Scott Daily

C950

12/28/2020

Rubric Requirements

**A.**

The core algorithm that is implemented to deliver the packages in a timely manner is a form of a greedy algorithm or nearest neighbor. The algorithm first sorts the packages loaded on a truck depending on whether they have a deadline attached or have no deadline. The route is first built by examining the packages that possess deadlines and ordering them so that the earliest delivery deadline comes before a package with a later delivery deadline. Next the packages without deadlines are sorted by comparing the last address added to the route and all the addresses of the remaining packages to deliver. The algorithm will then add the nearest address to the last address added to the route list.

**B1.**

**1)**

The core algorithm begins execution by creating six lists. The first list will contain a list of addresses the delivery truck must visit in the order they appear after being returned from the function. This is called the route list. The next is a list to contain the custom package objects of the packages that have an attached deadline. Another is a list of the packages that have no deadline. Next, two more list are created, the first to contain the addresses of the priority packages and the second to contain the addresses of the non-priority packages. Finally, another list is initialized that is a copy of the list of package ID’s from the specific Truck instance’s internal structure to refer to during the core algorithms execution.

**2)**

The first operation conducted is to iterate through the package ID’s in the package list and check to see if they contain a delivery deadline or not. If they do have a deadline, then it checks to see if the address already exist in the priority package address list. If it does not, then the packages address is added to this list, and the entire package object is also added to the priority package’s list. If the package is not a package with a deadline, then the same thing is done except it is added to the appropriate non-deadline package list.

**3)**

Next a while loop is used so that all the packages in the priority package list can be iterated over and added in the correct order to the route list of addresses. The while loop will break when the package list has only 1 package left, and will automatically append this final remaining package address to the route list that will be returned from the core algorithm.

**4)**

If there is more than one priority package, then the packages will be sorted by their delivery deadline time. This is done with the use of the python time module’s strptime method. This method converts a string into a time object. The purpose of this is so that the package’s delivery time can be compared to find out if a time is less than or greater than another package’s deadline. A variable is created called “sorted\_priority\_packages” that will be assigned the first package in the sorted package list. This package will have the earliest deadline or be tied for the earliest.

**5)**

Next a for loop will iterate over each package in the priority package list. This will initially compare each packages deadline and add the closest to the route list. If both times are equal, then the else clause will execute. In this loop, a variable to hold the “current\_least\_distance” will be initialized with the distance between the last address in the route list and the second address in the priority packages. All the packages remaining the priority package list will then have their distance between the last address added to the route list and the next package’s address. The current least distance will hold the closest set of addresses distance calculated so far. When all the packages have been iterated over, if the address is not already in the route list, it is added. Next, the package that was just added is removed from the “priority\_packages” list. This is so that the while loop will eventually hit the base case of only one package remaining and be able to exit without creating an infinite loop.

**6)**

Now that the priority packages are taken care of, the next section will append the packages that do not contain a deadline to the route list. First a list called “duplicate\_addresses” is created. This is performed in order to remove packages that have the same address as a package that had a deadline, so that the same address is not visited twice.

**7)**

Next, another while loop will begin and will break when the non-deadline package list has only one package left. This loop follows the exact same steps as step 5 above.

**8)**

Finally, the completed route list is returned by the function. This will be a parameter in the delivery simulation method that will actually iterate through each address in a route list and update the custom package hash table with required status updates, such as “En route” when the truck leaves the Hub or “Delivered” along with the delivery time when a package is “dropped off” at an address.

**B2)**

The programming environment used to the create the program was Visual Studio Code. The language utilized was Python 3.8.3. Because this program was created and run on a local machine, there is no communication protocol or defined interaction semantics. Since the excel files are located in the local folder as the python files, they are simply imported utilizing Python’s built in csv library.

**B3)**

The core route building algorithm has a space complexity of O(N) and a time complexity of O(N3 ). This is due to the algorithm containing a while loop and then two more nested for loops inside the while loop.

