**Core Algorithm Overview**

**PURPOSE STATEMENT:**

The purpose of this project is to develop a package delivery application using the Python programming language and algorithms to simulate effective and efficient delivery of packages around the Salt Lake City, Utah area. Provided were a list of packages and a list of locations to which those packages needed to be delivered. The goal of the application is to deliver all packages on time and fulfill any other requests per the special notes provided with the packages. The expected output of the program will achieve this goal, while providing capability to view the status of all packages at a time specified by the user.

**ALGORITHM SELECTION AND LOGIC COMMENTS:**

The application solution uses an implementation of the greedy algorithm. This is executed in the following manner:

1. An unsorted list is passed into the sorting algorithm.
2. The current location is set to the HUB
3. The current location is compared to the location of each other package in the list
4. The current location is set to the package location that is closest.
5. The package is then added to a sorted list and removed from the unsorted list.

Psuedocode:

*GREEDY ALGORITHM*

*Input: Unsorted List*

*Output: Sorted List*

*Current location := HUB*

*While length unsorted list > 0*

*Shortest := infinity*

*For each package in unsorted list*

*If current distance < shortest*

*Shortest := current distance*

*End If*

*End For*

*Append package to sorted list*

*Remove package from unsorted list*

*Return Sorted List*

*End While*

**APPLICATION OF PROGRAMMING MODELS:**

The programming model used in this application is limited as the application is hosted locally, written using Python 3.7, and executed in the PyCharm IDE. No communication protocol is needed since the application reads data from CSV files located in the project folder of the application. This read is performed through an import of the *csv* library in Python and calling the *csvreader* function. Data exchanges are therefore limited to the application and local machine. A network connection is not needed for this application, as written, so none is established, nor is a target host environment selected. As the application is hosted and run on a local machine, a set of interaction semantics to determine the flow of connection, data and information is not required.

**SPACE-TIME AND BIG-O:**

This application is has an overall time efficiency of at worst O(N2). The main components of the application include:

* Greedy Algorithm for sorting the packages.
  + This implementation of the greedy algorithm is O(N). It loops through the While loop for each package in the passed list. Then loops again using a for loop for comparison
  + The space complexity for this algorithm is O(N) due to the need to store the sorted and unsorted lists.
* The run\_truck function.
  + This function handles the delivery of the packages.
  + The tiem complexity is O(N). The while loop executes until there are no packages left on the truck.
  + Space complexity is O(1) since nothing need be stored that isn’t written over for later use.

**ADAPTABILITY:**

This application is somewhat scalable. Minor adjustments would need to be made to scale the application based on the number of packages, number of trucks used, and capacity of those trucks. The sorting algorithm could remain unchanged. The changes would be made in how the truck loads are broken down from the sorted list and which sublist gets loaded onto which truck.

The delivery location part of the algorithm is scalable. Minor tweaks would need to be made such as updated the location file and distance table. Once this was completed, the graph structure would adjust and account for the new location.

**SOFTWARE EFFICIENCY AND MAINTAINABILITY:**

Overall, the application is very efficient. Worst-case time complexity for the slowest comparisons in the application are O(N2). This is found in the chaining implementation of the Hash Table. It is also maintainable as the core code has been placed into functions so that it is reusable.

