Topocycle

Jeff Beorse, James Lee, Bo Qin, Daniel Suskin
CSE 454 Autumn 2009

1 - Goals

Our original goal was to create a Google Map-like website in which users can search for a route from point A to point B that takes both distance and elevation change into account. Our target audience is bike commuters who prefer to avoid hills to and from work. We would also like the site to allow bikers to share routes that they like, and to allow others to search through those routes to discover new wonders in their neighborhoods. Initially, our goal for routing coverage was the Seattle area, but we were able to cover the entire King County region.

As for the backend functionality, we wanted the routing finding algorithm to be fast. It'd be ideal if it returned a route in under five seconds, and favored long shallow slopes over short steep ones. On the user interface side, the routes should be marked accurately on the map. We also wanted the user interface to be very intuitive; users who are already familiar with the Google Map UI should be able to pick it up very easily. The bike route submission and searching interfaces should also be easy to use.

2 - Design and Algorithms

2.1 - Overview

The basic system consists of three portions: the front end, the routing back end, and the community back end. The back ends all communicate with the front end via PHP, and process data stored in a MySQL database. The database for routing contains a table of x, y, z triplets of longitude, latitude, and altitude for all of the polyline segments in King County, and another table containing the x, y endpoints of each polyline segment in King Couny. The front end interfaces with Google Maps API code.

2.2 - Front end

To show a map and our calculated routes to the user we used the Google Maps API. This provided us with a map object that users are already generally familiar with, the ability to place custom markers and polylines on that map, and geocoding functionality. The geocoder is used to convert the user input to corresponding Google LatLng objects which contain the latitude and longitude of the text address. An event listener is used to track where a user right clicks on the map, and returns a LatLng object with the coordinates of the click. Furthermore, a reverse geocoder is used to convert these resulting LatLng objects to text addresses to display in a text box input. This was done to create continuity between the two methods of inputting points. The LatLng objects corresponding to user selected addresses, both by text box and right click, are displayed on the map object using marker objects with custom images. These images are modeled after typical Google Maps markers, but each one has a letter on it corresponding to the text box that specifies it. Finally, routes are displayed on the map after creating an array of LatLng objects from the latitude longitude pairs returned by our route finding algorithm and displaying that array as a polyline.

2.3 - Community

The backend for the community side of Topocycle consists of three sections. The sections are route searching, route retrieval, and route submitting. The PHP service that handles requests to the backend has the three portions in the same file, using flags to switch to the appropriate section that handles corresponding requests.

Initially the database that contained the routes had no data; we had to scrape bike routes that had already been submitted to bikely.com. We built a crawler that traversed the relevant areas of the site grabbing the xml route data as well as any metadata we could find. The crawler grabbed the relevant routes over the course of three days running into problems in which it found critical errors that resulted from unexpected structures and downtimes of the bikely website. After the data was compiled, we wrote an additional script to parse and extract the start and end points as well as the total length of the route from the metadata. When writing the script, we had to handle and circumvent poorly written and unparseable xml documents.

After we inserted the routes into the database, we designed the query system to take user input. Before interfacing with the database, all input strings are escaped as to not cause any unintended and malicious problems with our systems.

After the user chooses a specific route, the front end sends requests to the back end to get the route data and all metadata from the database. The back end does a basic query and retrieves all relevant information for the front end.

When users want to submit information to the database, the service checks if the user specified a name and username, as all routes are associated with a username, and those are the only two required fields. The server then validates all input and matches the username with existing users and verifies that a specified pathname is not already in use. The server receives latitude and longitude coordinates from the front end of the website and parses them to get the start and end points and the total length of the route. Once all validation is confirmed, the server saves the route, the metadata, and creates new users as necessary.

