**A: Algorithm Identification**

This program utilizes a “nearest neighbor” algorithm (“nearest\_neighbor” in main.py). The algorithm is initialized with the hub as the starting location, 0.0 mileage, and a starting time per truck. The first package of each truck is initialized as the closest package and as the algorithm runs, it determines which, if any, package is closer than the current package. When the closest package is found and delivered, the mileage and delivery time is updated.

**B1: Logic Comments**

The algorithm runs as followed:

* *Initialize ‘current\_address’ to WGUPS hub address*
* *Initialize ‘total\_mileage’ to the ‘mileage’ parameter of the function*
* *Initialize ‘delivery\_time’ to the ‘start’ parameter of the function*
* *Iterate through each package in the truck*
  + *Set ‘package’ to each package object using the hash lookup function*
  + *Set ‘time\_left\_hub’ of each package to ‘start’ parameter*
* *Iterate through the truck list while the list has packages in it*
  + *Initialize the first package in the list to ‘start\_package’*
  + *Initialize ‘closest\_distance’ to the distance between the hub and ‘start\_package’*
  + *Initialize ‘closest\_package\_id’ to the first package in the truck list*
    - *Iterate through each item in the truck*
      * *Set ‘next\_package’ to the next item in the list*
      * *Set ‘distance’ to the float value between ‘current\_address’ and ‘next\_package’*
      * *If ‘distance’ is less than ‘closet\_distance’*
        + *Set ‘closest\_distance’ to ‘distance’*
        + *Set ‘closest\_package\_id’ to the current item in the list*
      * *Initialize ‘nearest\_package’ to ‘closest\_package\_id by the hash lookup*
      * *Set ‘current\_address’ to ‘nearest\_package’ address*
      * *Update ‘total\_mileage’ by adding ‘closest\_distance’ to existing ‘total\_mileage’*
      * *Update ‘delivery\_time’ by calculating time using ‘closest\_distance’ / 18*
      * *Update ‘delivery\_time’ of ‘nearest\_package’ object by setting to ‘delivery\_time’*
      * *Remove package from truck list using remove hash function*
    - *Return ‘total\_mileage’*

**B2: Development Environment**

This program utilizes the Python programming language. The code was written using PyCharm Community Edition 2023.1 and Python 3.11.3. Windows 10 was the operating system used on the machine to write the program.

**B3: Space-Time and Big-O**

In main.py, lines 17 and 50 have the big-o notation for each respective function. Distance.py big-o notation is on lines 12 and 20. In hash\_table.py, lines 4, 10, 24, and 33 have the big-o notation for each respective function. Lines 4 and 17 of package.py have the big-o notation for the functions.

**B4: Scalability and Adaptability**

The nearest neighbor algorithm and the hash-table defined in this program have great scalability and adaptability potential. Both functions are written to iterate through a list of objects, no matter the size of the list. However, this specific program has a shortcoming that will not allow it to scale well. The packages on each truck were pre-defined and it only allowed for 40 packages. If more packages were to be added, they would have to be manually added to each truck list.

**B5: Software Efficiency and Maintainability**

The efficiency of the program is O(n^3). The program isn’t too complex and it’s easily maintainable if another developer were to add to it. All the variables are appropriately named as to easily discern what their purpose is, the comments are clear and appropriately placed, and all code is written to standard format.

**B6: Self-Adjusting Data Structures**

The chaining hash table ensures that there is enough space for all packages while handling collisions well. If more packages are added, the linked list grows according to how many packages are added with no consequences. Inserting, updating, and removing packages are very easily implemented. The downside of the chaining has is that it’s essentially a list of lists. As the list grows, so does that time complexity of the program.

**C: Original Code**

The code is my own original program, and it runs without errors. Please see the screenshot included in section H of this file to see it ran without warnings or errors.

**C1: Identification Information**

Please see lines 1-3 in main.py.

**C2: Process and Flow Comments**

Comments have been added to all major blocks of code. Please see main.py, truck.py, distance.py, hash\_table.py, and package.py for comments specific to each block of code.

**D: Data Structure**

The self-adjusting data structure used in this program is a chaining hash-table. The hash-table can be observed in “hash\_table.py”. It uses the chaining method to combat collisions and to easily insert, retrieve, and remove data.

**D1: Explanation of Data Structure**

The hash table class is responsible for initiating the hash table. It uses a passed integer to determine the initial size of the table. The class also contains the following methods: insert, lookup, and remove. Each of these 3 methods utilize the key (package ID) to perform their respective function. Using the package ID as the key, the hash allows storage of package data such as the delivery address, delivery deadline, the time it left the hub, and the time the package was delivered.

**E: Hash Table**

Please see hash\_table.py to view the insert function.

**F: Look-Up Function**

Please see hash\_table.py to view the lookup function.

**G: Interface**

A simple console interface was created to navigate the program. Please see main.py to view the code or run the program to view the interface.

**G1: First Status Check**A screen shot of a computer

Description automatically generated with medium confidence

**G2: Second Status Check**

A screenshot of a computer program

Description automatically generated with medium confidence

**G3: Third Status Check**A screen shot of a computer program

Description automatically generated with low confidence

**H: Screenshots of code execution**

A screenshot of a computer

Description automatically generated with medium confidence

**I1: Strengths of the chosen Algorithm**

The Nearest Neighbor algorithm is very simple to understand, and it is easily scalable. The algorithm itself is quite easy to understand because it’s a very intuitive process. You have a list of packages and the only function it needs to perform is to decide what the next closest package is. Once that is determined, the rest is just updating existing values. The algorithm is also very scalable, which makes it very desirable for many applications. It iterates through a list of items, and it does not matter how many items are in that list. It will keep going until there are no more items left.

**I2: Verification of Algorithm**

The algorithm of this program delivers all 40 packages in 108.6 miles, which is observed as soon as the user runs the program. All packages were also delivered on time and to the correct locations as specified in the package details. Through the user interface, all delivery status can be observed as well as the total mileage. Selecting option 1 and entering a time will print all package information for that time. Selecting option 2, then entering a time and a package ID will print the package information for the selected package.

**I3: Other Possible Algorithms**

Two alternative algorithms that could have been used include Dijkstra’s shortest path or a depths-first search. Both algorithms would have satisfied the requirements of this project and delivered all packages in a timely and efficient manner.

**I3A: Algorithm Differences**

Both Dijkstra’s algorithm and a depths-first search would have resulted in a more efficient delivery route, but both would be harder to implement due to their complexity. Greater complexity of an application requires higher processing power of a machine, which increases initial costs.

**J: Different Approach**

If I were to do this project again, knowing what I know now, I would add 1 more algorithm to make the program more scalable. In this program, each truck was “manually loaded”, meaning I chose outside of the program which package would go on which truck. To improve the program, I would create an algorithm that reads the information from the csv files and sorts them into each truck according to delivery deadlines and/or special notes.

**K1: Verification of Data Structure**

The algorithm of this program delivers all 40 packages in 108.6 miles, which is observed as soon as the user runs the program. All packages were also delivered on time and to the correct locations as specified in the package details. Through the user interface, all delivery status can be observed as well as the total mileage. Selecting option 1 and entering a time will print all package information for that time. Selecting option 2, then entering a time and a package ID will print the package information for the selected package. The hash-table with the look-up function are present in the program and can be viewed in “hash\_table.py”.

**K1A: Efficiency**

The chaining hash table data structure is very efficient for key-value pair data management. However, the more data the hash table stores, the time complexity would increase. The look-up function of the hash table has a time complexity of O(n) and if packages are added, the time would be affected by a factor of ‘n’.

**K1B: Overhead**

The great thing about the chaining hash table is that it does not affect the structure space usage. In this program, the hash table is initialized with 10 spaces for the data. When more than 10 pieces of data is added, more spaces are not needed because the new data is chained to whatever other piece of data is currently in the allotted slot.

**K1C: Implications**

Adding more trucks and cities would slow down the program depending on how much data was added. The lookup function would have to iterate through more data, thus slowing down the program by a factor of ‘n’. However, since were utilizing a chaining has function, no more space would be required to add more data to the program.

**K2: Other Data Structures**

Two other data structures that could have been used would have been a graph and a Binary Search Tree. Both data structures would have worked within the provided constraints of the program.

**K2A: Data Structure Differences**

A graph would have been a good data structure to implement if there was a need for larger scalability. With graphs, multiple data points could be grouped in adjacent vertices. Binary search tree would have made it easier to get a specific data pint because the data would have been pre-sorted.

**L: Sources**

The only source used for this project was “C950 – Webinar-1 – Let’s Go Hashing – Recording” from the provided webinars. That source is observable in “hash\_table.py”.

# 