Project Overview

The Task

The purpose of this project is to write a Python program that will deliver a set of packages within various constraints, such as total mileage, delivery deadlines, and required start times and truck placement. The Nearest Neighbor algorithm was chosen as the problem-solving technique. Initially, we have 40 packages. But the program is written to be scalable to allow any number of packages. We only have two trucks and three drivers. Being that the trucks are only allowed to hold a maximum of 16 packages each, one of the trucks must complete their delivery and return to the hub to reload, allowing the third driver to take the truck out for a second load of deliveries. The maximum total mileage allowance for the three trucks is 140 miles.

Hardware / Software

To complete this project, a Dell G15 (Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz) laptop with 32GB RAM running the Windows 11 operating system was used. The code editor used was Visual Studio Code with the Python extension installed and enabled. Program was run through VS Code using the console native to that application. Python version 3.10.5 was installed and used for this project.

The Data Structure

The primary data structure used to store the package data is a list of dictionaries. The dictionary keys are the package IDs and the values are the rows of data read from the csv file. The length of the outer list of the data structure may be passed as an argument when the class object is initialized. The default length is 10. The index of the hash map for each package is determined by using the modulus operator (package ID % hash map length).

As a new package is added, a new dictionary is stored in memory in the designated index of the hash map.

The lookup function takes the given parameters and return the corresponding values from the hash map. The time an space needed to do so grows linearly with additional added packages, trucks, and cities.

Each additional truck would also require another truck list to be stored in memory and additional set of lookups to be performed (for the packages on that truck). In total, though, the space-time complexity of lookups to be performed would always be O(n). Additional trucks would simply divide those total lookups into more groups.

Strengths of the data structure:

1. Because the package IDs are consecutive integers, this modulus process distributes the packages into the indices at an equal rate.
2. Using a list of dictionaries negates the need to loop through the items during the lookup function or insert function as you would if a list of lists was used.

Weaknesses of the data structure:

1. By grouping the packages into the hash map indices by package ID, their locations and distances to each other are not considered. If they were grouped according to zip code, a function to load the trucks could be more easily executed.
2. If packages are deleted, certain indices could lose their balance of packages

As an alternative to the chosen data structure, there are many choices. A binary search tree could have been used. Each of the package distances from the hub could have been evaluated and sorted. The package at the midpoint of the sort could have been the head node, with branches left and right being the remaining packages' distance from that one. Once the trucks were loaded, this data structure would have allowed us to traverse up the tree from the leftmost node until we landed on a package that was in the current truck to find the next delivery address. That process would allow the Nearest Neighbor algorithm to run in O(n) time, whereas our chosen data structure forces the algorithm to run in O(n^2) because distances are not considered. A negative aspect of using a binary search tree is that it could very easily become unbalanced as additional packages, trucks, and cities are added. The chosen algorithm will general remain very balanced.

Another choice for the data structure could have been a linked list. Linked lists are great for scalability being that that don't have to have a given size at creation. Additional added packages can be continually linked to other nodes. It would not have worked so well for using the lookup function. Being that the packages would be linked in a linear fashion, a lookup would always be O(n). The advantage of dividing the packages into groups, as we did in the chosen data structure, would be lost.

The Nearest Neighbor Algorithm

After consideration of several named algorithms, the Nearest Neighbor algorithm was selected for this project due to the small number of packages and addresses. Nearest Neighbor follows the heuristic of making the optimal choice at the present moment in time, without consideration of what came before or what comes next. Since the WGUPS delivery area covers a small number of zip codes and short distances, the Nearest Neighbor algorithm works nicely.

The steps for executing the Nearest Neighbor algorithm are as follows:

A. Load the trucks.
    a. To load the trucks, certain constraints must be considered first. These are:
    1. Some packages may only be delivered on a certain truck. These can be placed immediately.
    2. Some of the packages are delayed and may not begin delivery until after a certain time. These are placed on the second or third truck.
    3. One of the packages has the wrong address and will be placed on the third truck to give WGUPS time to correct the address when information is available without delaying the delivery of the other packages.
    4. Several packages must all be delivered by the same truck, but do not have any other significant constraints. These will be loaded onto the first truck.

   b. Once those constraints are dealt with, the rest of the packages are selected based primarily on their address proximity to the addresses of the other packages currently loaded on the trucks. This philosophy allows the deliveries to succeed without surpassing the 140 total mile constraint defined by the project task. Delivery deadlines are also considered in this step.
   c. Trucks are loaded using a list for each truck

B. Find the delivery route for each truck using the Nearest Neighbor algorithm.
   a. An empty list is created for the delivered packages for each truck. Each of the addresses of the packages on the truck are compared to the current address of the truck (each truck begins at the WGU hub) unless the package is already on the delivered packages list. A variable is maintained to keep track of the address with the shortest distance.
   b. Once the truck package addresses have been looped through and compared, the address with the shortest distance is added to the delivered packages list and that address becomes the current address of the truck.
   c. This loop continues until the delivered packages list is the same length as the truck list, updating a total mileage variable as it runs

This algorithm runs in space-time complexity of O(n^2) due the need to run through the truck list for each of packages. It is the behavior of this algorithm which drives the overall complexity of the program to be O(n^2).

