Student: William Hellems-Moody
Date: 12-1-23
Assignment: Distribution Project - Final Assignment [Updated 27 November]

Documentation.txt Content:

Data Representation

Regarding the data representation for this program, we were instructed to use the provided Supplier.java and Transporter.java classes to represent the suppliers and transporters. Focusing more on the Supplier class. This class is designed to model the nodes in the graph, which includes fields such as the name, demand, inventory, and maxCapacity. The purpose of this class was to provide a clear representation of manufacturers and distributors that each have their own storage and demand needs. The transporter class was provided to represent the transporters used along the routes/paths and essentially represent the edges between two suppliers. It also has its required fields, including source, destination, capacity, cost, and allocation. The role of this class is to outline and provide a detailed overview of the logistical routes/paths within our graph representation.

I also decided to implement a graph class, which assisted in aiding me in representing and managing the network suppliers (nodes) and transporters (edges). Essentially, the suppliers represent the points along the router, and the transporters represent the edges along the directed graph, which contain their allotted costs and capacities.

This distribution network uses a graph-based structure, which uses HashMaps to represent my adjacency list to store the edges (transporters) and each node (suppliers) to represent a distribution network properly. To model the network, nodes and edges were used throughout the system to properly represent each supplier and transporter, characterized by key attributes like demand, maximum storage capacity, inventory, name, storage cost, and surplus. Shifting the focus to how transporters are represented. These are represented in this system as the edges of the graph that connect one supplier to another. The edges in the graph are also equipped with their own set of required attributes such as name, from, to, cost per unit, allocation (necessary for implementing most of the program), and max capacity.

This design aimed to encapsulate the entire or, at the very least, most of the flow network within the graph object, representing a directed graph. This was necessary as using a directed graph significantly helped me map and visualize the flow of scooters (goods) from one supplier to another. Aside from this, this system, particularly in the implementation of the two primary methods allocateForDemand and CheapestPath, primarily makes use of data structures like HashMaps, Sets, Collections, and Priority Queues for tasks like processing nodes and checking the minimum cost of an edge along a path, backtracking, implementing breadth-first search, updating and allocating transporters along a certain path, tracking

visited suppliers, and determining the lowest cost of a path, and allocating a collection of transporters along a max flow path with the incorporation of a superSource to a superSink. The combination and use cases for each approach were vital in separating the logic needed to satisfy the overall requirements of this lab (i.e., implementing cheapestPath and allocateForDemand).

Algorithm Descriptions (Brief Explanation)

Implementing the cheapestPath Method (using modified DijkstraÕs algorithm):

The cheapest path method was implemented using a modified version of DijkstraÕs algorithm. It was formulated to find the least-cost path between the suppliers represented in the graph that ultimately helps establish the lower transportation cost within the directed graph. First, the algorithm starts by assigning each supplierÕs initial cost (i.e., the distance) as the maximum possible value available (i.e., Integer.MAX_VALUE), excluding the starting point, which is set to zero. The graph is then developed through the addition of edges in the form of transporters being added to it. Next, the algorithm iteratively updates the cost of reaching adjacent suppliers by considering the current transporterÕs cost per unit, which is then updated until the destination node is reached. From here, the method backtracks from the destination supplier to the source and then uses the paths stored to determine the cheapest path between suppliers and transporters. This was accomplished through using a HashMap named minCost, which is used to constantly track the minimum cost to reach each supplier, and another set named unvisitedSuppliers to ensure that the algorithm can track which suppliers have been visited (as the name implies), ultimately leading to the cheapest path among suppliers being found.

Implementing the allocateForDemand method (using Ford-Fulkerson Algorithm)

I implemented the allocateForDemand method using the Ford-Fulkerson algorithm, which was appropriate for determining the max flow in a given flow network. To implement this method and use this algorithm, I began creating a graph representing my distribution flow network of suppliers and transporters (nodes and edges). Taking this approach allowed me to focus on the core functionality of the allocateForDemand method, which was finding augmenting paths and adjusting the allocation of transporters within the network to satisfy demand where possible. To assist in implementing the Ford-Fulkerson algorithm, I incorporated various private helper methods within the program to help maintain the readability and maintainability of my code and the logic used for allocating transporters along the max flow network.

