C950 Data Structures and Algorithms II

Hiep Pham

ID #001467696

WGU Email: hpham25@wgu.edu

Date: 07/26/2023

Introduction

This scenario describes WGUPS's company finding an efficient route to complete its daily local deliveries. The scenario has different constraints, to include: 2 drivers and 3 trucks, 40 packages, and specific packages having their constraints. The program is created using Python 3.11 and implements data structures, hashes, and algorithms to meet business requirements.

A. Algorithm Identification

In this program, the Nearest Neighbor Algorithm is used to determine the path the truck takes. This self-adjusting algorithm determines the order the packages are delivered based on how near each package is to the truck.

B1. Logic Comments

The algorithm does the following, along with its complexity:

1. Iterate through all packages of the load until empty and perform the following (while len(load) != 0):	O(N^3)
a. Set temporary distance "next_address_distance" to max (999999).	-O(1)
b. Iterate through all packages in load and perform the following:	-O(N^2)
i. Iterate through the address table to set the vehicle address id and parcel address id.	--O(N)
ii. Set the variable "next_address_distance" to the distance of the package with the lowest distance from the truck using enumerate_address_id_from_address to pull the address id.	**--O(1)** **--O(1)**
iii. Set the variable "next_package_instance" to the package with the lowest distance from the truck using enumerate_address_id_from_address to pull the address id.	**--O(1)**
c. Remove the package from the load and print delivered.	-O(1)
d. Add the time it took to deliver the package to the truck's time.	-O(1)
e. Update the package's delivery time to the truck's time (at arrival).	-O(1)
f. Update the package's departure time to the truck's departure time (when left) and print details of the delivery (departure & delivery).	-O(1)
g. Add the mileage to the total of all trucks' mileage and print the mileage it took to make delivery.	-O(1)
h. Set the truck's address to the package's address (as if the truck has just been delivered) and print the departure.	-O(1)

The bolded text is the Nearest Neighbor Algorithm.

B2. Development Environment

The programming environment used to create this program is PyCharm Community Edition 2023.1.3, compiled by Python 3.11. The program was built on Windows 11 using Ryzen 5600X and NVIDIA RTX 3060Ti.

B3. Space-Time and Big-O

The space-time complexity of this program is O(N^4), as calculated using the table below. The best case scenario would be O(1), for singular package delivery. The Nearest Neighbor Algorithm implementation within the program is O(N^2).

main.py

Method	**Line Number**	**Complexity**
N/a (GUI while loop)	21, 31, 55	O(N * (N + N)) = O(N^2)
TOTAL		**O(N^2) = O(N^2)**

structure.py

Method	**Line Number**	**Complexity**
add_key_and_value	11	O(N)
obtain_value_based_on_key	24	O(N)
__init__	34	O(N)
TOTAL		**3N = O(N)**

parcel.py

Method	**Line Number**	**Complexity**
__str__	15	O(1)
__init__	27	O(1)
n/a (load packages onto the hash map)	54	O(N)
TOTAL		**O(N) + 2 = O(N)**

delivery_vehicle.py

Method	**Line Number**	**Complexity**
__str__	16	O(1)
__init__	23	O(N)
update_status_after_delivery	48	O(1)
update_vehicle_address_after_delivery	63	O(1)
n/a (print vehicle fleet info)	114	O(N)
n/a (deliver packages)	123,129,135,139	O(N*N*N*N) = O(N^4)
n/a (print vehicle fleet info)	177	O(N)
TOTAL		**O(N^4) + 3O(N) + 3 = O(N^4)**

TOTAL of program: O(N^4) + O(N^2) + 2O(N) = O(N^4)

B4. Scalability and Adaptability

The application maintains an exponentially scaling computational time, enabling sets of input and packages, to scale at a reasonable pace. With a complexity of O(N^4), scaling is not problematic until the dataset grows extremely large. In an extremely large dataset, the business

requirements likely become changed due to the nature of a truck's limitations. This program is run locally, so network bandwidth scaling is irrelevant in this scenario.

B5. Software Efficiency and Maintainability

This software is efficient because it meets all constraints of the business requirements while maintaining a complexity of O(N^4). As mentioned in B4, reasonable scaling is easy, enabling the business to be efficient in computational resources. The program is created using a class & object approach, so it is modular as the business needs to adapt and grow, resulting in easy maintenance.

B6. Self-Adjusting Data Structures

The hash table used in this program is a self-adjusting data structure. A strength of the data structure (hash table) is its efficiency in adding and looking up values based on keys, enabling quick data addition and retrieval. Secondly, thanks to its efficiency, another strength of the data structure (hash table) is its scalability. The data structure (hash table) has an overall space-time complexity of O(N), along with its addition and look-up functions. This ensures linear scaling as datasets grow.

The data structure complexity is a strength when using normal dataset sizes. If the number of packages were to grow dramatically into the tens of thousands range, the complexity O(N) would no longer be effective. Linearly searching through thousands of entries in the hash table would dramatically slow down the program, consuming computational and memory bandwidth. With a data size that large, a more efficient data structure would be required.

C. Original Code

The program & code has been attached. No parts of the code were sourced or referenced from any external sources. Screenshots are provided below with the completion of the code being run free from warnings/errors.

C1. Identification Information

Each Python file has the author's first and last name and the author's ID.

C2. Process and Flow Comments

All Python files have detailed comments to explain the high-level logic of all code segments.

D. Data Structure

The project uses a hash table data structure because it meets business requirements. The hash table enables efficient storage of key-value pairs and lookup and add functions. These functions

operate with a complexity of O(N), suitable for meeting business requirements. The hash table utilizes the data points' keys to connect to its values. No code was externally sourced.

