COMPSCI 2XB3 L04 Lab Report #9

Mahad Aziz - azizm17 - 400250379 - azizm17@mcmaster.ca
Talha Amjad - amjadt1 - 400203592 - amjadt1@mcmaster.ca
Logan Brown - brownl33 - 400263889 - brownl33@mcmaster.ca
McMaster University

COMPSCI/SFWRENG 2XB3: Binding Theory To Practice

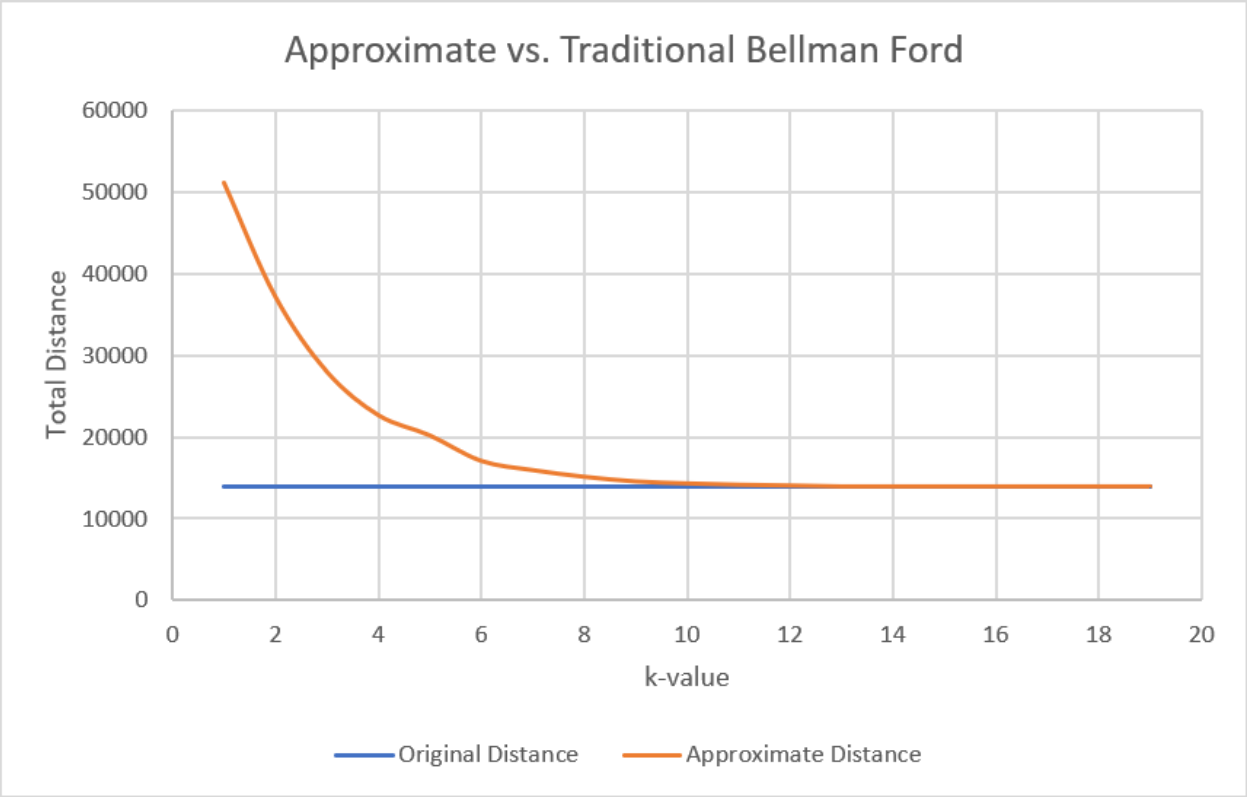
Dr. Vincent Maccio

TA: Amir Afzali

March 29, 2021

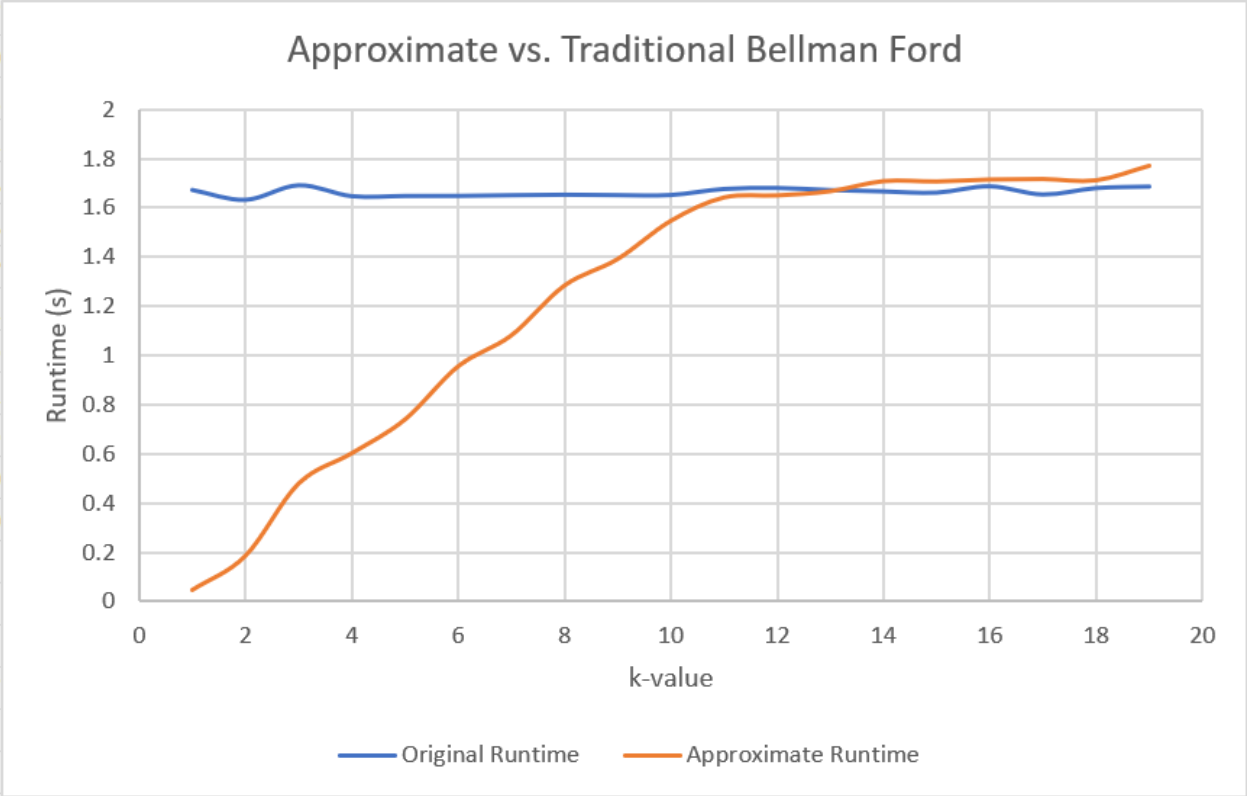
Bellman-Ford Approximation:

For the approximate variation of the Bellman-Ford function, we decided to use a dictionary to keep track of the number of times the distance of a node was changed. We also made sure that the distance of a specific node is updated up to a k number of times. For testing the runtime and the total distance of the Bellman-Ford approximation algorithm we created a random graph that has a size of 100 with an edge weight randomly chosen between 1 and 1000 and we also began with a k-value of 1 and incremented that until a k-value of 20. Using this random graph we calculate both the runtime and the total distance of the original Bellman-Ford function and the approximate version as well by taking the average of 5 runs.



