Vladan Djurkovic

Student ID #000695823

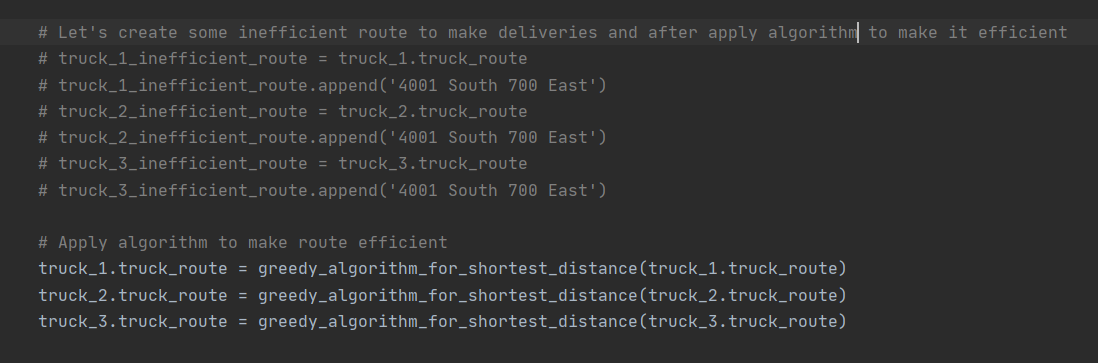
WGU

Date: 08/16/2020

**Data Structures and Algorithms II – task C950**

Algorithm Selection

For this task, my choice of algorithm is a greedy algorithm. I chose this because it is optimal and somewhat an efficient choice to solve a traveling salesman problem by sorting a list of delivery points based on the next closest location. It will probably not give the best possible solution but efficient enough. I named it greedy\_algorithm\_for\_shortest\_distance which gets applied in the deliveries.py file. It makes a previously created inefficient route more efficient, helps deliver packages and meets the requirements.



Logical Comments

All the major sections in the coding part of the project are commented on and explain the logic of each method applied. Here I will elaborate on the greedy\_algorithm\_for\_shortest\_distance. Applying greedy algorithm will help to solve our problem with three delivery trucks by finding the best possible route for each truck. All three trucks will start their deliveries from the warehouse (Hub). Every package will be loaded with its attached address. The algorithm will sort a given truck route in order to find the best possible route based on the next closest location from the current location. For instance, as I mentioned at the beginning, the trucks will start at the HUB- 4001 South 700 East. The starting point will be first added to a route and the algorithm will sort the route and look for a package with the attached address that is the closest to the starting point or the HUB and so on.

The algorithm function will take in a route that is passed to it. It has a starting point and it also has distance between the delivery points imported from the graph.py file. There is an initial route created as the packages are loaded onto truck. Then I created a list better\_route = [starting\_point] which will have a starting point and initially build a better route comparing to the one already given. While loop is applied to make sure that it goes through the passed route until all the locations are removed from the list and the algorithm returns possibly the shortest route. It starts with a distance of 0 (zero) and with a starting point. For loop nested inside the while loop finds the distance between every location from the list of distances. It then eliminates visits to the same delivery location from the route by removing an already visited location from the list. At the end, the result is returned greedy\_path list with the shortest path for a truck to go and deliver packages.

Input to the algorithm -> truck route

Return -> best possible route

Def greedy algorithm for shortest distance

Define starting point to 4001 South 700 East

Define distances by assigning distance {} from graph.py

Define route to sort = truck route

Define and initiate better route [] = starting point

Use while loop to loop through the length of a route until ! = 0

Assign and initiate beginning [] = 0mi and starting point

For loop -> in route to sort

Assign distance and initiate with last element from the better route and add next delivery point

If statements

If look for the shortest distance and one that is not 0 adds next delivery point

Checks if the distance is already in the route and appends it

Remove locations that are already visited to avoid doubles

Return better route sorted

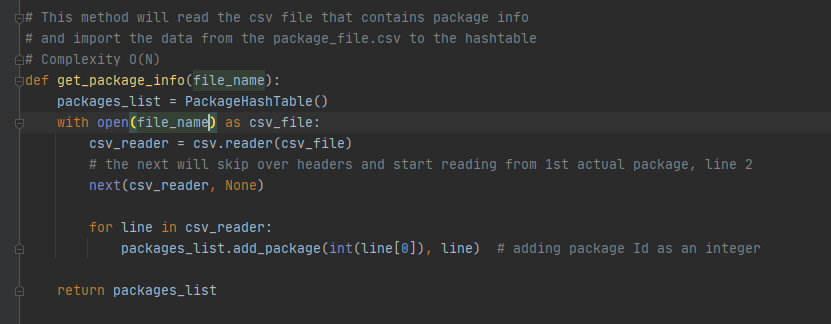
Application of Programming Models

This application runs from the local computer only. Csv files containing data are located inside the project folder. Therefore, there is no protocol that is used to exchange the data. There is no server that is hosting the application and the local machine does not connect anywhere to exchange the data. Therefore, there is no need for the application to disconnect from the server.

The current project is set up as follows: the project data files are stored on the same machine as the one that hosts the application. There are two csv files, distances\_file.csv and package\_file.csv that are stored under the project folder inside the data directory. The entire project is developed inside the PyCharm IDE using Python programing language. Name of the project is TSP\_data\_structures\_and\_algorithms\_II.

Space Time and Big-O

The time complexity is commented inside the project and every major block of code includes the comment where the time complexity is shown. Example shown on the picture:



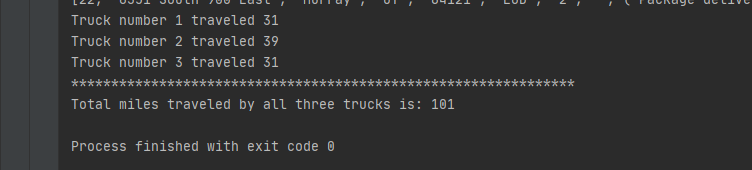
Adaptability, software efficiency and maintainability

In the future if there are routes with more destinations, the algorithm complexity function will keep growing smoothly. The worst case that the algorithm could take is O(N^2). Algorithm can also be improved later by presorting a given list of distances if the given route grows larger, which will help the sorting algorithm sort the given route faster.

The entire program can be easily understood because each major block contains comments on time complexity and what the given method does. This will help if somebody wants to make any modifications to the program or attempt to enhance it.

