Core Algorithm Overview

Stated Problem:

The purpose of this project is to compute the most optimal and efficient route for the WGUPS to use in order to deliver packages, using Python as the programming language. There are two trucks and three drivers to be used to deliver a total of 40 packages. The total combined mileage between the trucks must be under 140. Many of the packages have specific constraints that must be accounted for when planning the most optimal route to take. These constraints are used to determine which truck the packages should be loaded on to. Once the package's truck number is determined, a greedy algorithm is used to sort the packages in order.

Algorithm Overview:

The Greedy Algorithm is utilized in the following way:

1. A truck(list of packages) and a current location of the truck is passed in. The current location of the truck is at the hub always because the greedy algorithm is called before the truck departs. The other two parameters passed in, which have no effect on the algorithm, are a list for the newly sorted packages and a list that contains those packages indexes.
2. Since the current location is at the hub, all of the destinations of the packages are looked at to determine which ones are closest to the hub.
3. The package whose address is closest to the hub is removed from the truck list, signaling the truck driving to that address.
4. The value package removed is added to the sorted truck list and then the algorithm is recursively called again on the original truck list.
5. This keeps repeating until all packages in the truck list are added to the sorted truck list, therefore placing them in optimal order.

The worst case runtime for this greedy algorithm is O(n^2). The best case is O(1) but is very unlikely to occur unless the truck(package list) being passed in contains no packages. Therefore, if using this algorithm to optimize the ordering of packages on a truck, the worst case is almost always going to occur.

Greedy Algorithm Pseudocode:

Parameters passed in:

truck: list(Packages), **current_location**: int, **sorted_truck**: list(None), **sorted_truck_idx**: list([0])

```
O(1)
If there are no packages in the truck:
    Return the truck (empty list)
If the truck is not empty:
    O(1)
    Set the next_location the truck will be going to 0
    Set the shortest_distance which is the truck can go to next to 50
    O(n)
    For package in truck:
        Set d = to the packages location
        If current distance of d and current_location is <= shortest_distance:
            Shortest_distance = current distance of d and current_location
            Set next_location = d
    O(n^2)
    For package in truck:
        Set d equal to the packages location
        If current distance of d and current_location is == shortest_distance:
            sorted_truck.append(package)
            sorted_truck_idx.append(d)
            truck.pop(truck.index(package))
            Current_location = next_location
            optimized_route(truck, current_location, sorted_truck, sorted_truck_idx)
            **recursively call the greedy algorithm on the updated set of parameters)

Return sorted_truck, sorted_truck_idx
```

Overall time complexity = O(n^2)

Efficiency of functions throughout Program:

Function	**Time**	**Space**	**File**
set_location	O(n^2)	O(n^2)	load_trucks.py
computer_truck_distance	O(n)	O(n)	load_trucks.py
No function	O(n)	O(n)	load_trucks.py
get_total_distance	O(1)	O(1)	load_trucks.py
__init__	O(n)	O(n)	hashtable.py
insert	O(n)	O(n)	hashtable.py
remove	O(n)	O(n)	hashtable.py
search	O(1)	O(1)	hashtable.py
get_distance	O(1)	O(1)	distances.py
get_current_distance	O(1)	O(1)	distances.py
get_address	O(n)	O(n)	distances.py
get_time_left	O(n)	O(n)	distances.py
No function	O(n^2)	O(n^2)	main.py
get_all_packages	O(n)	O(n)	options.py
get_single_package	O(n)	O(n)	options.py
__str__	O(1)	O(1)	Package.py

The overall runtime of this program is going to be O(n^2) because that is the largest runtime present throughout all functions.

Algorithm Justification:

The greedy algorithm is so strong because it is able to determine the shortest path between the packages and the current location. It will do this recursively over all of the objects while updating the current location to then sort each package in the most optimal order. It is also able to scale and still work optimal with even larger sets of data. The size of the truck's packages do not affect the runtime as they grow. This algorithm meets the project's requirements because it can deliver all of the packages in less than 140 miles.

