Jacob Alo

C950

4/15/2019

For this paper, I will address items in the rubric specifically to ease perusal and grading as much as I can. To reduce redundancy and support that ease of perusal however, some subsections will not be explicitly indicated if the relevant content was included in some preceding paragraph.

*Section 1: Programming/Coding*

*A. Identify the algorithm used to create the program and delivery the packages.*

For this program, I selected Dijkstra's shortest path algorithm as the basis for my delivery algorithm, modified of course to meet the particular use case requirements.

*B. Write a core algorithm overview.*

Relevant psuedo-code will be provided in bolded text during this overview.

Dijkstra's employs a graph structure to build trees from a start node to other nodes in the graph, and recording in those nodes a distance value that represents the shortest tree traversed to arrive there. It is provided with two arguments, which are the graph itself and the start node from which all other distances are calculated.

**dijkstras\_shortest\_path(graph, start\_node):**

The graph employed requires two components:

* Vertex nodes to represent locations, hold distance values, and hold predecessor values for retaining the shortest path built from the start node.
* Weighted edges that connect those nodes where appropriate, with values appropriate for their distance.

When the algorithm is implemented, it first initializes the distance value of the start node to itself as zero, and intializes all other node distances as *infinite*. This represents cleaning the graph before each use, and is implemented as a separate method in my use case. After cleaning, each node is placed into a queue of unvisited nodes.

**for each node (curr\_node) in graph:**

**curr\_node[distance] = infinity // Initial value**

**curr\_node[predecessor\_node] = 0 // Initial value**

**push curr\_node to unvisited\_queue**

**start\_node[distance] = 0 // Initial node has distance of 0 from self**

After this has been done, the algorithm will loop through the queue, popping each node with the shortest distance and setting it as the current node. For each node adjacent to the current node, the algorithm begins computing the path distance from the start node to that adjacent node using the edge weights provided. If that path's distance is shorter than the adjacent node's distance value, then a shorter path has been found. The distance value of the adjacent node is updated to reflect the more efficient path discovery, and it's predecessor node value is pointed to the current node. This process is repeated until the queue is empty, and what remains will be a tree of efficient paths from each node to the start node.

**while (unvisited\_queue is not empty)**

**curr\_node = MinPop unvisited\_queue // Visit node with min distance**

**for each node (adj\_node) adjacent to curr\_node:**

**edge\_weight = edge weight between curr\_node and adj\_node**

**alt\_path = current\_node[distance] + edge\_weight**

**if alt\_path < adj\_node[distance] // If shorter path found**

**adj\_node[distance] = alt\_path // Update distance**

**adj\_node[predecessor\_node] = curr\_node // Update predecessor**

Because the algorithm visits each node in both the inner and outer loop, its runtime is O(N^2). In my implementation, there are only 27 nodes, so this runtime is trivial, but for very large delivery systems this runtime could strain computer resources. A more efficient implementation of the unvisited\_queue as a fast heap data structure will reduce the run time of this algorithm to O(NlogN) and is advised.

This is the basic logic behind Dijkstra's shortest path algorithm. The efficient path is calculated above, then the path is traversed starting from a specified end node and working backwards to the start node, with the distance traveled being the end node's distance value. For my use case, the delivery algorithm that traversed the path calculated by Dijkstra's had to be adapted to its particular needs.

As each package is loaded onto the truck (represented by an object), the hub it is to be delivered to is added to one of two lists tracked as truck attributes. These lists are called priority\_current\_packages and regular\_current\_packages. These lists are implemented in my delivery algorithm to ensure time sensitive packages are delivered on time. Because the lists are truck attributes, they do not need to be provided to the delivery algorithm as arguments. Additionally, the truck has attributes called priority\_current\_route and regular\_current\_route. These are also both lists.

First, the algorithm checks that there are packages to be delivered. If there are, it builds a list of hubs to be visited by checking those packages and populating the priority\_current\_route and regular\_current\_route attribute lists.

**deliver\_route:**

**while neither pack\_lists[ pri or reg ] are empty: // Check both package lists**

**for each package in pack\_lists[ pri or reg ]:**

**// Add package delivery hub to appropriate priority list**

**add package[hub] to route\_list[package priority]**

After the route lists are built, the current location of the truck is removed from the route lists, and a clean call of dijkstras\_shortest\_path is performed to update the route distances from the current location. This is implemented as separate methods in my implementation.

**if curr\_location is in route\_list[ pri or reg ]:**

**remove route\_list[current\_location]**

**call dijkstras\_shortest\_path(graph, curr\_location) // Update graph for new location**

After route management has been handled, the algorithm checks if the priority route list is empty. If there are hubs still remaining in that list, then there are still priority packages to be delivered. It implements a greedy algorithm that selects from that priority route list the nearest hub as its next delivery location. If there are no hubs in that priority list, then all priority packages have been delivered and it searches for the nearest hub in the regular route list and selects it as the next delivery.

**if route\_list[priority] is not empty:**

**next\_hub = minimum distance in route\_list[priority] // Find closest priority delivery**

**else:**

**next\_hub = minimum distance in route\_list[regular] // Find closest regular delivery**

With the next hub selected, the distance value associated with next\_hub is added to the truck mileage attribute, and the internal clock on truck is updated to reflect the travel time to get there, given the truck speed of 18mph. The truck's location attribute is updated to next\_hub.

**truck[mileage] += next\_hub[distance]**

**truck[time] += next\_hub[distance] / 18mph // Time = distance/speed**

**truck[location] = next\_hub**

Having arrived at the new hub, the truck delivers its packages. To do so, the algorithm creates four temporary lists. The first list is called priority\_new\_packages\_list, and it is a list of priority packages to be kept in the truck (not delivered). There is likewise a regular\_new\_packages\_list that does the same for regulat packages. The other two are priority and regular package lists that track the packages delivered at the hub (not retained in the truck). To discern which packages to deliver, the algorithm checks each package in its priority and regular package lists, and if the address in the package object's address attribute matches the current location, it is added to the list of packages to be delivered. If it does not match, it is added to the list of packages to be retained in the truck. Packages to be delivered are also appended to a global master list that tracks all packages that have been delivered. After this book keeping is handled, the truck's regular and priority lists of current packages are updated to reflect the contents of the temporary lists of packages retained, updating their values.

**pri\_new\_pack\_list = []**

**reg\_new\_pack\_list = []**

**pri\_delivered = []**

**reg\_delivered = []**

**for each package in each pack\_list[pri and reg]: // Check both package lists**

**if package address = current location: // Package is to be delivered here**

**add package to appropriate delivery list // Priority or regular**

**add package to global delivery list**

**else: // Package is to be kept in truck**

**add package to appropriate new package list // Priority or regular**

Once all packages have been delivered in this manner, the base case for the while loop will be met, and the algorithm will exit the loop. It then calls a return\_to\_wgu method which calls dijkstras to find the distance to the wgu package depot. It updates the time and mileage values as above, then updates the truck location to the wgu depot. The runtime of this delivery algorithm is superceded by its call to dijkstras\_shortest\_path, and so runs in the O(N^2) time that dijkstra's runs in.

My delivery algorithm is efficient for the requirements it must fulfill. It is limited by the day to day realities of the package delivery center, and so is bounded by the use case it is implemented for. Use case efficiency can be improved by integration with package depot inventory management software that efficiently loads trucks with deliveries based on depot needs (which packages will be late, which need priority delivery, trucks being brought online as new drivers clock in, etc). This algorithm can be easily updated and expanded to account for these efficiencies, and to track other package or truck details as the need arises.

