Linmei Mills

01/21/2023

Student ID: 001530700

**Project Overview**

**Introduction**

This application aims to deliver 40 packages for Western Governors University Parcel Service(WGUPS) using two CSV files that contain packages information and a table of addresses and distances between two different locations. The application must adhere to certain conditions, such as delivering packages at a specific time, handling flight delays, and ensuring that packages with specific requirements, such as being on a specific truck or delivered together, are met. In addition, there are only three trucks, two drivers, and each truck can only hold a maximum of 16 packages. To ensure all packages are delivered by the end of day, the program first category packages based on their requirements and conditions. This is followed by using a series of constant statement to load packages onto the first two trucks, leave the rest to the third truck. However, the truck number two will be return right before the delayed packages arrived to pick up the rest of flight delayed packages. The program then ensure that packages are delivered on time, starting from 8AM, by using the nearest neighbor algorithm to find the optimal delivery route.

**Programing Environment**

The programming is written in Python 3.10.9 and developed on IntelliJ IDE 2022.2 (Community Edition). It is only suitable for running on a local environment.

**Evaluation the Space-Time and Big-O**

The program’s major segments have been evaluated for space and time complexity and are commented in the program. The nearest neighbor algorithm that finds the fast route for package delivery runs in O(N^2) quadratic time with a space complexity of O(N) as a new list creates to store the determined optimize route. Searching and comparing the nearest address runs in linear time O(N) as it loops through a list of dictionaries, with a constant space complexity of O(1). Delivery of packages runs in linear time O(N) as it loops through the packages in the truck for delivery. The highest time complexity in this program is 0(N^2). O(N^2) time complexity not only occurs in the nearest neighbor algorithm function but also exists in multiple functions throughout the program, resulting in an overall time complexity of the application 0(N^2).

**Nearest Neighbor algorithm overview**

The nearest neighbor algorithm is optimized for finding the most efficient route for drivers to deliver packages. It is a common approach for solving problems related to finding the shortest route that visits a given set of locations and returns to the starting point, making it suitable for the purpose of the program. The pseudocode for the algorithm is provided as the following, the same as the time complexity and space complexity.

* Pseudocode
* Input: a start\_location as a pkg object, a list of package objects
* Output: a shortest path
* Function specific:

|  |
| --- |
| * Function find\_fast\_route(start\_location, packages): * Initialize an empty\_route list * Initialize an empty nearest\_address * If start\_location object == Hub location object: * Set travel distance attribute inside the object to 0 * Set current location to a start location * While package list is not empty * initialize a location\_holder variable to None * initialize and set a nearest\_distance variable to an infinity * for each package in the package list * get the specific address from package object * find the distance between the specific address and the current location * if the distance is not empty or None * cast the distance as a float * if the distance < the nearest\_distance variable * set location\_holder to hold this package object * set the nearest\_distance to the distance * repeat until all package looped through * if the location\_holder != None * if the current\_location = the nearest\_location * update travel\_distance = 0 * else, update the travel\_distance to the current package that has the nearest distance * append the nearest location package to the route list * remove this package from the packages list * set current\_location to the nearest location * repeat until the packages become empty and get out of the while loop * return a route |

* Time complexity is O(N^2)

A nested loop, the for loop, loops through the entire list again to find the nearest package while the outer loop, the while loop, loops through n times, so it is n\*n = n^2

* Space complexity is O(N)

The ‘route’ variable is a new list created within the function. It stores each package visited and its size grows linearly with the number of packages in the input list, resulting in the function’s space usage being directly proportional to the number of packages.

**The capability of the solution to scale and adaptivity**

The core functions of the program are designed to be able to scale and addresses changes in the number of packages, including changes to packages constraints, the number of locations. The program also read packages information and graphic (distance table) information from CSV files without requiring code modification when increase or decrease the size of the CSV files. However, the program is only designed for 2 drivers and 3 trucks. If the number of urgent deadline package exceeds 16, delivery may be delay unless additional resource, such as an additional driver, are added. In this case, code modification would be required to increase the scalability.

