**Tracking System Overview**

**(A.) Stated Problem:**

The tracking system is a delivery service that is seeking to optimize its DLD (Daily Local Deliveries). This overview is used to provide the most optimized routing solution for the delivery distribution for the tracking system. A brief summary of this project is that the tracking service has 40 packages needed to be delivered using 3 separate trucks. Only two drivers are available, so the third truck will not be out-delivering until one of the first two trucks returns to the hub. Some packages have special notes for handling the packages. To solve these business problems, we implemented a series of statements to insert packages into the trucks based on the conditions presented in the problem, including the Nearest Neighbor Algorithm to optimize routes and mileages by sorting and looking up the nearest address from the previous address. This will unload packages from a temporary list and cycle through the algorithm until none is left. This document will present the various components of the delivery tracking program and provide a descriptive summary of the application of statements and methods.

**(B2**.**) Programming Environment:**

The programming environment primarily uses the Pycharm IDE Community Edition with Python language version 3.9. All data are hosted locally (xlsx files) and then transformed into csv format (raw date format with no formatting). The IDE reads directly from the csv files and load them into the tracking program. Since no alteration (add, remove, update, etc.) to the original data is needed while running this program, other programming environments, such as SQL, are not needed. Furthermore, there is no need to utilize a network infrastructure in order for the program to contact a remote database since the information is hosted and processed on a local server/machine.

**(B1.) Algorithm Overview:**

The Nearest Neighbor Algorithm is implemented by following the below steps:

1. Initialize all addresses as unvisited and their corresponding packages as undelivered
2. Select the TRACKING SYSTEM Hub address as the initial vertex
3. Look up addresses to find the closest/shortest edge connecting the current vertex with the unvisited vertex/unvisited address
4. Mark the current vertex as visited/delivered, move onto that closest vertex and mark it as the new current vertex.
5. Repeat step 3 and 4 until all addresses are emptied from the “unvisited” list, then terminate.

The Nearest Neighbor Algorithm has the worst-case performance of O(N3) and worst-case space complexity of O(N). Though having a best-case complexity, the worst-case complexity is commonly used to indicate the upper bound on the resources (running time, memory, etc.) required by an algorithm. The worst-case complexity is the most likely upper bound when using the Nearest Neighbor Algorithm to sort addresses and append packages to the trucks. **(I1-2.)** Using this algorithm is very intuitive and simple to calculate efficient routes to deliver the packages. All we need to do is find the nearest address out of all and repeat the process. Because of its simplicity to implement in the program, it can reduce the difficulty to maintain the program and implement changes to meet future business needs as there is no need to alter the algorithm whether the amount of packages changes or not.

**(I3.)** Dijkstra’s algorithm can be an alternative to the Nearest Neighbor Algorithm. It would initialize the hub and assign every address to that hub a tentative distance value. Each of those addresses will calculate the distance between them and all their neighbours and then compare it to the previous value these addresses were assigned to. The smallest value with be kept. Repeat the process until the shortest route is found. This algorithm efficiently finds the shortest overall mileage a truck needs to travel through. However, the implementation of this algorithm is extremely complex and hard for a simple solution that TRACKING SYSTEM is aiming to find. Another alternative can be the Bell-Ford algorithm. This algorithm makes each address calculates the distances to all other addresses and store that information in a table. Then each of those addresses sends the table to all other addresses. Once an address receives the distance information from all its neighbours, it calculates the shortest distance and reflects the new information in its own table. This algorithm can be easily implemented for a small number of packages, but the scalability of this algorithm, compared to the Nearest Neighbor, is extremely inefficient (imagining 5000 packages, each has 5000 tables that contain 5000 distance information and cross-compare them).

