Kyle Kitchens

Data Structures and Algorithms 2

C191

WGUPS Routing Program Overview

**A.  Identify a named self-adjusting algorithm:**

I used a nearest-neighbor algorithm to determine the sequence of stops.

(optimization.py, lines 184 – 193).

**B.  Write an overview of your program, in which you do the following:**

*1.  Explain the algorithm’s logic using pseudocode:* **Pseudocode in bold**

The purpose of this program is to find a delivery path for trucks carrying packages. To determine the sequence in which the trucks will stop references to distance data and package data must be used. Truck objects are also used. The Hash table package objects, distance table, address table, and truck objects are initialized at the beginning of the program

**Initialize Trucks**

**Initialize Packages and Hash Table**

**Initialize distance table**

**Initialize address table**

The Trucks are then manually loaded onto the trucks. The truck objects have a manifest attribute which is an un-initialized dictionary. Initially, packages are put into this dictionary with the address index as the key and the package id as the value.

**Load trucks**

**For each truck**

**Initialize truck manifest dictionary**

**Add package list to manifest with address index as key and package id as the value**

In order to produce a route for the truck all of the values from the truck manifest are combined into a list and gone through to find like addresses and combine them under one key to remove duplicate keys or stops. Then a list of keys is made and a copy is made called temp. Then the original list has a 0 added to the first and last position to account for the index of the hub. At most, 16 addresses indexes will be in this queue.

**For each truck**

**For all packages on truck**

**If packages have the same address**

**group packages into lists in the truck manifest dictionary**

**initialize list of stop indexes**

**add truck manifest dictionary keys to list of stop indexes**

**temp = copy list of stop indexes**

**add 0 to first and last position of list of stop indexes**

The nearest neighbor algorithm uses the complete list of stop indexes which include stops at the hub and uses the temp list as a comparison in order to determine which stops are still available for comparison. The mileage is determined from the hub to all of the locations indexed in the temp list and the closest index is determined. Then the second position of the original list of stop indexes is updated to the closest index, and that index is removed from the temp list to prevent duplicate stops. This is then repeated for all positions of the list of stop indexes and the final result is saved to a truck object attribute called stop sequence and packages are delivered.

**for every position in list of stop indexes except the first and last position**

**initialize closest address**

**for every position in temp**

**reference distance table from the stop index of current position to stop index of temp.**

**if position in temp is closer than closest address**

**closest address = current position in temp**

**current position in stop indexes = closest address**

**remove closest address from temp**

**truck stop list = list of stop indexes**

**Deliver packages**

A package’s status is determined by determining the truck the package is on, then determining the total mileage for that trucks route and computing the time for delivery based on 18 miles per hour. That time is added to the start time and compared to the input time to determine the status of the package.

**Get package status**

**Determine route distance to destination**

**Time from hub = route distance / 18 mpg**

**Delivery time = truck start time + time from hub.**

**If current time is greater than delivery**

**Status = delivered**

**If current time is less than delivery, but greater than start time**

**Status = en route**

**If current time is less than delivery**

**Status = at the hub**

*2.  Describe the programming environment you used to create the Python application:*

Python 3.11.2 was used with virtualenv 20.16.7 virtual environment library.

PyCharm 2022.3.2 was used. The program was run on a local machine using local data. The machine make and model is an HP Laptop 17-by3xxx running Windows 10 version 21h2.

*3.  Evaluate the space-time complexity of each major segment of the program, and the entire program, using big-O notation.*

Complexity is in code comments

*4.  Explain the capability of your solution to scale and adapt to a growing number of packages:*

The software is capable of scaling the capacity of packages. given that package data and distance and address CSV files are all updated. An unlisted address could lead to errors. This program also “manually” loads the trucks, or decides what packages go on which truck. Without a function to sort the notes, deadlines, and other constraints, scalability will always require manual intervention. A systematic way to present the data to the existing software without stripping headers off of CSV files and deciphering notes would lessen the requirement for the continual effort to achieve scalability. Most of the program is for data access and data manipulation. The core workload of the program is the nearest neighbor algorithm which requires repeated access to the data structure that contains distances. The majority of the program will have its workload grow in a linear fashion with regard to the number of packages. The workload from the nearest neighbor algorithm only looks at the packages per truck though. As long as the limit is for a maximum of 16 packages, there will be a maximum of 16 addresses for the algorithm to compare distances per truck, but the number of trucks may increase. Since the data structure that contains distances has an access time complexity of O(2) (optimization.py line 45) and the limit on the number of comparisons the nearest neighbor algorithm can make, the workload on the nearest neighbor algorithm should grow linearly as well as more trucks are added.

*5.  Discuss why the software is efficient and easy to maintain:*

The software is only limited by the input CSV files, and the loading configuration (optimization.py lines 5-7). The software itself is easy to maintain, but the caveat is that the data presented to the program must be cleaned and presented in a particular format. The software’s efficiency is derived from its polynomial time complexity and worst-case scenario n^2 space complexity.

*6.  Discuss the strengths and weaknesses of the self-adjusting data structures:*

The hash table has an access time of O(1) time complexity but it returns a package object of which its attributes can be accessed in O(1) time complexity. Python uses dictionaries to instantiate objects and essentially there are two keys anyway. Tests done by jessicald on GitHub indicate that the performance of dictionaries alone are better than objects because objects have more overhead than dictionaries. Overall performance could be increased by associating a key with every package attribute, but this would be at the expense of space.

