C950 – Data Structures and Algorithms II

John R. McKahan II – Student ID:

Stated Problem: WGUPS(Western Governors University Parcel Service) requires a truck routing software solution, coded in Python, to assist with the company's Daily Local Deliveries (DLD) As of writing, packages are not being consistently delivered by their deadlines.

Provided Information: WGUPS has provided the following information:

- Truck capacity: sixteen (16) packages
- Trucks available for use: three (3)
- Average truck speed: Eighteen MPH(Miles per hour)
- Trucks are assumed to have an infinite amount of gasoline
- Delivery drivers start their route at 8:00AM
- Delivery day ends when all 40 packages have been delivered.
- Delivery time is factored into to the average truck speed.
- Each package may have up to one note attached
- Package #9 has an incorrect address, and will be corrected at 10:20AM
- Each package ID is unique
- A distance .XLSX file which contains mileage information between delivery locations • A package .XLSX file which contains information on the packages to be delivered
- A map of downtown Salt Lake City with delivery locations marked.

The following information pertains to the proposed truck routing software solution for WGUPS, tailored to the needs of WGUPS with scalability in mind should the needs become greater.

Requirement B1: Logic Comments

Overview of Route Algorithm:

The proposed routing algorithm takes a “greedy” approach to the routing problem and uses the following steps to determine the best route for the trucks to take.

1. First, the truck’s list of locations is input within the algorithm
2. The mileage between the hub location, and every single location within the list is calculated.
3. The location with the lowest mileage becomes the new hub value
4. The location with the lowest mileage is removed from the list of input locations to visit.
5. The location with the lowest mileage is added to the routing list.
6. The algorithm continues to loop until the original list is empty, and therefore all routes have been calculated and added to the new routing list.

Pseudocode of Route Algorithm:

```
Greedyroute(ListOfLocations)

1.  Input and initialize
        ListOfLocations is taken as an input, representing a list of locations to visit Hub
        is a variable assigned to be zero at first, or symbolically, the starting location
        BestRoute is an empty list, which will eventually hold the best route locations in
        order.

2.  Check length of ListOfLocations
    The algorithm will continue to loop until the ListOfLocations is equivalent to zero,
    and empty.

3.  Assign a large value to “TheLowestChoice”
    The lowest choice is assigned a large value for the first loop, to guarantee that the
    first location will result in the latter requirement returning true.

4.  For every locationID within ListOfLocations
            Ensure Hub location does not equal LocationID
                    Calculate mileage between Hub location and locationID
                    If calculated mileage is less than the lowest choice
                            TheLowestChoice becomes equivalent to the calculated mileage
                            LowestId becomes equivalent to the locationID
```

```
5. If locationID is not within BestRoute

        Hub = LowestID, making the new starting location, the previous
        step's ending location.
        Remove LowestID from ListOfLocations
        Add LowestID to BestRoute

6. Return the BestRoute
```

The proposed algorithm has proven to be valuable as routes determined by this algorithm meet all requirements defined by WGUPS, additionally, screenshots of functionality are provided within the project folder, which contain overall mileage within the range required by WGUPS.

A full overview of the Python code used in this proposed solution is available, defined on line 79 within the "DistanceHelper.py" file, contained within the attached C950 folder.

The Space/Time Complexity of this "Greedy" algorithm is O(n^2) in a worst-case scenario. This is a polynomial run time, and is considered efficient as per the requirements of WGUPS defined in

Space/Time Complexity breakdowns for files within the proposed solution are provided in the following section.

B3: Space-Time and Big-O

CSVHelper.py

Method/Function	**Time Complexity**	**Space Complexity**	**Line**
(Main File)	**O(N)**	**O(N)**	**14**
FixTypo()	**O(1)**	**O(1)**	**51**
FixWrongAddress()	**O(1)**	**O(1)**	**58**
PrintTheTable()	**O(1)**	**O(1)**	**64**
ResetTheValues()	**O(1)**	**O(1)**	**71**
ResetWrongAddress()	**O(1)**	**O(1)**	**76**
TOTAL	**O(N)**	**O(N)**	

DistanceHelper.py

Method/Function	**Time Complexity**	**Space Complexity**	**Line**
(Main)	**O(N)**	**O(N)**	**20,26**
Get_DeliverID	**O(1)**	**O(1)**	**40**
Get_Mileage	**O(1)**	**O(1)**	**52**
Get_DeliverIDNow	**O(1)**	**O(1)**	**60**

GreedyRoute	**O(N^2)**	**O(N^2)**	**79**
TOTAL	**O(N^2)**	**O(N^2)**	

Hashfile.py

Method/Function	**Time Complexity**	**Space Complexity**	**Line**
__init__	**O(1)**	**O(1)**	**6**
Get_hash	**O(1)**	**O(1)**	**12**
Add_value	**O(N)**	**O(1)**	**16**
Get_value	**O(1)**	**O(1)**	**31**
Delete_Value	**O(1)**	**O(1)**	**42**
TOTAL	**O(N)**	**O(1)**	

