Core Algorithm Overview

**Stated Problem:**

The goal of this project is to use the Python programming language to create a Nearest Neighbor Algorithm capable of finding an acceptably efficient driving route for the WGUPS package delivery service with which they can deliver multiple sets of packages on time according to each package’s stated delivery deadline while keeping the total mileage traveled under 140 miles.

Additionally, this program is to provide a system containing some form of user interface, allowing a user, presumably management, to track the status of any given package along it’s route as well as display the status of all packages at a given searched time, and show information relevant to those packages.

A.

My Nearest Neighbor Algorithm is created as follows:

1. After each truck is loaded, it’s list of packages is fed into the algorithm.
2. A secondary list is created with the purpose of sorting the package list into an efficient “delivery route”
3. Iterating through the entire package list, the algorithm compares the distance between the current location and the address of every other package until the nearest address is found.
4. The current location is changed to the nearest address, the package(s) is/are removed from the package list (decreasing it’s size by at least 1) and are added to the new sorted list.
5. Once all packages have been sorted into the new list or “route”, the algorithm returns the sorted list so the corresponding truck can “travel” along said “route”.

B1.

1. **(variable created to hold current address of truck, initialized to address of hub)**

**current\_location = hub**

**Accepts package\_list input**

**temp\_list = package\_list**

**(new list created to become the package list or ‘route’ sorted according to the algorithm)**

**sorted\_list = new empty list**

**(variable created to track closest location found so far, initialized to 25)**

**shortest\_distance = 25.0**

**(variable created to hold package id with closest delivery address, initialized to 0)**

**p = 0**

1. **As long as shortest\_found == False:**

**(Runs while loop)**

**(Iterates through temp\_list checking addresses)**

**For i in range (0 to (temp\_list size - 1)):**

**If distance between current\_location and address of package[i + 1] < current\_location and address of package[i]:**

**If distance between current\_location and package[i + 1] < shortest\_distance:**

**shortest\_distance = distance between current\_location and package[i + 1]**

**p = package[i + 1]**

**Else if current\_location and package[i] < current\_location and package[i + 1]:**

**shortest\_distance = distance between current\_location and package[i]**

**p = package[i]**

**Else if distance between current\_location and package[i +1] == current\_location and package[i] (Both addresses are equally distant from the current location):**

**(Default to selecting package[i + 1])**

**shortest\_distance = distance between current\_location and package[i + 1]**

**p = package[i + 1]**

**(after iterating through the entire package list and comparing all addresses, the closest address to current\_location is found)**

**(package variable ‘p’ is added to the sorted\_list)**

**sorted\_list.append(p)**

**(current location is changed to address of p)**

**current\_location = address of p**

**(p removed from package list upon ‘delivery’)**

**temp\_list.remove(p)**

1. **(checks to see if package list has been fully sorted into new sorted\_list)**

**If sorted\_list == ((package\_list length) – 1):**

**(break while loop)**

**shortest\_found = True**

**(adds last package into sorted list as final stop on route)**

**sorted\_list.append(final package)**

**(returns sorted list to be run through by truck)**

**Return sorted\_list**

**Else if packages have not been fully sorted:**

**(resets tracking variables to original values in preparation to search for the next closest address from the new current\_location)**

**shortest\_distance = 25.0**

**p = 0**

**(triggers while loop execution on remaining packages in temp\_list)**

**Shortest\_found = False**

**B2.**

The chosen Nearest Neighbor Algorithm, as well as the broader program, was built using the PyCharm IDE on an Asus Vivobook laptop with a Pentium(R) Silver N5030 CPU and 4GB of RAM.

**B4 – B6.**

While there may not be a perfect solution to the problem of delivering packages, the nearest neighbor algorithm, as chosen here, has proven to be very simple yet quite efficient.

Upon running my program, all packages are delivered in under 140 total miles across three separate delivery routes and all packages were delivered to their destinations on or before their given deadlines.

Despite the fact that the algorithm runs with a space-time complexity of O(N^2), there exists a reasonable capability to scale up. The algorithm only runs once per route to sort a truck load of packages into a more acceptably efficient delivery route. Given the inherent nature of this delivery scenario, N is limited to how many packages a truck can carry on a given trip, which in this case is a maximum of 16. Adding more than the current 40 packages would increase the number of trips, but the time spent executing the algorithm could not get any worse than the worst case run time on a maximum of 16 packages.

Use of the Nearest Neighbor Algorithm as well as thoughtful utilization of the hash table structure and consistent, clean coding practices allow for an efficient and easily maintained program. The use of well written comments throughout the code also contributes to maintenance efforts.

Instead of an algorithm that always chooses the location with the shortest distance, other algorithms could have used, such as:

1. Creating a list with 2 locations to start. Adding whichever location between the two that will create the shorter route, and repeating this step until a group of 16 packages can be delivered at those nearby locations in the list, then use that list as a delivery route. (Cheapest Insertion)
2. Randomizing the selection of locations, building a list of locations that can be visited to deliver up to 16 packages. (Random Insertion)

The chosen hash table structure offers the benefit of storing more than ten items within the ten buckets, however the more items that are added make retrieving any particular piece of data take longer. With that said, instead of iterating through every package to find the one you want, a hash table can take in a key(package id) and narrow down the search to a list of packages hashed into the same bucket.

Fortunately, increasing the number of trucks in use or the number of locations to visit should have no affect on time or space usage of the hash table in this scenario as only package data is being stored this way.

Each unique package ID is used as the key in a key/value pair that will be hashed.

The value that is tied to the key in the key/value pairing consists of a list containing data such as that package’s delivery deadline, delivery address, and weight.

Other potential alternative methods to the hash table that could have been used are such structures as:

1. A simple list of length 40. Each index contains a list holding a package’s data.
2. A simple dictionary. Instead of having a hash algorithm distribute packages into shared buckets based on a hashed key, each package id could be a unique key that would map to a list containing package data. Again we would have 40 lists, but each one could be retrieved in O(1) time by calling the key of the package in question.

If I were to repeat this project, I would work out a way to automate the loading of the trucks so that the program could scale to meet the demands of a more real world application using any locations, as well as more broad and varying package data.