C950 WGUPS Algorithm Overview

Kenna Heng

ID #011050292

WGU Email: kheng@wgu.edu

Date: 9/4/23

C950 Data Structures and Algorithms II

# Introduction

For the project of delivering 40 packages on time, under a certain mileage, and with some constraints: I have decided to use a greedy algorithm of The Nearest Neighbor. I was able to accomplish all the above task while having a worst-case time complexity of O(n^2). I will now go over each part of the code, data structure and hash map I used.

# A. Algorithm Identification

In this project, I have employed the "Nearest Neighbor algorithm" as the self-adjusting core algorithm to efficiently route and deliver packages while meeting the specified requirements outlined in the scenario. The self-adjusting aspect of the algorithm comes into play when the program dynamically evaluates and selects the nearest delivery location based on the current location of the delivery truck. This algorithm allows the program to adapt to changing conditions, such as the address of the packages and the location of the truck, ensuring that the most optimal route is determined dynamically during the delivery process.

# B1. Logic Comments

# Start of Truck Class

Class Truck:

Initialize truck with a list of packages, current time, and user-defined time

Initialize milesDriven, currentLocation, endTime, backHomeStatus, and userTime

Method startDelivery():

# Mark all packages in load as en route

For each package in packagesInLoad:

Set package.enrouteTime = currentTime

# Create a list of unvisited delivery locations

unvisited = List of addresses in packagesInLoad

# Using the Nearest Neighbor algorithm

While unvisited is not empty:

Initialize min\_distance to infinity

Initialize nearestStreet to None

# Find the nearest delivery location

For each address in unvisited:

Calculate the distance to address from currentLocation

If distance is less than min\_distance:

Update min\_distance and nearestStreet

# Update current location, unvisited list, and miles driven

Set currentLocation to nearestStreet

Remove nearestStreet from unvisited

Add min\_distance to milesDriven

Update currentTime based on travel time

# Deliver packages at the current location

For each package in packagesInLoad:

If package.getAddress() is equal to currentLocation:

Set package.deliveryTime = currentTime

Remove package from packagesInLoad

# Return to the hub once all packages are delivered

If packagesInLoad is empty:

Set currentLocation to hub

Calculate distance to hub and add to milesDriven

Set endTime and backHomeStatus to indicate completion

# End of Truck Class

This pseudocode provides a step-by-step outline of how the program finds a solution using the Nearest Neighbor algorithm to optimize the delivery route while dynamically adjusting to each package’s location. The algorithm's logic adapts to the real-time status of packages and the truck's current location during the delivery process.

Top of Form

Bottom of Form

# B2. Development Environment

I am using Windows 11 as the operating system. I am using Visual Studio Code 2023 as an IDE with the newest version of Python (v 3.11.4) as an extension. My laptop is an ASUS VivoBook with an Intel Core i5 CPU and an integrated GPU. More than enough to complete this assignment.

# B3. Space-Time and Big-O

I have included comments for each method in the program that specifies what it does and it’s time complexity. I will add the most complex method in the program here: The nearest neighbor algorithm inside of the truck class.

#Time complexity of this method is O(n^2)

    def startDelivery(self):

        #Mark all packages in load enroute time, O(n)

        for package in self.packagesInLoad:

            package.enrouteTime = self.currentTime

        # unvisited is a list of places where we need to drop off packages, O(n)

        unvisited = []

        for package in self.packagesInLoad:

            unvisited.append(package.getAddress())

        # Using nearest neighbor greedy algorithm, we visit the closest location from our current location

        # Loop through unvisited places as long as it's not empty, O(n^2) due to there being another for loop

        # In the distanceToAddress method which also has a for loop.

        while unvisited:

            min\_distance = float('inf')

            nearestStreet = None

            #O(n), but since distanceTo address is also O(n), makes this method O(n^2)

            for address in unvisited:

                #O(n)

                distance = distanceToAddress(self.currentLocation, address)

                if distance < min\_distance:

                    min\_distance = distance

                    nearestStreet = address

            #Update current location, unvisited list, and miles driven

            self.currentLocation = nearestStreet

            unvisited.remove(nearestStreet)

            self.milesDriven += min\_distance

            self.currentTime += datetime.timedelta(hours=min\_distance / 18)

            # Drop off packages if they match our current location

            for package in self.packagesInLoad:

                if package.getAddress() == self.currentLocation:

                    package.deliveryTime = self.currentTime

                    self.packagesInLoad.remove(package)  # Remove package from list

        #Go back home once load is empty

        if len(self.packagesInLoad) == 0:

            home = addressList[0]

            backHome = distanceToAddress(self.currentLocation, home)

            self.currentLocation = home

            self.milesDriven += backHome

            self.endTime = self.currentTime + datetime.timedelta(hours=backHome / 18)

            self.backHomeStatus = True

#Method to determine distance between two addresses. Uses two indexes to get a value in a

