C950 WGUPS Algorithm Overview

Robert Uhl

ID #010191437

WGU Email: ruhl4@wgu.edu

Date 12/2/2022

C950 Data Structures and Algorithms II

# A. Algorithm Identification

For my algorithm I used the greedy version of the nearest neighbor algorithm. This algorithm chooses its path by going to the next closest address. This is known as a greedy type of algorithm because, it calculates its choices only one step at a time and only explores one option. Because of this it is fast but doesn’t find the most optimal solution.

# B1. Logic Comments

Pseudocode Below

This method orders packages delivery by comparing what package is the closes to the truck at that given moment.

Initialize an empty list called not\_delivered

For all the packages in truck

Search for packages objects in hash table and add them to the not\_delivered list

Clear the truck package attribute of package IDs

End for loop

While there are packages in the not\_delivered list

Initialize variable named next\_address\_distance and set it to 10000

Initialize a variable named next\_package and set it to None

For all the packages in the not\_delivered list

Initialize a variable to hold the distance between the truck’s current location and the address of the current package

If the distance for the current\_package is less than the next\_pakage\_address

Then save the distance of the current\_distance to the next\_address\_distance

Then save the package object to the variable next package

Append the next package ID to the object truck’s attribute packages

Update the truck name in the package object

Remove the package from the not\_delivered list

Add the milage the truck will travel to get to the next address to the trucks attribute mileage

Change the trucks address attribute to the packages address

Add the time it will take the truck to get to the package’s address to its time attribute

Update the delivery time to the package object

Update the departure time to the package object

End For

End While

Initialize distance\_to\_hub variable to check the distance between the trucks last address and the hub address

Update trucks milage with the milage going back to the hub

Update the trucks time with the time it takes to drive back to the hub

# B2. Development Environment

I used a MacBook Pro with a 2GHz Quad-Core Intel Core i5 processor, 16GB of memory, and using mac OS 13.0.1 (22A400) software version. For my integrated development environment (IDE) I used PyCharm 2022.2.3(Community Edition). Finally for my Python interpreter I used version 3.10. memory, processor, PyCharm version

# B3. Space-Time and Big-O

Main.py: Space Complexity/ Time Complexity

deliver\_packages: O(N) / O(N^2)

main: O(N)/O(N^2)

Truck.py: Space Complexity/ Time Complexity

\_\_init\_\_: O(1) / O(1)

check\_distances: O(1)/ O(1)

get\_address\_id: O(1)/O(N)

Packages.py: Space Complexity/ Time Complexity

\_\_init\_\_: O(1) / O(1)

update\_status: O(1) / O(1)

update\_truck: O(1) / O(1)

load\_package\_data: O(N) / O(N)

Distance.py: Space Complexity/ Time Complexity

check\_distance: O(1) / O(1)

get\_address\_id: O(1) / O(N)

HashTable.py: Space Complexity/ Time Complexity

\_\_init\_\_: O(1) / O(1)

insert: O(1) / O(N)

search: O(1) / O(N)

remove: O(1) / O(N)

TOTAL: O(N) / O(N^2)

# B4. Scalability and Adaptability

This program is scalable because the packages are read from a csv file. If a csv file with more packages is loaded, then the program will be able to read in all of the packages. The chaining hash table in this program is self-adjusting allowing it to handle various amounts of packages. More truck objects can be easily created to keep up with demand. The algorithm runs in O(N) space time complexity which allows it to find solutions quickly even if the amount of packages increase.

# B5. Software Efficiency and Maintainability

The nearest neighbor algorithm used in this program has a time complexity of O(n). This allows the algorithm to stay fast if more packages are added.

I used an auto formatter to make sure my code is written to PEP8 standards. This allows for cleaner code so other programmers can update this code without any trouble.

# B6. Self-Adjusting Data Structures

The self-adjusting data structure in this program is a chaining hash table. While the number of buckets is statically assigned in the program the chaining allows for multiple key value pairs to be saved in each bucket. This allows the hash table to store as many values as needed.

# C. Original Code

Is included with this document.

# C1. Identification Information

Is at the top of Main.py.

# C2. Process and Flow Comments

Are inside of the code as comments.

# D. Data Structure

The chaining hash table is a self-adjusting data structure that can be used with the algorithm to store the package data.

# D1. Explanation of Data Structure

The chaining hash table has a method call insert. This method will take a key, and value as arguments. In the case of this program the key is the package id and the value is the package object. It then takes the key value pair and stores it in a bucket.

Then the search method takes a key argument which is the package id and will return the package object. The remove method will also take the package id as an argument, but instead of returning the package object it deletes the object from the hash table.

