Gross Summary of Versions 3 and 4 and the Website

# Version 3

This version was primarily optimizations to JavaScript mainly to reduce computation time and the amount of elevation data collected.

# Version 3.1

## Summary:

This version reduced computation time in the intersections function group, by reducing the number of for loops and function calls. Furthermore, a method for removing dead-end intersections was introduced, which decreased function calls further. Additionally, a method for detecting the directions of roads was created by adding extra directions data. Finally, an attempt was made to use the Elevation API instead of the Service, because of the decreased time between requests and the theoretical increase in queries at a time.

## Journal:

I downloaded Notepad++ 2.7.5 on 3/24/15 after looking some on the internet. I configured it so that it would automatically save changes.

I also brainstormed ways to improve the efficiency of the algorithm. It comes in three ways: Use of Elevation API and some interpolation, reversing the directions of the queries to generate one-way roads and also change it so that intersections can only connect forwards, not backwards also, and streamlining the Branch algorithm. I will conduct research into how to extract the information from the Geocoding URL without violating CORS. As for interpolation, I had the idea that it could be done, since I have to simultaneously include provisions for the query generating results for all pts with the same coordinate; it may or may not be more efficient. For the intersections/Branch algorithm improvements, I came up with the idea of lopping off intersections that only have one connection, and merging intersections into one long path if they only have two connections. This will drastically cut down on computing time, and therefore decrease the amount of time needed to complete the calculations. Also, I had the idea of making intermediate significant points, so that the Branch optimization process is split into two steps. Significant intersections will have the most fuel efficient route found between it and the other nearby major intersections, and then those will ultimately have the algorithm applied to them. It will sacrifice some accuracy, but this is a heuristic change, and will likely be necessary for the algorithm to speed up.

3/24/15: 1 hour

3/25/15: 45 min

3/26/15: 45 min

I got a new editor, which is called Notepad++

3/27/15: 1.5 hours

3/28/15: 1 hour

3/31/15: 2 hours

I had the idea that it would be least complicated to find *only* the intersections that connected without data, because a) it would take more computing power to resort the data twice and b) be more complicated to code and. This way that I found, I find all the intersections from the original series data, and build lists of connections. Then, I strip intersections altogether that have only 1 or 2 connections, and rebuild the series data from the remaining intersections. Then, normal computation carries on.

This method appears to work, but does not noticeably lower amount of time required. Added an intersection limit.

I added a while loop to continually remove forbidden intersections until none remained. I also discovered that the connections arrays did not reset after each time, so they had artificially high numbers of intersections, nor did they count the same connection index only once. This was fixed, and the result was that only two intersections (start and end) remained. This was actually caused by an error in logic. The error was corrected, and the original basic while loop was replaced. The result is seen in output\_02.png. That does not represent all the intersections.

I timed the amount of time it takes to run parts of the program:

* Intersections: generation, adding signals, compiling data: 3193 ms
* Branches: calculation of: 13 ms
* Optimal: sorting through branches, finding the best route: 25 ms

*Sample: 6936 Millbridge Road to 330 Knollwood Street*

After some modifications (removing functions defined in loops and removing all traces of points having once been reversed) and the adding of backwards connections in addition to forwards to not remove quite so many intersections:

* Intersections: 3230 ms
  + Intersection\_gen(): 33 ms
  + Add\_signals(): 2346 ms
  + Intersection\_build(): 851 ms

I then made all functions inside the intersection\_build() function inside the Network function scope, and removed extra logic from the add\_signals() function, because the logic was not crucial to the performance of the algorithm. It essentially just double-checked things.

* Intersections: 929 ms
  + Intersection\_gen(): 39 ms
  + Add\_signals: 178 ms
  + Intersection\_build(): 713 ms

This is satisfactory. I now measured the branch algorithm for the same route, and it took more than 3 seconds to complete. I decided to shorten the maximum distance for the route to be 1.4 times the linear distance. The new time is in 900 ms range, which is also acceptable.

I also made all of the routes used in the directions service go both ways, and made the intersections algorithm search only forward, so it should only choose routes which the vehicle would go forward on, instead of illegal turns or wrong sides of the road.

4/1/15: 2.5 hours

Time: 9.5 hours

# Version 3.2

## Summary:

This version optimized elevation more by generating intersections first that would be included, and then generating elevation data that is in those intersections used. This led to a small net decrease in elevation data. Finally, a Python Bottle server was made to facilitate the Elevation API.

## Journal:

Next step was to optimize the elevation data gathering. First, I re-wrote the program so that the intersections were generated first, then elevation data retrieved for the relevant points. This took about 4 hours to do, because there were issues with finding problems in code, and figuring out the best way to reconfigure the program. Intersections\_build() was split up, because the first part gets relevant intersections, and the second part assigns pts to the paths. Since a copy of the point is made, it was not possible to assign copies and then change the data.