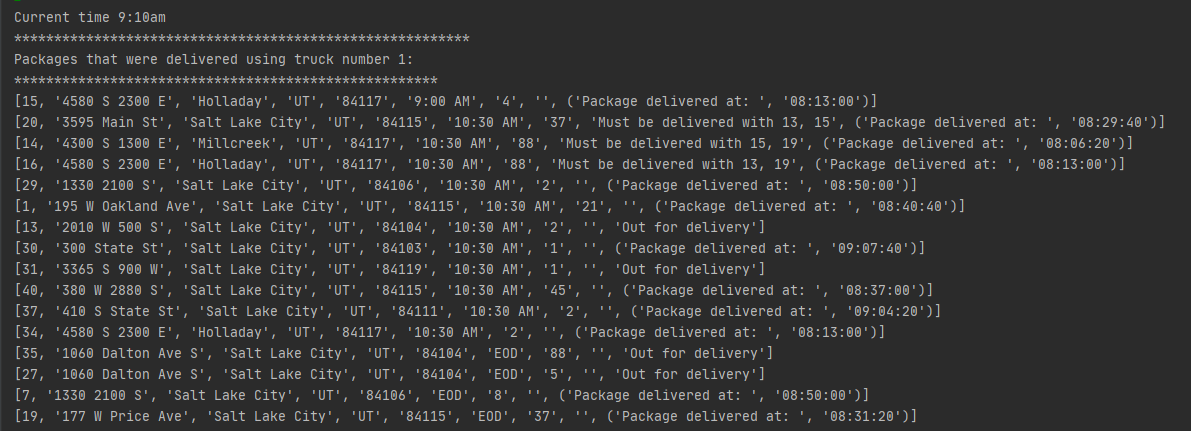
Original Code

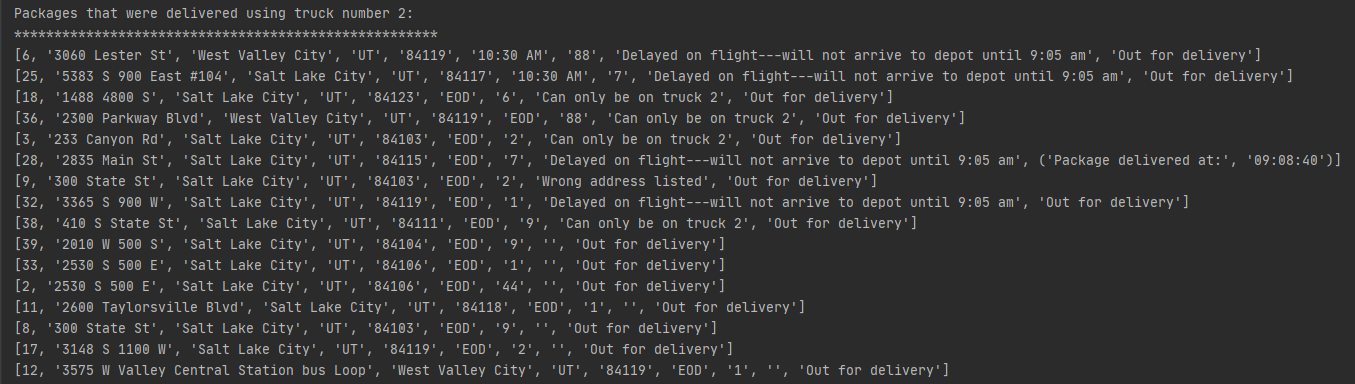
The provided solution meets the requirements because it delivers all the packages traveling less then 145mi by all three trucks together. Truck 1 traveled 31mi, truck 2 traveled 39mi and truck 3 completed deliveries traveling 31mi. Therefore, the total mileage traveled by all three trucks is 101 miles.

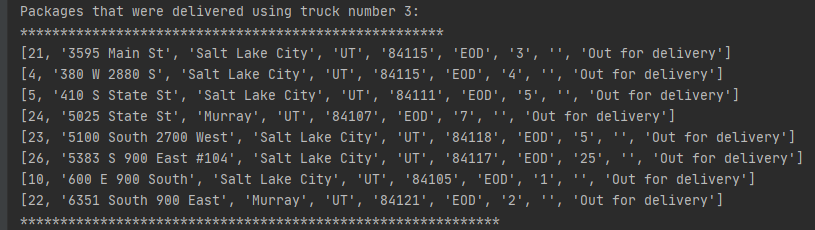


The program also clearly provides the status for every package at any given time. For example, from following screenshots, the status of every package can be seen at any given time.

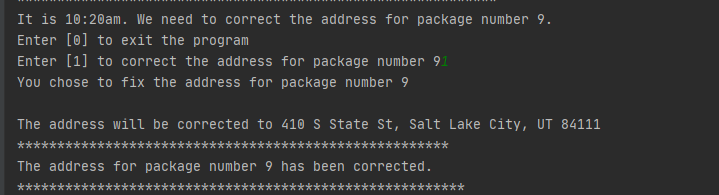
The status of every package between 8:35am and 9:25am. Given time is 9:10am.



