**Solution**

**Tyler Bradshaw**

**Data Structures and Algorithms II**

**F. Algorithm Evaluation and Alternatives**

The delivery simulation system's core strength lies in using the nearest-neighbor strategy. This heuristic method efficiently selects the next closest package destination, aiming to minimize the distance traveled. Such an approach often offers a good balance between computational efficiency and route optimization. Additionally, the system's ability to accommodate real-time modifications, such as the dynamic address update for package 9, showcases its adaptability to changing conditions.

The algorithm adheres to the provided requirements: it employs a hash table for package data storage, ensures trucks travel 18 miles per hour while carrying a maximum of 16 packages, and completes all deliveries within the day. All three trucks travel a combined 134.8 miles. All packages delayed in flight and package 9 are on truck three. Truck three does not depart until one of the drivers from truck one or truck two returns, and it is after 9:05. Package 9 receives an address update at 10:20 and reroutes all remaining packages on truck three after the correct address is received.

There are alternative methods that could enhance route optimization. For instance, a "Divide and Conquer" strategy could be employed to segment the delivery region into smaller zones, optimizing deliveries within each zone before amalgamating the results for a comprehensive route. On the other hand, "Dijkstra's Algorithm" could be used to ascertain the shortest path from the hub to all destinations, ensuring an optimal route based on the shortest paths between nodes. Both approaches could also meet all requirements.

The "Divide and Conquer" strategy fundamentally differs from the "Nearest-Neighbor" heuristic used in the solution. In "Divide and Conquer," the problem is recursively divided into smaller subproblems. These subproblems are solved independently, combining their solutions for a comprehensive answer. For the delivery system, this could mean segmenting the delivery region into smaller zones, optimizing deliveries within each zone, and then amalgamating the results for an overall route. In contrast, the "Nearest-Neighbor" approach used in the solution focuses on sequentially selecting the closest next destination, potentially leading to suboptimal paths over the entire set of destinations.

Dijkstra's algorithm is designed to find the shortest path between a starting node and all other nodes in a weighted graph. If applied to the delivery system, Dijkstra's algorithm would determine the shortest path from the hub to every other delivery point, ensuring optimal routes based on distance. This contrasts with the "Nearest-Neighbor" heuristic in the solution, which, at each step, chooses the next closest destination without considering the overall shortest route for all packages.

**G. Potential Enhancements**

If revisiting this project, several refinements come to mind. First, integrating a dynamic routing system would be invaluable, offering the capacity to adjust routes based on variables like traffic or unforeseen events, ensuring more accurate and efficient deliveries. Second, employing a priority queue for package deliveries could be pivotal, prioritizing deliveries based on their deadlines and ensuring timeliness. Lastly, enhancing the system with a GUI map visualization utilizing the provided downtown map would elevate the user experience and provide real-time visual insights into the delivery process.

**H. Data Structure Evaluation and Alternatives**

The custom hash table employed in the proposed solution has been designed to meet the specific demands of the scenario. It boasts an impressive average time complexity of *O(1)* for insertions and lookups. This efficiency stems from the underpinning hashing mechanism, which ensures swift access to any given entry when combined with apt collision resolution strategies. The primary allure of such a table lies in its ability to perform key-value pair operations with consistent speed, regardless of the size of the data set. The hash table was used to utilize each required data component appropriately.

An alternative data structure is a balanced binary search tree, like the AVL tree. While a binary search tree, in its basic form, can degenerate to a linked list in worst-case scenarios, leading to *O(n)* time complexity, self-balancing trees like AVL ensure that the tree remains relatively balanced. This balance ensures that operations such as insertions, deletions, and lookups are completed in *O(logn)* time complexity, a slight decrement from the ideal *O(1)* but still efficient. What sets balanced binary search trees apart, perhaps their most salient feature, is their ability to maintain order among the elements. This ordered structure can be invaluable for tasks requiring sorted data, range queries, or in-order traversal without additional sorting operations.

Python's built-in dictionary is another worthy contender for efficient data structures. At its core, it manifests as a hash table optimized for general-purpose applications. With an average time complexity of *O*(1) for most operations, it parallels the performance of the custom hash table. However, while they might seem functionally equivalent, the distinguishing factor is the level of control one can exercise. The custom hash table, by being tailor-made, offers nuanced control over aspects like the hashing mechanism and collision resolution strategies. Such flexibility can be invaluable, especially when the application demands specific behaviors, optimizations, or adaptations beyond Python's standard dictionary.