Core Algorithm Overview

**Stated Problem:**

The purpose of this project is to determine a practical route to deliver packages for the Western Governors University Parcel Service (WGUPS) system to minimize mileage on WGUPS trucks, using the python scripting language (vsn 3.9), and also meeting several special criteria including guaranteed delivery times, incorrect addresses, and delays in delivery to WGUPS hub. Our solution to this problem begins by sorting the trucks based upon the aforementioned criteria, with special attention to guaranteed delivery times using conditional statements, and ends with us using the nearest neighbor algorithm to build the route. We are using the nearest neighbor algorithm for algorithmic simplicity, ease of understanding for end users, and ease of scalability. What this algorithm does is very simple; it takes you to the nearest possible destination from your current one. This is the natural way most people in the field would process the problem, and will thus minimize any employee disgruntlement from changes to how they do their work.

**Algorithm Overview:**

the nearest neighbor algorithm works like this:

1. The current location is determined, and a list of possible destinations is defined.
2. The destination list is sorted by which is closest to the current location in terms of raw mileage.
3. The closest destination is selected, relevant packages are removed from the package list (and thus the list of possible locations), and the algorithm repeats.

The worst case runtime for this Algorithm would be O(N^3); a hypothetical best case of O(1) is not in actuality possible, as this would require the truck to not be used.

**Pseudocode:**

A pseudocode representation of the above algorithm:

1. Algorithm input: Truck package list
2. initialize location (hub, 0) and daily mileage (0)
3. initialize list of possible destinations based on address Ids of packages in truck (id\_list), populate list of distances to all destinations the truck may go to (col)
4. enter for loop for length of truck:
   1. populate list of possible destinations (id\_list)
   2. enter second for loop for length of id\_list:
      1. populate list of tuples (list\_of\_remaining\_addresses) by combining distances from col and ids from id\_list in format (distance, id)
   3. sort list of list\_of\_remaining\_addresses by minimum distance
   4. check to ensure that there are still packages on the truck; if there are, append the distance to the next location from list\_of\_remaining\_addresses[0]
   5. set destination, update package list, and proceed to next address
   6. if truck is at the final destination, calculate the distance back to the hub and add that to daily milage; otherwise, loop continues

**Methods:**

address.py

|  |  |  |
| --- | --- | --- |
| Method Name | Method Line | Method Runtime |
| \_\_init\_\_ | 6 | O(1) |
| load\_addresses | 17 | O(N) |
| return\_address\_hashtable | 33 | O(1) |
| return\_address\_list | 38 | O(1) |
| Totals |  | O(1)+O(N)+O(1)+O(1) = **O(N)** |

hash.py

|  |  |  |
| --- | --- | --- |
| Method Name | Method Line | Method Runtime |
| \_\_init\_\_ | 4 | O(1) |
| insert | 10 | O(N) |
| search | 22 | O(N) |
| Remove | 31 | O(N) |
| Totals |  | O(1)+O(N)+O(N)+O(N) = **O(N)** |

loading.py

|  |  |  |
| --- | --- | --- |
| Method Name | Method Line | Method Runtime |
| \_\_init\_\_ | 7 | O(1) |
| convert\_street\_id | 23 | O(N) |
| load\_trucks | 32 | O(N) |
| get\_first | 82 | O(1) |
| get\_second | 87 | O(1) |
| get\_third | 92 | O(1) |
| get\_table | 97 | O(1) |
| Totals |  | O(1)+O(N)+O(N)+O(1)+O(1)+O(1)+O(1)=**O(N)** |

main.py

|  |  |  |
| --- | --- | --- |
| Method Name | Method Line | Runtime |
| N/A | 17 | O(N) |
| Totals |  | **O(N)** |

routing.py

|  |  |  |
| --- | --- | --- |
| Method Name | Method Line | Runtime |
| distance\_between\_two | 14 | O(N) |
| update\_package\_list | 20 | O(N) |
| get\_distance\_in\_column | 33 | O(N) |
| get\_destination\_id | 43 | O(N) |
| get\_list\_of\_ids | 50 | O(N) |
| calc\_mileage | 72 | O(N^2) |
| N/A | 98 | O(N^2) |
| get\_mileage | 107 | O(1) |
| get\_time | 112 | O(1) |
| get\_delivered\_packages | 119 | O(1) |
| Totals |  | O(N)+O(N)+O(N)+O(N)+O(N)+O(N^2)+O(N^2)+O(1)+O(1)+O(1) = **O(N^2)** |

**Advantage of chosen algorithm:**

The advantages of the nearest neighbor algorithm (NNA) are in its simplicity, and intuitive nature. The algorithm matches with what “makes sense” so to speak to most people. Changes to the established system frequently can cause disgruntlement amongst the rank and file, who have become accustomed to doing their jobs a certain way; using an algorithm to make their routes that makes intuitive sense to them will minimize this. The NNA will consistently give a result within 25% of the Held-Karp lower bound [1]. Because of this, while the algorithm will not produce the optimal result, it will produce a result that is within reasonable distance from it, while at the same time being able to be quickly implemented, simple to maintain, and scalable within what could reasonably be expected each truck to carry with an O(N^2) runtime. Realistically, a delivery truck cannot be expected to hold enough packages for this to be a runtime in excess of what would be reasonable.

