**WGU Truck Route**

**Core Algorithm Overview**

**Stated Problem:**

The following algorithm overview covers the methods and coding used to solve a truck route problem. The problem consists of packages that need to be delivered to a host of addresses spread throughout Salt Lake City. The scenario consists of 27 locations including the central starting point that the delivery trucks must deliver packages to. There are 40 packages and each have varying amounts of requirements that need to be fulfilled. Some of them have early delivery times though most are due at the close of the business day. There are wrong addresses to correct, delayed packages, packages that must be on a certain truck, and others have to be delivered together. There are three trucks but only two drivers and only 16 packages are allowed on each truck. The programming language selected for this project was python 3.8.3 in the PyCharm 2020.1.2 community edition IED.

**Algorithm Overview:**

The truck route algorithm uses a greedy algorithm at the core to find the shortest points between each address. The process of the program can be explained in several steps:

1. Create a hash table that uses keys as identifiers or unique items to store data.
2. Read in CSV data from files and save the data to the hash table.
3. Assign routes as lists to each of the trucks and optimize the routes
4. Deliver the packages following the constraints
5. Create a user interface and code that allows a user to look up information about the packages

Once the data is read into the program, the csv data is stored in the hash tables using unique keys, much as a SQL table works. The various functions allow data to be read in, updated, read, and removed. Just before the data is saved to the hash tables, the trucks are assigned and loaded. Since there are two drivers, the first truck makes a run and then returns to fill up on packages and then goes on a second trip. With the trucks loaded, the package lists are then sent through a gauntlet of functions that order and optimize the list. Once the list is optimized, the list is then sent to a delivery function that timestamps and updates the packages. The program runs and delivers all of the packages and then a user is able to look at various elements of the packages. The main discussion that follows focuses on the main function used to optimize the routes.

As mentioned previously, the main algorithm used is a greedy algorithm. The lists sent to the getOptimalRoute function, as it’s named in the program, are used to create a new list that orders the addresses in a more optimal manner. The function takes advantage of dynamic programming in that it is recursive and breaks the steps down into smaller problems. Essentially a for loop finds the shortest distance in a set of addresses. This shortest distance is then saved and used as the first destination on the route. The location is used and the function calls itself recursively to find a new shortest distance. As each point is visited, the function continues to find the shortest distance between points and adds that point to a growing list. As the function continues this process, the original list is cut shorter and shorter. This is to simulate going from point to point as visited points are no longer needed. More importantly this gives the result of eventually going to zero so the base case of the recursion can end the function. If the base case were never met, which is that the original list goes to zero, then the function would call itself through infinity and never stop. The greedy aspect of the algorithm is the fact that the function always grabs the shortest distance from a list. This isn’t the most optimal method of solving the problem, but it works. This is a good example of dynamic programming in that it breaks the problem down into smaller steps. The greedy algorithm fully meets the requirements of the problem and successfully finds optimal routes. Draw backs and other possible solutions are discussed later. The pseudo code below outlines the getOptimalRoute function.

**getOptimalRoute- takes a list and integer location value**

**Base Case- if len( list) is 0**

**Return optimized list**

**For i ¬ list**

**Get distance between points**

**Check if it is shortest**

**Shortest is saved**

**Location saved**

**For i ¬ list**

**Get distance**

**if the distance is equal to the shortest**

**add this point to optimized list**

**save the location**

**remove this point from list**

**call the function again (shortened list, new location)**

The space time complexity of this algorithm is O(N^2) due primarily to the two for loops. The best case and the worst case are the same since the two for loops each cycle fully through their loops. A longer list will result in an increased number of recursive calls but this maintains a linear relationship to the number of items in the list. Though there is likely a more efficient way of finding the shortest distance without two for loops, this seemed to be the easiest way to implement the code. The following show other core functions and their complexity.

**Methods:**

assignLoctions(list) :: returns list

getDistance(row, col) :: returns distance

getOptimalRoute(list) :: returns optimized list

getTimeStamp(distance, truck number) :: returns timestamp

deliverPackages(list, truck number) :: returns total distance

**Running Times:**

assignLoctions: O(N^2)

getDistance: O(1)

getOptimalRoute: O(N^2)

getTimeStamp: O(N)

deliverPackages: O(N)

**Scalability**

The scalability of the whole truck route program has some strengths and weaknesses. The strengths lie in the way the hash table can be fully scaled and the central functions discussed above all take one list and calculate through it. If there were to be more trucks and far more packages, the hash table would be able to increase in size and the though the trucks were loaded manually, there is code that is commented out that would be able to be used. It would not be fully efficient but it would work. It would not take too much effort to improve the code for loading the trucks as this is the greatest current weakness. Once the trucks are loaded and routes created per truck, the scale could easily increase as the functions flow in a linear manner one at a time. This makes trouble shooting very easy and enables as many trucks as desired to be sent to through the core functions. Once complete, the code that draws the data from the hash table to the user would work essentially the same.

