Joshua Corpuz

C950 – Data Structures and Algorithms II

Western Governors University

C950 Documentation

**A: ALGORITHM SELECTION**

**Identify the algorithm that will be used to create a program to deliver the packages and meets all requirements specified in the scenario.**

I used the Nearest Neighbor Algorithm to calculate the best route for the packages to be delivered. The nature of the algorithm looks for each of the current vertex’s “nearest neighbor” or nearest other node that has yet to be visited. Before the algorithm is optimized, each package is assigned to each of the trucks and loaded depending on each of the package’s notes and dependencies like delivery deadlines, trucks they need to be on and other packages they need to be delivered with. Once all the packages have been loaded, the algorithm will be called to create an optimized route for each of the truck.

The algorithm successfully delivered all the packages on time and below the project limit of 140 aggregate miles travelled between the three trucks provided. See the screenshots on section **I2** as evidence of the deliveries with the package details, and the summary of the results.

**B1: LOGIC COMMENTS**

**Write a core algorithm overview, using the sample given, in which you do the following:**

Function nearest\_neighbor(truck, graph\_data\_to\_reference)

Step 1: initialize the following variables:

Minimum\_distance = 999999.99 (a ridiculously high value)

Nearest\_neighbor = [] #a list to hold the values for the nearest neighbor

vertex\_a = ‘HUB’ initialize vertex\_a to be first vertex

address\_list\_temp = [] # temporary holder for the addresses, will be used for error handling

adjacency\_list = graph\_data\_to\_reference.get\_edge\_weights(vertex\_a) #initialize the first adjacency list to the values for vertex\_a

truck.route.clear() # always clears the truck’s current route, allows for the function to be called again without causing any errors

Step 2: Find the nearest neighbor for the current vertex\_a from the adjacency list initialized in the start of the function. Looping through all the value pairs of vertex\_b and edge\_weight and returning the one with the shortest distance to the current vertex. Once found, the algorithm will append the nearest\_neighbor to the truck’s route, remove the found address from the truck’s address list and append it to the address\_list\_temp. The nearest\_neighbor will then be assigned as vertex\_a, and minimum\_distance will be re-assigned to its initial value. The algorithm will continue the loop until the truck’s address\_list has a length of 0.

while (len(truck.address\_list) > 0):

for location in adjacency\_list:

if location[0] not in truck.address\_list:

pass

elif location[0] == vertex\_a:

# removes the found nearest\_neighbor from address\_list

truck.address\_list.remove(location[0])

# adds the found nearest\_neighbor to temp\_address\_list, to reset the truck's address\_list when the algorithm is over

address\_list\_temp.append(location[0])

pass

else:

distance = float(location[1])

if (distance < min\_distance) and distance != 0:

min\_distance = distance

nearest\_neighbor = location

# removes the found nearest\_neighbor from address\_list

truck.address\_list.remove(nearest\_neighbor[0])

# adds the found nearest\_neighbor to temp\_address\_list, to reset the truck's address\_list when the algorithm is over

address\_list\_temp.append(nearest\_neighbor[0])

# adds the nearest\_neighbor and it's distance from vertex\_a to the route

truck.route.append([nearest\_neighbor[1], nearest\_neighbor[0]])

# sets vertex\_a to the current nearest\_neighbor

vertex\_a = nearest\_neighbor[0]

min\_distance = 99999.99 # resets min\_distance to an absurdly high number

# resets the adjacency list using the new vertex\_a

adjacency\_list = distance\_graph.get\_edge\_weights(vertex\_a)

Step 3: Complete the route, assign the route and re-assign address\_list

Prepend(‘0’, ‘HUB’)

distance\_to\_hub = distance\_graph.get\_edge\_weight(vertex\_a, "HUB")

Truck.route.append(distance\_to\_hub)

Truck.address\_list = address\_list\_temp

**B2: DEVELOPMENT ENVIRONMENT**

The program utilizes Python 3.10 with Visual Studio Code as the IDE. The program runs locally with the data provided as Excel files converted into CSV files in the root directory of the program, this data is then read by the program using Python’s csv reader module. The program runs completely as a console application.

**B3: SPACE-TIME AND BIG-O**

The Nearest Neighbor Algorithm has the space-time complexity of O{n^2). The other functions utilized in the program have their time-complexities commented above the function declaration.

**B4: ADAPTABILITY**

The data structures used for storing and reading the data given will be able to scale with an increase in the number of packages and locations. Given that the self-adjusting nature of the data structures (hash\_map and graph), their designs allow for a change in scale without affecting their insertion and lookup functionalities.

However, one of the most glaring issues with the solution as it currently stands, is its reliance on excel files as its data source. Combine the need to manually convert the data set into a CSV file and then load them to the program repeatedly would be a complicated and tedious task. This would prevent it from scaling up as market and business needs change.

The other potential source of problem would be the handling of changes in the package data. In the real world, changes to package data could be constant and should be instantaneous. There should be a way to handle such changes when inputted into the system.

**B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY**

The design of the project leverages OOP as much as possible. Given that, the program would be able to take on more trucks and packages to an extent. Because each part of the business process has been abstracted out to their own individual classes, the core logic and process of the program will be able to handle an increase in trucks and packages.

However, some aspects of the program like loading the package data, distance matrix, and assigning which packages go to which trucks are all custom built with the project requirements in mind. Given that, those parts of the program need to be adjusted as the data source, its format, and dependencies change.

**B6: SELF-ADJUSTING DATA STRUCTURES**

There are two self-adjusting data structure in the project: graph.py and hash\_map.py

Strengths:

The data structure in hash\_map.py was used to store a master list of packages, this allowed for insertions, deletions, updates, and lookups to be quicker than a traditional list would provide. The data structure in the graph.py allowed the storage of the adjacency matrix provided from wgups\_distance\_table.csv as an adjacency list which also allowed for insertion, deletion, updates, and lookups to be quicker than a traditional list would provide.

Weakness:

Their space complexity of a hash table largely depends on how well the hashing key is designed to be calculated. Without increasing the number of “containers” or “buckets” of a hash table, the distribution of values stored at certain keys may skew depending on how well made the hashing logic is. Essentially, if a certain key contains a long list of values, the lookup functions will essentially be looping through another long list of values to get the desired value stored.

**C: ORIGINAL CODE**

All the project’s code is original, and results meet major project requirements: All packages were delivered on time and total miles travelled by all trucks was 125.8 miles.

**C1: IDENTIFICATION INFORMATION**

Each file was labeled with first name, last name, and student id

**C2: PROCESS AND FLOW COMMENTS**

Each code block has comments to explain the process and flow of the coding

**D1: EXPLANATION OF DATA STRUCTURE**

I used a graph data structure for my distance data and a hash\_table for my package data.

The graph is loaded with data using the add\_vertex() and add\_edge\_weights() functions. First the list of the vertices is created from the addresses/locations in the wgups\_distance\_table.csv and then each of those locations are used as keys. The corresponding vertex and distance are then assigned as the values for each of those keys.

The hash\_table is loaded with package data in the beginning of the program using the insert() function of the hash\_table. First, each data set is used to create a package object which is then inserted into the hash\_table using the package id as the key and the package object as the value.

**E: HASH TABLE**

**Develop a hash table, without using any additional libraries or classes with an insertion function that takes the following components as inputs and inserts the components into the hash table:**

* Package ID Number
* Delivery address
* Delivery city
* Delivery zip code
* Package weight
* Delivery status

First, with the use of object-oriented design, I made a representation of a package by defining a class. The components stated above are then assigned as the class’s properties. The package id, delivery address, city, zip and weight will be retrieved from the WGUPS Package File while delivery status will be assigned by the program. With this class, the program will have an abstracted version which could be used in a hash table data structure as the value component.

Second, I created a chaining hash table to store these package classes. The hash table was made to have a configurable size by passing an integer in its constructor. The number tells the program how many “buckets” or “containers” the hash table will have. Next, to ensure that each key will be limited to the number of buckets, the hash key will be calculated by dividing the integer value of the chosen key by finding the remainder (modulo) against the size of the hash table (key % hash table size).

The next step would be to prepare the data given by cleaning up the format of the WGUPS Package Excel file by removing unnecessary spacings, headings and/or images from the file. It is then saved as a CSV file. With the data inculcated into a manageable format, it can then be loaded into the program.

I decided to use the package Id number as the key and used the created package object as the value that will be stored. A simple function was declared to consume this data by looping through all the rows of the CSV, reading each of the item in that row and setting the package object properties to those read values. Then, with the use of the package id number, the package is inserted into the hash\_table.

Also, since the delivery status is not part of the data set, all the packages were initialized to a default value of “AT HUB”.

**F: LOOK-UP FUNCTION**

Inside the hash\_map class, the search function passes in the inputted package ID number of the package the user is looking for. That integer is then used as the program internally calls the function that generates the hash key. This hash key will be used to identity where in the hash table the search function should look. If found, the program will loop through the values stored in that key and find the package that has a matching id. Once found, the search function will return the package object including all its stored properties.

**G. Provide an interface for the insert and look-up functions to view the status of any package at any time. This function should return all information about each package including delivery status.**

* The Look-up functionality, or the hash\_map’s search() function is implemented in the beginning of the program. See screenshot: Text

  Description automatically generated
* Text

  Description automatically generated
* Text

  Description automatically generated
* The Insert functionality of the hash table was implemented in a loop as the program went through each row of the package data CSV file, created a package object, and inserted each of those objects into the hash table. See screenshot:Text

  Description automatically generated

1. Provide screenshots to show package status of all packages at a time between 8:35 A.M. and 9:25 A.M.

- An arbitrary time between 8:35 A.M. and 9:25 A.M was chosen, please note the different statuses of the packages. See screenshot: Text

Description automatically generated

2. Provide screenshots to show package status of all packages at a between 9:35 A.M. and 10:25 A.M

- An arbitrary time between 9:35 A.M. and 10:25 A.M was chosen, please note the different statuses of the packages. See screenshot: Text

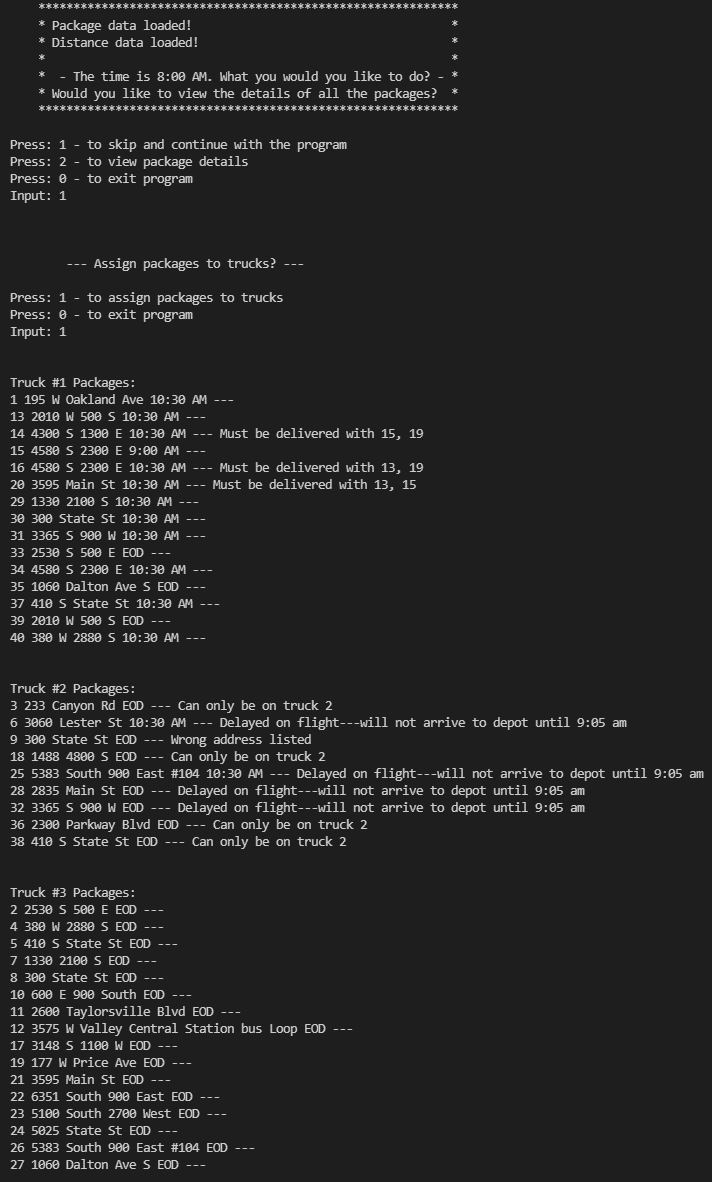
Description automatically generated

3. Provide screenshots to show package status of all packages at a time between 12:03 p.m. and 1:12 pm

- An arbitrary time between 12:03 P.M. and 1:12 P.M was chosen, please note the different statuses of the packages. See screenshot: Text

Description automatically generated

H. Run your code and provide screenshots to capture the complete execution of your code.

Program’s business logic starts.

Trucks 1 and 3 are delivered first as we only have two drivers. Truck 2’s delivery is interrupted as the updated package address for package 9 is received.

Text

Description automatically generated

At 10:20 AM, the corrected package address of package 9 was received, this prompts the user to update the package address, and to reoptimze the route for the truck that package 9 was on.Text, timeline

Description automatically generated

Text

Description automatically generated

A look up function for package status is also implemented to see package status throughout the day.

Text

Description automatically generated

Results:

Please note that truck 2 does not leave until truck 1 is back at the HUB.

Text

Description automatically generated

The program successfully meets project’s mileage requirements and truck restrictions.

**I1: STRENGTHS OF THE CHOSEN ALGORITHM**

Strength #1: The algorithm chosen is efficient. It has a space-time complexity of O(N^2).

Strength #2: As it says on the article by Marina Chatterjee, “A Quick Introduction to KNN Algorithm”, (KNN is another name for Nearest Neighbor Algorithm) a big advantage is that it is a simple algorithm. Making it easier for another to investigate the code and easily understand the process and if needed, use it for themselves. This ultimately leads to better maintainability.

**2. Verify that the algorithm you chose meets all the criteria and requirements given in the scenario**

The algorithm chosen met all the criteria and requirements given. As required, it is a self-adjusting or heuristic algorithm. The algorithm was also able to deliver all packages on time and found routes with an aggregate of 128.5 miles which is less than the limit of 140 miles.

Other project requirements are met in other parts of the program. Trucks were limited to 16 packages when assigned. The package notes, delivery times and dependencies were used to affect the loading process. The time calculation was based off the constant speed of the trucks that was at 0.3 miles per minute or 18 miles per hour.

**3. Identify two other algorithms that could be used and would have met the criteria and requirements given in the scenario**

Both a Brute-Force Algorithm and Dijkstra’s Algorithm could have been used to calculate optimal routes for the trucks.

However, a Brute-Force Algorithm has a time-complexity of O(m\*n). Although it would probably give us the best possible solution, it is very inefficient as it would look through every possible scenario to determine whether it is the best route to take. The computational overhead that it would take would be too much and would not be feasible for use in a real-world scenario. Therefore, the Nearest Neighbor Algorithm would be the better choice.

Dijkstra’s algorithm would also be able to find an optimal route that is within the project scenario. However, the biggest difference is the complexity of the algorithm itself and how it is written. First, it would need to initialize the vertices to infinity. Second, we would also have to keep track of all the other vertices that have already been visited and create a path from there. This process will then be repeated to all the other vertices to find the shortest distance. The implementation would be a lot more complex than the algorithm used in this project.

