C950 WGUPS Algorithm Overview

Daniel Crites

ID #002152169

WGU Email: dcrite5@my.wgu.edu

Date

C950 Data Structures and Algorithms II

# Introduction

Text goes here

# A. Algorithm Identification

The algorithm that I used to delivery packages is the greedy algorithm. I use the min function in python to find the closest route and through-out of the delivery process. As packages are delivered, they are removed from a list until no packages is left. Once that happens the truck returns to the Hub from last delivery.

# B1. Logic Comments

Trucks are manually loaded and assign to a specific truck. Once loaded the trucks begin their deliveries. The following pseudocode demonstrates how I used the greedy algorithm.

loadTruck1 = manually load packages

loadTruck2 = manually load packages

loadTruck3 = manually load packages

total\_mileage = 0

delivery\_process(truck)

packages = truck.packages

current\_position = 0

mileage\_list = mileage distances for current locations

duplicate\_list = handles duplicate distances on route

while len(packages) > 0  
 shortest\_route = min(mileage)

total\_mileage += shortest\_route

duplicate\_list.determine the right package that needs to be delivered

update.packages

update.current\_position

remove(package) from packages

return truck mileage

# B2. Development Environment

This program used Python 3.10 and Pycharm for the IDE. The OS use was Windows 11. GitHub was used as the repository. This allowed me to work on the project on two different machines, my laptop and desktop.

# B3. Space-Time and Big-O

HashTable.py

def \_\_init\_\_() = Space-time complexity O(1)

def insert() = Space-time complexity O(n)

def search() = Space-time complexity O(n)

def remove() = Space-time complexity O(n)

Main.py

def address\_list() = Space-time complexity O(n)

def package\_list() = Space-time complexity O(n)

def distance\_list() Space-time complexity O(n)

def load\_package\_data() = Space-time complexity O(1)

def delivery\_process() = Space-time complexity O(n^2)

def remove\_delivered\_package = Space-time complexity O(n)

def find\_indices() = Space-time complexity O(n)

def ui() = Space-time complexity O(1)

# 4. Scalability and Adaptability

For larger deliveries this program is scalable due to it’s ability to handle duplicate values of mileage and determine how to update those packages. Its space-time complexity would change much for larger package handle. The program’s adaptability could be improved by creating a more specific hash table, the current hash table is limited on what it can do.

# B5. Software Efficiency and Maintainability

This software is efficient since majority of its spacetime complexity is O(n) or O(1). The program should be easily maintained since there are not a lot of code to source through. This makes it easy to quickly find classes, functions, variables, etc, and alter as needed.

# B6. Self-Adjusting Data Structures

Self-adjusting data structures are beneficial in that they can handle large and small amounts of inputs and account for pattern changes. Self-adjusting data structures can be easy to read and increase updates' efficiency. One of the weaknesses of self-adjusting data structures would be data collisions. With these collisions happening, the cost and preventing can be complex and increase time-complexity.

# D1. Explanation of Data Structure

I used the chaining hash table from zyBooks 7.8.2. The hash table has the initializing capacity to control the data it can use. The insert function allows one to input new data or update existing data. The search function uses a Key ID to look up packages and returns the information associated with the Key ID. The remove function removes data from the hash table by inputting the Key ID.

