C950 WGUPS Algorithm Overview

Lyndon Pearson

ID # 010389649

WGU Email: lpear79@wgu.edu

Date: 03/08/2023

C950 Data Structures and Algorithms II

# A. Algorithm Identification

The greedy algorithm was used in the delivery of packages. The loading of packages was done manually and is not self-adjusting.

# B1. Logic Comments

**Algorithm Overview:**

The greedy algorithm is written in the following format:

1. Accepts a Truck object, an address list, and distance list as inputs
2. For each package in the Truck object, calculate the distance for delivery.
3. The package with the lowest delivery distance has its distance and package ID

returned from the function

Operation time for the greedy algorithm is O(n). This is because the algorithm iterates through the entire package list to identify the closest package for delivery. The pseudo code is below showing both the structure of the program and the operational time.

The operational complexity of GreedyAlgorithm is O(n) since it iterates through the entire package list.

**GreedyAlgorithm (TruckPackageList, AddressList, DistanceList)**

**For I length[TruckPackageList]**

**Get delivery index from AddressList for distance look up of package**

**Retrieve delivery distance from DistanceList for package**

**If delivery distance current lowest, store distance and package ID**

**If not, continue iterating through package list**

**Upon completion, return lowest distance and package ID**

# B2. Development Environment

PyCharm Community Edition IDE version 2022.3.2 was used exclusively to create this application.

Python version 3.10.1 was used exclusively in the creation of this application.

The application was exclusively developed and tested on a Microsoft Surface 4 Laptop which contains an AMD Ryzen 7 4980U, 16GB of RAM, and a 256GB SSD.

# B3. Space-Time and Big-O

Please see file comments for space-time complexity of all methods. The large portions of code will be detailed below:

The Python file “PackageDeliveryProgram.py” has a big-O complexity of O(n2) due to “RouteAndDeliver.py” being called for each package. “RouteAndDeliver.py” then calls “GreedyAlgorithm.py” for each package.

The Python file “GreedyAlgorithm.py” has O(n) complexity as it iterates through the package list of the input Truck object and returns the nearest package.

The Python file “RouteAndDeliver.py” can be considered O(n) complexity since it calls the greedy algorithm (noted above as O(n)) followed by looping through the packages in numerical order. Together these operations are O(2n) but since the constant can be ignored it reduces to O(n).

The PackageHashTable constructor found in “manageData.py” can be considered O(n) since the constructor loops through the number of packages and appends storage locations to the list.

PackageHashTable class methods such as “printPackageStatuses”, “insertPackage”, “searchHashTable”, and “removePackage” can also be considered O(n) since they iterate through the package list to find the matching Package object for operation.

Functions such as “loadPackageData”, “loadDistanceData” and “loadAddressData” are also O(n) complexity since they extract data from each row of a CSV file.

The constructor and all “set” and “get” methods of the Package class are all O(1) complexity since they either assign an input to a field value or return the requested field value of the Package object.

The constructor as well as all “load”, “set”, and “get” methods of the Truck class are all O(1) complexity since they either add the package input to the truck’s list (“load”), set a truck field value (“set”) or return a truck field value (“get”).

The Truck methods “getPackageByID” and “deliveryPackage” are both O(n) complexity since they iterate through the truck’s package list to retrieve and delivery the specified package.

# B4. Scalability and Adaptability

This application is capable of scaling to a larger number of packages fairly easily as the data currently appends to lists in the hash table, address list and distance list.

# B5. Software Efficiency and Maintainability

This software is efficient and easy to maintain due to its modular approach (appending new data to existing lists). The program will automatically expand when reading address or distance files of greater size.

# B6. Self-Adjusting Data Structures

The hash table is space efficient and can automatically grow in size given the need. If the data is large enough, collisions will be unavoidable. If there are large amounts of collisions, the structure’s efficiency will suffer.

# C. Original Code

See “dsa2Project” folder for complete code.

# C1. Identification Information

Completed in “main.py”

# C2. Process and Flow Comments

Completed in all code files.

# D. Data Structure

A hash table was selected to store package data as objects.

# D1. Explanation of Data Structure

A hash table was created to store all Package objects. The individual Package objects were created by reading the “WGUPS Package File.csv” file, extracting the relevant rows, and using the cells in each row as field inputs into the Package constructor.

The Package objects were then inserted into the hash table by hashing their Package ID. The Package ID – Package item was stored as a key-value pair with the Package ID as the key.

The Packages were then retrieved by their Package ID key and loaded/assigned to the delivery Truck object.

