**C950 Performance Assessment Overview**

Kyle Green ID: #001363834

1. The objective of this performance assessment task is to find an efficient route for multiple package delivery trucks, a variation of the “traveling salesman problem.” I chose to use the “nearest neighbor” algorithm as my solution to this challenge as it is able to generate a reasonably efficient route without taking an unacceptable amount of computation time. This algorithm simply iterates through a list of vertices in a graph data structure, and stores the smallest edge weight, i.e., the nearest neighbor. The algorithm runs repeatedly until all packages have been delivered.

1,3. Program Overview, Pseudocode and Space-Time Complexity

The program begins by loading all package data from the provided csv file and inserts it into a hash table. A hash table is used because it allows for insertion and search operations in constant – O (1) time complexity. This enables the insert and search methods to scale to an arbitrarily high number of packages. To load package data, I opened the file with python’s built-in csv library and iterated through the file. Each item in each line was then used to create a package object. This operation scales linearly with the number of packages in the file, leading to a time complexity of 0(n). The space complexity also scales linearly with the number of packages.

**Hash Table**

**Insert (package):**

index = hash (package id) % table length

table at index = package

**Search (package id):**

index = hash (package id) % table length

return table at index

**Load Package Data**

for line in csv file:

package = id, address, deadline

insert package into hash table

The next section of the program creates a graph data structure and loads the address data from another csv file. I chose a weighted graph structure for the address data and used the distance information as the weights for the edges. This is somewhat more complicated than the package data because each address has a connection to every other address. This leads to quadratic complexity of O (n^2) because the program must iterate through each address and then iterate through each connection for that address.

**Graph**

add vertex (address):

append address to graph

add weighted edge:

Graph edge weights [address, other address] = distance

**Load Address Data**

for line in csv file

address = label, street address

add vertex (address)

for other addresses in line

add weighted edge (address, other address, distance)

From here, the program can “load” the packages onto the trucks. This is a relatively simple operation that iterates through the packages until all have been delivered. There are some conditional statements used in each iteration to check special notes on the package, but these do not significantly add to the time complexity. This operation occurs in O (n) linear time. The operation repeats until all packages have been loaded, so n is multiplied by the number of repetitions, but this is a constant that does not alter the time complexity.

**Load Trucks**

while any package is still at hub:

for package in all packages:

load truck (package)

The final aspect of the program is to create the delivery route. This is done using the nearest neighbor algorithm on the list of packages loaded onto the truck. The truck finds the edge weights between its current address and the delivery address of all packages. The lowest edge weight (i.e., the shortest distance) is chosen as the next delivery stop. For each address, this is a linear time operation, O (n). It must be repeated for each package in the list, however, rendering an overall time complexity of O (n^2) quadratic time.

**Nearest Neighbor**

while truck is not empty:

shortest distance = 1000

for package in packages on truck:

if edge weight (truck location, package address) is less than shortest distance:

shortest distance = edge weight (truck location, package address)

nearest neighbor = package address

move to nearest neighbor

The nearest neighbor algorithm would dominate the computational requirements of the program. The other elements are fast enough that they don’t need to be considered for the overall complexity of the program. Combining all the segments of the program leads to a quadratic O (n^2) time complexity.

B.2. I developed this application using Python version 3.9.1 and the Pycharm Community 2020.3 edition IDE. I used the csv methods from the python standard library and did not employ any external libraries at all.

B.4. My solution to this version of the traveling salesman problem is able to scale to much larger number of packages. The first scalable aspect of the program is the use of a hash table to store packages. Because a hash table creates an index for each package by “hashing” a unique identifier, search and insert methods are constant time operations. This means they are unaffected by the number of packages in the table. My implementation of the nearest neighbor algorithm also enables the program to scale very well. By keeping the route creation to quadratic time, the algorithm can scale reasonably well to an increasing number of packages, certainly better than a brute force method. This scaling has its limits, though. Quadratic time can become unacceptable at large numbers of packages and different algorithms would be needed to yield an answer in a reasonable time frame.

B.5. My software is efficient and easy to maintain because I used the object-oriented programming model and followed clean code practices. My code is contained in clearly named classes and instances of those classes. Additionally, all methods have clear functionality and limited scope. I have also provided comments that make it clear to anyone reading the code what the flow of the program is, and what each segment is doing.

B.6. Hash tables are very useful data structures when search and insertion time is important. They are able to scale to an arbitrarily high number of items and maintain the celerity of these operations. The biggest drawback is that it is somewhat difficult to search for items by anything other than the unique identifier. If one wants to search by special notes of each package, for example, the program will have to iterate through all packages and return the items with matching values. This will begin to take an unacceptably long amount of time as the number of packages increases.

