**Core Algorithm Overview**

**Stated Problem**

The Western Governors University Parcel Service (WGUPS) is having trouble consistently delivering packages in the desired time for their Daily Local Deliveries (DLD) within Salt Lake City, UT. The purpose of this project is to create an algorithm that determines an efficient route and delivery distribution system that takes into consideration special criteria and delivery requirements. The Salt Lake City DLD operates with three trucks, two drivers, and on average has 40 packages to deliver daily.

**Assumptions**

* Each truck can carry a maximum of 16 packages.
* The trucks travel at an average speed of 18 mph and have no need to refuel.
* There are no collisions.
* Three trucks and two drivers are available for deliveries. Each driver stays with the same truck as long as that truck is in service.
* Drivers cannot leave the hub before 8:00 a.m.
* Drivers can return to the hub to reload packages if needed.
* The delivery and loading times are instantaneous, i.e., no time passes while at a delivery or when moving packages to a truck at the hub (that time is factored into the calculation of the average speed of the trucks).
* There is up to one special note associated with a package.
* The delivery address for package #9, *Third District Juvenile Court,* is wrong and will be corrected at 10:20 a.m. The correct address is: 410 S State St., Salt Lake City, UT 84111.
* The distance provided in the WGUPS Distance Table are equal regardless of the direction travelled.
* The day ends when all 40 packages have been delivered.

**Package Restrictions**

Each package is allowed one special requirement, the following are the restrictions that were present in review of the package data.

* Certain packages were delayed on a flight and would not be arriving to the Hub until after the 8:00 a.m. truck departure time.
* Certain packages were required to be on a specified truck.
* A package could request to be delivered with other packages.
* A package had a specified delivery deadline.

**Algorithm Identification**

A: Algorithm identification, B2: development environment

Nearest Neighbor is a type of greedy algorithm that is useful in determining a path to the next closest location from the current location. Because of its efficiency in both space and time complexity with the amount of data that was being processed in this project, in addition to its ease of implementation, I found the algorithm to be an acceptable solution for WGUPS’s delivery route problem. My program implementation of this algorithm meets all the requirements by having a total milage under 140 miles after delivering all 40 packages with the use of all three trucks and two drivers.

To implement this algorithm, I used the high-level programming language Python 3.9.5 with the development environment Visual Studio Code on Windows 10. The hardware that was used to run this program was an HP Envy laptop with a Ryzen 7 microprocessor with 16 GB of RAM. Bandwidth was not a factor for this project because the program was ran locally and I was not using a network to transfer any data.

**Algorithm Overview**

B1: Logic comments

My algorithm works as follows:

1. In load\_data.py
   1. I create an array of Truck objects and load each truck with an array of packages.

**Def load\_all\_trucks( # pass in the hash table) :**

**# Create three truck objects**

**# create an array of package id’s for each truck object**

**# ‘load’ each truck by appending each package object (function called that takes package id and appends package object into package array)**

**# create an array of all three trucks with package data**

**# returns the array of trucks**

1. In main.py
   1. I loop through the array of Trucks and for each truck I set update the delivery status for each package and capture the start time for the route.
   2. I call NN\_shortest\_path() algorithm

**# loop through the array of trucks and for each truck**

**# loop through array of package objects**

**# if the ID is 9 update the address**

**# otherwise update the status’s of each package to ‘en route’**

**# call NN\_shortest\_path(# pass in graph # pass in start vertex (start location), # pass in each truck, # pass in array of vertices (array of all locations.))**

1. In algorithm.py
   1. I create a queue for the vertices (locations) that have not been visited
   2. I create a queue for the packages that have a delivery deadline and append those if on Truck 1.
      1. I only do truck one because this is the only truck were I was violating the delivery deadline.
   3. I compare the addresses of each package to the array of all addresses and append the addresses that match onto the unvisited\_queue.
   4. I then loop until the unvisited\_queue is empty and once it equals zero I send the truck back to the hub.
      1. If it is not empty then this is where I am comparing vertices and determining the closest location.
      2. Once this is determined I remove the address from the unvisited\_queue and start the process over again until the queue is empty.
      3. Once the location has been removed from the unvisited\_queue I update the package delivery status to be delivered and update the distance the truck as traveled.
   5. The process then repeats for each truck.

