Core Algorithm Overview

**Stated Problem:**

The purpose of this project is to create an algorithm that successfully determines the optimal or near-optimal route for package delivery by the [Company] Parcel Service in [City, State], using a common programming language; Python 3.7 in this case. The algorithm and code should be designed in a way so as to be scalable and replicable for use by [Company] Parcel Service in other cities, with minimal rework or modification. In this scenario there are 40 packages, many with special condition parameters, to be delivered to 26 different locations as quickly as possible while meeting the package parameters, such as delivery time deadlines, delays or groups of packages, etc. [Company] Parcel Service has two drivers and three trucks with which to deliver all the packages.

The proposed solution to this problem will first sort and separate the packages by their requirements and load them on to the trucks as truck space is available. Once loaded, a greedy algorithm (based on Dijkstra’s Shortest Path algorithm) will determine which packages need to be delivered first (time deadlines) and then determine which location has the next shortest path from the last delivery location in a loop-like fashion until no packages remain. Once a truck is empty it will return to the ‘HUB’ to either pick up more packages if needed or simply wait for the other truck to return. This paper will examine the proposed algorithm solution and assess its structure, efficiency and viability for future use by [Company] Parcel Service.

**Algorithm Overview:**

The greedy algorithm is implemented as follows:

1. Once a truck’s packages have been loaded, the location addresses of those packages are all assigned paths to every other location on the truck’s route.
2. The start location is always the [Company] Parcel Service HUB. The current location’s available paths are analyzed and the shortest path is selected for the next delivery location if no other package constraints exist. If packages have early delivery deadlines, they are delivered first, regardless of their shortest path status.
3. Once the truck arrives at its next location, the package is delivered and the previous location and all paths to that previous location are removed from the remaining available location paths.
4. After the previous location and its path are removed, the shortest path to the next location is determined and the process is repeated

The worst- and average-case runtime of this algorithm is O(N2). The fewer the packages and the fewer constraints, the better it performs. The more packages and more constraints, the poorer the algorithm performs. Finding the next shortest path is an O(N2) operation and determining if any package constraints exist is also an O(N2) operation. An overview of the algorithm’s structure and pseudocode is presented below.

**Greedy Algorithm**

1. **Given parameters**
2. **truck ( an object of the truck class, has package and location lists as well as time and distance   
    markers)**
3. **truck\_package\_list ( a truck’s assigned list of packages )**
4. **truck\_location\_list ( a truck’s assigned list of locationNodes )**
5. **locationNode ( an object of the locationNode class, represents locations and each Node has an   
    edge\_list of paths or “edges” to all other locationNodes )**
6. **edge ( every edge is assigned a “weight” that represents the distance in miles between any two   
    locations )**
7. **edge\_list ( a list of all paths from a specific locationNode to all locationNodes )**
8. **min\_edge ( variable that represents the shortest path in current\_location’s edge\_list)**
9. **current\_location ( variable that represents where the truck’s current location)**
10. **next\_location ( variable that represents the truck’s next destination )**

**Time-Complexity is O(1)**

1. **Assign edges to locationNodes’ edge\_lists in truck\_location\_list**

**for locationNode\_1 in truck\_location\_list:**

**for locationNode\_2 in truck\_location\_list:**

**if locationNode.address != locationNode2.address:**

**create new edge of distance between locationNode\_1 and locationNode\_2**

**locationNode\_1.edge\_list.append(new edge)**

**remove duplicate edges**

**Time-Complexity is O(N2)**

1. **Determine the base case to break the algorithm’s Recursive Loop**

**if length of truck\_location\_list equals 1, then deliver final package at current\_location, exit loop   
 and return to HUB   
  
  
Time-Complexity is O(1)**

1. **Enter the Recursive Loop**

**While truck\_location\_list > 1:**

**set first edge in current\_location’s edge\_list to be min\_edge**

**for edges in current\_location’s edge\_list:**

**if any other edge in edge\_list is smaller than min\_edge,   
 that edge becomes min\_edge (shortest path)**

**for packages in truck\_package\_list:**

**if any package address matches current\_location, deliver package**

**for locationNode in truck\_location\_list:**

**for edges in locationNode.edge\_list:**

**if any edge in any location\_Node’s edge\_list points to current\_location,   
remove that edge from edge\_list**

**set next\_location to be min\_edge’s target location (shortest path)**

**remove current\_location from truck\_location\_list**

**set next\_location to be current\_location (shortest path)**

**[SUB-LOOP checks remaining packages for constraints, if none, SUB-LOOP is ignored]**

**for locationNode in truck\_location\_list:**

**for packages in truck\_package\_list:**

**if package address matches locationNode AND package has (time) constraint:**

**set that locationNode to be current\_location**

**Time-Complexity is O(N2) = O(N) + O(N) + O(N2) + O(1) + O(1) + O(1) + O(N2)**

**Total Time Complexity is O(N2)**

**A. + B. + C. + D. = O(1) + O(N2) + O(1) + O(N2) = O(N2)**

Below is a table of methods and their space and time complexities for each file in the program:

**HashTable.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| \_\_init\_\_ | 4 | O(1) | O(1) |
| \_get\_hash | 10 | O(1) | O(1) |
| add | 17 | O(N) | O(N) |
| get | 35 | O(N) | O(N) |
| delete | 46 | O(N) | O(N) |
| update | 58 | O(1) | O(N) |
|  |  |  |  |
| **Total** |  | 3N + 3  = O(N) | 4N + 2  = O(N) |

**CSVPackages.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| None (create truck objects) | 10 | O(1) | O(1) |
| None (set times) | 16 | O(1) | O(1) |
| None (read packages.csv) | 24 | O(1) | O(1) |
| None (create package objects) | 51 | O(N) | O(N) |
| None (set package parameters) | 90 | O(N) | O(N) |
| None (assign package objects) | 101, 109, 117 | O(N) | O(N) |
| None (read DistanceData.csv) | 150 | O(1) | O(1) |
| None (read Locations.csv) | 158, 275 | O(1) | O(1) |
| None (create LocationNode objects) | 170, 284 | O(N) | O(N) |
| get\_truck\_locations | 188, 223, 301 | O(N2) | O(N2) |
| None (create\_edges\_for\_truck\_location\_list) | 229, 312 | O(N2) | O(N2) |
| None (optimize\_truck\_location\_list) | 261, 322 | O(N2) | O(N2) |
| get\_hash\_map | 326 | O(1) | O(1) |
|  |  |  |  |
| **Total** |  | 2N2 + 4N + 5  = O(N2) | 2N2 + 4N + 5  = O(N2) |

**DistanceChecker.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| None (read DistanceData.csv) | 4 | O(1) | O(1) |
| None (read Locations.csv) | 10 | O(1) | O(1) |
| current\_distance | 17 | O(1) | O(1) |
| total\_distance | 26 | (O1) | O(1) |
|  |  |  |  |
| **Total** |  | 4 = O(1) | 4 = O(1) |

**Main.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| None (start) | 16 | O(1) | O(N) |
| None (‘time value’) | 26 | O(N) | O(N) |
| None (‘search’) | 90 | O(1) | O(1) |
|  |  |  |  |
| **Total** |  | N + 2  = O(N) | 2N + 1  = O(N) |

**Truck.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| create\_edges\_for\_truck\_location\_list | 30 | O(N2) | O(N2) |
| optimize\_truck\_location\_list | 66 | O(N2) | O(N2) |
| None (while truck\_location\_list > 1) | 93 | O(N2) | O(N2) |
| None (calculate current time) | 127 | O(N) | O(N) |
| None (deliver package) | 137 | O(N2) | O(N2) |
| None (remove edge) | 156 | O(N2) | O(N2) |
| None (set current\_vertex) | 177 | O(1) | O(1) |
| None (check for package constraints) | 190 | O(N2) | O(N2) |
| None (deliver final package) | 215 | O(N2) | O(N2) |
| None (back to HUB) | 235 | O(1) | O(1) |
| None (calculate total trip time) | 251 | O(N) | O(N) |
| current\_distance | 272 | O(N) | O(N) |
|  |  |  |  |
| **Total** |  | 7N2 + 3N + 2 = O(N2) | 7N2 + 3N + 2 = O(N2) |

Most of the methods outside the main algorithm (contained in Truck.py) are linear or constant space and time complexity. The user would experience a fairly fast program to access package information and delivery status in real time. The main algorithm however is mostly O(N2). This could create a scalability problem as more and more packages or constraints are added, without adding more trucks or drivers or further optimizing the code. Ultimately, there are only so many hours in a workday and with the trucks limited to 18 mph, the maximum number of packages deliverable with 2 drivers would be approached very quickly, even if the algorithm could accommodate more.

**Advantages of the Main Algorithm**

The main algorithm meets all the requirements and performs all of the needed functions. Packages are delivered with approximately 104 miles of distance travelled in just under 4 hours. The algorithm does allow for easy modification and by using truck, location and package objects, any level of customization is possible without much extra work.

The algorithm delivers any package with a time deadline first and then proceeds to deliver the rest of the packages in the shortest path method. It is a greedy algorithm that uses Dijkstra’s Shortest Path algorithm. The graph of available locations to travel to is adjusted (locations and edges are removed) during each iteration. This ensures that each location is only visited once.

