



**Artificial Intelligence**  
**ENCS3340**

**Project Report**

---

**Name: Nagham Maali**

**ID: 1212312**

**Sec.: 2**

**Instructor: Dr. Samah Alaydi**

## **Contents**

<b>PROJECT OVERVIEW:</b>	<b>3</b>
<b>USED TECHNOLOGY:</b>	<b>4</b>
<b>BALANCE HANDLING:</b>	<b>6</b>
<b>PROBLEM FORMULATION:</b>	<b>9</b>
<b>INPUT:</b>	<b>11</b>
<b>IMPLEMENTATION:</b>	<b>12</b>
<b>OUTPUT:</b>	<b>37</b>
<b>TESTING AND VERIFICATION:</b>	<b>39</b>
<b>RESULT AND CONCLUSION:</b>	<b>48</b>

## **Project overview:**

This project focuses on optimizing the daily operations of a small local package delivery shop. Every day, the shop receives multiple packages, each with a destination, a weight, and a priority level (1 to 5, where 1 is the highest priority). The goal is to assign these packages to a limited number of delivery vehicles and plan the delivery routes in a way that reduces the total distance traveled, while also giving preference to delivering high-priority packages earlier.

Each vehicle has a limited carrying capacity, so the assignment must make sure that no vehicle is overloaded. The challenge is to find a good balance between two goals: minimizing the total travel distance (to reduce cost) and delivering high-priority packages as early as possible. This balance is important to ensure both efficiency and quality of service.

To solve this, two optimization algorithms are used: Simulated Annealing and Genetic Algorithms. The program takes input through a graphical interface, where the user provides two files—one containing vehicle information, and the other containing package details. The system then loads the data, processes it, and displays an optimized delivery plan.

In short, this project simulates real-world delivery logistics and applies AI techniques to find smart, practical, and balanced solutions for package delivery.

## Used technology:

This project was developed using Java along with JavaFX for the graphical user interface. Scene Builder was used to design and organize the interface through .FXML files, allowing for a clean separation between the layout and the logic.

The user interface includes multiple scenes, starting with a Cover Page featuring the app's name "Delivra" and a visually appealing background. The interface uses a consistent color theme of #001D5E (dark blue), #E85E2C (orange), #35E5F3 (cyan), and White, creating a modern and engaging look. Visual elements like images (Cover\_Page.png, vehicle.jpg) and styled buttons, labels, and text fields are used to improve user experience.

Users interact with the system by entering the names of two .txt files (one for vehicles and one for packages). A second scene lets users choose between the Simulated Annealing and Genetic Algorithm methods. Once a method is selected, the system computes the optimized routes scene, and the Output Page visualizes the solution dynamically. Horizontal and vertical zoom sliders allow users to explore the canvas in detail.

To build this system, several Java libraries were used, including:

- `javafx.application.Application`
- `javafx.fxml.FXMLLoader`
- `javafx.scene.control` components like `TextField`, `Button`, and `Alert`
- `javafx.scene.image.Image` and `ImageView` for visuals

- `java.util.*` and `java.io.*` for file handling and data structures
- `javafx.animation.PauseTransition` and `javafx.util.Duration` for handling delays and transitions
- `javafx.scene.canvas.*`

The combination of JavaFX and Scene Builder allowed for a user-friendly, colorful interface that helps users interact smoothly with the system.

Java was chosen for this project due to its strong object-oriented capabilities, reliability, and performance in building desktop applications. JavaFX, along with Scene Builder, provides a flexible and visually rich framework for creating modern user interfaces, making it ideal for building a multi-scene application like this one. The combination allows for clean separation between logic and design, easy event handling, and efficient file processing.

## **Balance handling:**

In both the Simulated Annealing and Genetic Algorithm implementations, I designed a custom cost function to guide the optimization process toward solutions that balance operational cost with delivery priorities and vehicle load usage. This function is central to maintaining the trade-off between delivering high-priority packages and minimizing total distance traveled.

The total cost of a solution is calculated using three key components:

- **Route Distance** – the total kilometers traveled by each vehicle, from the shop to all assigned package destinations and back.
- **Priority Penalty** – a penalty that increases when high-priority packages are delivered later in the route.
- **Load Balance Penalty** – a penalty based on how much unused capacity a vehicle has; encourages efficient use of vehicle load.

For each vehicle, the algorithm:

1. Calculates the Euclidean distance traveled (including return to the shop).
2. Applies a priority penalty, computed as the sum of (priority  $\times$  delivery order index). This discourages delivering important packages last.
3. Applies a load balance penalty, calculated as the absolute difference between the vehicle's capacity and the total

weight it carries. This penalizes underutilized or overloaded vehicles.

- ✓ These penalties are scaled with coefficients (0.1 for priority, 0.05 for load balance) to keep them proportional and meaningful without overpowering the route cost.

The reason I chose this cost function design is to strike a balance between different goals—mainly minimizing travel distance, respecting package priority, and using vehicle capacity wisely. While reducing the total route distance is the main objective, I didn't want the algorithm to ignore priority altogether. That's why I added a priority penalty: it gently encourages the system to deliver high-priority packages earlier without forcing it. Sometimes, it may still delay a high-priority package if doing so greatly reduces the total travel distance, which is actually closer to a realistic and practical solution. The load penalty was added to avoid situations where one vehicle is overloaded and another is nearly empty. Instead, the algorithm tries to fill each vehicle closer to its full capacity, but only when it's smart to do so in terms of total cost.

The values I used (0.1 for the priority penalty and 0.05 for the load balance penalty) were chosen after experimenting and observing how the algorithm behaves. I wanted the route distance to remain the most important factor, so the penalties had to be smaller but still strong enough to influence the decision-making. If the priority penalty was too high, the algorithm would start making poor decisions just to deliver important packages first. If it was too low, it would completely ignore them. The same applies to the load balance: a gentle

penalty (0.05) helps keep the load reasonable without making the algorithm obsess over perfect balance. This mix leads to more flexible, realistic, and high-quality solutions, especially in real-world delivery scenarios where perfect optimization isn't always practical.



## **Problem formulation:**

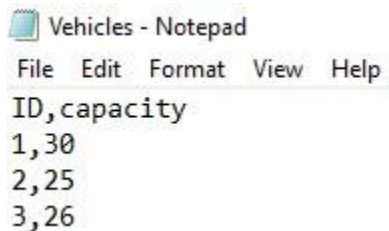
- **Initial State:** The system starts with a list of available delivery vehicles (each with a known capacity) and a list of packages (each with a destination, weight, and priority).
- **Actions:** Assigning packages to vehicles and determining the optimal delivery route for each vehicle.
- **State Space:** All possible combinations of package assignments and delivery routes for the given vehicles.
- **Goal State:** A complete assignment of packages to vehicles along with delivery routes that minimize operational cost while balancing priority delivery.
- **Goal Test:** Checks if all packages are assigned within vehicle capacity limits and whether a valid route is generated for each vehicle.
- **Path:** A specific sequence of package assignments and route decisions taken from the initial state to reach a solution.
- **Path Cost:** The cost is a weighted combination of three factors:
  - Total distance traveled by all vehicles (using Euclidean distance)
  - Penalty for delivering high-priority packages later in the route
  - Penalty for underusing or overloading vehicle capacity
- **Solution:** The system outputs an assignment of packages to vehicles and optimized delivery routes using either Simulated

Annealing or Genetic Algorithm, based on user selection. The solution seeks a balanced trade-off between minimizing distance and delivering higher-priority packages earlier.

## Input:

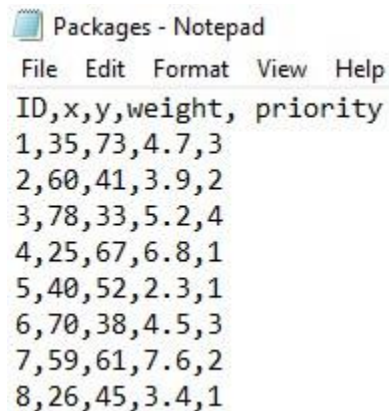
To load the data into the system, I built a user-friendly interface using JavaFX that lets the user enter the names of two input files: one containing vehicle information (with each vehicle's ID and capacity) and the other containing package details (ID, location as x and y coordinates, weight, and priority). Once the user enters the filenames through labeled text fields and submits them, the program reads the data from the files and organizes it into useful structures. Specifically, all vehicles are stored in a list of Vehicle objects, all packages are stored in a list of Package objects, and a map (`Map<Vehicle, List<Package>>`) is used to keep track of which packages are assigned to which vehicle. This approach makes it easy to access, assign, and analyze the data later during the optimization process.

### Vehicles.txt file example:



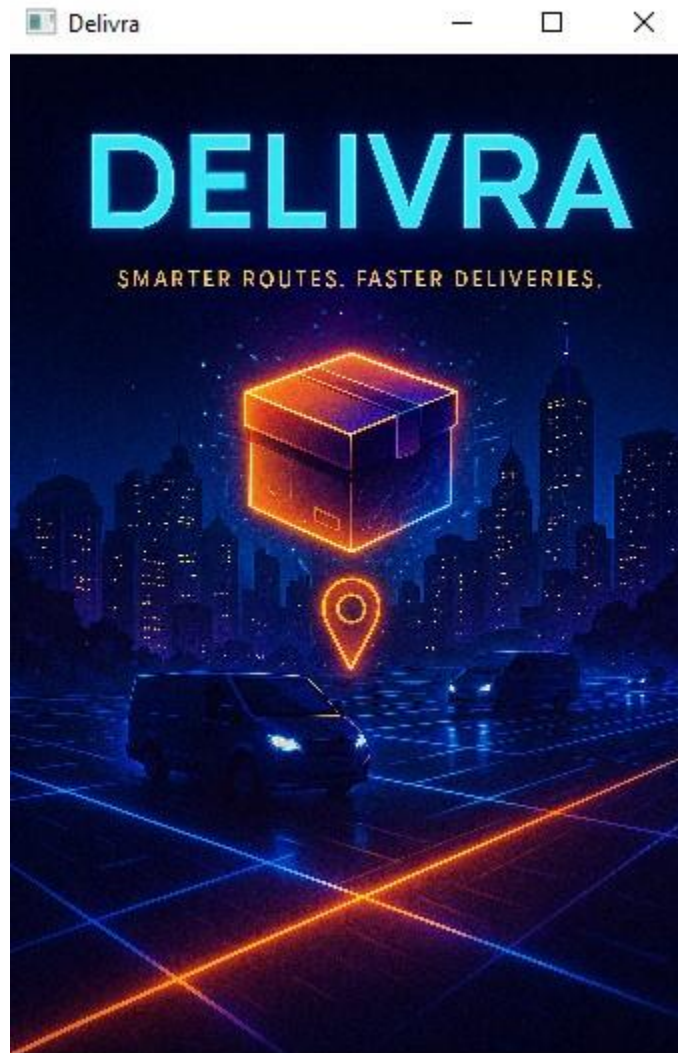
```
Vehicles - Notepad
File Edit Format View Help
ID,capacity
1,30
2,25
3,26
```

### Packages.txt file example:



```
Packages - Notepad
File Edit Format View Help
ID,x,y,weight, priority
1,35,73,4.7,3
2,60,41,3.9,2
3,78,33,5.2,4
4,25,67,6.8,1
5,40,52,2.3,1
6,70,38,4.5,3
7,59,61,7.6,2
8,26,45,3.4,1
```

## Implementation:

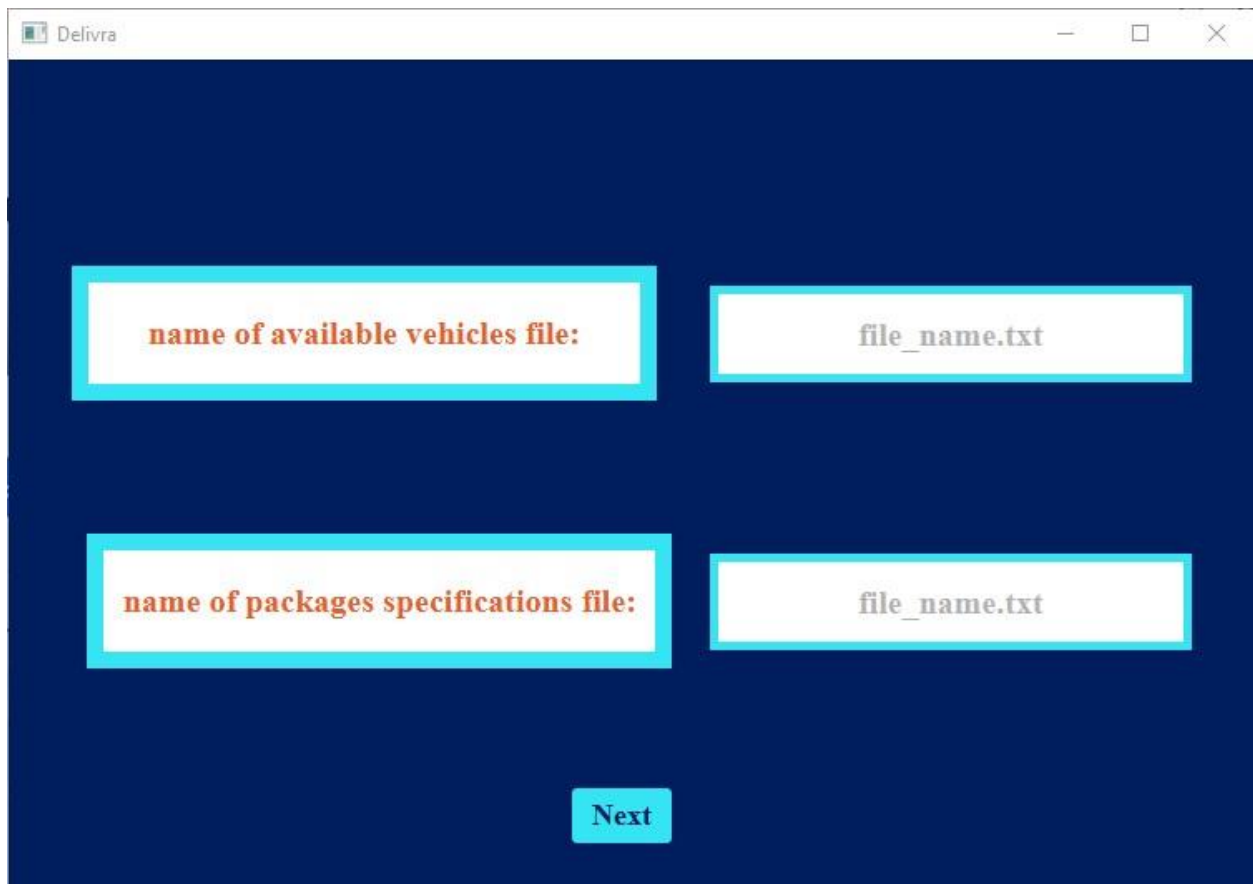


The JavaFX application starts from the Main class, which serves as the entry point. When the program runs, this class is responsible for opening the first window that the user sees. It begins by loading a file called CoverPage.fxml, which defines the layout for the welcome or splash screen. This screen is built using an AnchorPane layout and includes a background image that covers the whole window. A scene is created using this

layout and displayed on a small window (340 by 500 pixels) with the title “Delivra.”

Right after showing the splash screen, the program waits for 3 seconds using something called `PauseTransition`. This short pause is meant to give users time to view the cover page.

Once the 3 seconds are over, the application automatically switches to another scene, defined in the file `Input.fxml`, which contains the actual input form for entering file names.

The image shows a screenshot of a JavaFX application window titled "Delivra". The window has a dark blue background. It contains two rows of input fields. The first row has a label "name of available vehicles file:" in orange text inside a white box with a cyan border, followed by a text input field containing "file\_name.txt". The second row has a label "name of packages specifications file:" in orange text inside a white box with a cyan border, followed by another text input field containing "file\_name.txt". At the bottom center, there is a cyan button with the text "Next".

To do this, the code uses a loader that reads the new FXML file, replaces the current scene with the new one, and also connects the controller (`InputController`) with the main window (`Stage`) so that future events in the app can be handled properly.

The `CoverPageController` class is linked to the cover page FXML file. Its main job is to load and display an image when the cover page is shown. This is done in the `initialize()` method, which runs automatically when the scene loads. The image file path is hardcoded in the class (you can update it if needed), and if the file exists, it will be shown inside the `ImageView` on the screen. If the file is missing or the path is incorrect, an error message will appear in the console.

The FXML file `CoverPage.fxml` defines what the welcome screen looks like. It uses an `AnchorPane` layout to position elements and contains only one main component: an `ImageView`. This `ImageView` is where the cover image appears, and it's stretched to fit the full size of the window (768 by 512 pixels) while keeping its original aspect ratio. The link between the `ImageView` in the FXML and the code in the controller is made using the `fx:id` attribute, which allows the controller to access and update the image when the program runs.

This combination of the `Main` class, `CoverPageController`, and `CoverPage.fxml` creates a smooth and professional-looking start to the application, giving users a few seconds to view a branded splash screen before moving on to the input stage.

There is also `InputController.java` and `Input.FXML`, that work like a simple interface to let the user input two text file names: one file that contains vehicle data and another that contains package data. It then reads the content of these files, stores the data into lists, and moves to another scene if everything goes well.

The program is divided into two parts: the Java controller class (InputController.java) and the FXML layout file (InputScene.fxml). The controller class handles logic and user interaction, while the FXML file designs the appearance of the user interface.

At the very top of the controller class, we see package application;, which means this class is part of a Java package named application. Then, a bunch of libraries are imported. These include tools for reading files (java.io.\*, java.util.\*) and for creating the JavaFX graphical user interface (javafx.\*).

Now, inside the class InputController, there are two main variables: stage and scene, which refer to the current application window and its contents. These are used to control what is shown on the screen and to change between screens. Then, there are two TextField components that allow the user to type in file names: one for the vehicle file and one for the packages file. There is also a button called Next, which, when clicked, will run the method Next\_btn() to handle the file-loading and screen-switching.

The vehicles and packages lists are created to hold the data read from the files. Each vehicle and package will be stored as an object of a custom class named Vehicle or Package. These classes are not shown here but are expected to contain details like ID, weight, capacity, etc.

The setStage() method sets up the main stage for this screen and gives it a title “Delivra”. It is usually called from the main application class when launching the app.

There is a helper method named `displayMessage()` that shows a message box (alert window) to the user. It can either show an error message or an information message, depending on the second argument passed to it ("error" or "information").

