# C950 WGUPS Delivery Tracker Overview

Uri W. Easter

10/17/2023

C950 Data Structures and Algorithms II

## A.  Algorithm Identification

The Nearest Neighbor Algorithm was elected in this program as the solution for this traveling salesman problem. Nearest Neighbor finds the minimum distance between a grouping of points and maps to the closest proximal location using the smallest distance identified. This algorithm was one of the best choices given for this problem since it serves the purpose of mapping an efficient path in terms of distance. This suggests that the trucks will travel on one of most optimized paths for delivering all packages quickly and well within the required distance limit of 140 miles.

## B.  Program Logic Overview

### B1. Algorithm Pseudocode

Pkg\_load = [list of package ids] // initialize pkg list

Truck\_current\_location = 0 //set current location of truck

Minimum\_distance = 1500// set minimum distance to a big number for comparison

next\_package = None // the next package is set to null

While pkg\_load > 0: //will keep going while pkg\_load is not empty

For pid in pkg\_load: // iterates for each package in the list

       if calc\_distance(address\_index(truck.address), //see if the addresses

                             address\_index(p.address)) <= next\_address: // of the truck and package are less

//than the next address

                next\_address = calc\_distance(address\_index(truck.address), //set the next address  
                                             address\_index(p.address))  
                next\_pkg = p //set the next package to be delivered  
                # print("next package = { " + str(next\_pkg) + "}\n")  
        # Adds next closest package to the truck package list  
        truck.pkg\_load\_r2.append(next\_pkg.package\_id)

print(str(truck.truck\_name) + " TIME: " + str(truck.time) + ", DISTANCE: " + str(truck.tot\_miles) + "\n" +

              str(next\_pkg) + "\n") // outputs truck name, current time, current distance

// and last package delivered

### B2.  Development Environment

The programming environment for this program uses Pycharm 2023.2, and Python 3.10.7

The hardware is an Aspire A515-54 with an Intel Core i5-8265U, a CPU with processing speeds of 1.60GHz to 1.80 GHz and a 64-bit operating system using Windows 11.

B3. Program Space-Time Complexity  
This program runs in O(n2) time complexity. See the attached program

### B4.  Program Adaptability

For a growing number of packages, we could first increase the number of trucks to deliver more packages, and if they have a different maximum, we could reset the value for the attribute of pkg\_max to a higher number to deliver more packages. Further actions we could take are implementing an add package function to allow users to add packages and increasing the hashindex (after clearing the hashmap) relative to the number of growing packages.

One of the difficult things to account for scaling up is that more addresses means we would need to switch from adjacency matrix as that isn't maintainable when you have large numbers of addresses (greater than fifty or one hundred).We could keep writing data to a csv file, but in reality, for industry we may want to connect to a database like SQL using SQLite library in python or MongoDB using a Pymongo driver, depending on the  app design and preference

If the trucks were allowed to go faster, like average speed in real life, that may also hasten the number of deliveries (Union Pacific 2023).  
I would also need an algorithm to predict which truck should load which packages. Nearest neighbor predicts nearest locations to each other, but given the complexity of constraints in this problem, for massive amounts of packages, and existing package limits, a more advanced prediction tool may be needed to allot a number of packages to the correct trucks (Saxena 2020).  
I knew the inconveniences because they were project requisites, in real life people most likely will not know many of the inconveniences faced ahead of time, nor will packages be delivered instantaneously and therefore delays in predicted delivery times will be slower.

### B5.  Software Efficiency & Maintainability

The time complexity of the entire program O(n2) by worst case scenario. Other functions within the program function at O(1), or O(n) time, which suggests that on the whole this program is efficient especially since it does not run in O(n!) time or O(2n).  
Further, I commented extensively on each major block of code, followed naming conventions, and employed functional programming practices, which means if a new developer needed to understand or revise my code, they should in theory be able to do so with ease (Finer 2018). Lastly, but not least, I also pasted sources that helped me derive some of the code solutions to the traveling salesman problem. The sources give insight to conclusions about the efficiency of the how the code operates, and the logic that has been adapted to the expectations of this program.

