Lucas Roberts

WGU C950 – Data Structures and Algorithms II

29 January 2020

WGUPS Core Algorithm Overview

**Stated Problem:**

The Wester Governors University Parcel Service (WGUPS) needs to find the best route and distribution to deliver 40 packages listed on a manifest utilizing two drivers, and three trucks. A solution needs to be presented that delivers all 40 packages by the end of the day with package-specific requirements to the provided locations. Locations, a map, and distances between the locations are provided as well as a list of packages and their specific deadlines, locations, and specific requirements. In addition, a manager should be able to see the status of each package at a given time during the day.

**Algorithm Overview:**

Using a greedy algorithm, I was able to deliver all 40 packages in 128.9 miles. Alongside this greedy algorithm, I created and implemented a hash table from scratch to store package data and utilized a dictionary to store the location data. Within the greedy algorithm, a priority algorithm was used sparingly to ensure that packages with deadlines and special requirements were delivered on time. The algorithm keeps track of the time by advancing time at a specific rate proportionate to the 18 mph average speed of the truck during travel. While all packages are delivered on time and meet their stated requirements, the travel distance of each truck was not at a completely optimum level. That being said, with some small changes, this algorithm could be scaled up or down depending on different requirements.

In general terms, the algorithm functions as follows:

1. Parse through package data and load packages to trucks depending on requirements. Packages are subject to a number of filters before being loaded. Trucks are loaded in full before they embark.
2. Create a list of addresses for each truck that correspond to the truck’s cargo manifest.
3. Each truck begins its delivery route. The route is created by looking at a sorted list of the truck’s current locations neighbor addresses that also have packages currently on the truck that need to be delivered. The closest neighbor location is chosen, and the truck embarks to that location to drop off the package. This repeats until the truck is empty.
4. The truck drives back to the hub to pick up the next round of packages. Packages are manually sorted via code here to ensure timely delivery.
5. The truck repeats the previous steps but creates a high priority manifest prior to embarking so that it can deliver the packages with high priority deadlines first.
6. The truck returns to the hub if required and repeats these steps.

The delivery route method “.next\_stop()” is illustrated using pseudocode below:

**closest\_neighbor(cargo\_list):**

**temp\_list = []  
for Package, Location in cargo\_list.items():  
 for k in location.neighbors.items():  
 if k[0].name == Location.name:  
 temp\_list.append(k)  
  
temp\_list.sort(key=lambda loc: float(loc[1]))  
return temp\_list[0]**

**add\_miles(miles)**

**truck.odometer = truck.odometer + miles  
time\_advance = miles / ((18 / 60) / 60)  
if time\_to\_stop > truck.time + timedelta(seconds=time\_advance):  
 truck.time = truck.time + timedelta(seconds=time\_advance)  
else:  
 while time\_to\_stop > truck.time:  
 truck.time = truck.time + datetime.timedelta(seconds=10)  
 truck.time\_up = True**

**next\_stop():**

**while not time\_up:  
 next\_location = truck.location.closest\_neighbor(cargo\_addresses)  
 miles = float(next\_location[1])  
 add\_miles(miles)  
 truck.location = next\_location[0]  
  
 for Package, Location in cargo\_addresses.items():  
 if Location == truck.location:  
 package\_to\_deliver = Package  
 break;  
 if truck.time\_up == False:  
 truck.remove\_package(package\_to\_deliver)  
 package\_to\_deliver.delivered\_at\_time = truck.time  
 break;**

While programming models do not explicitly apply to this specific scenario, if the requirements were to be scaled up as discussed in the project rubric, persistent data should be used in the form of a database using tables for package, location, truck and perhaps even driver data. Presently, the location and package data is parsed in from formatted CSV files via the csvreader that is built into Python 3.8. The data is utilized to create package and location object instances using \_\_init\_\_ methods and hash table insert methods.

Throughout the coded algorithm, comments are left for each code block to describe what it does and what time complexity it has. The entire algorithm has O(N^2) time complexity. While the hash table that the package data utilizes is extremely efficient and the sort method completed with each use of the closest\_neighbor() method only has a O(n logn) complexity, the worst case run time is O(N^2).