Main.py

Method/Function	**Time Complexity**	**Space Complexity**	**Line**
Calculatetotalmileageoftruckroute	**O(1)**	**O(1)**	**11**
Get_distances	**O(1)**	**O(1)**	**29**
Mileagecalculator	**O(1)**	**O(1)**	**56**
Simulations	**O(N^3)**	**O(N^3)**	**67**
DelayedSimulation	**O(N^3)**	**O(N^3)**	**236**
Simulationsforuser	**O(N^3)**	**O(N^3)**	**285**
Simulatetheentireday	**O(N^3)**	**O(N^3)**	**444**
Simulatetheentiredaygiventime	**O(N^3)**	**O(N^3)**	**459**
TOTAL	**O(N^3)**	**O(N^3)**	

Package.py

Method/Function	**Time Complexity**	**Space Complexity**	**Line**
__init__	**O(1)**	**O(1)**	**15**
Gen_package	**O(1)**	**O(1)**	**27**
Set_id/Get_ID	**O(1)**	**O(1)**	**45,49**
Set_Address/Get_Address	**O(1)**	**O(1)**	**53,57**
Get_city/Set_City	**O(1)**	**O(1)**	**61,65**
Get_state/Set_state	**O(1)**	**O(1)**	**69,73**
Set_Zip/Get_Zip	**O(1)**	**O(1)**	**77,81**
Set_Deadline/Get_deadline	**O(1)**	**O(1)**	**85,89**

Set_weight/Get_weight	**O(1)**	**O(1)**	**93,97**
Set_Notes/Get_Notes	**O(1)**	**O(1)**	**101,105**
Set_status/Get_status	**O(1)**	**O(1)**	**109,113**
TOTAL	**O(1)**	**O(1)**	

Truck.py

Method/Function	**Time Complexity**	**Space Complexity**	**Line**
__init__	**O(1)**	**O(1)**	**6**
Add_Package	**O(1)**	**O(1)**	**14**
Add_Miles	**O(1)**	**O(1)**	**18**
Get_Miles	**O(1)**	**O(1)**	**22**
Show_Packages	**O(1)**	**O(1)**	**26**
Remove_package	**O(1)**	**O(1)**	**30**
Remove_route	**O(1)**	**O(1)**	**35**
Add_route	**O(1)**	**O(1)**	**40**
Show_route	**O(1)**	**O(1)**	**44**
TOTAL	**O(1)**	**O(1)**	

B2: Application of Programming Models:

WGUPS requires this proposed solution to be stored and hosted on the local machine. As this program will be used on the same computer, which is hosting the application, there is no need for a communication protocol or interaction semantics. “CSV.Reader” is used to read CSV files to provide the data needed for this program. This proposed solution was developed in the PyCharm Community Edition 2020.2 IDE (Integrated development environment), using Python version 3.8.5.

B4: Adaptability:

The chosen algorithm, as defined above under B1, uses a greedy approach to determine a route to take. This algorithm would scale well with larger routes, and with additional locations that may be added in the future, as WGUPS plans to expand their business. There would be slight changes, as some placeholder values within the program are valid for solutions under 100 location identification numbers, but these changes would only take minutes.

B5: Software Efficiency and Maintainability:

The entire software solution runs, in a worst-case scenario, O(N^3), which is polynomial time, and efficient as per the terms of WGUPS’ requirements. A further breakdown of the program’s efficiency and Big-O time complexity is provided in section B3.

This software solution was designed with code readability, and maintainability in mind. Throughout the program, comments are provided at every major block of code to explain what each method does. These comments help to explain what is being done in every step of the code, so that if future changes are necessary, and WGUPS wants to enhance this solution without consulting the original developer, they have ample information to do so.

B6: Self-Adjusting Data Structures/D: Data Structure/Explanation of Data Structure/K: Efficiency, Overhead, Implications, Other Data Structures, Differences:

The proposed software solution uses a chaining hash table to store package data, arranged in key value pairs, with the key being the product ID, and the value containing all of the product's information, stored within a product object. The chaining hash table used within this program can add new values, retrieve values, and delete values. This data structure will grow as new information is added. A great benefit of this data structure is that interaction with it, such as looking up data, is fast, running in constant O(1) time complexity no matter how large this hash table grows. Combined with the ability for the chaining hash table to grow as new values are added to it, this allows for a time efficient, and scalable solution, should the program need more product data in the future. The chaining hash stores data in a list of lists.

