Julian Chavez
Student ID: 000966293
C950: DS&A II

Objective Overview:

WGUPS is a shipping company that needs a program that determines the daily route for three truckloads of packages, optimized for least mileage. The program is restricted to using Python 3, as well as only built-in libraries. There are 40 packages to be delivered by two drivers. Some packages have special conditions that must be addressed, such as delivery times, delayed shipments, and load priority groups of packages. The packages are required to be loaded systematically into a custom Hash Table, and the algorithm must optimize a mileage efficient route. The user must be able to look up packages and their information as well.

Section A:

The algorithm I used for finding an optimal path is based on a common algorithm called a 3-opt algorithm. The algorithm is able to take in a variety of route lengths and find an optimal path. The algorithm is also able to adjust to immovable path stops, as in starting at the Hub is required and necessary. Every shipment will leave from the Hub, so that stop in the path is not movable, similarly for priority shipments.

Section B1:

First, the program must distribute the packages into truck loads. There is a significant amount of efficiency at stake in this process. After several failed iterations in my project I was able to finally develop more optimal heuristics to guide the loading process. Essentially, the process is guided by geographic zip codes. However, most importantly, the packages have to be delivered according to the special restrictions- delivery time, delayed shipments, package groups, and others.

Step 1: Create the Address Table

- In this step, the locations and distance data are processed into usable lists for creating routes, calculating distances, and being able to associate packages with that data. Locations are of type Vertex.

Step 2: Create the Package Hash Table

- Package data is processed and entered into a hash table for efficient access

Step 3: Process Packages with conditions

- Iterate through the hash table to find packages with conditions
- Packages with load priority, as in those that have to be grouped with other packages, #13,#14,#15,#16,#19,#20, are associated with Truck 1
- Delayed packages with delivery times are associated with Truck 2

- Delayed packages without delivery times are associated with Truck 3

Step 4: Process Packages with delivery times

- Iterate through the hash table to find packages with delivery times
- Packages are first distributed into trucks based on their zip codes
- Each truck serves an area with associated zip codes that are geographically near
- Some zip codes are shared between trucks so more heuristics are involved that determine where the packages are associated
    - These truck associations were based on the proximity of the addresses to zip codes that had clear associations with trucks 1-3

Step 5: Process remaining Packages

- Iterate through the hash table to find packages without truck association
- Packages are first distributed into trucks based on their zip codes
- Packages are then combined with other truck associated packages with the same address
- Packages are then distributed to the shared zip codes similarly to the previous step
    - Some of these packages are run through the optimizing algorithm

Step 6: Finalizing the preload

- When the packages are distributed to preloads, a list of IDs and list of Vertex objects are created.
    - The ID list make it simple to debug unwanted behaviors and inefficiencies
    - The Vertex list becomes the route that will be optimized
- The truck routes are shuffled.
    - I found through testing that this step tended to provide better results
- Custom optimizations
    - I found through testing that this step tended to provide better results
    - Certain stops in the route are set to not be swappable by the optimizer
- Routes are then attached to the Hub and then optimized
- The packages are now ready for the program to use
- Delivery times are based on the distances along the route

The second part is the actual optimizing algorithm. Through trial and error I finally settled on a 3-opt algorithm combined with simulated annealing.

Step 1: A Route is sent to the ThreeOpt class

- The ThreeOpt class takes a route and a two-dimensional list of distances

Step 2: Subsets of the Route

- Three subsets of the route are taken, distances are calculated, and then the subsets are compared against each other in different combinations to find a more optimal route

Step 3: Step 2 is repeated many times to capture all possible subsets

- With the introduction of non-swappable locations(HUB, priority packages), the algorithm was not able to always cope. I had to also make a counter that serves as a time-out flag in case the algorithm is stuck in an infinite loop.

Step 4: Simulated Annealing

- The purpose of simulated annealing is to break up local minimums in pursuit of finding a better global minimum. The nature of greedy algorithms is that they highly prioritize local minimums, sometimes at the expense of global optimization. Simulated annealing uses randomization to find indices not usually associated in greedy algorithms.
- The random indices are swapped and compared to find potentially better arrangements.
- This is repeated many times until the initial “temperature” of the route has cooled
- In the small truck load sizes required by the project, annealing is borderline unnecessary, however there were times that it did find more optimal solutions.
- The benefits of annealing were even more apparent in early testing of my project when I was testing optimal routes of all 40 packages. So I would expect that my solution is beneficial and scalable to larger applications.

Section B2:

It appears that a previous iteration of this project may have involved reading package information from a server. Admittedly it would have made more sense for that to be the case. For this project we are required to read in provided distance, location, and package data from Excel files. We are also required to use Python 3. Python makes reading the files a simple task. Excel files can be easily converted into CSV files. Python then can read the CSV files straight into multi-dimensional lists, ready to use.

