NICHOLAS HARTMAN

WGU C950 Task

Part A:

File	Func Name	Type	Line	Big O
PackageController.py	load_packages_on_truck	Greedy	152	
Graph.py	dijkstra	Greedy	133	N^3
Truck.py	__plan_route_nearest_neighbor	Nearest neighbor	107	N^5
PackageController.py	Any function that updates or gets		288, 293, 305	Constant Time

Part B:

Pseudocode to plan route by nearest neighbor

```
set current location variable to appropriate starting point
while # of packages delivered < number of packages
    initialize a variable max_miles to largest possible miles
    for package in packages
        get miles from current location to package destination
        if miles < max_miles
            then update max_miles to miles and set package as closest
            package
    set current location to the closest package destination
    add closest package to a list
```

Pseudo code for dijkstra

```
    set start
    Array[] keeping track of distances from node to any other node
    Queue of all nodes in graph
    set of nodes seen already
    while(Queue is not empty)
        get node in queue that has no yet been seen and has smallest distance
            to starting node
        add node to seen and remove from queue
        if the end node id is in seen return distance
        else get the next connections from the new smallest distance node
            and add new distances to total distance
```

I used Visual Studio Code running on a Mac to write, debug and run my application as I was developing it. Visual Studio Code was an IDE I was a little familiar with before starting this project so I thought I would use that over other options like PyCharm. Visual Studio Code has an excellent python plugin and a really good python linter called Pylance.

The hash table I have created handles expanding for more packages just fine. As packages are added the size requirements of the hash table are checked and increased if needed. The fact that my hash table uses a one to one hash essential means that all look ups are done in O(1) time. There is no probing or chaining.

The software is efficient and easy to maintain for a few reasons. One such reason is that both the hash table and graph only require that a object that is going to be stored in it be a sub class of Identifiable. The benefit of this is that all an object has to do is subclass Identifiable, then supply a unique integer and the hash table and graph can store and retrieve the object just fine. Another part that makes this easy to maintain is that all the package handling logic is in a package controller class. This separates the data structure from the implementation. If you want to use a different data structure for the packages just change the controller.

The primary weakness of my current implementation of a hash table is that it takes up more memory than needed. A linear probed hash table that fills up empty spots before resizing would save on memory. The strength is that it is relatively straight forward to use because of the one to one mapping of the id of the object to the location in the hash table

Part D:

I used a hash table to store the package data and a graph to store the location and distance data. I created a Identifiable class that any class can inherit from that insures that the subclass will hash correctly and will also be usable in both the hash table and the graph. However, a side effect of python is that hashing an integer just returns that integer for a hash so the items that get put into the hash table are indexed at the same location as the id of the object if the id is an integer. Both the graph and the hash table are matched up with the data points through the id that is loaded up from the CSV file. Packages are stored into the hash table based on the id of the package and locations are stored into the graph based on the id of the location. Packages also end up getting stored in the graph structure to account for the fact that some packages must be delivered with other packages.

Part I:

One strength of a greedy algorithm is that for this particular problem you can easily set things up based on priority and fill the trucks based on that priority. Also, in this particular case it is fast because we can just loop through the packages and see what packages are left to be delivered and make choices based on those packages. It can be verified that the algorithms chosen for this particular problem work because the total distance of the trucks is under 140 miles and all the packages are delivered according to there special notes and deadlines. A self adjusting heuristic for loading the trucks could have been used and would have been able to arrange the packages on the truck not just by priority but also by distance from last package loaded on the truck. A dynamic algorithm could have been used to plan the route for the packages loaded on the truck. I implemented a brute force traveling sales man algorithm which would have been way to slow to use on a group of 16 packages but on groups of 4 packages the shortest distance could have been found. Those routes could be stored and those routes could be compared against other groupings of 4 or recombined using another algorithm to find a good short distance between the end points of the groups of 4.

Part K:

A hash table is pretty straight foreword. A package class can hold all the needed data and then can just be put into a hash table based on a package ID. There is no time change to a look up when the number of packages changes because there is always a one to one hash to the hash table. The space usage is sizeOf(package class) * number of packages held + over head of the sized hash table. The way my program is currently setup there would be no change to the worst case time if more trucks were added however the more cities that get added the longer the nearest neighbor / dijkstra would get. A big O of n^5 is pretty bad.