C950 WGUPS Algorithm Overview

Syed Saad Khurshid

ID #010081191

WGU Email: skhurs2@wgu.edu

April 21st, 2023

C950 Data Structures and Algorithms II

# Introduction

The purpose of the project is to determine the best route and delivery distribution for the Western Governors University Parcel Services (WGUPS) using a high-level programming language which in this case is Python v3.11.3. There being 40 packages that are split between 3 trucks and 3 different drivers and determining an algorithm that will deliver the required packages within the actual constraints and remaining under the maximum alotted travel of under 140 miles.

# A. Algorithm Identification

The greedy algorithm is being utilized by doing the following:

* A list of packages on the truck is passed along with the associated truck which have been assigned as first truck, second truck and third truck and the package location which by default is 0 or at “At Hub”.
* The current location is then compared to the location of all packages in the truck to determine the closest location
* Once all the objects in the truck have been determined to have the lowest value and have been compared in the truck, the lowest value is then moved from the truck list and moves to the address location
* The value is appended to an optimized locations list than the algorithm updates the current location with the new address and calls the function again with the smaller truck list

# B1. Logic Comments

Considering the following parameters

1. Truck\_list (represents a nested list of packages on a given truck)
2. Assigned\_Truck\_Number (Represents the truck you are working with in this case 3)
3. Current\_location (Using a recursive variable that is used to show where the truck is)

Creating the base case to break the recursion in the algorithm

If the length of first\_truck is 0 then return an empty list

(Recursive step)

If the length of the truck\_list is not empty

***Set lowest\_value (Distance to the closest deliverable location from current location) to 50***

***Set new\_location to 0***

***(represents where the truck is moving next to deliver a package)***

***If the truck\_list is not 0:***

***For index in truck\_list:***

***Check if current\_distance of current location and the next index in the truck package list <= lowest\_value:***

***(Searches all reachable locations and updates the lowest\_value if a lower distance path exists)***

***If lowest\_value is found then update lowest\_value and current\_path***

***The second loop in the recursive section of the algorithm is initiated***

***For index in truck\_list:***

***Check if current\_distance of current\_location and the next index in the truck package list is equal to the lowest\_value (This step we search through the list of packages again to find the package address value that would be lowest value, thus allowing multiple package delivery to same address if they are being delivered under the same truck)***

***If truck\_number is equal to 1, 2 or 3 (Based on the truck object passed an optimized optimized list for that truck is created)***

***optimizedTruck.append(current package) (The selected package from truck\_list is passed to the optimezedTruck list)***

***optimizedTruck\_index.append(current package index) (Append the selected package address index from the truck\_list to the optimizedTruck index list)***

***pop\_value = truck\_list.index(current package)***

***truck\_list.pop(pop\_value) (Remove the selected package from truck\_list)***

***current\_location = new\_location ( Update the current location to show truck at a new location)***

***calculate shortest\_route(truck\_list, truck\_number, current\_location)***

The algorithm goes again in a loop through the shorter truck\_list with an updated location

# B2. Development Environment

The programming model for this application is currently hosted on a local machine. The application is utilizing Python 3.11.3 and is executed under Visual Studio Code. All the data is currently being stored locally in a CSV file format that is currently stored in a data folder where using a data\_reading function utilizing the csv library where the data is then red and thus data exchange is only local and thus no external threat as one would on a network connected environment.

# B3. Space-Time and Big-O

Text goes here

# B4. Scalability and Adaptability

The core functions of the application are designed to be able to scale and address changes in the number of packages. Currently the application depends entirely on CSV files that are stored and address changes and locations are determined by the CSV files stored and can be changed if the data is connected to a database instead of a determined fixed csv file. Another method is to have the location of csv files be saved automatically by having a database system automatically save the file at an addressed location at the host. Additional packages can be inserted and the program will determine where to place the packages. This approach allows a great deal of control when implementing numerous sub-applications as the design allows the input set to change freely. Current problem is that the data as mentioned above is saved manually to meet all the constraints and deadlines. An optimized solution would have a database utilization approach that would have the data automated would have the ability of the application to adapt to new changes and environments.