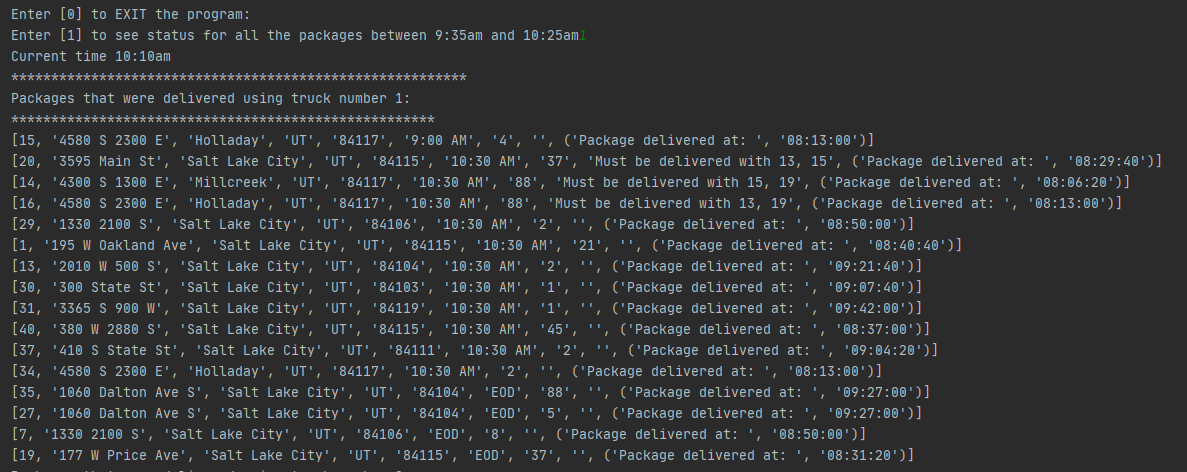


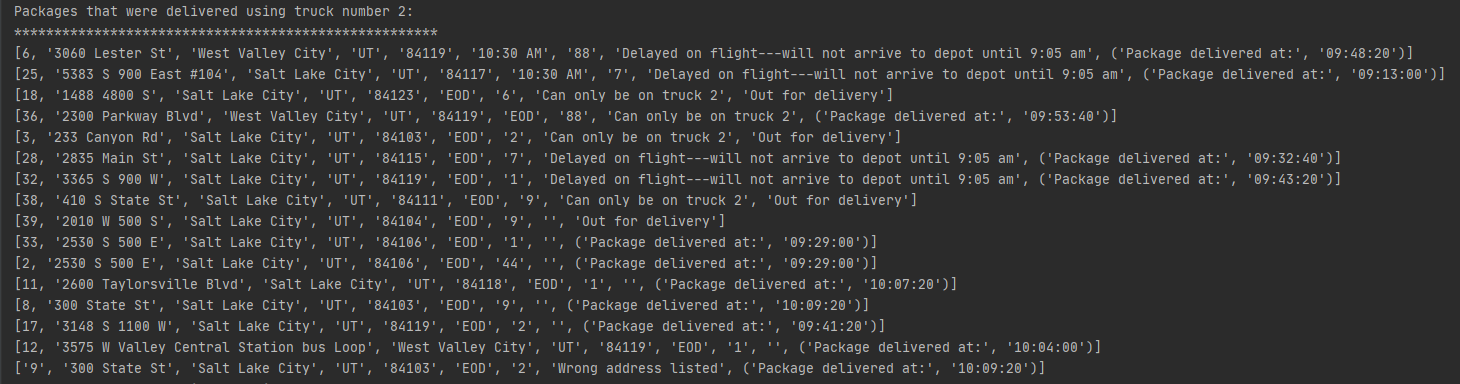


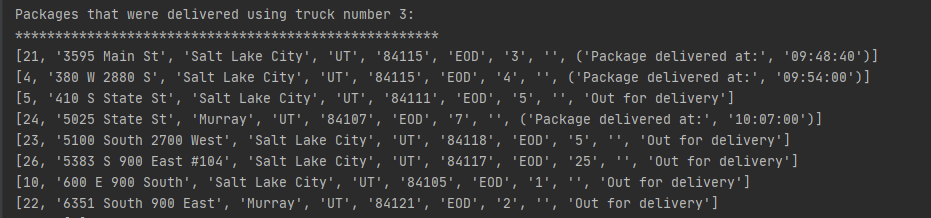
Then the program correctly fixes the address for package number 9.



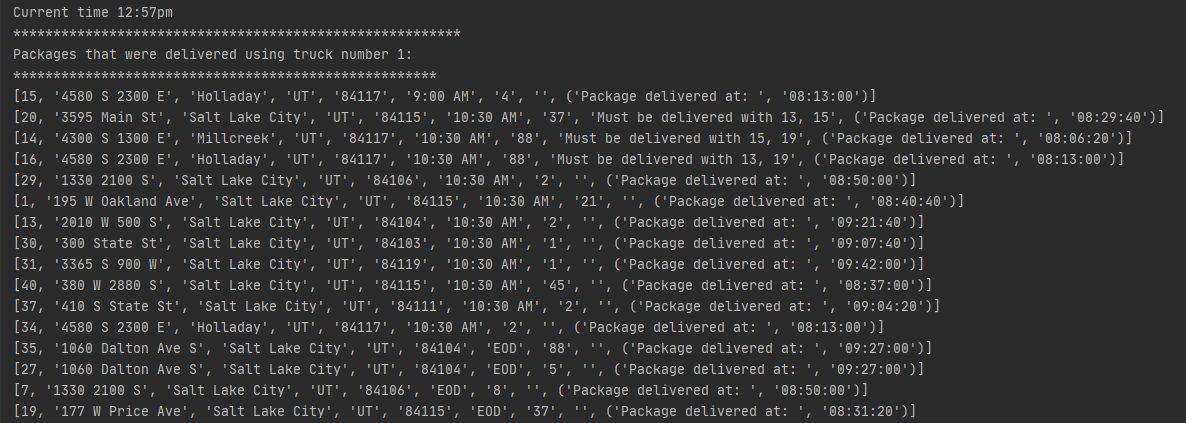
Then the program needs to show the status for all the packages between 9:35am and 10:20am. Given time is 10:10am.