D1. Explanation of Data Structure

The data structure utilizes the package file to enumerate the hash table. The structure utilizes the package's ID to create a hash and connects a value to create a key-value pair. A constraint this requires is for the package IDs to be all unique, which this business has met. Because all IDs are unique, there is a surety that there will be no collisions.

E. Hash Table

The structure.py contains all implementations of the hash table. This file has the insertion function on line 11: add_key_and_value.

```
8       # PART E of scenario
9       # Insert a new package value into the hash table based on key,
10      # O(N) space-time complexity
        1 usage (1 dynamic)
11      def add_key_and_value(self, key, value):
12          # iterate through all the key value pairs in hash table
13          for key_value_pair in self.list[int(key) % len(self.list)]:
14              # if the key of the pair matches the inputted key
15              if key_value_pair[0] == key:
16                  # set the value of the key to be the updated key value pair
17                  key_value_pair[1] = [key, value]
18          # append the value of the key to the list at the hash location
19          self.list[int(key) % len(self.list)].append([key, value])
```

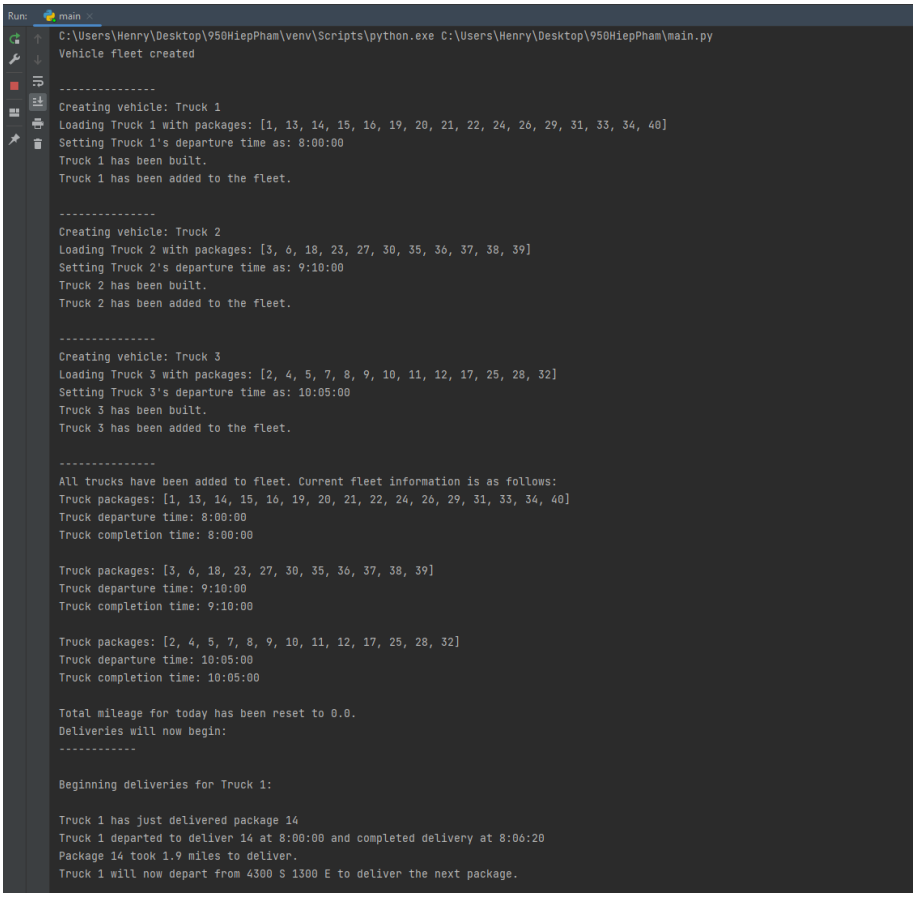
F. Look-Up Function

The structure.py contains all implementations of the hash table. This file has the lookup function on line 24: obtain_value_based_on_key.

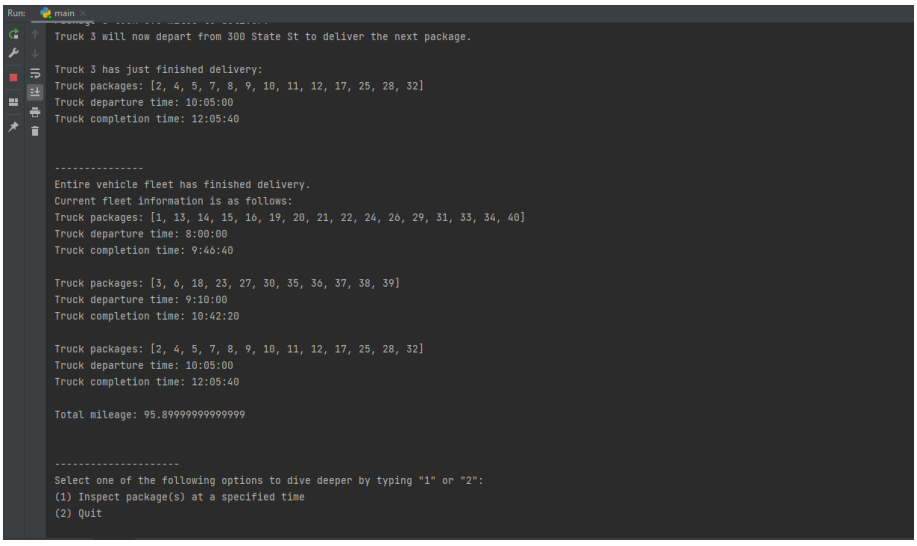
```
21      # Part F of scenario
22      # Returns value from the hash table based on key if exists. If it doesn't exist, return None.
23      # O(N) space-time complexity
        10 usages (10 dynamic)
24      def obtain_value_based_on_key(self, key):
25          # iterate through all key value pairs in hash table
26          for key_value_pair in self.list[int(key) % len(self.list)]:
27              # if the key in the key value pair matches inputted key
28              if key_value_pair[0] == key:
29                  # return the value of the key value pair
30                  return key_value_pair[1]
```

