SNOMOR

We Forgive Bashar for AFIT

aristide.cuny
elisey.balakhnichev
elvïn.foulon
maureen.rauss
mehdi.oueslati

June 2021

Contents

1 Defining the problem: Ice, Graphs, and Maple Syrup

We are tasked to optimize the de-icing of Montréal's roads. Since this is usually done through the means of specialized vehicles, we may tackle this problem by using graphs. More specifically, what we are trying to find is a way to determine the path of each unit of vehicles in such a way that they de-ice in an optimal manner. The definition of what is optimal is however somewhat vague, as it could cover a wide range of parameters and trade-offs such as time to de-ice versus cost of de-icing. In the end, we simply decided to solve the problem in a generic fashion.

2 The theory: Euler's Adventures in Wonderland

2.1 From maple syrup to spicy noodles: the Chinese Postman Problem

Since we are asked to de-ice the whole city, given an arbitrary number of units of vehicles, we must be able to compute a path for each vehicle such that every edge is visited. Moreover, we are obviously looking for the shortest of these paths. Given a single unit of vehicle, our problem definition is thus equivalent to that of the Chinese postman. The definition of that problem, according to (American) National Institute of Standards and Technology (NIST) is indeed to "Find a minimum length closed walk that traverses each edge at least once.".

2.2 Keeping It Simple, Stupid: starting with an easy problem

Following the previous definition, NIST also adds that "Finding an optimal solution in a graph with both directed and undirected edges is NP-complete.". As our lord and saviour Donald Knuth said, early optimization is the root of all sins, so let us start by keeping it simple and look at the case where we only have undirected edges. The problem is now to find the smallest path going through every edge of the graph at least once. That seems simpler, but the "smallest" phrasing may be somehow ambiguous, so let us explicit it. Ideally, we would like to go through every edge once and only once. How can we do so?

2.3 From spicy noodles to greasy meatballs: the Königsberg problem

The problem of finding a path that goes through every edge of a graph only once is one of the oldest in graph theory. As a matter of fact, it is called the Königsberg bridges problem and was solved by Leonhard Euler himself in the first half of the 18th century. In his honour, such a path was even named a Eulerian cycle. Unfortunately, a graph only has a Eulerian cycle if and only if every vertex of that graph has even degree, yet we can be sure that that is not the case for the graph of Montréal just because it is highly probable that there is at least one cul-de-sac or three-way in such a big city.

2.4 If you cannot make it, fake it: Eulerization

Our problem as detailed in 2.3 is thus insolvable. Too bad, but we can benefit from working abstractly and "hack" it through. How so? By "Eulerizing" the graph, that is tweaking the graph such that every one of its vertices is of even degree. Obviously, we would not do that by adding arbitrary non existing nodes, but rather by duplicating nodes of odd degree such that they become their own duplicate's neighbour, thus adding one new neighbour and making them even. We are allowed to do so because we are working on an undirected graph, so going to neighbouring node is semantically equivalent to crossing back the road in the other direction. This process runs in polynomial time regarding the number of vertices, so that is nice.

2.5 Adding road signs: directed edges

So far, we have an answer which makes wholesome paths on our Montréal maps, but units of vehicles obviously cannot always do U-turns and go back from whence they came, so working on undirected graphs is a bit of stretch. We hence need to update our model by adding directions to our graphs. What does that mean to our solution? Can it be kept? If the graph is Eulerian, yes of course. If it is not, then we need to Eulerize it once more. However, as we already established, we cannot do so by adding U-turns to unfitting vertices. Fortunately, there is still a way to tell if a directed graph has a Eulerian cycle by looking at its vertices: that is by checking that every vertex has an equal number of incoming and outgoing nodes. How do we fake it this time? By finding paths in-between nodes with more indegrees than outdegrees and those with more outdegrees than indegrees in such a way that every indegree excess is matched to every outdegree excess. This can be done if and only if the graph is strongly connected, which is a fair assumption for a city's road network.

