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 problem is to find 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. The packages are grouped by delivery deadlines; packages that must be delivered by 9:00 AM, packages that must be delivered by 10:30 AM, and packages that must be delivered by EOD which we will define as 5:00 PM. Additional challenges to the problem is that there is a limit of three trucks, two available drivers for delivery, and capacity limits. This means no more than two trucks can be out for delivery at any given time and load balancing is required to ensure that trucks are not loaded with more than 16 packages. To make the problem even more complicated, we must also account for delays in package arrival, requirements like grouped packages for delivery, and error handling of packages with incorrect delivery addresses. To address these challenges a systematic approach is required.

# Abstract

To address the problem and these challenges, this paper presents a comprehensive solution that utilizes algorithms and data structures to optimize package deliveries. The solution encompasses truck prioritization, the implementation of the nearest neighbor and two-opt algorithms for route optimization, and the utilization of Dijkstra's shortest path algorithm for precise route calculations. By integrating these components, this aims to enhance the efficiency and accuracy of package deliveries for WGUPS.

# Introduction

The Western Governor University Parcel Service (WGUPS) plays a pivotal role in facilitating timely package deliveries in a dynamic and demanding environment. With the objective of delivering 40 packages by the end of the day, WGUPS faces multiple challenges that necessitate a systematic approach to ensure successful and optimized deliveries. These challenges encompass the management of limited trucks and available drivers, compliance with diverse delivery deadlines, consideration of specific truck loading constraints, and the accommodation of special requirements such as grouped package deliveries and designated trucks for specific packages.

In response to these challenges, this paper presents a comprehensive solution approach designed to optimize package deliveries and maximize operational efficiency. The proposed approach is grounded in the utilization of algorithms and data structures, which collectively contribute to the optimization of package routes, adherence to deadlines, and efficient allocation of resources.

This paper is organized as follows: **Section 1** presents the classes and methods that we will utilize in managing packages, trucks, and delivery tracking. **Section 2** delves into the implementation details of each algorithm, emphasizing their role in optimizing package deliveries as well as the space-time complexity of the implementation**. Section 3** delves into the process and flow of the code, loading data using our data structures and manipulating the data using our algorithms. **Section 4** delves into the command line interface where we will simulate loading of packages, optimizing the routes, and the monitoring of packages during their delivery. **Section 5** delves into the evaluation of the strengths of our selected data structures and algorithms, alternative approaches , key findings of our approach, challenges faced, and possible areas of improvement, followed by a conclusion that summarizes the main contributions of the paper and underscores its significance for the field of logistics and route optimization.

By presenting a holistic solution approach, this paper aims to provide valuable insights and practical strategies that can be applied to real-world scenarios, enhancing the efficiency, accuracy, and effectiveness of package deliveries for organizations like WGUPS.

# Proposed Solution and Approach

Our approach consists of several 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 load packages onto each truck. It starts from the hub (starting point) and iteratively selects the nearest package or destination to visit next while keeping in mind individual package constraints. 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(Yenigün, O.). In the context of the code, the Nearest Neighbor algorithm is used to load packages for each truck and create an initial route, which is then further optimized using Two-opt algorithm, and finally 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(Brilliant Math & Science Wiki. (n.d.)), 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.

# Major Code Blocks

Implementation of data structures and created methods to manage packages, trucks, and the logic of loading and transportation. Our data is stored in two csv files. First, ‘WGUPS Package File Formatted.csv’ contains all the 40 packages required for delivery with their respective details such as package ID, delivery address, delivery deadline, and special notes. Second, ‘WGUPS\_Distances.csv’ contains distance information between various locations. Each row represents a specific location, and the columns provide distance values between that location and other locations.

## HashMapEntry Class

The HashMapEntry Class defines a package object with its details. To be loaded into HashMap Object

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 22 | Initialize package object with details | \_\_init\_\_ | O(1) | O(1) |
| 69 | String representation of package details | \_\_repr\_\_ | O(1) | O(1) |
| Total |  |  | O(1) | O(1) |

## HashMap Class

The Hash Map Class is responsible for managing all 40 packages as key-value pairs. Where HashMapEntry Class defines a package object with its details to be loaded into HashMap. Using the package ID associated with each package as the key and the package object as the value for our key-value pairs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 87 | Initialize an empty hash map | \_init\_\_ | O(N) | O(N) |
| 100 | Compute the hash bucket for a given key | get\_hash | O(1) | O(1) |
| 114 | Insert a new key-value pair into hash map | insert\_package | O(N) | O(N) |
| 149 | Retrieve value using a given key | get\_value\_from\_key | O(1) | O(N) |
| 170 | Retrieve key using given address | get\_address\_from\_key | O(1) | O(N) |
| 189 | Retrieve address using given key | get\_key\_from\_address | O(1) | O(N) |
| 205 | Retrieve the entire hash map | get\_hashmap | O(1) | O(1) |
| 211 | Retrieve packages in the hash map | get\_packages | O(1) | O(1) |
| 218 | Print details of all packages | print\_all\_packages | O(1) | O(N) |
| 237 | Update value for a given key | update\_value | O(1) | O(N) |
| 261 | Delete key-value pair using given key | delete\_key\_value | O(1) | O(N) |
| 286 | Load CSV data into the hash map | load\_hash\_map | O(N) | O(N) |
| 332 | Print details of all packages for testing | print\_hash\_map\_all | O(1) | O(N) |
| 346 | Check if all packages exist in the hash map | check\_all\_packages | O(1) | O(N) |
| Total |  |  | O(N) | O(N) |

## Graph Class

The Graph Class represents a graph with vertices and edges. Used to store delivery locations and distances associated with the packages. It also creates associations between vertices(addresses) and package objects stored in the HashMap object. Allows for efficient retrieval and manipulation of package data needed to perform operations or calculations specific to the packages at a given location.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 19 | Initialize graph with vertices and edges | \_\_init\_\_ | O(N) | O(1) |
| 30 | Add new vertex to graph | add\_vertex | O(1) | O(1) |
| 41 | Get all vertices from graph | get\_vertices | O(1) | O1) |
| 51 | Get all edge weights from the graph | get\_edge\_weight | O(1) | O(1) |
| 62 | Add an edge between two vertices | add\_edge | O(1) | O(1) |
| 97 | Associate packages with graph vertices | insert\_packages\_vertex\_associate | O(1) | O(N^2) |
| 130 | Get vertex distances from CSV data | get\_csv\_vertex\_distances | O(N) | O(N) |
| 154 | Load graph with vertices and edges | load\_graph | O(N) | O(N^2) |
| 176 | Print edges and weights in the graph | print\_graph\_edge\_weights | O(1) | O(N^2) |
| 195 | Print edges and associated packages | print\_edges\_packages\_asc | O(1) | O(N) |
| 217 | Print vertices and their associated packages | print\_vertices\_packages\_asc | O(1) | O(N^2) |
| 233 | Print packages with specific deadline | print\_package\_deadline\_asc | O( N) | O(N^2) |
| Total |  |  | O(N) | O(N^2) |

## Trucks Class

The Trucks class is used to create the high priority, medium priority, and low priority trucks. It encapsulates attributes and methods related to truck management and simulating our delivery. Facilitating the organization and optimization of package delivery operations using our implemented data structures and algorithms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 47 | Initialize class with all necessary components to handle loading and delivery | \_\_init\_\_ | O(1) | O(1) |
| 70 | Get the id of current truck | truck\_id | O(1) | O(1) |
| 80 | Get name of current truck | truck\_name | O(1) | O(1) |
| 91 | Insert packages and route into truck | insert\_packages | O(1) | O(1) |
| 104 | Insert filtered packages into truck | insert\_filtered\_packages | O(1) | O(1) |
| 111 | Get packages in truck | get\_packages | O(1) | O(1) |
| 122 | Remove package from truck | remove\_packages | O(1) | O(1) |
| 135 | Insert distance and predecessor vertices into truck | insert\_distances\_pred\_vertex | O(1) | O(1) |
| 149 | Get distances dictionary of current truck | get\_distances | O(1) | O(1) |
| 160 | Print all packages of current truck | print\_packages | O(1) | O(N) |
| 167 | Print filtered packages during delivery | print\_filtered\_packages | O(1) | O(N) |
| 175 | Print route taken by truck | print\_route | O(1) | O(1) |
| 190 | Get number of packages on truck | get\_package\_count | O(1) | O(1) |
| 204 | Load all trucks | load\_trucks | O(1) | O(N) |
| 236 | Find the shortest route for each truck | find\_shortest\_route\_to\_deliver | O(N) | O(N^2) |
| 281 | Deliver packages for truck 1 and 2 | deliver\_packages | O(N) | O(N^2) |
| 353 |  | deliver\_truck3\_packages | O(1) | O(N^2) |
| Total |  |  | O(N) | O(N^2) |