G. Interface

The interface is provided to the user through the Command Line Interface. At the initial startup, the program outputs the creation and loading of the trucks. Then, the program outputs the trucks' deliveries as it happens. A screenshot is shown below:



Then, print all three trucks' total mileage, and each truck's specific details are shown:



Then, the program asks the user for input out of the two options below:

1. Information for the package(s) at a specified time.
2. Quit program.

```
--------------------
Select one of the following options to dive deeper by typing "1" or "2":
(1) Inspect package(s) at a specified time
(2) Quit
```

If the user picks option 1, the user is then asked for a specified time in “HH:MM:SS” format. Then, the program asks again with the two options below:

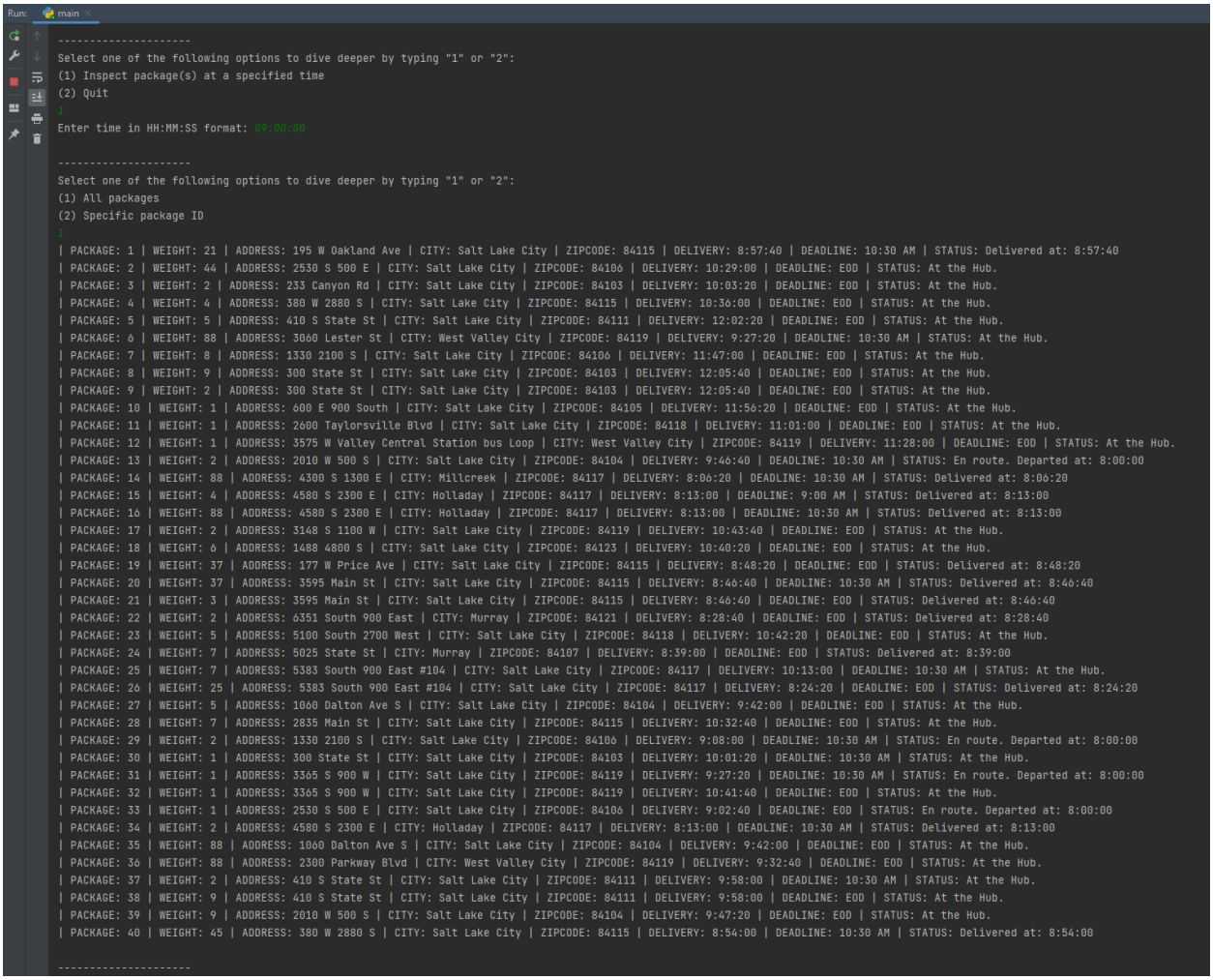
1. All packages
2. Specific package ID

```
--------------------
Select one of the following options to dive deeper by typing "1" or "2":
(1) Inspect package(s) at a specified time
(2) Quit
1
Enter time in HH:MM:SS format: 12:00:00

--------------------
Select one of the following options to dive deeper by typing "1" or "2":
(1) All packages
(2) Specific package ID
```

