MQP GFP 1002

Rush Hour Traffic Optimization

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Matthew Runkle

Jennifer Spinney

Date: April 28, 2011

Approved:

Professor Gary F. Pollice, Major Advisor

1. Google Maps
2. Rush hour traffic
3. Geographic directions
4. Path finding

ABSTRACT

This project develops a route-planning web application that is specifically designed to work well for recurring rush hour conditions. Users can specify their trip's start and end points, the desired time of arrival (or departure), and obtain driving directions and a time estimate based on the system's knowledge of average traffic patterns in that area. Users may also provide typical traffic data for areas with which they are familiar. With each piece of user-submitted data, the system's knowledge base grows to improve accuracy for future users.

ACKNOWLEDGEMENTS

We would like to thank Professor Gary Pollice for his invaluable guidance, patience, and endless supply of amusing anecdotes. Without his support, this project would not have been possible and our experience significantly less enjoyable.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. INTRODUCTION

Computers have come a long way in the past decade to help automobile drivers discover better routes and estimate driving time. Instead of relying on road maps and the advice of friendly gas station attendants, we can now find near optimal routes with services like MapQuest® and Google Maps™. Handheld Global Positioning System (GPS) devices allow us to benefit from route-planning on the go and often have access to real-time traffic data and other important information.

Technology helps drivers with route-planning, yet a large gap still remains in the capabilities of these systems. Many people commute to work or school during typical rush-hour times when traffic patterns are drastically different than they are at 3:00AM. However, the current mapping systems have no concept of rush hour and cannot give the user an optimal route nor an accurate time estimate for their journey during this time. Some services, like Google Maps and TrafficPredict.com™, can display a general indicator of the typical traffic flow at a certain time on the major roads. In these situations, the driver is still left to piece together what his or her best route is, given this fuzzy data. Other systems, like SmarTraveler® and some GPS units, provide real-time traffic data, but this assumes the user is either currently driving or will begin shortly.

Our system fills in this gap. We designed our route-planner from the onset to work well with users who plan to drive during rush hour. This service could be very useful for a person who has an early morning interview or doctor appointment in a busy city with which they are not intimately familiar. Not only will they have access to a route designed to take advantage of the particular rush hour traffic patterns in that city, but they will also have a better idea of when to wake up in the morning and start their trip because of our more accurate time estimate. The system also benefits regular commuters by showing them routes that will (on average) cut down on their daily commute.

To accomplish this, we implemented a Web interface that allows end users to interact with our service in a familiar fashion. Users may specify their trip's start and end points and the desired time of departure in order to receive driving directions and a time estimate based on the system's knowledge of average traffic patterns in that area. Users also can provide their own observations of typical speeds at a specific time of day. With each piece of user-submitted data, the knowledge of the system grows and it provides more accurate directions.

This report summarizes the process and results of our Major Qualifying Project (MQP), which developed this proof-of-concept system over the course of an academic year. By incorporating open source software, freely available data, and information from previous research, our vision of a smart route-planning system developed into a functional, albeit limited, Web service.

2. Background

To better understand the complexities of routing vehicles through a transportation network, we review some relevant background material to provide a foundation for understanding the focus of the project. First, we explore the evolution of mapping technologies to demonstrate the importance of route planning. Next, we outline notable path finding algorithms, in particular those optimized for transportation networks. Finally, we briefly describe the technologies leveraged by our project, namely Google App Engine™ and Google Maps.

2.1. The Evolution of Road Maps

As long as automobiles have existed, road maps have been commonplace. As the cost of street cars fell such that middle-income families could own them, long distance land travel increased rapidly. Production of local, regional, and country wide road maps increased to meet the wide spread consumer demand. Vehicle owners required these maps in order to appropriately plan their trips abroad. For the majority of the twentieth century, however, people were resigned to manually determine the optimal route for their trip.

With the invention and widespread adoption of the personal computer, families were presented with a faster method of trip planning. More detailed, digital maps were available as software packages, increasing users' interaction with road data. As the Internet matured, primitive route planning services emerged to simplify street directions. Big players, like MapQuest and later Google Maps, pushed the abilities of online mapping to new levels through innovative user interfaces and robust data integration.

During this time, GPS technologies matured from exclusive use in military technologies to household devices. As their adoption increased, real-time, turn-by-turn directions evolved to become a standard, straightforward way to navigate. Many modern cars come equipped with GPS devices, making route planning as simple as driving.

The final step in the evolution from paper route maps to automatic routing GPS devices was the wide scale adoption of Internet enabled smart phones. As the Internet evolved to accommodate mobile devices and GPS units shrank in size, smart phones incorporated the technologies of both, enabling users to receive real-time, turn-by-turn directions made scalable by the size and inherent parallelism of the cloud.

2.2.Graph Theory and Route Planning

As the platform to display road maps evolved, so did the underlying algorithms to determine optimal driving paths. Competing web and GPS services constantly tweak their algorithms to gain a competitive advantage over their counterparts. Static path finding algorithms were replaced by adaptive ones to account for changing driving conditions. The realm of mathematics that concerns itself with interconnected networks such as road maps and the operations on those networks is called graph theory.

Graph theory is not a new field, but it has become increasingly relevant in the domain of transportation networks as services compete to offer the most accurate and most efficient route planner available. A graph is a made up of a set of nodes and a set of edges. Each edge describes a connection between two nodes. We use the formal notion G = (V, E) to denote that a graph G is comprised of a set of vertices (nodes), V, as well as a set of edges, E. In a directed graph, edges are unidirectional, and have a notion of their start node and end node. In a weighted graph, each edge has a weight associated with it, which often represents the cost of traversing that edge. In a road map graph, each junction can be considered a node and each road segment connecting two junctions is an edge. In this example, the graph would most likely be directed, so that one-way streets could be represented. The graph might also be weighted, where each edge weight could be the distance of that road segment. In our application, we used a directed, weighted graph where each edge weight represented the estimated time it would take to drive that road segment at a given time of day.

When a user wants to get driving directions from point A to point B, it is the direction service's job to find a path – that is, a sequence of connected edges – through the graph such that A is the first node and B is the last node. Path finding is a very common operation on graphs, and there exist many popular algorithms for doing this. One of the simplest techniques is to use to a breadth-first traversal originating from point A and stopping once we encounter node B. Breadth-first search works by expanding the search radially outward from the start, using a queue to store the nodes we should expand next. More formally, given a graph G = (V, E) and the start and end vertices *s* and *e* in V, we start by assigning every vertex in V – *s* the color white. The start vertex *s* is colored gray. We then process every gray node *g* in V (in the order in which the nodes became gray), such that if *g* is equal to *e*, the search is over. Otherwise, all of the white neighbors of *g* are colored gray and *g* itself is colored black, and we continue processing the next gray node.[1]

[1] Thomas H. Cormen, *Introduction to Algorithms*, Third ed. (Cambridge, MA: The MIT Press, 2009), 595.

As an example, consider the graph below. Instead of coloring the nodes, we will use a queue to represent the gray vertices.