I also experimented with bottle.py beginning on April 8th. I wrote a short program to make a local server (which in theory could be hosted from a domain) to retrieve the elevation data from the Elevation API (since CORS restrictions do not apply in Python) and then pass it on to the Ajax request from script.js. This took about 3 hours, with experimenting and then needing to find and add the appropriate response headers (since initially the ‘no “Access-Control-Allow-Origin’ error message was present). I used the Bottle Documentation as a source to fix this problem on April 12, 2015.

To decrease the number of queries used, I looked up the format for the Encoded Polyline Algorithm. I tried to make a parser for that in Python, but I had problems with rendering negatives in binary. I later discovered a built-in function under google.maps.geometry.encoding.encodePath() that accomplished this. I substituted this format in lieu of “lat,lng|lat,lng|…”, but there were errors.

Next step was to optimize the elevation data gathering. First, I re-wrote the program so that the intersections were generated first, then elevation data retrieved for the relevant points. This took about 4 hours to do, because there were issues with finding problems in code, and figuring out the best way to reconfigure the program. Intersections\_build() was split up, because the first part gets relevant intersections, and the second part assigns pts to the paths. Since a copy of the point is made, it was not possible t assign copies and then change the data.

I also experimented with bottle.py beginning on April 8th. I wrote a short program to make a local server (which in theory could be hosted from a domain) to retrieve the elevation data from the Elevation API (since CORS restrictions do not apply in Python) and then pass it on to the Ajax request from script.js. This took about 3 hours, with experimenting and then needing to find and add the appropriate response headers (since initially the ‘no “Access-Control-Allow-Origin’ error message was present). I used the Bottle Documentation as a source to fix this problem on April 12, 2015.

To decrease the number of queries used, I looked up the format for the Encoded Polyline Algorithm. I tried to make a parser for that in Python, but I had problems with rendering negatives in binary. I later discovered a built-in function under google.maps.geometry.encoding.encodePath() that accomplished this. I substituted this format in lieu of “lat,lng|lat,lng|…”, but there were errors.

# Version 4

This version involved the transfer of the algorithm from JavaScript to Python, and the corresponding changes to connect the Python to HTML using PHP and file saving with Ajax. Also, optimizations were made to the algorithm that now gives it roughly a 50 mile range. Finally, a website was created and hosted on devtano.com.

# Version 4.1

## Summary:

This version included preliminary translation into Python, as well as a number of innovations from version 3. On a structural level, the Python module was organized into sub-modules encompassing general methods, data classes, query classes, IO, and miscellaneous functions, as well as external modules such as Six and Polyline. The program makes use of tuples instead of objects and specific data structures for entities like Pts and Intersections, and was designed in such a way as to utilize the iterative abilities of Python.

The innovations included a greater degree of Object Oriented Programming; gathering elevation data only used in a relatively small number of reasonable routes, instead of every point reference; a separate module (added during development) to house query classes; using road names gathered in the Directions API instead of geocoding; and drastic speed increases, because Python is more efficient than JavaScript in general, and because of the data structures used in the program.

# Version 4.2

## Summary:

Added input/output interfacing, so that the application can be called from PHP (popen) or command line, and write real-time console data to HTML and output final instructions to HTML and JSON (like Google Directions API).

Interpolation was finally added, which utilized a method of using the *connections*, not intersections, which were used in the preliminary list of reasonable routes. This decreased the amount of elevation data needed by roughly 70%, when removed connections and a (roughly) 45% decrease in total elevation is considered.

Increased speed in key areas: assigning stoplights to points by chunking and iterating through smaller datasets; finding relevant intersections to get elevation data for by using sets instead of indexing lists; and other small tweaks.

# Website

## Summary:

Initial pages placed by Akshat, along with some CSS styling stuff, from a template. Daniel made edits to this. Ali entered content. Collin gathered data for the drop-down menu in the input section of the main application page. Daniel re-formatted the CSS for the website to make it more readable and easy to edit. Ali and Daniel wrote a new version of the End-User Documentation. Initially, the pages were in HTML, and had copies of the same headers. They were changed by Daniel to PHP, and they loaded a header and footer from separate master files to increase compatibility and make editing a less strenuous process. Daniel wrote the main application page, with the JavaScript, from scratch. Daniel also spent a lot of time looking for a way to connect HTML to Python, and ended up finding and using PHP.

All file transfers were made to the remote server over FTP. Daniel kept a mirror of the appropriate directory tree on his flash drive.