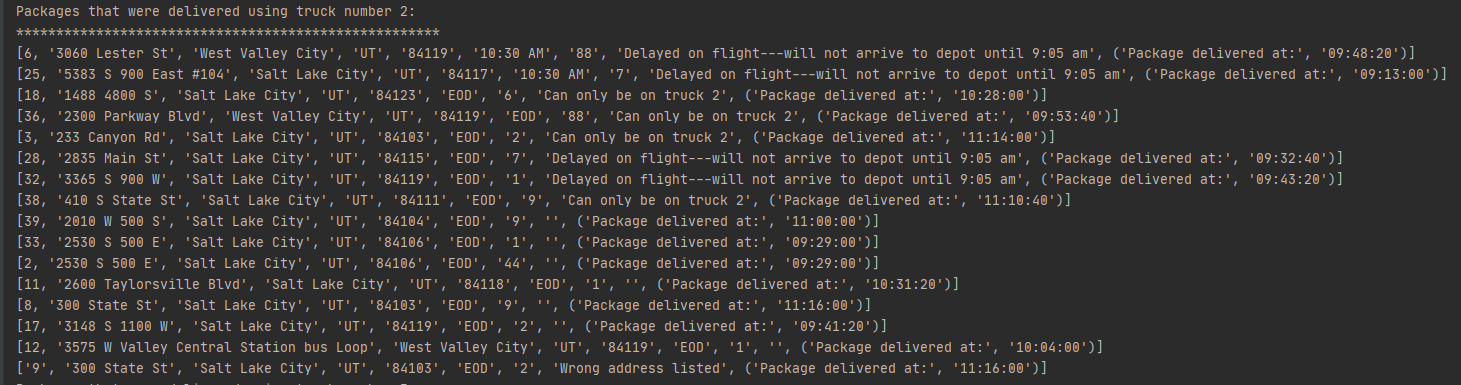


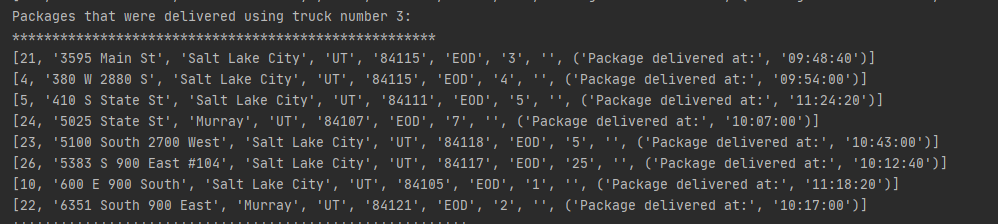




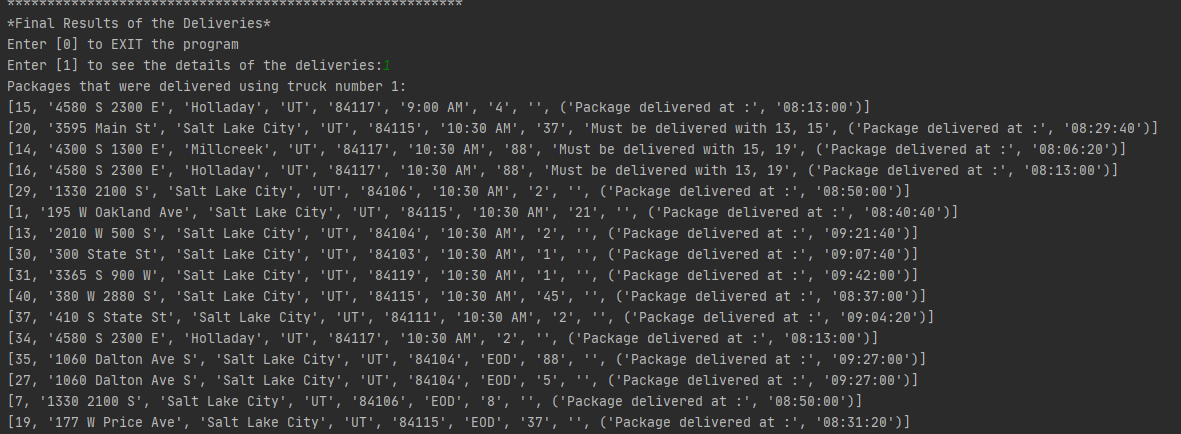
After that, the program checks the status for all the packages between 12:03pm and 1:12pm. Given time is 12:57pm.

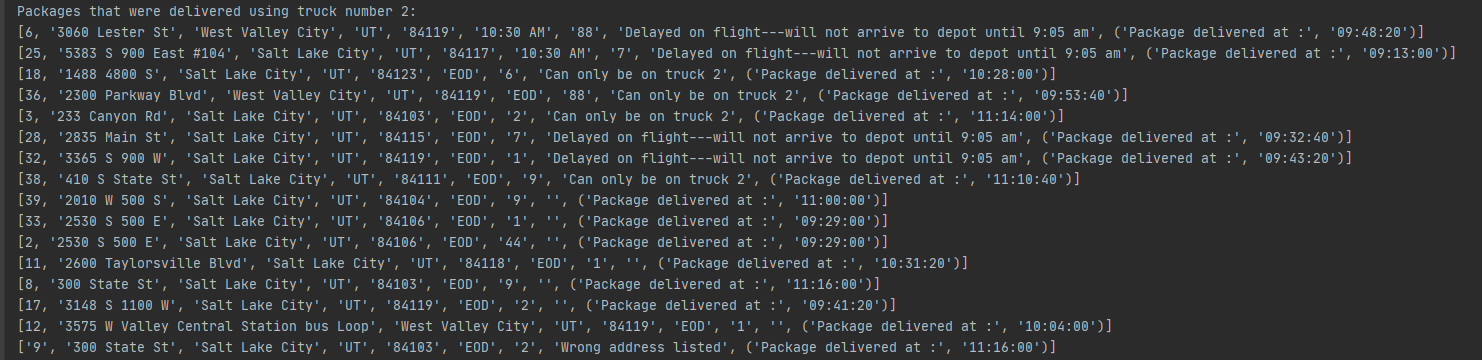


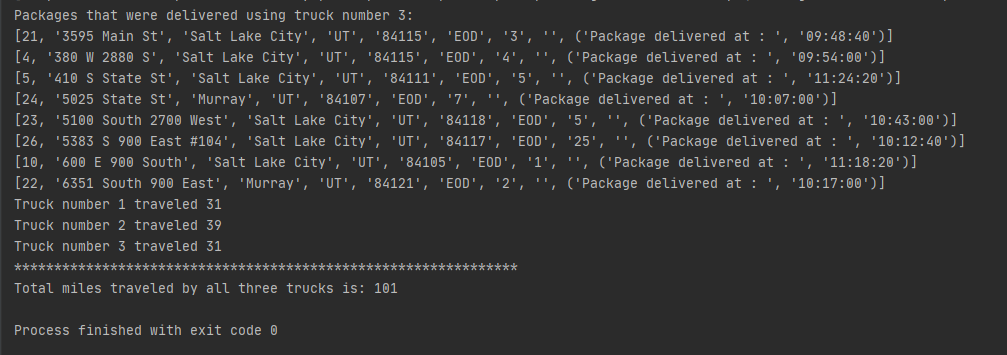




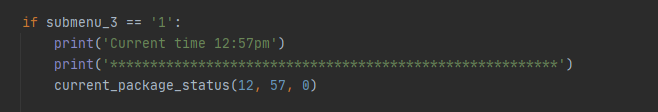
After that, the program displays the result and status of all the deliveries that were done by all three trucks.



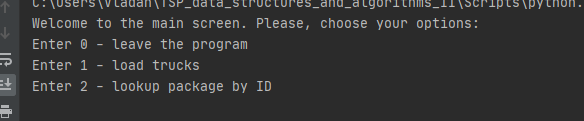


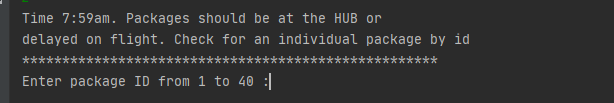


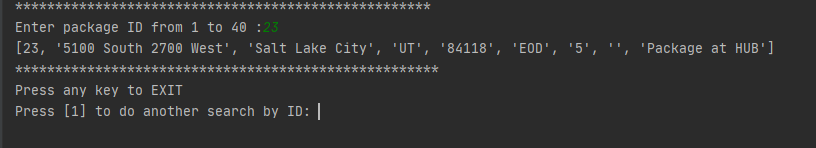
Passing different time value to current\_package\_status (), inside the user\_interface.py file will display current package status for all deliveries and packages at any given time.



It can also show the current package status by ID. By entering number 2 at the main screen of user interface:

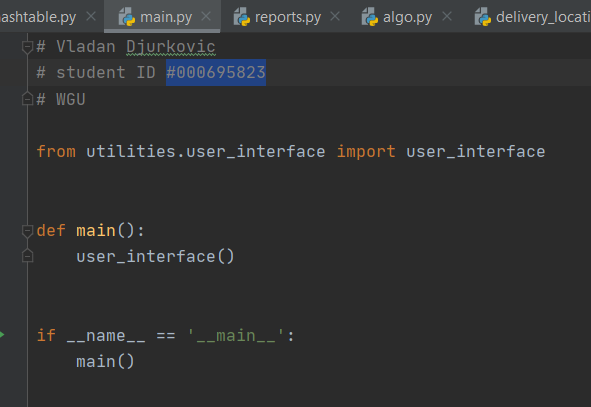






Identification Information

Comment with identification information included in main.py file.