**Maintainability**

The creator of this project is limited in experience and a senior develop would likely disagree, but the author of this project considers the code of the truck route algorithm to be quite maintainable. There are multiple functions and creating new ones or adjusting the current ones in operation would be easily accomplished. Since there are several files with well defined functions for each key step of the algorithm, the flow of the program is easily followed and adding new functions would tie right into existing ones with ease. The repeatability of code is minimized and each function flows linearly from one to the next.

**Self-Adjusting Data Structures**

The choice of a hash table and basic functions makes the truck algorithm quite static or non-self-adjusting overall. The implementation of heaps and trees etc. would greatly increase the self-adjusting nature of the algorithm. Trees in particular have systems and ways to maintain balance where heaps maintain relationships when one is reassigned or removed. The size of the hash table is likely the only structure to really change in the current form of this project. The list sizes would increase or decrease depending on the needs but this is not a self-adjusting structure. There are essentially no self-adjusting structures in the code in its current state. An advantage to this is simplicity. To scale production would simply require increasing the volume of the storage space used. This would only work to a point however. If the scale were to increase substantially a number of issues would arise. Of primary concern would be the speed at which the algorithm would be able to maintain. Also of interest would be keeping the information organized and structured in a useful manner. The advantage of a self-adjusting structure is the faster, more efficient manner in which more utilized data can be found and accessed. The height of trees would be of concern but otherwise highly efficient. For simplicity sake and ease of use, the algorithm uses fewer complex structures and solves the problem in a dynamic way.

**Conclusion**

The greedy algorithm is simple and attacks the problem one small piece at a time. By finding the shortest route, a series of points can be traversed and yield a fairly efficient solution. It isn’t the most optimal way of solving the problem, but it does so none the less. The code is easily implemented and does not require complex data structures to do so. Though the algorithm solves the problem, there are other possible means by which there could be ascertained a solution.

One that came to mind early on in the planning phase of the project was dijkstra's algorithm. This algorithm relies on finding the shortest distance between nodes in a graph. It is highly relational and is able to find complex routes by comparing edges of and finding the shortest route between a starting destination and a final destination with any number of points between. The run time increases with the number of edges and vertices. A draw back would be the implementation of a complex graph structure capable of accurately fulfilling the algorithm’s needs. Once set up however, the results could be highly efficient. Though efficient, they would still be an approximation and more of dynamic approach to the solution. Another useful algorithm that could be used is the Bellman-Ford algorithm. This is also a graph dependent algorithm that finds the shortest distance path from a single source to many other vertices. A draw back is it is rather slow, even slower than dijkstra's. It is more versatile however in that in can handle more dynamic relationships as well as negative numbers.

There are other interesting algorithms that could be used but this is not by any means an exhaustive list. If this project were to be repeated, I might attempt dijkstra's. Though it is more complex to fully implement, it would prove interesting. Indeed, the most interesting would be to see the results of the algorithm’s output. If this project were to be redone, there is little that I would do differently. The hash table proved to be simple and affective. Though it would have been good to better learn how heaps, trees, and graphs work, a simple solution saved time and effort. As a first python project this was challenging and quite engaging. Future endeavors to utilize more complex structures would be very interesting for a greater challenge.

**Sources**

Advanced Algorithms Session 2015-16 Lecture 8: Self adjusting data structures, Jan. 12, 2016 Faculty: Chowdhary, K.R.

zyBook C950: Data Structures and Algorithms II**.** Jun 2018. [Lysecky](http://www.zybooks.com/authoring-team), R. [Vahid](http://www.zybooks.com/authoring-team), F

*Structure in communication nets*. Proceedings of the Symposium on Information Networks. New York, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn. pp. 199–203. (1955). Shimbel, A.

Algorithms, What is Dynamic Programming With Python Examples. 5 Jun, 2019. Brandon Skerritt.