C950 Task-2 WGUPS Write-Up

(Task-2: The implementation phase of the WGUPS Routing Program).

(Zip your source code and upload it with this file)

Brandon Nguyen

ID #003964281

WGU Email: bnguy80@wgu.edu

Date: 8/11/23

C950 Data Structures and Algorithms II

**Stated Problem:**

The objective of this program is to establish efficient routes and distribution plans for package deliveries within the Western Governor University Parcel Service (WGUPS), utilizing Python 3.11. The task involves handling 40 packages scheduled for delivery by the end of the day, each with unique delivery deadlines. Challenges include limit of three trucks and two available drivers for delivery, accounting for delays in package arrival, adhering to specific truck loading constraints, and accommodating special requirements like grouped package deliveries and designated trucks for certain packages. To address these challenges a systematic approach is proposed.

**Proposed Solution and Approach:**

Our approach consists of several strategic steps aimed at optimizing package deliveries for the Western Governor University Parcel Service (WGUPS):

1. **Truck Prioritization:** The three available trucks will be prioritized based on package delivery deadlines. High priority(truck1) for packages with a delivery deadline of 9:00 AM, medium priority(truck2) with delivery deadline of 10:30 AM, and low priority(truck3) for packages with deliver deadline by EOD which will be defined as 5: 00 PM. This prioritization will for the foundation for efficient route planning and distribution.

2**. Nearest Neighbor Greedy Algorithm:** The core of our approach lies in the implementation of the self-adjusting heuristic nearest neighbor algorithm. The nearest neighbor algorithm is used to create an initial delivery route for each truck. It starts from the hub (starting point) and iteratively selects the nearest package or destination to visit next. This process continues until all packages are assigned to the route.

The nearest neighbor algorithm is relatively simple and quick but may not always result in the most optimal route. In the context of the code, the nearest neighbor algorithm is used to create an initial route for each truck, which is then further optimized using the two-opt algorithm and Dijkstra's shortest path algorithm.

3. **Two-Opt Greedy Algorithm:** The two-opt algorithm is then applied to the initial routes created by the nearest neighbor algorithm. This algorithm attempts to improve the route by iteratively swapping pairs of edges to eliminate intersections and reduce the overall route distance. It helps refine the initial routes and make them more optimal.

After applying the two-opt algorithm to the initial routes, Dijkstra's algorithm is used once again to calculate the shortest paths for the refined routes. This ensures that the routes remain optimized even further after the two-opt optimization.

4. **Dijkstra's Shortest Path Greedy Algorithm:** Dijkstra's algorithm is a graph traversal algorithm used to find the shortest paths from a source vertex (hub) to all other vertices (package delivery locations) in a weighted graph. In the code, Dijkstra's algorithm is used to calculate the shortest route from the hub to each package's delivery location, considering the distances between vertices (locations).

Dijkstra's algorithm ensures that the calculated routes are truly optimized in terms of minimizing the distance traveled. This algorithm provides a more accurate measure of the shortest routes compared to the initial routes generated by the nearest neighbor algorithm and two-opt algorithm. The final total distance calculated.

**Core Algorithm Overview:**

The self-adjusting heurist algorithm is executed by doing the following:

1. For each truck, iterate through the packages, find nearest package to current vertex(address).

2. Selects the nearest package with the shortest distance to current vertex.

3. Moves onto nearest package's address, updating current vertex

4. This process continues until all packages are delivered, ensuring the truck travels to the nearest package at each step and return to hub.

**Space Complexity:**

The space complexity of the given nearest neighbor algorithm is O(n), where n represents the number of packages loaded onto the trucks.

**Time Complexity:**

The time complexity of this algorithm is O(n^2), where n is the number of packages on the truck. This is because for each package on the truck, there is an inner loop that iterates through the remaining packages to find the nearest package.

As the number of packages on the truck decreases with each iteration, the average number of iterations in the inner loop decreases as well, resulting in a total time complexity of O(n^2).

**Best- and Worst-Case Time Complexity:**

**Best Case:** The best-case time complexity of the nearest neighbor algorithm is O(n), where n is the number of packages on the truck. This occurs when the packages are already optimally ordered in a way that minimizes the distance traveled. For each package, the algorithm calculates the distance to all remaining packages and immediately finds the nearest one without needing to iterate through all the other packages. As a result, the algorithm completes in a linear time.

**Worst Case:** The worst-case time complexity of the algorithm is O(n^2), which happens when the packages are in an order that maximizes the distance traveled. In this case, the algorithm iterates through all the packages on the truck for each package, leading to a quadratic time complexity.

**REWORK------Dijkstra's Shortest Path Algorithm Overview:**

The greedy algorithm is executed by doing the following:

1. For each truck, find the final optimized routes for minimum distances traveled.

2. Utilize a min-heap as a priority queue to prioritize processing vertices with the shortest known distance. This queue is continually updated to ensure efficient shortest path determination during execution.

3. Iterate over neighbors of a vertex and calculate the distance. If the calculated distance is smaller than the current known distance, update the minimum distance.

4. The algorithm returns minimum distances for all addresses associated with each truck package.

**Space Complexity:**

**Priority Queue:** Our implementation a priority queue (min-heap) to manage vertices by their distances. The space complexity the priority queue is O(n), where n is the number of vertices.

**Best- and Worst-Case Time Complexity:**

**Priority Queue (Min-Heap) Operations:** The core operation in Dijkstra's algorithm involves extracting the minimum element from a priority queue (min-heap) and updating its neighbors. The number of times this operation is performed depends on the number of vertices and edges in the graph.

**Best Case**: O ((V + E) \* log V), where V is the number of vertices and E is the number of edges. In the best case, the priority queue operations may be more efficient due to the specific distribution of edge weights and vertex connections.

**Worst Case**: O ((V + E) \* log V), where V is the number of vertices and E is the number of edges. In the worst case, all vertices and edges need to be processed.

**Updating Neighbors:** In each iteration, the algorithm updates the distances of neighboring vertices. This operation involves checking the distances and possibly updating them.

**Best Case**: O(V), where V is the number of vertices. In the best case, only a few neighbors need to be updated.

**Worst Case**: O(V), where V is the number of vertices. In the worst case, all neighbors need to be updated.

Considering these factors, the overall time complexity of Dijkstra's algorithm in this context is O ((V + E) \* log V). As of now there are only 40 packages.

**Dijkstra's Algorithm Pseudocode:**

1. Takes two parameters graph object (associated packages with address acting as vertices and edge weights acting as distance to each address) and src the current location (by default always at hub), and route (the route the truck takes after using nearest-neighbor and two-opt algorithm)

2. Initializes 'distances' dictionary of all vertices in the graph to infinity that is in the route, except for the src, which is set to 0.

3. A min-heap represented as a list of tuples (distances, vertex), is created to prioritize processing of vertices with the shortest known distance first. Initially, the src and distance of 0 are added to the min-heap.

4. Enters while loop that continues until the min-heap is empty. In each iteration, the vertex with the smallest distance is popped from the min-heap. If the vertex has already been visited and its current distance is greater than the distance stored in 'distances' dictionary, then the iteration is skipped.

5. Popped vertex is marked as visited and its neighbors are iterated over. For each neighbor, the function calculates the distance from the source vertex to the neighbor by adding the weight of the edge connecting them to the current distance. If this calculated distance is smaller than the current known distance to the neighbor, the `distances` dictionary is updated with the new, smaller distance.

6. Finally, the updated distance and neighbor vertex are pushed into the min-heap, ensuring that the neighbor vertex will be processed later with its updated distance.

7. Once loop is finished, function 'distance' dictionary will contain the shortest distances for packages to be returned

**Worst Case Space-Time Complexity:**

1.Hash Map Class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 61 | Initialize an empty hash map | \_init\_\_ | O(N) | O(N) |
| 68 | Compute the hash bucket for a given key | get\_hash | O(1) | O(1) |
| 73 | Insert a new key-value pair into hash map | insert\_package | O(N) | O(N) |
| 98 | Retrieve value using a given key | get\_value\_from\_key | O(1) | O(N) |
| 110 | Retrieve key using given address | get\_address\_from\_key | O(1) | O(N) |
| 120 | Retrieve address using given key | get\_key\_from\_address | O(1) | O(N) |
| 127 | Retrieve the entire hash map | get\_hashmap | O(1) | O(1) |
| 130 | Retrieve packages in the hash map | get\_packages | O(1) | O(1) |
| 134 | Print details of all packages | print\_all\_packages | O(1) | O(N) |
| 150 | Update value for a given key | update\_value | O(1) | O(N) |
| 164 | Delete key-value pair using given key | delete\_key\_value | O(1) | O(N) |
| 181 | Load CSV data into the hash map | load\_hash\_map | O(N) | O(N) |
| 222 | Print details of all packages for testing | get\_hash\_map\_all | O(1) | O(N) |
| 228 | Check if all packages exist in the hash map | check\_all\_packages | O(1) | O(N) |
| Total |  |  | O(N) | O(N) |

