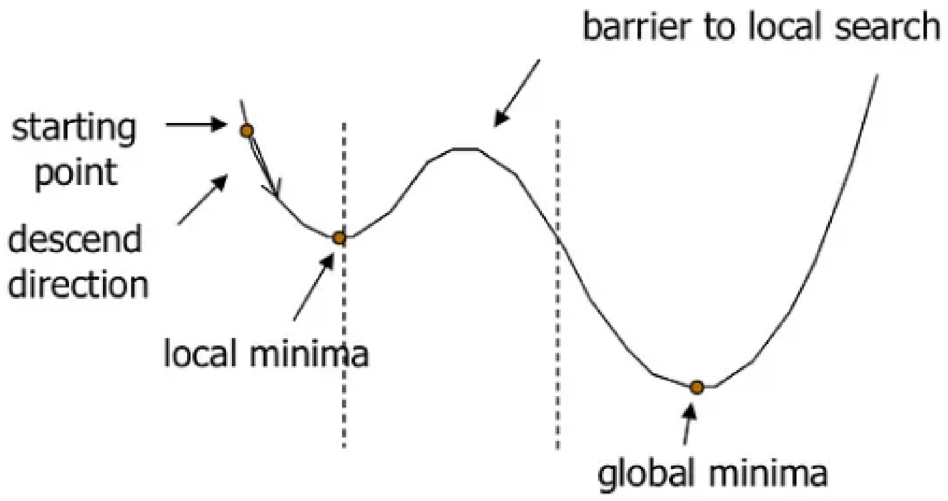
Simulated Annealing

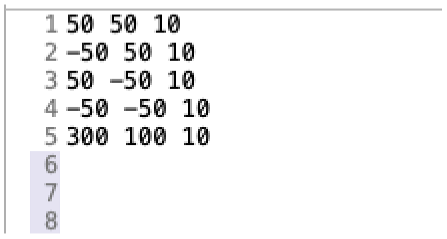
Simulated Annealing is a local search algorithm that is not guaranteed to give the optimal solution, however it does find a great approximate solution that is good enough for many search problems. This Algorithm has mainly two parameters, which are the temperature and cooling factor inspired by the process of metallurgy. The algorithm starts with an initially random solution, then it iteratively generates other solutions (neighboring solutions) by making local modifications. It will accept the neighboring solution if it has a better cost or if it meets a certain probability function, that is if the solution is worse. Initially, this algorithm is more likely to accept worse solutions but with time this decreases, becoming better at finding the global minima. The value of the temperature controls the probability of accepting the worse solution. This algorithm is beneficial for solving complex problems since it can explore the solution space probabilistically.



As Can be seen in the figure above, if the gradient descent begins at the starting point, it will end up in the local minima, and won't be able to reach the global minima. Thus, it is very important that this algorithm introduces randomness when initializing the starting state.

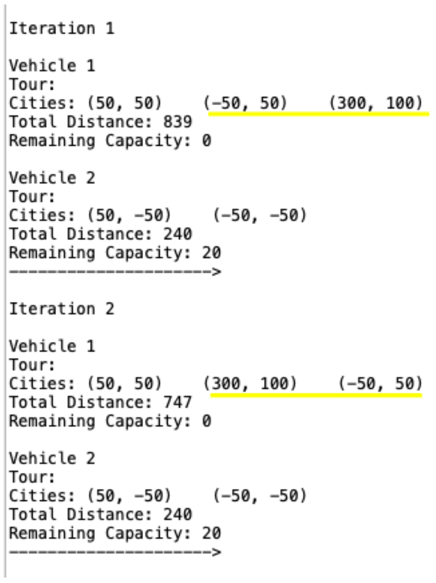
Problem Formulation and Code Explanation

