CS/SE 2XC3 Lab 9 Report

Glotov, Oleg
L03, 400174037
glotovo@mcmaster.ca

Willson, Emma
L02, 400309856
willsone@mcmaster.ca

March 25, 2022

This report includes the main observations that we found in this week's lab, along with the analysis of our results.

1 Shortest Path

In this section, we discuss algorithms for finding the shortest path in directed weighted graphs, potentially with negative edge weights.

1.1 Bellman-Ford Approximation

In this implementation of the Bellman-Ford algorithm, there is an array of counters that keeps track of how many times the line `dist[neighbour] = dist[node] + G.w(node, neighbour)` is run on each node. When the count for a node surpasses the input value k, the algorithm stops updating the shortest distance to that node.
We tested our approximation of Bellman-Ford using connected graphs of size $n = 100$ (where n is the number of nodes) with edge weights randomly chosen between 1 and 1000. The testing method used was consistent with earlier labs. We used `create_random_connected_graph()` to generate a random graph given a number of nodes and maximum edge weight. We ran the original `bellman_ford()` and our `bellman_ford_approximation()` on each graph using k values ranging from 1 to 40. For each k value, each algorithm was tested three times on three different graphs. When testing, we recorded the runtime for each algorithm and the calculated total distance to every other node. Each data point is the average of these three trials.
We expect our approximation algorithm to perform better than the original Bellman-Ford algorithm for low values of k because k puts a ceiling on the number of times the distance to a node can be updated. This is one of the most expensive aspects of the algorithm, so capping it should allow it to terminate sooner. The performance of both algorithms is shown in the chart below.

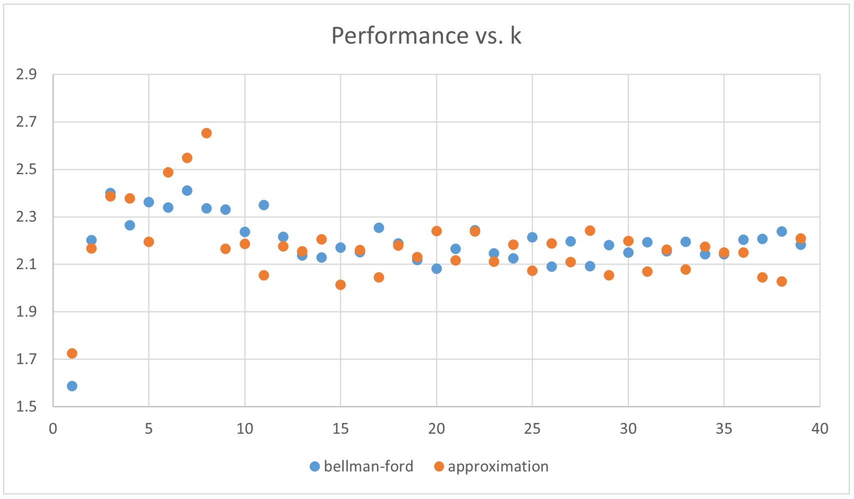

Figure 1: performance of bellman-ford and an approximation vs. k

The timing data we collected doesn't show any clear correlation between k and performance. In fact, some trials resulted in the approximation performing worse than the original algorithm, which seems erroneous. This may be due to the additional overhead in our approximation algorithm. That algorithm updates each element of an array of length n up to k times, which can be expensive as n becomes large. Our approximation generally seems to terminate slightly before the original algorithm does, but this improvement in performance is negligible compared to the imprecision of the data.
We expect the total distance to each node to become more accurate as k increases, with low k values resulting in the approximation overestimating the total distance. This is because `dist[neighbour] = dist[node] + G.w(node, neighbour)` is only run when the algorithm finds a shorter path to a node. If we restrict this process with k, the shortest distance to a specific node might not be found. The total distance to each node (or the minimum spanning tree) is shown in the chart below.