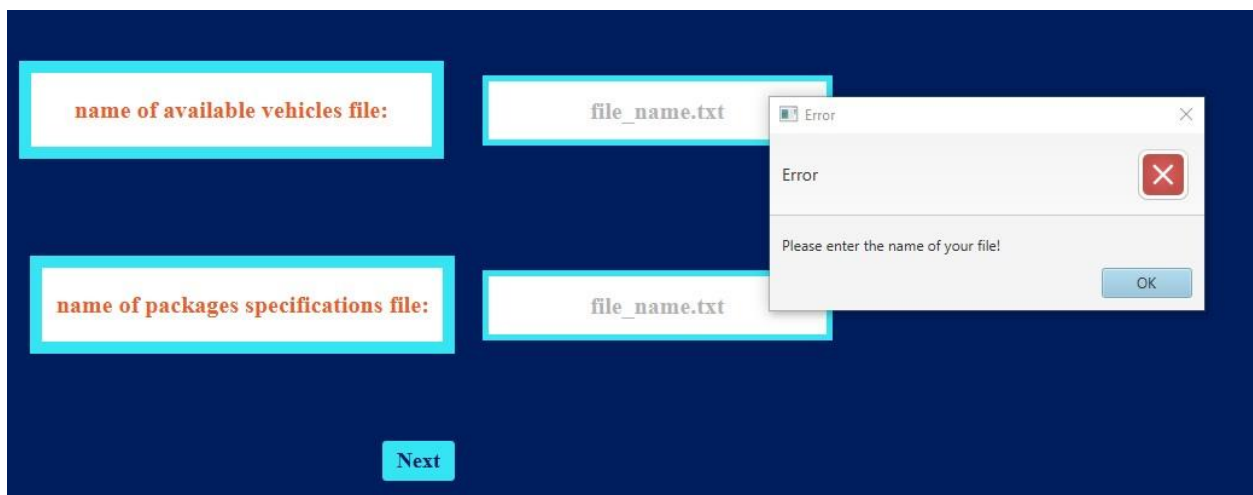
Now, the most important method is `Next_btn()`, which runs when the user clicks the "Next" button. It first reads the file names from the two text fields, removes any extra spaces, and checks if either field is empty. If they are empty, it shows an error message asking the user to enter the names. If both fields have content, it clears the existing lists of vehicles and packages to avoid duplicate data and then tries to load the new data by calling `loadVehicles()` and `loadPackages()`.

These two methods (`loadVehicles()` and `loadPackages()`) use the Java Scanner class to read lines from the file. Each line is expected to contain comma-separated values. For the vehicle file, each line should have two values (vehicle ID and capacity). For the packages file, each line should have five values (ID, X and Y coordinates, weight, and priority). The program reads each value, converts it to the correct type, and creates a new object which is added to the appropriate list.

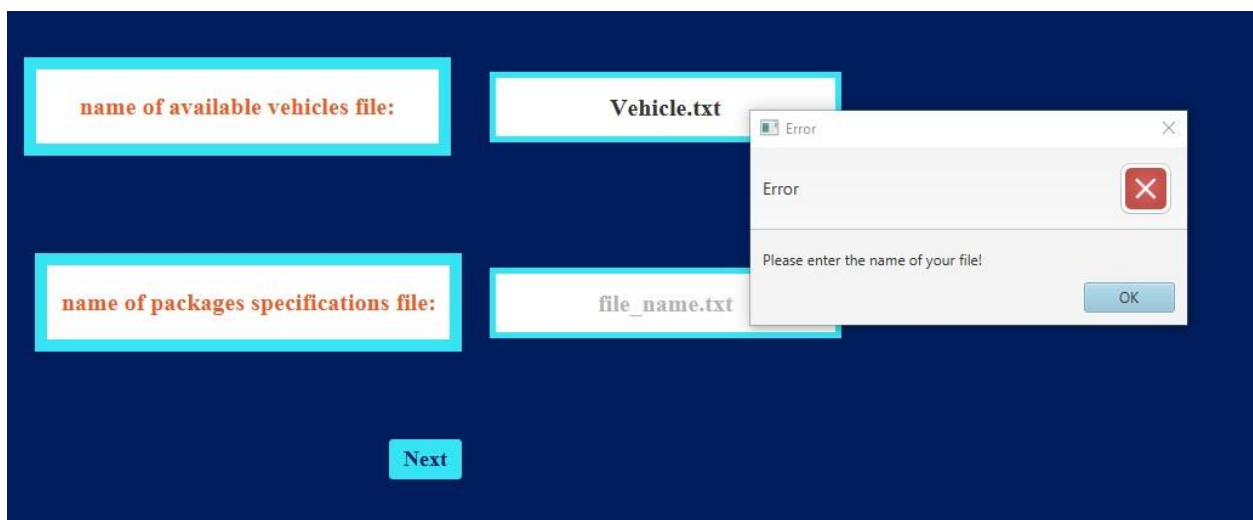
If both files load correctly, the program shows a success message saying “Delivra has your info :)” and prints the loaded data to the console for testing purposes. After that, the program prepares to move to the next scene, which is defined in a file named `Algorithms.fxml`. It loads that file, gets its controller (likely named `AlgorithmsController`), sends the loaded lists to it, and then shows the new screen.



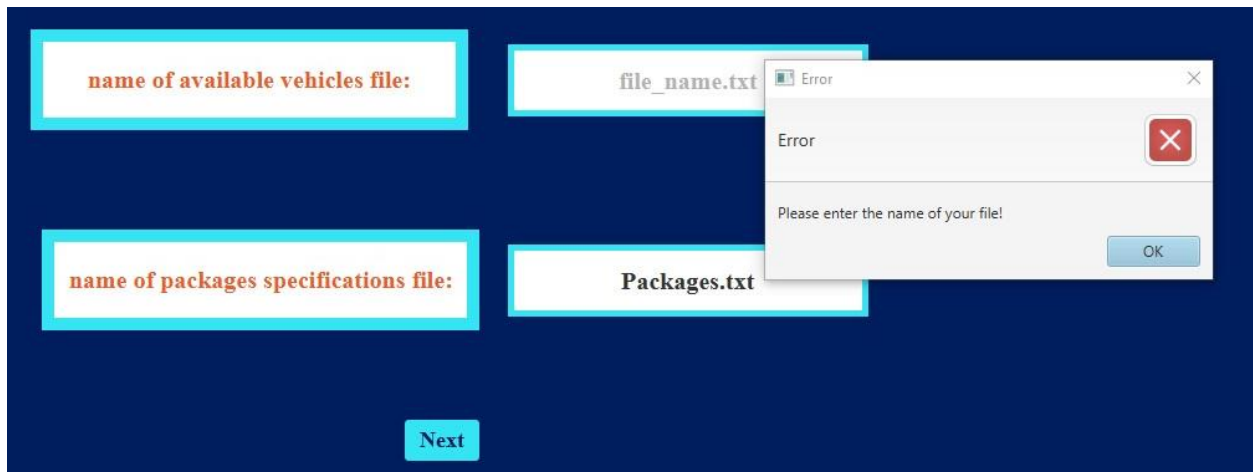
The FXML file creates the visual design of this first screen. It has a background color of dark blue (#001D5E). It contains two labels asking the user to enter the file names (one for vehicles, one for packages), two text fields for input (styled with borders and prompt text), and a button labeled “Next” which is linked to the Next\_btn() method in the controller. The layout positions are defined using values like layoutX and layoutY, and fonts are set to be bold Times New Roman.



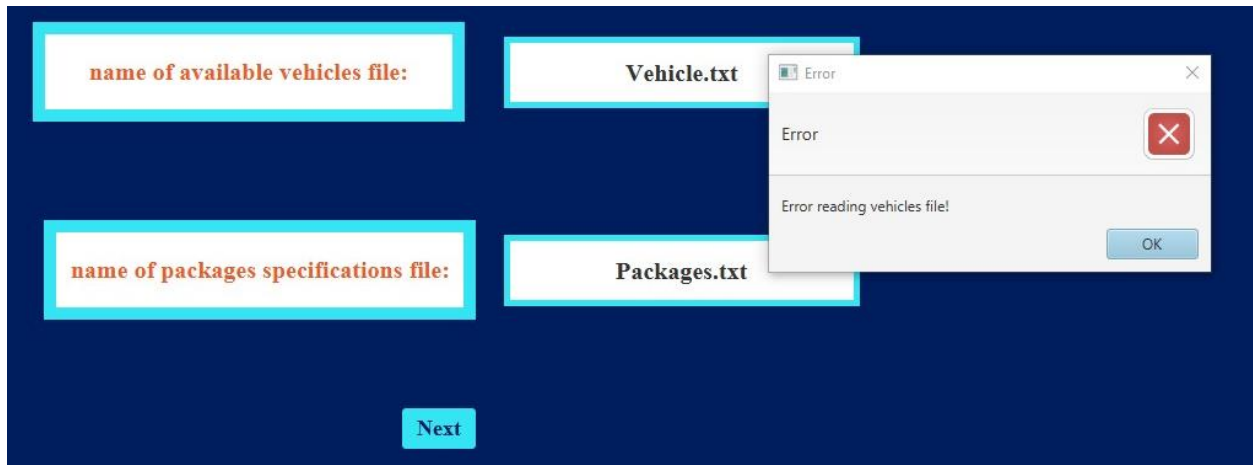
*when the user doesn't enter files names*