B6.  Self-Adjusting Data Structures  
One of the strengths of the HashMap is the O(1) time complexity for object access time (Saxena 2020). Another one of the benefits of using the HashMap data structure is that values map sequentially from the order set by the hash index (Saxena 2020).  
One of the weaknesses I observed is that it is difficult to get the size of all objects within the HashMap for scaling without a list or for loop since a HashMap does not "maintain the same order of items in a collection"(Strmecki 2020). There can be many values per index, and there appears to be no way of getting the count of items other than directly calling the number of items as a parameter from the lookup function, or through iteration rather than using len() or size() functions to measure the amount of values in the HashMap.   
Another one of the weaknesses of HashMap is that a greater number of keys increases the chances of collision. When there are fairly limited amount of keys, operation stays at O(1) time. As the keys increase, and potentially the hash index, it is possible for the HashMap to start processing closer to O(n) time if the key-value ratio begins to get overloaded (Ruane 2011).

## C.  Original Code

1. Identification Information  
   The identifying comments is in the code.

2.  Process and Flow Comments

Comments in the code explain the flow of the program.

## D.  Data Structure Explanation

The data structure used for this program to facilitate the access needs of the nearest neighbor algorithm is a chaining hash table/ hash map which is a form hash map. The book for this course defines a hash table as "..a data structure that stores unordered items by mapping (or hashing) each item to a location in an array (or vector)…the modulo (remainder) operator can be used to map[to].. a bucket index from the item's key".  
  
The chaining hash map differs from the hash map data structure in that it has a method of mapping a list of values to a bucket for insertion and search from the key, sometimes referred to as the hash index. Still, as the Chaining HashMap is a form of HashMap it still shares the benefits of O(1) time complexity which is faster than the O(n) time complexity of linear search, and the reduced space complexity of the Chaining HashMap means that when a bucket is identified less time is executed per search for fewer items as compared to having to search an entire list for an item in a linear search.

In the HashMap a list of packages is designated to a bucket using a modulus of ten on the package id to ascertain their location. In the nearest neighbor algorithm, we use the attribute of the package id from the Truck’s list of package id’s known as package\_load. This enables us to search for a package object within the HashMap using package id and add it to a list of package objects.

   Then while the list of package objects is greater than zero, we use the calc\_distance function to compare the address indices of the truck and packages to the nearest possible address, then we assign that address as the next address.  
The HashMap here plays a pivotal role in allowing the program to go from taking the id of the package, to looking up the package object so the addresses can be compared using the address\_index and calc\_distance function.

## E.  Hash Table Insertion

See the code.

The hash table has an insertion function that takes the following components as input and inserts the components into the hash table: delivery address, deadline, city, zip code, status, package ID number and weight.

## F.  Search function evaluation

See the code The track\_one() and track\_all() functions in the code (options 3 and 4 in the user interface) outputs all the following data elements for the package ID number:delivery address, deadline, city, zip code, status, and package weight.

## G.  Status Info Screenshots

--add files

See “Part G” of the “Screenshots” folder for parts G1 through G3.

H.  Code Execution screenshots

--add files  
See “Part H” of the “Screenshots” folder.

## I.  Core Algorithm Justification

### 1. Two Strengths of Nearest Neighbor Algorithm

Two strengths of Nearest Neighbor algorithm are that it does not need a training step, and that it uses a “point pattern analysis model to find closest points nearest each other in space”(Xristica 2018, Madhushree 2012). When searching for how to implement the Nearest Neighbor algorithm many results for K-Nearest Neighbor would show up. Some difference between the two is that K-Nearest Neighbor uses a hyper-parameter k to define the number of points to cluster for training. In the training phase the algorithm identifies the number of points that can be classified as a group to depict patterns in data. K-Nearest Neighbor was beyond the scope of this project, and using point classification was not needed for the trucks to identify where to travel since the calc\_distance and address\_index functions both served to return nearest distances as if by adjacency matrix. Both functions just mentioned helped within the nearest neighbor algorithm executed “point pattern analysis” by calculating the next shortest point within distance of the current truck and package locations, thereby keeping the distance well within range of the requirements.

