C950 WGUPS Algorithm Overview

Wyatt Durbin

ID #002184855

WGU Email: wdurbi5@wgu.edu

04/25/2023

C950 Data Structures and Algorithms II

# Introduction

The purpose of this program is to provide a solution to WGU Postal Services package delivery needs. The program loads three trucks and finds an optimized route for them to delivery their packages. Special delivery notes, such as time delays are also considered and accommodated using conditional statements when the trucks are loaded. Dijkstra’s Algorithm is used to sort packages based on distance information contained in a weighted graph. A basic but effective command line interface provides the user a means of interactive with the program.

# A. Algorithm Identification

The program uses Dijkstra’s Algorithm, which uses a weighted graph comprised of nodes and edges. Each node is a delivery address, and each edge is the numerical distance between each node. The package, address, and distance information is parsed into the program using comma separated values (CSV) files. The Dijkstra function in the program is named “dijkstra\_shortest\_path” and has two parameters: graph, and truck. The graph parameter takes the weighted graph of the truck being passed into the function, and the truck parameter is the truck object. The algorithm evaluates each package via a recursive loop to find the closest address to the trucks current location and sends the truck to that location. Mileage and the time it took for the truck to get to the location is recorded. At the end of the algorithm, the truck is returned back to the hub.

# B1. Logic Comments

def dijkstra\_shortest\_path(graph, truck):

undelivered\_package = []

for ID in truck.packages:

package = PackageTable.search(ID)

if package.ID == 9 and truck.time > time(hours=10, minutes=20, seconds=00):

package.address = ‘410 S State St’

package.zip = ‘84111’

package.address = address\_ID\_dict.get(package.address)

ubdelivered\_package.append(package)

truck.packages = []

truck.address = 0

while length(undelivered\_package) > 0:

smallest\_dist = 0

for i in range(1, length(undelivered\_package)):

if graph.edge\_weights[truck.address], undelivered\_package[i].address)] <

graph.edge\_weights[truck.address, undelivered\_package[smallest\_dist].address]

smallest\_dist = i

truck.time += time(hours=graph.edge\_weights[truck.address,

undelivered\_package[smallest\_dist].address) / 18

truck.mileage = truck.mileage + graph.edge\_weights[truck.address, undelivered\_package[smallest\_dist].address

truck.address = undelivered\_package[smallest\_dist].address

undelivered\_package[smallest\_dist].delivery\_time = truck.time

undelivered\_package[smallest\_dist].status = ‘Delivered: ‘

undelivered\_package[smallest\_dist].address = address\_dict.get undelivered\_package[smallest\_dist].address)

truck.packages.append(undelivered\_package[smallest\_dist])

undelivered\_package.pop(smallest\_dist)

truck.time += time(hours= graph.edge\_weights[truck.address, 0] / 18)

truck.mileage += graph,edge\_weights[truck.address, 0]

truck.address = ‘4001 South 700 East’

# B2. Development Environment

During development of this application, the following software and hardware was used:

Python 3.9

PyCharm Community Edition 2022.3.2

Windows 11 Pro Version 22H2

Processor: 11th Gen Intel(R) Core(TM) i5-11400F @ 2.60GHz

RAM: Vulcan Z 16.0 GB DDR4 3600MHz

Drive: Western Digital WD Green SN350 NVMe M.2 2280 1TB

Graphics Card: RTX 3060TI 8GB

# B3. Space-Time and Big-O

This program has a time complexity of O(N^2). This time complexity is shared with the sorting function “dijkstra\_shortest\_path”, which has a while loop of length

Main.py

|  |  |  |
| --- | --- | --- |
|  | CPU Time | Memory Space |
| load\_package\_date() | O(n^2) | O(n^2) |
| dijkstra\_shortest\_path() | O(n^2) | O(n) |
| add\_vertex() | O(n) | O(n) |
| add\_edge() | O(n^2) | O(n) |
| delivery\_update() | O(n) | O(1) |
| Main | O(n) | O(1) |
| Main.py | O(n^2) | O(n^2) |

PackageHashTable.py

|  |  |  |
| --- | --- | --- |
|  | CPU Time | Memory Space |
| \_\_init\_\_() | O(1) | O(1) |
| insert() | O(n) | O(n) |
| search() | O(n) | O(1) |
| remove() | O(1) | O(1) |