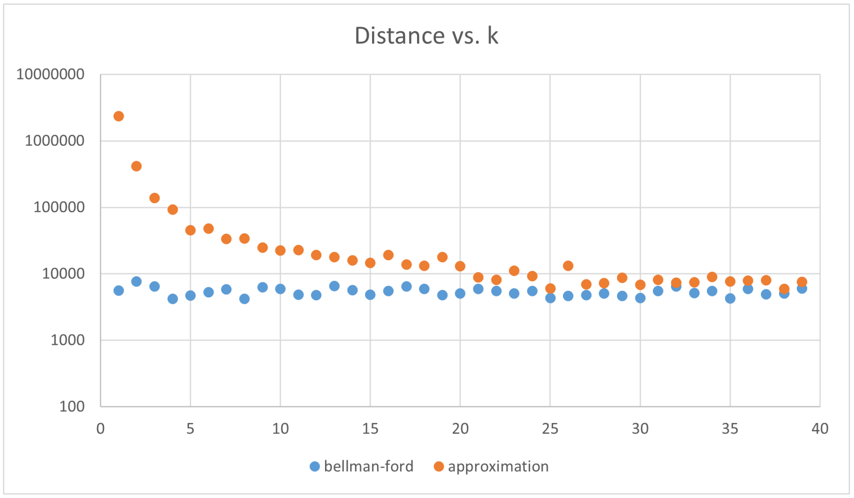

Figure 2: total distance calculated by bellman-ford and an approximation vs. k

Our approximated Bellman-Ford performed as expected. For low k values, the approximate total distance was much higher than the distance calculated by the original method. Around $k = 20$, the approximate and actual total distances converge. We would recommend a k value above 20 because that is when the error made by the approximated distance becomes reasonably small. Our timing data does not suggest that a specific value of k is better for performance.

1.2 All Pairs

Our implementations for `all_pairs_dijkstra()` and `all_pairs_bellman_ford()` use the same approach as the algorithms given in `lab9.py`. To make each algorithm find the shortest path from every node to every other node of a graph, we compute the array of distances from a source node to every other node for each node in the graph using `dijkstra()` and `bellman_ford()` respectively. Since Dijkstra has complexity $O(V^2)$ for dense graphs and Bellman-Ford has $O(V^3)$ for dense graphs, I would expect the all-pairs versions of these algorithms to have $O(V^3)$ and $O(V^4)$ complexities. This is because they consist of running Dijkstra and Bellman-Ford V times and $O(V) \times O(V^2) = O(V^3)$, $O(V) \times O(V^3) = O(V^4)$.

1.3 Mystery

The `mystery()` function in `lab9.py` uses a helper function `init_d()` that takes in a graph and initializes the shortest distance matrix. For each path from node i to node j, the function checks if these nodes are connected. If they are connected, then the matrix entry for `d[i][j]` is assigned the weight of the edge between them. Otherwise, it is assigned 999999. The `mystery()` function iterates through each node V^2 times. It checks if the recorded path from some node i to j is greater than the sum of the paths from i to another node k and k to j. If so, the shortest path distance is updated. After some testing, we concluded that `mystery()` does not work on graphs with negative edge weights. We ran some experiments on `mystery()` and `all_pairs_dijkstra()` and `all_pairs_bellman_ford()` to see how they perform. We used the same method as in the Bellman-Ford Approximation, except we chose to vary V, the number of nodes instead of k. We still tested these algorithms on connected graphs with positive edge weights. First, we verified that all three algorithms output the same path matrix when given the same graph. Then we tested the time each algorithm takes to generate the path matrix when given the same graph. We conducted our tests on graphs with V ranging from 1 to 40. The results are shown in the chart below.

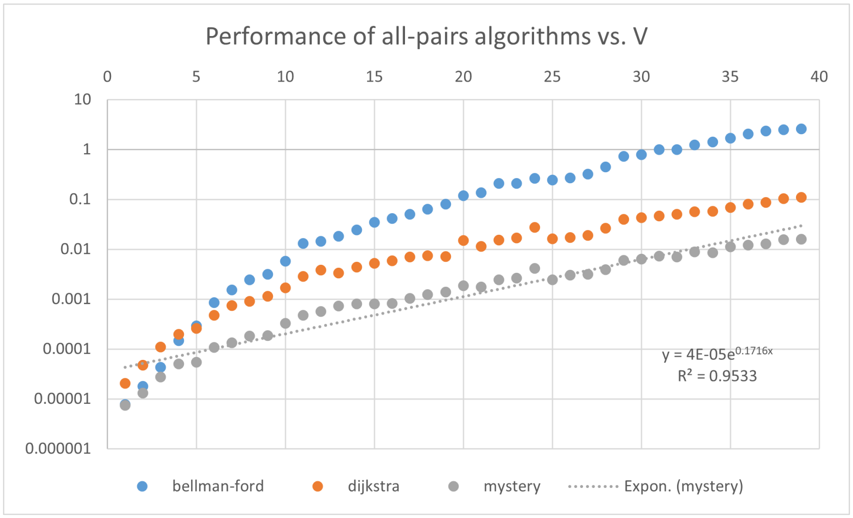

Figure 3: performance of mystery function vs. V

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