C950 Data Structures and Algorithms II

Joseph McClintock
#004154968

Core Algorithm Overview

Stated Problem:

The intent of this project is to create an efficient package delivery system for WGUPS that determines the best route for a truck full of packages to take. The driving factor for making this program is that the current system is not working, and packages are not being delivered by the promised deadline. WGUPS has two trucks and delivers 40 packages on average every day, they would like the mileage to be under 140 miles driven in total between both trucks.

Algorithm Overview:

Requirements A and B1:

The core algorithm of the program is based on the nearest neighbor algorithm. It takes in three arguments, the addresses to visit, the packages for those addresses, and the departure time of the truck. When the algorithm begins it checks the size of the truck making sure that there are addresses that are meant to be delivered to, if no addresses are found it exits the algorithm. It then creates a list of indexes of the addresses that need to be visited. Then the core part of the algorithm begins. The first address is selected from the truck and then looped through to find the nearest valid address. Once the closest address is traveled to it is marked as visited and not allowed to be visited again. This process repeats until every address has been visited, and once done the truck will head back to the HUB. This is all done with a worst-case runtime of O(n^2), the best-case would-be O(1) but that is only possible if the truck has no addresses to visit.

Nearest Neighbor Algorithm Pseudocode

First check to make sure the truck is not empty.

```
if truck is empty, then return
```

Creates a list of address indexes that need to be visited.

```
for size of truck:
    add index of address from truck to truck_index list
```

If the truck is not empty the core algorithm begins first selecting the first address. Current index equals the current address index that the algorithm is on.

```
for size of truck
    current index equals truck[address][0]
```

Next the distance to travel to other addresses is looped through finding the closest one.

```
for length of current address in truck
```

Then it confirms that the closet address is a valid address to visit and makes that distance the current smallest.

```
if current smallest > current distance from truck > 0
    if address index in truck index list and index not 0
        smallest = current distance
        current index = address index
```

If the size of the truck index list is larger than one, then that address index is removed from the list and is moved in the truck to indicate it is the current address we are at.

```
if truck indexes > 1
    remove current index from truck index
    for size of truck
        if truck[index][0] equals current index
        move address to current index + 1
```

And lastly a statement is printed out to the console saying that the packages(s) have been delivered to that address.

```
print address delivered to and current time
```

The worst-case run time for the whole algorithm is O(n^2). If the truck that is entered is not empty, then it will always execute in the worst-case time.

Development Environment:

Requirements B2:

The program was made with a mixture of Visual Studio Code and PyCharm and written in python. The whole program is running through a command-line interface. All the data that the program uses is pulled from CSV files and stored on the user's local machine, no databases or external resources are used.

Space-time and Big-O:

Requirements B3:

HashTable.py

Method	Space Complexity	Time Complexity
__init__	O(n)	O(n)
get_hash	O(1)	O(1)
insert	O(n)	O(n)
get	O(n)	O(n)
delete	O(n)	O(n)

ReadCSV.py

Method	Space Complexity	Time Complexity
get_hash_map	O(1)	O(1)
get_first_deliveries	O(1)	O(1)
get_second_deliveries	O(1)	O(1)
get_third_deliveries	O(1)	O(1)

Distance.py

Method	Space Complexity	Time Complexity
organize_truck	O(n^2)	O(n^2)
miles_to_time	O(1)	O(1)
set_packages_en_route_and_depart_time	O(n)	O(n)
set_delivered	O(n)	O(n)
organize_truck_route	O(n^2)	O(n^2)
get_first_truck	O(1)	O(1)
get_second_truck	O(1)	O(1)
get_last_deliveries	O(1)	O(1)
get_package_distance	O(1)	O(1)

Main.py

Method	Space Complexity	Time Complexity
Line 16	O(n)	O(n)
Line 67	O(n)	O(n)

Adaptability:

Requirements B4:

The main elements of the algorithm are made with adaptability in mind. The algorithm allows you to enter any number of addresses to visit, but since the run time of the algorithm is O(n^2) the runtime can scale poorly. One of the less adaptable features is that you need to manually organize the trucks, but once the trucks are loaded everything works and the shortest path is found. Other less adaptable parts of the program are some of the for loops, they are hardcoded to the current size of the package list which is 40. A simple fix for that would be to add a size function to HashTable.py that returns the size of the given hash table.