**(B1. Continue…)** Below is the rundown of the Nearest Neighbor Algorithm implemented in this TRACKING SYSTEM program:

**def nearest\_neighb(delivery\_vehicle):**

**temp = []**

**mileage\_tracking\_list = []**

**vehicle\_location\_tracking = []**

I created 3 temporary lists to hold different data. The first list is temp, which is used to hold packages waiting to be delivered. The second list is used to hold mileage information as the delivery truck visits each location. The third list is used to hold the data of the location as a given truck traverses through. the last item in this list will be the newest location of that truck. The Big O for this part is a constant O(1)

**for package\_id in delivery\_vehicle.r:**

**temp.append(pkg\_hashTable.lookup\_key(package\_id))**

I used the for loop here to append the packages that were preloaded into a given truck will all be appended to the temporary list. Big O here is O(N).

**while len(temp) >= 0:**

A while loop is created to sort packages in the temp list and remove them from the temp list. Once they are added to the package list of a given vehicle. All codes under it will go through N iterations. The Big O is O(N).

**upcoming\_pkg = None**

declare the next package the truck will travel to be upcoming\_pkg and temporarily set it to None. This value will get updated after the first iteration. Big O – O(1)

**if len(temp) == 0:**

**break**

if statement is introduced to prevent while loop from going indefinitely once there is no longer anything in the temp list. It will only be checked once for each while loop. Big O – O(1)

**else:**

Here, I am to compare all the addresses of the existing packages in the pkg\_awaits\_delivery list with the current address to find the next nearest package.

**temp\_list = []**

the temp\_list is created to hold distance data for comparison

**for package in temp:**

for statement is created to iterate through all packages in the temp list (as long as the temp [] is not empty). Big O – O(N)

**local\_file = "TRACKING SYSTEM\_Address\_File.csv"**

**with open(local\_file) as address\_data:**

**for x in list(csv.reader(address\_data)):**

**if delivery\_vehicle.n in x[2]:**

**a = int(x[0])**

**if package.b in x[2]:**

**b = int(x[0])**

Open and read the address csv file. Use the delivery truck’s current location as variable a and the location of all packages as variable b. These two variables will be used to find distance values. Big O – O(1)

**local\_file1 = "TRACKING SYSTEM\_Distance\_Table.csv"**

**with open(local\_file1) as len\_edge:**

**\_edge\_len\_data = list(csv.reader(len\_edge))**

**vertex\_vertex = \_edge\_len\_data[a][b]**

**if vertex\_vertex == '':**

**vertex\_vertex = \_edge\_len\_data[b][a]**

Open the distance csv file and use the a and b variables from above to find the list of addresses that the current truck is connected to and the distances between them. The list of distances will be stored in the vertex\_vertext variable. Since there are empty cells in the distance table, when the csv reader encounters an empty cell, I use an if statement to flip the row (a) and columns (b) to columns (b) and row (a) to avoid index out-of-range error. Big O – O(1)

**temp\_list.append(vertex\_vertex)**

**shortest\_d = min(temp\_list)**

All elements in the vertex\_vertext list are appended to the temp\_list. I used the min function to find the smallest value in that list and assign that to the shorest\_d variable. Big O – O(1)

**for y in vertex\_vertex:**

I used the for statement again here to iterate all items in the vertex\_vertex list (list that hold all distance data). The y variable represents any item within the list as long as it’s not empty. Big O – O(N)

**if y == shortest\_d:**

I used the if statement here to find if any y value equals the shortest distance value obtained previously

**for b in x[0]:**

**delivery\_vehicle.n = x[2]**

for b (package address) of the y value in the address table, the address index of y retrieves the address string and is assigned to the delivery vehicle location. Big O – O(1)

**\_v = float(shortest\_d)**

I convert the shortest\_d into float so it can used to calculate mileage. Big O – O(1)

**upcoming\_pkg = package**

the package the \_v represents will be the next package the truck will go to. Big O – O(1)

**if len(temp) != 0:**

creating an if statement to prevent popping even when the list has been emptied to avoid errors. Big O – O(1)

**temp.pop(temp.index(upcoming\_pkg))**

this pops the index id of the next package from the temp list to indicate it's no longer there. Big O - O(1)

**else:**

**break**

**mileage\_tracking\_list.append(\_v)**

append the value of distance between the last package and the next one to the mileage list. Big O – O(1)

**vehicle\_location\_tracking.append(upcoming\_pkg.b)**

Append each location the truck has been through. Big O – O(1)

**delivery\_vehicle.q += datetime.timedelta(hours=\_v / 18)**

Newest time of the truck to the closest address/pkg. Distance/Speed = Time.

