

C950 WGUPS Algorithm Overview

Andrew Duliba

ID #003300689

WGU Email: aduliba@wgu.edu

Date 06/02/2023

C950 Data Structures and Algorithms II

Introduction

The following rubric will cover all of my documentation and answering of questions and requirements for the NHP2 — NHP2 Task 1: WGUPS Routing Program.

A. Algorithm Identification

I choose the Nearest Neighbor algorithm for my program. The nearest neighbor algorithm loops over a list of possible destinations (packages) and selects the one that is closest to the current location. The selected package becomes the next destination and is removed from the list of possible destinations. The algorithm repeats this loop until the list of possible destinations (packages) is empty.

B1. Logic Comments

Algorithm Overview for Package Delivery System:

Stated Problem: The purpose of this project is to create a package delivery system that utilizes an algorithm in the Python programming language to track the status of packages and calculate the total mileage of the cars delivering those packages. The algorithm should be able to manage package information and optimize delivery routes for three different trucks. The program should also be able to read in-package data from CSV files and implement the algorithm of choice to strategically place packages on each truck.

Algorithm Overview: The package delivery system algorithm is implemented as follows:

1. Read and Assign CSV Files:
2. Load Package Data into Hash Table:
3. Calculate Distance between Addresses:
4. Extract Address Number from Address String:
5. Create Truck Objects:
6. Create Hash Table for Packages:
7. Load Package Data into Hash Table:
8. Order Packages on Trucks:
9. Load Trucks:
10. Execute program

PROCEDURE deliveringPackages(trucks):

notDelivered EQUALS []

FOR packageID in trucks.packages:

package EQUALS packageHashTable.search(packageID)

notDelivered.append(package)

trucks.packages.clear()

WHILE len(notDelivered) GREATER THAN 0:

nextAddress EQUALS 2000

nextPackage EQUALS None

FOR package IN notDelivered:

IF distance EQUALS distanceInBetween(extractAddress(trucks.address),
extractAddress(package.address)) LESS THAN OR EQUAL TO nextAddress:

nextAddress EQUALS distance

nextPackage EQUALS package

trucks.packages.append(nextPackage.ID)

notDelivered.remove(nextPackage)

trucks.mileage ADDITION ASSIGNMENT nextAddress

trucks.address EQUALS nextPackage.address

trucks.time ADDITION ASSIGNMENT datetime.timedelta(hours EQUALS nextAddress
DIVIDED BY 18)

nextPackage.deliveryTime EQUALS trucks.time

nextPackage.departureTime EQUALS trucks.departureTime

B2. Development Environment

Hardware used: Desktop computer with the following specifications

- OS: Microsoft Windows 10 Home
- Version: 10.0.19045 Build 19045
- Memory: 1terabyte SSD
- Processor: AMD Ryzen 5 3600 6-Core
- BIOS - UEFI ASRock B450M-HDV R4.0

Software used: PyCharm Community Edition 2023.1

Python Version used: Python 3.11

The programming environment that I used for this application is PyCharm through IntelliJ IDEA. PyCharm is an Integrated Development Environment (IDE) specifically designed for Python development. I was familiar with using other IntelliJ products like JetBrains, so using PyCharm was quick and easy to learn. Additionally, I was prompted by teachers and the class study guide to go with this option for a coding environment.

B3. Space-Time and Big O

```
def loadPackageData(fileName, packageHashTable):  
  
    with open(fileName) as packageInfo:  
  
        packageData = csv.reader(packageInfo)  
  
        for package in packageData:  
  
            pID = int(package[0])  
  
            pAddress = package[1]  
  
            pCity = package[2]  
  
            pState = package[3]  
  
            pZipcode = package[4]  
  
            pDeadline = package[5]  
  
            pWeight = package[6]  
  
            pStatus = "A Hub"
```

```
        p = Package(pID, pAddress, pCity, pState, pZipcode, pDeadline,
pWeight, pStatus)

        packageHashTable.insert(pID, p)

# Time complexity: O(N), where N is the number of packages in the file.
```

```
def distanceInBetween(xValue, yValue):

    distance = distance_csv[xValue][yValue]

    if distance == '':

        distance = distance_csv[yValue][xValue]

    return float(distance)

# Time complexity: O(1)
```

```
def extractAddress(address):

    for row in address_csv:

        if address in row[2]:

            return int(row[0])

# Time complexity: O(N), where N is the number of addresses in the CSV file.
```

```
def deliveringPackages(trucks):

    notDelivered = []

    for packageID in trucks.packages:

        package = packageHashTable.search(packageID)

        notDelivered.append(package)

    trucks.packages.clear()

    while len(notDelivered) > 0:
```

```

nextAddress = 2000

nextPackage = None

for package in notDelivered:

    if distanceInBetween(extractAddress(trucks.address),
extractAddress(package.address)) <= nextAddress:

        nextAddress = distanceInBetween(extractAddress(trucks.address),
extractAddress(package.address))

        nextPackage = package

trucks.packages.append(nextPackage.ID)

notDelivered.remove(nextPackage)

trucks.mileage += nextAddress

trucks.address = nextPackage.address

trucks.time += datetime.timedelta(hours=nextAddress / 18)

nextPackage.deliveryTime = trucks.time

nextPackage.departureTime = trucks.departureTime

# Time complexity: O(N^2), where N is the number of packages on the truck.