Section B3:

The program has the time complexity in the comments above each major method.

Section B4:

The three-opt algorithm performs better with larger sets than the current project requirement. When I tested this algorithm along a route consisting of all 40 packages, it out-performed other algorithms that I had tried previously. In combination with the simulated annealing, I believe my algorithm will scale well with larger data sets. The project requirements necessitated close proximity between the segment routes derived from the algorithm and unswappable elements. The effect of this was that many segment routes were not able to be swapped even if they were more efficient. In larger data sets, the simulated annealing will be able to find more local minimum clusters to break down and swap.

Section B5:

Julian A. Chavez
I.D: 000966293

This program runs in polynomial time. The most complex function in the program is the three-opt algorithm that runs at $O(n^3)$ time-complexity. Most other functions run at O(n) or better. The hash table data structure will typically run in constant O(1) time.

This program will allow custom optimization in the form of specifying zip code-truck associations. For larger data sets the optimization algorithm can easily have new functions added that can modify desired package pathing. For instance, the current the program will sort a remainder package based on a least distance optimization among the 3 trucks. In a larger data set, a more accurate comparison might involve a least average distance modifier.

With the documentation and comments the code should be fairly simple to understand by other developers. In the future I might make an effort to streamline certain processes, such as reducing the amount of separate lists for packages, routes, and IDs. It would also be a beneficial improvement to design the code more pythonically. Previously my experience was in Java, so many of those practices dictated my approach to this project.

Section B6:

The self-adjusting data structure used is a simple Hash Table implementation. This hash table is able to adjust its size to the initial delivery total for the day, making resizing unnecessary unless the fictional WGUPS business practices were to change. This Hash Table can also adjust for collision. If there is a collision, which there should not be under current project requirements, the put() function will append a second key/value pair to the index location.

Section C:

The project is all original code. I did adapt or take inspiration from sources listed below in Section L in more detail. The hash table class, vertex class, and simulated annealing function was adapted from Udemy course by Holczer Balazs. The three-opt algorithm was adapted from the wikipedia article on that type of algorithm.

Requirements have been met as far as my understanding. The user can look up package information at many points during the run. Performance tends to result in a final mileage between 68-74 miles.

Section C1:

See Main.py

Section C2:

All major/important blocks of code have comments explaining the function. Uncommented blocks are either unimportant or seldom used.

Section D:

See Section B6 above.

Section D1:

See Section B6 above.

Julian A. Chavez
I.D: 000966293

Hash tables are used when the developer needs to access data in constant time, and when the user will essentially always use a key to refer to the desired data. Keys are run through a hash function that maps a key to a particular index. The data structure Hash Table in this program uses a package ID as key and a package object containing package data as a value. The package ID as key greatly simplifies the project required ability to lookup data. The Hash Table is a simple implementation, similar to many others. It has the typical hash, insertion, and retrieval functions, along with a function to return the size of the Hash Table for iteration purposes in other classes.

Section E:

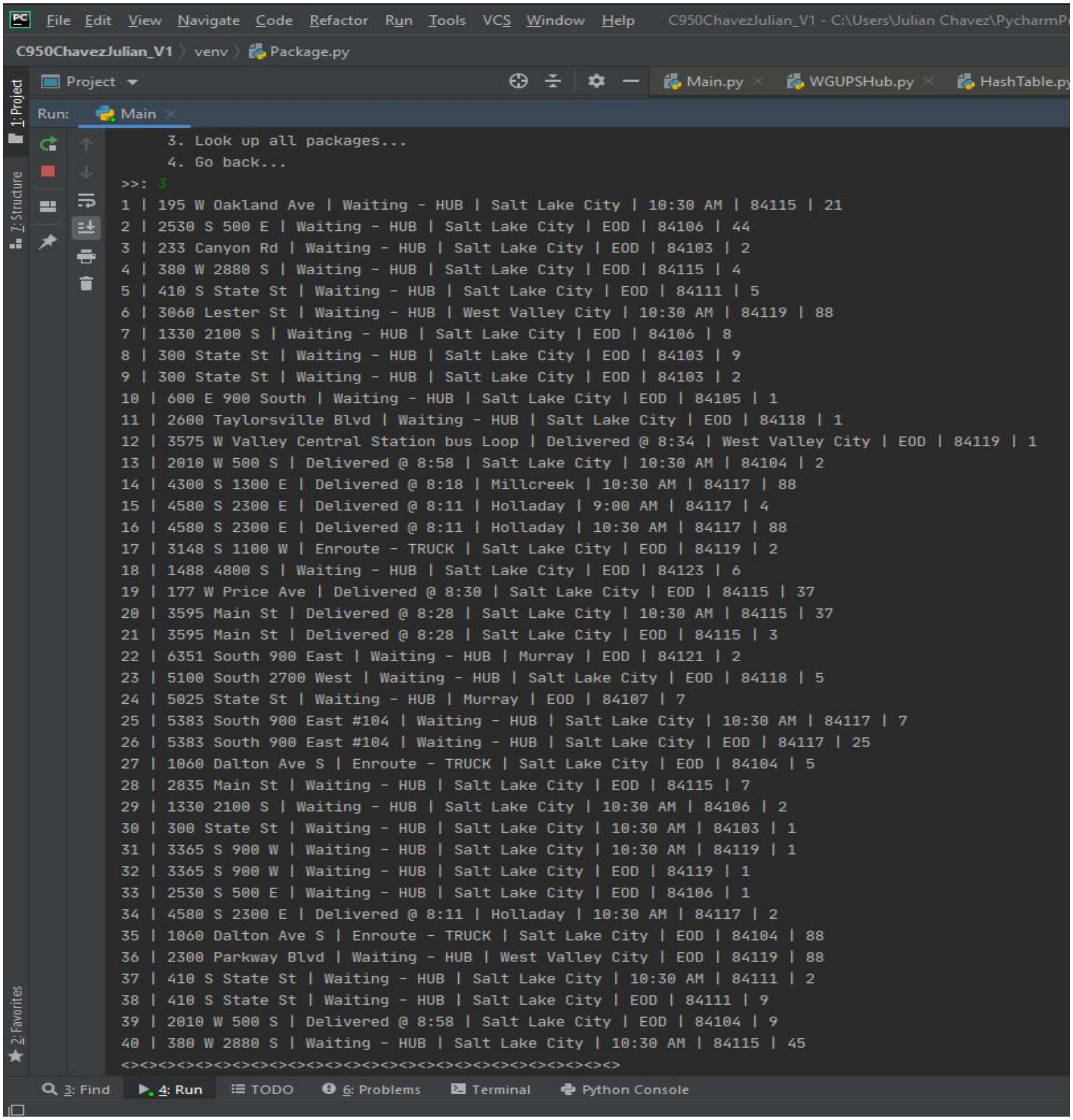
See sections B6, D1 above

Section F:

See sections B6, D1 above

Section G1: 8:35am to 9:25am screenshot

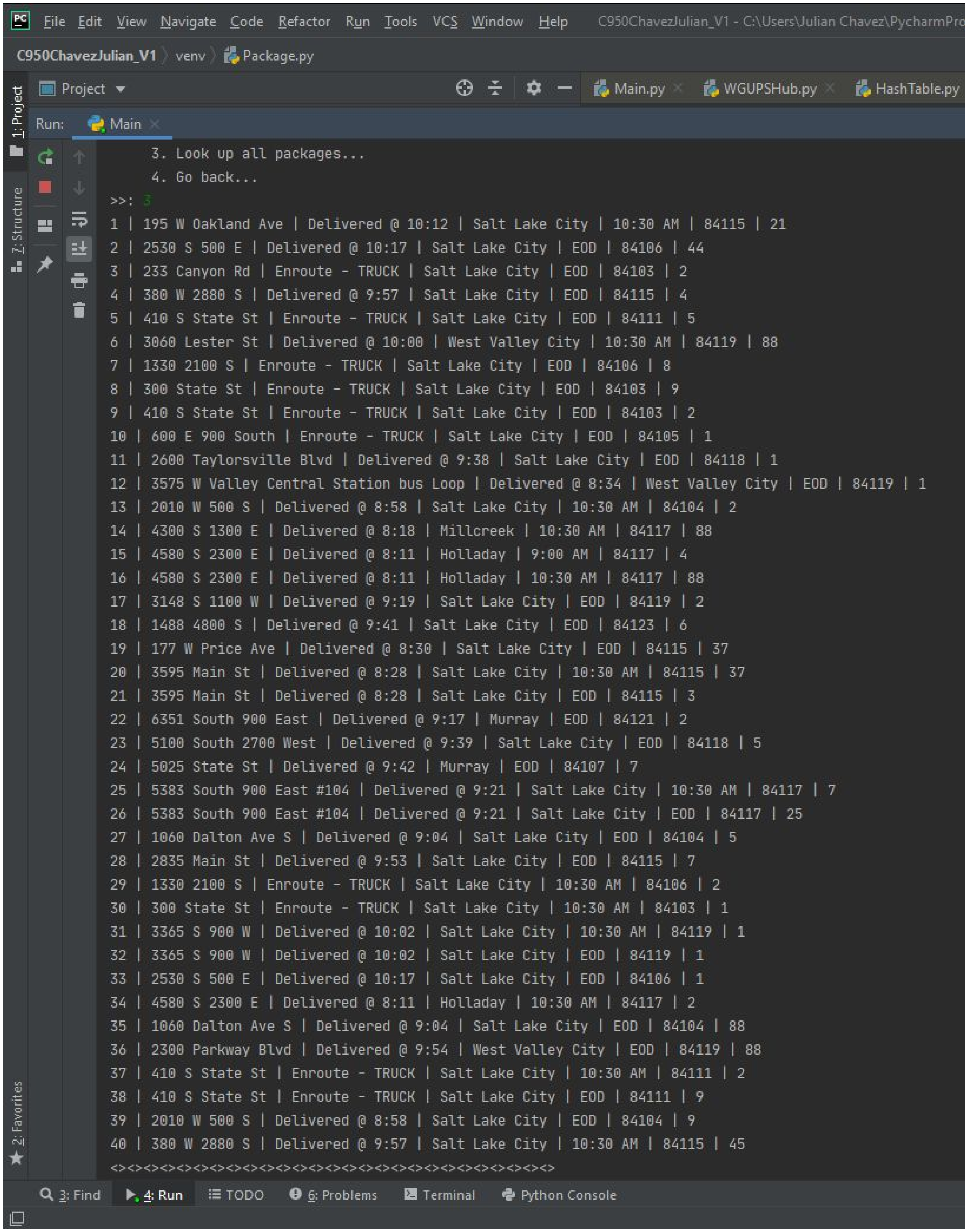
Julian A. Chavez
I.D: 000966293



```
    3. Look up all packages...
    4. Go back...
>>: 3
1 | 195 W Oakland Ave | Waiting - HUB | Salt Lake City | 10:30 AM | 84115 | 21
2 | 2530 S 500 E | Waiting - HUB | Salt Lake City | EOD | 84106 | 44
3 | 233 Canyon Rd | Waiting - HUB | Salt Lake City | EOD | 84103 | 2
4 | 380 W 2880 S | Waiting - HUB | Salt Lake City | EOD | 84115 | 4
5 | 410 S State St | Waiting - HUB | Salt Lake City | EOD | 84111 | 5
6 | 3060 Lester St | Waiting - HUB | West Valley City | 10:30 AM | 84119 | 88
7 | 1330 2100 S | Waiting - HUB | Salt Lake City | EOD | 84106 | 8
8 | 300 State St | Waiting - HUB | Salt Lake City | EOD | 84103 | 9
9 | 300 State St | Waiting - HUB | Salt Lake City | EOD | 84103 | 2
10 | 600 E 900 South | Waiting - HUB | Salt Lake City | EOD | 84105 | 1
11 | 2600 Taylorsville Blvd | Waiting - HUB | Salt Lake City | EOD | 84118 | 1
12 | 3575 W Valley Central Station bus Loop | Delivered @ 8:34 | West Valley City | EOD | 84119 | 1
13 | 2010 W 500 S | Delivered @ 8:58 | Salt Lake City | 10:30 AM | 84104 | 2
14 | 4300 S 1300 E | Delivered @ 8:18 | Millcreek | 10:30 AM | 84117 | 88
15 | 4580 S 2300 E | Delivered @ 8:11 | Holladay | 9:00 AM | 84117 | 4
16 | 4580 S 2300 E | Delivered @ 8:11 | Holladay | 10:30 AM | 84117 | 88
17 | 3148 S 1100 W | Enroute - TRUCK | Salt Lake City | EOD | 84119 | 2
18 | 1488 4800 S | Waiting - HUB | Salt Lake City | EOD | 84123 | 6
19 | 177 W Price Ave | Delivered @ 8:30 | Salt Lake City | EOD | 84115 | 37
20 | 3595 Main St | Delivered @ 8:28 | Salt Lake City | 10:30 AM | 84115 | 37
21 | 3595 Main St | Delivered @ 8:28 | Salt Lake City | EOD | 84115 | 3
22 | 6351 South 900 East | Waiting - HUB | Murray | EOD | 84121 | 2
23 | 5100 South 2700 West | Waiting - HUB | Salt Lake City | EOD | 84118 | 5
24 | 5025 State St | Waiting - HUB | Murray | EOD | 84107 | 7
25 | 5383 South 900 East #104 | Waiting - HUB | Salt Lake City | 10:30 AM | 84117 | 7
26 | 5383 South 900 East #104 | Waiting - HUB | Salt Lake City | EOD | 84117 | 25
27 | 1060 Dalton Ave S | Enroute - TRUCK | Salt Lake City | EOD | 84104 | 5
28 | 2835 Main St | Waiting - HUB | Salt Lake City | EOD | 84115 | 7
29 | 1330 2100 S | Waiting - HUB | Salt Lake City | 10:30 AM | 84106 | 2
30 | 300 State St | Waiting - HUB | Salt Lake City | 10:30 AM | 84103 | 1
31 | 3365 S 900 W | Waiting - HUB | Salt Lake City | 10:30 AM | 84119 | 1
32 | 3365 S 900 W | Waiting - HUB | Salt Lake City | EOD | 84119 | 1
33 | 2530 S 500 E | Waiting - HUB | Salt Lake City | EOD | 84106 | 1
34 | 4580 S 2300 E | Delivered @ 8:11 | Holladay | 10:30 AM | 84117 | 2
35 | 1060 Dalton Ave S | Enroute - TRUCK | Salt Lake City | EOD | 84104 | 88
36 | 2300 Parkway Blvd | Waiting - HUB | West Valley City | EOD | 84119 | 88
37 | 410 S State St | Waiting - HUB | Salt Lake City | 10:30 AM | 84111 | 2
38 | 410 S State St | Waiting - HUB | Salt Lake City | EOD | 84111 | 9
39 | 2010 W 500 S | Delivered @ 8:58 | Salt Lake City | EOD | 84104 | 9
40 | 380 W 2880 S | Waiting - HUB | Salt Lake City | 10:30 AM | 84115 | 45
<><><><><><><><><><><><><><><><><><><><><><><><><><><>
```

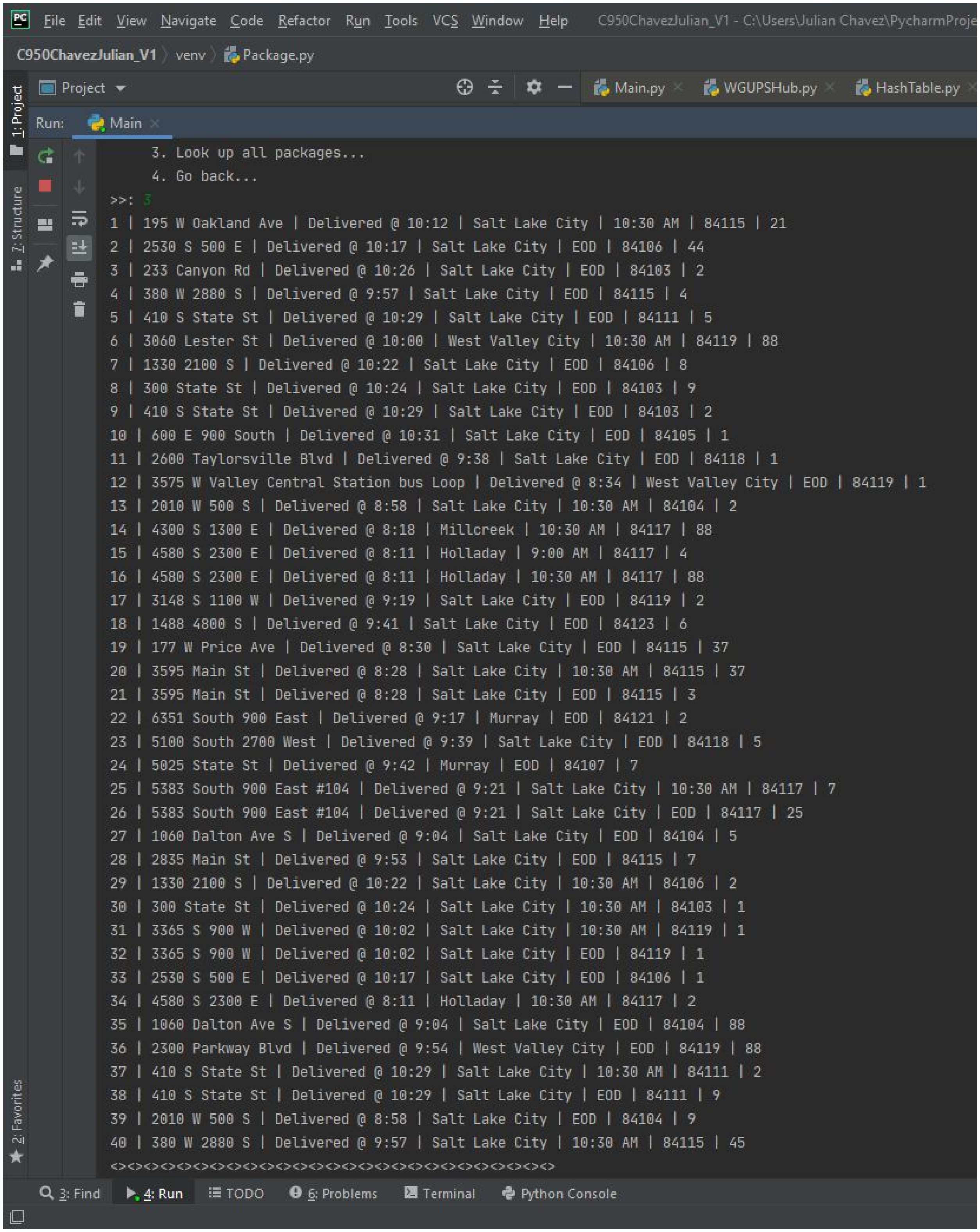
Section G2: 9:35am to 10:25am screenshot