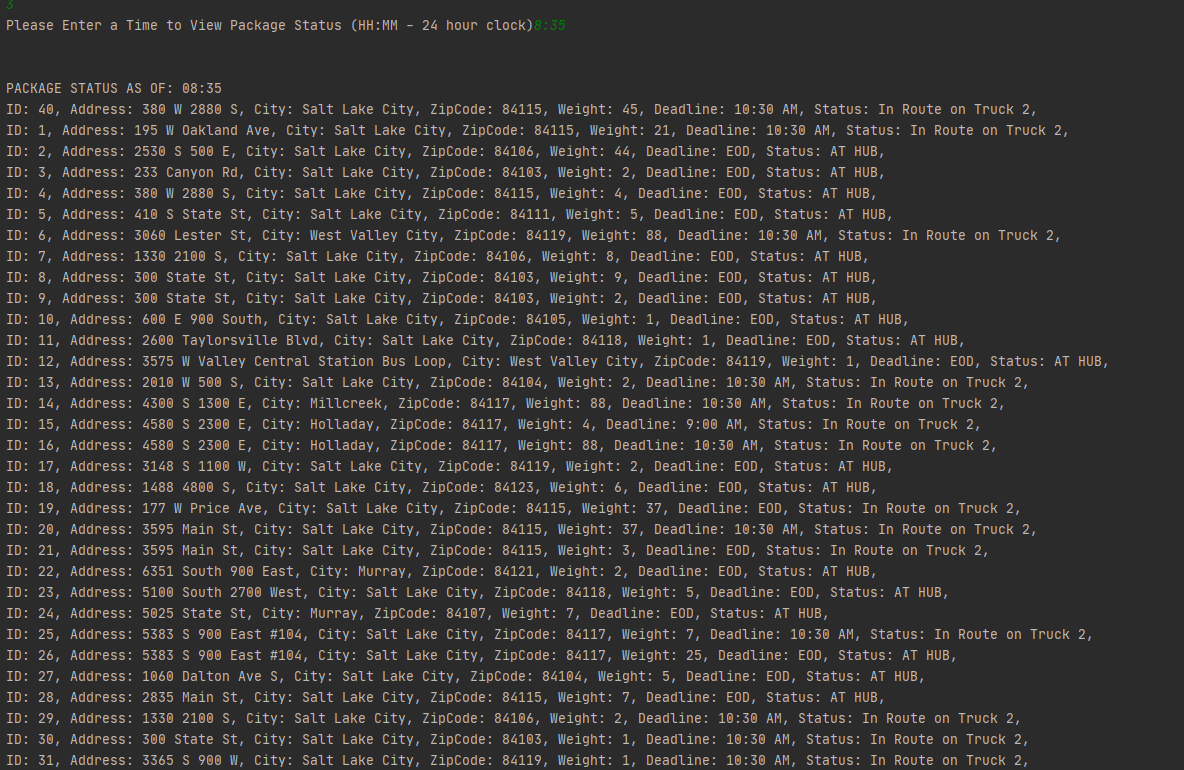
**SELF-ADJUSTING DATA STRUCTURES:**

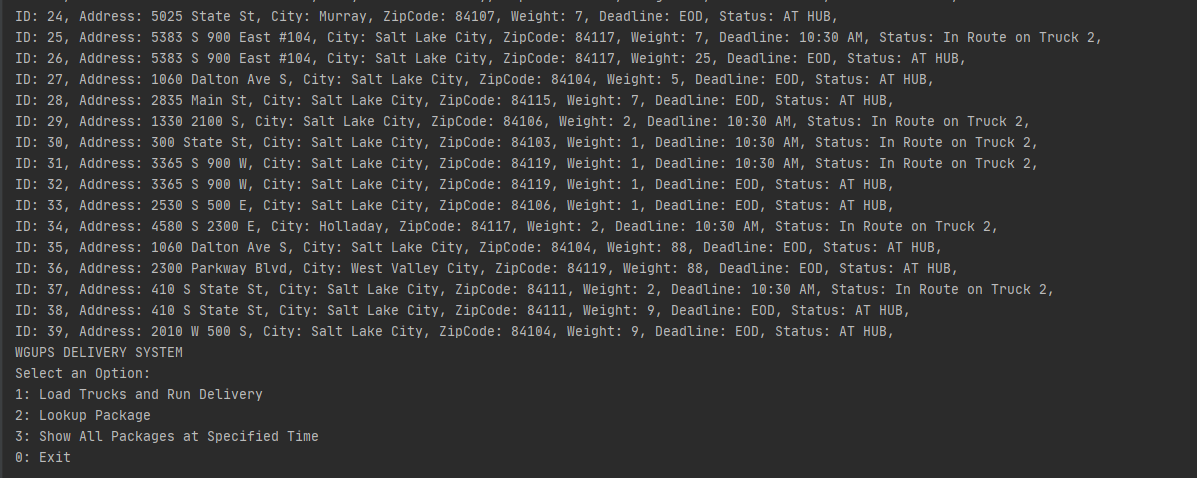
The main self-adjusting data structure used in this application is the graph that tracks all locations and the distances between those locations. If a location were to be added or removed the graph would adjust itself and update the edges going into and out of the vertex corresponding to that location. An adjustment to this data structure would not affect the running time of the application.

