C950 WGUPS Algorithm Overview

Chelsey Machin

ID #001221846

WGU Email: cmachi2@wgu.edu

June 23rd, 2022

C950 Data Structures and Algorithms II

# Introduction

This project’s intended purpose is to develop an algorithm and code to present a solution where a specified number of theoretical packages are loaded onto trucks and delivered according to specific parameters and delivery deadlines. The total mileage of the delivery trucks should remain under 140 miles. The project uses distance, address, and package data provided directly by WGU, but could easily be used with any similarly formatted data available.

# A. Algorithm Identification

The project utilizes a greedy, nearest neighbor algorithm (Lysecky & Vahid) to determine the most efficient delivery manner for all the packages currently on a designated delivery truck. The greedy portion of the algorithm makes different decisions for loading based on the special notes section of the package data. The nearest neighbor portion generates a list of closest distance addresses sorted by distance away from the current address and then loads and delivers the packages according to that. The algorithm is logically explained below step by step:

1. All the packages are visited to check if they have special notes
   1. If they need to be delivered early, they are loaded onto the truck leaving first
   2. If they need to be delivered with specific other packages, they are loaded into the second truck
2. The first loaded package address from the first truck is then given to the nearest neighbor function as a string
3. It creates an empty list to store distances between addresses
4. Every package in the table is visited
   1. If the current package’s address matches the input address, it is skipped
   2. Otherwise the distance between the input address and the current package’s address is found in the data and added to the empty distances list
5. The distances list is sorted and returns a list of package IDs and distances sorted by closest to the given address
6. The packages from that list are then loaded onto the first truck until it reaches its package limit (16)
   1. Repeated for the second truck
7. The remaining packages are loaded onto the third truck in their nearest neighbor list order
8. Packages are delivered one by one on each truck according to the same nearest neighbor algorithm, and get a time of delivery update and a status change to “delivered”

# B1. Logic Comments

This function is the greedy portion of the algorithm which goes through the packages to determine which need special considerations for loading, then executes that first loading. After, it gets the first package of a semi-loaded truck and uses the get\_nearest\_neighbor function to generate a list of nearest packages that would be the most efficient to load together with the first package already on the truck. The nearest neighbor list is then iterated over and loaded one by one onto the indicated truck until that truck has 16 packages on it. At that point, it ends. This function can be reused for however many trucks you need to load. Any remaining packages are loaded into the last truck using the same method to generate the most efficient route to deliver them as well, based on the nearest distances of the package addresses. After loading, the truck’s packages are gone through one by one and delivered according to the nearest neighbor’s address list. For delivery, each package is marked as “delivered” and given a time that it was delivered at. This data is stored in the hash table for packages as part of the package’s information.

**Get\_truck\_load\_lists():**

For package in packages\_table:

If package.special\_notes = ‘can only be on x truck’:

Truck\_x.add(package)

Continue

First\_package\_truck\_x = packages\_table.lookup(truck\_x[0])

Nearest\_neighbor\_list = get\_nearest\_neightbors(first\_package\_truck\_x.address)

For package in nearest\_neighbor\_list:

If package.loaded:

Continue

Else:

While (length(truck\_x)) < 16:

Truck\_x.add(package)

Break

**Get\_nearest\_neighbors(input address):**

distances = []

For package in packages\_table:

if package address == input address:

Continue to next package

else:

distance = distance\_lookup(input address, package address)

distances.add(distance)

distances.sort()

return distances

# B2. Development Environment

I utilized Python 3.8 as the application language, with the following utility modules: tabulate, datetime, and csv. I used PyCharm as an integrated development environment and GitHub as a version control repository source. The program has been run and tested successfully on a desktop PC running the latest Windows 10 Home OS (19044.1766), and on a MacBook Pro running the latest version of macOS Monterey (12.3.1). Both systems have 16 GB of memory and storage drives of 1 TB.