Graph.py

|  |  |  |
| --- | --- | --- |
|  | CPU Time | Memory Space |
| \_\_init\_\_() | O(1) | O(1) |
| add\_vertex() | O(1) | O(1) |
| add\_directed\_edge() | O(1) | O(1) |
| add\_undirected\_edge() | O(1) | O(1) |

Vertex.py

|  |  |  |
| --- | --- | --- |
|  | CPU Time | Memory Space |
| \_\_init\_\_ | O(1) | O(1) |

Truck.py

|  |  |  |
| --- | --- | --- |
|  | CPU Time | Memory Space |
| \_\_init\_\_ | O(1) | O(1) |
| \_\_str\_\_ | O(1) | O(1) |

Package.py

|  |  |  |
| --- | --- | --- |
|  | CPU Time | Memory Space |
| \_\_init\_\_ | O(1) | O(1) |
| \_\_str\_\_ | O(1) | O(1) |

# B4. Scalability and Adaptability

This program was designed with scalability and adaptability in mind. Truck packages can easily be added, deleted, and removed within the “load\_package\_date()” function. Packages are automatically read from the packageFile and created into a package object. The sorting algorithm in “dijkstra\_shortest\_path works independently of package size, so it can theoretically be scaled indefinitely to the number of packages. We also can easily adjust the capacity of the truck object, in the case WGUPS purchases trucks with larger capacity (Such as the capability to carry 24 packages, rather than 16.), the program can easily be modified by simply increasing the truck object “capacity”. The sorting algorithm will automatically adapt, and sort based on the trucks new capacity. One shortcoming of the program however is that the packages do mostly need to be manually loaded onto the trucks. There is logic that will read the notes associated with the package and automatically assign it to a truck. (Such as a package reading “Can only be on truck 2”), but the package loading is partially manual still.

# B5. Software Efficiency and Maintainability

This delivery solution that is efficient due to the sorting algorithm, package information, delivery and distance data, automatically scaling to any size read into them. The program doesn’t require much maintenance, but does require the program user to partially manually load the packages into the truck, which means the program needs adjusted each day to the packages being delivered.

# B6. Self-Adjusting Data Structures

The data structure implemented in “PackageHashTable.py” has a current capacity of 20, which can easily handle the current package count of 40. The has table offers excellent time complexity speeds, which is a strength of hash tables. If the program starts to take on especially large package courts, such as 100, 200, etc, then the PackageHashTable class can easily be adapted, with the downside that the code would need an edit to increase the capacity to the desired size.

# C. Original Code

The code is in the same archive as this word document.