**D.  Identify a self-adjusting data structure, such as a hash table, that can be used with the algorithm identified in part A to store the package data.**

*1.  Explain how your data structure accounts for the relationship between the data points you are storing.*

The package data is stored as a package object of the package class (package.py). the chaining hash table takes the ID of the package and uses the integer value as a key to return the package object as a value. The package attributes can be manipulated by using the search method in the “HashTable” class. (hashtable.py line 49) and addressing the value for each attribute. The attributes that can be manipulated this way are the Id number, destination address, destination city, destination state, destination zip code, delivery deadline, package weight, package notes, and package status.

**I.  Justify the core algorithm you identified in part A and used in the solution by doing the following:**

*1.  Describe*at least ***two****strengths of the algorithm used in the solution:*

The algorithm is simple and easy to troubleshoot, and it is scalable given that all of the data scales with it. The nearest neighbor function used in this application addresses each truck individually. So, while the algorithm is n^2, the n passed to the nearest neighbor algorithm will only include a subset of all packages which are then coupled if they have a like address before running through the nearest neighbor algorithm.

*2.  Verify that the algorithm used in the solution meets*all*requirements in the scenario:*

The total mileage came to 103.8 miles. The other requirements have more to do with the way the truck was loaded and the times that trucks left the hub.

*3.  Identify****two****other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario:*

A depth-first search algorithm could be used to determine a route in this scenario as well. The depth-first search algorithm starts with a minimum spanning tree and starts at the root node. A path is found to the bottommost and leftmost mode of the tree and goes back up the tree toward the root as necessary to reach other branches. The resulting path is each edge from node to node is traveled twice. Duplicate nodes are then deleted from the sequence leaving a route where every node has been visited just once. This is different from the nearest neighbor algorithm because it starts with a feasible overall route and subtracts duplicate routes and the nearest neighbor algorithm only looks at the shortest distance from the current node. The depth-first search algorithm takes the approach of looking at the wider picture of the entire route while the nearest neighbor algorithm is only concerned with the next node.

The Christofides’ algorithm would also work in this scenario. The Christofides’ algorithm also starts with a minimum spanning tree but works to find possible edges between nodes with only one edge leading to the node which then creates a cycle when another edge is added to the route. This is also different from the nearest neighbor algorithm because it starts with a partially defined route and connects edges to complete the route.

**J.  Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again:**

The Primary thing I would do differently would be to use a multi-threaded approach to the truck and use a simulated real-time nearest neighbor algorithm where the lists of addresses are in a queue and the nearest neighbor is not calculated until the current stop is completed. The approach I took was to determine the entire route before the truck was leaving, but this requires the route to be more inflexible to changes.

**K.  Justify the data structure you identified in part D by doing the following:**

*1.  Verify that the data structure used in the solution meets*all*requirements in the scenario.*

a.  Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered:

The lookup function has a time complexity of 0(2). The distance is accessed by a dictionary of indexes of addresses (0-26) with a list of corresponding address indexes (0-26). It is accessed in the format “distance\_table[current\_index][new\_address\_index]”.(“Complexity of Python Operations”, https://www.ics.uci.edu/) The “distance\_table” hash map is the primary data structure as it has the most repeated access. The address table and the package hash table have the most accesses while gathering package information initially where the distance data is accessed repeatedly in the nearest neighbor algorithm. That being said, the lookup function is primarily affected by the number of addresses, not necessarily the number of packages.

b.  Explain how the data structure space usage is affected by changes in the number of packages to be delivered:

The Space complexity is primarily N^2 since every address must have a reference to every other address. The address data has a linear space complexity in a worst-case scenario if every address is unique and becomes better with every matching address.

c.  Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure:

The lookup time would remain constant, but the space usage will continue to be n^2. The number of cities would have no direct effect, but the number of addresses within the cities would have an effect, assuming that there are packages associated with those addresses. The lookup function starts by checking the address of a package, so if there is no package associated with the address the data structure would remain unaffected except for the initial loading. The number of trucks has no influence because the search function is based on the number of packages regardless of the number of trucks.

*2.  Identify****two****other data structures that could meet the same requirements:*

The data structure mentioned in part D of the requirements is a chaining hash table initialized with 10 buckets. Each bucket contains lists that contain key-value pairs for each corresponding remainder after running the hash function and the values that the hash table return are package objects. Alternatively, instead of having objects at all, a single dictionary could be used to store all package data, which would make the time complexity O(1) for accessing any package attribute and reduce the program overhead by not initializing any objects. I decided against this because it would increase the necessary complexity of a search function. Another data structure that could have been used is a list. According to the University of California, when Python lists are accessed by their index, they have an access time complexity of O(1) as well, but with even less overhead than a dictionary. The index could be used like a key in a dictionary or hash table, but would not result in any substantial improvements and would add unnecessary complexity to a search function.

**Sources:**

Class vs. Dictionary; or, An Example of Space–Time Tradeoff in Python, <https://gist.github.com/jessicald/2861038>

Joe James, TSP Approximation Algorithms | Solving the Traveling Salesman Problem

<https://www.youtube.com/watch?v=M5UggIrAOME&t=460s>

Python Complexity Classes, <https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt>