C950 Task-1 WGUPS Algorithm Overview

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

John Mahon

ID 001193994

WGU Email: jmaho24@wgu.edu

6/1/2025

C950 Data Structures and Algorithms II

# Introduction

The goal of this project is to build a simulation for the WGUPS package delivery system that efficiently delivers all packages using three trucks while meeting delivery deadlines and minimizing total mileage. I plan to implement a custom hash table to store package data for fast lookup. I will use the Nearest Neighbor algorithm to construct delivery routes. This algorithm fits my problem-solving style and allows me to focus on simplicity and fast computation. I expect to include logic to handle special cases like delayed packages and address corrections. My overall approach is object-oriented and modular and will prioritize scalability and maintainability.

# A. Algorithm Identification

My program will use a Nearest Neighbor Algorithm, a self-adjusting, greedy algorithm. A greedy algorithm is “an algorithm that, when presented with a list of options, chooses the option that is optimal at that point in time” (Lysecky & Vahid, 2018). The algorithm in my project will take a truck’s position and choose the next package stop based on which address is immediately closest. It will iterate through the package manifest until all packages are delivered and then return to the hub.

# B. Data Structure Identification

I will use a self-adjusting, chaining hash table to store and manage package data for this project.

# B1. Explanation of Data Structure

I will implement a hash table, as described in the “Let’s Go Hashing” webinar, is “a data structure that stores unordered items by mapping (or hashing) each item to a location in an array” (Tepe, 2024). with chaining to store and fetch package objects (my project will be object-oriented). The hash table will be defined as a Python class in its own file. It will contain a list of 40 buckets (matching the number of packages provided), and each bucket will be initialized as an empty list to support separate chaining for collision handling.

The native Python hash() function will be used to generate a numeric hash from each WGUPS-provided package ID. The formula hash(packageID) % table\_size (with a table size of 40) will ensure that all keys are mapped to a bucket index between 0 and 39.

When a collision occurs (i.e., two package IDs hash to the same index), both entries will be stored in the same bucket as key-value pairs with the key being the package ID and the value being the full package object. This chaining mechanism allows multiple packages to coexist in the same bucket without overwriting one another, ensuring packages can be inserted, retrieved, and updated based on ID.

This structure will allow for fast lookups and status updates, which is essential for a time-based simulation like where delayed packages become available or corrected addresses applied mid-simulation.

# C1. Algorithm’s Logic

START the simulation

FOR each truck

LOAD the assigned packages into the truck

SET truck departure time based on dispatch schedule

SET current location to the hub

SET truck mileage to 0

WHILE the truck has undelivered packages

SET shortest distance to infinity

FOR each remaining package on the truck

CALCULATE distance from current location to the package’s address

IF this distance is shorter than the current shortest distance

SET this package as the closest package

UPDATE shortest distance

ENDIF

ENDFOR

TRAVEL to the closest package’s address

UPDATE current location to the delivery address

UPDATE truck mileage by adding the shortest distance

CALCULATE time taken using distance and speed of 18 MPH

ADD that time to the truck’s current time

DELIVER the package

REMOVE the package from the truck’s manifest

ENDWHILE

RETURN to the hub

CALCULATE return distance to hub

UPDATE truck mileage by adding return distance

CALCULATE time to return using speed of 18 MPH

ADD that time to the truck’s current time

ENDFOR

END the simulation

# C2. Development Environment

**Development environment hardware:**

Intel i5 6600K

16 GB DDR4 RAM

nVidia GeForce RTX 3050

1TB HDD

Windows 10 Home

USB Keyboard + Mouse

24” LCD Monitor via DisplayPort

**Development environment software environment:**

I am using the PyCharm Community 2024.3.1.1 IDE using Python version 3.8.

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

**Hash Table**: The hash table in my program has an average case time complexity of O(1) for insert and search operations using Python’s hash() function to assign keys to buckets. It uses separate chaining for collisions, where each bucket is a list of key value pairs. When many keys collide, the worst case, the function would be O(n). The space complexity is O(n) with one entry per package.