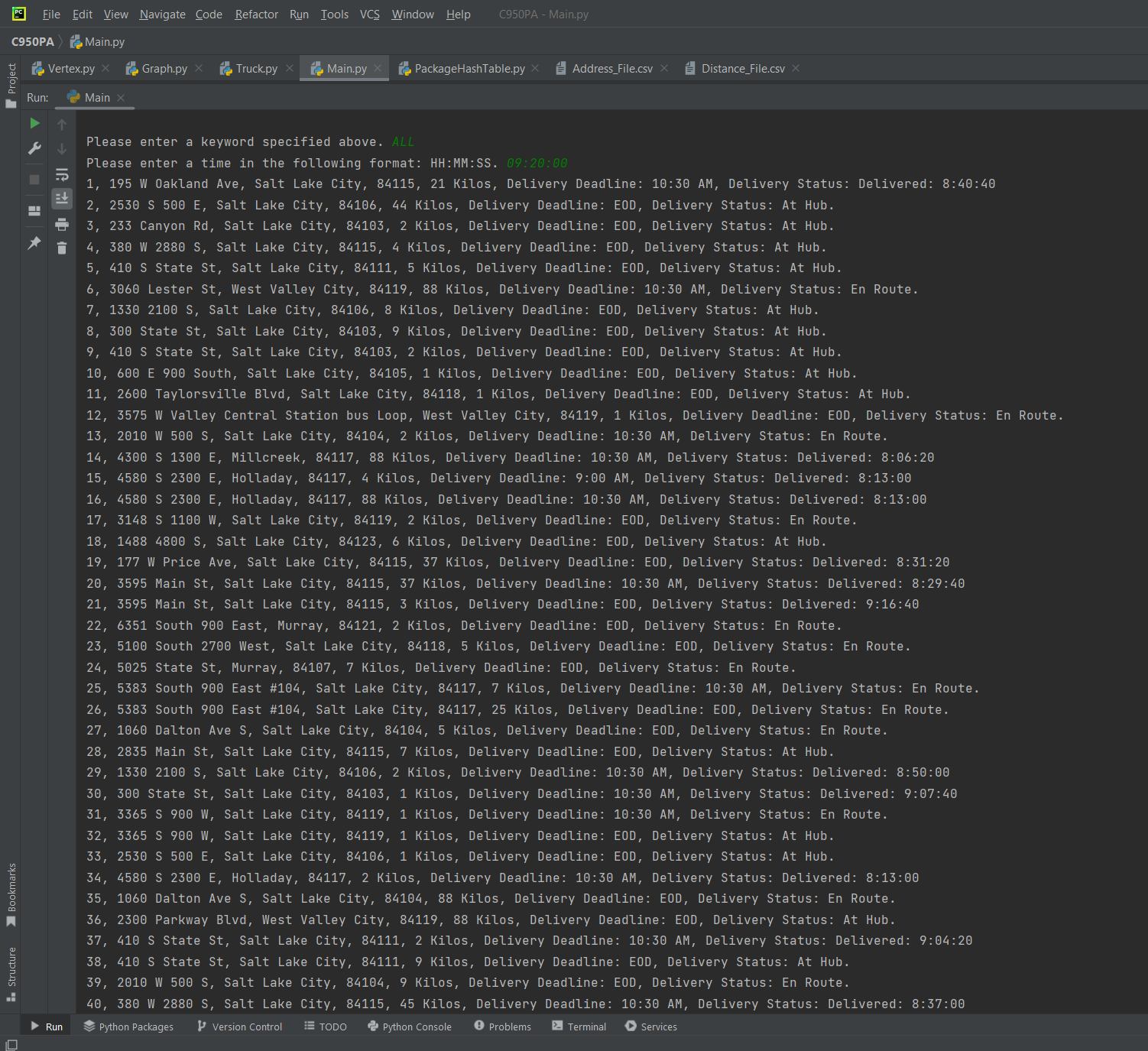
