Core Algorithm Overview

**Provided Problem:**

The purpose of this project is to use Python 3 to determine the best distribution route for the Western Governors University Parcel Service. The company has three trucks and two drivers assigned to deliver forty packages. The packages will come with a multitude of constraints such as the possibility of the package being delayed, a package having a wrong address, or packages that are required to be on a specific truck. To solve this problem my program must be designed to load the trucks efficiently based on the data and constraints that are provided. The solution I selected is to implement Dijkstra's Shortest Path Algorithm to find an optimal route to deliver each package. This algorithm is optimal because while it is not fast, it will always find the shortest path. This document will overview the reasoning behind the choices made and explore the specifics of the different components of the algorithm.

**Algorithm Overview:**

Dijkstra’s algorithm is executed by these steps:

1. The current time, distance, truck number, and a bool determining whether or not the truck needs to return for delays are passed in to the method;
2. The algorithm checks which truck for which the route is being calculated and assigns the selected\_truck reference appropriately. It then checks if the selected truck has packages on board, and if not, it returns the current distance and time. This is the best case scenario.
3. The algorithm then checks whether the available destinations have been setup, and if not, it sets up the available destinations as well as the distance graph. Only one check is needed because both will always be setup and deallocated together.
4. Next a for loop will convert all packages that have been loaded in the truck to destination points that the algorithm can use
5. Dijkstra’s algorithm will then calculate the distance from the current point(the main hub by default since the truck will always return to the hub before picking up more packages)
6. The while loop executing the calculation will then first determine whether it needs to stop delivering packages and return to receive delayed ones. This only happens if the return for delayed packages bool is true and if the current time is past or equal to the time it would need to return. If these values are not true the truck will then greedily choose the package with the closest destination with the shortest time left before it is late and deliver it. Distance traveled and time traveled are then added to the total distance and time.
7. After delivering all packages or being interrupted to pick up delayed packages, the truck will automatically add the time and distance traveled from the current point back to the main hub.
8. Lastly, the algorithm will return the new current time, and total distance traveled.

**Advantages of chosen Algorithm**

This algorithm performs all the required functions for the project and delivers the packages within the range of 140 miles. The main advantage of this algorithm is that it will always find the optimal path. Another massive advantage of my algorithm is that it can find the optimal path even if it not one of the paths provided. For example, if path A to path C is 50, but path A to B is 20 and B to C is 20; The algorithm will recognize this and choose bath A-B-C instead of A-C. Additionally, its use of an adjacency list makes it far more compatible with future changes and additions.

Another algorithm I could have used to optimize the packages would be a simple greedy algorithm. The advantage of using a normal greedy approach is that it would run faster since there is much less processing needing to be done. While this algorithm may have been faster, it won’t necessarily always find the optimal path as it is only comparing the provided distances and not an alternative path. A second algorithm which I could have implemented is a Dynamic Algorithm. A dynamic approach would break the package lists into smaller lists to compare. This would drastically speed up the execution of the program, but at the cost of a much higher space complexity. Not only would it raise the space complexity, but it also has the potential of being much harder to debug by adding more instances of data being moved around.

**Algorithm Pseudo-code**

The space-time complexity of this algorithm has a worst case runtime of O(N^3) and a best case runtime of O(1). The worst case is practically guaranteed because the best case can only happen when the list of packages being loaded onto a truck is empty. Below I have provided the pseudo-code for the algorithm:

**Dijkstra’s Algorithm**

A. The algorithm must first take the following parameters. The time complexity for this section is O(1)

**1. current\_distance**

**2. current\_time**

**3. truck\_num (variable representing which of the available truck objects to use)**

**4. return\_for\_delays (variable representing whether or not the truck will need to return for delayed packages)**

B. Next it will assign a reference to the correct truck object based on truck\_num. The time complexity for this section is O(1)

**if truck\_num equals 1:**

**assign reference selected\_truck to truck\_one**

**else:**

**assign reference selected\_truck to truck\_two**

C. After that it will check whether the truck is empty or not. It will skip execution and return the current time and distance if empty and execute the algorithm if not. The time complexity for this section is O(1)

**if selected\_truck.packages\_on\_truck is not Empty:**

**execute the algorithm**

**else:**

**return current\_distance, current\_time**

D. Following that my algorithm will setup available\_destinations and the distance graph for those destinations if it hasn’t been done. This is done by only checking available destinations; We only need to check whether available\_destinations has been populated because testing shows both lists will always deallocate and be created at the same time. Time Complexity O(N^2)

**if destinations\_available is Empty:**

**setup\_available\_destinations() (O(N))**

**setup\_graph() (O(N^2))**

E. After assigning some needed variables, the algorithm will then create a destination for every package on the truck and assign the package to the destination. I use enumerate because a reference the current index is needed to assign the package to the proper destination. I use references to available\_destination to solve data integrity issues within the algorithm when comparing destination objects. The time complexity for this section is O(N)

