C950 WGUPS Algorithm Overview

Syed Saad Khurshid

ID #010081191

WGU Email: skhurs2@wgu.edu

May 10th, 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 with the actual constraints and remaining under the maximum allotted 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, second and third truck and the package location by default is “0” or “At Hub”.
* The trucks are then assigned based on certain criteria and assigned to each Truck separately and those with strict criteria are then allocated to either the Second or Third Truck.
* Once all the trucks have been assigned a package to, then using the nearest neighbor algorithm is utilized where the nearest address is driven to and then from there the next nearest. Each time the shortest route is determined, it’s moved to a shortest path list with the location id only.
* During this time the distance travelled and the delivery time are calculated and updated in each trucks list

# B1. Logic Comments

When originally first creating this, the following was considered:

* Have the packages input into a hash table
* The Packages are to be defined as well as the Trucks
* The Packages are placed into the Hash Table and Packages have to be placed into the Trucks List
* Based on certain constraints the trucks are loaded with packages
* Packages are then delivered using the shortest route which is using the nearest neighbor

1. Creating of the Hash table

***Hash\_table = []***

***A key is generated using the hash hey generator***

***Hash\_key\_generator(key):***

***Return hash(key) % length(Hash\_table)***

Data is inserted in the insert function where it first checks if the Hash key already exists or not otherwise the data is inserted. If it exists then its chained with the key. The data is stored with the original key and data as a list together in another list which is the Hash Table

***Insert(key, data):***

***hashKey = key\_generator(key)***

***if we return empty then it means we can store it in the space***

***Hash\_table[hashKey] = [key, data] // Which takes in and stores the new data***

***Else:***

***For keyValue in Hash\_table[Hash\_key]:***

***If keyValue[0] == key:***

***keyValue[1] == data // This is the new data***

The cost of the Insert function is O(N) due to the for loops.

1. The Truck class is created and consists of a list

***Truck Class:***

***Truck\_List = []***

***Add\_to\_truck(Package):***

***If address is not None:***

***Truck\_List.append(Package)***

***Else return false***

The Package Class is then created which takes in the parameters:

***Package Class:***

***Constructor requirements of id, address, city, state, zip, delivery, size, notes, starting\_time, delivery\_address, status, truck\_number, delivery\_time\_expected***

Package is instantiated and then stored in a list

Package data is looped over. This has a run time of O(N) and sorting into Truck Lists

***Truck\_List\_1 = []***

***Truck\_List\_2 = []***

***Truck\_List\_3 = []***

***For row in data\_source:***

***PackageA = Packages(id address, city, state, zip, delivery, size, notes, starting\_time, delivery\_address, status, truck\_number, delivery\_time\_expected)***

***Value = [PackageA]***

Based on conditions the Packages are then sorted to each truck. For example, if the expected Delivery time is not a earlier time then its added to Truck\_List\_3, or other conditions its appended to other trucks

***Truck\_List\_(Based on Condition, truck 1, 2, or 3 could be selected).append(Value)***

***Insert\_Into\_Hash\_Table (id, Value)***

1. Index list is created of the address location the packages on each truck

This is run time of O(N) and its used to retrieve the index from the address data by searching an comparing address

***Truck\_List\_Index = []***

***Get\_index(Sorted\_Truck\_List\_1,2 or 3):***

***For address in address\_data:***

***If address[2] == Truck\_List[2]***

***Return Index***

Adding to the Index list

***Set\_Index\_Values (Truck\_List\_Truck\_Number):***

***For Values in Truck\_List***

***Index = Get\_Index(Address from Value)***

***Truck\_List\_Index.append(Index)***

1. Now determining the shortest path after all the indexes are sorted out. Using a list to store the next best point in the Truck\_Lists and appending it. The Current Starting Point is always going to be the hub but is updated to the next visited location

***get\_shortest\_path(Truck\_List\_Index):***

***Starting\_Point\_List = [‘0’] with 0 being at the Hub***

***Current\_Point = “0” which is the hub***

This is at the cost of O(N4)

