Bryan Schloesser

Student ID# 001103177

C950 – Data Structures and Algorithms II

A: Self-Adjusting Algorithm Identification

The Self-adjusting Algorithm that I used was a Nearest Neighbor Algorithm. The location in the code for this the function Distance.find\_closest. This is located in the Distance.py file.

B1: Logic Comments

Methods:

Location.get\_location\_index(truck.delivery\_locations[index])

This method takes the delivery location data currently being worked on in the loop and finds the index of the address in the addresses\_list. This is used to find the address in the distances\_list to find how far away the location is from where the truck currently is.

get\_distance(current\_locale, destination)

This method takes the current truck location and the destination the loop is currently trying to find the distance for. It looks these up based on the value of the first index. This value determines what column and row to look at in the distances\_list. It will find the correct dinstance between the two locations.

Run-time:

Location.get\_location\_index: O(n)

get\_distance: O(1)

DEFINE find\_closest(location of truck, truck object)

INITIALIZE and SET closest to 100.0 (float)

INITIALIZE and SET next\_destination\_index to -1

INITIALIZE and SET index\_to\_delete to -1

FOR the length of truck.delivery\_locations

INITIALIZE destination and set to the integer value of Location.get\_location\_index(truck.delivery\_locations[index]) (The index of where the truck is in the locations list)

INITIALIZE distance and set to the float value of get\_distance(current\_local, destination). (The distance between the truck current location and the current destination being looked at.)

IF distance does not equal 0.0 and distance is less than or equal to closest:

SET closest to current value of distance

SET next\_destination\_index to current value of destination

SET index\_to\_delete to the current index of truck.delivery\_locations

END IF

END FOR

REMOVE truck.delivery\_locations[index\_to\_delete] from truck.delivery\_locations

SET truck.mileage to truck.time + closest

SET truck.time to truck.time + (closest/truck.speed)

RETURN value of Location.addresses\_list[next\_destination\_index] (The address info for the next closest destination)

B2: Development Environment

The development environment I used to develop this location was, for starters, on a Macbook Pro running macOS Big Sur version 11.5.2. The IDE I chose was PyCharm Community Edition 2021.2. I chose this one because it was heavily recommended by multiple sources.

The version of Python my development environment was running during creation was Python 3.9.7.

B3: Space-Time and Big-O

In the comments of the code, I went into some detail explaining space-time complexity for each loop. I will provide a break down here as well. The “major segments” of the program I have listed here:

Distance.find\_closest(current\_locale, truck)

This method has a time complexity O(n). This is due to the one for loop in the method. The method is reliant on the length of the truck objects delievery\_locations list. We actually know the max length of each trucks delivery\_locations list currently. So one could argue that the complexity is O(1). However, in terms of scalability, the algorithm’s loop will run N times based on the size of the lists.

Location.get\_location\_index(address\_name)

This method has a time complexity of O(n). Just like the method above, this methods time complexity also relies on the length of a list. Whatever the length of the addresses\_list is, determines how many times this loop will run.

Package.package\_data\_to\_hash\_table(hash\_table)

This function (identical to the other data extraction functions) all have a time complexity of O(n). This is due to the loop that looks at each row in the csv files and adds the row to the respective list. The time complexity is based entirely on how many rows of data are in the csv files. The function similar to this is CSV\_Extractor.import\_csv(csv\_file). This function is used for generating the lists for the distances and locations and has the same time complexity of O(n).

While user\_input != ‘quit’

This function has a time complexity of O(n^3). The amount of times it will run is based solely on what the user enters. Unlike the above listed functions and methods, this is O(n^3) time complexity because all the other loops are nested within this one. Those will run N times based on variables such as list length.

The entire program’s time complexity overall comes down to O(n^3). This is because the program is only as slow as the slowest piece. This will come down to the find\_closest function O(n^2) and the fact it is nested within the while loop.

B4: Scalability and Adaptability

The program is entirely capable and designed well enough to support scaling in any aspect. From data extraction, to adding additional trucks and even hash\_table expansions, there is a way to increase most aspects of the application.

The data is extracted from CSV files. The program will take however many packages or locations that are present in those CSV files. This allows any amount of locations and packages to be entered in.

The hash table also is designed to be customizable in size. It also has collision detection. If there are already packages in the selected bucket, it will simply appended the package to the bucket. This is more likely to occur the more packages there are.

More trucks will also become more probable with any sort of expansion to the business. The application is designed to make most aspects of the truck customizable. This makes it adaptable to almost any situation. The departure time, speed, and package lists can all be easily adjusted to meet any need. It is also very simple to add additional trucks. While it does require additional coding, the framework for the truck objects is designed to easily add any amount of trucks.

All these different way provide scalability and adaptability when it comes to the eventual growth of packages needing to be delivered. The hash table can dynamically handle additional packages with bucket size increases and collision support. With the package number increasing, the truck amount will also eventually increase. With being able to add trucks easily on demand and the self-adjusting hash table, this program has great scalability and adaptability.

B5: Software Efficiency and Maintainability

The software is efficient due to it doing what it needs to do with a relatively low time complexity. The efficiently all comes down to how many packages and locations there are. The more there are, the slower the application will be. That being said, each algorithm has a time complexity of O(n). With an application that supports scalability, this is as efficient as you can get.

From a maintainability aspect, I split the application in multiple different files based on what the function is working with. If it was determining distance, it would be in the distance.py file. If it was working with the trucks, it is located in the truck.py file. This makes it significantly easier to add more functionality and it is overall easier to work with.

B6: Self-Adjusting Data Structures

The hash table for storing package data is one of the self-adjusting data structures. A strength of using a hash table is the lookup speed to find the packages. Other methods would require multiple loops to look through the lists. A weakness is that collisions are inevitable. The more collisions you have, the slower the hash table becomes. However, we know how many packages on average there are. So we can adjust the hash table to avoid collisions as much as possible.

Another self-adjusting data structure is the nearest neighbor algorithm that is used to determine the next closest destination. The nearest neighbor algorithm is that it is guaranteed to find a good solution. It might not be the most optimal, but the solution it comes up with is good enough in most cases. For example, in my nearest neighbor algorithm, it looks at all the locations distances from where the truck is currently at and finds the closest place. However, it does not think about whether or not the full route is longer than choosing some other options. As in, might be more optimal to choose a destination farther away if they are closer to multiple other destinations.

D1: Explanation of Data Structure

The hash table is where all the package data is stored. This data is then retrieved for use by sending a copy of the data into package lists for each truck. The data is then used to determine where the truck needs to go. There is a separate list of all destinations that is based on the delivery addresses for each package within the package list. This data is then used to determine how far away each location is from where the truck currently is. This is where the connection of my nearest neighbor algorithm comes into play. It does the calculation of where to go next. Once it determines where to go next, it changes the trucks currently location to that address and updates the mileage. Lastly, the truck then gets to the location, checks the package data for any packages that need to be delivered and will deliver them. This updates the status and delivery time of the packages.

Overall, the hash table and the nearest neighbor are connected by the data being used. The data gets passed from the hash table and into the algorithm and then said data gets updated in the hash table based on results of the algorithm.

G1: First Status Check

See attachment “First\_status\_check\_screenshot”

G2: Second Status Check

See attachment “Second\_status\_check\_screenshot”

G3: Third Status Check

See attachment “Third\_status\_check\_screenshot”

H: Screenshots of Code Execution

This information is in the three screenshots. This gets printed out with each execution of the program.

I1: Strengths of the Chosen Algorithm

The nearest neighbor algorithm always chooses the most optimal choice at the current time. For example, in the application, it will compare the distances for all destinations for where the truck currently is at. It finds the closest destination (shortest distance) and goes there. This will always be a relatively efficient way to determine a quick route. Another strength is that it is very easy to understand. The logic is not too hard to wrap your head around. This allows anyone to analyze the code and improve upon it or add functionality to it.

I2: Verification of Algorithm

I have verified that my nearest neighbor algorithm works by making sure all packages are delivered on time. I also made sure that the trucks do not have a combined mileage that is greater than 140 miles. Lastly, all the trucks complete their routes before the end of the day. (4PM)

I3 and I3A: Other Possible Algorithm

One example of a possible algorithm to use instead of the nearest neighbor method is to use a variant Dijkstra’s algorithm. A brief explanation is that Dijkstra’s algorithm will look at the starting point and find the distance between all the connecting nodes as well as the nodes those are connected to. With the nearest neighbor algorithm, it only looks at the distances that the truck is currently at and chooses the closest destination to said point. However, with Dijkstra’s algorithm, it checks distances between all nodes to find an optimal route. This is biggest flaw I thought about when implementing the nearest neighbor algorithm. There may be a shorter path later in the route, but where the truck currently is at, it would not be the closest next location. Dijkstra’s algorithm considers that possibility.

A second example would be to use the 2-opt algorithm type. This algorithm was designed in the thought that, with the traveling salesman problem, a route that crosses over itself is not an optimal route. The time complexity of this is O(n^k) where k is the amount of edges it needs to switch to create a route that does not overlap itself. The time complexity of this algorithm is higher than the nearest neighbor option. However, it will provide a more optimal route in most cases. (Lawrence Weru, Stem Lounge)

J: Different Approach

If I were to do this program again, I would like to look into a different solution than a hash table to store the package data. Either that, or at least look into a different method for the actual hashing. The formula for hashing I used (package\_ID mod size\_of\_hash\_table) works, but will result in more frequent collisions.

K1A: Efficiency

The lookup function for the packages will suffer the more packages get added into the hash table. The reason for this is due to collisions. In the search function for the hash table, the performance is O(n). It relies on how many packages are in the same bucket. The more items in the same bucket, the longer it will take to search through each index and find the correct package to return.

K1B and K1C: Overhead and Implications

The space usage of the application will grow with the increase in packages. The hash table will need to get increased to avoid collisions as much as possible. With the increase of packages, the various lists used in the application will also use up additional space in memory.

The trucks package lists will get larger and it would be assumed that the location to deliver to will also increase. This increases the lists used for locations and the destination distances as well. Since the calculations with both distance and time in the application rely on float numbers, they take up more space in memory. With the addition of more packages to deliver, there will be more space required in memory for these additional numbers as well.

K2A: Data Structure Differences

One example is to use a HashMap instead of a HashTable. Hash Maps allow multiple threads to access the data at the same time. This makes it faster than a hash table. A HashMap is also fail-fast. This means that if data is being modified and another thread is attempting to handle the data, it will throw an error. HashTables are not fail-fast so they do not throw an error. Overall, the use of a HashMap would be an overall better implementation. (Chaitanya Signh, Beginners Book)

A second example would be using a Skip List. A skip list is essentially just two sorted linked list in parallel. The bottom list is where the data elements are while the top is built randomly. A skip list uses less memory than any other structure. It is also comparable to a Hash Table in construction time. That being said, hash tables are still quicker. (Mark P Neyer, UNC.edu)

L: Sources

Neyer, Mark P. “A Comparison of Dictionary Implementations.” *UNC.edu*,

<https://www.cs.unc.edu/~plaisted/comp550/Neyer%20paper.pdf>. Accessed 20 September 2021.

Signh, Chaitanya. “Difference between HashMap and HashTable.” *Beginners Book*,

<https://beginnersbook.com/2014/06/difference-between-hashmap-and-hashtable/>. Accessed 20 September 2021

Weru, Lawrence. “11 Algorithms for the Traveling Salesman Problem.” Stemlounge,

<https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/> Accessed 16 September 2021