## TimeTracker Class

The TimeTracker Class manages and tracks various aspects of package delivery, including package status such as ‘AT\_HUB’, ‘IN\_TRANSIT’, ‘DELIVERED’, delivery times, and miles traveled. The Trucks class ‘has a’ relationship with the TimeTracker class to allow time, status, miles traveled tracking functionalities specific to trucks.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 17 | Initialize TimeTracker object with package and truck delivery information | \_\_init\_\_ | O(1) | O(1) |
| 48 | Get speed of truck | get\_truck\_speed | O(1) | O(1) |
| 54 | Get miles traveled for truck | get\_miles\_traveled | O(1) | O(1) |
| 62 | Get all package statuses for truck | get\_package\_status | O(1) | O(1) |
| 71 | Set the time of truck 3 after truck 1 finishes delivery | set\_truck\_current\_time | O(1) | O(1) |
| 90 | Look up individual package status during delivery | lookup\_single\_package\_status | O(1) | O(N) |
| 123 | Update status of packages in TimeTracker package\_status dictionary | update\_package\_status | O(1) | O(1) |
| 166 | Print status of all packages | print\_all\_package\_status | O(1) | O(N) |
| 184 | Initialize multiple package statuses | initialize\_multiple\_package\_status | O(1) | O(N) |
| 211 | Update time to start delivery of truck | update\_time\_to\_start\_delivery | O(1) | O(N) |
| 225 | Get single package status | get\_single\_package\_status | O(1) | O(1) |
| 229 | Increment time of truck when delivering packages | increment\_current\_truck\_time | O(1) | O(1) |
| 242 | Get current time of truck | get\_current\_truck\_time | O(1) | O(1) |
| 255 | Update miles traveled of truck when delivering packages | update\_miles\_traveled | O(1) | O(1) |
| 272 | Print miles traveled of current truck | print\_current\_truck\_miles | O(1) | O(1) |
| 286 | Calculate total miles traveled of single truck | calculate\_total\_miles\_traveled | O(1) | O(1) |
| 299 | Calculate travel time minutes | calculate\_travel\_time\_minutes | O(1) | O(1) |
| 313 | Update truck time when delivering packages | update\_current\_truck\_time | O(1) | O(1) |
| 333 | Insert time of delivery with package | insert\_current\_truck\_time\_to\_package | O(1) | O(1) |
| 349 | Check if trucks are ready for delivery | is\_ready\_to\_deliver | O(1) | O(N) |
| 365 | Check if truck’s delivery is complete | is\_delivery\_completed | O(1) | O(1) |
| 381 | Show packages within certain time ranges | filter\_packages\_by\_time\_range | O(N) | O(N) |
| Total |  |  | O(N) | O(N) |

## Tracking\_Util

The Tracking\_Util consist of various functions to help with tracking the time of delivery in the TimeTracker class.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 5 | Float number to string 24-hour time format | float\_time\_24hr\_str | O(1) | O(1) |
| 21 | Convert time string to datetime | convert\_time\_str\_to\_datatime | O(1) | O(1) |
| 38 | Convert 12-hour format time string to 24-hour datetime | convert\_12hr\_to\_24hr\_datetime | O(1) | O(1) |
| 55 | Validate correct format of time inputted | validate\_time\_format | O(1) | O(1) |
| Total |  |  | O(1) | O(1) |

## Load\_Util

The Load\_Util consist of various helper functions for loading packages onto trucks in our approach. Methods with stars denote methods that are not utilized for our current implementation and were part of approach to manually load packages onto trucks.

(For the table below: k is number of special notes in the package constraints)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Space Complexity | Time Complexity |
| 10 | Get packages to load | get\_all\_packages\_to\_load | O(N) | O(N) |
| 31 | Check if packages can be loaded for current truck by constraints delivery deadline, special notes | has\_load\_packages | O(1) | O(1) |
| 72 | Check if package has any constraints; special notes | has\_package\_constraints | O(1) | O(K) |
| 97 | Randomize order of packages list | randomize\_packages | O(1) | O(N log N) (Shuffle) |
| 112 | Sort packages by distance from current vertex in graph | sort\_packages\_by\_distance | O(1) | O(N log N) |
| 129 | Load packages onto trucks based on constraints in less algorithmic approach | \*load\_packages\* | O(1) | O(N^2) |
| 218 | Get left over packages remaining by tracking packages already loaded | get\_left\_over\_packages | O(N) | O(N) |
| 245 | Load left over packages onto trucks | load\_left\_over\_packages | O(1) | O(N) |
| 307 | Return list of packages that meet low priority truck constraints | \*get\_package\_deadline\_constraints\_low\_asc | O(n) | O(N^2) |
| 340 | Returns list of packages that meet medium priority truck constraints | \*get\_package\_deadline\_constraints\_med\_asc | O(N) | O(N^2) |
| 373 | Return list of packages that meet high priority truck constraints | \*get\_package\_deadline\_constraints\_high\_asc | O(N) | O(N^2) |

# Algorithms

## Core Algorithm Overview

The core algorithm focuses on efficiently loading packages onto trucks by utilizing a self-adjusting heuristic algorithm. The algorithm ensures that each truck follows a systematic approach to select and load packages. This overview highlights the algorithm's key steps, space complexity, and time complexity.

**Space Complexity:**

The space complexity of the given nearest neighbor algorithm is O(n^2), where n represents the number of packages loaded onto the trucks. The primary contributors to the space complexity are the graph representation (which depends on the number of locations) and the lists storing package information (which depend on the number of packages).

**Time Complexity:**

The time complexity of this algorithm is O(n), where n is the number of packages on the truck. This is because package loading loop, that iterates until the truck’s package count reaches 14 or all available packages have been loaded onto the truck.

**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 occurs when the algorithm needs to consider all available packages multiple times and perform additional constraint checks leading to O(n^2). Here are the key factors contributing to the worst-case time complexity:

**Package Loading Loop**: In the worst case, the loop that loads packages onto the trucks needs to iterate through all available packages multiple times until each truck is fully loaded. This results in O(n) iterations, where n is the total number of packages.

**Constraint Checks**: For each available package, there are constraint checks to determine if it can be loaded onto the current truck. In the worst case, these checks could involve iterating through all packages again, leading to an additional O(n) factor.

### Core Algorithm Implementation

The core implementation details the implementation steps and processes involved in the truck package loading procedure. It covers initialization, package separation, nearest-neighbor loading, this process continues until truck package count reaches 14 or all available packages are loaded, ensuring the truck travels to the nearest package at each step and return to hub, and completion and return steps.

1. Initialization:
2. Set the starting vertex of the truck's route to the hub vertex ('4001 South 700 East').
3. Get the list of remaining packages to load onto the truck using the graph and tracked package IDs.
4. Initialize the truck's route list with the hub vertex.

for each truck in trucks:

current\_vertex = '4001 South 700 East'

remaining\_packages = get\_all\_packages\_to\_load(graph, track\_package\_id)

truck.route = [current\_vertex]

1. Package Separation, iterate through the remaining packages:
2. Check if each package can be loaded onto the current truck:
   1. If it meets the current truck's constraints, add it to the constrained packages list.
   2. Otherwise, add it to the unconstrained packages list.

constrained\_packages = []

unconstrained\_packages = []

for each package in remaining\_packages:

if can\_load\_package(truck, package):

constrained\_packages.append(package)

else:

unconstrained\_packages.append(package)

1. Handling Unconstrained Packages:
   1. Remove unconstrained packages with constraints for the current truck from the unconstrained packages list.

unconstrained\_packages = filter out unconstrained packages with constraints

1. Nearest-Neighbor Loading:
2. Combine the constrained and unconstrained packages to form the list of all packages for the current truck.

all\_packages = constrained\_packages + unconstrained\_packages

1. While the truck's package count is less than 14 and there are still packages to load:
   1. Find the nearest package to the current vertex by calculating distances to all packages.
   2. If a suitable nearest package is found:
      1. Update the current vertex to the address of the loaded package.
      2. Insert the package onto the truck.
      3. Add the package's ID to the set of tracked package IDs.
      4. Remove the package from the list of all packages.
   3. If no suitable package is found, print a message indicating that no package was found.

while truck.get\_package\_count() < 14 and len(all\_packages) > 0:

min\_distance = infinity

nearest\_package = None

for each package in all\_packages:

dest\_vertex = package.address

distance = graph.edge\_weight[current\_vertex][dest\_vertex]

if distance < min\_distance:

min\_distance = distance

nearest\_package = package

if nearest\_package is not None:

current\_vertex = nearest\_package.address

truck.insert\_packages(nearest\_package)

track\_package\_id.add(nearest\_package.package\_id)

all\_packages.remove(nearest\_package)

else:

print "No suitable package found for truck", truck.truck\_id

1. Completion and Return:
2. Add the hub vertex to the truck's route to complete the route loop.
3. Print the truck's route.

truck.route.append('4001 South 700 East')

print "Truck", truck.truck\_id, "- Route:", truck.route

## Two-opt Algorithm Overview

The Two-opt algorithm aims to improve the existing routes obtained from the nearest neighbor algorithm by rearranging the sequence of addresses visited, with the aim of optimizing truck routes for package. This overview highlights the algorithm's key steps, space complexity, and time complexity. This process continues until no further improvements can be made.

**Space Complexity**

The space complexity of the 2-opt algorithm is determined by the additional memory used for storing the optimized route and other variables.

**Unique Route Storage**: Storing the unique route without repeated vertices requires O(n) space, where "n" is the number of addresses in the original route.

**Temporary Route Storage**: During the optimization process, temporary routes are created for comparison and swapping. The space used for these temporary routes is also O(n).

Considering these factors, the overall worst-case space complexity of the algorithm can be approximated as O(n)

**Time Complexity**

The worst-case time complexity of the 2-opt algorithm for optimizing route is O(n^2), where "n" is the number of addresses (vertices) in the route. This is because the algorithm iterates through pairs of vertices in a nested loop, resulting in comparisons for every possible pair of vertices. The total number of iterations is proportional to n \* (n - 1) / 2, which simplifies to O(n^2).