A listing of all functions within the program and associated space & time complexities are below, they are also listed in comments by each function. They are organized below according to which file they belong to.

**hashtable.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| insert | O(N) | O(N) |
| get | O(1) | O(1) |
| \_\_getitem\_\_ | O(1) | O(1) |

**data.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| import\_packages | O(N) | O(N) |
| import\_distances | O(N) | O(N) |

**truck.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| \_\_init\_\_ | O(1) | O(1) |
| loadPackage | O(1) | O(1) |
| unloadPackage | O(1) | O(1) |
| \_\_repr\_\_ | O(1) | O(1) |

**package.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| \_\_init\_\_ | O(1) | O(1) |
| packageView | O(1) | O(1) |
| setLocation | O(1) | O(1) |
| get | O(1) | O(1) |
| \_\_getitem\_\_ | O(1) | O(1) |
| \_\_setitem\_\_ | O(1) | O(1) |
| \_\_repr\_\_ | O(1) | O(1) |

**minutes.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| timeToMinutes | O(1) | O(1) |
| minutesToTime | O(1) | O(1) |

**routebuilder.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| buildRoute | O(N) | O(N4 \* logN) |

**deliverysim.py:**

|  |  |  |
| --- | --- | --- |
| **FUNCTION** | **SPACE COMPLEXITY** | **TIME COMPLEXITY** |
| runDeliverySim | O(N) | O(N7 \* logN) |

**B4)**

The core routing algorithm is scalable. It will work with more or less packages than the 40 packages that were delivered. It can also handle any amount of addresses. It will automatically calculate distances between all remaining package addresses and the last address added to the route. It will then choose whichever address had the shortest distance. This can be repeated an unlimited amount of times. The efficiency of the algorithm itself, which is O(N3), could likely be improved upon by approaching the solution differently. Since the time complexity growth rate is exponential, the more packages that must be routed will make the program exponentially slower as the number of packages increases. Similarly, adding additional addresses that must be delivered to would drastically slow down the program once it reaches a certain threshold.

**B5)**

The overall program’s time complexity is not optimally efficient. It can easily run with the amount of addresses and packages that were required, but if the number of either was to increase by a large factor, the program would slow down quickly. Therefore, a more efficient algorithm would likely be needed for large-scale delivery operations.

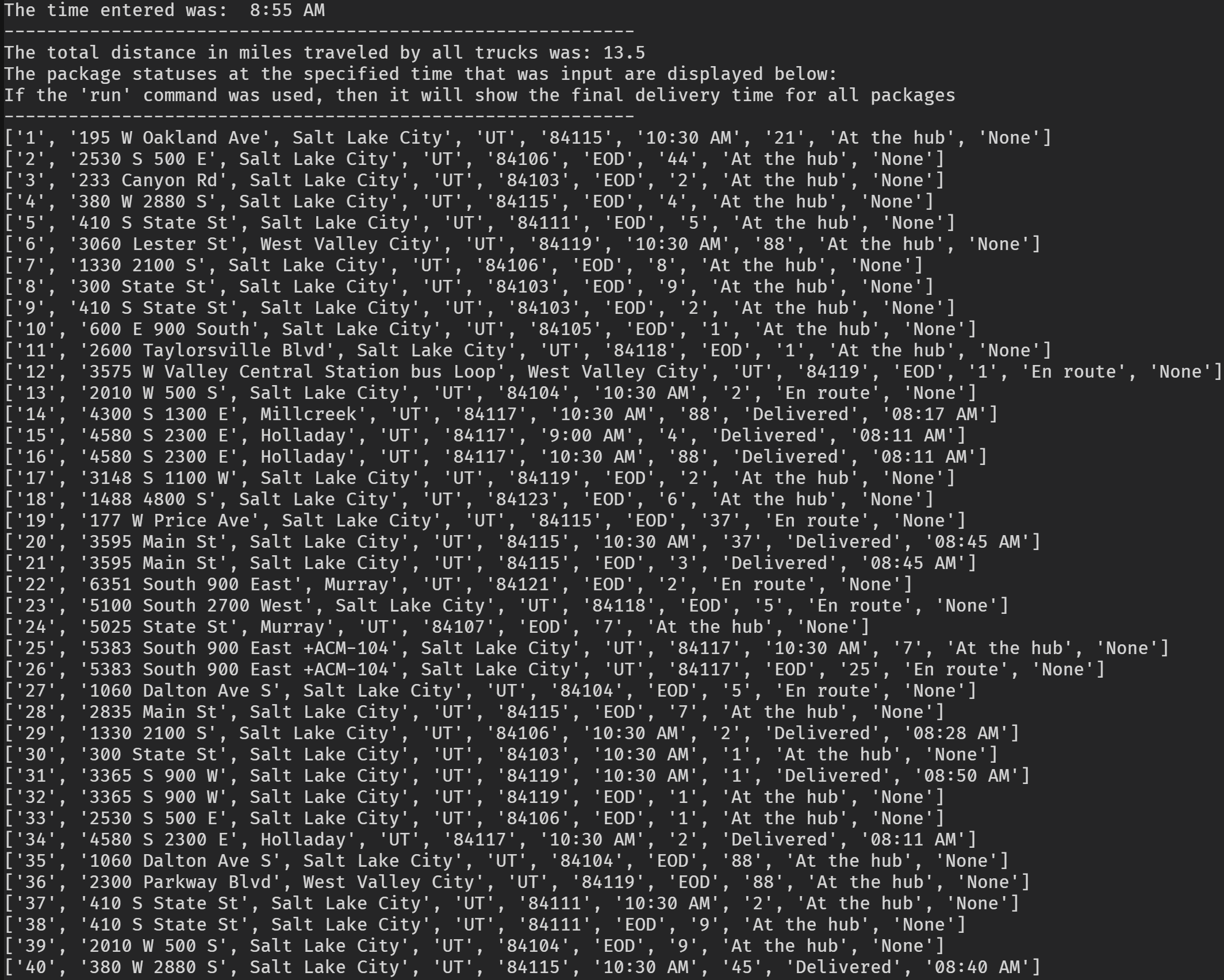
The program is well commented throughout to explain each block of code’s purpose. Within each function, more significant sections also include clarifying comments. This would allow someone else to understand and modify the program more efficiently. The program is also separated logically into different files with semantic file names. These are used to group similar functionality together and also make it easier to understand and change the program as needed by others.