Julian A. Chavez
I.D: 000966293



Julian A. Chavez
I.D: 000966293

Section G3: 12:03pm to 1:12pm screenshot

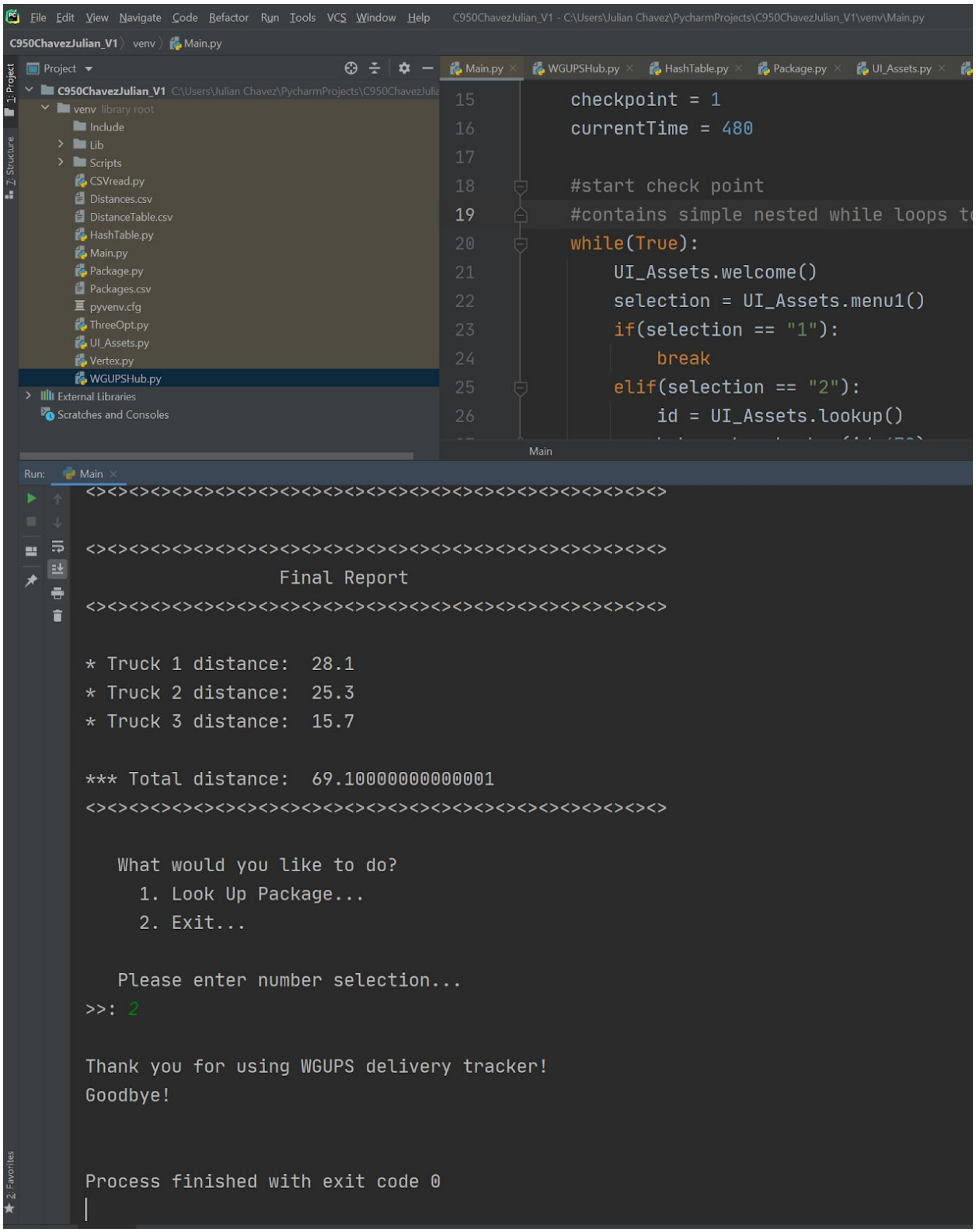



```
    3. Look up all packages...
    4. Go back...
>>: 3
1 | 195 W Oakland Ave | Delivered @ 10:12 | Salt Lake City | 10:30 AM | 84115 | 21
2 | 2530 S 500 E | Delivered @ 10:17 | Salt Lake City | EOD | 84106 | 44
3 | 233 Canyon Rd | Delivered @ 10:26 | Salt Lake City | EOD | 84103 | 2
4 | 380 W 2880 S | Delivered @ 9:57 | Salt Lake City | EOD | 84115 | 4
5 | 410 S State St | Delivered @ 10:29 | Salt Lake City | EOD | 84111 | 5
6 | 3060 Lester St | Delivered @ 10:00 | West Valley City | 10:30 AM | 84119 | 88
7 | 1330 2100 S | Delivered @ 10:22 | Salt Lake City | EOD | 84106 | 8
8 | 300 State St | Delivered @ 10:24 | Salt Lake City | EOD | 84103 | 9
9 | 410 S State St | Delivered @ 10:29 | Salt Lake City | EOD | 84103 | 2
10 | 600 E 900 South | Delivered @ 10:31 | Salt Lake City | EOD | 84105 | 1
11 | 2600 Taylorsville Blvd | Delivered @ 9:38 | Salt Lake City | EOD | 84118 | 1
12 | 3575 W Valley Central Station bus Loop | Delivered @ 8:34 | West Valley City | EOD | 84119 | 1
13 | 2010 W 500 S | Delivered @ 8:58 | Salt Lake City | 10:30 AM | 84104 | 2
14 | 4300 S 1300 E | Delivered @ 8:18 | Millcreek | 10:30 AM | 84117 | 88
15 | 4580 S 2300 E | Delivered @ 8:11 | Holladay | 9:00 AM | 84117 | 4
16 | 4580 S 2300 E | Delivered @ 8:11 | Holladay | 10:30 AM | 84117 | 88
17 | 3148 S 1100 W | Delivered @ 9:19 | Salt Lake City | EOD | 84119 | 2
18 | 1488 4800 S | Delivered @ 9:41 | Salt Lake City | EOD | 84123 | 6
19 | 177 W Price Ave | Delivered @ 8:30 | Salt Lake City | EOD | 84115 | 37