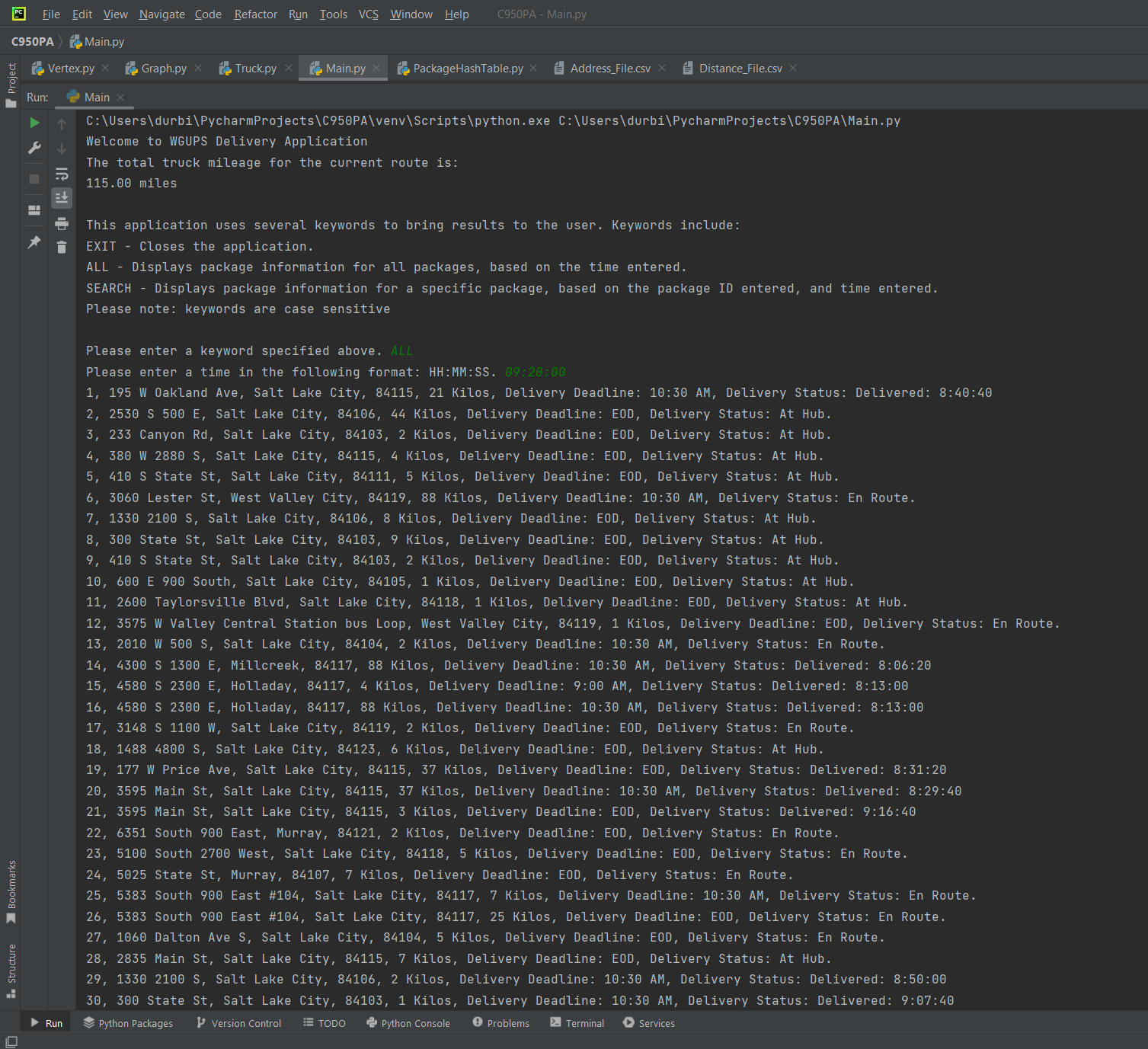
# D. Data Structure