Big O – O(1)

**upcoming\_pkg.k = delivery\_vehicle.q**

**s**et the delivery time of the package. Big O – O(1)

**upcoming\_pkg.j = delivery\_vehicle.l**

set the departure time of the delivery trucks. Big O – O(1)

**delivery\_vehicle.n = vehicle\_location\_tracking[-1]**

I use the last item in the truck location list for the value of the truck's latest location.

Big O – O(1)

**delivery\_vehicle.p = sum(mileage\_tracking\_list) #**

I then sum the cumulative mileage of a truck to get the total distance of a given truck.

Big O – O(1)

If we look back at the worst-case complexity of entire the Nearest Neighbor Algorithm that’s being implemented in this TRACKING SYSTEM program, it would be as below:

**O(1) + O(N) + O(N(1 + 1 + N(1 + 1 + 1 + N(1) + 1 + 1)) + 1 + 1(1) + 1 + 1 + 1 + 1 + 1 + 1) + O(1) =**

**O(1) + O(N) + O(N(2 + N(5 + N(1))) + 8) + O(1) =**

**O(1) + O(N) + O(N3) + O(1) =**

**O(N3) =**

**(B3.)** Below is the rundown of each major segment of the TRACKING SYSTEM program in using the big-O notation:

**class delivery\_Vehicle(object)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Big O - Space | Big O - Time |
| \_\_init\_\_ | 69 – 77 | O(1)/constant | O(1)/constant |
| Worst-case complexity |  | O(1)/constant | O(1)/constant |

**class myPkg (object)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Big O - Space | Big O - Time |
| \_\_init\_\_ | 84 – 95 | O(1)/constant | O(1)/constant |
| \_\_str\_\_ | 97 – 103 | O(1)/constant | O(1)/constant |
| Worst-case complexity |  | O(1)/constant | O(1)/constant |

**class hash\_table\_initiate**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Space Complexity | Time Complexity |
| \_\_init\_\_ | 111 – 114 | O(1)/constant | O(1)/constant |
| get\_hash\_key | 117 – 118 | O(1)/constant | O(1)/constant |
| Insert\_into\_hash | 121 – 130 | O(N) | O(N) |
| lookup\_key | 133 – 137 | O(1)/constant | O(1)/constant |
| Worst-case complexity |  | O(N) | O(N) |

**Load packages into truck**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Space Complexity | Time Complexity |
| Packages\_for\_truck\_A | 143 | O(1)/constant | O(1)/constant |
| Packages\_for\_truck\_B | 144 | O(1)/constant | O(1)/constant |
| Packages\_for\_truck\_C | 145 | O(1)/constant | O(1)/constant |
| Worst-case complexity |  | O(1)/constant | O(1)/constant |

**Instantiate package objects and load them into trucks automatically**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Big O - Space | Big O - Time |
| open cvs files | 148 – 161 | O(1)/constant | O(1)/constant |
| Instantiate package | 164 | O(1)/constant | O(1)/constant |
| Insert into hash table | 166 | O(N) | O(N) |
| Worst-case complexity |  | O(N) | O(N) |

**Instantiate truck objects**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Big O - Space | Big O - Time |
| Assign attributes to trucks | 175 - 184 | O(1)/constant | O(1)/constant |
| Worst-case complexity |  | O(1)/constant | O(1)/constant |

**UI interface**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Line Number | Big O - Space | Big O - Time |
| def ui\_menu () | 192 – 217 | O(1)/constant | O(1)/constant |
| def truck\_mileage\_check () | 220 - 260 | O(1)/constant | O(1)/constant |
| def status\_check() | 263 - 326 | O(1)/constant | O(1)/constant |
| class main | 329 - 339 | O(1)/constant | O(1)/constant |
| Worst-case complexity |  | O(1)/constant | O(1) |