Nearest Neighbor Algorithm Pseudocode

The algorithm runs in the run_route.py. For the sake of this pseudocode, the word 'truck' is used to signify any of the three deliveries (being that there are only 2 real trucks). Truck 3 is assumed to be the second trip of truck 1

- `current_hash = get current hashmap from csv_to_hashmap module`
- `DistData = get current distance dictionary from csv_to_hashmap module`
- `Manually Load Trucks using philosophy from The Nearest Neighbor Algorithm step A above`
  - `truck1 = ['7', '10', '13', '14', '15', '16', '19', '20', '27', '29', '30', '31', '34', '35', '37', '39']`
  - `truck2 = ['1', '2', '3', '4', '5', '6', '8', '12', '17', '18', '22', '25', '28', '32', '36', '38']`
  - `truck3 = ['9', '11', '21', '23', '24', '26', '33', '40']`
- `Run Route`
  - `Set current_location to WGU hub address`
  - `Set initial least distance to 1000`
  - `Set route start times`
    - `Truck1_start = 8am (no time constraints), truck2_start = 9:05am (due to delays), truck3_start = time truck1 returns to hub after delivery`

- Run truck_route for each of the three trucks, accepting three parameters (truck = list of packages, start = route start times, whichTruck = truck1, truck2, or truck3)
  - address_correction = False (whether or not package #9 has had it's address corrected. Starts out false)
  - while route length less than truck list length:
    - if pkg #9 is on truck list AND current time >= 10:20:
      - change pkg #9 address in current_hash to correct address
      - address_correction = TRUE
    - for each pkg in truck:
      - if package is not already delivered,
        - if package ID is not 9, OR (if it is 9 AND address_correction == TRUE):
          - get package address from current_hash
          - check miles from pkg to last delivery address (begins at WGU Hub)
          - if miles < current least distance:
            - update least_distance variable to equal miles
            - update next_location to equal pkg address (this ensures that the address stored in next_location at the end of the loop is the address closest to the last delivery address
            - update current_pkg to equal pkg ID (this ensures that the value stored in current_pkg at the end of the loop is the package ID that is closest to the last delivery address)
        - next iteration of for loop
      - finish for loop
    - append route list with value stored in next_location after loop finishes (this will be an address)
    - append delivered package list with value stored in current_pkg after loop finishes (this will be a pkg ID)
    - add least_distance to total truck distance
    - compute time of delivery (start time + (total truck distance / 18))
    - store delivery time in the hash table for current_pkg
    - set current_location = next_location (this adjusts the current address of the truck for the next interation of the while loop)
    - set next_location to [“”] for next iteration of while loop
    - set least_distance back to 1000 for next iteration of while loop
    - next iteration of while loop
  - finish while loop

    - calculate and distance from current location of truck back to WGU Hub and add it to the total truck distance
    - return [route list, truck distance, delivered package list, start time]
- Print package status information to the console for single or all packages (as requested by user)

Upon completion of the program, all packages are delivered on time and within the 140-mile total distance constraint. With the current set of packages, this program completes all deliveries with a total distance of 115.1 miles.

Efficiency and Scalability

The program runs very efficiently, with a worst-case scenario of O(n^2) complexity. With n representing the number of packages on each truck and, being that each truck can only hold a certain number of packages (in this case, 16), we could scale the total number of packages upward without significantly increasing the runtime. We would simply have to increase the number of trucks, or each truck would have to make multiple trips to the Hub to reload. This is the greatest strength of the algorithm. Even if we increased the number of packages to 1000, the number of trucks needed would be a constant and the run_route function would still run in worst-case time of O(16^2). It would simply have to run one more time for each additional truck.

Ease of maintenance is another strength of the algorithm. As we add packages to the csv file, the algorithm needs no adjustment. Package information would automatically be pulled into the hash map data structure and the distance dictionary for use during the run_route function.

This execution of the Nearest Neighbor algorithm is not perfect. Its primary weakness is that it does not take total distance into consideration. Nearest Neighbor, by definition, accepts the optimal solution at the present moment. The closest address to its present location is determined to be the next delivery address. With such a small number of packages within a relatively small delivery area, this works. If the WGUPS delivery area was much larger, Nearest Neighbor would be much less effective at maintaining a minimal mileage constraint. The process of always choosing the address that is closest to the current location would keep mileage low at first. But eventually, the truck could be led extremely far out in one direction or another, with exceedingly longer distances to the remaining addresses.

Alternatives to the Nearest Neighbor algorithm that could have been considered include the Brute Force and Dijkstra's algorithms.

Brute Force could have been used to recursively check every possible route for each truck list, resulting in a guaranteed shortest route. This comes at a price though, as the algorithm runs at a time complexity of O(n!) vs the O(n^2) of the chosen Nearest Neighbor.

Dijkstra is best used if the data structure chosen is a graph, using graph and vertex classes. The algorithm computes the shortest path from a given starting vertex to all other vertices in the graph (Zybooks, 2019). This is another algorithm that would eventually return the shortest overall route for the packages on the truck. In order to use it, the data structure would have to be the much more complicated graph structure. And the run time is much greater than Nearest Neighbor. If the number of

packages is expected to scale to very large numbers, or if the distances (from added cities) scales greatly, this algorithm would be preferable due to the accurate determination of the overall shortest route. Nearest Neighbor tends to become less accurate in determining overall shortest route as more packages and cities are added.

If I had to do the project again, I would have chosen to call the run_route function after the time was selected by the user. That time would be passed in as an argument and used as a comparison to the current time as the delivery progresses. This would allow the delivery status (at hub, en route, delivered) to be stored in the hash table as each package is selected and printed to the console after run_route finishes. As the code stands now, the run_route function is called first. After the time of query is input by the user, each of the packages must be looked up again and its delivery time compared to the user input time to determine if it is at the hub, en route, or delivered. Since the run_route function is already cycling through the packages, this extra loop could have been avoided by incorporating it into the run_route function itself.

Sources and Citations

Learn.zybooks.com. (2019). zyBooks.
https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/6/section/12