**truck\_distance = 0**

**truck\_destinations = []**

**current\_time\_for\_truck = current\_time**

**for index, package in enumerate(selected\_truck.packages\_on\_truck):**

**Assign destinations from destinations available based on the address index provided by find\_index**

**assign the destination added with the correlating package**

F. The algorithm will then calculate the distance from the current point to every point using Dijkstra's algorithm. This algorithm takes an adjacency graph(g) and the starting point that the distances will be made from. The time complexity for this section is O(N^2)

**unvisited\_queue = []**

**for each destination in g.adjancency\_list:**

**assign current\_distance with infinity**

**append destination to the unvisited\_queue list**

**starting\_destination.distance = 0**

**while unvisited\_queue is not empty:**

**current\_index = 0**

**for I in range 1 to unvisited\_queue’s length:**

**if unvisited\_queue at index I has a distance less than at the destination in the current\_index:**

**asssign current\_index to I**

**pop the value from unvisited\_queue at index current\_index and assign to current destination**

**for adjancent\_destination in g.adjacency\_list:**

**edge\_weight = g.edge\_weight(from current destination to adj\_destination)**

**alternative\_path\_distance = current\_destination\_distance + edge\_weight**

**if alternative\_path\_distance < adj\_destination.distance:**

**adj\_destination.distance = alternative\_path\_distance**

**adj\_destination.prev = current\_destination**

G. Next the algorithm enters the main while loop. First the while loop checks if it needs to return for delays and if the current time is past the time to return. If these checks are both true the truck will exit the while loop and automatically return to the hub to retrieve packages. The time complexity for this section is O(1)

**if current\_time\_for\_truck is less than 9:05 and return for delays is true:**

**break the while loop**

H. Next the algorithm will find the closest destination among the packages with the least time left until they are overdue. The time complexity for this section is O(N)

**current\_destination = truck\_destinations[0]**

**for destination in truck\_destinations:**

**if the destination is not the current destination:**

**if the destination deadline is less than the current destination’s deadline:**

**assign current destination with destination**

**if the destination deadline is less than or equal to the current destination and the destination distance is less than the current distance:**

**assign current destination with destination**

**I.** Lastly this while loop will calculate the distance traveled, and the time traveled for and add it to the current time and total distance. It will also unload packages being delivered to that destination from the truc.k. Lastly, it will run dijsktra’s algroithm with the point traveled to as the starting point. Time complexity of O(N^2).

**current\_time\_for\_truck = current\_time\_for\_truck + distance\_travel\_time(current\_destination.distance)**

**truck\_distance = truck\_distance + current\_destination.distance**

**selected\_truck.unload\_packages(current\_destination.package.address, current\_time\_for\_truck)**

**truck\_destinations[:] = [x for x in truck\_destinations if x.index != current\_destination.index]**

**dijkstra\_shortest\_path(distance\_graph, current\_destination)**

**J.** When the while loop completes, the distance and time traveled from the current point to the main hub is added to the current\_time\_for\_truck and truck\_distance automatically because the truck will always return to the main hub. Lastly, it adds the time and distance to the total and returns those values. Time Complexity of O(1)

**truck\_distance += destinations\_available[0].distance**

**current\_time\_for\_truck = current\_time\_for\_truck + distance\_travel\_time(current\_destination.distance)**

**current\_time = current\_time\_for\_truck**

**current\_distance += truck\_distance**

**return current\_distance, current\_time**

**Data Structure Chosen**

The primary data structure I implemented throughout the application is combination of lists and an adjacency graph for calculation portion. I choose this data structure because it is very easy to work with and our hash-table that holds all the package info can be easily converted into lists for processing. All the info that needs to be saved and not processed is stored in a hash table for quick and easy access. All data being processed is stored in lists rather than a hash-table since it will mostly be looped through rather than individually accessed. The biggest advantage for me personally was it allowed me to debug the program much more quickly; This resulted in a far shorter development time. One weakness I encountered while coding this project using lists is that in the very few cases of needing to find a specific package it makes the time complexity much higher. The use of an adjacency graph during the distance calculation portion streamlines the development process as I’ve already implemented Dijkstra’s algorithm with a graph before in school. It also allows me to group package destinations together by adjacency lowering the runtime of the calculation.

There are many different data structures that I could have used in place of my own that still meet the project requirements. I think the best alternative would be a Binary Search Tree. A BST would cut the time complexity in half for every instance in the program that loops through the lists, but at the cost of a far greater space-complexity. Another data structure I could have implemented would be to only use the Hash Table for the packages. A huge advantage of using only a single Hash Table would be that I wouldn’t have to keep track of as many data elements when debugging. This would also speed up searching for individual packages in the few cases it was needed. A major disadvantage of using only a hash table is it has double the space and time complexity when looping through it because two for loops are needed.

