**A: Algorithm Selection**

My algorithm delivers all packages on time, meeting all the truck requirements (when the package must be delivered by a specific truck) and the package requirements (when the package must be delivered with specific packages). The delayed packages are also picked up only when the current time passes their delayed time of arrival, or in other words, when the package has arrived at the warehouse.

My loading algorithm is considered a greedy algorithm, and consists of these steps:

1. Calculate all the best paths between locations using Dijkstra’s Algorithm[1] and place them in a 2D array as a lookup table, with the row index being the start location id of the path, and the col index being the end location id of the path.
2. Use two trucks.
   1. The first truck is assigned a behavior that automatically loads the time-sensitive, not delayed packages into the truck.
   2. The second truck is assigned a behavior that waits for the earliest delayed packages and loads them in.
3. Both trucks have these behaviors for loading in more packages beyond their responsibility:
   1. If the truck still has space for more packages, then it checks the packages it already has for package requirements (when the package must be delivered with specific packages). If the truck has already loaded in packages like those, then it looks for packages in the warehouse that need to be in the same truck to fulfill those requirements, and loads them in.
   2. If the truck still has space for more packages, then it checks the warehouse for any packages that has a truck requirement to be in this specific truck, and loads them in.
   3. If that truck still has space for more packages, then it checks the warehouse for any packages that need to go to the same location as any of the packages currently in the truck, and loads them in.
   4. If that truck still has space for more packages, then it fills it up with groups of packages that go to the same locations together.
   5. If the warehouse ran out of packages, the truck stops its delivery and is done for the day.
   6. The truck only loads packages when it ran out of packages and it is currently at the warehouse/hub.
4. After loading all the packages, all the location ids are grabbed from the packages, and the packages going to the same location are grouped under the same location id.
5. Those location ids are then sent to the TSP (Traveling Salesman Problem) solver to find an optimal, if not the most optimal, path to take to deliver all the packages.
6. The trucks are then simulated to drive and deliver all the packages to their locations, keeping track of the time each package is delivered and how many packages failed to be delivered on time or at all.

My TSP Solver algorithm is considered a greedy algorithm, and consists of these steps, given a queue of locations and the best paths between locations, as solved by Dijkstra’s Algorithm earlier:

1. Track the shortest distance, shortest time if desired, and shortest path.
2. A location is removed from the queue and inserted to the path list at each index.
3. The overall distance of the current path is calculated. If the overall distance of this path is shorter than the overall distance of an earlier shortest path, the shortest distance and shortest path is changed to this path.
4. After the location has been tried to be inserted to all the indexes, get the shortest distance and path, and replace the overall shortest distance and path with them.
5. Repeat this until the queue is empty.

**B1: Logic Comments**

Comments are provided where necessary and all functions are documented.

**B2: Application of Programming Models (Revised)**

The different classes in the program can access data from manager classes using the Singleton pattern through the Borg Idiom.[3] Data is also passed around using constructors. The data (packages, addresses, and distance adjacency matrix) are all read into the program from .csv files placed in the “assets” folder. Because all the data are stored in local files on the same local machine, no communication protocol is necessary to access those data from a server. The application runs on Python 3.8 in Windows 10 and coded using PyCharm 2020.1.

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

The space-time complexity is provided for each function using the Big-O notation to show the worst-case space and time complexities.

**B4: Adaptability**

The algorithm can handle more packages and more locations to go through, as long as the data provided are formatted properly (correct type, format, and number of columns) and as a .csv file of the correct name. However, the algorithm will not always guarantee that packages are delivered on time or that mileage will always be below 140 miles once the data is scaled up. More varying truck loading behavior is needed to be able to guarantee that.

**B5: Software Efficiency and Maintainability (Revised)**

The algorithm is efficient in a way that it provides a solution to the problem that will result in overall truck mileage of only 104.5 miles. The algorithm can be optimized further in different ways, as will be discussed below. Overall, the program runs in polynomial time complexity. The algorithm is also separated into different classes and functions in a way that makes sense (no class is given too much responsibility), and each function is documented, making it easy to maintain.

**B6: Self-adjusting Data Structures**

The data structures used in this software are hashmaps and queues. Hashmaps are as efficient as arrays when accessing an item. However, it does not need numerical indices as ids to determine where an item is stored in the data structure. Hashmaps allow strings to be used as keys by hashing this string into an index of its own internal array. Hashmaps are also more efficient on insert and delete operations than arrays, becoming O(1) for both operations if properly implemented.

