**ALGORITHM IDENTIFICATION**

The algorithm I used for this project is Dijkstra's algorithm, and proper implementation was researched before production. The algorithm uses a graph-like node system, each one being iterated and compared to the other to find a semi-optimal path. This works well with manual loading of the trucks but can be modified to determine automatic loading in the future. The algorithm is effective and

**programming environment**

I used the recommended programming environment Pycharm using the standard libraries. This paired with multiple adjacent python files allows for the multiple moving parts of the program to stay separated and avoid any unwanted alterations.

**Evaluate the space-time complexity of each major segment of the program, and the entire program, using big-O notation.**

The Algorithm itself runs at O(N^2) Quadratic time, Retrieving the information runs at O(N^2) Quadratic time, Creating and Comparing nodes O(N) Linear Time, The Delivery Function runs at O(N^3) Cubic Time, Retrieving Packages run at O(N) Linear Time, and the run Function that determines where the trucks are runs at O(1) Constant Time.

The highest time complexity in the software is O(N^3) Cubic time, meaning that the software runs at O(N^3) Cubic time

**Algorithm Overview**

Dijkstra’s algorithm works as such

* Populate a graph with nodes
* Mark every node as unvisited initially
* Sequentially Mark every node with a distance located in the csv documents provided
* From the current node consider all the distances between itself and all other unvisited nodes
* After all possibilities are considered remove the current node from unvisited and place it in the visited set
* Continue this until all of the destinations have been met.

**Dijkstra’s Algorithm:**

**StartNode minimum distance =0**

**For every item in the csv file create an unvisited node**

*O(N)*

**While length(Unvisited) >0**

**Current node = pop.(unvisited)**

**Current node.visited = true**

*O(N)*

**For each Adjacent node in graph**

**If the adjacent node hasn’t been visited then**

**Calculate weight of edge between current node and Adjacent node**

**Set new distance = Current node minimum distance**

**Check if this is the shortest available distance**

**Mark best solution as visited and start over**

**Repeat loop**

**Repeat Loop**

*This will continue until all of the Unvisited nodes are marked visited and a solution has been discovered*

*O(N^2)*

**Explain the capability of your solution to scale and adapt to a growing number of packages or trucks.**

The solution is easily scalable from a programming perspective. The software is expecting the number of values that are given initially. So adding to the csv will yield no difference in the output, but alterations can be made to the code to allow for a growing list of packages if needed

Adding more trucks or cities would not drastically impact the lookup time because the hash table is specifically in regards to the packages, so looking up where they are is more a case of finding the package( O(N) ) because the trucks are loaded manually. The cities are a variable in the package object so adding cities will not impact the time complexity of the software.

**Discuss why the software is efficient and easy to maintain.**

The software is set up to carry out these very specific instructions. However, each of the moving parts is separated from the other to avoid unwanted results. So iteration on the individual pieces of code is as straightforward as creating new methods in the various python files.

**Discuss the strengths and weaknesses of the self-adjusting data structures**

The hash table is a self-regulating data structure that I find most interesting and useful, avoiding collisions using a key-value retrieval system, meaning that it can be added to and retrieved easily without risking an error. However, the use of the hash table is not as straightforward as the use of an array or list, where an index can be used for data retrieval, which requires a solid design plan must be laid out for the system to function. When adding objects to the hash table, the Table runs a function called a hash function which manipulates the data so that it can be sorted into buckets, These buckets contain additional lists and are iterated through to add and retrieve values to avoid data collisions.

The data structure was able to efficiently store packages that have multiple collisions, such as delivery times, destinations, and what city they are to be delivered in. This is perfect for this scenario because the hash table is able to self-sort that data without anything being overwritten or removed unintentionally.

The data structure is also very fast when it comes to information retrieval, Having a Time complexity of O(N), meaning no matter how many things are added into the table it will return the selected item in O(N) Linear time. The amount of space used by the Hash Table is also O(N) meaning that it will scale linearly with the amount of information placed within it rather than growing at an excessive rate.

**Describe *at least* two strengths of the algorithm used in the solution.**

It is a very simple algorithm, specifically Quadratic time, and is good at finding the shortest distance from a selected node and all other available nodes, the trouble arises with making sure there are no revisiting old locations

**Identify two other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario.**

***Bellman-Ford algorithm***: According to (“Bellman–Ford Algorithm: DP-23.”, geeksforgeeks), The algorithm is more complex, takes more time, and more accurate, looks at the edges one at a time rather than all at once.

***Floyd’s Algorithm***: According to (“Floyd Warshall Algorithm: DP-16.”geeksforgeeks), The algorithm is More complex, stores distances differently, would check against every vertice every iteration, including backward, all in all not that different from Dijkstra’s

**Identify two other Data Structures that could have been used instead of the Hashtable**

A key-value pair dictionary for the containers for packages. It is O(1) for retrieval, and O(N) for data searches, this wouldn’t be very different from the hash table I used in this implementation if anything it might have been faster. While a dictionary does not have to use a hash function to sort values into buckets and avoid collisions, it is able to differentiate multiple key-value sets without added complexity.

Another solution would be to simply iterate through a list of package objects, however, I believe handling each one that way would have proved more confusing in the end due to the fact that finding values in lists or arrays is done with the use of integers denoting placement in said list or array.

**In conclusion**

If I were to do this project over or had more patience I would likely use the Bellman-Ford method and implement a self-loading system. I may still do this in the future so that I can have a more compelling piece for my portfolio, of course, once I'm more proud of the outcome.

**Sources**

I looked at multiple examples of the implementation of code for this kind of problem on these sites.

“Where Developers Learn, Share, & Build Careers.” *Stack Overflow*, stackoverflow.com/.

“Dijsktra's Algorithm.” *GeeksforGeeks*, 30 Sept. 2020, www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/.

“Bellman–Ford Algorithm: DP-23.” *GeeksforGeeks*, 23 Apr. 2020, www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/.

“Floyd Warshall Algorithm: DP-16.” *GeeksforGeeks*, 29 Jan. 2021, www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/.

“Where the World Builds Software.” *GitHub*, github.com/.