# C950 Task-1 WGUPS Algorithm Overview

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

Jessica Short
ID #010500487
WGU Email: jnic331@wgu.edu
11/29/2024

C950 Data Structures and Algorithms II

## Introduction

This document provides a summary of the planning phase for the Western Governors University Parcel Service (WGUPS) route and delivery program. The solution is designed to ensure timely delivery of all packages while limiting the total distance traveled by the three delivery trucks to no more than 140 miles. A complete list of project specifications is available at: https://tasks.wgu.edu/student/010500487/course/30860017/task/4042/overview.

## A. Algorithm Identification

The nearest neighbor algorithm is well-suited for this project because it selects the closest stop at each step, effectively minimizing the total travel distance. Its simplicity and ability to adapt dynamically to changing conditions make it an ideal choice for this solution.

## B. Data Structure Identification

A hash table is a dynamic data structure that is ideal for storing package data in this project. It offers fast access times, a key-value pair structure for efficient organization, support for real-time updates, and scalability.

## B1. Explanation of Data Structure

The hash table used in this project manages the relationship between data components through a key-value pair structure. Each package is uniquely identified by its package ID, which serves as the key. The corresponding package details, stored as the value, include attributes such as the address, city, state, zip code, deadline, weight, status, departure time, delivery time, and additional notes, all organized within a structured object.

## C1. Algorithm’s Logic

In this section, the proposed implementation of the nearest neighbor algorithm is demonstrated via pseudocode:

**function find_address_id(target_address):**

FOR each row in address_data

    IF target_address is found in the third column of row THEN

        OUTPUT the first column of row as an integer

    ENDIF

ENDFOR

**function calculate_distance(location1, location2):**

SET dist to the distance value from location 1 to location 2

IF dist does not contain a value THEN

    SET dist to the distance value from location 1 to location 2

ENDIF

OUTPUT dist as a floating-point number

**function deliverTruckPackages(truck):**

CREATE an empty list, packages_in_transit, to store packages to go out for delivery

FOR each package assigned to a truck

    ADD the package to the list packages_in_transit

ENDFOR

CLEAR the truck's package list

WHILE the list packages_in_transit is not empty

    CALL find_next_package() which will OUTPUT next_package_delivery and next_delivery_distance

    IF next_package_delivery is not empty THEN

        CALL update_truck_and_package_status(truck, next_package_delivery, next_delivery_distance)

        REMOVE next_package_delivery from the list packages_in_transit

    ENDIF

ENDWHILE

**function find_next_package(truck, packages_in_transit):**

FOR each package in packages_in_transit:

    IF the package ID is 25 or 6 THEN

        SET distance to calculate_distance(calculate_distance(current truck location), find_address_id(package delivery address))

        OUTPUT package, distance

    ENDIF

ENDFOR

SET nearest_package to None

FOR each package in packages_in_transit

    SET distance to

    calculate_distance(calculate_distance(current truck location), find_address_id(package delivery address))

    IF distance < 30 THEN

        SET shortest_distance to distance

        SET nearest_package to package

    ENDIF

ENDFOR

OUTPUT nearest_package, shortest_distance

**function update_truck_and_package_status(truck, package, distance):**

UPDATE the truck's mileage by adding the distance traveled

UPDATE the truck's current location to the package's delivery address

CALCULTAE the time taken for the truck to travel the distance and update the truck's time

SET the package's delivery time to the truck's current time

SET the package's departure time to the truck's departure time

*Deliver the packages*

CALL deliverTruckPackages(truck1)

CALL deliverTruckPackages(truck3)
SET truck2's departure time to the earlier of Truck 1's or Truck 3's finish time
CALL deliverTruckPackages(truck2)

## C2. Development Environment

In this section, I will describe the hardware and software that I will use to create this application.

Hardware:

For my hardware, I will use a HP Pavilion laptop, model number 15T-EG100. The specifications for this laptop are listed below:

- Processor: 11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz
- RAM: 16.0 GB (15.8 GB usable)
- System Type: 64-bit operating system, x64-based processor

The laptop will be connected to 2 external monitors via HDMI and an external keyboard and mouse via USB.

Software:

I will use Windows 11 version 23H2 for my operating system.

I will use PyCharm 2024.3 Community Edition as my IDE.

I will create the program with Python 3.13.0.

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

The worst-case space complexity of the application is O(N), while the worst-case time complexity is $O(N^2)$.

The best-case space and time complexity for the application are both O(N).

