Sample Core Algorithm Overview

Jonathan Nguyen, id:000918228

March 31, 2019

Stated Problem:

The purpose of this project is to find the quickest delivery for 3 trucks delivering 40 packages and program a solution for said problem. (In this case, Python). The code should put all the package information into a hash table, and then use a chosen algorithm to find the quickest delivery routes for the trucks. After the packages have been delivered, we are to provide a lookup function and time function for the user to be able to find delivery/package information and how many total miles the trucks drove throughout the day.

Algorithm Overview:

For my chosen algorithm, I decided to use Dijkstra's Algorithm. The algorithm works by:

- Make a hashtable with all the package data
- Make a graph with the distances and locations
- Make truck lists with the packages and their information
- Determining a starting point in a graph
- For each following vertex(location), the algorithm will determine the next shortest path
- Repeat this process until all locations necessary have been visited

Hashtable

Due to the nature of python, dictionaries are considered to be hash tables. Also because of the wide range of possible package numbers, we can simply use the package ID as the key for adding, searching, and removing the packages from our table. Because of the wide range of packages we could be appending, the complexity is linear (O(N)). Here is an example of the code:

```
def Hashtable():

    def __init__(self):
        self.hashtable = {}

    def add_to_hash_table(package, ID):
        key = hash(ID)
        self.hashtable[key] = package

    def search(ID):
        key = hash(ID)
        if key in hashtable:
            print(hashtable[key])
        else:
            return None

    def remove(ID):
        key = hash(ID)
        if key in hashtable:
            self.hashtable.pop(key)
```

Another data structure that could also be used is lists. A list, unlike a dictionary, does not have a key and values. Therefore in order to avoid collisions, one could simply have lists within the main list, which would allow multiples of the same key without collisions.

Graphs

The next step would be to setup our graph. A simple choice would be a dictionary, with the current location as the key, and another dictionary inside that dictionary which would contain the locations and distances relative to the main key. This could be an issue however if the amount of locations grow too much. Keeping such a large dictionary file could prove to be troublesome. Here is a quick example of a python dictionary graph:

```
graph = {
    'home': {'A': 1, 'B': 2},
    'A': {'home': 3, 'B': 5},
    'B': {'A': 5, 'home': 2},
}
```

Another possible interpretation of a graph could be done by making a function that contains all of the locations, and simply building weighted edges that represent miles between certain key locations. This would simply the end result and growth, but the execution and coding could be more difficult than a dictionary.

Trucks

For this project, a list was used to represent the three trucks. To sort priority packages, we used an if statement that said if a packages' delivery time was not End of Day, append the package to our priority truck. After that, we simply used a heuristic algorithm to fill the trucks to maximum capacity (16), of which the complexity is O(N). This is because the workflow increases/decreases depending on the number of packages there are. Here is an example of that code:

```
priority_truck = []
second_truck = []
third_truck = []

if the package delivery time != End of day and length(priority_truck) <= 16:
    priority_truck.append(package)

elif second_truck <= 16:
    second_truck.append(package)

else:
    third_truck.append(package)
```

Perhaps another data type could be a dictionary or a tuple. But given that the values are already in the package data, a dictionary might not have enough values to justify being used. A tuple, compared to a list, is immutable and therefore unchangeable. This would be okay in an ideal world, but in a program that may require changing the truck information, this data structure might also not be plausible for this scenario.

Dijkstra's Algorithm

Once that is done, we run dijkstra's algorithm on each truck list.

We do this because each package has a distance table attached to it, so the algorithm will first have to search for the location, and the the next location inside the values of the current place. In the pseudo code, we can get an idea of how this is done. Assume that a graph (dictionary) with location and distance data has already been created.

```
def Djikstras_Algorithm(self):

    for neighbor, distances in graph.items():
        if the neighbor is not in untouched:
            continue

        self.touched[self.current_place] = self.current_distance

        sorted_locations = sorted(graph[location], key = lambda x: x[1])

        for i in range(len(sorted_locations)):
            self.current_place == sorted_locations[i][0]
            self.current_distance == sorted_locations[i][1]
```

The cost of this is ON^2 because we start off with the initial for loop, which runs through the graph dictionary. While inside that loop, we run the sort on the values of our current location and run a range loop to determine the next location and distance. Because we are looking at the shortest distance to the next location on an individual basis, as opposed to searching only from the hub, we can be sure that the path the trucks will deliver on is the most efficient and accurate.

Total Distances Traveled

Another objective is to find the total distance traveled by the truck. This can be done by simply taking the "touched" dictionary that our algorithm inputted and adding up all the values together. Once they are added together, we then need to pull the distance from the final location to the hub. The first part determines how far the truck went to deliver the packages, and the second part is how long it takes to return home. Because the length of the list varies, the complexity would directly correlate with the size of the list, or O(N). Below is the pseudo code for such a program:

```
For values in touched.items():
    total_distance += values

    if len(touched) == 1:
        for last, distance in graph[location].items():
            if home == 'home':
                total_distance += distance
```

Lookup Function

For the lookup function, the user will be prompted to state if they would like to see an individual package details, or if they would like to see the status of all packages at any given time. Lastly, the code will prompt the user to see the total distance of all the trucks. Because it simply returns a value after an input, the complexity is constant, or O(1). Here is the pseudo code:

```
While True:

    question = input('Do you want to see 1 package or all?')

    if question == '1':
        packagenumber = input('What is the package number?')
```

```
            print(hastable.search(packagenumber))
    else:
        time = ('What time did you want to see the status for?')

        for i in range(len(hashtable)):
            if time > delivery_time:
                print(f'The delivery time was {delivery status}')
            else:
                print(f'The status is {current status}')


    print(f'The total distance traveled by the trucks are {totaldistance}')
```

In the above pseudo code, if the user wants to see the information for 1 package, the program will run the hash table search function on the package ID listed and return that information. If the user wants to see the total information, they are given an option to see what time they to see the status of all the packages. To determine the status, the delivery time information is compared to the inputted time. If the inputted time is greater, that would mean that the package has been delivered, thus it returns the delivery time. Otherwise we simply return where the package is at the time.

Implications

If this project's code were to be expanded into other cities and/or more locations and packages were added, most of the structures in place would be fine. For the hashtable, because there is no shortage of characters/numbers that can be used for package names, the hash keys should have no problem inserting a growth in the number of packages.

For the graph, should the project expand and grow into cities, should be sufficient with growth. This is assuming that there is no overlap in destinations in said cities. Because we chose a dictionary as the data structure for our graph, adding a new delivery location and distance is as simple as appending the values as needed. The biggest change however, would be to set the graph variable name as the city, to avoid confusion. Another thing that could be done is to use the addresses as keys, rather than letters as was done in this project.

For our delivery trucks, given that a list is mutable, a list should be sufficient to store and hold the package data even when accommodating exponential growth. Because of the limitations of the truck capacity, continuing to use a heuristic algorithm to load them would be viable for growth and expansion.

For our core algorithm, as long as the prior data structures are correct, using Dijkstra's algorithm to find the shortest route will continue to be beneficial in delivering all packages in a timely manner. Even when adding new routes or locations, the algorithm will find the shortest path to the next location dependent on where the truck currently is. So for scaling, the complexity should remain $O(N^2)$ no matter how large the delivery route is.

Sources

Lyseky, Roman, and Frank Vahid. "C950: Data Structures & Algorithms II." ZyBooks, learn.zybooks.com/zybook/WGUC950AY20182019.