2.HashMapEntry Class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 6 | Initialize package object with details | \_\_init\_\_ | O(1) | O(1) |
| 55 | String representation of package details | \_\_repr\_\_ | O(1) | O(1) |
| Total |  |  | O(1) | O(1) |

3.Graph Class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 8 | Initialize graph with vertices and edges | \_\_init\_\_ | O(N) | O(1) |
| 16 | Add new vertex to graph | add\_vertex | O(1) | O(1) |
| 19 | Get all vertices from graph | get\_all\_vertices | O(N) | ON) |
| 24 | Add an edge between two vertices | add\_edge | O(1) | O(1) |
| 62 | Associate packages with graph vertices | insert\_packages\_vertex\_associate | O(N^2) | O(N^2) |
| 86 | Get vertex distances from CSV data | get\_csv\_vertex\_distances | O(N) | O(N) |
| 99 | Load graph with vertices and edges | load\_graph | O(N) | O(N^2) |
| 114 | Print edges and weights in the graph | print\_graph\_edge\_weights | O(1) | O(N^2) |
| 124 | Print edges and associated packages | print\_edges\_packages\_asc | O(1) | O(N) |
| 134 | Print vertices and their associated packages | print\_vertices\_packages\_asc | O(1) | O(N^2) |
| 154 | Print packages with specific deadline | print\_package\_deadline\_asc | O( N) | O(N^2) |
| Total |  |  | O(N^2) | O(N^2) |

4.Trucks Class

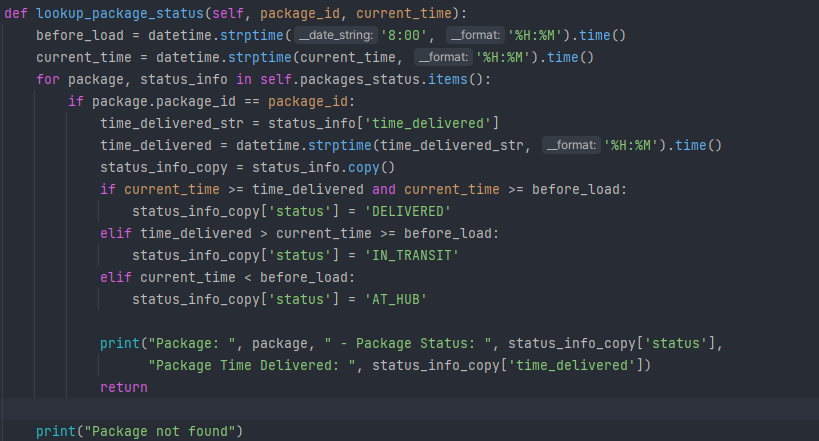
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 33 |  | \_\_init\_\_ | O(1) | O(1) |
| 43 |  | insert\_truck\_id | O(1) | O(1) |
| 48 |  | insert\_packages | O(1) | O(1) |
| 53 |  | get\_packages | O(1) | O(1) |
| 58 |  | remove\_packages | O(1) | O(1) |
| 62 |  | insert\_distances\_pred\_vertex | O(1) | O(1) |
| 66 |  | get\_distances | O(1) | O(1) |
| 71 |  | print\_packages | O(1) | O(N) |
| 76 |  | print\_route | O(1) | O(1) |
|  |  | get\_package\_count | O(1) | O(1) |
|  |  | set\_edge\_weights | O(1) | O(N) |
| \ |  | get\_package\_deadline\_constraints\_low\_asc | O(N) | O(N) |
| \ |  | get\_package\_deadline\_constraints\_med\_asc | O(N) | O(N) |
| \ |  | get\_package\_deadline\_constraints\_high\_asc | O(N) | O(N) |
| \ |  | load\_trucks | O(N) | O(N) |
| \ |  | load\_packages | O(N) | O(N) |
|  |  | get\_left\_over\_packages | O(N) | O(N) |
|  |  | load\_left\_over\_packages | O(N) | O(N) |
|  |  | find\_shortest\_route\_to\_deliver | O(N) | O(N^2) |
|  |  | deliver\_packages | O(N) | O(N^2) |
|  |  | deliver\_truck3\_packages | O(1) | O(N^2) |
| Total |  |  | O(N) | O(N^2) |