**Def NN\_shortest\_path(pass in graph, pass in start location vertex, pass in truck object, pass in array of vertices (array of all locations))**

**# get truck’s leave time**

**# create queue for unvisited vertices (unvisited\_queue)**

**#create queue for packages with delivery deadlines (priority\_queue)**

**# loop through the array of package objects**

**# loop through array of vertices**

**# if running for Truck 1**

**# append packages with delivery deadlines onto priority queue  
# otherwise append packages without deliver deadlines to unvisited queue**

**# otherwise**

**# compare the address of the package being added and each delivery vertex. If equal then add to unvisited\_queue**

**# deduplicate addresses by changing the unvisited\_queue to a set (holds unique values only) and then back to an array (iterable).**

**# Run loop until unvisited\_queue is empty**

**# if queue is empty bring truck back to hub**

**# otherwise**

**# loop through delivery locations (edge list). Edge list is a dictionary with key being a tuple (from\_vertex, to\_vertex) with the value being the miles between vertices:**

**# compare the distance to all other delivery options distance and keep the shortest values**

**# remove the vertex with the shortest distance from unvisited\_queue and update the current vertex, delivery status of package, and miles traveled by truck.**

**# once unvisited\_queue is empty, update time value and record completed time**

**# once all trucks are completed and at hub, update total miles travelled by all trucks.**

**Algorithm Runtime**B3: Space-time and Big-O

I have fully documented my code, including the Big O notation but I am listing them below for convenience.

**Algorithm.py**

|  |  |  |
| --- | --- | --- |
| Function | Space complexity | Time complexity |
| NN\_shortest\_path | O(N^2) | O(N^2) |
| Calculate\_distance | O(N^2) | O(N^2) |

**Load\_data.py**

|  |  |  |
| --- | --- | --- |
| Function | Space complexity | Time complexity |
| Load\_package\_data | O(N) | O(N) |
| Load\_distance\_data | O(N^2) | O(N^2) |
| Load\_all\_trucks | O(N) | O(N) |
| Load\_truck | O(N) | O(N) |

**Hash\_table.py**

My hash table includes both an insertion method and a look up method, the space and time complexity are included within the program but are listed below for continence.

|  |  |  |
| --- | --- | --- |
| Function | Space complexity | Time complexity |
| Get\_hash\_index | O(N) | O(N) |
| Insert | O(N) | O(N) |
| Look\_up | O(N) | O(N) |
| Remove | O(N) | O(N) |
| Print | O(N) | O(N) |

The program has a time complexity of O(N^2).

**Algorithm Scalability and Adaptability**

B4: scalability and adaptability

The problem that WGUPS is facing is similar to the popular and classic problem known as the Traveling Salesman Problem. The traveling salesman problem looks at a given list of cities and the known distances between pairs of cities and then asks the question of, what is the shortest route that one can take to visit each city exactly once and return to the starting city? I have discovered that this problem can be difficult to solve and optimize, especially when restrictions are also involved. The implementation of the Nearest Neighbors algorithm in attempt to solve this Traveling Salesman problem for WGUPS resulted in a run time-complexity of O(N^2). This means that it will grow exponentially with the number of packages involved. With this, I do not think that this algorithm is the most efficient or optimal choice for a problem that needs to linearly scale, but it will adapt and continue to run with the addition of more packages, trucks, and destinations.

**Algorithm Strengths**

I1: strengths of chosen algorithm

One strength of the Nearest Neighbors algorithm is that it is easy to understand. I implemented the algorithm after thinking about the problem from a logical perspective of how a driver would most likely drive a delivery route without any instructions. From the starting location, they would commonly choose to deliver the package that is closest to them and continue to apply that logic until all packages were delivered. Nearest Neighbor programmatically does just that as it chooses the next location based on which of the options is closest to the current location and continues until you have arrived back at the starting location.

Another strength of this algorithm is that it is relatively simple from a programmable point of view. Unlike a comparable algorithm such as Dijkstra’s Shortest Path algorithm which has many working relationships regarding a graph data structure, the Nearest Neighbor algorithm is mostly a recursive loop that compares all delivery distance options to each other and returns the closest choice. This can be achieved with basic understanding of loops and recursion.

**Algorithm alternatives**

I3: other possible algorithms

**Dijkstra’s shortest path algorithm**

Dijkstra’s shortest path algorithm, a comparable path finding algorithm, determines the shortest path from a start vertex to a destination vertex in a graph data structure. Although similar to the implementation of Nearest Neighbors algorithm, Dijkstra’s is best suited when there is a definitive direction one needs to take to get from a start location to an end location. To achieve this, the algorithm compares the distance of the path to each vertex along the way to the destination vertex. To determine the shortest path, it checks each option to every vertex in the graph and stores the value of the shortest distance to that vertex. Once all vertices have been visited and the shortest distance to each has been found, one can generate the shortest path by backtracking the path from the destination vertex back to the start vertex.

Where this differs from the Nearest Neighbor algorithm is that the implementation does not take into consideration every possible paths. Although Dijkstra’s algorithm could result in a more optimized and shorter path, the implementation Dijkstra’s is very difficult due to the nature of the data used in the WGUPS problem. The difficulty lies in that all vertices are adjacent to each other, creating a bidirectional graph. This creates an issue in where there are N factorial routes possible, and the implementation would be closer to a brute force method where the programmer would need to check all possible solutions.

**Greedy Best-First-Search algorithm**

The Greedy Best-First-Search algorithm works in a similar way to Dijkstra’s shortest path algorithm but implements heuristics to find how far from the destination point any vertex is. Starting by selecting the vertex closest to the goal and uses its heuristics to guide its way towards the goal quickly (Patel). This approach can generate a path faster in runtime-complexity compared to Dijkstra’s and Nearest Neighbors but not always the most optimal in terms of shortest distance.

Once again, this algorithm is far more difficult to implement for the WGUPS problem due to the starting point and ending point are both the same and all vertices are adjacent to each other. Since there is a limited and undetermined gain in distance optimization, the additional complexity was not worth it over choosing the Nearest Neighbor approach.

**Efficiency and Maintainability**

B5: software efficiency and maintainability

To keep the code organized and maintainable, the program was created using an object-oriented approach. This programming paradigm allows the programmer to create reusable classes to generate unique objects holding similar data. This was utilized for the package data, the truck implementation, as well as storing all of the delivery locations (vertices and edge data). Each class and its components, as well as all functions have been clearly documented with comments throughout the files and are easy to understand. Other developers, including myself, can look at this program now or in the future and ascertain the working relationships involved in the program.

**Data Structures**

B6: self-adjusting data structures, D: Data structure, D1: explanation of data structures, E: hash table, K1C: Implications

**Hash Table**

The hash table that I created does not utilize any third-party libraries nor is it implemented using a dictionary. I created an empty array large enough to fit all package objects, which can scale if needed. As each package object was added to the array, a hash function was utilized to find the correct index from the formula: *key modulo T* (Key equaling package ID number and T equaling size of array). In the case of when a collision occurs at the hash index, the chaining technique was implemented and an iterable list was created at each index to hold multiple package objects.

**Weakness of hash tables**

K1B: overhead

A weakness of the hash table is that it can become slow if a large number of collisions happen. As mentioned above, the chaining technique was implemented to handle collisions by creating an iterable list at each index. When utilizing a hash table to find the desired package, it will perform a linear search by taking in the key value and run that value through the hash function to obtain the desired hash index. At that point, it either finds the correct key or reaches the end of the list, returning an invalid ID message to the user. With a relatively small amount of package data in this case, it is not a problem, however it can become one when working at the scale that most delivery companies work with. A programmer can get around this by re-hashing or re-indexing as the data grows, but each time that is needed it is O(N) runtime-complexity and that grows linearly.

**Strengths of hash tables**

A hash table is both efficient and can be fast when implemented correctly. If a perfect hash can be achieved the runtime-complexity is O(1). However, a hash table most commonly runs at O(N) and the runtime-complexity remains the same even as the data that is being stored grows.

**Dictionary Data Structure**

Other than the hash table, I used two dictionaries to hold data to create the graph. One dictionary was utilized as an adjacency list, where the key is a vertex (a delivery destination) and the value was an array of vertices it was adjacent to. Since the destination data required that each location be adjacent to all other locations, each value array held every other location, including itself. The other dictionary held the miles between each location, commonly referred to as the edge list. This list was composed of tuples as the key, each being the start vertex and end vertex from one location to another, and the value for each tuple was the miles it took to travel between them (bidirectional).

**Strengths of a Dictionary Data Structure**

The dictionary data structure proved to be useful in the WGUPS project as it was common to have relational data between one delivery location to another (or many others). One of the main strengths of the dictionary is the ease of implementation, scalability, and generally good runtime-complexity.

**Weaknesses of a Dictionary Data Structure**

The major weakness of a dictionary is that the data is unordered. This makes the dictionary impossible to access by index or order in which the data was added. Another weakness worth noting is as the data grows the memory space taken up by the dictionary will grow, as well as the look up time for each key/value pair. For example, in the dictionary that holds the adjacency list, as more destinations are added, the array associated with the value will grow linearly, causing the recursive comparisons to grow as well.