In our code, our initial state is a randomly generated initial solution or route for each other's vehicles, taking into account the capacity of each vehicle and the package's weights. In order to generate the neighboring states we must apply the successor function or operators. In our code, this can be done by either swapping two destination points from the same route, or moving a package from one vehicle to another. Our state space is set inside a text file, where each line represents the x-y coordinate points (representing the locations to be delivered to) and the weights of the packages. The following figure shows the state space.



```
50 50 10
-50 50 10
50 -50 10
-50 -50 10
300 100 10
```

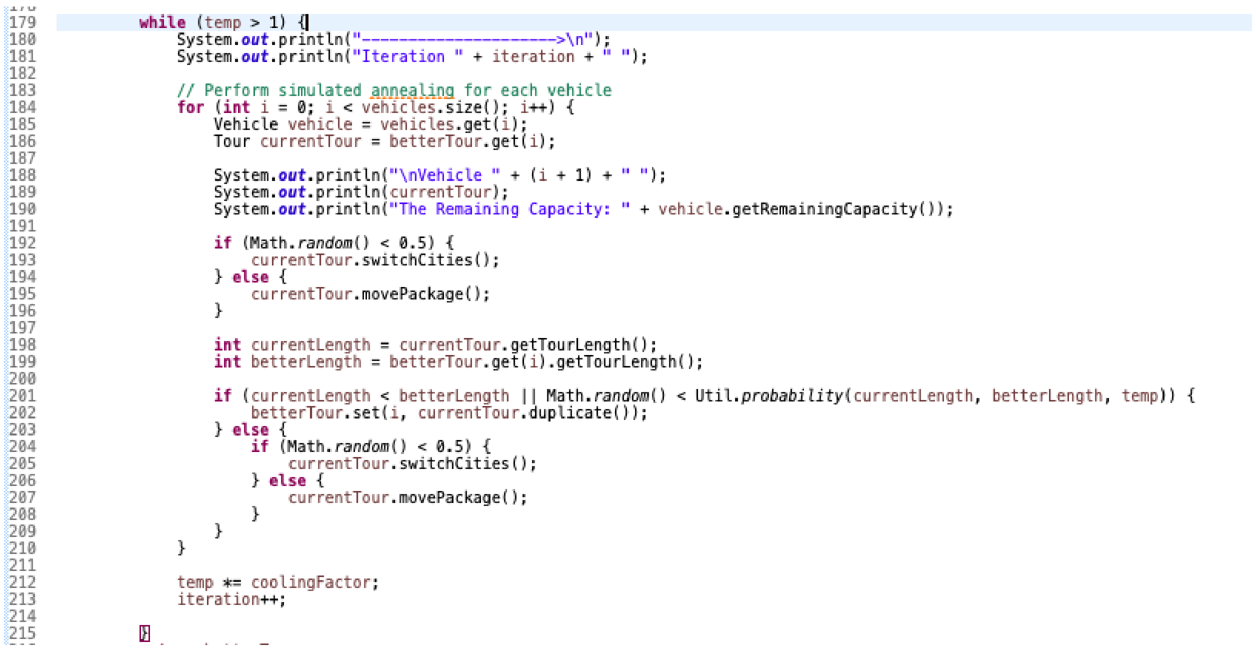
The goal state in our algorithm is a path that has a low path cost with a smooth route, it is not necessarily the optimal solution (the path that has the lowest path cost). However from the following output we can see the local changes that happen between the iterations, resulting in different cost paths. As can be seen below, the first Vehicle's path cost is 839, in the second iteration a local change happened swapping two cities resulting in a new path cost 747.



```
Iteration 1

Vehicle 1
Tour:
Cities: (50, 50)    (-50, 50)    (300, 100)
Total Distance: 839
Remaining Capacity: 0

Vehicle 2
Tour:
Cities: (50, -50)    (-50, -50)
Total Distance: 240
Remaining Capacity: 20
---------------------->

Iteration 2

Vehicle 1
Tour:
Cities: (50, 50)    (300, 100)    (-50, 50)
Total Distance: 747
Remaining Capacity: 0

Vehicle 2
Tour:
Cities: (50, -50)    (-50, -50)
Total Distance: 240
Remaining Capacity: 20
---------------------->
```

The path cost is the total distance measured from the starting point (0,0) to the destination points and back to the starting point. This is calculated using the euclidean distance formula as seen below. We use the path cost as an evaluation or objective function to compare between two different solutions and find the path that minimizes this cost.

