C950 grading rubric

A:

The algorithm I used is the “Greedy Algorithm”.

B1:

set current location to the variable passed as such

While the number of packages is > 1

For each package available

get the distance between the current location and the package being checked

If it’s the smallest distance checked so far, save both the distance and the package

retrieve the saved package and set your new current location to it

add it as the next the package to deliver in the list you will return

add the last package to the list as it’s the only package and therefore by definition the shortest distance

lastly return the list of packages that are now in the order you should deliver them

B2:

I am using the PyCharm IDE 2021.2.3 edition with the Python 3.10 Interpreter.

I am running Windows 10 Pro on an Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz.

B3:

The space and time complexity of reading and storing the package data from the package file is O(N) where N is the number of packages as each additional package adds a single row to be read and stored.

The space and time complexity of reading and storing the distance data is O(N2) where N is the number of addresses because each address adds an additional row with length equal to N leading to N rows \* N length.

The space complexity of the getDistance method is static as the variables involved increase but the time complexity is O(N) as it must loop through the list of addresses twice which simplifies form 2N to N. This could have been made in to a O(1) runtime if I had stored the addresses in a hash table.

The space complexity for my chooseGreedyPath method is static as the variables involved increase but the time complexity is O(N2) where N is at most 16 because that is the amount each truck can hold. It is O(N2) because each node must be compared to every other node not yet removed.

The program as a whole has a space complexity of O(N2) where N is the number of addresses and a time complexity of O(N2) Where N is the number of packages. At sample sizes smaller than 16 the time complexity can be O(N3) because the time will be dominated by the chooseGreedyPath running N2 times (N <= 16) and each time it runs it calls the getDistance method N times.

B4:

This solution scales very well as the number of packages increases because the algorithm is only run on 16 packages at a time which keeps the runtime low. The main concern would be the getDistance method whose runtime would increase linearly as the number of addresses increased. If the addresses were stored in a hash table this concern would be negated.

B5:

The code is well commented and organized to allow for easy maintainability. The items being worked with are made in to objects and each major action was made in to its own method which again helps with maintainability. The code is efficient because the runtimes are kept low by only running the time intensive methods on at most 16 packages at a time.

B6 & D:

My self-adjusting data structure is the hash table that I built titled myHashtable. The main strength of this data structure is that its very well suited for the unique ID’s that each package has which means 0 collisions and a consistent O(1) runtime. The second strength is that it doesn’t include extraneous methods or complications because of its simplicity in both how it works and the number of things it can do. Its main weakness is because it’s so simple it would be difficult to use for much other than its designed task of storing packages.

D1:

The hash table that I created has a number of buckets equal to the number of packages and each buck is mapped to a hashed version of the package ID leading to zero collisions. The data is stored so that a package object is the value and the packageID value of that package is the key. So each package can be found by hashing its key at which point all the data stored in that package object can be retrieved.

I1:

The first strength of my chosen algorithm the “Greedy algorithm” is that it’s very good at reaching a close to optimal path for this type (traveling salesman) of problem. The second strength is that I’m only giving it chunks of 16 packages at a time which keeps its runtime relatively low making it a fast and effective solution.

I2:

The algorithm I chose meets all of the requirements. It is a self adjusting algorithm that delivers all of the packages on time, with in their individual constraints, and only going 127.8 miles (including returning to hub after all deliveries) which meets the requirement of below 140.

I3:

Dijkstra’s and Nearest Neighbor algorithm would have worked in this scenario.

I3A:

Dijkstra’s algorithm is very useful for finding a shortest path between 2 points in a quick runtime and would have been a great choice for this problem. The problem with Dijkstra’s algorithm in this problem is that there are so many requirements that have to be juggled and met it would require more work pre-coding to determine which addresses Dijkstra’s algorithm should be used on, my greedy algorithm requires less planning but may have a higher runtime. Similarly, the Nearest Neighbor algorithm also struggles with handling the requirements of all the packages. It may also run in to bad scalability and excessive runtimes if it doesn’t take the problem in chunks like my Greedy algorithm. They all have their strengths and weaknesses, but I like mine because unlike the others it requires little planning and scales great.

J:

If I were to do this project again I would store the addresses table in a hash table with the address as the key and its list of distances as the values. This way my getDistance method could have an O(1) runtime

K1:

My hash table meets all the requirements. It is a self-adjusting data structure with the required methods, all packages are delivered in 127.8 miles (including returning to hub after all deliveries), and the lookup function runs efficiently (O(1) runtime 0 collisions) and return accurate information.

K1A, K1B, & K1C:

As the number of packages increases the runtime remains O(1) and there are 0 collisions assuming unique ID’s are maintained. The space usage scales linearly because with each new package only 1 new bucket needs to be created. The number of trucks and addresses do not affect the runtime or space complexity of the lookup function because it always receives an ID that is hashed which allows the program to immediately check 1 location and return that package.

K2 & K2A:

An array could have been used since the ID’s are all small ascending integers. It would have been easy to maintain an array of 40 elements and check the array index that matches the packageID. This data structure would have been simpler to maintain and use, however it doesn’t have the scalability or versatility of the hash table. A hash set would also be a good choice as each ID is unique and the lookup time is important. The hash set is a similar data structure and would have worked well but would fail when two seemingly identical packages were being saved where as mine would still work.