Brian Guevara

Student ID: 001003681

Sample Core Algorithm Overview

**Stated Problem:**

The purpose of this program is to find an optimal route to deliver 40 packages, with emphasis on keeping the total mileage low and having packages delivered on time. In our problem, we have 3 trucks, 2 drivers, and an average of 40 packages with special notes. The goal of this program aims to have every package delivered in the least amount of time as possible. On top of these requirements there are parameters within our program we must account for. Each truck can only have 16 packages, travels at a speed of about 18 miles per hour, and has an infinite amount of gas.

**Algorithm overview:**

I’ll be using a common greedy algorithm for sorting the addresses by distance. The way the program works is passing in a truck (in this case a binary tree of addresses) and looking for the closest location in a list of delivery addresses (another binary tree). We will then move to the closest location and repeat the process with a smaller binary tree.

1. While the truck isn’t empty, search for the next stop
   1. Check distance to one of the possible addresses from the current address
   2. Compare distances between multiple addresses and the current address.
2. Get the distance and time difference between the next stop and current
3. Go to the new stop and repeat

**Algorithm Pseudocode**

**while address tree is not empty:**

**next stop = find closest location**

**remove next stop from the tree of addresses**

**get the difference in distance and time between current location and next stop**

**truck distance sum =+ distance**

**assign the delivery time for all the packages at this stop**

**assign next stop as the current**

**find closest location(address list, current location, lowest value:**

**if node is empty**

**return**

**else**

**check if distance between the current stop and this address < lowest value**

**lowest value = this distance**

**destination = this address**

**keep searching the tree**

**return destination**

**and package == lowest value**

**add package to final list**

**Programming Model**

In its current form, this algorithm can work across different programming models and structures. It is easy to implement in other languages like C++, Java, and even Perl (due to recursion). The data structures compatible include but are not limited to Binary Trees, Lists/Arrays, Graphs, and Hash tables. My pseudocode was written with binary trees in mind due to the need of recursion. For other structures, the algorithm would be simpler, but likely less efficient (more on that later).

The program uses simple Python packages to complete its process (csv and datetime). CSV files are easy to setup, extract from Excel files, and could be used across multiple programming languages and structures. In this instance, our entire program is run on a local machine. Currently, there are no communications over the net. However, we could implement our algorithm and methods within another program and create an GUI/interface to allow a polished UI, input buttons/fields, and more. To add on, we can integrate an SQL database to have our packages stores, updated, and added from remote locations. Currently, the program is set to just need python and will run on a computer with Python integrated into its environments. But, the potential for this algorithm (and application) can be implemented in most programming languages, models, and structures.

**Big-O Notation**

Tables will be listed in a ‘bottom-top’ structure so that Main is at the end and our Hash/Binary Tree is what we start with.

Binary\_Tree.py

|  |  |
| --- | --- |
| **Method** | **Big-O** |
| \_\_init\_\_ (for Node) | O(1) |
| get\_size | O(logN) |
| has\_package | O(logN) |
| has\_address | O(logN) |
| \_\_init\_\_ (for truck\_bt) | O(l) |
| insert(for truck\_bt) | O(N) |
| \_\_init\_\_ (for address\_bt) | O(l) |
| insert(for address\_bt) | O(N) |
| remove\_add\_id | O(N) |
| **Largest in class:** | **O(N)** |

Hash.py

|  |  |
| --- | --- |
| **Method** | **Big-O** |
| \_\_init\_\_ (for Map) | O(N) |
| getHash | O(1) |
| insert | O(N) |
| update | O(N) |
| get | O(N) |
| delete | O(N) |
| **Largest in class:** | **O(N)** |

InputDataCSV.py

|  |  |
| --- | --- |
| **Method** | **Big-O** |
| With open(‘WGUInputData.csv’) | O(N2) |
| get\_hash\_map | O(1) |
| get\_first\_packages | O(1) |
| get\_first\_addresses | O(1) |
| get\_second\_packages | O(1) |
| get\_second\_addresses | O(1) |
| get\_third\_packages | O(1) |
| get\_third\_addresses | O(1) |
| **Largest in class:** | **O(N2 )** |

Distance.py

|  |  |
| --- | --- |
| **Method** | **Big-O** |
| with open (‘DistanceData.csv’) | O(N) |
| with open (‘WGUDistanceNameData.csv’) | O(N) |
| distance\_between | O(1) |
| find\_time\_difference | O(1) |
| find\_closest\_place | O(logN) |
| get\_Address | O(1) |
| third\_optimized\_truck\_list | O(1) |
| **Largest in class:** | **O(N)** |

Packages.py

|  |  |
| --- | --- |
| **Method** | **Big-O** |
| set\_initial\_time | O(logN) |
| set\_delivery\_time | O(logN) |
| while(get\_size(packagenode)) | O(N) |
| 3 for loops | O(3N) |
| get\_first\_finish\_time | O(1) |
| get\_second\_finish\_time | O(1) |
| get\_third\_finish\_time | O(1) |
| total\_distance | O(1) |
| **Largest in class:** | **O(N2)** |

Main.py

|  |  |
| --- | --- |
| **Method** | **Big-O** |
| print\_lines | O(1) |
| main | O(N) |
| **Largest in class:** | **O(N)** |

Due to the use of lists, the whole program has an Big-O of about **O(N2 )**. Despite having much ‘better’ algorithm complexities throughout my program, the nested loops can seriously drag out performance if our package list was to increase substantially. Performance would suffer if we added a lot more addresses as well… To accomplish this, we would add new addresses to csv files and more packages to the main input file.