While this program could not be immediately ran with a completely different data set without adjustment, the algorithm itself could be used on a larger scale in a different region. Because the data is read in from an outside source and includes distances to relevant neighboring locations, the greedy algorithm itself would work. As the need for more package deliveries increases in society, it can be assumed that new challenges will present themselves. Occasionally, the algorithm will need to be adjusted in itself to offer solutions to these new issues. At a certain point, from a workforce point of view, a third driver would need to be hired to handle the rise in deadline requirements and general deliveries. Furthermore, with a different dataset, the algorithm would need to be adjusted to properly account for the new dataset’s requirements. For instance, at a certain point in the algorithm, a truck is instructed to ignore a particular package due to a delay in an address update. This is hard coded in and obviously would not be applicable to a different package manifest.

The program could be more efficient. As stated by Lysecky and Vahid, a greedy algorithm “chooses the option that is optimal at that point in time. The choice of option does not consider additional subsequent options, and may or may not lead to an optimal solution.” While this particular greedy algorithm does incorporate the use of a priority list at times, it does this out of necessity. This software is maintainable in the sense that is object-oriented and thoroughly commented. Someone who did not originally work on the project can at the very least follow the algorithm’s path by looking at the comments. The section of the program that parses in the data is separate from the algorithm and could be adjusted to support a persistent data technology such as a MySQL database. If a new data attribute needs to be created for insight purposes, it can be added to one of the objects easily.

As previously stated, I chose to implement and create a hash table data structure from scratch due to requirements presented in the project. This hash table includes an insert and a lookup function with packages in mind. This hash table takes in the package’s ID and runs it through a hash function to decide which bucket and bucket list to place it in. There, the package object is stored with all of its data attributes. A benefit of the hash table data structure is that lookups and inserts with the data structure have an O(1) time complexity. This is extremely ideal for algorithms that require constant inserts and removals and is very efficient. However, iterating through the hash table is still O(N) complexity despite the ease with which it looks up specific items. While the hash table is useful with a small data set, with much larger data sets, collisions can be hard to avoid. This could create inefficiencies and problems for the algorithm and ultimately cause packages to not be delivered if left unchecked.

Other data structures could have been used to store the package data. One data structure that could have been used is a list. While a list could have easily stored the data for the entire program, an insertion takes O(N) time complexity and would be almost useless with an extremely large data set. Another data structure that could have been used is the built-in dictionary data structure. While this is essentially a hash table, it stores data in key-value pairs whereas a the hash table that was implemented in this project takes in a key, runs it through a hash function, and sorts the objects into bucket lists to avoid collisions. While this may not be considered a typical data structure, the package data could also have been stored in a database as previously mentioned. This would prove more feasible for larger data sets but comes at the cost of a larger insertion, lookup and removal time complexity.

I also utilized a dictionary, a type of hash table built into python, to store location data. This was particularly useful for storing a list of neighboring locations and their distances for each location object. Before implementing this data structure for this purpose, I had originally began using a graph data structure. However, it proved simpler for me to use the dictionary with neighboring locations as keys and their distances from their parent object as values.

The modified greedy algorithm I chose fulfilled the requirements of each package and the scenario and was able to deliver the packages in a reasonable amount of time and distance. At its core, it is adjustable for many types of scenarios and it is fairly easy to explain as well as understand for those that may be working with it in the future. That being said, there are more efficient algorithms for this type of scenario in existence that could have cut the distance travelled down significantly. Dijkstra’s algorithm would have cut the distance traveled down significantly but may not have been as easy to manipulate so that requirements were met. Dijkstra’s algorithm works by determining the shortest distance from a starting vertex to a destination vertex in a graph. Combined with a priority list, this algorithm could have cut down the distance traveled immensely. However, I believe it would have been difficult to have implemented this alongside a priority list. Another algorithm that could have been used was a non-modified greedy algorithm. Also called the nearest neighbor algorithm, I had originally implemented this before ensuring that package requirements were met. The total miles travelled using this algorithm was under 100 but several packages were not delivered on time. This algorithm works similarly to mine in that it is always searching for the next closest location, except that it does not include any kind of priority list.

If I were to do this project again, I would have spent more time looking over the data before beginning the coding portion of this algorithm. I feel that I could have saved time by getting a more insightful look at the data requirements. I also would have researched more about the travelling salesman problem, as it is very similar to this scenario.

Sources Cited:

Lysecky, Roman, and Frank Vahid. *C950: Data Structures and Algorithms II*. Zyante, Inc., 2018.