Core Algorithm Overview

**Stated Problem:**

This project’s purpose is to create an algorithm designed to determine the best route for a WGUPS delivery truck to take to efficiently deliver a given number of packages. The project is written in Python 3.8. There are 40 packages to deliver which must be divided up between three trucks. However, there are only two truck drivers so only two trucks can be out at any point in time. Furthermore, some packages have delivery restrictions. For example, one package has the incorrect address and will be updated at a specified time. There are also packages that must be delivered by a certain time. To solve this problem, the program opens the necessary CSV files that supply the distance and package data and then iterates through the files to standardize and populate three lists that contain header information and distance information. The program then implements a greedy algorithm to iterate through the packages on a WGUPS truck, find the closest destination from the truck’s current location, and drop off any packages that need to be delivered at that address. The greedy algorithm repeats until there are no more packages on the WGUPS delivery truck. This algorithm is considered greedy because it determines the shortest path based on the delivery truck’s current location and does not consider future options. This article will analyze the use of this greedy algorithm and provide a detailed overview of the application’s components.

**A: Algorithm Selection**

The program’s core algorithm is a greedy algorithm that uses a “nearest neighbor” concept in which the algorithm selects the next closest delivery stop based on the truck’s current location without regard to future options. The algorithm also implements self-adjusting method where each WGUPS delivery truck object can scale its path based on how many packages it must deliver. This self-adjusting method helps ensure that a truck can deliver as many or as little packages as necessary.

**B1: Logic Comments**

My core algorithm finds a solution to the stated problem by reading three CSV files. Two of those CSV files are header information, which is read into a one-dimensional list. The remaining CSV file contains distances in floating-point notation, which is read into a two-dimensional array. The algorithm also standardizes the header information to create a graph and compare vertices and edges within that graph more easily. The algorithm then populates a graph object that represents all the stops the truck must take and the routes between those stops. This graph is then used to find the nearest neighbor from the truck’s starting point, the WGUPS hub. Once the truck arrives at the next delivery point, the truck drops off any packages that need to be delivered there, finds the next nearest delivery point and travels to that delivery stop. This process is repeated until there are no more packages for that truck to deliver. Once the truck has no more packages to drop off, the truck will return to the hub.

The following pseudocode provides an overview of how the greedy algorithm operates.

Greedy Algorithm Pseudocode

1. Read in the following CSV files:
   1. Distance Table as a two-dimensional array
   2. Horizontal headers as a list
   3. Vertical headers as a list
2. Standardize the verticalHeaders and horizontalHeaders list
   1. Look through both lists and replace “South” with “S”, “North” with “N”, “East” with “E”, and “West” with “W”.
   2. Special case: “West” occurs in the hub name, “Western Governors”. If “Western Governors” is currently being worked on, do not replace “West” with “W”. Continue to the next street address.
   3. Special case: “Sta” is in one header list and “Station” is in another header list. Replace “Sta” with the word “Station” except if “State” is in the street address.
   4. Remove zip codes from vertical header list for standardization
   5. Look at only the street address in the horizontal header list for standardization
   6. Replace the word “HUB” in the vertical headers list for standardization
3. Create a graph object for all packages and edges
4. Populate the graph object with all the destinations and weighted edges between those destinations

For every row in the distance table

For every column in that row

If accessing that row and column results in ‘’, go to the next row

If accessing the street address in the vertical headers and horizontal headers are the same, go to the next row

Add an bidirectional edge between two street addresses, using the distance table to provide the weight for the edge

* 1. This requires a nested for loop, which results in a time complexity of O(N^2)

1. Compare the delivery truck’s package manifest and add a vertex to the truck’s path graph object if the package’s street address matches a vertex in the overall graph object.

For every package in the delivery truck

For every vertex in the overall graph

Compare the package’s street address and the vertex’s label. If they match, add that vertex to the truck’s path graph object.

* 1. This requires a nested for loop, which results in a time complexity of O(N^2)

1. Set the hub’s location
2. Add the hub’s adjacency list and edge weights to the truck object’s path

For every vertex in the truck object’s vertex list

Add a bidirectional weighted edge between that vertex and the hub

* 1. This requires a single for loop, which results in a time complexity of O(N)

1. Add all the adjacency lists and edge weights for all the stops to the truck’s path

For every vertex (vertexA) in the truck object’s vertex list

For every vertex (vertexB) in the truck object’s vertex list

If vertexA and vertex are the same, compare vertexA to the next vertex in the truck object’s vertex list, do not execute the next line of code

Add a bidirectional weighted edge between the two different vertices

* 1. The two nested for loops result in a runtime complexity of O(N^2)

1. Set an arbitrarily high floating-point variable to assist in finding the nearest neighbor to the hub (closestDistToHub)
2. For every edge in the truck’s list of edges, if the edge’s weight is smaller than closestDistToHub, the edge’s weight is not 0, and the truck object’s current location is on the left side of the (vertexA, vertexB) tuple (which represents the truck’s current location), then make closestDistToHub equal to the edge weight and make the edge the nearest neighbor
   1. Results in a runtime complexity of O(N)
3. Depart the hub and update all the packages on the truck object to reflect what time the package left the hub
4. Calculate the time needed to travel to the closest destination and add that time difference to the truck object’s timer
5. Set the truck object’s new location as the destination it traveled to
6. Add how far the truck traveled to the truck’s odometer
7. Add the truck object’s new location to the locations the truck has already visited so that it does not visit it again
8. Drop off any packages at the truck object’s new current location and update their statuses to say “DELIVERED” and time to reflect what time the package was dropped off
9. Remove the path the truck object took from the truck object’s edge weight list to prevent the truck from taking the path again.
10. Until the truck object no longer has packages to deliver:
    1. Set an arbitrarily high floating-point variable to assist in finding the nearest neighbor to the truck object’s current location (closestDist)
    2. Iterate through every edge in the truck’s list of edges, if the edge’s weight is smaller than closestDist, the edge’s weight is not 0, the truck object’s current location is on the left side of the (vertexA, vertexB) tuple (which represents the truck’s current location), and the truck has not visited the location before, make closestDist equal to the edge weight and make the edge the nearest neighbor
       1. Results in a runtime complexity of O(N)
    3. Calculate the time needed to travel to the closest destination and add that time difference to the truck object’s timer
    4. Set the truck object’s new location as the destination it traveled to
    5. Add how far the truck traveled to the truck’s odometer
    6. Add the truck object’s new location to the locations the truck has already visited so that it does not visit it again
    7. Drop off any packages at the truck object’s new current location and update their statuses to say “DELIVERED” and time to reflect what time the package was dropped off
    8. Remove the path the truck object took from the truck object’s edge weight list to prevent the truck from taking the path again
11. After the last package has been delivered, find closest path back to the hub
    1. Set an arbitrarily high floating-point variable to assist in finding the nearest neighbor to the truck object’s current location (closestDistToHub)
    2. Iterate through every edge in the truck’s list of edges, if the edge’s weight is smaller than closestDist, the edge’s weight is not 0, the truck object’s current location is on the left side of the (vertexA, vertexB) tuple (which represents the truck’s current location), and vertexB in the (vertexA, vertexB tuple) (which represents the potential next destination) is the hub, make closestDist equal to the edge weight and make the edge the candidate for the closest way back to the hub
    3. Calculate the time needed to travel back to the hub and add that time difference to the truck object’s timer
    4. Set the truck’s new current location as the hub
    5. Add the miles traveled to the truck’s odometer
12. Overall runtime complexity of the greedy algorithm used is O(N^2)

**B2: Application of Programming Models**

The programming language used in this project is Python 3.8.5. The environment used to develop this project was PyCharm. As all the necessary files are included with this project, there are no communication protocols or server-client communication taking place.