**EXPLANATION OF DATA STRUCTURE:**

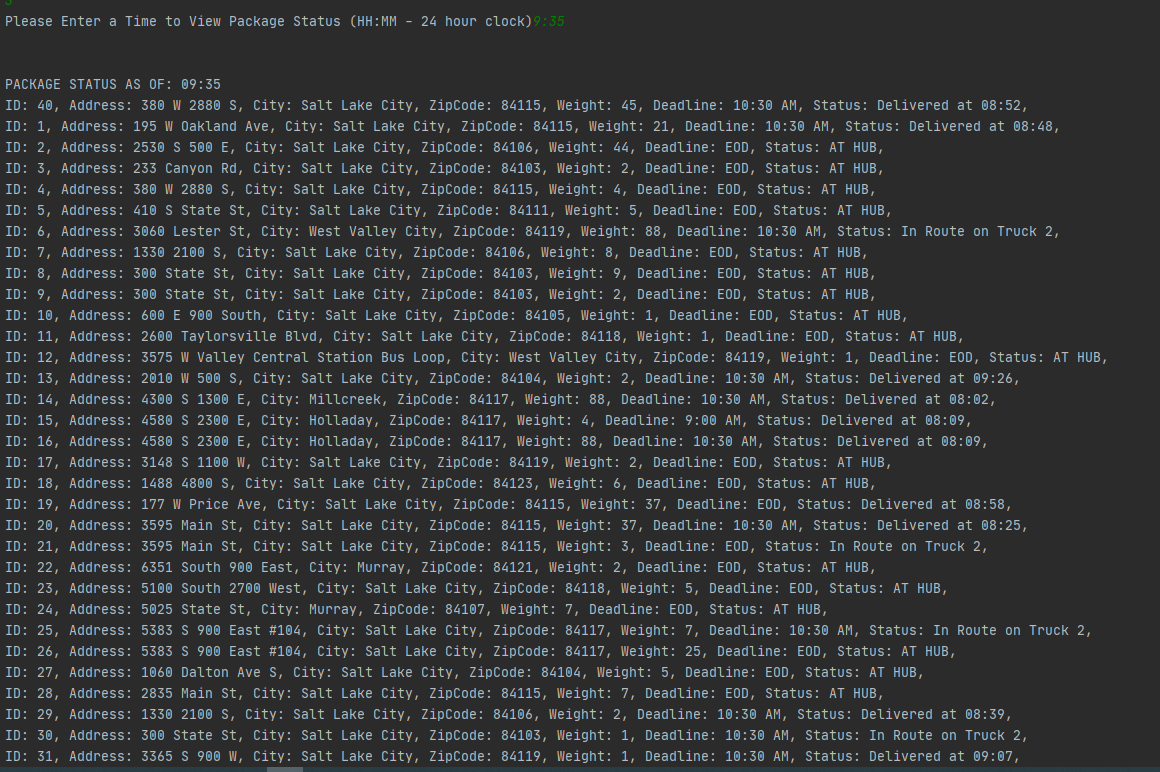
As mentioned above, this application uses a graph data structure to track locations for delivery as vertices of that graph and undirected weighted edges to track the distances between those vertices. Each address in the Location List is assigned a Location ID and a Vertex that corresponds with that Location ID. The graph is then populated with edges going from each Vertex to each Vertex adding an undirected weighted edge. The edge weights are found in the Distance Table with is imported into the application using a Matrix structure. This matrix can then be iterated over and the pair of Vertices corresponding to the indices of the matrix rows and columns can be assigned the corresponding weight value.