**(B4-5.)** The overall implementation of the algorithm and statements provide a relatively straightforward solution for the TRACKING SYSTEM program. For the majority of the time, the complexity (both time and space) is fairly linear to execute. Overall, with space complexity being O(N) and time complexity being O(N3), the program's scalability is extremely easy to adapt to a growing number of packages. Minor changes to the codes will be needed while scaling the program. For example, the maximum load and/or the number of trucks can be changed to fit the scalability. The conditions need to be revised to accustom the special handling of certain packages (e.g. won’t arrive at the hub until 9:05 AM, has to be on truck 2, etc.). Another issue with the original programming code is, it has a specific implementation to solve the not enough drivers for the number of trucks (2 drivers for 3 trucks) by making the third truck wait until at least one of the first two trucks finished delivering. This section needs to be changed manually depending on the driver: truck situation. Other than minor updates of the program to better solve future business needs, no major alteration is needed for both the core algorithm or UI interface. This makes the TRACKING SYSTEM program easy to use and maintain. It is also extremely efficient with the time complexity of O(N3) and space complexity of O(N).

**(B6., K1-2.)** **Self-Adjusting Data Structure:**

I used a hash table (hash map) to implement a flexible data structure for the TRACKING SYSTEM program. A hash table is a data structure that collects a series of keys and uses those keys to map to the actual data they are corresponding. **(K1a.)** Hash table is very efficient for referencing data and extremely friendly for time complexity in this program. When a statement is looking up information on a specific object, the program will reference key indices only without the necessity to load the entire data. This can reduce the reading of information from N to 1. As the amount of packages increases, because of a hash table, the time increased to read data is merely restricted to going through a list of indices. **(K1b.)** With this method, it can also hugely reduce the space complexity and avoid using a large amount of memory to host data temporarily. For example, a key can map to a series of information hosted locally. When the program executes statements, it only needs to reference and load data as needed to compute the necessary statements. Thus, the space complexity for a majority of methods is O(1) because of this reason, as there is no need to host a large amount of data that are not needed for computation. With the increase in the number of packages, the increase of space complexity for a hash table is O(1), as it only needs to store additional keys, which is a constant number. Above is the strength of using a hash table. However, should be program requires a heavy addition, removal, or update of the data (even to the original data), then a hash table can prevent data from being readily available for alteration. For example, if the TRACKING SYSTEM program requires accessing data from a remote database, and will have to change information of the original data in the database, the hash table can add an extra layer because of the incessant referencing of indices and keys before you can reach the data.

**(K1c.)** One thing that can impact the hash table data structure in the TRACKING SYSTEM program is the change in truck numbers. Each truck is an object that stores a set of keys that references the packages that are on that truck. So, if the number of trucks increases, the amount of hash table will increase proportionally.

**(K2a.)** Two other dynamic data structures that can also be used for this type of program are the heap and stack. Heap is a tree-based data structure, where the priority of the elements is in ascending or descending order. Unlike a hash table, where indices are used to reference the actual data, a heap data structure is more cumbersome in terms of space complexity (storing data). However, a heap data structure is very flexible as you can access data in any particular order. One way I can use heap in this program is using the hub as the top node and from there, branch into three child nodes (the first package of the 3 trucks), then branch further based on the algorithm sorted order. The stack data structure uses the last in first out principle. Very efficient program but lacks flexibility. Using stack for this program will increase the efficiency of delivery by loading the sorted packages (using the algorithm) in backward order. In other words, the first address that the truck will arrive at from the hub will be loaded last.

**(J.)** If I do this project again, I would use the stack data structure for the TRACKING SYSTEM program. This data structure is highly efficient. It can optimize the loading and removal speed of packages to/from each truck. I will be able to implement it by presorting the packages using the Nearest Neighbor Algorithm and manually adjusting packages with special requirements before loading using stack. Another implementation I can possibly include is the live-graph feature on top of the plain-text status for the packages. This implementation will require the use of an outside program to visually display the paths each truck traverses through as they status updates.