**B6)**

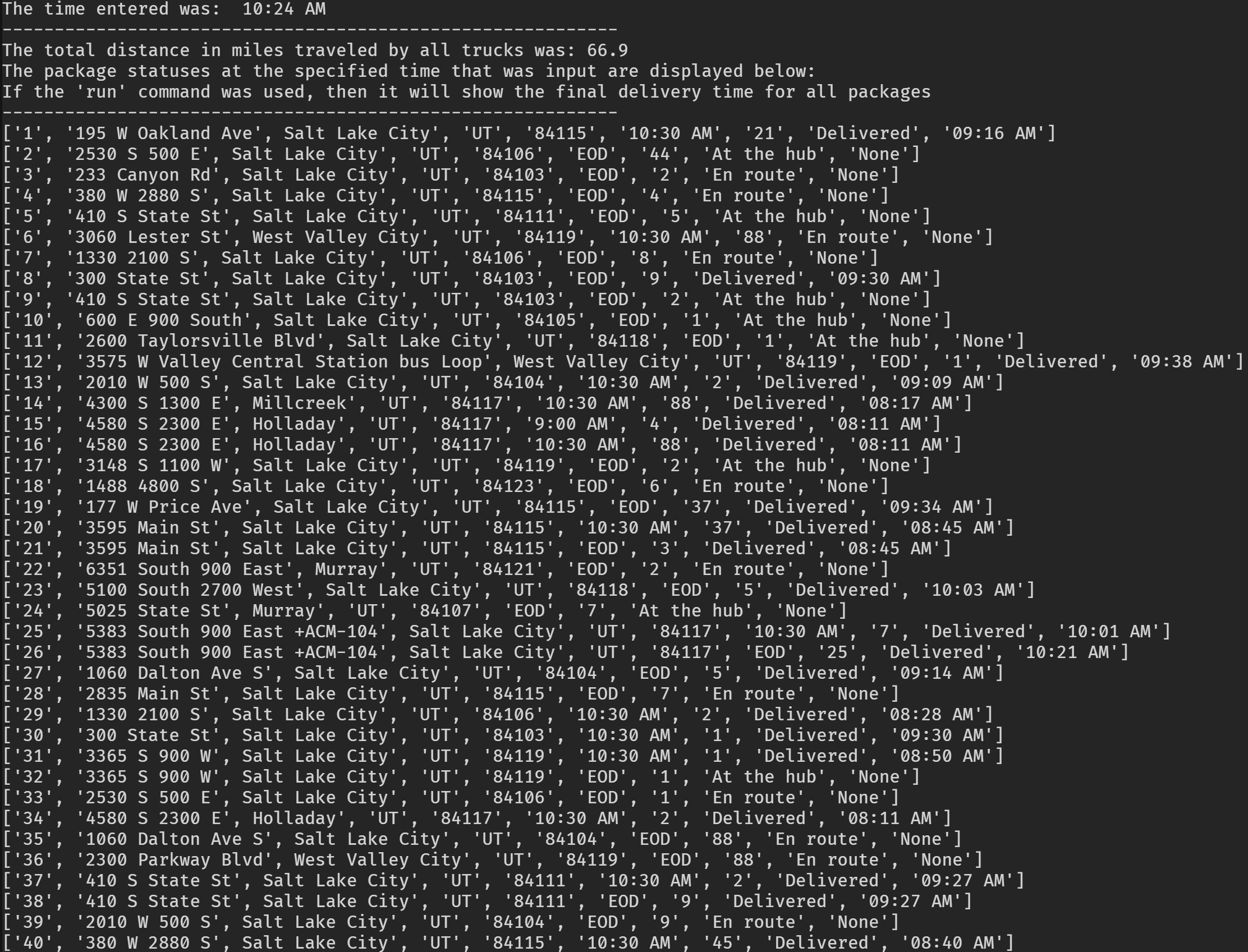
The self-adjusting data structure is a direct access hash table. This upside of this type of hash table is it always allows for O(1) access time, since only one object will ever occupy a bucket.

This implementation has two downsides: One is that the space complexity is equal to the number of objects that the hash table must hold. When dealing with a relatively small number of objects, it is fairly simple to implement and should have no issues. If there were a large amount of items, then this method could use too much space. Another reason is that when an object is inserted the hash table automatically adjusts it's capacity if the current size is not less than the current capacity. To increase it’s capacity it simply copies the old list of items and increases the list’s size by one more than the previous capacity, leaving space for the newly inserted object. This makes it self-adjusting. It also makes adding an object have a Big-Oh time complexity of O(N) and space complexity of O(N).