```

```

class ChainingHashTable:

    def __init__(self, initial_capacity=40):

        self.table = []

        for i in range(initial_capacity):

            self.table.append([])

    def insert(self, key, item):

        bucket = hash(key) % len(self.table)

        bucket_list = self.table[bucket]

```

```

        for kv in bucket_list:

            #print (key_value)

            if kv[0] == key:

                kv[1] = item

                return True

        key_value = [key, item]

        bucket_list.append(key_value)

        return True

def search(self, key):

    bucket = hash(key) % len(self.table)

    bucket_list = self.table[bucket]

    for kv in bucket_list:

        if kv[0] == key:

            return kv[1] # value

    return None

```

```

def remove(self, key):

    bucket = hash(key) % len(self.table)

    bucket_list = self.table[bucket]

    for kv in bucket_list:

        if kv[0] == key:

            bucket_list.remove([kv[0], kv[1]])

```

Time Complexity: $O(1)$ on average, $O(n)$ in the worst case (due to potential collisions) for insert, search, and remove

The overall time complexity of the program is $O(n^2)$. At the end of each chunk of code, there should be a comment analyzing and describing the runtime complexity in Big O notation.

B4. Scalability and Adaptability

In terms of adaptability and scalability, my program is able to handle adding a moderate amount of additional packages, but as the package begins to reach the hundreds, there will be potential performance issues as well as more needed coding. I opted to manually hard code each package into the truck objects rather than create a function for loading them for me, so in this way, the program is only as adaptable as it needed to be for this specific scenario. If hundreds of packages were to be added it would require manually coding hundreds more package IDs to each truck, which for some individuals could be tedious, but for others more preferable. In this way, the program does exhibit the necessary functionality to adapt to a new scenario with hundreds more packages. Additionally, as it pertains to scalability, if hundreds of packages were added to the program in its current state, there could be potential collisions. This could occur due to a change in the load factor of the hash function when more elements are added to the table, so in this way, the program does have the capability to handle hundreds more packages, but in handling those new packages, there will be the potential for a loss in performance. It is important to note the use of a hashtable data structure to hold all the package and address data was a requirement of the scenario. There are other data structures that could have been used to make the program more to scale, but the requirement of a hashtable intrinsically makes the program vulnerable to a loss in performance as it scales.

B5. Software Efficiency and Maintainability

To start, the project has a well-organized structure, separating each class into modules, and leaving extensive programmer notes for each line of code so the program can be easily picked up from where it was left off. This separation allows for easier maintainability and readability for other programmers. Additionally, in its current state, the program is acting as a template for a package delivery system that can be easily built onto and expanded. In these ways the program is efficient in what it can do for its relatively small size, and maintainable for its well-organized structure.

B6. Self-Adjusting Data Structures

The major benefit of the hashtable is its ability to add, remove, and delete items into the data structure with a runtime complexity of $O(1)$. This is extremely efficient and one of if not the quickest data structures for querying data from the structure, but where the hashtable begins to decrease in performance is when the number of packages becomes too big for the size of the list. When this happens the load factor of the hashtable changes and results in more collisions, which can hamper the performance of the hashtable. So, in these ways the hashtable is beneficial for querying data but has the potential, if not properly maintained, to have poor performance related to collision and storage.

C1 Identification Information, C2. Process and Flow Comments

In attached programming files

D. Data Structure

The self-adjusting data structure I implemented for my program is the chaining hash table.

