C950 WGUPS Algorithm Overview

Isom Brown

ID #004478495

WGU Email: ibrow71@my.wgu.edu

Date 11/14/2022

C950 Data Structures and Algorithms II

# Introduction

This project required me to implement an algorithm that finds a delivery route for a hypothetical delivery company (WGUPS) given a specific set of restraints. The algorithm’s goal was to have the company make all its deliveries on time with its delivery trucks logging less than 140 total miles.

# A. Algorithm Identification

The self-adjusting algorithm in my project used to deliver my packages was “start\_route” and was associated with my truck class. The algorithm is essentially a nearest neighbor algorithm. It decides which package should be delivered based on which package's address is closest to the delivery truck's current location. The algorithm does this until all the packages loaded onto the truck have been delivered.

# B1. Logic Comments

1. Creating data structures to store data from WGUPS package CSV file and WGUPS distance table
   1. This includes an adjacency list (“list\_of\_locations”) for the distance table
   2. It also includes a hash table (“the\_hash\_table”) for the package information
2. Create a class for packages that will allow you to create multiple instances of a package object which will be stored in the hash table
   1. Packages need to store all information relevant to the package (i.e. package ID, delivery address, delivery deadline, etc.)
3. Once the data structures are in place and a package class has been established, the files can be read from row by row, and their data stored in their respective data structures
   1. Packages will be created from individual rows in the package file and then added to the hash table
4. A delivery truck class needs to be created.
   1. Trucks need to be able to hold a package manifest (list of packages to be delivered), the distance traveled, and the “current time”
5. Three instances of delivery truck objects will be created to deliver packages.
6. After the trucks are created, they are manually loaded to help deal with restraints outlined in the project instructions.
   1. To load the trucks, packages are passed from the hash table to the truck’s manifest using package ID #.
7. With the trucks loaded an algorithm to establish a delivery route needs to be created. In my project, the “start\_route” function outlined in Section A is used to do this.
   1. This function is associated with my truck class. The algorithm is essentially the nearest neighbor algorithm. It decides which package should be delivered based on which package's address is closest to the delivery truck's current location. The algorithm loops through the truck’s manifest using a while loop until all packages loaded onto the truck have been delivered.
      1. As the truck makes its delivery its current time and distance traveled are updated
8. The algorithm has one truck start making deliveries at 08:00 to ensure packages with early delivery deadlines get to their destination on time (all trucks will be using “start\_route” function to make deliveries)
9. A second truck starts making deliveries at 09:05 so it can take packages arriving at the hub late, with early delivery deadlines, and still get to their destination on time
10. The third truck will start making deliveries at 10:20, once package #9 has its address updated and truck first delivery truck has returned.
11. Once all trucks have completed their routes. A user interface will be printed out on the console. Allowing program users to look at the delivery status of all packages or the status of specific packages at any given time
12. Based on the user’s input one of three functions will be selected.
    1. The first (“get\_all\_package\_info”) lets a user simply look at the total mileage traveled by the trucks and the delivery time of each package
       1. The mileage will be calculated by adding together the distance traveled by each truck
       2. The delivery time of each package will be obtained by iterating through the hash table and pulling each package from it and printing the package’s data to the console.
    2. The second (“status\_of\_package\_given\_time”) will take a package ID and time from a user’s input.
       1. It then pulls the package from the hash table using the package ID and stores it in a temporary package object.
       2. Next the time the user entered is compared with the package’s delivery time and the time the package left the hub, to determine the status of the package
       3. Lastly the status of the package is printed
    3. The third (“status\_of\_all\_packages\_given\_time”) will take a time from the user via the console.
       1. It then iterates through the hash map checking the time the user entered with each package’s delivery time and the time the package left the hub, to determine the status of the package.
       2. Lastly it prints the results.
    4. Users can look up the status of different package(s) at different times until they choose to exit the program.

# B2. Development Environment

This program was developed on a Windows 11 machine. The integrated development environment used was PyCharm (2022.2 Community Edition) by Jet Brains. The programming language used was Python version 3.10. All files used were stored locally.

