Find the Sneaky Path

CS404 Project, Fall Semester 2016

Version: October 27, 2016.

Due: Saturday November 26, 2016 (or earlier)

Summary

Finding an optimal path in a network is not always the shortest path, shortest distance, or cheapest cost, but rather: most scenic, least traffic, get-away path, and so on. This project explores finding a *SneakyPath*.

Motivation

Commercial Airlines, roads, and computer and communication networks often model their routes as graphs. There are other examples in social networks and cloud computing, and so on. In such a network, nodes are the cities, or intersections, or server nodes (sources of demand, such as traffic or communication sites), and the links connect hem. If you want to go from one node to another (by plane, by car, or routing internet traffic or information packet), then you need a plan: source, destination, and a planned path in between. Let us explain this in terms of an road network between cities as the motivating application, but keep in mind that this project might have additional uses. When planning a car-trip, say from a to b, most people would want to find the shortest route, and there are several algorithms known that will help you find the shortest path. But this may not always be the criterion used. Some people may be interested in the least travelled road, to avoid as much congestion as possible. They want to avoid other cars as much as possible, they do not want to see other cars, they do not other cars to see them, or whatever. May be you are towing a 12-foot wide boat, and you do not want to inconvenience other drivers, so you look for the least travelled road. Even in computer networks this might be of interest: lowest probability of dropped packets. We will call such a path a *SneakyPath*.
You will be given a number of different input scenario's. Each scenario will have

1. The size n of the system (the total number of cities),
2. an adjacency matrix $\mathbf{E}$ with edge-weights. The $(i,j)^{\text{th}}$ entry $\mathbf{E}[i,j]$ represents the traveling time on the (direct) edge between the two specific nodes, i and j.

The matrix $\mathbf{E}$ is will allow you to compute the shortest path from a to b, and several algorithms could be used for this. The shortest path might take several links, and if you are the only one on the road, then this is the path you will take. However, you are not the only person/car on the road. You will also be given

3. a flow matrix $\mathbf{F}$, whose $(i,j)^{\text{th}}$ entry $\mathbf{F}[i,j]$ represents the number of (other) cars that travel from node i to node j every hour.

For the sake of argument, say that there are $\mathbf{F}[21,47] = 115$ cars traveling on the path from node v_{21} to node v_{47}, then there are 115 cars on each edge of that path. Suppose edge (v_{34}, v_{38}) is on the shortest path from node v_{21} to node v_{47}, then there are also 115 cars/hour on the edge (v_{34}, v_{38}) due to the flow $\mathbf{F}[21,47]$. But is very well possible that the edge (v_{34}, v_{38}) is also on the shortest path from node v_{17} to node v_{29}, and that $\mathbf{F}[17,29] = 75$. This means that there are at least 190 cars/hour on the edge (v_{34}, v_{38}). Thus for each edge (i,j), you can calculate the carried traffic load from all source-destination pairs on that edge. Put all this information in a new matrix $\mathbf{L}$, each entry representing the total load on the edge. Finally, you will be given

4. the starting node a and the terminal node b

And for this particular pair, you are asked to calculate (and print):

a) The *SneakyPath* from a to b, such that the total number of other cars on the road encountered is as small as possible,

b) the edge on this *SneakyPath* with the lowest number of other cars,

c) the edge on this *SneakyPath* with the highest number of other cars,

d) the average number of other cars on the *SneakyPath*, averaged over the number of links on the path.

This is however a course on algorithms, and you also have to report on the algorithms you used to actually find the information asked above. In particular,

e) The worst case time complexity for the algorithm(s) you used,

f) the measured CPU-usage of the programs when it solved each case,

g) the empirical validation that your programs have the correct asymptotic complexity.

Project expectations

You need to explore and expand the algorithms for finding the information as outlined above, and you should feel free to explore additional algorithms that you feel are appropriate. The goal is to find an efficient algorithm for the problem and to implement them efficiently. You can write separate algorithms/programs for the separate parts, or use the same one, whichever is most appropriate for the problem. You need to write a report in which you address the following issues:

1. Explain the reasoning for using the algorithms you implemented (as opposed to alternate choices), explain any supporting algorithms and data structures. Obvious potential choices are Dijkstra's SPA, and Floyd-Marshall SPA.
2. Explain the algorithm you used and implemented to actually generate the paths between nodes, so that the load per edge can be calculated. If there were two ideas you were working on, what are the pro's and con's? You may use secondary metrics, such as ease of programming, ease of software maintenance, portability, and so on.
3. Provide convincing arguments that your algorithms find a correct *SneakyPath* or demonstrate by counter example that it does not always find a *SneakyPath* and explain why this should be acceptable.
4. Present an analysis of your your algorithms for best-case and worst-case time complexities as well as space complexities. Make sure to mention the key-and-basic operation you are using.
5. Implement the algorithms that is best suited (or acceptably suited) for networks where n is bounded above by $1,000$. Best-suited means better performance.
6. Provide convincing arguments that your have implemented your algorithms correctly and efficiently with the appropriate data structure.
7. Perform a timing study of your implementation and show that this conforms with your (pre-implementation) time complexity study. In other words, measure the run times of your programs to experimentally verify your analytical findings. The least you can do is a plot (size n versus observed times).
8. For the cases that will be provided, present the *SneakyPath*.
9. Feel free to add test cases yourself that illustrate certain algorithmic or program features
10. Indicate whether or not you believe that there is a better (i.e. faster run-time) solution with an explanation on why you believe this to be the case or not to be the case.
11. As a last item, reflect back on this project and on the design and data structure decisions you made. How did you handle any unforeseen situations and any 'newly discovered' problems? Now that you have done the project, what have you learned? If you have the opportunity to do this project again, perhaps with additional requirements, what would you do the same, and what would you do differently? What are the lasting lessons you have learned? What are the lessons for future students?