Ryan Hildebrant
C950: Data Structures & Algorithms II Student # 000709587

Core Algorithm Overview

Stated Problem:

The purpose of this project is to determine the best route and delivery distribution for the Western Governors University Parcel Service (WGUPS) using a common high-level programming language (Python 3.7). There are 40 packages that must be split up across two trucks with 3 different drivers. Some packages have delivery constraints, and some have been delayed. Additionally, there is a package that initially has the wrong address and packages that must go out on the second truck. To solve this problem, we must first use a series of conditional statements to load the trucks based on the data that is provided. We then must implement a greedy algorithm to optimize delivery of each package along the truck route. This algorithm is greedy because it determines the shortest available path from its current location then continues to do this until no additional packages remain. This article will analyze the use of this algorithm and provide a descriptive overview of the applications methods and components.

Algorithm Overview:

The greedy algorithm is executed by doing the following…

1.) A list of packages on the truck is passed in along with the associated truck number and the current location (always at hub by default)
2.) The current location is compared to the locations of all the packages in the truck to determine the closest location.
3.) The lowest value is determined once all objects in the truck have been compared then that value is removed from the truck list and the truck moves to that address location
4.) That value is appended to an optimized location list than the algorithm updates current location with the new address and calls the function again with the smaller truck list.

The space-time complexity of this self-adjusting greedy algorithm has a worst case runtime O(N^2) and a best case runtime of O(1). The worst case is almost always guaranteed as the best case is only possible when the list of packages being loaded onto a truck is empty. A pseudocode representation is provided below to prove this…

```
Greedy Algorithm

A. Take the following parameters
        1. truck_list (represents a nested list of packages on a given truck)
        2. truck_number (represents the truck you are working with)
        3. current_location (recursive variable that is used to show where the truck is)
B. Create the base case to break the recursion in the algorithm (see to see recursive step)

        if length of truck_list is 0 then return the empty list
```

C. Enter the recursive sections if the truck_list is not empty

```
set lowest_value to 50.0
(represents the closest deliverable location from the current location in miles, ex: 3.2)
set new_location to 0
(represents where the truck is moving next to deliver a package)
if length of truck_list is not 0:
    for index in truck_list:
        if check_current_distance of current_location and the next index in the truck
        package list <= lowest_value:
        (Searches all reachable locations and updates the lowest_value if a smaller mileage
        path is found).
            new lowest_value exists then update lowest_value and current_path
```

The time complexity of the algorithm so far is as follows:

A.) O(1) or constant time B.) O(1) or constant time C.) O(N)

O(1) + O(1) + O(N) = O(N)

D. Enter the second for loop in the recursive section of the algorithm

```
for index in truck_list:
    if check_current_distance of current_location and the next index in the truck package list is
    equal to lowest_value:
    (search through the list of packages again to find the package address value that is equal to
    lowest_value (see section c). This allows us to deliver multiple packages to the same address
    if they are in the same truck)
        if truck_number is equal to 1, 2, or 3
        (determines which truck object was originally passed in so it can create an
        optimized list for that truck)
            optimized_truck.append(current package)
            (Append the selected package from truck_list to the optimized_truck list)
            optimized_truck_index.append(current package index)
            (Append the selected package address index from truck_list to the
            optimized_truck index list)
            pop_value = truck_list.index(current package)
            truck_list.pop(pop_value)
            (Remove the selected package from truck_list)
```

current_location = new_location

(Update the current location to show the truck has moved to a new location)

calculate_shortest_distance(truck_list, truck_number, current_location)

calls algorithm function again to loop through the shorter truck_list with an updated location

The total time complexity of the algorithm is as follows:

A.) O(1) or constant time B.) O(1) or constant time C.) O(N) D.) O(N^2)

O(1) + O(1) + O(N) + O(N^2) = O(N^2)

Below is a table breakdown of the worst-case space-time complexity of each file in the Python application:

HashTable.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
__init__	12	O(1)	O(1)
_get_hash	20	O(1)	O(1)
insert	26	O(N)	O(N)
update	42	O(1)	O(N)
get	55	O(N)	O(N)
delete	64	O(N)	O(N)
Total		3N + 3 = O(N)	4N + 2 = O(N)

ReadCSV.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
None	15	O(N)	O(N)
get_hash_map	60	O(1)	O(1)
check_first_truck_trip	65	O(1)	O(1)
check_second_truck_trip	70	O(1)	O(1)
check_third_truck_trip	75	O(1)	O(1)
Total		N + 4 = O(N)	N + 4 = O(N)

Packages.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
None	43	O(1)	O(N)
None	51	O(N^2)	O(N^2)
None	66	O(N)	O(N)
None	80	O(1)	O(N)
None	88	O(N^2)	O(N^2)
None	102	O(N)	O(N)

None	117	O(1)	O(N)
None	125	O(N^2)	O(N^2)
None	139	O(N)	O(N)
total_distance	153	O(1)	O(1)
Total		3N^2 + 3N + 3 = O(N^2)	3N^2 + 6N + 1 = O(N^2)

Main.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
None	15	O(1)	O(N)
None	24	O(N)	O(N^2)
Total		N + 1 = O(N)	2N = O(N^2)

Distances.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
check_distance	17	O(1)	O(1)
check_current_distance	27	O(1)	O(1)
check_time_first_truck	41	O(N)	O(N)
check_time_second_truck	53	O(N)	O(N)
check_time_third_truck	65	O(N)	O(N)
check_address	79	O(1)	O(1)
calculate_shortest_distance	110	O(N^2)	O(N^2)
first_optimized_truck_index	150	O(1)	O(1)
first_optimized_truck_list	155	O(1)	O(1)
second_optimized_truck_index	160	O(1)	O(1)
second_optimized_truck_list	164	O(1)	O(1)
third_optimized_truck_index	170	O(1)	O(1)
third_optimized_truck_list	174	O(1)	O(1)
Total		N^2 + 3N + 9 = O(N^2)	N^2 + 3N + 9 = O(N^2)

Memory and computational time remain nearly linear throughout the entire application. This allows the available set of inputs to scale without being overburdened by memory availability constraints. Bandwidth is not a factor in the current implementation as the application is run and managed on a local machine that does not require network resources.