2.4 - Routing

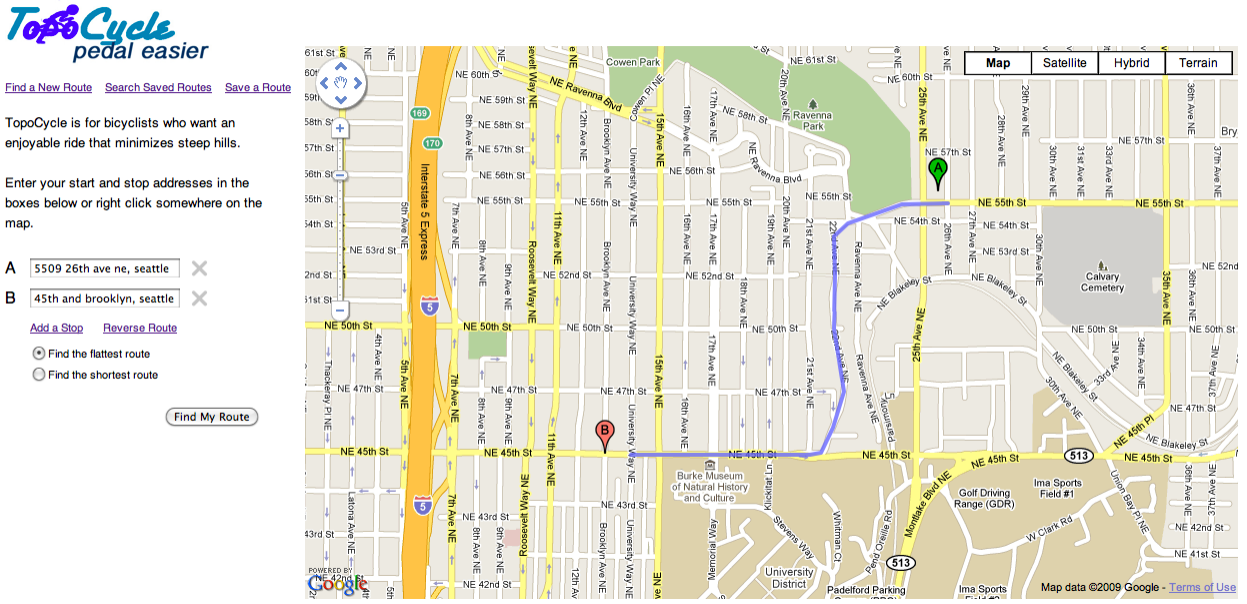
To create the route searching portion of our website, we began by looking for open-source routing packages that we could build upon. The most prominent candidate we found was pgRouting, which uses stored routines in PostgreSQL to find routes so long as the database records follow the correct format. It supports multiple search algorithms. We decided to use its A* search and eventually modify the cost function to factor elevation change into the cost, but that proved too difficult for how easy it would be to implement our own A* search. Additionally, pgRouting didn't meet our performance goals; route queries generally took between five and ten seconds to return a result.

So we decided to move on and write our own version, using Java. To better explain the A* heuristic we used, let us consider the case of calculating the distance from point A to point B. Let x_A, y_A, z_A, and x_B, y_B, z_B be the longitudes, latitudes, and altitudes of points A and B, respectively. We calculate the cost from point A to B using the following snippet of pseudocode:

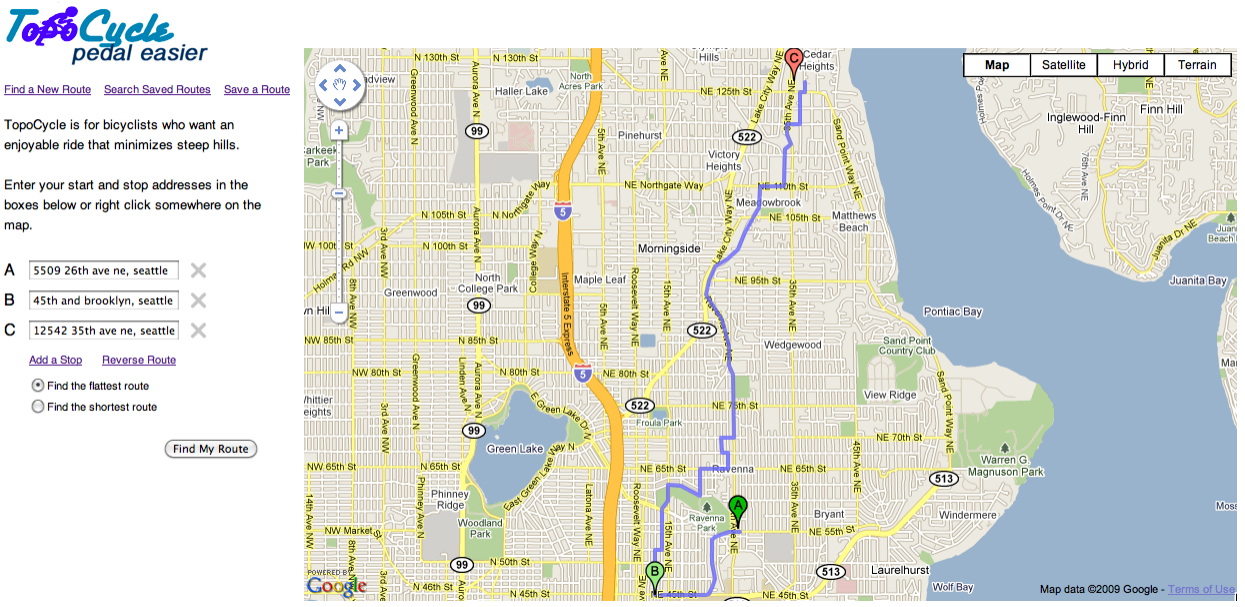
$Cost = sqrt((x_A - x_B)^2 + (y_A - y_B)^2)$
$If\ z_B - z_A > 0$
  $Cost \mathrel{+}= sqrt((z_A - z_B)^2 + Cost^2) * (z_B - z_A) / Cost$
End if

To speed up the route-finding time, we created new database tables with all of the unnecessary data that pgRouting used stripped out, and with indices in place to speed up the database queries. We also only query the database once per route, and only for a limited area. The area currently is the area formed between the start and end points as the rectangle for which the start and end point are points diagonal to each other. The rectangle's width and height are both increased by a constant factor, to account for some routes which double around behind the start or end point.

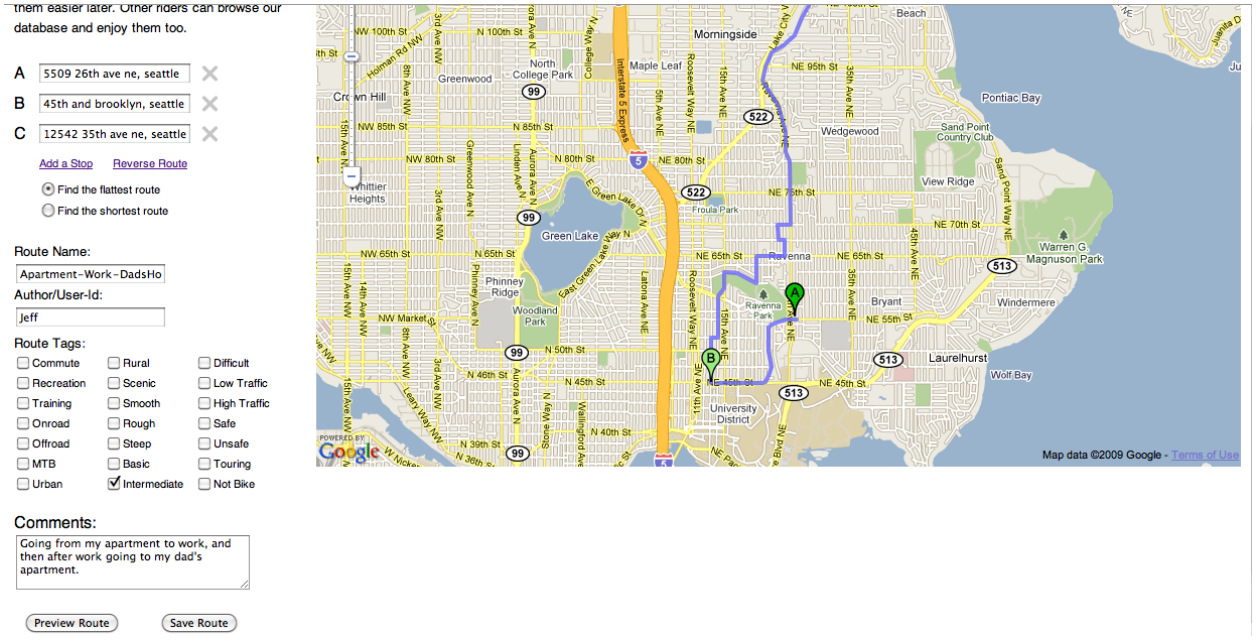
3 - The System in Use



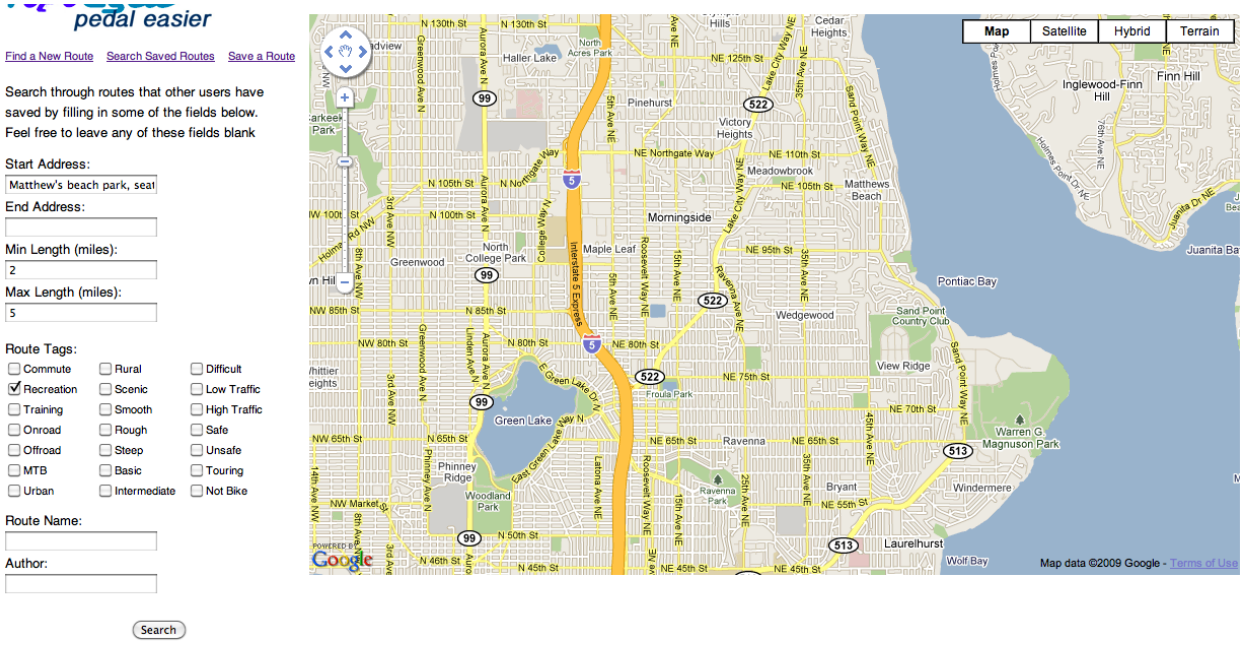
This is a single leg route finding the flattest route between the start and end point. This is the most typical use case of the site.



This is an expansion of the single leg route to include another leg. This is used when the user has an idea of the places they want to go, but not the specific route to get there.

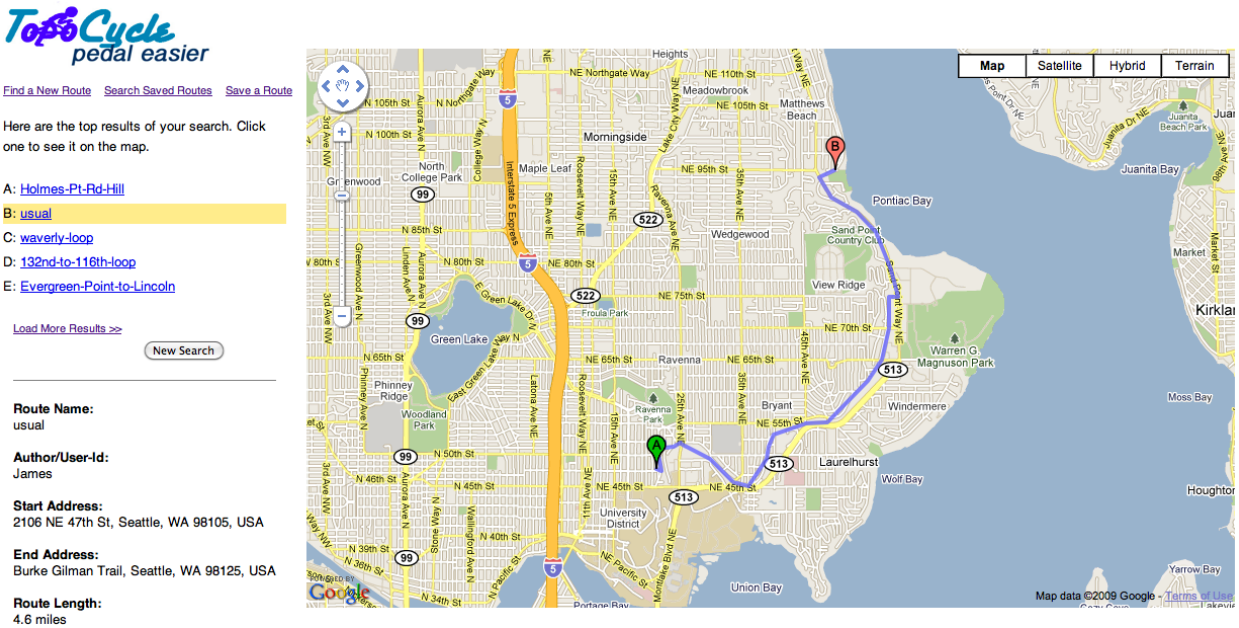


This is an example of a user saving a route for future use. This makes searching for the same route multiple times easier, because the user can now search by the route name or their author name.



This is a typical search for saved routes.

This is used when a user has a general idea of where they want to ride, the distance they want, and/or some characteristics of the route they want, but they don't have a specific route in mind. The user can also specify a route name or user name if they have a specific route in mind.

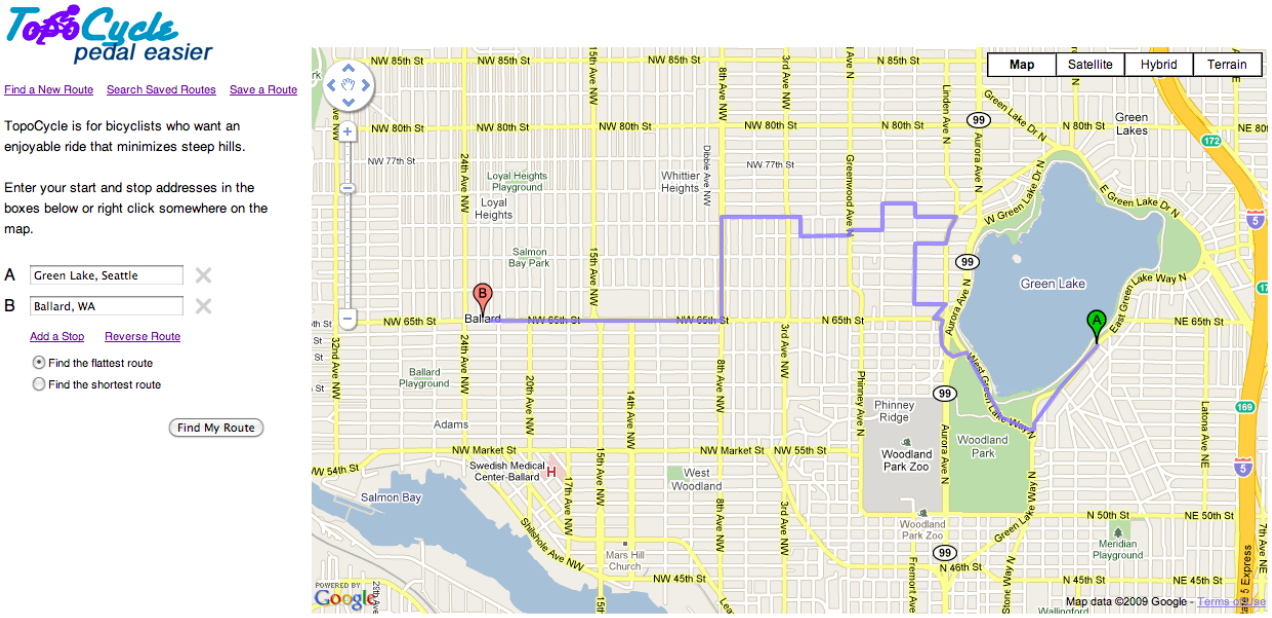


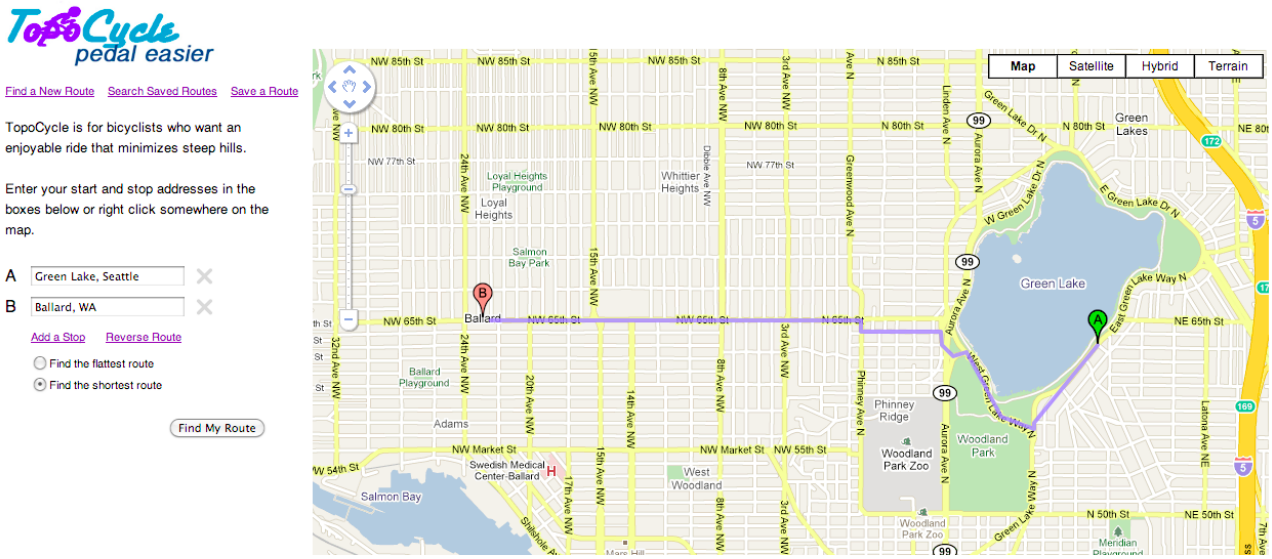
This is the results page for the above search. This is used to browse through the search results.

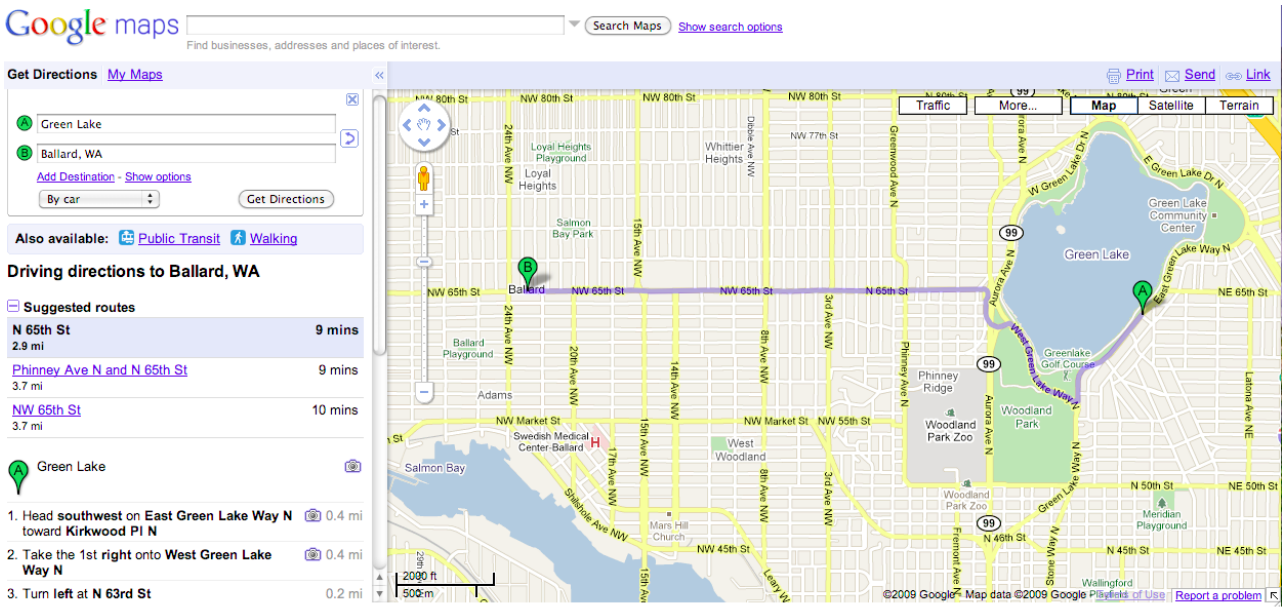
4 - Evaluation

The system was tested in four ways: comparing our custom route finding algorithm to the pgRouting algorithm and the Google Maps driving directions, testing the accuracy of the “search for saved routes” functionality, going out and riding the routes, and user testing.

To compare our route finding algorithm against existing algorithms we used the “shortest path” functionality of our algorithm (elevation cost set to zero). When compared against pgRouting our algorithm returned very similar results, though our results were calculated much faster: pgRouting took approximately 12 seconds to complete, on average, while our algorithm took approximately 4. Furthermore, pgRouting would occasionally return routes that made small loops or doubled back. When compared against Google Maps’ driving directions our routes were similar, but not nearly as similar as to pgRouting. One reason for this discrepancy is that pgRouting and our algorithm both use the same data. The other reason is that Google does not always return the shortest route, but rather the “best” route. This means that Google Maps is more likely to return a route that takes a major road and makes less turns than our route would. An example of this can be seen in the following picture, where the routes in order are the flattest according to us, shortest according to us, and best according to Google:

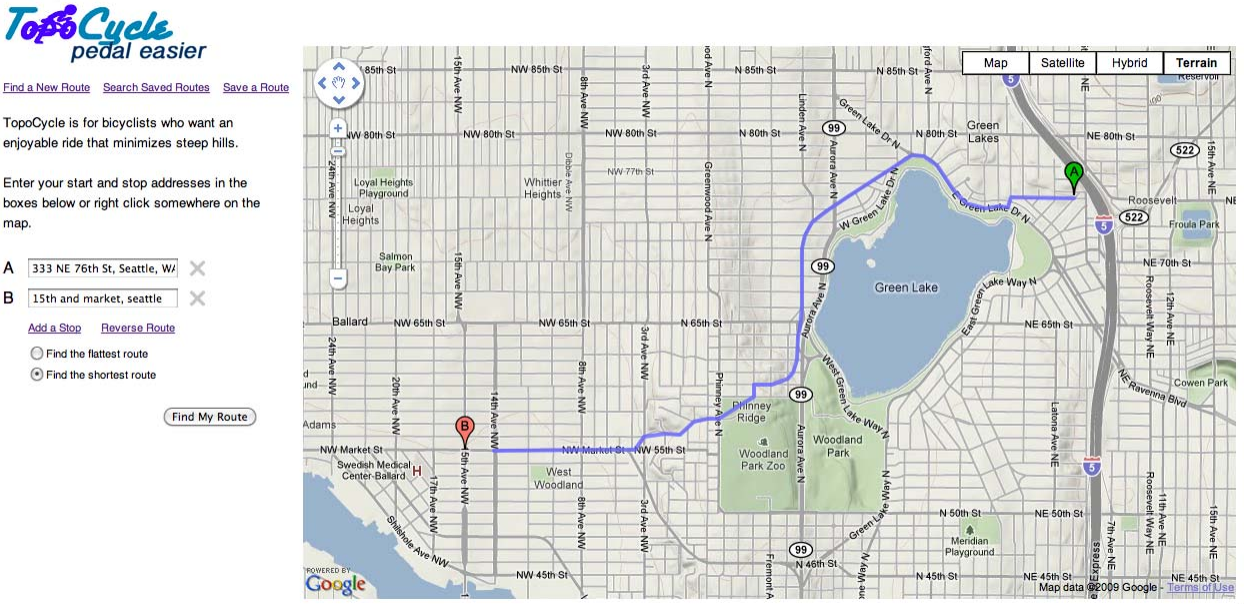


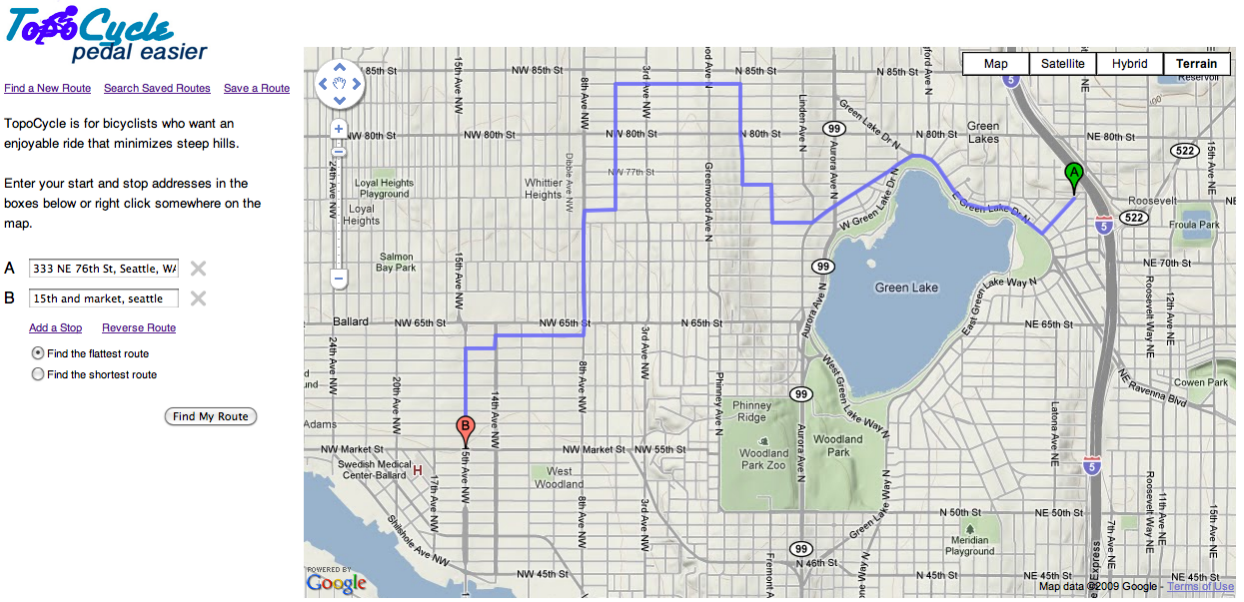




Testing the “search saved results” algorithm was done individually for searching by location, searching by route length, searching by tags, searching by route name, and searching by author. To test searching by location, we entered a specific start address and checked that the first result was close to the start point, and that each subsequent result was farther away. This was repeated with the end point and with both specified, where the total distance from the start and end points was the measure. To test searching by tags we entered a specific list of tags and verified that all returned routes contained the specified tags. This same process was used for testing search by route length, route author, and route name, where route name should only return one result. All of these tests were completely successful.

The first test of the validity of our flattest route finding algorithm was to go out and ride the routes ourselves, both in cars and on bicycles. To effectively test this we chose flat routes whose shortest-route versions went up steep hills and compared the two versions. These tests produced positive results, with the flat routes effectively avoiding the steep hills found in the shortest path routes. However, some of the returned routes added a significant distance on top of the length of the shortest route. An example of this is included below, with the first picture as the shortest route and the second as the flattest. This example effectively avoids a long, steep hill, but makes the route 50% longer.





User testing was performed in two ways. First, we asked six bikers to enter start and end addresses for routes that they already use and know well, and compare these to the routes our algorithm returned. This produced generally positive results with all six bikers reporting that the routes our algorithm returned generally matched the routes they would take, with minor discrepancies. The main source of these differences was similar to our difference with the Google Maps driving directions: our algorithm would return routes with lots of turns and no preference for major streets over alleys and back ways. This issue was reported by four of the bikers tested. The other source of differences was a complaint that the algorithm weighted elevation change too high, creating routes that added too much distance with respect to the amount of effort saved by avoiding hills. This imbalance was reported by three of the bikers tested. However, in both cases the tested bikers did not rate these discrepancies as unacceptable and reported that they would prefer our route finding algorithm over Google Maps driving directions when searching for bike routes. It should be noted that in these tests the users exclusively found the "flattest route" and did not use the "shortest route" option.

User testing was also used to test the user interface. The six bikers referenced above, along with five other people, were asked to use the site to perform the following tasks and report on their experience: find a single leg route, find a multiple leg route, alter a route they just searched, save a route to the database, search for a route in the database using geographic constraints, and search for the route in the database that they had saved. Furthermore, the users were not given instructions on how to use the site, but asked to figure out how to perform the tasks only using the instructions in the site itself. Of the eleven tested, ten reported that the site was easy to use overall and that they would use the site given the need. The most common compliments for the site were the ability to click on the map to add points, the continuity between right clicking to add points and entering addresses in the text boxes, and the ease of changing rates that they had searched by deleting points with the "x" buttons next to the boxes. There were also common complaints which included the need to search for a route again before removing points would update the map, the lack of turn by turn directions, and the inability to drag points. The general consensus among the users tested was that the correlation to the Google Maps interface was helpful, but that they would like to see more of the features that Google Maps offers. It should be noted that all of the users tested knew members of our team personally and we were often present during testing. This may have skewed the results to be artificially more positive.

