Jack Compton
ID #001511804
jcomp40@wgu.edu
April 11th, 2023

C950 Data Structures and Algorithms II

## Introduction

The WGUPS program is a package delivery service that operates in the greater Salt Lake City area. With a fleet of delivery trucks, the company strives to deliver packages efficiently and on time. To achieve this goal, the WGUPS program uses the nearest neighbor algorithm, a popular method for solving the traveling salesman problem. This algorithm is used to determine the most efficient delivery routes for each truck based on the proximity of packages to one another. By minimizing the distance traveled by each truck, the WGUPS program is able to reduce delivery times and costs, while maximizing customer satisfaction.

### A. Algorithm Identification

The self-adjusting algorithm used in the WGUPS project is the Nearest Neighbor algorithm, which starts at a predefined hub location and selects the nearest unvisited location until all packages have been delivered. However, the code for choosing the packages is not an algorithm. Instead, the trucks are manually loaded based on the package specifications and delivery requirements specified in the scenario.

## **B1.** Logic Comments

Pseudocode for the Nearest Neighbor Algorithm used

```
for each delivery stop in truck_route:
    set min_distance to the distance between current and next stop
    set min_index to the index of the next stop in the route
    for each remaining stop in the route:
        set check_distance to the distance between current and remaining stops
        if check_distance is less than min_distance:
            set min_distance to check_distance
            set min_index to the index of the remaining stop
    if min_index is not equal to the index of the next stop:
        swap locations of the next stop and the stop with the minimum distance
    return the optimized truck_route
```

## **B2.** Development Environment

Programming Language: Python 3.11

IDE: PyCharm

OS: Mac OS

Hardware: M1 Chip MacBook Pro with 16gb of RAM

## B3. Space-Time and Big-O

Function Name	Space Complexity	Time Complexity
optimize_route()	O(n^2)	O(1)
plan_route()	O(n^3)	O(n)
get_route_data()	O(1)	O(n)
entire program	O(1) however, this grows if there are additional packages	O(n^3) where n is the number of packages that are delivered

### B4. Scalability and Adaptability

The application is designed to handle an increasing number of packages through the use of scalable elements such as the Nearest Neighbor Algorithm for optimizing delivery routes and the Chaining Hash Table data structure for efficient package lookup. These components allow the application to adapt to changes in package volume while maintaining optimal performance.

However, there are imitations to scalability. For example, the number of trucks available may not be enough to handle a larger number of packages, which may impact deadlines on delivery times.

## B5. Software Efficiency and Maintainability

Nearest Neighbor Algorithm used in the solution has a time complexity of  $O(n^2)$ , which is polynomial time, and the Chaining Hash Table used for package lookups has an average case time complexity of O(1), which is constant time. Therefore, the program runs efficiently, as it operates in polynomial time or better.

In terms of maintainability, the code has been structured in a modular and compartmentalized way, with functions for each specific task. The code also includes clear comments and naming conventions, making it easy for future developers to understand the logic and flow of the code. Additionally, the use of object-oriented programming principles make it easier to update and modify in the future.

## B6. Self-Adjusting Data Structures

### Strengths:

- 1. Efficient lookup time: Using a hash table allows for constant time O(1) lookup of package data. This is because the hash function can quickly identify the location of the package data in the table.
- 2. Dynamic sizing: Chaining hash tables allow for dynamic sizing, meaning the table can grow or shrink as necessary depending on the number of packages to deliver. This can lead to a more efficient use of memory.

#### Weaknesses:

- 3. Collision resolution: Chaining hash tables can experience collisions when two or more keys are mapped to the same slot in the table. This can slow down the lookup time and decrease the efficiency of the data structure.
- 4. Overhead for small data sets: If the number of packages to deliver is small, the overhead required to set up and maintain the hash table may outweigh the benefits of using the data structure. In such cases, simpler data structures may be more appropriate.

#### C. Original Code

The program was written using Python 3.11 in PyCharm and requires no special installations. The UI is developed in the command line and can immediately be ran and interacted with from within PyCharm (or another IDE supporting Python)

### D1. Explanation of Data Structure

A hash table is a data structure that uses a hash function to map keys (in this case, package IDs) to slots in an array. In our program, each package is assigned a unique ID, which is used as the key in the hash table.

