Core Algorithm Overview

**Stated Problem:**

The purpose of this project was to efficiently sort and deliver packages with Python using a specific algorithm of the student’s choice. This program represents the Western Governer’s University Parcel Service or WGUPS, 40 individual packages with constraints and the routes along the Salt Lake City delivery route. Three trucks and two drivers were issued, with the trucks being loaded instantly, and having no need for refueling or stopping at a destination.

**Algorithm Identification:**

For this project, I have chosen a Greedy algorithm. While it may not be the most efficient, it chooses the shortest destination from a given truck’s current location, and moves the truck/package to the next destination, and so on. This implementation uses recursion to move the truck and reduce the size of the truck’s list.

* Truck’s delivery list is passed into the function with the truck number and current location.
* The truck’s current location is compared to the distances and address values of each other package to find the shortest distance.
* The delivery and location distance are added to their respective lists.
* The shortest distance is set as the distance value, the list item that contained that value is removed, and the function is called again using recursion with the next item in the list.
* Once the function sees a list with zero items, it ends.

**B1: Logic Comments**

Method takes three values: truck\_list, truck\_number, current\_location

*(base case)* If length of truck\_list is 0 then return the empty truck\_list

set distance\_value to 30.0

*(to make sure the next distance is always less than this value)*

set new\_truck\_position to 0.0

*(sets the truck’s position after a package is delivered or leaving hub)*

for index in truck\_list:

if current\_distance of (current\_location, (int(index[9]) in truck\_list)) <= distance\_value

*(checks the length between the two values and determines if it is smaller than the provided value, index[9] is equal to the address’ distance value column in the distances.csv)*

set distance\_value to current\_distance(current\_location, int(index[9]))

*(sets distance value to be compared as the distance between the two values)*

set new\_truck\_position to int(index[9])

*(sets the new truck position to the second or end value)*

for index in truck\_list:

if current\_distance(current\_location, int(index[9]) is equal to distance\_value:

*(compares current distance value to the value in the truck\_list at a given index)*

If truck\_num is equal to 1, 2, or 3

*(finds truck number to add to that truck’s specific packages and locations lists)*

truck\_deliveries.append(index)

*(adds current truck’s delivery item to the list)*

truck\_location.append(index[9])

*(adds the delivery distance to the list)*

truck\_list.pop(truck\_list.index(index))

*(removed the item from the truck list)*

set current\_location to new\_truck\_position

*(sets the current location to the new position to pass back into the method)*

get\_route(truck\_list, truck\_num, current\_location)

*(recalls the method recursively with the new shortened truck list, truck number, and updated location of the truck)*

**B2: Development Environment**

**Software:** This program was developed in PyCharm 2021.1.1 Community Edition, using Python 3.8.

**Hardware:** This was developed on my personal desktop with a Ryzen 7 5800X at 4.67GHz, 32GB of 3200MHz CL16 RAM, and an NVMe SSD.

**B3: Big-O and Space-time**

**Main.py**

|  |  |  |
| --- | --- | --- |
| **Line Number** | **Space Complexity** | **Time Complexity** |
| **26** | **O(1)** | **O(N)** |
| **33** | **O(N)** | **O(N^2)** |
| **Total** | **O(N)** | **O(N^2)** |

**Deliveries.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| **23** | **-** | **O(1)** | **O(N)** |
| **28** | **-** | **O(1)** | **O(N)** |
| **33** | **-** | **O(1)** | **O(N)** |
| **41** | **-** | **O(N)** | **O(N^2)** |
| **47** | **-** | **O(N)** | **O(N^2)** |
| **53** | **-** | **O(N)** | **O(N^2)** |
| **74** | **-** | **O(N)** | **O(N)** |
| **92** | **-** | **O(N)** | **O(N)** |
| **110** | **-** | **O(N)** | **O(N)** |
| **131** | **get\_first\_truck\_total\_distance** | **O(1)** | **O(1)** |
| **137** | **get\_second\_truck\_total\_distance** | **O(1)** | **O(1)** |
| **143** | **get\_third\_truck\_total\_distance** | **O(1)** | **O(1)** |
| **149** | **get\_total\_distances** | **O(1)** | **O(1)** |
| **Total** |  | **O(N)** | **O(N^2)** |

**Hashtable.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| **3** | **\_\_init\_\_** | **O(1)** | **O(1)** |
| **10** | **\_\_init\_\_** | **O(1)** | **O(1)** |
| **17** | **key\_gen** | **O(1)** | **O(1)** |
| **22** | **insert** | **O(N)** | **O(N)** |
| **39** | **update** | **O(1)** | **O(N)** |
| **50** | **get\_key** | **O(N)** | **O(N)** |
| **60** | **delete** | **O(N)** | **O(N)** |
| **Total** |  | **O(N)** | **O(N)** |

**Packages.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| **17** | **-** | **O(N)** | **O(N)** |
| **98** | **get\_hashtable** | **O(1)** | **O(1)** |
| **103** | **get\_first\_truck\_deliveries** | **O(1)** | **O(1)** |
| **108** | **get\_second\_truck\_deliveries** | **O(1)** | **O(1)** |
| **113** | **get\_third\_truck\_deliveries** | **O(1)** | **O(1)** |
| **119** | **get\_packages** | **O(N)** | **O(N)** |
| **Total** |  | **O(N)** | **O(N)** |