### 2.  Requirement Fulfillment of Nearest Neighbor

The algorithm used in this solution meets all requirements in this scenario. Each truck can carry a maximum of 16 packages, and the ID number of each package is unique. The truck can carry up to 16 packages since its representation of the truck object has its attribute of pkg\_max is set to sixteen. The pkg\_max attribute maintains the maximum number of packages until manually altered. To maintain the uniqueness of id, each id is entered individually into one of two lists per truck object in accordance with a planned route associated with the list the id is sorted into.

To ensure that the trucks travel at an average speed of eighteen, the truck objects have the value of avg\_mph set to eighteen once the objects are instantiated. Representing a limitless supply of gas originated from not setting any variable for gas supply. Expressing no need for the trucks to stop in this program was the result of not developing any conditional logic that emulates driving rules which would require the trucks to stop. Not developing any conditional logic that emulates driving rules also negates the ability for collisions.

Assuming there is one driver per truck, and that each driver stays with the same truck if it is in service, the minimum requirement in this instance is simply that each driver has a truck. Therefore, only two truck objects were instantiated in this program and their combined total distance of 124.7 miles meets the requirement of being under 140 miles.

Neither driver leaves the hub earlier than 8:00 a.m. referred to as the pkg\_loadtime in this program, and only after pkg\_loadtime do packages begin to go en route. The drivers do return to the hub to reload packages for their second round of deliveries. This can be confirmed in the user interface by selecting any of the options from option four to option seven followed by selecting option eight.

Loading packages onto the truck is an instantaneous operation, and delivering packages is instantaneous as well since no time delay was factored into the time measured for these processes. Each package has the capacity for one special message at most. The conditional logic for package status in the track\_one() and track\_all() functions correct the delivery address for package nine to 410 S State St., Salt Lake City, UT 84111 at the specified time of 10:20 in the morning. Lastly, due to the calc\_distance function distances returned are equal regardless of the direction traveled. The results of this can be observed in folder “PART G” of the “Screenshots” folder by checking the “10AM\_TEST” and the “1245PM\_TEST” folders.

Finally, between both sets of delivery routes executed by the trucks all 40 packages were delivered on time, according to their delivery requirements.  One could check this information in the user interface by selecting option one “Check Full Delivery Cycle”, in which the program will have both trucks run both of their routes before outputting a report of the final delivery status of all packages and the final mileage status for both trucks.

3.  Alternative Algorithm Choices

Prior to electing Nearest Neighbor algorithm, I researched many different algorithm types to see which may be best suited to solve this traveling salesman problem in python. Nearest Neighbor naturally was one of the best, however two other contenders for top place that I would explore later in depth are the k-Opt Algorithm, and the genetic algorithm.

1. Algorithm Differences  
     
   Nearest Neighbor searches for the nearest location given a series of locations, and then moves in that direction. Both algorithms below review a series of iterations to find the most optimal path being the path with the least total distance.  
   The first algorithm to discuss is the k-Opt Algorithm more commonly expressed in the form of the 2-Opt Algorithm. The 2-Opt Improvement finds the most optimal pathway by continuing to swap pairs of edges until the best route is found (Reducible 2022). This method has been described as a “local search algorithm for solving the TSP” (Solving Optimization Problems 2021).   
   If the numbers of edges being swapped were now three edges at a time rather than two the algorithm would be a 3-Opt Algorithm, and for any “k” number of edges being swapped using this algorithm the logic follows that the method classifies as a k-Opt algorithm (Reducible 2022). For larger numbers of k, the number of potential solutions increases since this algorithm generates every possible combination as it swaps edges, and this output lengthens the time of execution for the algorithm (Solving Optimization Problems 2021, Reducible 2022). While this is a prime candidate for solving this form of Traveling Salesman Problem known as the Vehicle Routing Problem, it has been suggested that one of the better approaches to implement this technique may be to pair it with heuristic algorithms like the Nearest Neighbor (Solving Optimization Problems 2021, Reducible 2022).  
   The next algorithm to discuss is the Genetic Algorithm.  John Holland and his students created the Genetic Algorithm from the inspiration of Charles Darwin’s Theory of evolution known as the survival of the fittest (Ruiz 2022). In survival of the fittest the most fit organisms survive from generation to generation to pass on their more adaptive genes while the least fit organisms tend to perish thereby ending the possibility for their genetic trait to proliferate. The result is that the gene pool of generations tends to improve by producing more fit organisms that in turn reproduce. While the expression of the Genetic Algorithm occurs randomly in nature, the random behavior of the Genetic Algorithm outperforms random behavior expressed by local search algorithms such as k-Opt (Ruiz 2022). The Genetic Algorithm is very useful when there is a large quantity of variables to consider, especially since it can produce a list of conducive solutions in a proficient and expeditious manner (Ruiz 2022).

