Name: Frank Morales
Student ID: 001122005

Program Overview Document

Data Structures and Algorithms II – C950

Purpose of Program

The purpose of this program is to create a delivery application that will accept a list of deliveries along with at most one requirement per package, then deliver them on time according to each package's delivery deadline. There are three trucks, and only two drivers, and the trucks can only go 18 mph. A matrix of distances between the destinations of the packages also has been provided. The packages must be delivered before the end of the day and under 140 miles total.

Algorithm's Logic (Requirements B.1, B.3)

The following is a pseudocode that explains how the delivery program works. I will also include the space-time complexity of each segment of the code, as well as the entire program, using the Big-O notation.

main.py

Call PackageHashTable object to instantiate hash table object

while loop that runs deliver() function until all packages have been delivered
The cost of the first while loop *is n = number of packages/14*, or *O(n)*.

print results of running algorithm

while loop print prompt that allows users to enter a time to check the status of packages at that time.
The second while loop that has to do with the prompt is **n = number of times user inputs info times m = the number of packages**. So worst case would be **O(n^2)** if the user wants to check the status as many times or more times as there are packages.

PackageHashTable.py

For the data structure that holds the package information I used a self-adjusting hash table. The following information explains what each method does. This file also contains the algorithm used to deliver the packages, which is a *greedy algorithm*. This is in the deliver() method and will be discussed there.

Methods:
__init__() :: initializes the hash table and runs the functions that fill the table and the matrix with package info

insert_package(package) :: accepts a package object as a parameter and "hashes" it to the main packages_array hash table using the package id as the index

Name: Frank Morales
Student ID: 001122005

search_package(package) :: accepts a package object as parameter and returns the package if the bucket which corresponds to the package id being searched for is not empty, but returns None if that bucket is empty.

all_delivered() :: this method searches through the table through each bucket and returns false once a package with a status other than “Delivered” is found. Otherwise, the method returns true because that means that all the packages have their status as “Delivered”. This controls the while loop in the main.py file and stops it from executing indefinitely.

print_package_hash_table(timedate object) :: this method accepts a timedate object and runs through all the packages in the hash table comparing them to that timedate object to determine how to print the status of each package.

import_package_info() :: this method runs through the csv file containing the package information and fills the hash table with that information.

import_distance_info() :: this method runs through the csv file containing the distance information and fills the matrix with that information.

Is_appropriate_time_for_special_group() :: returns false if at least one of the packages in the special package list can’t go out due to a delayed delivery time.

deliver() :: so finally we get to the good stuff: the greedy algorithm. In theory, this algorithm is pretty simple. If the program has to choose what to do next, this algorithm will try to search for the closest or easiest thing to do first, and continue to do so until done. My algorithm picks the closest package in the truck until there are no more packages in the truck, then it goes back to the hub for more until all the packages have been delivered. Here is some pseudocode to help visualize this:

---

if able to deliver special package list, the ones that have to get delivered together, “load” these onto the “truck” list
else
    Search through hash table for all packages with status “Undelivered” and put into list.
    Then sort list by deadline so packages with earliest deadlines go first

for each package in prioritized list
    if after delayed time
        “load” package onto truck list until we reach 14 packages

***** HEREIN LIES THE CODE FOR THE ALGORITHM *****
while there are packages on truck
    for each package in truck
        find the package with the closest destination

once the closest package has been found:
-add miles to that destination
-add time it takes to reach that destination
-record time delivered and change package status to “Delivered”
-remove that package from the truck

finally after all packages on truck have been delivered, record the miles and time it takes to get back to the hub to deliver more packages.

Run times for the methods in PackageHashTable.py:
__init__() :: O(1)

insert_package(package) :: O(1)

search_package(package) :: O(1)

all_delivered() :: O(N) for N = number of buckets in hash table

print_package_hash_table(timedate object) :: O(N) for N = number of buckets in hash table

import_package_info() :: O(N) for N = number of rows in csv file

import_distance_info() :: O(N^2) for N = number of rows in csv file

Is_appropriate_time_for_special_group() :: O(N) for N = number of special packages in list

deliver() :: O(N^2) for N = number of packages in hash table

Run time for the entire program :: O(N^2)

Name: Frank Morales
Student ID: 001122005

Programming Environment (Requirement B.2)

To write this code I used PyCharm 2020.3 (Community Edition) software. Build #PC-203.5981.165, built on December 1, 2020. Runtime version: 11.0.9+11-b1145.21 amd64. VM: OpenJDK 64-bit Server VM by JetBrains s.r.o.

My computer uses Windows 10 Home edition and has an Intel® Core™ i7-1065G7 CPU @ 1.30GHz 1.50GHz, with 12 GB of RAM, 64-bit operating system, x64-based processor.

Efficiency and maintainability of software (Requirement B.5)

I believe the code is easy to maintain because I put comments above each important piece of code to explain what that code does. I also did not put everything in just one file, but I broke it up so that if the code needs to be tweaked, it would be easier to understand where to implement that change. For example, if another characteristic is needed to be added to a package, that can be added in the Package class and the method that imports that information from the csv file can be modified to accept another variable quite easily.

