WGUPS Routing Program

Section 1 - Programming/Coding

A. Algorithm Selection:

The algorithm used is the greedy algorithm

B1. Logic Comments:

Found in lines 9-15 of “algorithm.py”

B2. Application of Programming Models:

This scenario calls for finding the shortest distance possible for a truck to travel to deliver packages. To do this, you must consider the distances from any address to any other address, special requirements associated with each package, and the number of packages allowed on each truck at once. The greedy algorithm is in my opinion the best solution for finding the shortest path because it simply finds the shortest distance to travel next at any given moment. Built into the program are stipulations that allow packages with earlier required delivery times to be prioritized over those that can just be delivered by the end of the day. What is specifically important about how the algorithm sorts through the packages is that it focuses on the addresses to which they will be delivered, because in some situations, multiple packages are being sent to the same address. It can always be assumed that there are more packages than there are addresses. This program also finds the packages that have special loading requirements and allows you to manually load them onto whichever truck is necessary. The rest of the packages are sorted through depending on addresses and delivery times to create a priority list, and systematically moves through the list delivering each accordingly.

B3. Space-Time and Big O:

Big O notation evaluation for individual blocks of code can be found throughout the files. The Big O notation for the entire program is O(n^2)

B4. Adaptability:

This solution is easily scalable and adaptable to any city. All that is necessary is to find associated distances between addresses in the new location. The hash table the package data is stored in has an insert function and prompt to continuously add in new packages to add to the algorithm if necessary.

B5. Software Efficiency and Maintainability:

It is true that the “shortest path” is not always the most efficient path, but this solution takes it a step further. It not only finds the shortest path available to the truck in question, but also parses through the packages remaining to be delivered to find those that require a specified delivery time, and loads those packages into the trucks first. It also allows the user the option to manually load packages with special notes, such as requiring specific packages to be on a specific truck.

B6. Self-Adjusting Data Structures:

The “self adjusting data structure” chosen is a chaining hash table that is able to hold any number of packages within the created buckets.

C1. Identification Information:

Found in lines 1-5 of each file

C2. Process and Flow Comments:

Found throughout all of the code of each file

D. Data Structure:

Hash tables are great for storing data in an easily sorted list. The entirety of the package data was able to be stored in a hash table, making retrieving it a simple task, since it is all sorted using the package ID number as a key.

D1. Explanation of Data Structure:

The package data is stored within a data structure called a chaining hash table. They are sorted within the table using the “package ID” as a unique key index. As stated in the “assumptions”, this key is unique to each package as there are no collisions.

E. Hash Table:

The hash table created is found in the file “hashTable.py”. It uses the package ID of each package to sort the data into buckets within the hashtable.

F. Look-Up Function:

(Lines 25-45 in “main.py”) Allows the user to input the package ID for any package and find the rest of the information regarding that package.

G. Interface:

Screenshots are provided separately

H. Screenshots of Code Execution:

Screenshots are provided separately

Section 2 - Annotations

I1. Strengths of the Chosen Algorithm:

The two strengths of the greedy algorithm are that it is easily scalable, and simple to implement. The greedy algorithm only cares about the shortest possible route. This makes using it (and understanding it's process) quite easy. Adding in new information is also quite simple. All that is necessary is the address of any new package to be inserted into the program, and the distances to and from this possibly new address for the algorithm to consider in its computations.

I2. Verification of Algorithm:

The greedy algorithm allows the program to select the packages with the shortest delivery distances. Add in the fact that the program prioritizes packages with earlier delivery times and allows those with special notes to be manually loaded as needed, and the greedy algorithm fits all requirements given in the scenario.

I3. Other Possible Algorithms:

Two other algorithms that could have been used are Bellman Ford's algorithm and Dijkstra's Shortest Path.

I3a. Algorithm Differences:

Bellman Ford's (BFs) algorithm is used to find the shortest path through a dynamic programming approach. BFs algorithm uses a graph and a source vertex (which would be the Hub in our scenario). It differs from the greedy algorithm in that it could also account for negative values in a graph with weighted paths.

Dijkstra's Shortest Path is unsurprisingly similar to the greedy algorithm, but it differs in that it also accounts for future destinations, not just the best choice at that given moment. It does this by working backwards, starting at the ending destination and finding the shortest path possible from vertex to vertex until the last one is the starting point.