Alternative Algorithms:
A-star search algorithm and Dijkstra's Algorithm are two algorithms that could've been used instead to meet the project requirements. Both of these algorithms work effectively with graphs so if a graph was used as the data structure in this project then these algorithms would be able to find the shortest distance between two vertices (represented addresses). This differs from the algorithm I used because I used the greedy algorithm to traverse across Lists.

Programming Environment:

The programming I used to create this application was writing the Python code inside of VSCode IDE. The version of python used was 3.10.6 and was developed using a python virtual environment to contain all project dependencies. External libraries used within this project include csv, and datetime. Hardware used was a Macbook Air M1 with an M1 processor, 8gm of Ram, and 256gb of internal storage.

Scalability:

All functions and algorithms within this project are able to be applied to an infinite amount of trucks and packages. Changes would need to be made in load_trucks.py as well as the Package.py files if a user wanted to associate more information to the packages. The packages were also loaded in manually with conditional statements, which was possible due to the requirements specifying specific restraints associated with certain packages. A more optimal way to load the trucks with packages should be used to prevent loading issues as the project scales. The greedy algorithm would not have any issues though handling more trucks/packages to sort. One last potential issue that may arise while scaling, which could easily be mitigated by minor code changes, is the way in which certain functions were placed in external files. The structure of this project works well in a small scale environment, but for larger projects I think it would be advantageous to organize the code in a more structured manner.

Efficiency:

The software developed for this project is efficient due to the requirements given within this project. For example, we are told that each truck can only contain at most 16 packages. This is very important because even though the greedy algorithm has a complexity of O(n^2), it isn't very resource intensive when n<=16. If the truck was holding hundreds/thousands of packages then that may be a different story, but in this instance, that isn't the case. It is easy to maintain due to how important functions were handled. If an error arose in the future, it would be very easy to pinpoint its origin due to the compartmentalization of the project's functions.

Hash Table Strengths:

There are many strengths of the data structure used (Hash Table). The fact that all operations such as insertion, deletion, and searching are all O(1) or O(n) is very powerful in maintaining the efficiency of the program. All of these operations were essential in the development of this program, and if any of them were even for example O(n^2), this would've caused a dramatic decrease in program efficiency. The hash table is justified to use for the project because the search function is so efficient and due to the nature of packages needing to be searched for many times, O(1) allows for an efficient program. The lookup time is always going to be O(1) so regardless of how many packages need to be delivered, the lookup functions run-time will never change. No parameters change the run-time of the lookup function. Again as previously stated in the last few sentences, NO parameters whatsoever, including cities and trucks will affect the lookup time or space efficiency! The amount of indices in the hashmap should grow though to avoid collisions with an increasing amount of input. The hash table is used to store the packages from the truck, it's great because we can look up the packages by their IDs. This is the relationship between the project's data points and the hash table data structure.

Hash Table Weaknesses:

The weakness of this data structure could have been derived from collisions during insertion, thankfully a chaining hashmap was implemented which mitigates collision problems.

Alternative Data Structures:

I could have implemented a variety of different data structures if need be but two that stand out to me are Graphs and BST's. Graphs would have allowed me to store the packages closely together with adjacent vertices. This would mean that packages with similarities were grouped together leading to an easier time doing computation in the program. A BST would allow the packages to be sorted by any of their attributes such as address, weight, or delivery time. This could help lead to an easier time sorting them for the most efficient route. Graphs and BSTs differ from a hashtable because they are non-linear data structures whereas hash tables are linear depending on the implementation. Hash Tables take a key to search for elements whereas Graphs and BSTs need to be traversed.

Reference List:

No external sources were used in the creation of the core algorithm overview. All concepts and their explanations were explained based on my knowledge of the material.

In my code, I referenced a chaining hashtable from the zybooks which was properly sited within the python file.

https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/7/section/8