The maximum capacity for the program is 48 packages, using 3 trucks each capable of carrying 16 packages. If more than 48 packages are present, the program may display message indicating that more drivers may be needed for timely delivery. Note that the 16-package count is based on the latest delivery start time of 8:00 AM and the latest deadline of 10:30 AM. A recursive algorithm is implemented in the code to determine the number of packages that can be loaded on a truck for deliveries with a 10:30 AM deadline. (See deliverable\_package\_count() function in utils.py)

**Software efficiency and maintenance**

The software is efficient with a time complexity of 0(N^2) and is easy to maintain, despite its time complexity not being the best. Its small data set, such as only 3 trucks, 2 drivers, and 40 packages, makes it easy to manage and fast. The code is well-documented with comments throughout, making it easy to understand when reading it. The class instance variable for trucks, graphs, and packages are always available during debug mode regardless of location of the debug point. Additionally, the program features a clean and clear user interface for user interaction.

**The strengths and weaknesses of the self-adjusting data structures and the hash table**

The self-adjusting data structure is used in the program for calculating a fast route is highly flexible and reusable. It can be applied to a wide range of input data and is also a common solution for the purpose of this project. Its strengths include:

Flexibility and reusability: Given a start location and a list of all locations, it will find a sufficient route and return to the caller regardless of where it is called within the program.

Travel distance calculation: The function updates each package with the travel distance.

Same address handling: If the next iteration has the same address, the distance between those two points updates to 0, as those packages can be delivered at one time without additional driving.

Use of deep copy method: The function uses a *copy.deepcopy()* method to make a package list to ensure the original data is not modified.

However, it also has the following weaknesses:

* As data input gets larger, the runtime increases exponentially with the number of locations. For large number of locations, it may not be possible to find an optimal solution in a reasonable amount of time.
* The nearest neighbor approach may not always find the optimal route, particularly if the locations are not distributed uniformly.
* The function only consider distance between locations and not the time. If take the time as part of optimization, this may not result the best measure of the fastest route.

Hash table is highly efficient for lookups, insertions, and deletions. In this program, the key is the package ID and it is unique, the value is the package object which contains all the package information. Thus, it is easy to retrieve packages based on the package ID.

However, hash tables are sensitive to has collisions, which occur when two or more keys are mapped to the same index in the table. This can lead to poor performance if the number of collisions is high. In my case, it the package ID increases, it can lead to poor performance. Also, the keys are not in order, so sorting by package ID is required additional step to get the list of package information from 1 – 40.

**What would done differently?**

If I were to start over, I would store the same references to the original packages list (maybe have a copy list as a backup) in the trucks instead of a copying of the packages. This way, when a package is updated during delivery, it would also update in the original list of packages, which is not only a better object-oriented programming but also can reduce an large amount of redundant code. Additionally, I would begin by creating a delivery high level flow diagram before writing any code. This would have saved me time, as I had to revise my program multiple times due to unclear high-level flow.

