Jaxon Adams

ID 011441603

jada900@wgu.edu

C950 Data Structures & Algorithms II

PA Task 2: WGUPS Routing Program Retrospective

F1 – Algorithm Strengths

For this project, I opted to use the “nearest neighbor” algorithm. This algorithm had many strengths which made it an appealing choice. The first was the algorithm’s simplicity – the pseudocode for the algorithm is concise and easy to understand, and thus easy to implement in Python. The overall concept is very simple as well – it makes sense that always traveling to the next nearest location would lead to a short delivery route, although it may not be the most optimal.

Another major strength of this algorithm is its time-space complexity. With a time complexity of O(n^2) and a space complexity of O(n), it’s rather quick compared to an algorithm which might produce a more optimized route. The algorithm’s decent scalability and the sufficiently optimal routes it produces make it an excellent choice for this project.

F2 – Algorithm Requirements Met

My implementation of the nearest neighbor algorithm is a greedy algorithm which finds a sufficiently optimized route for the trucks to deliver all packages. As seen in screenshot “partE.png” (included in the /screenshots directory of this project), all packages are delivered, and the total number of miles travelled is 113.6. This satisfies the requirement that no more than 140 miles can be travelled in the program. The algorithm runs in polynomial time (O(n^2)), so it is considered sufficiently optimized for this project. Additionally, all packages’ special instructions are followed, and they are delivered before their deadline (see screenshot “partE.png”).

F3 – Other Choices of Algorithm

An alternative algorithm which could be used for this project is Dijkstra’s algorithm. Dijkstra’s algorithm is an algorithm which determines the shortest distance between two points. One could use Dijkstra’s algorithm to find the distance between a truck’s current location and another address, dropping off packages at addresses visited along the way. Dijkstra’s algorithm would likely need to be executed several times to ensure all addresses are visited. This is different from the nearest neighbor algorithm in that Dijkstra’s algorithm operates on a graph, where any address can be used in the path to the target address. My implementation of the nearest neighbor algorithm operates on a list of addresses rather than a graph, and it builds up the truck’s route by visiting one address at a time.

Another algorithm which could be used for this project is the 2-opt (or pairwise exchange) algorithm. It would involve taking an initial route (for example, one generated by the nearest-neighbor algorithm) and repeatedly swapping two addresses in the route to see if it produces a more optimized path than the initial route. It would be different from the nearest neighbor algorithm in that it takes an initial route and searches for ways to improve it, where the nearest neighbor algorithm builds a new route from a set of addresses.

Both alternative algorithms listed above would provide sufficiently optimal routes to deliver all packages and would satisfy the requirements of this project.

G – What I’d Do Differently

The biggest change I’d make if I did this project again would be to make it easier to handle different addresses and packages. Currently, the program manually loads the trucks. It would be an interesting challenge to build out functionality for the program to choose which packages should go on which trucks. This would include parsing delivery deadlines and special instructions on each package and accounting for them.

Another change I’d make is that I’d simplify the logic around running the simulation. Currently, my program uses an event-loop and coroutines to allow for trucks to deliver packages concurrently. I think a simpler approach would be to simulate each truck’s deliveries one at a time, resetting the simulation time when the second truck leaves the hub. The solution I came up with certainly satisfies the requirements of the project, but I think simpler simulation logic would be acceptable and would have saved implementation time.

H – Data Structure Requirements Met

For my self-adjusting data structure, I implemented a hash table to store package data. It holds instances of a “Package” class to relate all package data, such as delivery deadline and package weight. The hash table accepts a package ID as the key for a new entry. It allows for efficient searching for packages, which was very helpful for this project. The hash table also has a “resize” method it uses to adjust its own size when the table grows too full, which helps with the program’s scalability.

H1 – Data Structure Alternatives

One other data structure that could meet the requirements of this project is a list which stores instances of a “Package” class. A list would be self-adjusting in that its size can increase to accommodate more packages. Furthermore, one could introduce more self-adjusting functionality by having the list move packages which are searched for frequently to the front of the list. This would increase the speed of a linear search on the list for packages which are searched for frequently. A list is different from a hash table in that a list doesn’t store key-value pairs, and to find a specific package one would have to perform a search algorithm on the list (e.g. a linear search or binary search algorithm).

Another alternative to the hash table would be a graph data structure, where each edge is a “Package” instance and each vertex is the distance from one package’s delivery address to another. In addition to meeting the requirements for this project, using a graph to store package data would make it easier to lookup the distance from one address to another, simplifying the core algorithm’s logic. A graph is different from a hash table in that rather than storing key-value pairs, it links elements with vertexes representing their relationships.

I – Sources

N/A – No resources were used in preparation for this program or retrospective.