***While Truck\_List\_Index is not empty:***

(This is at the cost of O(N)

***For index in Truck\_List\_Index:***

(Another function checks which is the closest point starting from the Hub)

(This process returns at a cost of O(N2)

***Shortest\_Point = get\_shortest\_point(current\_Point, Truck\_List\_Index)***

(Please refer below to see function explanation. The shortest point is determined by checking the distance from the current point to the

Next point by checking each Point in the Truck\_List\_Index by looping over each point and checking the distance from the current point at a cost of O(N2). The once with the shortest

Distance would be the next best point)

***Starting\_Point\_List.append(Shortest\_Point)***

(The New Starting point is then changed as the current point)

***Current\_Point = Shortest\_Point***

***Truck\_List\_Index.pop(Shortest\_Point)***

***(Remove the Point from the Truck Index List, as its now visited)***

After the Truck\_Index\_List is now empty. The distance is calculated using another function

***Total Truck Distance = Calculate\_Distance(Starting\_Point\_List)***

The function above reaches out to “**get\_shortest\_point**” which also reaches out to another function called “get\_current\_distance” and is defined as below. It takes the current\_Point from the “**get\_shortest\_path**” function above and the next point in the Truck\_List\_index to get the next point with the shortest distance excluding those with distance “0” because that would mean already at the same location. This has a run time complexity of O(N2)

***get\_shortest\_point (takes current point from the above function, Truck\_List\_Index passed from the above function):***

***temp\_distance = 0. (Temporary distance measure)***

***index = 0*** (This is the default starting point for all trucks which is the hub)

***For ind in range of Truck\_List\_Index:***

***distance = get\_current\_distance (current\_Point, Truck\_List\_Index[ind])***

(This is explained in the lines below but it obtains the distance from the current point to the next point. Run-time complexity of O(N))

***If temp\_distance equals Zero:***

***Then temp is equal to the distance*** (this is a check in case if both current\_Point and Truck\_List\_Index [Ind] does not produce a distance of zero)

***Else if distance < temp\_distance then temp\_distance is equal to distance***

***Index = Ind***

***Return Index***

In this function the distance from current\_Point to the next point in the Truck\_List\_Index[index] passed from the **get\_shortest\_point** function above. It searches for the distance between the points and returns the distance once it finds it. This has a run-time complexity of O(N)

***get\_current\_distance (current\_Point, Truck\_List\_Index[index]:***

***distance = distance\_CSV[current\_Point][Truck\_List\_Index[index]]***

(distance\_CSV is a csv file that has the distance of each location and the table recognizing each location by index. The table has rows and columns that points to each distance. So, for example the Hub’s index is “0” and the next point has a index of “1”, in the distance\_CSV the table would look at row “0” and column “1” to get the distance of for example “2.1”. Run time complexity is O(N))

***If distance search in distance\_CSV is zero:***

***distance = distance\_CSV[Truck\_List\_Index[index]][row]***

***return distance*** (We are able to get the distance from Point A to B)

# B2. Development Environment

The programming model for this application is currently hosted on a local machine. The specifications are 2018 MacBook Pro Intel Core i7, 16gb RAM, 1TB M.2 Samsung SSD with MacOS Ventura 13.3.1. IDE being utilized is Visual Studio Code v1.78.1 with Python version 3.11.3 64-bit version. All data is stored locally on local hard drive in CSV file format that is currently stored in a data folder were using a data reading function utilizing the “csv” python library where the data is then read. The data exchange is only local and thus no external threat as one would on a network connected environment.

# B3. Space-Time and Big-O

The runtime complexity overall of this application is O(N4). The main function behind the complexity is the to get the shortest\_path function which has nested functions that also have nested loops that in order to determine the shortest path and has a run time complexity of O(N2).