5.TimeTracker Class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 7 | Initialize TimeTracker object with package and truck delivery information | \_\_init\_\_ | O(1) | O(1) |
| 19 | Convert delivery deadline to time | convert\_deadline\_time | O(1) | O(1) |
| 27 | Format time to 24-hour format | format\_time | O(1) | O(1) |
| 36 | Get fixed truck speed | get\_truck\_speed | O(1) | O(1) |
| 39 | Initialize truck data | insert\_current\_truck | O(1) | O(1) |
| 51 | Get all trucks | get\_all\_trucks | O(N) | O(N)) |
| 57 | Remove truck’s data | remove\_current\_truck | O(1) | O(1) |
| 62 | Get status of all packages | get\_all\_package\_status | O(1) | O(1) |
| 66 | Lookup package and it’s status from package ID | lookup\_package\_status | O(1) | O(N) |
|  | Update packages status | update\_package\_status | O(1) | O(1) |
|  | Print status of all packages | print\_all\_package\_status | O(1) | O(N) |
|  | Initialize multiple package statuses | initialize\_multiple\_package\_status | O(1) | O(N) |
|  | Update time to start delivery of truck | update\_time\_to\_start\_delivery | O(1) | O(N) |
|  | Increment time of truck when delivering packages | increment\_current\_truck\_time | O(1) | O(1) |
|  | Get current time of truck | get\_current\_truck\_time | O(1) | O(1) |
|  | Update miles traveled of truck when delivering packages | update\_miles\_traveled | O(1) | O(1) |
|  | Print miles traveled of truck | print\_miles\_traveled | O(1) | O(1) |
|  | Calculate travel time | calculate\_travel\_time\_minutes | O(1) | O(1) |
|  | Update truck time when delivering packages | update\_current\_truck\_time | O(1) | O(1) |
|  | Insert time of delivery with package | insert\_current\_truck\_time\_to\_package | O(1) | O(1) |
|  | Check if trucks are ready for delivery | is\_ready\_to\_deliver | O(1) | O(N) |
|  | Check if truck’s delivery is complete | is\_delivery\_completed | O(1) | O(1) |
|  | Update package status | updated\_delivered\_delivery | O(1) | O(N) |
|  | Print packages with ‘DELIVERED’ status | print\_delivered\_status | O(N) | O(N) |
|  | Show packages within certain time ranges | filter\_packages\_by\_time\_range | O(N) | O(N) |
| Total |  |  | O(N) | O(N) |

B. Look-Up Function:

Trucks extends TimeTracker class to keep track of packages during their delivery. During delivery, ability to retrieve specific package from ID and the associated package data components and delivery status and time delivered based on the current time.



C. Original Code

Major code blocks screenshots go here showing implementation.

C1. Identification Information

main.py screenshot goes here showing Student ID

C2. Process and Flow Comments

Some code blocks screenshots go here showing comments.

D. Interface

Interface screenshot goes here

D1. First Status Check

Screen shot goes here

D2. Second Status Check

Screenshot goes here

D3. Third Status Check

Screenshot goes here

E. Screenshot of Code Execution

Screenshot goes here

F1. Strengths of the Chosen Algorithm

Text goes here

F2. Verification of Algorithm

Text goes here

F3. Other Possible Algorithms

Text goes here

F3a. Algorithm Differences

Text goes here

G. Different Approach

Text goes here

H. Verification of Data Structure

Text goes here

H1. Other Data Structures

Text goes here

H1a. Data Structure Differences

Text goes here

I. Sources

Text goes here

An example:

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Retrieved March 22, 2021, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>

J. Professional Communication