Graphs are another useful data structure for simulating geographic locations and distances. The use of an edge weight list allows for easy lookup of distances between vertices for use in the nearest neighbor algorithm. One issue with weighted graphs is that the edge weight list could become exceedingly large as the number of vertices increases, but it may still prove to be the most viable structure to use.

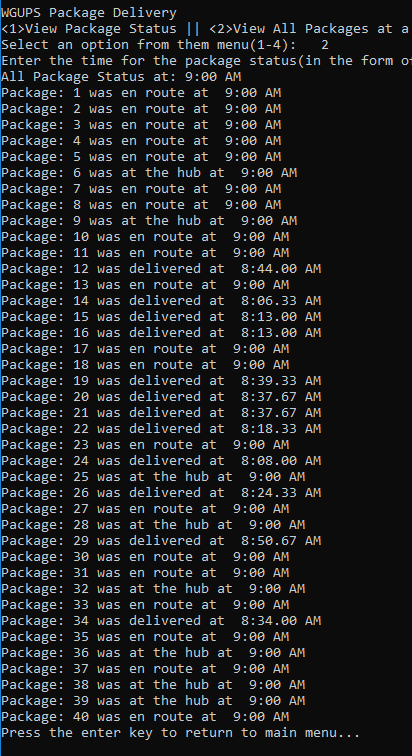
D.1. The nearest neighbor algorithm interacts with both the hash table containing all package data and the graph containing all addresses and distances. The code begins by looking up the package in the hash table and returning the address of said package. This address, combined with the current location address, are used to create a key for the edge weights list in the graph structure. This key will return the distance between each address in the package list and the algorithm stores the lowest one.

E. The hash table class is in the HashTable.py file in the DataStructures directory of my project.

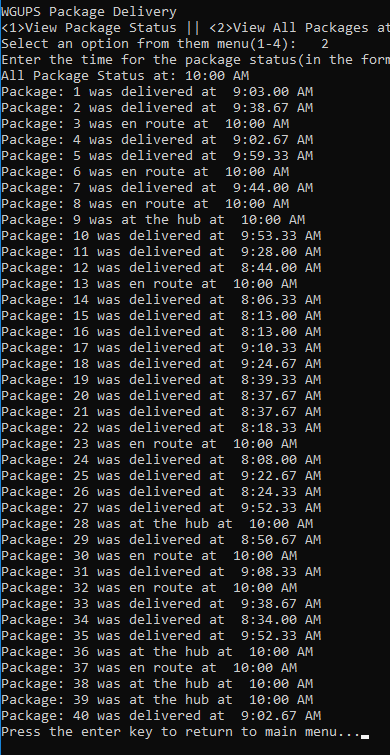
F. There are two methods in the hash table class that provide lookup functionality. The “search” method takes a package object and hashes its “package\_id” data member to find its index location. The “search\_by\_id” method takes either a string or an integer as the id argument and hashes that id to find the index of the package.

G.1.

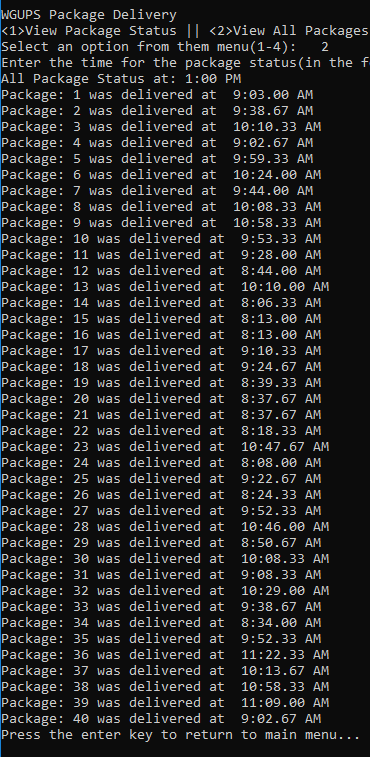
**All Packages at a time between 8:35 AM and 9:25 AM:**



**G.2. All Packages at a time between 9:35 AM and 10:25 AM:**

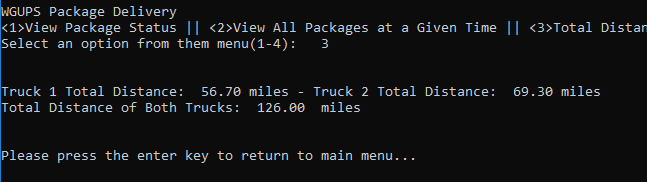


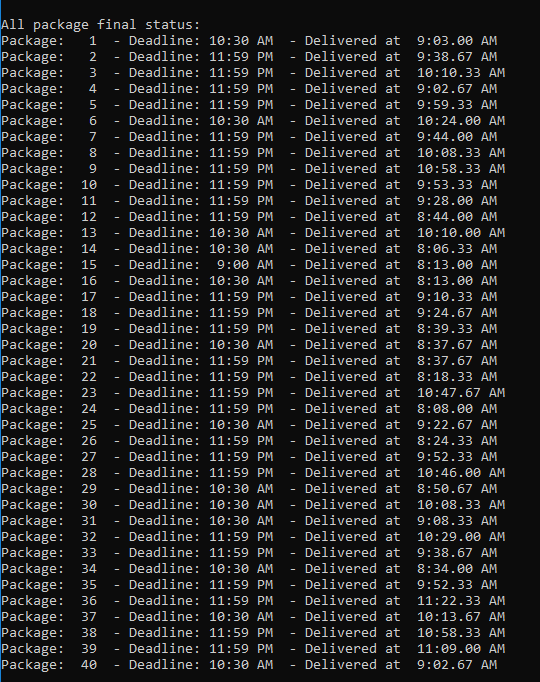
**G.3. All Packages at a time between 12:03 PM and 1:12 PM:**



**H.**

**Total Mileage of All Trucks:**



**Final Delivery Status:**

**I.1.** The nearest neighbor (NN) algorithm is not a perfect solution to the traveling salesman problem. Some research has shown that the route it generates can be 15% longer than the optimal solution. This doesn’t mean that it isn’t useful, however. NN is able to create a solution in sub quadratic time, which is significantly faster than other, more accurate solutions. Furthermore, NN is relatively easy to implement, requiring only one for loop and a comparison to find the smallest distance. For these reasons, I believe the nearest neighbor algorithm was the best choice for my program.

**I.2.** My implementation of the nearest neighbor algorithm was able to meet all requirements of this task, as evidenced in section H of this overview. All packages were delivered by their deadlines and the total combined distance driven by the trucks was only 126 miles.

**I.3.** There are some alternatives to the nearest neighbor algorithm for solving this problem. The simplest would be a “brute-force search”, where the program tries all permutations of possible routes and picks the one with the shortest distance. Although this is easy to implement in code, it would have extremely long computation times for even relatively small numbers of nodes. Since the number of permutations is a factorial of the number of nodes, this type of algorithm would have a time complexity of O(n!), out of the range of acceptability for this application (Johnson, 1997).

Another alternative algorithm is called Ant Colony Optimization. This solution is modelled after ants and the method they use to find efficient routes in the real world. “Ants” in the code travel along edges in the graph and leave “pheromones” on each edge (Blum, 2005). The pheromones are subtracted from the edge weights, and other ants will choose the edges with the most pheromones. This leads to near optimal solutions after many ants have traveled the graph. Although Ant Colony Optimization can produce near optimal results in a reasonable amount of time, the ant and pheromone systems are rather complicated to implement. For the scale of this application, I determined nearest neighbor was a better solution.

**J.** If I were todo this project again, I would try to implement the nearest neighbor algorithm to find the best packages before loading the trucks. While iterating through the packages, I would compare address distances with those packages already on the truck. I believe this would create even more efficient routes than simply finding the nearest neighbor of the packages that just happened to be loaded on to the truck. The drawback of this idea is that it would make the loading algorithm more complicated and increase its time complexity, so I’m not sure it would be an overall superior solution.

K.1.

a. The hash table that I used for this program is able to scale very well with the number of items that it contains. Since the index of each item is found using a hashing algorithm on a unique identifier, the search and insertion functions are constant time operations. Thus, the total number of packages do not affect the time needed to insert or search for a package.

b. The size of the hash table will increase linearly with the number of packages it contains. A package object will be created for each package and then inserted into the hash table. Each package object is not overly large, however, so modern computers should be able to store large amounts of packages.

c. The number of trucks should not affect the lookup time of the hash table at all. With two trucks, the hash table is accessed to load packages on to the truck and to create the delivery route multiple times. With additional trucks, these operations will be spread out over those trucks, but it would not require additional calculations.

If more cities, or addresses were added, than that would affect the size of the graph object, primarily the edge weights list. This list contains connections between all addresses and so every new address will create new entries in the edge weights for its connections to all other addresses. This means its space requirement is quadratic in nature. The time complexity would not be affected, though, the edge weights are indexed by combined addresses as keys and do not need to be iterated through.

K.2. One data structure that could meet the scenario’s requirement is a simple list/array. This would be relatively easy to implement as most programming language have arrays as part of their standard library. The big disadvantage of arrays would be their lookup time. Unlike with hash tables, the program would need to iterate over the array and compare each element to the search criteria. This would lead to linear time complexity of this operation. A linked list could also potentially serve as the data structure to store package data. This suffers the same issue that the array does, though. One would need to iterate through the list to do any kind of search operation.

# Bibliography

Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 354-368.

Johnson, D. S. (1997). The Traveling Salesman Problem: A Case Study in Local Optimization. In D. S. Johnson, *Local Search in Combinatorial Optimization* (pp. 215-310). London: John Wiley and.