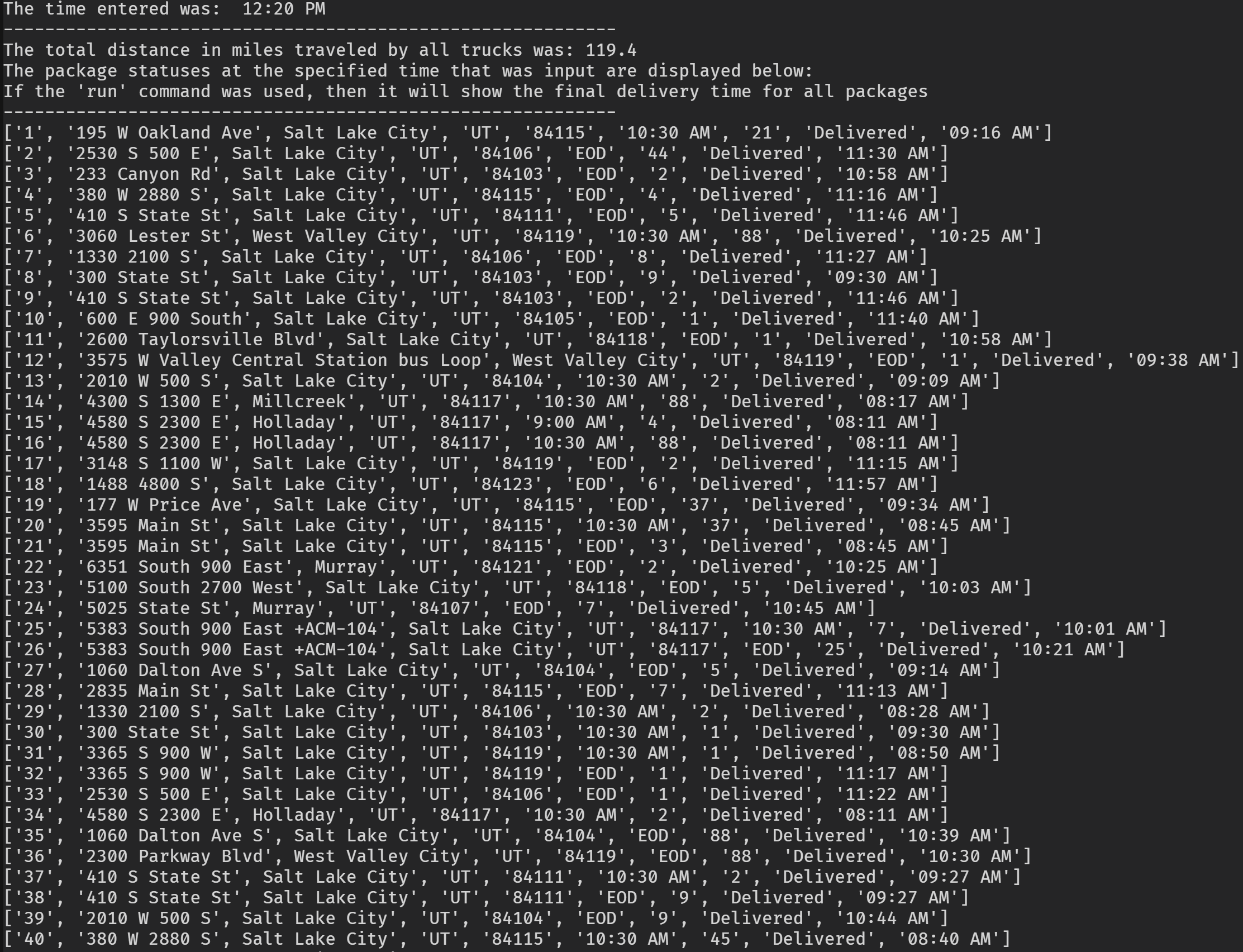
**G1)**

****

**G2)**

****

**G3)**



**I1)**

One strength of the core algorithm chosen is that it can adapt to a varying amount of packages or addresses that must be delivered to. It automatically splits packages up based on whether they are priority or not and is able to iterate through each group and build a route that is reasonably efficient, although not optimal. Since this is an NP-Hard problem, an optimal solution would take far too much time to compute.

Another strength is that the algorithm automatically checks to make sure if there are enough priority packages or non-priority packages to require sorting and if not, skips ahead. This is a self-adjusting feature of the algorithm.

**I3)**

One alternative algorithm that could have been used would be an algorithm that works in conjunction with a graph data structure that uses nodes to represent addresses and weighted edges as the distances between addresses. We could run a sort of Monte Carlo simulation where we calculate the total distance for a specific number of randomized routes and then choose the route that has the least distance required to deliver all packages.