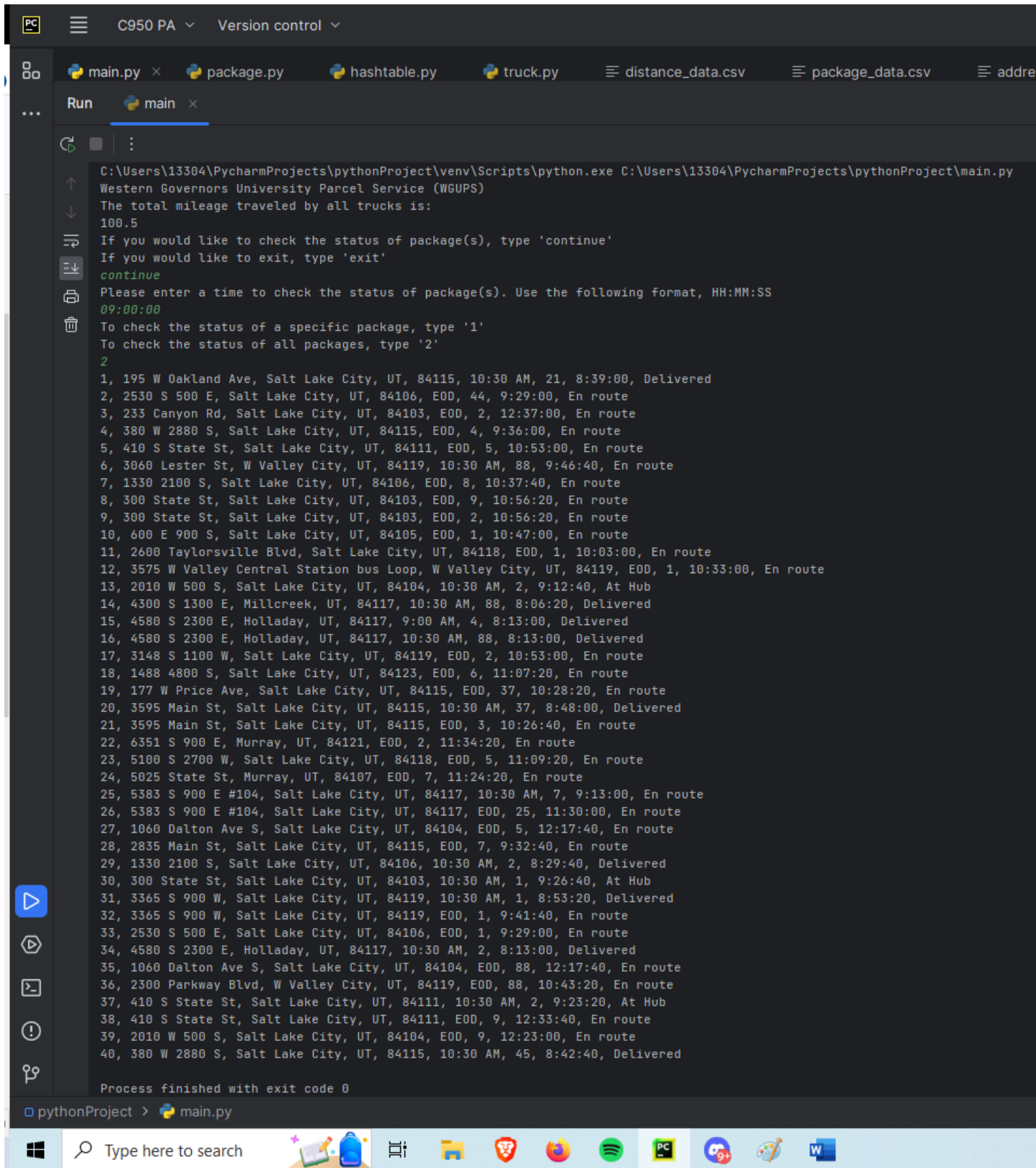
D1. Explanation of Data Structure

The code used in my program for the hash table structure comes directly from the Zybooks curriculum as well as the 4-part C950 webinar series with Dr. Cemal Tepe (Lysecky, section 7). It uses key-value pairs for storing and retrieving data within a series of nested lists, which serves as the basis for the hashtable data structure. Each nested list is a “bucket” which can be queried with the modulo % operator. Each package ID can be queried with either an insert, search, or delete into the various buckets within the hashtable. In this way, all of the necessary package functionality is implemented into the program.

E Hash Table, F. Lookup Function

In attached programming files

G1. First Status Check, time between 8:35 a.m. and 9:25 a.m.



```
C:\Users\13304\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\13304\PycharmProjects\pythonProject\main.py
Western Governors University Parcel Service (WGUPS)
The total mileage traveled by all trucks is:
100.5
If you would like to check the status of package(s), type 'continue'
If you would like to exit, type 'exit'
continue
Please enter a time to check the status of package(s). Use the following format, HH:MM:SS
09:00:00
To check the status of a specific package, type '1'
To check the status of all packages, type '2'
2
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:39:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 9:29:00, En route
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 12:37:00, En route
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:36:00, En route
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:53:00, En route
6, 3060 Lester St, W Valley City, UT, 84119, 10:30 AM, 88, 9:46:40, En route
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:37:40, En route
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 10:56:20, En route
9, 300 State St, Salt Lake City, UT, 84103, EOD, 2, 10:56:20, En route
10, 600 E 900 S, Salt Lake City, UT, 84105, EOD, 1, 10:47:00, En route
11, 2600 Taylorsville Blvd, Salt Lake City, UT, 84118, EOD, 1, 10:03:00, En route
12, 3575 W Valley Central Station bus Loop, W Valley City, UT, 84119, EOD, 1, 10:33:00, En route
13, 2010 W 500 S, Salt Lake City, UT, 84104, 10:30 AM, 2, 9:12:40, At Hub
14, 4300 S 1300 E, Millcreek, UT, 84117, 10:30 AM, 88, 8:06:20, Delivered
15, 4580 S 2300 E, Holladay, UT, 84117, 9:00 AM, 4, 8:13:00, Delivered
16, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 88, 8:13:00, Delivered
17, 3148 S 1100 W, Salt Lake City, UT, 84119, EOD, 2, 10:53:00, En route
18, 1488 4800 S, Salt Lake City, UT, 84123, EOD, 6, 11:07:20, En route
19, 177 W Price Ave, Salt Lake City, UT, 84115, EOD, 37, 10:28:20, En route
20, 3595 Main St, Salt Lake City, UT, 84115, 10:30 AM, 37, 8:48:00, Delivered
21, 3595 Main St, Salt Lake City, UT, 84115, EOD, 3, 10:26:40, En route
22, 6351 S 900 E, Murray, UT, 84121, EOD, 2, 11:34:20, En route
23, 5100 S 2700 W, Salt Lake City, UT, 84118, EOD, 5, 11:09:20, En route
24, 5025 State St, Murray, UT, 84107, EOD, 7, 11:24:20, En route
25, 5383 S 900 E #104, Salt Lake City, UT, 84117, 10:30 AM, 7, 9:13:00, En route
26, 5383 S 900 E #104, Salt Lake City, UT, 84117, EOD, 25, 11:30:00, En route
27, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 5, 12:17:40, En route
28, 2835 Main St, Salt Lake City, UT, 84115, EOD, 7, 9:32:40, En route
29, 1330 2100 S, Salt Lake City, UT, 84106, 10:30 AM, 2, 8:29:40, Delivered
30, 300 State St, Salt Lake City, UT, 84103, 10:30 AM, 1, 9:26:40, At Hub
31, 3365 S 900 W, Salt Lake City, UT, 84119, 10:30 AM, 1, 8:53:20, Delivered
32, 3365 S 900 W, Salt Lake City, UT, 84119, EOD, 1, 9:41:40, En route
33, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 1, 9:29:00, En route
34, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 2, 8:13:00, Delivered
35, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 88, 12:17:40, En route
36, 2300 Parkway Blvd, W Valley City, UT, 84119, EOD, 88, 10:43:20, En route
37, 410 S State St, Salt Lake City, UT, 84111, 10:30 AM, 2, 9:23:20, At Hub
38, 410 S State St, Salt Lake City, UT, 84111, EOD, 9, 12:33:40, En route
39, 2010 W 500 S, Salt Lake City, UT, 84104, EOD, 9, 12:23:00, En route
40, 380 W 2880 S, Salt Lake City, UT, 84115, 10:30 AM, 45, 8:42:40, Delivered

Process finished with exit code 0
pythonProject > main.py
```

