# WGUPS Algorithm Overview

John McGinnes

02/24/2025

## Introduction

The performance assessment involves creating a package delivery tracking system using the Python programming language (Python Version 3.13.1). The provided files included data for 40 packages and their addresses, to be divided into three trucks with only two drivers operating at a time. This process also considers the special instructions with each package, including one case where the address is initially incorrect and corrected later in the day.

To complete this task, I intend to use conditional statements to load the trucks based on the special instructions first, then implement the Nearest Neighbor algorithm to determine the shortest distance between locations. This algorithm is greedy since it will continue to determine the immediate shortest distance between addresses, only terminating when no addresses remain.

## A. Algorithm Identification

I have decided to use the Nearest Neighbor greedy algorithm as the core component of this program. The algorithm will find the immediate shortest path to the next location on the route, it does not use any previous or future decisions only the current shortest path.

## B. Data Structure Identification

I will use a hash table as the primary data structure for this program. This will allow searching, inserting, and updating in constant time (on average) for efficiently storing the data.

## B1. Explanation of Data Structure

The hash table will be using the package_id as a key since these are unique identifiers for each package. Each key will correspond to a value, which in this case will be an instance of the Package class. The Package Class contains the attributes for each of the package records extracted from the CSV file (address, location, weight, etc.)

The hash table is the ideal choice as a data structure for this program because it has very efficient time complexity, O(1) for searching, inserting, and deleting. This feature is crucial for larger data sets, but even for a fixed set of data used here, it is beneficial.

Because the package ID is a unique value and is used as a key, the hash table will allow constant time access to the package data, including addresses, status, and delivery time. This will significantly speed up the process of locating and updating package information as the trucks are engaged in their routes, to ensure that there are no delays in delivery schedules.

The hash table is also able to grow and shrink dynamically as packages are added and removed, which provides strong scalability for the program. If additional packages are added the table will accommodate them easily, and as packages are removed the memory used will adjust accordingly. It performs this dynamic resizing based on a load factor (Often

0.75), once the load factor is exceeded, the hash table will double in size and rehash the existing data to ensure fewer collisions while maintaining constant time operations for most functions.

The hash table ensures quick searches and easy updates for the package information, which is crucial in a system where package locations will update and delivery statuses change frequently.

## C1. Algorithm's Logic

The Nearest Neighbor Greedy Algorithm as implemented here will perform four basic steps:

1. Initialize the starting location for the truck.
2. Create a list of packages to be delivered.
3. While there are packages in the list above
    a. Find the closest package location
    b. Update the truck's total distance.
    c. Update the truck's current address to the closest package location.
    d. Change the package departure time to the truck's last departure time.
    e. Update the package arrival time to match the truck arrival time.
    f. Remove the delivered package from the list of packages left to be delivered.
4. Repeat the process until no packages remain.

The complete pseudocode is as follows:

```
START

SET truck.current_address = “Hub”
SET truck.total_distance = 0
SET unvisited_packages = all packages assigned to the truck

WHILE unvisited_packages is not empty
    SET closest_package = None
    SET closest_distance = INFINITY

    FOR each package in unvisited_packages
        SET distance = determine_distance(truck.current_address,
package.address)
        IF distance < closest_distance
            SET closest_distance = distance
            SET closest_package = package
    END FOR
    UPDATE truck.current_address = closest_package.address
    UPDATE truck.total_miles += closest_distance
    SET closest_package.departure_time = truck.latest_departure
```

```
        SET closest_package.arrival_time = truck.latest_departure +
(closest_distance / truck. speed)


        REMOVE closest_package from unvisited_packages
END WHILE
END
```

## C2. Development Environment

I will be using the PyCharm Community Edition 2024 version 3.2 by JetBrains as a development platform. As mentioned above, this will be using Python version 3.13.1 on Windows 11. The hardware that I am using to build and run the program is a Lenovo Legion laptop, with a 13th Gen Intel Core i9 2200Mhz/24 core processor and 16GB of main memory. It also has a 2TB Solid State drive for secondary storage and an NVIDIA GeForce 4070 integrated graphics chip, though those are not likely to influence this program.

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

I have broken down the time and space complexity to each of the following main functions:

1. Initialization and reading data from the CSV files:
    a. This stage includes reading through the CSV files, with each function iterating over the entire file one time.
    b. Time complexity for this stage would be O(n) with "n" being the number of rows in the CSV files.

   c. The space complexity is also O(n) based on the space used to store “n” rows in the package, address, and distance files.

2. Hash table functions:
   a. The table_insert function also has a time complexity of O(n) with “n” being the number of packages. This would override the time complexity of the table_search function which is O(1).
   b. The space complexity for this section would be O(n) with “n” as the number of packages.
3. Adding packages to the trucks:
   a. This part of the program involves sorting the packages and assigning them to the trucks, iterating once for O(n) time complexity, though there is a nested loop to compare the packages for distance and assign which runs with O(n^2) time complexity, resulting in a total of O(n^2) time complexity here.
   b. The space complexity for this section would be O(n) with “n” as the number of packages. Specifically, it is affected by creating the truck1_pkg_list, truck2_pkg_list, and truck3_pkg_list and a list for low-priority packages.
4. Delivering packages using the Nearest Neighbor Algorithm:
   a. This section involves iterating over the undelivered packages, and within each iteration calculating the distance from the current location to the next nearest location. The time complexity for this section is O(n^2) with “n” being the number of packages, due to the nested loops.
   b. Storing the temporary variables (like nearest_package and nearest_distance) results in O(n) space complexity.
5. User Interface section:

a. The user interface section has a time complexity of O(1) for most options that the user can select, though none would ever exceed O(n) here.
b. The space complexity for this section is O(1) as well as it does not use any data structures of its own, only those created and addressed above.

The overall time complexity for the program in the worst case would be O(n^2). The add_packages_to_trucks function uses nested loops to assign low-priority packages based on distance (O(n^2)) and the deliver_packages function uses a nested iteration as well to implement the Nearest Neighbor algorithm (O(n^2) for each truck, which is technically O(3n^2) that simplifies to O(n^2)). The best case time complexity would be O(n) assuming that the packages were already sorted and needed no comparison.

The overall space complexity would be O(n) (with “n” as the number of packages) as the program primarily uses lists and the hash table to store data.

## C4. Scalability and Adaptability

The program uses a hash table to store the data, which supports O(1) searches, inserts, and updates, so even with a larger set of data it would not be an issue to store the information efficiently.

The bottleneck to scalability would most certainly be the implementation of the Nearest Neighbor algorithm in the deliver_packages function. As the number of packages increases, this would cause significant decreases in performance because of the O(n^2) time complexity detailed in the previous section. Using alternative algorithms (like Dijkstra's Algorithm) would be more complicated to implement but could result in improvements in the scalability. The sorting functions could also be addressed using external libraries to eliminate the remaining nested loops.

The program is very adaptable, however. With the current functionality other statuses can easily be added and additional trucks created by adding minimal code.

## C5. Software Efficiency and Maintainability

The program was designed to be as efficient as possible given the constraints provided, with most operations using O(1) to O(n) time complexity, though the Nearest Neighbor algorithm in the deliver_packages method does limit the efficiency with a larger set of data, running with O(n^2) time complexity.

The code is easily maintained by design, with the package, truck, and hash table classes being broken out into separate files for easy configuration and updating. I have also included concise comments throughout the program to allow for easy debugging and enhancements by other programmers if needed.

## C6. Self-Adjusting Data Structures

The benefit of using a hash table data structure to store the package data is the ability to search, insert, and update in O(1) time complexity. This would make this data structure highly scalable beyond the given dataset. It will also adjust based on the size if needed.

Issues could occur with a hash table in the form of data collisions if multiple keys have the same hash (though this can be mitigated by chaining) and the order is not preserved, which would make sequential data retrieval difficult if it was required.

Hash tables are also able to dynamically self-adjust to the capacity required. As the hash table fills, the chance of data collision increases. The hash table solves this issue by using a load factor (a ratio of the number of entries to the table size). When the load factor exceeds a specific threshold, usually 0.5 or 0.75, the hash table automatically resizes (doubles its capacity of buckets) to maintain an efficient operation.

During this process, all existing entries must be rehashed. This may cause a process to take longer than expected. Once the resizing is complete, however, the occurrence of collisions will be minimized and the average time complexity will still be O(1) for searching, inserting, and deleting. As the number of packages may vary each day, a dynamically adjusting system is ideal for the WGUPS application.

## C7. Data Key

The package ID was chosen as the key for my program. This decision was mainly because it is a unique value for each package. Other values could have been used in certain circumstances (address, deadline, or status) but the package ID is the optimal choice in this case to keep the program efficient and the hash table operations within O(1) time complexity.

## D. Sources

Lysecky, R., & Vahid, F. (2018, June). C950: Data Structures and Algorithms II. zyBooks. Retrieved February 24, 2025, from

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