C950 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, error handling of packages with incorrect delivery addresses, and lastly ensuring delivery does not exceed 140 miles. 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, accommodation of special requirements such as grouped package deliveries and designated trucks for specific packages and ensuring the total delivery distance does not exceed 140 miles.

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: **Proposed Solution and Approach** discussing the choice of algorithms used and reasoning for each algorithm used in the program. **Development Environment**  delves into the program environment, dependencies, and set up of program to run. **Software Efficiency and Maintainability** presents the codebase structure and organization of the program. **Major Code Blocks** presents the classes and methods in the codebase that we will utilize in managing packages, trucks, and delivery tracking. **Algorithms** delves into the implementation details of each algorithm in the codebase, emphasizing their role in optimizing package deliveries as well as the space-time complexity of the implementation**. Process and Flow** delves into the process and flow of the code, loading data using our data structures and manipulating the data using our algorithms. **User Interface and Status Check** delves into the command line interface where we will simulate loading of packages, running delivery simulation, and the monitoring of packages during their delivery. **Data Structure and Algorithm Evaluation** 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 **Results and Discussion** that summarizes the main findings of the paper, and finally **Different Approach to Project** discussing future improvements and modification of the program.

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.

# Development Environment

The development environment that we will be using is **PyCharm 2023.2.1 (Professional Edition)** IDE with **Python 3.11** as the interpreter with usage of **matplotlib==3.7.2** and **Requests==2.31.0** packages noted in the requirements.txt dependencies in the **Package\_Delivery\_Program\_New** file.

1. **cd path\_to\_Package\_Delivery\_Program\_New** to navigate to project directory.
2. **venv\Scripts\activate** to activate virtual environment(venv).
3. **pip install -r requirements.text** to setup program before running simulations.

# Software Efficiency and Maintainability

## Modular Architecture

Our software design is inherently modular, meaning distinct functionalities are isolated in separate modules or classes. This allows for a clear separation of concerns. For example, the algorithms, data structures, and delivery logistics components all reside in their individual modules. Such a set up makes it easier to understand the codebase and simplifies the process of updating or swapping out specific functionalities without affecting others.

## Clear Documentation and Commenting

Throughout our codebase, clear comments and documentation have been added. This ensures that anyone reviewing or working on the code in the future will have a clear communication of understanding the logic and flow. Making maintenance and refactoring more straightforward.

## Error Handling

The UI where user interaction will be taken place is equipped with error handling mechanisms to ensure correct flow of execution. Preventing unexpected errors when simulating delivery throughout the day under this specific scenario discussed in the stated problem.

## Efficiency

The overall dominating factor of the time complexity of the program are the algorithms used for routing and package loading is O(N^2). The nearest neighbor algorithm is the primary algorithm, where the worst-case time complexity is O(N^2) for N packages. The two-opt algorithm used to further refine the initial routes calculated by the nearest neighbor algorithm has a time complexity of O(N^2) for N locations. Finally, dijkstra’s shortest path algorithm used to find the final total distance on the optimized routes have a time complexity of O(E + V log V), but this is overshadowed by the O(N^2) of the nearest neighbor and two-opt in most practical scenarios.

The two data structures used are the hash map and graph. The hash map’s space complexity is O(N) for N packages, while the graph’s space complexity is O(V + E). Combing these, the overall space complexity is O(N + V + E) which can be simplified to O(N).

For our current scenario, the number of packages N isn’t very large. For smaller values of N, even an O(N^2) algorithm can run quickly in practice. As N grows, the time taken by O(N^2) grows quadratically. For example, if N goes from 100 to 1000, the operations will not just increase by a factor of 10 but by a factor of 100. In conclusion, while O(N^2) efficient for the current scenario given the problem size and constraints, for scalability, especially with a significantly larger number of packages, it's important to monitor performance and be ready make changes to the program to adapt to different scenarios.