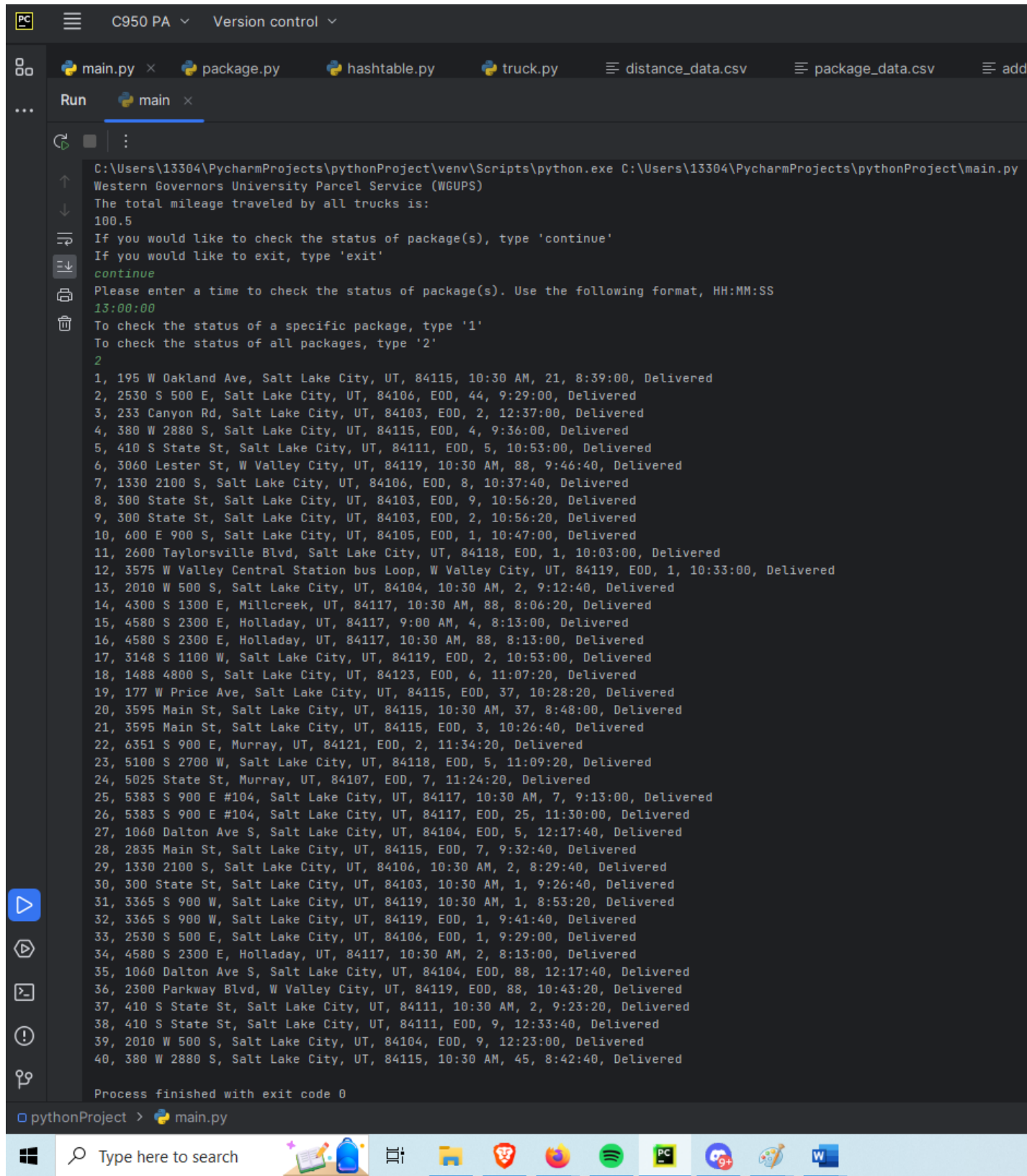
G2. Second Status Check, time between 9:35 a.m. and 10:25 a.m.

```
PC  C950 PA  Version control  ...
main.py  package.py  hashtable.py  truck.py  distance_data.csv  package_data.csv  address...
Run  main  x

C:\Users\13304\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\13304\PycharmProjects\pythonProject\main.py
Western Governors University Parcel Service (WGUPS)
The total mileage traveled by all trucks is:
100.5
If you would like to check the status of package(s), type 'continue'
If you would like to exit, type 'exit'
continue
Please enter a time to check the status of package(s). Use the following format, HH:MM:SS
10:00:00
To check the status of a specific package, type '1'
To check the status of all packages, type '2'
2
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:39:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 9:29:00, Delivered
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 12:37:00, En route
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:36:00, Delivered
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:53:00, At Hub
6, 3060 Lester St, W Valley City, UT, 84119, 10:30 AM, 88, 9:46:40, Delivered
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:37:40, At Hub
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 10:56:20, At Hub
9, 300 State St, Salt Lake City, UT, 84103, EOD, 2, 10:56:20, At Hub
10, 600 E 900 S, Salt Lake City, UT, 84105, EOD, 1, 10:47:00, At Hub
11, 2600 Taylorsville Blvd, Salt Lake City, UT, 84118, EOD, 1, 10:03:00, At Hub
12, 3575 W Valley Central Station bus Loop, W Valley City, UT, 84119, EOD, 1, 10:33:00, En route
13, 2010 W 500 S, Salt Lake City, UT, 84104, 10:30 AM, 2, 9:12:40, Delivered
14, 4300 S 1300 E, Millcreek, UT, 84117, 10:30 AM, 88, 8:06:20, Delivered
15, 4580 S 2300 E, Holladay, UT, 84117, 9:00 AM, 4, 8:13:00, Delivered
16, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 88, 8:13:00, Delivered
17, 3148 S 1100 W, Salt Lake City, UT, 84119, EOD, 2, 10:53:00, En route
18, 1488 4800 S, Salt Lake City, UT, 84123, EOD, 6, 11:07:20, En route
19, 177 W Price Ave, Salt Lake City, UT, 84115, EOD, 37, 10:28:20, En route
20, 3595 Main St, Salt Lake City, UT, 84115, 10:30 AM, 37, 8:48:00, Delivered
21, 3595 Main St, Salt Lake City, UT, 84115, EOD, 3, 10:26:40, En route
22, 6351 S 900 E, Murray, UT, 84121, EOD, 2, 11:34:20, En route
23, 5100 S 2700 W, Salt Lake City, UT, 84118, EOD, 5, 11:09:20, En route
24, 5025 State St, Murray, UT, 84107, EOD, 7, 11:24:20, En route
25, 5383 S 900 E #104, Salt Lake City, UT, 84117, 10:30 AM, 7, 9:13:00, Delivered
26, 5383 S 900 E #104, Salt Lake City, UT, 84117, EOD, 25, 11:30:00, En route
27, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 5, 12:17:40, En route
28, 2835 Main St, Salt Lake City, UT, 84115, EOD, 7, 9:32:40, Delivered
29, 1330 2100 S, Salt Lake City, UT, 84106, 10:30 AM, 2, 8:29:40, Delivered
30, 300 State St, Salt Lake City, UT, 84103, 10:30 AM, 1, 9:26:40, Delivered
31, 3365 S 900 W, Salt Lake City, UT, 84119, 10:30 AM, 1, 8:53:20, Delivered
32, 3365 S 900 W, Salt Lake City, UT, 84119, EOD, 1, 9:41:40, Delivered
33, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 1, 9:29:00, Delivered
34, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 2, 8:13:00, Delivered
35, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 88, 12:17:40, En route
36, 2300 Parkway Blvd, W Valley City, UT, 84119, EOD, 88, 10:43:20, En route
37, 410 S State St, Salt Lake City, UT, 84111, 10:30 AM, 2, 9:23:20, Delivered
38, 410 S State St, Salt Lake City, UT, 84111, EOD, 9, 12:33:40, En route
39, 2010 W 500 S, Salt Lake City, UT, 84104, EOD, 9, 12:23:00, En route
40, 380 W 2880 S, Salt Lake City, UT, 84115, 10:30 AM, 45, 8:42:40, Delivered

Process finished with exit code 0
pythonProject > main.py
```