This graph shows the comparison between the total distance of both variations of the Bellman-Ford with respect to k. The original algorithm is not based on the k-value so the total

distance remains constant as the k-value increases. However, for the approximate algorithm, we can see that the total distance does indeed change drastically as the k-value increases. The total distance is largest when the k-value is small and it decreases until it gets to a k-value of 11 where both algorithms have the same total distance after that for increasing k-values.



For the graph that compares the runtime of the original and approximate Bellman-Ford algorithms with respect to k. The original algorithm is not based on the k-value so the runtime remains constant as the k-value increments. On the other hand, the approximate algorithm is based on the k-value so the runtime does indeed change when it comes to an increasing k-value. This makes sense because in the approximate function we only update the distance either a k number of times or when the distance could potentially increase. From the graph we can see that for k-values up to 13, the runtime of the approximate is lower than the runtime for the original Bellman-Ford which makes sense because we set a limit to the number of times the distance will

get updated. For k-values greater than 13, the approximate function has more runtime because it does more calculations than the original and the condition where the distance has the potential to get updated is met before the value of k is met.

From both of these graphs, we can conclude that as k values approach infinity for the Bellman-Ford approximate algorithm they produce the same results as the original Bellman-Ford algorithm. The noticeable trade-off that occurs when using the Bellman-Ford approximate algorithm for small k values is that you achieve faster runtimes but larger total distances. On the other hand, for larger k-values, you achieve smaller total distances but with slower runtimes similar to the original Bellman-Ford algorithm. From our experiment, we noticed that using a k-value of 10 in the approximate Bellman-Ford algorithm has a faster runtime compared to the original algorithm but it gives a total distance value that is almost identical to the total distance from the original algorithm.

All Pairs Shortest Paths:

To implement the two all pairs algorithms, we used either Dijkstra or the Bellman-Ford algorithm on each node in the graph. We then appended the list of values we got from each run of either algorithm into another list, effectively making a nested matrix of values. The all-pairs algorithm that uses Dijkstra can be used for graphs with all positively weighted edges and the one that uses Bellman-Ford can be used for graphs containing potentially negative edge weights. To determine the complexity of these algorithms for dense graphs, we can use the complexities of Dijkstra's algorithm and Bellman-Ford on their own which are $\Theta(V2)$ and $\Theta(V3)$ respectively. Since we use each algorithm on every vertex of the graph, it's simple to reason that our complexity becomes $\Theta(V*V2) = \Theta(V3)$ and $\Theta(V*V3) = \Theta(V4)$ respectively.

