C950 WGUPS Algorithm Overview

Miguel-Angel Garcia

ID #001347769

WGU Email: mgar574@my.wgu.edu

05/19/2023

C950 Data Structures and Algorithms II

# Introduction

For this course, C950 Data Structures & Algorithms II, students are to solve a problem for a delivery business by creating a program. 40 packages must be delivered while maintaining a total mileage driven under 140 miles as well as satisfying other various constraints (e.g., time deadlines, incomplete address information, special notes, etc.). Students are to use the Python programming language to create a self-adjusting heuristic algorithm to solve this problem. In addition, the program must be able to allow the user to check the status of any package at any given time. Finally, the program must make use of a hash table to store package information.

# A. Algorithm Identification

The self-adjusting algorithm used in this program is the “Nearest Neighbour” algorithm. The algorithm was able to follow all constraints, deliver all packages, and maintain a total mileage between all three delivery trucks under 140 miles. The final mileage was 124.2 miles.

The self-adjusting part of the algorithm was the piece that decided which package to deliver next. The algorithm checks all distances for its given packages and delivers the package with the nearest address. After delivery of said package, the algorithm removes it from the inventory, again checks addresses of all remaining packages, and again delivers to the nearest address.

# B1. Logic Comments

currentLocation = startingLocation

nonDeliveredPackages = []

delvieryOrderOfPackages = []

mileage = 0

function distanceToNextDelivery(nextLocationDistance, presentLocation)

nextLocationDistance = nextLocationDistance - presentLocation

return nextLocationDistance

for each Package:

nonDeliveredPackages.append(package)

#initalize nextDeliveredPackage

nextDeliveredPackage = none

while nonDeliveredPackages.length > 0:

#sets the first item in a list to the shortest next address by default

smallestDeliveryDistance = distanceBetween(nonDeliveredPackages[0].deliveryAddress, currentLocation)

for presentPackage in nonDeliveredPackages:

#sets the first item in the remaining packages list to the next package by default

nextDeliveredPackage = nonDeliveredPackages[0]

#checks distances between current location and all remaining packages.

distanceToNextDelivery = distanceBetween(presentPackage.deliveryAddress, currentLocation)

#if a new closest delivery distance is found, it becomes the next package to be delivered

if distanceToNextDelivery <= smallestDeliveryDistance:

smallestDeliveryDistance = distanceToNextDelivery

nextDeliveredPackage = presentPackage

#after next package to deliver is found, it gets added in order to be delivered

delvieryOrderOfPackages.append(presentPackage)

#after finding next delivery package, it is removed from nonDelivered list

nonDeliveredPackages.remove(nextDeliveredPackage)

#add distance to mileage total

mileage += smallestDeliveryDistance

#setting current location to recently delivered package

currentLocation = presentPackage.deliveryAddress

# B2. Development Environment

For my development environment, I used the PyCharm Community Edition on My Macintosh computer, running MacOS. The application was written using Python 3.9. Running the “main.py” file started the program, which communicates with the user via the Command Line. All files, including CSV and Modules, are in the same directory and are pictured below.

# B3. Space-Time and Big-O

One of the first steps of the program is to extract data from the CSV files. Grabbing each row of information and placing it in the created array would give a worst-case runtime of O(N). This was done for the Distance and Address files. This is done again in the loadPackageInfo() function. Each package’s information is grabbed and then that information is used to create a package object. This again is O(N). Finally, the is the getAddress() function. This function takes a given address, and loops through each index of the address list until it finds the appropriate address. The worst-case time complexity is O(N). Space-time complexity is also O(N). As more items get added, more space will be required to store them.

Another important step is looking up address information. This is done in the findDistanceBetween() function. This could have a time complexity of O(1), as the function is given an x and y coordinate to look up. It goes to the y row and grabs the x item. If it is not found, the function tries the reverse, grabbing the x row and then obtaining the y item within. Both time and space complexity amortize to O(1).

The last major part of the program is the delivery algorithm itself. The algorithm takes the package IDs in the truck and adds them to the list of non-delivered packages. Both time and space complexity of O(N).

The next segment of the code runs until the number of packages in the non-delivered list is exhausted. The algorithms will iterate through the array. This runtime complexity is O(N). For each package in the array, we will get its delivery address, and calculate how far away it is from our present location. After going through each package, and comparing distances, we will find the package with the closest delivery address and deliver it. From there, the algorithm will again check all remaining packages and compare distances until there are no packages to deliver. This runtime complexity is O(N). There is a loop inside of a loop.

