A: ALGORITHM SELECTION

Stated Problem:

This project aims to leverage an algorithm to determine an ideal route/delivery distribution for the WGUPS Delivery company using the common high-level programming language Python 3.7. Forty packages must be split across three trucks with two drivers and must be delivered following a very strict set of requirements. These requirements include priority deliveries, mileage limitations, late arrivals, etc. To solve this issue, the delivery system can be built around these requirements and around an object-oriented approach so that the attributes of various entities (trucks, packages) can be modified at will.

The self-adjusting **“Nearest Neighbor”** algorithm was used in the program to deliver the packages. This algorithm would allow the truck to find the nearest possible delivery from its current location out of all the remaining packages on the given truck.

B1: LOGIC COMMENTS

**The Nearest Neighbor Algorithm**

For the WGUPS Routing System, the “nearest neighbor algorithm” has been adopted using the following method:

Pre-Requisite Steps:

1. A hash table is initialized to hold all of the packages.
2. A dictionary is created that maps each address with a corresponding address number.
3. Packages are loaded on their respective trucks, each with its own list called a remaining package list.

**The “minDistanceFrom” function / The nearest neighbor algorithm explanation**

Each truck uses the algorithm on its own package list

1. Create placeholder variables to store the distance to the closest delivery, the location of the new delivery, and the assigned number of the package .
2. Go through each of the items in the remaining packages list and reference the address of the package with the address dictionary to get the assigned address number.
3. Using the distance CSV, find the distance to the delivery, and if it is lower than the current value of the closest delivery: replace the closest delivery value, the location of the package, and the package’s assigned number.

The space-time complexity of the nearest neighbor algorithm is O(N) because the function iterates through a list to find the lowest distance value. While the best-case scenario of the runtime could be O(1), it is not realistic in this scenario unless there are no packages on the truck.

Pseudocode (Also provided in Pseudocode file)

A picture containing text, screenshot, font

Description automatically generated

B2: Programming Environment

This program was built using the high-level programming language Python 3.7. The program was created using the IDE: PyCharm 2022.1.2 (Community Edition) running on Windows 10 Home Edition. The PC is comprised of an Intel Core i7 10750h with 16 GB of DDR4 ram.

B3: Space-Time and Big O

Importing the CSV files, all have a time complexity of **O(N)** and a space complexity of O(N), where N is the number of rows to read through. The code simply parses the CSV files and adds information to the correct lists.

A screen shot of a computer program

Description automatically generated with low confidence

The initialization of the truck class has a space and time complexity of **O(N)** as it searches through the hashtable for every package and updates its status as “En Route.”.

def unload(self, packageID):

The “unload” truck function has a time complexity of O(N) as it iterates through all the packages in its list until it finds one with the correct id and removes it.

def distance\_in\_between(current, address):

The “distance\_in\_between” function has a space-time complexity of O(1).

def minDistanceFrom(currentLocation, remainingPackagesList):

The “minDistanceFrom” function uses the nearest neighbor algorithm. Since it iterates through the remainingPackagesList, which can have (N) number of packages, it has a time complexity of O(N).

def truckDeliverPackages(truck):

The “truckDeliverPackages” function has a space-time complexity of O(N^2). This function contains the function “minDistanceFrom” with a time complexity of O(N). The “minDistanceFrom” function is nested within a while loop which checks if there are any remaining packages. Therefore the overall space-time complexity is O(N^2).

def loadPackages(truck):

The “loadPackages” function has a space-time complexity of O(N) as it iterates through the packages on the truck, and the truck can have a variable number of packages (represented by N).

def validateTime(package, timeStamp):

The “validateTime” function has a space-time complexity of O(1) as it uses the given values and does not iterate through any lists

def loadTrucks():

The loadTrucks function has a space-time complexity of O(N^2) as it inherits the time complexity of the function “truckDeliverPacakges.”

class Main:

The “Main” class has a time complexity of O(N^2) as it uses the function “loadTrucks”. The GUI and text sections of the class have a time complexity of O(1) as it displays text.

Overall the space-time complexity for the entire program is O(N^2). For a more in-depth breakdown of the time complexity of each function, please refer to the annotations in the main.py file.

Section B4 Adaptability

The program would not scale exceedingly well as while the algorithm and the hashtable (data structure) are self-adjusting, a space-time complexity of O(N^2) means that the program would be slow at scaling up.

Additionally, as the packages are hard-coded onto the trucks, any changes to the trucks would require a modification directly to the code. This would be incredibly cumbersome and is not manageable if the number of packages or the conditions of the packages change.

Section B5: Software Efficiency and Maintainability

The overall program has a Big-O time complexity of O(N^2), which means the program is running in polynomial time. It is not the most optimal solution overall as it only gets the optimal next location from its current location.

The maintainability of the code is sub-par. Some abstraction would certainly make the code much more manageable. The various functions could be split into separate .py files, but in its current format, most functions are located simply in the main.py file.

B6: SELF-ADJUSTING DATA STRUCTURES

The hashtable is a self-adjusting data structure.

Strengths:

* Simple to search/insert

Weaknesses:

* A possible downside of collisions if packages are entered with the same package ID
* CSV data is not formatted correctly

Section C

All code is original and runs without errors or warnings thanks to try/catch blocks. The code for the Data Structure (Hashmap) was provided by instructors. However, it has been altered slightly for more straightforward use

C1: IDENTIFICATION INFORMATION

A comment with the relevant information has been provided.

C2: PROCESS AND FLOW COMMENTS

Comments have been included for each major “block” of code.

D: DATA STRUCTURE

The hash table has an initialization method to create itself. It also has methods to insert an item, search for a specific item or remove an item from the data structure. Thanks to its key, the hash function allows access straight to the item without iterating through all of the elements. The hash table is more efficient than a linear search as the time to complete operations remains constant regardless of the number of items in the table.

E: HASH TABLE

The program uses a hash table with an insertion function.

F: LOOK-UP FUNCTION

The program uses a hash table with a look-up function.

G: INTERFACE

The program provides an interface that reports the status of the package, its destination, ID, etc.

G1: Status Check at 8:35 a.m.

A screen shot of a computer

Description automatically generated with medium confidence

G2: Status Check at 9:35 a.m.

A screen shot of a computer

Description automatically generated with medium confidence

G3: Status Check at 1:11 p.m.

A screen shot of a computer screen

Description automatically generated with medium confidence

H: SCREENSHOTS OF CODE EXECUTION

A screenshot of a computer

Description automatically generated with medium confidence

A screen shot of a computer

Description automatically generated with medium confidence

I1: STRENGTHS OF THE CHOSEN ALGORITHM

The chosen “nearest neighbor” algorithm was selected primarily due to its simplicity. The implementation of this algorithm is straightforward to read, use and modify. The algorithm is also incredibly adaptable; if the proper information is provided, it can handle an increasing amount of packages to deliver.

I2: VERIFICATION OF ALGORITHM

The total number of combined miles between all trucks is 122 miles. This fulfills the requirement of being below 140 miles.

All packages are delivered before the 5 p.m. cut-off and follow the delivery specifications. This can be confirmed by using the “Status for all Packages and Total Mileage for all deliveries” option in the program, which provides the package’s delivery time and any special requirements they may have had.

All of the following can be confirmed using the “Status for all Packages and Total Mileage for all deliveries.”

I3: OTHER POSSIBLE ALGORITHMS

The program could also have used the branch and bound algorithm or Dijkstra’s algorithm to solve its problems. Neither of these solutions was chosen as the “nearest neighbor” algorithm had a higher level of simplicity.

I3A: ALGORITHM DIFFERENCES

1. Branch and bound algorithm

* Creates multiple paths and selects the ideal one
* Uses lower bounds to stop considering unideal branches

1. Dijkstra’s algorithm
   * Complex to maintain
   * Less accessible or manageable than the nearest neighbor algorithm

J: DIFFERENT APPROACH

If the packages also had to go to various cities/states, the program should be refactored around these new conditions. There is a very strong probability of having matching addresses across different cities.

K1: VERIFICATION OF DATA STRUCTURE

A screen shot of a computer

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

K1A: EFFICIENCY

Adding a package does not affect the hash table efficiency when looking up package information. Since the package is indexed, we do not have to iterate through every element in the table.

K1B: Implications

Adding additional trucks would not have any implications on the hash table itself. The operation to add information to the package regarding its truck is done in O(N) constant time. Therefore, this would not affect hashtable performance.

K2: OTHER DATA STRUCTURES

A queue or a stack would be possible options for alternative data structures.

K2A: DATA STRUCTURES DIFFERENCES

Queue: A queue would be a great alternative as we could sort the nodes by their priority. We’d only have to sort packages by their location once to find the optimal route and then set up the queue to store their locations in an ordered fashion.

Stacks: A stack is another alternative that could be used similarly to the queue, where once the nodes are sorted, they could be stored using a stack based on the priority of the delivery.

L: Sources

Ferdinand, R. (2023, March 28). Data Structures And Algorithms Support [Live Meeting]. WebEx@WGU University. https://wgu.webex.com/meet/robert.ferdinand