Further consideration of the Hash Table

Capability of solution to scale and adapt to growing number of packages (Requirement B.4)

The type of table that I used to store the packages is a direct access table. These are great because they eliminate the risk of collisions because all the buckets are uniquely based on the id of each package. This works just fine and is a much simpler solution when dealing with a relatively small number of packages. Unfortunately, the space-time complexity gets worse as the number of packages grows. The table would also not be a good option if the package ids were strings or started out as a high number. If this were the case, then possibly chaining would greatly help reduce the space-time complexity of the program if it had to handle a large number of packages.

Strengths and weaknesses of the hash table (Requirement B.6)

As said above, the **strength** of the hash table is the direct access to the packages if searching with the unique id that corresponds to the bucket. This makes the time-complexity O(1) when searching with the package id in the case of our program. There is no need to search through a list or data structure. I believe that is one of the biggest strengths of this time of data structure.

I would say a **weakness** would be the fact that the table cannot be sorted according to an object attribute. For example, in my program I would like to sort the packages according to the deadline, but I have to load the packages into a different list and sort them there using the sort method. That is an added step I might not have had to do using a different structure such as a tree.

Storing the data points in the Hash Table (Requirement D.1)

The key used to store the package objects into the array is the package id. This makes searching and pulling the package object from the table extremely efficient if the id is known. I can also store arrays in each bucket that could hold the information of each package but in this case I

decided to formulate a Package object to hold that information. The unordered items happen to be ordered by the id but cannot be stored in the table in any other ordered way.

Time affected by number of packages (Requirement K.1.a)

The number of packages wouldn't affect the time needed to complete the look-up function if the lookup function uses the package id as the parameter because this is the key of the buckets. Even if we do chaining it would barely make it slightly worse because you would just have to hash the id, and then search the array stored at that bucket for the object using the id. Now if there was another look-up function that accepts something other than the id to look up the object, then that would require much more time to look up if there are more packages, because the program would have to literally search through every single bucket and object in the table to find the object that matches that variable. For example, to sort the packages according to the deadline, I have to put the ids and dates in a separate array and then call the sort function, which would take much longer the more packages we have to deliver.

Data structure space usage affected by number of packages (Requirement K.1.b)

The data structure space usage is affected by changes in the number of packages to be delivered because the more packages there are, the more buckets that need to be used. The best way to deal with more packages would probably be to do the chaining on the table.

Changes to number of trucks or cities (Requirement K.1.c)

If changes to the number of trucks or the number of cities were made, it would affect the look-up time and the space usage of the data structure by making the distance matrix bigger, making looking up distances between the destinations longer, and the array would require more space. It might not affect the code itself, but I believe it would make the program require more time and memory to run.

Identify two other data structures that would work in this scenario (Requirement K.2.a)

One data structure that could work in the scenario would be a **dictionary**. A dictionary is different than a hash table because the hash table only uses integers as keys and uses the term buckets to store the objects. A key in a dictionary can be different types, even strings. So if each package had a distinct name, then I could've used a dictionary to store the packages instead of the id. For simplicity's sake though, the direct access hash table is easier in this situation where the id of each package is unique and can be used as the key.

Another data structure that could have been used is a **binary search tree**. This data structure uses nodes that has up to two children and orders the objects as they are added. This structure does not use keys to store objects like the hash table does. This structure would be great to store the packages according to the deadline date so after the information is pulled from the csv file, they can be "loaded" directly on the truck.

Name: Frank Morales
Student ID: 001122005

Further Consideration of the Greedy Algorithm

Two strengths of the algorithm (Requirement I.1)

One strength of the greedy algorithm used in the solution is that it is a **simple** and relatively **easy** choice to implement. Just pick the closest package every time it is run, nothing else. Of course, the end result might not be the most efficient, but this algorithm makes it easy to choose what to do next.

Another strength is that generally the work that is performed by the algorithm is **linear**. We still have to take into account the time and resources it takes to sort the list first and then search for the closest distance, but it is still better than other algorithms that are more complex to be more efficient.

Verify algorithm meets all requirements (Requirement I.2)

The screenshots show that the algorithm used in the solution delivers the packages before the end of the day, before 5pm, and stays under 140 miles. The algorithm also delivers all the packages on time before their respective deadlines. The packages with delayed time also don't get loaded or delivered until the appropriate time. The packages that need to go on truck 2 do so since the only truck used is truck 2. All the packages that are supposed to be delivered together end up going out together too.

Two other algorithms that would work (Requirement I.3, I.3.a)

One algorithm that is different that the greedy algorithm is the Dynamic Programming. Dynamic programming stores solutions to subproblems in memory to then help find the solution to the main problem. This type of algorithm is apparently much more efficient, but uses more memory and is more difficult to implement.

Another algorithm that could have been used is Dijkstra's shortest path algorithm. This one is different than the one I used because it uses the graph data structures with weighted edges and vertices to find the shortest path. The idea behind this algorithm is very similar to the one I used, but the data structure used to store the packages would be different.

What I would do differently (Requirement J)

Honestly, I like the way the program turned out. I think I would possibly just consolidate some of the methods I wrote in the hash table class for the code to be a little more streamlined. For example, instead of having in the main class the while loop that checks if all the packages have been delivered, I would just put that loop in the deliver method and have it run in there until all the packages have been delivered.