### Two-opt Algorithm Implementation

The Two-opt implementation details the implementation steps and processes involved in the truck route optimization. It covers unique route creation, two-opt swap, and completion and return steps.

1. Initialization:
   1. Remove repeated vertices from the truck's route using the remove\_repeated\_vertices function.
   2. Initialize the current route by adding the hub address at the beginning and end of the unique route.
   3. Set the best route as the current route.

unique\_route = remove\_repeated\_vertices(truck.route) excluding hub address

current\_route = ['hub'] + unique\_route + ['hub']

best\_route = current\_route

1. Two-opt swap:
   1. Enter a loop to optimize the route until no further improvements are made.
   2. Calculate the distance of the current route using the calculate\_route\_distance function.
   3. Iterate through each pair of vertices in the route (except the first and last).
   4. Swap the order of the vertices between the pair using the two\_opt\_swap function to create a new route.
   5. Calculate the distance of the new route.
   6. If an improvement is made, set the current route as the best route.

For i from 1 to (length of current route - 2):

For j from (i + 1) to (length of current route - 1):

- Swap the order of vertices between index i and j using the two\_opt\_swap function to create a new route.

- Calculate the distance of the new route.

- If the new route distance is shorter than the current best route distance:

- Update the current route to be the new route.

- Set the current route as the best route.

1. Completion and Return:
   1. Update the truck's route with the optimized route.
   2. Optimize the order of packages to reflect the optimized route.

truck.route = best\_route

optimized\_packages = []

For each address in current\_route:

For each package in truck.packages:

If package.address == address:

Add package to optimized\_packages

Break loop

truck.packages = optimized\_packages

## Dijkstra's Shortest Path Algorithm Overview

The Dijkstra's algorithm is a graph traversal technique that efficiently finds the shortest path between a source vertex and all other vertices in a weighted graph. It employs a priority queue and dynamic distance updating to ensure optimal path discovery. In the context of the provided problem, the algorithm helps determine the most efficient routes for trucks to reach their destinations, ensuring timely package deliveries This overview highlights the algorithm's key steps, space complexity, and time complexity.

**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 Implementation

The Dijkstra implementation details the steps and processes involved in the final route optimization and distance traveled for each truck. It covers initialization, shortest-path discovery, and completion and return steps.

* + - 1. Initialization:

1. Initialize the distances dictionary with all vertices in the route set to infinity, indicating that their distances are unknown initially.
2. Initialize the pred\_vertex dictionary with all vertices in the graph to None, indicating that their predecessor vertices are unknown initially.
3. Set the distance of the source vertex (src) to 0.
4. Create a min-heap queue (min\_heap) of tuples containing the distance and vertex.
5. Create an empty list (visited\_queue) to track visited vertices.

distances = {vertex: infinity for vertex in route}

pred\_vertex = {vertex: None for vertex in graph.vertices}

distances[src] = 0

min\_heap = create\_min\_heap()

visited\_queue = []

* + - 1. Shortest-Path Discovery:
         1. Pop the vertex with the smallest distance from the min-heap.
         2. If the vertex is already in the visited queue and the current distance is greater than the stored distance, skip to the next iteration.
         3. Mark the vertex as visited by adding it to the visited queue.
         4. Get the neighbors of the current vertex that are also in the route.
         5. Iterate over the neighbors and calculate the total distance to each neighbor.
         6. If the total distance is smaller than the current known distance to the neighbor, update the distances dictionary and predecessor vertex.
         7. Push the updated distance and neighbor vertex into the min-heap.

while min\_heap is not empty:

current\_distance, current\_vertex = extract\_min\_from\_heap(min\_heap)

if current\_vertex in visited\_queue and current\_distance > distances[current\_vertex]:

continue

visited\_queue.append(current\_vertex)

neighbors = get\_neighbors\_within\_route(graph, current\_vertex, route)

for neighbor in neighbors:

weight = graph.edge\_weight[current\_vertex][neighbor]

total\_distance = current\_distance + weight

if total\_distance < distances[neighbor]:

distances[neighbor] = total\_distance

pred\_vertex[neighbor] = current\_vertex

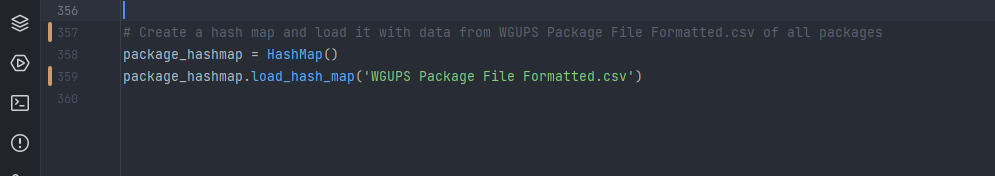
insert\_into\_heap(min\_heap, (total\_distance, neighbor))

* + - 1. Completion and Return:
         1. Return the distances and pred\_vertex dictionaries.

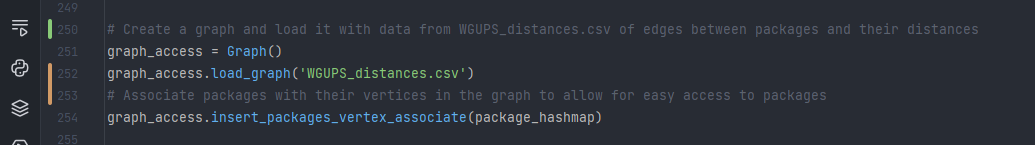
return distances, pred\_vertex

# Process and Flow

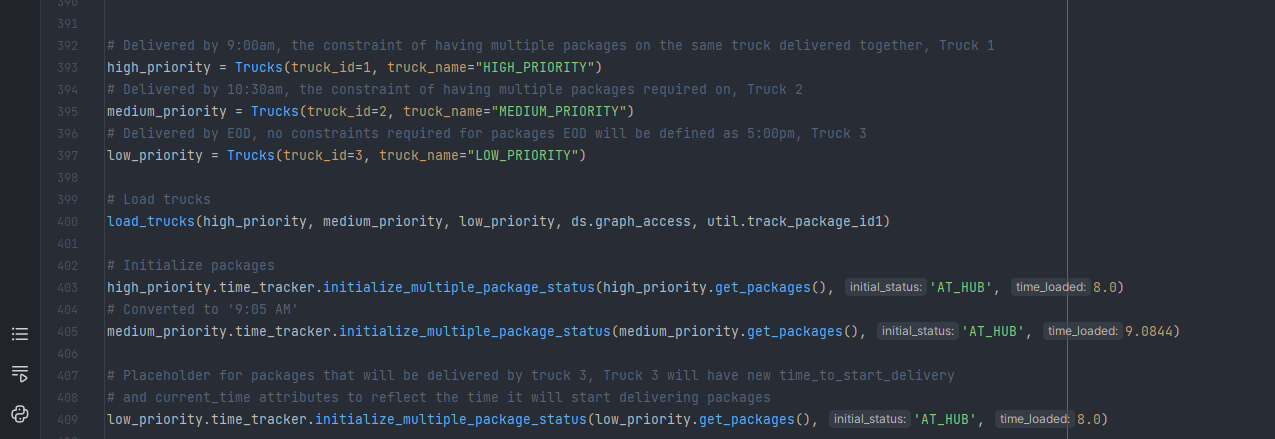
Create hash map and load it.



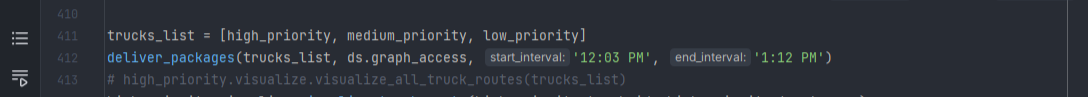
Create a graph of edges and weights between packages and create associations between packages and vertices. Allow for easier package access and package delivery operations.



Create three trucks, load packages onto trucks and initialize packages to start tracking delivery.



Can begin delivery of packages and see their status during certain time ranges. By manually calling the deliver\_packages method. For our implementation, we will use a CLI to allow for easier utilization of the program.

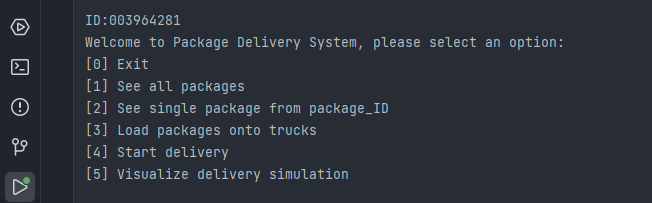


# User Interface and Status Check

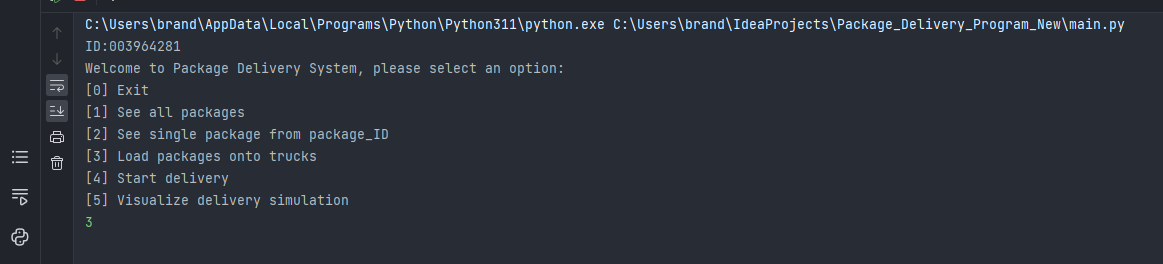
## User Interface

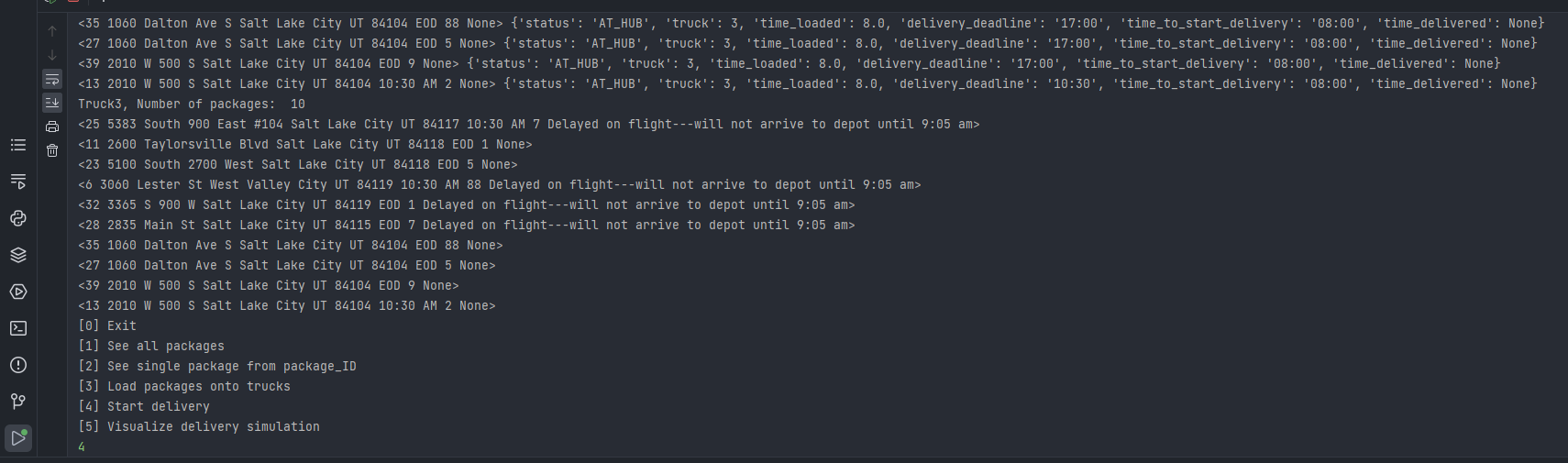
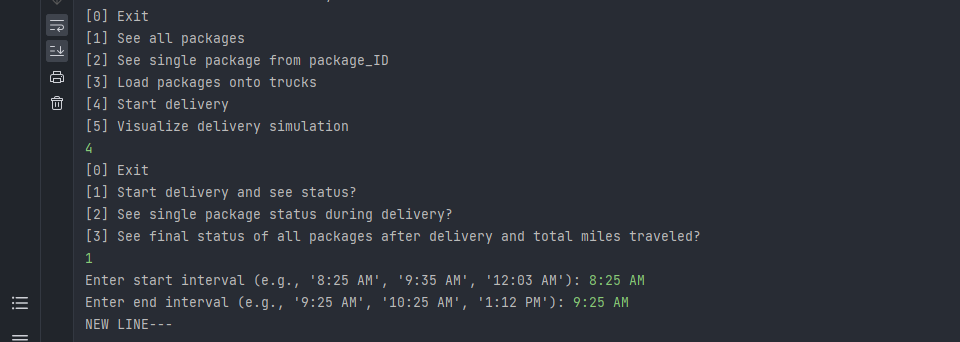
Main UI to run program.

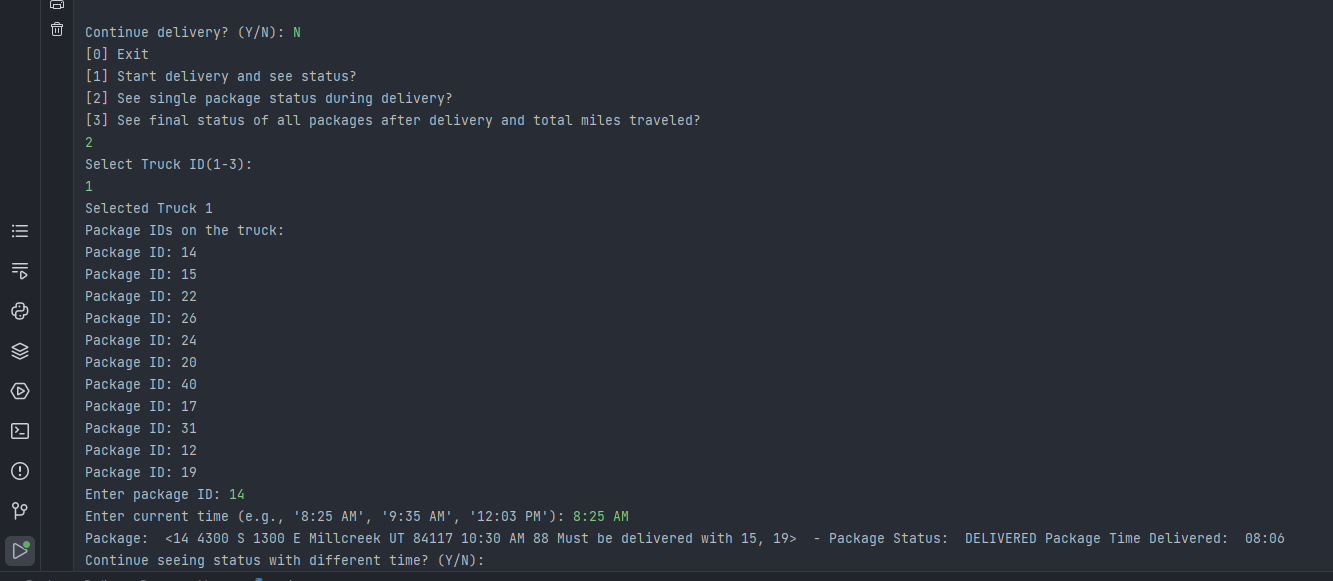
* Select 1 to see all packages and check if packages are all present.
* Select 2 to see single package data of specified package ID.
* Select 3 to prepare trucks with packages to deliver.
* Select 4 to start delivery and see status of packages during delivery.
* Select 5 to see visualization of delivery.



1. Load packages onto truck.

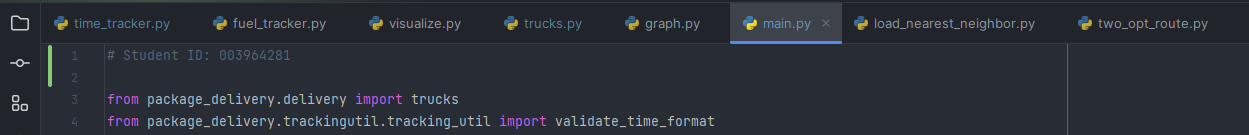


1. After packages are loaded. Can start delivery. 
2. Choose time frame to see status of delivery, use 12-hour format for input. Can continue with different time frames by user. After starting delivery, selecting choice 3 will see final status of all packages and the distance traveled.
3. After starting delivery, can see status of individual packages loaded on trucks at different times.



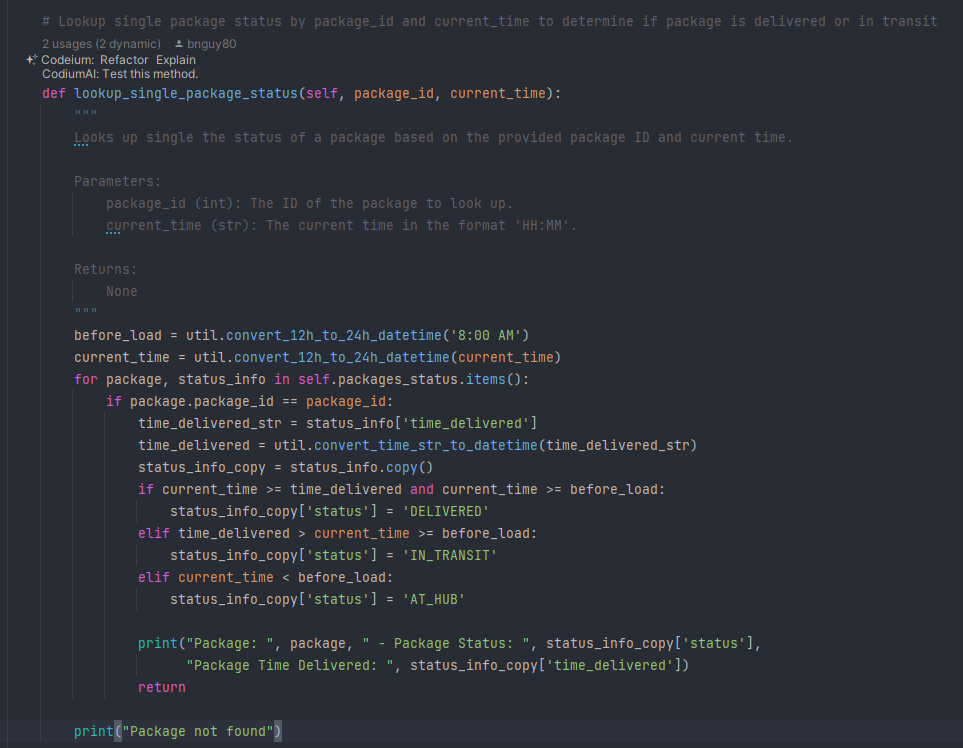
## Identification Information

Main.py with student ID at top of file



## Look-Up Function

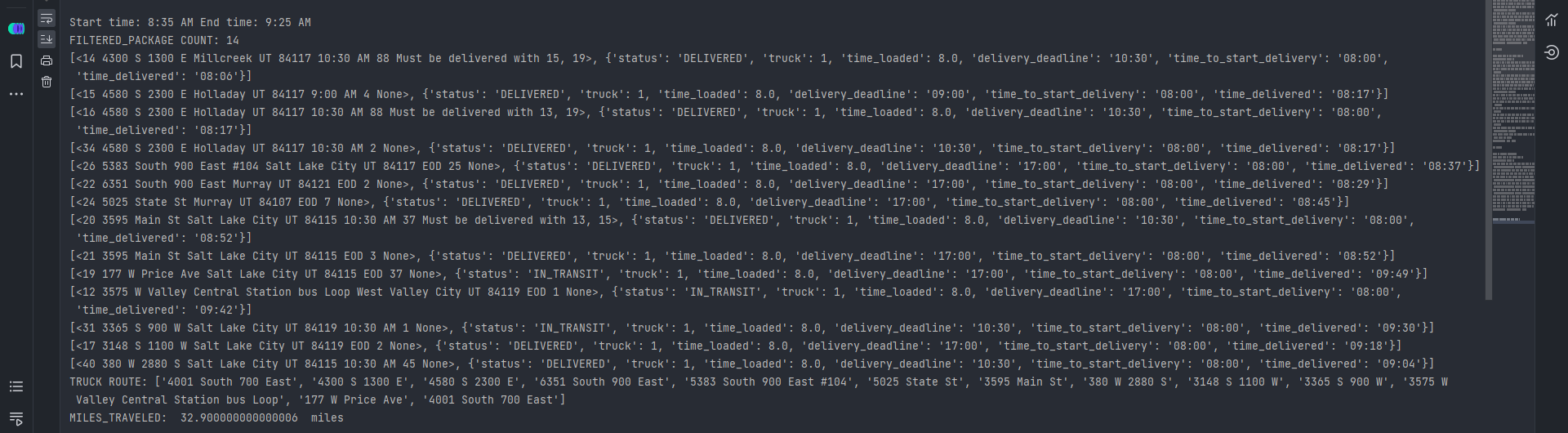
Trucks class ‘has a’ relationship(composite-component) with TimeTracker class to keep track of packages during their delivery. During delivery, ability to retrieve specific package from ID and the associated package data components and delivery status and time delivered based on the current time. Allowing for single package status tracking as well as overall package status tracking.



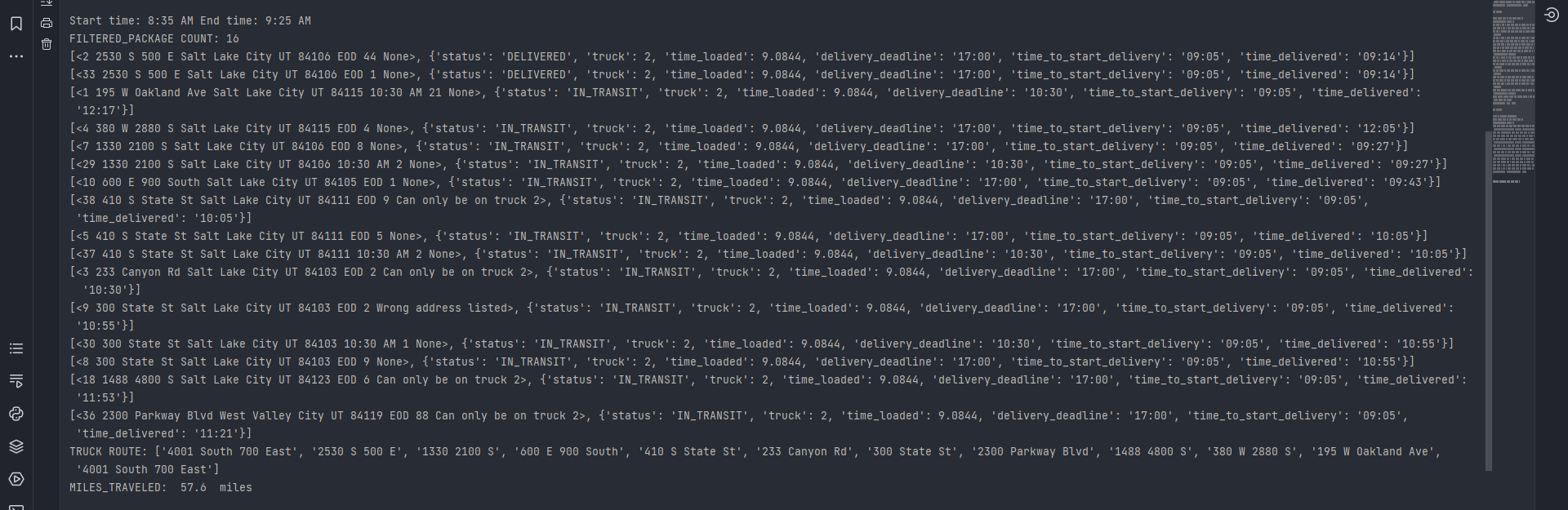
## Status Check

### First Status Check 8:35 AM – 9:25 AM

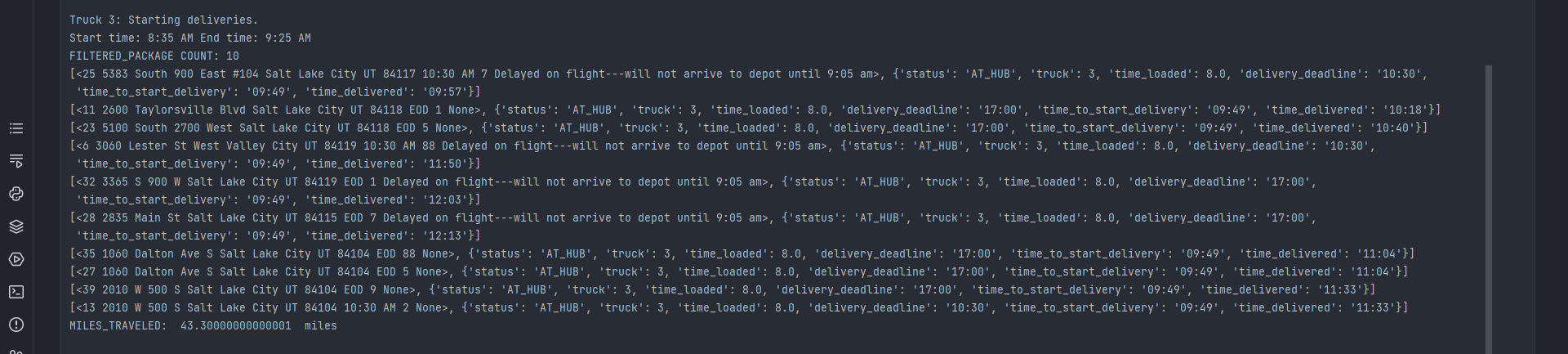
Truck 1



Truck 2

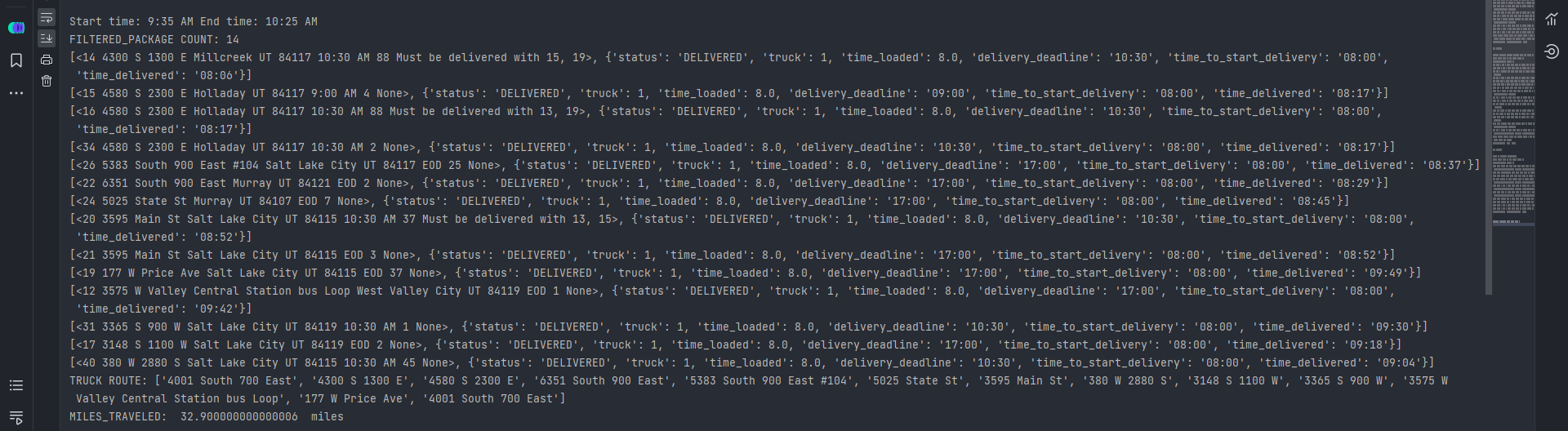


Truck 3

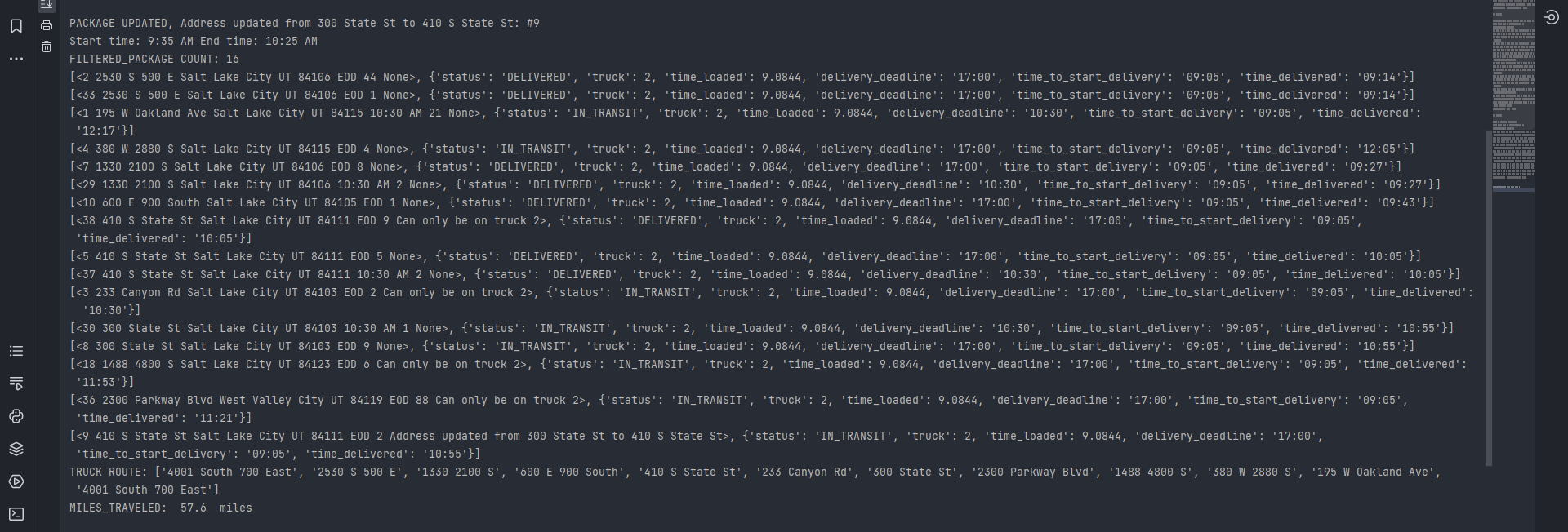


### Second Status Check 9:35 AM – 10:25 AM

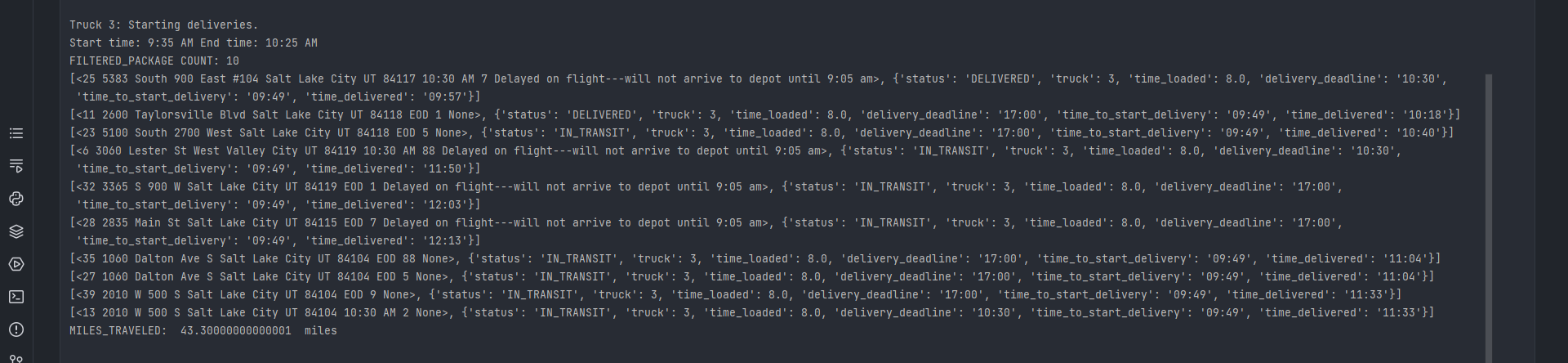
Truck 1



Truck 2

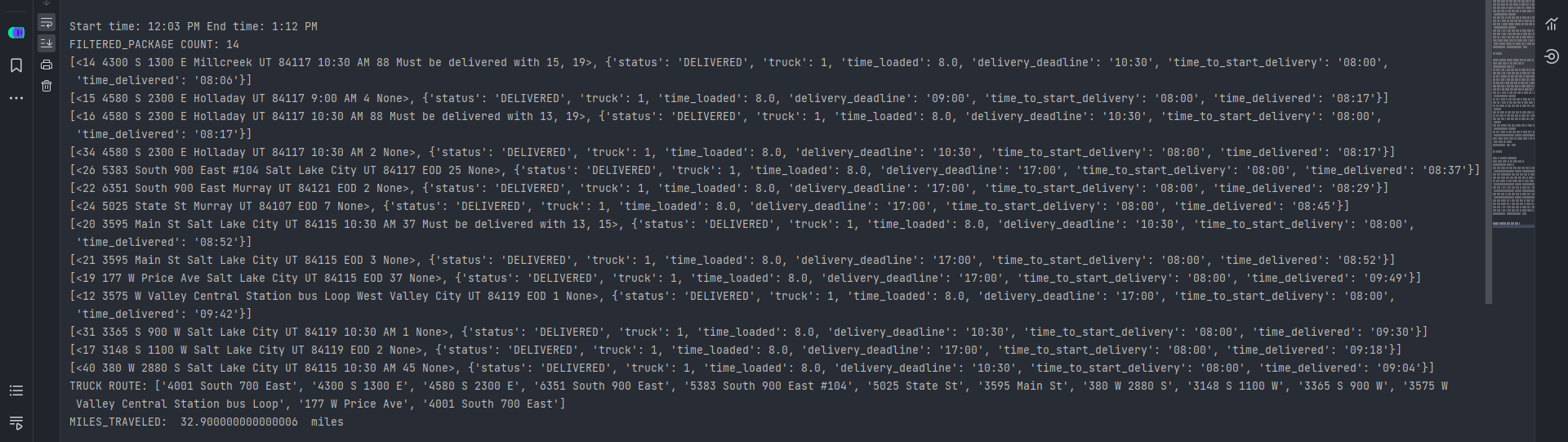


Truck 3

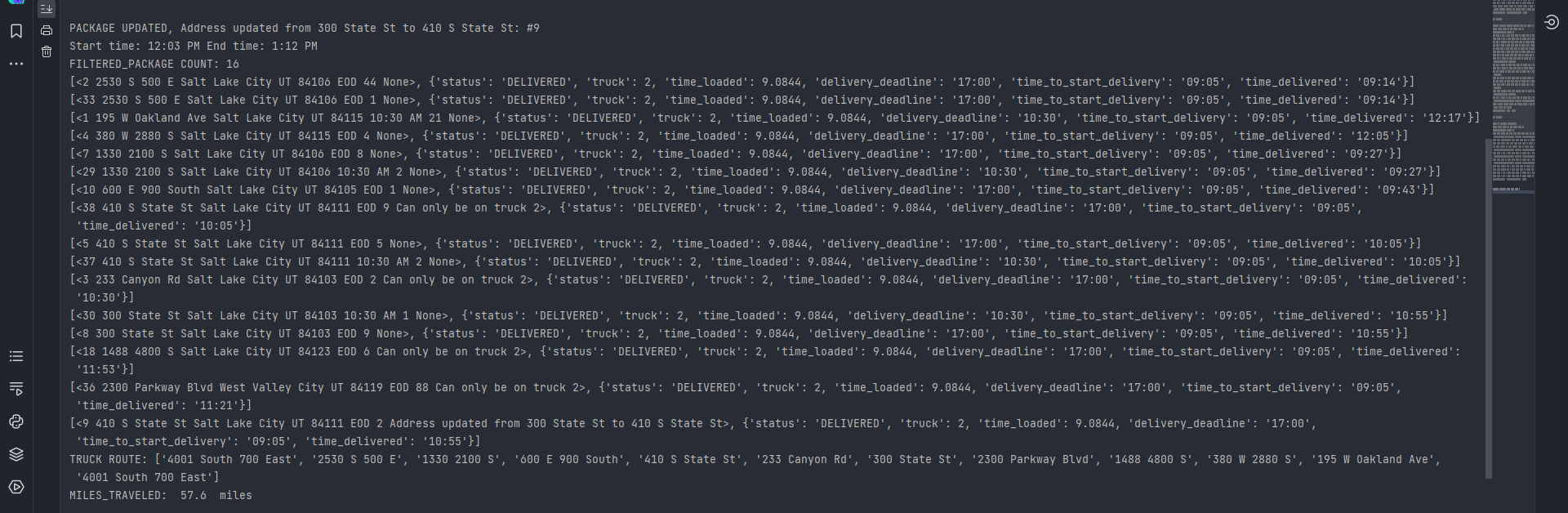


### Third Status Check 12:03 PM – 1:12 PM

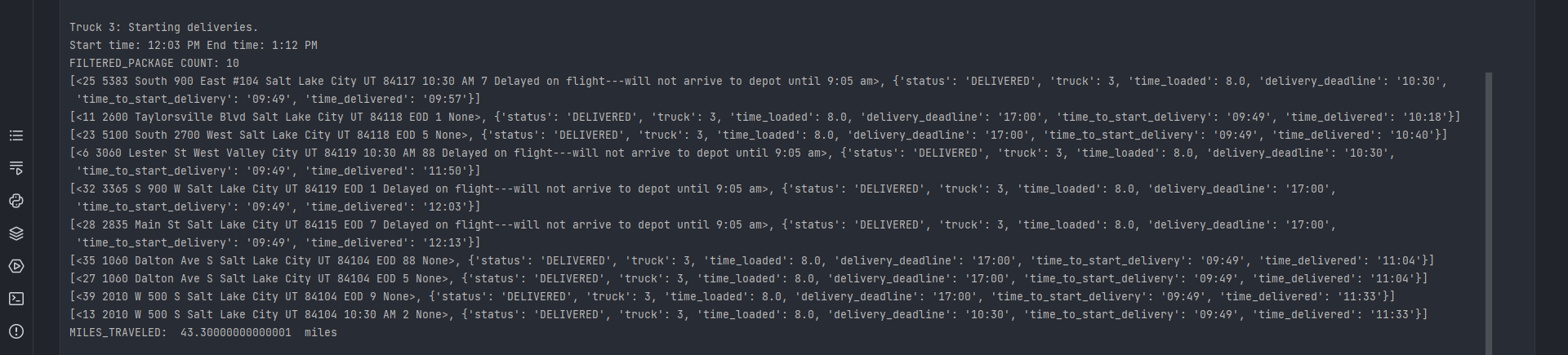
Truck 1



Truck 2

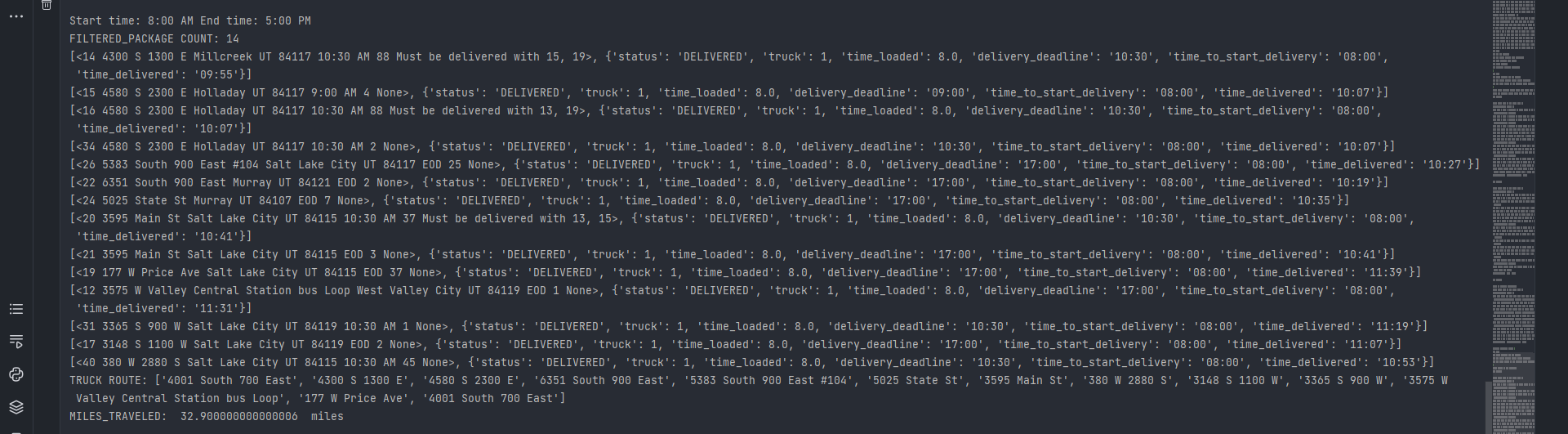


Truck 3

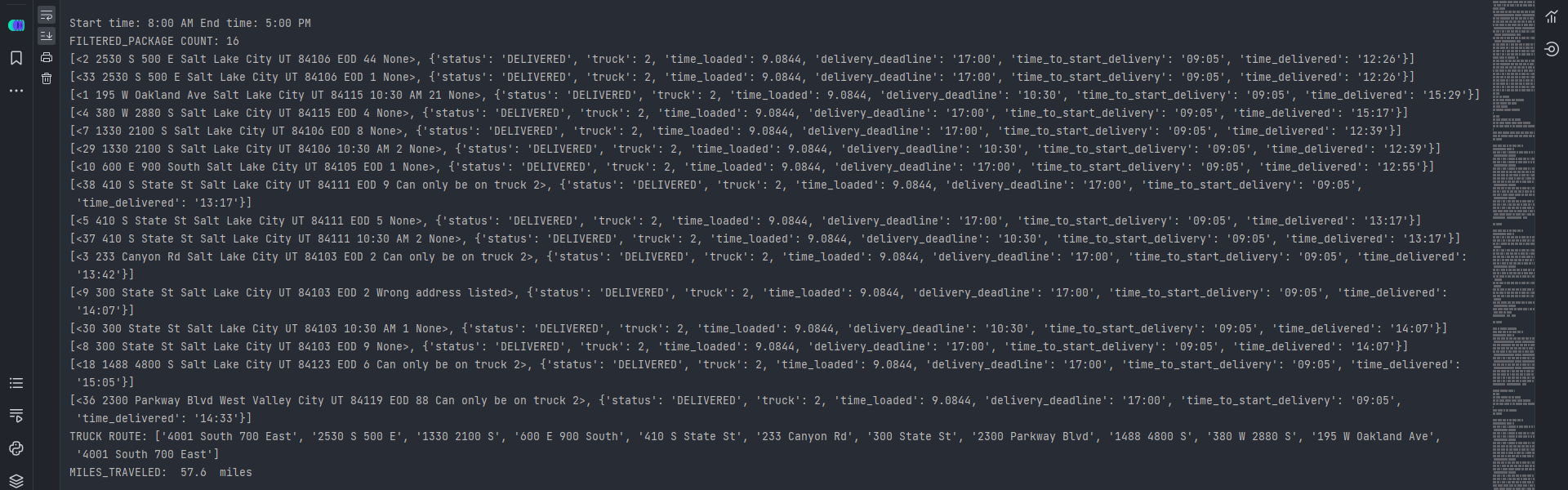


### Screenshot of Code Execution - Final Status and Total Miles Traveled

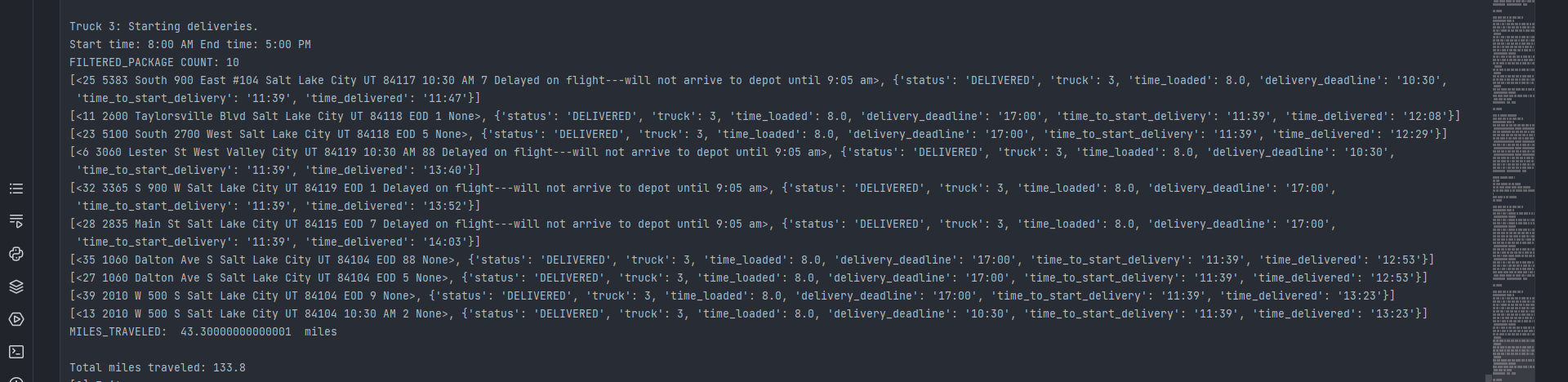
Truck 1



Truck 2



Truck 3



# Data Structure and Algorithm Evaluation

## Data Structures Evaluation

### HashMap Verification

Separation of concerns between managing packages and tracking package status. Hash map has the single responsibility to manage static data of available packages to be loaded onto trucks for delivery. Keeping track of all packages allows for easier ability to maintain and update package information in the future. Each package has the following components of delivery address, delivery deadline, delivery city, delivery zip code, and package weight. By keeping the logic of updating package status separate in TimeTracker allows to handle the single responsibility of tracking the various dynamic aspects of package delivery such as time delivered for each package and package status.

### Graph Verification

To model the relationship between packages and the delivery locations and distances associated with each package a graph is needed. A vertices dictionary which are the delivery locations and edge weights dictionary which are the distances between those locations(Python. Educative. (n.d.)). The graph interacts with the Hash map to create associations between packages and their edge weights. To allow for faster lookup and manipulation of package data for package delivery related operations(Python. Educative. (n.d.)).

### Other Possible Data Structures

An alternative to the existing hash map for storing packages could be a Binary Search Tree (BST). The BST could efficiently sort packages based on attributes like delivery deadline. Operations such as insertion, deletion, and lookup in a BST take O(log N) time complexity, making it particularly useful for dynamic adjustments(Online Courses and eBooks Library. (n.d.)). This data structure allows for both maintaining a sorted list of packages and efficient iteration, allowing loading of trucks with packages in adherence to delivery deadlines.

An alternative to the existing hash map for storing packages could be the Skip List. Organizing the data in multiple levels allowing for insertion, deletion, and lookup to be performed in O(log N) time complexity(Skip list: Set 2 (insertion) GeeksforGeeks). This could be useful for searching packages based on specific constraints or updating the status of the package, while keeping the data sorted based on various criteria such as delivery deadlines, addresses, or priority levels. Allowing more efficient route planning.

### Data Structure Differences

The hash map, while efficient for quick data access with an average time complexity of O(1) for operations like insertion, deletion, and lookup, but doesn't keep the packages in any sorted order. Its space complexity is O(N)( Hash map in python. GeeksforGeeks), based on the number of packages.

In contrast, the binary search tree (BST) keeps packages in a sorted order, an attribute that can be useful for organizing deliveries based on deadlines. It offers a time complexity of O(log N) for operations like insertion, deletion, and lookup, and its space complexity is also O(N). This makes the BST an effective choice for scenarios that require frequent data manipulations while keeping the information ordered.

Similarly, the Skip List also maintains sorted order while providing a time complexity of O(log N) for essential operations. Although its space complexity is also O(N), additional pointers may increase its space requirements. Skip Lists are especially handy for data sets that require both fast access and sorted organization.

Compared to these alternatives, while the hash map excels in fast data retrieval, it falls short in its ability to maintain sorted data, which could be advantageous for tasks like pre-sorting packages before optimizing delivery routes.

## Algorithms Evaluation

### Verification of Algorithm

The core algorithm used in this program is the self-adjusting nearest neighbor algorithm. The algorithm starts at a given ‘current vertex’ and iteratively selects the closest neighboring vertex as the next point to visit. The algorithm continues this process until it either reaches a specified condition or visits all vertices. In our implementation, it is used to dynamically load packages onto trucks and plan their routes. The packages are successfully loaded onto trucks with considerations of individual package constraints and truck loading limitations. The algorithm provides reasonably optimized routes within acceptable time limits.

### Strengths of the Chosen Algorithm

In the fast-paced environment of package delivery, where decisions need to be made in real-time, this algorithm provides a quick yet reasonably accurate routing solution. This is crucial for ensuring that packages are delivered within the tight time windows often required. Another strength of the Nearest Neighbor Algorithm is its adaptability. The algorithm can be re-run whenever the conditions change, such as the addition of new packages or adjustments in package priorities. This makes it highly flexible and suited for a dynamic operation where variables can change frequently.

## Alternative Algorithm Approaches

### Other Possible Algorithms

An alternative to our current nearest neighbor algorithm is the probabilistic approach of simulated annealing. While the current algorithm may become trapped in local optima—choosing routes that appear shortest only in the immediate context—simulated annealing has the capability to break free from such restrictions. It does this by occasionally selecting suboptimal routes, thereby increasing the likelihood of finding a more globally efficient path for delivery. In contrast to merely settling for the nearest available option, this approach broadens the search to potentially discover better overall solutions.

An alternative to our current nearest neighbor algorithm is genetic algorithm. Like discussed earlier the current algorithm selects the current immediate optimal route. The genetic algorithm considers a broader range of possibilities by generating a population of routes and iteratively improving on them. Combining elements of different routes to create a potentially more optimal solution. Particularly useful for more complex delivery network and constraints as it allows for comprehensive search for best routes.

### Algorithm Differences

The nearest neighbor algorithm starts from the hub and iteratively selects the nearest neighbor or destination based on immediate context. Quick and easy to implement and provides a good-enough solution in acceptable time requirements. Simulated annealing provides a more nuanced approach by occasionally selects suboptimal routes, allowing for the algorithm to escape local minima and explore broader solutions to potentially find global optimal solutions. Genetic algorithm generates a population of routes and iteratively improves upon them through crossover and mutation. The algorithm goes beyond the immediate locally optimal route and explore a wider range of route combinations. More adaptable and can handle complex delivery scenarios.

# Different Approach to Project

If I were to revisit this project, I would employ the strategy design pattern to create an interface for the various route optimization algorithms. This would allow for interchangeability between different algorithms, allowing easier testing and comparison of their efficiency. By adhering to this design pattern, the codebase would become more maintainable and extensible, as incorporating new algorithms would not need changes to existing code. Here's a detailed explanation of how I would implement the strategy design pattern.

Define Strategy Interface:

Define a common interface for all route optimization strategies to implement.

from abc import ABC, abstractmethod

class RouteOptimizationStrategy(ABC):

@abstractmethod

def optimize\_route(self, packages, trucks, graph):

pass

Implement Concrete Strategies:

Define classes that implement interface for each optimization algorithm.

class NearestNeighbor(RouteOptimizationStrategy):

def optimize\_route(self, packages, trucks, graph):

# Implement nearest neighbor algorithm

pass

class SimulatedAnnealing(RouteOptimizationStrategy):

def optimize\_route(self, packages, trucks, graph):

# Implement simulated annealing algorithm

pass

class GeneticAlgorithm(RouteOptimizationStrategy):

def optimize\_route(self, packages, trucks, graph):

# Implement genetic algorithm

Pass

Create Class to Use Strategy:

class DeliverySystem:

def \_\_init\_\_(self, graph, strategy: RouteOptimizationStrategy):

self.strategy = strategy

self.packages = []

self.trucks = []

self.graph = graph

def set\_strategy(self, strategy: RouteOptimizationStrategy):

self.strategy = strategy

def optimize\_route(self):

self.strategy.optimize\_route(self.packages, self.trucks, self.graph)

Using the Strategy:

Can easily swap out optimization algorithms without altering the code that uses it.

def main():

# Initialize with nearest neighbor strategy

delivery\_system = DeliverySystem(NearestNeighbor())

delivery\_system.optimize\_route()

# Switch to simulated annealing strategy

delivery\_system.set\_strategy(SimulatedAnnealing())

delivery\_system.optimize\_route()

# Switch to genetic algorithm strategy

delivery\_system.set\_strategy(GeneticAlgorithm())

delivery\_system.optimize\_route()

if \_\_name\_\_ == "\_\_main\_\_":

main()

# Results and Discussion

Our core algorithm, which uses a combination of the Nearest Neighbor, Two-opt, and Dijkstra's algorithms, has demonstrated the ability to deliver all packages within the stipulated time frame. The Nearest Neighbor algorithm was primarily used for initial route planning and was further optimized using the Two-opt algorithm. Finally, Dijkstra’s algorithm was used to find the shortest path for each route. One of the significant strengths of this approach is its efficiency in terms of time complexity. The Nearest Neighbor algorithm offers quick initial route planning, while Dijkstra's ensures that the shortest path is chosen for each route. Additionally, the Two-opt algorithm helps in optimizing the route by eliminating crossings, thereby making the path more streamlined. While the algorithm is efficient, there are limitations. The Nearest Neighbor algorithm can sometimes get trapped in local minima, leading to suboptimal solutions. Furthermore, the hash map data structure used for storing package information is not best suited for maintaining the packages in sorted order, which could be useful for more advanced sorting algorithms. We also considered alternative algorithms like Simulated Annealing and Genetic Algorithm, which could potentially offer more optimal solutions by escaping local minima and exploring a broader range of solutions, respectively. Alternative data structures like Binary Search Trees and Skip Lists were also considered for better sorting and retrieval functionalities. If we were to revisit this project, adopting a more modular codebase through the strategy design pattern could be beneficial. This would make the code more maintainable and extensible, allowing for easy swapping of algorithms and features without affecting the existing codebase.

# References

Data Structure - Binary Search Tree. Online Courses and eBooks Library. (n.d.). <https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm>

GeeksforGeeks. (2023, February 13). Skip list: Set 2 (insertion). GeeksforGeeks. <https://www.geeksforgeeks.org/skip-list-set-2-insertion/>

Dijkstra’s shortest path algorithm. Brilliant Math &amp; Science Wiki. (n.d.-a). <https://brilliant.org/wiki/dijkstras-short-path-finder/>

Yenigün, O. (2023, June 16). Traveling salesman problem: Nearest Neighbor Algorithm Solution. Medium. <https://blog.devgenius.io/traveling-salesman-problem-nearest-neighbor-algorithm-solution-e78399d0ab0c>

2-opt algorithm: Solving the travelling salesman problem in Python. Saturn Cloud Blog. (2023, August 25). <https://saturncloud.io/blog/2opt-algorithm-solving-the-travelling-salesman-problem-in-python/#:~:text=Python%20Implementation%20of%20the%202%2Dopt%20Algorithm&amp;text=The%20two_opt%20function%20implements%20the,returns%20the%20best%20route%20found>.

GeeksforGeeks. (2023b, August 20). Hash map in python. GeeksforGeeks. <https://www.geeksforgeeks.org/hash-map-in-python/>

How to implement a graph in Python. Educative. (n.d.). <https://www.educative.io/answers/how-to-implement-a-graph-in-python>