Inside the simulated annealing function, both parameters are initialized. The loop inside the function iterates, each time doing a local change. The choice between the two different local changes is decided randomly. Then, the new tour path cost is calculated and compared with the current tour path cost. If the new tour is shorter than the current tour, then the tour is updated. However, if the new tour is not better than the current tour, then there is still a possibility that it will be accepted based on a probability function shown below. In each iteration the temperature is being reduced by multiplying it with the cooling factor. This loop continues as long as the temperature is greater than one.



```
        while (temp > 1) {
            System.out.println("--------------------->\n");
            System.out.println("Iteration " + iteration + " ");

            // Perform simulated annealing for each vehicle
            for (int i = 0; i < vehicles.size(); i++) {
                Vehicle vehicle = vehicles.get(i);
                Tour currentTour = betterTour.get(i);

                System.out.println("\nVehicle " + (i + 1) + " ");
                System.out.println(currentTour);
                System.out.println("The Remaining Capacity: " + vehicle.getRemainingCapacity());

                if (Math.random() < 0.5) {
                    currentTour.switchCities();
                } else {
                    currentTour.movePackage();
                }

                int currentLength = currentTour.getTourLength();
                int betterLength = betterTour.get(i).getTourLength();

                if (currentLength < betterLength || Math.random() < Util.probability(currentLength, betterLength, temp)) {
                    betterTour.set(i, currentTour.duplicate());
                } else {
                    if (Math.random() < 0.5) {
                        currentTour.switchCities();
                    } else {
                        currentTour.movePackage();
                    }
                }
            }

            temp *= coolingFactor;
            iteration++;

        }
```

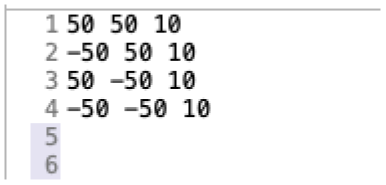
As can be seen from our simulated annealing function, there are random factors that we have included in order to avoid the Local minima problem. For Example, choosing between the two local changes was decided randomly. Moreover, the algorithm chooses a worse tour sometimes based on a probability function in order to increase the randomness and decrease the chances to fall into a Local minima. The formula below is used as our probability function for accepting a worse solution.

```
public static double probability(double f1, double f2, double temp) { // f1 and f2 cities
    if (f2 < f1) return 1;
    return Math.exp((f1 - f2) / temp); // prob e^e/T
}
```

Genetic Algorithm

Genetic algorithm is a local optimization technique that maintains a population of solutions, and develops these solutions by iteratively applying a set of stochastic operators. This algorithm is inspired by the process of natural selection and the idea of survival of the fittest, which means that better solutions will be produced from previous generations until a near optimal solution is reached. This algorithm first works by randomly generating an initial population. Using an evaluation function, the fitness of the population is evaluated. Based on this evaluation function, the parents are selected to reproduce the next generation by doing a crossover. Another factor that increases the randomness in this algorithm is the use of a mutation, which randomly perturbs a solution by making a local modification.

In our code, the population represents a set of solutions. These solutions are paths that start from the initial point (0,0). A chromosome represents one of these solutions, implemented using an array of cities. A gene is represented as a package in our code. We have started with 5 populations generated randomly using the initializePopulation()function. The state space is inside a file named pack.txt, where the destination and the package weights are stored as seen in the figure below.

