Jaxon Weis
Data Structures and Algorithms II — C950

C950 WGUPS Routing Program

A. Algorithm Identification

Nearest Neighbor Algorithm

The goal for my program was to sort the packages for the shortest distance between packages. The first step was to make a table of distances from one point to another. I used a 2 dimension list called mileageTable. To find the distance between 2 points use the destination id of where you were to where you would go example: hub = 0, 195 W Oakland Ave = 5. So mileageTable[0][5] = 3.5.

B.

1. Logic Comments

```
currentLocation = 0 the hub
while truck.packageList has packages
    For every package in truck.packageList
            Find the shortest distance package from currentLocation
    Truck.mileage add package distance
    Truck.route add shortest package
    truck.packageList remove package
    Truck.currentLocation = package
Truck.route add hub
Truck.mileage add distance to hub
Truck.currentLocation = hub
```

This chunk of code creates the route for each truck. Starting with the hub going to whichever package is closest and keeps going. Until every package is in the route and the truck is sent back to the hub. This algorithm does well with getting a very low mileage count. I've compared the numbers to a brute force algorithm and it will usually be within 20%. Where this algorithm will go wrong is if there is a point where a close group of destinations leads the truck astray from making a delivery nearby but not the closest. Causing the truck to turn around and catch that destination. This algorithm is also quick. With only being O(n) meaning only having to loop the amount packages in the truck. Compared to the brute force algorithm which can take hours to run if the package list gets above 10.

2. Development Environment

This program was created in pyCharm 2020.3 with Python 3.8.6. Using a 64bit Windows 10 PC with Intel i3-8130U Cpu and 8gigs of ram.

3. Space-Time and Big-O

In Code.

4. Scalability and adaptability

The final big-O notation for the entire program is O(p*d) this came from the package constructor operation. The biggest hurder this program has is when it makes packages it sorts through the entire list of destinations to find the one with the same address. Then it applies the Id number from the address to the package to calculate mileage later. This could be sped up with a dictionary or hash table using the address as the key. From there the number of packages can be scaled to large numbers. The only other problem being the bubble sort algorithm in the sorting of the trucks which can only have 16 loops because of the size of the truck.

5. Software Efficiency and Maintainability

The software is fairly versatile in its current state. The trucks page can be easily updated if found that the trucks are traveling at a higher speed. Also have the ability to add more trucks if available. Those numbers can be updated just in the constructor of a truck, or even moved to the initializing statement if different trucks have different speeds. The sorting algorithm is efficient only needing to cycle through a loop for as many items are in the package list. To get better efficiency in the program will lead to worse mileage.

6. Self adjusting data structures

The strength of the Hash table is quick lookup times because there is a key that relates to the object's location. This would be better shown if the package id's were not incremented. Random id numbers would cause a problem of possible making a list of a thousand just so all 3 digit id numbers would have a dedicated spot in the list, or searching every item in a list for the proper id number. The problems that arise are if 2 objects fall in the same location this is a collision. You can either overwrite the position or you can make it a list to have multiple objects in the same spot.

C. Original Code

In code.

1. Identification information

In code.

2. Process and flow comments

In code.

D. Data Structure

The self adjusting data structure I chose was the Hash Table. It won over others because of the simplicity. Different data structures would help in the long run with the algorithm. Binary trees would be required to sort items on the left and right side of the node and could help with loading them onto a truck. Graphs could relate packages to each other making routes on their own. But Hash tables were simple and only require each item to have a key which would be the package id.

1. Explanation of data structure

The hash table is set up like a 2d array. With a set number of rows and an infinite amount of columns. The rows numbers are set when the table is created. Then when you insert an item you give a key to send it to a row. The key I used was the package id. Which was then processed through the hash function and modded to the size of the array to find its bucket. This would even work if the package numbers were not incremented. Unlike a list where I would add a new item and the index of the list was one less than the id number. Which would not work for random id numbers or id numbers with ‘a’ thru ‘z’ in them.

E. Hash table

In code.

F. Look-up function

In code.