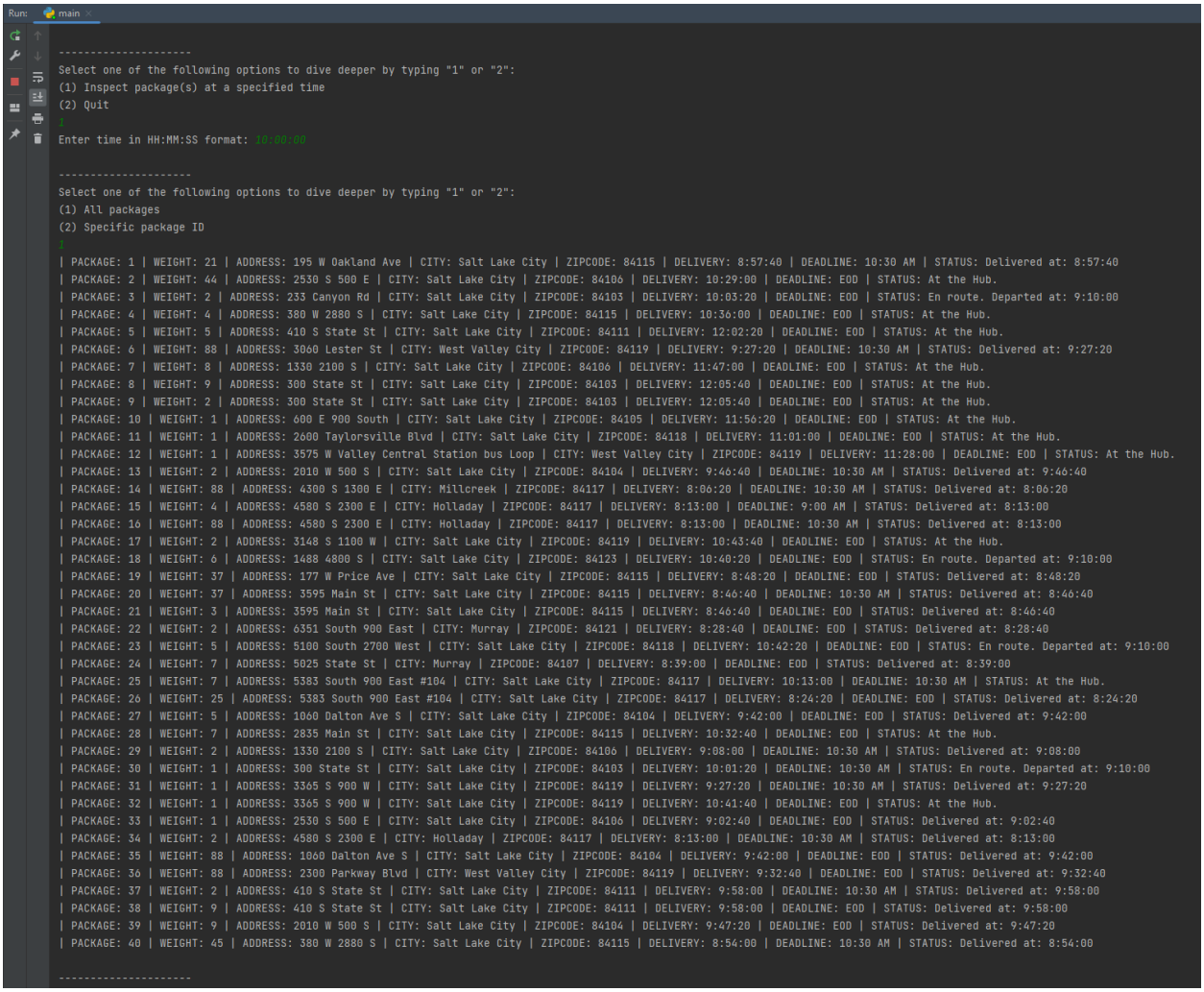
If option 1 was picked, the program outputs all packages’ information at the specified time. If option 2 was picked, the program requests the specific package ID the user is looking for, then outputs the package’s information at the specified time. The program will loop itself until the user has finished finding all information the user needs. If the user picks option 2, the program is ended.

```
--------------------
Select one of the following options to dive deeper by typing "1" or "2":
(1) Inspect package(s) at a specified time
(2) Quit
2
Program complete

Process finished with exit code 0
```