In reduced form the process of the genetic algorithm functions by creating an initial population, calculating the fitness of that population, then repeating the steps for successive generations until we find an optimal fitness (Auctux 2021). This process improves the results of the genetic within each execution phase (Auctux 2021, Ruiz 2022).

J.  Different Approach  
  
If this project were to be an industry facing solution to scale for over 4000 packages or more, there are in fact a few changes I would make. The first is that I would add functions for the user to interact with where they could insert packages into the HashMap or delete as needed if possible. My observations with big companies such as FedEx or Amazon are not that they merely load 4000 entries of data at a time, but rather that any person who needs their services can submit a shipment order from wherever they are. Therefore, with an add button, that could help scale this project beyond its current capacity, and likewise for the delete, so users could undo unintentional orders. I began working on an add function at the end of the main.py file, and while the code is unfinished, it can be considered as pseudocode that showcases some of thoughts towards how I thought at present I could try to add packages. I did attempt to write an added package to file, and upon succeeding found I had overwritten the file. Luckily, I had a backup file on GitHub that had all the former csv rows to undo my mistake, but that mistake further solidified the idea that a database may be needed. The database could be a MongoDB or SQL database, but I would most likely use a SQL database for value binding in queries and search.  
Further, to increase the number of orders I would try to dispatch more trucks. I would also see if there’s an API, I could pull from to continue the adjacency matrix for more locations or see if another geolocation API could remedy the goal of working with shortest distances.  
Given a chance to do this again, I would also do more research into how to implement a Tkinter Gui with this project. I nearly completed it but I got stuck on how to display more rows into one of my Tree views. There was a neat project I was following on the Codemy.com channel of youtube that linked TreeViews to a SQL database, and I would have loved to emulate aspects of that project into the final user interface for this program.

## K.  Data Structure Justification

1.  Requirement Fulfillment of the Chaining HashMap  
  
To verify all attributes of the packages and statuses that pertain to them, the user can select option eight in the user interface and verify the accuracy of the information as they please by selecting option eight after selecting any of the options from option four to option seven. Selecting options two or three after any of the options from four to seven also demonstrates how quickly the hash map can retrieve current information on the statuses of all and any package of choosing.  
A user could also check the intended results of the program by selecting option one “Check Full Delivery Cycle”. Option one models the intended use of the program by executing the call to each option of option four to option seven once, which represents each truck traveling both routes to deliver all packages according to their delivery requirements in timely fashion. The “Check Full Delivery Cycle” also calls option eight to report that the truck’s total mileage is 124.7 which is still beneath the 140 miles requirement.

a.  Efficiency  
The lookup function takes a package id and searches for the package object within one bucket’s list of values in the Chaining HashMap. Since the lookup function corresponds to the HashMap it is affected by the processing of the HashMap data structure. The HashMap data structure processes operations in O(1) time, therefore theoretically it should be that even as packages increase the time complexity would stay the same. However, as the number of values increases per index of the Chaining HashMap, the space-complexity increases therefore increasing the time complexity on the operation of number of searches since the Chaining HashMap will identify one bucket on which to perform a sequential search for a value in a list of values analogous to a sequential search (Ruane 2011). Also, as space complexity increases, there is a possibility for a greater number of collisions (Seppänen 2012).

