**Core Algorithm Overview**

**Stated Problem:**

The goal of this project is to use the Python programming language to construct the best feasible path for the delivery of Western Governors University Parcel Service (WGUPS) parcels. Three trucks and two drivers are needed to deliver 40 parcels. Some packages were late, while others had problems that needed to be fixed. We must parse and filter the data in order to put the packages into the appropriate trucks. To find the shortest path from the truck's present position to the next destination, a greedy algorithm is used. This method is called “greedy” since it finds the shortest available path from its present location and repeats the process until no more packages are available. This article will examine how this technique is used, as well as offer a description of the application's methods and components.

**Algorithm Overview:**

The Greedy Algorithm performs in the following steps:

1. A group of manually loaded packages, as well as the corresponding truck number and starting point are provided (starting point is always the hub).
2. Shortest distance is initiated by maximum value.
3. The function will loop through each package on the truck and get the distance between current location and the next location.
4. Shortest distance variable gets updated when the function found a location with shorter distance.
5. Repeat this process until the function loops through all packages available on the truck.
6. The algorithm will return the ID to the next address, next package object and the shortest distance.

This self-adjusting greedy algorithm's operation time or also known as Big O is O(N2) in the worst case and O( N^ 2) in the best scenario O( 1 ). The worst-case is virtually always assured, whereas the best-case scenario is only conceivable when a truck's list of items to be loaded is empty.

*Pseudocode of the WGUPS Greedy Algorithm:*

Time complexity: O(1)

1. **The function called algorithm will take in 2 parameters:**
2. **current\_address\_ID**
3. **truck\_load**

Time complexity: O(1)

1. **Initialize nearest\_distance variable with the largest value in the distance\_data 2D array.**

**near\_distance = 15**

Time complexity: O(n^2)

1. **Iterate through each package ID in truck load**

**for each ID number in truck\_load**

**set package = package\_table.get(ID number)**

**package\_address\_ID (to get the address ID from address csv file)**

**for each address in address table (loop through all addresses in address csv file to find an address that matches the package’s address.)**

**if address equals package.address:**

**Set package\_address\_ID = address.ID**

1. **Get the distance between the two locations using the distance\_data 2D array.**

**Distance = distance\_data[current\_address\_ID][package\_address\_ID]**

1. **If distance is shorter than previous distance**

**If distance < nearest\_distance**

**Set nearest\_distance = distance**

**Set next\_package = package (this variable holds the next package the truck will deliver)**

**Set next\_address\_ID = package\_address\_ID**

Time complexity: O(1)

1. **Once the iteration finished**

**Return next\_package object (this is the package the truck will deliver next)**

**Return next\_address\_ID**

**Return nearest\_distance**

The total time complexity of this entire algorithm is O(n^2)

**Method and Class Overview**

**ReadCSV.py**

| **Method** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| parse\_package\_csv | $O(n)$ | $O(n)$ |
| Parse\_distance\_csv | $O(n)$ | $O(n)$ |
| Parse\_address\_csv | $O(n)$ | $O(n)$ |

**HashTable.py**

| **Method** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| \_\_init\_\_ | $O(1)$ | $O(1)$ |
| \_hash\_key\_generator | $O(1)$ | $O(1)$ |
| add | $O(n)$ | $O(n)$ |
| get | $O(n)$ | $O(n)$ |

**Package.py**

| **Method** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| \_\_init\_\_ | $O(1)$ | $O(1)$ |
| print\_delivered | $O(1)$ | $O(1)$ |
| print\_not\_delivered | $O(1)$ | $O(1)$ |
| print\_details | $O(1)$ | $O(1)$ |

**Main.py**

| **Method** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| time\_calculator | $O(1)$ | $O(1)$ |
| go\_next\_location | $O(n)$ | $O(n)$ |
| algorithm | $O(n^2)$ | $O(n^2)$ |

**Advantages of Greedy Algorithm**

The implemented Greedy Algorithm performs all of the necessary operations to satisfy the project's requirements and distributes the 40 packages within a 140 mile constraint. The ability of this algorithm to swiftly determine the best possible route for a certain vehicle is its greatest strength. Another significant benefit of the greedy algorithm technique is that it is highly efficient when working with a small dataset. Because of its simplicity to implement, this algorithm is extremely flexible to be modified in order to meet the given requirements.

Because they do not consider all of the data, greedy algorithms can fail to identify the absolutely optimum solution. A greedy algorithm's decision may be influenced by previous decisions, but it is unaware of potential future decisions.

**Alternate Algorithm**

An alternate technique of utilizing a Floyd-Warshall algorithm may have been utilized instead of the present approach of using Greedy algorithm and DFS to locate the cycle. This has the potential to discover a quicker path, but at Big O(N3 ), it is slower. Given the existing size of activities, running at this pace would probably be suitable, but it would be more difficult to scale up. “The Floyd-Warshall algorithm is an example of [dynamic programming](https://brilliant.org/wiki/problem-solving-dynamic-programming/). It breaks the problem down into smaller subproblems, then combines the answers to those subproblems to solve the big, initial problem.” (Brilliant, 2019). A self-adjusting heuristic is a second approach I might have used to find the best path. Starting at the hub, this method would identify the shortest way to the hub. It would then figure out all of the packages that needed to be delivered to that truck and load them into it. It would then begin at the new site and find the quickest path from there. If a vehicle has previously visited a place, it will continue on to the next closest location until all 40 items have been assigned to a truck and a path has been established. In the sense that the shortest accessible path is always taken, my method and the self-adjusting heuristic have something in common.

**Data Structure Overview**

Since the one of the requirements of this project is to use hashtable, a hashtable was made to be the primary data structure employed in this project. Because each bucket can only hold one package, direct hashing has the disadvantage of creating the biggest hash table. However, because the WGUPS only ships 40 packages each day on average, the memory usage will be minimal. The hash table comes 10 buckets by default, however this may be sized up or down by changing the size of the table when we initialize the constructor. The hashtable generates a hashkey from the unique package ID, which is a simple modulo generator that determines which bucket index the package will be placed in. I was able to build a hash table with chaining using a list, which resulted in extremely quick look-up, deletion, modification and insertion. Since most of the operations on this hash table rely on package ID, the runtime performance will be unaffected because those methods execute at Big O(1). The main benefit for me was that it allowed me to double-check the accuracy of my vehicles and packages along the route while incurring minimal additional time.

**Alternate Data Structure**

Reflecting closely, I could have utilized a variety of data structures to meet the project's objectives. A binary search tree is a very common data structure. The Binary Search Tree would be different in that I could presort the packages depending on an attribute and retrieve them fast via a tree. Binary search trees are used to construct dynamic sets and lookup tables and enable for rapid lookup, adding, and removal of data elements. A graph was another data structure I might have used. The ability to put comparable packages together as neighboring vertices was a significant benefit of utilizing a graph. After that, I could have explored the graphs till I reached a maximum traversal length of 16. This would also be an excellent way for scaling in the future.

**Adaptability**

The application's basic operations are built to scale, and they take into account changes in the quantity of packages, trucks, and locations. To scale with any of these components, minor modifications are necessary. Another set of location csv files, for example, can be entered into the application to compute a new route and identify the best way. Additional packages can be added, and the algorithm will decide where they should go. This method also gives you a lot of flexibility when it comes to developing several sub-applications because the architecture allows you to alter the input set at any time. The method I use to load the items into the truck might be an issue for future scalability in this application. To fulfill all of the package limitations and deadlines, it is presently done manually. Because the architecture allows for easy changes to the input set, this method allows for a lot of flexibility in the creation of numerous sub-applications. The technique I use to put the products onto the truck might be a problem for the application's future expansion. It is currently done manually to meet all of the package limitations and deadlines. Instead, if I had the chance to enhance this project, I would translate the special notes of each package to key code which can be used to automate the loading package task of the program. This will save a lot of time when we’re dealing with a large number of packages.

**Implications**

The lookup time and space usage are both dependent on the number of cities and trucks, therefore, changes to these two parameters will certainly affect the space and time complexity. Increasing the number of cities implies we'll have more addresses to iterate through in our algorithm. As the number of cities increases, our program's time and space complexity will increase as well. To compensate for this discrepancy, we may add extra trucks, which will assist to evenly distribute the amount of packages and addresses the algorithm must loop through, thereby compensating for the increased number of cities.

**Efficiency and Maintainability**

Since we’re dealing with a manageable amount of packages, our program is highly efficient, with only two comparisons taking O ( n ^ 2 ) time to complete. While this isn't the most time-consuming method, it works nicely with the 16-package restriction per vehicle. That’s why Greedy Algorithm was the first choice for this problem, thanks to its simplicity and adjustability. The program is also incredibly easy to maintain and modify because so much of the software is the same basic functionalities that have been tweaked for the specific use case. In terms of maintainability, each function is easily accessible because of the usage of object-oriented programming.

**Programming Model:**

Because this application is presently hosted on a local memory, the programming paradigm is constrained. The Pycharm IDE is used to run the program, which was written in Python 3.9. The program collects data from CSV files in the project folder on the local system, thus there is no connection protocol. To do this, we utilize Python's csvreader function from the csv package. As a result, data transfers are restricted to interactions between the program and the local computer. We also use the datetime library, which assists with time tracking and formatting. The system console is the primary user interface, which accepts user inputs and uses that information to generate outputs.

**Alternate Approach**

If I were to do this project over, the handling of package limitations would be one of the things I would change. The present approach solely takes into account the problem's specific constraints. However, my objective is to create a more flexible system that employs a more general limitation mechanism that allows for more complicated and diverse constraints. The chosen algorithm is another part of the present approach that I would reconsider. A greedy method solves the issue as described, but it does not scale well as the problem's search space grows. Furthermore, because it does not evaluate solutions that partially fill trucks, the current implementation is unlikely to find the most efficient option. A metaheuristic approach, in my opinion, would avoid these global optima by taking into account a much broader search area in a more efficient manner.

**Source:**

1. *Floyd-Warshall Algorithm | Brilliant Math & Science Wiki*. (2019, October 12). Brilliant. https://brilliant.org/wiki/floyd-warshall-algorithm/