Still, the use of binary trees to insert packages into the trucks and use of binary trees for our algorithm create a very efficient program that will handle a larger input just fine. Our processes are still very efficient, although our input is not. The algorithm itself has a Big-O of **O(log N)**

**Changing Market and Scalability + Efficiency and Maintainability**

Given this program is intended for delivery services/companies, it will stand well against changes in the market/industry it is in and changes in the world economy. People will always need their mail. Delivery routes are subject to change. Above all, they’ll want it as soon as possible and delivery companies want to exhaust as little gas (miles) as possible. As long as we have a federal government that provides a mailing system, there is a market for our program. The real issue becomes scalability. For the most part this program can scale up just fine. Initial input of reading the csv file line by line is one of the most inefficient parts of the program. It has an O(N) space-time complexity. What matters is our process to compute the total distance, set delivery times, and go through every delivery address are efficient. We use binary trees to check multiple address ‘at once.’ Our logarithmic process can handle an increase in scalability just fine.

For maintainability: In the event we have more addresses added to our route, we can add them without much issue and hardcode their values into a csv file. But when making my program I realized that the algorithm doesn’t prioritize 10:30 deliveries. I had to setup a package to be first in the delivery queue, otherwise it would have been late! With this in mind, it would make sense to readjust our algorithm to handle those issues automatically. Having to manually hardcode these values could give us undesired results. In the end, the program is easy to add more packages and (if needed) add more addresses.

If we further developed our program, we could make another application to create the package file and add it to a csv file. I initially wanted to read the information directly from the excel file provided, but that would require third party packages and would not be good for maintainability and accessibility. To add on, every value would have needed to be hardcoded. What we have, right now, is ideal: simple and efficient.

**Data Structure Explanation**

For this project, I decided to use a binary tree data structure. This structure is very efficient for sorting and finding data. I believe this data structure to be more efficient that arrays, graphs, and more. This structure is built by nodes. Those nodes contain: data, a link to the left node, and a link to the right node. **1**

We start by inserting a value as the root/start to our binary tree. Then we add another value, compare it to the current value, and move it to the right/left node depending if the data is more/less (respectively) than the current node’s data. Values are found by looking at the current node, and checking if the value we search for is less than or greater than the current node. These conditional checks are running ‘at the same time’ which is why these trees are so great. They can process multiple pieces of information in the same iteration. This is what makes finding our closest location so efficient. We use trees to search through the values and compare them to all previous ones checked.

**Algorithm Selection**

I went with a greedy algorithm when trying to optimize the route for the trucks. The main reasons for the greedy algorithm are simple: it is easy to setup, good for scalability, and efficient. Creating it for a binary tree felt very intuitive. I set up the algorithm to simply check the distance of a possible address and a passed in address (current location). Then, I would compare the distance to the left and right nodes… This would continue until every node is searched and distance is compared. We then go with the immediate, best option. While greedy algorithms have their benefits (and work well with the benefits of binary trees), they still have the disadvantage of picking the best option right now. **2**

**Other Algorithms and Data structures**

Another algorithm I could have used was Dynamic Programming. If my goal was to do this, I probably would have setup the input Binary Trees in the same way (one for each truck). Then, I could get different zip codes (I would pick ones that are a bit far from each other) and start my recursion algorithm with addresses around there. This may give a better result in miles as it would give us more possibilities than the simple greedy algorithm, but would increase the complexity of the ‘finding’ algorithm.

Another algorithm type I could have used was a (non-greedy) Heuristic. I think this option would work best with a graph data structure as we would need more factors (parameters) for our non-greedy Heuristic. With this approach, I would probably only have the addresses in graphs and then use the distance variable to find the closest values between all addresses. The edges in a graph data structure could easily collect that data, especially since graphs use vertices that can create a more in-depth view of our routes.**3**  Packages would likely be sorted as they are now (linearly). I can imagine we’d have the best possible route this way, but the efficiency and sustainability would not pan out well.

On top of graphs, I could have used another Hash table for the addresses and packages. For packages, the key I would use would likely be the truck that that the package is to be delivered on. For the address hash table, I would set the key to either be a zip code, or create a number for each quadrant of the SLC map.

**Different Approaches**

One of the things I would’ve liked to do differently is finding a better method to insert the packages into the correct trucks. In the first sense, I mean the actual file we import from. It would have been great to read from an Excel file but it was vital to stick to base packages from Python. CSV files are a great tool, but most people don’t use CSV files. In another sense, from testing I saw that the non-special (EOD delivery and no special notes) packages had a very big influence on mileage, regardless of algorithm. That in itself can be an algorithm/method.

Another change to add on to that would be the actual binary trees I have for the packages. The address binary trees would be the same, but I could have at least 4 separate trees. 3 for the 3 trucks and their needed packages, and a 4th tree for non-special packages. After sorting the packages and build the 3 address trees for the trucks, I can sort through the 4th tree and look for the best places for each package (whether it’s truck 1, 2 or 3… and where in the route they would be).

**Sources:**

1. *4.1 Binary Trees*. Zybooks. <https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/4/section/1>
2. Jimin Khim, Karleigh Moore, and Eli Ross. *Greedy Algorithms.* Brilliant.org. <https://brilliant.org/wiki/greedy-algorithm/>
3. *6.1 Graphs: Introduction*. Zybooks. https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/6/section/1