5 – Surprises, Lessons

One of the earlier surprises we ran into was the poor documentation on pgRouting's website. The examples for setting up pgRouting and all the steps required to run its A* function are all outdated. While some example code didn't work at all, others lacked associating explanation for us to understand the code at all. Since we are using the TIGER/Line dataset, the data was also different from the example dataset in the tutorial, which complicated things even more. PgRouting was also not very popularly used, so finding answers through Google was difficult. The second surprise with pgRouting was its A* function's speed. Even when the correct columns were indexed, the speed was still slow. It was also very difficult to debug since it's a SQL function. If we were to start the project differently, we would avoid

pgRouting in the first place since most of the work spent on it was not even used in the final version of the program.

Other than that, we learned that it's useful to build your own projects on top of other code. We used the Google Maps API and pgRouting to start, and while pgRouting didn't end up in the final cut of the product, those two codebases still gave us a very good start that we wouldn't have gotten without using other code to begin our project. Additionally, it's good to keep in mind that at some point the software made by others may not fit perfectly for the role I'm fitting, and I'll have to write my own.

6 - Future Work

One of the biggest limitations that the service faced was that it was hindered by the bounding box problem: for routes that need to go around obstacles, the pathfinder did not always cover a large enough area to find a route between the start and end points and around the obstacle. In the future, we would upgrade the service so that the size of the bounding box in which the service looks for a path would be increased several times before concluding that no solution exists. Another potential future change is that users should be able to submit routes from outside of King County. Currently users can only submit routes that have been found on the site, which limits the routes to King County. There are many ways to expand this, one of which is to expand the dataset to include more data so that our service can search a wider area. Another solution is to allow users to submit valid longitude and latitude data that was acquired independently of our search. A big and ambitious project that solves this problem would be to develop an application for users to collect and submit GPS data using a mobile device. Although the project has met the goals we set in the project proposal, many changes large and small could be implemented to improve the service.

7 - Conclusion

Even though our service suffers from a few limitations, we deem our project a success, as it delivers on all of the points that we set out to do. Namely, the service can find flat routes adequately; most of the test runs of the routes discovered showed them to be better than the alternatives. The service also successfully supports a library of routes submitted by other individuals which can be queried in a variety of ways.

Appendix A: Work division

Everyone contributed writing to the paper.

Jeff

- Coded the html, css and javascript for the pages.
- Wrote the code which uses the Google Maps API.
- Designed the user interface for the site.
- Performed most of the testing, both user and module.

James

- Crawled Bikely.com for data
- Inserted the seed data from Bikely into our database
- Created the search retrieved routes and submit route features
- Helped debug some javascript

Daniel

- Wrote the custom A* search and all other Java
- Located, imported, and managed the elevation data for all lines described by the TIGER/LINE shape files
- Edited and compiled the paper, and added connective tissue
- Helped to debug javascript

Bo

- Obtaining and converting the TIGER/Line data into database tables.
- Setting up pgRouting, creating the topology in pgRouting, and making A* function work on our dataset.
- Creating the PHP script that ties the pgRouting route finding algorithm with the Google Map API front end.
- Eliminating Highways and Waterlines in the TIGER/Line data.
- Designing the test cases and user study.
- Conducting tests and user study.

Appendix B: External code

We used code from the following sources:

- Google Maps
    - For geocoding
    - For the map API
- pgRouting
    - To convert TIGER/Line data to SQL records
    - As the initial A* search
- U.S. Geological Survey Web Services
    - To collect elevation data for our TIGER/Line coordinates

We also used data from the following sources:

- U.S. Census Bureau
    - For TIGER/Line data
- U.S. Geological Survey
    - For elevation data
- Bikely.com
    - For user-submitted routes

Appendix C: Readme

To use the path finding algorithm you can use the site as it is when it first loads, or click the “Find a New Route” link. To add stops to your route enter addresses in the text boxes on the left or right click on the map. You can have up to 26 total stops by clicking the “add stop” link or right clicking more. If you with to reverse the route click the reverse route link below the text boxes. If you wish to execute our route finding algorithm, click the “Find My Route” button or press enter while in a text box. To switch between finding the flattest routes and the shortest routes change the value of the radio button.

To save a route click the “Save a Route” link on the top under the logo. From here the route that was already entered in the text boxes will still be there, and the controls can be used the same as above to find a new one. Enter a route name and author (these are required), and tags and comments if desired. To preview the route click the “Preview Route” button at the bottom. This has identical functionality to the “Find My Route” button. To save the route in our database click the “Save Route” button.

To search for saved routes in the database click the “Search Saved Routes” link at the top under the logo. Enter your search criteria in the text boxes and check boxes and click the search button. The results page should pop up. Click on one of the listed routes on the left to see it on the map and see its stats below. To view the next 5 results click the “Load More Results” link. Once this has been clicked you can click the “Previous Results” link to view the previous 5 results. To perform a new search click the “New Search” button or the “Search Saved Routes” Link at the top.