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 creating this, the following parameters were being considered:

* Truck List (represents a nested list of packages on a given truck)
* Assigning a truck number (Represents the truck you are working with in this case is 3 Trucks)
* Current Position or Location of the trucks. Determining the best route for the truck to take using the nearest neighbor method for each package starting from the Hub

We create the truck list of indices of each truck location

***Truck\_index\_list is created with only the index of each package address***

The Truck index List is then passed to the shortest path function to obtain the shortest route

***Set Starting\_Point List to “0” which is the hub***

***We remove any “0” from existing Truck Index List that was passed we cannot have repeated Hub points***

***Set Current Location = “0” which is the Hub***

***While the Truck index List is not empty:***

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

***We get the shortest point in the Truck Array List with the Current Location and the next best point in the Truck\_Index\_list (Goes through and compares the next best point in the Truck\_Index List, for example if [0,4,5,9,10], the next best point to go from 0 would be 9 and 9 would be returned)***

***Next best point is appended to the Starting\_Point List***

***We remove best point from the Truck\_Index\_list (This is to ensure that the same points are not revisited)***

This loops until the Truck\_Index\_list is completely empty

***Once the shortest path is determined and stored in the Starting\_Point\_List. Shortest distance is calculated***

***In another function we assign the Shortest Path list, the Truck List and Truck number it belongs to and update its time***

***Calculate set\_time(Shortest\_Point list, original Truck List with Addresses, truck number and hash table):***

***Each truck is assigned a starting time at first and stored in a temp list***

***TempTime List which has different starting time***

***For index in Shortest\_Point\_list):***

***Distance is calculated for the shortest route from starting point to the next point in the Shortest Point list***

***Time\_travelled is calculated get\_time\_travelled(Passing, Distance, TempTime) (This calculates and adds time to the distance travelled resulting in the delivery time)***

***For index in original Truck List:***

***If the address matches the address in the shortest path list:***

***Original Truck list delivery time is updated and updated also in the hash table***

The algorithm continues until all the truck lists are updates with the delivery time

# 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 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). 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 zip code or status

# 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]:

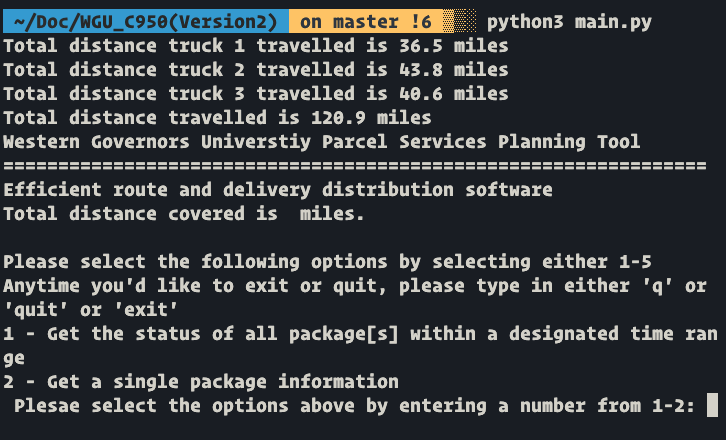
10. if items[0] == key:

11. return items[1]

12. return None

13.