**Distances.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| **19** | **current\_distance** | **O(1)** | **O(1)** |
| **28** | **get\_distance** | **O(1)** | **O(1)** |
| **38** | **get\_truck\_time** | **O(N)** | **O(N)** |
| **69** | **get\_route** | **O(N^2)** | **O(N^2)** |
| **106** | **get\_addresses** | **O(1)** | **O(1)** |
| **111** | **get\_first\_truck\_deliveries** | **O(1)** | **O(1)** |
| **116** | **get\_second\_truck\_deliveries** | **O(1)** | **O(1)** |
| **121** | **get\_third\_truck\_deliveries** | **O(1)** | **O(1)** |
| **126** | **get\_first\_truck\_locations** | **O(1)** | **O(1)** |
| **131** | **get\_second\_truck\_locations** | **O(1)** | **O(1)** |
| **136** | **get\_third\_truck\_locations** | **O(1)** | **O(1)** |
| **Total** |  | **O(N^2)** | **O(N^2)** |

**B4: Scalability and Adaptability**

Due to using a Greedy algorithm, this can be used to sort any number of packages (within reason) in a reasonable amount of time, and find the shortest distance to travel. The program can be scaled to include either more trucks, more packages, and more addresses. The easiest way would be to input more/new values into the CSV, and change some of the parameters to work with the new set of information. I would most likely also change the method of sorting to be more sophisticated or accept more overall generic terms, that way truck sorting can be completely automated.

**B5: Software Efficiency and Mantainability**

Overall, the software is relatively efficient with a worst-case space and time complexity of O(N^2). A couple of potential improvements would be to develop methods rather than using functions, to clean up the code and make it easier to maintain rather than splitting out each portion for each truck. For sorting packages again, it could be separated and implemented in a way that could only take certain constraints in order to sort the packages automatically.

**B6: Self-Adjusting Data Structures**

Since I used basically all lists and a chaining hash table in this program, searching for the lists and nested list was rather quick. However, if I had chosen to nest an additional list inside of a nested list, that would have significantly increased both the space and time requirements of the project. That’s where having either a Truck or Package class/object would be better, since those could be passed in and out of lists/methods without the need for nesting a list inside of a list. Another method I would have used to store the address or delivery data would have been a graph and mapped out each distance between the addresses or vertices to find a path that would be shortest.

**Data Structure and Explanation**

For this application, I used lists and nested lists in order to store and convert the information for each truck’s delivery and location data. Overall information was stored in a chaining hashtable for relative ease in accessing each element. In this certain use-case, since there were no collisons and I only used lists, accessing each element is much faster than if there were collisons.

For the lookup function, since I was using a hashtable and nested lists, finding the packages completed rather quickly. However, if a truck was able to carry many more packages or the distance was increased between the deliveries, the lookup function would remain relatively the same, but the processing would increase overall in order to print every package.

For the space used by the program, a nested list or hashtable is rather space and memory efficient. The issue would be if I decided to implement many nested lists, or nested lists in order to store and look up that information. However, in this certain scenario, the current lookup and space usage is fairly efficient based on a list being created by the distance and addresses CSV files that could be added to with relative ease.

**Strength of the Chosen Algorithm**

The main aspect of the greedy algorithm is that as soon as it finds a value that is lower than the one being passed into it, is chooses that option. In this case, that would be to choose the next distance along the path and delivering all packages within 140 miles. It also is able to reference the addresses and package data passed into it, and can be scaled upwards based on the values entered into the CSV file.

**Other Possible Algorithms**

Two other algorithms I could have used would have been either developing a Dynamic way, or by using Hueristics. In a Dynamic situation, I would have most likely stored address and distance data to determine if a path might be shorter by a different route than what was immediately found by the Greedy approach. In the Greedy algoritm, the distance and address data is called each time in order to set the function up for the next recursive call. If I would have chosen a Huerstic approach, the way of sorting the packages and the distances would have been somewhat similar, but I would most likely choose the best path per truck based on the specific packages that needed to be loaded due to constraints. However, with the Greedy approach, sometimes the shortest route is not chosen based on the function choosing what is the best choice given the options presented to it.

**J: Different Approach**

If I was given another opportunity to create this project again, or a similar project, I would most likely take the approach of making it more object-oriented rather than using a method of lists. I would most likely implement a truck class with contained private objects, as well as a package class with contained objects. An interesting method would be to read/convert all the package data before loading the trucks, calculate the distances between where the address or package is in relative distance to the hub as well as to each other, and lay out the best path for a certain truck containing packages with certain constraints (time, truck exclusivity, etc.). In that case, I would most likely either still use a list for distance data or create an “id” containing each data/distance point pulled from the CSV file and compare the package class with that. For the sake of space-time complexity, I believe storing the data for distances as well as the objects would be more taxing, but it may not be so much so that it wouldn’t be worth exploring.

**K2/A: Other Data Structures**

As I had touched on above, I would most likely try to implement a graph, which would be best suited for use with the distances, and “vertices” or lengths between each address/package. This might work best with the different method I had listed above for calculating out the shortest path per package per truck before a truck has been loaded.

Another data structure I could have implemented would be a binary search tree. It appears to be about as complex space-time wise to searching a list, and better than searching a nested list. For this method, I would most likely insert the packages after sorting them into a tree that could be accessed to load a truck, rather than using a list. I could possibly also use this for inserting distance or address data, and use that to compare the distances.