As more data is added to the hash table, the insertion time will remain at O(1). No collisions will occur in this chaining hash table, as each hash table index contains a linked list, this is an advantage to using a chaining hash table. In the worst case, if this hash table were to grow extremely large, the runtime for a search on the table could become O(N). Trucks and cities could also be stored within a chaining hash table, and as the program grows larger with the addition of more trucks and cities more storage space would become necessary, and would approach O(N) for searching their respective hash tables, while remaining O(1) for insertions, as mentioned previously. The data within this hash table uses the product ID as the key, and contains the entire product's information within an object, as the value. Product objects contain both string and integer data types. Products can be searched for in this hash table by the key, and the value, containing the product object, can both retrieve data, or update data, through getter and setter methods. The package objects contain data which allow later functions, to determine the location by ID where a package needs to be delivered. Additionally, these object getter and setter methods allow the program to change the status of a package from "At the Hub" to "On the truck" to "Delivered" allowing a user to determine if the packages have been successfully delivered.

Additionally, it is important to note, as WGUPS will be hosting this program from their local machine, memory and bandwidth are not noted concerns, as the overall size of the program is relatively small, and the processing power required to perform the calculations is a fraction of what a modern computer is capable of. If the amount of data handled by this program were to increase substantially in size, more hard drive space may be needed for the CSV files, provided new locations are being added. Perhaps a solid-state drive may be of use in a scenario like this to allow for fast access of files stored within it. If the data size were to grow substantially, more processing power would be needed, but as of right now, with 26 routes and 40 packages, this program uses a minimal amount of memory and processing power. Additionally, as this software is to be hosted on the local machine, no internet bandwidth is necessary for the function of this program. The data input of this program currently uses CSV files, and Python version 3.8.5's "CSV Reader". This program aims to be easy to update in the future if sub-applications are developed by allowing the inputs to be altered to whatever data source WGUPS chooses to use. These changes would be easy to control as sub-applications are developed and used within the program. An important note is that within the program, currently, packages are manually determined for delivery, and this may be updated in the future if desired by WGUPS, by creation of functions which will generate the lists for delivery from the list of packages. The ability to respond to a growing market, as needed by WGUPS, is a major strength of this software solution

A binary search tree, BST, could have been used in place of the hash table used within this solution. I chose to use a hash table as inserting, deleting, and searching within the hash table is faster on average than the $\Theta(\log(n))$ time complexity provided by a binary search tree. Binary search trees are still efficient, however, searching can take a much longer time than a hash table.

Another data structure that could have been used is an array. Arrays would have stored values, like the hash table used, but they would not have a corresponding key. A hash table is faster than an array and can be searched easily by the key which is the product ID in this scenario. Another downside to using an array, is that to access a certain package object, as used in this project, you would have to search through the entire array, or know which index the package object was in.

I1: Strengths of the Chosen Algorithm:

The chosen greedy algorithm, as defined in B1, has distinct strengths that make it a viable choice for this project. This algorithm will scale as larger lists of locations to visit are added. This is one strength, as if WGUPS acquires trucks that have a larger cargo capacity, more locations will need to be visited, and this algorithm will be able to handle these scalable changes. Another advantage of this algorithm is that it has a lower overhead and time complexity compared with other algorithms, and runs much faster than an algorithm that would use a "brute-force" approach solve the routing issue.

I3: Other Possible Algorithms:

A dynamic programming algorithm could have been used to solve this routing problem. The dynamic algorithm would break the problem into smaller problems. This approach would reuse previously calculations as to not have to solve it again. This way, the mileages would only have to be calculated once, and would save time on future comparisons.

A genetic algorithm, a heuristic algorithm, could have also been used to determine the route. In a genetic algorithm, in this scenario, lists of places to visit, in order, would have been randomized, each given a value of their total mileage, and the "fittest", that is, the ones with the lowest mileages would go on to "crossover" to create "offspring" with other lists. For example, a list [1,2,3,4,5] may cross over with a list [2,3,1,5,4] creating an "offspring" composed of elements of both, perhaps [1,2,3,5,4]. Many generations of these crossovers could be created, and a route determined through this method. Although this may provide more optimal solutions than the greedy algorithm used within this project, the simplicity and relative optimality of the greedy algorithm suffices for the needs of WGUPS.

J: Different Approach:

If I had to do this project using different methods than what was performed, I would have created a weighted graph to obtain mileages between location points. In this project, I transposed the excel data table to make it easier to look up mileages based upon a nested dictionary. I would have allowed the truck objects to store mileage on them in order to allow the user to look up the mileage at any given time, and not just the total mileage of the route. The current algorithm may output different routes if there is a point within the route where two mileages are equal. This has not been an issue, as any route determined by this algorithm is well within WGUPS' requirement of under 145 miles. I would modify the algorithm to then check the next shortest path and choose the duplicate location that leads to the shorter next path. Though, for the needs of WGUPS the provided is more than sufficient.

Sources:

No sources were used in the production of this document.