J. Different Approach:

If I did this project again, I would use Dikstra's Shortest Path for my algorithm. I honestly don't know very much about Dijkstra's Shortest Path but I'm certain it would prove to be very useful for this scenario. Dijkstra's Shortest Path algorithm would have worked well because it not only looks for the most efficient path at that given moment, but also takes into account future paths as well, trying to make the most efficient path possible overall. To implement this, I would make it so the truck has to make 16 deliveries and end up back at the hub. The algorithm would find the shortest possible path for the truck to take in order for 16 deliveries (the maximum number of packages a truck can carry at once) to be made and coming back to the hub for an efficient route. It would still work like the current solution in that it would give higher priority to packages with earlier required delivery times and special instructions, but the paths would become more efficient while still making the maximum amount of deliveries per delivery route.

K1. Verification of Data Structure:

Verifying that the data structure I used meets all requirements, I created a hash table without using any libraries, or any dictionaries of any kind.

K1a. Efficiency:

Chaining hash tables allow for any number of items to be placed in the hash table for storage. This allows for efficient scaling should the shipping business expand to more destinations. The way a hash table works is that it essentially stores a vertex (street address) and information regarding it's distance to other vertices that is readily available to be retrieved. Chaining hash tables do sacrifice memory for efficiency, but since this is a localized project (and not on any servers of any kind), the trade off is well worth it.

K1b. Overhead:

This solution does sacrifice memory overhead in order to gain back some efficiency. As stated before in K1a, the use of chaining hash tables sacrifices memory usage for efficiency. However this project is entirely localized, so worrying about bandwidth or server connections is not a problem currently. The program's space-time complexity is O(n^2) so there will come a point where there will simply be too many packages and addresses to efficiently wort through to find the best route. However, at the current (and foreseeable) size of the company, that shouldn't be a problem for a while.

K1c. Implications:

Adding more package data or changing the location of the shipping business altogether would not be a big problem. Adding new packages is not a problem at all if the program already has the address within it’s system, and adding a new address simply requires the distances to other addresses within the system for the algorithm to pick up the work from there.

Should the company expand beyond what would be feasible for run time to stay low, one way to allow the program to scale up would be to store the package data in an AVL balanced tree. This would prevent possible collisions between packages and allow the search time to be O(logN).

A different solution might be to implement multiple hash tables, each one for a different region. Allowing the user to choose the appropriate hash table, instead of forcing the algorithm to have to search for it everytime it tried to choose packages to load onto a truck, would allow the routing algorithm to remain at the current run time complexity of O(n^2).

K2. Other Data Structures:

Two other data structures that could have been used to meet all requirements would be graphs and balanced trees.

K2a. Data Structure Differences:

Graphs are different from hash tables because they focus on how each data point is connected to each other, while hash tables simply focus on presenting the data as raw vertices to be used in an algorithm.

Balanced trees are much like graphs in that they work around the connection from a data point to the other data points in the memory. They allow the algorithm to find the most efficient route to a specific data point.

L. Sources:

Information on Greedy Algorithms - Section 3.3 in reading material

Information on Dijkstra’s Shortest Path - Section 6.11 in reading material

Information on Dynamic Programming - Section 3.5 & 3.6 in reading material

Information on the Bellman Ford algorithm - GeeksforGeeks.org at the following URL: https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

Information on Graphs - Section 6.1 & 6.6 in reading material

Information on Balanced Trees - Section 5.1 & 5.5 in reading material

Information on memory overhead and hash tables - article on Medium.com “Hash table tradeoffs: CPU, memory, and variability” at the following URL: https://medium.com/@leventov/hash-table-tradeoffs-cpu-memory-and-variability-22dc944e6b9a

M. Professional Communication:

When beginning the program there are a few things to note. First off is that the time is based on a 24 hour clock (not 12). Screenshots of the code running provide a sort of “walk through” of how to run the code, but ideally you should load the trucks first (pretty obvious right?). The prompts will ask for instructions regarding packages with special notes. My choices are shown in the screenshots. Once that is done, send the first truck on it’s route by hitting ‘B’ and jump to a desired time. Given the requirements asked for 3 specific time windows, I chose 9:00. Jumping to 9 will then print the packages that have been delivered thus far. Typing ‘C’ will print the status of all packages in the system. Then send truck 2 on it’s route and jump to a new desired time, which in this case I chose 10:00. Same as before, the program prints off the newly delivered packages and asks for a new prompt where you can ask for the status of all packages again by typing ‘C’. Finally send truck 3 on it’s way and jump to a new time (I chose 1:00pm or 13:00). This prints off even more delivered packages. Hitting ‘C’ at this time will reveal that all packages have been delivered by 1:00pm (13:00). To bring the program to an end, and see the distances of the three trucks’ routes, hit ‘X’. Depending on which trucks you chose to load the packages with special notes, your results will be printed below. My results came out to be 104.5 miles.