Christopher Brown
Student #000968168
6/7/2020

WGUPS Routing Program

A: ALGORITHM SELECTION

The core algorithm I used to load and deliver packages was a form of the Greedy Algorithm. In order to meet the various conditions, such as time deadlines and capacity limits, I slightly modified the Greedy Algorithm to meet the programs needs but the prominent concept of the algorithm remains.

B1: LOGIC COMMENTS

To begin my algorithm checks to ensure the truck is located at the hub so that it can be loaded with packages. Some packages are delayed therefore they are not loaded onto the truck until the time they arrive at the hub.

Next the algorithm finds the packages that must be delivered together and have not yet been loaded onto a truck. These packages are discovered early in the process so that they can be easily identified throughout the following steps.

The algorithm then begins the loading process. In the next step the algorithm loads all of the packages that have a time deadline. Early packages are first loaded because they need to be quickly dispersed to customers, therefore I send these packages out before anything else. Any package that is not marked 'EOD' will be evenly divided between the trucks leaving at the start of the day. The Greedy Algorithm is used on the early packages to determine what the closest package to the current location is. After that the current location is updated to the loaded packages address and the algorithm looks for the next closest package with an early deadline.

After all of the early deadline packages are loaded the packages that must be on a truck with a specific name (Truck 2) are loaded. These grouped packages will be sorted at the end of the algorithm.

At this stage most of the packages with strict conditions have been loaded or eliminated because they are not yet ready to deliver. The remaining packages are

loaded in a one by one until the truck's capacity is full or there are no more packages to load. The algorithm first will find the closest package in the list of remaining packages. It will load this package and then update the current address. Next the algorithm looks for the closest package to the current address that is available. This process continues until one of the previously noted conditions is satisfied.

The final step is reordering the grouped packages that have been loaded onto the truck. The packages that are in groups are most likely out of order so the Greedy Algorithm is used once more to sort every currently on the truck that does not have a time deadline according to their distance just as before.

When all of the packages have been loaded and sorted the truck is then sent out for delivery and packages are popped off from the front of the list one by one until the truck is empty.

B2: APPLICATION OF PROGRAMMING MODELS

Python was the programming language used to complete the project and PyCharm was my chosen IDE. This project uses locally stored .csv files with the necessary distance and location data. To access data from these .csv files I used the csv package in python and the open function from this package. Everything is needed for the delivery system is run locally so I did not need to use a specific communication protocol as I would have if the data was located externally on a database or server.

B3: SPACE-TIME AND BIG-O

The code has been throughly documented with comments stating the Big O complexity for every section of code. There are descriptions throughout the program describing what each section of code does. In addition I have included a document with an overview of the Greedy Algorithm and its Big-O complexity.

B4: ADAPTABILITY

The algorithm will scale to work with a larger package list but the truck capacity will not self adjust. The list of locations and addresses is another component that can adjust to a larger dataset without needing to change any of the code. The programmer would have to adjust the capacity of the truck in the code if

trucks could handle more packages. In the future I could easily change the capacity of a truck if each truck had specific information about its unique capacity. It is also recommended to manually adjust the hash table size according to the number of packages to avoid collisions which may slow the project down.

B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

The worst case runtime found in my program is O(n^2). Due to the fact that the goal of the project is to optimize miles the time complexity is not necessarily the best that it could be. Although O(n^2) is not a perfect solution in terms of efficiency, the small size of the lists still allow the program to run fast enough for the user to experience no noticeable delays.

There are several aspects of the project that enhance future maintainability. I included comments for every portion of the code so that other developers can more clearly understand what the code is doing. I also decided to use datetime rather than time for much of the program so if in the future a client wants to expand the program to schedule deliveries for more than one day at a time it will be easier to scale. I also split the code up into multiple files and classes to make the code more organized.

C: ORIGINAL CODE

When user starts the application they can select the results section by entering the number 3 to show the final mileage and time. The final mileage is 108.7 miles and the final delivery is completed at 11:59AM. That status of every package can be shown at any time the user wishes to see. All packages have met every constraint such as time deadlines, being delivered with other specific packages, etc.

D1: EXPLANATION OF DATA STRUCTURE

The use of a hash table was extremely beneficial in this project. Hash tables are very fast as long as they are the properly sized. I created a custom hash table by utilizing lists. The hash table stored all of the package information including the address and status for each package that needed to be delivered. Using my custom hash table enabled me to quickly look up any information I needed. My hash table size is 40 elements and there are 40 elements in the package list so each time I look

up a key the operation only costs O(1) time. In my project I also update the status and time of the package within a hash function and use that to check what packages are remaining to deliver.

G1-G3: 1st, 2nd, and 3rd status checks.

Screenshots are located in the folder at the times of 9AM, 10AM, and 12:30PM.

H: SCREENSHOTS OF CODE EXECUTION

Screenshots are located in the folder documenting successful completion of the project. These screenshots capture the final delivery time and final mileage in the results section.

I1: STRENGTHS OF THE CHOSEN ALGORITHM

The Greedy Algorithm has many strengths in a delivery situation. One strength is the ability for a truck to reduce the miles driven by staying in the closest proximity to the current location. The truck is not likely to backtrack across long distances often. Another strength of the Greedy Algorithm is that it can efficiently scale with a larger set of data. As the company grows and changes to incorporate larger cities with more addresses to deliver to the algorithm will still be able to manage deliveries with no changes needed to the code.