# B3. Space-Time and Big-O

Each function and data structure has a comment describing the worst-case runtime complexity (Big-O) or the space-time complexity accordingly. The entire application has a Big-O of O(n3) which means the worst-case runtime complexity of a function is O(n3). The function with this runtime complexity is the package delivery function as it utilizes a function that is O(n2) (get\_nearest\_undelivered\_package) within it’s own while loop, increasing the complexity of the entire outer function to O(n3). The space-time complexity for the largest data structure is O(n2) (distance\_list) as it contains a 2-dimensional array.

# B4. Scalability and Adaptability

This program has a lot of scalability but would have some strict parameters around how adaptable it could be. It is highly dependent on data being imported by provided and pre-formatted CSV files. If the input data follows the formatting of the original WGU provided data, you could scale this up to as many packages and delivery vehicles as you wanted without much change in the code at all. The nearest neighbor algorithm will still work in the same way to determine the closest list of packages for all packages in the table. The algorithm for loading and delivering packages would need to be run many more times depending on how many packages are added to the table. Each truck holds up to 16 packages so a new truck load list of packages would need to be generated for each new truck/set of 16 nearest packages. This would increase the amount of time it would take to load all the packages as more nearest neighbor package lists would need to be generated as the number of trucks needing to deliver them increased.

# B5. Software Efficiency and Maintainability

The software is written in Python, which is one of the oldest and most stable programming languages available (Balasubramaniam). It doesn’t use any other external libraries other than pre-installed modules in the program that are long term stable release versions. It will be easy to maintain with very minimal updates for years to come. This also means it loads its dependencies quickly and does not take up a lot of space on a device while executing, making it an efficiently running program. The worst-case runtime of the entire application is O(n3) no matter how many packages you add to it, meaning it will still be efficient/run just as quickly as it gets larger, too.

# B6. Self-Adjusting Data Structures

The self-adjusting data structure implemented in the project is a chaining hash table. It scales itself up with data stored in it by simply making the linked lists in each “bucket” longer each time would-be collision data is added to it, just slightly increasing the lookup time for each “bucket” within the table (Lysecky & Vahid). This means we’ll technically never run out of space in the hash table, and it is easy to interface with the linked lists within each data storage bucket.

There are some slight cons to using this type of self-adjusting data structure including slight space waste (some linked lists could be quite long while some buckets do not even contain entries in their lists), and if the chain gets long enough, the runtime complexity could shoot up to O(n) for the bucket (Gudabayev).

# C. Original Code

All the code in this project has been written by me for the express purpose of this project. There are no errors or warnings that occur during runtime through extensive testing.

# C1. Identification Information

An identifying comment has been made in the first line of my main.py file that includes the name and student ID.

# C2. Process and Flow Comments

There are quite a few comments that detail the ideological process and the general flow of the code for the program. Variables all have a comment giving the reason/use case for the variable; functions all have comments describing the input/output/reasoning for use.

# D. Data Structure

The self-adjusting data structure that stores the package data is a chaining hash table. The hash table buckets are created as empty lists and new objects are added to the list using the Python list’s append and remove methods (Lysecky & Vahid).

# D1. Explanation of Data Structure

The chaining hash table stores each package as an object in a bucket according to its index (defined as the object’s key divided by the length of the table). All updates and removals are made according to this index. A lookup can be performed to get the index of the object and then use that to access the data object in the table to get a package’s information. Packages are loaded into the data structure by looping through the CSV file to create a package data object, then that object is added to the table using the insert function.

# E. Hash Table

The hash table in the project is implemented in Python and has an insertion/deletion/lookup/update interface through functions. It also has a get\_hash function that easily allows a hash key to be returned based on the key given.

# F. Look-Up Function

The look-up function exists in the hash table and is used as a package look-up function for the packages class. In the hash table, it is implemented by returning a bucket index for the key using get\_hash and then if that hash exists, it returns the paired value data point for that key.

Get\_hash(key):

Index = key % length(self)

Return index

Lookup(self, key):

Index = get\_hash(key)

If table[index] exists:

For values in table[index]:

If values[0] == key:

Return values[1]

# G. Interface

The interface runs on program start. It immediately reports the total mileage of the delivered packages, then asks the user to make a choice between 1: getting package updates for all packages based on an input time, 2: getting package updates individually based on a package ID, 3: exit the program.

# G1. First Status Check

Text

Description automatically generated

# G2. Second Status Check

Text

Description automatically generated

# G3. Third Status Check

Text

Description automatically generated

# H. Screenshots of Code Execution

Text

Description automatically generated

Initial program screen is seen here. The mileage of each truck and total mileage for all deliveries is displayed on the first few print lines. Then the interface options are indicated to the user. They can enter a time to see an update of all packages at that time, or they can input an individual package ID to look up the final loading and delivery times for it.

# I1. Strengths of Chosen Algorithm

The nearest neighbor algorithm that is utilized to determine a list of closest packages sorted by distance from current address has a few different strengths in regards to this project. The first is that it is non-dependent on any other data point for the packages except the address. This means it is doing a lot less to determine the most “efficient” path as it only determines efficiency based on the distance between the current address and the next, which leads to faster results and sorting. Another strength is that the algorithm functions independently of any other loading/delivery functions so it can be utilized in different ways/for different applications that need to utilize a list of nearest packages sorted by closest distance.

# I2. Verification of Algorithm

The algorithm used in my project delivers all packages with just under 114 miles driven. All packages are delivered in accordance with their special notes, and on time. There are no undelivered packages.

# I3. Other possible Algorithms

Two other algorithms that could’ve been used to determine an efficient delivery path would be Djikstra’s Shortest Path and a Breadth-First Search. Djikstra’s Shortest Path is an algorithm that finds the shortest path between a “source node” and the rest of the nodes in the data structure (a graph, typically) (Cassingena Navone). A Breadth-First search doesn’t require edge weights for the graph, like Djikstra’s does, but still finds the shortest path between graph nodes (Baum).

# I3A. Algorithm Differences

Both algorithms could be tailored to fit this program’s needs with some slight alterations to the structure of the code. For Djikstra’s path, the data structure housing the packages’ address distance from the hub would need to be made into a traversable graph (Baum). For Breadth-First, you’d also need a traversable graph of the distances for the addresses, but you wouldn’t need edge weights to make a nearest path determination for the packages (Baum). The nearest neighbor algorithm in the program is able to be a lot more flexible with the information it’s given, as it interacts with the distance table directly to compare a list of already generated distances from the hub that coordinate with package addresses.

# J. Different Approach

If I were to do the project again, I would create an additional data structure, a graph, that just contained the addresses and distances. I would then implement a version of Djikstra’s Shortest Path to try and return the most efficient route for the packages to be delivered in. I would then divide that list between the trucks and hopefully come in with a little less mileage than this implementation.

# K1. Verification of Data Structure

The hash table used in my project meets all requirements laid out in the rubric. Each truck’s mileage is reported in the first print lines of the program’s interface, as well as the total mileage for all deliveries combined. All packages are delivered on time, and none are left undelivered. The interface uses the hash table’s lookup function to get package statuses for the user.

# K1A. Efficiency

The overall complexity of the program is O(n3). This means that directly speaking, the larger the number of packages in the data source, the longer the program takes to run as it has to loop through each package in the data set to make decisions. The variable “n” in this case can be thought of as representing the total size of your package data for the program. In this implementation, there are only 40 packages, and the program runs quickly. It would take longer for a data set of 400 packages based on the program’s complexity and this concept. The hash table package look up time would not increase with every package added, but once the initial 10 buckets of the hash table start to get more filled, the lookup function will need to compare more indexes in the buckets to get the correct item, and that will increase the package look up time.

# K1B. Overhead

Increasing the size of the package data set would also require a slight restructuring of the hash table. It is currently initialized with ten “buckets”, which works great for our purposes. If we had a much larger data set, we would probably need to revisit that as we could run into many collisions based on a much larger set of packages needing to be stored.

# K1C. Implications

Changes to the number of trucks or number of cities delivered to wouldn’t affect the look-up time of the packages at all, as they’re stored in a hash table and the trucks and addresses/distances are separate data structures. Increasing the number of delivery trucks wouldn’t affect the runtime complexity of the program at all, as the algorithm assigns a maximum of 16 packages to each truck and the overall runtime complexity of the delivery algorithm (O(n3)) wouldn’t be better just by adding more trucks as it would still need to loop through the same information in the same way for each individual truck. Increasing the number of delivery trucks would also add more O(n) lists that act as “trucks” to store/deliver the packages. Decreasing the number of trucks might make the runtime complexity of the delivery algorithm worse as it would mean the trucks would need to come back to the hub and be loaded with more packages and then deliver them. This might require an additional loop in the algorithm to have the truck come back and get re-loaded and then deliver its packages again, worsening the runtime complexity to O(n4).

Increasing the number of cities to deliver packages wouldn’t change the space complexity of the delivery address distances two-dimensional array structure (O(n2), but it would need to store more address comparisons to get a larger number of distances as cities are added. Increasing the number of comparisons the nearest neighbor algorithm needs to make in order to determine the next closest undelivered package, which again would not necessarily change the runtime complexity of the delivery algorithm, but would generally take longer to get the nearest neighbor list from that part of the algorithm increasing the time it takes for the program to run slightly. Decreasing the number of cities to deliver packages to would have the opposite effect. It wouldn’t change the space complexity of the data structure, but would theoretically decrease the comparison time for a nearest neighbor list to be generated.

# K2. Other Data Structures

Some other data structures that could work are a nested array and a graph (Lysecky & Vahid). Both would work for this project’s purpose and allow us to store and lookup our packages for algorithmic purposes.

# K2a. Data Structure Differences

As previously mentioned, Djikstra’s nearest path directly relies on graph traversal so that would be a great data structure pairing if we used that algorithm instead. It would somewhat limit the amount of direct interface we have with the data. In our hash table, all records can be easily accessed and manipulated with referenced key/value pairs. A nested array faces a similar issue. Because it is really only iterable as a way to access it, all of the interface functionality would take longer (look ups, as an example) because the data structure is less flexible in how data is stored versus the hash table where the buckets can be directly accessed quickly without a whole structure comparison as in the array.

# M. Professional Communication

All language in this project outline is directed towards someone with a basic understanding of programming principles, and is made to feel direct and professional. All language and ideas are my own, and all utilized sources have been cited below.

# L. Sources - Works Cited

Baum, J. (2020, February). *5 Ways to Find the Shortest Path In A Graph*. Better Programming.

Retrieved July 14, 2022, from

<https://betterprogramming.pub/5-ways-to-find-the-shortest-path-in-a-graph-88cfefd0030f>.

Balasubramaniam, S. (2020, November). *Using the Graph Data Structure in Python*. Section.io.

Retrieved June 18, 2022, from

<https://www.section.io/engineering-education/graph-data-structure-python/>.

Cassingena Navone, E. (2020, September). *Djikstra’s Shortest Path Algorithm – A Detailed and Visual Introduction*. freeCodeCamp.

Retrieved June 18, 2022, from

<https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/#:~:text=Dijkstra's%20Algorithm%20finds%20the%20shortest,node%20and%20all%20other%20nodes>.

Gudabayev, T. (2021, May). *Data Structures: Hash Tables*. Dev.to.

Retrieved July 14, 2022 from

<https://dev.to/tamerlang/data-structures-hash-tables-e1c>.

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

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