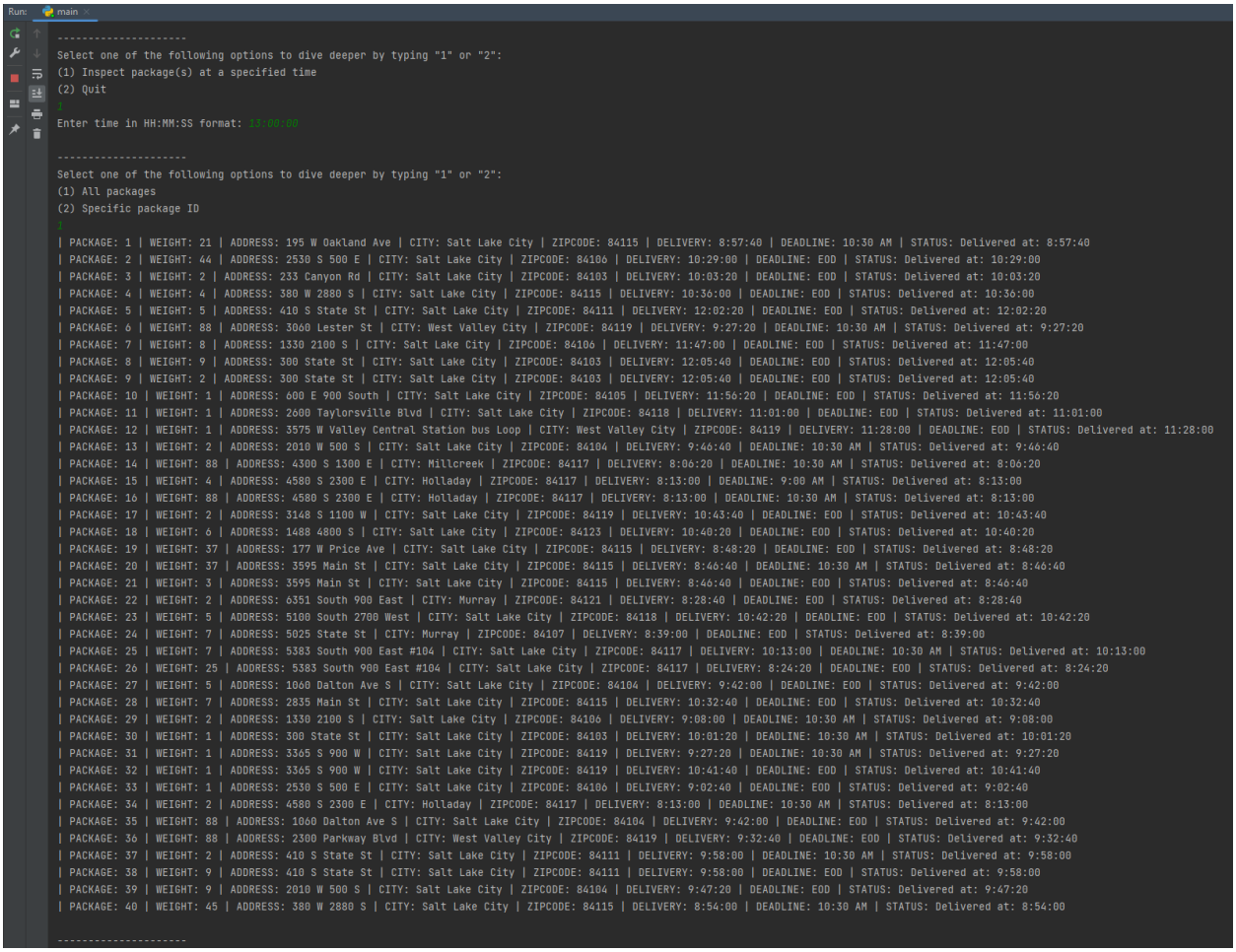
G1. First Status Check



G2. Second Status Check

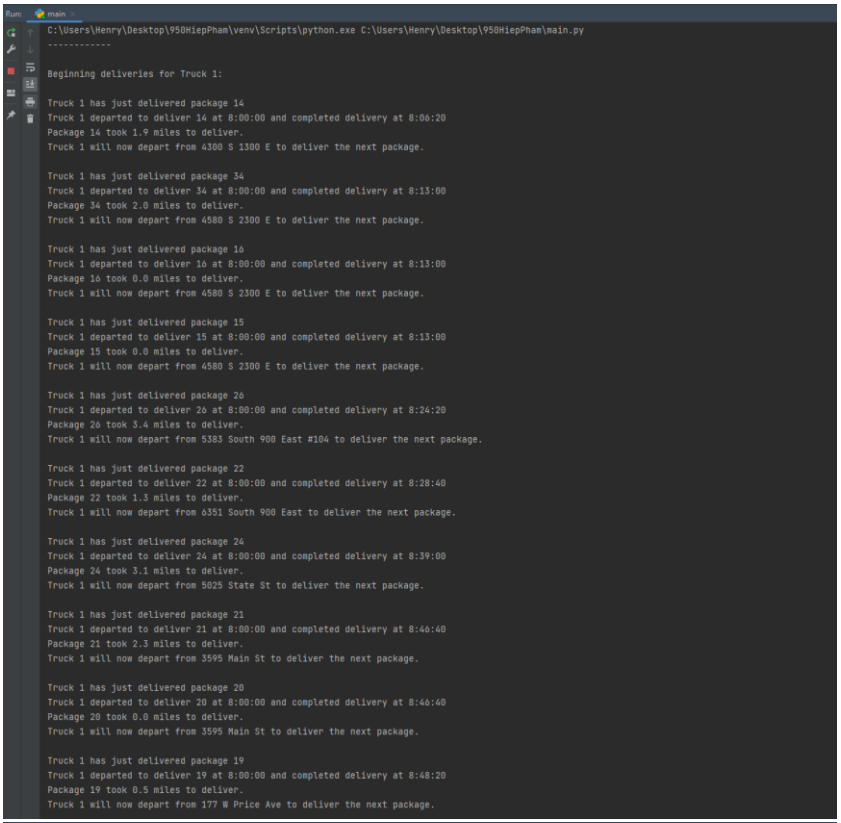


G3. Third Status Check

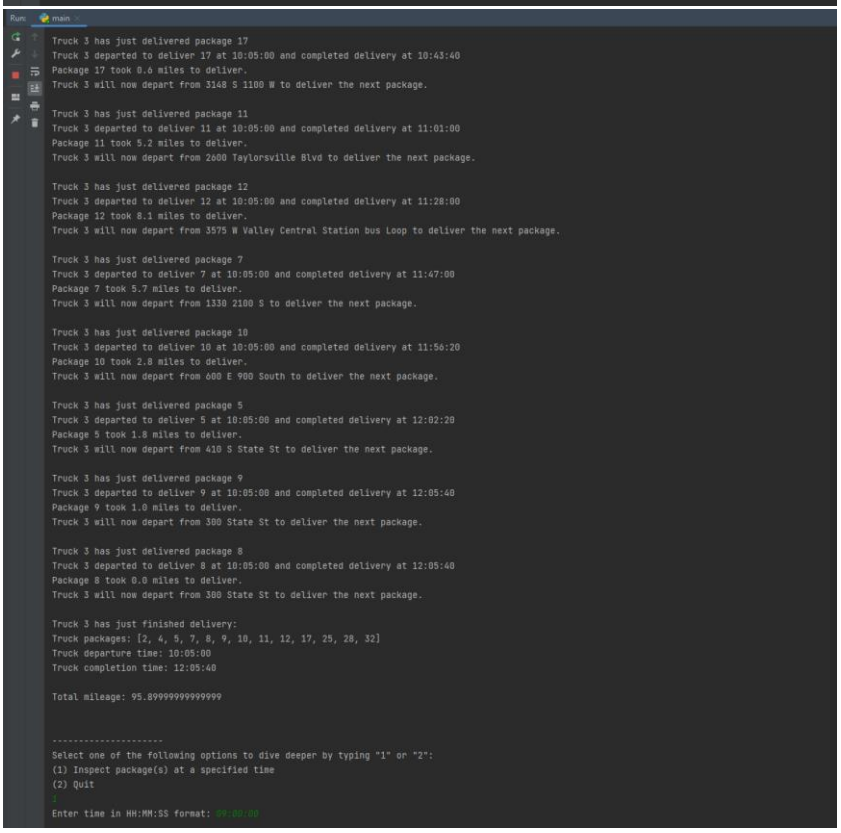


H. Screenshots of Code Execution

Menu 1, at time 09:00:00

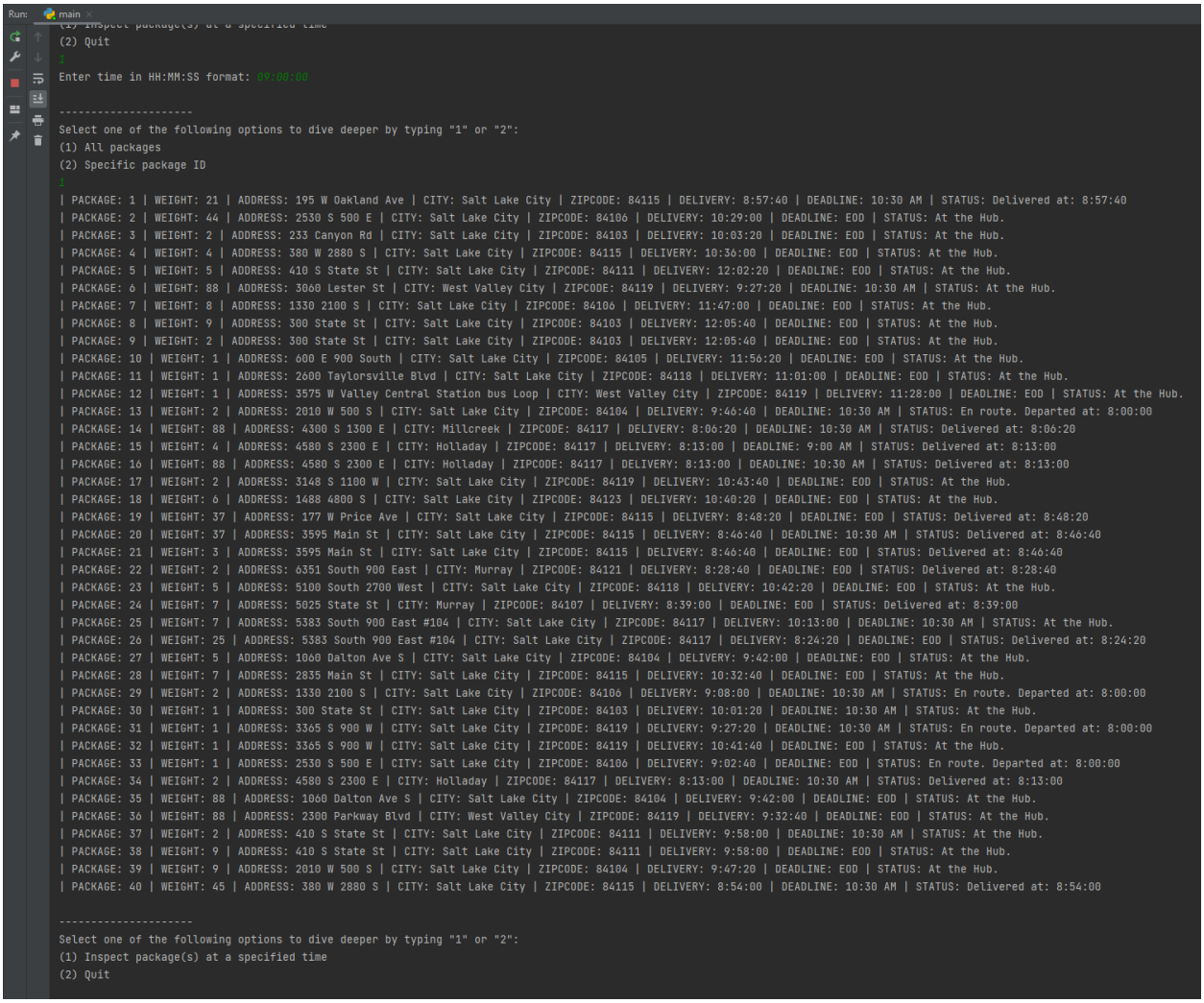


```
Run:  main
C:\Users\Henry\Desktop\950HiepPham\venv\Scripts\python.exe C:\Users\Henry\Desktop\950HiepPham\main.py
-----------

Beginning deliveries for Truck 1:

Truck 1 has just delivered package 14
Truck 1 departed to deliver 14 at 8:00:00 and completed delivery at 8:06:20
Package 14 took 1.9 miles to deliver.
Truck 1 will now depart from 4300 S 1300 E to deliver the next package.

Truck 1 has just delivered package 34
Truck 1 departed to deliver 34 at 8:00:00 and completed delivery at 8:13:00
Package 34 took 2.0 miles to deliver.
Truck 1 will now depart from 4580 S 2300 E to deliver the next package.

Truck 1 has just delivered package 16
Truck 1 departed to deliver 16 at 8:00:00 and completed delivery at 8:13:00
Package 16 took 0.0 miles to deliver.
Truck 1 will now depart from 4580 S 2300 E to deliver the next package.

Truck 1 has just delivered package 15
Truck 1 departed to deliver 15 at 8:00:00 and completed delivery at 8:13:00
Package 15 took 0.0 miles to deliver.
Truck 1 will now depart from 4580 S 2300 E to deliver the next package.