2.6 Uh Oh! Two clouds in our model

2.6.1 The more the merrier: the k-Chinese Postman Problem

So far, we assumed that we were only looking for a single cycle as if we only had a single unit of vehicles to de-ice the whole city. This assumption is obviously false: we have many units of vehicles that can work in parallel. So, the task is not so much to find the perfect city pathway for one, but rather to divide the city up into smaller sectors with minimal overlaps in such a way that every unit of vehicles would de-ice its sector without impeding on another's. This problem is unfortunately NP-Complete.

2.6.2 From greasy meatballs to saucy hot-dogs: the New York City Street Sweeping Problem

Moreover, it is swell to have a perfect solution for both undirected graphs and directed graphs, but what happens if we combine them? What would happen if some edges only needed to be crossed once but could be passed by both ways? Well, it turns out that this problem is NP-Complete as well and that the Eulerization solution does not work here.

2.7 Thank you, next!

Arriving at this point, we were happy, mainly because of the insight and of the understanding of the problem that we developed. We also had a few Python scripts (*such as brrr.py in the shipped files*) that showed nice results. Nonetheless, we deemed the Eulerization useless and impractical, so we decided to stop thinking like Computer Scientists and to start behaving like ~~hackers~~ engineers.

3 The practice: Snap back to Réal-ity

3.1 Thee shalt not reinvent the wheel!

Now that we identified which theoretical problems we were facing, we no longer needed to think about solutions. We could find the work of people who had much more time than we do to solve the problems we were facing. We thus read a few papers online and decided to settle for the work of Kaj Holmberg from the Linköping Institute of Technology in Sweden. His paper solves the k-rural postman problem for urban snow removal. The rural postman problem is not exactly the New York City problem as it allows itself to ignore some edges as input, we can thus emulate a New York City problem to the best of our capacities.

3.2 Real Montréal.js

We made a standalone software to load OpenStreetMap (OSM) data into a graph and export graphs into JSON files so that they can be looked at on `https://geojson.io/`. We wrote this software in NodeJS because it was quicker than Python while still providing ergonomic manners of interacting with OSM.

3.3 Writing the heuristics: Crab or Snake?

We quickly decided to ditch Python, which we had used at every step so far to test things out. The main reason behind this deliberate choice is that moving to a lower-level language is free optimization for our heuristics. The only downside is the loss of programming comfort but considering how (un)fortunate we are to have survived "rushes", "piscines" and hellish projects, we came to the point where we did not deem it peculiarly difficult to write low level code. We thus considered that the advantage of having code that runs fast and uses less memory more valuable as it allowed us to locally test our software on the real Montréal graph. The question then was whether to pick C++ or Rust ~~or C or Assembly~~. Being lazy millennials who hate the thought of cleaning our own pointer garbage, we took the *Oh-so-2021!* choice of picking Rust. We manage to load the whole Montréal map data and run our heuristics in less than 30 minutes on a 4GHz processor.

3.4 Reconnecting everything

So far, we ended up having two pieces of software that do exactly what we want them to do but are completely disconnected from each other. We thus needed to make them smoothly communicate. In order to do so, we decided to write Input/Output standards. Those standards are simple extensions of the GeoJSON specification and can be found on the folder Spec as JSON file examples.

4 Our solution's limitations

4.1 Local minima hell

Our algorithm often reaches local minima which forces us to make multiple tries with different inputs to find better solutions.

4.2 Poor cost modelling

Our cost modelling only considers the weights of the graph and the initial cost for the k units of vehicles, but it does not look for an optimal k, and thus still heavily relies on user input. It also does not consider the costs of having snow blocking roads.

4.3 Imperfect city model

The data we load from OSM seems to be incomplete for our usage. Our software is capable of handling both roads and sidewalks, but OSM has barely any information on Montréal's sidewalks. We also found that there are multiple strongly connected components in our graph, which does not make much of sense because it is a city so you should be able to go from anywhere to anywhere else. This leads us to believe that our data is partially incomplete.

5 Conclusion

SNOMOR is a powerful yet imperfect collection of software that takes an Open Street Map file as an input, converts it to graph and then takes extra parameters to solve a weighted k-rural postman problem. This being said, it is still imperfect both from an algorithmic and a data perspective. Furthermore, as a software, it could be fine tuned more to fit even better to the actual cost optimization problem of de-icing.