C950 WGUPS Algorithm Overview

Samuel Diaz

ID #001361588

WGU Email: sdiaz@wgu.edu

07/03/2022

C950 Data Structures and Algorithms II

# Introduction

The purpose of this document is to lay out and respond to all the requested requirements from the rubric in a visually appealing, easy-to-follow format.

# A. Algorithm Identification

For this specific project, the nearest neighbor algorithm was deemed efficient and therefore implemented for identifying the order in which packages are to be delivered by each truck.

# B1. Logic Comments

In this section, I describe how I implemented the nearest neighbor algorithm into my program and use pseudocode to explain the developmental process.

As previously stated, the NN algorithm was used to find the shortest distance from the current delivery address to the next delivery address. I implemented the algorithm as a function that passes truck objects as its parameter. This truck object contains a list of packages it must drive around and deliver. For starters, I created an inner and outer loop that both serve the function of iterating through the list of packages within the truck object. Inside the inner loop, we check to see if the time has reached 10:20 a.m., so we can update package nine which has the wrong address written. Moving on, the hash table search function is used to retrieve package information and store it inside a variable. This variable retrieves the address of the current package and compares it to other potential addresses in order to attain the optimal distance. After, if the next package has a shorter distance than the previous package, we store it as the optimal package. The optimal package becomes the next package to be delivered and we use it for our next comparison.

Below is the pseudo-code that summarized the implementation of my algorithm.

**Algorithm Class simplification:**

for i in array []

optimal length

optimal object

for j in array[]

if time > 10:20

adjust object with wrong information

object = j.information

length = length between current object and next object

**NN Pseudocode**

if length < optimal length and object time == null:

optimal object = object

optimal length = length

object = optimal object

**Runtime:**

O(N²) – quadratic

This class contains two for loops, one nested inside of the other. The outer loop iteration runs through every single package within x truck, and then the inner loop repeats this process by running through each package ID within x truck. This loop iteration assists with the creation of a 2D array that allows for the comparison between the current address of drop-off and a potential future drop-off address. Given that our table contains a nested for loop, the space-time complexity of this function is quadratic, or O(N²). So, worst-case scenario, the function compares every single package inside truck x to every single package ID inside truck x.

# B2. Development Environment

The Python programming language was used to develop this project. My program was developed and run through PyCharm, an integrated development environment developed and maintained by JetBrains. The version of Python utilized was 3.11.0, while the version for PyCharm was 2023.1.3 (Community Edition). No outside libraries or packages were used to create this program. As for hardware, I used a MacBook Air with the M2 chip and 8 gigabytes of RAM.

# B3. Space-Time using Big-O

There are a total of four major segments in my source code: the hash table class, both the truck and package class, the algorithm function, and finally the piece of code that displays the packages at three distinct points in time. For the most part, all of these four prominent code segments use a space-time complexity of O(N). Therefore, the program as a whole holds a linear runtime.

A first major part of my code is the class holding my data structure. My hash table contains the functions of insert, search, and remove that alter the table in various ways: the insert function adds a new element, the search function looks for a specific element, and the remove function deletes a specific element. Not surprisingly, each of these functions has a linear space-time runtime of O(N). Meaning, that worst case scenario, the algorithm will have to search through every single element within the hash table at most once to find the value it needs.

A second main component of my program is the creation of both the truck and package objects using their corresponding classes. In this scenario, the truck class creates delivery truck objects while the package class creates package objects. To create either a package or truck object the space-time complexity is O(1) because every single package and truck object takes the same amount of time to create. For example, each time the package class is called to create a package object, the data types and amount of the indices in the package object remain the same. When creating a new package object, the parameters will always be id, street, city, state, zip, deadline, weight, and special notes. Each parameter passed will have the same data type. Such that, ID will always have an integer value, while street will always have a string value.

Third, another major segment of my code is the class that runs the nearest neighbor algorithm. This class contains two for loops, one nested inside of the other. The outer loop iteration runs through every single package within x truck, and then the inner loop repeats this process by running through each package ID within x truck. This loop iteration assists with the creation of a 2D array that allows for the comparison between the current address of drop-off and a potential future drop-off address. Given that our table contains a nested for loop, the space-time complexity of this function is quadratic, or O(N²). So, worst-case scenario, the function compares every single package inside truck x to every single package ID inside truck x.

Finally, the last major segment of the code involves displaying all packages at three different particular points in time. Each of these segments of code contains a space-time complexity of O(N) due to the fact that all packages are iterated through at most once.

# B4. Scalability and Adaptability

As the number of packages to be delivered grows, my algorithm’s time and space complexity increase at an exponential rate since my algorithm is located inside two nested loops. Both, the inner and outer loops, iterate through all packages within a truck. This is done so we are able to retrieve the distance between the current delivery address and a future package destination. Therefore, the more packages we have to deliver, the more items the inner and outer loops have to iterate through. Worst case scenario, the space-time complexity of the algorithm will be O(N²). For example, assuming it takes 1 millisecond to iterate through a single item, it will take 100 milliseconds to iterate through 10 items: (10)² = x milliseconds. In short, my hash table is capable of growing and adapting to a growing number of packages, but it will become exponentially slower the more packages it has to sort through. Though, it is currently efficient given there are only 40 packages it has to sort through.

# B5. Software Efficiency and Maintainability

The software is easy to maintain since it requires nearly no backend maintenance or intervention from a software engineer to upkeep the source code. The only prominent requirement the user must abide by is to add new packages formatted with the elements in the following order: id, street, city, state, zip, deadline, weight, and special notes. Failing to add packages as listed would result in an error. Other than that, the program is straightforward and can be used for an extensive period of time without breaking down or producing errors. Its low maintainability means the software is an efficient, cost-effective choice that avoids incurring significant expenses and using extensive resources. For most of the software life cycle, the program can remain untouched making it a robust, efficient piece of code.

# B6. Self-Adjusting Data Structures

Strengths

The hash table implemented has the following strengths:

1. Fast: Hash tables offer a constant time for searching, removing, and inserting items. On average, they tend to be quicker at searching through the items than other data structures such as an array. In the best-case scenario, the hash table will have a lookup time of O(1).
2. Versatile: Hash tables are suitable for a wide range of applications where key-value lookups are essential.
3. Space efficient: Hash tables are space efficient since they place values into buckets rather than a fixed-sized memory block.

Weaknesses

The hash table holds the following weaknesses:

1. Unordered structure: Hash tables do not maintain the order of insertion, which may be a drawback if the programmer wants to preserve the order in which items were inserted into the table.
2. Hash collisions: A collision occurs when two distinct keys map to the same hash value. Resolving for collisions adds complexity to the program and increases the space-time complexity.
3. Overhead costs: For large unstructured data sets, building a hash table becomes a very complex project and requires a lot of effort and time spent on its development. Developing an efficient hash table becomes a challenge and a poor choice of hash function may lead to performance issues.

# C. Original Code

\* See attached Python file \*

# C1. Identification Information

\* See attached Python file \*

Within the first line of my code, my name and student idea are commented:

Samuel Diaz #001361588

# C2. Process and Flow Comments

\* See attached Python file \*

My code includes necessary comments all throughout that explain the process and flow.

# D. Identification of Data Structure

The data structure I used to store all packages was a hash table with chaining.

# D1. Explanation of Data Structure

My data structure inserts, removes, and searches packages with an O(N) linear runtime. A hash table was deemed an appropriate way of storing the packages given that each individual package is its own entity, separate from one another. Each time a new package is added to the table, they receive a unique ID. In order to avoid collisions, chaining was implemented within the table. My hash table contains three main functions that allow it to be modified. The insert function adds a new package object, the search function looks up a specified package object, and the remove function deletes a specified package object. When adding a new package, the data structure assigns the package to a specific bucket and uses its unique ID as an identifier. This allows the table to account for proper storage of packages without any collisions.

# E. Hash Table

\* See attached Python file \*

My code contains a hash table data structure with an ‘insert’ function that allows package objects to be added. My hash table handles package objects which include the following components: ID, street, deadline, city, zip code, weight, delivery status, and delivery time.

# F. Look-Up Function

\* See attached Python file \*

My code contains a hash table data structure with a ‘search’ function that allows for the retrieval of package objects. My hash table handles package objects which include the following components: ID, street, deadline, city, zip code, weight, delivery status, and delivery time.

# G. Interface

\* See attached Python file \*

The code provides an interface for the user to view any package at any time, and the total mileage traveled by all trucks.

# G1. First Status Check (8:35 am – 9:25 am)

Screenshot that shows the status of all packages at 09:20.

A screen shot of a computer

Description automatically generated A black screen with white text

Description automatically generated

# G2. Second Status Check (9:35 am – 10:25 am)

Screenshot that shows the status of all packages at 10:20.

A screen shot of a computer

Description automatically generated A screen shot of a black box

Description automatically generated

# G3. Third Status Check (12:03 pm – 1:12 pm)

Screenshot that shows the status of all packages at 1:00.

A screen shot of a computer

Description automatically generated A screen shot of a black screen

Description automatically generated

# H. Screenshots of Code Execution

**Screenshot(s) showing successful code completion, free from runtime or warning errors, including the total mileage traveled by all trucks.A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

# I1. Strengths of Nearest Neighbor Algorithm

For one, the best advantage this algorithm has over others is that it is simple yet very effective. Its simplicity enhances the program’s readability which in turn makes the program easier to understand by third parties not directly involved in the creation of the code. For example, if a programmer were to look through this algorithm’s source code, they wouldn’t have trouble understanding what the algorithm is attempting to do as its design is very minimalistic and intuitive. Furthermore, this is a very popular algorithm used in several different areas of computer science research and technological development. So, it is safe to claim that most computer scientists have stumbled upon it given its popularity. So, both the algorithm’s simplicity and popularity make it a strong choice that enhances the readability and comprehension of the source code.

Second, another advantage of this algorithm is that it is non-parametric, meaning that it needs not to be too involved with the dataset to make assumptions. In this case, the algorithm is a perfect choice given that we are attempting to implement it in a package delivery system where there could be hundreds if not thousands of packages with a ton of information (in this case we only have 40). Also, as new data is added to the data set, the nearest neighbor adapts accordingly. There is no need to be too involved with the data, it’s just a matter of making minor adjustments before adding more packages to deliver. So, the NN algorithm being non-parametric and adaptable makes it a great choice.

# I2. Verification of Algorithm

I have verified that the algorithm meets all the requirements provided by the scenario such that the total mileage added to all trucks is 118 (less than the maximum limit of 140), and all packages were delivered on time.

General requirements met.

✓ Each truck can carry a maximum of 16 packages, and the ID number of each package is unique.

✓ The trucks travel at an average speed of 18 miles per hour and have an infinite amount of gas with no need to stop.

✓ There are no collisions.

✓ Three trucks and two drivers are available for deliveries. Each driver stays with the same truck as long as that truck is in service.

✓ Drivers leave the hub no earlier than 8:00 a.m., with the truck loaded, and can return to the hub for packages if needed.

✓ The delivery and loading times are instantaneous, i.e., no time passes while at a delivery or when moving packages to a truck at the hub (that time is factored into the calculation of the average speed of the trucks).

✓ There is up to one special note associated with a package.

✓ The delivery address for package #9, Third District Juvenile Court, is wrong and will be corrected at 10:20 a.m. WGUPS is aware that the address is incorrect and will be updated at 10:20 a.m. However, WGUPS does not know the correct address (410 S State St., Salt Lake City, UT 84111) until 10:20 a.m.

✓ The distances provided in the WGUPS Distance Table are equal regardless of the direction traveled.