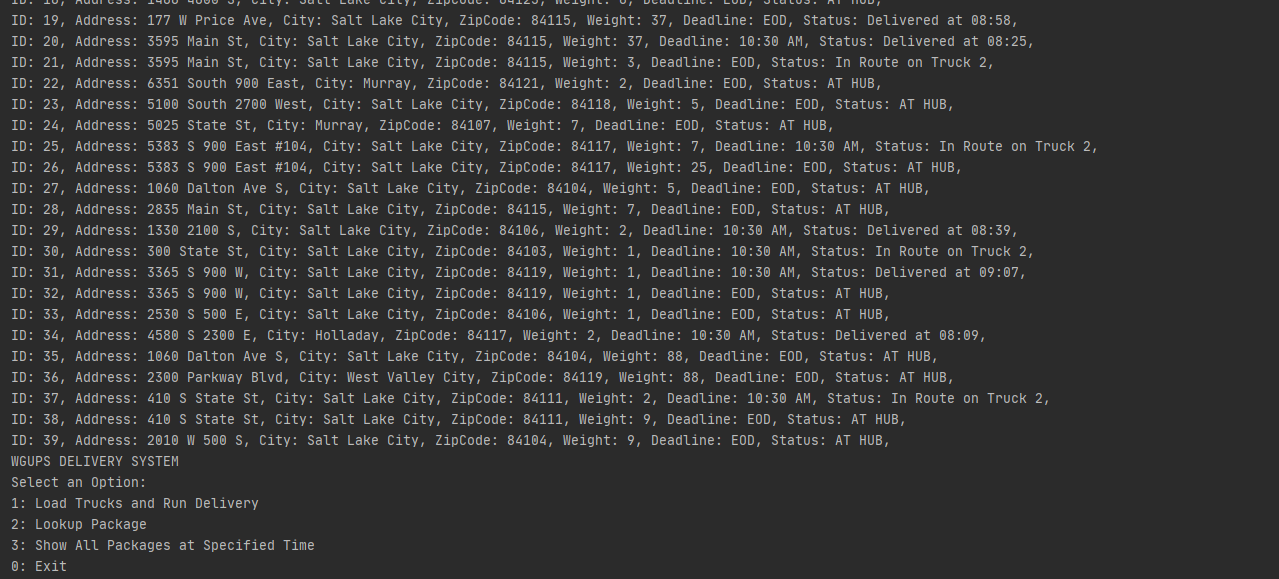
**FIRST STATUS CHECK:**

****

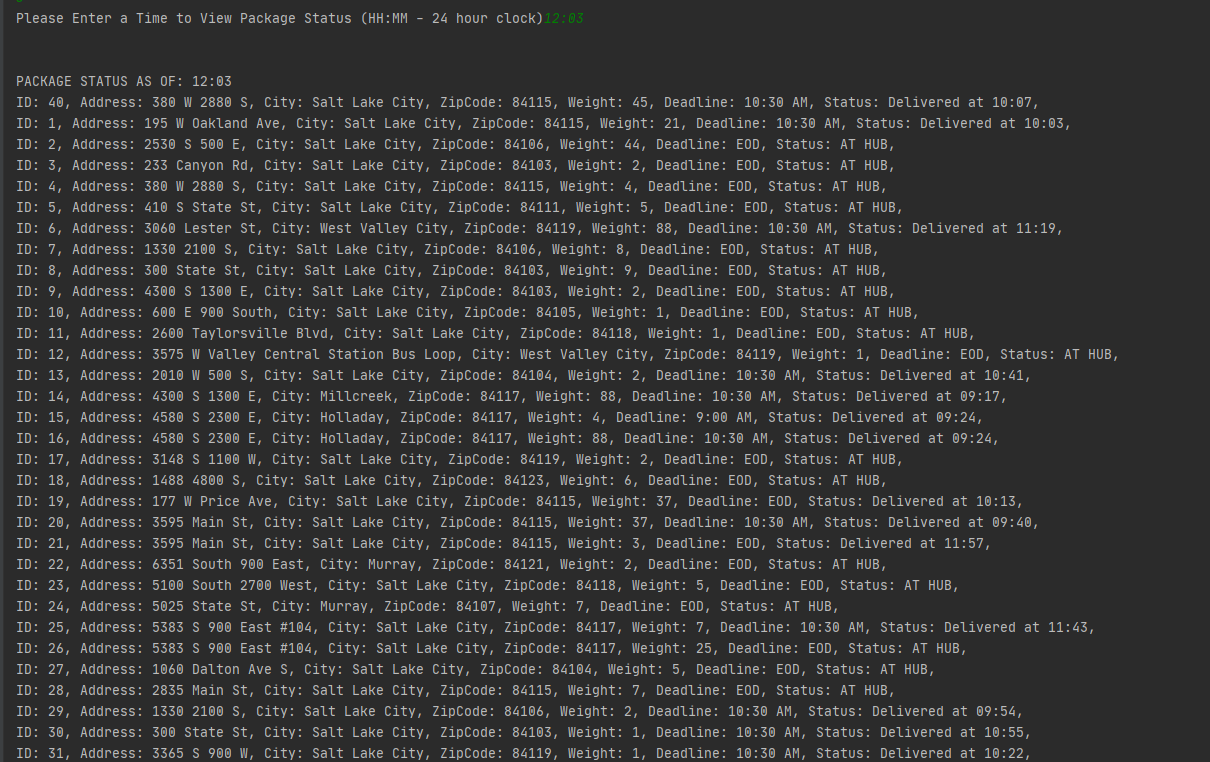
****

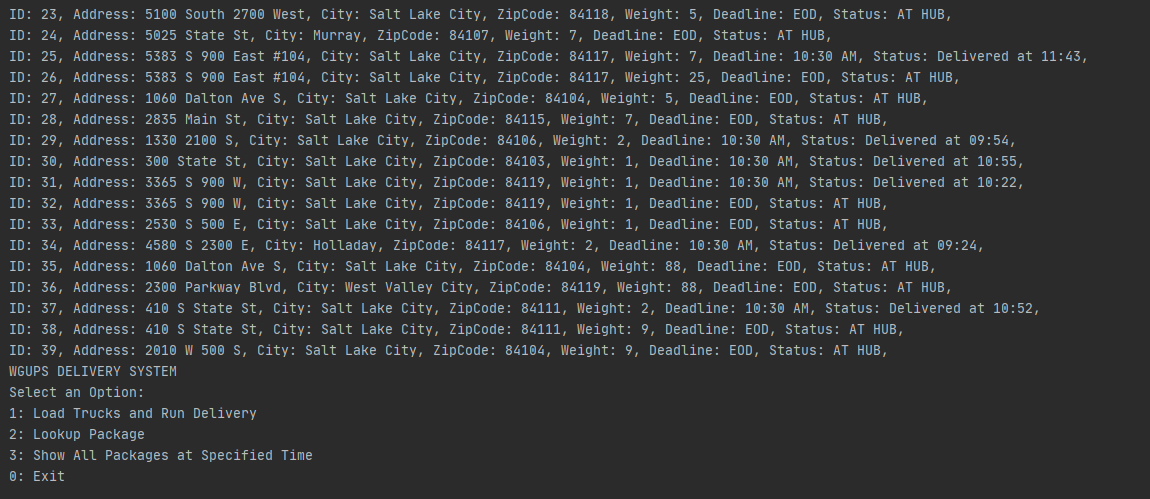
**SECOND STATUS CHECK:**

****

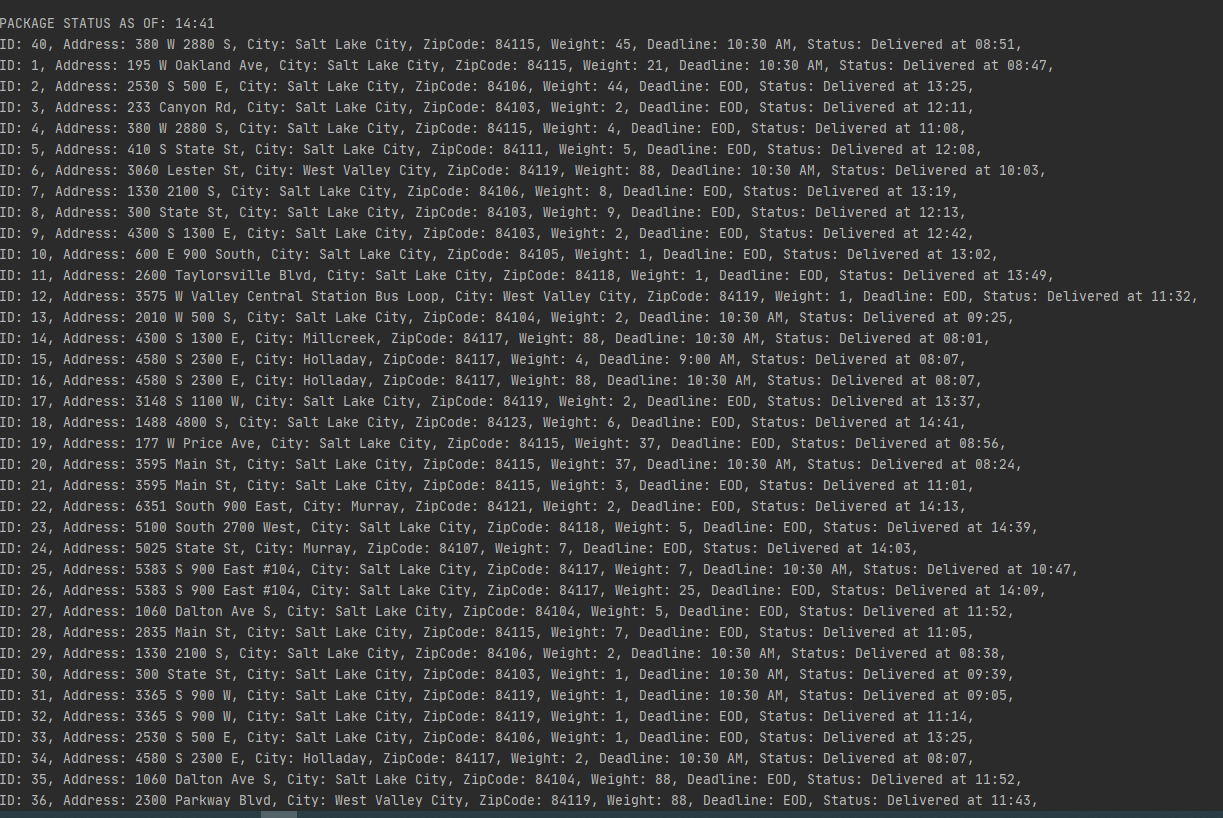
****

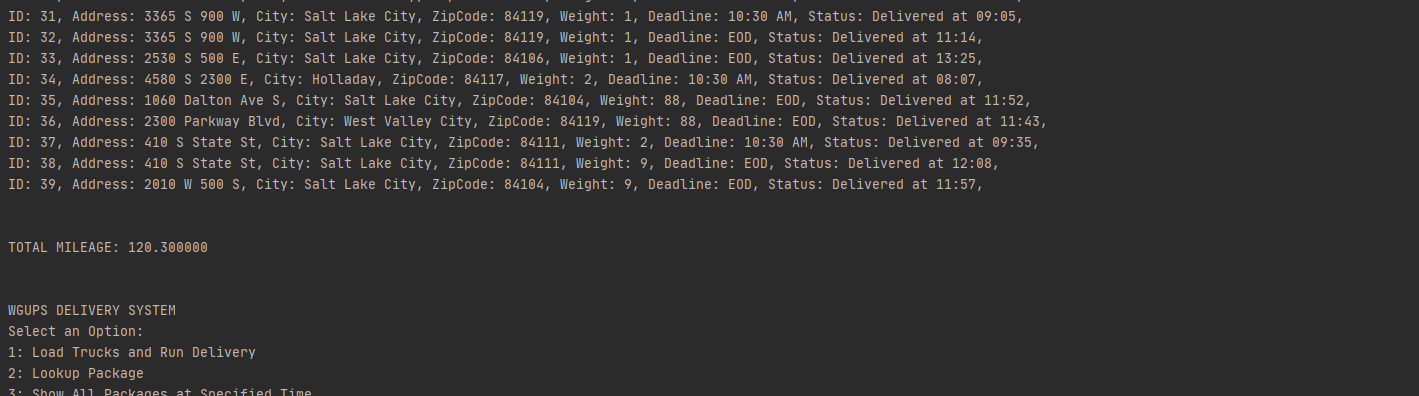
**THIRD STATUS CHECK:**

****

****

**SCREENSHOTS OF CODE EXECUTION:**

****

****

**STRENGTHS OF THE CHOSEN ALGORITHM:**

The Greedy Algorithm chosen here has many advantages. It is simple to implement. It is scalable. It is easily maintainable. The greedy algorithm is also efficient. Compared to other algorithms used to solve optimization problems it can solve them with less overhead.

**OTHER POSSIBLE ALGORITHMS AND ALGORITHM DIFFERENCES:**

Dijkstra’s algorithm takes a start node and an ending node and finds the shortest distance between those nodes. This algorithm has