*Explain how your data structure includes the relationship between the data points you are storing.*

Each package, hub, and truck is an object with associated attributes to track information. I employed a direct hash for storing information about packages and hubs. Each package has a package ID number from 1-40, and that ID number is an attribute for each package object. When reading the package CSV file in to instantiate class objects, I also populated a master\_package\_list that uses that package ID direct hash as the index value of each package in it for references. I followed this same model with each delivery hub, but because no hub ID number was provided, I assigned them myself. The direct has assignment was somewhat arbitrary. Each delivery hub is assigned a hub ID number ranging from 0-26. These numbers were assigned automatically while reading in the CSV file with information about hub distances and are assigned in order based on the row in that file. Each hub has a hub\_id attribute similar to how packages have, and there is a master\_hubs\_list using the direct hash as the index for each hub.

*Section 2: Annotations*

*Describe two strengths of the algorithm you chose.*

Two strengths of my algorithm are simplicity and efficiency. Dijkstra's shortest path is an efficient industry standard algorithm that is well understood. Anybody with a relevant professional background will be able to read my code and quickly understand how and why it works the way it does. This makes it easy to maintain and update as needs require.

*Verify the algorithm you chose meets criterion.*

After several refactors to fine tune mileage efficiency, all packages are delivered on time, all special instructions are fulfilled, and the total mileage is only 120 miles, which I am told by my instructor is an efficient value. This is verified in the provided screenshots.

*Identify two other algorithms that could be used and would have met the criteria*

In researching approaches to adopt, one solution I discovered involved implementing innovations from genetic coding to fine tune Dijkstra's algorithm. This method should have, in theory, produced a similar result, and may have even reduced the mileage further. However, as I was instructed not to include any external libraries, this solution was not appropriate. Another approach that I could have adopted would have been a simple greedy algorithm run on the distance values provided in the CSV. I would have loaded up the trucks by zip coded or some other condition to reduce potential variance, then just navigated to the closest distance for each delivery. This likely could have been tuned to meet package requirements, but would have represented an inferior solution to running Dijkstra's on those distance values to find more efficient routes, and so was rejected in favor of my choice.

*What would you do differently if you did this project again?*

This project was a great learning tool for me. I had absolutely zero experience with Python prior to this, and almost zero programming experience as I do not come from any kind of technical background. Consequently, early on in the project, I made decisions that I later came to regret. If I were able to do this project again, with more knowledge than I had at first, I would firstly invest my time differently. My initial approach was ignorant of how intensely individualized each algorithm must be. I wasted a lot of time trying to find examples of people doing similar overall projects that I could extrapolate from until I realized the better way to learn from others here was to study perhaps unrelated projects line by line to see how more abstract coding concepts develop. I also would have probably implemented the class structures a bit differently. When I first started the project I had only a very vague idea of how class objects worked, and so when I was trying to instantiate objects from the CSV reader, I wasted a lot of time trying to reference values that didn't exist, or navigate attributes that didn't work the way I thought they did. I regard this as an indicator that I have successfully grown in my understanding for having done this project, however.

*Identify two other data structures that can meet the same criteria*

I employed class objects, graphs, vertexes, and direct hash lists as my primary data structures. They worked well for my use case, and I do not regret those choices. However, there are some other options that would have been available to me. Instead of class objects, I could have used a variety of nested dictionaries to track associated values. I did not, because this would have introduced unnecessary complication, and class objects essentially do the same thing underneath the class abstraction. Another structure I could have used would have been to implement binary trees to store and find things like distance values when delivering packages. I briefly considered exploring something in this direction, but ultimately rejected it because it did not integrate as smoothly with other elements of my program, and again seemed to introduce an unnecessary layer of complexity for little real payoff.

*Acknowledge sources*

Other than the Zybooks course material, there are no academic sources to cite and reference. I consulted with my instructor, Eric Christianson (thank you so much for our many meetings!), and two different friends (thank you Justin for your help with understanding how csv readers and the enumerate method work, and Josh for listening patiently while I worked my way through algorithm logic) on a few occasions.

Thank you for taking the time to read this.