Below is a breakdown of the worst-case space and time complexity for each file in the my application:

**DistanceCalculator.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| Destination.\_\_\_init\_\_\_ | 7 | O(1) | O(1) |
| Graph.\_\_\_init\_\_\_ | 28 | O(1) | O(1) |
| add\_delivery | 23 | O(1) | O(1) |
| add\_street | 26 | O(1) | O(N) |
| setup\_available\_destinations | 48 | O(N) | O(N) |
| find\_index | 62 | O(1) | O(1) |
| find\_address | 71 | O(N) | O(N) |
| find\_distance | 80 | O(1) | O(1) |
| distance\_travel\_time | 89 | O(1) | O(1) |
| load\_truck\_one | 96 | O(N) | O(N) |
| load\_truck\_two | 110 | O(N) | O(N) |
| setup\_graph | 122 | O(N^2) | O(N^2) |
| deliever\_packages | 137 | O(N^3) | O(N^3) |
| dijkstra\_shortest\_path | 201 | O(N^2) | O(N^2) |
| **Total** |  | N^3 + 2N^2 + 4N + 7 = O(N^3) | N^3 + 2N^2 + 4N + 7 = O(N^3) |

**sorter.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| sort | 15 | O(N^2) | O(N^2) |
| **Total** |  | O(N^2) | O(N^2) |

**database.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| DeliveryStatus.\_\_str\_\_ | 14 | O(1) | O(1) |
| Package.\_\_init\_\_ | 20 | O(1) | O(1) |
| Package.\_\_str\_\_\_ | 54 | O(1) | O(1) |
| set\_delievery\_status | 80 | O(1) | O(1) |
| lookup\_by\_time | 72 | O(N^2) | O(N^2) |
| lookup\_packages | 100 | O(N^2) | O(N^2) |
| Truck.\_\_\_init\_\_\_ | 140 | O(1) | O(N) |
| load\_package | 146 | O(1) | O(1) |
| unload\_packages | 165 | O(N) | O(N) |
| **Total** |  | 2N^2 + N + 6 = O(N^2) | 2N^2 + N + 6 = O(N^2) |

**CSVParser.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| load\_file | 9 | O(N) | O(N) |
| **Total** |  | N = O(N) | N = O(N) |

**HashTable.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| \_\_\_init\_\_\_ | 5 | O(1) | O(1) |
| create\_hash | 12 | O(1) | O(1) |
| insert | 18 | O(N) | O(N) |
| search | 29 | O(N) | O(N) |
| remove | 39 | O(N) | O(N) |
| update | 51 | O(N) | O(N) |
| **Total** |  | 4N + 2 = O(4N) | 4N + 2 = O(4N) |

**main.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Line Number** | **Space Complexity** | **Time Complexity** |
| create\_package\_from\_input | 13 | O(1) | O(1) |
| handle\_user\_input | 28 | O(N) | O(N) |
| None | 94-111 | O(N) | O(N) |
| None | 112 | O(N^2) | O(N^2) |
| None | 113-135 | O(N^4) | O(N^4) |
| None | 136-155 | O(N^3) | O(N^3) |
| None | 156-end | O(N^2) | O(N^2) |
| **Total** |  | N^4 + N^3 + 2N^2 + 1 = O(N^4) | N^4 + N^3 + 2N^2 + 1 = O(N^4) |

**Program efficiency and maintainability with scale**

The ability to scale with the number of cities is one of the weaker aspects of my program, but it still meets all the scaling requirements. This program has been heavily optimized for the area and scenario provided and little testing has been done on alternative data sets. There are several changes required within the program for effective scaling: The sorter.py file will need to be completely rewritten because it is designed so that it can be constantly changed as the type of constraints change; The job of this file is to properly sort the packages so the algorithm can run at it’s most efficient level; Lastly, the sorting algorithm(sorter.py) will need to be changed if there are new constraints added. However, if the type of constraints are the same, my program is prepared to scale with higher number of packages. The program loops through all provided packages and automatically adjusts the hash-table size to deal with a larger number of packages. The program is also easy to maintain since all major parts of the program are broken into different files and functions.My program will easily scale with the number of trucks because the only change required is to add additional truck objects and an additional check for those trucks’ ID numbers in the algorithm.

The theoretical efficiency of my program isn’t great with the time and space complexity of main.py alone reaching O(N^4); I decided to sacrifice efficiency to insure that the program always returns an optimal result. In practice this will not be an issue because the program only needs to be ran once a day, and the program only needs to keep up with how long it takes to deliver each load of packages; Any modern computer should be able to execute a million packages in less than an hour. The total time and space complexity of the algorithm itself is O(N^3), but this won’t cause any issues since the truck has a package limit of sixteen.

**What I would do differently**

If I was to do this project again I would put more focus into the package sorting system at the start and optimize the algorithm around how the packages are provided to the algorithm instead of the other way around. I would also consider something like a heuristic approach, or a some type of balanced tree to bring my algorithm efficiency down to N^2 or lower. I also would have done more testing on specific features of my program before implementation; I didn’t do this because I was overly confident in my pseudocode being correct. Lastly, I wouldn’t have used Dijkstra's algorithm, even if I still wanted to use a greedy algorithm to approach this problem. Dijkstra’s is far more reliable than a standard greedy algorithm in finding the shortest path, but in this type of application the difference is negligible.

**Sources**

The only resource I used was the Zybooks material provided by WGU for general information. Nothing in my project was paraphrased or quoted.

Lysecky, Roman, and Frank Vahid. *ZyBooks*, learn.zybooks.com/zybook/WGUC950AY20182019.