Queues are more efficient than arrays when it comes to inserting an item to the back of the queue and deleting an item from the front of the queue, taking only O(1) time for both operations. This is especially useful for queueing up the order of locations the truck needs to go through to deliver the packages.

**C: Original Code**

The code runs a simulation of trucks going to different locations to deliver the packages on time, passing all the requirements the packages needed, all while getting the overall mileage of all trucks down to only 104.5 miles.

**C1: Identification Information**

My first and last name, and student id is provided as the first comment on the first line of each code file.

**C2: Process and Flow Comments**

Comments are provided in the code at large blocks of code, where necessary. Documentation is also provided for all functions of the program.

**D: Data Structure**

The program uses hashmaps to store packages and locations, and act as a lookup table the performs much more efficiently than arrays.

**D1: Explanation of Data Structure**

Hashmaps contain key-value pairs to store data. The key for each pair is made by using a hashing function to hash the id of the value, either the package id or location id.

**E: Hash Table**

The hashmap created from scratch contains an insertion function that can insert anything, including objects, into the hashmap. Package and location data are first aggregated into a package and location object and then inserted into their own hashmaps using their ids as the key.

**F: Look-up Function**

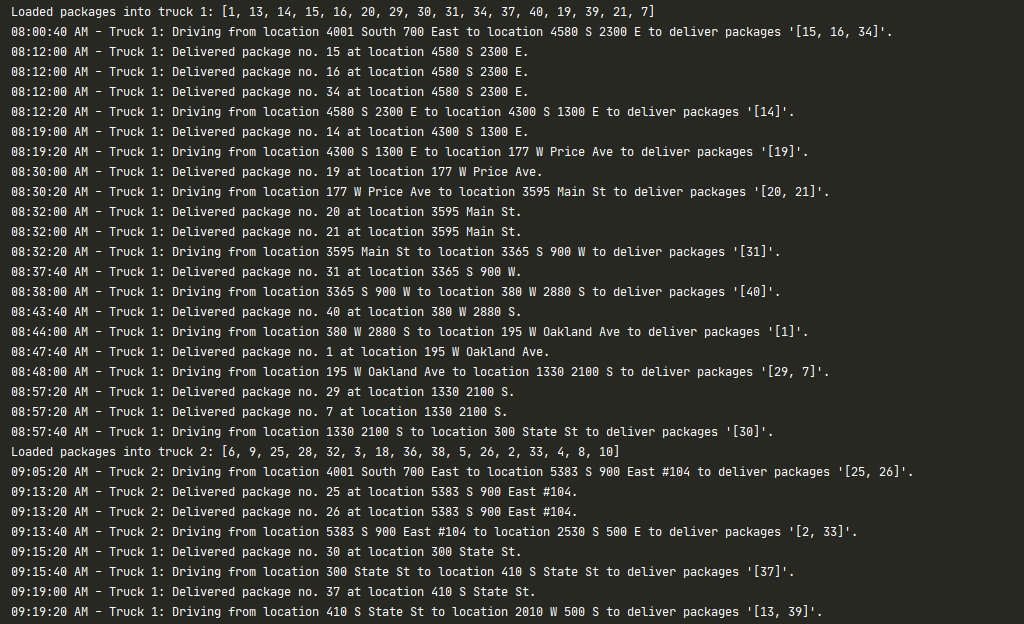
The hashmap look-up function is implemented using the `\_\_getitem\_\_` premade function of Python. This allows for getting items with keys using the bracket notation like that of arrays.

**G: Interface**

The interface shows a log of timestamps when trucks are about to drive to a location to deliver packages and when they deliver those packages to their location. After all packages are delivered, all the packages and their information are shown, alongside their delivery status and the time they were delivered. Afterwards, the total truck mileage, the number of packages successfully delivered, and the number of packages failed to deliver (either late or not delivered) are shown.

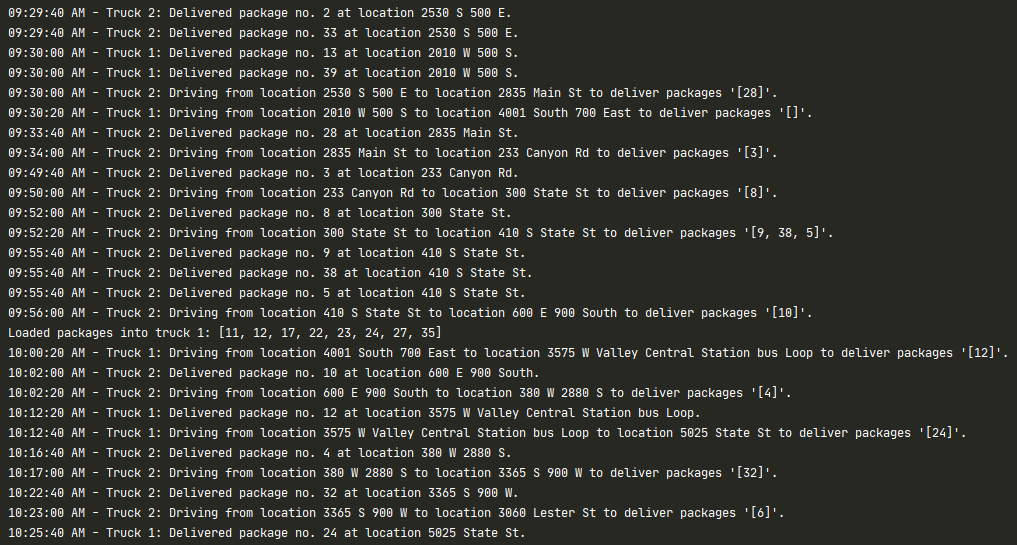
**G1: First Status Check**

This is a screenshot of the simulation of the package deliveries between 8:00 am and 9:25 am:



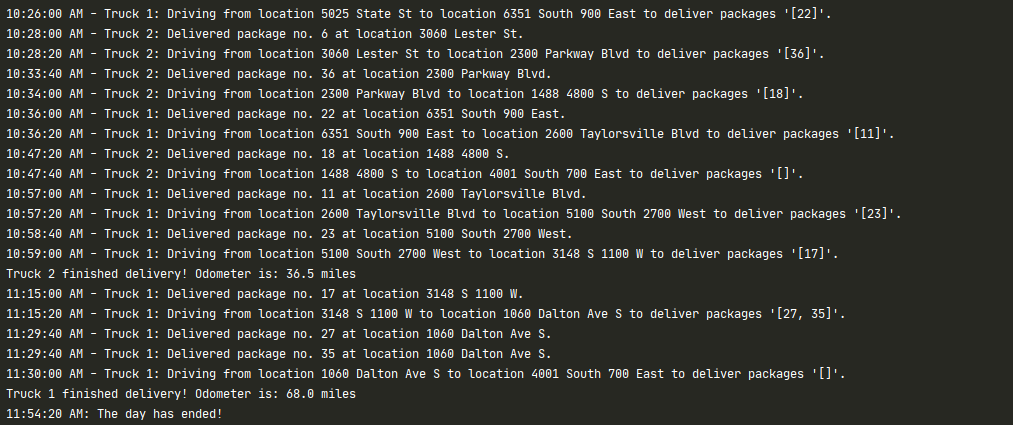
**G2: Second Status Check**

This is a screenshot of the simulation of the package deliveries between 9:25 am and 10:25 am:



**G3: Third Status Check**

This is a screenshot of the simulation of the package deliveries between 10:25 am until the end of the simulation, at 11:55 am:



**H: Screenshots of Code Execution**

Since the screenshots of the entire code execution are too tall to fit in this document, it has been provided in a separate folder called “screenshots”. All the other screenshots provided above are included in that folder as well.

**I1: Strengths of the Chosen Algorithm**

1. Easy to understand and implement
2. Provides an optimal, if not the most optimal, solution to the problem

**I2: Verification of Algorithm**

At the end of the code execution, after all the simulation of the trucks delivering the packages, a verification is shown that shows the overall mileage of both trucks combined. It also shows the number of successfully delivered packages (on time and meeting all the other requirements) and the number of failed to deliver packages (late or not delivered).

**I3: Other Possible Algorithms**

1. Clustering Algorithms
2. Solving the Vehicle Routing Problem (a variation of the Traveling Salesman Problem) using Tabu Search Algorithm

**I3A: Algorithm Differences**

1. For Clustering Algorithm, the locations are first clustered. Then, besides grouping packages based on their requirements, each package is grouped with other packages based on which cluster its delivery location is in. In other words, packages to be delivered in the same cluster should be grouped together. This should be able to plan a better route than the algorithm used, depending on the clustering algorithm used.
2. For Tabu Search, the locations to go through are placed into different routes, and locations are moved between routes to find where they best belong in an overall route. This should also be able to plan a better route than the algorithm used, depending on the implementation of the “short-term memory” data structure and the number of iterations called the “tabu tenure”.[2]

**J: Different Approach**

In the TSP Solver code, the calculation of the overall distance when calculating the best path to take can be further improved. Instead of getting the overall distance by always summing the distance between two consecutive points, only the inserted location and the adjacent locations in the array need to be recalculated. This can be done using the steps below:

1. Given the index *i* where the new location (locnew) should go into the array, get the location currently at index *i* of the array and let this location be called loci. Then get the location currently at index *i – 1* of the array and let this location be called loci – 1.
2. Get the distance between loci and loci – 1. Subtract that distance from the current overall distance.
3. Get the distance from loci – 1 to locnew. Then get the distance from locnew to loci.
4. Add the distances in step 3 together. Then add the resulting sum to the current overall distance.

This will lead to an O(1) time complexity instead of the previous O(n) time complexity. Here is an example:

Given a path array 0 -> 1 -> 2, where the distance of 0 -> 1 is 3 and 1 -> 2 is 5, making the current overall distance (3 + 5 = 8).

Say that we want to insert 3 into index 1 of the array. The loci is 1 and the loci – 1 is 0. The distance of 0 -> 3 is 4 and the distance of 3 -> 1 is 2.

The distance of 0 -> 1 is 3, so we subtract that from the current overall distance (8 – 3 = 5).

Then we add the distance of 0 -> 3 and the distance of 3 -> 1 together (4 + 2 = 6).

Then we add this resulting sum to the current overall distance (6 + 5 = 11), giving us 11 as the new overall distance.

**K1: Verification of Data Structure**

At the end of the code execution, after all the simulation of the trucks delivering the packages, a verification is shown that shows the overall mileage of both trucks combined. It also shows the number of successfully delivered packages (on time and meeting all the other requirements) and the number of failed to deliver packages (late or not delivered). The hashmap or hashtable with a lookup function is also present and used for storing the package and location objects.

**K1A: Efficiency**

The type of data used in the hashmaps are the Package and Location objects. The Package object stores the properties of a package (id, address id to go to, delivery time, etc.) while the Location object stores the properties of a location (address id, the name of the location, the address, city, etc.)

**K1B: Overhead**

For the hashmap, its access, search, insert, and delete operations run on O(1) time complexity, as long as the hashing function is implemented well. It has O(n) space complexity. Since this is a local application that does not need to connect to any servers or databases, there are no bandwidth constraints.

**K1C: Implications**

When the number of packages increase, the algorithm needs to warn the user about possibly not meeting the package requirement, if there are more packages that need to be delivered together than the capacity of the truck, and the truck requirement, if there are more packages that need to be in a specific truck than the capacity of the truck. More truck behaviors have to be implemented to handle more loading strategies to be able to handle the increased amount of packages to meet all their requirements.

When the number of trucks increase, some truck behaviors need to change to accommodate for the increase in trucks while making sure all the requirements of packages are met. For example, the time-sensitive trucks can grab a subset of time-sensitive packages depending on how many other time-sensitive trucks are there.

When the number of cities or locations increase, as long as the distance adjacency matrix provided represents a fully connected graph, the algorithm should still work without any changes needed to be made.

**K2: Other Data Structures**

Other data structures that meet the criteria are:

1. Min Heap instead of an array for Dijkstra’s Algorithm in my Pathfinder, using the distance of the node as the key, to get the next closest node in O(1) time, compared to the array’s O(n) time.
2. Singly Linked List instead of an array for the solve() method of my TSP Solver. The array in question is the copy of the shortest path array used to test for the shortest distance path. The insertion of an item to an index will take only O(1) time compared to the array’s O(n) time.

**References:**

[1] Dijkstra's Algorithm for Adjacency List Representation: Greedy Algo-8. (2020, April 27). Retrieved July 27, 2020, from https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/

[2] Glover, F., &amp; Kochenberger, G. A. (2003). An Introduction To Tabu Search. In Handbook of Metaheuristics (pp. 37-54). Boston, MA: Kluwer Academic.

[3] Martelli, A. (n.d.). Avoiding the Singleton Design Pattern with the Borg Idiom. Retrieved July 27, 2020, from https://www.oreilly.com/library/view/python-cookbook/0596001673/ch05s23.html