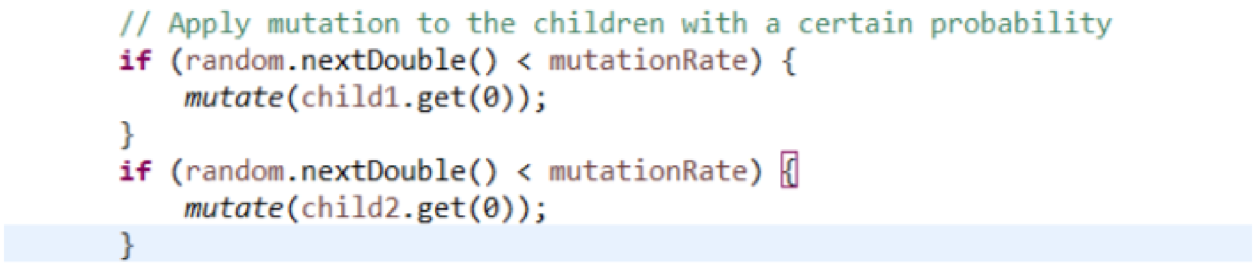


```
50 50 10
-50 50 10
50 -50 10
-50 -50 10
```

We have assumed that our shop has two vehicles, meaning each population has two routes. The packages are randomly distributed among the two vehicles such that both have different routes. As mentioned previously the fitness function (objective function) is the total distance traveled to all the destinations by the two vehicles. The higher the Fitness value is, the better chance for reproduction of population. This evaluation function is measured by the euclidean distance and it is used to compare between the solutions; thus, the best two solutions are chosen as the two parents. The successor operation to move from one state to another is the crossover and mutation operations.

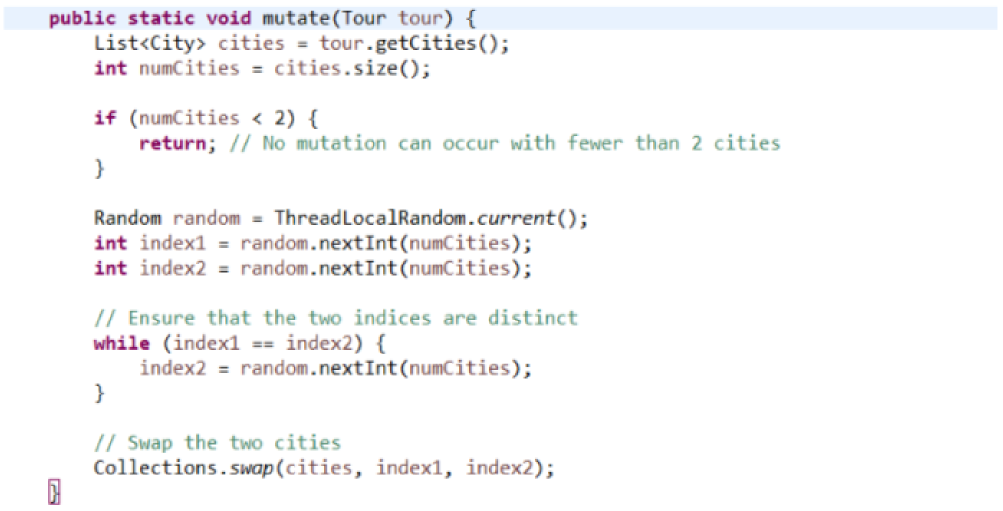
The crossover between the two parents Exchanges genetic information to create one or more offspring, in our code we generate two offspring. The first child is a crossover between vehicle 1 from parent 1 with vehicle 1 in parent 2. The second child is a crossover between vehicle 2 from parent 1 with vehicle 1 in parent 2. Finally, we add these 2 offsprings to the population.

Mutation is another factor that increases the randomness in order to avoid local minima; this is done by doing Local modification, swapping between packages in the same route.The variable named mutationRate is set to a value of 0.80. This means that only 80% of children will undergo mutation. We generate a random number, if the number is greater than 0.80, then mutation will occur to generate a new gene.



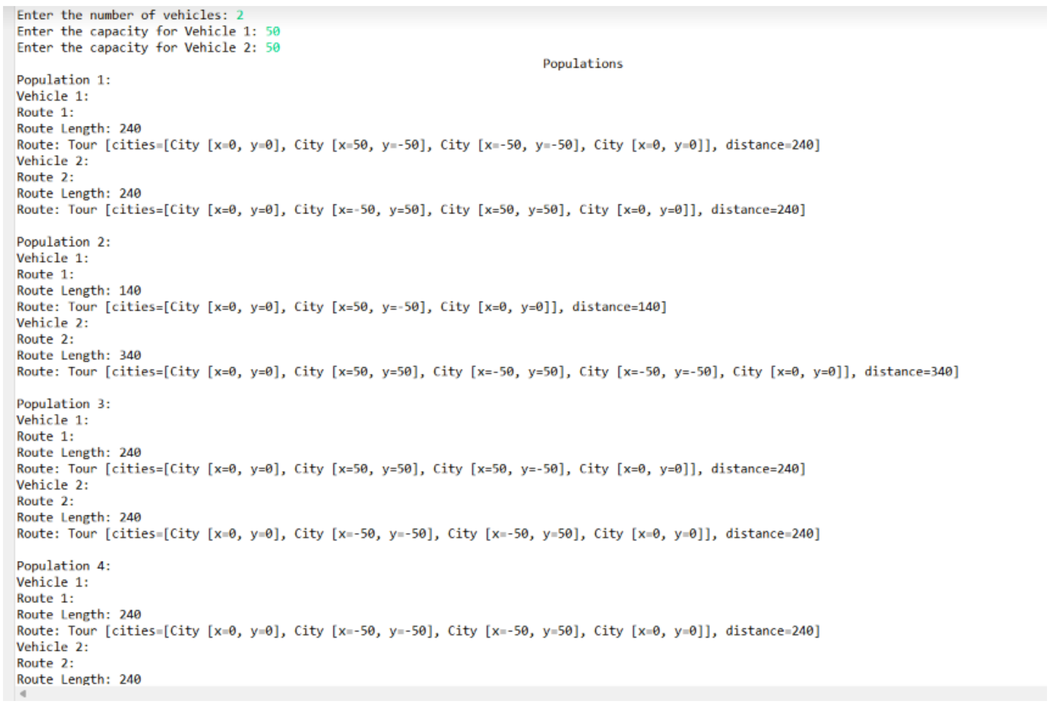
```
// Apply mutation to the children with a certain probability
if (random.nextDouble() < mutationRate) {
    mutate(child1.get(0));
}
if (random.nextDouble() < mutationRate) {
    mutate(child2.get(0));
}
```

The figure below shows the mutation function, where the swapping of the packages occurs in the same tour.

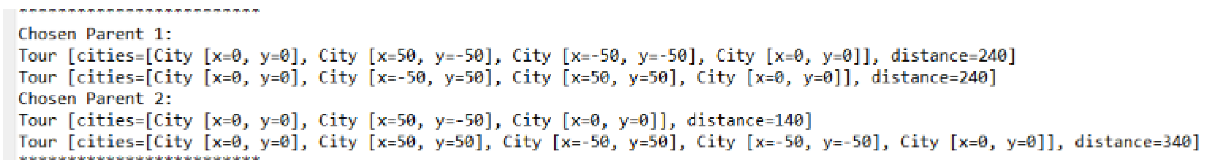


```java
public static void mutate(Tour tour) {
    List<City> cities = tour.getCities();
    int numCities = cities.size();

    if (numCities < 2) {
        return; // No mutation can occur with fewer than 2 cities
    }

    Random random = ThreadLocalRandom.current();
    int index1 = random.nextInt(numCities);
    int index2 = random.nextInt(numCities);

    // Ensure that the two indices are distinct
    while (index1 == index2) {
        index2 = random.nextInt(numCities);
    }

    // Swap the two cities
    Collections.swap(cities, index1, index2);
}
```

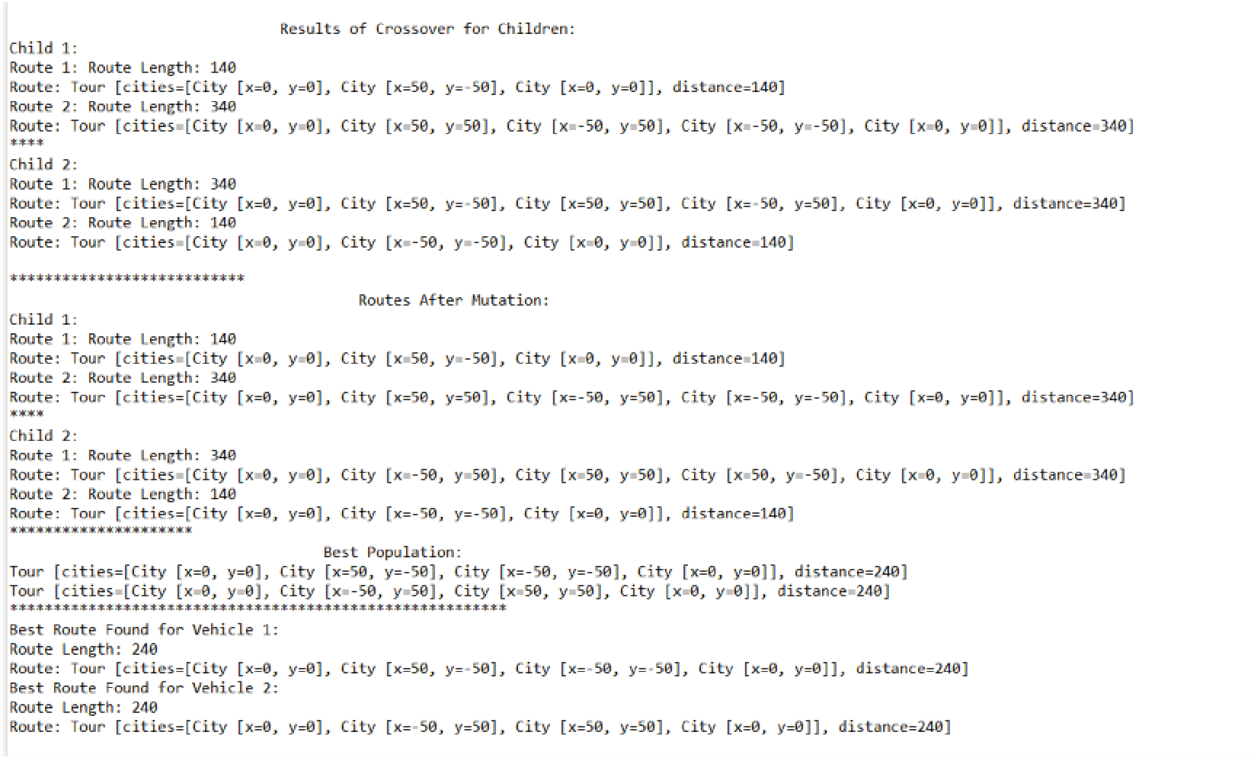
The figure below is the output seen when crossover and mutation occurs. As seen below 5 populations were generated, the best two were picked as parents. As seen below the crossover generated two offspring.



```
Enter the number of vehicles: 2
Enter the capacity for Vehicle 1: 50
Enter the capacity for Vehicle 2: 50
                                                  Populations
Population 1:
Vehicle 1:
Route 1:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=-50, y=-50], City [x=0, y=0]], distance=240]
Vehicle 2:
Route 2:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=50], City [x=50, y=50], City [x=0, y=0]], distance=240]

Population 2:
Vehicle 1:
Route 1:
Route Length: 140
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=0, y=0]], distance=140]
Vehicle 2:
Route 2:
Route Length: 340
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=50], City [x=-50, y=50], City [x=-50, y=-50], City [x=0, y=0]], distance=340]

Population 3:
Vehicle 1:
Route 1:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=50], City [x=50, y=-50], City [x=0, y=0]], distance=240]
Vehicle 2:
Route 2:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=-50], City [x=-50, y=50], City [x=0, y=0]], distance=240]

Population 4:
Vehicle 1:
Route 1:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=-50], City [x=-50, y=50], City [x=0, y=0]], distance=240]
Vehicle 2:
Route 2:
Route Length: 240
```



```
**************************
Chosen Parent 1:
Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=-50, y=-50], City [x=0, y=0]], distance=240]
Tour [cities=[City [x=0, y=0], City [x=-50, y=50], City [x=50, y=50], City [x=0, y=0]], distance=240]
Chosen Parent 2:
Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=0, y=0]], distance=140]
Tour [cities=[City [x=0, y=0], City [x=50, y=50], City [x=-50, y=50], City [x=-50, y=-50], City [x=0, y=0]], distance=340]
**************************
```



```
                              Results of Crossover for Children:
Child 1:
Route 1: Route Length: 140
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=0, y=0]], distance=140]
Route 2: Route Length: 340
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=50], City [x=-50, y=50], City [x=-50, y=-50], City [x=0, y=0]], distance=340]
****
Child 2:
Route 1: Route Length: 340
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=50, y=50], City [x=-50, y=50], City [x=0, y=0]], distance=340]
Route 2: Route Length: 140
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=-50], City [x=0, y=0]], distance=140]

***************************
                                       Routes After Mutation:
Child 1:
Route 1: Route Length: 140
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=0, y=0]], distance=140]
Route 2: Route Length: 340
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=50], City [x=-50, y=50], City [x=-50, y=-50], City [x=0, y=0]], distance=340]
****
Child 2:
Route 1: Route Length: 340
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=50], City [x=50, y=50], City [x=50, y=-50], City [x=0, y=0]], distance=340]
Route 2: Route Length: 140
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=-50], City [x=0, y=0]], distance=140]
*********************
                                    Best Population:
Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=-50, y=-50], City [x=0, y=0]], distance=240]
Tour [cities=[City [x=0, y=0], City [x=-50, y=50], City [x=50, y=50], City [x=0, y=0]], distance=240]
*********************************************************
Best Route Found for Vehicle 1:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=50, y=-50], City [x=-50, y=-50], City [x=0, y=0]], distance=240]
Best Route Found for Vehicle 2:
Route Length: 240
Route: Tour [cities=[City [x=0, y=0], City [x=-50, y=50], City [x=50, y=50], City [x=0, y=0]], distance=240]
```

Conclusion

By doing this project, we have learned so much about many algorithms and how they are used to solve real world problems. By implementing them and seeing the results of each algorithm, we were able to better understand how these algorithms work and why they don't guarantee to give the optimal solution. Having that said, we still explore the benefits and advantages of using local optimization.

References

1) https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7#:~:text=Simulated%20Annealing%20(SA)%20mimics%20the,Descent%20would%20be%20stuck%20at.

2) https://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/?fbclid=IwAR3-OyFF1YcwXX_B8iOz1u2Gsm59zt3-e4oT9-d4JdjdjnDJXdbAapNtbbc

3) https://stackabuse.com/introduction-to-genetic-algorithms-in-java/?fbclid=IwAR2xZhDE4cUdaMEvGRO0lNtuSu1t7vLfVQLlfg5jf1UhmaKLNkhjOarormE

4) https://stackabuse.com/traveling-salesman-problem-with-genetic-algorithms-in-java/?fbclid=IwAR3z7kuZU0RiGKiEXfdKx7o2lFDPVBHuunbo63ZyjUx4GVpMwgJdRgEmdPM