Other algorithms such as a greedy algorithm implementing a graph structure, or Christofides algorithm implementing a minimal spanning tree structure could also be used, and probably would produce a somewhat lower overall mileage.

In the case of the greedy algorithm, this would be a 5-10% Held-Kard reduction (25% vs 15-20%)[1], as well as a smaller runtime (O(N^2) vs O(N^2log2(N))[1]. In regards to Christofides algorithm, the mileage reduction gains are more significant (Approximately a 15% reduction)[1], but with a runtime of O(N^3), the runtime becomes more of an issue, and with enough packages runtime issues would be more likely to occur, especially at scale.

The immediate issue and the cause for this project was that WGUPS was having issue meeting their deadlines; this issue is solved with the NNA. If Mileage reduction is a bigger priority, this can be adjusted at a later date. By solving the deadline issue, WGUPS’ plans to increase in scale will be more practical since they likely will increase their business opportunity to do so with their reputation improving. The added complexity, however, of implementing either of these algorithms would result in a larger, longer to develop program which would be to the disinterest of WGUPS, since the potential gains from implementing this type of algorithm would be minimal at best at the type of scale WGUPS currently operates at, and even if there was a significant expansion, for the gains to really justify the additional complexity, this would require such a massive growth that it would not be realistic for several years, and even then the program could be relatively painlessly adapted to a new, more complex structure, at a later date.

**Programming models:**

Currently, the programming model for this project is limited; it only functions on a local machine in the Pycharm IDE, with no deployment plans at present; realistically, this project is at best a proof of concept. It relies on a very specifically formatted grouping of CSV files, and very minute errors in formatting could produce errors in output very easliy. For this project to function in an actual deployment, there would need to be significant additions made, such as networking, actual route output, a way to ensure accurate CSV formatting (Such as a data entry method to reduce the possibility of user error), and the sourcing of some form of host environment.

**Adaptability:**

The algorithm should be scalable with additional trucks, packages, and addresses with very little issue, assuming that the format of the CSV files is unchanged. With a realistic amount of packages per truck, our overall runtime of O(N^2) should be perfectly reasonable. Issues may arise from how loading of trucks is handled, and in how specific the CSV formatting needs to be; some form of input form would help reduce any possible errors in these issues. Assuming these are addressed, there is no reason that this algorithm would not function at a significantly larger scale than it is currently implemented for.

**Efficiency and Maintainability:**

With an overall runtime of O(N^2), this algorithm is acceptably efficient with any reasonable amount of packages per truck, be that 16, 160, or 1500 packages, since the algorithm is run per truck, the practical upper limit of how many packages a truck physically can deliver per day is going to be reached well before any runtime issues come into play.

As for maintainability, since all the core functions are actually rather simple, debugging and maintenance should not be a significant concern. Since very descriptive names and documentation is used, maintenance should be rather straightforward.

**Data Structures:**

This project was approached with a very Object Oriented perspective. The primary data structure used was a list of package objects, with the algorithm extracting data based upon the various objects in the lists to determine the route. Using a list of objects made the hashtable creation very convinient, as well as making structuring the program very straightforward. A weakness of how I have structured the code is that it would require manually running each truck in the routing.py file. Since there are a finite amount of trucks, this is not an issue, but in practical application this would most likely require desinging some way for the driver to initiate running of the truck or some similar way to account for varying numbers of trucks. This should scale with an increase in the number of packages, as the amount of packages a truck can reasonably be expected to deliver in one day (Or the addresses it could reasonably be expected to visit) is low enough that there should not be any issue with scaling with this data structure.

I could have also focused more heavily on using the hashtable, and also could have used a list of lists to run all of the truck objects in one passage of the algorithm. Using the hashtable more would have been equally effective as what I did, with a reduction in the runtime of some of the methods I wrote; it would not, however, have reduced the overall runtime of the program. I could also have created a list of lists of packages in a truck to address the need for running the algorithm for each truck, but this would again not reduce the actual runtime of the program, as well as increase the damage presented by inaccurately entered information.

If I had to do this over again, I likely would stress less about coming up with an optimal route. I spent an inordinate amount of time trying to perfect the algorithm to get the mileage below 100, but the more complex I made the algorithm, the worse the mileage got. Eventually I got tired of it, and came up with a simplistic algorithm that actually ended up resulting in a 40 mile lower result than the more complicated algorithm I had come up with.

**Bibliography:**

1. Nilsson, Christian. *Heuristics for the Traveling Salesman Problem*. Linkoping University, 160592857366.free.fr/joe/ebooks/ShareData/Heuristics%20for%20the%20Traveling %20Salesman%20Problem%20By%20Christian%20Nillson.pdf.

(https://bit.ly/3dfOE7V)