Process and Flow Comments

Every major block of code is commented by explaining what every method does. Comments are also there to improve legibility of the code and to explain more complicated parts in case somebody wants to go back and modify it.

Data Structure and Explanation of Data Structure, Hash Table, Self-adjusting Data Structure

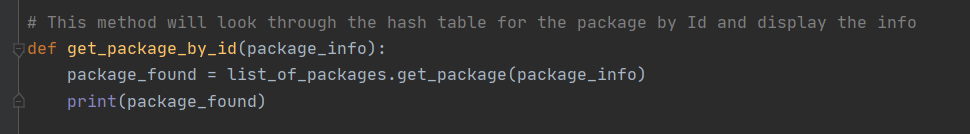
For this project I implemented a hash table called class PackageHashTable. The purpose of the hash table is to improve the speed of accessing packages that are stored inside the backets. Time complexity is O(N). Hash table has an initial capacity of 32 backets and each bucket can store a list with more than one package. For loop will loop through the hash table and append each package to the initial list.

Method, def add\_package (self, key, package) inserts a new package into a hash table using a first index from package\_file.csv as a key, because the first column inside the csv file is the package Id. It will insert a package into a backet based on the package Id modulo length of the table. Based on mod taken, it will place a package in the bucket. The preceding method also includes two if statements that will check if the package is delayed on flight. Method, def get\_package() will take a key as a parameter and first it will look if the package is in any of the buckets. If it is, it will loop through the list and find the matching key if it exists. If not, it will return None. Method, def del\_package () will do the similar thing as the previous method. The difference is that if the matching key or package Id is found, it will remove it from the list.

Inside the hashtable.py file there is a method get\_package\_info () that takes a csv file as a parameter. It reads the csv file package\_file.csv line by line and it will append values from the csv file to package\_list which is an instance of PackageHashTable class. Values that will be appended are package Id as an integer. The second part is the rest of the line that includes address, city, state, zip, delivery deadline, weight of a package and special notes if present. The method will return a list of stored packages.

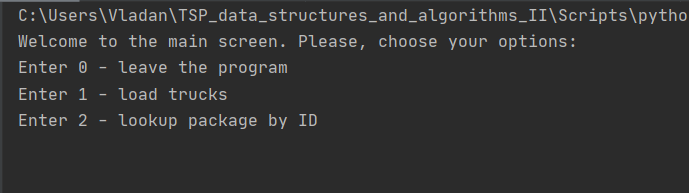
Look-up Function

Hashtable.py file also contains get\_package\_by\_id() that is used inside the user\_inteface.py file It takes package\_info as a parameter, which in this case is package id. It will look through the hash table and display the information about the package, if matched.



Interface

For this project, I provided a command line interface for the user to interact with the program and view the status of packages and other info.