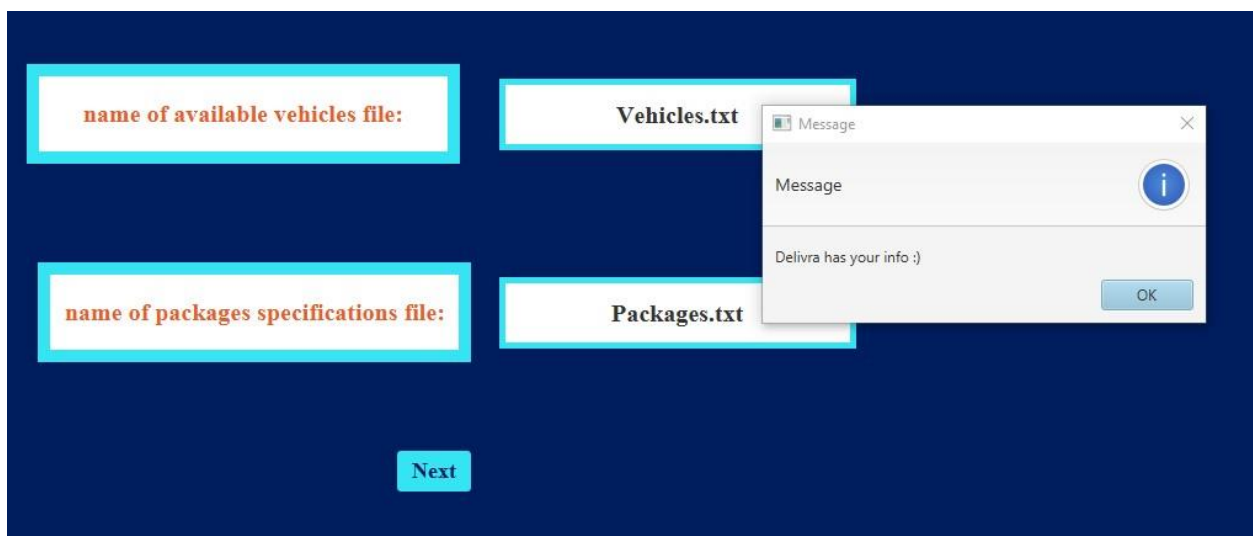
*when the user doesn't enter the Packages file*



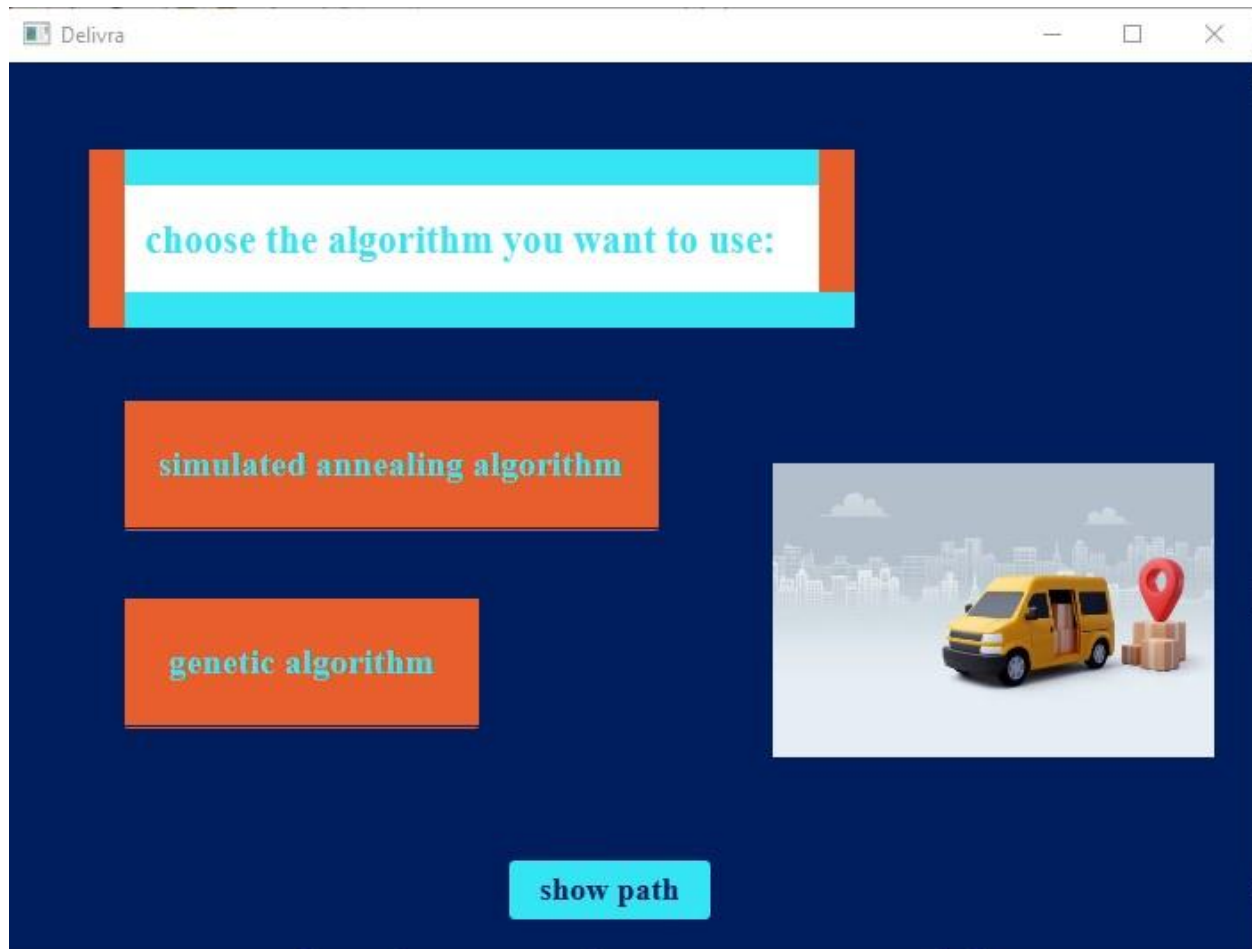
*when the user doesn't enter the vehicles file*



*when a file name or both files name is wrong*



*when both files names are correct*



This screen appears after the user has entered the input data for vehicles and packages. Here, the user chooses which algorithm they want to use to solve the delivery problem: Simulated Annealing or Genetic Algorithm. The screen also includes a button to view the generated result and a decorative image for visual appeal.

The AlgorithmsController.java file connects the user interface (FXML) with the backend logic. It defines three buttons: one for Simulated Annealing (SA\_btn), one for Genetic Algorithm (GA\_btn), and a third button called SP to display the solution. There's also an ImageView called

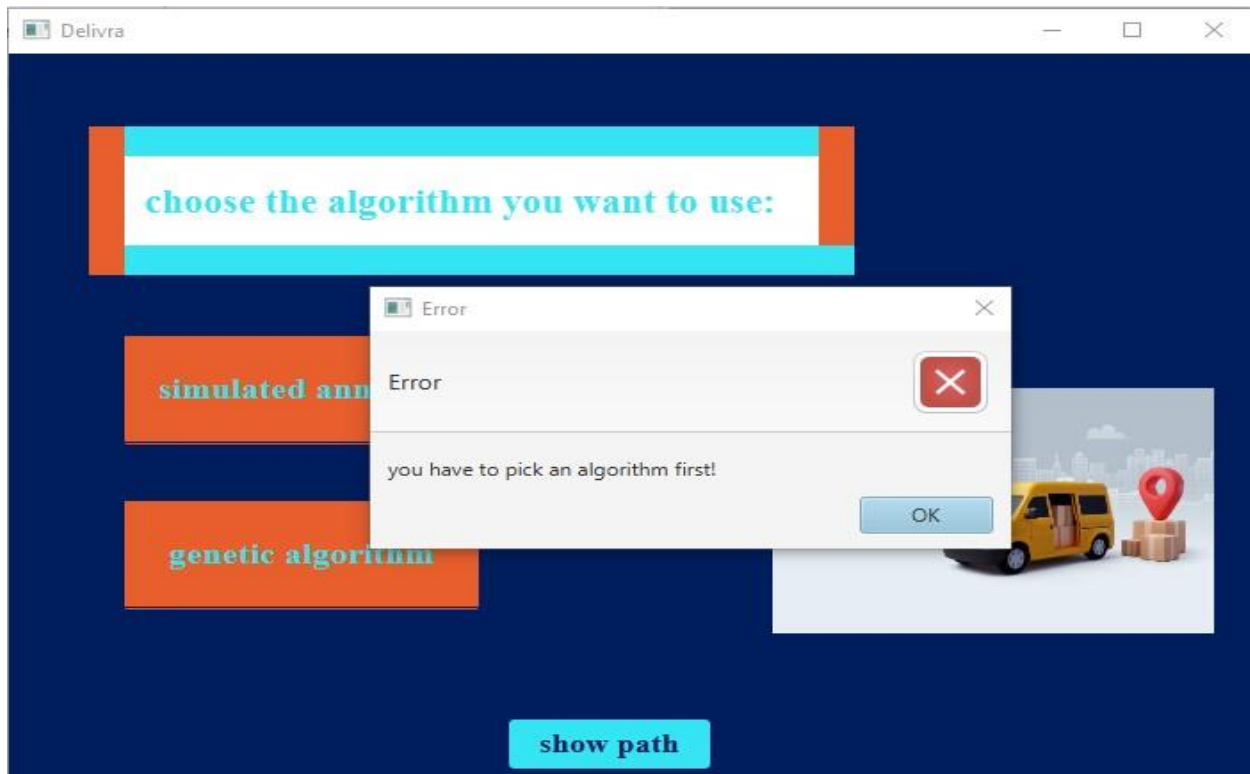
Del\_img which displays an image for design purposes. When this screen is loaded, the initialize() method is called automatically. Inside this method, the application first tries to load an image from a specific path on the computer. If the image file is found, it is shown on the interface using Del\_img. If the image is missing, an error is printed in the console. Then, button actions are defined. When the user clicks the Simulated Annealing button (SA\_btn), the program runs the SAA() method. This method calls a function from the SimulatedAnnealing class which processes the input data (vehicles and packages) and stores the result in a variable called result. It also prints the solution in the console for testing. After that, a new action is defined for the SP button. When this button is clicked, it checks if the result exists. If not, it shows an error message. If the result is valid, it loads the Output.fxml file and passes the result to the next controller (OutputController) to display the final solution on a new screen. The same steps happen in the GA() method when the user clicks the Genetic Algorithm button (GA\_btn). It runs the optimization using the Genetic Algorithm logic (from the Genetic class), stores the result, and sets up the SP button to show the output if available.

If the user clicks the SP button before selecting any algorithm, the initialize() method makes sure to show a message saying that an algorithm needs to be selected first. This prevents errors or blank outputs.

There is also a helper method called displayMessage() which shows a pop-up message on the screen. This method is used to alert the user when something goes wrong, like when the result is missing. Finally, the setStage() method allows this controller to receive and use the main application stage from the Main class. This is useful when changing scenes between different windows of the application.

The Algorithms.fxml file describes the visual layout of the scene. The background color is set to dark blue. There are two large buttons on the left: one for Simulated Annealing and one for Genetic Algorithm. Each button has a custom color style with orange background and cyan text. A third button labeled “show path” is placed at the bottom, and it’s used to go to the output screen after the solution is generated. There is also a label at the top asking the user to choose an algorithm, and an image view on the right side where the vehicle image will appear if the file is found. Together, the Java controller and the FXML file allow the user to visually interact with the two

algorithms and display their results in a clean and guided way.



*if the user clicks the button before applying one of the algorithms*

the program also prints to console for checking:

```
Simulated Annealing selected.
Vehicle ID: 1
Capacity: 30.0kg
Current Load: 29.599999999999998kg
Assigned Packages:
- Package 5 | (40.0,52.0) | 2.3kg | prio 1
- Package 7 | (59.0,61.0) | 7.6kg | prio 2
- Package 1 | (35.0,73.0) | 4.7kg | prio 3
- Package 9 | (63.0,77.0) | 8.2kg | prio 4
- Package 4 | (25.0,67.0) | 6.8kg | prio 1
Delivery Path: Shop(0,0) -> P5(40.0,52.0) -> P7(59.0,61.0) -> P9(63.0,77.0) -> P1(35.0,73.0) -> P4(25.0,67.0) -> Shop(0,0)
-----
Vehicle ID: 2
Capacity: 25.0kg
Current Load: 21.0kg
Assigned Packages:
- Package 8 | (26.0,45.0) | 3.4kg | prio 1
- Package 2 | (60.0,41.0) | 3.9kg | prio 2
- Package 6 | (70.0,38.0) | 4.5kg | prio 3
- Package 3 | (78.0,33.0) | 5.2kg | prio 4
- Package 10 | (31.0,55.0) | 4.0kg | prio 5
Delivery Path: Shop(0,0) -> P8(26.0,45.0) -> P10(31.0,55.0) -> P2(60.0,41.0) -> P6(70.0,38.0) -> P3(78.0,33.0) -> Shop(0,0)
-----
Vehicle ID: 3
Capacity: 26.0kg
Current Load: 0.0kg
Assigned Packages: (none—stays at shop)
-----
```

The Package class is a blueprint for creating package objects that will be delivered. Each package is defined by five main pieces of information: an ID (like a tag number), a location (where it's going, shown by x and y coordinates), a weight (how heavy it is), and a priority (how important it is, from 1 being most important to 5 being least). The constructor method `public Package(int Id , double x , double y , double weight , int priority)` is used when we want to create a new package and fill in all these values at once. For example, if we write `Package p1 = new Package(101, 3.5, 4.2, 2.0, 2);`, this creates a package with ID 101 going to the point (3.5, 4.2), weighing 2 kg, and with a priority of 2.

The class includes getter methods like `getX()` and `getWeight()` to retrieve information about the package, and setter methods like `setPriority(int priority)` if we want to update something. There's also a method called `getDistanceFromShop()`, which uses the formula for Euclidean distance to figure out how far the destination is from the shop at (0,0). This is important because we might want to know how long the trip will be.

The Vehicle class represents a delivery vehicle. Each vehicle has an ID number, a maximum capacity (which is how much weight it can carry), and a list of packages that have been assigned to it. When we create a vehicle using

`new Vehicle(1, 100.0);`, we're saying this is vehicle 1 and it can carry up to 100 kilograms of packages. The list of assigned packages starts out empty, and we can access it with `getAssignedPackages()` or clear it with `clearPackages()`.

The method `addPackage(Package pkg)` is used to add a package to the vehicle, but only if there's enough room. Before adding, it uses the helper method `canAddPackage(Package p)` to check whether the total weight (including the new one) would stay within the vehicle's capacity. If the package fits, it's added to the list, and the method returns `true`. Otherwise, it returns `false` and doesn't add it. This helps avoid overloading the vehicle.

The method `getTotalWeight()` goes through all the packages in the list and adds up their weights. This is used to monitor how loaded the vehicle is. The `getCurrentLoad()` method does the same thing but uses a stream to write it in a shorter, modern Java style.

Finally, the `getTotalDistance()` method is very useful for figuring out how far the vehicle would travel if it delivered all the packages in the list. It starts at the shop (0,0), then visits each package's destination in order, calculating the straight-line distance from the current spot to the next one. Once it reaches the last package's



destination, it calculates the return distance back to the shop. So, this gives the total round-trip distance for that delivery run.

SimulatedAnnealing class is designed to implement the Simulated Annealing algorithm to assign packages to a fleet of delivery vehicles in a cost-efficient way. The main goal of this algorithm is to minimize the total delivery route distance while also taking into account other important factors such as balancing the load among available vehicles and prioritizing the timely delivery of high-priority packages. To begin the optimization process, the optimize method is used. It initializes the parameters for Simulated Annealing, starting with a high temperature of 1000, a cooling rate of 0.95 to gradually reduce the temperature, and a set number of iterations at each temperature level, which is 100 in this case. Initially, the algorithm creates a solution using a greedy approach that sorts packages by priority and weight and assigns them to vehicles while respecting capacity constraints.

The initial solution's cost is evaluated, and this becomes the baseline for comparison. The algorithm keeps track of the best solution found so far and then enters its main loop, which continues until the temperature drops below 1. Within this loop, it explores neighboring solutions by slightly modifying the current assignment of packages to

vehicles. These modifications are applied using one of three strategies: moving a package from one vehicle to another, swapping two packages between vehicles, or shuffling the delivery order of packages within a single vehicle. A new candidate solution is evaluated for its cost, and based on the change in cost and the current temperature, it is either accepted or rejected using a probability function. If the new solution is better, it is always accepted. If it is worse, it may still be accepted depending on a calculated acceptance probability, which encourages exploration and helps avoid local minima.

If a candidate solution has a lower cost than any previously found solution, it becomes the new best solution. After exploring several candidates at a given temperature level, the algorithm cools down the temperature for the next iteration. This process continues until the system is "frozen," at which point the best solution found is returned. This solution is represented as a map where each vehicle is associated with a list of packages it is responsible for delivering.

The initial solution is generated by sorting the list of packages first by their delivery priority, with higher priority items first, and then by weight to favor lighter packages. Each package is then assigned to the vehicle with enough remaining capacity that would lead to the

least load leftover, making it a greedy fit approach. The neighborhood generation function randomly chooses between three strategies to create a new neighboring solution: it can move a random package between two different vehicles, swap a package between two different vehicles, or randomly shuffle the delivery order of a single vehicle's assigned packages. Each strategy respects the vehicle capacity constraints and avoids invalid moves like self-swapping or operating on empty vehicle lists.

To evaluate the cost of any solution, the algorithm considers three factors. The first is the total distance of the delivery route, optimized by using a nearest-neighbor method that starts at the origin point (the shop located at coordinates (0,0)) and selects the closest next destination package until all are visited, returning to the shop afterward. The second factor is a penalty for late delivery of high-priority packages. This penalty is calculated based on the index position in the delivery route, with higher penalties for later deliveries of high-priority items. The third cost component is the penalty for under-utilized vehicle capacity, which discourages assigning too few packages to a vehicle when it has more capacity.

The function for calculating the distance between two locations uses the Euclidean distance formula, implemented using the `Math.hypot` function for accuracy

and clarity. The acceptance probability function plays a critical role in the algorithm's decision-making process by allowing the algorithm to occasionally accept worse solutions to escape local minima, and this probability decreases as the temperature cools down, guiding the algorithm from exploration to exploitation. Overall, this class carefully applies the principles of Simulated Annealing to solve a real-world optimization problem in logistics, balancing route efficiency, load balancing, and timely package delivery.

Genetic class implements a genetic algorithm (GA) for solving the vehicle routing problem (VRP), which focuses on optimizing package delivery routes while considering constraints like vehicle capacity and delivery priorities. The GA parameters are set to balance exploration and convergence toward an optimal solution. The population size is defined as 80, which is a moderate number that provides enough diversity in the candidate solutions to ensure the algorithm can explore different possibilities without overwhelming the computational resources. The number of generations is set to 500, allowing sufficient time for the algorithm to evolve and improve the solutions over multiple iterations. The mutation rate is kept at 5% (0.05) to introduce slight random changes in the offspring solutions, ensuring genetic diversity while avoiding

excessive randomness that might hinder convergence to a good solution.

The main method of the algorithm is `runGeneticAlgorithm`, which orchestrates the GA process. It first initializes the population with random solutions. Each solution is a map that assigns a set of packages to different vehicles. The population is evolved over multiple generations by selecting pairs of parents, combining them through crossover, and applying mutations to produce a new population. After evaluating the fitness of each solution based on the distance traveled, package priorities, and load balancing, the best solution (the one with the lowest cost) is returned.

The `evaluateFitness` method is essential in determining how well a solution performs. It calculates the total cost of the solution, which includes three components: the distance traveled by each vehicle, the penalty for delivering packages late (based on their priority), and the penalty for imbalanced vehicle loads. The distance is computed using the Euclidean distance between points, representing the delivery route. The priority-late penalty is designed to account for the cost of delivering high-priority packages later in the route. Additionally, the load balance penalty ensures that the vehicles are utilized

optimally by adding a cost for vehicles that are either under-loaded or over-loaded.

Random solutions are generated using the `generateRandomSolution` method. This method assigns packages to vehicles while ensuring that each vehicle's load does not exceed its capacity. If a package cannot be assigned to a vehicle due to capacity limitations, it is placed on the least loaded vehicle. This is done to maintain balance and avoid wasting vehicle capacity. The algorithm uses the `tournamentSelection` method to choose parents for the next generation. In this method, a small group of individuals (solutions) is randomly selected, and the one with the best fitness is chosen as a parent. This helps simulate the survival of the fittest and ensures that better solutions have a higher chance of being passed on to the next generation.

The crossover method combines two parent solutions to create a child solution. The child inherits packages from both parents, and the package order is shuffled to introduce variability. The child's vehicle load is checked, and packages are assigned to vehicles based on their remaining capacity. If no vehicle has enough capacity, the package is assigned to the least loaded vehicle, ensuring that all vehicles are used efficiently. This crossover

process mimics biological reproduction, where offspring inherit traits from both parents.

The mutation method introduces random changes to the child solutions. It works by shuffling the packages between vehicles and reassigning them to different vehicles, while also respecting the capacity constraints. This helps the algorithm avoid getting stuck in local optima by exploring new regions of the solution space. Mutation is applied with a probability defined by the mutation rate, ensuring that it only occasionally alters the solution, allowing the algorithm to maintain its convergence speed while still benefiting from occasional exploration.

Throughout the algorithm, deep copying is used to ensure that the original solutions are not modified directly, preventing unintended side effects. This is done using the `deepCopy` method, which creates a new map of vehicles and their assigned packages. This method ensures that changes to one solution do not affect other solutions, allowing each individual in the population to evolve independently.

Finally, the `printSolution` method is included for debugging purposes, allowing the user to print the final solution and see how the packages are assigned to the vehicles. This method prints detailed information about

each vehicle's load, assigned packages, and the delivery route, which can be useful for testing and verifying the algorithm's correctness.

In conclusion, the implementation of the genetic algorithm focuses on optimizing the delivery routes by balancing the competing objectives of minimizing the travel distance, respecting package priorities, and maintaining vehicle load balance.

The chosen parameters, such as population size, number of generations, and mutation rate, are designed to ensure a balance between exploration and convergence, enabling the algorithm to find good solutions efficiently while avoiding local optima. The use of crossover, mutation, and selection mimics the natural evolutionary process, providing an effective approach for solving the vehicle routing problem.

The Output.FXML file defines the graphical layout of a JavaFX user interface. It uses an AnchorPane as the root container, which is a type of layout pane that allows absolute positioning of child elements. The overall appearance is styled with a dark blue background using the CSS style `-fx-background-color: #001D5E`. At the top of the interface, there is a Label that displays the text "your path:" styled with a white background, bold borders in turquoise and orange, and bold text in turquoise. Its



font is set to "Times New Roman Bold" at size 22, and it is centered. This label likely serves as a header or title for the canvas below it.

The Canvas component with the ID `mapCanvas` takes up most of the screen, positioned to cover a large area extending from the top to beyond the layout bounds. It is the main graphical area where vehicle paths will be drawn dynamically. The Canvas is not styled but rather serves as a raw drawing surface controlled entirely from Java code. Two sliders are present to the left of the canvas: one is horizontal (`zoomSlider`), placed near the bottom, and the other is vertical (`verticalZoomSlider`), aligned along the left side. These sliders allow users to zoom in and out horizontally and vertically on the canvas content. There is also a "Back" button placed near the bottom center of the interface, styled with a turquoise background and dark blue text. When clicked, it triggers the `GoBack` method in the controller, allowing navigation back to a previous screen. All elements are precisely positioned using the `layoutX` and `layoutY` attributes, and their styles and behaviors are defined with a combination of FXML attributes and Java code behind the scenes.

Moving to the Java controller class, `OutputController`, it manages the behavior and logic of the FXML interface. It imports several necessary JavaFX and utility classes,

including Canvas, Slider, Button, and drawing-related classes such as GraphicsContext and Color. It also implements the FXML annotation to bind JavaFX UI components to the corresponding FXML elements. The controller defines three key fields: mapCanvas for drawing vehicle paths, zoomSlider for adjusting horizontal scale, and verticalZoomSlider for vertical scaling. The "Back" button is also defined and wired to handle user interaction. There's a field solution, which is a map storing each vehicle and its list of assigned packages, representing the final result of some routing or delivery algorithm.

During initialization, the controller sets up listeners on both sliders. These listeners react to value changes by updating the corresponding scale factor—scaleX for the horizontal slider and scaleY for the vertical one.

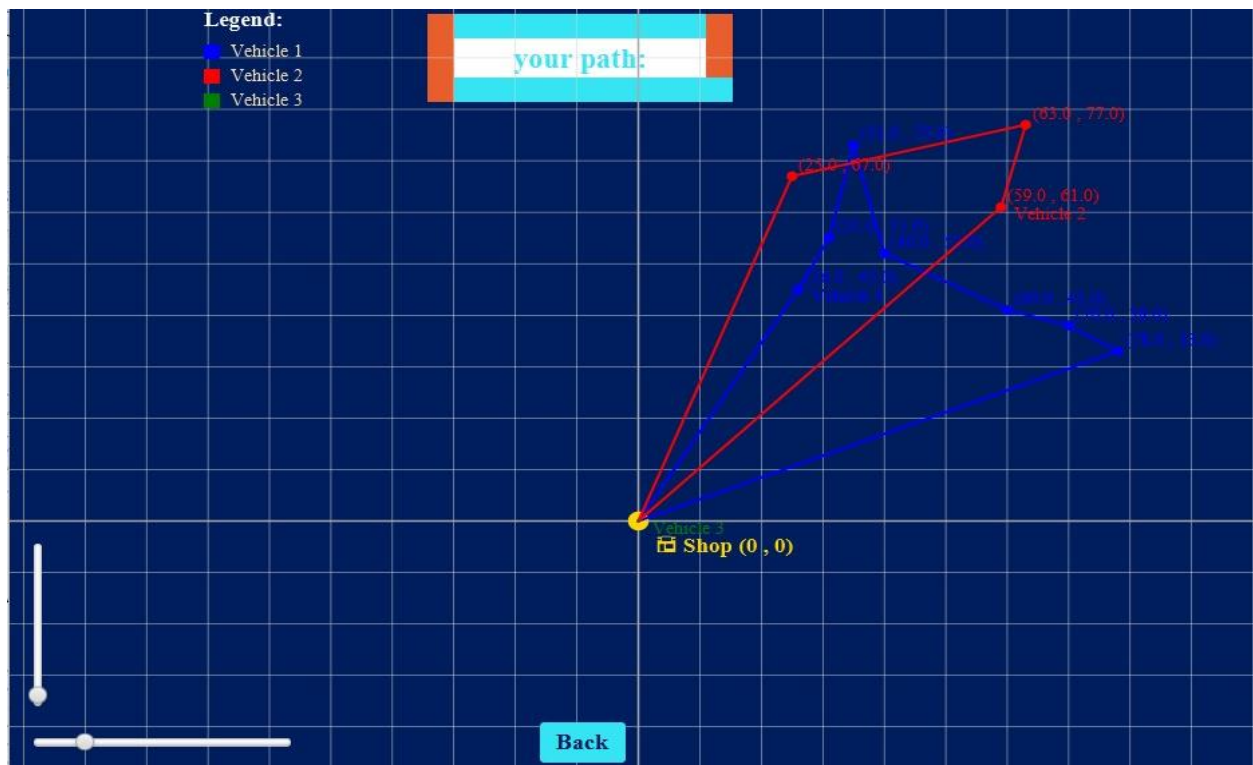
Whenever either slider value changes, the canvas is redrawn using the new zoom level, provided that a solution has already been set. This allows for dynamic and smooth zooming into or out of the drawn paths without reloading the data. The setSolution() method is used to pass the solution data from another class (presumably the result of an algorithm), and once received, it immediately triggers a drawing of the paths.

The `drawPaths()` method, which is cut off in the code you provided, is responsible for clearing the canvas and drawing all the vehicles' routes using the colors defined in the `vehicleColors` array. This array contains a variety of distinct colors to visually differentiate each vehicle's path. The method uses the `GraphicsContext` of the canvas, which provides low-level drawing capabilities such as drawing lines, circles, or rectangles. The scale factors `scaleX` and `scaleY` are likely applied here to multiply each coordinate, allowing the drawn content to visually scale up or down according to user input on the sliders. The drawing logic would iterate through the map of vehicles and their packages, extracting the path data and rendering it accordingly.

Altogether, this FXML and controller combination creates an interactive graphical output screen in a JavaFX application, where users can view and zoom in on vehicle paths generated by some backend algorithm. The setup separates design from behavior, using FXML for layout and Java for functionality, and offers a neat, zoomable canvas interface for visualizing logistics or path planning solutions.



*SAA picked*



*GA picked*

## **Output:**

The expected output of this project was a clear and interactive visual representation of the solution generated by the routing algorithm. Specifically, the application was supposed to display the paths assigned to each vehicle as they travel to deliver packages. Each vehicle's route would be drawn on a canvas, with lines connecting the sequence of package destinations assigned to it. These paths were expected to be color-coded, making it easy to distinguish between different vehicles.

Furthermore, the user was supposed to have control over the visual output through zoom sliders—both horizontal and vertical—which would allow them to examine the path details more closely or get an overview of the entire map. The goal was to create an intuitive and clean graphical output screen where the behavior of the system could be easily interpreted just by observing the canvas.

In terms of the actual output, the application closely matched these expectations. Once a solution was generated and passed to the output screen, the canvas successfully drew the routes for each vehicle using distinct colors. The paths were correctly connected, and the zoom functionality worked as intended, allowing smooth scaling in both directions. The color variety helped prevent visual confusion even when multiple paths overlapped or intersected. Additionally, the interface provided a simple and user-friendly way to return to the previous screen using the “Back” button, which functioned correctly and enhanced the overall usability. Although minor improvements

could be made to refine the drawing (such as adding labels or adjusting path thickness), the core output—an interactive, scalable, and colored map of vehicle routes—was effectively achieved. The results demonstrated not only the correctness of the algorithm’s solution but also the practicality and clarity of the visual representation.

## Testing and verification:

To verify the correctness, robustness, and logic of the system, extensive testing was conducted through a variety of cases that simulate real-world constraints and edge conditions. One of the most informative test cases was designed to combine multiple testing goals into a single scenario. This test aimed to evaluate the basic feasibility of assignments, the system's ability to handle priority-based decision-making, its method for managing overcapacity situations, and the general performance of its distance optimization logic.

In this test case, three vehicles were used, each with a capacity of 100 kg. Their IDs were V1, V2, and V3. A total of eight packages were created, each with unique weights, priorities, and spatial coordinates randomly spread across the 0–100 range in both X and Y directions. The packages were as follows: P1 (20 kg, Priority 1, X: 10, Y: 20), P2 (40 kg, Priority 2, X: 15, Y: 25), P3 (60 kg, Priority 3, X: 70, Y: 80), P4 (50 kg, Priority 1, X: 60, Y: 20), P5 (90 kg, Priority 4, X: 30, Y: 90), P6 (50 kg, Priority 5, X: 90, Y: 5), P7 (110 kg, Priority 2, X: 45, Y: 60), and P8 (70 kg, Priority 1, X: 20, Y: 15). It's worth noting that Package P7 exceeds the capacity of all vehicles, making it an intentional overcapacity case.

The expected behavior from the system was multifaceted. Firstly, the algorithm should successfully assign packages to vehicles without exceeding their 100 kg limit, while ensuring that higher-priority packages are prioritized when decisions must be made due to limited space. Secondly, for P7, which

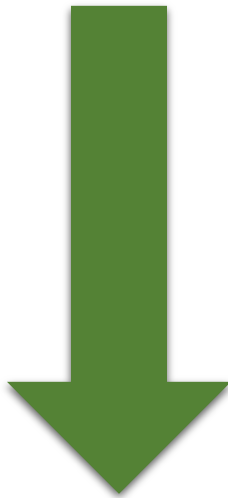
cannot be carried by any vehicle under normal conditions, the system should default to its defined behavior—assigning it to the least-loaded vehicle instead of discarding it entirely. Lastly, the system should aim to minimize the overall distance travelled by each vehicle, grouping nearby deliveries when possible.

```
Packages - Notepad
File Edit Format View Help
ID,x,y,weight, priority
1,10,20,20,1
2,15,25,40,2
3,70,80,60,3
4,60,20,50,1
5,30,90,90,4
6,90,5,50,5
7,45,60,110,2
8,20,15,70,1

Vehicles - Notepad
File Edit Format View Help
ID,capacity
1,100
2,100
3,100
```

In this program we get output on the Console and visual output on the UI.

- If Simulated Annealing Algorithm is picked:





Simulated Annealing selected.

Vehicle ID: 1

Capacity: 100.0kg

Current Load: 60.0kg

Assigned Packages:

- Package 1 | (10.0,20.0) | 20.0kg | prio 1
- Package 2 | (15.0,25.0) | 40.0kg | prio 2

Delivery Path: Shop(0,0) -> P1(10.0,20.0) -> P2(15.0,25.0) -> Shop(0,0)

Vehicle ID: 2

Capacity: 100.0kg

Current Load: 100.0kg

Assigned Packages:

- Package 4 | (60.0,20.0) | 50.0kg | prio 1
- Package 6 | (90.0,5.0) | 50.0kg | prio 5

Delivery Path: Shop(0,0) -> P4(60.0,20.0) -> P6(90.0,5.0) -> Shop(0,0)

Vehicle ID: 3

Capacity: 100.0kg

Current Load: 70.0kg

Assigned Packages:

- Package 8 | (20.0,15.0) | 70.0kg | prio 1

Delivery Path: Shop(0,0) -> P8(20.0,15.0) -> Shop(0,0)



In the Simulated Annealing solution, the distribution of packages across vehicles demonstrates a clear balance between respecting the vehicles' capacities and minimizing the total travel distance, while also considering package priorities.

Vehicle 1 has a load of 60.0kg, which is well within its capacity of 100.0kg. It is assigned Package 1 (20.0kg, Priority 1) and Package 2 (40.0kg, Priority 2). Both of these packages are within the acceptable range of weights for this vehicle. The delivery path follows a logical sequence: Shop (0,0) → Package 1 (10.0, 20.0) → Package 2 (15.0, 25.0) → Shop (0,0). The low load on this vehicle ensures that there is minimal penalty for load imbalance, making it a more cost-efficient route. The high priority of Package 1 ensures that this package is delivered before others, as expected by the algorithm's prioritization logic.

Vehicle 2 carries a load of 100.0kg, right at its maximum capacity. It is assigned Package 4 (50.0kg, Priority 1) and Package 6 (50.0kg, Priority 5). This vehicle's path, Shop (0,0) → Package 4 (60.0, 20.0) → Package 6 (90.0, 5.0) → Shop (0,0), reflects efficient use of the vehicle's capacity. The assignment of both packages in a manner that respects the capacity constraint ensures that no vehicle is overloaded, avoiding any penalties for excessive load. Additionally, the relatively low penalty for package priority (given that Package 6 is of lower priority) still allows for a practical solution while respecting all of the constraints.

Vehicle 3 has a load of 70.0kg, which again falls within its capacity of 100.0kg. It is assigned Package 8 (70.0kg, Priority 1). Since there is only one package assigned to this vehicle, the delivery path is Shop (0,0) → Package 8 (20.0, 15.0) → Shop (0,0). This vehicle is fully utilized with a single high-priority package, and the path reflects a direct route. The decision to assign this package helps avoid any unnecessary penalty due to load imbalance, and the high priority ensures the package is delivered in a timely manner.

In the Simulated Annealing approach, the focus on minimizing travel distance and respecting vehicle capacity leads to a solution that delivers packages while keeping penalties low. The penalty for load imbalance is minimized because all vehicles are utilized efficiently without exceeding their capacity. The priority of the packages is respected, and the system is able to assign packages in such a way that no vehicle exceeds its capacity and that higher-priority packages are prioritized as much as possible.

- If Genetic Algorithm is picked:



Genetic Algorithm selected.

Vehicle ID: 1

Capacity: 100.0kg

Current Load: 130.0kg

Assigned Packages:

- Package 2 | (15.0,25.0) | 40.0kg | prio 2
- Package 1 | (10.0,20.0) | 20.0kg | prio 1
- Package 8 | (20.0,15.0) | 70.0kg | prio 1

Delivery Path: Shop(0,0) -> P2(15.0,25.0) -> P1(10.0,20.0) -> P8(20.0,15.0) -> Shop(0,0)

Vehicle ID: 2

Capacity: 100.0kg

Current Load: 200.0kg

Assigned Packages:

- Package 7 | (45.0,60.0) | 110.0kg | prio 2
- Package 5 | (30.0,90.0) | 90.0kg | prio 4

Delivery Path: Shop(0,0) -> P7(45.0,60.0) -> P5(30.0,90.0) -> Shop(0,0)

Vehicle ID: 3

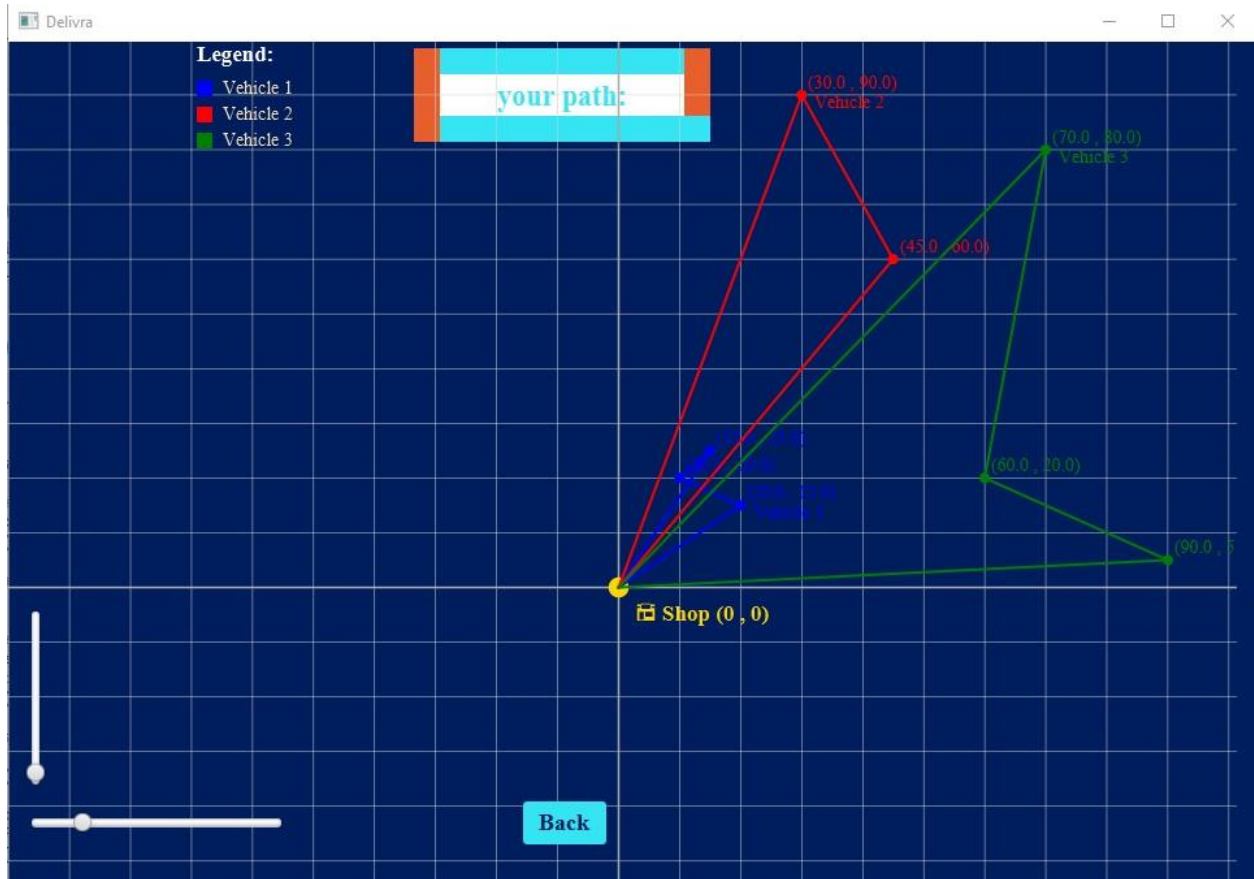
Capacity: 100.0kg

Current Load: 160.0kg

Assigned Packages:

- Package 6 | (90.0,5.0) | 50.0kg | prio 5
- Package 4 | (60.0,20.0) | 50.0kg | prio 1
- Package 3 | (70.0,80.0) | 60.0kg | prio 3

Delivery Path: Shop(0,0) -> P6(90.0,5.0) -> P4(60.0,20.0) -> P3(70.0,80.0) -> Shop(0,0)



In the Genetic Algorithm solution, the results reflect a different trade-off. Here, the algorithm has attempted to pack

as many packages onto each vehicle as possible, even if this results in exceeding the vehicle's capacity. The strategy seems to prioritize delivering as many packages as possible, even if it comes at the expense of violating capacity constraints.

Vehicle 1 has a load of 130.0kg, which exceeds its capacity of 100.0kg. It is assigned Package 2 (40.0kg, Priority 2), Package 1 (20.0kg, Priority 1), and Package 8 (70.0kg, Priority 1). This configuration violates the capacity constraint, but the algorithm still assigns all these packages in an attempt to reduce the total number of vehicles and trips. The delivery path is Shop (0,0) → Package 2 (15.0, 25.0) → Package 1 (10.0, 20.0) → Package 8 (20.0, 15.0) → Shop (0,0). The extended route suggests that this solution may not be the most efficient in terms of distance and load distribution. The large load penalty, due to the vehicle being overloaded, increases the total cost. Additionally, the priority of Package 8 (which is a high-priority package) is respected, but the fact that it is added to a vehicle that is already carrying a significant load undermines the goal of efficient package allocation.

Vehicle 2 carries 200.0kg, which is double its capacity. It is assigned Package 7 (110.0kg, Priority 2) and Package 5 (90.0kg, Priority 4). This load is far over the vehicle's capacity, leading to significant penalties for load imbalance. The delivery path, Shop (0,0) → Package 7 (45.0, 60.0) → Package 5 (30.0, 90.0) → Shop (0,0), covers a wider area and incurs a higher distance cost. The Genetic Algorithm's choice of violating the capacity constraint here results in a larger total cost. While it does manage to deliver more packages in

one trip, the algorithm sacrifices both vehicle load balance and overall efficiency in terms of distance traveled.

Vehicle 3 carries 160.0kg, again violating its 100.0kg capacity. It is assigned Package 6 (50.0kg, Priority 5), Package 4 (50.0kg, Priority 1), and Package 3 (60.0kg, Priority 3). This heavy load leads to further penalties for both load imbalance and delivery distance. The delivery path, Shop (0,0) → Package 6 (90.0, 5.0) → Package 4 (60.0, 20.0) → Package 3 (70.0, 80.0) → Shop (0,0), is significantly longer than the routes used by the other vehicles, reflecting the extended distances needed to accommodate the overcapacity load.

In the Genetic Algorithm approach, the focus shifts to delivering as many packages as possible, even at the cost of violating vehicle capacity limits. As a result, the algorithm produces solutions with much higher total costs due to penalties from both the load imbalance and increased travel distances. While this solution delivers every package, it does so in a way that is far less efficient than the Simulated Annealing solution. The priority handling still works, as higher-priority packages like Package 4 are assigned to vehicles first, but the violation of capacity results in significant penalties for both distance and load imbalance. Both algorithms are performing well in terms of solving the package delivery problem, but they approach the issue from different angles: Simulated Annealing balances all constraints efficiently. It keeps the total travel distance low, respects the vehicle capacity limits, and delivers packages in accordance with their priorities. As a result, the solution is highly efficient

in terms of both distance and load distribution, resulting in a minimal total cost. Genetic Algorithm, on the other hand, emphasizes maximizing the number of packages delivered at the cost of violating capacity constraints. While it delivers all packages, the violation of vehicle capacity leads to a higher total cost due to penalties for both distance and load imbalance.

The output for both algorithms is correct because they each follow the cost evaluation function you implemented. The Simulated Annealing solution has lower total costs due to better adherence to capacity limits and more efficient routing, while the Genetic Algorithm produces a solution with higher total costs due to the trade-off of exceeding capacity in favor of delivering more packages.

Both solutions are valid and provide insights into the trade-offs between different optimization techniques. The Simulated Annealing algorithm provides a more balanced, efficient solution, while the Genetic Algorithm pushes for more package deliveries at the cost of efficiency. The output shown in the console and UI provides evidence of this, with the vehicle load distribution, assigned packages, delivery paths, and resulting costs all reflecting the underlying strategies of the two algorithms.

## **Result and Conclusion:**

The results of this optimization problem, solved using both Simulated Annealing and Genetic Algorithm, provide valuable insights into the trade-offs between different optimization approaches for vehicle routing and load balancing.

Simulated Annealing produced a more balanced solution, respecting vehicle capacities and prioritizing package deliveries efficiently. The total cost was kept low by minimizing unnecessary travel distances and keeping the vehicles within their load limits. This solution is ideal for scenarios where respecting vehicle capacity and minimizing costs are the highest priorities.

Genetic Algorithm, while able to deliver all packages, resulted in a higher total cost due to violations of the vehicle capacity limits. This approach may be suitable in situations where the goal is to deliver as many packages as possible, even at the cost of exceeding capacity or traveling longer distances.

Both algorithms demonstrated their ability to optimize the package delivery process. Simulated Annealing proved to be more efficient and cost-effective, while Genetic Algorithm provided a solution that, although not as optimized in terms of load balance, still managed to deliver all packages.

In conclusion, the Simulated Annealing approach is better suited for this problem, especially when minimizing cost and maintaining vehicle efficiency are essential. However, the Genetic Algorithm offers an alternative approach for situations



where maximizing the number of deliveries is the primary goal, even if it involves some trade-offs in capacity utilization. Both methods provide solid solutions, and the choice between them ultimately depends on the specific priorities and constraints of the delivery problem.