# E. Hash Table

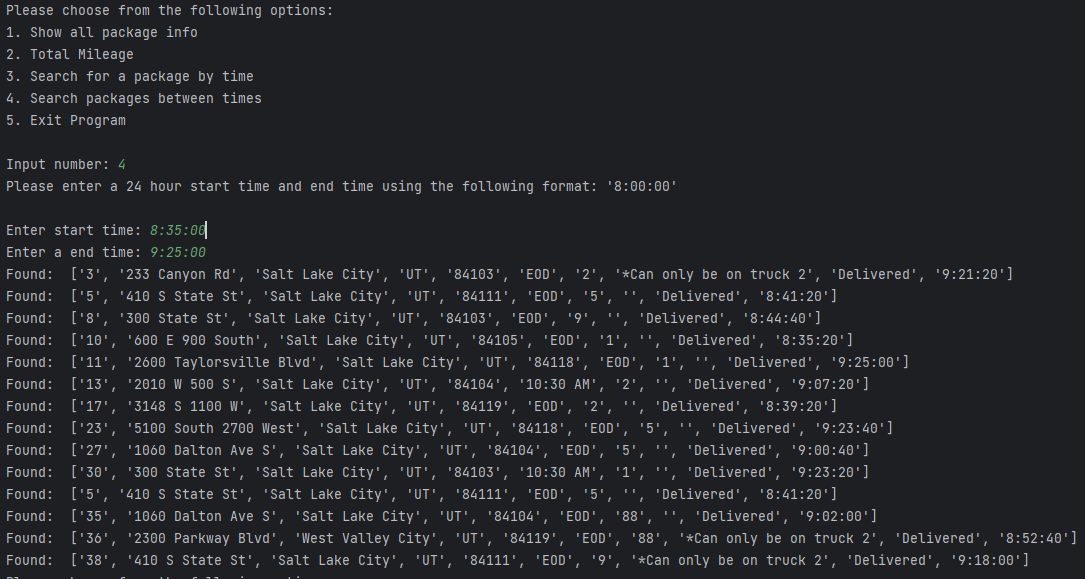


# F. Look-Up Function

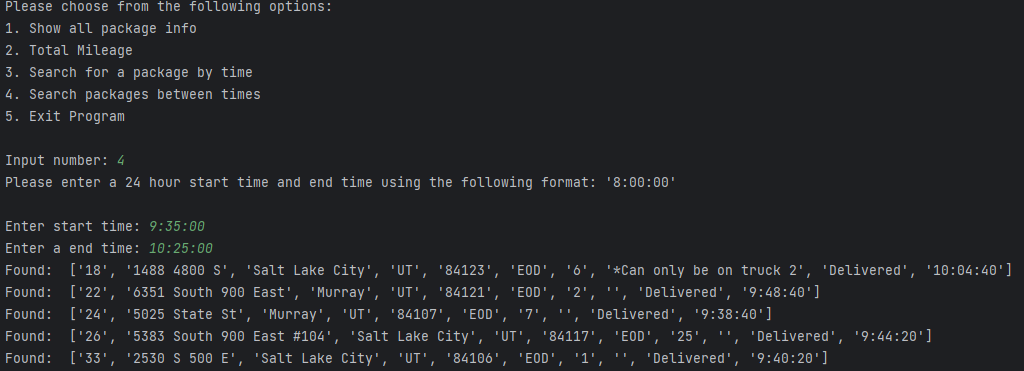


# G. Interface

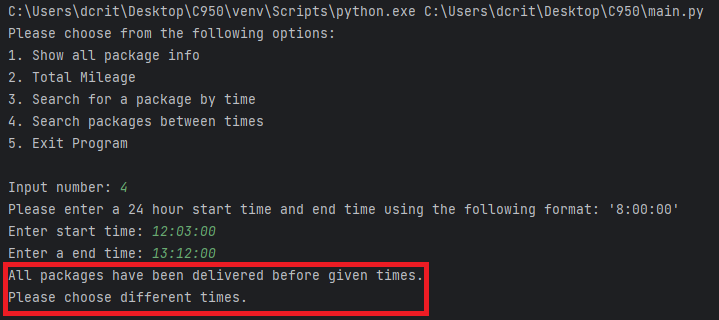
# G1. First Status Check



# G2. Second Status Check



# G3. Third Status Check



# H. Screenshots of Code Execution

A screenshot of a computer program

Description automatically generated

# I1. Strengths of Chosen Algorithm

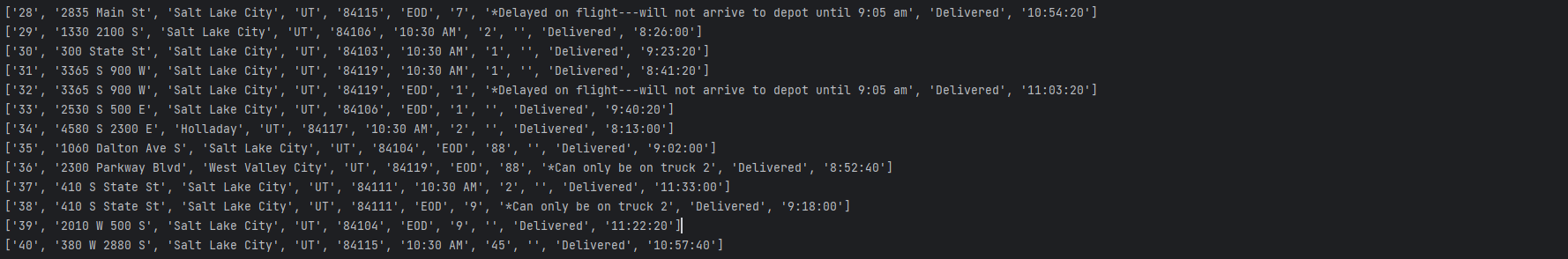
I used the greedy algorithm to complete the project because of its simplicity. It finds the closest route from each location until deliveries are completed. The greedy algorithm scales well with a large or small amount of data. The efficiency is usually constant and makes it easier to predict space-time complexity.

# I2. Verification of Algorithm

My final mileage was 124.7, which is below the 140 mileage requirement. Packages were manually loaded based on the requirements and delivered on time.

A screen shot of a computer

Description automatically generated



# I3. Other possible Algorithms

Brute-force search algorithm

Farthest Insertion algorithm

# I3A. Algorithm Differences

One of the algorithms that I would use would be a brute-force search algorithm. This could be an optimal algorithm because of the increase in hardware performance. The faster the computation, the less time it takes to complete. However, increased electricity consumption would be expected, and vast amounts of data still could take some time to compute. The other algorithm I would like to use would be the Farthest Insertion. This algorithm finds the farthest point from the starting location and repeatedly finds a point where it causes the smallest increase on the route.

# J. Different Approach

The first thing I would change would be to develop a function that automatically sorts packages in trucks and does not need to load them manually. I would also change my delivery algorithm to the brute-force search algorithm. I believe that it would be the most optimal solution.

# K1. Verification of Data Structure

The total distance is under 140 miles. All packages were delivered on time and to their specifications. A hash table with a look-up function was used. The user interface reports all packages, and the information is correct.

A screen shot of a computer

Description automatically generated

A black background with many small colored lines

Description automatically generated with medium confidence

A screen shot of a computer program

Description automatically generated

# K1A. Efficiency

The lookup function has a space-time complexity of O(n). Since n is constant, a large data set would take more time to search the entirety of the list. The more packages, the longer the list becomes and the longer it takes to search it.

# K1B. Overhead

The space usage is only O(1) and would be consistent regardless of how many packages are added. This means the algorithm will continuously allocate memory at a constant of 1. The overhead is low for the given scenario.

# K1C. Implications

Adding additional trucks and cities would increase space and time usage. Since the look-up function is O(n), the larger the inputs, the more time it takes to find them. However, this would be an acceptable scalability feature that keeps time and space complexity reasonably low.

# K2. Other Data Structures

Binary Tree

Linked List

# K2a. Data Structure Differences

The binary tree data structure would work well with efficiency and searching of large amount of data since there can only be at most two nodes. The scalability would better than the hash table I used for the project. The other data structure I would use would be a linked list. Linked lists allow for easy insertion and removal of data because the nodes are linked together. Linked list doesn’t require a fixed size or initial size. This would also be a better scalability option than hash table.

# M. Professional Communication

None

# L. Sources - Works Cited

Text goes here

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

Retrieved March 22, 2021, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>

<https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/>

<https://www.baeldung.com/cs/tsp-exact-solutions-vs-heuristic-vs-approximation-algorithms>