C950 WGUPS Algorithm Overview

Meagan Wood

ID # 000933763

WGU Email: mwoo195@wgu.edu

C950 Data Structures and Algorithms II

# Introduction

This project was to create a routing/delivery system for Western Governors University’s parcel service (WGUPS). It has been created to meet all package, truck, and driver parameters that have been provided. As the user runs this program it will display the total mileage driven to complete all deliveries and the trucks with their assigned packages and departure times. The user will then be presented with 3 options to continue:

* Option one, the user will enter a package ID followed by a time. The user will then be provided with the specific package details and the status of the delivery at the provided time.
* Option two, the user will enter only a time. This will return the packages that have been delivered by that time, and the truck’s that may still be ‘en route’, or at ‘the hub’ with a list of the packages still on each truck.
* Option three has been provided for simple confirmation that all package parameters have been met. It will return all packages with their associated data: ID, address package was delivered to, deadline, weight(mass), special notes, and delivery time.

# A. Algorithm Identification

The algorithm used in the program is the Nearest Neighbor algorithm, which optimizes each truck route using the nearest address from the trucks current location.

# B1. Logic Comments

1. The nearest neighbor algorithm (get\_closest\_distance) uses functions that were created to help retrieve addresses associated with a packages (lookup\_address) and to help retrieve the distance between two addresses (get\_mileage) from csv files.
2. The get\_closest\_distance function takes a parameter of the truck class which keeps track of trucks packages, mileage, time, speed, current location. As the packages from the truck are accessed the package class updates to keep track of the package status, and delivery time.
3. Initialize min\_mileage to fifty
4. Initialize location to zero
5. For each package in truck packages get package address

a. If mileage of current location and address of package is less or equal to min\_mileage

1. Set min\_mileage to new mileage

2. Set location to address of package

3. End when all packages compared and min\_milage found

1. Set truck location equal to new location
2. Delivery\_duration equals min\_mileage multiplied by truck speed
3. Truck time equals truck time plus delivery\_duration
4. Truck total mileage adds min\_mileage and stores as truck total mileage
5. For each package in truck packages

a. If location equals package address

1. Set package departure time to truck departure

2. Set package delivery time to truck time

3. Set package status to delivered

4. Remove package from truck

5. End when all packages with matching address are removed

1. Do while there are packages on truck one
2. Do while there are packages on truck two
3. Do while there are packages on truck three

# B2. Development Environment

Operating system: MacOS Big Sur Version 11.6.

IDE: PyCharm 2022.2.3 (Community Edition)

Language: Python 3.10

Hardware: MacBook Pro (Retina, 13-inch, Early 2015), 2.7 GHz Dual-Core Intel Core i5

# B3. Space-Time and Big-O

Below are the space and time complexities for each major segment of the project. I have organized them by file to make it easy to reference, and included the entire program total.

Entire program:

|  |  |  |
| --- | --- | --- |
|  | **Space Complexity** | **Time Complexity** |
| **Total program** | O(n) | O(n^2) |

File: main.py

|  |  |  |
| --- | --- | --- |
| **Function** | **Space Complexity** | **Time Complexity** |
| def app | O(1) | O(n) |
| **Total** | O(1) | O(n) |

File: hashTable.py

|  |  |  |
| --- | --- | --- |
| **Function** | **Space Complexity** | **Time Complexity** |
| init | O(1) | O(n) |
| insert | O(n) | O(n) |
| search | O(1) | O(n) |
| remove | O(n) | O(n) |
| **Total** | O(n) | O(n) |

File: package.py

|  |  |  |
| --- | --- | --- |
| **Function** | **Space Complexity** | **Time Complexity** |
| def read\_package\_data | O(n) | O(n) |
| def get\_package\_data | O(n) | O(n) |
| def get\_truck1 | O(1) | O(1) |
| def get\_truck2 | O(1) | O(1) |
| def get\_truck3 | O(1) | O(1) |
| **Total** | O(n) | O(n) |

File: truck.py

|  |  |  |
| --- | --- | --- |
| **Function** | **Space Complexity** | **Time Complexity** |
| def lookup\_address | O(1) | O(n) |
| def get\_mileage | O(1) | O(1) |
| def delivered\_packages | O(n) | O(n) |
| def get\_closest\_distance | O(1) | O(n^2) |
| def total\_mileage | O(1) | O(1) |
| **Total** | O(n) | O(n^2) |

# B4. Scalability and Adaptability

The program has been designed with growth in mind. Meaning that if the quantity of packages changes, each function can accommodate that change. From the sorting algorithm to the hash table. All will adjust properly and each package will be delivered.