The data structure this program uses is name “PackageHashTable” and is a hash table data structure.

# D1. Explanation of Data Structure

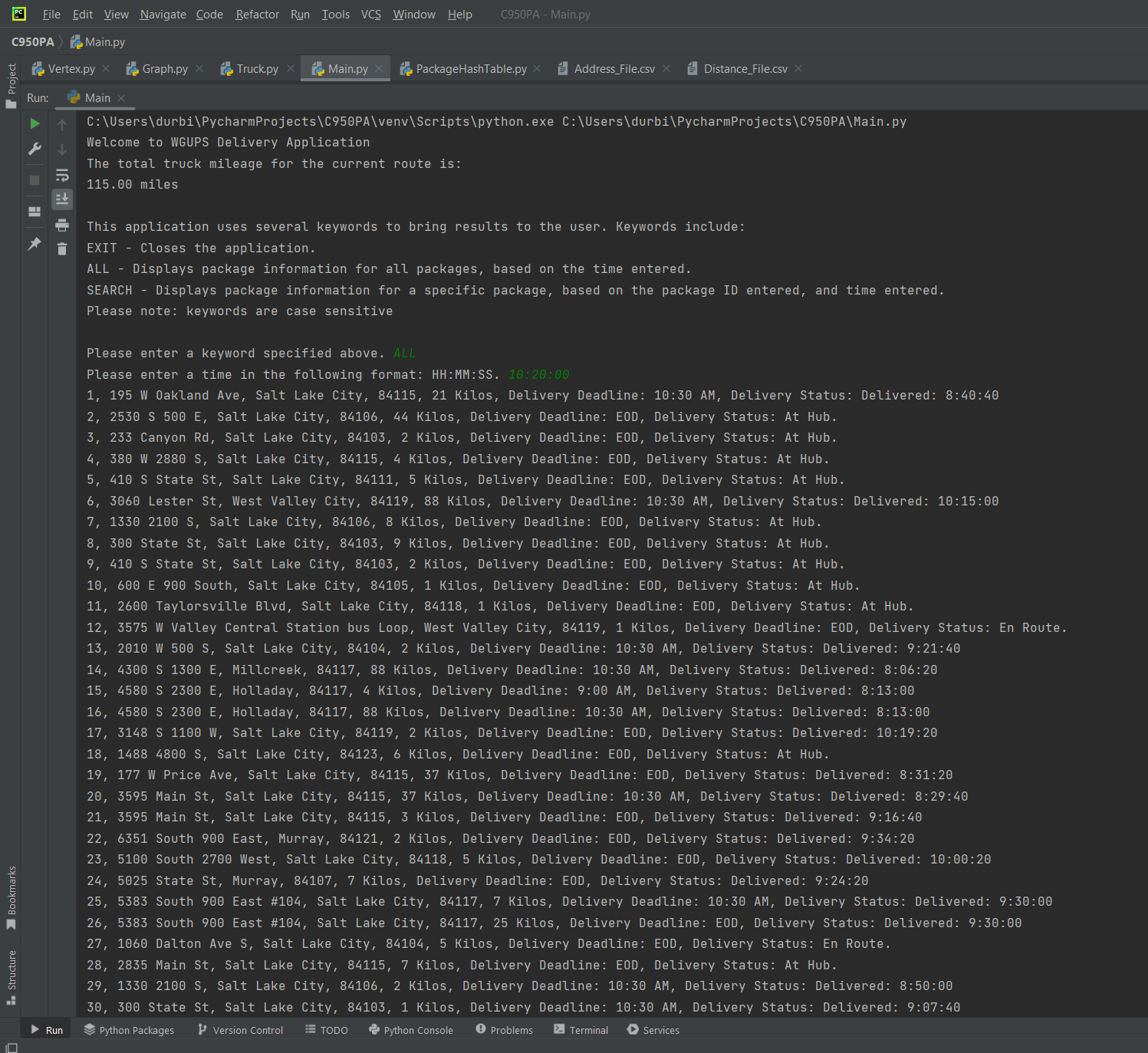
For my has table, I stored package object, created via the Package class. The package ID was used to store the package into the hash table, in a bucket, being converted into a key via modulo in the insert() method. The package object was then stored as the bucket’s item. This approach allows each object to retain it’s integrity and not get package data points mixed up.