# G. Interface



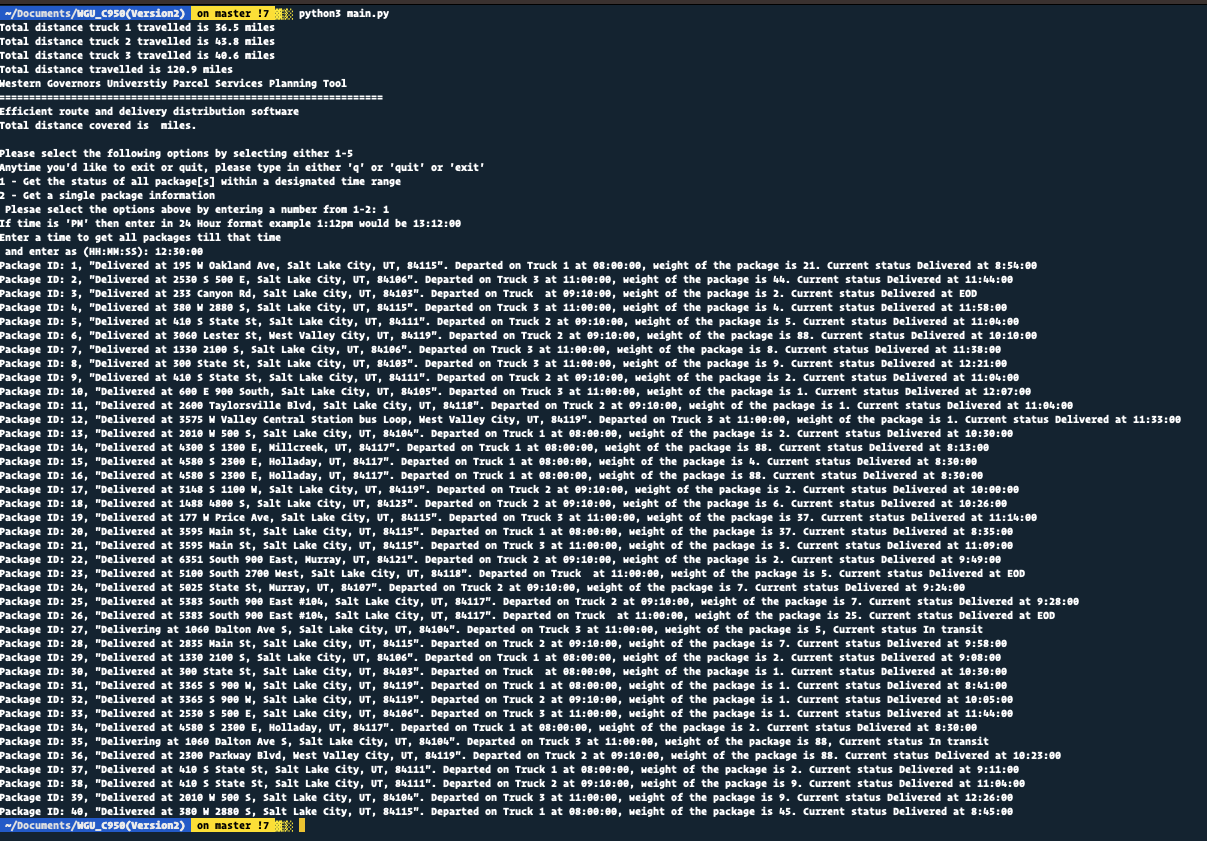
# G1. First Status Check



# G2. Second Status Check

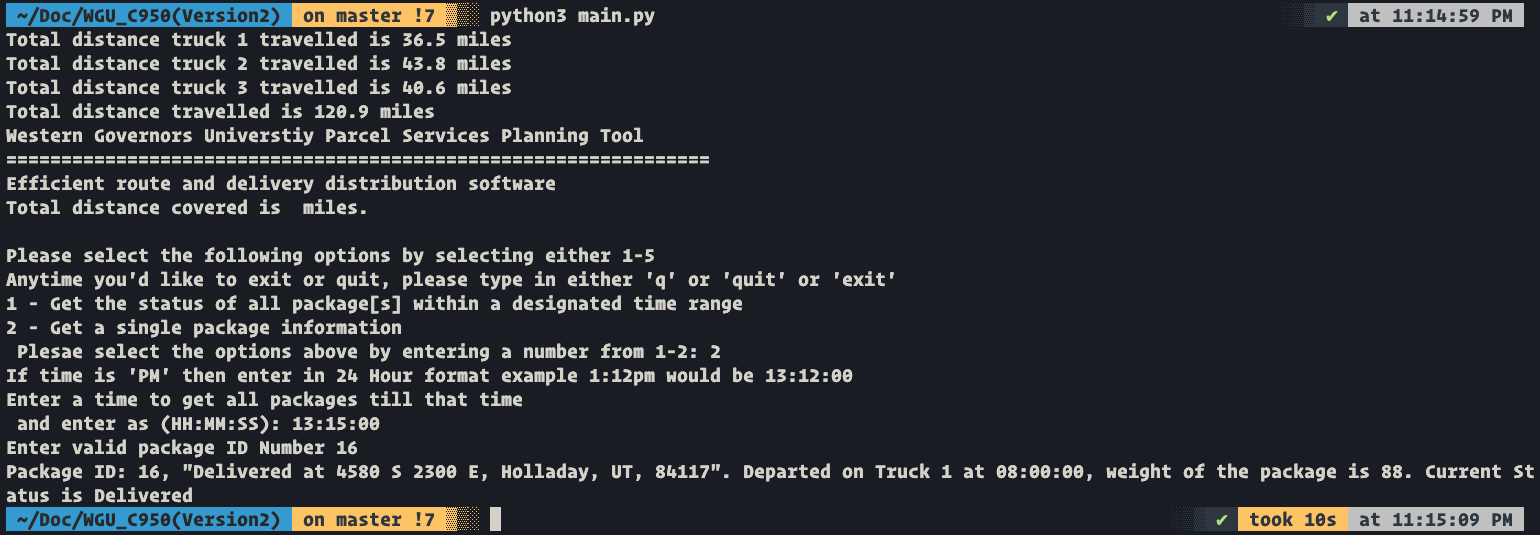


# G3. Third Status Check



# H. Screenshots of Code Execution

Search Package by ID

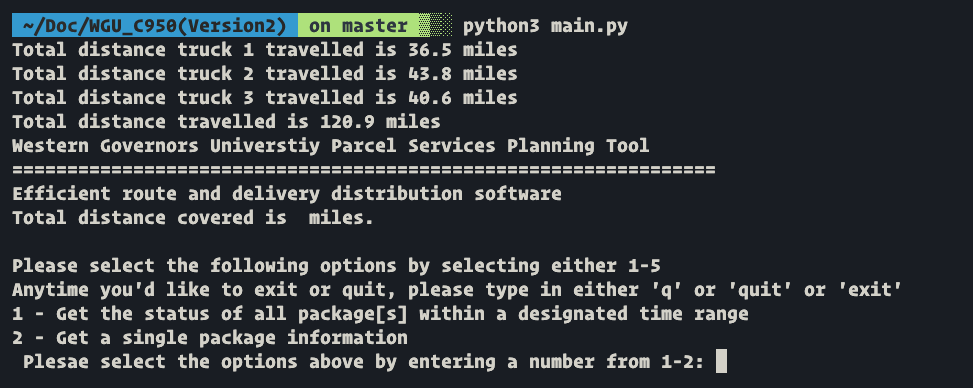


# I1. Strengths of Chosen Algorithm

One of the strengths of choosing the Nearest Neighbor method is due to the fact that its simple to implement. It has no assumptions and simple constraints can be applied easily for example limiting the number of packages a truck can carry which in this case is 16. Due to this simplicity, it would be able to easily scale further up with more packages and larger data points.

# 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 120.9 miles.



# I3. Other possible Algorithms

Looking back a different Algorithm could have been utilized for example the Dijkstra’s Algorithm could have been utilized. Although we are using greedy algorithm, we could have utilized graphs and vertex points. This algorithm visits each vertex points and then determines which is the best point. In this case the distance from each point can be considered a vertex point and the Dijkstra’s Algorithm would visit each point and based on that determine the best possible route until the maximum travel length is achieved. This would be good for future scaling.

# I3A. Algorithm Differences

The difference between Greedy algorithm and Dijkstra Algorithm is that the greedy algorithm simply considers the next best point and then from there which would be the next best point. This is simpler in terms of achieving a shorter route but is not exactly the most optimal route. The Dijkstra Algorithm on the other visits each point which is determined as a vertex, the vertex distance is considered from the predecessor point. It then determines the optimal route by visiting each vertex and then determining the shortest route which may not necessarily be the nearest point and in this case the nearest point could actually be better point to visit on the way back. Dijkstra also takes into account other factors and in this case things like the package weight can be considered.

# J. Different Approach

Dijkstra determines the shortest path from the start vertex to each vertex in the graph (Lysecky, 6.11). The start vertex is the Hub point and the package delivery address are the next vertex points. Dijkstra visits each vertex point and then determines the most optimal route. The greedy method only simply looks at the shortest path from Point A to Point B and then the next shortest path after reaching the next point. This may bring in a shorter router but it’s not necessarily the most efficient route. Another difference is that with the Dijkstra Algorithm other constraints or attributes can be applied when determining the shortest route for example for heavier packages the shortest route would not best route so it can be used as a factor as we further scale in the future

# 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 Djikstra 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 then I would use the Binary Search Tree data structure which would also work for this application

# 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

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/>