C950 WGUPS Algorithm Overview

Lydia Husser

ID #00142799

WGU Email: lhusser@wgu.edu

Sept. 12, 2021

C950 Data Structures and Algorithms II

# Introduction

The following paper describes the package routing program that I developed for the WGUPS Routing Program performance assessment. The program takes a list of packages on three trucks, reads package, address, and distance data from CSV files, and finds the optimized route for each truck. An interface allows the user to access statuses for all packages, data for a specific package, and the total delivery time for the trucks.

The program has two major components: a chaining hash table (ChainingHashTable class) and a greedy algorithm (find\_optimized\_route()) . This paper presents my reasoning for choosing the major data structures and algorithms in the program, describes alternative structures, and provides an analysis of the program’s complexity.

# A. Algorithm Identification

The find\_optimized\_route() function is a greedy algorithm that finds the shortest route for a truck based on the distance from one address to the next.

Two strengths of this algorithm are:

1. It only looks at each address once. If multiple packages on a truck go to the same address, duplicate addresses are disregarded and eliminated from the route.
2. It uses data already gathered to get the distance. You don’t have to store distance separately because you can deduce it from the list of package IDs on the truck.

Two other algorithms that would have worked:

1. Sorting algorithm
2. Dijkstra’s algorithm (Lysecky & Vahid, 2018)

# B1. Logic Comments

Pseudocode for find\_optimized\_route(), as referenced from the lesson Tepe, “Getting Greedy” (n.d.)

1. Function input:
   1. truck\_packages list
2. Initialize:
   1. Create package\_data hash table
   2. Initialize optimized\_route list to start at the hub (address ID 1)
   3. Copy truck\_packages list as packages
   4. Initialize empty packages\_on\_truck\_data list
3. For each package on the truck:
   1. Get package data from package\_data
   2. Append data to packages\_on\_truck\_data
4. For each package on the truck:
   1. Get the address for previous\_stop
   2. Find next\_closest\_stop in the copied packages list
      1. Get address for each remaining package
      2. Find the distance between previous\_stop and each address
      3. Return the address closest to previous\_stop
   3. Append optimized\_route with next\_closest\_stop
   4. If a stop has been found for a package:
      1. Remove package from the copied packages list
5. If optimized\_route contains None:
   1. Remove None
6. Append address ID 1 to optimized\_route to end at the hub
7. Return optimized\_route

# B2. Development Environment

This program was written using Python 3.9 in PyCharm 2021.2.1 (Community Edition).

# B3. Space-Time and Big-O

This table shows the complexity analysis for each major section in the program, as well as the total complexity. Calculations are based on Lysecky & Vahid (2018).

|  |  |
| --- | --- |
| **Function** | **Big-O complexity** |
| **main.py** | |
| interface() | O(n) |
| **Total** | O(n) |
| **hashtable.py** | |
| \_\_init\_\_() | O(1) |
| insert() | O(n) |
| search() | O(n) |
| remove() | O(n) |
| **Total** | 3n + 1 = O(n) |
| **distance.py** | |
| get\_distance\_data() | O(1) |
| get\_address\_data() | O(1) |
| distance\_between\_addresses() | O(n) |
| find\_next\_stop() | O(n) |
| find\_optimized\_route() | O(n2) |
| get\_address\_index() | O(n) |
| get\_address\_from\_id() | O(n) |
| get\_total\_distance() | O(n) |
| get\_package\_delivery\_times() | O(n) |
| update\_package\_statuses() | O(n2) |
| **Total** | 2n2 + 6n + 2 = O(n2) |
| **package.py** | |
| create\_package\_hash() | O(n) |
| get\_package\_info() | O(1) |
| **Total** | n + 1 = O(n) |
| **Total program complexity** | 3n + n2 = O(n2) |

# B4. Scalability and Adaptability

This program is scalable for additional trucks and packages. The hash table that creates the package list can be resized as needed. The optimized route algorithm is not dependent on the number of trucks, but calculates a route for each truck separately.

# B5. Software Efficiency and Maintainability

The polynomial complexity of this program means that it is relatively efficient (Lysecky & Vahid, 2018). The program is maintainable because it is organized into intuitive modules and is well-commented.

# B6. Self-Adjusting Data Structures

Strengths of the hash table are that it can be scaled according to the number of packages and it is easy to update and search. A weakness is that the more packages that are added, the longer it will take to build the hash table and to search it (Tepe, “Let’s Go Hashing”, n.d.).

Strengths of the optimized\_truck\_route algorithm are that it adjusts according to the packages on the truck and that it accounts for packages delivered at the same address. A weakness is that it only looks for the next closest distance to the previous stop instead of accounting for all stops, so may not always result in the shortest possible route. (Tepe, “Getting Greedy”, n.d.)

# C. Original Code

I delivered an original program that meets all requirements.

# C1. Identification Information

I added identifying information to the top of main.py.

# C2. Process and Flow Comments

I included comments in my code to explain the process and flow of the program.

# D. Data Structure

I created the self-adjusting ChainingHashTable class to store the package data. The hash table is based on the algorithm in Tepe, “Let’s Go Hashing” (n.d.)

# D1. Explanation of Data Structure

The chaining hash table stores the data for each package object. When all package data has been inserted into the hash table, it is a well-organized, searchable, and updateable data structure.

# E. Hash Table

The ChainingHashTable class fulfils the hash table requirement.

# F. Look-Up Function

The ChainingHashTable search() function takes the package ID as a key and returns all required package components.

# G. Interface

When the program runs, the user is prompted to choose an option:

1. View status of all packages at a specified time
2. View data for a package
3. View total distance traveled by all trucks
4. Exit the program

In PyCharm, the user can input commands using the built-in terminal.

# G1. First Status Check

Screenshot showing status of all packages at 9:00 a.m.

Text

Description automatically generated

# G2. Second Status Check

Screenshot showing status of all packages at 10:00 a.m.

Text

Description automatically generated

# G3. Third Status Check

Screenshot showing status of all packages at 12:30 p.m.

Text

Description automatically generated

# H. Screenshot of Code Execution

Screenshot of successful code completion and total mileage of trucks.

Text

Description automatically generated

# I1. Strengths of Chosen Algorithm

Two strengths of the optimized\_truck\_route() algorithm:

1. It works independently of number of packages on the truck or the number of trucks in operation.
2. It accounts for packages delivered to the same address and adjusts the route so that no address is visited twice by the same truck.

# I2. Verification of Algorithm

The optimized\_truck\_route() algorithm meets all requirements in the scenario.

# I3. Other possible Algorithms

Two other possible algorithms that could be used to find the truck route are:

1. Dijkstra’s algorithm
2. A sorting algorithm (Lysecky & Vahid, 2018)

# I3A. Algorithm Differences

This is how the two algorithms above differ from the optimized\_truck\_route() algorithm.

1. Dijkstra’s algorithm adjusts according to each remaining set of distances instead of just looking at the next closest distance. It would require knowing the package address and distance between each two addresses ahead of time.
2. A sorting algorithm would sort by shortest distance, but would keep duplicate addresses. It would also require knowing the package address ahead of time, but the distance between each two addresses could be calculated as the program runs. (Lysecky & Vahid, 2018)

# J. Different Approach

Some things I would do differently if I did this project again:

* Create a truck object and write the optimized\_truck\_route() algorithm to accept a scalable list of trucks instead of inserting each truck package list and departure time manually.
* Add the address ID as a secondary key in the packages.csv file because matching an address string is imprecise and converting a string to an ID adds to the complexity of the program.
* Build the package hash table just once and update it instead of rebuilding it each time the optimized\_truck\_route() function runs.

# K1. Verification of Data Structure

The ChainingHashTable meets all requirements for the scenario.

# K1A. Efficiency

As the number of packages increases, the search() function will need to perform more comparisons to find a matching key.

# K1B. Overhead

Increasing the number of packages will not increase the number of buckets in the hash table because data in buckets can be chained so that they can hold multiple objects.

# K1C. Implications

Increasing the number of trucks would mean increasing the number of calculations for optimizing truck routes and updating package statuses. Increasing the number of cities would require more careful sorting of packages onto trucks to ensure trucks stay under the maximum mileage.

# K2. Other Data Structures

Two other data structures that could meet the requirements for the hash table in this scenario are:

1. A dictionary
2. A heap (Lysecky & Vahid, 2018)

# K2a. Data Structure Differences

The two algorithms above differ from a hash table in the following ways:

1. A dictionary would mean the package data is more immediately accessible. A package object wouldn’t need to be formatted and converted to readable text before you can use individual components of it.
2. A heap would still use package objects, but could prioritize them based on package ID. This could be useful for scaling if there were more trucks and cities involved and package IDs were assigned based on city of delivery. (Lysecky & Vahid, 2018)

# M. Professional Communication

Professional communication is demonstrated in the content and presentation of this project.

# L. Sources - Works Cited

Lysecky, R., & Vahid, F. (2018, June). *C949: Data Structures and Algorithms I*. zyBooks.

Retrieved September 12, 2021, from

<https://learn.zybooks.com/zybook/WGUC9492018/>

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

Retrieved September 12, 2021, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>

Tepe, C. “C950 – Webinar-1 – Let’s Go Hashing”. C950 Supplemental Material.

Retrieved September 12, 2021, from <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71>

Tepe, C. “C950 – Webinar-2 – Getting Greedy, who moved my data”. C950 Supplemental Material.

Retrieved September 12, 2021, from <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=eee77a88-4de8-4d42-a3c3-ac8000ece256>