Core Algorithm Overview

**Stated Problem:**

The purpose of this project is to create an algorithm that will allow trucks to make deliveries to various local locations within constraints such as time deadlines and delays while keeping the total duration traveled by delivery trucks under 140 miles

**Approach to solving stated problem:**

In order to achieve these requirements, I adopted the strategy of using a graph data structure with edges representing distances between delivery locations. My approach utilizes the programming language python (version 3.13.0) and the Pandas library for excel data extraction. The software is run on a standard 64-bit MacOS personal computer architecture. The editor I used to develop this application is VSCode My code had 4 distinct parts:

1. A Package class, which created instances of package orders and was contained in a PackageList class. The PackageList class contains two separate dictionaries for priority packages (packages that have a time deadline) and non priority packages (that do not have a time constraint aside from the end of day)
2. A Truck class that would deliver packages and calculate the time each delivery would take as well as total distance duration
3. A DeliveryGraph class that would store the DeliveryVertex (representing a delivery location) as well as the distance between each vertex

**Core algorithm breakdown:**

In order to find the most ideal set of packages for the trucks to deliver to I implemented a variation of the nearest neighbor algorithm. This is a greedy algorithm, which uses the DeliveryMap’s adjacency matrix and edge weights to find the most optimum next location to deliver based on whichever package does not have a status of ‘delivered’ and the current vertex. This algorithmic approach is composed of several parts:

*// in DeliveryGraph class from DeliveryMap.py*

*// Function to determine a route with up to package\_capacity using the nearest neighbor approach*

**function nearest\_neighbor\_route(starting\_location, packages, package\_capacity = 16)**

**Visited = empty set**

**current\_vertex = starting\_location**

**assigned\_packages = empty list**

**assignments = empty dictionary**

**// Group packages by delivery vertex**

**For each package in packages:**

**If package.delivery\_vertex is in assignments:**

**Add package to assignments at key package.delivery\_vertex**

**Else:**

**Create a new list for package.delivery\_vertex in assignments and add package**

**all\_destinations = all keys from assignments**

**// Ensures that the driver offloads all packages at a location in one visit**

**While current\_vertex and length(assigned\_packages) < package\_capacity:**

**Min\_route = infinity**

**Next\_vertex = null**

**Vertices = filter vertices in adjacency list of current\_vertex that are in all\_destinations**

**// Find the next nearest vertex**

**For each vertex in Vertices:**

**If edge weight from current\_vertex to vertex < Min\_route and vertex is not in Visited:**

**Min\_route = edge weight from current\_vertex to vertex**

**Next\_vertex = vertex**

**// If no next vertex is found, end the loop**

**If Next\_vertex is null:**

**Break**

**Add Next\_vertex to Visited set**

**next\_package = assignments[Next\_vertex]**

**Append next\_package to assigned\_packages**

**// Move to the next vertex**

**current\_vertex = Next\_vertex**

**Return assigned\_packages**

The cost of this method is O(n) + O(k \* m) runtime, where n represents the number of packages in variable ‘packages’, k represents the package\_capacity variable, and m represents the total number of adjacent edges. Since package\_capacity (k) is a constant of 16 in this exercise, this big O notation can be simplified to O(n + m). The space-time can be represented as O(V) for the visited list, O(n) for the assignments dictionary, and O(D) for the all destinations list for a combined spacetime big-O of O(V + n + D). However with V never exceeding 16 packages, this can be considered a constant and reduced to O(n + D)

*//Generate\_next\_deliveries function (found in main.py)*

*//Function that utilizes nearest\_neighbor\_route on both the PackageList’s priority queue and non priority queue and combines the result to create an appropriate route considering time constraints*

**function generate\_next\_deliveries():**

**// Get the list of priority packages ready for delivery**

**priorities = orders.list\_ready\_priorities()**

// Generate the route for the priority packages using the nearest neighbor algorithm