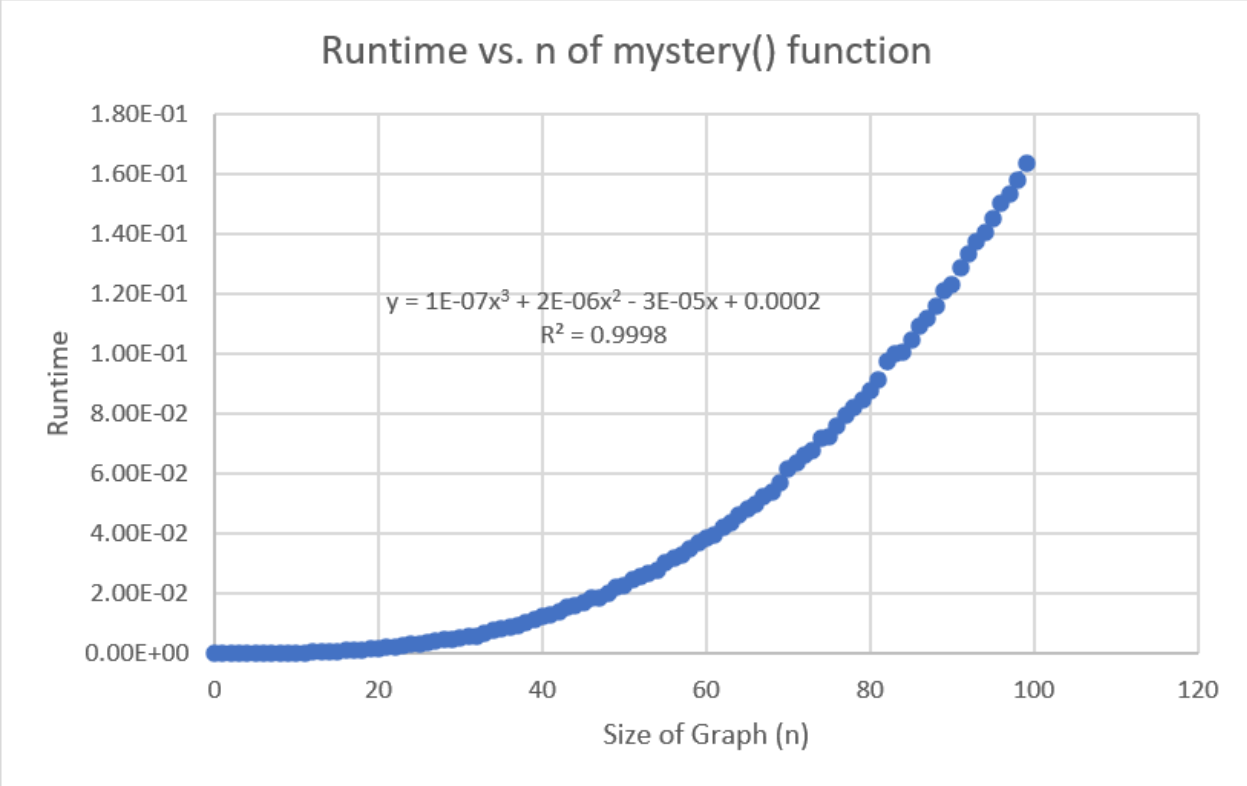
Mystery Algorithm

To figure out what the mystery function does we must first look at what its helper function init_d is doing. It first creates a matrix of size nxn where n is the number of nodes in the given graph, and it fills every index with 999999. Then, if two nodes i and j are connected, the weight of the edge between them is inserted into the index (i, j) in our matrix. All diagonal indices are filled with zeros. The result is a matrix that gives us another representation of the graph, where nodes i and j have an edge between them with weight d[i][j]. If there's a weight of 999999 between two nodes, then there's essentially an "infinite" distance between them and they're disconnected. The diagonal of the matrix is filled with zeros because all nodes have a weight of zero to themselves. Using our new understanding that this matrix is essentially the graph, we can see that the three for loops in mystery() are traversing every path from i to j with one potential in-between node k for every node k in the graph. Then in the inner, if statement, if the path between i to k to k to j is shorter than the path from i to j, we can replace the weight of

the path from i to j with this new path. Essentially, we're again finding the shortest path from each node to every other node.

When testing mystery with graphs with negative edge weights it functioned correctly every time and gave the same results as Bellman-Ford. As expected, it also didn't work with negative edge cycles. In the below analysis we determined that the mystery algorithm has complexity Θ(V3) which is a straight upgrade over the all-pairs algorithm using Bellman-Ford which we determined to be Θ(V4). In terms of finding all pairs of a graph which may contain negative edge weights, this seems to be the superior algorithm.

To understand the complexity of the mystery algorithm, we created a function that creates a random complete graph. This test function then runs the mystery algorithm on random graphs of increasing size n.



Taking a look at the algorithm, we see that 3 loops add up to the time complexity of O(V^3). Furthermore, the variable declarations and if statements are constant time complexity so we end up with the O(V^3) complexity for the algorithm.

Furthermore, a log-log graph was created to prove if the time complexity is O(V^3). As you can see in the graph below, the slope from the log-log graph shows us a slope of approximately 2.8 which indicates that it is close enough to be the time complexity of O(V^3)