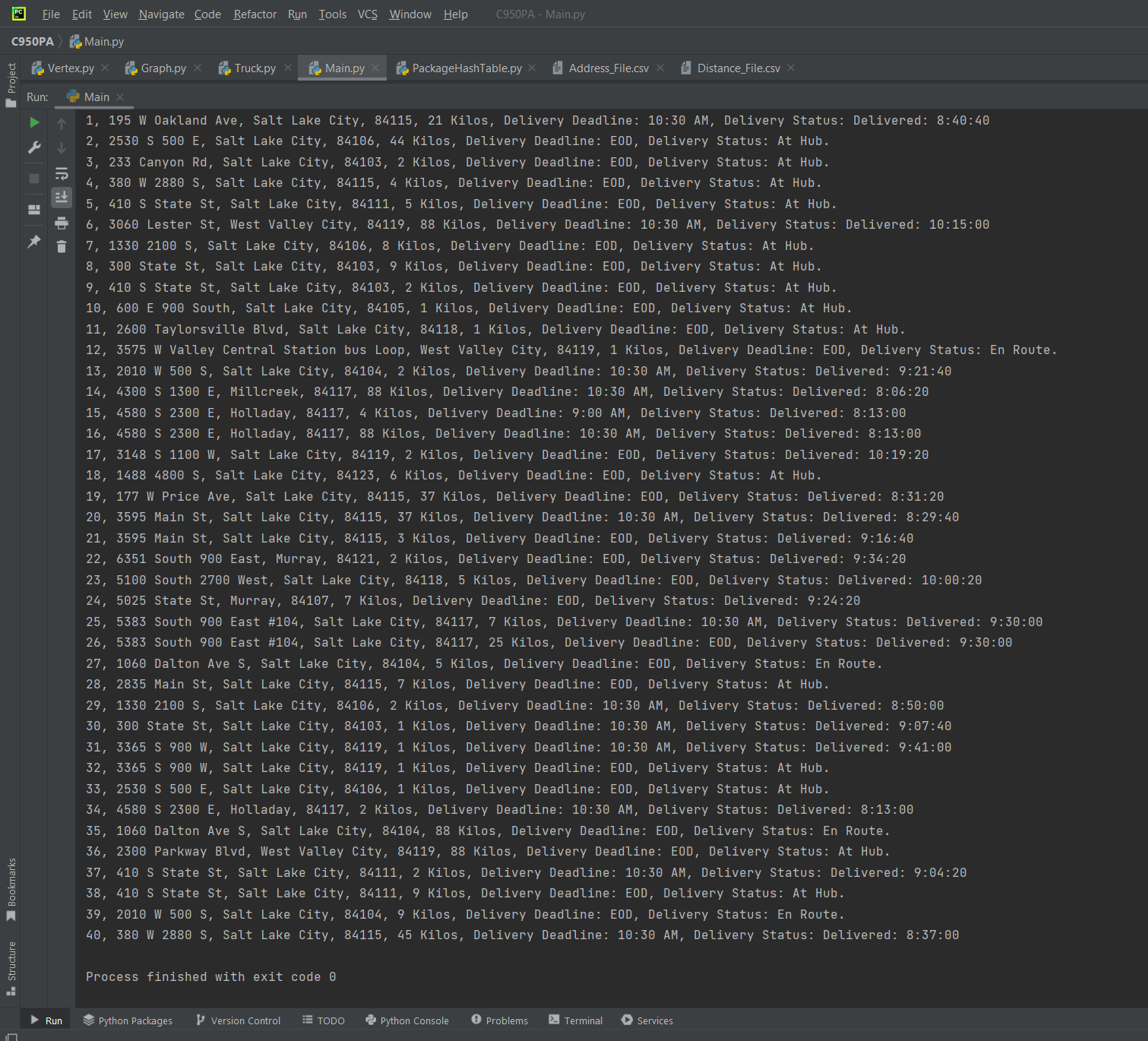
# G1. First Status Check

09:20:00 Status Check

# G2. Second Status Check

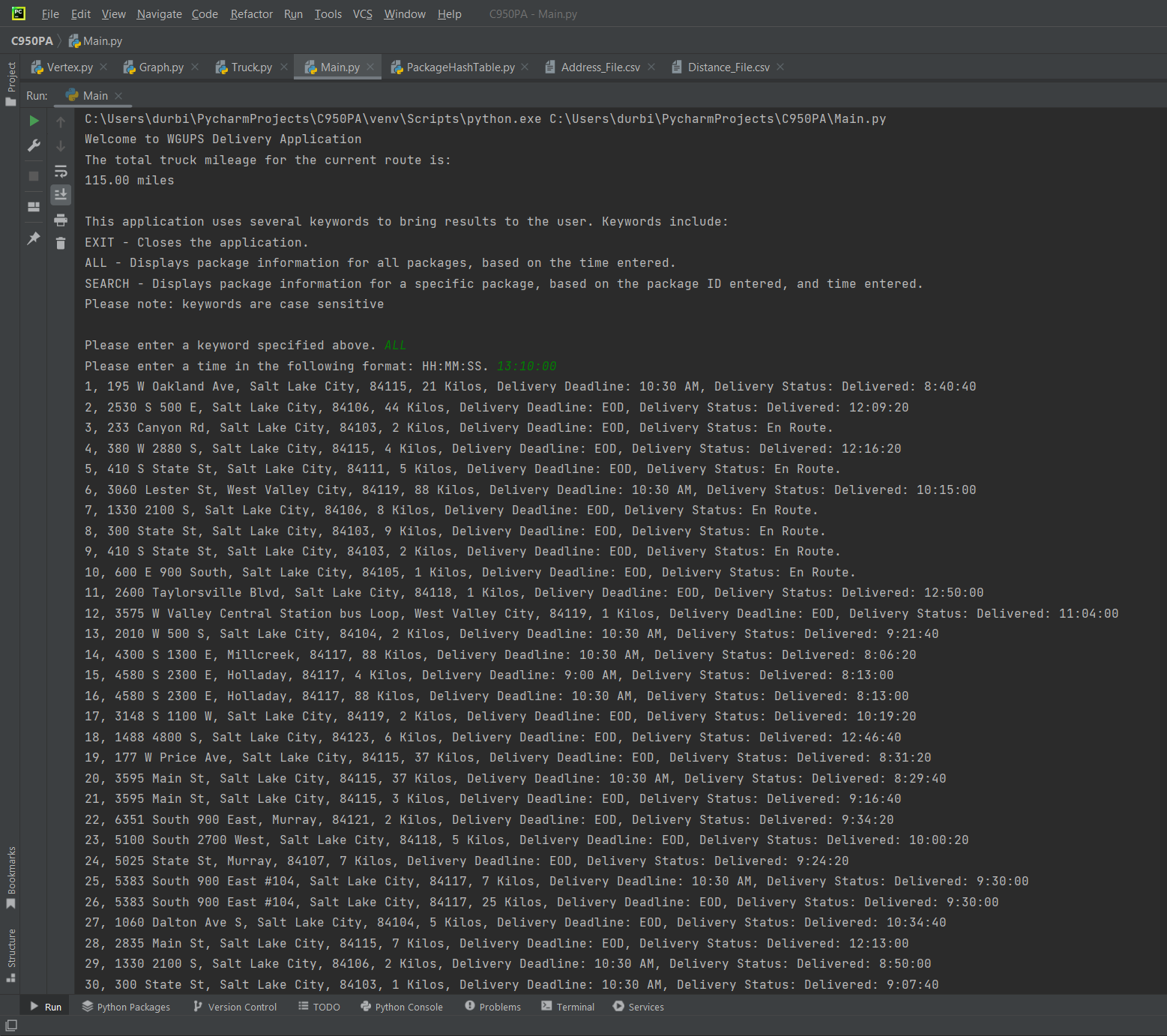
10:20:00 Status Chcck

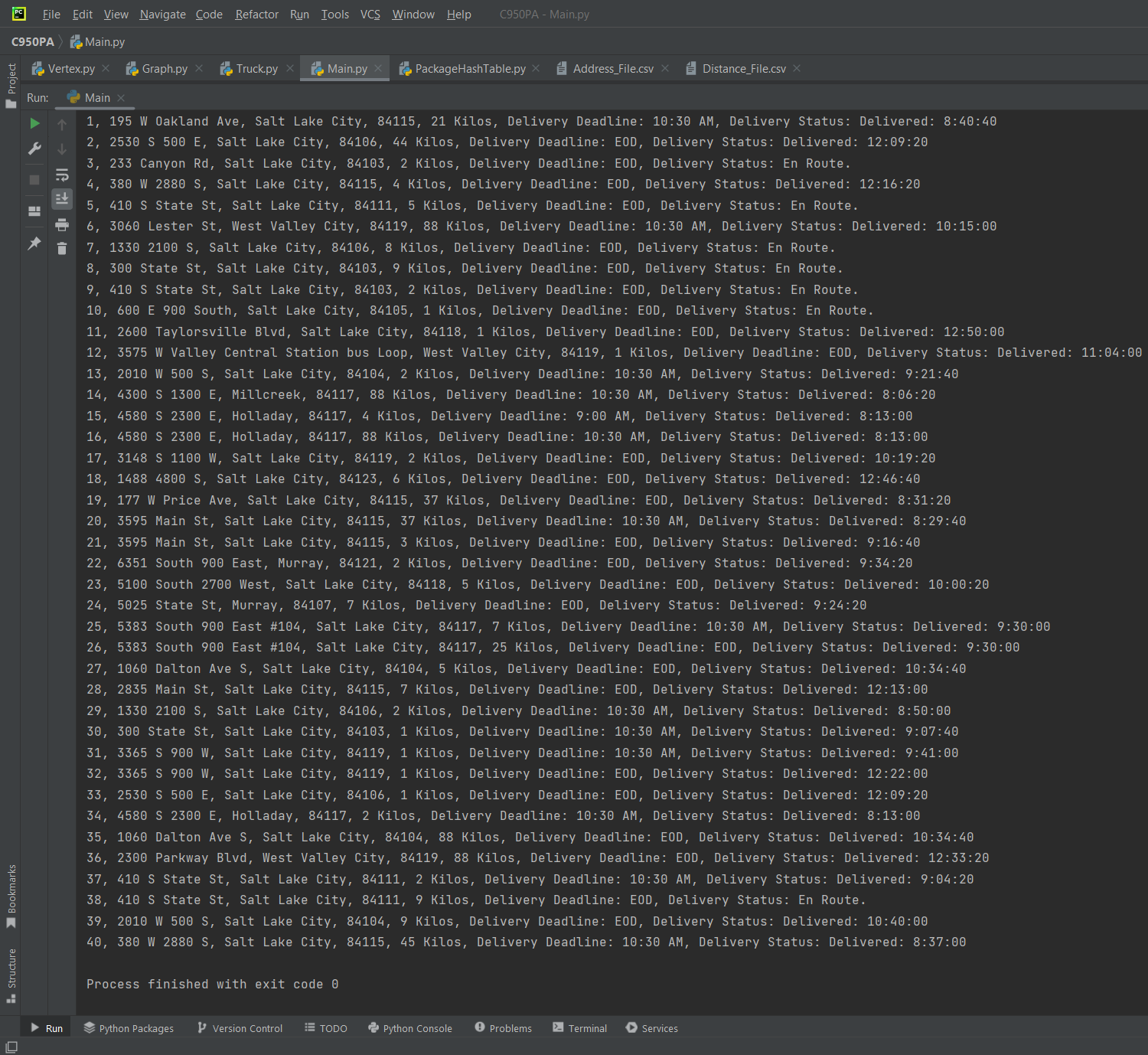




# G3. Third Status Check

13:10:00 Status Check





# H. Screenshots of Code Execution

Command line interface output of total truck mileage.

Text

Description automatically generated

# I1. Strengths of Chosen Algorithm

The Dijkstra’s Algorithm I implemented was chosen due to the easily comprehensible nature of a weighted graph, and how easily a weighted graph can be related to the variety of package address locations and distances between them. With Dijkstra’s Algorithm being a variation of a greedy algorithm, it offers a decently optimized route that manages to get the total truck mileage to 115 miles.

# I3. Other possible Algorithms

Two alternatives to the Dijkstra’s Algorithm I chose to use would be a brute-force search algorithm and a branch and bound algorithm.

# I3A. Algorithm Differences

Out of the three algorithms, Dijkstra’s Algorithm offers the best overall speed, at the cost of a potentially less optimal route. On the opposite end of the spectrum, we have the brute-force search algorithm. A brute-force search would ultimately give the most optimal route, which could be valuable if truck mileage a priority concern, but this algorithm has a far worse time complexity than either Dijkstra’s algorithm or a branch and bound algorithm (Wikipedia, Brute-force-search). In the middle ground, a branch and bound algorithm could be used. A branch and bound algorithm is a dynamic programming