The overall complexity of the algorithm is O(N^2)

This space complexity is O(N^2)

# B4. Scalability and Adaptability

An example of something in this program that gets affected by scalability is the Hash Table. The more items that get added, the more collisions or chaining occur. We only created a hash table with 16 buckets. This could easily be a problem if the number of packages increases by a large amount. The worst-case run-time would become O(N), which could present a problem.

Another example of something that could be influenced by scalability is our algorithm used. We chose the “Nearest Neighbour” algorithm to solve our problem. The worst-case run-time of the algorithm is O(N^2). Although it may not be the most efficient, we can at least say that our algorithm is efficient. As more packages get added, the time it takes to set the package delivery order will increase quadratically. This problem could be alleviated by adding delivery trucks and especially drivers.

# B5. Software Efficiency and Maintainability

Appropriate steps were taken to ensure the maintainability of the code. Variable names are intuitive, as well as function names. Comments are plentiful, explain the logic, and allow a reader to understand the flow of the application. The goal of the program was to create a modular application. For example, if a developer decides to change how the trucks get loaded, the program will still run appropriately. If a developer sees that something needs to be repaired, the comments will illustrate the logic of the original developer. This will help the developer see which parts are supposed to do what, making repairing easier, as it will eliminate the overhead of learning time.

The program is efficient. This can be stated because Big-O time complexity is O(N^2), or polynomial time. The program will scale appropriately when the number of packages to deliver in a day increases.

# B6. Self-Adjusting Data Structures

The self-adjusting data structure used was the Hash Table. Like anything, it has its strengths and weakness. One strength is its lookup time. When provided the key, a user can access the appropriate bucket in one step. Another strength is it collision handling. If items get placed in the same bucket, they will begin to chain. Both key-value pairs are stored in a list at the same bucket.

One weakness could be its size. Once it is created, the size of a hash table is fixed. This could slow down look-up time if the number of items added becomes large. Another weakness is how the hash table handles integers. If you pass an integer as a key, it may hash it properly. But upon look-up, the passed integer key may not get hashed. This was a problem experienced when coding the program. When passing the key “1,” we saw that the package was not found.

# C. Original Code

Here is an example of the code written. This screenshot shows the first few lines of code belonging to the program.

A screenshot of a computer program

Description automatically generated with medium confidence

# C1. Identification Information

A screenshot of a computer

Description automatically generated with medium confidence

# C2. Process and Flow Comments

The program code was written with comments to help explain the flow of what is going on. These comments help me to remember what the intent of the code is. These comments would also help other developers understand the “what” and “why” of what is happening.

# D. Data Structure

The self-adjusting data structure used in this program was the Hash Table. Only one instance was used. This instance was used for the package objects created in this program.

# D1. Explanation of Data Structure

The self-adjusting data structure used in this program was the Hash Table. The hash table takes a package ID, turns an int into a string, and used the hash() method to return a hash value. This hashed value has a modulo operation performed on it. The remainder, between 0-15, is the   
“bucket” that will contain the package information. This method is faster than a linear search because it will find the value in its bucket (if it exists) in 1 step. This method O(1) is faster than a linear search, which has a time complexity of O(N).

# E. Hash Table

The hash table created resides in the ‘HashTable.py’ file. The hash table has 3 methods: insert, find, and remove. The insert() method uses package IDs to store information as required by the assignment.