# E. Hash Table

The hash table is implemented in “manageData.py”

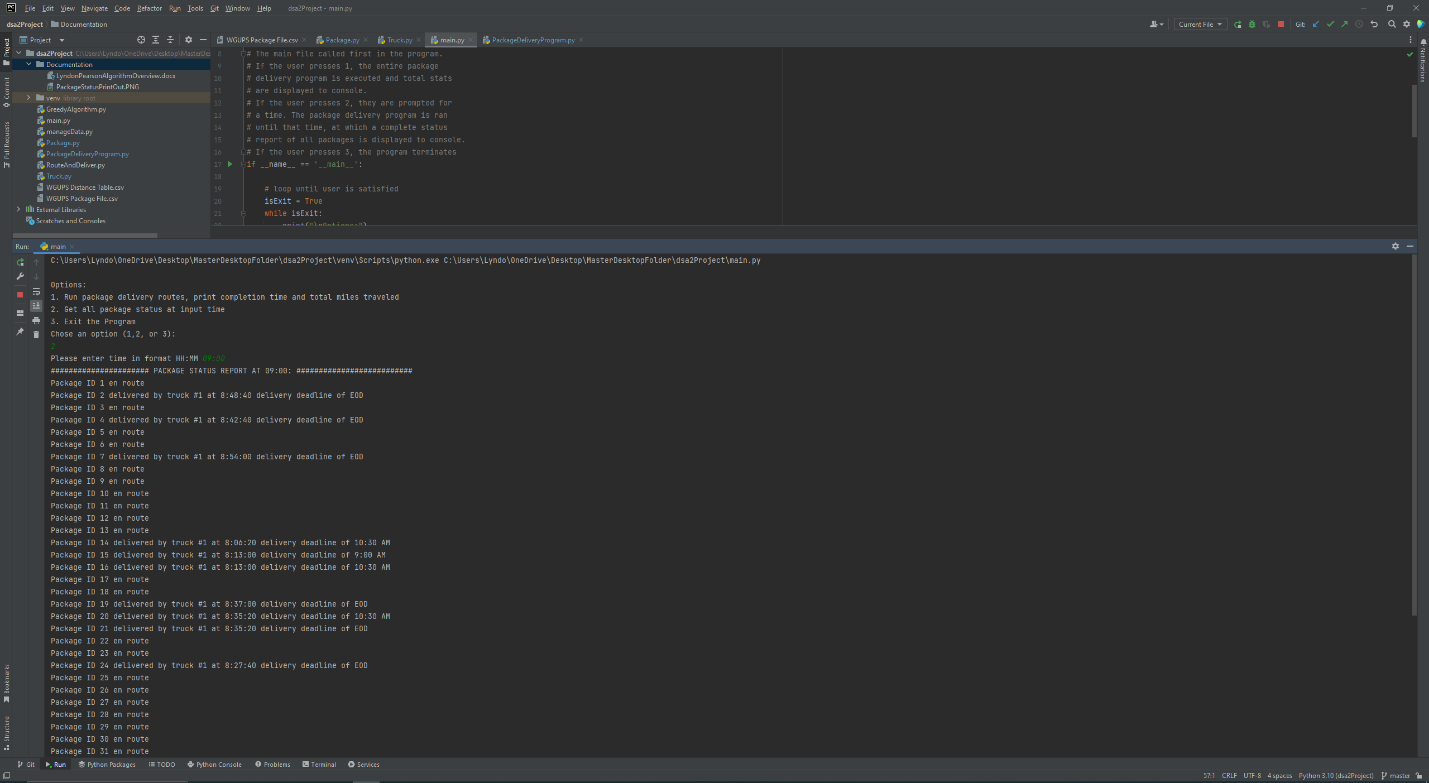
# F. Look-Up Function

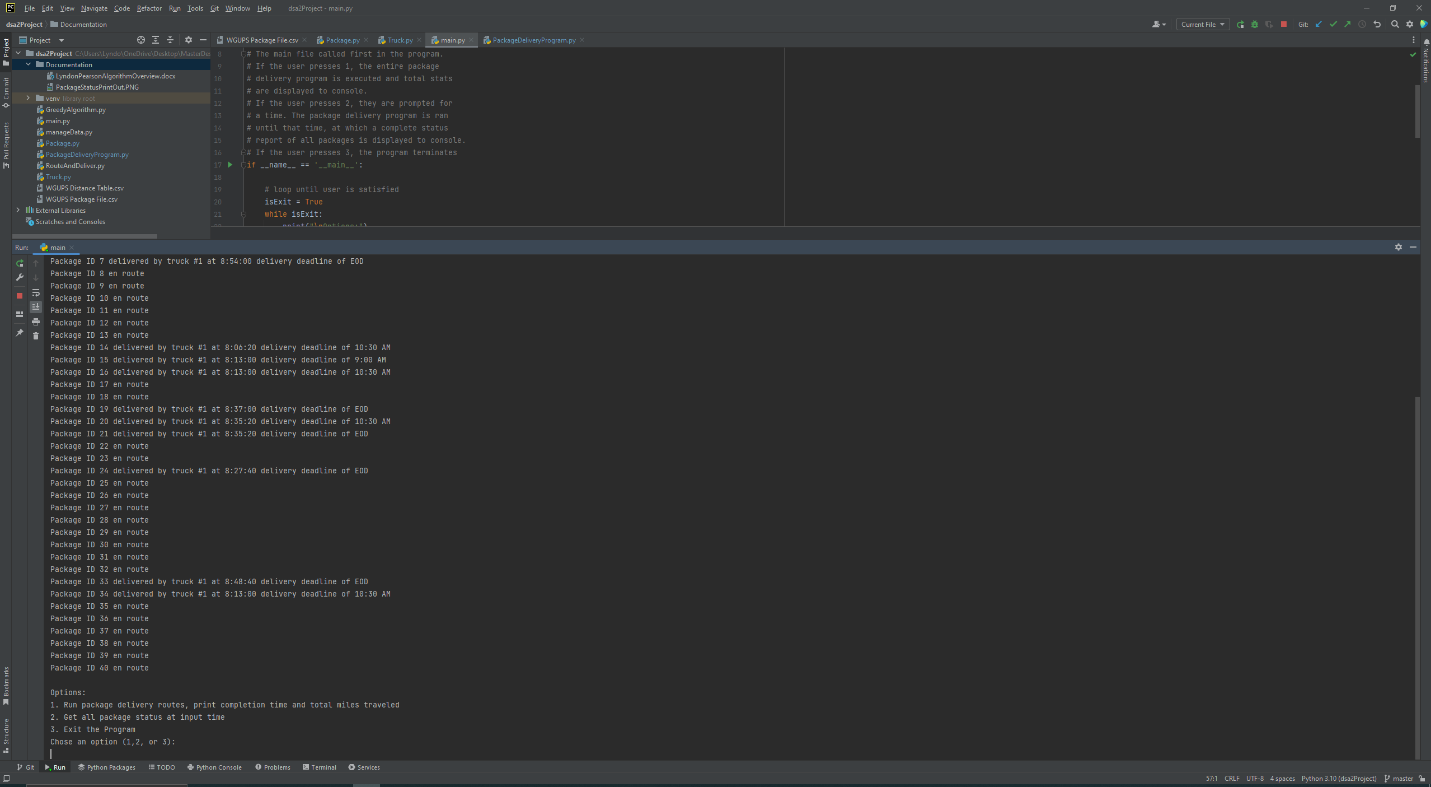
The look-up function is also implemented in “manageData.py”

# G. Interface

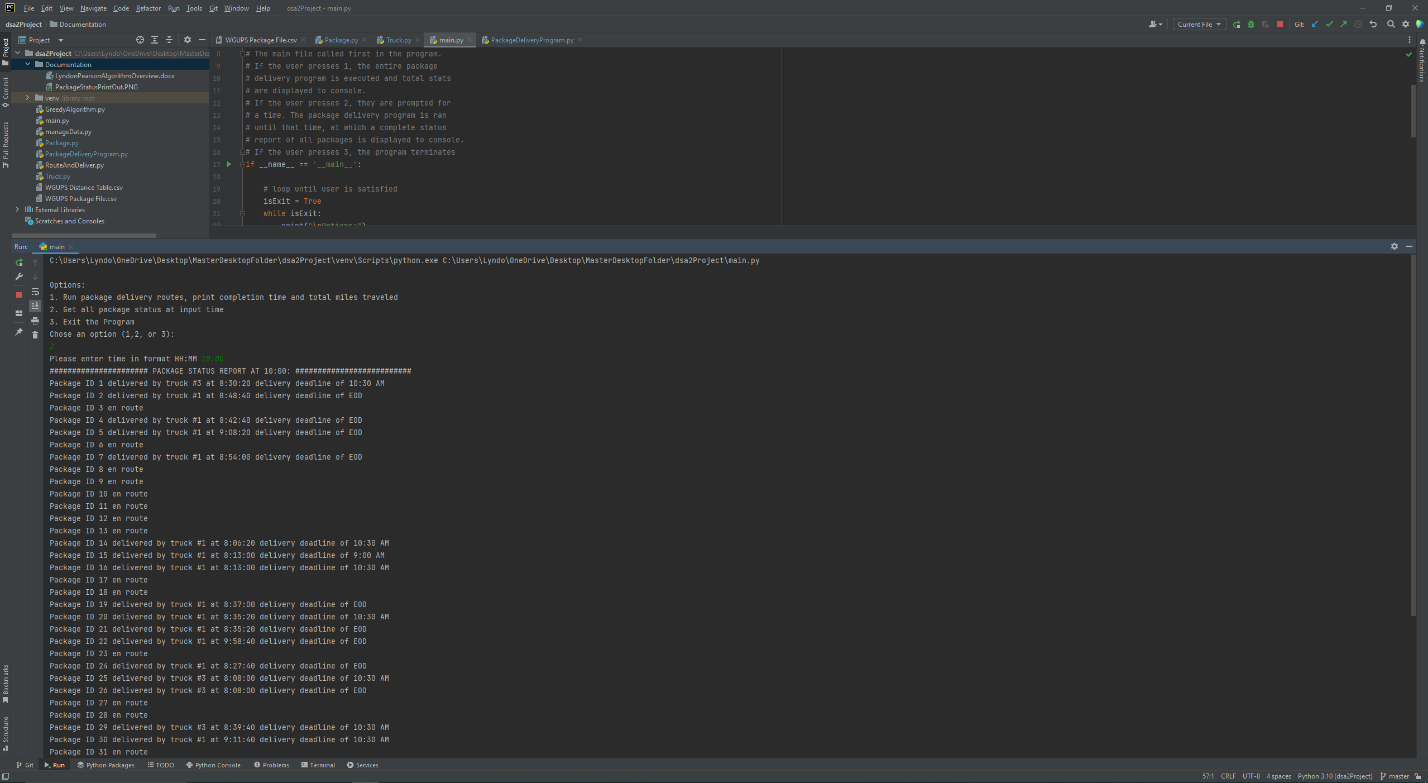
The interface is accessed by selecting option 2 and specifying a time in HH:MM format. Three example scenarios follow in G1, G2, and G3.

