NHP1: WGUPS Routing Program
Jarred Wilson
Student #001277133

Core Algorithm Overview

Stated Problem:

The goal of this project is to build a system capable of evaluating and simulating the most efficient route for delivering packages logistically. The intent of the project is to be used by WGUPS for their Daily Local Deliveries (DLD) because of inconsistency in delivering packages by the desired time. An algorithm was selected and implemented in order to meet the given requirements. Data was provided in the assignment that contained a list of packages each with a list of properties, destinations, and the associated distances between all provided locations. A greedy algorithm was implemented due to its balance of effectiveness and ease of implementation. This document will provide a thorough overview of the implementation.

Algorithm Overview:

The greedy algorithm works as follows:

1. Packages are divided into lists associated with their respective trucks.
2. A comparison is performed between the current location and all destinations of packages in the truck list.
3. The closest destination is selected, the truck then moves to that location.
4. All packages with that destination are unloaded and removed from the truck list
5. Algorithm is repeated with the updated current position.

Operation time for this implementation of a greedy algorithm is in general going to be O(N^2). The data can be provided either sorted or unsorted in their respective lists. If there are no priority items on the truck, it will choose the best closest destination and will operate consistently at O(N^2) where N is the number of packages on the trucks. In the presence of priority items, it will choose the closest location from the list of priority items. Here is a pseudo-code representation of the algorithm:

Greedy Algorithm

if truck_list > 0 and time is not up:

    Provide truck_list and current_position

    Set closest_distance to 100

From data provided, Distances iterates through relevant section of the data and sets the new lowest value only from destinations of packages in truck list

for distance in list comprehension[in destinations and not current_position]

if distance is less than closest_distance: closest_distance = distance

Distances returns tuple (closest_location, closest_distance)

Use returned tuple to advance truck

unload(truck_list, priority_list, current_position)

for item in truck_list: add destination to removal list

for item in removal_list: remove from truck_list and priorit_list

current_position = closest_location

Time complexity of the algorithm above:

O(N) + O(N) + O(N) = O(N)

This process is repeated until the truck list is empty.

HashTable.py

Method	Space Complexity (Worst)	Time Complexity (Worst)
__init__	O(1)	O(1)
__get_hash	O(1)	O(1)
add (also updates)	O(N)	O(N)
get	O(N)	O(N)
delete	O(N)	O(N)

Distances.py

Method	Space Complexity (Worst)	Time Complexity (Worst)
find_closest_distance	O(N)	O(N)

distance_to_home	O(1)	O(1)

Packages.py

Method	Space Complexity (Worst)	Time Complexity (Worst)
__init__	O(N)	O(N)

Simulation.py

Method	Space Complexity (Worst)	Time Complexity (Worst)
__init__	O(1)	O(1)
reset	O(1)	O(1)
get_total_distance	O(1)	O(1)
__build_lists	O(N)	O(N)
__unload	O(N)	O(N)
run	O(N)	O(N^2)
print	O(N)	O(N)

Algorithm Advantages:

The advantages of the algorithm selected is it's ease of implementation and debugging as well as its moderately efficient results. While the greedy algorithm does not always result in the absolute shortest path, it was well suited for the data that had been provided. Overall, the algorithm quickly finds the shortest path due to its rather rudimentary evaluation of the best possible path. Generally, the algorithm works in linear time, with a worst case of O(N^2).

Programming Model:

Building and running the software was done completely on a local workstation. The data was stored in CSV files and was manipulated using built in Python 3 libraries. A network connection was not needed as there was no connection to a database or any other network connected resource. No additional libraries were installed using pip or any other tool. An IDE was not used in the construction of this project and it was executed using python 3.8 on a Linux

workstation. Since the implementation and required systems are defined as only the a local workstation, there is no practical reason to define the set of interaction semantics.

Adaptability:

This algorithm works very well to adapt to changes in the package lists and locations. Updates to the package lists result in updated destination lists. Once the destination list has been updated, it will continue to operate in the same greedy algorithm by choosing the closest destination. The program currently, however, does not adapt well to the changing of truck lists. The current implementation requires a careful evaluation and distribution manually to yield favorable

Efficiency and Maintainability:

Not all approaches to a problem are equal. While this algorithm is not the most efficient, it strikes a crucial balance of efficiency and maintainability. This algorithm is not the most efficient solution to this problem, but it is considerably more maintainable due to its simplicity. A majority of the core functions have been separated and simplified in ways that promote good maintainability. Minor changes to the truck lists can result in minor changes in overall distance the trucks are driven, but major changes can result in major differences currently.

Data Structures:

The data structures that were used in this project were primarily a number of lists containing basic information such as object identification numbers. Lists in Python are incredibly easy to implement and are very flexible data structures. Python supports list comprehensions and has built in functions that abstract a lot of the tedious iterations required. While they are not particularly the most resource efficient to use, they are light enough given the project requirements and easy to use.

Numbers from the lists were then used to store data pulled from CSV files and stored in hash tables and using the identification numbers as keys in the hash tables. This allowed for very quick and constant lookup performance.

Alternate Algorithms:

Dijkstras's algorithm would be a great solution to this problem. It results in a very efficient solution to the problem with decent maintainability. It looks deeper into what defines what the shortest path is. Dijkstra's algorithm would require that destinations were stored as nodes and that the paths between all of the nodes and their associated distances be stored as weighted edges. This approach can evaluate the shortest distance between two points as a path between two or more points.

Another possible solution to the problem would be to implement a self-adjusting algorithm that determines the route before it leaves the hub. This would allow for it to plan ahead and

load up only the packages that are along the chosen route. This would result in greater flexibility by taking advantage of a trucks ability to return to the hub at any point, as well as, allowing for more efficient distribution of packages among the trucks.

Reflection:

If I could start over on this project, I would reach out to a course instructor initially as I did not find the project requirements to be very clear. While the requirements said that there were no additional assumptions allowed, I found myself having to make assumptions to be able to move forward. Reaching out to a course instructor could have cleared up some of my questions and allowed for a quicker development time.

Also, I would have evaluated additional algorithms to allow for more intelligent evaluation of closest destination. The current implementation is very simple and only looks at closest destinations from one node away. Allowing for multiple depths in evaluating the closest location could have yielded more efficient mileage in the results.