approach that splits the package delivery problem into a set of smaller problems (Zybooks, 3.5). Although this approach will offer a well optimized route, it still falls behind Dijkstra’s Algorithm in worst-case time complexity. Considering the above, I still find Dijkstra’s to be the best solution due to it delivering a decent combined truck mileage of 115 miles, yet having a relatively good worst-case time complexity.

.

# J. Different Approach

If I started the program over, I would consider building a function to load the trucks automatically. At first, I set off to do that, but I found it unnecessary to meet requirements of the program. To accomplish this, I would start with sorting all package destinations and create an ordered list of packages based on their distance from each other. After an ordered list is created, I would implement a function so check for special notes associated with the packages. (Such as searching for the keyword “delayed” ). This would require studying the package list and creating a series of if statements. After evaluating the list, the packages could finally be loaded into the trucks.

# K1A. Efficiency

This program has an overall efficiency of O(n^2). This would remain constant regardless of the number of packages (such as 60, 80, etc). With that in mind, the hash table search method will still grow slower as more items are inserted and the overall capacity is filled. As the hash table is filled, we will approach a package search efficiency of O(n) due to collisions becoming more likely, rather than an efficiency of O(1), which occurs when the table has no collisions.

# K1B. Overhead

The change in package count does have a direct impact on the data structure’s memory usage. The data structure has an O(n) memory space usage, so as the number of packages increased, so will the memory used at the rate of n.

# K1C. Implications

The number of trucks or cities doesn’t impact the data structure CPU time or memory space, since the data structure simply stores the package object, and doesn’t involve in where the package is going/what truck it is going on. Memory space usage only increased when n packages increases, not when the number of trucks are increased.

# K2. Other Data Structures

Two data structures I could have implemented, other than a hash table, would be a binary search tree, and a linked list.

# K2a. Data Structure Differences

Out of the three data structures, a binary search tree (bst) would offer the best adaptability to fluctuating number of packages. A hash table however offers a better search time complexity when compared to a bst, and even more so when compared to a linked list. Considering this program was built with the 40 packages in mind, and frequent item searching, a hash table was still the best data structure solution when compared to a binary search tree and a linked list.

# L. Sources - Works Cited

Wikipedia contributors. (2022, November 23). Brute-force search. In *Wikipedia, The Free Encyclopedia*. Retrieved 23:41, April 30, 2023, from

<https://en.wikipedia.org/w/index.php?title=Brute-force_search&oldid=1123297599>

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

Retrieved April 30th, 2023, from https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/3/section/5