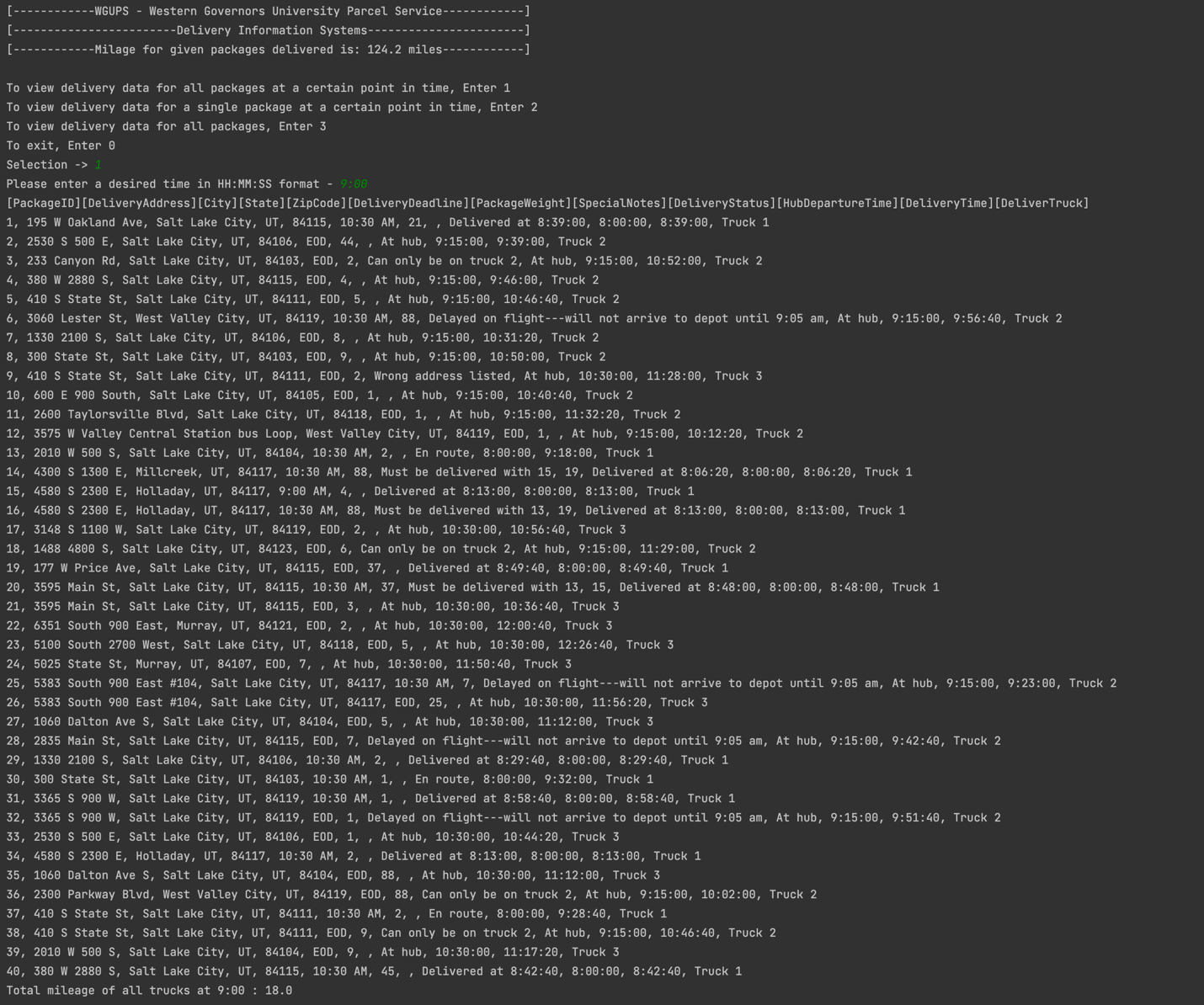
# F. Look-Up Function

The look-up function in this program’s Hash Table is called ‘find().’ Like above, this can be found in the ‘HashTable.py’ file. The function uses a package ID as a key to retrieve the values specified in part D (package ID number, delivery address, delivery deadline, delivery city, delivery zip code, package weight, and delivery status.)

# G. Interface

For my user interface, I made an interactive text user interface. The user will be presented with 3 options. The first is information for all package statuses at a desired time. The second option is the delivery status of a specific package at a desired time. The third and final option will present the user with all package delivery statuses at the end of the workday. The provided screenshots for package delivery statuses at three different times, 9:00am, 10:15am, and 12:30pm. The status of a package has three options: “at hub,” “en route,” and “delivered at [time].”

# G1. First Status Check

9:00am Status Check.