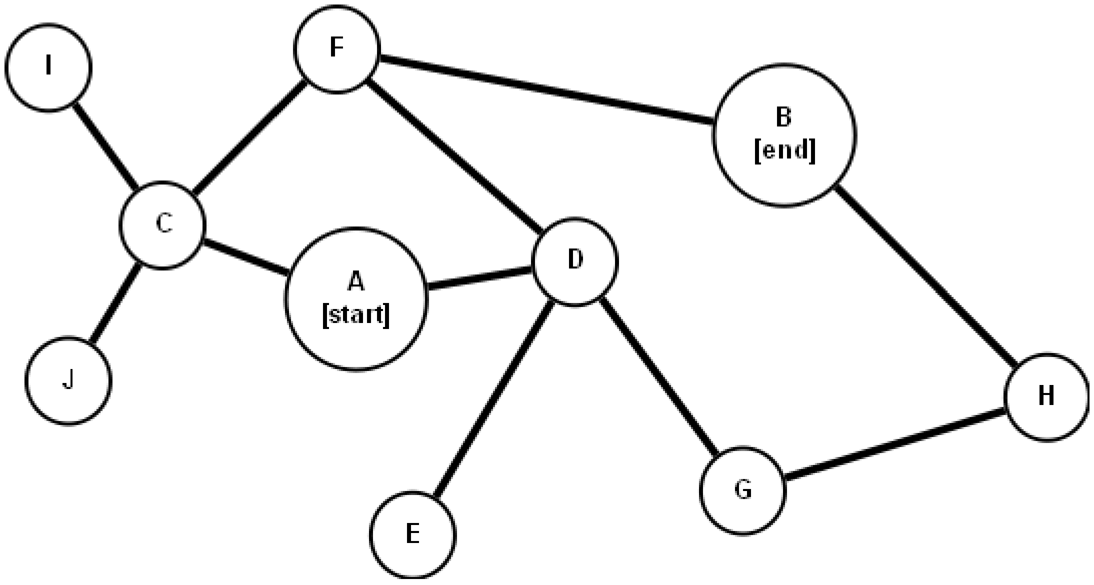


Figure 1 A sample graph, for which the user wants a path from A to B

For this sample graph, we would first add the start node, A, to our queue. We would then pop A off of the queue, inspect it to see if it is our goal node (B), and then proceed to add all of A's neighbors to the queue. The queue now contains C and D. We inspect C, see that it is not our goal node, and then add its neighbors to the back of the queue. The next node to be considered will be D, before any of C's neighbors. More generally, a breadth-first search will not consider a node n + 1 hops away from the start node until it has considered every node that is n or fewer hops away. Because of this property, breadth-first search is guaranteed to find the path from A to B that involves the fewest number of edges possible.

While breadth-first search does give the optimal path for unweighted graphs, where the number of hops is the metric being minimized, it may not give the shortest path when dealing with weighted graphs where the path distance is the sum of the edge costs. If we consider the same graph as before, but now with edge weights, we can see clearly that breadth-first search will not give the path with the lowest total path cost.

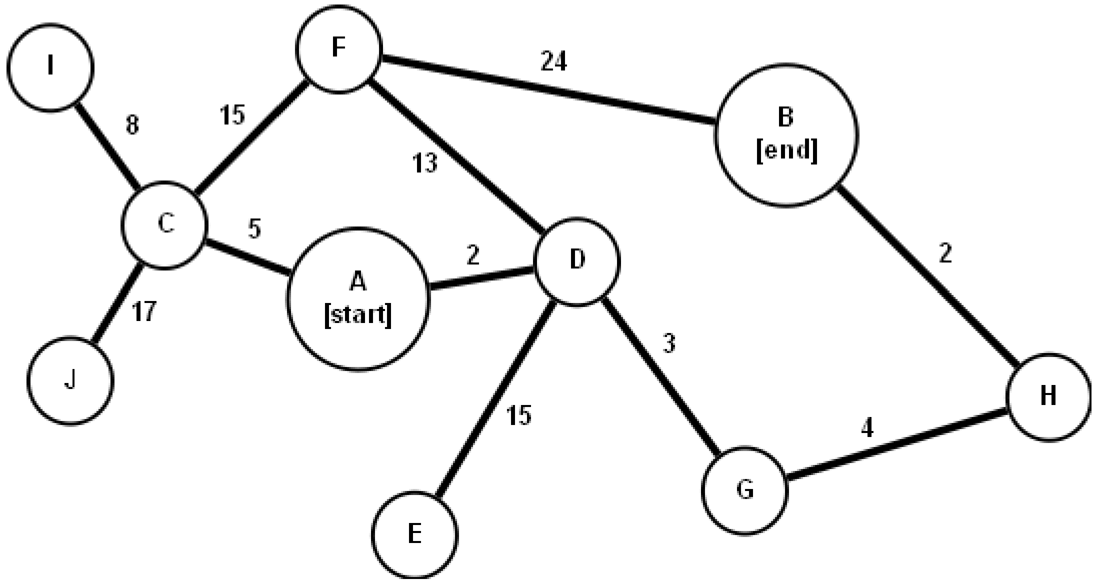


Figure 2 A weighted version of the previous graph. Breadth-first search would yield either the path A->C->F->B or A->D->F->D, both of which have a much higher total edge cost than A->D->G->H->B.

A similar search technique that does yield the path with the lowest total path cost is uniform cost search. Uniform cost search is implemented similarly to breadth-first search, except that nodes are added to a priority queue instead of a regular queue. The priority values in the priority queue represent the total cost of the minimal path from the start node to the node in question. Formally, uniform cost search is identical to breadth-first search, except that instead of considering the gray nodes in the order in which they were colored gray, we process the gray nodes in order of ascending priority values. In a uniform cost traversal of the weighted graph above, we would start by adding A to the priority queue with a value of 0. A is then popped off the queue, and the nodes C and D are added, with values of 5 and 2 respectively. The next node to be popped off will be D, since it has the lower path cost. D's neighbors are added, such that the nodes F, G, and E are assigned respective values of 15, 5, and 17. From here, either C or G will be expanded next, as they are tied with a path cost of 5. This technique continues until the minimal path is found. Whereas simple breadth-first search is guaranteed to produce the path with the fewest number of edges, uniform cost search is guaranteed to produce the path with the lowest total path cost, which is much more valuable when trying to find useful driving directions.

Breadth-first search and uniform cost search are both uninformed search techniques, but we can search more efficiently by incorporating additional information about the problem domain. An informed search strategy is defined as "one that uses problem-specific knowledge beyond the definition of the problem itself."[2] That is, if we are concerned with geographic road data, we know that there exists a computable straight line distance between any two points, whether or not they have an edge between them. Furthermore, we can say that it is often smarter to consider nodes for which the straight line distance to the goal node is small than nodes that are far away from the end node. The straight line distance in this domain acts as a heuristic – that is, a rough estimate of how good we predict a certain node will be.

A basic and somewhat naïve informed approach is to simply always expand the neighbor node with the best heuristic value. This is called greedy best-first search. The disadvantage to this approach is that it offers no guarantee of optimality. It is, however, very memory and time efficient.

[2] Stuart J. Russell and Peter Norvig, "Informed (Heuristic) Search Strategies," in *Artificial Intelligence: A Modern Approach*, Third ed. (Boston: Pearson, 2010), 92.