The run-time complexity overall of this application is O(N4). The main function behind the complexity is the get\_shortest\_path function that while it has to look over the Truck index array list list that is passed in, that it is searching through and removing points that have been confirmed to be the next shortest point. That is determined while looping over the remaining points in the Truck Index List and looping over it which has another function that has nested loop O(N) and has internal function which also has a run-time of O(N). The function is also looping again to determine the next closest point until the Truck Index List is empty and all points have been visited, that has a run-time of O(N2). Combined it has a run time complexity of O(N4)

The hash table on the other hand has a space time complexity of O(n) which each of its functions having an O(1) time complexity (Lysecky, Section 2.6)

# B4. Scalability and Adaptability

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, its 16 packages. Also, the nearest neighbor method looks at the closest distance to the next point. So, if additional travel destinations were added then it would still find the nearest point till the end. Since it doesn’t depend on anything else besides the destination points and distance along with packages one truck can carry, it can scale with additional capacity of trucks and thus also allowing additional destination points to travel while traveling the shortest path.

# B5. Software Efficiency and Maintainability

The software has a good structure to build upon and is efficient when compared to similar applications that do not use a hash map. I refactored what was over 300 lines of code to be as minimalistic as I could make it. The goal was to build upon a skeleton using classes and functions that make it easier to scale up with the classes being easily maintainable while also being able to adjust accordingly if further scaling is necessary. Having a mix of classes for Trucks, Packages, Hashing and Travel while function based can you used for any other purpose if necessary, allowing it easily scalable and maintainable.

# B6. Self-Adjusting Data Structures

The main strength of the chaining hash table is how fast its data can be interacted with. Having a data structure that can update, remove, and insert data in linear time is a major upside. Another upside is that it is able to handle collision gracefully. These strengths combined allow for further scalability for further larger data. As mentioned, that it’s a chaining hash table, with chaining its ability to handle large amount of data may result in some of the buckets assigned to the table being left empty and wasting space or not being fully utilized. All can be considered a minor disadvantage since its speed is a major advantage.

# 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). The hash table has a limited capacity of having 10 buckets then using pythons hash function and mod (%) function a hash key is generated which is then inserted into the respective bucket. However, to avoid collision inside the bucket, a new package is appended to the list of packages already present in the respective bucket.

Having a hash key helps provide a more efficient way of searching than a linear search due to the fixed number of buckets (10 buckets) and has a space complexity of O(1). The search may not need to search through all the items as would happen in a linear search.

# E. Hash Table

1.

1. class HashingData:

2. # Constructor with optional initial capacity parameter

3. # Buckets are assigned to an empty list

4. def \_\_init\_\_(self, initial\_capacity=10):

5. self.data = []

6. for \_ in range(initial\_capacity):

7. self.data.append([])

8.

9. # Generating hash-key which is a O(1)

10. def hash\_key\_generator(self, key):

11. return hash(key) % len(self.data)

12.

13. # Inserting a new item into the hash table

14. def insert(self, key, item):

15. Hkey = self.hash\_key\_generator(key)

16. # First to check if the Hash Key does exist of not

17. if self.data[Hkey] == None:

18. self.data[Hkey] = [key, item]

19. return True # End the function to continue to add more later

20. else:

21. for keyValue in self.data[Hkey]:

22. if keyValue[0] == key:

23. keyValue[1] = item

24. return True

25.

26. self.data[Hkey].append([key, item])

27. return True

28.

29. # Searching for an item with matching key in the hash table

30. # Returns the item if not found, or None if not found

31.

32. def search(self, key):

33. # get the bucket list where the key is located

34. Hkey = self.hash\_key\_generator(key)

35. # In case the key doesn't exist

36. if self.data[Hkey] != None:

37. for items in self.data[Hkey]:

38. if items[0] == key:

39. return items[1]

40. return None

41.

42. # Removes an item with matching key from hash table.