✓ The day ends when all 40 packages have been delivered.

Package requirements met.

✓ Package 3 - can only be in truck 2  
✓ Package 6 - will not arrive at the depot until 9:05 am  
✓ Package 9 - wrong address listed  
✓ Package 14 - must be delivered with 15, 19  
✓ Package 16 - must be delivered with 13, 19  
✓ Package 18 - can only be on truck 2  
✓ Package 20 - must be delivered with 13, 15  
✓ Package 25 - will not arrive to the depot until 9:05 am  
✓ Package 28 - will not arrive to the depot until 9:05 am  
✓ Package 32 - will not arrive to the depot until 9:05 am  
✓ Package 36 - can only be on truck 2  
✓ Package 38 - can only be on truck 2

# I3. Other possible Algorithms

The following are two algorithms distinct from the nearest neighbor algorithm that can be used to establish a solution for the given problem:

1. Greedy Algorithm
2. Dijkstra Algorithm

# I3A. Algorithm Differences

Greedy algorithm vs. nearest neighbor algorithm

The greedy algorithm differs from the nearest neighbor algorithm which was implemented in the solution. The greedy algorithm is more niche in terms of which problems it can be applied to, while the nearest neighbor algorithm can be applied to almost any problem. Though, if applied correctly, the greedy algorithm can be much more efficient and produce a faster runtime. For example, take into consideration a cash register that distributes change: quarters, dimes, nickels, and pennies. An effective greedy algorithm would first look at the biggest type of coin to return and try to fit as many as possible into the change that needs to be handed. After it can’t fit any more quarters it would move to the next biggest coin, dimes. Then, it would go through nickels, and finally pennies. In comparison, the nearest neighbor algorithm will look at all the coins at once after every single run, until it can no longer return change. So, the greedy algorithm can only be applied to specific scenarios, but its smaller search space produces a faster runtime than the nearest neighbor algorithm if applied correctly.

Also, the greedy algorithm tends to be much more complex than the NN algorithm. In this situation, if we were to have used the greedy algorithm to find a solution for the package delivery system there would have been much more time and effort invested in its creation. I would’ve had to be more involved, perhaps establishing a hierarchy representing the best package option to choose from that wouldn’t affect the following packages.

Dijkstra algorithm vs. nearest neighbor algorithm

The Dijkstra algorithm is used to find the shortest path between the current point and the surrounding points. In this case, after delivering a package, the algorithm would search and move to the next address with the shortest distance. In comparison, the NN algorithm branches to the closest unvisited point, without considering the overall global optimality of the chosen path. The Dijkstra algorithm would find the most optimal solution, but it would require a lot more time and effort to develop as its implementation is much more complex. Hence, developing a Dijkstra algorithm would require more overhead but will provide a more optimal solution. Whereas the nearest neighbor would provide an approximate solution that may not be as optimal but doesn’t require an extensive amount of overhead and time spent on it.

# J. Different Approach

If I were to do this program one more time, one thing I would change would be the order of how I went about constructing my project from start to finish. First, I constructed the algorithm as I deemed it to be the most important objective. Second, I knocked out the interface given that it was pretty easy to create. Last, I constructed the HashTable, Truck, and Package classes. The process in which I developed my program made the subject matter more confusing which caused me to spend more time on the project than needed. This backfired because by the end when I tried to put everything together, it created mayhem and I had to go back and reformat all of my functions and classes. If I were to start again, I would first create the HashTable, Truck, and Package class. This would give me a strong foundation to follow. Then, I would append the data to the Hash table and create the interface. Lastly, I would develop the algorithm and piece everything together. This way, I have a clear picture of what my program is set to do, and I don’t have to spend too much time reformatting to fit the picture.

# K1. Verification of Data Structure

The provided data structure, a hash table, meet all the requirements requested by the scenario. The total mileage added to all trucks is 118 (less than the maximum limit of 140), and all packages were delivered on time. Each package gets a unique ID, there are no collisions, there is up to one special note per package, it has an insertion function and a lookup function, etc. The reporting of packages is accurate and efficient since it keeps a record of a specific package given a unique ID.

# K1A. Efficiency

My hash table is able to adapt to a growing number of packages, though, the more packages there are, the slower it would be for my lookup function to search for a specific package. My search function has a time complexity is O(N), meaning that worst case scenario, the time it takes to search for an item is proportional to the number of items in that set. If all items were to be clumped into the same bucket, and the item we were searching for was at the very end of the bucket, we would have to search through every single element to find the one we want. For example, let’s pretend it takes 1 millisecond to look up a single item and that our hash table contains five buckets with five distinct items. If all items were clumped into the first bucket, and the item we were searching for was at the end of the list, we would have to search through every single item to locate the one we wanted. So, it would take a total of 5 milliseconds to retrieve the value. On the contrary, if all five items where distributed equally into each of the five buckets, it would only take the searching of a single item to retrieve the value. So, it would take 1 millisecond to retrieve the value. Hence, best case scenario for our hash table is O(1). Given this information we can infer the time needed to complete the lookup function increases the more packages there are to be delivered.

# K1B. Overhead

My hash table has a space complexity of O(N) which means the space usage by the table increases in proportion to the number of items. As the number of packages to be delivered increases the space usage of the data structure also increases. If a single item takes up 1 MB of space, 10 packages would take up 10 MB, 100 packages would take 100 MB, and so on. On the contrary, packages removed from the hash table free up an amount of space proportional to the number of items removed.

# K1C. Implications

Increasing either the number of trucks or the number of cities would not have an effect on the lookup time or space usage of the data structure. All of the packages from every truck get placed inside the hash table. If we were to add more trucks to our system but keep the number of packages the same, there wouldn’t be a change in our hash table. Our data structure makes a decision to store a package based on its unique identifier, unrelated to the truck or city the package is tied to. Therefore, increasing the number of trucks or cities there doesn’t affect the lookup time and space usage of the table.

# K2. Other Data Structures

The following two other data structures could have been implemented to meet the requirements:

1. Array
2. Linked List

# K2a. Data Structure Differences

Array vs. hash table

An array differs from a hash table in that it does not have buckets that store groups of data. Instead, an array stores data in a linear fashion. Each time a new object is added to an array, it gets appended to the end and given a new index value. Usually, this index value is one greater than the previous. For example, given a set of numbers from 1 to 10, if we were to store these in an array we would end up with 10 different values stored in indices from 0-9.

Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Index: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Now let’s say we would like to store the same set of numbers 1 to 10 in a hash table with 5 buckets. Having 5 buckets implies that our hash value is 5. So, *(value of 1) % (5 buckets) = (1st bucket)* which means the value 1 would be stored in the first bucket. We do this for all values until every single one is placed inside a bucket. Each bucket represents an identifier, and each element inside a bucket contains an index that identifies each individual element. Usually, for large datasets hash tables tend to have a faster runtime than arrays.

Hash table: [0] [5, 10]

[1] [1, 6]

[2] [2, 7]

[3] [3, 8]

[4] [4, 9]

In brief, an array has a linear approach of storing data by appending new data to the end of the list. While a hash table stores data into groups called buckets and uses the modulus function to determine where each data value should go.

Linked list vs. hash table

A linked list is a type of data structure with a header, nodes, and a tail pointing at a null value. This, of course, differs from the hash table since as explained, it assigns the given elements to buckets. It is possible to use this data structure to store the parcel packages, but it wouldn’t be as efficient as using a hash table. Once the last parcel value gets delivered, the last value points to null which lets us know that there aren’t more values in our data set. The structure of a linked list can be summarized as follows.

Linked list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Index: [header (first value) 🡪 node (each value in between) 🡪 tail (last value) 🡪 null]

# M. Professional Communication

I have demonstrated professional communication in the content and presentation of my report.

# L. Sources - Works Cited

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

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