Advantages of chosen Algorithm

This greedy algorithm preforms all the required functions to meet the project constraints and delivers the packages within the range of 100 miles. The biggest strength of this algorithm is its ability to quickly find the optimal path for a given truck (defined above). Another huge advantage of the greedy algorithm approach is that it can scale with any set of data and addresses provided to it. Additionally, the user interface provides the ability for a user to see all package details and their delivery status at a given time. It can also search for an individual package based on its package ID.

Another algorithmic approach I could have used to optimize the packages was a dynamic programming approach. “Dynamic programming is a problem solving technique that splits a problem into smaller subproblems” (Zybooks, 3.5). I use an exhaustive approach instead of trying to break the program down into smaller functions. The advantage of using a dynamic approach is that I can store paths along a route and check if there is a faster way to get to a location by first traveling to another location. Storing these different paths may have created a large space complexity but it could have yielded a shorter path. A second algorithm I could have implemented for the optimal route is a self-adjusting heuristic. This approach would start at the hub then determine the closest path to the hub. It would then determine all packages that need to be delivered to that truck and load them into the truck. From there it would start at the new location and determine the shortest path from that location. If a location was already visited by that truck than it would move on the to the next closest location until all 40 packages had been allocated to a truck and a path was set. My approach and the self-adjusting heuristic share a similar attribute in the sense that the shortest available path is always chosen.

Programming Models:

The programming model for this application is limited as it is currently hosted on a local machine. The application is written using Python 3.7 and is executed in the Pycharm IDE. There is no communication protocol present as the application pulls data from CSV files that are in the project folder on the local machine. To do this we use the csvreader function from the csv library in Python. Thus, data exchanges are limited to the interactions of the application and the local machine. Additionally, there is no target host environment for the application as a network connection is not needed or established. In terms of information semantics, the current software requirements are organized in a functional manner. Defining a set of interaction semantics to determine the flow of connection, data, and the resulting information would not be required with the application being hosted on a local machine.

Ability to Adapt

The core functions of the application are designed to be able to scale and addresses changes in the number of packages, the number of trucks, and the number of locations. Minor changes are required to scale with any of these components. For example, another set of location csv files can be inputted into the application to calculate a new route and determine an optimal path. Additional packages can be inserted and the program will determine where to place the packages. This approach also allows a great deal of control when implementing numerous sub-applications as the design allows the input set to change freely. A potential problem with future scaling in this application is the approach I take to load the packages into the truck. It is currently done manually to meet all the package constraints and deadlines. Given the opportunity to improve this project, I would instead work on developing a heuristic approach to determining what packages go onto which truck. Having this process automated would benefit both the ability for the application to adapt to new environments and potentially provide a better path set for the greedy algorithm to work with. However, the core functions of the application have great potential to scale with changing markets as the greedy algorithm handles limited data sets very quickly.

Efficiency and Maintainability

Overall the software is very efficient, with two comparisons that have a time efficiency of O(N^2). While this may not be the best time complexity, it scales well with the 16-package limit per truck. It is also very maintainable as much of the software is the same core functions that have been modified for use case. Debugging is easier because you can always refer to another instance of the function to determine where potential errors might be.

Data Structures

The primary data structure I implemented throughout the application is a list of lists. I choose this data structure because it is flexible and easy to work with. It also works well with a hash table and data retrieval. Additionally, the project required to be able to have quick access to many elements of a package. Using a list allowed me to create a hash table with chaining resulting in very fast lookup, insertion, deletion, and access to specific data elements. In fact, most operations involving the data structure had a constant time complexity of O(1). The biggest advantage for me was it allowed me to verify the correctness of my trucks and packages along the route with very little additional overhead. One weakness I identified while working through this project using a list of lists is that it is hard to utilize the lists in an object-oriented way. For example, I had to apply the same operations to each list in order to delivery the packages for a given truck. Having a package factory class and a truck object would have simplified the coding and made it easier to manage scaling.

Looking back, I could have used many different data structures to fit the requirements of this project. One popular data structure would be a binary search tree. The BST would be different in the sense that I can presort the packages based on an attribute and quickly access them through a tree. Another data structure I could have implemented was a graph. A huge advantage of using a graph is that I could have grouped similar packages together as adjacent vertices. Then I could have traversed the graphs until a maximum traversal length of 16 was reached. This would also be a good method for future scaling.

Sources and In-Text Citations

The primary resource I used when modeling this overview was the reading material provided by Zybooks. The quote in the section, **Advantages of chosen Algorithm** is cited below:

Learn.zybooks.com. (n.d.). *zyBooks*. [online] Available at: https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/3/section/5 [Accessed 10 Nov. 2019].