Another approach could be to use supervised training with neural networks and have the neural network try many different routes and then we would use training to show if a route was good or not. Once the neural network was trained on a sufficient number of iterations of this, it should be able to come up with reasonably efficient routes quickly.

**13A)**

The core algorithm used was very different than the first alternative approach using a graph structure to represent the problem. The Big O time complexity of a graph based solution should be significantly faster than the core algorithm that was used. The core algorithm utilized many loops to operate, which slows it down considerably when the amount of packages or addresses goes up. A graph based algorithm might be able to as fast as O(N2) if built well.

The neural network approach is also very different than the core algorithm used. The neural network approach relies on the neural network learning how to build a route by being trained on many prior examples and being told whether it’s solution is good or not. This might require a lot of time upfront to train the network enough to be useful.

**J)**

One aspect of the program I would change would be to create a graph class and approach the problem by building an algorithm that can traverse the graph from nodes that represent addresses along weighted edges that represent the distances between addresses. This approach would be much more efficient than the core algorithm utilized.

Another part of the program that could be changed would be to utilize a different type of hash table, such as a linear probing hash table. This would allow several packages to be located in a single bucket, but would save space in memory compared to the direct access hash table.

**K1)**

All evidence requested can be seen in the included screenshots of the program running or in section B6.

**K1A & K1B & K1C)**

The hash table used in the solution stores data in the form of the custom Package object as a value. It uses the package’s unique ID as the key for the key-value pair. The direct access hash table allows for constant time access when retrieving a stored value. This is because each Package object stored is stored in it’s own unique bucket that is accessed by the Package’s ID. The efficiency is not as good when inserting new Package objects after the initial capacity is reached, since each time an object is added the Package objects internal list is copied and then the new item to insert is added. This makes the space complexity O(N) and time complexity O(N) on insertion.

Since the program only inserts the packages initially and the capacity is specified correctly to begin, this is not an issue. If a large number of packages was needed and the initial capacity requirement was unknown, then it could become a bottleneck in the program. The program uses the O(1) access time much more frequently to examine the attributes of the Package objects stored in the hash table.

Since the program is run on a local machine, bandwidth is not relevant. Memory is not a concern with such a small number of packages, but if this were to increase, then memory could an overhead issue.

The implications when more packages are added are noted already, as the time complexity to add new items is O(N) along with a space complexity of O(N). Adding additional cities or trucks would not have an impact on the current hash table.

**K2)**

Another type of hash table that could be used is known as a Linear Probing hash table. This kind of hash table can store more items in less space than an open addressing or direct access hash table, such as the one used in the program. In a linear probing hash table, each Package would be assigned a hashed key. If this ended up the same as another Package’s hash key, then there would be a collision. So the linear probing table would then check a predetermined amount of space after this location to check if it is available.

A second type could be a chaining hash table that generates a hash code that is the modulus of the size of the hash table, so that the resulting hash is guaranteed to match one of the current indexes. If two packages have the same hash, then they would be chained together in the form of a linked list.

**K2A)**

A Linear Probing hash table would insert a Package object differently than the hash table used. It would have to calculate a hash and then attempt to store the Package at this location. If the location was already used, then it would probe a set number of spaces ahead and check this location. The table would need to be resized once the amount of packages stored reached around 70% of the total capacity, otherwise the efficiency would fall. Most of the time an insertion would not require a time complexity of O(N) but would be O(1). In comparison, the direct access table used is implemented to require O(N) operations every time a new package is inserted after the initial capacity is reached. Accessing an item would have the same run time as the direct access table used.

The chaining hash table would require much less space, since each Package object would be converted into a hash code with the result being a location in the current hash table. Packages would build up as a linked list if two hashes went to the same index. The access time could slow down if too many collisions occurred resulting in long linked list on a single bucket.