# B3. Space-Time and Big-O

The time complexity of my main function or method is O(N). The space complexity of my main function or method is O(N^2). For a breakdown of the space-time complexity of individual segments of code see the comments in the project itself.

# B4. Scalability and Adaptability

In the project, I created a hash table to store packages. The hash table I created uses chaining to handle collisions. Chaining handles hash table collisions by using a list for each bucket or element, where each list may store multiple items that map to the same bucket/element. This essentially creates a nested 2-D array and allows the hash table to effectively handle an increase in the total number of packages without changes needing to be made to the code. The list used to store addresses and distances can easily increase or decrease in size as well.

While some of the data structures in my project can easily be scaled, some of the segments of my project would struggle with scalability/adaptability. One segment would be the loading of delivery trucks. Currently, the delivery trucks are loaded manually to achieve efficient results and meet the goal of the project. If the number of packages needing to be delivered changed, then the methods/functions that load the trucks would have to be manually changed.

# B5. Software Efficiency and Maintainability

The overall time complexity of the program is O(N) meaning as the problem grows larger the time it takes to solve the problem will increase linearly. The overall space complexity of the program was O(N^2). This was primarily due to the self-adjusting data structures (i.e. hash map used to store package data). The space-time complexity of the program is reasonably efficient, and the program requires little maintenance besides the manual loading of the delivery trucks.

# B6. Self-Adjusting Data Structures

For this project to be scalable and to have some level of adaptability self-adjusting data structures needed to be created. The primary example of this would be the hash table used to store packages. The hash table I created uses chaining to handle collisions. Chaining handles hash table collisions by using a list for each bucket or element, where each list may store multiple items that map to the same bucket/element. Allowing the hash table to effectively handle an increase in the total number of packages without changes needing to be made to the code.

# C. Original Code

\*Refer to project code\*

# C1. Identification Information

\*Refer to project code\*

# C2. Process and Flow Comments

\*Refer to project code\*

# D. Data Structure

The self-adjusting data structure, such as a hash table, which can be used with the algorithm identified in part A to store the package data in my project is “the\_hash\_table.” It is an instance of a hash table created from my “packageHashTable” class. It provides each instance of a package a key hash value based on the package’s ID number, which will be used to sort where the package is placed in the hash table.

# D1. Explanation of Data Structure

Rather than store the data for each package separately. A package class was created. This allowed me to wrap all the data/information associated with a specific package into one object. Once I was able to create a package, a simple hash function was used on the package’s ID number to determine where in the hash map the package would be placed. The package and its associated information can then be retrieved by looking at the key-value pair of the package’s ID number and the package itself.

# E. Hash Table

\*Refer to project code\*

# F. Look-Up Function

\*Refer to project code\*

# G. Interface

\*Refer to project code\*

# G1. First Status Check

Status of all packages at 08:45…

Text

Description automatically generated

Status of packages 1-20 (above)

Text

Description automatically generated

Status of packages 21-40 (above)

# G2. Second Status Check

Status of all packages at 09:45…

Text

Description automatically generated

Status of packages 1-20 (above)

A picture containing background pattern

Description automatically generated

Status of packages 21-40 (above)

# G3. Third Status Check

Status of all packages at 12:15…

Text

Description automatically generated

Status of packages 1-20 (above)

A picture containing background pattern

Description automatically generated

Status of packages 21-40 (above)

# H. Screenshots of Code Execution

Successful completion of code….

Text

Description automatically generated

Delivery times for packages 1-20 (above)

Text

Description automatically generated

Delivery times for packages 21-40 (above)

# I1. Strengths of Chosen Algorithm

A couple of strengths of the algorithm I chose were its ease of implementation and that it provides linear time complexity. The solution is also somewhat scalable and would be able to route a truck with more than 16 packages.

# I2. Verification of Algorithm

If you refer to section H. you can see all the packages were delivered before their deadline, with the total mileage traveled by the trucks less than 140 miles. I checked the special notes associated with each package and observed each package was on its designated truck. I also checked the special notes and confirmed no packages were sent out for delivery before they arrived at the hub. All packages delayed getting to the hub were loaded onto truck #2 and left the hub at 09:05. I confirmed package #9 left on truck #3 at 10:20 when its address was updated. Lastly, I made sure no more than two trucks were out for delivery at a time.

# I3. Other possible Algorithms

Other algorithms that could have been used to solve the routing problem are Dijkstra’s algorithm and A\* search

# I3A. Algorithm Differences

Dijkstra’s and A\* searches are similar but different. Both traverse a graph’s vertexes. However, the A\* algorithm only finds the shortest path from one specific vertex to another specific vertex, while Dijkstra’s algorithm finds the shortest path from one specific vertex to all other vertices on the graph. Both differ from my nearest neighbor algorithm in that they use a weighted graph rather than a list.

# J. Different Approach

If I were able to implement my algorithm again, I would try and look at a solution that gave my truck the shortest overall route. Currently, my algorithm seeks the next closest address to its current location. By doing this there’s no guarantee I end up with the optimum solution.

# K1. Verification of Data Structure

The hash table I implemented was able to accurately store package information without error. It also was able to look up packages being stored without error.

# K1A. Efficiency

The best-case time complexity for looking up packages in my hash table is O(1). This is when each package has its own element or bucket. As the number of packages increases and chaining begins occurring the time complexity for looking up packages becomes O(N) as you have to search through the chain or list of elements stored at the specific index.

# K1B. Overhead

The space complexity when adding new objects to the hash table would be similar to that of the time complexity. Initially, the space complexity would start out constant in the best-case scenario and then would become linear as you progress to the worst-case scenario.

# K1C. Implications

I don’t believe the number of trucks or number of cities would change the look-up time or space usage of my hash table. Packages are given a hash value based on their package ID number. I also don’t believe adding more trucks would affect my hash table. I do think overall adding more cities would increase the size of the list used to store addresses and distances, therefore increasing space usage and lookup time of addresses. Adding more trucks would also cause an increase in space usage as new truck objects would have to be created.

# K2. Other Data Structures

I think an AVL tree or priority queue could also meet the requirements identified in section D of the project rubric

# K2a. Data Structure Differences

An AVL tree is a self-adjusting heuristic data structure like a hash Table. However, in the worst-case scenario, an AVL tree performs better than a hash table. An AVL tree has a time complexity for insertion, deletion, and searches of O(log n). Making an AVL tree a better option for scaling.

A priority queue would have similar time complexity for a worst-case scenario to a hash table. However, assigning each package a level of priority could also help with automating the truck-loading process.

# L. Sources - Works Cited

References

*AVL tree*. (2021, June 4). Wikipedia. <https://en.wikipedia.org/wiki/AVL_tree>

Belwariar, R. (2018, September 7). *A\* Search Algorithm - GeeksforGeeks*. GeeksforGeeks. <https://www.geeksforgeeks.org/a-search-algorithm/>

Drake, V. (2018, May 30). *A coffee-break introduction to time complexity of algorithms*. Victoria.dev. <https://victoria.dev/blog/a-coffee-break-introduction-to-time-complexity-of-algorithms/#:~:text=Its%20time%20complexity%20depends%20on>

*graphs - Dijsktra’s algorithm applied to travelling salesman problem*. (2012, May 9). Computer Science Stack Exchange. <https://cs.stackexchange.com/questions/1749/dijsktras-algorithm-applied-to-travelling-salesman-problem>

James, J. (2016, January 22). *Python: Creating a HASHMAP using Lists*. Www.youtube.com. <https://www.youtube.com/watch?v=9HFbhPscPU0>

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Retrieved March 22, 2021, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>

*Priority queue*. (2020, May 15). Wikipedia. <https://en.wikipedia.org/wiki/Priority_queue>