b.  Overheard  
In a standard linear sort, there is a parity of searching one value per index, and traversing each index until the value is found. In theory, if the list was one item long it retrieve in O(1) time, but if the item was at position “n” which represented being the end of a very long list, it would take “n” searches to find that item. The pattern for Linear sort seems to always be relative in this way, meaning any “n” number of indices relates to the “n” number of searches that will be conducted to find an item.  
The Chaining HashMap is more efficient in that per fewer indices, more values can be stored. In this instance there were 40 package objects, but 4 objects stored per index. This means that when a search is conducted one bucket found and searched up to four times, rather than searching one list up to forty times for the same value. With fewer packages of course space usage decreases and the search of values per bucket decreases. Respectively, as the space usage increases with a greater number of packages per bucket the length of the search on values per bucket can also increase, steadily approaching O(n) times per search if in theory massive numbers of packages were being stored to this data structure.

#### c.  Implications

Each truck object has two lists of packages ids per route. Each list of package ids referred to as either pkg\_load or pkg\_load\_r2 goes through an iteration of calling the lookup function to search for the package object that corresponds to its own package id before appending that package id to one of the inventory\_lists. Iterating through the inventory list per route ensures that all package objects are delivered in accordance with the nearest neighbor logic executed below. Following these processes, if there were a greater number of trucks (with drivers) there could be fewer packages to deliver in total per truck, thereby eliminating the number of routes per truck (with drivers) and decreasing look-up time as the distribution function would only need to be called once for this delivery setup. If the capacity of packages increased per truck, and the number of trucks with drivers increased, that could increase the look-up time per search while again decreasing the number of times a distribution function would need to be called. Each time the distribution function is called it executes an O(n2) time complexity per number of operations such as a search in this instance. Increasing the number of trucks (with drivers) therefore could reduce the number of calls to between similar functions, potentially enable more efficient coding with more conditional logic, and in the long-term, decrease look-up time. If there are more trucks without drivers, the bottleneck may remain essentially the same.

I do not think the lookup function itself would be impacted directly by a greater number of cities, other than the fact that there would be more packages in theory sent to those cities. The greater number of packages being sent to more cities could increase the processing for nearest neighbor algorithm in comparing the address index for each of those cities to calculate the distance of the next closest point, thereby increasing the mileage report and processing time. It should also be noted that more important than the number of cities is the location of those cities/ addresses. If in theory there were several small cities close to each other, and they were all a part of an existing route, per the nearest neighbor algorithm, there may be little difference if any for them to be added, and processing time would stay constant. In ordering the packages, I often experimented with this logic, and found I could move packages to different order by this principle while keeping the mileage low while also maintaining an intended route. However, with a reduction of cities and counterintuitive structuring for package delivery mileage could increase, and delivery times could increase even while package lookup time remained constant (or decreased relative to the number of packages in the HashMap). The significant variable regarding lookup time is really the number of packages, so if the number of cities is increasing the number of packages, or decreasing the number of packages, the number of packages would affect the lookup time accordingly.

An interesting feature to implement for this question would be setting up one HashMap per route or pair of routes as was done in this situation, per city. In the current execution of this algorithm, two rounds of delivery are required per route. With more trucks (manned by drivers), this model could be implemented across many more cities or pairs of cities, while keeping mileage low or relative to the intended routes. In creating more Hash Maps relative to the routes, as mentioned above, more packages relative to the routes could be delivered without affecting mileage, and more deliveries could be made across additional cities. The increasing number of packages would be relative to the city or cities in question since a HashMap could be dedicated to specific routes. There would be more truck to divide the load of those packages, thereby decreasing lookup time or theoretically keeping constant with O(1) time complexity.

### 2.  Alternative Data Structures

Two different data structures that could have still met the needs of this assignment are the dictionary data structure and the list data structure. Both data structures can store at least one object per index for later retrieval in the nearest neighbor algorithm.