When a package is loaded onto a truck, its information (destination, delivery deadline, etc.) is stored in the hash table at the index corresponding to its ID. This allows us to retrieve the package's information in constant time, regardless of the size of the data set.

The hash function used in our program is designed to evenly distribute the keys across the array, reducing the likelihood of collisions (when multiple keys map to the same index). In the rare event of a collision, our hash table uses chaining to resolve it.

Compared to a simple linear search, which would require iterating through every package in the list to find the one with the desired ID, a hash table allows for much faster and more efficient retrieval of package information. This is because the hash function provides an index to the exact location of the desired package's information, eliminating the need for iteration.

### G1. First Status Check

09:00AM

```
Processes 13. St. Folk Address: 208 5 State at Salt Lake City, UF 4611 | Deadline: COD | Salto: 9 | Delivery Status: or not the hold | Delivery Time: 224-6726 | Departure Time: 8:08:00 | Frances Deadline: COD | Folk Address: 208 00 Salt Lake City, UF 4615 | Deadline: 200 | Miles 9 | Delivery Status: or not collectively Time: 22-700 | Delivery Time: 8:00:00 | France Deadline: COD | Salt Lake City, UF 4615 | Deadline: 200 | Miles 9 | Delivery Time: 8:00:00 | France Status: 8:00:00 | France Sta
```

#### G2. Second Status Check

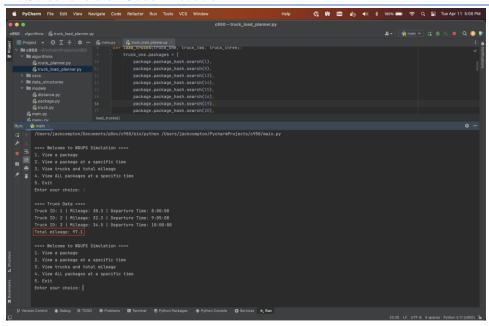
10:00AM

## G3. Third Status Check

12:05PM

schape ID: 1 | Full Address: 198 W Oakland Ave Salt Lake City,UT 84135 | Deadline: 500 | Kilo: 24 | Delivery Status: delivered at 80:38:20 | Delivery Time: 8:40:40 | Departure Time: schape ID: 2 | Full Address: 235 | South East City,UT 84136 | Deadline: 500 | Kilo: 24 | Delivery Status: delivered at 10:38:20 | Delivery Time: 10:38:20 | Departure Time: schape ID: 3 | Full Address: 235 | Campen RB Salt Lake City,UT 84135 | Deadline: 500 | Kilo: 2 | Delivery Status: delivered at 10:45:24 | Delivery Time: 10:42:40 | Departure Time: 10:46:40 | Deadline: 500 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:28 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:20 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:20 | Deadline: 100 | Kilo: 2 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:20 | Deadline: 100 | Kilo: 4 | Delivery Status: delivered at 10:47:20 | Delivery Time: 10:40:20 | Delivery T Process D: 38 | Folk Address: Cold 5 State of State Law City, UP 8011 | Dearline: Did | Nation V | Solitivery Status: delivered at 19:30-700 | Dearline: Did | Process D: 10 | Process D: 10 | State State Law City, UP 8015 | Dearline: Did | Nation V | Solitivery Status: delivered at 19:30-700 | Departure Time: 19:700 | Departure Time: Did | Process D: 10 | State State Law City, UP 8015 | Dearline: Did | Nation V | State Stat

#### H. Screenshots of Code Execution



## I1. Strengths of Chosen Algorithm

**Efficiency**: The Nearest Neighbor Algorithm is an efficient way to determine a close-to-optimal solution to the traveling salesman problem. The algorithm can quickly calculate distances between locations and determine the nearest "neighbor", allowing for the efficient construction of truck routes that minimize distance traveled.

**Adaptability**: The Nearest Neighbor Algorithm is also highly adaptable and can easily adjust to changes in package delivery requirements, such as adding or removing packages from the delivery list or adjusting delivery priorities.

## I2. Verification of Algorithm

The implemented algorithm meets all of the following requirements:

- Provide the total combined miles traveled by all trucks. It must be less than 140
  - confirmed by selecting Option 2 'View trucks and total mileage'
- State that all packages were delivered on time.
  - confirmed by selecting Option 4 'View ALL packages at a specific time' and setting the time to EOD
- State that all packages were delivered according to their delivery specifications.
  - confirmed by selecting Option 2 'View a package at a specific time' and setting the time to EOD and reviewing the package notes

### I3. Other possible Algorithms

1. Greedy Algorithm

## 2. Held-Karp Algorithm

## I3A. Algorithm Differences

## Approach:

- Nearest Neighbor Algorithm: starts at a random city and repeatedly selects the nearest unvisited city to the current city.
- Greedy Algorithm: selects the edge with the smallest weight and moves to the next node, iteratively.
- Held-Karp Algorithm: uses dynamic programming and examines all possible subproblems of the TSP to calculate the optimal solution.

## Time complexity:

- Nearest Neighbor Algorithm: O(n^2)
- Greedy Algorithm: O(n^2 log n)
- Held-Karp Algorithm: O(n^2 \* 2^n)

### Accuracy:

- Nearest Neighbor Algorithm: can quickly provide a decent approximation but may not always find the optimal solution.
- Greedy Algorithm: can provide a good approximation for some instances of the TSP but not all.
- Held-Karp Algorithm: can find the optimal solution for small instances of the TSP but becomes impractical for larger instances.

#### **Space complexity:**

- Nearest Neighbor Algorithm: O(n)
- Greedy Algorithm: O(n^2)
- Held-Karp Algorithm: O(n \* 2^n)

### J. Different Approach

During the project, I explored various software design styles and did not limit myself to a specific one while attempting to solve the problem. In hindsight, I realize that a better approach would have been to stick to a single design style, such as OOP or Functional programming, instead of using a multi-paradigm approach.

#### K1. Verification of Data Structure

- The total mileage traveled is 97.1
- All packages were delivered on time
- All packages are delivered per their specifications
- An efficient chaining hash table is present with a lookup (search()) function

All of these points can be verified in the UI

## K1A. Efficiency

When adding packages to a chaining hash table, the number of elements in the table increases. As a result, the time needed to complete the search for packages also increases. This is because more packages mean that more elements need to be stored and retrieved from the hash table, which takes additional time.

Specifically, the search time for a chaining hash table is directly proportional to the number of elements in each bucket (i.e., the number of packages with the same hash value). Therefore, if there are more packages with the same hash value, the search time for retrieving package information will be longer.

#### K1B. Overhead

As more packages are added to a chaining hash table, the space usage of the data structure will increase. This is because chaining hash tables typically use linked lists to store the key-value pairs, and each new key-value pair added will require additional memory to store the linked list nodes.

Additionally, as the number of packages grows, the number of collisions in the hash table may increase, which can further impact space usage.

## K1C. Implications

Adding more trucks or cities would not directly affect the lookup time of the chaining hash table for packages. The lookup time is dependent on the number of packages in the hash table, not the number of trucks or cities. However, adding more trucks or cities could indirectly affect the lookup time as it will result in more packages being added to the hash table.

For space usage, similar logic applies that there would be an indirect effect on the chaining hash tables space per package with a new city or any potential added requirements for additional trucks.

#### K2. Other Data Structures

#### 1. Binary Search Tree

a. A BST could have been implemented to store packages in a sorted order based on package ID, allowing for quick look-up times with an average time complexity of O(log n)

#### 2. AVL Tree

a. An AVL tree could have been used to maintain the balance of the tree, ensuring an even faster look-up time and avoiding the worst-case scenario of a skewed tree, where the time complexity could degrade to O(n).

### K2a. Data Structure Differences

## **Chaining Hash Table vs. Binary Search Tree (BST):**

The chaining hash table uses hash functions to map keys to indices in the table while a BST sorts keys and compares them to traverse the tree.

In a chaining hash table, collisions occur when two or more keys are mapped to the same index and a linked list is used to store these collided keys while in a BST, collisions don't happen as each key is sorted and unique, thus the tree will never have duplicate nodes.

## **Chaining Hash Table vs. AVL Tree:**

Similar to the BST, the AVL tree sorts keys and compares them to traverse the tree.

AVL trees guarantee that the tree remains balanced, unlike BSTs, which could become imbalanced and degrade the search time to O(n).