**J. Describe what you would do differently if you did this project again**

If I were to do this project again, I would inspect my data set more before designing the hashing function for it. I did not realize that the only unique identifier for each package was its Id and not the address to which each package was to be delivered to.

I had initially designed my hashing function to take in the sum of all char values from the address to find the modulus against the size of the hashing table. I would have saved time in the development stage had I taken note of the fact that it was more feasible to use the package id as the key.

Also, I would also design the database structure to store and process the data more efficiently. This would alleviate the issue mentioned in section B4 as the task of data processing would be streamlined. As the core logic is already designed with scalability in mind, the project would be better prepared for the changes in the inputs.

**K. Justify your choice of data structure by doing the following:**

**1. Verify that the data structure you chose meets all the criteria and requirements given in the scenario.**

The data structure in hash\_map.py was all original code and was made without the use of any python dictionaries as restricted by the project requirements.

**a. Describe the efficiency of the data structure chosen:**

A hash map has a worst time complexity of O(N), this means that searching for items within the hash table is very efficient. The O(N) is since each “bucket” or “container” in the hash table might have several values inside of it in the form of a list which would require a for loop to go through. However, a well thought out hashing function should mitigate the risk of having too many items mapped to each “container”. The data structure could also increase in size as needed, although this would obviously increase its space complexity.

**b. Explain the expected overhead when linking to the next data item**

The overhead of linking another data item within the same “container” of a hash table would be largely determined by the use of the “for” loop that is needed to go through a list under a certain a hash key. As it says on the python’s own documentation, “the interpreter overhead of the for loop itself can be a substantial amount of the overhead.” which means that for the program to still be efficient, the number of items in each of these lists should not be too large. However, as previously state, a good hashing function as well as an increase in size for the hash table should mitigate this issue.

**c. Describe the implications of when more package data is added to the system or other changes in scale occur**

When more package data is added to the system, the hash table’s size can easily be adjusted to a necessary value. The way the hashing table is implemented in this program is that it takes in an integer for size in its constructor. This way, the size is not hard coded into the program. The passed integer size is also utilized in the hashing function, meaning the whole system can easily adapt to changes as they are all dependent to a user set value. This was implemented by design as a hash table becomes more inefficient as the number of items stored increases compared to the size of the hash table itself.

The program would also be able to handle an increase in the number of trucks. Since a truck was abstracted into its own class, creating, and declaring properties of a truck could be done with ease. However, the loading process would not be able to assign the packages properly as it was designed with the project requirements in mind. The loading process needs to be refactored to allow for an indefinite number of trucks to be able to scale.

**2. Identify two other data structures that can meet the same criteria and requirements given in the scenario**

A list and dictionary could both have met the same criteria and requirements.

The packages could have easily been stored in a list. However, the functions that an algorithm would need to do its job would create more overhead than necessary. If the program ever needed data store at the last item on that list, it would require a for loop that checks every item on that list. A hash table on the other hand has fast lookups and insertions as each item is mapped by a hash key. The same key that could also uniquely identify the item that being searched for. It drastically reduces the number of items to search through as it would ignore all the other items under different containers.

A dictionary could have also been used to store our package data. However, as the project required, we were not allowed to use python’s built-in dictionary. A dictionary works the same way as a hash table where you provide key value pairs. However, since a dictionary is built into Python, we do not have control over its size, hashing function and its overall logic.

**L. Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.**

M. C., By, -, Marina Chatterjee https://www.mygreatlearning.com/ (2021, April 19). *A quick introduction to KNN algorithm*. GreatLearning Blog: Free Resources what Matters to shape your Career! Retrieved February 25, 2022, from https://www.mygreatlearning.com/blog/knn-algorithm-introduction/

“PythonSpeed/PerformanceTips.” Python, 26 Apr. 2020, wiki.python.org/moin/PythonSpeed/PerformanceTips

**M. Demonstrate professional communication in the content and presentation of your submission**

The contents of this document are well organized. The code of the project is organized and has proper documentation though comments. The user interface has an easy and intuitive presentation and works per project specifications.