**The time affect to changes in the number of packages**

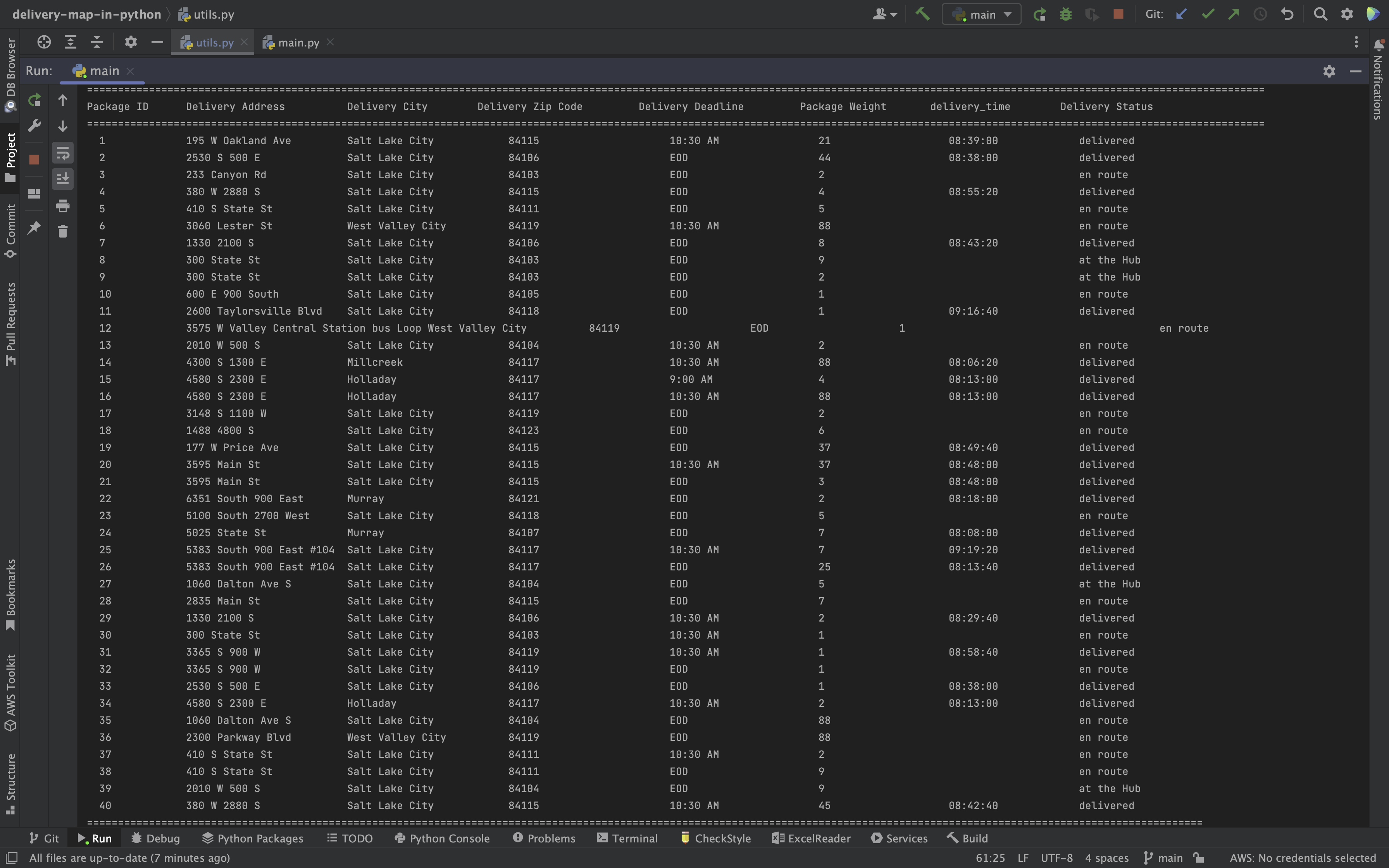
* The lookup/search function for looking up packages has an average time complexity of 0(1) but can be as high as O(N) as the worst case scenario. It results 0(1) when it uses hash function to find the bucket in which is a key-value pair is located. Usually, the number of items in the bucket is small, so it is fast to search. However, if all items are stored in the same bucket, then the time complexity would be the maximum size of items, resulting in the worst-case scenario.
* The space complexity is 0(1) as it only requires a constant amount of memory regardless of the number of key-value pair in the bucket. The variable ‘bucket’ in the search function (see Hashtable.py, line# 33) stores a reference to the bucket list and a single variable ‘kv’ in the for loop stores the current key-value pair.