20 | 3595 Main St | Delivered @ 8:28 | Salt Lake City | 10:30 AM | 84115 | 37
21 | 3595 Main St | Delivered @ 8:28 | Salt Lake City | EOD | 84115 | 3
22 | 6351 South 900 East | Delivered @ 9:17 | Murray | EOD | 84121 | 2
23 | 5100 South 2700 West | Delivered @ 9:39 | Salt Lake City | EOD | 84118 | 5
24 | 5025 State St | Delivered @ 9:42 | Murray | EOD | 84107 | 7
25 | 5383 South 900 East #104 | Delivered @ 9:21 | Salt Lake City | 10:30 AM | 84117 | 7
26 | 5383 South 900 East #104 | Delivered @ 9:21 | Salt Lake City | EOD | 84117 | 25
27 | 1060 Dalton Ave S | Delivered @ 9:04 | Salt Lake City | EOD | 84104 | 5
28 | 2835 Main St | Delivered @ 9:53 | Salt Lake City | EOD | 84115 | 7
29 | 1330 2100 S | Delivered @ 10:22 | Salt Lake City | 10:30 AM | 84106 | 2
30 | 300 State St | Delivered @ 10:24 | Salt Lake City | 10:30 AM | 84103 | 1
31 | 3365 S 900 W | Delivered @ 10:02 | Salt Lake City | 10:30 AM | 84119 | 1
32 | 3365 S 900 W | Delivered @ 10:02 | Salt Lake City | EOD | 84119 | 1
33 | 2530 S 500 E | Delivered @ 10:17 | Salt Lake City | EOD | 84106 | 1
34 | 4580 S 2300 E | Delivered @ 8:11 | Holladay | 10:30 AM | 84117 | 2
35 | 1060 Dalton Ave S | Delivered @ 9:04 | Salt Lake City | EOD | 84104 | 88
36 | 2300 Parkway Blvd | Delivered @ 9:54 | West Valley City | EOD | 84119 | 88
37 | 410 S State St | Delivered @ 10:29 | Salt Lake City | 10:30 AM | 84111 | 2
38 | 410 S State St | Delivered @ 10:29 | Salt Lake City | EOD | 84111 | 9
39 | 2010 W 500 S | Delivered @ 8:58 | Salt Lake City | EOD | 84104 | 9
40 | 380 W 2880 S | Delivered @ 9:57 | Salt Lake City | 10:30 AM | 84115 | 45
<><><><><><><><><><><><><><><><><><><><><><><><><><><>
```

Julian A. Chavez
I.D: 000966293

Section H: Code Execution screenshot

- Shows successful execution and final output of trip distance



Julian A. Chavez
I.D: 000966293

Section I1:

The strength of the 3-opt algorithm lies in its segment swapping. Many other, more simple, algorithms rely on finding the nearest neighbor. The 3-opt algorithm compares segments of a route rather than only node to node. Comparing one node to another will tend to focus on local minimization rather than global minimization. The 3-opt algorithm compares segments that consist of multiple nodes to find more optimal solutions. Simulated annealing also assists in breaking up local minimums that do occur.

Another strength of the 3-opt algorithm is that it is suited for this project in particular. If the number of stops were high, the 3-opt algorithm would be an inefficient solution. However, in this situation, the 3-opt performs at an acceptable time cost with a better solution than 2-opt and other similar algorithms.

Section I2:

See sections H, G3

Section I3/A:

In previous iterations of this project, I used a minimum spanning tree and Prim's algorithm. This algorithm was adequate when using a route consisting of all 40 packages, but there was difficulty adapting it to the truckload limit, as well as being able to optimize a solution that begins at the Hub rather than a general solution without the Hub. With more work it would be possible to make this algorithm work correctly to meet project requirements. This algorithm uses vertices that contain an adjacency list that contain its proximity to other vertices. Then this algorithm essentially functions as a nearest neighbor/greedy algorithm where after the first vertex is chosen, each successive and nearest vertex is added to the tree until every vertex has been visited.

Another similar algorithm is a 2-opt algorithm. Where the 3-opt algorithm uses 3 segments of the route, the 2-opt algorithm uses 2 pairs of vertices for comparison and then proceeds similarly. Both of these alternative algorithms have better time complexities, however, the project requirements dictate a relatively small data set, so the difference is insignificant.

Section J:

As mentioned before, I was not as well acquainted with Python when beginning this project, so my knowledge of Java heavily influenced the design of this program. In a future iteration I would design this program more pythonically. I would also streamline my approach in handling the package data and find a solution that engaged the Hash Table more effectively. The nature of the project allowed me to heavily optimize the loading algorithm toward this particular data set. So in theory, if this data set does not represent a typical day at WGUPS, then the loading algorithm may not prove efficient. Some of my design considerations did take this into account, and future users could conceivably optimize to the day without trouble. My optimization also depends on minimizing the total distance as much as possible so that priority

packages all arrive on time without overly burdening the loading algorithm. My solution may not be optimal or deliver packages on time depending on the location of delivery addresses.

Section K1:

See section H, G3

Section K1A:

See sections D1, B6 above

The Hash Table allows constant time access which improves the program and makes lookups extremely simple. Using the package id as a key gives direct access for the look up, and using a package object as a value allows the use of methods on package data.

Section K1B:

Hash Table operational complexity

Algorithm	**Average**	**Worst case**
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

The project constraints eliminate the need to address bandwidth or memory considerations. If the project required communication over a network then there would be some consideration. An inefficient program can slow down performance for users and impact other functions downstream.

Section K1C:

The project constraints did not necessitate the ability to add packages, stops, or trucks so there is not functionality to easily do so. If all the packages are known at the beginning of the day, then the hash table will initialize to the size of that excel file. This program can easily handle scenarios where the package total is different than 40. There is no functionality to add packages during the day. Stops/address are similar to packages in that if the data is set at the beginning, the program can handle differing data sets. This program was specifically designed with three trucks in mind, so that would be more difficult to implement in future iterations.

Section K2/A:

Julian A. Chavez
I.D: 000966293

Lists or dictionaries could be used in place of the required Hash Table. Lists would be slower since they require iteration to find values. It would be possible to match indices to package IDs but HashTables are more foolproof and provide more consistent behavior since the hashing function will always match a key to its value. Dictionaries are essentially the python implementation of hash tables. They have similar functionality and provide similar benefits. A dictionary would have performed the same if the project constraints did not preclude its use.

Section L: Sources

Hash Table, Vertex:

Balazs, H. (n.d.). Algorithms and Data Structures in Python. Retrieved August 02, 2020, from https://www.udemy.com/course/algorithms-and-data-structures-in-python/

Three-Opt:

3-opt. (2020, June 03). Retrieved August 02, 2020, from https://en.wikipedia.org/wiki/3-opt

Simulated Annealing:

Balazs, H. (2020, July 09). Learn Advanced Data Structures and Algorithms in Java with Practice. Retrieved August 02, 2020, from https://www.udemy.com/course/advanced-algorithms-in-java/