C950 Task-1 WGUPS Algorithm Overview

(Task-1: The planning phase of the WGUPS Routing Program)

Austin Pelley

ID #011191773

WGU Email: apell52@wgu.edu

October 29, 2024

C950 Data Structures and Algorithms II

# Introduction

The purpose of this project is to develop a program that effectively routes deliveries for WGUPS by adhering to specified delivery requirements and constraints. My approach will focus on constructing a data structure that efficiently stores and organizes package information, allowing for quick access and updates. I plan to integrate an algorithm that optimizes routing by minimizing travel distance and time, ultimately enhancing delivery efficiency. The program will also feature a user-friendly interface to enable users to view package delivery statuses and monitor the cumulative distance traveled by WGUPS trucks. Python will be my primary programming language, as it offers robust libraries and structures suitable for handling the data and logic requirements of this project.

# A. Algorithm Identification

For my project, I have selected the Nearest Neighbor Algorithm as the core approach to efficiently deliver packages. This algorithm will enable my program to determine the optimal route dynamically, starting at the hub and selecting the next closest delivery location based on both proximity and any priority criteria, such as delivery deadlines. As a self-adjusting algorithm, Nearest Neighbor continually adapts to each completed delivery by recalculating the route based on remaining package destinations and their respective time constraints. This method is well-suited to meet the project’s requirements, as it reduces overall travel distance while ensuring that high-priority packages reach their destinations on time. By allowing the program to self-adjust in real-time, this algorithm supports both route efficiency and flexible response to any changes in delivery priorities or deadlines.

# B. Data Structure Identification

For my project, I have chosen to use a hash table as the self-adjusting data structure to store package data efficiently alongside the Nearest Neighbor algorithm. The hash table will store each package’s unique data as key-value pairs, where the key is the package’s unique ID and the value holds additional information, such as the delivery address, city, state, zip code, delivery deadline, package weight, and any special notes. I selected this structure because it enables efficient, near-instant access to specific packages using only the package ID, which is essential for quickly updating and retrieving package information during the routing process. The hash table's organization allows all relevant package data to be stored and accessed in a way that keeps each package’s attributes connected, making it easy to retrieve and update details like status or delivery time in real-time. This setup will allow my program to both manage and adjust package details fluidly as the delivery route progresses.

# B1. Explanation of Data Structure

In my project, the hash table data structure ensures that each package’s information is linked directly to a unique package ID, which serves as the key to access the entire data set for that package. This approach allows me to keep all critical information—such as delivery address, city, state, zip code, deadline, weight, and special instructions—associated with one identifier, simplifying how the program retrieves and updates data. By maintaining these components in a single key-value pair, the hash table enables quick access to related attributes, ensuring that any modifications to package status, delivery time, or truck assignment can be updated efficiently without having to search through unrelated data. This organization also supports the Nearest Neighbor algorithm by enabling immediate access to each package’s destination and deadline information, which is crucial for making route adjustments during delivery.

# C1. Algorithm’s Logic

1. DECLARE `truck` as the starting delivery vehicle

2. INITIALIZE `truck` at the hub location

3. FOR each package in the `package list`

- ASSIGN status as "in hub"

- IF the package has a strict delivery deadline

- MARK as a high-priority package

ENDFOR

4. WHILE there are undelivered packages in the `package list`

- SET `nearest package` as the package closest to the current location of `truck`

- IF multiple packages have a priority deadline

- SELECT the nearest high-priority package to deliver next

- UPDATE `truck` mileage and time for distance traveled to `nearest package` address

- MARK `nearest package` as “delivered”

- UPDATE `truck` location to the address of the `nearest package`

ENDWHILE

5. OUTPUT total mileage traveled by each `truck`

To deliver packages efficiently, I plan to implement a Greedy Nearest Neighbor algorithm that selects the closest package delivery location to the current stop, prioritizing packages with stricter delivery deadlines. The algorithm will start each truck from the hub, search for the nearest undelivered package location within the package list, and calculate the shortest distance to each eligible package address, updating its status as "en route." Using a while-loop, I will iterate until all packages are delivered, moving from one address to the next closest available, updating the truck’s current location, time, and mileage at each stop. This approach minimizes the total distance traveled, as each truck dynamically adjusts its route based on the location of the closest remaining packages, aiming to optimize efficiency. Through this structured route optimization, my code addresses the project's delivery constraints while ensuring a quick and effective delivery plan.

# C2. Development Environment

The development environment I will use for the project will include PyCharm Professional (Version 2024.2.4) as my IDE, Windows 11 Pro Education (Version 23H2) as my operating system, and the version of Python version for the project will be 3.12.2. The hardware components of my development environment are an Intel i9-13900K, a Nvidia RTX 4080, 32gb of RAM, and a 27’ Asus monitor.

# C3. Space and Time complexity using Big-O notation

To evaluate the efficiency of my program, I analyzed both time and space complexity for each significant segment and for the program as a whole. The hash table for storing package data is designed with a time complexity of O(1) for insertion and retrieval. This constant time performance is achieved by using unique keys, which minimizes lookup times regardless of the total number of packages. In terms of space complexity, the hash table requires O(n), where n is the number of packages. This linear space complexity accounts for each package's data being stored as a unique entry, ensuring each entry only stores necessary information for efficient retrieval. The function for calculating distances between two addresses operates with a time complexity of O(n). It iterates over a list of addresses sequentially to identify the correct entry, so its time complexity is linear. The space complexity for this function is also O(n) because it relies on a pre-existing list of distances between addresses, stored in memory for rapid access during delivery calculations. My nearest neighbor algorithm, which determines the order of package deliveries, has a time complexity of O(n^2) due to the nested loop structure. For each package on a truck, the algorithm iterates over all undelivered packages to find the shortest available route. The space complexity for this algorithm is O(n), as it only requires storage for the list of undelivered packages and the temporary storage needed for the nearest neighbor calculations. The function to print a single package has a time complexity of O(1). This efficiency results from using the hash table and accessing a package by its ID, which avoids any iteration. The space complexity for this operation is O(1), as no additional data structures or temporary storage are required.

Considering all elements, the overall time complexity of the program is O(n^2). This is primarily due to the nested loops within the nearest neighbor algorithm, which scales quadratically with the number of packages. The overall space complexity of the program is O(n). Each data structure (the hash table for packages, the list for distances, and the truck and package lists) operates with linear space requirements, maintaining an efficient use of memory as the program scales with more packages. In summary, the program’s design ensures efficient memory usage with O(n) space complexity and effective performance, although the nearest neighbor algorithm contributes to the O(n^2) time complexity for overall package delivery operations.

# C4. Scalability and Adaptability

The capability of my solution to scale and adapt to a growing number of packages is a crucial aspect of its design as a software solution for package delivery. By employing the nearest neighbor algorithm, I can efficiently manage an increasing volume of packages. This algorithm's self-adjusting nature means that as new packages are added, the program can seamlessly incorporate them into the delivery route without a significant drop in performance. The use of a hash table for storing package data enhances this adaptability, allowing for quick retrieval and management of package information, regardless of the volume.

However, there are some limitations to consider. As the number of packages grows, the algorithm's efficiency may decline due to its O(n^2) complexity when searching for the nearest neighbor. This means that while the system can handle additional packages, performance may degrade significantly with a substantial increase in volume. Additionally, if the geographical area expands or delivery conditions change, the algorithm may require further optimization to maintain efficiency. Therefore, while the solution is designed to scale, continuous monitoring and potential adjustments will be necessary to ensure optimal performance in real-world deployment.

# C5. Software Efficiency and Maintainability

The software design of my package delivery solution is structured to be both efficient and easy to maintain, ensuring it can adapt to future needs. In terms of efficiency, the overall time complexity of my program is primarily driven by the nearest neighbor algorithm, which has a complexity of O(n²) when processing deliveries. While this is quadratic time complexity, the use of a hash table for package data retrieval ensures that certain operations, like accessing individual package information, run in constant time O(1). This design choice helps mitigate some of the performance issues associated with larger datasets, allowing the program to perform adequately even as the number of packages grows.

From a maintainability perspective, I have focused on writing clear, modular code that adheres to good software design principles. Each component of the program, such as the package class, truck class, and routing algorithm, is encapsulated within its own module. This modular approach makes it easier for other developers to understand the logic and functionality of the code, as they can work on individual parts without needing to navigate the entire codebase. Additionally, I have used meaningful variable names and added comments throughout the code to clarify the purpose of each section. By following these practices, I aim to create a program that can be easily understood, repaired, and enhanced by developers beyond myself, ensuring that the software can evolve as new requirements arise.

# C6. Self-Adjusting Data Structures

The hash table I have chosen as the self-adjusting data structure for my package delivery solution offers several strengths and weaknesses that are important to consider as I plan my project. One of the key strengths of a hash table is its ability to provide fast data retrieval. With an average time complexity of O(1) for lookup operations, the hash table allows me to access package information quickly using unique package IDs as keys. This efficiency is crucial for ensuring that my program can handle multiple package queries in real-time, enhancing overall performance. Another strength of the hash table is its adaptability to dynamic datasets. As the number of packages fluctuates, I can easily insert or delete entries without significant performance degradation. This flexibility is vital for my project, as package deliveries can vary greatly depending on the time of day or season.

However, there are also weaknesses to consider. One notable issue is that hash tables can suffer from collisions, where multiple keys hash to the same index. While techniques like chaining or open addressing can mitigate this, collisions can still lead to increased time complexity, particularly in the worst-case scenarios where lookups may degrade to O(n). Additionally, if the hash table is not adequately sized or if the hash function is poorly designed, it can lead to inefficient memory usage and performance bottlenecks. Overall, while the hash table is a powerful tool for my project's data management needs, I must remain mindful of its limitations and take steps to ensure optimal performance as I implement it.

# C7. Data Key

In my project focused on efficient delivery management, I will use the package ID as the key for my hash table. The primary reason for this decision is that the package ID serves as a unique identifier for each package in the system. By assigning a distinct ID to every package, I can ensure that each entry in the hash table corresponds to only one specific package. This uniqueness allows for quick and accurate retrieval of all associated information, such as the delivery address, deadline, weight, and status. Using package ID as the key also simplifies the management of delivery data. Unlike other components, such as the delivery address or city, which may not be unique and could lead to potential conflicts or confusion, the package ID provides a reliable reference point. For example, multiple packages could be delivered to the same address, especially in densely populated areas, but each of those packages will still have its own unique ID. This helps maintain clarity in the data and prevents any mix-ups during the retrieval process. The package ID is less likely to change compared to other attributes, such as delivery status or deadlines, which can be modified during the delivery process. This stability is advantageous because it reduces the chances of needing to rehash or adjust the data structure frequently, thus contributing to the overall efficiency of the program.

In summary, I selected the package ID as the key for my hash table due to its uniqueness, stability, and effectiveness in providing quick access to package-related information. This choice is crucial for ensuring efficient delivery management and optimizing the performance of my program as I continue to develop it.

# D. Sources

C950 WGUPS Project Implementation Steps - Example - Nearest Neighbor

Retrieved October 16, 2024, from

https://srm--c.vf.force.com/apex/coursearticle?Id=kA03x000001DbBGCA0

# E. Professional Communication