The methods used include findAugmentingPath, updateResidual, and redistribSurp. The findAugmentingPath method represents a modified breadth-first search, which iterates through the graph to identify paths where additional flow can ideally be sent. After locating an augmenting

path, the updateResidual method is called to adjust the flows along this path, considering the capacity constraints and current flow within the network. Next, the goal of the private helper redistribSurp is to ensure and check if there is any remaining capacity in the network and redistribute surplus inventory. The process of updating the flow network of goods between suppliers and transporters halts when no more augmented paths are discovered in the network and when the entire demand of 1010 (for the test case provided by the instructor) is either satisfied or the max flow to satisfy a set of suppliers within the network is achieved. Afterward, the algorithm is set to return a collection of transporters allocated along that path, whose allocated amounts tally up and equal the influx of demand within the network, ensuring that the inflow matches the networkÕs outflow. This ensures that the algorithm either fully satisfies the total demand or maximizes the possible throughput given the constraints set along the distribution network and by each supplier and transporter.

Assessment of Correctness (Brief Explanation)

This program implements two primary algorithms: cheapestPath and allocateForDemand. The cheapestPath method uses the modified Dijkstra algorithm to determine the least cost path between any two suppliers within the graph representation of my flow network, considering factors like the cost and capacity constraints outlined for each transporter. This method returns the appropriate collection of transporters along the desired cheapest path in the network. This was tested against fabricated data (graph provided by instructor) for each supplier and transporter within the graph, ensuring the algorithm consistently finds the least costly path for any given scenario. Otherwise, an empty collection of transporters is provided, representing that no cheapest path was found.

Regarding the allocateForDemand method implemented for this program, this method uses Ford-FulkersonÕs to determine the maximum network flow to satisfy the suppliersÕ demand where possible. This method was tested alongside the cheapestPath, which also utilized the same fabricated data. Using the same data allowed me to assess the algorithmÕs capacity to ascertain the optimal flow within the network. The testing showcased the methodÕs ability to allocate transporters to establish a maximum from a given source to specified destinations. Likewise, I realize that further testing is needed, in particular with more diverse and complex data sets, which may expose additional insights and areas of improvement.

Overall, after testing for the provided scenario, it was concluded that both functions operate as intended, where one determines the cheapest path by using the basic structure of DijkstraÕs and the other implements Ford-FulkersonÕs algorithm for determining the max flow along a network through its allocation of transporters within the network.

Reflection (Less than 300 words as requested during lecture)

Reflecting on the program SimpleAllocator, this experience provided me with real-world experience using algorithms to manage a network of

suppliers and transporters to find the least cost-effective paths and maximum flow of a network. Likewise, optimizing and allocating resources gave me a deeper understanding of graph theory and how it can be used to solve complex logistical issues in the supply chain. In addition to learning more about flowgraphs in the logistical sense, I also found that the use of Ford-FulkersonÕs for optimizing the allocation of scooters along a path requires careful consideration of the method being applied and ensuring that the relationship between nodes was maintained and applied appropriately. Given this is my personal reflection, I found that the allocateForDemand method was very tedious and challenging, especially when dealing with complex systems such as a supply chain network that deals with product distribution.

A significant takeaway for me was the benefit of implementing a modified version of standard algorithms like Ford-Fulkerson and DijkstraÕs algorithm solely for the purpose of finding the cheapest path of a distribution network and allocating transporters to achieve a maximum flow in a given network. I found this algorithmic adaptation exercise quite valuable for gaining real-world experience on how algorithms learned in a class setting can be modified and used beyond their initial or standard implementation. Overall, this project underscored the significance of precise, rigorous testing and debugging and having an in-depth understanding of the algorithms you are using to achieve the desired overall outcome. Finally, working through this project, I feel that I have gained extremely valuable skills, regardless of the correctness of my program, that will undoubtedly benefit my future endeavors in computer science and software development beyond this course.