# G2. Second Status Check

10:15am Status CheckA screenshot of a computer screen

Description automatically generated with medium confidence

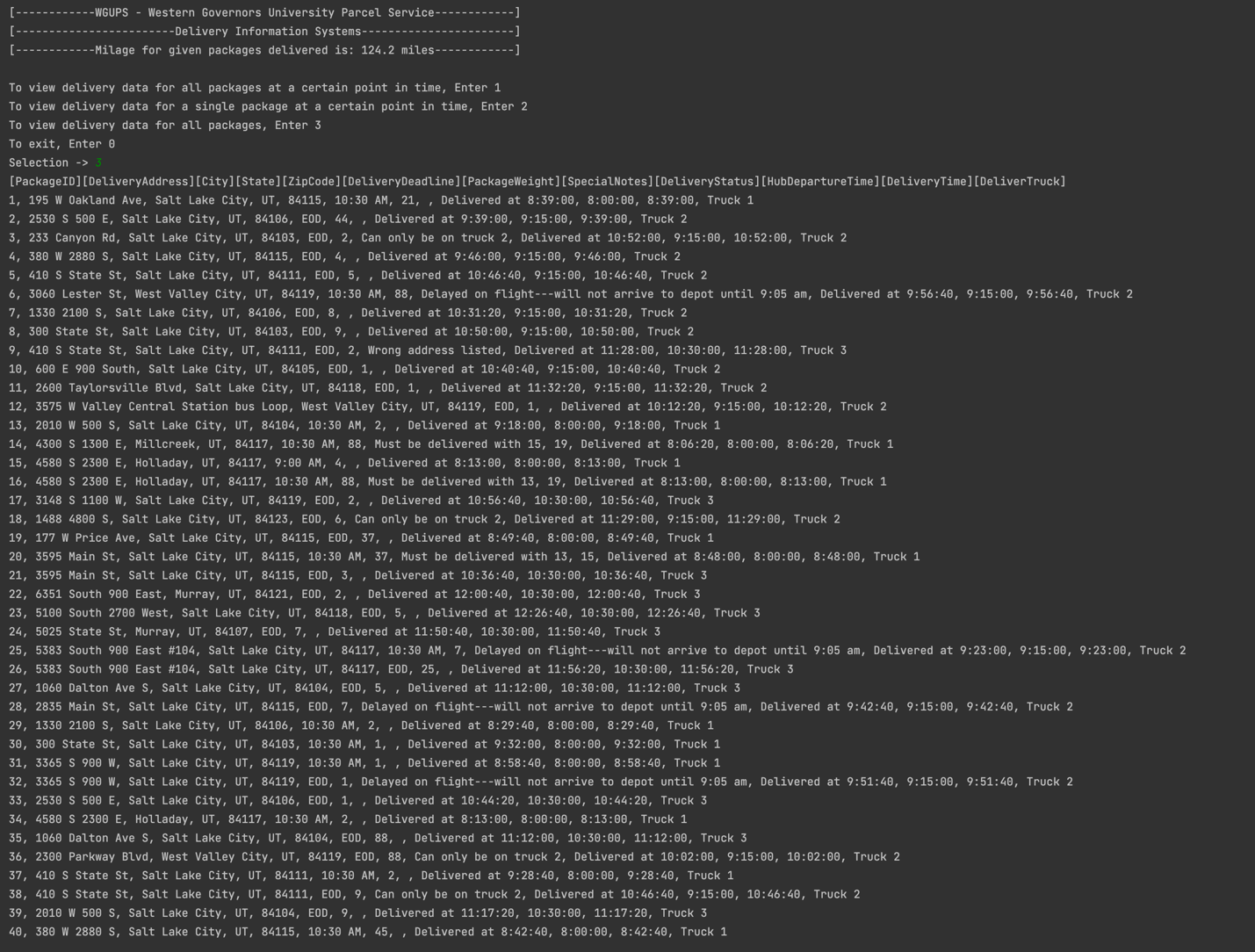
# G3. Third Status Check

12:30pm Status Check

# 

# H. Screenshots of Code Execution

End of Day Status Check



End of Day Status Check With File Folders

A screenshot of a computer program

Description automatically generated with medium confidence

# I1. Strengths of Chosen Algorithm

The algorithm used for this assignment was the Nearest-Neighbour Algorithm, which comes with many strengths. For example, it was one of the first algorithms to answer the traveling salesman problems. Another strength is its age. Because it is so old, it has been extensively studied, researched, and discussed. This makes answering any questions one has about implementing said algorithm, much easier to find. Finally, the algorithm is usually successful in solving the problem of finding the most efficient route. This algorithm was able to meet the project goal of delivering all packages in under 140 miles.