#Triangular array. O(n)

def distanceToAddress(address1, address2):

    indexAddress1 = 0

    indexAddress2 = 0

    #get the indexes for both addresses

    for i in range(len(addressList)):

        if address1 == addressList[i]:

            indexAddress1 = i

    for j in range(len(addressList)):

        if address2 == addressList[j]:

            indexAddress2 = j

    #Sets the distance using indexs in distanceList

    distance = distanceList[indexAddress1][indexAddress2]

    #Since it's a triangle array, will switch indexes if = ""

    if distance == "":

        distance = distanceList[indexAddress2][indexAddress1]

    #Return a float value

    fltDistance = float(distance)

    return fltDistance

# B4. Scalability and Adaptability

The application I've developed for package delivery demonstrates scalability and adaptability in handling an increasing number of packages. Although, the packages must be pre-loaded into the trucks since there’s no package sorting algorithm in place. The addresses in relation to all the current addresses would also need to be updated in the corresponding csv files, if adding new packages with new addresses. The scalability of the program might be affected if the number of packages becomes exceedingly large, as the algorithm's time complexity may introduce overhead.

# B5. Software Efficiency and Maintainability

The software developed for the package delivery system is designed with both efficiency and maintainability in mind. In terms of efficiency, the overall Big-O time complexity of the program is primarily driven by the Nearest Neighbor algorithm, resulting in a time complexity of O(n^2) for the worst case, where n represents the number of packages. While this time complexity is acceptable for the expected scale of operations, further optimization could be considered for handling extremely large datasets.

For maintainability, the codebase is structured in a clear and organized manner, with meaningful variable and function names, making it easy for future developers to understand. Additionally, detailed comments are included throughout the code to explain the logic, algorithms, and key decision points. The code is compartmentalized into distinct classes (Truck and Package), each with its responsibilities, enhancing code modularity and making it more straightforward to identify and fix issues or add enhancements. These practices contribute to the maintainability of the software, ensuring that it remains comprehensible and adaptable for future development and maintenance tasks.

# B6. Self-Adjusting Data Structures

**Strengths:**

1. **Fast Key-Value Retrieval:** The hash table provides rapid key-value retrieval, making it efficient for accessing package data by package ID. This efficiency is crucial for quick look-ups and updates of package information, contributing to the overall speed of the delivery system.
2. **Optimal Space Utilization:** Hash tables typically offer excellent space utilization, ensuring that memory is used efficiently. The use of hash functions and buckets allows for the distribution of data, reducing collisions and minimizing wasted memory.

**Weaknesses:**

1. **Limited Sorting Capabilities:** Hash tables are not designed for sorting data based on keys, which can be a limitation when specific sorting requirements are needed. Sorting or iterating through the entire data set can be less efficient in hash tables compared to other data structures like balanced trees.
2. **Collisions Impact Performance:** While hash functions aim to distribute data evenly across buckets, collisions can still occur. Collisions require additional processing to resolve, which can impact performance when dealing with a large number of packages. In extreme cases, collisions can degrade the time complexity from O(1) to O(n). Highly likely since I only implemented the key to match the package ID. If in the future ID’s are reused, will cause issues in the future.

# C. Original Code

My code is original. It’s made using the latest version of python and the popular Visual Studio Code 2023 IDE.

# C1. Identification Information

The first three lines and comments include my name, student ID, and class information

# C2. Process and Flow Comments

There is comments for every block of code and method. For all the methods, the time complexity is also included. Each comment specifies what the below code block or method does.

# D. Data Structure

My Hash map code. It can adjust to any new packages added to the list:

from Package import Package

class HashMap:

def \_\_init\_\_(self, packageList):

self.packageList = packageList

self.buckets = [[] for \_ in range(len(self.packageList))]

for package in packageList:

key = package.package\_id

value = package

self.add(key, value)

# returns a hash code, O(1)

def \_hash(self, key):

return hash(key) % len(self.packageList)

# adds a key to the hashmap, O(n)

def add(self, key, value):

hash\_key = self.\_hash(key)

bucket = self.buckets[hash\_key]

for i, (existing\_key, \_) in enumerate(bucket):

if existing\_key == key:

bucket[i] = (key, value)

return

bucket.append((key, value))

# returns a value from a key, O(n)

def get(self, key):

hash\_key = self.\_hash(key)

bucket = self.buckets[hash\_key]

for existing\_key, value in bucket:

if existing\_key == key:

return print(value.packageInfo())

return print("Package not found")

# deletes a key and value, O(n)

def delete(self, key):

hash\_key = self.\_hash(key)

bucket = self.buckets[hash\_key]

for i, (existing\_key, \_) in enumerate(bucket):

if existing\_key == key:

del bucket[i]

return

# returns all keys and values, O(n^2)

def get\_all\_Packages(self):

for bucket in self.buckets:

for key, package in bucket:

print(package.packageInfo())

# D1. Explanation of Data Structure

The implemented hash table above, efficiently stores and retrieves package information using package IDs as keys and the package object itself as a value. The logic behind the hash table involves generating a hash code for each package ID through the \_hash method, which maps the key to a specific index in the table. This approach allows for constant time complexity O(1) in calculating the hash code, making it a more efficient retrieval method compared to a simple linear search that iterates though all values in it.