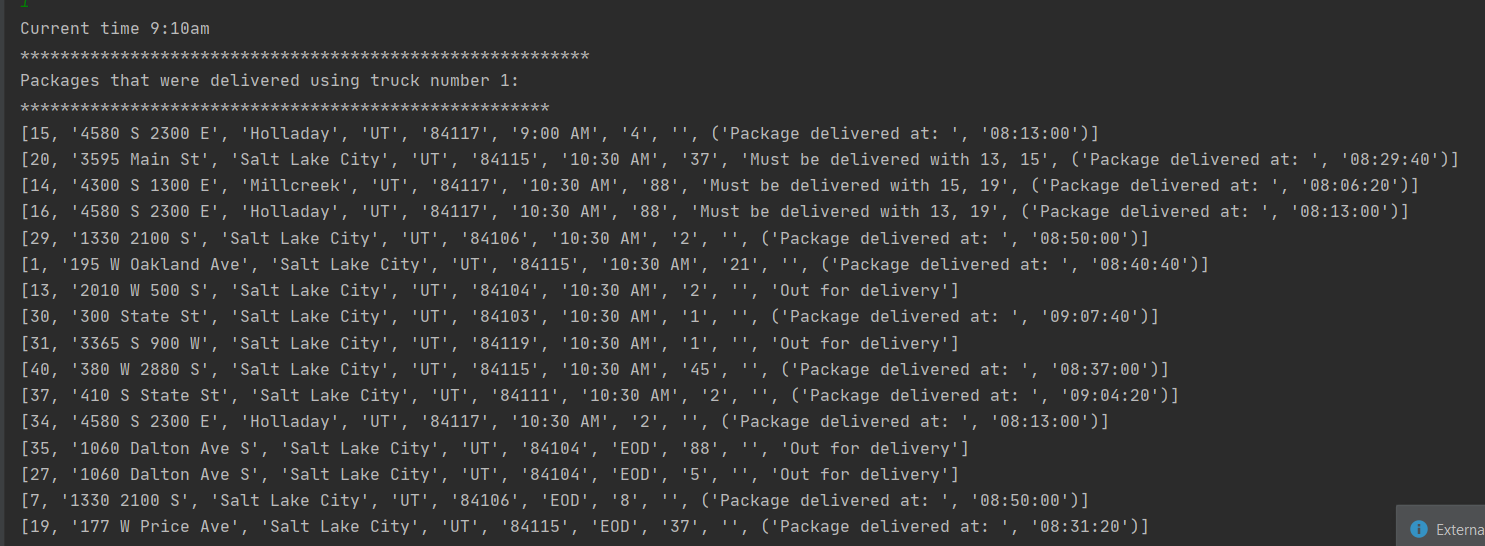


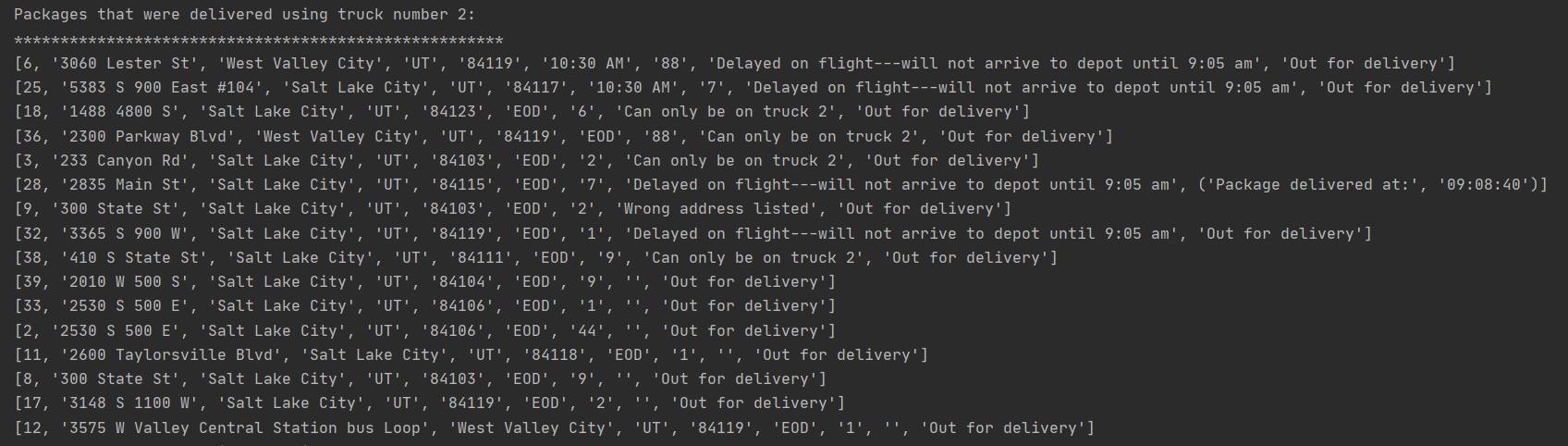
1st, 2nd and 3rd Status Checks

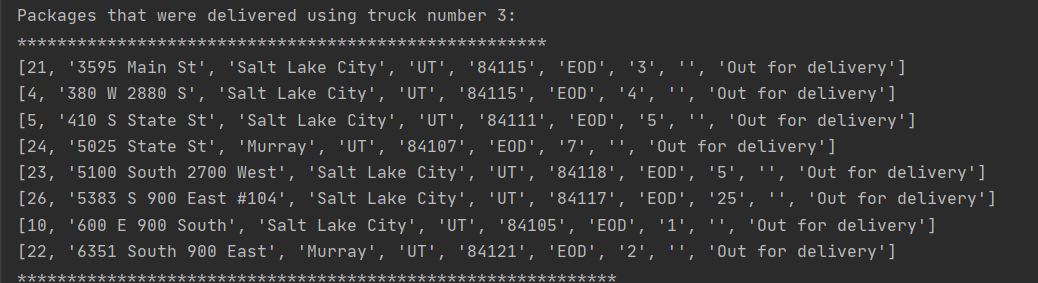
The following screenshots provide info and status of the packages between:

* 8:35am and 9:25am
* 9:35am and 10:25am
* 12:03pm and 1:12pm

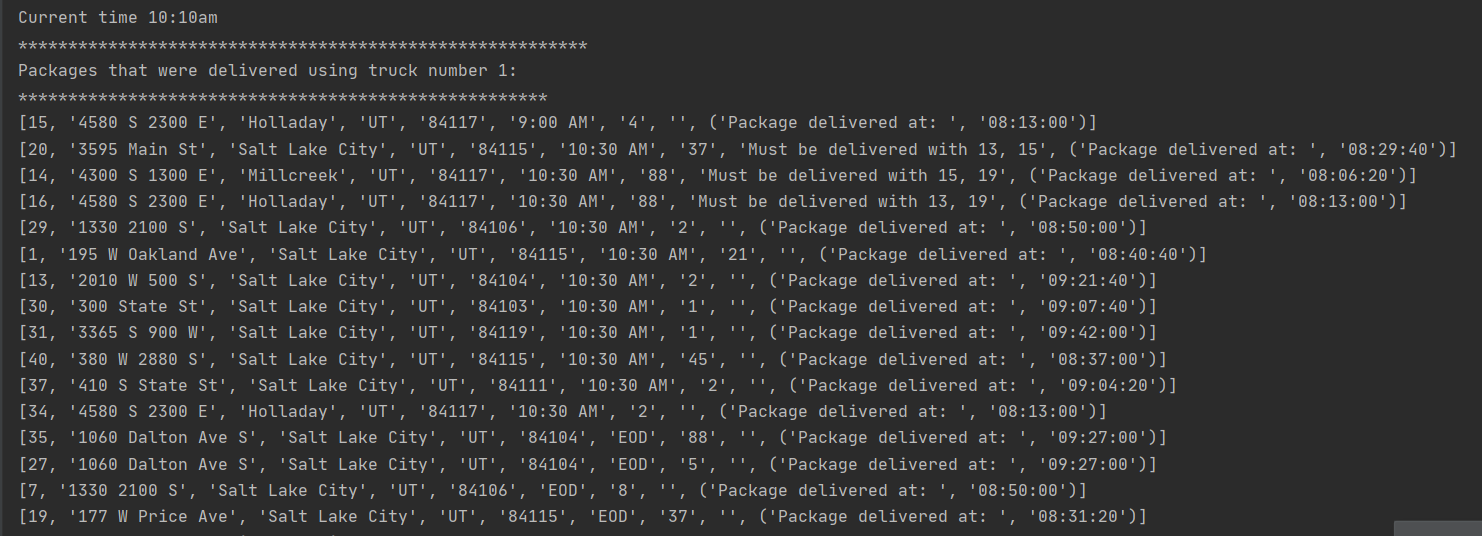
1st time check 9:10am

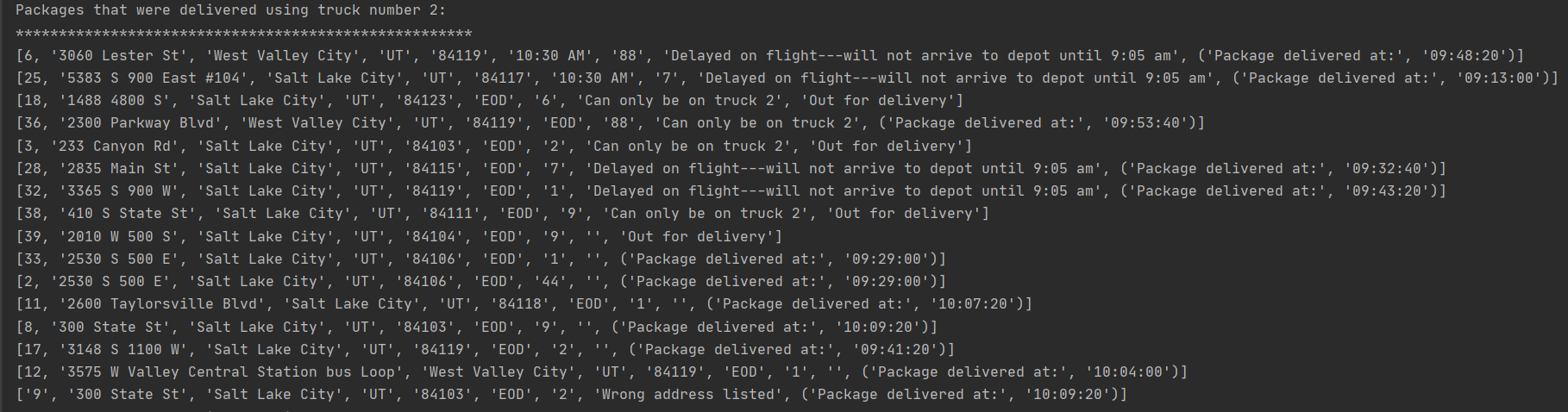


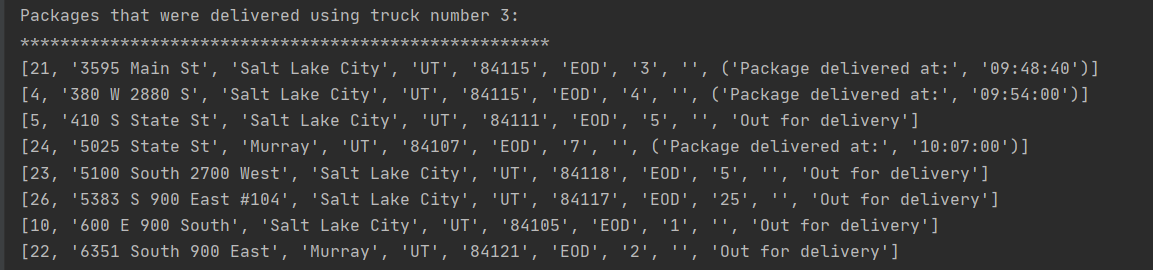




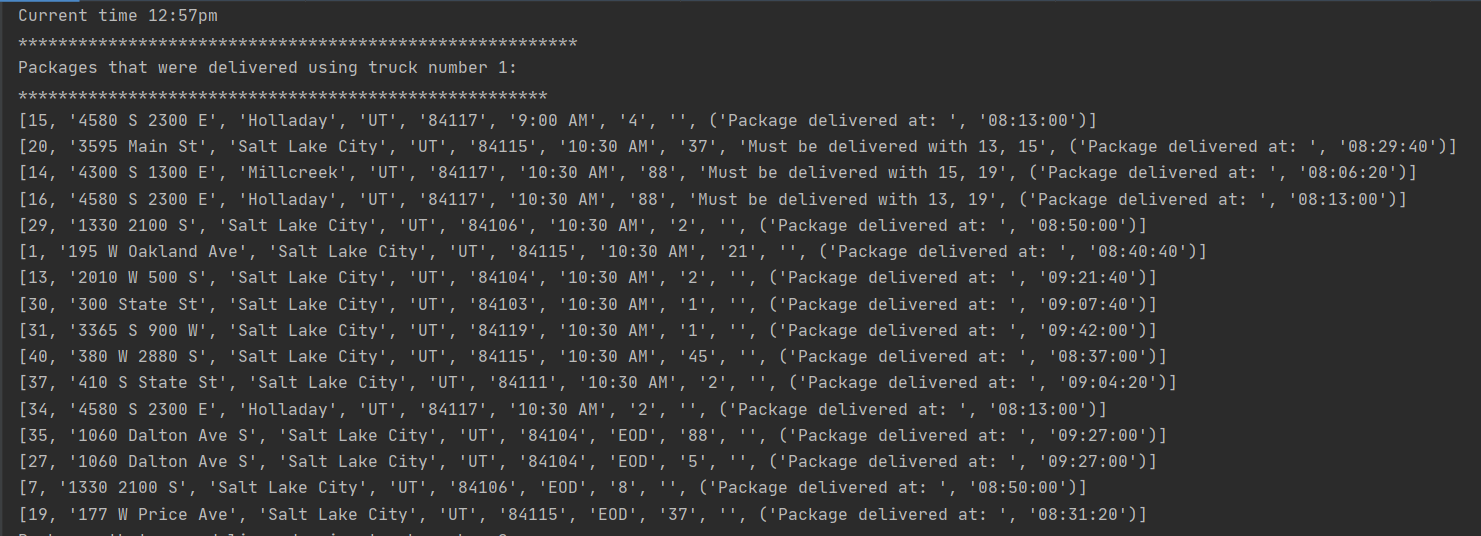
2nd time check at 10:10am

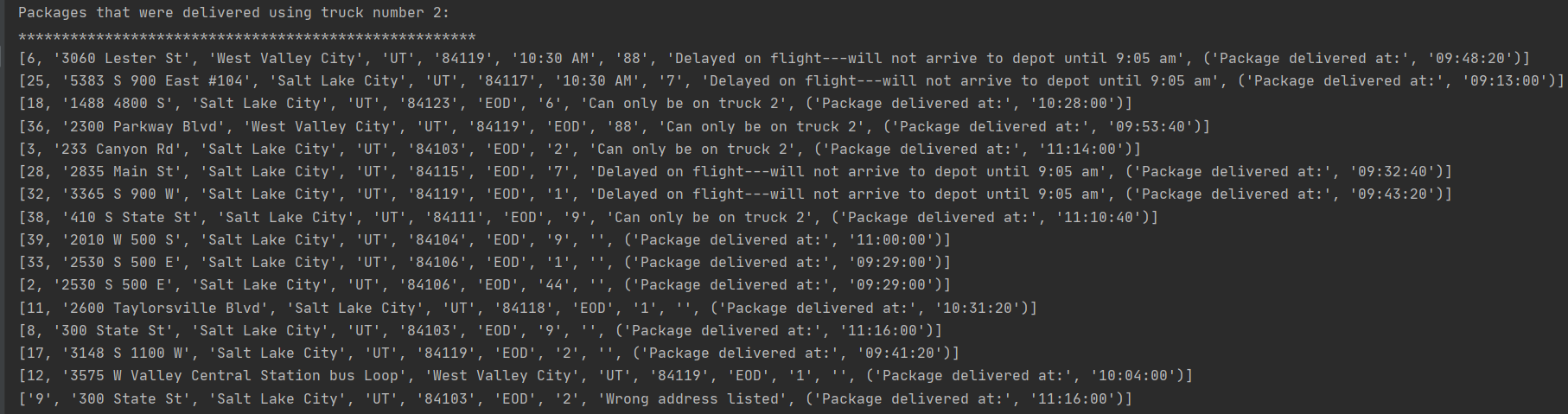


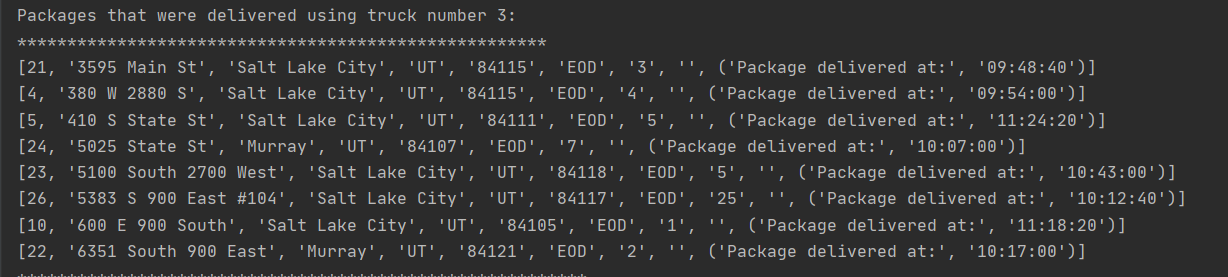




3rd time check at 12:57pm

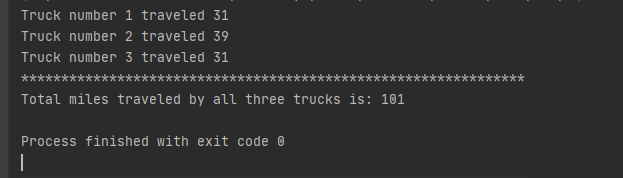






Strengths of the Chosen Algorithm, Verification, Other Possible Algorithms and Differences

Going back to the algorithm chosen for the problem, I can conclude that the greedy algorithm was a good choice because in our case it will always choose the next closest available location from the list. However, it usually requires sorting of choices from the list. Taking the next best choice is a linear sort. Greedy algorithm is a very good replacement for exhaustive search algorithms. In our case, it worked out well since it provided a solution that delivered all forty packages and the trucks were driven only 101 miles.



Some other algorithms that could have been used for our problem are the Nearest Insertion Algorithm and Multi-Fragment Algorithm. The Nearest Insertion algorithm works in a way that finds the two points that are furthest apart and connects them. Afterwards, it identifies the furthest location from any point from the previously connected distance of the furthest points. It incorporates the last found point into the route and deletes the previously connected furthest distance. It repeats the process until only one point remains and it is the only choice to connect to complete the tour.

A Multi Fragment algorithm takes the list of all the distances and identifies two closest points, joins them together and forms one edge. Then it finds the next two locations that are closest to each other and connects them. After this step, it looks for two points where joining them will not result in a closed path and each point has an edge degree of either 0 or 1. If the point has an edge degree of 2 it cannot be connected to any other points. The steps are repeated until only two points remain with an edge degree less than two. Finally, it joins them, and the route is completed.

Looking at these two approaches, greedy algorithm that looks for the shortest distance is more efficient because it needs to look for only one edge at the time that is the closest to the previously identified one. While the other two algorithms previously mentioned always look for two edges.

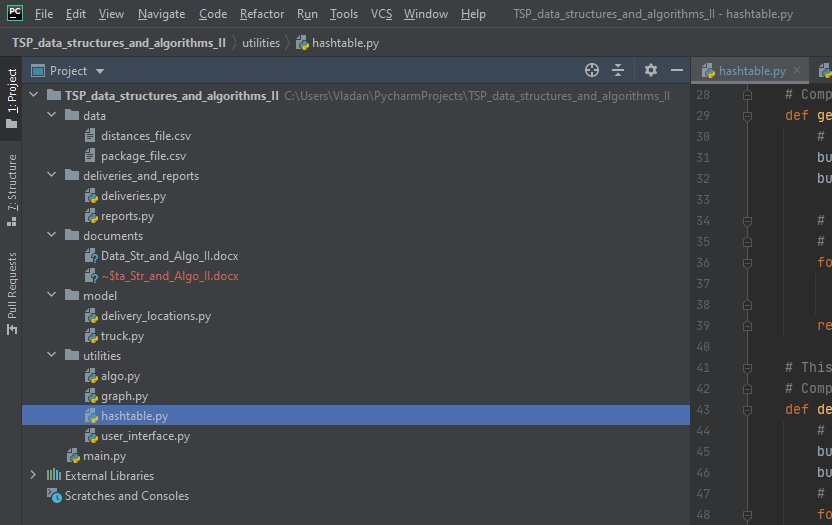
Different Approach, Verification of Data Structure, Efficiency

The one thing that I would maybe modify later in my program is the hash table which I initiated with 32 buckets. If the demand increases, it could be extended to 64 so it can have more buckets and each one of them can store a list with multiple packages. As a result of having more buckets, each bucket will store less packages inside which will reduce the time of looking through the bucket. Method add\_package that is defined inside of a hash table will add the package based on package id from a csv file and its complexity is O (1). Get package method that searches for an existing package from the hash table has a complexity of O(N). The delete package method has the same complexity and its purpose is to remove a package from a hash table based on package id. Adding more packages to the hash table requires more time to loop through the bucket and find the correct package. By adding more packages, the performance of the program will be affected.

Overhead and Implications

Hash table consists of three methods. Add\_package() method has a complexity of 1 and it will add a package to a hash table from the csv file package\_file.csv based on index[0], which is package id. Get\_package() method has a complexity of N since it loops through a list and finds the matching id for a package. It is there to retrieve the package from the list, if it exists. If it doesn’t find it, it will return None. Delete\_package() method has the same complexity as the preceding get\_package() method, but instead of retrieving a package it deletes the package, if found. The time for the hash table increases linearly with the size of input. This means there is a constant k such that the running time is at the most kN for every input size N. I believe that this is the best possible time complexity for the hash table because the algorithm must sequentially read the entire list.

There is no application server hosting the application for this project. In our case there is not much concern for bandwidth and memory usage since the data files, distances\_file.csv and package\_file.csv are located inside the project folder under data directory. The application runs on the local machine only.



Adding more load to the hash table, meaning if the delivery task expands to more cities, requires some structural changes. In general, one cannot predict the size of the load, so one of the solutions could be a resizable array. Instead of representing the hash table as a bucket array, we can introduce a header pointer object that contains a pointer to the current bucket array and keeps track of number of elements in the hash table (cs.cornell.edu, 2020).

Other Data Structures and Differences

Instead of using a hash table to store our package information, some other possible options would be a python built-in data structures dictionary and a list. Dictionary could have been used because it consists of key and value pair. It could work for our solution by storing package id as a key of the dictionary and the rest of the row for that package could have been stored as a value of the dictionary. The other option is python lists. The package data can be stored in a list by nesting a dictionary inside a list. In this case, each dictionary inside the list would contain package information. Comparing these, hash table that has been used for the project is more efficient because it contains buckets where we can place packages by simply applying package id number modulo size of a hash table. That operation places a package in a particular bucket.

References

cs.cornell.edu (2020). Lecture 13: Hash tables. <https://www.cs.cornell.edu/courses/cs3110/2014fa/lectures/13/lec13.html>