# B5. Software Efficiency and Maintainability

Maintainability:

The program has been designed to contain separate files for each major portion of the program. Meaning that if a programmer were to look at the program, it would be evident that there is a file to handle each major attribute. The files are:

* Main.py, where all the logic for the UI is.
* Hashtable.py, where all the packages are inserted and searched.
* Packages.py, which includes a package class to keep track of all the packages and each of their attributes.
* Truck.py, to maintain and route all the trucks, delivery all the packages and ensure packages are being updated as changes are occurring.
* PackageFile.csv, where all the package data is stored
* AddressFile.csv, where all the address is stored
* DistanceTable.csv, this file contains the distances between two addresses

All of the files and functions are easy to edit and maintain, with logic comments throughout to help understand the use of each item.

Efficiency:

The program has been designed with efficiency in mind. There are solutions that would provide a more ‘optimal’ route, but that would greatly affect the time complexity of the program, particularly if the amount of packages increase. Currently, as the number of packages increase it would incrementally effect the time-complexity, and the change would be unremarkable

# B6. Self-Adjusting Data Structures

This program utilizes a hash table, which is self-adjusting. A has table has many advantages and disadvantages, which you will find below.

*(ZYBooks, chapter 7)*

Advantages:

* Fast and efficient even with complex data
* Allows for quick operations (e.g. inserts, lookups, deletions)
* Stores data with keys for quick access
* Has ability to manage collisions
* Space-efficient

Disadvantages:

* Many collisions can affect the efficiency
* Cannot handle null values
* Can be difficult to implement
* Does not keep data in specific order

# C. Original Code

All the code in the program is original and written by me.

# C1. Identification Information

Have included my full name, email, and student ID on the top of the main.py file.

# C2. Process and Flow Comments

I have included many comments to describe the functions and what they are doing.

# D. Data Structure

The self-adjusting data structure is the hash table, if can be found in hash.py. This data structure works well with the nearest neighbor algorithm. Packages insert quickly and store the package information no matter how many packages there are. They are easily found with a key that maps to the memory location which allows for quick lookup when the algorithm is running.

# D1. Explanation of Data Structure

When packages are inserted into the hash table, they are mapped by a unique key to a specific location in memory called a bucket. That key is then used for quick lookup, updates, and deletions. *(ZYBooks chapter 7)* I have created a package class object with all packages attributes stored as indexes. When the packages are inserted into the hash table, the package class helps to ensure that the package with all attributes is stored and returned correctly, free of errors.

# E. Hash Table

The hash table can be found in the hash.py file of the program. All required components of the hash table have been included and no additional libraries or classes were used. Components included are: Insert function, Search function, and Remove function.

# F. Look-Up Function

All data elements of the lookup function have been included. Look up function does return the package along with its’ ID, delivery address, city, zip, delivery deadline, package weight, and the status.

# G. Interface

When the program runs the user will see the three trucks with their assigned packages, and the total mileage it took for all packages to be delivered. Then the user will choose one of the options that are listed to return the data they wish to view.

The interface includes the following options:

* Option one, the user will enter a package ID followed by a time. The user will then be provided with the specific package details and the status of the delivery at the provided time.
* Option two, the user will enter only a time. This will return ALL packages and their status at that time.
* Option three has been provided for simple confirmation that all package parameters have been met. It will return all packages with their associated data: ID, address package was delivered to, deadline, weight(mass), special notes, and delivery time.

Below you will find the required screen shots. I have also included a screen shot of option 3 above which is provided first.

Text

Description automatically generated

Text

Description automatically generated

# G1. First Status Check

1. Provide screenshots to show the status of *all* packages at a time between 8:35 a.m. and 9:25 a.m. You can seen the time entered at the top of the program is 09:00

Text

Description automatically generatedText

Description automatically generated

# G2. Second Status Check

2. Provide screenshots to show the status of *all* packages at a time between 9:35 a.m. and 10:25 a.m. You can seen the time entered at the top of the program is 10:00

Text

Description automatically generatedText

Description automatically generated

# G3. Third Status Check

3. Provide screenshots to show the status of *all* packages at a time between 12:03 p.m. and 1:12 p.m. You can seen the time entered at the top of the program is 12:30.