## Scalability and Adaptability

The time complexity of the nearest neighbor algorithm is O(N^2), where N is the number of packages. As the number of packages grows, the time taken by the algorithm grows quadratically. This means that for a small number of packages, the algorithm is quick, but as more packages are added, it will start to slow down significantly. In addition, the algorithm is deterministic, which means that it will produce the same route for the same set of packages. If more packages are added it might not produce the most optimal route. As more packages are added, real world constraints become more complex. For example, in addition to the current delivery windows different days packages must be delivered and delivery windows at 11:00 AM or 4:30 PM, additional package constraints(one is very large and takes up a lot of space), etc. The current implementation might need further tweaking or additional constraints to handle these additional requirements.

# 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

# Individual packages with their delivery relevant information to be represented as HashMap entries  
# Will be the values in key\_value pair to enter into Hash Map  
class HashMapEntry:  
 """  
 Class to represent a HashMap entry to be used in a HashMap.  
  
 Attributes:  
 package\_id (int): The ID of the package.  
 address (str): The address of the package.  
 city (str): The city of the package.  
 state (str): The state of the package.  
 zipcode (str): The zipcode of the package.  
 delivery\_deadline (str): The delivery deadline of the package.  
 mass (str): The mass of the package.  
 special\_notes (str): Any special notes for the package.  
 delivery\_status (str): The delivery status of the package.  
 """  
  
 def \_\_init\_\_(self, package\_id, address, city, state, zipcode, delivery\_deadline, mass, special\_notes, delivery\_status):  
 """  
 Initialize a package object.  
  
 Args:  
 package\_id (int): The ID of the package.  
 address (str): The address of the package.  
 city (str): The city of the package.  
 state (str): The state of the package.  
 zipcode (str): The zipcode of the package.  
 delivery\_deadline (str): The delivery deadline of the package.  
 mass (str): The mass of the package.  
 special\_notes (str): Any special notes for the package.  
 delivery\_status (str): The delivery status of the package.  
  
 Raises:  
 TypeError: If any of the arguments have an invalid type.  
  
 Returns:  
 None  
 """  
 if not isinstance(package\_id, int):  
 raise TypeError('Package ID must be an integer')  
 if not isinstance(address, str):  
 raise TypeError('Address must be a string')  
 if not isinstance(city, str):  
 raise TypeError('City must be a string')  
 if not isinstance(state, str):  
 raise TypeError('State must be a string')  
 if not isinstance(zipcode, str):  
 raise TypeError('Zipcode must be a string')  
 if not isinstance(delivery\_deadline, str):  
 raise TypeError('Delivery deadline must be a string')  
 if not isinstance(mass, str):  
 raise TypeError('Mass must be a string')  
 if not isinstance(special\_notes, str):  
 raise TypeError('Special notes must be a string')  
 if not isinstance(delivery\_status, str):  
 raise TypeError('Delivery status must be a string')  
  
 self.package\_id = package\_id  
 self.address = address  
 self.city = city  
 self.state = state  
 self.zipcode = zipcode  
 self.delivery\_deadline = delivery\_deadline  
 self.mass = mass  
 self.special\_notes = special\_notes  
 self.delivery\_status = delivery\_status

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Worst Case Space Complexity | Worst Case Time Complexity |
| 23 | Initialize package object with details | \_\_init\_\_ | O(1) | O(1) |
| 74 | String representation of package details | \_\_repr\_\_ | O(1) | O(1) |
| Total |  |  | O(1) | O(1) |

## Task 2 A. 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.

class HashMap:  
 """  
 Class to represent a HashMap to store package information.  
  
 Attributes:  
 map (list): A list of buckets.  
 """  
  
 def \_\_init\_\_(self, initial\_size=10):  
 """  
 Initialize a HashMap.  
  
 Parameters:  
 initial\_size (int): The initial size of the HashMap.  
 """  
 # Initialize HashMap with empty buckets entries i.e. list  
 self.map = []  
 for i in range(initial\_size):  
 self.map.append([])

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Worst Case Space Complexity | Worse Case Time Complexity |
| 92 | Initialize an empty hash map | \_init\_\_ | O(N) | O(N) |
| 105 | Compute the hash bucket for a given key | get\_hash | O(1) | O(1) |
| 119 | Insert a new key-value pair into hash map | insert\_package | O(N) | O(N) |
| 154 | Retrieve value using a given key | get\_value\_from\_key | O(1) | O(N) |
| 175 | Retrieve key using given address | get\_address\_from\_key | O(1) | O(N) |
| 194 | Retrieve address using given key | get\_key\_from\_address | O(1) | O(N) |
| 210 | Retrieve the entire hash map | get\_hashmap | O(1) | O(1) |
| 216 | Retrieve packages in the hash map | get\_packages | O(1) | O(1) |
| 223 | Print details of all packages | print\_all\_packages | O(1) | O(N) |
| 242 | Update value for a given key | update\_key\_value\_pair | O(1) | O(N) |
| 266 | Delete key-value pair using given key | delete\_key\_value\_pair | O(1) | O(N) |
| 291 | Load CSV data into the hash map | load\_hash\_map | O(N) | O(N) |
| 338 | Print details of all packages for testing | print\_hash\_map\_all | O(1) | O(N) |
| 352 | 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 | Worst Case Space Complexity | Worst Case 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 | Worst Case Space Complexity | Worst Case 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) |
| 158 | Reset distance of truck | reset\_distance | O(1) | O(1) |
| 181 | Print filtered packages during delivery | print\_filtered\_packages | O(1) | O(N) |
| 188 | Get list of filtered packages during delivery | get\_filtered\_packages | O(1) | O(1) |
| 195 | Print route taken by truck | print\_route | O(1) | O(1) |
| 210 | Get number of packages on truck | get\_package\_count | O(1) | O(1) |
| 224 | Load all trucks | load\_trucks | O(1) | O(N) |
| 256 | Find the shortest route for each truck | find\_shortest\_route\_to\_deliver | O(N) | O(N^2) |
| 306 | Deliver packages for truck 1 and 2 | deliver\_packages | O(N) | O(N^2) |
| 368 | Deliver packages for truck 3 | deliver\_truck3\_packages | O(1) | O(N^2) |
| 416 | Print all packages on all three trucks | print\_all\_package\_status\_delivery | O(1) | O(N) |
| 437 | Print all individual packages on each truck | printr\_truck\_delivery\_status | O(1) | O(N) |
| 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 | Worst Case Space Complexity | Worst Case Time Complexity |
| 18 | Initialize TimeTracker object with package and truck delivery information | \_\_init\_\_ | O(1) | O(1) |
| 48 | Reset miles when running simulation multiple times | reset\_truck\_current\_time | O(1) | O(1) |
| 58 | Get speed of truck | get\_truck\_speed | O(1) | O(1) |
| 64 | Get miles traveled for truck | get\_miles\_traveled | O(1) | O(1) |
| 71 | Get all package statuses for truck | get\_package\_status | O(1) | O(1) |
| 81 | Set the time of truck 3 after truck 1 finishes delivery | set\_truck\_current\_time | O(1) | O(1) |
| 100 | Look up individual package status during delivery | lookup\_single\_package\_status | O(1) | O(N) |
| 133 | Update status of packages in TimeTracker package\_status dictionary | update\_package\_status | O(1) | O(1) |
| 177 | Print status of packages | print\_all\_package\_status | O(1) | O(N) |
| 198 | Initialize multiple package statuses | initialize\_multiple\_package\_status | O(1) | O(N) |
| 223 | Update time to start delivery of truck | update\_time\_to\_start\_delivery | O(1) | O(N) |
| 237 | Get single package status | get\_single\_package\_status | O(1) | O(1) |
| 241 | Increment time of truck when delivering packages | increment\_current\_truck\_time | O(1) | O(1) |
| 254 | Get current time of truck | get\_current\_truck\_time | O(1) | O(1) |
| 267 | Update miles traveled of truck when delivering packages | update\_miles\_traveled | O(1) | O(1) |
| 283 | Calculate total miles traveled of single truck | calculate\_total\_miles\_traveled | O(1) | O(1) |
| 296 | Calculate travel time minutes | calculate\_travel\_time\_minutes | O(1) | O(1) |
| 310 | Update truck time when delivering packages | update\_current\_truck\_time | O(1) | O(1) |
| 330 | Insert time of delivery with package | insert\_current\_truck\_time\_to\_package | O(1) | O(1) |
| 346 | Check if trucks are ready for delivery | is\_ready\_to\_deliver | O(1) | O(N) |
| 361 | Check if truck’s delivery is complete | is\_delivery\_completed | O(1) | O(1) |
| 378 | Dynamically set package status depending on time range | get\_filtered\_packages\_by\_time\_range | O(N) | O(N) |
| 428 | Format package information during delivery | format\_ouput | O(1) | O(1) |
| Total |  |  | O(N) | O(N) |

## timeutil

The time\_util consist of various functions to help with managing logic of tracking time during delivery.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line Number | Summary | Method | Worst Case Space Complexity | Worst Case 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) |
| 73 | Add time to datetime.time object | time\_plus\_delta | O(1) | O(1) |
| Total |  |  | O(1) | O(1) |

## loadutil

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 | Worst Case Space Complexity | Worst Case Time Complexity |
| 12 | Get packages to load | get\_all\_packages\_to\_load | O(N) | O(N) |
| 34 | Check if packages can be loaded for current truck by constraints delivery deadline, special notes | has\_load\_packages | O(1) | O(1) |
| 82 | Sort packages by distance from current vertex in graph | sort\_packages\_by\_distance | O(1) | O(N log N) |
| 101 | Check if package has any constraints; special notes | has\_package\_constraints | O(1) | O(K) |
| 124 | Randomize order of packages list | randomize\_packages | O(1) | O(N log N) (Shuffle) |
| 139 | Load packages onto trucks based on constraints in less algorithmic approach | \*load\_packages\* | O(1) | O(N^2) |
| 228 | Get left over packages remaining by tracking packages already loaded | get\_left\_over\_packages | O(N) | O(N) |
| 254 | Load left over packages onto trucks | load\_left\_over\_packages | O(1) | O(N) |
| 315 | Return list of packages that meet low priority truck constraints | \*get\_package\_deadline\_constraints\_low\_asc | O(n) | O(N^2) |
| 348 | Returns list of packages that meet medium priority truck constraints | \*get\_package\_deadline\_constraints\_med\_asc | O(N) | O(N^2) |
| 381 | 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 15 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. Handle Packages Meeting Constraints
   * + - 1. Prioritize loading of constrained packages for truck High Priority(truck1) and Medium Priority(truck2) are met.

if truck == 1 or truck == 2:

for package in constrained packages:

current\_vertex = package.address

truck.insert\_packages(package)

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

constrained\_packages.remove(package)

1. Handle 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 15 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() < 15:

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:

break

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

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

## 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 with the unique route.
   3. Set the best route as the current route.

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

current\_route = unique\_route

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

# Task2 C2. Process and Flow

Create hash map and load it.

A screen shot of a computer

Description automatically generated

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.

A screen shot of a computer code

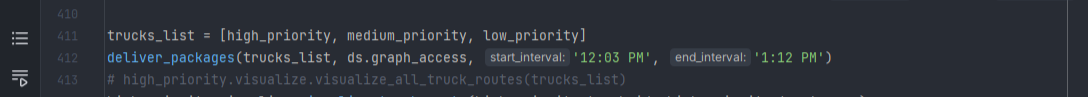
Description automatically generated

Create three trucks, load packages onto trucks and initialize packages to start tracking delivery. Where load\_trucks method implements the nearest neighbor algorithm to load packages onto trucks. And initializing the status of all packages to ‘AT\_HUB’ and start time to begin delivery tracked in the TimeTracker class.

A screenshot of a computer code

Description automatically generated

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.



Inside the deliver\_packages method our two-opt algorithm implementation is used to futher optimize the route and finally find\_shortest\_route\_to\_deliver is called to calculate the final distance using dijkstra’s shortest path algorithm on the optimized route after applying nearest neighbor and two-opt algorithm for each truck. As well as update the time delivered for each truck and filter the packages to be shown during specific time intervals of delivery calling methods from the TimeTracker class.

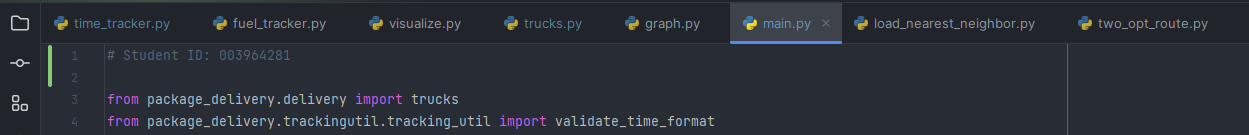
# Calculate the shortest route to deliver packages to destination  
# and back to hub after applying Dijkstra's algorithm  
def \_find\_shortest\_route\_to\_deliver(truck, graph):  
 """  
Find the shortest route for a truck to deliver packages.  
  
 Parameters:  
 truck (Truck): The truck object representing the delivery truck.  
 graph (Graph): The graph object representing the delivery network.  
  
 Returns:  
 int: The total distance traveled by the truck to deliver all the packages.  
 """  
 # Make a copy of the packages on the truck  
 packages\_copy = truck.get\_packages().copy()  
 # Starting location for all trucks, i.e., the hub  
 start\_vertex1 = truck.route[0]  
 # Initialize total\_distance traveled by truck to 0  
 total\_distance = 0  
 # Calculate the shortest distances and pred\_vertex using Dijkstra's algorithm of the truck's route  
 distances1, pred\_vertex1 = algo.dijkstra(graph, start\_vertex1, truck.route)  
 # Use a for loop to iterate over the packages  
 for package in packages\_copy:  
 # Vertex to travel to  
 dest\_vertex = package.address  
  
 # Insert the calculated distances and pred\_vertex into the Trucks object  
 truck.insert\_distances\_pred\_vertex(distances1[dest\_vertex], pred\_vertex1[dest\_vertex])  
 # Update time during delivery  
 time\_delivered = truck.time\_tracker.update\_current\_truck\_time(distances1[dest\_vertex])  
 # Insert the time\_delivered into package  
 truck.time\_tracker.insert\_current\_truck\_time\_to\_package(package, time\_delivered)  
 # Update the time for other packages with the same address  
 for other\_package, status\_info in truck.time\_tracker.get\_package\_status.items():  
 if (other\_package != package and other\_package.address == package.address and status\_info['status'] !=  
 'DELIVERED'):  
 truck.time\_tracker.insert\_current\_truck\_time\_to\_package(other\_package, time\_delivered)  
 # Skip adding the distance if the next package is already at the dest\_vertex, share addresses  
 if distances1[dest\_vertex] != 0:  
 total = distances1[dest\_vertex]  
 total\_distance += total  
  
 return total\_distance

# User Interface and Status Check

Note: If the values in the 'All Packages' or ‘Truck 1-3’ screenshots are not discernible from the current screenshots, additional screenshots are available in the 'Status\_Checks' directory of 'Package\_Delivery\_Program\_New'.

## Task 2 C1. Identification Information

Main.py with student ID at top of file



## Task 2 D. User Interface

Running Delivery Simulation:

Firstly, choose option 3 to load packages onto the trucks, as this is a prerequisite for running the simulation. Next, opt for choice 4 to access the delivery submenu and start the delivery simulation. Note that the output is organized into four sections:

* While running delivery simulation, the first three sections of print output display the status of each individual package status on their respective trucks and the total distance, number of packages loaded, and route traveled from the hub of each truck.
* The final section provides an overview of all 40 package statuses and the cumulative distance traveled by all three trucks during the delivery simulation.

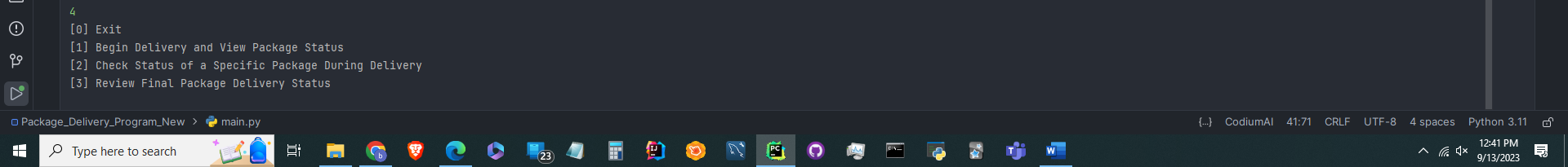
Main UI:

A screenshot of a computer program

Description automatically generated

* Select 1 to see all packages and their relevant information and check if packages are all present loaded from WGUPS Package File Formatted.csv.
* Select 2 to see single package data populated in hash map of specified package ID acting as key.
* Select 3 to navigate to delivery submenu to run delivery simulation.
* Select 4 to see visualization of delivery simulation(not required for our approach in solving the stated problem).

Delivery Submenu:



1. Selecting choice 1, make sure to choose time frames to see status of delivery, use 12-hour format for input(i.e. 9:35 AM). Can continue with different time frames by user.
2. Selecting choice 2, see individual package statuses during delivery on user Package ID and desired time.
3. Selecting choice 3,shows all packages delivered at end of day, from 8:00 AM- 5:00PM.

Single Look Up:

A screenshot of a computer

Description automatically generated

1) Select package ID

2) Use 12-hour format as mentioned above to see status during the specificed time

## Task 2 B. 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.

A screen shot of a computer code

Description automatically generated

## Status Check

### Task 2 D2. First Status Check 8:35 AM – 9:25 AM

Truck 1

A blurry image of a computer screen

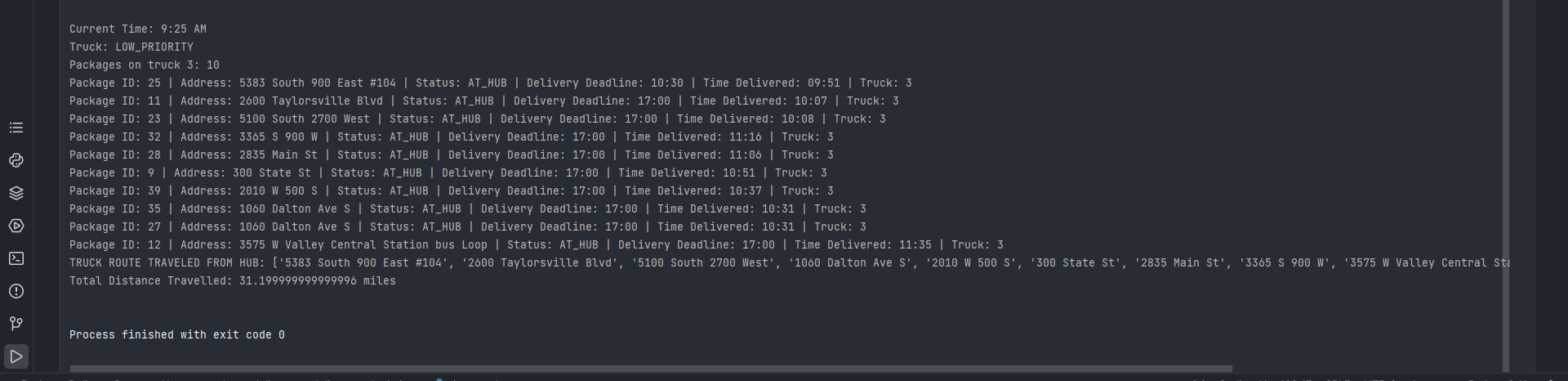
Description automatically generated

Truck 2

A blurry image of a square

Description automatically generated

Truck 3



All Packages

A screenshot of a computer

Description automatically generated

### Task 2 D3. Second Status Check 9:35 AM – 10:25 AM

Truck 1

A computer screen with a city in the background

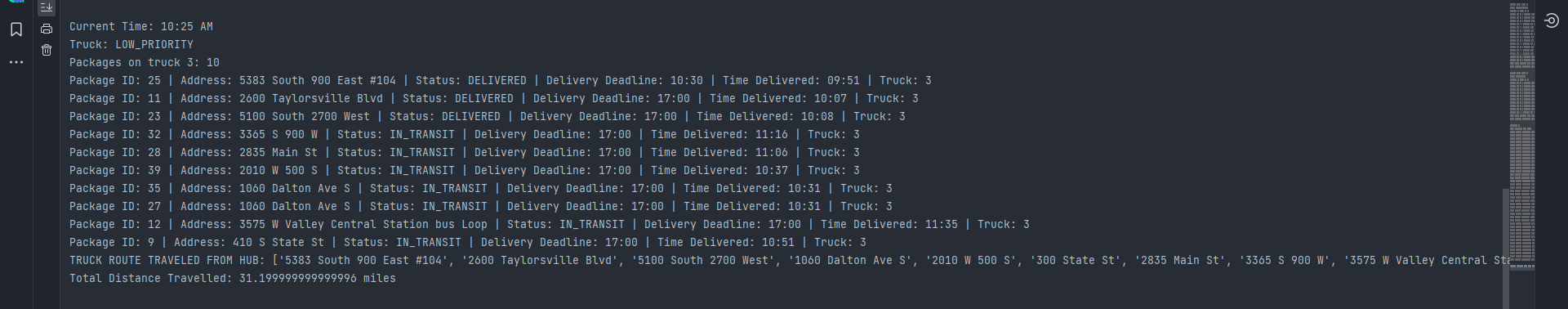
Description automatically generated with medium confidence

Truck 2

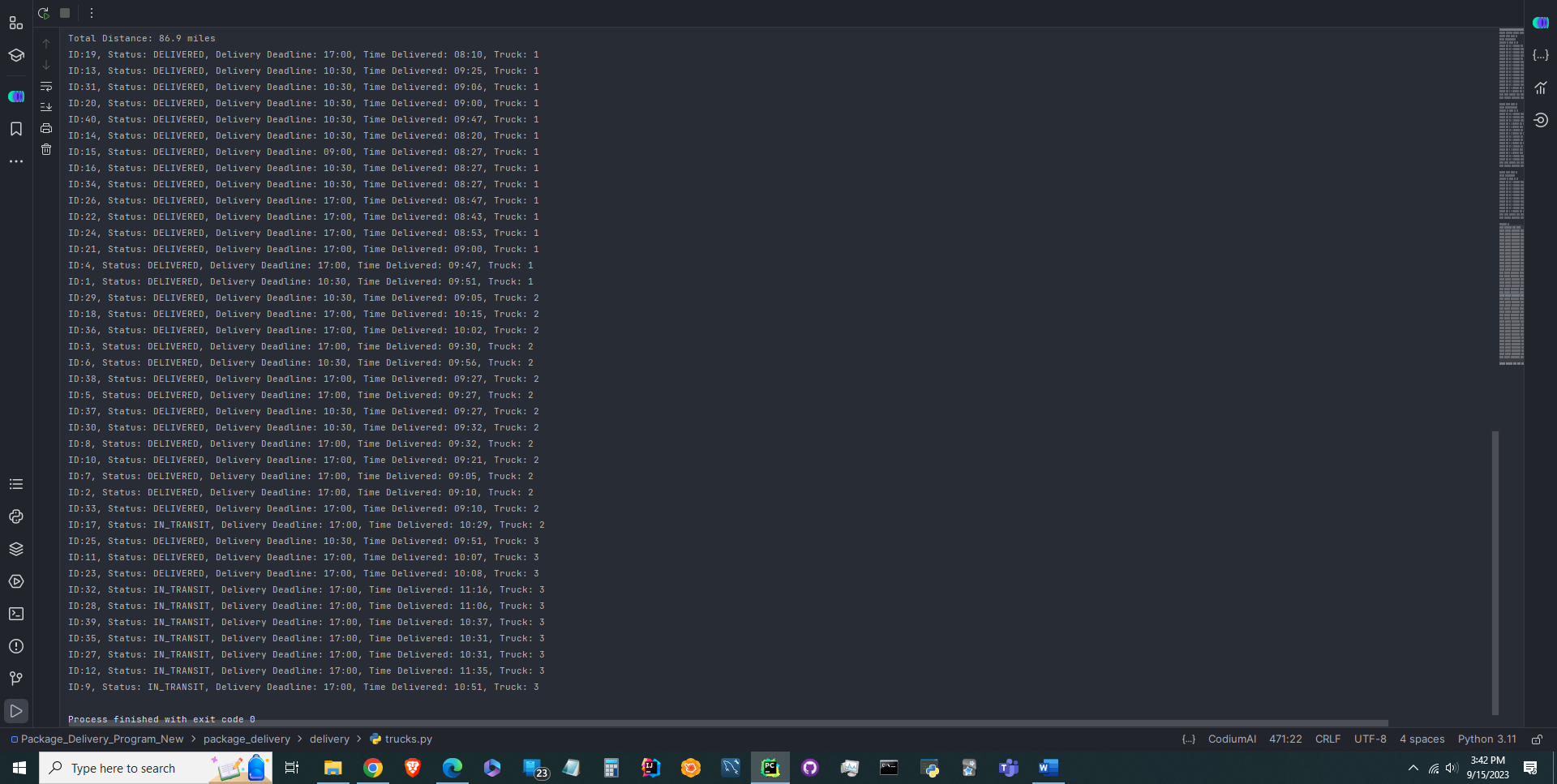
A pixelated image of a computer screen

Description automatically generated

Truck 3



All Packages



### Task 2 D4. Third Status Check 12:03 PM – 1:12 PM

Truck 1

A blurry image of a building

Description automatically generated

Truck 2

A pixelated image of a computer screen

Description automatically generated

Truck 3

A black background with many small squares

Description automatically generated with medium confidence

All Packages

A screenshot of a computer

Description automatically generated

# Data Structure and Algorithm Evaluation

## Data Structures Evaluation

### Task 2 H. HashMap Verification

The hash map is designed to manage the details of package data, populated by extracting data from the WGUPS Package File Formatted.csv file. It uses the package ID as the key and the associated package details as the value. In addition, the TimeTracker class is tasked with monitoring package statuses and handling time-related logistics. By separating these functionalities, the TimeTracker focuses solely on the dynamic elements of package delivery, such as each package's delivery time and updating status. Every package is characterized by attributes including its delivery address, deadline, city, zip code, weight, special notes, status, assigned truck, delivery start time, and actual delivery time. Through the user interface, by selecting the first option from the main menu, users can confirm the display of all 40 packages, complete with their specifications.

### Strengths of Chosen Data Structure

In our implementation the key will be the package id of each package. Every package in our scenario will have a unique ID. By using unique keys, we can effectively avoid overwriting data and ensure that each package's information is preserved distinctly. Because of the nature of our delivery program needing to access key-value pairs quickly. Instant retrieval of package details can be accessed without having to search for multiple attributes, instead it directly accesses the package details using the unique identifier of each package. As our delivery program grows and more packages are added, the significance of efficient data access becomes even more pronounced. By using package ID as a key, the system's performance remains consistent, regardless of the number of packages as deletion, insertion, and lookup has an average time complexity of O(1). This ensures that data retrieval and access remain efficient.

### Weakness of Chose Data Structure

A challenge with hash maps is collisions, where two keys have the same hash value. In our implementation handles collision using chaining but is still a concern when the load factor is high. In a scenario where there is large amount of chaining, it can lead to additional space overhead making the data structure less space efficient. Lastly, hash maps do have inherently maintain data order. For example, if the sequence of packages upon insertion is important.

### Task 2 H. 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.)). As shown by running the delivery simulation distance is calculated and the total distance of all three trucks is below the limit of 140 miles.

### Task 2 H1. 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.

### Task 2 H1a. 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.

## Core Algorithm Evaluation

### Task 2 F2. 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 load packages onto trucks and plan their routes. All 40 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 as total distance traveled is 89 miles and no packages are delivered past their delivery deadline.

### Task 2 F1. Strength of the Chosen Algorithm

The nearest neighbor algorithm is straightforward and easy to understand. It follows a simple principle: from the current location, go to the nearest unvisited neighbor. This simplicity makes it a good starting point in solving our scenario. Unlike other algorithms, the nearest neighbor algorithm provides consistent results. Given the same set of packages, it will produce the same route, makings its behavior predictable. For our scenario where there is a small data set, this allows us to compute a route quickly, where quick decisions are required. All in all, the nearest neighbor algorithm serves as an excellent baseline. It allows for easy comparison with more advance algorithms, to help in gauging their effectiveness.

## Alternative Algorithm Approaches

### Task 2 F3. 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, especially as more packages are added.

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. The time complexity is roughly O(M x N^2) where M is the number of generations and N is the number of packages. This algorithm can handle a larger number of packages but will still slow down as the number of packages increases. Particularly useful for more complex delivery networks and constraints as it allows for comprehensive search mechanism to pinpoint the best routes.

### Task 2 F3a. 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.

# 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.

# Task 2 G. 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()

# Task1 S, Task 2 S. 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>