Truck 1 has just delivered package 26
Truck 1 departed to deliver 26 at 8:00:00 and completed delivery at 8:24:20
Package 26 took 3.4 miles to deliver.
Truck 1 will now depart from 5383 South 900 East #104 to deliver the next package.

Truck 1 has just delivered package 22
Truck 1 departed to deliver 22 at 8:00:00 and completed delivery at 8:28:40
Package 22 took 1.3 miles to deliver.
Truck 1 will now depart from 6351 South 900 East to deliver the next package.

Truck 1 has just delivered package 24
Truck 1 departed to deliver 24 at 8:00:00 and completed delivery at 8:39:00
Package 24 took 3.1 miles to deliver.
Truck 1 will now depart from 5025 State St to deliver the next package.

Truck 1 has just delivered package 21
Truck 1 departed to deliver 21 at 8:00:00 and completed delivery at 8:46:40
Package 21 took 2.3 miles to deliver.
Truck 1 will now depart from 3595 Main St to deliver the next package.

Truck 1 has just delivered package 20
Truck 1 departed to deliver 20 at 8:00:00 and completed delivery at 8:46:40
Package 20 took 0.0 miles to deliver.
Truck 1 will now depart from 3595 Main St to deliver the next package.

Truck 1 has just delivered package 19
Truck 1 departed to deliver 19 at 8:00:00 and completed delivery at 8:48:20
Package 19 took 0.5 miles to deliver.
Truck 1 will now depart from 177 W Price Ave to deliver the next package.
```