Efficiency and Maintainability

Requirements B5, I1, I2:

The core algorithm has a runtime of O(n^2), and although it might not be the most efficient algorithm it is quite maintainable. It is just two for loops and two if statements that handle the logic of deciding what addresses to visit. Some of the simpler parts that do not need to be included in the core algorithm are in their own functions to help keep things readable. Variable names are made with the thought that other programmers might need to update or change the program and each function has its own short comment describing what the function does and its time complexity. The algorithm allowed the truck to deliver the packages under the 140-mile requirement and every package was delivered in accordance with their delivery specifications and delivered on time. All of this is verifiable in the user interface where the total truck miles are shown at the end of the simulation and if you search for the packages, it will show their delivered time.

Other Algorithms

Requirements I3, I3A, J

Two other algorithms that would have been able to solve the problems posed by WGUPS would have been Dijkstra's algorithm or A star. The main benefit of using Dijkstra's is its lower run time which is O(n log n) compared to my current algorithm which is O(n^2). It is also quite simple and well documented on. Dijkstra's uses a graph to determine the shortest path instead of a list used in my algorithm. A star is a version of Dijkstra's algorithm that uses heuristics to find the shortest path. It is one of the fastest pathing finding algorithms if you have good heuristics but if you have poor heuristics then it can become slow.

If I had to do anything differently it would be how I approached developing the algorithm. I chose to develop my own algorithm as opposed to looking up any standard pathfinding algorithms. Although I learned quite a bit in the process, I ended up spending more time on it than I believe was necessary. Reflecting on this portion I should have just used Dijkstra's for its simplicity and efficiency.

Data Structure

Requirement D1, B6, K1, K1A, K1B, K1C, K2, K2A:

There are only two data structures used in the program, lists and the created hash-table. I found no use for any other data structure while making the program. The created hash-table has four key features built into it, a get_hash, insert, get, and delete functions. The get_hash function is used to create the key for the inserted data. Insert allows the user to insert data with a key such as a package ID and a value, delete allows the user to delete the data by using the key. And lastly, the get function allows the user to return a hash-table value with a key such as a package ID, the time complexity of the get function never changes no matter the size of the hash-table. The size of the hash-table has a default size of ten, which is then dynamically updated as more items are added to the hash-table which causes the hash-table to take up more space. The hash-table is not affected at all with the addition of more trucks and cities. The only thing that affects the hash-table size is the number of packages.

The main strengths of using this data structure are that it allows the user to search in O(n) time and its simplicity. But there are drawbacks, a large amount of data can cause unavoidable collisions which can cause the hash-table to become inefficient. Another drawback is you must parse through the hash-table if you want to search for a specific value if you do not know what key it belongs to. With the hash-table, the values were updated on time allowing the truck to travel under the allowed 140 miles. All the packages were delivered on time, delivered in accordance with their directions and specifications. And has an easy-to-use search function that allows the user to search with a key. When the user searches in the main interface it displays all the appropriate values from the hash table with no problems.

Lists were used to hold certain packages that belonged to each truck. Those lists of packages are then searched for addresses that the packages belong to and then those addresses are added to another list. The address list is then used to tell the trucks which address to go to. If I could go back, I would try to make use of more types of data structures such as queues or graphs. An ordered queue of packages instead of a hash-table would have allowed for easier delivery. And the use of a graph would have allowed me to use Dijkstra's algorithm.

References and Sources

My primary source of information is from the Zybooks that goes along with the class.

Wikimedia Foundation. (2021, December 13). *A* search algorithm*. Wikipedia. Retrieved February 6, 2022, from https://en.wikipedia.org/wiki/A*_search_algorithm