# B5. Software Efficiency and Maintainability

While this method may not be the best time complexity, it scales well with constraints for example the limited number of packages the truck can carry and in this case it’s 16 packages per truck. The software is very efficient and in comparison, has a time efficiency of O(N^2). Errors can be traced back to the instance of the function making it easier to debug.

# B6. Self-Adjusting Data Structures

The data structure that was implemented throughout the application is list of lists. This structure was chosen because of its flexibility and being easy to work with. One of the advantages of having a list was its ability to allow creation of hash tables with chaining resulting in very fast lookup, insertion, removal, and access to specific data elements. Most of the operations had a constant time complexity of O(1). Due to this, it has the ability to grow with the large data set

# C. Original Code

See attached files

# C1. Identification Information

See attached files

# C2. Process and Flow Comments

See attached files

# D. Data Structure

The self adjusting data structure that can be used with the algorithm have been discussed in part A and B1

# D1. Explanation of Data Structure

The hash table used for this project is influenced by the chaining hash table structure described in Zybooks (Lysecky, section 7). It implements an insertion function that uses a key value pair to store and retrieve pieces of information. The hash table has an initial size of 40 and the key represents each of the 40 buckets where the objects will be stored in those buckets. In this case the project implemented has lists of lists where the first List/bucket to be identified by the hash key itself.

The hash table accounts for relationship between the stored data points first by using the Hash Key that is the ID. This can then be sued to retrieve the entire package object being stored or one of its specific attributes such as its Zipcode or status

# E. Hash Table

class HashingTable:

def \_\_init\_\_(self, capacity=10):

*self*.table = []

for \_ in range(capacity):

*self*.table.append([])

# Create hash key -> O(1)

def create\_hash\_key(self, key):

return int(key) % len(*self*.table)

# Insert package into hash table -> O(n)

def insert(self, key, value):

hashKey = *self*.create\_hash\_key(key)

values = [key, value]

if *self*.table[hashKey] == None:

*self*.table[hashKey] = list([values])

return True

else:

for pair in *self*.table[hashKey]:

if pair[0] == key:

pair[1] = values

return True

*self*.table[hashKey].append(values)

return True

# Update package in hash table -> O(n)

def update(self, key, value):

hashKey = *self*.create\_hash\_key(key)

if *self*.table[hashKey] != None:

for pair in *self*.table[hashKey]:

if pair[0] == key:

pair[1] = value

return True

else:

print('Unsuccessful Key Updation: ' + key)

# Get a value from hash table -> O(n)

def get\_value(self, key):

hashKey = *self*.create\_hash\_key(key)

if *self*.table[hashKey] != None:

for pair in *self*.table[hashKey]:

if pair[0] == key:

return pair[1]

return None

# Delete a value from hash table -> O(n)

def remove(self, key):

hashKey = *self*.create\_hash\_key(key)

if *self*.table[hashKey] == None:

return False

for i in range(0, len(*self*.table[hashKey])):

if *self*.table[hashKey][i][0] == key:

*self*.table[hashKey].pop(i)

return True

return False

# F. Look-Up Function

# Get a value from hash table -> O(n)

def get\_value(self, key):

hashKey = *self*.create\_hash\_key(key)

if *self*.table[hashKey] != None:

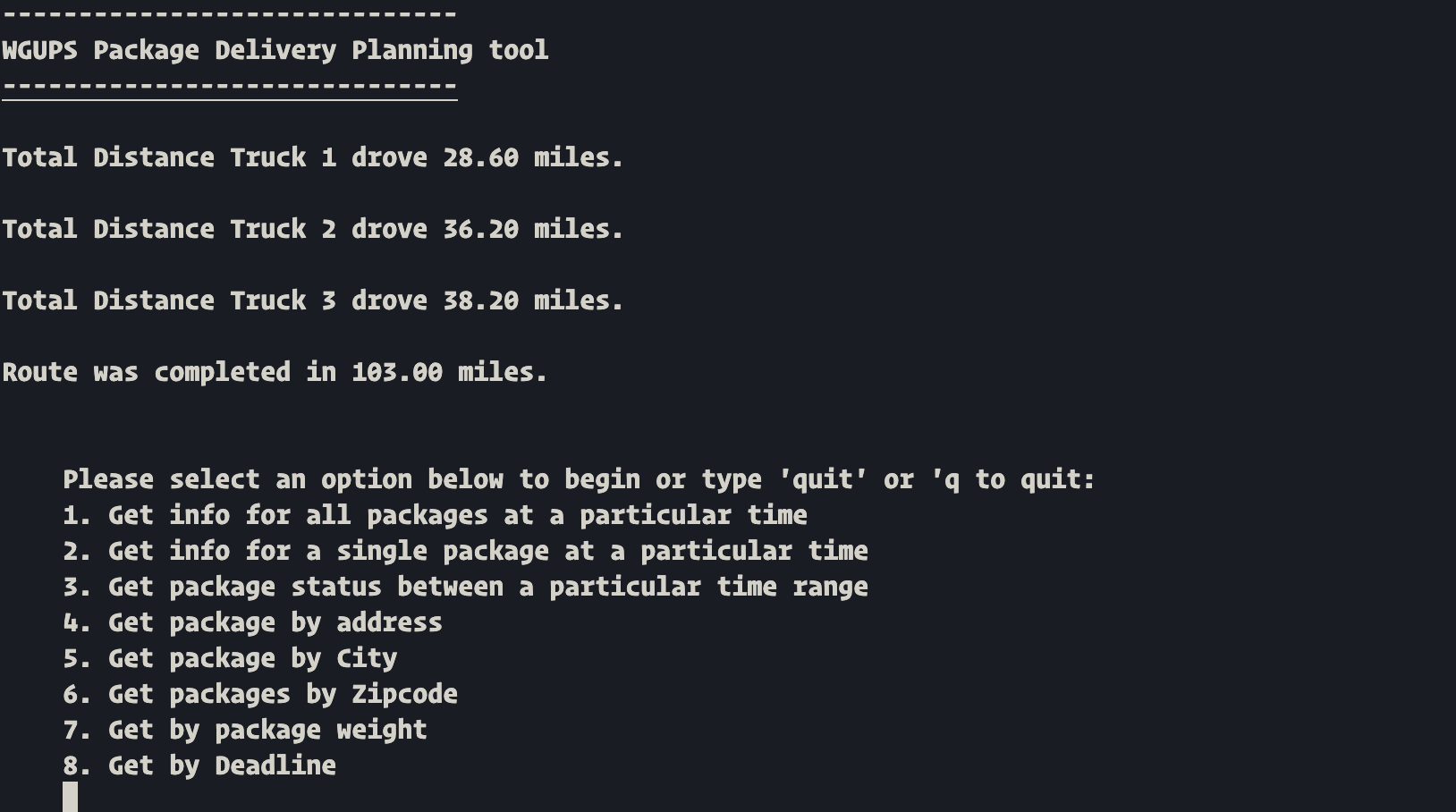
for pair in *self*.table[hashKey]:

if pair[0] == key:

return pair[1]

return None

# G. Interface



# G1. First Status Check

Searching all package status between 08:25am and 09:25am

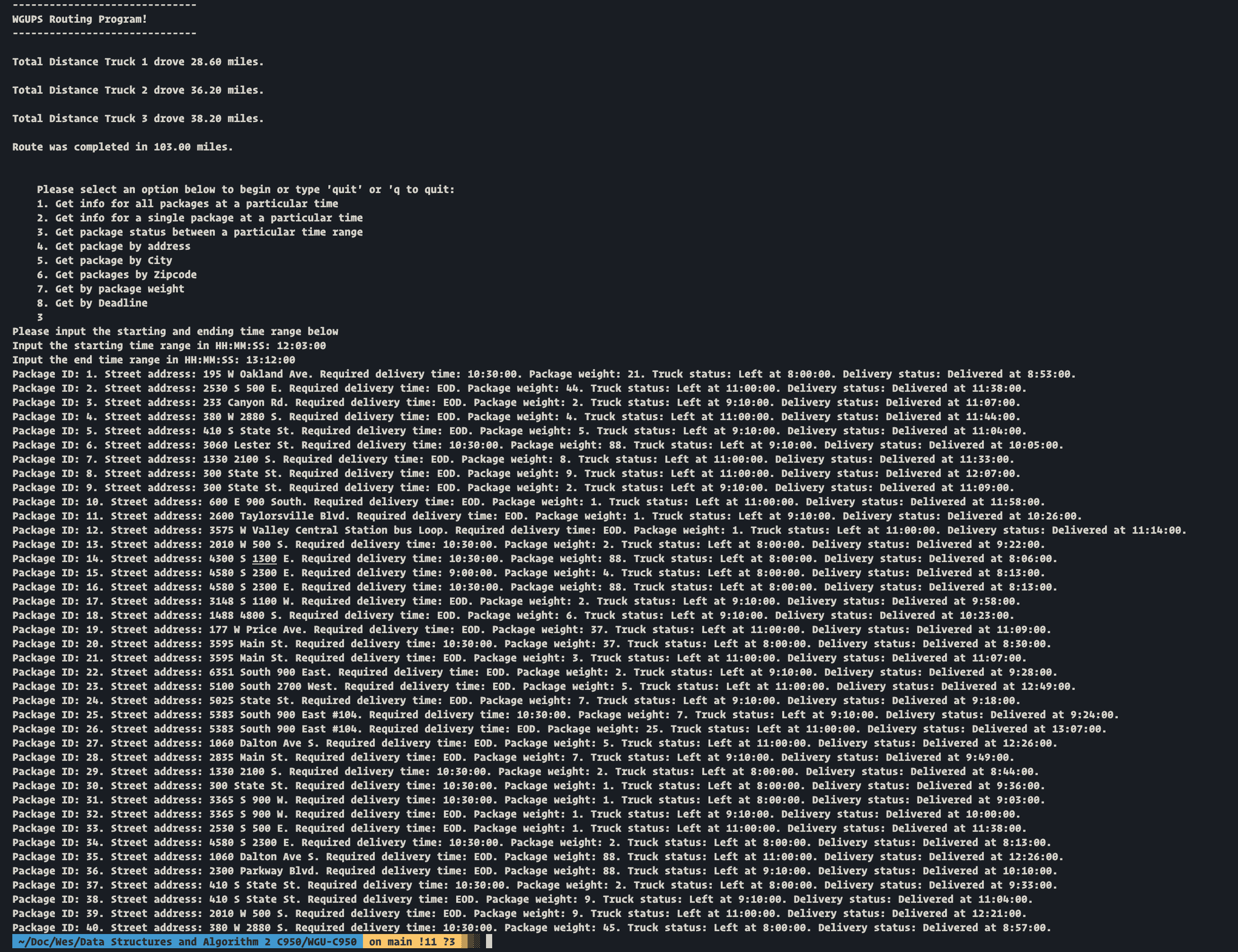


# G2. Second Status Check

Status of Packages between 09:35am and 10:25am

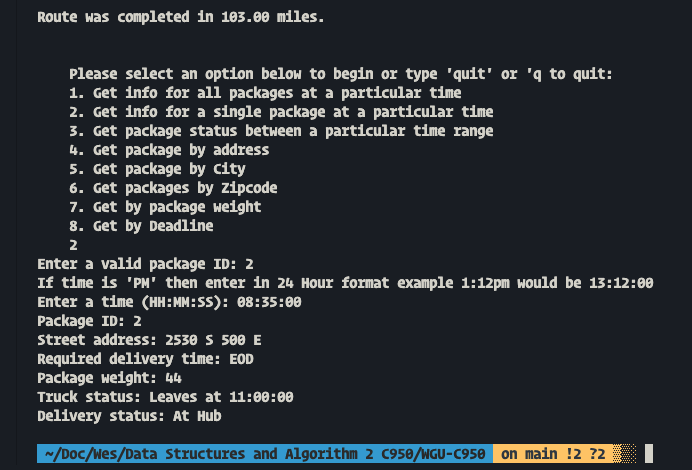


# G3. Third Status Check

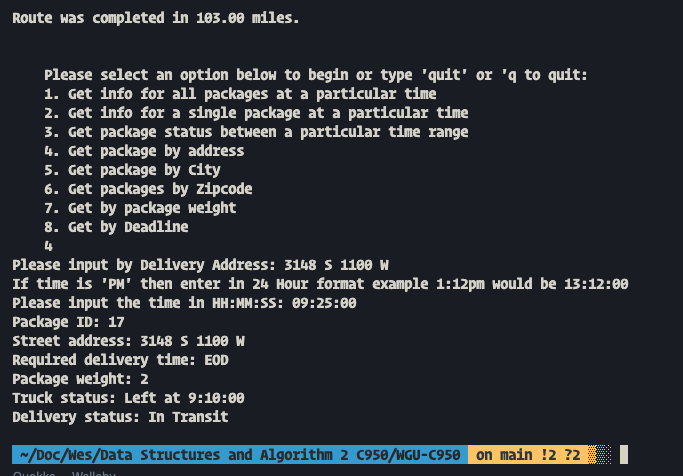


# H. Screenshots of Code Execution

Searching by package ID:



Search by Address



Search by City:



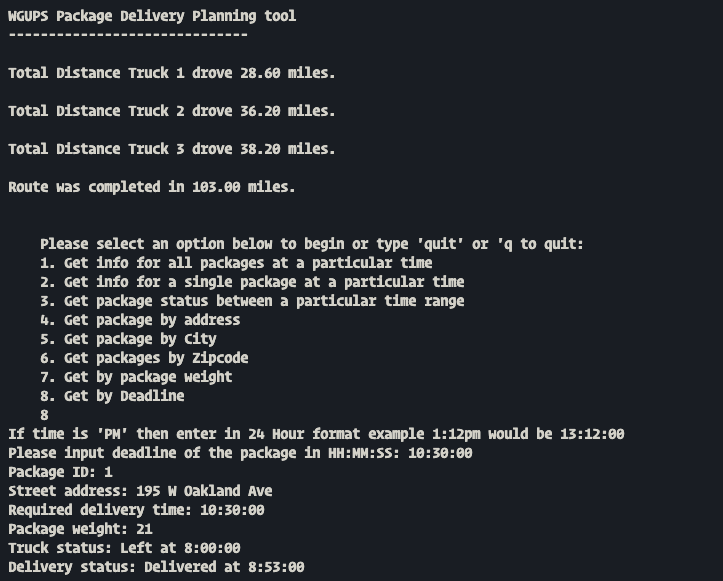
Search by Zipcode:



Search by weight:



Search by Deadline:

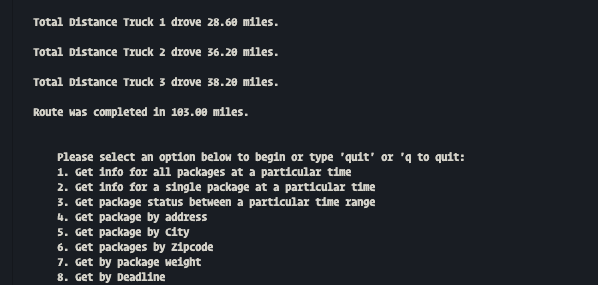


# I1. Strengths of Chosen Algorithm

The current method I chose the greedy algorithm is that it is due to the ability to scale with constrains. Having constraints like the limited number of packages that the truck can carry which in this case is 16 packages per truck. Increasing the size of the data would not deter the application and due to the algorithm being function based can result in errors being able to be traced back to the functions instance.