Is there a way that we can keep the optimality guarantee of uniform cost search but also get some of the efficiency benefits of greedy best-first search? This is goal of A* search. A* is similar to uniform cost search in that it uses a priority queue to organize the nodes to be explored next. The difference is that the priority value for each node is the cost to reach that node (as it was in uniform cost search) plus the estimated cost from that node to the goal node. Thus the value of each node is a combination of its value so far with our prediction for the remainder of the route. More formally, the priority value of each node x in the queue is given by $f(x)$, where $f(x) = g(x) + h(x)$, where $g(x)$ is the total path cost from the start node to x, and $h(x)$ is the heuristic estimate from x to the goal node.

An important property to keep in mind when designing implementations of A* search is that the algorithm is guaranteed to yield an optimal path only if the heuristic is admissible and consistent. An admissible heuristic is one which never overestimates the true cost of getting from a node to the goal. In the context of geographic data, the straight-line distance between two points is an admissible heuristic because it is impossible to have a path between the two nodes with a smaller distance. A heuristic must also be consistent for A* to perform optimally. A consistent heuristic is one in which, for every node, the estimated cost from that node to the goal does not exceed, for each of its neighbors, the edge cost to that neighbor plus the estimated cost from the neighbor to the goal. Since the point of our application was to provide users with the optimal path, it was very important for us to understand and follow the criteria for admissibility and consistency in our heuristics.

2.3.Google App Engine

Google App Engine is an open platform provided by Google to develop and deploy web-based applications. The service offers a simple web hosting environment and provides numerous tools developers may leverage to rapidly deploy workable applications. Free for low resource projects (approximately five million page views per month), App Engine contains standard Java libraries, including a Java Virtual Machine (JVM), as well as standard Python libraries[3]. This allows developers to use any language which can be compiled by the JVM or parsed by the Python interpreter. Python implementations, like our own, may incorporate components from the popular Django web framework. Django provides straightforward web development strategies like templating among others, making web development simpler and more efficient[4]. Additionally, the Python runtime in App Engine provides webapp, a framework for creating simple web applications that conform to the WSGI standard[5].

App Engine also contains Google's proprietary object-based data store, Big Table. Currently, it comes in two flavors: Master/Slave and High Replication[6]. The former is the traditional datastore which replicates data from one repository (master) to another, physically distinct repository (slave); the latter replicates data among multiple, always-

---

[3] "What Is Google App Engine? - Google App Engine," Google Code, under "The Application Environment," accessed April 27, 2011, http://code.google.com/appengine/docs/whatisgoogleappengine.html.
[4] Django | The Web Framework for Perfectionists with Deadlines, accessed April 27, 2011, http://www.djangoproject.com/.
[5] "The Webapp Framework - Google App Engine," Google Code, accessed April 27, 2011, http://code.google.com/appengine/docs/python/tools/webapp/.
[6] "Choosing a Datastore (Java) - Google App Engine," Google Code, under "Comparing the Data Storage Options," accessed April 27, 2011, http://code.google.com/appengine/docs/java/datastore/hr/.

available servers in a stable, robust manner. The datastore itself is "schemaless" and maintains data in the form of entity objects[7].

Developers wishing to use App Engine to deploy their applications must download a Software Development Kit (SDK). This creates a pseudo App Engine environment on a development machine to ensure programs are created and tested within the confines of the Google cloud. The SDK also facilitates deployment to a sandboxed web, which allows programs to run securely across multiple servers without noticeable impact on the underlying hardware.

2.4.Google Maps

Another Google product, Maps, exposes numerous APIs that allow developers to build services on top of their existing mapping service[8]. In particular, the JavaScript API allows developers to create applications that operate in a manner similar to Google's official Maps service[9]. Since the Google Maps user interface has become the de facto standard method with which to interact with a map, Google's APIs enable developers to use this interface as part of their software, reducing the learning curve for clients. The Maps APIs also expose Google's path finding and routing algorithms, transferring this complex calculation from the average developer to Google engineers.

While our final product uses its own routing algorithm and simply displays the results via the Google Map interface, the other services offered through the Google Maps APIs were critical to the development of our system. These included methods for drawing

---

[7] Mark C. Chu-Carroll, *Code in the Cloud.* (Raleigh, NC: Pragmatic Bookshelf, 2011), 50.
[8] "Google Maps API Family," Google Code, accessed April 27, 2011, http://code.google.com/apis/maps/.
[9] "The Google Maps Javascript API V3 - Google Maps JavaScript API V3," Google Code, accessed April 27, 2011, http://code.google.com/apis/maps/documentation/javascript/.

paths and creating information dialogs, as well as translating human-readable addresses into latitude and longitude coordinates.

3. METHODOLOGY

Through an iterative process, we implemented our driving directions application on top of existing Web technologies. We imported data from existing open-source stores as well as obtaining user generated input. The resulting service was exposed through Google's App Engine.

3.1. Solution Overview

The final solution was implemented as a Web interface, which allows users to interact with a street map and determine driving directions. This complex implementation is made up of several sub-systems, each responsible for a particular aspect. A breakdown of the overall system is provided in figure 3 and further described below.