I2: VERIFICATION OF ALGORITHM

When user starts the application they can select the results section by entering the number 3 to show the final mileage and time. The final mileage is 108.7 miles and the final delivery is completed at 11:59AM. That status of every package can be shown at any time the user wishes to see. All packages have met every constraint such as time deadlines, being delivered with other specific packages, etc.

I3: OTHER POSSIBLE ALGORITHMS

A slightly modified version of Dijkstra's Shortest Path Algorithm could be used to complete all of the deliveries. The algorithm would have to be modified to

only add the first 16 packages to the truck instead of every package. After the 16 packages are delivered the next 16 in the shortest path could be added and delivered. This pattern would continue until all packages are delivered.

Breath First Search could also be used to find the shortest path to every other node. The process would have to be done recursively after each delivery to ensure every package is delivered. Several there constraints would need to be included to keep the trucks capacity 16 or less packages and to choose which packages to deliver first but with some modifications Breath First Search could be useful in this delivery scenario.

I3A: ALGORITHM DIFFERENCES

Although both Dijkstra's Shortest Path and Breath First Search could potentially be used to help find a solution the Greedy Algorithm still seems to be the clearest way to solve the problem. There would need to be several modifications to the other algorithms when it comes to deciding how to implement them.

J: DIFFERENT APPROACH

If I were to start this project over from scratch there are several things I would do differently. First I would include a driver class to keep track of what drivers are assigned to specific trucks and it would prevent trucks from departing if no drivers are currently available. I would used also try using the city or zip code to deliver the packages according to their district. The packages with the same zip code are most likely going to be close together. I would use the greedy approach to deliver all packages within a zip code. Each truck would have an assigned zip code and work on that area until it is complete. When the truck delivers all of the packages in that district it can then move on to the next district. I would also try a different approach where early packages are not the first loaded. Instead after I ordered every package according to distance I would check if all of the packages meet their deadlines. If they don't meet their deadline I would move them forward in the list until every package is delivered on time.

K1: VERIFICATION OF DATA STRUCTURE

When user starts the application they can select the results section by entering the number 3 to show the final mileage and time. The final mileage is 108.7 miles and the final delivery is completed at 11:59AM. That status of every

package can be shown at any time the user wishes to see. All packages have met every constraint such as time deadlines, being delivered with other specific packages, etc.

K1A: EFFICIENCY

The hash table is created using a lists of lists. The hash table operates in O(1) time which makes it extremely efficient and fast. If I were using another structure such as a list I would have to search through each item to find what I am looking for which would require O(N) time to find a given package ID. I used the hash table to store all of the package data including the delivery address, delivery status, notes, deadlines, and delays.

KB: OVERHEAD

Hash tables are a very fast data structure. The time complexity for reading, storing, and looking up package information in my project from the hash table is O(1). There are no collisions in my hash table which allows all of these operations to be completed in constant time. If there were collisions these operations could take O(n) time because there could be multiple items in a single bucket of the hash table.

All necessary data for the application is stored on the users local machine therefore bandwidth is of no concern. Memory should not be a major concern because the program uses very little memory on the local machine. No transferring of information is done to external machines making both of bandwidth and memory nearly irrelevant.

K1C: IMPLICATIONS

When more items are added to the hash table the search time remains O(1) as long as the size of the hash table is set appropriately and the package ids are incremented by 1 for each new package. If the hash table size is not set appropriately there will be collisions which will slow the search time down. If the hash table size is too large excess memory will be used. If more packages were to be added to the list it would be very important for the programmer to increase the size of the hash table accordingly if they wanted hash table operations to be completed in O(1) time. One shortcoming of the current hash table design that I have is that it doesn't detect how many packages are in the list. In the future I

could potentially add a feature to dynamically size the hash table to the size of the detected package list.

K2: OTHER DATA STRUCTURES

One different data structure that could have been used in this project is an undirected graph. Each node in the graph would correspond to an address and have edges that go to every other address in the system. The edges of the graph would have weights corresponding to the distance between locations. This information could be utilized to traverse a route between packages in the list. An undirected graph has the same weight in both directions therefore a truck could travel in either direction between nodes.

There are many instances in the project when I could have also implemented a Binary Search Tree as a data structure. This kind of tree would allow me to organize the packages according to a specific element. For instance I could have used the package ID, the package deadline, or distance from hub to order the nodes in the tree. The tree would have lookup times of O(log n).

K2A: DATA STRUCTURES DIFFERENCES

The binary search tree would have average access, search, insert, and delete times of O(log n). If the tree was not properly balanced the worst case time complexity for these operations is O(n). The lookup times would not be quite as fast as a properly managed hash table which has lookup times of O(1).

In an undirected graph adding a vertex or an edge takes a constant time of O(1). The edges and vertex's would only need to be added once and would not need to be modified. Therefore the time to load the whole graph would depend on the size of the address list, O(n). To Query the vertices of the graph it would take O(v) time. This again is longer than a proper hash table look up time of O(1).

L: SOURCES

I primarily used Zybook's Data Structures and Algorithms II (zyBook ISBN: 978-1-5418-4355-4) as a source to gather the concepts and information for my algorithm.