#### a.  Data Structure Difference

        In this scenario, another option could have been to implement a dictionary. A dictionary I have discovered is a form of HashMap in python, as both the dictionary and the HashMap derive from the Map class and implement key value stores (r/learnpython 2021). The main difference between the dictionary and HashMap is that the HashMap relies upon the indexing of the bucket array value, whereas dictionaries tend to link string values. Extracting dictionary object values per key, or list of values per object from the csv file, I found was a little more difficult with using the dictionary rather than the HashMap since the dictionary depends on a one-to-one parity of key to value. This led to the observation that in this implementation of the HashMap being the Chaining HashMap, the greater distinction between the dictionary and the Chaining HashMap it that the dictionary stores one value per key, while the Chaining HashMap stores a list of values per key. As both data structures derive from the Map class though, they both share O (1) time complexity in access time therefore, either data structure would suffice for the problem (r/learnpython 2021, Samuel Klatchko 2010).  
A list also could have been used in this scenario. The list data structure shares both the benefit and the bane of storing one item per index. The benefit of this data structure is that for n elements it is very easy to retrieve the length view the len() function or the size with the size() function. However, as discussed at length above, as lists increase, per n items a search is conducted n times, thereby increasing length of the search directly proportional to the number of items at an O(n) time complexity, which is worse than the O(1) time complexity produced by searches using Map data structures.

## L.  REFERENCES

Auctux. (2021, October 25). *Solve the Traveling salesman problem (Genetic Algorithm, Ant Colony Optimization)*. YouTube. <https://youtu.be/Sk9QQUGMdY8?si=gKrksv2UquDhawNc>

Finer, J. (2018, December 19). *How to write beautiful Python code with PEP 8*. Realpython.com; Real Python. <https://realpython.com/python-pep8/>

Madhushree. (2012). Point Pattern and Nearest Neighbour Analysis. *Point Pattern and Nearest Neighbour Analysis*, 12.<https://epgp.inflibnet.ac.in/epgpdata/uploads/epgp_content/S000017GE/P001787/M031066/ET/1527502623NNA_text(Final(1.pdf>

Reducible. (2022, July 26). *The Traveling Salesman Problem: When good enough beats perfect*. YouTube. <https://youtu.be/GiDsjIBOVoA?si=3lytf46Zt9YbTCIX&t=1087>

r/learnpython. (2021). *Hash Maps vs. Dictionaries*. Reddit.com. <https://www.reddit.com/r/learnpython/comments/s8oz0f/hash_maps_vs_dictionaries/>

Ruane, P. (2011, August 3). *What are the disadvantages to hashmaps?* Stack Overflow. <https://stackoverflow.com/questions/6924852/what-are-the-disadvantages-to-hashmaps>

Ruiz, G. (2022, May 29). *The travelling salesman problem using genetic algorithm*. YouTube. <https://youtu.be/1Mt0HB2eAdY?si=qqiwurxA-1W9xVFQ>

Samuel Klatchko, R. (2010, January 14). *What is the true difference between a dictionary and a hash table?* Stack Overflow. <https://stackoverflow.com/questions/2061222/what-is-the-true-difference-between-a-dictionary-and-a-hash-table>

Saxena, A. (2020, December 6). *Hash map in python*. GeeksforGeeks. <https://www.geeksforgeeks.org/hash-map-in-python/>

Seppänen, J. K. (2012, March 24). *Hash table vs. Linear list*. Stack Overflow. <https://stackoverflow.com/questions/9849633/hash-table-vs-linear-list>

Solving Optimization Problems [@SolvingOptimizationProblems]. (2021, January 17). *Python code of the 2-Opt Algorithm for solving the travelling salesman problems (TSP)*. Youtube. <https://www.youtube.com/watch?v=ayIsRZAGyi4>

Track Record/Union Pacific. (2023, January 3). *The Pros & Cons of Truck Shipping: Cost, Speed, Capacity and More*. [up.com](http://up.com). <https://www.up.com/customers/track-record/tr081319-truck-pros-cons.htm#:~:text=Trucks%20travel%20at%20an%20average,and%20cost%2Deffective%20shipping%20solution>

Xristica, A. (2018, July 18). *10 Reasons for loving Nearest Neighbors algorithm | Quantdare*. Quantdare.com. view-source:<https://quantdare.com/10-reasons-for-loving-nearest-neighbors-algorithm/>