# E. Hash Table

The insertion function in the hash map above. Has a complexity of O(n) to make sure that the key doesn’t already exist:

# adds a key to the hashmap, O(n)

def add(self, key, value):

hash\_key = self.\_hash(key)

bucket = self.buckets[hash\_key]

for i, (existing\_key, \_) in enumerate(bucket):

if existing\_key == key:

bucket[i] = (key, value)

return

bucket.append((key, value))

# F. Look-Up Function

The get method returns a value from the key the user calls:

# returns a value from a key, O(n)

def get(self, key):

hash\_key = self.\_hash(key)

bucket = self.buckets[hash\_key]

for existing\_key, value in bucket:

if existing\_key == key:

return print(value.packageInfo())

return print("Package not found")

# G. Interface

The code for the interface, located in the main.py file. Starts by running the algorithm with the time all trucks are back at base and all packages are delivered. The total mile is returned. Then the user is asked for two inputs, what time they want to view the status of a package(s). And if they want to view a single package or all the packages:

# Name: Kenna Heng

# Student ID: 011050292

# Class: C950

from PackageLoader import packageReader

from HashMap import HashMap

from Truck import Truck

from Package import Package

import datetime

# Load data and populate the hashmap

file = 'WGUPS Package File.csv'

PackageInfo = packageReader(file)

hashMap = HashMap(PackageInfo)

#Opening message

print("WGU Parcel Service")

#Time where route is finished

userTime = datetime.timedelta(hours=13, minutes=12)

#Create 3 trucks with packages loaded

truck1 = Truck ([PackageInfo[0], PackageInfo[12], PackageInfo[13], PackageInfo[14], PackageInfo[15], PackageInfo[19], PackageInfo[28], PackageInfo[29], PackageInfo[30], PackageInfo[33], PackageInfo[36], PackageInfo[39]], datetime.timedelta(hours=8), userTime)

truck2 = Truck ([PackageInfo[2], PackageInfo[5], PackageInfo[11], PackageInfo[16], PackageInfo[17], PackageInfo[18], PackageInfo[20], PackageInfo[21], PackageInfo[22], PackageInfo[23], PackageInfo[25], PackageInfo[26], PackageInfo[34], PackageInfo[35], PackageInfo[37], PackageInfo[38]], datetime.timedelta(hours=10, minutes=20), userTime)

truck3 = Truck ([PackageInfo[1], PackageInfo[3], PackageInfo[4], PackageInfo[5], PackageInfo[6], PackageInfo[7], PackageInfo[8], PackageInfo[9], PackageInfo[10], PackageInfo[24], PackageInfo[27], PackageInfo[31], PackageInfo[32]], datetime.timedelta(hours=9, minutes=5), userTime)

#Start Truck 1

if truck1.currentTime < userTime:

truck1.startDelivery()

#Start truck 2 at this time

if truck2.currentTime < userTime:

truck2.startDelivery()

#Change package info for package 9 @10:20am

PackageInfo[8].address = "410 S State St"

PackageInfo[8].zip\_code = "84111"

#Start truck 3 when either truck 1 or 2 make it back to base

if truck1.backHomeStatus == True:

truck3.currentTime = truck1.endTime

if truck3.currentTime < userTime: # type: ignore

truck3.startDelivery()

elif truck2.backHomeStatus == True:

truck3.currentTime = truck2.endTime

if truck3.currentTime < userTime: # type: ignore

truck3.startDelivery()

#Update all packages with status of final delivery time

for i in range(len(PackageInfo)):

PackageInfo[i].updateStatus(userTime)

#Output the total miles driven

print(f"Total miles of route: {round(truck1.milesDriven + truck2.milesDriven + truck3.milesDriven, 2)} miles")

#Input time to see status of package

try:

# Input time to see status of package

userInput = input("Please enter a time to check the status of package(s). Please use HH:MM military time format. Invalid entry will exit program: ")

(h, m) = userInput.split(":")

userTimeInput = datetime.timedelta(hours=int(h), minutes=int(m))