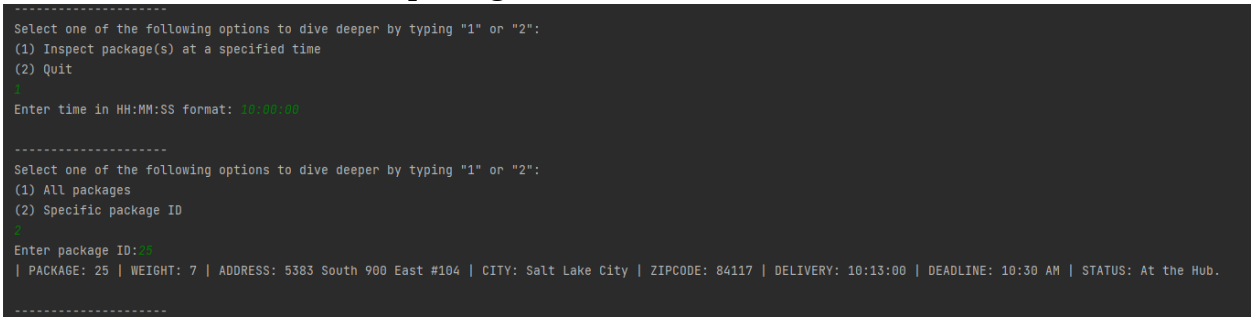


```
Run:  main
Truck 3 has just delivered package 17
Truck 3 departed to deliver 17 at 10:05:00 and completed delivery at 10:43:40
Package 17 took 0.6 miles to deliver.
Truck 3 will now depart from 3148 S 1100 W to deliver the next package.

Truck 3 has just delivered package 11
Truck 3 departed to deliver 11 at 10:05:00 and completed delivery at 11:01:00
Package 11 took 5.2 miles to deliver.
Truck 3 will now depart from 2600 Taylorsville Blvd to deliver the next package.

Truck 3 has just delivered package 12
Truck 3 departed to deliver 12 at 10:05:00 and completed delivery at 11:28:00
Package 12 took 8.1 miles to deliver.
Truck 3 will now depart from 3575 W Valley Central Station bus Loop to deliver the next package.

Truck 3 has just delivered package 7
Truck 3 departed to deliver 7 at 10:05:00 and completed delivery at 11:47:00
Package 7 took 5.7 miles to deliver.
Truck 3 will now depart from 1330 2100 S to deliver the next package.

Truck 3 has just delivered package 10
Truck 3 departed to deliver 10 at 10:05:00 and completed delivery at 11:56:20
Package 10 took 2.8 miles to deliver.
Truck 3 will now depart from 600 E 900 South to deliver the next package.

Truck 3 has just delivered package 5
Truck 3 departed to deliver 5 at 10:05:00 and completed delivery at 12:02:20
Package 5 took 1.8 miles to deliver.
Truck 3 will now depart from 410 S State St to deliver the next package.

Truck 3 has just delivered package 9
Truck 3 departed to deliver 9 at 10:05:00 and completed delivery at 12:05:40
Package 9 took 1.0 miles to deliver.
Truck 3 will now depart from 300 State St to deliver the next package.

Truck 3 has just delivered package 8
Truck 3 departed to deliver 8 at 10:05:00 and completed delivery at 12:05:40
Package 8 took 0.0 miles to deliver.
Truck 3 will now depart from 300 State St to deliver the next package.

Truck 3 has just finished delivery:
Truck packages: [2, 4, 5, 7, 8, 9, 10, 11, 12, 17, 25, 28, 32]
Truck departure time: 10:05:00
Truck completion time: 12:05:40

Total mileage: 95.89999999999999


--------------------
Select one of the following options to dive deeper by typing "1" or "2":
(1) Inspect package(s) at a specified time
(2) Quit
1
Enter time in HH:MM:SS format: 09:00:00
```

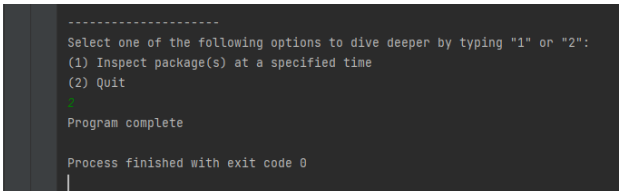
...