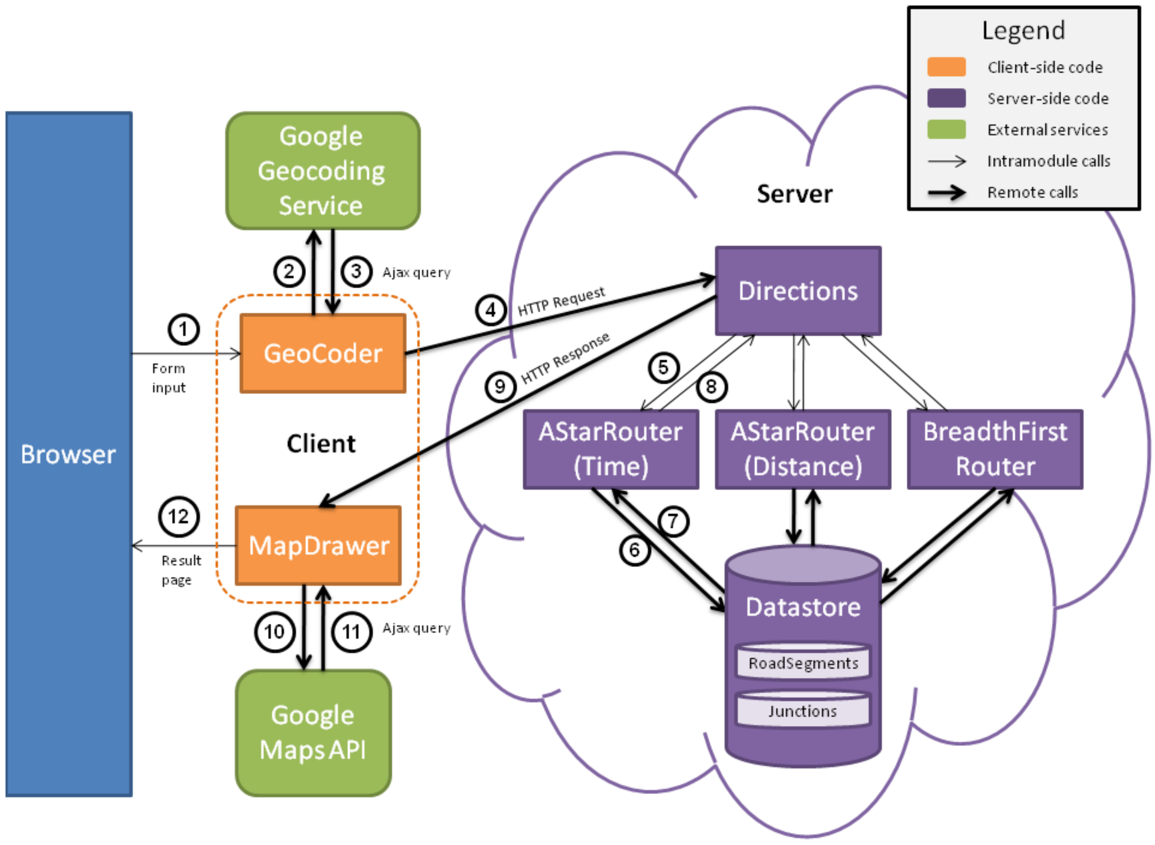


Figure 3 System Overview

A user accesses our application through a Web browser, where he or she is shown a street map of a particular area, for example Worcester, MA centered on WPI. From there, the user enters a starting location, ending location, and desired time of departure, provided both locations are within Worcester town limits (for this demo). The user's specified locations are captured client-side (1) and sent via asynchronous JavaScript (Ajax) calls to the Google Geocoding Service (2), provided through the Google Maps API. The result (3), along with the specified time of departure, is sent (4) to the server as an HTTP Request. The server accepts this request through its exposed API directions layer. From there, an appropriate routing algorithm is selected (5), namely the AStarRouter based on a time heuristic as this provides the fastest driving route. The actual driving route is calculated through numerous reads and comparisons from the datastore (6 & 7). Once the complete route is formulated, the directed path and statistics are returned to the browser (8 & 9). There the result is parsed by JavaScript methods and drawn on the map (10 & 11) via several Google Maps API calls. Finally, the complete route, time and distance information, and driving directions are displayed to the user (12).

3.2. Technologies and Implementation

The actual system is broken into two distinct parts: the client and server. The server performs most of the calculations as well as containing both street and speed data. The client, on the other hand, is primarily concerned with accepting user input and displaying the results in a meaningful fashion.

3.2.1. Server-side Implementation

When we set out to implement our driving directions service, one of the first decisions we faced was how and where to host our web application. We were impressed by the inherent scalability and apparent ease of setup associated with Google's cloud platform, Google App Engine. The fact that App Engine is a free service until certain bandwidth or computation thresholds are reached was also an important factor in our decision.

Google App Engine provides both Java and Python Software Development Kits (SDKs) for programmers using their cloud services. We chose the Python SDK as it seemed a bit simpler and more straight-forward than the Java counterpart. The Python SDK provides several classes and modules which applications must build off of when developing for App Engine.

One striking difference between programming on a cloud like Google App Engine and a traditional single server is the type of database one can use. Relational database management systems, such as Oracle and MySQL, are very popular with the traditional model, and are the paradigm most programmers are familiar with. In a relational database, data entities are modeled as tables, where each column of the table corresponds to a property of that data entity. Relations between different entity types are typically represented as their own tables. Thus, relational databases are usually optimized for performing data queries that make use of inter-table references. In a highly parallelized environment such as a server cloud, relational database are inefficient and not easy to scale. For this reason, Google App Engine uses a non-relational proprietary database management system called BigTable, which is more similar to a multi-dimensional hash map than a traditional database system. BigTable stores each individual piece of data (e.g. the phone number of John Smith) as a flat byte string. Each such value is accessed via a triple of keys: a row key, a column key, and a timestamp (which isn't used in App Engine code). Each row can have any number or type of columns within it. Each row is its own atomic unit and isn't affected by the schema or data in other rows. This allows BigTable to partition groups of rows together and store these partitions on many different machines in the cloud. However, this design means that a lot of the features and design patterns associated with typical relational databases no longer work. We developed our first draft of a database design using many techniques used in the relational system, but which didn't work well in BigTable. Figures 4 and 5 below illustrate our initial design.

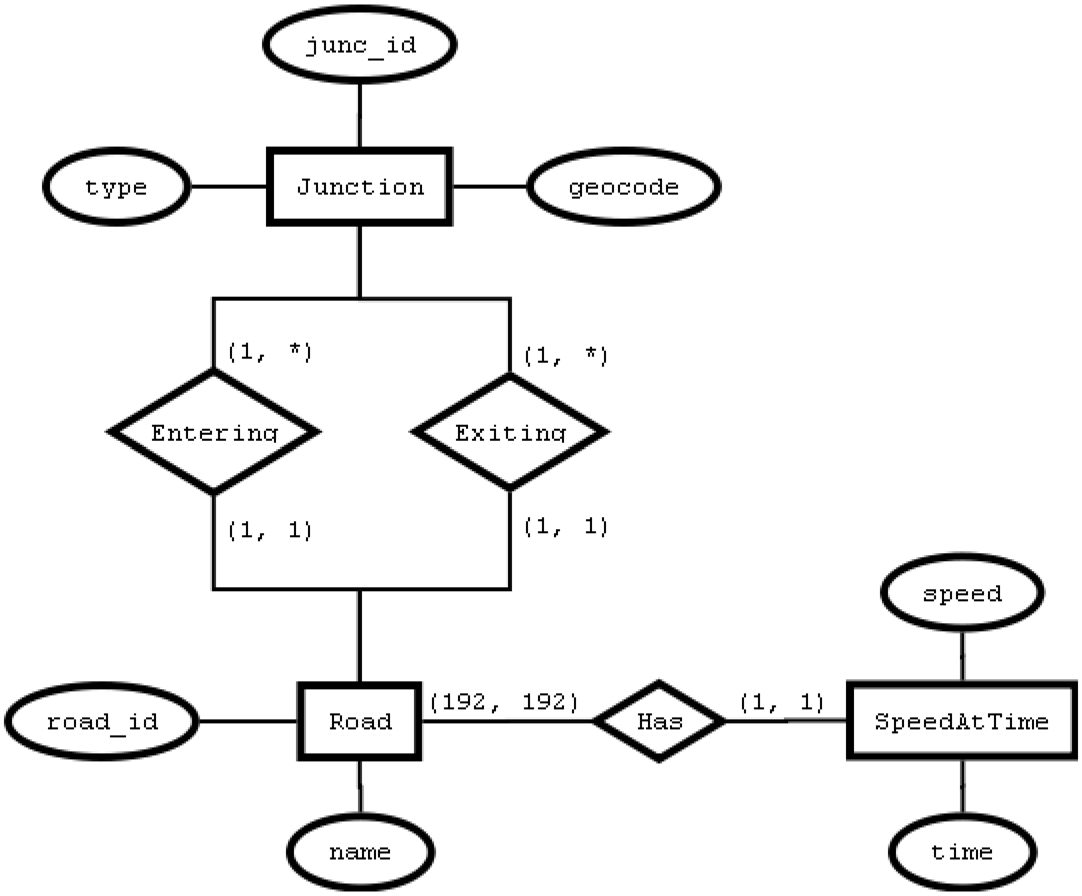


Figure 4 Entity-Relationship (ER) diagram of our initial database design. Each Road is involved in exactly 192 relationships with SpeedAtTime entities because we store a speed for each 15 minute interval throughout all 24 hours of the day, and we maintain two separate 24 hour models for weekdays and weekends.