**B3: Space-Time Complexity and Big-O**

The greedy algorithm has a worst-case runtime of O(N^2). Below is a table detailing the worst-case space-time complexity of each file of the application:

Hashtable.py

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Line Number | Space Complexity | Time Complexity |
| \_\_init\_\_ | 5 | O(N^2) | O(N) |
| insertPackage | 10 | O(1) | O(1) |
| Search | 19 | O(1) | O(1) |
| Remove | 28 | O(1) | O(1) |
| **Total** |  | O(N^2) | O(N) |

Main.py

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Line Number | Space Complexity | Time Complexity |
| None | 10 | O(N^2) | O(N^2) |
| None | 15 | O(N) | O(N) |
| None | 23 | O(N) | O(N) |
| None | 32 and 33 | O(N) | O(1) |
| None | 37 | O(1) | O(N) |
| None | 45 | O(N^2) | O(1) |
| None | 49 | O(1) | O(N) |
| None | 64 | O(1) | O(N) |
| None | 68-74 | O(N) | O(1) |
| None | 77 | O(1) | O(N) |
| None | 94 | O(N^2) | O(N^2) |
| None | 97 | O(N) | O(1) |
| None | 115 | O(N^2) | O(N^2) |
| None | 118 | O(N) | O(1) |
| None | 139 | O(N^2) | O(N^2) |
| **Total** |  | O(N^2) | O(N^2) |

Truck.py

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Line Number | Space Complexity | Time Complexity |
| \_\_init\_\_ | 10 | O(N) | O(1) |
| buildRouteAndTravel | 52 | O(N^2) | O(N^2) |
| **Total** |  | O(N^2) | O(N^2) |

Graph.py

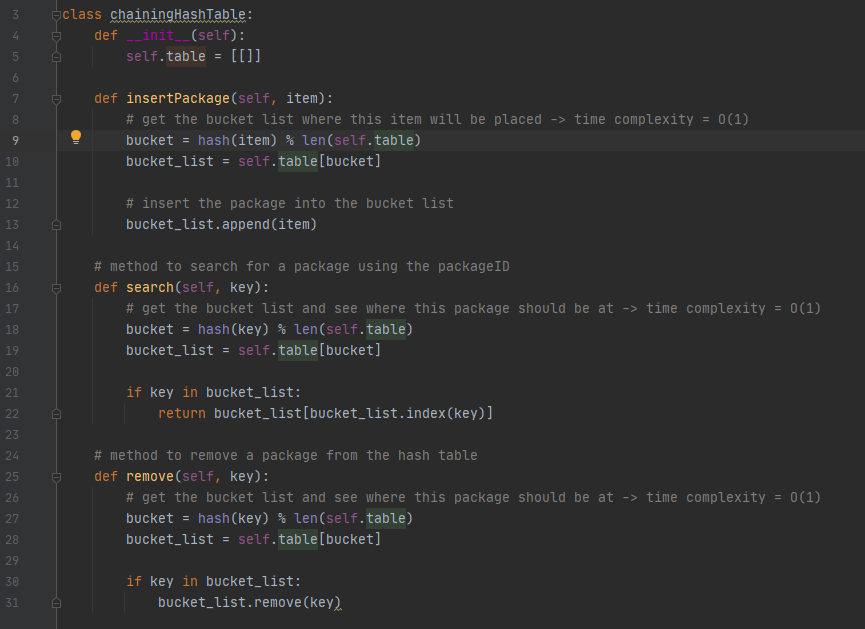
|  |  |  |  |
| --- | --- | --- | --- |
| Method | Line Number | Space Complexity | Time Complexity |
| \_\_init\_\_ | 4 | O(N) | O(1) |
| Add\_vertex | 10 | O(N^2) | O(1) |
| Add\_directed\_edge | 17 | O(1) | O(1) |
| Add\_undirected\_edge | 22 | O(1) | O(1) |
| getVertex | 31 | O(1) | O(N) |
| getDistance | 37 | O(1) | O(1) |
| **Total** |  | O(N^2) | O(N) |

packageClass.py

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Line Number | Space Complexity | Time Complexity |
| None | 3 | O(N) | O(1) |
| \_\_init\_\_ | 7 | O(1) | O(1) |
| \_\_hash\_\_ | 26 | O(1) | O(1) |
| \_\_eq\_\_ | 30 | O(1) | O(1) |
| **Total** |  | O(N) | O(1) |

**B4: Adaptability**

The self-adjusting algorithm used by the greedy algorithm scales well as the truck can add as many packages as it needs, and the routing will still work the same. The self-adjusting algorithm also uses a hash table that is built to handle any number of packages. As the number of packages grows, the self-adjusting hash table grows to hold that number of packages. Line 13, which is a part of the insertPackage method of the chainingHashTable class, highlights the self-adjusting component of the hash table. Bucket\_list, which is a list object within the “table” list object, holds the package objects. Bucket\_list can append a large amount of package objects with no impact to the runtime complexity as chaining hash tables have a runtime complexity of O(1).



**B5: Software Efficiency and Maintainability**

The entire program has a runtime and space complexity of O(N^2). Regarding maintainability, the program is easily maintained as variables are appropriately named, comments precede all major code blocks, and opportunities exist for enhancement by developers other than the author.

**B6: Self-Adjusting Data Structures**

The hash table built is self-adjusting as any number of packages can be inserted into the hash table with little regard to hash collisions. Each bucket in the hash table can accommodate any number of packages. Considering runtime, the complexity of accessing a package object will always be O(1) despite the number of packages that are inserted into the hash table.

**C: Original Code**

The code produced is original and designed by the author. The total mileage traveled by all three trucks is 103.3 miles.

**C1: Identification Information**

Author information is included in the first line of each Python file of this project.

**C2: Process and Flow Comments**

Every major block of code includes preceding comments that discuss the logic and process of that block of code.

**D: Data Structure**

The hash table used in the program is a data structure that was written using only primitive data structures. The main data structure used in building the hash table is a list which incorporates the use of buckets to help find packages and prevent hash collisions. Buckets are essentially a list within a list.

**D1: Explanation of Data Structure**

The program uses a chaining hash table that provides both scalability and prevention of hash collisions. The hash table constructor creates the initial table. The insertPackage method takes the package object, hashes it, and performs a modulo operation with the size of the table to determine which bucket the package should be inserted. The modulo operation uses the size of the table to ensure that there will be a bucket for the package to go into. The search method uses the package’s ID number to search for which bucket the package should be in and searches through the bucket to find the package. If the package is found, it returns the package. The remove method works the same as the search method. The difference between the remove method and the search method is the remove method will remove the package if it is found rather than returning the package object to the calling function. The hash table is mainly used to load packages onto the WGUPS delivery trucks.

**E: Hash Table**

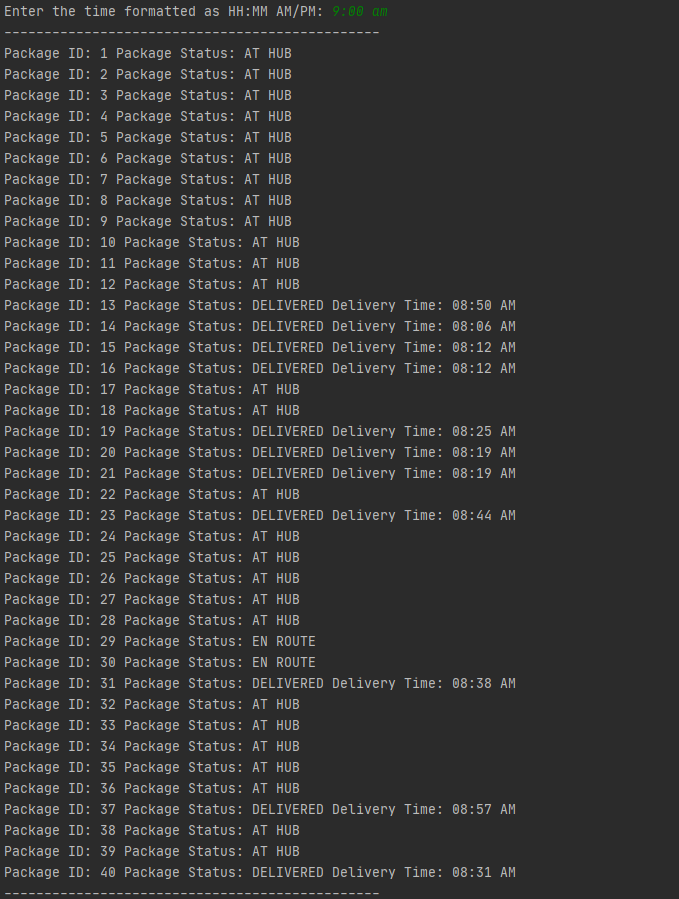
The hash table incorporates an insertion function which allows for the insertion of package objects.

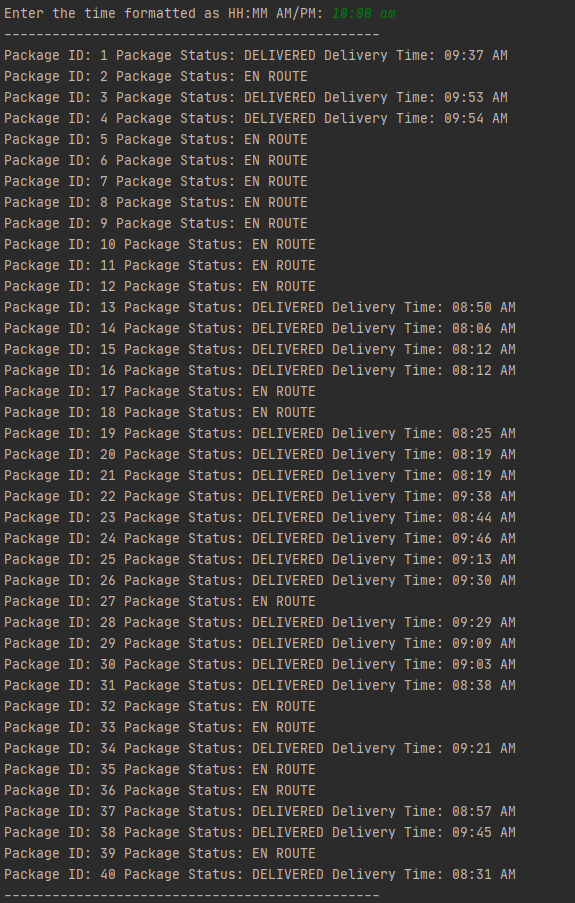
**F: Look-up Function**

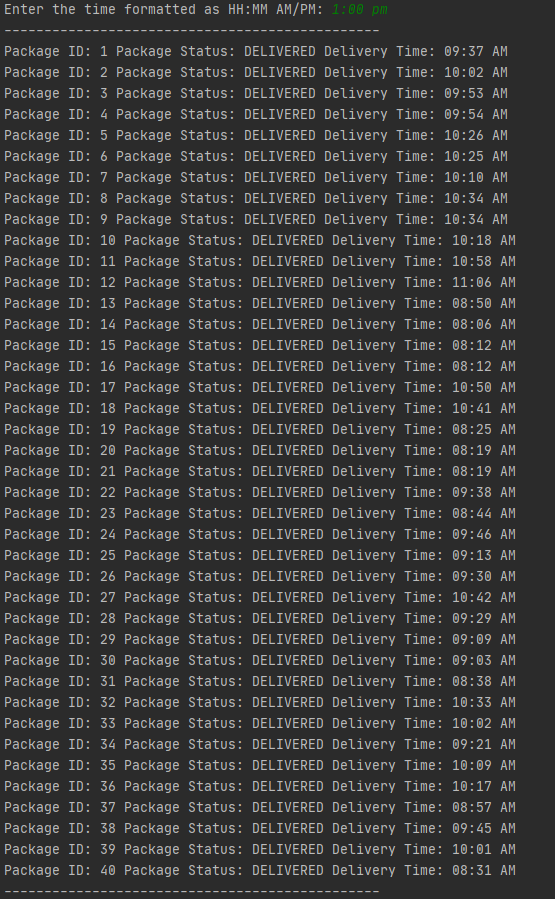
The hash table provides a look-up function that takes the package’s ID number as a parameter, hashes and modulos the ID number in order to find which bucket the package should be in, and searches that bucket to find the package. If the package is found, the method returns the package and all its information.

**G: Interface**

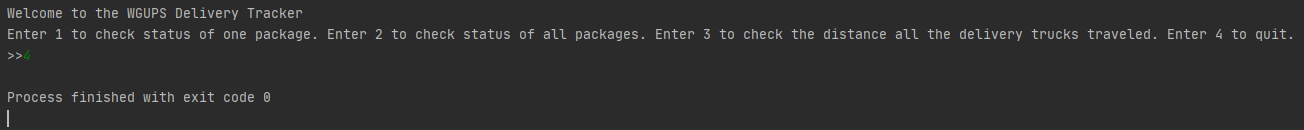
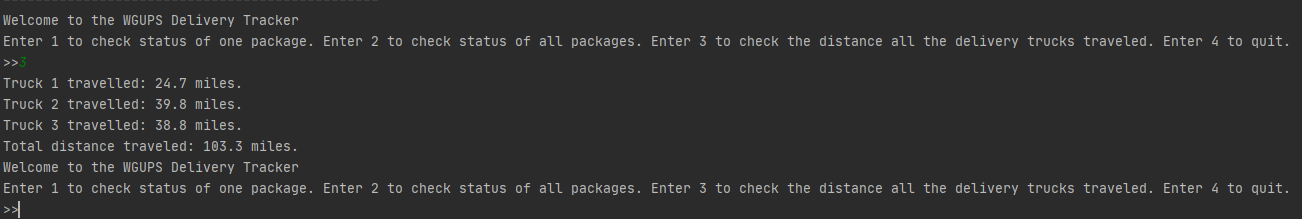
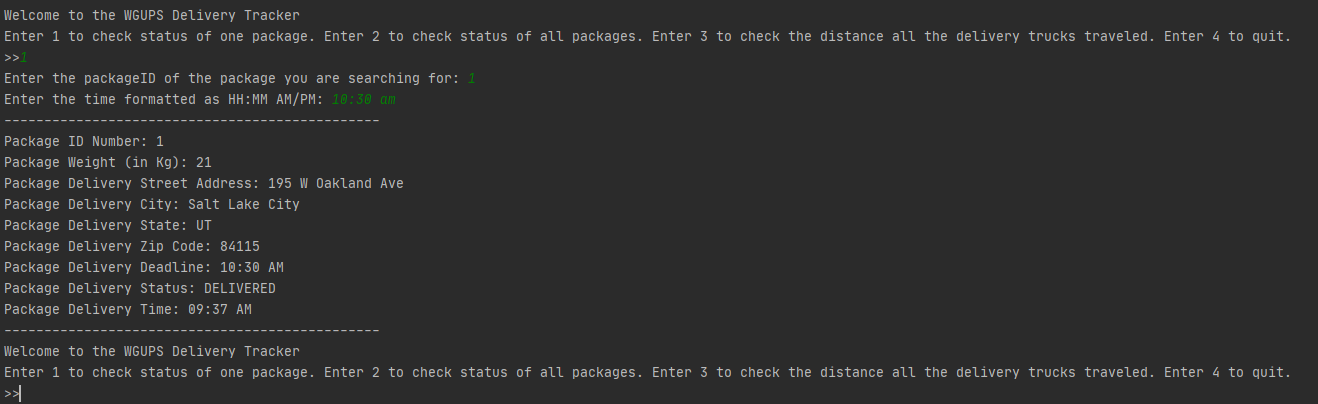
The program provides a basic command-line user interface to allow the user to check the status of one package at a specified time, statuses of all packages at a specified time, distance that all the delivery trucks traveled, or end the program.

**G1: Screenshot of All Packages between 8:35am and 9:25am**

**G2: Screenshot of All Packages between 9:35am and 10:25am**

**G3: Screenshot of All Packages between 12:03pm and 1:12pm**

**H: Screenshots of Code Execution**



**I1: Strengths of the Chosen Algorithm**

The strengths of the greedy algorithm used are that it was easy to implement and works relatively well for solving this problem while staying under 140 miles travelled by all trucks.

**I2: Verification of Algorithm**

The algorithm is successful in solving the problem as proven by the total mileage of the trucks and all packages being delivered within their constraints. The total mileage traveled by the trucks is 103.3 miles.

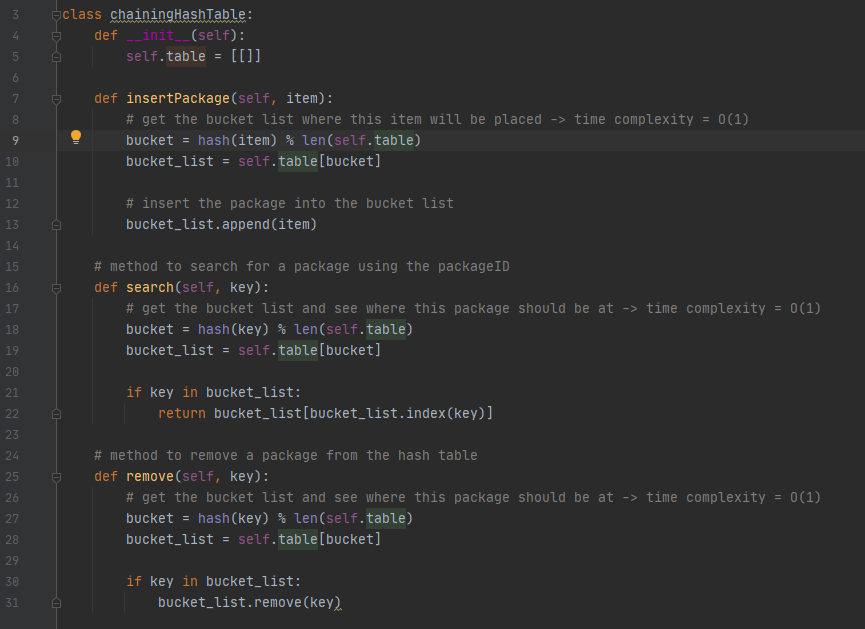
**I3: Other Possible Algorithms**

Two other algorithms that could have worked would be a genetic algorithm or Dijkstra’s shortest path algorithm. Dijksta’s algorithm would have worked better than the implemented algorithm; however, it is more difficult to implement. A genetic algorithm works by creating a random route, calculating how efficient it is, comparing how efficient it is to other random routes, then using the best paths from the random routes to generate a better initial route (Stoltz, 2018). I chose to implement a greedy algorithm over a genetic algorithm due the complexity involved with a genetic algorithm.