Menu 1, at time 10:00:00, for package 25



Menu 2, to quit the program.



I1. Strengths of Chosen Algorithm

The Nearest Neighbor Algorithm's primary strength is its ability to meet business requirements while running at an efficient complexity of O(N^2), allowing for easy scalability. A secondary strength is its simplicity in implementation and logic. Any business owner would be able to read the code and understand its mission.

I2. Verification of Algorithm

The algorithm solution meets all requirements as shown in the runtime screenshots. The route was completed with a total mileage of 95.9 miles. All packages were checked manually to ensure delivery according to package specifications. All packages were checked manually to ensure delivery according to package specifications. Details of the implementation are in the main.py file.

I3. Other possible Algorithms

Two other algorithms that would've met the requirements in the scenario are Dijkstra's algorithm and Bellman-Ford's algorithm.

I3A. Algorithm Differences

Dijkstra's graph traversal algorithm seeks "the shortest path from a source node to all other nodes". Dijkstra's algorithm searches for the shortest path overall, meanwhile, the Nearest Neighbor algorithm searches for the shortest path at a point in time. Dijkstra's algorithm would've used more computational resources depending on the implementation ("Find Shortest Paths from Source to All Vertices Using Dijkstra's Algorithm").

Like Dijkstra's algorithm, the Bellman-Ford algorithm also seeks "the shortest path from the source to all other nodes". However, Bellman-Ford also "considers negative edge weights and identifies negative cycles" in its optimization of the shortest path. Bellman-Ford is also unlike the Nearest Neighbor algorithm which only seeks the nearest node from each source node. Like Dijkstra's algorithm, the Bellman-Ford algorithm also would've used more computational resources depending on the implementation ("Bellman–Ford Algorithm | DP-23 - GeeksforGeeks").

J. Different Approach

The different approach I would take is to implement a function to dynamically load the trucks rather than manually load them. Enabling dynamic loading would aid in scalability as businesses grow in dataset size. The dynamic loading function would account for late packages, errors in addresses, and delivery deadline times to load to the correct truck most optimally.

K1. Verification of Data Structure

All packages were delivered before the deadline and checked manually. All packages were checked manually to ensure delivery according to package specifications. The total mileage from all trucks is 95.9 miles. The efficiency of the hash table can be verified in part D the look-up function can be verified in part F. Screenshots from sections G & H verify the completion of business requirements, and the implementation of data structure code can be found in structure.py.

K1A. Efficiency

The obtain_value_based_on_key function (lookup) of the data structure has a complexity of O(N), scaling linearly with the size of the dataset. This enables scalability for small and large businesses as their datasets grow. In this solution, the dataset is referring to the number of packages WGUPS requires to be delivered. The obtain_value_based_on_key function (lookup) of the data structure will grow and downsize linearly as WGUPS grows and downsizes its package deliveries. The lack of exponential growth ensures growing datasets will not affect the program's efficiency dramatically.

K1B. Overhead

The data structure’s space usage is linear, scaling with the size of the dataset or total packages. The hash table has a total of complexity O(N), increasing at the same rate as the dataset inputted. Because the hash table has a complexity of O(N), computational time will scale linearly as the size of the packages grows. Additionally, as the number of packages increases, the hash table will need to allocate additional memory to store each package’s data. The memory overhead will also scale linearly as the number of packages increases. Network overhead is not an issue because this is a local application, not using any bandwidth. If the number of packages, or the size of the dataset, were to grow into the tens of thousands, the linear scaling would become problematic in terms of computational and memory overhead. Performing a simple look-up or adding a key-value pair would take dramatically longer, scaling with the size of the number of packages. At this point, it is recommended to reprogram the data structure to use a smaller complexity, such as O(log N).

K1C. Implications

The program has an overall complexity of O(N^4), exponentially increasing computational requirements as datasets grow very large. Small changes to the dataset do not affect computational requirements & usage for the data structure dramatically, but large changes are not advised. As the number of packages grows, it will affect the program linearly because the data structure (hash table) has a complexity of O(N). This is not problematic until the number of packages grows extremely large, because a simple addition of a package or a package look-up could take the same amount of time as the size of the dataset. As for the trucks, adding another truck would result in the exponential growth of computational and memory requirements. When the business is small, or the count of trucks is low, this is not a problem. However, as the

business grows and the number of trucks increases, the implications of high complexity will be dramatic. The deliver package logic has a complexity of O(N^4), so adding a truck would exponentially increase the computational requirements. If the business is expanding to multiple cities concurrently, it is advised to run the program on multiple systems in parallel to decrease the computational requirements of each program. If infeasible, then it is advised to fund more efficient and optimized programs to better adhere to business requirements.

K2. Other Data Structures

Other data structures that would've satisfied the same requirements for the business are trees and stacks.

K2a. Data Structure Differences

Stacks could've been used to satisfy the same solution. Stacks are different from hash maps by functioning with one dimension, identified by its position rather than a key. Stacks would've required more computational resources for look-up functions because their position is dynamic, while hash maps are instant if we know the key.

Another data structure that could satisfy the same solution is a tree. The binary search tree can be organized and sorted by attributes, and accessed by traversing through the tree compared to a hash map where we access through a key. Trees could be traversed and organized by attributes solely, removing the need for a primary key.

M. Professional Communication

All code segments have comments explaining logic in Python files.

L. Sources - Works Cited

"Bellman–Ford Algorithm | DP-23 - GeeksforGeeks." GeeksforGeeks, Dec. 2012, www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/.

"Find Shortest Paths from Source to All Vertices Using Dijkstra's Algorithm." *GeeksforGeeks*, 22 June 2023, www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/.