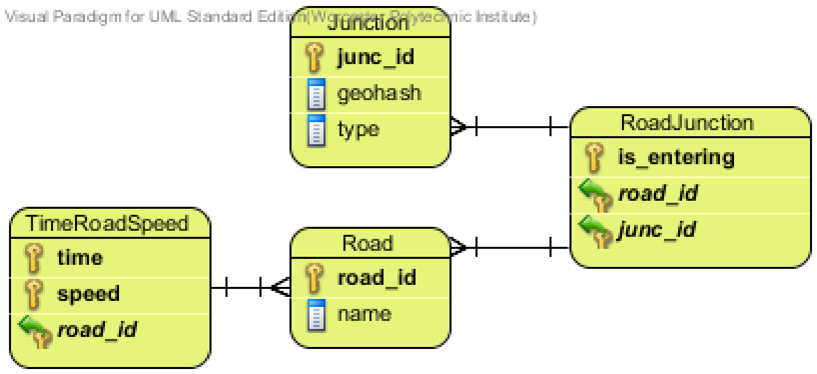


Figure 5 Database table diagram of our initial database design.

In Figure 4, we used an Entity Relationship (ER) diagram to illustrate the relationships between entity types in our data model. Each rectangle represents an entity type, and the connected ovals indicate properties or columns of that entity. Each diamond represents an inter-entity relationship. The numbers on the line connecting an entity and a relationship indicate the lower and upper bound for the number of the times the given entity is involved in the particular relationship.

Figure 5 shows the actual tables that we decided to create based on our ER diagram. The RoadJunction table is a typical relational abstraction. A RoadJunction doesn't model a physical entity itself, but rather it holds information about relationships between Roads and Junctions. While this technique makes sense in a relational setting where joins and inter-table queries are supported and efficient, we quickly learned that this layout simply doesn't make sense for BigTable, where data is stored in a way that is not optimized for relational operations.

Our final database design looked like the tables pictured in Figure 6 below. We used fewer tables than we normally would in a relational database, and much more information is stored in the RoadSegment entity.

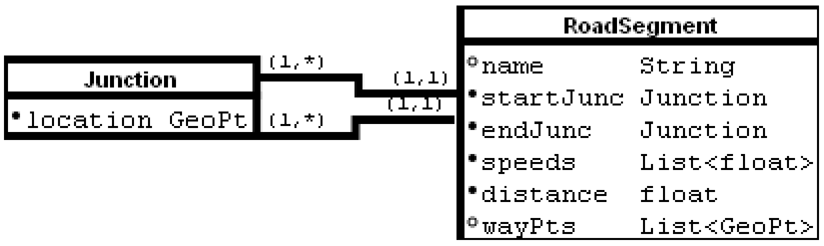


Figure 6 Final database structure

3.2.2. Client-side Implementation

For a user-driven and interactive service such as ours to be successful, it must have an intuitive and user-friendly front-end. One of our primary goals for the front-end was to create a familiar experience that reminds users of mapping and driving directions websites they've already used. We used the Google Maps JavaScript API to achieve this sense of familiarity as well as to avoid re-implementing complex functionality that already exists and is freely available. Through the API, we were able to allow our application to draw custom routes and markers on top of the popular Google Maps interface. The API also provides a Geocoding service, which returns the latitude and longitude coordinates of a given address, such as "100 Institute Rd, Worcester MA" or "Institute Rd at West St, 01609". We use this service so that users can provide the start and end locations in a natural language format.

The actual code making up the client interface is structured according to web design best practices and adapted to work with the Google App Engine framework. Additionally, the Django framework is used to achieve a fluid templating scheme that is easily maintainable. This allows us to use display code, like HTML, as a template to dynamically create web pages based on the content returned from the server. A master template exists containing site-wide components, like site navigation as well as the structure of the entire website. Additional components can then be generated and inserted into this master template, essentially allowing inheritance within HTML.

Styling and dynamic manipulation of Webpages are handled through Cascading Style Sheets (CSS) and JavaScript files respectfully. Both of these file types, along with image files, are considered static by Google App Engine since they do not change regardless of the server output and are simply linked by the final webpage. Therefore, they are stored within a specific folder marked by App Engine to be static. This increases efficiency as the files are stored in memory as a single copy and linked as necessary (as opposed to generating a new copy upon each page request).

Each page, therefore, is a mix of dynamic and static files. When a user clicks a link or submits a form, the server processes the request, then uses the result to assemble the HTML website code through numerous substitutions, and then links in the appropriate static files. The result is a custom webpage displayed as HTML, styled with CSS and images, and enhanced with JavaScript.

3.3.Path-finding Algorithms

The core of our application rests on efficient and intelligent path-finding through a weighted graph. Several types of path-finding algorithms are described in Section 2. When implementing the path-finding portion of our application, we started out as simply as possible, in order to test that the basic structure of our application made sense and gave legitimate results. Thus, we started with a simple breadth-first search that does not take into account edge weights at all, meaning that the path generated will be optimal in the number of edges traversed, but not optimal in the distance or time elapsed over the course of the journey. Starting with a simple algorithm also forced us to think very carefully about exactly how we access our data. As we developed this router, we began to realize a lot of the problems inherent in our initial database design, as described in Section 3.2.1. The way in which we interacted with the data from a router's point of view guided the way in which we restructured our database schema.

Once we were satisfied with the behavior of our breadth-first algorithm, we implemented an A* Router which finds the route with the smallest total distance. As soon as we began implementing this additional class, we saw an opportunity for abstraction between multiple routing algorithm implementations. We pulled out some methods from the BreadthFirstRouter that would be useful to other router types and put them in a Router superclass, which is extended by both BreadthFirstRouter and AStarRouter. We also wanted to use this superclass to define a set of common methods for interacting with routers. In a language like Java, this goal could be easily achieved with an interface, but Python does not have such a construct. Python also does not provide built-in support for abstract methods, so create this interface-like construct, we defined methods in the Router class which would throw exceptions if they were ever directly called.

Once we were comfortable with our distance-based A* router, we implemented a time-based version of the same router, that used our road speed time and took into account the departure time of the trip. The two A* routers shared a lot of code, so we refactored our design again. We pulled out the common A* algorithm functionality into an AStarRouter class, and used a compositional approach to differentiate between distance-based and time-based routing. We created a superclass called Heuristic to capture the type of metric used and heuristic function associated with that metric. We had two subclasses of Heuristic: DistanceHeuristic and TimeHeuristic. An AStarRouter path-finding function has a reference to a generic Heuristic object, so to create an AStarTimeRouter, we just initialized an AStarRouter with the TimeHeuristic for its heuristic object. Our final routing class hierarchy is shown in Figure 7.

