COMPSCI-2XC3 — Lab 9

DUE : Sunday, March 27, 2022 11:59 PM EST

Submit the lab on Avenue. Note, Avenue must receive your lab by the due date. **Do not leave your submission to the last minute! Late submissions, even by seconds, will not be graded.** You have been warned.

This lab is worth (59/7)% of your final grade in the course. Read this document carefully and completely. **Whenever I ask you to discuss something, I implicitly mean to include that discussion in your lab report. Furthermore, when I ask you to run an experiment or evaluate the performance of a function/method include scatter plots where appropriate. Do not import external libraries, especially those containing data structures you are meant to implement!** Include all timing experiments in your `code.py` file.

Purpose

The goals of this lab are as follows:

1. Implement different shortest path algorithms
2. Understand the trade-offs of different approaches
3. Understand different shortest path problems
4. Identify when and why to use a specific algorithm

Submission

You will submit your lab via Avenue. You will submit your lab as three separate files:

- `code.py`
- `shortest_paths.py`
- report.pdf (or docx, etc.)

Only one member of your lab group will submit to Avenue – see the section below for more details on that. Your report .pdf will contain all information regarding your group members, i.e. name, students number, McMaster email, and enrolled lab section. This will be given on the

title page of the report. For the remainder of this document, read carefully to gauge what other material is required in the report. Your report should be professional and free from egregious grammar, spelling, and formatting errors. Moreover, all graphs/figures in your report should be professional and clear. You may lose grades if this is not the case. Organize the report in a such a way to make things easy for the TA to grade. You are not doing yourself any favors by making your report difficult to mark!

Testing

For the purposes of testing the performance of your algorithms, unless told otherwise test it on complete graphs with randomly assigned weights. That is, there is an edge between all nodes. Use the `create_random_connected_graph` function in `lab9.py` to help you.

Bellman-Ford Approximation [50%]

You have seen Dijkstra's algorithm, but as you know it does not work all the time. Specifically, Dijkstra's algorithm will not necessarily return the shortest path if the graph in question has negative edge weights. Bellman-Ford is another shortest path algorithm which does work with negative edges (but not negative cycles), however, its runtime is much higher than Dijkstra's. This stems from the fact that Bellman-Ford in essence brute forces relaxations of edges/nodes. Specifically, $V - 1$ times Bellman-Ford will iterate through all the edges of the graph, potentially updating the currently known shortest distant to each node.

In `lab9.py` you can find code for both Dijkstra's and the Bellman-Ford algorithm. You need to write a modified version of the Bellman-Ford algorithm. Your modification will take in a third parameter k – each node's currently known shortest distance will be updated at most k times. That is, the line:

```
dist[neighbour] = dist[node] + G.w(node, neighbour)
```

for any specific node should not be run more than k times. Name this function `bellman_ford_approx` in your `shortest_paths.py`. Run some experiments on your approximation compared to the original Bellman-Ford implementation. Specifically limit your experiments to graphs of size 100, which have edge weight randomly chosen between 1 and 1000. You will evaluate your approximation on two metrics:

1. Runtime
2. Total distance returned by the algorithm

By total distance I mean the sum of all the distances to each node from the distance dictionary. See `total_dist` for more information/clarification. In your experiments start with $k = 1$ and increase it one-by-one. Present relevant scatter plots for your two metrics. For this experiment do you recommend a specific value of k? Discuss the trade-off your approximation gives the user.

All Pairs Shortest Paths [50%]

Dijkstra's and the Bellman-Ford algorithm are both single-source shortest path algorithms. You give them a node v (a single source) and they return the shortest path from v to every other node. An all-pairs shortest path algorithm takes a graph as input and returns the shortest path from every node to every other node. This is usually returned as a matrix of values.

You need to write two all-pairs shortest path algorithms. One for positive edge weights and one for potentially negative edge weights. Do not reinvent the wheel here. Use Dijkstra and Bellman-Ford as a starting point for each algorithm. Name them `all_pairs_dijkstra()` and `all_pairs_bellman_ford()` in your `shortest_paths.py` file. Each method should return a matrix (as a nested list), where `matrix[i][j]` is the shortest path distance from node i, to node j. For all i it should be the case that `matrix[i][i] = 0`.

From 2C03 (and Wikipedia) you know Dijkstra has complexity $\Theta(E + V \log V)$, or $\Theta(V^2)$ if the graph is dense. Moreover, Bellman-Ford has complexity $\Theta(VE)$, or $\Theta(V^3)$ if the graph is dense. Knowing this, what would you conclude the complexity of your two algorithms to be for dense graphs? Explain your conclusion in your report. You do not need to verify this empirically.

In `lab9.py` you will find a `mystery()` function. It takes a graph as input. Do some reverse engineering. Try to figure out what exactly this function is accomplishing. You should explore the possibility of testing it on graphs with negative edge weights (create some small graphs manually for this). Determine the complexity of this function by running some experiments as well as inspecting the code. Given what this code does, is the complexity surprising? Why or why not?