**Project Code Overview**

**Stated Problem**:

The goal of the project was to develop a program and algorithm that determines the best distribution of packages amongst delivery trucks and to optimize the delivery route. There are three trucks but only two drivers, so only two trucks can be out at a time. There are also some restrictions and special cases that come along with the 40 packages. For example, some packages need to be delivered by a certain time, and some packages were delivered to the warehouse late. First, the program determines the optimal way to load the trucks, starting with the first, based on their priorities and deadlines. Each truck is then assigned locations they must visit to deliver the packages. The program then takes these locations and uses a greedy algorithm to return a sub-optimized route that will lead to a low mileage. The reason I say sub-optimized, is because to find the absolute best path for the truck to take, it must check every possible path which has a complexity of O(N!). Our trucks have a capacity of 16 packages, which means at worst-case 16 different locations, or 16! (about 2.1 X 1013 different paths to compare).

**Program Overview:**

**The Program**

The program covers all the requirements given using the given assumptions. The program was created using Python and the data is loaded from CVS files. The CSV files were modified slightly in format to make loading the data more convenient. The location and package data are all loaded into a hash table created for this program. While the packages are being loaded from the CVS file and into a hash table, it is also being sorted through nested if and else statements. These statements are based on assumptions and special notes. Priority mail with deadlines and not delayed were sorted into truck one that leaves at 8 AM. Priority mail which was delayed till 9:05 AM where sorted to truck two. Packages that must be delivered in the same truck or had to be in truck two are also put in accordingly. Once the priority mail was sorted, if the trucks still had space, packages with similar locations were put in until full. All the remaining packages were put into truck three, which leaves at 10:20 AM after the package correction and truck one driver returns.

The program then takes the three truck locations and optimizes a route using a greedy algorithm, to give the least possible miles. This greedy algorithm will be discussed further in the following section. The next part of the program is to run the trucks and time stamp when the package was delivered. An interface gives the user the ability to check the status of packages at any given time. This was accomplished by taking the time of day and calculating the time the truck has traveled in that time frame. The distance it has covered in that time frame is calculated and then checks which locations it has visited in that distance according to the optimized route. Packages associated with that location are then time stamped.

**Program Big-O**

There are methods in the program that has an O(N2) time complexity including the greedy algorithm. For example, when loading the truck after the priority ones, it iterates through a list and then has a nested linear search to match locations. Another example is when running the simulation, if a package should be time-stamped it runs through a loop and with a linear search. However, O(N2) is the worst-case throughout the program. The program code has comments on complexities for further information.

**The Route Algorithm**

Forthis program, I choose to go with a self-adjusting greedy algorithm to optimize the truck routes. The reason for this was because the program requirements were not demanding, and it was easier to implement. The requirements only had 40 packages, 26 locations, and three trucks. In a worst-case scenario, a truck would have to travel to 16 different locations. Therefore, a greedy algorithm would suffice in this situation.

The greedy algorithm works by taking the starting location, which is the hub or point 0, and compares the distances to all the other locations that the truck needs to go to. The shortest distance is then picked as its next destination. The greedy algorithm then starts again with the starting location updated as the previously chosen location and compares it to the remaining locations. This is done until all the locations have been visited. The following is the pseudocode representing the algorithm to calculate the shortest route.

**Next Shortest Path Greedy Algorithm Pseudocode**

route\_optimize(truck\_locations)

route = empty list

current\_location = 0

while length(truck\_locations) != 0

# This is the greedy algorithm part

# Assumes the first location in list is the shortest

next\_location = truck\_locations[0]

next\_location\_distance = get distance from start to truck\_locations[0]

for i in truck\_locations

if (get distance from start to truck\_location[i]) is less than (next\_location\_distance)

next\_location = truck\_location[i]

next\_location\_distance = get distance from start to truck\_locations[i]

next\_location append to route list

next\_location removed from truck\_locations list

return route

**Greedy Algorithm Big-O**

As seen from the pseudocode, a for loop is nested inside a while loop, therefore the time complexity for my algorithm is O(N2). This is the worst-case and best-case since the algorithm checks each distance even if the shortest one is found at the start. The space complexity remains O(N) as only two lists are used to compare data (O(2N) = O(N)).

**Choosing Greedy Algorithm**

As mentioned before, the requirements of this project weren’t demanding and used pre-determined data, a greedy algorithm was the simplest process to meet the project constraints. Using this algorithm, the total traveled distance was 115 miles, and all packages were delivered before their deadlines. The best advantage is that it can find a suboptimal - but still adequate - path very fast. It also can adjust well with different location inputs. For example, after the initial testing, I changed the data structure in my loading method and re-ran the program. The greedy algorithm self-adjusted and produced an even better route optimization (125 miles to 115 miles). Therefore, I choose a greedy algorithm since it was easy to implement and gave a good route, even when the locations changed.

While the greedy algorithm gave good results and meet the requirements, it is probably not the best solution. It performs well but it does not give the best route with a time complexity of O(N2). One method with the same complexity that could have been used is the Dijkstra’s Shortest path algorithm. It is considered one of the most efficient shortest path algorithms while having a complexity of O(N2). It uses a graph data structure and weighted vertices to calculate the best path, while the greedy algorithm uses a simple list. Even though it is more efficient, it is more complex to implement and unnecessary.

Another algorithm that could have been used is a Dynamic method. These methods break down the problem into smaller subproblems and store information on the way. Using this method will help to keep track of the best routes and check if there is a faster way. This again is also more complex to implement and not as efficient as Dijkstra’s algorithm. In the end, the greedy algorithm, while not being the most efficient, works well for this program.

**Data Structure Overview:**

**Hash Table**

For this project I created a simple hash table instead of using one provided by python. It is used for the location data, package data, and truck data. It is initialized with a size input. Because the size of the data was predetermined, it was easier to implement a pre-determined size hash table. The hash table is created using a list and filled with nulls/none matching the size input. This makes a one-time cost of complexity O(N). Not making it self-adjusting helped to make the insert and search/get methods remain at O(1) complexity. Because the program uses the insert and get methods more frequently, having those methods O(1) was the preferable choice. Also, since there was no possible collision in the data, we did not have to provide collision resolution to help maintain the hash table simple and constant complexity. This provides easy access for my algorithm to the desired data. It was also easy to iterate through a hash table by just using a for loop.

The issue with my hash table is that it won’t scale very well with unknown sizes. Therefore, it probably will not scale well with changes in packages and locations, such as a real-world parcel service would have. However, this is adjustable, because the number of packages to be delivered at the beginning of the day is known. Lastly, since the hash table is used to hold pre-determined data and quick retrievals, it works well with the program and delivers all the requirements.

**Lists**

Most of my data structure involves lists. They are simple to implement and versatile. Appending has only a complexity of O(1). Since the program keeps adding to a list, this was very helpful in keeping overhead low. Both of the main algorithms rely heavily on lists. My route optimization greedy algorithm works very well with lists as most of their complexity is O(1) since it adds and removes data from lists quite frequently. Lists are also self-adjusting, so frequent use and unknown sizes of some data worked well with this data structure. Using lists for testing and debugging is also very useful, as they are easy to sort and print to console.

One issue with lists is the search function. Its worst and average case is O(N). This is one of the main reasons why my program has a complexity of O(N2). The program has several loops and searches combined. However, at the program's scale and the versatility list provides, this disadvantage is negligible.

**Alternative Data Structures**

Even though the list is simple and versatile, a better replacement would have been a Binary search tree or an AVL (self-adjusting balanced tree). A tree sorts and holds its data with a lower value on the left side of a node and a higher value on the right side of the node. This incorporates a binary search, instead of the linear search my program uses. This would have helped to reduce the search time complexity to O(log N). It would also help to prevent data repetitions since binary trees don’t allow any similar additions. The program sorts out packages and destinations to trucks. Having repeated values cause additional if statements and search in lists to maintain the program. Implementing a tree data structure would have made the process easier to maintain and scale. Also incorporating the distances into a tree would have helped the greedy algorithm reduce its complexity and give faster results.

The program could also have used graphs to incorporate location data, helping to implement a better route optimization, and even with sorting packages by where they are to be delivered. This could have helped to reduce the number of times different trucks had to visit the same location. However, graphs have a higher spatial complexity and again more complex to implement.

**Conclusion:**

**Ability to Adapt and Scalability**

This program has some setbacks that were neglected due to the requirements. Since the requirements weren’t intensive, the program was built around low data sizes and predetermined variables. For example, the hash tables are not self-adjusting, so the change in data sizes had to be hardcoded, allow an interface to change the size before loading the data or change the hash table structure. The loading of the trucks is largely manual and determined based on the given data. To change and optimize the method would require more data or test scenarios.

The greedy algorithm works fine and is very scalable. Since its self-adjusting to optimize routes, it works well with any number of locations and changes in trucks. It doesn’t provide the best solution but works well and will be good for a small company to use.

Therefore, my program isn’t very scalable regarding input data or any change to the company’s structure and requirements. However, these changes aren’t major and can be easily changed to fit a growing company’s needs.

**Efficiency and Maintainability**

I think the program is very efficient for its requirements and delivers all the mail well before deadlines. Even though the complexity is O(N2) the program is still efficient. Brute force for optimization will take forever and using just a first come first serve algorithm leads to about 225 miles, compared to 115 miles. Even though it has issues with scalability, these are minor and easy to change.

For maintainability, I used object-orientated programming principles and kept each aspect of the program separate. The sorting of packages, route optimization, interface, and simulation were built and kept separate for later changes. Commenting was done to explain the program's process and flow for debugging and changes.

**Future Approach**

First, I would try to optimize my sort and loading method. As mentioned, before it seems very manual and based on the given data. I would change it to optimize it for any package data. This would require more scenarios and test data. Also, I would consider using a Relational Database in the best practice. It would help maintain data integrity and will make the program highly scalable.

Even though I am happy with the greedy algorithms used in this program, it is not the best. Using Dijkstra’s algorithm would have provided a better result and give the company a competitive edge. However, incorporating this will require data structure changes. Also, I would implement a binary tree to help the sort and search.

In conclusion, I was very happy with the programs results and delivery outcomes. It meets all the requirements at 115miles.

**Sources:**

Hartley, J. (2017, June 5). Time Complexity. Retrieved May 10, 2020, from <https://wiki.python.org/moin/TimeComplexity>

Learn.zybooks.com. (n.d.). *zyBooks*

https://learn.zybooks.com/zybook/WGUC950AY20182019/