#Package 9 address is different depending on time

if userTimeInput < datetime.timedelta(hours=10, minutes=20):

PackageInfo[8].address = "300 State St"

PackageInfo[8].zip\_code = "84103"

#Update all packages with status of inputted time

for i in range(len(PackageInfo)):

PackageInfo[i].updateStatus(userTimeInput)

# Ask to see status of single or all packages

userInputSingleOrAll = input("If you would like to see the status of one package, please enter 'single', or enter 'all'. Invalid entry will exit program: ")

if userInputSingleOrAll == "all":

for i in range(len(PackageInfo)):

print(PackageInfo[i].packageInfo())

elif userInputSingleOrAll == "single":

userSingleLookUp = int(input("Enter package ID: ")) - 1

print(PackageInfo[userSingleLookUp].packageInfo())

else:

print("Invalid input. Please enter 'single' or 'all'.")

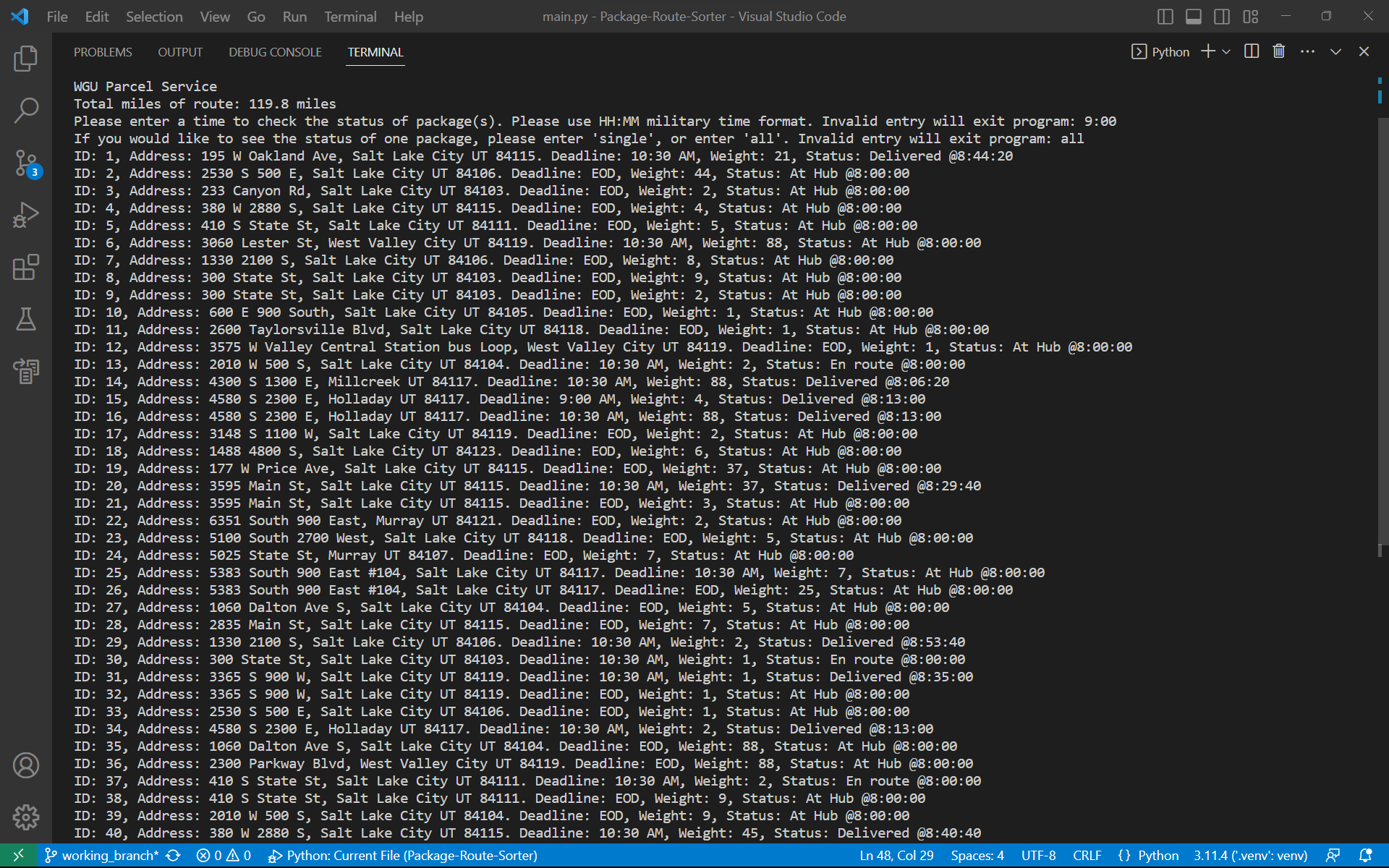
except ValueError as e:

print("Invalid input format. Please use HH:MM military time format for time input.")

print("Exiting program...")