some challenges with this application as it is not designed to return to the starting node. This can be easily accounted for in a manual loading of the trucks.

Another approach would be to implement a heap approach. This would have greatly improved the sorting efficiency. The challenge would have been to tie the distances and their locations together in the heap.

**DIFFERENT APPROACH:**

If I were to do this project over again, I would go into it with a better plan. I think I would look at my algorithm choices and perhaps even explore some that were not presented in the text. There are many other choices that would have been more efficient, provided a better solution, but have been much much harder to implement.

**EFFICIENCY:**

The graph data structure used in this application is very efficient. The worst case time complexity would be O(N2) and that would only be needed if one were to desire a list of each node with the distance to each other node. The most common use of the graph as implemented is for reading the distance between two specific nodes. This can be done in O(1) time.

**OVERHEAD:**

As this application is hosted and run on a local machine the bandwidth aspect of application overhead is nonexistent. To link to the next data item in the data structure is done in at worst O(N) time. This would involve iterating over the list of all vertices connected to the current vertex. Memory overhead is also minimal. You would need to only know this current vertex and the list of all adjacent vertices.

**IMPLICATIONS:**

This solution is scalable. It will continue to function with small changes. New cities can be incorporated with the addition of new hubs. Package sorting is also scalable as the package sorting grows at a rate of O(N2). New classes could be explored for adding these new cities and hubs.

**OTHER DATA STRUCTURES AND DIFFERENCES:**

Other data structures that could have been used include nested dictionaries and a binary search tree. Nested dictionaries would have allowed for fast efficient access to the distance data. Know the address for your current location and the location to which you would travel next would provide this distance that needed to be traversed. Implementing this data structure instead of a graph would have allowed for the address to be used outright as opposed to having a list of addresses and relying on index numbers to find vertices in the graph. The BST would have provided a different solution. I could have sorted the packages into this tree based on certain attributes providing for quick and efficient access.