**Two additional data structure**

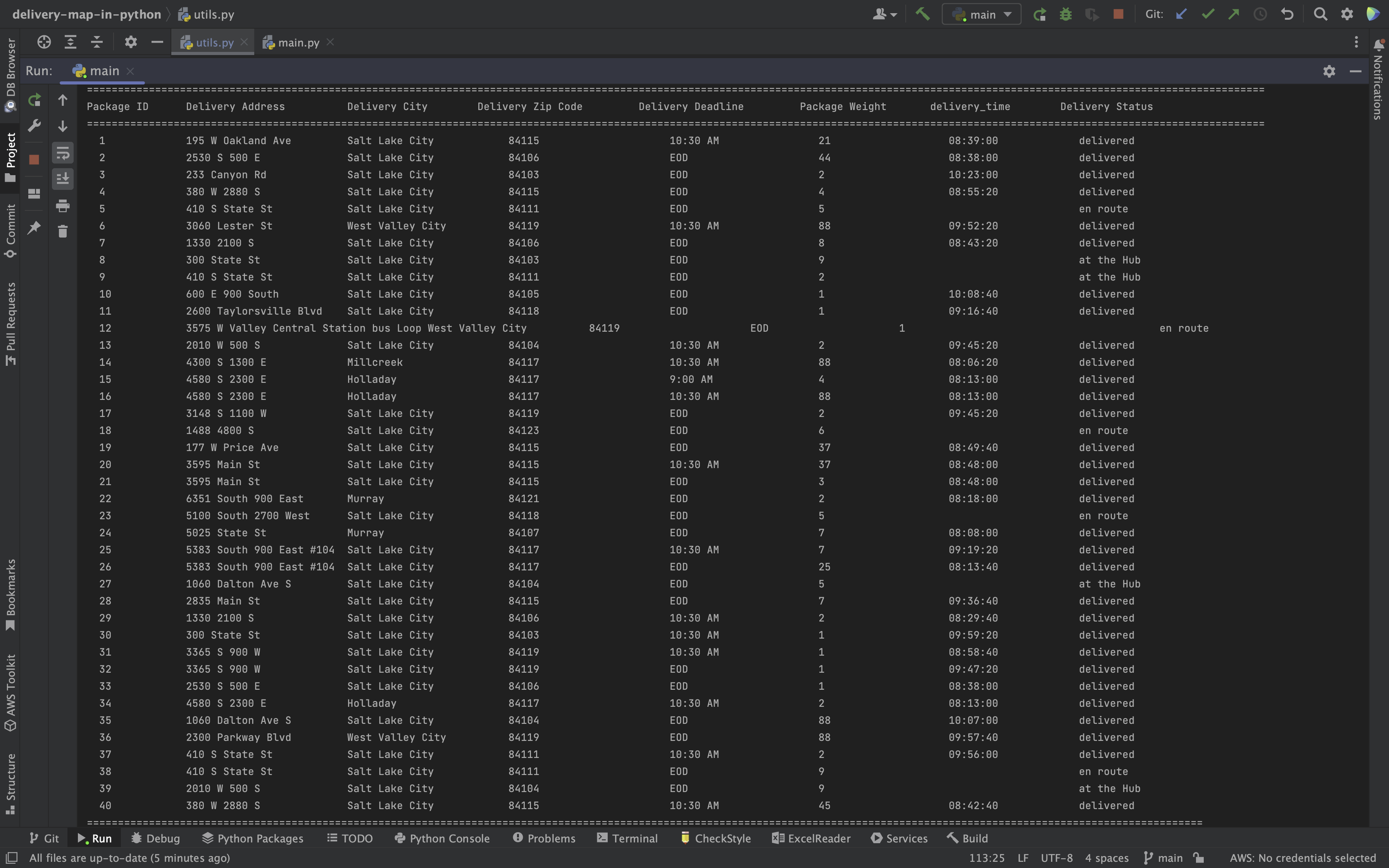
* List data structure is used throughout the program. For example, the packages in truck1, truck2 and truck2 are defined as list; they store a list of package objects. (See truck.py, line#2, 3, 4)
* Dictionary data structure is used to store addresses and its distance (attribute: address\_with\_distance). The keys are each location, and the value for each key is a list of locations connected to that specific location. (See graph.py)

**Screenshots**

* The status of all packages at a time between 8:35 a.m. and 9:25 a.m.



* The status of all packages at a time between 9:35 a.m. and 10:25 a.m.



* The status of all packages at a time between 12:03 p.m. and 1:12 p.m. Meanwhile, the following screenshot also indicates all packages are delivered on time by comparing package delivery deadline and the package delivery time.

Calendar

Description automatically generated with medium confidence

* A complete execution of the code and include the total delivery mileage.

Graphical user interface, text

Description automatically generated

**Citation**

* Lysechky, R., & Vahid, F. (2013). C950: Data Structures and Algorithms II. <https://learn.zybooks.com/zybook/WGUC950AY20182019> (The reference for the HashTable.py is from ‘Hash Table class using Chaining’ in C950 ZyBook Chapter 9)
* Nearest Neighbor algorithm, geeksforgeerks, Nov 28. 2022. https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/