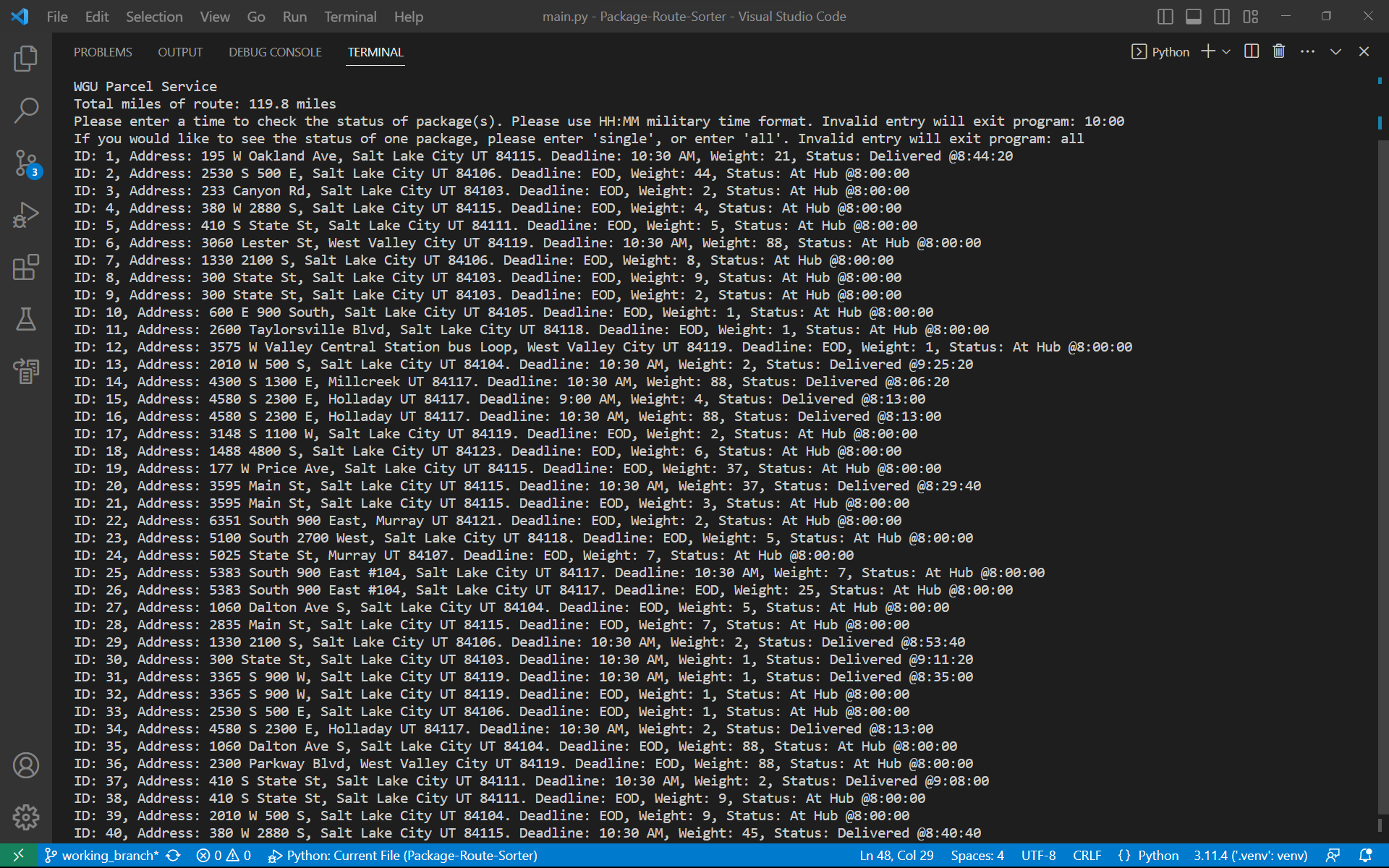
# G1. First Status Check

9AM status check:



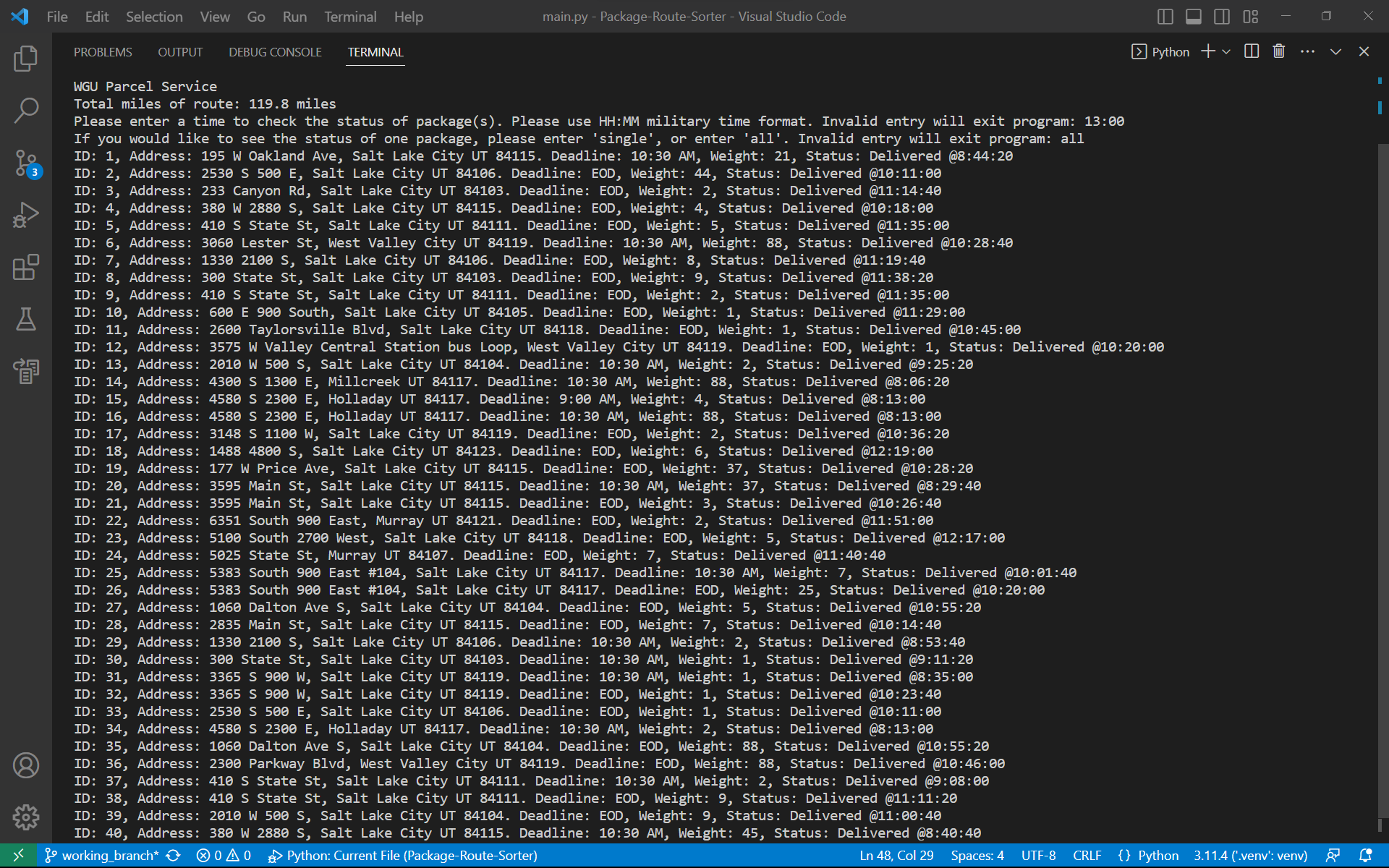
# G2. Second Status Check

10AM status check:



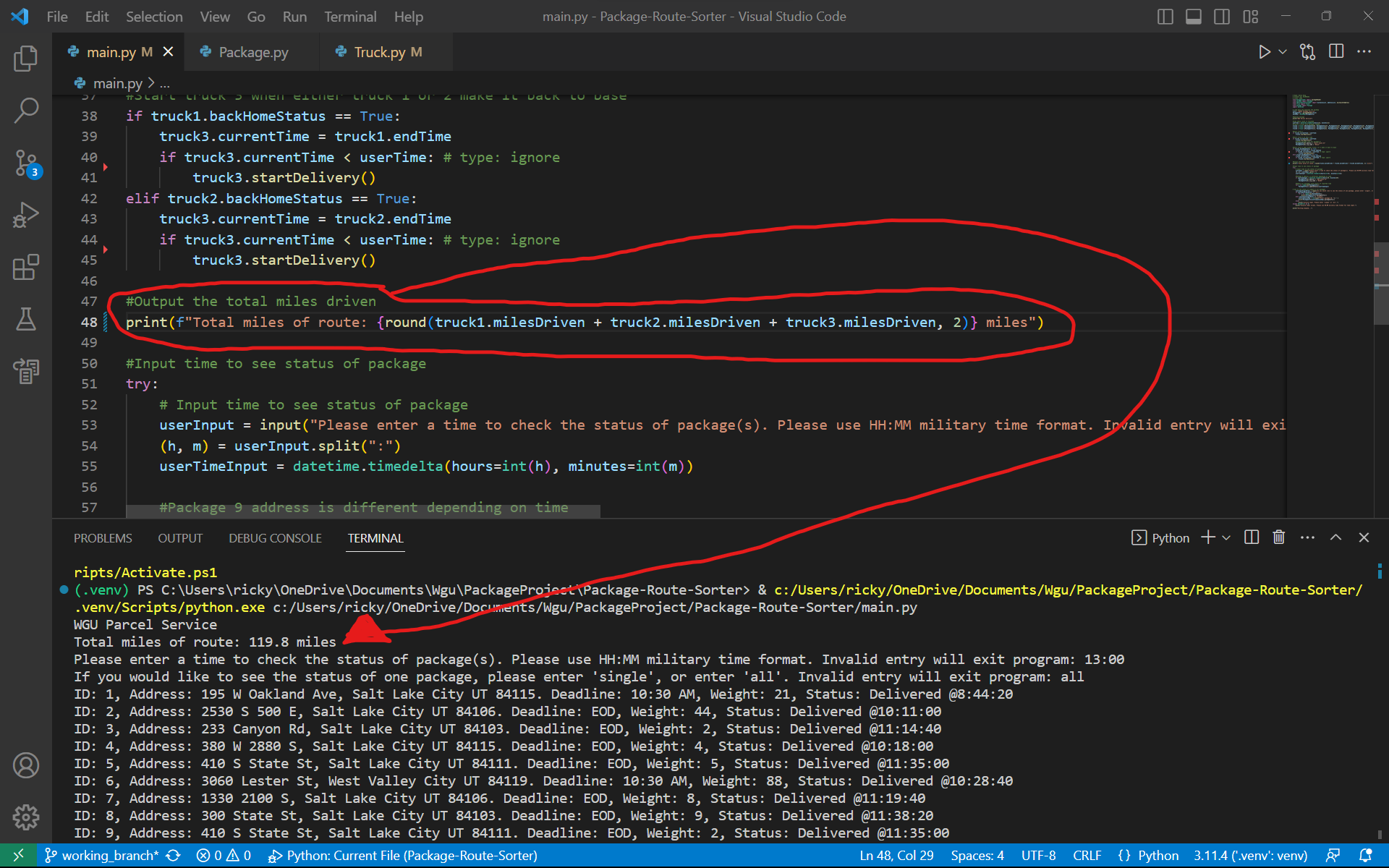
# G3. Third Status Check

1PM status check:



# H. Screenshots of Code Execution

Part of the code that returns the miles of all trucks after dropping off all packages. It points to where it shows in the console. There are zero errors shown in the console and IDE:



# I1. Strengths of Chosen Algorithm

Two specific strengths of the Nearest Neighbor algorithm used in the project are its efficiency and adaptability. Firstly, its efficiency lies in its ability to quickly determine an optimal delivery route by selecting the nearest delivery location, reducing the overall travel distance. This efficiency is crucial in ensuring that packages are delivered within the specified time frame while keeping the total distance traveled by the trucks under control. Secondly, its adaptability is a strength relevant to this project as it dynamically adjusts the delivery route based on the locations of packages and the truck. This adaptability ensures that the program can handle varying package scenarios and changes in delivery requirements, making it well-suited for the dynamic nature of package delivery.

# I2. Verification of Algorithm

As you can see from the above screenshots. Total miles traveled by all three trucks is about 120 miles. All packages were delivered by their deadline. All packages that had special instructions, especially package number 9, were delivered correctly. The can verify with the UI because the deadline and delivery time is also provided when you look up a specific package.

# I3. Other possible Algorithms

1. **Genetic Algorithm**: Genetic algorithms are a powerful optimization technique inspired by the process of natural selection. In the context of package delivery, this algorithm could be applied to evolve and improve delivery routes over time. Initially, a population of random delivery routes is generated. These routes are then evaluated based on criteria such as total distance, delivery time adherence, and truck utilization. Routes that perform better are selected and combined (crossover operation) to create new routes, introducing genetic diversity. Random mutations are also applied to some routes. This process continues for multiple generations, gradually refining the delivery routes. Genetic algorithms can handle complex scenarios and are known for their ability to find near-optimal solutions. However, they may require more computational resources compared to simpler algorithms.
2. **Ant Colony Optimization (ACO)**: ACO is another nature-inspired algorithm that can be applied to solve the Traveling Salesperson Problem efficiently. In this algorithm, artificial ants are used to explore and optimize delivery routes. Initially, ants are placed at the depot (hub), and they traverse the streets, dropping pheromones along their paths. Ants make decisions based on a combination of pheromone levels and a heuristic function that estimates the quality of the path. Over time, paths with higher pheromone levels become more attractive to other ants, and the algorithm converges toward an optimal or near-optimal solution. ACO can adapt to changing conditions and is suitable for scenarios where routes need to be continually optimized. However, it may require fine-tuning of parameters and a larger number of iterations to converge to the optimal solution. (Rahman, 2020)