In the table below, the worst-case space and time complexities are summarized using Big-O notation for each major section of the application.

| Method | Line Number | Space Complexity | Time Complexity |
|---|---|---|---|
| class ChainingHashTable | | | |
| init | 33 | O(1) | O(1) |
| insert | 40 | O(1) | O(N) |
| search | 58 | O(1) | O(N) |
| remove | 70 | O(1) | O(N) |
| class Package | | | |
| updateStatus | 112 | O(1) | O(1) |
| _updatePackage9Address | 124 | O(1) | O(1) |
| loadPackagesFromCSV | 136 | O(N) | O(N) |
| find_address_id | 189 | O(1) | O(N) |
| calculate_distance | 197 | O(1) | O(1) |
| deliverTruckPackages | 230 | O(N) | O(N^2) |
| find_next_package | 269 | O(N) | O(N) |
| update_truck_and_package_status | 289 | O(1) | O(1) |
| update_and_print_status | 321 | O(1) | O(1) |
| display_package_status_by_truck | 330 | O(1) | O(N) |
| UI loop | | | |
| None | 346 | O(1) | O(N) |
| **Total** | | 11 + 3N = O(N) | 6 + 7N + N^2 = O(N^2) |

## C4. Scalability and Adaptability

This solution is designed with scalability in mind. The hash table is efficient, providing O(1) average time complexity for insertion, search, and removal. As the number of packages increases, the hash table can easily accommodate them without significant performance degradation. Additionally, the size of the hash table can be adjusted if needed to maintain optimal performance.

The modularity of this program supports scalability by allowing the current, well-defined classes and functions to be easily modified or extended as new requirements for packages, deliveries, or trucks arise in the future.

## C5. Software Efficiency and Maintainability

This application is designed to be both efficient and easy to maintain. The use of a hash table ensures an average-case time complexity of O(1) for inserting, searching, and deleting packages, enabling quick lookups even as the number of packages

increases. The nearest neighbor algorithm, though simple, effectively minimizes overall mileage and delivery time.

The application follows object-oriented principles, featuring modular design with well-defined classes and functions. Comprehensive and clear comments throughout the code further enhance its readability and ease of debugging. Additionally, the data-driven design allows updates to package or address data by simply modifying the CSV files, eliminating the need for code changes.

The user interaction loop offers a clear and intuitive interface, with robust error handling to improve reliability and reduce the risk of runtime errors.

**C6. Self-Adjusting Data Structures**

Strengths of the hash table:

A hash table is an ideal data structure for this project, offering several key advantages. It provides an average-case time complexity of O(1) for insert, search, and delete operations, enabling fast lookups of package details. The chaining mechanism allows multiple key-value pairs to be stored in a single bucket, effectively resolving hash collisions while maintaining data integrity.

The hash table is both modular and scalable, implemented as a separate class that is easily adaptable and self-contained. The initialcapacity parameter can be easily adjusted to accommodate varying data sizes. By using package IDs as keys, the hash table simplifies the process of locating, updating, or deleting package information. Furthermore, the insert method efficiently handles updates—if a package ID already exists, it modifies the existing entry rather than creating a duplicate.

Weakness of the hash table:

There are some disadvantages to using a hash table for this project, however. If the load factor becomes too high or the hash function doesn’t distribute keys evenly, the time complexity can degrade to O(N) for search, insert, and delete operations. Although the initialcapacity can be easily adjusted to accommodate more packages, the hash table does not resize automatically. As the number of packages increases, performance may suffer unless the initialcapacity is manually updated to a higher value. Additionally, the current implementation lacks error handling for cases where the

table's capacity is exceeded, or an invalid key type is provided. Without these checks, the program may experience runtime errors or unpredictable behavior.

**C7. Data Key**

Package ID is the best choice for the key in the hash table for this project. It is unique, ensuring that each package's data can be accessed directly and without ambiguity. Additionally, because the Package ID is an integer, it serves as a simple and efficient key.

Opting for Package ID as the key helps reduce the likelihood of collisions.  This is because package IDs are generally assigned in a consecutive sequence, which creates a predictable pattern. This predictability leads to a more balanced distribution of hash values, ensuring that the hash function spreads the keys more evenly across the available buckets. As a result, the chances of multiple keys landing in the same bucket are minimized.

**D. Sources**

The hash table used in this solution is a modified version of the one in zyBooks.

Figure 7.8.2: Hash table using chaining

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Accessed 11/29/2024 from

https://learn.zybooks.com/zybook/WGUC950Template2023