# E. Hash Table

The insert function is included in HashTable.py

# F. Look-Up Function

Text

Description automatically generated

# G. Interface

Text

Description automatically generated

# G1. First Status Check

9am

Text

Description automatically generated

# G2. Second Status Check

10am

Text

Description automatically generated with medium confidence

# G3. Third Status Check

12:45pmText

Description automatically generated with medium confidence

# H. Screenshots of Code Execution

Screenshots (and possibly labels) go here

A picture containing calendar

Description automatically generated

# I1. Strengths of Chosen Algorithm

A strength of the greedy nearest neighbor algorithm is that it is simple to implement. This also makes the code easy to maintain and update in the future. The other strength is that is fast at finding a solution. Since this algorithm only explores one option it doesn’t spend a lot of time trying to find the most optimal solution. Since it can find a solution quickly it can handle receiving more packages.

# I2. Verification of Algorithm

The screen shot below shows the total milage is 126.8.

A picture containing calendar

Description automatically generated

The screen shot below show all packages were delivered on time

Text

Description automatically generated with medium confidence

# I3. Other possible Algorithms

One algorithm that could have been used is the Dijkstra. This algorithm can use a graph as its data structure and saves the addresses as nodes. The algorithm starts by checking the distance from the root node to all the other nodes. We then save the shortest distance to a shortest path tree. The algorithm then looks the shortest vertex from that node and save it to the tree. It continues this process until all the nodes have been analyzed and the shortest distances between nodes are saved to the tree (“Dijsktra’s Algorithm” | GeeksforGeeks).

Another algorithm is the Bellman-Ford algorithm. This algorithm uses a bottom-up approach where is first calculates the shortest distances that has 1 edge(or vertex) in the path. It then calculates the shortest path that has no more than 2 edges. It then repeats this over and over until its complete (“Bellman–Ford Algorithm | DP-23 - GeeksforGeeks”).

# I3A. Algorithm Differences

The Dijkstra Algorithm would find a more efficient route but would be harder to implement and have a larger space complexity.

The Bellman-Ford algorithm is simple to implement but the time complexity is 0(V\*E) so larger than both the greedy nearest neighbor and the Dijkstra (“Bellman–Ford Algorithm | DP-23 - GeeksforGeeks”).

# J. Different Approach

If I were to remake this project, I would like to use pandas to pull the data right in from the excel files instead of converting them into CSVs. This would make it easier to pull the data when given an excel spread sheet.

# K1. Verification of Data Structure

Build a self-adjusting chaining hash table without any additional libraries or classes, with an insertion function.

# K1A. Efficiency

The more packages there are the more chaining that is required to store the packages. The current bucket count is 16. The hash table uses modulus to determine the bucket a package should go into. This means that if the package ID numbers stay sequential, they will be evenly distributed across the buckets. However, the time it takes to find the package in the individual buckets will increase because it will need to iterate through the list. So the time it takes to look up a package will be the number of packages divided by the number of buckets. In order to get it faster you could increase the number of buckets. Also, if our ID don’t stay sequential then this could make the distribution uneven and cause our time complexity to go to O(N), but that is unlikely it would end up being that bad.

# K1B. Overhead

As the packages grow the space needed to store them will grow at the same rate. So the space complexity is constant O(1)

# K1C. Implications

Adding more trucks and cities would increase the amount of packages in the hash table which would increase the amount of chaining that would be needed. If the chaining is evenly distributed then the amount of time the look up function needs will increase by the amount of packages divided by the amount of buckets. This is because the hash table will have to integrate through the bucket to find the key. If the distribution is off then all the packages could be in one list and the worst runtime would be linear or O(n).

The space runs for adding more package is constant, so the more we add packages the more we add space at an equal rate.

# K2. Other Data Structures

Other data structures I could have used are an Adjacency Matrix or a Binary Tree. Both of these data structure would meet the requirements.

# K2a. Data Structure Differences

An adjacency matrix uses a matrix instead of a list to store the values this allows the matrix to add, delete and search edges very quickly in O(1) but it takes a lot of space O(V^2). It also takes a lot of time to add or delete a node O(V^2) because you are adding and subtracting a row and column each time.

A binary tree can perform operations like insertion, searching and are on average very fast in O(log(n)). However, deleting a node in a binary tree is more difficult because to have to make sure to point one of the children nodes to the parent and you have to keep doing this down the tree. Binary trees are more memory efficient at storage because it does not reserve more memory than need. While hash tables can have more buckets then key, value pairs.

# L. Sources - Works Cited

GeeksforGeeks. (2022, August 31). *Dijsktra’s algorithm*. <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

“Bellman–Ford Algorithm | DP-23 - GeeksforGeeks.” *GeeksforGeeks*, 1 Dec. 2012, [www.geeksforgeeks.org/bellman-ford-algorithm-dp-23](http://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23).

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

Retrieved November 22, 2022, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>