# I3A. Algorithm Differences

The Genetic Algorithm and Ant Colony Optimization (ACO) present distinctive attributes when compared to the Nearest Neighbor algorithm identified in Part A. One notable difference is their approach to optimization. While the Nearest Neighbor algorithm relies on a greedy strategy, selecting the nearest delivery point at each step, Genetic Algorithms and ACO take a more evolutionary approach. Genetic Algorithms use a population-based method, evolving multiple routes over generations, which can lead to a broader exploration of possible solutions and potentially find a global optimum. ACO, on the other hand, relies on the collective behavior of artificial ants and pheromone trails to guide route selection, making it a more collaborative and adaptable approach. Additionally, both Genetic Algorithms and ACO have the advantage of adaptability to changing conditions, making them well-suited for scenarios where routes need to be continually optimized. However, these methods may require more computational resources and parameter tuning compared to what’s needed for this small parcel delivery company at the moment.

# J. Different Approach

In the future, when I have more experience and become a seasoned programmer, I would like to add a package sorter algorithm to load the trucks based on the constraints of the package. As the code is now, trucks must be manually loaded to handle the packages with special instructions and to keep the miles below 140. I also would like to change the hash map to better create hashes for packages and store the information. Collision could be handled better, and the time complexities could be lowered for the hash map.

# K1. Verification of Data Structure

The data structure I used meets all requirements. The combined miles of all the trucks is 120 miles. All packages are delivered on time. Packages are delivered according to their special instructions, if any. The hash table I used has a lookup function. The user interface is able to return information of a single package or all of them.

# K1A. Efficiency

My hash table look up function has a time complexity of O(n). Which means the more packages added, the more time it will take to perform this function. It scales linearly.

# K1B. Overhead

Adding more packages will also affect data storage. Each key corresponds to a package ID. And since they won’t be reused, will take up a lot of data if the company scales dramatically in the future.

# K1C. Implications

Adding more trucks and cities would mean updating the address csv and each new location would need to be related to all existing locations. Since must of the hash maps functions scale linearly, the more variables would increase the stress placed on the algorithm.

# K2. Other Data Structures

A Binary Search Tree could have been a viable alternative due to its ability to maintain data in sorted order, which could facilitate efficient search operations. Each package ID could serve as the key, allowing for quick retrieval of package information with an average time complexity of O(log n). However, the main drawback of BSTs is that they may not be as efficient as hash tables for scenarios with a large number of packages due to potential imbalance issues that can lead to a worst-case time complexity of O(n). Nevertheless, for relatively smaller datasets, BSTs can offer a straightforward and organized way to store and retrieve package information.

Linked Lists could also be considered as an alternative data structure. Each node in the linked list could represent a package, and the list could be sorted based on package IDs for efficient search. While linked lists may not offer the same level of search efficiency as hash tables or BSTs, they have their advantages, such as simplicity and lower memory overhead. They could be particularly useful in scenarios where memory resources are limited or when the dataset is relatively small. However, their search time complexity is O(n), which may not be ideal for larger datasets.

# K2a. Data Structure Differences

Comparing Binary Search Trees (BSTs) and Linked Lists to the hash table used in Part D, there are notable differences in terms of their attributes. Firstly, in terms of search efficiency, the hash table, with its constant-time average search complexity (O(1)), outperforms both BSTs (average O(log n)) and Linked Lists (O(n)). This is a significant advantage for quick retrieval of package information, especially when dealing with a large number of packages. Secondly, in terms of memory overhead, Linked Lists have an advantage over both BSTs and hash tables. Linked Lists use memory more efficiently as they only allocate memory for each individual package, while hash tables and BSTs require additional memory for data structures such as buckets or nodes. However, Linked Lists sacrifice search efficiency for this lower memory footprint, which may not be suitable for larger datasets or scenarios with frequent data lookups. Overall, the choice of data structure depends on the specific project requirements, considering factors like dataset size, memory constraints, and desired search efficiency.

# M. Professional Communication

I believe my content to be organized and in line with the rubric and task instructions. Please feel free to reach me to discuss and criticisms and changes I can work on to better improve this assignment and myself going forward.

# L. Sources - Works Cited

Rahman, A. (2020, April). *Introduction to Ant colony optimization(ACO).* Towards Data Science

Retrieved September 5, 2023, from

<https://towardsdatascience.com/the-inspiration-of-an-ant-colony-optimization-f377568ea03f>