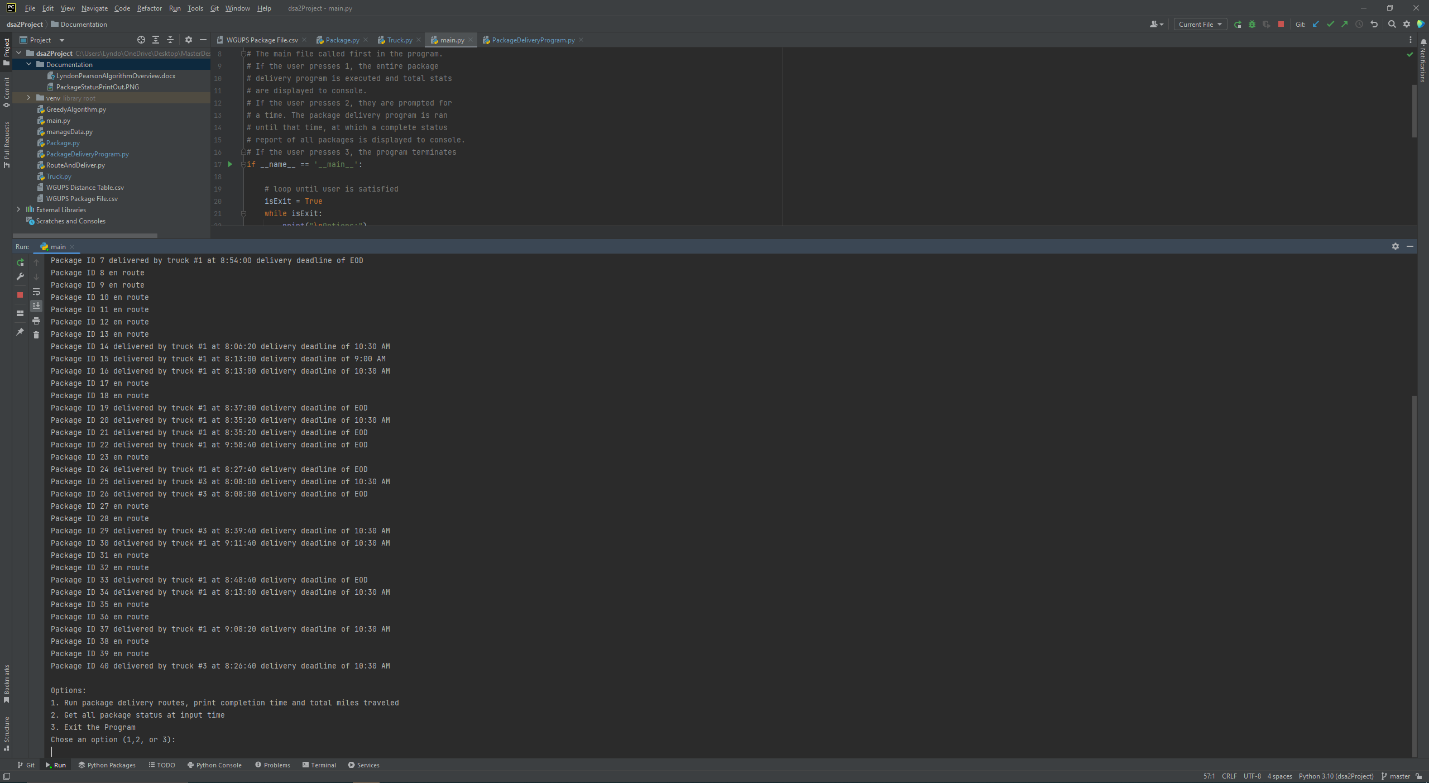
# G1. First Status Check



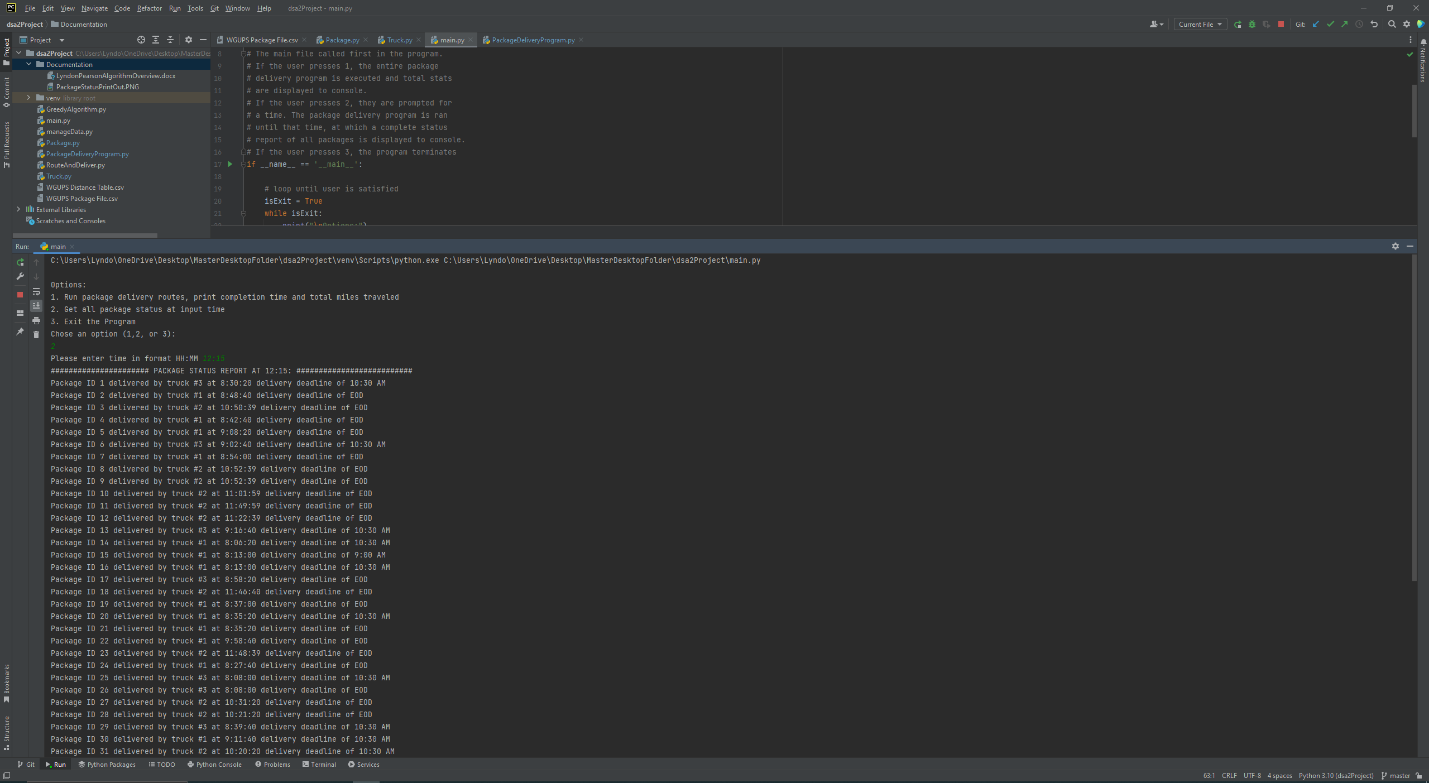


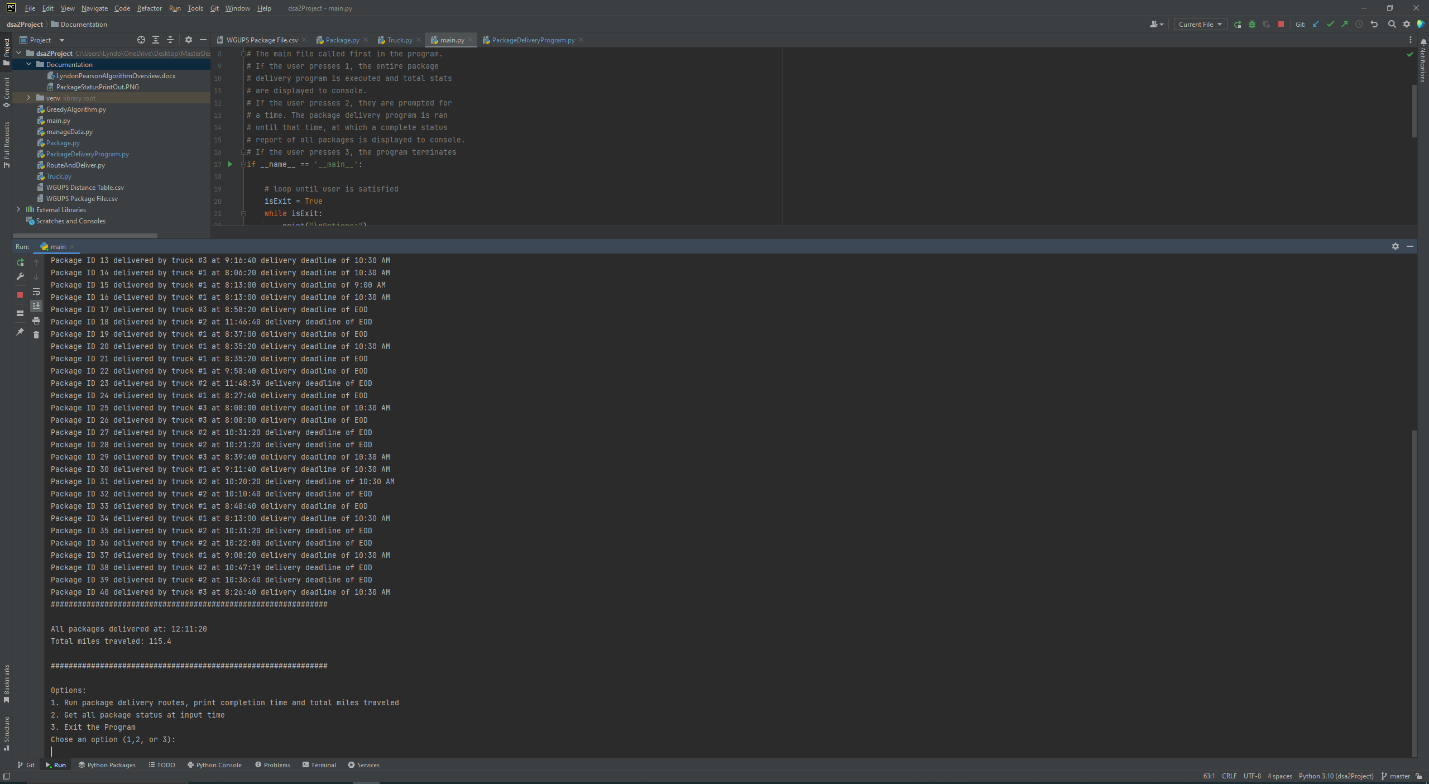
# G2. Second Status Check



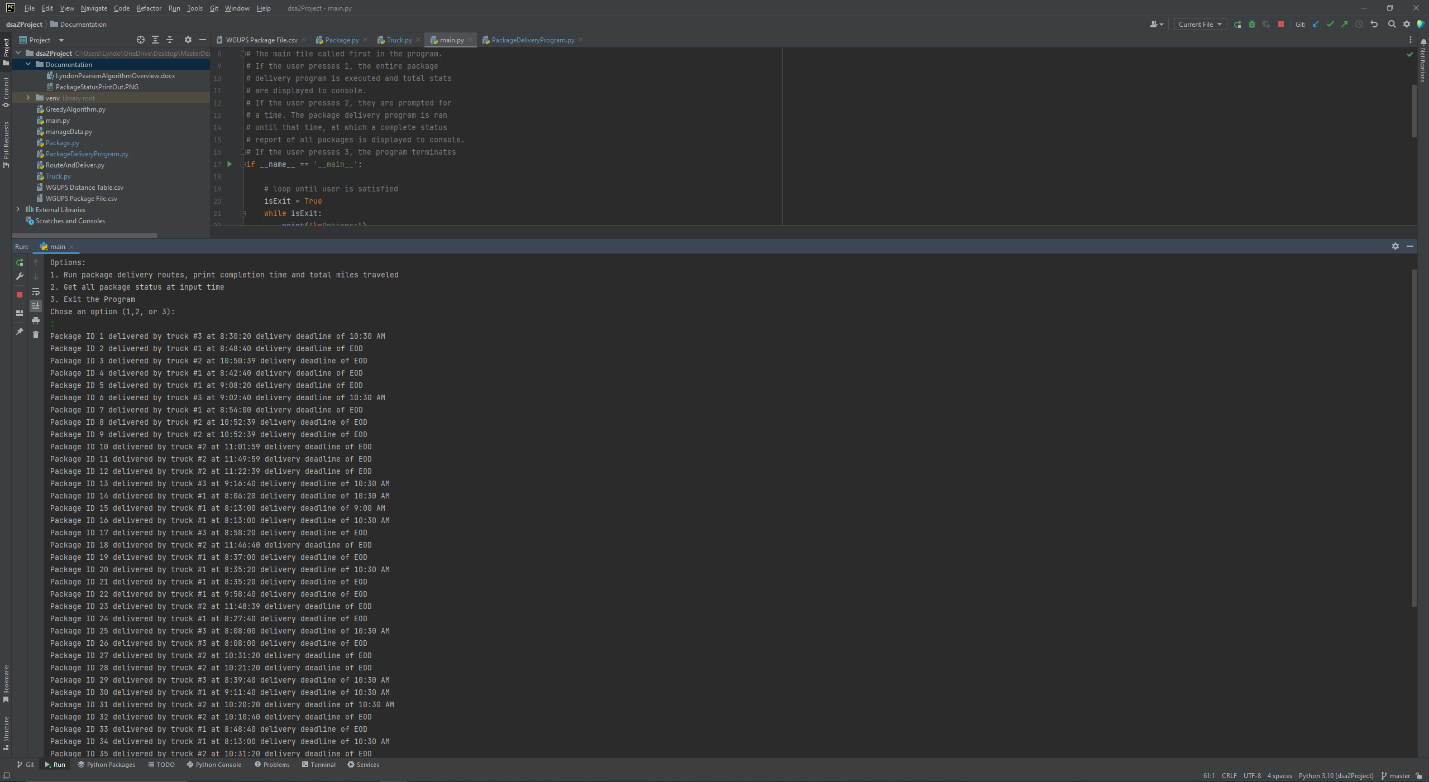


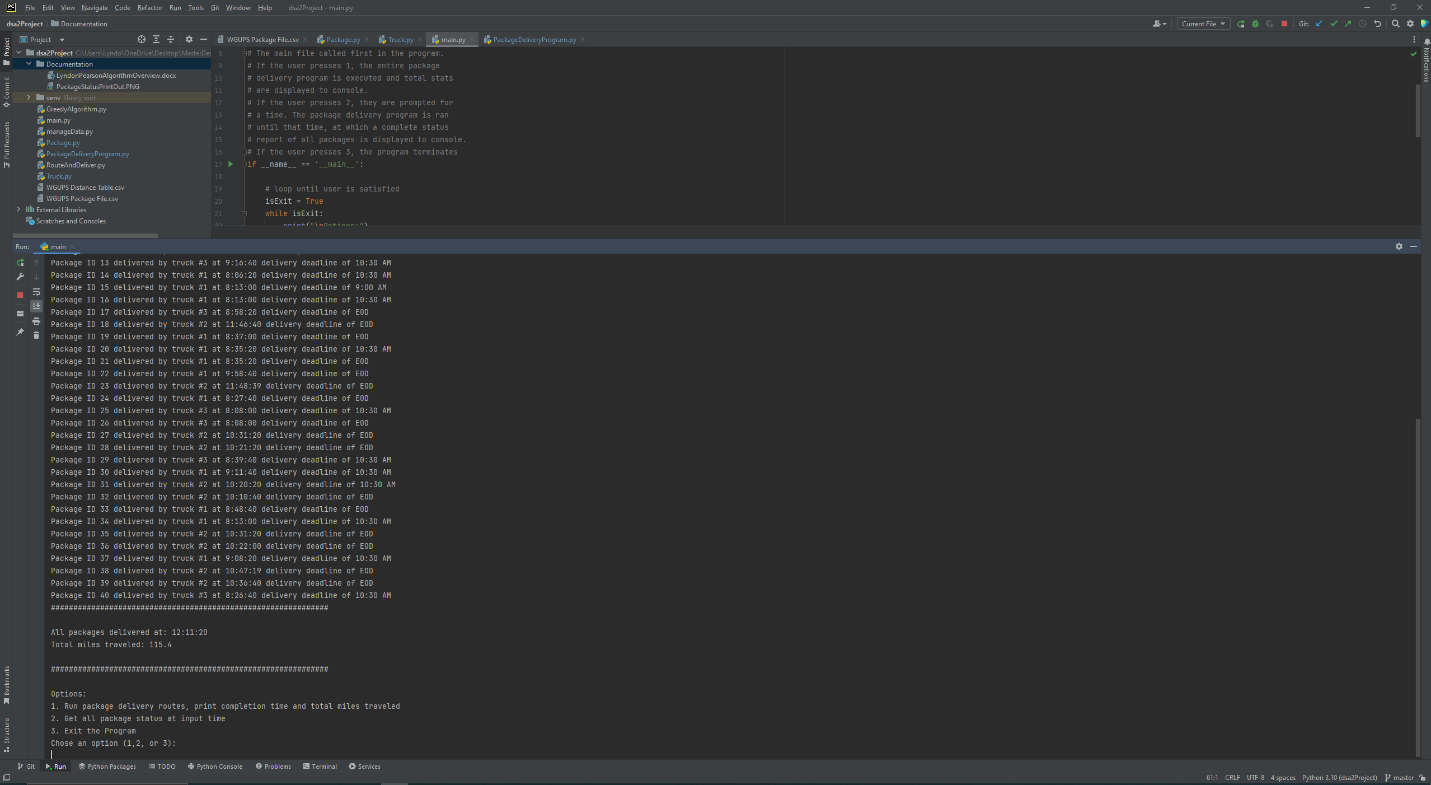
# G3. Third Status Check





# H. Screenshots of Code Execution





# I1. Strengths of Chosen Algorithm

The greedy algorithm used in this program is simple to implement and understand. In less than ten lines of code, the algorithm iterates through all packages, identifies the one with the shortest delivery distance, and returns that distance and the package’s ID.