G. Interface

To run this application you have to load packages manually one at a time. To load a package press 'l' and it will show all packages available to load. Enter the package id you want to load and then the truck id to load it on. Once you have a truck loaded with packages you need to sort them in a route with 's'. Once the trucks are sorted dispatch the trucks with 'd'. Once the trucks are dispatched you can jump to the time when the truck is done with 'j' or enter a time to jump to with 't'. You can view the status of trucks with 1, 2, and 3, or view all packages with ‘v’.

1. First status check

In screenshots.

2. Second status check

In screenshots.

3. Third status check

In screenshots.

H. Screenshots and Code Execution

In screenshots.

I.

1. Strength of chosen algorithm

Nearest neighbor was chosen because of its ease of use. The biggest obstacle of the algorithm is that I used a bubble sort to get the closest destination and removed it from the list. Which has created a terribly slow O(n^2) algorithm. If the package list was stored differently it could have helped the algorithm to be faster. But I compared the mileage and time taken to a brute force algorithm and found that the shortest neighbor runtime was not noticeable compared to brute force and had comparable mileages.

2. Verification of algorithm

In screen shots.

3. Other possible algorithm

I experimented with the idea of doing a brute force algorithm which would show every possible route and choose the best one. It was showing promising results with low package numbers < 10 and could actually be used as a solution keeping the package list low. That is why it still remains in the code. The problem is as soon as you crossed into the 10 package mark you would see a considerable time delay in finishing the sorting. So much so that I coded in a progress bar to let you know the program didn't stall. My quick math suggests that a full 16 package load could take a day to run.

I had the beginning thought of an algorithm that would deliver packages closest to the hub first. This would fix the problem with small clusters causing my truck to run away from a nearby destination. But would trade that problem with the problem of having the next delivery being slightly farther from the hub but on the other side of the hub. Making the next run twice the distance of the destination from the hub.

a. Algorithm Differences

The brute force algorithm is a very different approach than the current method. Brute force takes the package list and comes up with every possible combination of routes. Then it chooses the best one. Which would give the best mileage results but at the cost of long run times. Which is why it wasn't chosen.

The closest to the hub first algorithm is not too different from what I have currently. It would only take a couple of line changes. The biggest difference would be the errors that it could make. If all the packages were north this wouldn't be a bad algorithm. But if the packages listed contained destinations from the north and south then the truck would by driving by the hub at worst case after every delivery. Which is why it wasn't chosen.

J. Different approach

Doing this project again I would have liked to have done better on the class structure. When starting this project I hadn't done a lot of python projects. Which made me set up my project like I would have with java and that isn't the proper way to do it.

K.
1.

In code.

a. Efficiency

The more the number of packages the slower this program will run. Which is true of any program like this. But the biggest slowdown will come from the package creation. Where every package is matched with a destination. This is especially true if the increase in packages comes with an increase in destinations. From there the biggest slowdown is sorting the trucks. Which with a max load of 16 doesn't show any delay.

b. Overhead

The space complexity of adding more packages isn't a problem. With each new package is just one more spot in the hash table. At a certain point it will be helpful to increase the bucket size to deal with the large load of packages. The average size of the buckets would only be the number of packages divided by the number of buckets. The good thing about the hash table is that it doesn't leave empty holes for data to fill. Which would waste a lot of space.

c. Implications

Increasing the number of trucks would be a nice way to increase the number of packages to be delivered in a certain time. But with that luxury comes the difficulty of managing the trucks and loading them properly so they are more isolated to their own area than walking all over each other. Another thing is the memory usage is going to increase. The same thing will happen if this program is used for multiple cities. Currently the program only has to look through 20 or so destinations to assign a destination id to a package but if we include multiple cities that could take ages and each city should have its own destinations list so one package search is contained to its own city.

2. Other data structures

Other data structures I would have liked to try to implement were trees and graphs. Trees would have been a nice way to connect all the elements back to the hub and essentially give a truck a branch to deliver and come back. Graphs would make it interesting because the points can have multiple vertices showing multiple paths a truck could take.

a. Data structure differences

Trees have to have a starting node and branch off from there. The tree can only have 2 branches so deciding how each package would fill the tree would be difficult. Graphs are not limited like that. Graphs can have every point connected to every point. But to keep navigation times smaller each package should only connect to a nearby package.