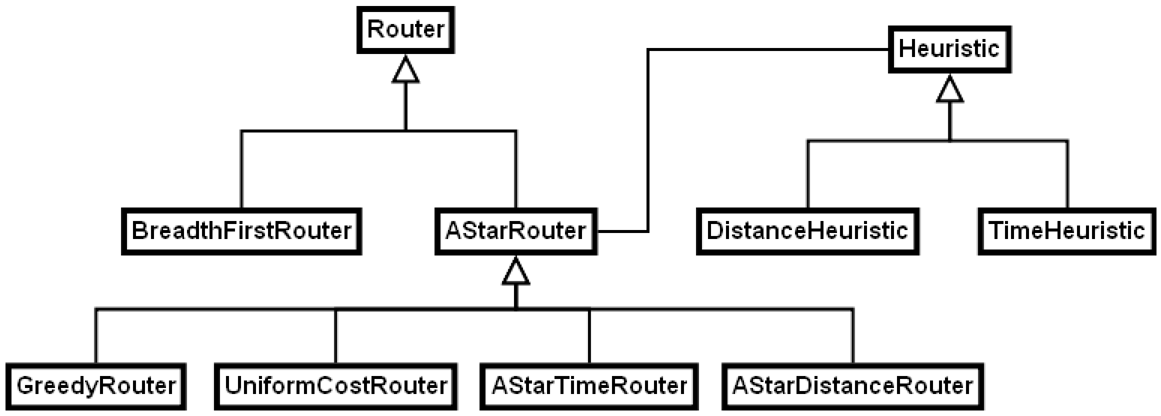


Figure 7 Class hierarchy for our router (path-finding) implementations

3.4. Acquiring and Importing Road Data

One of the biggest challenges during the course of this project was finding and integrating real road data into our system. Services such as Google Maps provide a means to programmatically interact with their maps, but it is against their terms of service and not supported in their API to download and store road map information. Thus, we had to find a source dedicated to provided free and open source data. The first such resource we considered was OpenStreetMap™, a free and user-maintained database of road information spanning the globe. Their data is available under the Creative Commons Attribution-ShareAlike 2.0 license[10] and freely downloadable in Extensible Markup Language (XML) files. Unfortunately, we ran into difficulties porting their representation of road data into our own. Their data does not have the notion of an intersection which can be differentiated from other points of interest, such as restaurants or shops. We struggled for a while trying to develop our own means of determining whether a node was an intersection or not, but

[10] "Copyright and License," OpenStreetMap, accessed April 27, 2011, http://www.openstreetmap.org/copyright.

no of our attempts were successful. For this reason, we looked into a different source of data, the Commonwealth of Massachusetts government.

The Massachusetts Office of Geographic Information (MassGIS) makes available to the public state-wide road data as maintained by the Massachusetts Department of Transporation (MassDOT). An obvious limitation of this approach is that this data only applies to Massachusetts and other states and countries have other ways of representing their road data. For a simple prototype however, limiting ourselves to Massachusetts was not a problem. Compared to the simple XML of OpenStreetMap data, however, the MassGIS roads layer was more complicated to parse and translate into our data representation. The data is stored in the shapefile format, a popular format used by many (Geographic Information System) GIS software applications. A shapefile is actually a collection of several files, which must be processed as a whole. On top of the basic shapefile structure, MassGIS uses its own schema for describing road information. While more complex, this data was also much more accurate and more complete than the OpenStreetMap data. It was also not a problem to determine intersections between roads. The only limitation of the MassGIS data was the fact that while one-way streets were clearly indicated, there was no mechanism for representing which way the street went. Aside from this sacrifice, we were successful using the MassGIS data.

Another challenge was uploading the road data into our datastore in the cloud. Google App Engine has strict limits on how much time a query can take and how big an externally referenced file can be. For this reason, we initially developed a compact custom file format that we could use to communicate road data to our cloud application as efficiently as possible. In the end, however, this turned out to be unnecessary. As we continued to be frustrated by the time and resource limits, we found a feature of Google App Engine specifically designed for bulk data transfers, called the bulk loader. We eventually got our data successfully uploaded through this API.

3.5.Acquiring Speed Data

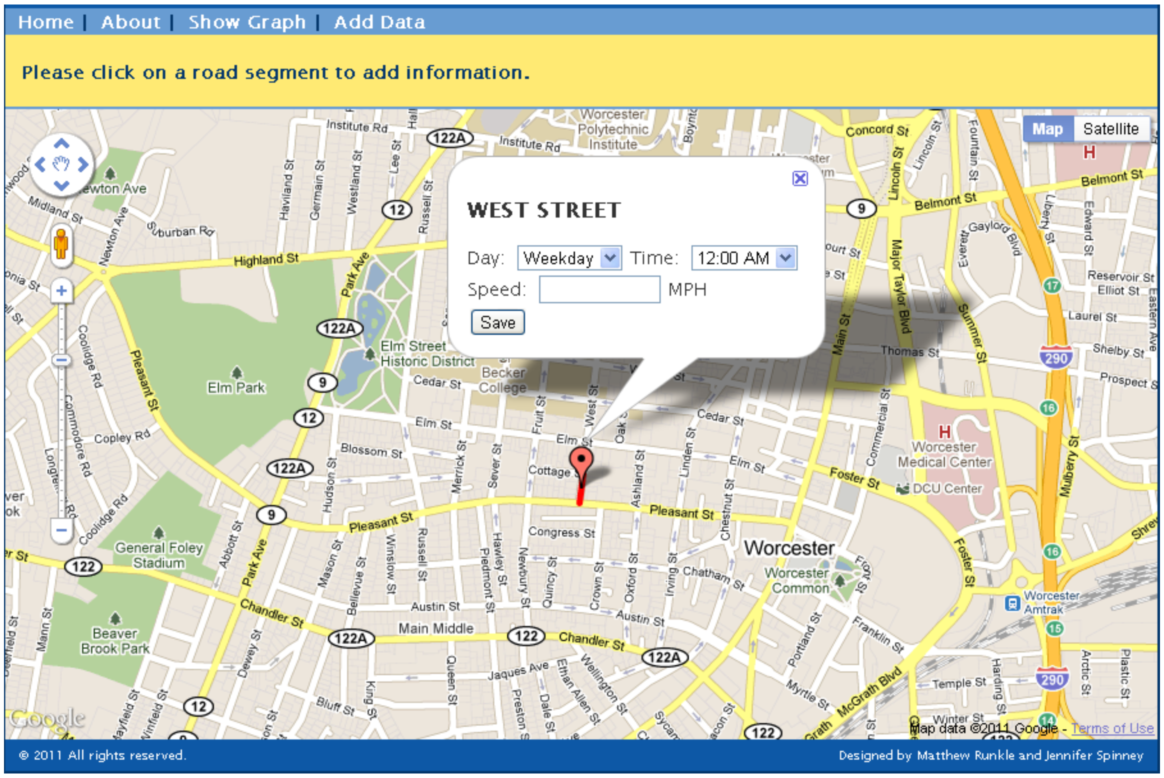
Unlike street data, there is not an easily accessible source of historical rush hour traffic data. Even contemporary mapping services do not maintain accurate traffic data beyond color coding main highways. Since our applications focuses on local roads, we were left with few options to obtain effect road speeds. Realizing the only true source of traffic conditions resides within the minds of the drivers caught in traffic, we set out to centralize this information. We designed an interface in our application to gather as much speed information as possible from our users through a process known as crowdsourcing.

Crowdsourcing “is the act of outsourcing tasks, traditionally performed by an employee or contractor, to an undefined, large group of people or community … through an open call.”[11] With enough user input, the effective speeds entered will converge upon an average value that should accurately approximate the true speed of a particular road segment during each fifteen-minute time interval. Initially, each speed value was set to the speed limit of the road; adding data points then alters this value.

The interface for adding speed information is contained within an alternate tab on the demonstration website. A user simply clicks on the road he or she wishes to add information about. The system highlights the appropriate road segment and generates a

[11] "Crowdsourcing," Wikipedia, the Free Encyclopedia, accessed April 27, 2011, http://en.wikipedia.org/wiki/Crowdsourcing.

form for the user to fill out. The user selects the day and time and specifies the effective speed at that location. From there, the server manipulates the datastore to incorporate the new information. Figure 8 shows a screen shot of this.


Home | About | Show Graph | Add Data
Map
Satellite
Save
Worcester
Elm Park
Google
Map data ©2011 Google - Terms of Use

Please click on a road segment to add information.

WEST STREET

Day: Weekday Time: 12:00 AM

Speed: MPH

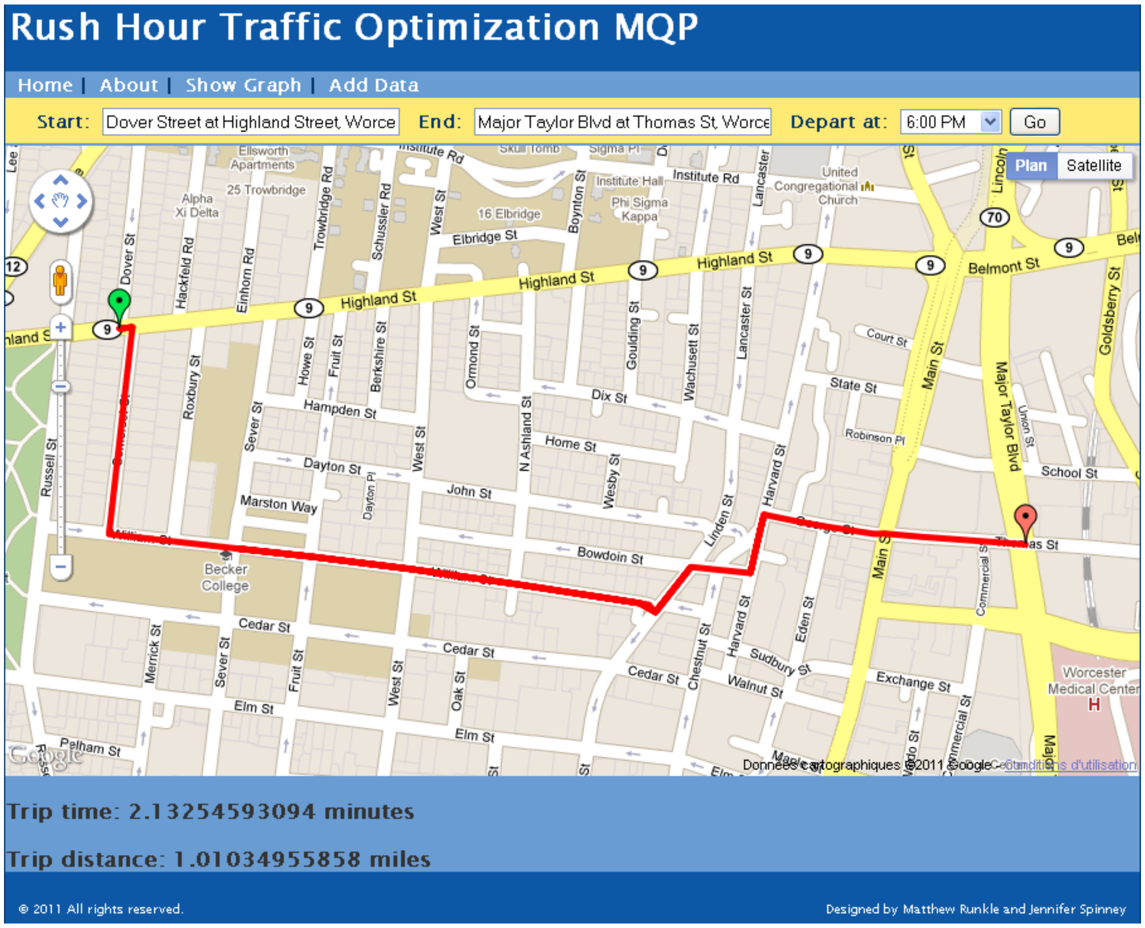
© 2011 All rights reserved.

Designed by Matthew Runkle and Jennifer Spinney

Figure 8 Crowdsourcing interface

4. Results and Analysis

Our project was successful in that we created a functional website where users could query for directions within Worcester, MA and get different routes depending on the departure time they selected. Our front-end succeeded in converting user-submitted addresses to geo-coordinates, sending a request our backend, and then correctly visually displaying the route along with a correct textual description of the directions. The backend correctly generated different paths for different times of the day, all of which appeared to be optimal for the speed data at the given time of day. An example of a rush hour route that our system discovers and displays is shown in figure 9.


Rush Hour Traffic Optimization MQP

Home | About | Show Graph | Add Data

Start: Dover Street at Highland Street, Worce End: Major Taylor Blvd at Thomas St, Worce Depart at: 6:00 PM Go

Trip time: 2.13254593094 minutes

Trip distance: 1.01034955858 miles

© 2011 All rights reserved. Designed by Matthew Runkle and Jennifer Spinney

Figure 9 When routing from the green marker to the red maker, our algorithm correctly selects a back-roads path instead of the usually path through Highland St. Highland St is correctly used when the user departs at an off-peak time.

While our application met our original goals, it is still a prototype and not ready for widespread general use. For instance, we only imported road data from the city of Worcester, so inter-city directions are not possible at this time. We also have a problem with the data itself, in that we cannot determine from MassGIS which direction one-way streets travel. In addition, the running time of our algorithm is quite a bit slower than what most users on the web have come to expect, since we did not implement any caching techniques.

While the general running time was subjectively somewhat slow, we did investigate the relative running time of the different algorithms in our application. For each algorithm, we ran five trials of the same start and end point query and measured the elapsed wall-clock time. The average for these five trials is represented in the graph in Figure 10 below.