43. def delete(self, key):

44. Hkey = self.hash\_key\_generator(key)

45. bucket\_list = self.data[Hkey]

46. if self.data[Hkey] == None:

47. return False

48. else:

49. for keyValue in bucket\_list:

50. if keyValue[0] == key:

51. bucket\_list.remove(keyValue[0], keyValue[1])

52. return True

53. return False

54.

55. # To Update the item in a key

56. def update(self, key, value):

57. Hkey = self.hash\_key\_generator(key)

58. # Now to search for hashed line item if its been found or not

59. if self.data[Hkey] != None:

60. # Now if they match then we update the second list in the data

61. for items in self.data[Hkey]:

62. if items[0] == key:

63. items[1] = value

64. return True # This should stop any further updates

65. # In case the key doesn't match

66. return None

67.

68. # This is to get the self table

69. def get\_hash\_Table(self):

70. return self.data

71.

72.

# F. Look-Up Function

1. # Searching for an item with matching key in the hash table

2. # Returns the item if not found, or None if not found

3.

4. def search(self, key):

5. # get the bucket list where the key is located

6. Hkey = self.hash\_key\_generator(key)

7. # In case the key doesn't exist

8. if self.data[Hkey] != None:

9. for items in self.data[Hkey]:

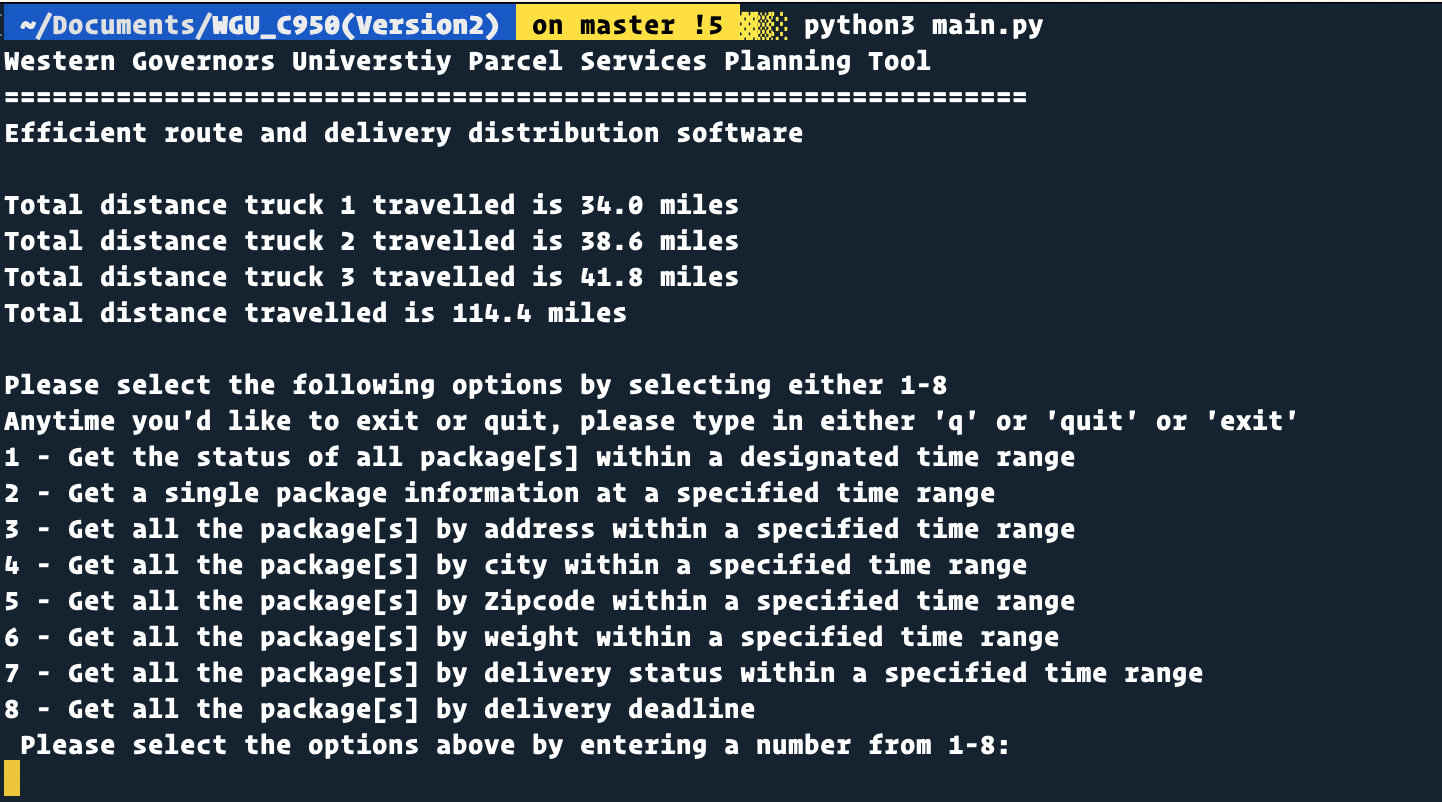
10. if items[0] == key:

11. return items[1]

12. return None

13.

# G. Interface



# G1. First Status Check (Also can be viewed in Screenshot folder)



# G2. Second Status Check (Also can be viewed in the Screenshot folder



# G3. Third Status Check (Can also be checked in Screen shot Folder)



# H. Screenshots of Code Execution

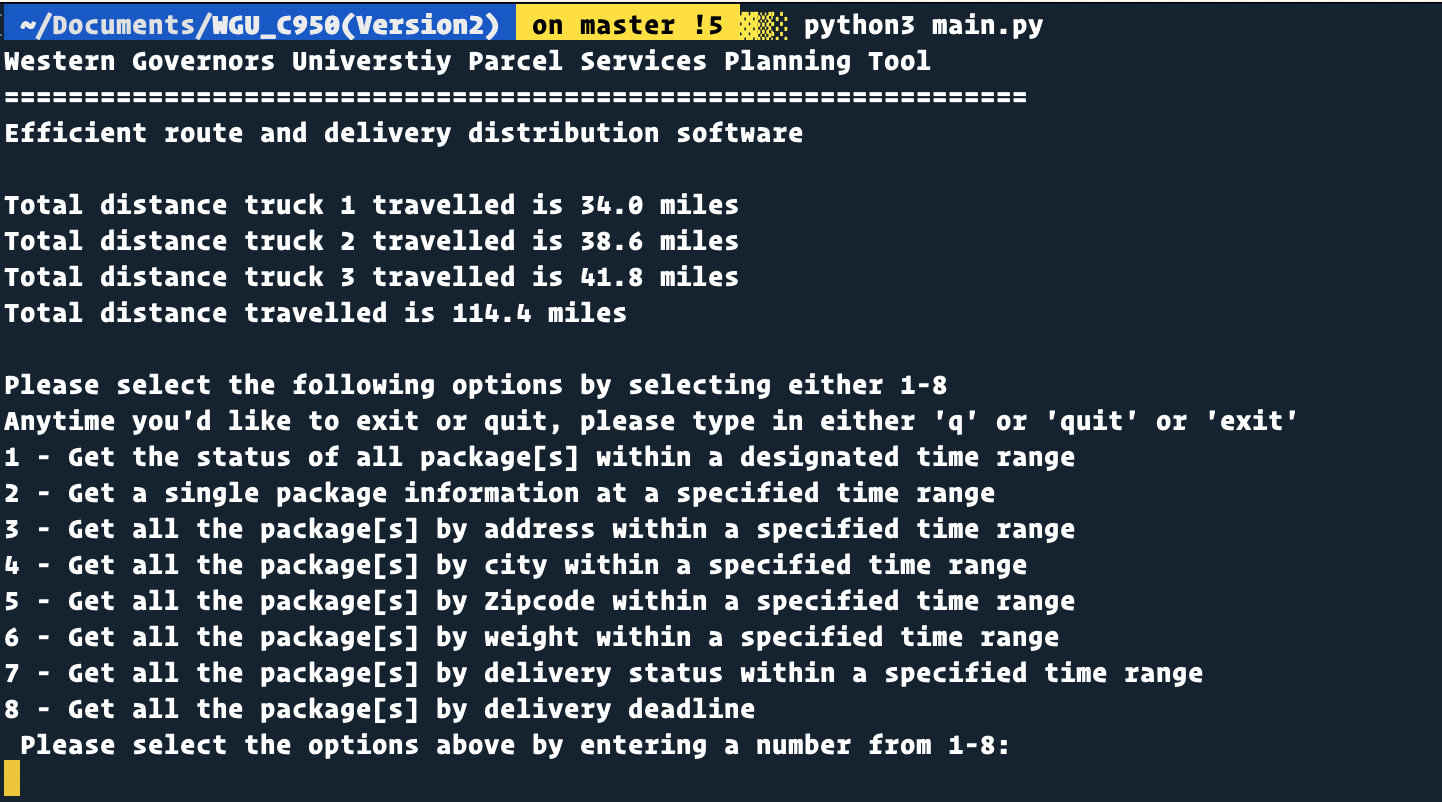
Please see in the Screenshot folder to see by Package ID, Address, City, Delivery Deadline, Weight of Package, Delivery Status

# I1. Strengths of Chosen Algorithm

One of the strengths of choosing the Nearest Neighbor Algorithm is its simplicity to implement where it only assumes the shortest or nearest distance to the next point or vertex. This makes it easier to scale as we increase the size of data which in this case can be more packages to deliver. The other advantage is that it requires very few parameters to be implemented which in this project means that only 16 data points are required as that’s the limit of packages each truck can carry. Should we scale further, the limitations can be changed accordingly for in this case more than 16 points and the Nearest Neighbor Algorithm will not be affected by the large size of data and determine equally the shortest route accordingly

# 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 114.40 miles.



# I3. Other possible Algorithms

Another Algorithmic approach I could have used is Breadth-First-Search Algorithm when trying to find the optimal route for Delivery for the trucks. Breadth-First-Search algorithm is for traversing or searching a tree or graph data structure which visits each vertex starting from the nearest vertex and then expands to other vertices without revisiting a vertex and determines the optimal solution which in this case would be the best delivery route for the trucks to utilize

A second algorithmic approach I could have utilized for the delivery routes is to utilize the Dijkstra Algorithm. Dijkstra Algorithm determines the shortest path from a start vertex to each vertex in a graph. For each vertex, Dijkstra’s algorithm traverses through each node visiting each vertex and its determination of the shortest path are based on weight of the edge for example in this case it can be the distance, or the time to travel to that vertex.

# I3A. Algorithm Differences

A Breadth-First-Search (BFS) is a traversal that visits a starting vertex, then all vertices of distance 1 from that vertex, then of distance 2, and so on, without revisiting a vertex (Lysecky, Section 6.5). In this case the BFS Algorithm determines each vertex weight of the edges by distance. This can provide and an optimal route. One of the ways it differentiates from the current project is that it places the nodes as visited and not visited which would avoid revisiting the same location twice in the same route that currently the nearest neighbor algorithm doesn’t avoid.

Dijkstra Algorithm unlike Breadth-First-Search doesn’t simply determine the optimal route by its distance only. Rather Dijkstra applies weight to each Edge so it can be the travel time as well as the distance. Dijkstra also visits each vertex by the weight of edges and then determines by shortest path which is not necessarily the shortest distance from each vertex and the next that BFS does. This can give an additional advantage when determining the shortest delivery route as we can add additional factors like the package weight, travel time or stop signs or intersections that can add weight to the edge and thus coming up with the shortest delivery route

# J. Different Approach

Currently the project uses nearest neighbor which does not take into account of any other limitations for example if there is a stop sign on the road or other intersections in the route and simply assumes it’s a straight path. This is unrealistic and as we scale further up, the nearest neighbor might not actually provide us the shortest route. Dijkstra algorithm traverses through each node visiting each vertex and takes into account any weights of the edges, for example in the delivery of package as mentioned if there is an intersection with a stop sign that may add time to travel, these conditions can add further weight to the edges and may end up being a longer route. By having these weight in the edges, Dijkstra goes over each point starting from the starting node which is the hub in this case and visits each point considering the weights of the edges, and determines the shortest path which can actually bring in a shorter delivery time and shorter distance travelled.

This project also provided weights of the packages, the heavier the package the more fuel the truck would have to use to run such a heavy load. Using the package weight can be used as additional weight for the edges in consideration and can come up with a different route. Having the truck run on routes that has less stop signs for example and having the truck move at faster speeds would mean faster deliveries and making the truck fuel efficient.

# K1. Verification of Data Structure

If I were to do this project again, I would perform the data structure in a graph based route using Dijkstra Algorithm instead of the greed methodology because of the factors taken into the fact that using Dijkstra 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

The chaining hash table is a great option for this assignment as it is able to use load/update packages as needed in a speedy manner. The biggest justification is that the packages are delivered on time taking into account the constraints. The time needed to look for packages would not change with more or less packages since hash tables perform searches in O(1) time. With current restraint of the hash table set to 10, if that turns out to be too small as we scale further, then the worst-case scenario is a search that takes O(n) time which is still optimal.

# K1B. Overhead

Currently the hash table is constrained to 10 buckets, if the number of packages were to increase as we scale further, the hash table could have a space complexity of O(n) which is still optimal.

# K1C. Implications

Increase in delivery trucks and cities would increase space usage proportionally to the Number of Trucks in the fleet as each truck has a own list of packages and time to deliver. Should further expansion of cities to deliver, would then increase further usage of space possibly exponentially since the Greedy method is looking currently only references the nearest point. The time to look up will remain O(n) regardless of these changes

# K2. Other Data Structures

If I were to utilize anything besides Hash table then that would be using Dictionaries instead of Lists which is currently what the Hash Table consist of. A dictionary associates or maps keys with values; key is a term that can be located in a dictionary and value is that is associated with a key (Lysecky, Section 9.1). The packages can each have a key which in this case would be the Package ID and be an alternative to using a Hash Key for accessing the Package data.

Another form if I were to use would be utilizing the Binary Tree for storing of the Packages. In a Binary Tree, each node has up to 2 children, known as a left child and a right child (Lysecky, Section 4.1). A Binary tree is represented by a pointer to the topmost node which is the root of the tree. We can apply this to sub categories the packages into further categories, for example currently we have weights for Packages, and is currently not a determining factor, but in this case the weight of the package can be categorized which can then further be split into further categories. This can be useful in allocating packages and other factors can be utilized for example it can be stored by type of package for example if its electronic package, or letters, etc.

# K2a. Data Structure Differences

The Current Hash Table in the project creates a hash key to be associated with the List itself and is then appended into another lists. The Dictionary on the other had already uses a hash key that is unique and does not allow duplicates to exists. Due to its unique way of already creating a hash key, it’s easier to search through a dictionary and has a constant time complexity of O(1) whereas the current hash table being a list has to iterate over each hash key till the key that is being looked for is found, this adds a complexity of O(n) which would increase as the size of the list increases where as Dictionaries remains constant and would be more efficient as we scale further.

If I were to use Binary Tree Data Structure uses a sorted List which is an alternative to a Hash Table Data Structure. As mentioned above, Binary Tree uses a sorted List which can be used to store the Packages by different categories that can further be categorized for example the packages could be categorized as electronics, equipment, etc., or in this project then weight of the Packages is one factor that can categorized with. Binary Tree has a run time complexity of O(Log N) which means much faster due its non-linear nature and more memory efficient than the current Hash Table search complexity of O(n) which would not be efficient should we scale further and thus can help when larger data for packages can be sorted.

# M. Professional Communication

Text goes here

# 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/>