# Version 4.2 Optimizations

I ran a test of the algorithm for a 40 mile distance (through IDLE) (test20/console.html), and the algorithm took minutes to run. There was also an error that read:

Traceback (most recent call last):

File "D:\TSA\devtano.com\software\eco\python\main.py", line 449, in <module>

route.getInstructions(vehicle)

File "D:\TSA\devtano.com\software\eco\python\classes.py", line 80, in getInstructions

vehicle

File "D:\TSA\devtano.com\software\eco\python\classes.py", line 39, in \_\_init\_\_

self.string = 'Continue straight onto ' + path[0].roadname + ' and continue for ' + util.mileFormat(meters=self.distance)

TypeError: cannot concatenate 'str' and 'NoneType' objects

I added in time recording to measure which parts of the algorithm most needed improvement. I began with “6936 Millbridge Road, Clemmons, NC” and “330 Knollwood Street, Winston-Salem, NC”, which is the standard testing route I use (~8 miles). Results as found in test 28/console.html. There were only two functions in the algorithm that took over 0.1 seconds to execute, and those were the functions for assigning road data to pts (and making the points) and the function for iterating through the overpass stoplights and assigning them to pts. This route (visually) takes around 20-22 seconds to run in the program (including queries).

In query.Overpasser.overpass, I split the if statement that decided if the point was in bounds into two if statements, and the time went from 0.158 seconds to 0.120 seconds. The increased cyclomatic complexity was probably offset by making considerably fewer calculations.

I ran tests in IDLE comparing map, for loop comprehensions and for loops with appends. The code was:

**def** **usingMap(**coords**):**

t0 **=** time**.**time**()**

output **=** map**(lambda** a**:** Pt**(**LatLng**(**a**[**1**][**0**],**a**[**1**][**1**]),** a**[**0**],** 35**,** 'stuff'**),** enumerate**(**coords**))**

**return** time**.**time**()-**t0

**def** **usingForEnumerate(**coords**):**

t0 **=** time**.**time**()**

output **=** **[]**

output\_append **=** output**.**append

**for** index**,** **(**lat**,** lng**)** **in** enumerate**(**coords**):**

output\_append**(**Pt**(**LatLng**(**lat**,**lng**),** index**,** 35**,** 'stuff'**))**

**return** time**.**time**()-**t0

**def** **usingForLCE(**coords**):**

t0 **=** time**.**time**()**

output **=** **[**Pt**(**LatLng**(**lat**,**lng**),** index**,** 35**,** 'stuff'**)** **for** index**,** **(**lat**,** lng**)** **in** enumerate**(**coords**)]**

**return** time**.**time**()-**t0

The average results were (after 20 consecutive runs for each):

* Using Map: 0.114
* Using For Loop List Comprehension: 0.113
* Using For Loop: 0.355

I ran the long route again with the timers in place (“6936 Millbridge Road, Clemmons, NC” to “Greensboro, NC”, about 40 miles [64 km]). The biggest one by far was Interpolations, which took 1 minute 51 seconds to complete. This is probably because there 1730 routes, and each one had about 18 intersections. Each intersection has two “in” operations on a list of connections eventually with (probably) a couple hundred items. I can reduce this by using a set instead of a list and organizing the intersection pairs alphabetically (since order does not matter and that will reduce the “in” operations from 2 to 1).

The changes I ended up making were to make elevations\_out and interpolations\_out into sets, since there was a problem with intersection dictionary references if the order was mixed up in any way. I also made pairs a set, and did not worry about having duplicates of reversed intersections. The two aforementioned sets were there to handle that problem, and also obviate the need for costly util.uniquify function calls at the end (since it would have to sort through about 3,000 elements, meaning 9,000,000 comparisons or so). The time for the short route went down from test29 at 0.045 to 0.005 at test33. Also, ValidReferences() was taken out of the script, since it is now obsolete.

On the long route, the initial 111 second time was reduced to 0.095 seconds in test36. However, there was a 400 error returned by Google Elevation API. This did not happen with the same code to the short route.

Another problem is the amount of time it takes to process the Overpass data. On the long route (test34), it took 13.3 seconds for the stoplights to be matched up. This was reduced to 10.2 seconds in test36. In summary, I changed it so that once a close pt was found, the two adjacent points are tested. The closest is chosen for the stoplight, and a break is made.

I made a test of using abs and -a < b-c < a. The test was made using the following functions:

**def** **absval(**a**):**

x **=** **[**random**.**random**()** **for** \_ **in** range**(**a**)]**

t0 **=** time**.**time**()**

y **=** **[**i **for** i **in** x **if** 0.05 **>** abs**(**i**-**0.5**)]**

**return** time**.**time**()-**t0

**def** **comparisons(**a**):**

x **=** **[**random**.**random**()** **for** \_ **in** range**(**a**)]**

t0 **=** time**.**time**()**

y **=** **[**i **for** i **in** x **if** 0.05 **>** 0.5**-**i **>** **-**0.05**]**

**return** time**.**time**()-**t0

* Absolute Value: 0.0021 seconds
* Comparisons: 0.01965 seconds

This was implemented, but made no significant difference.

I had been using the original set of intersections, which in this example, would be 24,917. The combined elev\_refs and inteprol\_refs are 11,375 long, which is less than half. In test42, the time for Overpass matching was 4.44 seconds.

I had a final idea to split up the references into a dictionary of chunks based on coordinates. This is accomplished by using a round function to determine the 0.1 x 0.1 rectangle the coordinate falls in, and add it to the correct dictionary element. This narrows down the possible references to a manageable number (anywhere from near-zero to about 1500 in the long route). It took 0.076 seconds to make the chunks, and 0.127 seconds to assign Overpass data. This is finally satisfactory.

Finally, there is still the problem with getting invalid responses from Google Maps Elevation API. This was solved by reducing the query length from 300 points to 200 points, as per the suggestion of a user on Stack Overflow.