For the Bellman-Ford algorithm, I chose to implement a private “Node” class that uses a “destinationToWeightMap” HashMap within each node. Since each node is connected to another destination node via a weighted edge, the key for each HashMap was the integer for the destination node, and the corresponding value for the HashMap was the weighted edge. This structure seemed to be more intuitive than using an ArrayList or LinkedList since the nodes connected by edges are already given in the input file. Each node contains a HashMap to associate its destination with an edge, AND another HashMap is also used again to structure all the nodes together into one graph. I implemented a “nodeMap” HashMap that takes a node’s integer as the key and maps it to that node in the graph as its value. I used a HashMap for the main execution of the algorithm because it is very efficient to grab a node via its integer key, and then access any of its attributes (one of which is the destinationToWeightMap HashMap). The same approach was used for the Floyd-Warshall algorithm, were the destinationToWeight HashMap was later used for an adjacency array for each node to help initialize the main structure of the matrix. HashMap operations are O(1) on average and O(n) for worst case or for iteration. However, when it comes to searching a shortest path within the Bellman-Ford algorithm, my program should have a runtime of O(V*E), where V and E are the number of vertices and edges. The algorithm does V-1 iterations, but since I’ve included in my program the ability to finish sooner when no changes have been make during a single iteration, it is possible for the best case to be O(E).The Floyd-Warshall algorithm should take an average runtime of O(V^3) since updating the data of each node’s indexes of a V x V matrix is done by a 3-level nested for loop. I’ve already accounted for the graphs to be doubly linked, thus the process of updating each index should be faster since the weight from node A to node B should be the same vice versa. Also, all the diagonal positions in the matrix will all be set to zero and will not have to be checked again since a node connected to itself is zero by default. After doing some research and given that my program uses a HashMap data structure, I would assume that my program uses an average memory space of 32 * SIZE + 4 * CAPACITY number of bytes. Since a pointer to the next entry is contained within a HashMap along with a key and a value, each entry is considered to typically occupy 32 bytes. Thus, my program with size being the number of vertices must be multiplied by 32 for entry storage alone. In addition, 4 times the amount that the map can hold will be used for the array of entries. Even though a HashMap is very space efficient compared to other data structures such as LinkedList or TreeMaps, a HashMap can still be improved. A THashMap can replace an implementation for a HashMap to save memory. It contains one array for keys, and another for values, so it uses an average memory space of 8 * CAPACITY number of bytes for storage instead. The ease of implementation with HashMaps that was first created when making Dijktra’s algorithm helped to grab information from the input text file and populate the Node data quickly. For this reason, I simply reused the idea of a private “Node” class and all its attributes to be able to establish the initialization of each node without having to give it a second thought. The main difference between Assignment 2 and the algorithms implemented in this Assignment 3, are the different algorithm methods itself.

Online Sources Used:

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

… and lecture slides