**first\_deliveries = delivery\_graph.nearest\_neighbor\_route(delivery\_hub, priorities)**

// Determine the last endpoint of the first delivery route

**if first\_deliveries is not empty:**

**last\_endpoint = last package's destination vertex in first\_deliveries**

**else:**

**last\_endpoint = delivery\_hub**

// Get the list of non-priority packages ready for delivery

**non\_priorities = orders.list\_ready\_non\_priorities()**

// Generate the route for non-priority packages starting from the last endpoint

**second\_deliveries = delivery\_graph.nearest\_neighbor\_route(last\_endpoint, non\_priorities)**

// Combine both priority and non-priority deliveries

**deliveries = first\_deliveries + second\_deliveries**

// Create a dictionary to ensure unique delivery locations

**unique\_locations = empty dictionary**

**for each delivery in deliveries:**

**if delivery's address exists in unique\_locations:**

// Append the package to the existing list for this address

**add delivery's package to the 'packages' list at this address in unique\_locations**

**else:**

// Create a new entry for this address with distance and packages

**unique\_locations[destination address] = dictionary with 'distance' and 'packages'**

// Convert the dictionary back to a list of deliveries

**deliveries = convert unique\_locations' values to a list**

// Initialize variables for counting packages and tracking the last index

**package\_count = 0**

**last\_index = 0**

// Iterate through the list of deliveries to limit total packages to 16

for index, grouping in deliveries:

**package\_count += number of packages in grouping's 'packages'**

**if package\_count <= 16:**

// Update the last valid index that stays within the package limit

**last\_index = index**

**else:**

// Stop iteration once the package limit is reached

**break**

// Slice the deliveries list up to the last valid index, including that index

**deliveries = deliveries from start to last\_index + 1**

// Calculate the return distance from the last delivery's destination back to the hub

**return\_distance = distance from the last delivery's destination to delivery\_hub in delivery\_graph.edge\_weights**

// Append the return distance to the deliveries list

**add return\_distance to deliveries**

// Return the final list of deliveries with the return distance included

**return deliveries**

The big O runtime for this algorithm takes into consideration the runtimes for nearest\_neighbor\_route algorithm for the priority and non-priority queues, which leads to a runtime of O(n1 + m1) + O(n2 + m2) which can be simplified to be O(n + m), the same as nearest\_neighbor\_route algorithm. The space time complexity can be represented with O(n1 + n2) for the priority and non-priority lists and O(u) for the unique locations list, for a combined big O of O(n1 + n2 + u) spacetime. However since u is proportional to the number of packages this can be simplified to O(n1 + n2)

**Scalability of approach:**

A benefit of my approach is that much of the program is completely automated, and thus would respond well to increased packages. It prioritizes packages with deadlines and thus takes minimal user input in order to function. Packages are automatically loaded into priority or standard queues based on the excel input data. Thus if the package count were to increase the program would function the same and be able to accommodate the additional orders. Additionally package lookups can easily be done by referencing the combined package list by using the order ID as a key.

**Strengths and weaknesses of approach:**

A strength of my approach is the graph nature of the vertices and how they map to packages. This allows me to easily reference corresponding objects in the program via reference and object oriented design principles. The greedy algorithm may not find the most ideal route possible, but its runtime is less than quadratic and can generate appropriate routes for delivery without straining system resources.

A weakness of the approach I’ve taken is the lack of flexibility in changing a package destination once a truck has left. Once a truck has left, the entire route has been mapped out. This leads to trucks needing to wait for a delivery location to be updated before departing with that package, which results in unnecessary delays for other packages.

**Justifying Keys**

* Delivery Address: Using this as a key can be acceptable in a small delivery application, however this approach is not scalable since street names can be shared by different zip codes
* Delivery Deadline: Using this as a key can be a beneficial way to organize priority in packages, however should not be the primary way of identifying a package
* Delivery City and Zip Code: This can be useful for grouping packages for expansive delivery networks, however should not be used to identify individual packages
* Package ID: This is the most appropriate key for uniquely identifying a package. It also incorporates best with relational database design
* Package Weight: This is not a good key for most circumstances but can be useful for grouping oversize packages
* Delivery Status: Can be useful for grouping status for use in an alternative list, however this value is not static and would not make for a suitable primary key