# I2. Verification of Algorithm

The goal of the project was to deliver all packages in under 140 total miles traveled by the three trucks. The solution provided delivered all packages with a total of 124.2 miles driven. Each package was delivered before its deadline, making it on-time, and meeting that requirement. Each package with delivery specifications was delivered with its requirements met. For example, certain packages had a time deadline. Other packages were arriving at 9:05am and could not be delivered until after they arrived. Some packages had to be on a certain truck, and others had to be delivered with certain other packages. Each specification was met. If a user would like to verify this information, they can select option 3 to get delivery data for all packages. The output with includes the package ID, its delivery deadline, any delivery specification if one exists, its delivery time, and delivery truck.

# I3. Other possible Algorithms

One possible alternative to the algorithm used in the assignment is the “Branch and Bound” algorithm. A second possible alternative algorithm could be the “Brute Force Algorithm.”

# I3. Algorithms Differences

The “Branch and Bound” algorithm is another solution to the traveling salesman problem, similar to the problem in this project. One attribute of the algorithm is that it finds the minimal path to reach the optimal solution. This means the algorithm assigns costs for each option, and will select the options with the lowest cost at each node. Another attribute is the algorithm's state space tree. The tree contains nodes, and each node is a partial solution. At each level of the tree, a decision must be made on which node will be included in the solution.

One attribute of the “Brute Force” algorithm is that it finds out every possible solution, also called exhaustion. It calculates the mileage of every possible route considering any delivery specifications. Brute force algorithm strengths include their simplicity and consistency. One weakness of the algorithm is its speed, which is slow.

# J. Different Approach

If I were to do this project again and do it differently, I would use Dijkstra’s Algorithm. Although they appear to be incredibly similar, there is one distinct difference, you can travel back to a location you have previously visited. With the Nearest Neighbour Algorithm, you choose the shortest path from a list of unvisited places. With Dijkstra’s algorithm, if the shortest path involves going back to a previously visited location, you are free to do so. This can lead to a solution that is more optimal.

# K1. Verification of Data Structure

The data structure used in this application, the Hash Table, was able to meet all requirements. All packages were delivered in 124.2 miles, meeting the goal of under 140 miles. Each package was delivered on time, delivered before its deadline, and met any delivery specifications if they were given.  
  
The data structure was an efficient hash table with a lookup function. Package status and information can easily be verified with the provided user-interface. Finally, the data structure stores information that is accurate.

# K1A. Efficiency

One thing that gets affected by an increasing number of packages is the look-up time. The hash table has 16 buckets, this means that some buckets may hold more than one package at a time. This is certainly the case when there are more than 16 packages. When there is more than one package in a bucket, collision handling occurs. This means the look-up function will have to iterate through each chained item until it finds what it is looking for, or in the event there is no match, it will need to iterate through each item before confirming.

# K1B. Overhead

The data structure space usage will increase linearly with each package added. If the number of packages goes from 40 to 80, the amount of space needed will double. With each extra package inserted, an extra space will be needed to store it.

# K1C. Implications

In this application, the hash table was used to store package information. In the event that trucks or cities are added, look-up time and space would not be affected. With extra trucks, the table will get accessed more, but neither space nor time will not be affected. With cities being added, the same is true. The address data inside the package information may change, but once again, space and look-up time will not be affected.

# K2. Other Data Structures

Besides the Hash Table, there are other data structures that exist. One example would be the array. To access an array, one only needs the index where it is stored. Once you access that index, you will retrieve your package information. Another data structure could be a queue.

# K2a. Data Structure Differences

Let us begin with the array. Both are similar in that they can have a look-up time of O(1). One difference is how they look up things. An array is given an index, an integer, and it goes to said index to retrieve information. This is much different that the Hash Table’s lookup, which has a hashing function. This means, hash table keys are not limited to only integers. For example, it can also use strings. Another difference is their worst-case performance. An array will always be O(1). Whereas a hash table can also be O(1), but its worst case is O(n), as a bucket can store more than one item.

Looking at a queue data structure, we see the differences are much starker. The queue is a first-in-first-out data structure, meaning there is an order in how things must be retrieved. This is much different from the hash table, where one can go into any bucket. Another difference is how things can get added to each data structure. For a queue, things can only get added to the rear, whereas for the hash table, you can add an item to any of its buckets.

# M. Professional Communication

I appreciate you taking the time to read my work. I look forward to any notes or feedback. I wish you a great day.

# L. Sources - Works Cited

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

Retrieved March 25, 2023, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>