SE 2XB3 Group 4 Report 9

Huang, Kehao
400235182
huangk53@mcmaster.ca
L01

Jiao, Anhao
400251837
jiaoa3@mcmaster.ca
L01

Ye, Xunzhou
400268576
yex33@mcmaster.ca
L01

30 March 2021

1 Bellman-Ford Approximation

Two versions of Bellman-Ford approximation are implemented. One is a direct modification of the original Bellman-Ford implementation with a frequency map which keeps track of the number of relaxation performed on each node. The other version is our attempt to exploit the frequency map with the hope to exit the function and return when all nodes are relaxed exactly k times. Figure 1 and 2 are the results of experiments carried out to evaluate the implementations on two metrics. The optimized version and the original approximation operates in the same way in terms of how the distances are computed. Therefore, Figure 2 only includes one total distance for different approximation versions.

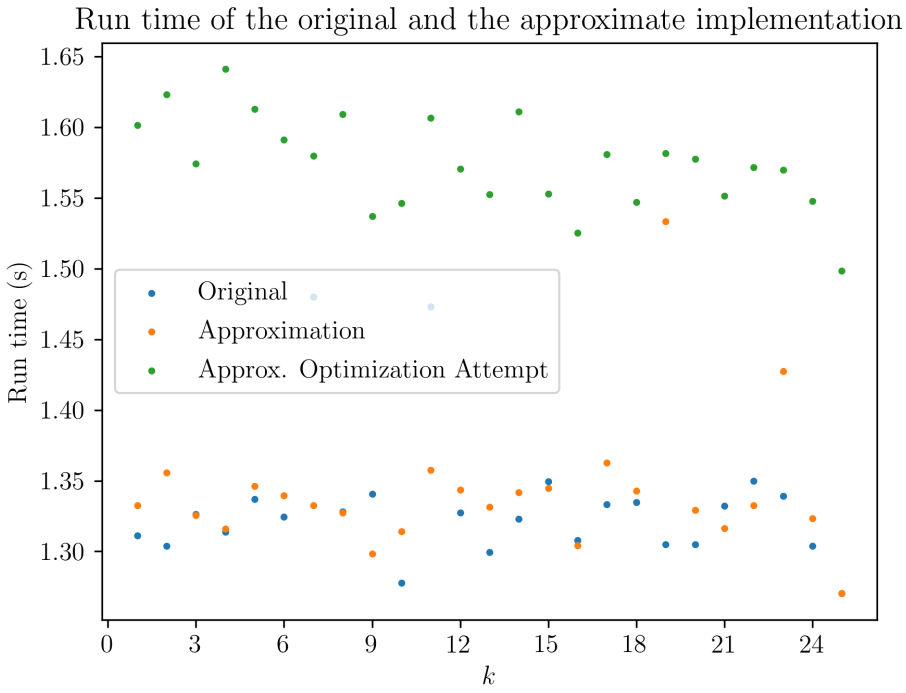

Figure 1: Run time comparison

Surprisingly, our attempt for optimization is observably slower than the other, while the difference in run time between the other two is insignificant. One possible reason is that the extra `if` statement and `set` operations introduced into the optimization attempt bring overheads to the function. The time saved from breaking the loops is shorter than the time wasted on processing the overheads.

Observing that in Figure 2, the approximated total distance merged with the accurate total distance computed by the original Bellman-Ford implementation starting on $k = 9$, the approximation implementation strictly requires less relaxations in general. However, as indicated in Figure 1, the performance gain from the avoided relaxations is not enough to make a consistent empirical difference. Considering that approximations are not 100% accurate, we propose that the original Bellman-Ford implementation ensures the accuracy of the result and is overall the best among the three.

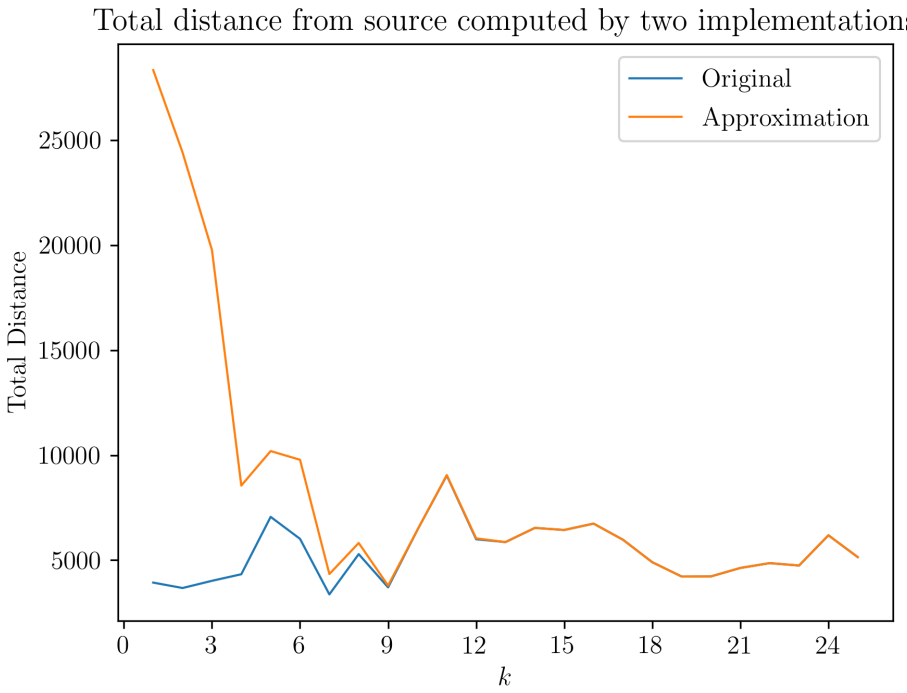

Figure 2: Total distance comparison

2 All Pairs Shortest Paths

Our implementations of `all_pairs_dijkstra` and `all_pairs_bellman_ford` simply return the list of the results of `dijkstra` and `bellman_ford` called on every vertex. By knowing the complexity of Dijkstra's and Bellman-Ford's algorithms are $\Theta(V^2)$ and $\Theta(V^3)$, respectively, for dense graphs, calling the function for V vertices results in the complexities of $\Theta(V^3)$ for Dijkstra's and $\Theta(V^4)$ for Bellman-Ford's.

As for the `mystery` function, it is an implementation of Floyd's algorithm. It finds the shortest paths between all pairs of the vertices. By inspecting the code, the three nested `for` loops over the number of vertices V confirm its time complexity of $\Theta(V^3)$. After running some small experiments with the implementation, we found that it can handle graphs with negative edge weights. Given what the code does, the complexity is not surprising because processing all possible pairs of vertices, or in another word, filling the $V \times V$ matrix itself is a $\Theta(V^2)$ task. The overall $\Theta(V^3)$ complexity can be roughly perceived as taking $\Theta(V)$ time to process $\Theta(V^2)$ pairs of vertices. In fact, Floyd's algorithm can be understood as for any pair of vertices i and j, try every other node k to see if the path $i \to k, k \to j$ can improve the current path estimate $i \to j$.