**Other data structures**

K2: other data structures, K2A: Data structure differences

An alternative to using a hash table to hold the package data would be to use a dictionary. Since the package data requires the need of a key/value pair, a dictionary would be a suitable replacement. A hash table and a dictionary are very similar in functionality and how one works with the data stored within the structure.

Rather than using an edge vertex graph to hold the addresses (vertices) and the associated miles (edges), a two-dimensional (2D) array could have been implemented to hold the distance data. A 2D array matrix acts similarly to a table where there are rows and columns, and the data can be accessed via position in the nested arrays.

**Screen Shots**G1: first status check, G2: second status check, G3: third status check, H: screenshots of code execution

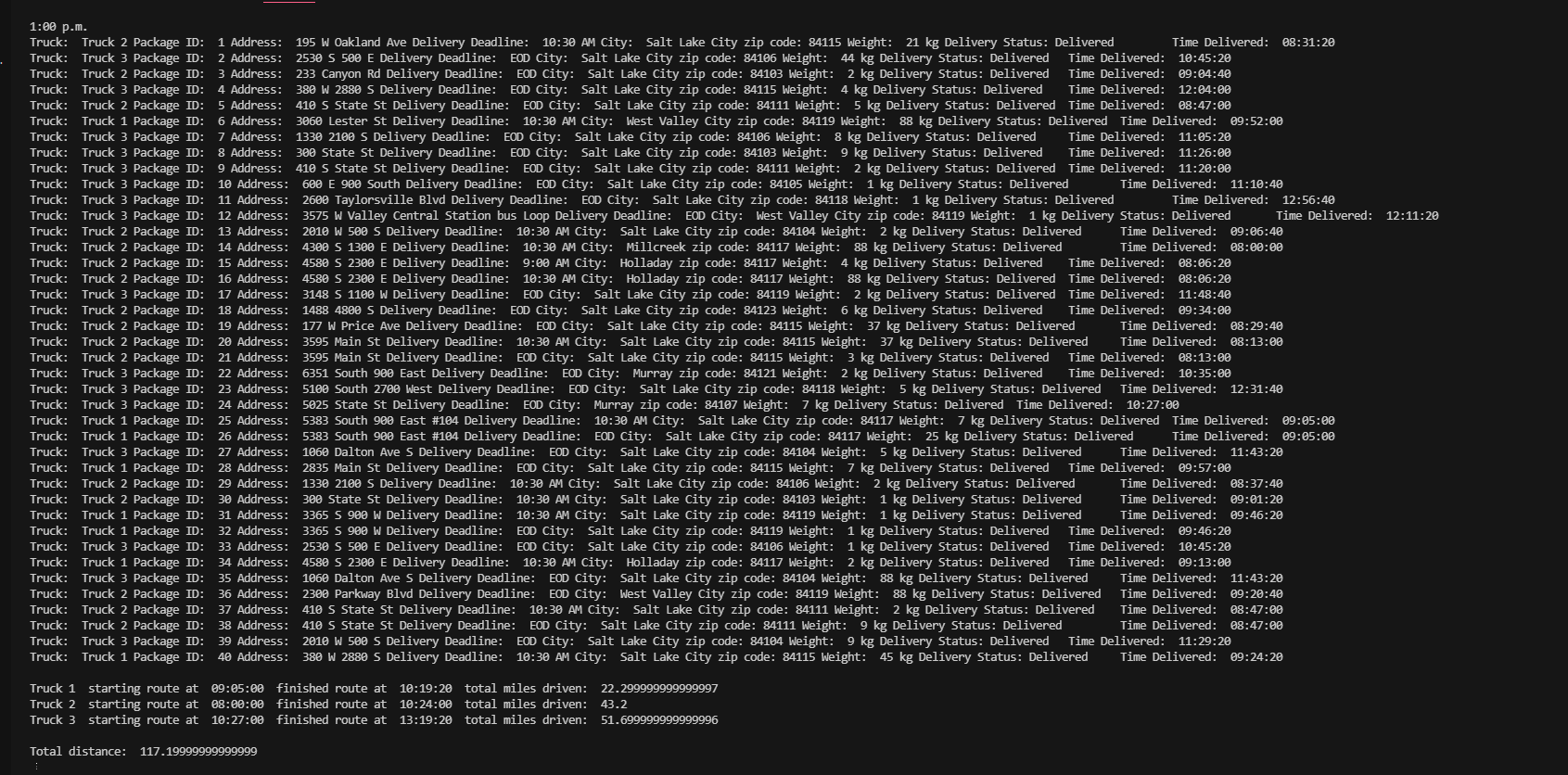
**G1: Screenshot of all packages for each truck at 8:35 a.m.**



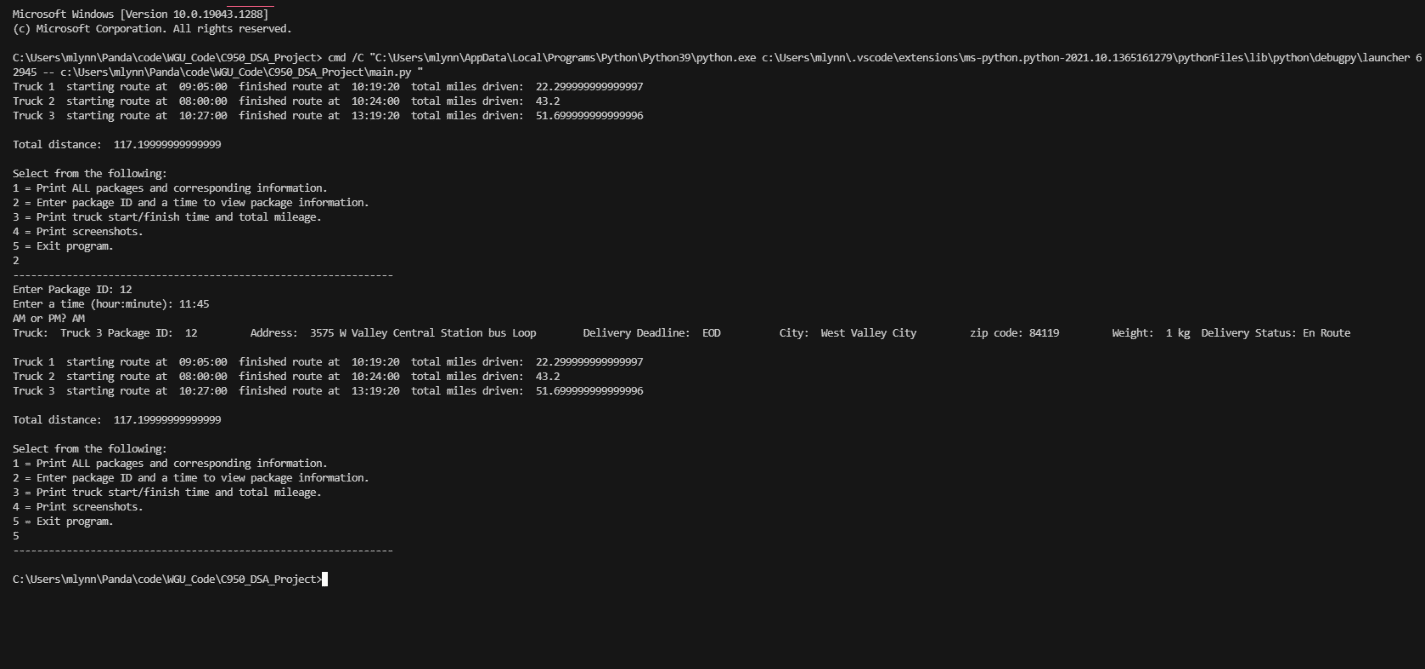
**G2: Screenshot of all packages for each truck at 9:40 a.m.**



**G3: Screenshot of all packages for each truck at 1:00 p.m.**



**H: Screenshot of code execution**



**Reflections**  
J: different approach

During the development of this program, I decided to “manually” load each package onto each of the trucks. This means that the packages were hardcoded to specific trucks utilizing arrays rather than programmatically solving which package goes on which truck. If I were to re-attempt this project, I would develop a method for sorting the package data to dynamically load the trucks with the packages, which would allow the program more flexibility such as accepting last minute packages and the ability to optimize the total driven distance through different packages being loaded on different trucks. Even though the use of the Nearest Neighbor algorithm worked well, I would also attempt a different algorithm, such as Dijkstra’s algorithm, to further optimize the truck milage and to have better scalability without the increase in runtime-complexity.

**Sources:**

Patel, Amit. "Introduction To A\*". Theory.Stanford.Edu, http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.