Matthew Shelbourn
C950: Data Structures & Algorithms II
Student ID: #001059665
WGUPS Delivery Manager: Program Overview and Analysis

General Program Overview

The guidelines of this assignment are to utilize 3 trucks and 2 drivers to deliver 40 packages with the main constraint that the total mileage for all trucks has to be under 140 miles. In addition, there are several other constraints that need to be accounted for, including: package delivery deadlines, package delivery dependencies (a package must be on the same truck as another package), package delays, and incorrect addresses. Without these additional constraints, this challenge could easily be tackled using an application of a nearest neighbor algorithm. Instead, I used the nearest neighbor (greedy) algorithm as the basis for my solution, but I added in extra logic to meet the requirements of the additional constraints.

Overall, I am mostly content with the outcome of my algorithm and program. The total distance travelled for all trucks using my algorithm was 104 miles, which is well under the 140 mile maximum. There are areas for improvement, which I may work on in the future purely for my own learning. One area for improvement that sticks out is my handling of the packages that have delivery deadlines. For my algorithm, I pushed these packages to the front of the delivery queue for each truck that they were on, then optimized them based on distance to their next destination. This works fairly well, but I know that I could shave at least a few more miles off if I used both the elapsed time for a truck on its delivery route and the miles to the next destination. If I factored in both time and mileage then I could ensure that packages with delivery deadlines were delivered on time and also delivered optimally based on mileage given the constraints.

In summary, the program works quite efficiently, although there is room for improvement. All packages are delivered within their constraints (if they have any) and the total mileage for all trucks to execute their routes was 104 miles. The entire program has a linear space complexity **O(N)** and a polynomial time complexity **$O(N^2)$**. Most of the program's components operate with a linear space-time complexity, with the notable exceptions of the truck load optimization algorithms, which have linear time complexities due to them each having a while-loop with a nested for-loop.

A: Algorithm Identification

The type of function that I use for routing optimization in my program is an adaptation of the K-Nearest Neighbor Algorithm (KNN), which is a type of "Greedy" algorithm. Although, since we are given fixed distances between all destinations there is no need to calculate the Euclidean distances between each location, which makes it much more straightforward. My optimization algorithm is considered to be "Greedy" because it doesn't necessarily care about the overall outcome of its execution. It iterates through a list of given packages and optimizes for mileage at each iteration. Since it optimizes at each iteration, it provides a very efficient (optimized) result, but it doesn't optimize for the entire outcome, just each component on a step-by-step basis.

I have three different versions of my optimization algorithm (one customized for each truckload to account for different constraints) and you may find them in my program in ***truckloads.py*** on lines 97-191. The three specific algorithms are ***get_truck_1, get_truck_2,*** and***, get_truck_3.*** They each take no parameters and have a linear space complexity of **O(N)** and polynomial time complexity of **O(N^2).** The space complexity is due to the fact that the space required for the algorithm to run is limited by the number of elements in each truck load array (***truck_1, truck_2, truck_3***). Even though each algorithm has a for-loop nested in a while-loop, the number variables in the local memory and the output values of the algorithm remain fixed to the number of elements that are in each truck load array. On the other hand, the time complexity is due to the fact that each algorithm has a nested for-loop in a while-loop. Therefore, as the size of each truck load array increases (N), the number of operations performed on each element in the array may grow polynomially, in the worst case scenario (N^2).

I decided to optimize each truck's load before the truck departed the hub because I believe this to be more similar to how logistics companies operate in the real world. More often than not, a truck would have all of it's deliveries set for the day before the truck even leaves the hub and would not optimize its load as it is delivering on its route. The optimization ends up being the same, it's just that I optimize each route on the front end and not while it is already under way.

The pseudocode for each of the truck's optimization algorithms is below:

Truck 1 Optimization Algorithm

def** get_truck_1():* ***# truckloads.py -- lines 97-128:

1. **Parameters: none**

2. **Set the local variables**

   truck_1: the unoptimized truckload for truck 1 that has been created by another function (**get_truck_loads** on line 13). Packages are added to truck 1 based on necessity (delivery deadlines, package dependencies, etc)

   truck_1_optimized: initializes an empty list to store the optimized truck load

   previous_package: initializes an empty string to store the previous package that was added to **truck_1_optimized**. The next package to be delivered will be determined by comparing its address and how far it is from the address of **previous_package**. The package that has a destination address that is the closest will be appended to the **truck_1_optimized** list and will then be assigned to **previous_package**

   shortest_distance: variable to store the shortest distance, which will be used when calculating the package that has the address closest to the address of **previous_package**.

shortest_distance is initialized with a value of 15 since the farthest distance between any two destinations is 14.2 miles. I rounded up to 15 because everyone likes whole numbers.

truck_1_optimized_current_index: variable to store the current last index of the **truck_1_optimized** list. As the algorithm steps through its iterations, the package with the closest destination address for a given iteration will be inserted into the **truck_1_optimized** list at this index.

3. **First for-loop**

Truck 1 contains most of the packages that have delivery deadlines and package delivery dependencies (must be delivered with other packages). All but one package with a delivery deadline must be delivered by 10:30 am. The exception is one package must be delivered by 9:00 am. The package that must be delivered by 9:00 am (package ID: 15) is on truck 1 and I decided to have it delivered first just so I don't run the risk of missing the delivery deadline. (Note: I'm pretty sure that I could have saved some mileage by not delivering this package first and instead running an optimization algorithm that also accounted for delivery deadlines as well as mileage).

This for-loop iterates through all elements of the **truck_1** list and searches for the package that has a delivery deadline of 9:00am and then appends that package to the **truck_1_optimized** list at index 0. Since the **truck_1** list will be used throughout the entire algorithm, once an element from it is added to the **truck_1_optimized** list it is deleted from the **truck_1** list to prevent duplicate packages in the **truck_1_optimized** list.

```
for every element (package) in the truck_1 list:
    if the string '9' is found in the element at index 5 (delivery deadline column):
        - append package to the truck_1_optimized list (index 0)
        - remove the package from the truck_1 list
        - assign the previous_package temporary variable with package
        - increment truck_1_optimized_current_index by one
```

4. **While-loop with nested for-loop**

In this while-loop is where the actual optimization of the truck's route happens. Before the while-loop is actually entered, the length of the **truck_1** list is checked and if it is equal to zero then the while-loop never executes. This is because when the length of the **truck_1 list** is zero then that means there are no remaining packages that need to be optimized.

For truck 1, each package except the package that must be delivered by 9:00 am (package ID: 15) is sorted in order based on the distance of its destination from the previous truck location (address of **previous_package**). The for-loop nested in the outer while-loop iterates through the packages that need to be optimized (remaining packages in the **truck_1** list) and calculates the distance between each package's destination address and the previous location's and then compares that value to the current **shortest_distance** value, which starts at 15 miles. If a distance value for a package is calculated to be less than the current value for **shortest_distance** then that calculated distance becomes the new value for **shortest_distance**. At the end of the for-loop the package that has the closest destination address to the address of the **previous_package** becomes the next package to be delivered. It is inserted into the **truck_1_optimized** list at the **truck_1_optimized_current_index**, removed from the **truck_1** list (so that it is not included in the next for-loop iteration), the **truck_1_optimized_current_index** is incremented by 1, and **shortest_distance** is reassigned with a value of 15.

```
while the length of the truck_1 list is not equal to zero:
    for every element (package) in truck_1 list:
        if the distance between the address of previous_package and the current
        package is less than the current value of shortest_distance:
            try:
                remove the package in the truck_1_optimized list at the
                truck_1_optimized_current_index (if an element exists at
                this index)
            except IndexError:
                if there is an IndexError (in this case that would mean that
                there is no element [package] in the truck_1_optimized list
                at the truck_1_optimized_current_index) then skip this
                operation and move on to the next statement in the “if” block.
            - insert the current package in the truck_1_optimized_list at the
            truck_1_optimized_current_index
            - set the value of shortest_distance to be the distance calculated
            between the address of previous_package and the address of the
            current package
    - assign the value of previous_package with the package that was just inserted
    into truck_1_optimized
    - remove the package that was just added to truck_1_optimized from the
    truck_1 list
    - increment the truck_1_optimized_current_index by 1
    - reassign a value of 15 to the shortest_distance temporary variable
```

5. **Return statement**

At the end of the function, the **truck_1_optimized** list is returned and ready to be used in the **exec_routes** function (**routes.py**).

```
return truck_1_optimized
```

Truck 2 Optimization Algorithm

def** get_truck_2():* ***# truckloads.py -- lines 131-163:

The optimization function for truck 2 is almost identical to the function for truck 1, with the minor exception that instead of prepending the optimized package list with the package that must be delivered by 9:00 am, it prepends the optimized package list with all packages that have delivery deadlines and then appends the list with the packages that must be delivered by end of day (EOD). Therefore, the only part of the code that is different is the for-loop at the beginning of the function.

```
for every package in the truck_2 list:
    if the string ‘EOD’ is not found (package must have a delivery deadline) in the
    element (package) at index 5 (delivery deadline column):
        insert the element into the truck_2_pre_sort list at index 0 (if there is
        already an element at index zero then all elements will be shifted to the
        right [their indices will be incremented by 1])
    else (if ‘EOD’ is found in the element at index 5):
        append the element to the truck_2_pre_sort list (order doesn’t matter
        right now, we just want to ensure that all packages with delivery deadlines
        are at the front of the truck_2_pre_sort list)
```

Truck 3 Optimization Algorithm

def** get_truck_3():* ***# truckloads.py -- lines 166-191:

The optimization algorithm for truck 3 is the most simplistic and straightforward out of the three optimization algorithms. This is due to the fact that truck 3 is not responsible for delivering any packages that have delivery constraints, deadlines, or dependencies. All of the packages that have such constraints will be delivered by trucks 1 and 2. Therefore, the optimization algorithm for truck 3 simply takes the portion of the code from **get_truck_1** and **get_truck_2** that includes the while-loop with the nested for-loop and iterates over the packages in the **truck_3** list and adds them in delivery order based on distance from the previous package’s address to the next package’s address.

B1: Logic Comments

Throughout the application I have included in-line comments for each of the major components, as well as the function declarations. Additionally, I have included in-line comments for some of the more-important variables that are used in the application.

I have included a detailed, step-by-step walkthrough of the logic for the main optimization algorithms used in my program and pseudocode versions of these algorithms in section A above.

B2: Development Environment

To develop my program, which I call *WGUPS Delivery Manager*, I used the following:

Software:

- PyCharm IDE (v2020.3.1 Community Edition)
- Python v3.9
- macOS Big Sur (v11.1)
- Github with PyCharm's integrated Git functionality was used for version control

Hardware:

- MacBook Pro (16-inch, 2019) - 2.6 Ghz 6-Core Intel Core i7 - 16 GB 2667 MHz DDR4 Memory
- Local development environment was used to run, test, and debug my program. No external server was used and therefore neither bandwidth, processing capacity, nor storage capacity were of concern

B3: Space-Time and Big-O

I have included space-time and Big-O analysis as in-line comments for each function in the code of my application. In addition, below you will find a breakdown of the space-time complexity and Big-O analysis for each function and the entire program as a whole.

> *Entire Program:* The majority of my application runs with a linear space-time complexity **O(N)**. Most of the functions and methods utilize at most one for-loop and work with a predefined set of variables. If any of these variables are lists, then the lists are of a fixed length and do not grow in size. Therefore the space-time complexity remains linear.
>
> Due to the fact that my optimization algorithms (**get_truck_1, get_truck_2,** and **get_truck_3**) utilize a while-loop with a nested for-loop, they have an polynomial time complexity **O(N^2)**, which makes the time complexity of my entire application polynomial **O(N^2)**. In a worst case scenario, the optimization functions will need to iterate through each of the packages and during each iteration, iterate through the list of packages again. This justifies the polynomial time complexity. The optimization algorithms have a linear space complexity since the lists that they iterate through are of fixed length and do not grow polynomially as the function executes. Since the space required for each of the optimization algorithms remains linear, each has a space complexity of **O(N)**.
>
> To summarize, the space complexity of my entire program is linear **O(N)** and the time complexity of my entire program is polynomial **O(N^2)**.

def main() - main.py -- lines 12-142: Linear space-time complexity **O(N)**

def __init__(self) - hash_table.py -- lines 13-18: Linear space-time complexity **O(N)**
def _get_hash(self, key) - hash_table.py -- lines 20-23: Constant space-time complexity **O(1)**

def create(self, key, val) - hash_table.py -- lines 27-45: Linear space-time complexity **O(N)**

def read(self, key) - hash_table.py -- lines 47-62: Linear space-time complexity **O(N)**

def update(self, key, val) - hash_table.py -- lines 64-80: Linear space-time complexity **O(N)**

def delete(self, key) - hash_table.py -- lines 82-98: Linear space-time complexity **O(N)**

def get_package_table - package_table.py -- lines 10-61: Linear space complexity **O(N)**, Polynomial time complexity **O(N^2)**

def get_truck_loads - truckloads.py -- lines 8-94: Linear space-time complexity **O(N)**

def get_truck_1 - truckloads.py -- lines 97-128: Linear space complexity **O(N)**, Polynomial time complexity **O(N^2)**

def get_truck_2 - truckloads.py -- lines 131-163: Linear space complexity **O(N)**, Polynomial time complexity **O(N^2)**

def get_truck_3 - truckloads.py -- lines 166-191: Linear space complexity **O(N)**, Polynomial time complexity **O(N^2)**

def exec_truck_routes - routes.py -- lines 13-144: Linear space-time complexity **O(N)**

def get_truck_departure_arrival_times - routes.py -- lines 147-156: Constant space-time complexity **O(1)**

def get_total_transit_time - routes.py -- lines 159-166: Constant space-time complexity **O(1)**

def get_total_transit_time_truck_1 - routes.py -- lines 169-176: Constant space-time complexity **O(1)**

def get_total_transit_time_truck_2 - routes.py -- lines 179-186: Constant space-time complexity **O(1)**

def get_total_transit_time_truck_2 - routes.py -- lines 189-196: Constant space-time complexity **O(1)**

def get_total_mileage - routes.py -- lines 199-202: Constant space-time complexity **O(1)**

def get_total_mileage_truck_1 - routes.py -- lines 205-208: Constant space-time complexity **O(1)**
def get_total_mileage_truck_2 - routes.py -- lines 211-214: Constant space-time complexity **O(1)**

def get_total_mileage_truck_3 - routes.py -- lines 217-220: Constant space-time complexity **O(1)**

def get_package_statuses(time_param) - package_statuses.py -- lines 12-150: Linear space-time complexity **O(N)**

def get_package_status(package_id, time_param) - package_statuses.py -- lines 153-292: Linear space-time complexity **O(N)**

distances.py -- lines 6-10: Linear space-time complexity **O(N)**

distances.py -- lines 12-16: Linear space-time complexity **O(N)**

def get_distances - distances.py -- lines 19-22: Constant space-time complexity **O(1)**

def get_addresses - distances.py -- lines 25-28: Constant space-time complexity **O(1)**

def calc_distance(add_1, add_2) - distances.py -- lines 31-51: Constant space-time complexity **O(1)**

def get_dest_name(package) - distances.py -- lines 54-61: Constant space-time complexity **O(1)**

def calc_dest_transit_time(distance) - durations.py - lines 7-13: Constant space-time complexity **O(1)**

def calc_delivery_time(current_time, transit_minutes) - durations.py -- lines 16-21: Constant space-time complexity **O(1)**

B4: Scalability and Adaptability

I designed this application to be fairly scalable, adaptable, and extensible. The program is composed of several modular components, which makes scaling and maintaining the program quite easy. If one part of the program needs to be changed then that change can be implemented in the appropriate components, without having to alter the entire program. Also, if errors arise with the program then it should be straightforward to diagnose the problem/problems and determine which component/components need to be looked at and modified.

My entire program is capable of scaling quite well. The hash table, optimization algorithms, all of the helper functions, and the interface are able to handle a greater number of inputs (packages, trucks), however there is one significant caveat. The capacity of each truck (16 packages) does hinder the scalability of the application as it stands now. Currently, the application is capable of easily handling 48 packages (3 truck loads), but if the truck capacity remained the same and more than 48 packages were fed into the application then there would be more packages than could fit on three trucks. Additional trucks would need to be added in order to handle the increased number of packages. Also, if the packages that were added had delivery deadlines or other dependencies then the package sorting algorithms would need to be refactored or duplicated to handle additional truck loads.

Besides the complexity of adding packages, every other component of the application with the exception of the package sorting algorithms are capable of handling an infinite number of packages. However, my application does not currently use a hash table that can dynamically adjust its maximum size so efficiency would drop with an increased number of inputs. But to offset this, the maximum size (buckets) of the hash table could be manually adjusted if the program was to be used with more than 40 packages. Currently, the program sets the max size of the hash table to be 10, but that can be adjusted easily in **hash_table.py**.

B5: Software Efficiency and Maintainability

I am a software developer professionally and understand the importance of writing code that is easy to interpret and maintain. As a result, I wrote this entire program with comprehension and maintainability in mind. I used modular components to make refactoring and debugging much easier than if I used a one or two big functions. I included in-line comments for each of the function declarations explaining what the function does to make comprehension of the code much easier. Also, I used readable, self-explanatory variable and function names so that the user/reader doesn’t have to guess what a variable or function does/is for.

The modular nature of my application makes it fairly easy to perform code adjustments when necessary. I could have lumped several functions into one or two big functions and put them into one or two files, but I instead used a “separation of concerns” approach to break each function down into its smallest components, made those components a function of their own, and grouped all similar functions together and included them in their own file. To me, this makes sense and it will hopefully make sense to anyone reviewing the application.

B6: Self-Adjusting Data Structures

As I previously mentioned, the hash table data structure I used for my application (hash table found in **hash_table.py**) is not completely self-adjusting. It is partially self-adjusting in that each bucket contains an empty list that can grow or shrink in length using the **create** or **delete** functions found in **hash_table.py**. This means that it can accommodate any number of inputs, but the size of the hash table is fixed and will not self-adjust to optimize for efficiency. For 40 packages, a fixed size of 10 buckets in the hash table makes perfect sense and works efficiently. However, if the number of packages was dynamic and not fixed then the efficiency of my hash table could drop dramatically. If there were, say 100 packages, or even 1000 packages, then a hash table with a size of 10 would be inefficient, to say the least. In a worst case scenario based on my hash function, the hash table could end up with 1000 elements in one bucket, making it highly inefficient.

If I wanted to maximize scalability and efficiency for my program, I could have designed a fully self-adjusting data structure to dynamically accommodate for greater or fewer inputs. Actually, it wouldn't be too much work to add a method to **hash_table.py** to recalculate the size of the hash table based on the number of inputs. If I implemented this then the hash table would be sized optimally regardless of how many packages were fed into the program, which would optimize the efficiency of all hash table Create-Read-Update-Delete (CRUD) functions.

The downside with using a dynamically adjusting (self-adjusting) data structure like a self-adjusting hash table in this way is that the algorithm used to recalculate the hash table size would most-likely have a polynomial space-time complexity **O(N^2)** which could hamper the efficiency of the program if it was being called often. It's a tradeoff and some thought would have to go into how to best code it so that the size recalculation algorithm wouldn't be called more frequently than necessary.

C: Original Code

All of the code found in this application is my own and nothing is plagiarized. I'm sure that there are many areas in my program where the code could be written more efficiently, but I never use code from other people without proper attribution and I used none in this application. Apart from it being unethical, I choose to write all the code myself because it is the best way for me to learn how to be a better programmer, especially when working with a language that I am unfamiliar with.

C1: Identification Information

I have included an identifying line comment on line #1 of each file, including **main.py**. This comments includes my full name, student ID number, school email address, and the month and year in which I worked on this project.

C2: Process and Flow Comments

As I previously stated, in-line comments have been included throughout this application to explain what chunks of code are doing. Primarily, I included line comments at the beginning of each function declaration to explain what the function does and also what its space-time complexity is.

D: Data Structure

The data structure that I designed and implemented for this application is a self-adjusting hash table. The code for the hash table is found in **hash_table.py**. This hash table initializes as a list of lists and builds itself out with a fixed number of "buckets" (nested lists). The number of nested lists may be changed by altering the value of **MAX** which is found on line 7 of **hash_table.py**. Once the hash table is initialized, elements are added to the buckets based on a hash function (**_get_hash_**) which is found on lines 19-22 of **hash_table.py**. For the purposes of this assignment, the **MAX** value was initialized with a value of 10 because 10 buckets are adequate for efficiently storing information for 40 packages. CRUD operations are performed on the hash table with four separate methods -- **create, read, update, and delete**. The aspect of this hash table that makes it self-adjusting is that each bucket is capable of storing multiple entries. This is made possible by the hash table's collision handling capabilities. This hash table uses "chaining" for collision handling. This means that if two or more hash table entries are calculated by the hash function to be stored in the same bucket then the entries will be "chained" together. This essentially means that this particular bucket will contain two or more nested lists, one for each element belonging to the bucket.

As I previously mentioned, I could have made this hash table even more capable of dynamically adjusting to the number of elements being stored in it by making it able to automatically recalculate its size. That wasn't necessary for this project since the number of packages being stored in the hash table is fixed. However, if I was using this program for a variable number of packages then I would refactor my hash table so that it could dynamically adjust in this way.

D1: Explanation of Data Structure

I provided an overview of how my hash table works in section D above, but now I will provide a more-detailed explanation. First, the hash table is initialized as a list with a length corresponding to the **MAX** value. In this case, the **MAX** value is 10. This value corresponds to the number of elements in the hash table (which is essentially a list of lists). So in this case my hash table will be a list with a last index value of 9. Next, an empty list ("bucket") is added at each index in the hash table. These "buckets" will store the hash table entries.

The hash function I used for the hash table is nothing overly complicated. It simply takes a key (package ID) and value (package details) as arguments, then calculates the bucket where the entry will be stored by finding the result of the key (package ID) modulo length of the hash table (10 in this case). So if the key (package ID) is 11 then the bucket where this entry will be stored is 11 % 10 which is 1. So the package with ID 11 will be stored in bucket 1 of the hash table.

This hash table uses four separate functions to perform Create-Read-Update-Delete (CRUD) functions. The **create** function adds an entry to the hash table. It takes a key (package ID) and value (package details) as arguments, calculates the "bucket" where the entry will be stored using the hash function, and inserts the entry into the appropriate bucket with the package's ID as the key. The **create** function checks to see whether there is an entry in the assigned bucket with an identical key and if there is then it updates the information for that entry (to handle the situation where a package's information has been updated and the user called the **create** function instead of the **update** function). This failsafe prevents duplicate entries in the hash table.

The **read** function takes a key (package ID) as an argument, calculates the "bucket" where the package should be stored using the hash function, searches the appropriate "bucket" for an entry with the given key (package ID). If a matching entry is found then the function returns the value. If an entry with the given key is not found in the appropriate “bucket” then an error is printed to the console, informing the user that a package with that ID was not found in the hash table.

The **update** function works much like the **read** function does except that it also takes a value (package details) as an argument. If a matching entry is found in the appropriate “bucket” then the value of the entry is updated with the value that is passed into the function as an argument. If an entry with the given key is not found in the appropriate “bucket” then an error is printed to the console, informing the user that a package with that ID was not found in the hash table.

The **delete** function also works much like the **read** function does except that it deletes a “bucket”entry if one is found with a key (package ID) that matches the key passed as an argument into the **delete** function. If an entry with the given key is not found in the appropriate “bucket” then an error is printed to the console, informing the user that a package with that ID was not found in the hash table.

E: Hash Table

The insertion function used for adding entries to my hash table is the **create** function found in **hash_table.py** on lines 26-44. I explained how this function works in section D1 above. I did not use any external libraries or frameworks to build the hash table or any of its associated methods, including the **create** function.

F: Look-Up Function

The look-up functions used for retrieving entries from the hash table is the **read** function found in **hash_table.py** on lines 46-61. I explained how this function works in section D! above. As with the other methods in my hash table, I did not use any external libraries or frameworks to design the **read** function.

G: Interface

I built what I consider to be a very intuitive interface for navigating the functionality of my application. My interface is descriptive and self-explanatory, and I have designed it to be as user-friendly as possible, requiring very little input from the user to view the desired information.

My interface consists of a main menu consisting of numbered options. The options are as follows: View total transit time of all trucks to complete their routes, View total mileage of all trucks to complete their routes, View the arrival and departures times for each truck to and from the Hub, View the total transit time for an individual truck to complete its route, View the total mileage for an individual truck to compete its route, View the package statuses for all packages at a given time (user provided), View the status of an individual package at a given time (user provided), Return to the main menu, and Exit the program.

One note about the options requiring time inputs by the user: I programmed the application to accept any time in 24-hour format (e.g. 11:33 or 14:02), but the program also accepts integer inputs such as 12 for 12:00 or 15 for 15:00.

G1 - G3: First - Third Status Checks

Please see the screenshots included in the "Program Screenshots" folder which is found in the project directory.

H: Screenshots of Code Execution

Please see the screenshots included in the "Program Screenshots" folder which is found in the project directory.

I1: Strengths of the Chosen Algorithm

There are a few strengths of using the "Greedy" algorithm that I wrote for optimizing each truck's delivery routes. Some of the strengths are more significant than others, but I will mention a couple of the more important advantages. First, the program is working with a fixed number of inputs (packages) with associated addresses that are a predefined distance from any other given destination address. Because of this, sorting the packages based on how close their destination addresses are to one another is fairly simple and does not require excessive compute power, storage, or calculation time, resulting in a linear space complexity **O(N)** and polynomial time complexity **O(N^2)**.

Secondly, using a K-Nearest Neighbor algorithm to optimize the truck routes for this project made perfect sense since many packages had the same destination address, which drastically improves the efficiency of a given truck's route and significantly reduces total mileage needed to complete all routes. Using anything other than a nearest neighbor algorithm may have resulted in a package being delivered at one time and then another package delivered at a different time but to the same address, which doesn't make logical sense. It is very rare that UPS will deliver a package to you in the morning and then return in the afternoon to deliver another package, unless the packages were shipped via different services (UPS Next Day Air, UPS Same Day, etc).

The last major advantage of using the type of optimization algorithm I used in my program is that we are tasked with optimizing for mileage, without needing to care about time unless it's regarding a package with a delivery deadline. The ancillary benefit of optimizing for total mileage is that mileage is tied to transit time so in making each truck's route as efficient as possible we are also make the route as time efficient as possible also. This came in handy for me especially because I was able to optimize the truck's routes to the point where most of the packages with delivery deadlines were delivered far earlier than their deadlines without even needing to prioritize their delivery first.

I2: Verification of Algorithm

You are able to verify that my algorithm meets the requirements for this project by viewing the screenshots of the program's output, which are in the "Program Screenshots" folder found in the project directory. Additionally, while running the program you may select option 2 to view the total mileage for all trucks to complete their routes and then select option 6 and enter a time of say 14:00 to see the statuses of all packages to verify that they were successfully delivered, when they were delivered, and if they made their delivery deadlines if they had one.

I3: Other Possible Algorithms

I chose an algorithm that I felt best fit the scenario, dataset, project requirements, and constraints. However, the challenge for this project is a variation of the Travelling Salesman Problem (TSP) and there are a plethora of different heuristic optimization algorithms that exist to tackle these types of problems.

One alternative is the Hill Climbing algorithm, which starts with an arbitrary solution and then makes small changes in each iteration. The result of each iteration is compared with the result of the previous iteration and if it is more optimal then it becomes the basis for comparison in the next iteration. The Hill Climbing algorithm has a polynomial space-time complexity **$O(N^2)$**.[1]

A second alternative to the Nearest Neighbor algorithm I used for optimization in my program is the Pairwise Exchange (2-Opt) algorithm. This algorithm starts with a base route and then removes two edges (two sets of connected destinations) in each iteration, replacing them with two different edges and comparing the total route distance to the previous total route distance. If the new route distance is more optimal than the previous route distance then the new route becomes the basis for comparison. This algorithm can run for a fixed number of iterations or until it's iterations do not return more optimal routes. The Pairwise Exchange algorithm has a polynomial space-time complexity **$O(N^2)$**.[2]

I3A: Algorithm Differences

I have already explained what the Hill Climbing and Pairwise Exchange (2-Opt) algorithms are in section I3 above. These two algorithms in addition to the Nearest Neighbor algorithm I chose to use for my program all attempt to achieve the same goal - to optimize the result based on a given set of inputs. For this project our task was to optimize truck routes for mileage. All three algorithms are capable of achieving this, however they go about it in different ways. The Nearest Neighbor algorithm is a type of Greedy algorithm because it doesn't care about the end result of its execution. It only attempts to optimize at each iteration, in this case finding the next destination with the shortest distance from the current location. On the other hand, both the Hill Climbing and Pairwise Exchange algorithms care

[1] Hill climbing. (2020, December 21). Retrieved January 01, 2021, from https://en.wikipedia.org/wiki/Hill_climbing

[2] Travelling salesman problem. (2020, December 31). Retrieved January 01, 2021, from https://en.wikipedia.org/wiki/Travelling_salesman_problem

about the entire result of the algorithm's execution. They both begin with a base solution and then optimize the entire solution in each iteration. If the solution resulting from a given iteration is more optimal than the previous solution then it becomes the base case for the next iteration. Both the Hill Climbing and Pairwise Exchange algorithms each use somewhat of a trial and error approach. Small changes are made in each iteration and if the outcome is more appealing then small changes are made to that outcome and so on. Conversely, the Nearest Neighbor algorithm is more refined in its approach. There is only one destination that is the closest to the current location and that becomes the next destination. For this reason, I decided that the Nearest Neighbor algorithm was the best option for the purposes of this project. We were given a fixed set of data and constraints to work with and the Nearest Neighbor algorithm made it easier to control the result of the algorithm's execution given a set of constraints.

Speaking of constraints, since both the Hill Climbing and Pairwise Exchange algorithms aim to optimize the result of the entire input set, it would be much more difficult to factor in things like delivery deadlines and package dependencies. This is another reason why I opted for the Nearest Neighbor algorithm.

J: Different Approach

I have already mentioned it elsewhere in this report, but one aspect of my program that I would change is that I would refactor my optimization algorithms to account for both mileage and time, instead of just mileage. As they are now, they deliver the packages with delivery deadlines first, regardless of how far their destination addresses are from each other of the hub. It is quite possible that on the way to a time-constrained package's destination there are one or more destinations that are passed by. If that were the case then the trucks could deliver these packages on the way to the destination for the time-constrained package(s) and still deliver them by the deadline.

There are a few ways that I could implement this change and I would have to do some testing to determine which option performs the best, but one method would be to run the current optimization algorithm on a subset of the packages on each truck that include delivery deadlines. Next I would need to calculate the total transit time required to deliver each of the packages with delivery deadlines. Once I had this information I could start to execute that main optimization algorithm, but not force a package to be delivered first and instead let the algorithm choose based on its calculations. In each iteration of the algorithm it would need to calculate the transit time from the next chosen destination to the destination of the first package in the subset of packages with delivery deadlines. If the transit time plus the total time required to deliver all the packages with deadlines pushed the current time past any package's delivery deadline, then the truck would immediately begin to deliver the packages in the subset with delivery deadlines. If not, then the next chosen package would be added to the optimized package list and the algorithm would move on to the next iteration.

There is no guarantee that this would actually result in a better overall mileage. In fact, it may result in less optimal mileage, but I was curious as to how it would work when I was originally writing the code for this application.