**Nearest Neighbor Routing Algorithm:** The nearest neighbor algorithm has a time complexity of O(n^2), since it repeatedly searches through the package list to find the closest one, resulting in nested iteration. The space complexity is O(n) because it tracks undelivered packages during routing.

**Distance Lookup Function:** The distance lookup function has a time complexity of O(1) using a preloaded 2D matrix to retrieve distances by index. Its space complexity is O(n^2) because the matrix stores distances between all address pairs.

**CSV Loading:** Loading package date from CSV has a time complexity of O(n), loading the distance matrix is O(n^2). Space complexity is O(n) for package csv and O(n^2) for the distances.

**CLI / Print All Packages:** Printing all packages is O(n) as it loops through the hash table. Searching and printing by package ID is O(1) due to direct hash table access.

**Overall Program:** Overall, the program has a time complexity of O(n^2) due to the nearest neighbor routing algorithm’s performance of nested iterations over the package list to determine the delivery sequence. The space complexity is also O(n^2) because of the preloaded 2D distance matrix storing distances between all address pairs. Other components, like the hash table grow linearly with the number of packages O(n) but are outweighed by the matrix’s quadratic growth.

# C4. Scalability and Adaptability

My solution is designed to scale with a growing number of packages by using modularly designed classes and flexible data structures. The hash table can store and retrieve any number of packages without changes to its structure, and the hash function adapts automatically based on the input keys. The nearest neighbor routing algorithm is self-adjusting, meaning it recalculates the next closest delivery address at each step based on the remaining package list. If needed, the truck fleet could be expanded easily by instantiating more Truck objects. While the system remains fully functional with larger input sizes, performance may decrease due to the O(n^2) time complexity of the routing algorithm.

# C5. Software Efficiency and Maintainability

The program uses a hash table with separate chaining to store and retrieve packages. It is self-adjusting because it handles collisions dynamically by placing multiple key-value pairs into a list within the same bucket when needed. As described in the course text, this is “a collision resolution technique where each bucket has a list of items” (Lysecky & Vahid, 2018). No manual rebalancing or resizing needs to be implemented.

The structure is very fast on average — O(1) for insert, lookup, and delete — and it continues to work well as the number of packages scales. However, if too many keys hash to the same bucket, performance could degrade toward O(n), especially without load balancing or resizing. Still, for the problem this project is meant to solve, the chaining hash table is expected to scale well and remains a clean and efficient solution.

# C6. Self-Adjusting Data Structures

The program uses a hash table with separate chaining to store and retrieve packages. It is self-adjusting because it handles collisions dynamically by placing multiple key-value pairs into a list within the same bucket when needed. As described in the course text, this is “a collision resolution technique where each bucket has a list of items” (Lysecky & Vahid, 2018). No manual rebalancing or resizing needs to be implemented.

The structure is very fast on average — O(1) for insert, lookup, and delete — and it continues to work well as the number of packages scales. However, if too many keys hash to the same bucket, performance could degrade toward O(n), especially without load balancing or resizing. Still, for the problem this project is meant to solve, the chaining hash table is expected to scale well and remains a clean and efficient solution.

# C7. Data Key

I chose package ID as my key because its unique and will not change throughout runtime. This makes it ideal for storing and fetching package data in a hash table with O(1) lookup. Other fields like address, zip code, or status aren’t unique and could change during runtime, making them poor candidates for a stable key. With package ID as the key, I can instantly access any package to add, update, or track progress within the implementation or via the CLI.

# D. Sources

Lysecky, R., & Vahid, F. (2018, June). *C950: Data structures and algorithms II.* zyBooks. Retrieved June 5, 2025, from https://learn.zybooks.com/zybook/WGUC950AY20182019/

Tepe, C. (2020). *Let’s go hashing* [Webinar]. Western Governors University.

Tepe, C. (2020). *Getting greedy, who moved my data?* [Webinar]. Western Governors University.