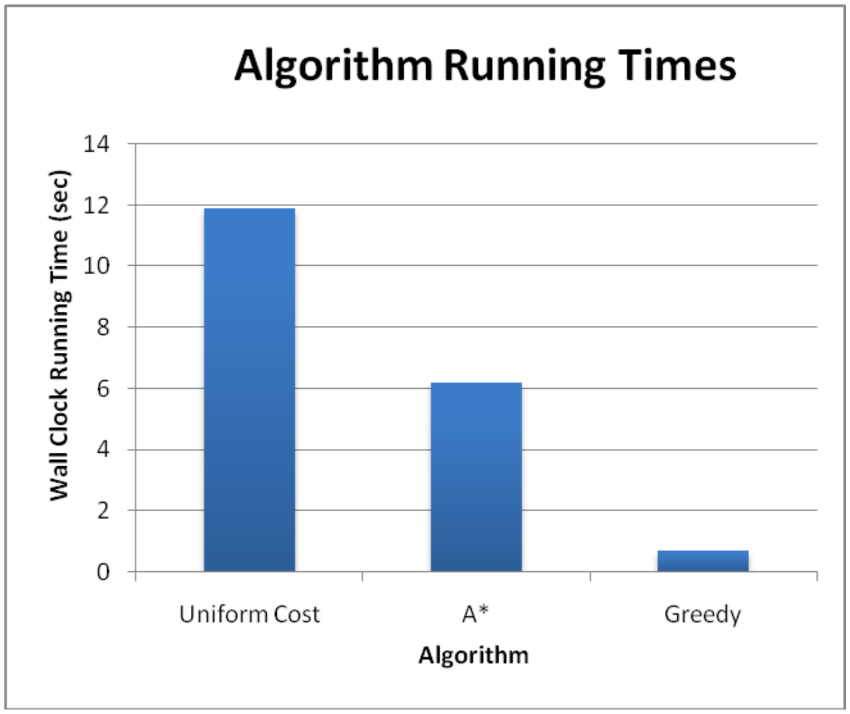


Figure 10 A comparison of the running times of our path-finding algorithms

We could clearly see that the heuristic of A* significantly cut down on the running time of the uninformed uniform cost search. Greedy best-first search was the fastest of all our algorithms, but because it does not yield an optimal solution, we cannot use it in our application. Overall, A* was the best combination of optimality and performance.

5. FUTURE WORK AND CONCLUSIONS

While we were able to complete a functional proof-of-concept system that accomplishes the initial goals of the project, there are a few improvements and extensions that would transform this MQP from a demonstration system to a robust application. Future developers should consider enhancements like the addition of a mobile application to shoring up the integrity of the street data to increasing the efficiency of the code.

5.1.Mobile Application

After creating the web interface for adding speed data, we realized the need for automatic reporting. Two issues arose almost immediately: 1) the interface was cumbersome and time consuming, and 2) the data was inaccurate. Both issues, however, stem from roughly the same fact: drivers do not remember the exact times and speeds they were traveling for every leg of their trip. Asking people to recall this data after their trip is complete and then force them to insert the values repeatedly for each road segment is simply not practical. This served as a great interface to show proof-of-concept, but not for mainstream use.

Therefore, a mobile application to be run on smart phones seems like an appropriate solution to mitigate this issue. Drivers are already used to operating handheld GPS devices or mobile navigation programs while traveling. A mobile application for this project would allow drivers to use our service anywhere, while simultaneously recording accurate time and speed data to improve our application. When speaking to numerous people over the course of this project, a mobile application sporting these features was requested almost unanimously.

5.2.Street Data

While there is an abundant source of street data available outlining the topography of cities, it seems there is not a single, fully accurate source. The street data contained within our datastore is accurate enough for demonstration purposes; however, it should not be used for actual direction generation since some inaccuracies remain. For example, the provider of our final source of street data, MassGIS, did not provide the directionality of one-way streets. Therefore, the current data is a "best guess" approximation of the direction. Lancaster Street, for example, flows in the opposite direction within our application than it does in reality. This could lead to unfortunate mishaps if used for actual routing. A truly accurate information source needs to be located to ensure the street data is completely correct. This may cost money, but would be worth the investment.

5.3.Real-time Traffic

This application initially set out to route drivers around rush hour traffic based on historical data. It was mainly to be used for route planning rather than real-time navigation. However, it would be useful if a person were able to see live-traffic conditions as well. Imagine a situation where a road is normally clear. The current implementation may route someone down this road since it usually has no traffic. What if an accident occurs? A driver might find him- or herself stuck in traffic and frustrated at our service. Real-time updates would prevent such occurrences from happening, adding additional robustness.

5.4. Efficiency

Since the application is implemented on the Google cloud, there exists the possibility of massive parallelization, which is currently untapped. Google App Engine has this functionality built-in so such a change would not be overly difficult. Our routing algorithm could be reconstructed to split the workload of constructing and traversing a weighted-graph into multiple concurrent threads. The initial graph creation step could also factor-in the time-difference as one moves away from the starting point. Currently, this is done on the fly, which impacts performance.

Google App Engine also has a built-in caching feature. This could be used to store commonly traversed routes to reduce datastore queries. Reducing calls to the datastore will dramatically improve performance as each query is rather expensive. The current algorithm queries the datastore at least once for each RoadSegment traversed.

5.5. Security

As a proof-of-concept application, security was not considered during development. Therefore, numerous vulnerabilities exist for malicious users to exploit. Popular web exploits, like Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) vulnerabilities could exist within the application, namely the form submission code. Another area of concern would be the exposed API points in the back-end. There is no validation key or location checking. Therefore, it would be possible for external sites to use this to access our data, or serve as a launching point for a Denial of Service attack.

5.6.Conclusion

Even though there exists numerous areas in which this project could be improved, it remains a successful proof-of-concept application. Clearly this MQP demonstrates the feasibility of building a crowdsourced, rush hour traffic prediction system that serves to alert drivers of potential congestion points and accurate driving times. With the improvements specified above, this application could become a viable and useful source of information.

REFERENCES

"Choosing a Datastore (Java) - Google App Engine." Google Code. Accessed April 27, 2011. http://code.google.com/appengine/docs/java/datastore/hr/.

Chu-Carroll, Mark C. *Code in the Cloud.* Raleigh, NC: Pragmatic Bookshelf, 2011.

"Copyright and License." OpenStreetMap. Accessed April 27, 2011. http://www.openstreetmap.org/copyright.

Cormen, Thomas H. *Introduction to Algorithms*. Third ed. Cambridge, MA: MIT Press, 2009.

"Crowdsourcing." Wikipedia, the Free Encyclopedia. Accessed April 27, 2011. http://en.wikipedia.org/wiki/Crowdsourcing.

Django | The Web Framework for Perfectionists with Deadlines. Accessed April 27, 2011. http://www.djangoproject.com/.

"Google Maps API Family." Google Code. Accessed April 27, 2011. http://code.google.com/apis/maps/.

"The Google Maps Javascript API V3 - Google Maps JavaScript API V3." Google Code. Accessed April 27, 2011. http://code.google.com/apis/maps/documentation/javascript/.

Russell, Stuart J., and Peter Norvig. "Informed (Heuristic) Search Strategies." In *Artificial Intelligence: A Modern Approach*, 92-102. Third ed. Boston: Pearson, 2010.

"The Webapp Framework - Google App Engine." Google Code. Accessed April 27, 2011. http://code.google.com/appengine/docs/python/tools/webapp/.

"What Is Google App Engine? -- Google App Engine." Google Code. Accessed April 27, 2011. http://code.google.com/appengine/docs/whatisgoogleappengine.html.