Version 4.3

**Summary:**

This version introduces a new method for dealing with connections, which is contained inside network.py. Essentially, it drastically reduces computation time by making the energy requirement calculations one phase instead of two phases, one of which being extremely long, iterative and inefficient.

**Notes:**

On 6/25/15, our team had an email interview with a professional software developer, Mehrdad Niknami, at Microsoft. This is the main body of advice we received:

Actually, I'm not saying the data is inaccurate—the accuracy of the data isn't my concern. When I say "optimal path", I mean the correct answer with respect to the data you already have.

My concern was regarding the algorithm, not the data. What I was saying was that even if the data was perfectly accurate, the fact that the costs are not additive means that—depending on how you take into account each factor—the problem itself might be very "hard" (in a mathematically precise sense) and thus your algorithm will (in general) return roads that are "far" from the optimal solution, even though in many practical cases it may do "well enough" (I don't know).

If you're more interested in the details—here's a college-level description that might help your friend. It's **way** beyond high school level, so I don't really expect either of you to understand much of it, but it seems like it might interest him so I'll mention it anyway. :)

My understanding is that your algorithm tries to find a bunch of paths (say 100 of them) based on some heuristics that are hopefully "good enough" in practice, and then calculates the cost of each path and picks the best one.

If I understand this correctly, the only way this approach could return the optimal path for non-additive costs is if it tries an exponential number of paths in the worst case (perhaps O(3^|V|) for typical graphs) which basically means that there are reasonably-sized maps for which it might take > billions of years for it to give you the answer. This should be exponential even if you limit the path sizes to 2x the direct distance.

If that's not the case (i.e. if you're not actually looking at all the paths that are ≤ 2x the direct distance), it's probably just (semi-)guessing a bunch of essentially random paths and picking the best one it finds—in which case there's no reason to believe the paths it returns are any closer to the optimal path than what (say) Google Maps would give you, although perhaps typically they are better.

I think, however, that if you reason about the problem more carefully—it takes a fair amount of college-level math and CS background to understand the importance of this—you may notice that your criteria satisfy the Markov property: basically, this means that the cost of adding a new road to the path depends on the last road in the path (which is used to enter it), but not on any road before that.\* (It is possible to come up with criteria that don't satisfy this, and in that case the problem will likely become NP-hard.) I believe your friend noticed this, but it's likely he didn't notice why this actually works to your advantage. What happens is that you can essentially transform the graph by combining every pair\* of roads into a new meta-road. The details are pretty tricky to get right (I haven't tried working through all of the details), but this would blow up the graph size by a factor of O(|E|d-1), where d is the number of roads that are joined into a meta-road (so joining every pair of roads would result in the graph becoming larger by a O(|E|) factor). This is slow for a very large map (since O(|E|d) running time is bad for d > 1), but it should actually guarantee that you will find the optimal path, and it's (infinitely!) better than an exponential time algorithm (which yours seems to be, if it were to guarantee that you would find the optimal path).

\*Actually, "pairs" is a bit of a lie... if your roads are too short, the speed & acceleration on a much earlier road might still be relevant, so you might have to take into account the last triple or quadruple. Nevertheless, the Markov property still holds, and you can determine how many pairs you need to join based on the road lengths and the 0-to-60 acceleration of the car under consideration.

As a completely unrelated side note, note that assuming a deceleration at every stoplight is pretty pessimistic and unlikely to give you a good estimate of fuel efficiency... for that, you would need to know a lot more, including the speed distributions of cars. Gathering that information is itself a massive undertaking, but now, even if you have that information, the problem becomes extremely difficult to solve in a reasonable amount of time (a multi-year project for a company, university, or government agency).

Hope this was useful for your learning! :) Let me know if you have any questions.

Daniel:

I did not understand what he was describing with meta-roads and so, and how it will always produce the optimal route, and how it will be more efficient. However, I did understand in some sense about the significance of Markov, but probably not the intended sense. The idea that the comment about Markov property working to my advantage gave me was to redo the data structure relating to energy calculation. Immediately previous to this email exchange, I had noticed that on a 65 mile route it took 7.5 seconds to calculate the final force values for only 30 routes.

The original algorithm had energy calculations happening between nodes, with a second phase for the most promising routes over the nodes, including all the other points in the whole path. This was very inefficient, especially for long routes. In these particular 30 routes, each one had about 2,500 points in it. Since each point required a whole series of conditional statements and math functions, this took a long time to complete. The reason I couldn’t just do the deep calculation for each connection was because the problem had the Markov property, where the energy requirement changed based on the state of adjacent connections; since the connections were variable, it was not possible to do a one-time calculation.

The new idea called for connections between nodes to have an extension afterwards, which is variable depending on which intersection it connects to next. Thus, the deep energy requirements can be calculated exactly once for each point (actually one, two or three times, due to algorithm structuring) instead of up to 30 times.

This idea was implemented by making new Intersection, Connection, Micropath and Longpath classes. Intersection was pretty much identical; Connection started with a path (which was the equivalent of self in the previous version, since the type was list) to act as a starting point used for distance calculations and so on, but was actually a dictionary type that would later have the keys filled by the intersection keys following self.end. This was done after elevation and stoplights were calculated, since it required the copying of objects. The path that occupied the Connections dictionary were of the Micropath type, which was a type of list, and was essentially a placeholder for a list that had the ability to calculate deep energy requirements similarly to the old Path class. Afterwards, the energy cost for each path extension in the Connections of routes was summed for each route generated using the recursive function, similarly to old OptimalEnergyInitial. The top entries were uniquified using util.filterRoutes, and finally the number of requested routes was compiled into a list similar to the old Path, now called Longpath, and the instructions were generated.

This had enormous time benefits. In test97, the combined time of OptimalEnergyInitial and OptimalEnergyDetailed 7.612 seconds, the latter being 7.547 of those. In test128, on the same route, it took 1.343 seconds, but since the computer was running nearly half as slowly as it did in test97, it is probably closer to 0.7 or 0.8 seconds. This is a tremendous improvement.

Later on, I took the needed connections generated in interpolation, and used them to calculate only the connections I needed, instead of doing literally every connection.