# I2. Verification of Algorithm

The algorithm used in the solutions meets all requirements because each of the packages are delivered before the deadline. Based on the constraints being applied here such as the limited number of packages each truck can be delivered and utilizing the lowest distance resulted in truck carrying the packages with the shortest route and resulting in total distance all 3 trucks traveling to be in 103 miles.



# I3. Other possible Algorithms

Looking back a different Algorithm could have been utilized for example utilizing the Binary search tree. The Binary Search Tree (BST) would have been different as packages could have been sorted based on its attributes and quickly access them through a tree.

Graphs is another possible data structure that could have been implemented because as one of the advantages of utilizing graphs would be that similar packages could be grouped together as adjacent to vertices. Then I could have utilized the graphs to travel or traversed the graphs until the maximum length of 16 (Current package constraint of 16 packages per truck) was reached and increasing the constraint limit could also help in future scaling.

# I3A. Algorithm Differences

As mentioned in section 13, the Graphs is a data structure for representing connections amount items, and consists of vertices connected by edges. Two vertices are adjacent if connected by an edge and path is the sequence of edges leading from a source vertex to a destination. In this case the path length would be the distance from one point to another. The advantage I feel if I were to utilize graph based data structure would have been that we can quickly group similar packages together as an adjacent vertices and we would have traversed the graphs until a maximum travel length is achieved. This would be good for future scaling.

# J. Different Approach

Two vertices are adjacent if connected by an edge and path is the sequence of edges leading from a source vertex to a destination. In this case the path length would be the distance from one point to another. The advantage I feel if I were to utilize graph based data structure would have been that we can quickly group similar packages together as an adjacent vertices and we would have traversed the graphs until a maximum travel length is achieved. This would be good for future scaling.

# K1. Verification of Data Structure

If I were to do this project again, I would perform the data structure in a graph based route instead of the greed methodology because of the factors taken into the fact that using graph based method, the packages can be grouped based on attribute for example weight class or delivery range of time as adjacent vertices and that based on the distance of the destination from the graph one could reach the maximum traversal length of 16 which is the total number of packages one can deliver. This would also be good for future scaling

# K1A. Efficiency

Overall, the software is every efficient, with two comparisons that have a time efficiency of O(N^2). Based on the constraints being applied here of 16 packages per truck, it may not be the best time complexity. It is maintainable as the core functions have been modified for use cases. Debugging is easier because you can always refer to another instance of the function to determine the potential errors.

# K1B. Overhead

The amount of space usage of a hashmap changes proportionally to the number of packages to be delivered since hash tables have a space complexity of O(n).

# K1C. Implications

Changes to the number of trucks would increase space usage proportionately to the number of trucks in WGUPS fleet as each truck has a list of its own packages to deliver. Adding more addresses and cities would increase space usage exponentially, since each destination references one another in the delivery algorithm.

# K2. Other Data Structures

Alternatives to the hash map that would work for this application would be binary search tree or some type of queue-based structure

# K2a. Data Structure Differences

Binary search tree completes all the same operations as the hash map in O(log n) time. Although all of these common operations are generally less efficient, binary search trees always perform them at this rate whereas hashing method because it every list was being utilized in a similar way, this can result in slow downs and inefficient utilization of memory as the number of packages and locations increases.

# M. Professional Communication

# L. Sources - Works Cited

The primary recourse I used when modeling this overview was reading material provided by Zybooks.

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

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