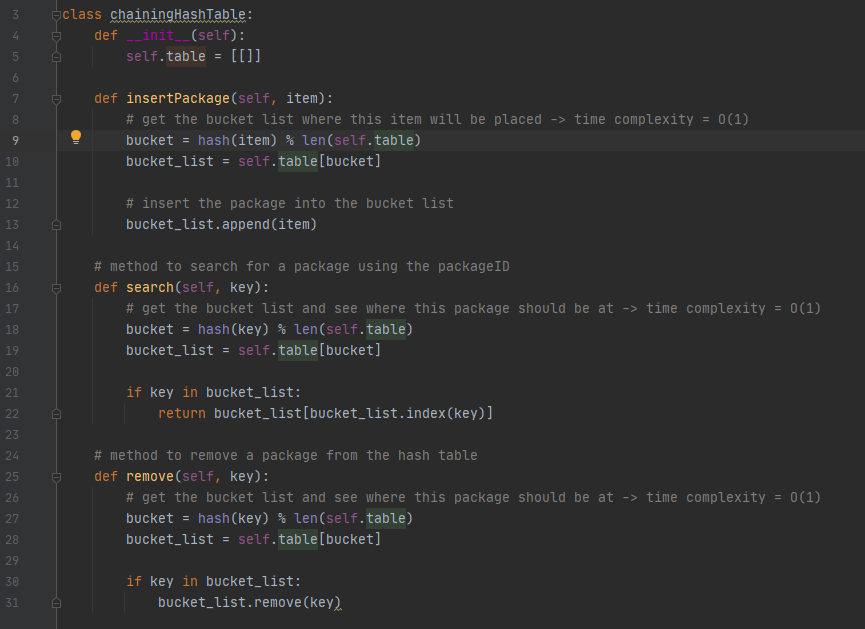
**J: Different Approach**

If I were to build this project again, one thing I would do differently would be to implement a way to automate the loading of packages onto the various WGUPS trucks versus manually loading the trucks.

**K1: Verification of Data Structure and Solution**

As seen in Section H of this article, the total mileage of the trucks stays within the bounds of mileage constraint. Total mileage traveled by the trucks is 103.3 miles. Section G3 of this article details the delivery times of all the packages. All packages are delivered by their deadline and according to their special notes. Section H and G3 also verify that reports on the total mileage traveled and packages’ statuses are accurate and displayed in a user-friendly manner. Below is a screenshot of the source code that verifies that a hash table was implemented and includes a look-up function. 

**K1A: Efficiency**

The hash table stores package objects and is primarily used to allow for quick and easy searching of package objects by their package ID number. The insertPackage function of the hash table takes the package object as an argument and uses the \_\_hash\_\_ magic function declared in packageClass to hash the package’s ID number, which is an integer, to find which bucket the package belongs. The search function of the hash table uses the package’s ID number to determine which bucket the package belongs and searches that bucket for the package object. The remove function of the hash table hashes the package’s ID number, which is an integer, to determine which bucket the package object should be in, searches that bucket, and removes the package object if it is found. The hash table allows the program to be more efficient due to the fact that its runtime complexity is O(1), which is significantly quicker than implementing a list and iterating through the list one item at a time. 

**K1B: Overhead**

The hash table implemented provides for a quick runtime complexity of O(1) and space complexity of O(N^2) due to its use of lists within a list. Bandwidth and memory used is not a concern for this project due to its lack of communication protocols and server-client communication.

**K1C: Implications**

Adding more packages to the hash table does not impact hash table performance to a big extent. The hash table can scale relatively well to handling large package manifests. This is evidenced by the fact the hash table prevents collisions between package objects. However, this hash table can be improved upon by implementing more buckets if the total number of packages exceeds a very large number. Implementing more buckets would provide for more scalability.

**K2: Other Data Structures**

Two alternative data structures that could have been used would be a linear probing hash table or a direct hash table.

**K2A: Data Structure Differences**

A linear probing hash table prevents collisions by starting at the bucket where the package would initially map to then linearly searching subsequent buckets until an empty bucket is found. This type of hash table would result in a large single list to prevent collisions. A direct hash table works by hashing the package ID number then placing the package in the bucket that corresponds to that hash. The problem with this type of hash table is that it results in a very large list.

**L: Sources**

Stoltz, E. (2018, July 17). Evolution of a salesman: A complete genetic algorithm tutorial for Python. Retrieved October 21, 2020, from https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35