Text

Description automatically generatedText

Description automatically generated

# H. Screenshots of Code Execution

Provide a screenshot or screenshots showing successful completion of the code, free from runtime errors or warnings, that includes the total mileage traveled by *all* trucks.

Text

Description automatically generated

# I1. Strengths of Chosen Algorithm

Nearest Neighbor Algorithm:

Advantages:

* Simple and accurate
* Easy to implement
* Adjusts easily to account for changes in data

# I2. Verification of Algorithm

The nearest neighbor algorithm meets all the requirements. All 40 packages were delivered on time and all the special notes for the packages were followed. The package with the wrong address updated at 10:20 and did not leave the hub until 11:00. Since there were only 3 drivers, the third truck did not leave the hub until one returned. As they packages were delivered the status was updated to “Delivered” and the time it was delivered. The total mileage to deliver all packages was 105.90.

# I3. Other possible Algorithms

Other possible algorithms for this project are:

* Dijkstra’s algorithm
* Prim’s algorithm

# I3A. Algorithm Differences

Dijkstra’s algorithm -vs- Nearest Neighbor algorithm:

The nearest neighbor algorithm stores a list and calculates the distance for each item in the list, until it finds the closest distance. Whereas the Dijkstra’s algorithm takes a weighted graph with nodes rather than a list. It using a starting node and then a target node to provide the shortest path. *(Wikipedia)*

Prim’s algorithm vs Nearest Neighbor algorithm

The nearest neighbor algorithm stores a list and calculates the distance for each item in the list, in a greedy manner until it finds the closest distance. Whereas Prim’s algorithm finds the minimum spanning tree. It takes a weighted graph, starting at a vertex and finds a subset of the edges adding the closest connection from to tree to the vertex including every vertex. *(Wikipedia-2)*

# J. Different Approach

If I were to take a different approach with the program, I would choose a different algorithm. The Nearest Neighbor algorithm works well with the provided data set, but as the data set increase, this algorithm would slow down. This algorithm also does not necessarily provided the most optimal route. Since Nearest Neighbor only collects the closest point, it can take more time to make it through the entire list.

If I were to use dynamic programming approach I would get a more optimal route then what Nearest Neighbor can find. This approach calculates all of the locations and finds the nearest one, without re-visiting any of the locations. Since we know the size of the data set in this project, this solution would work. However, this solution is the most time consuming, and if the data set were to grow, it would largely affect the time-complexity of the algorithm.

If I were to use Dijkstra’s algorithm, the ability to handle a larger data sets would improve, since it compares edge weights and will revisit nodes if the path’s cost is smaller. The change in the algorithm would help efficiency with larger data sets.

# K1. Verification of Data Structure

The hash table meets all the requirements of the program. The trucks traveled a total of 105.9 miles and all packages were delivered on time. The hash tables look up function does return the package and all the corresponding information which includes: ID, delivery address, city, zip code, package weights, and delivery status with delivery time.

# K1A. Efficiency

The hash table is very efficient when looking up a package. Each package, when inserted is stored using a unique key that maps it to its place in memory, ensuring quick and efficient return of the data. Because of this, the hash table is more efficient in handling growing data sets than other options. Since a hash table uses buckets to map data, the search is very quick. As the amount of packages increase, it will incrementally effect searching through the buckets, increasing time by one object iteration.

# K1B. Overhead

For a hash table, as the data set grows, so does the space used. The space complexity of a hash table is O(n) and the space required increases as n increases.

# K1C. Implications

If the number of cities or the number of trucks increased, the look up time and space usage would remain the same. The only thing that would increase is the space usage as the number of packages increases.

# K2. Other Data Structures

The other data structures that could be using in place of the hash table are:

* Binary Search Tree
* AVL Tree

# K2a. Data Structure Differences

Binary Search Tree -vs- Hash Table

A Binary Search tree inserts according to the node’s key, comparing it to the current key. A Hash Table inserts by providing a key and taking the modulo of the key to map it to its respective bucket.

AVL Tree -vs- Hash Table

An AVL tree insertion involves searching for the insert location, inserting the new node, updating balance factors, and rebalancing.

A Hash Table inserts by providing a key and taking the modulo of the key to map it to its respective bucket.

# L. Sources - Works Cited

*Wikipedia : Dijkstra’s algorithm*

*https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm*

*Wikipedia-2 : Prim’s algorithm*

*https://en.wikipedia.org/wiki/Prim's\_algorithm*

Resources provided from class: *ZYBooks*