K1: Verification of Data Structure

You may verify that my data structure and program in general meet the requirements of this assignment by examining my code and by going through the menu options in my user interface. In addition, the screenshots in the “Program Screenshots” folder found in the project directory also prove that the program meets the requirements for this task.

K1A: Efficiency

Since my hash table does not have a method to automatically adjust its maximum size based on the number of elements it is storing, the time required for the look-up function (**read**) to execute will grow linearly as the number of packages grows. This is because as a package is added to the hash table it is added into one of ten “buckets”. When the **read** function is called to look-up an entry in the hash table it calculates the bucket where the entry will be stored and then iterates through every element in that bucket until it finds the correct entry. As N-number of packages are added to the hash table, the time required for the **read** function to execute becomes N+1.

K1B: Overhead

As with the efficiency of the **read** function, the amount of space used by the hash table I used grows linearly as the number of packages increases. As N-number of packages are added to the hash table, the space required to store the hash table becomes N+1.

K1C: Implications

An increase in the number of trucks and/or number of cities would result in no change to the look-up time or space usage in the data structure I used for this project. The information for each package is stored in the data structure as a list with values for Package ID, Street Address, City, etc. I do not store which truck the package belongs to in the data structure, therefore there will be no change there. And additional cities will just mean that there will be a greater variety of cities found in the package table’s city column. A change to either of these would impact space or time complexity. An increase in the number of trucks would however alter how my optimization algorithms work so there would be some changes there. The algorithms would still have a linear space complexity **O(N)** and a polynomial time complexity **O(N^2)** but there would be a need for additional algorithms since I use one for each truck. Unless the added trucks only contained packages with no constraints. If that were the case then the algorithm **get_truck_3** could be used.

K2: Other Data Structures

The data structure that I chose to use for the purposes of this assignment is essentially a self-adjusting list of lists (or a list of lists and then each of those lists contains multiple tuples if there is more than one element) that uses a hashing function to assign each table entry (element/package) to a nested list with the package ID being the identifier used for the look-up (**read**) function. This data structure works quite well for this use-case because it keeps the storage size minimal when a predefined number of inputs is given. Additionally, the hash table I used is fairly adaptable and can be scaled quite well with a few minor adjustments, as long as the number of inputs is known. If the number of inputs is unknown then a different data structure or an altered version of the hash table I used may be a better option.

There were a couple of other types of data structures that could have been used for this project. One being Python's built-in dictionary data structure with key-value pairs. Another type of data structure that could have been used is a Graph. Either would have been able to store that required data for this application to function properly, although performance would vary depending on which option was selected. Each of these options will be compared more closely in the next section.

K2A: Data Structure Differences

As I mentioned in section K2 above, I opted for a self-adjusting hash table to store the package data in my program because it is adaptable, somewhat easy to scale, easy to maintain, and dynamically adjusts to the number of inputs provided, to a certain extent.

If I were to have used a dictionary it would have been incredibly easy to implement. I would not have needed to code my own data structure and it would include all of the methods that come with a built-in Python data structure. That being said, there would have been plenty of drawbacks to using a dictionary for this use-case. One major flaw would be its inability to be self-adjusting. Also, I would need to manually add each item to the dictionary, updating entries would be fairly simple but verbose, reading entries would be straightforward, and removing entries would be a little tricky since I would have to accommodate key exceptions (if the key being removed does not exist). The flaws for using a dictionary outweigh any benefits so the hash table still remains the best choice so far.[3]

Next we have the graph. This is an interesting option. It would be somewhat tricky to implement, but once it was it could prove to be more efficient than a hash table for the purposes of this project. This is because the graph data structure is great at optimizing for efficiency. If I added each package entry to the graph (as nodes) and then connected the nodes with the edges being the distance between the addresses then I may be able to pre-optimize the packages based on distance before even passing the data to the optimization algorithm. This could save time, compute power, and may result in a further optimized result. However, the complexity of implementing a graph data structure made it impractical for this project and therefore a hash table still made the most sense. If the dataset for this project was much larger or if the number of trucks, packages, destinations were unknown then a graph could be a better option.[4]

L: Sources

Citations for the sources I used for this project can be found as footnotes where applicable.

[3] Hash Tables and Hashmaps in Python: Hash Table vs Hashmap. (2020, November 25). Retrieved January 01, 2021, from https://www.edureka.co/blog/hash-tables-and-hashmaps-in-python/

[4] Data Structures in Python: List, Tuple, Dict, Sets, Stack, Queue. (2020, November 25). Retrieved January 01, 2021, from https://www.edureka.co/blog/data-structures-in-python/

M: Professional Communication

I have attempted to use professional communication throughout my code and report to the best of my abilities.