The greedy algorithm also does well optimizing these types of scenarios although the solution may not be globally optimized.

# I2. Verification of Algorithm

The algorithm has been verified to meet all requirements in the scenario such as delivery deadlines. Delivery deadlines are printed to console with all package statuses showing each deadline has been met.

# I3. Other possible Algorithms

Since there are no negative distance values, Dijkstra’s algorithm would be a suitable alternative. Another algorithm would be Bellman-Ford which, while slower than Dijkstra’s algorithm, does provide the capability for handling negative numbers.

# I3A. Algorithm Differences

Dijkstra’s algorithm typically finds the shortest distances from an assigned “source” node to all other nodes. It can be used in a similar way as the greedy algorithm is implemented if it is stopped once the destination node distance has been determined.

The Bellman-Ford algorithm is similar but with the added flexibility of handling negative values. This comes at a cost of increased time consumption and overhead.

# J. Different Approach

I would likely track time using some of the standard Python library tools as there are likely better and easier implementations. In addition, having a self-adjusting component that reads in package data and loads trucks accordingly would make the program more automated and flexible.

# K1. Verification of Data Structure

The hash table used in this program meets all requirements in the scenario and laid out by the

rubric. The table has a look-up complexity of O(1) as the hashing on the input key returns the

containing list index. Data storage for the table would grow by O(n) as the number of packages

increased as each package would need to be stored in the table. The program now outputs each

package delivery deadline to console by using the “getDeadline” method for each Package

object.

# K1A. Efficiency

For my implementation of the scenario, increasing the number of packages would not alter the

look-up time for the hash table. As a result, the complexity would be considered O(1) as the

hashing and look-up is independent of the number of packages.

# K1B. Overhead

The space usage would be considered O(n) as given an increased number of packages, the

storage space for the hash table would be grow linearly.

For my implementation of the scenario, increasing the number of trucks would not alter the look-

up time. As a result, the complexity would be considered O(1) as the hashing and look-up is

independent of the number of trucks.

The space usage would also be considered O(1) as given an increased number of trucks, the

storage space for the hash table would be unimpacted.

# K1C. Implications

Increasing the number of cities would impact lookup times by increasing to an O(n) operation.

This assumes each city has its own hash table with a unique identifier, and the need to loop

through a list of the hash tables to retrieve the appropriate city. Once this step has completed, the

lookup time and algorithm would be identical to the single-city application in this project.

Space usage would also grow at O(n), where n would grow linearly with the number of cities and

represent the hash table for each city.

# K2. Other Data Structures

Two other data structures that could meet project requirements would be a standard Python dictionary and a linked list.

# K2a. Data Structure Differences

A standard dictionary in Python would be very similar to the hash table implemented in the

program. There would be some loss in flexibility using standard dictionary when compared to

the hash table implementation of hashing, for example. The benefit would be gaining access to

Python’s built-in library of dictionary methods instead of creating them.

A linked list is a data structure that could meet the requirements in this scenario. In a linked list

approach, there would be no “buckets” containing items (key-value pairs), but rather each item

would have tail pointer that could be set to the following item. The benefit here would be a set

order, but searching, removing, or adding an item would be O(n) since the list would need to be

traversed entirely if many cases.

# L. Sources - Works Cited

zyBooks (n.d.). Hash Tables. ZyBooks Catalog. Retrieved March 6, 2023, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/7/section/1>

Ashe, J. (n.d.). C950 Data Structures & Algorithms II. C950 Webinar Getting Started. Retrieved March 1, 2023, from <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=2ef1dff8-7cf5-4d9a-aa57-ad1900e87c2c>

Tepe, C. (n.d.). C950 Data Structures & Algorithms II. C950 Webinar I - Let's Go Hashing. Retrieved February 27, 2023, from <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71>

Tepe, C. (n.d.). C950 Data Structures & Algorithms II. C950 Webinar II - Getting Greedy, who Moved my Data? Retrieved February 27, 2023, from <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=eee77a88-4de8-4d42-a3c3-ac8000ece256>

Wikipedia (n.d.). Dijkstra's algorithm. Retrieved March 3, 2023, from <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

Wikipedia (n.d.). Bellman-Ford algorithm. Retrieved March 3, 2023, from <https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm>