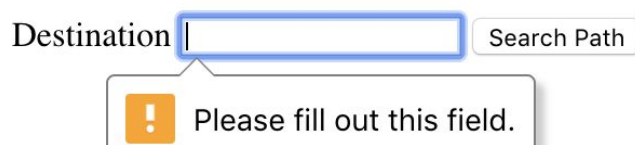
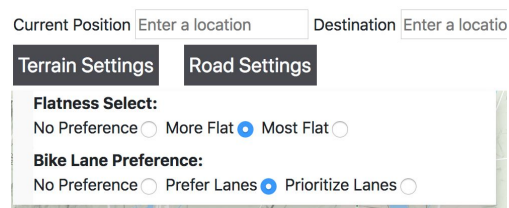


Figure 7. Address not found error message

Current position or destination was not found. Please provide a full, valid address.

Preference of Route: There are five user preferences that influence the route optimization algorithm. We allow users to choose one of three values for each domain, ranging from having no preference to having strong preferences. These preferences are displayed in two drop-down menus (categorized as “terrain settings” and “road settings”) with preference selection buttons (Figure 8).

Figure 8. Terrain and Road Settings for User Preferences



Road Settings

Motorway Preference:
 No Preference ☐ Prefer Motorways ☐ **Prioritize Motorways** ☒

Highway Preference:
 No Preference ☐ Prefer Highways ☐ **Prioritize Highways** ☒

Residential Road Preference:
 No Preference ☐ Prefer Residential ☐ **Prioritize Residential** ☒

Route Display:

We use the Google Map API to embed an interactive map that displays the returned route (Figure 9). The map is located in Amherst by default, but relocates to the start coordinate of the user's route once a route is found.

Figure 9. Route Display