A great feature of this algorithm is that it provides the time that each package was delivered. It also has the ability to display which truck any package is on. This information can be vital to the user if he or she needs to contact a driver or customer about a package while the driver is in route. The user will know if and when the package has been delivered without anything from the driver. This feature could also be used to provide customers with estimated delivery times. The algorithm is scalable with more packages, but more optimization would be needed beyond more than perhaps 100 packages with only two drivers.

**Programming Models**

The programming model for this application is limited solely to the local machine it is running on. It is written in Python 3.7 and executed in PyCharm. There are no communication, network or bandwidth protocols as there are no external connections required for or compatible with the application in its current state. All data exchange is implemented at the local machine level by the user. CSV files for locations, distances or package information could be modified at a single location and distributed out, but again, the user would need to manually update the application files to accommodate the new information.

**Other Solutions**

Another algorithmic approach available is the “heuristic algorithm”:

A heuristic algorithm is an algorithm that quickly determines a near optimal or approximate solution . . . Such an algorithm uses the heuristic of choosing the highest [or lowest] value item, without considering the impact on the remaining choices. While the algorithm's simple choices are aimed at optimizing the total value, the final result may not be optimal. (zyBooks, 3.1)

A heuristic solution to the [Company] Parcel Service problem could be to simply always choose the next closest location without regard to constraints or later paths. Run the algorithm a certain number of times and find the shortest route that still meets the prescribed conditions. An advantage of this method is its simplicity and speed. It may not deliver the best solution, but it will deliver a good solution quickly and trades speed/scalability for the absolute best solution. The primary disadvantage is that the optimal solution is never known and as complexity grows, more and more iterations must be run to find a better and better solution. This paper’s proposed solution delivers the optimal solution each time and guarantees that time deadlines are met every time. It only has to be run once.

Another option would be to create a Binary Search Tree structure. Each node would represent a location and each node would contain an adjacency list of edges to other locations. The algorithm would traverse recursively to each location to deliver packages and search that node’s edge\_list for the next closest node. This method may be faster as many BST operations are

O(N log N) where in this paper’s proposed solution, lists of objects are used heavily, requiring many O(N2) operations. While a BST option may find the shortest path faster and more efficiently, it would have to search the entire tree and each node’s edge\_list with every iteration, where with this paper’s proposed solution, the truck’s location\_list and number of available edges gets smaller and smaller with every iteration.

**Data Structures**

The main data structure used throughout the program were lists of objects or lists of lists. This data structure was chosen for its flexibility. Linked lists are dynamic in that they can change in size throughout a recursive method call. There is some overhead involved with retrieving items from a list as the entire list has to be searched. However, lists do allow for very specific searching criteria which can be more difficult with other data structures. Since lists can contain objects, objects can be modified inside the list without changing the list structure. Since this program relies heavily on lists of objects, object creation and destruction, lists provide a dynamic and memory efficient way to handle these objects. It does lose time efficiency as a result however, as searching a list requires O(N) time.

A hashtable was also used to store package objects. The objects can be adjusted within the main algorithm (delivery time, delivery status, etc.) and those changes are reflected in the hashtable in real time. The hashtable provides very efficient search and retrieval operations for user experience.

Arrays could be used to store locations, edges or packages. However, this would be difficult to implement with Dijkstra’s shortest path algorithm. Once a location or edge was located, it would then need to be removed. Every iteration an item is removed and the index of all other items in the array changes. Lists do not have this problem. It can also be more difficult to search for and retrieve a particular object.attribute\_value of an object within an array. Without knowing the index of the item you are looking for, the array retrieval method loses its speed and becomes comparable to a list at O(N) for sequential search.

A Binary Search Tree could also be used as a primary data structure to store locations, edges or packages. A BST could make pre-sorting packages simpler and package retrieval on a truck faster than using lists alone.

**Review**

The proposed algorithm visits each node only once. This may not always be the optimal solution and that is something that may have drastically affected the outcome. Time and distance travelled could be decreased further by optimizing the sub-loop of packages with delivery time deadlines. The algorithm does not check for the closest location within the sub-loop, it only checks if a location has a package with a delivery deadline and visits them all until none remain. Additionally, the sorting of packages prior to loading them on the trucks is done in a way only to meet the basic requirements. Sorting could be optimized prior to loading to decrease time and distance travelled.

Overall the software is efficient, readable and well commented for the user experience with most operations occurring in O(N) or O(N2) time. Since the application makes extensive use of object-oriented programming, the codebase is easily maintainable. Despite the number of lines of code, a single truck route can be completed in 3 method calls with all packages delivered guaranteed on time every time. What it may lack in memory and computational efficiency it makes up for in ease of use and reliability.

**Sources**

Learn.zybooks.com. (n.d.). *zyBooks*. [online] Available at: https://learn.zybooks.com/zybook/C950AY20182019/chapter/3/section/1 [Accessed 18 Jan. 2020].