G3. Third Status Check, time between 12:03 p.m. and 1:12 p.m.

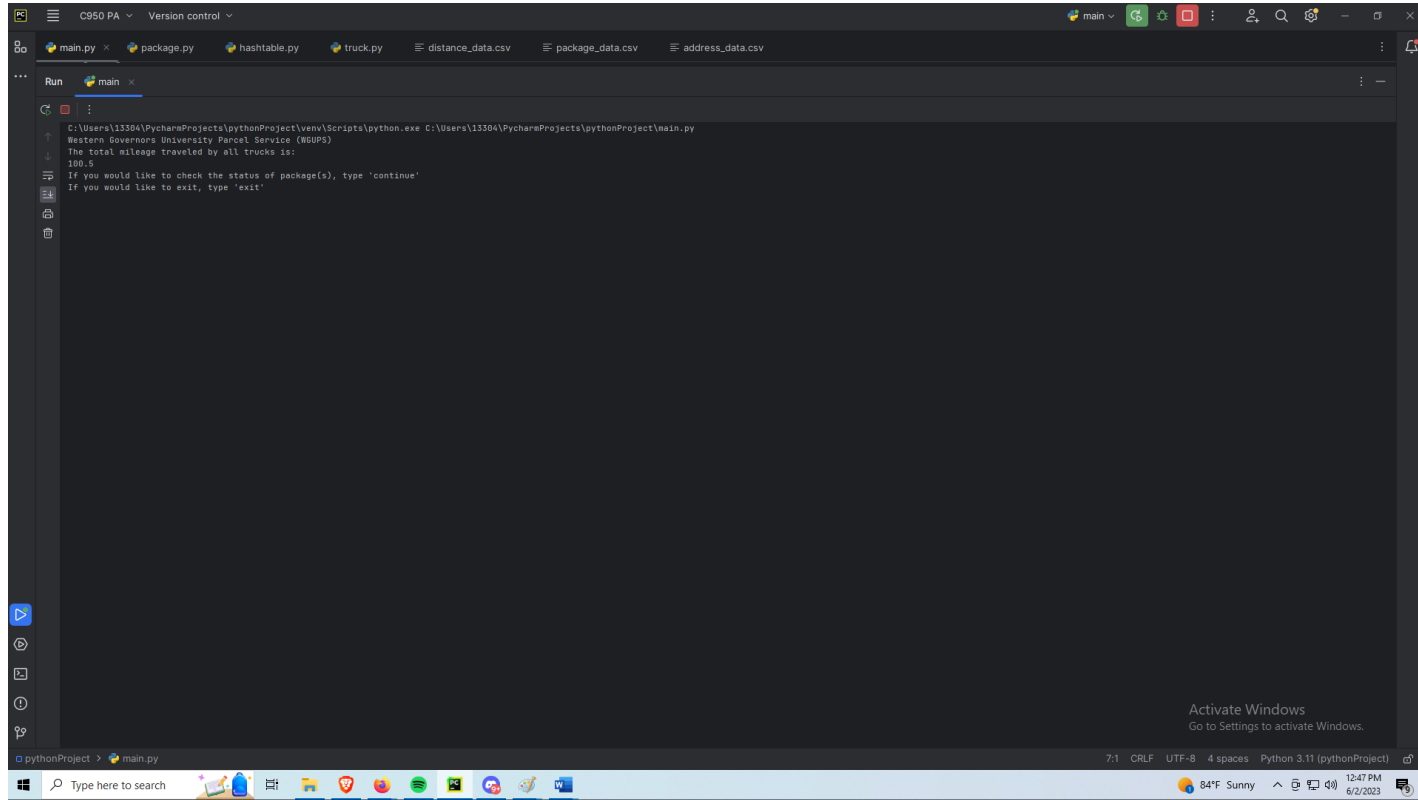


```
C:\Users\13304\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\13304\PycharmProjects\pythonProject\main.py
Western Governors University Parcel Service (WGUPS)
The total mileage traveled by all trucks is:
100.5
If you would like to check the status of package(s), type 'continue'
If you would like to exit, type 'exit'
continue
Please enter a time to check the status of package(s). Use the following format, HH:MM:SS
13:00:00
To check the status of a specific package, type '1'
To check the status of all packages, type '2'
2
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:39:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 9:29:00, Delivered
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 12:37:00, Delivered
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:36:00, Delivered
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:53:00, Delivered
6, 3060 Lester St, W Valley City, UT, 84119, 10:30 AM, 88, 9:46:40, Delivered
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:37:40, Delivered
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 10:56:20, Delivered
9, 300 State St, Salt Lake City, UT, 84103, EOD, 2, 10:56:20, Delivered
10, 600 E 900 S, Salt Lake City, UT, 84105, EOD, 1, 10:47:00, Delivered
11, 2600 Taylorsville Blvd, Salt Lake City, UT, 84118, EOD, 1, 10:03:00, Delivered
12, 3575 W Valley Central Station bus Loop, W Valley City, UT, 84119, EOD, 1, 10:33:00, Delivered
13, 2010 W 500 S, Salt Lake City, UT, 84104, 10:30 AM, 2, 9:12:40, Delivered
14, 4300 S 1300 E, Millcreek, UT, 84117, 10:30 AM, 88, 8:06:20, Delivered
15, 4580 S 2300 E, Holladay, UT, 84117, 9:00 AM, 4, 8:13:00, Delivered
16, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 88, 8:13:00, Delivered
17, 3148 S 1100 W, Salt Lake City, UT, 84119, EOD, 2, 10:53:00, Delivered
18, 1488 4800 S, Salt Lake City, UT, 84123, EOD, 6, 11:07:20, Delivered
19, 177 W Price Ave, Salt Lake City, UT, 84115, EOD, 37, 10:28:20, Delivered
20, 3595 Main St, Salt Lake City, UT, 84115, 10:30 AM, 37, 8:48:00, Delivered
21, 3595 Main St, Salt Lake City, UT, 84115, EOD, 3, 10:26:40, Delivered
22, 6351 S 900 E, Murray, UT, 84121, EOD, 2, 11:34:20, Delivered
23, 5100 S 2700 W, Salt Lake City, UT, 84118, EOD, 5, 11:09:20, Delivered
24, 5025 State St, Murray, UT, 84107, EOD, 7, 11:24:20, Delivered
25, 5383 S 900 E #104, Salt Lake City, UT, 84117, 10:30 AM, 7, 9:13:00, Delivered
26, 5383 S 900 E #104, Salt Lake City, UT, 84117, EOD, 25, 11:30:00, Delivered
27, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 5, 12:17:40, Delivered
28, 2835 Main St, Salt Lake City, UT, 84115, EOD, 7, 9:32:40, Delivered
29, 1330 2100 S, Salt Lake City, UT, 84106, 10:30 AM, 2, 8:29:40, Delivered
30, 300 State St, Salt Lake City, UT, 84103, 10:30 AM, 1, 9:26:40, Delivered
31, 3365 S 900 W, Salt Lake City, UT, 84119, 10:30 AM, 1, 8:53:20, Delivered
32, 3365 S 900 W, Salt Lake City, UT, 84119, EOD, 1, 9:41:40, Delivered
33, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 1, 9:29:00, Delivered
34, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 2, 8:13:00, Delivered
35, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 88, 12:17:40, Delivered
36, 2300 Parkway Blvd, W Valley City, UT, 84119, EOD, 88, 10:43:20, Delivered
37, 410 S State St, Salt Lake City, UT, 84111, 10:30 AM, 2, 9:23:20, Delivered
38, 410 S State St, Salt Lake City, UT, 84111, EOD, 9, 12:33:40, Delivered
39, 2010 W 500 S, Salt Lake City, UT, 84104, EOD, 9, 12:23:00, Delivered
40, 380 W 2880 S, Salt Lake City, UT, 84115, 10:30 AM, 45, 8:42:40, Delivered

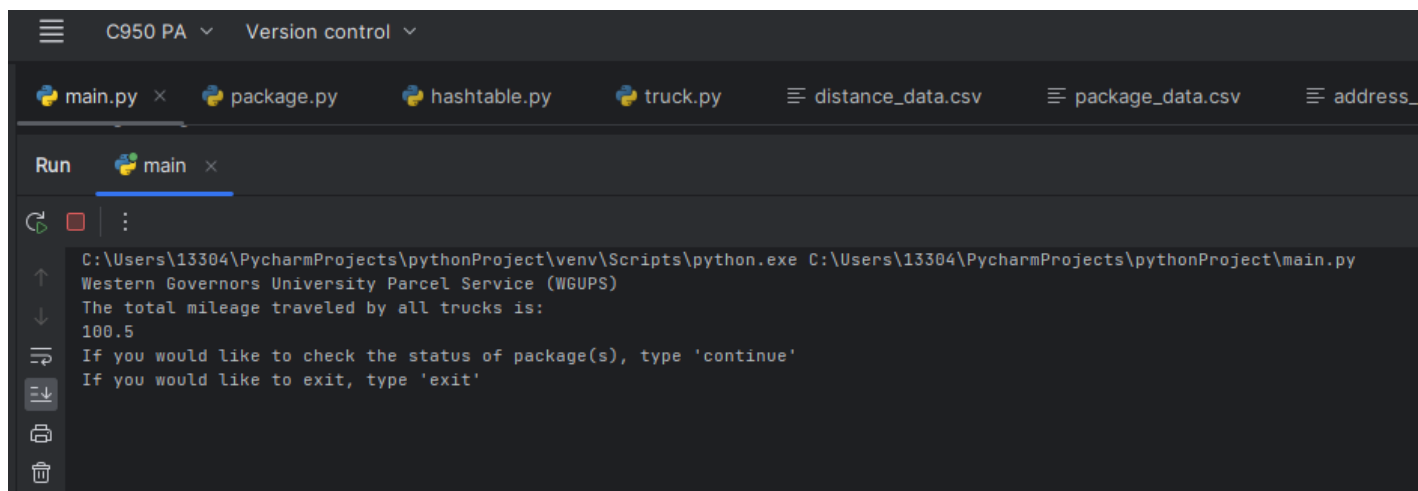
Process finished with exit code 0
pythonProject > main.py
```

H. Screen shots of Code Execution

The screenshots show the execution of the code where total truck mileage is calculated.



```
C:\Users\13304\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\13304\PycharmProjects\pythonProject\main.py
Western Governors University Parcel Service (WGUPS)
The total mileage traveled by all trucks is:
100.5
If you would like to check the status of package(s), type 'continue'
If you would like to exit, type 'exit'
```



```
C:\Users\13304\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\13304\PycharmProjects\pythonProject\main.py
Western Governors University Parcel Service (WGUPS)
The total mileage traveled by all trucks is:
100.5
If you would like to check the status of package(s), type 'continue'
If you would like to exit, type 'exit'
```

11. Strengths of Chosen Algorithm

The nearest neighbor algorithm used to order packages on each truck has the following strengths:

- Simplicity: The algorithm is relatively simple to implement and understand.
- Efficiency: It has a time complexity of $O(N^2)$, where N is the number of packages on the truck. While not the most efficient algorithm, it is efficient enough to maintain a total truck mileage under 140 miles.

Additionally, this algorithm was the easiest of all the algorithms to implement, because of the ability to manually hard code loading each of the trucks.

I2. Verification of Algorithm

Verification of meeting requirements:

- The algorithm assigns packages to trucks based on their delivery locations and deadlines.
- The algorithm uses the nearest neighbor algorithm to determine the order of package deliveries, which minimizes the distance traveled.
- The algorithm considers the constraint that truck 3 cannot depart until both truck 1 and truck 2 have finished delivering their packages.
- The algorithm updates the delivery time and status of each package accordingly.
- The algorithm produces a total mileage of all three trucks under 140 miles.
- The algorithm ends when all 40 packages have been delivered.
- The algorithm makes it so no driver leaves before 8:00:00AM.
- The algorithm produces no collisions
- The algorithm allows no more than 16 deliveries per truck.

I3. Other Possible Algorithms

Rather than use the Nearest Neighbor Algorithm, I could have implemented Dijkstra's shortest path or the farthest insertion algorithm.

I3a. Algorithm Differences

- Dijkstra's algorithm and the nearest neighbor algorithm approach the problem of finding the most optimal path for delivering packages in different ways. The nearest neighbor algorithm follows a greedy approach and prioritizes whatever address is closest to where it is currently. This means whatever address is closest to the current address the truck is at will be given top priority in selecting the next package to deliver. Dijkstra's algorithm is different in that it ensures the shortest path is selected by stopping before the next delivery and scanning all packages at each stop to find the most optimal path. This may create a larger runtime complexity for Dijkstra's algorithm but will guarantee a shorter total mileage for the cars by the end of their delivery routes.
- The farthest insertion algorithm and the nearest neighbor algorithm approach the problem of finding the most optimal path for delivering packages in different ways. The

farthest insertion algorithm differs from the nearest neighbor algorithm in terms of the initial selection strategy. Before departing the farthest insertion considers balance and equal distribution of packages across the route rather than whatever is closest to the truck upon exiting the hub. The farthest insertion algorithm is trying to shorten the overall route length by considering packages farthest away from where it is currently at, whereas the nearest neighbor only cares to look at what is closest to its current location. So in this way, the farthest insertion algorithm might have a larger runtime complexity, but it will benefit from having a shorter total mileage traveled.

J. Different Approach

I choose the nearest neighbor algorithm because it was easiest to implement by hardcoding the loading of the trucks and comparing distances to find the address with the shortest distance from the truck's current location. This reasoning helped me get through the project quickly but did not strengthen my programming skills as much as they could have been had I gone with another algorithm. This is why if I were to do this project again, I would choose to follow the instructor's webinars and use another method of loading the trucks with packages rather than manually hardcoding them into the truck.

K1. Verification of Data Structure

Verification of meeting requirements: The data structure used in the solution, a Chaining Hash Table, meets the requirements in the scenario:

- It allows efficient insertion and retrieval of package objects based on their package ID.
- It ensures unique keys (package IDs) and handles collisions through chaining, which ensures the integrity of the data.
- It provides a constant-time average complexity for insertion and retrieval operations, making it efficient for large-scale problems.

K1a. Efficiency

In terms of time complexity, the time complexity of the search, insert, and remove methods in the hash table are all on average constant meaning they have a time complexity of $O(1)$, but in the worst case, they are $O(N)$. Changing the number of packages to be delivered will not affect the Hash table's overall time complexity as it relates to its functionality, other than if the table were too small for the number of packages. This would yield a worst-case time complexity of $O(N)$ due to many collisions, but this metric would not change if more packages were added, the worst case will always be $O(N)$.

K1b. Overhead

In terms of space complexity, the space complexity of the hash table is constant meaning $O(1)$. Changing the number of packages to be delivered will directly affect the space complexity of the hash table because as more packages are added to the hash table, the space required to store the key-value pairs increases.

K1c. Implications

Changes to the number of trucks or the number of cities would not directly affect the look-up time of the hash table, because the look-up time is always on average constant $O(1)$. However, changes to the number of trucks or cities will directly affect the space usage of the data structure because the number of packages to deliver is proportional to the number of spaces needed inside the hashtable. If more trucks were added the space complexity will grow in proportion to the added trucks. If more cities were added the space complexity would grow there would be more space/slots to in the hash table for the new cities. Additionally, by adding more slots you will change the hash tables load factor, which could cause more collisions, ultimately increasing space complexity.

K2. Other Data Structures

The requirements specifically mention not using a dictionary, so the two other data structures that could be used for this scenario would be a binary search tree (BST), and a queue.

K2a. Data Structure Differences

A binary search tree differs from a hash table in many ways, I will list a few:

1. Structure
 - a. A binary search tree is a hierarchical structure, whereas a hashtable is an array of nested lists.
2. Operations
 - a. Both binary search trees and hashtables have add, delete, and search functionality, but where binary trees differ is in the way that they achieve this functionality. In a binary search tree, the tree moves from either the bottom up or the top down left to right with children, parents, and root nodes. Whereas, in a hashtable, the hashtable is a series of lists within a list known as buckets that are queried with the modulo operator %.
3. Search efficiency
 - a. A binary search tree's search time complexity is $O(\log N)$ on average and $O(N)$ in the worst case. Whereas, a hash table's search time complexity is constant $O(1)$ and worst case with collisions $O(N)$.

K2b. Data Structure Differences

A queue differs from a hash table in many ways, I will list a few:

1. Structure
 - a. A queue is a linear data structure meaning it is a list of items that follow the principles of first in first out (FIFO). Similar to stacking plates, whatever plate you just put on the stack you have to first take that one off to get to the next plate, till you find your desired plate. This is different from the hash table structure which searches through a series of buckets or nested lists with the modulo operator.
2. Operations
 - a. A queue has distinctly different operations than a hash table by utilizing operations such as enqueue, which adds elements to the rear of the queue, and dequeue, which removes elements from the top of the queue. This is different from a hash table that utilizes insert, search, and delete from anywhere within the hash table.
3. Search efficiency
 - a. A queue does not provide direct search capabilities as it follows the FIFO principle as was previously stated above, because of this, a queues search time complexity is $O(N)$ since reaching any specific element isn't possible without popping of an element from the top. This is different from a hash table where you can search any element in a bucket directly with a search time complexity of $O(1)$.

M. Professional Communication

Thank you for reading through this write-up, I hope it has been appropriately structured for your benefit.

L. Sources - Work Cited

Work Cited

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

Retrieved June 02, 2023, from

<https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/7/section/2>

Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @Ericdrowell.

www.bigocheatsheet.com.

“Panopto.” *Panopto*,

wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=a6e33b6d-9753-4ba4-a1b6-a
c8000f5d250.

Jason Brownlee. “Tutorial to Implement K-Nearest Neighbors in Python from Scratch.” *Machine Learning Mastery*, 14 Apr. 2018,

machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/.