Transilvania
University
of Brasov
FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE

Transilvania University of Brasov

- University of Mathematics and Computer Science -

Solving the Vehicle Routing Problem (VRP) and Package Selection Problem (PSP) using Genetic Algorithms

Lopataru Mihnea

Dobre Cosmin

Edves Alexandru

2021-2022

Content

Introduction

Ever since the dawn of modern civilization, couriers have had an important role in the well-functioning of a society. Starting with a simple man which had to walk to each destination in order to ensure that every individual will get their package, to huge companies equipped with a fleet of cars and drivers being able to transport thousands of parcels each day.

The importance of couriers has been especially outlined in this pandemic where, due to the Covid-19 virus, many people were forced to quarantine and stay in their homes for days on end. As a result, we saw an increase of online orders, ranging from food related items to house appliances, delivery companies being faced with a higher demand than usual. Since a car load limit is finite and fuel prices sky rocketed in the recent times, this begs the question: which packages do we deliver and how, in order to make sure both the customer and the provider are happy?

In this project we will provide the answer as well as a solution for the question mentioned above, creating an algorithm that will be able to tell the company which packages should be loaded in a car as well as the route the car needs to follow, all done with the help of genetic algorithms. These are heuristic search and optimization techniques which find their source in the natural selection process, being a simulation of Charles Darwin's evolution model. The keyword in this project is "heuristic"; this means that for certain inputs we will not be able to provide the optimal solution due to the way the algorithm functions, we will, however, provide a result that is as close as possible to the ideal one. This is a sacrifice we have to take when using GAs, as the brute force approach could take billions of years to provide a solution, while our method takes only a few seconds. As a consequence, we choose to take the chance of not providing the best solution possible, gaining massively on the computational time side.

To achieve both the packages that we need to load and how to deliver them we will need to divide our project into two problems: **Package Selection Problem (PSP)** and **Vehicle Routing Problem (VRP),** the merge between the two giving us the solution to our main problem.

In the next chapters we will dive a bit into detail about how each problem is solved and how we took the genetic algorithm concept and adapt it to fit our demands. It is also important we define the following terms, as they will be primarily used in the problem's description:

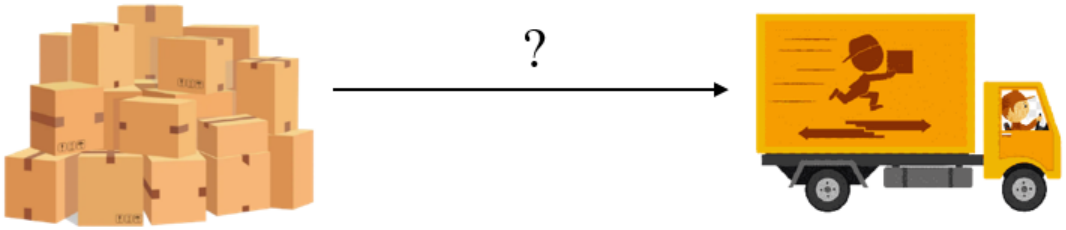
Gene	→	information relevant to the solving of the problem
Chromosome	→	a combination of genes which represents the problem's solution
Individual	→	a possible solution of the problem (made out of one chromosome)
Generation	→	the total number of individuals
Fitness	→	how good is our individual is (solution)
Selection Process	→	choosing the best individuals out of our generation
Crossover	→	mixing the genetic information of two chromosomes
Mutation	→	hanging the information of one chromosome
Parents	→	two individuals chosen for the crossover process
Offspring	→	the resulting individuals after the crossover process is applied with the chosen parents

Package Selection Problem

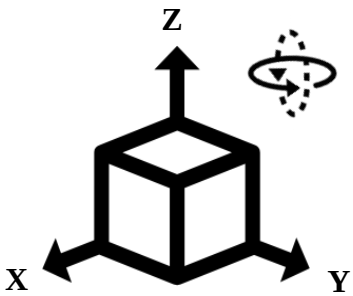
(PSP)

Problem description

The aim of this problem is to obtain an optimal solution (or as close to one as possible) that will tell us which packages we should choose in such a way that profit is maximized and the constraints imposed by the delivery truck's loading space.



Firstly, we will need to find out if there are any packages that can't fit inside our delivery truck. For this we will compare the car's dimensions (length, width, height) and see if there is a possible translation of our object in order to make it fit. Since all packages have a cuboid shape and we work on a 3D space, we will rotate our object 90° on all three axes OX, OY and OZ. In the event that we find a package that can't be placed in our delivery truck we will discard it from our solution.



This process can be seen as a first selection process because we discard all these unfit individuals from the start. While we could keep them and check in the fitness function if our chromosome has a package that can't be fit inside our truck, this would only slow down the algorithm having to recalculate the same thing over and over again. In consequence, for optimization purposes, we will only keep the packages that we know can be placed inside the truck.

Secondly, apart from the constraints imposed by the delivery truck's dimensions, the package selection process will be done based on certain criteria:

1. **If the package has or not priority** (has priority → higher chance picking it)
2. **The package weight and volume** (too heavy/big → won't be picked)
3. **The package value** (higher price → higher chance of picking it)

By optimizing this process with the help of a genetic algorithm we try to satisfy both the customer and the provider. The provider is satisfied by delivering their most valuable products (maximizing profit) and the customer receives their package as fast as possible (the priority of a package determines the quickness of which it is delivered).

Technology and innovation

As we have mentioned in the beginning, we will solve this problem using a genetic algorithm (GA) which mimics the evolution process found in nature. This works on the "survival of the fittest" principle, meaning only the best individuals out of a population will have the chance to reproduce, making sure that we get better and better result with each generation.

The reason why this algorithm is so much faster than the brute force approach is because rather than trying to find a solution for a given problem, it is much easier to verify a result and see if it is or not good. Basically, we generate a bunch of solutions, see which where closest to an ideal one and try to obtain better and better results by merging previously obtained ones.

At their core, all genetic algorithms are very much alike, the main difference being made on how the selection, crossover and the mutation processes are done. In our case, we will use the tournament selection, k-point crossover and the bitflip mutation technique. The mutation process is an optional but important step, as it can help us leave a local convergence point and get us closer to the global one. Its absence could mean we would not be able to break out a local maximum/minimum thus never getting the optimal solution.

One important aspect of using a GA, which is a heuristic optimization technique, is that it does not guarantee that we will always be able to reach the best solution possible, but we will try to get as close to it as possible. For this problem we will try to obtain an above 90% compatibility rate (if an above 90% result is possible).

$$\text{compatibilityRate} = \text{valuePercentage} + \text{priorityPercentage}$$

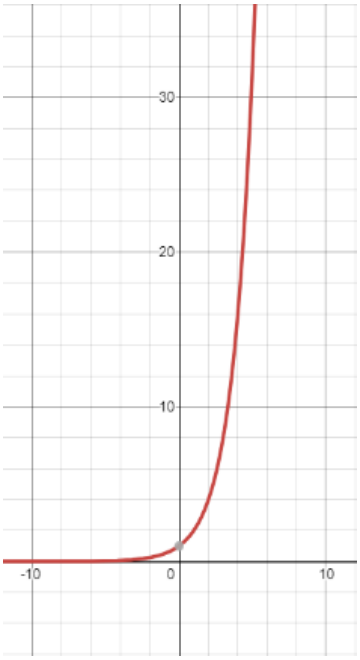
$$\text{valuePercentage} = \text{p\% solutionValue out of totalValue}$$

$$\text{priorityPercentage} = \text{p\% solutionPriorityItems out of totalPriorityItems}$$

Brute Force VS Genetic Algorithm

As stated in the introduction part of our problem, the reason we decided to use a Genetic Algorithm to solve our problem is due to the time complexity. This issue is very important because we can't let the user wait for hours, days or even years for a solution. As a consequence, we need to provide a **reliable** result in as short of a time as possible. Notice the word **reliable**, this means that our algorithm does not guarantee the best solution possible, unlike the brute force approach which will surely give us the optimal result, but in order to reach it we could wait for long periods of time.

To put it into perspective, let's take the knapsack problem, the foundation of the PSP. The brute force approach consists of generating every possible combination of 0's and 1's evaluates each one of them in order to find the optimal solution. The resulting complexity is therefore exponential $O(2^n)$ (in **Figure 1** we have a visual representation for the growth of the function), compared to the Genetic Algorithm's complexity which is in a polynomial form O(g*n*m) (where g-number of generations, n-population size, m-the size of the individual).



(Figure 1)

We can quickly see how for small data sets, the brute force approach is faster than the genetic algorithm, but as the demands increase, the brute force becomes unusable. To give a perspective on how the exponential time complexity affects the running of the program, here are some examples:

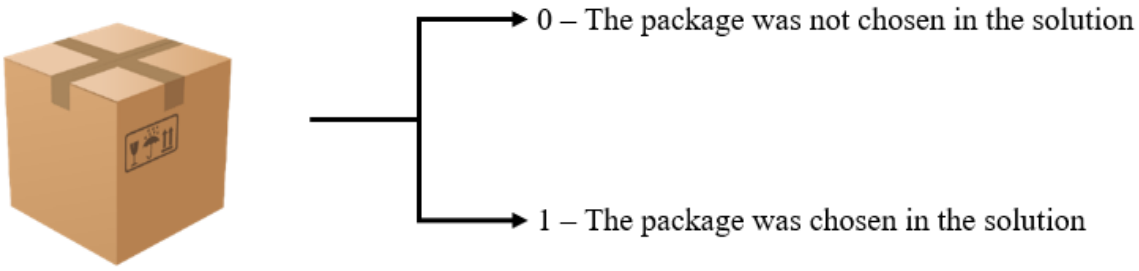
Number of items	Combinations	Brute Force	Genetic Algorithm
5	32	0.000003 seconds	0.0584717 seconds
10	1,024	0.000807 seconds	0.0621908 seconds
20	1,048,576	1.000000 seconds	0.0656834 seconds
21	2,097,152	1.940000 seconds	0.0675304 seconds
22	4,194,304	3.910000 seconds	0.0725825 seconds

A better insight on the times the brute force approach needs to find a solution is given in the video “Genetic Algorithms Explained by Example” by Kie Codes. This is the source we used for the brute force times mentioned above and one more important information that is provided in that video is “only 77 items are enough to occupy my MacBook for 5 billion years”. This just proves that the brute force approach cannot handle bigger data sets, while the genetic algorithm passes with flying colors the time test.

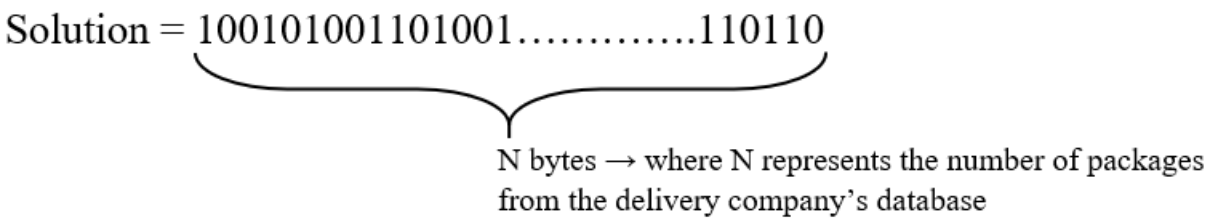
In conclusion, we take the risk of not getting the best solution possible, but something close to it, in just a few seconds, rather than assuring the optimal result in the time the solar system will cease to exist.

How does it work?

For this project we will assume that we have access to the entire delivery company's database, meaning we know how many packages there are, their value and which ones should be prioritized, as well as their dimensions. The genetic algorithm will generate at first a bunch of random solutions, pass them through a fitness function and choose the best ones. After that, those solutions will be intertwined and random mutations will occur, giving us new (and hopefully better) results with each iteration (**Figure 2**).

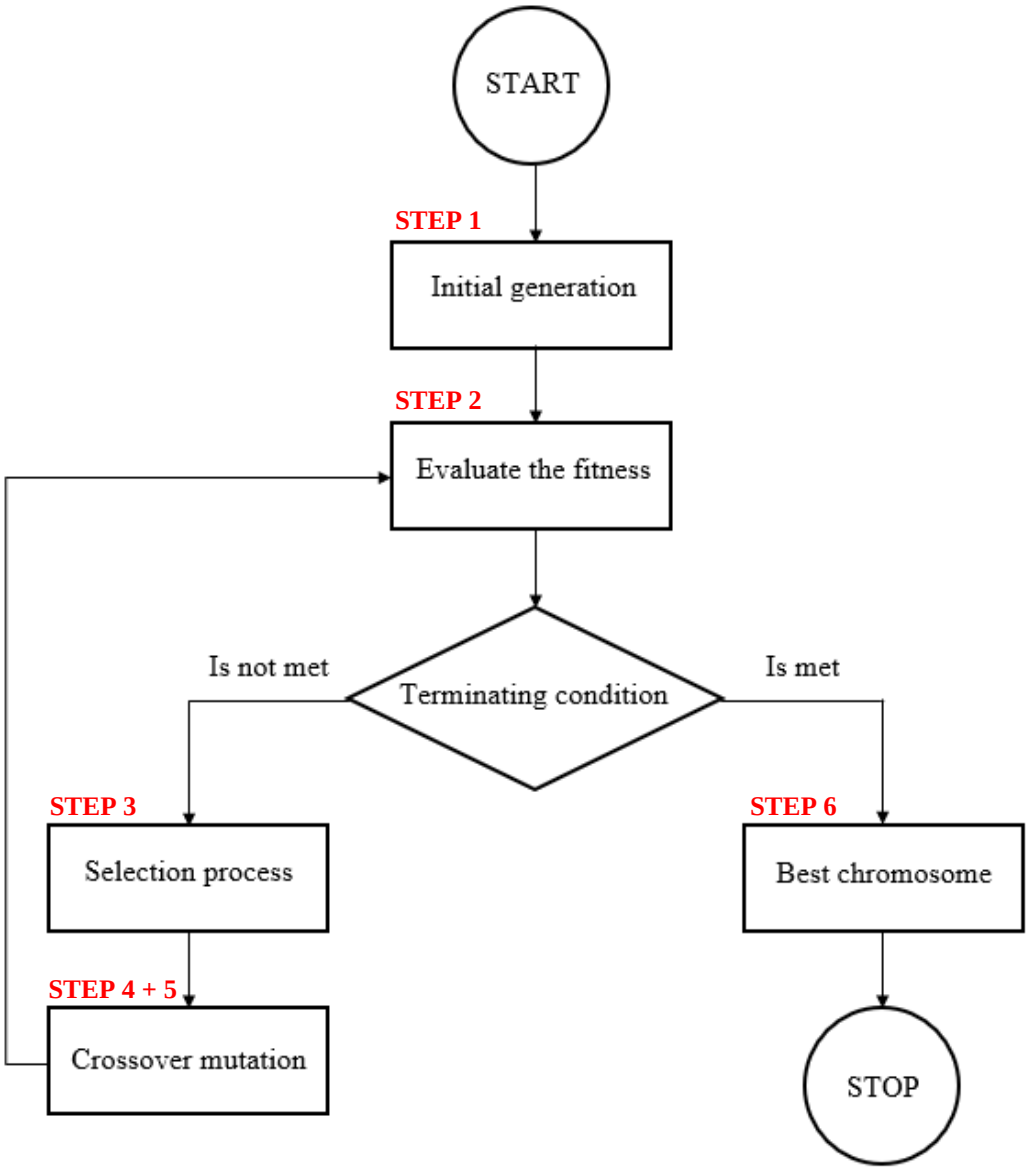


Each solution will be represented as a series of 1's and 0's which will specify if the package $\mathbf{c_i}$ on position **i** was chosen or not. This series of 1's one 0's will represent the gene of an individual (solution) and each solution cluster will represent a generation (the current population).



Steps

- **Step 1** – Initial generation
- **Step 2** – Fitness Evaluation
- **Step 3 –** Selection process
- **Step 4 –** Crossover
- **Step 5 –** Mutation
- **Step 6 –** Selecting the best solution and stopping the execution



(Figure 2)

Step 1 – Initial Generation

The initial generation, or "generation 0" as it is also called, represents the starting point for our algorithm. In order to cover a wider range of solutions we will randomly generate each individual's chromosome, meaning for every position, there will be a random chance that it will either be a 1 or a 0.

Let's take the following example, where for ease of representation, we chose the maximum generation size to be 6, as well as the number of items. As a result, each chromosome will have a length of 6 bytes, each having a complete random chance of being either a 1 or a 0. We get the following generation:


Solution 1 - 101101


Solution 2 - 110001


Solution 3 - 000101


Solution 4 - 111111


Solution 5 - 101011

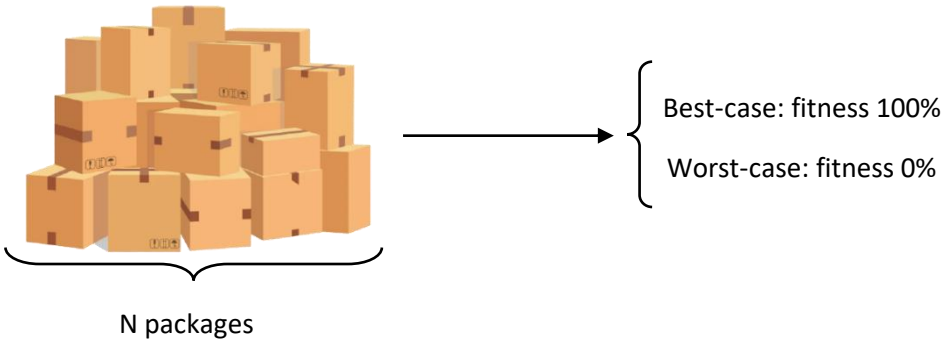

Solution 6 - 110000

It is important to specify that in the real algorithm, the generation size will need to have a bigger value in order to make sure we get more variety through our individuals. For small values we run the risk of not covering a wide enough range and thus having to run the algorithm for more generations than it would be necessary. Also, the random approach is more suitable for this kind of problem as trying to find a generating formula that could also cover the range the random one does is almost impossible, or it would take a lot of time without guaranteeing better results.

Step 2 – Fitness Evaluation

Fitness evaluation is an essential and vital part of our algorithm, as it will provide information about how good a solution is. Every individual of a generation will be passed through the fitness function and a value will be attributed to it. After getting these values for the entire generation, we can start the selection process which will always choose the best individuals from the current population.

As stated before, the fitness function will give us a value, but in order to interpret it, we will have to put it into context. To do so, we need to think what the ideal case and the worst case are for our problem; after establishing these parameters, we can determine how close our solution is to the perfect scenario.



Best-case: in the best-case scenario all the packages we have fit perfectly in our delivery truck. This means not only that we can achieve the maximum value possible, but also that we can transport all the prioritized packages. Taking this into consideration and using the calculating formula described below, we will get a total fitness value of 100%.

Worst-case: in the worst-case scenario we have two possible situations: either the packages we tried to load are too heavy or too big for the truck to carry and in that case the fitness value will be automatically set to 0, or out of all the packages that we have available we can't load any of them. This is just a theory as the chances of having a package that is much bigger than a delivery truck or heavier than its load capacity is very slim, but in that event, the fitness value will as well be set to 0.

Fitness formula

For starters, we need to know out of all the packages that we have selected to load into our truck, how many of them are prioritized and the load's total value.

$$\text{value} = \sum_{i=0}^{n} \begin{cases} v_i, & g_i = 1 \\ 0, & g_i = 0 \end{cases} \qquad \text{priorityItems} = \sum_{i=0}^{n} \begin{cases} 1, & p_i = \text{true} \\ 0, & p_i = \text{false} \end{cases}$$

Where v_i and p_i represents the value / priority of the item at index i. g_i is the chromosome value at position i and n is the total number of items available.

After getting these values, we can calculate the percentage they represent out of all the available packages.

$$\text{valuePercentage} = \frac{100 * \text{value}}{\text{totalValue}} \quad \text{priorityItemsPercentage} = \frac{100 * \text{priorityItems}}{\text{totalPriorityItems}}$$

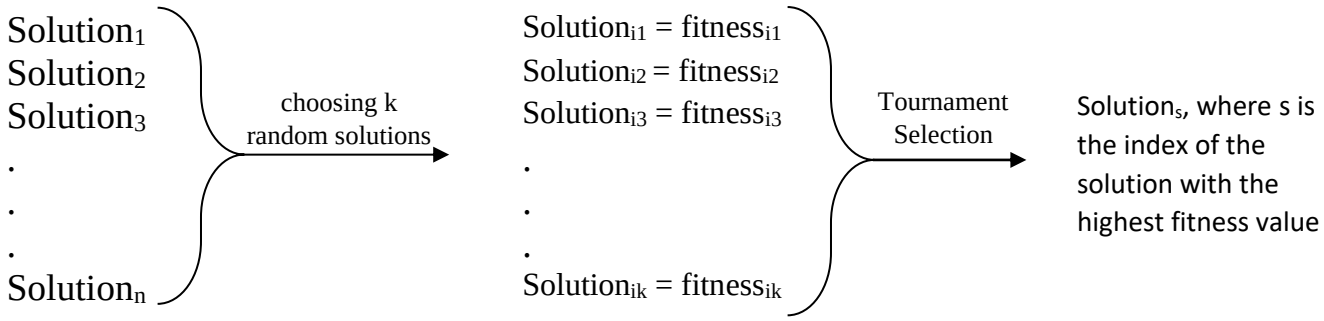
We now have all the necessary information to calculate the fitness value of one individual. This will be equal to the average of the two variables mentioned above.

$$\text{fitness} = \frac{\text{valuePercentage} + \text{priorityItemsPercentage}}{2}$$

The final value will represent the "compatibility rate", or how close we are to an ideal case. There are also some conventions we are going to make, such as if the total mass of the load is higher than what the truck can carry the fitness value will be defaulted to 0, same goes for when the volume exceeds the limits (as stated in the worst-case scenario explanation).

Step 3 – Selection Process

After each generation is created, we will need to select the best individuals that will produce our offspring. This process will be done using the Tournament Selection method. The way this works is that each time we want to choose a parent, we will have a predefined number of contestants (noted with k < the number of items), and we will choose, out of all our individuals, k random contestants. The winner of this so called "tournament" will be the individual with the biggest fitness value, thus earning the chance to pass its genes.



This selection method will be used each time we want to choose a parent, in order to make sure that only the best individuals will pass their genes over to the next generation. After the crossover of these parents, we will be left with two offspring, both of them becoming individuals of the next generation. The process of parent selection and offspring generation will keep running until the new generation's size matches the previous one.

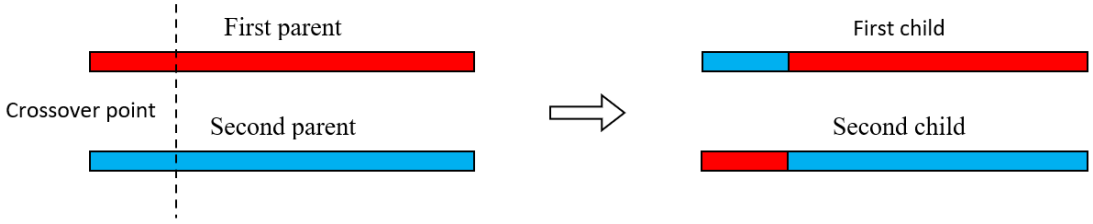
In the event that the number of individuals that are in a population is not even, before starting the parent selection and crossover process, we will apply Elitism. With Elitism we will ensure that the best individual of our current generation will make it into the next one. This is done by specially targeting the fittest individual from our current population and automatically choosing it in our next generation.

Last but not least, the reason why we randomize the selection of our contestants is to increase the diversity of our offspring. Choosing only the best ones could possibly get us stuck in a local convergence point.

Step 4 – Crossover

The crossover process is the most important part of a genetic algorithm as it dictates how our offspring will be creating, each method yielding different results. For our problem we decided create our new genes using **one-point crossover** (also named k-point crossover where k = 1). This process consists of randomly generating a position p, taking the first p genes and swapping them.

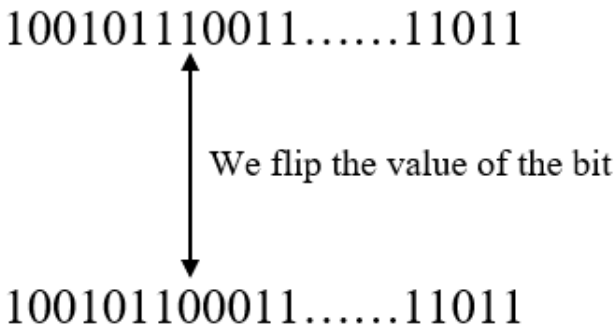
After each crossover operation is applied to our selected individuals, we will be resulted with two offspring, both containing genetic information from their parents. These new solutions will now form the individuals for our new generation.



Step 5 – Mutation

Mutation, just as crossover, is vital for the functionality of our genetic algorithm. Usually, when we find a good solution, we want to keep it and ignore the others, but there are events where the crossover of two bad parents can result in a very good offspring. Without mutation, we will only select the best individuals and this could result in our algorithm getting stuck in a local maximum/minimum interval. Altering an offspring's genome (with a very low probability) could help us break out of such zones and reach the global optimum.

The technique used to solve this problem is **bitflip mutation**. This works by randomly selecting a gene in our chromosome and flipping its value, meaning if the current value is 0, we will turn it into 1 and vice versa if the value is 1 it will be turned into 0.



Step 6 – Selecting the best solution and stopping the execution

This process of replacing the old generation with the new improved created one (also called an iteration) will run until a stopping criterion is met or we have reached our max iterations count. In our case, if we have multiple generations where the best solution is the same, we will consider we have reached the point of stopping the execution, as the individuals are no longer evolving. The number of generations that we will look back at will depend on our preference.

One observation that we have to make is with each iteration, we are not guaranteed that the offspring will be better than its parents and this could lead to a final generation worse than the previous ones. This is what our stopping condition is trying to avoid but to be sure we won't lose a good solution, our generation size will always be an uneven number, forcing the Elitism process on our population. As a result, we make sure that whatever the best solution currently is, it will make it into the next generation.

After the execution is stopped, we will simply select the best individual out of our generation. This will represent our final solution and also the best one regarding the packages that should be placed inside our delivery truck.

Where we stand and what already exists

Since the arrangement of a given number of items in a confined space is a pretty common optimization problem, it is not a surprise that there are many ways people have interpreted it over the years, adding their own twists. Our method is inspired by the very well-known Knapsack Problem which asks the question of placing a series of object inside a rucksack in such a manner that the profit is maximized.

In order to find the answer to our question, we took the concept of the Knapsack Problem which works on a 2D plane and modify it to work in a 3D space. We also added additional constraints such as the form factor, which plays an important role in choosing (or not) a certain package, as well as taking into consideration the priority of a certain item.

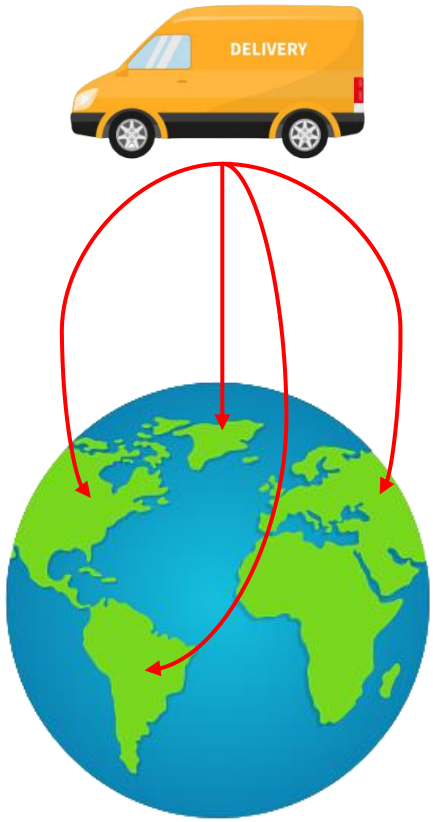
In the end, what we tried to do is take an already existing concept, improve on it and modify it in order to get something that can be applied in a real-life scenario. After getting the optimal solution, this will be passed onto the routing algorithm, which will calculate the optimal route for delivering all the packages to their destinations.

Vehicle Routing Problem

(VRP)

Problem Description

The question that we are trying to answer in this part of the project is how, or better said, in which order should we deliver our packages in order to minimize the total distance travelled by our car. This problem works together with the PSP part of our project because once we know which packages should be loaded in our truck, we also get the information regarding their destinations.



As we have mentioned in the first part, our project aims to ensure both customer and provider satisfaction, but we can't forget the delivery company which also need to turn in a profit. This is the reason why we try to optimize the route which the vehicle will take, so transportation costs are kept as low as possible, especially in a time were fossil fuels are getting more and more expensive.

For our problem, we will see each destination as a point in a 2D plane, where its position is given by the location coordinates (latitude and longitude). We end up with a system with points places at certain known coordinates (meaning we can calculate the distance from point A to point B) and we are trying to connect each one of them in a way that the total distance is as short as possible.

Last but not least, since we need to go through all our points, there will be no criteria which will dictate if a certain point will or not be chosen. Our generation will be composed of permutations of the destination's indices.

Technology and innovation

The vehicle routing problem, just as the Travelling Salesman Problem, is a np-complete problem, meaning there is no specific efficient algorithm that can provide a reliable solution. We previously mentioned the Travelling Salesman Problem or TSP for short because since we are only working with one vehicle, our solution finds its roots planted deeper in the TSP rather than the VRP, as the VRP approach handles multiple cars at once.

Considering there is no defined way to approach these types of problems, we decided to use a genetic algorithm again. While the main ideas of a GA stay the same, the biggest change will be seen in the way we make our crossover and mutation operation, the variant used in the PSP part of our project not working anymore. Another drastic change will be seen in the encoding of our solution, while previously we were using a bunch of 0's and 1's in order to determine which package was chosen, here we will need to find a better solution in order to represent our data.

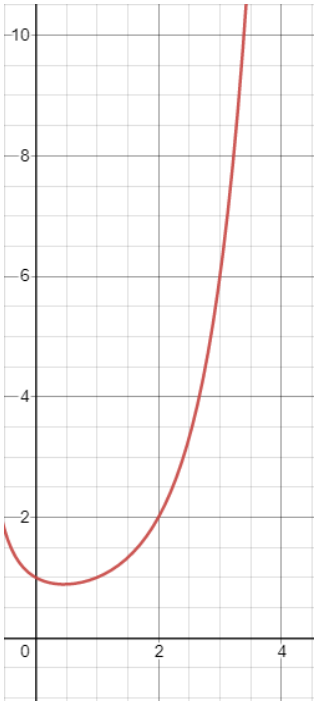
In the end, the mechanics of the genetic algorithm presented in the introduction won't change, however, the way we attack each step and try to solve it will require new methods and processes. These things are necessary in this context, as we no longer work with packages that can be chosen or not, but we try to generate path for our car to take and new constraints had to be introduced.

Brute Force VS Genetic Algorithm

Once again, the reason behind using a Genetic Algorithm instead of the brute force approach is time related. This time, instead of trying to choose each combination of 1's and 0's, since we work with a series of numbers that need to have a certain order, the brute force will have to calculate every possible permutation of our destination's indexes.

As a result, the complexity of the brute force algorithm will have a factorial form, more precisely, it is O(n!), where n! represents the number of permutations of n destinations. Once again, the complexity of the Genetic Algorithm will be O(g*n*m) (g-number of generations, n-population size, m-the size of the individual).

In **Figure 3** we have a visual representation of how the factorial function progresses as the data sets get larger and larger. We can see a sudden rise of the graph when the algorithm hits a certain value, ending up in the same scenario as with the PSP. The choice of choosing a Genetic Algorithm over a brute force one becomes obvious, as providing a good solution that is close to the optimal one in just a few seconds is much better than waiting for billions of years to get the perfect one.



(Figure 3)

To get a better understanding of how much time the brute force approach needs, we have the following table, the source of the brute force times being the site "toward data science", which gives a very good insight on how data size can affect the running of the algorithm.

Number of destinations	Permutations	Brute force	Genetic Algorithm
10	3,628,800	0.003 seconds	0.0747338 seconds
20	2,432,902,008,176,640,000	77 years	0.0847372 seconds
30	2.652528598 E+32	490 million years	0.4779625 seconds
40	8.159152832 E+4	10^{15} years	0.4999309 seconds
50	3.04140932 E+64	10^{49} years	0.5131843 seconds

As we can see, the time difference between the genetic algorithm that finds the best route and the one that selects the packages that need to be placed inside the truck is not that big. We can conclude that even though our algorithms solve two completely different things, the time complexity will be similar. This results from the fact that the complexity is dependent on the generation size, population size and individual size and not on the task that we need to solve. In comparison, the brute force's complexity will change with the task, the time complexity being better for some cases and worse for other; however, in all scenarios, once the data sets get larger this type of approach becomes unusable.

Times were calculated considering a computer can make 10^9 operations / second.

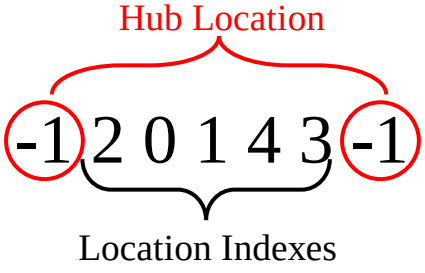
How does it work?

As stated previously, we will no longer be able to apply the same logic as we did in the PSP part of our project. The first problem that we will encounter is how we will encode our solution. This time, instead of using a series of 1's and 0's, our individual's chromosome will represent the path of the car. Each gene will be a city index, and the order of the indexes will represent the route our delivery truck will take (**Figure 4**).

Let's say the packages from our solution have the following destinations:

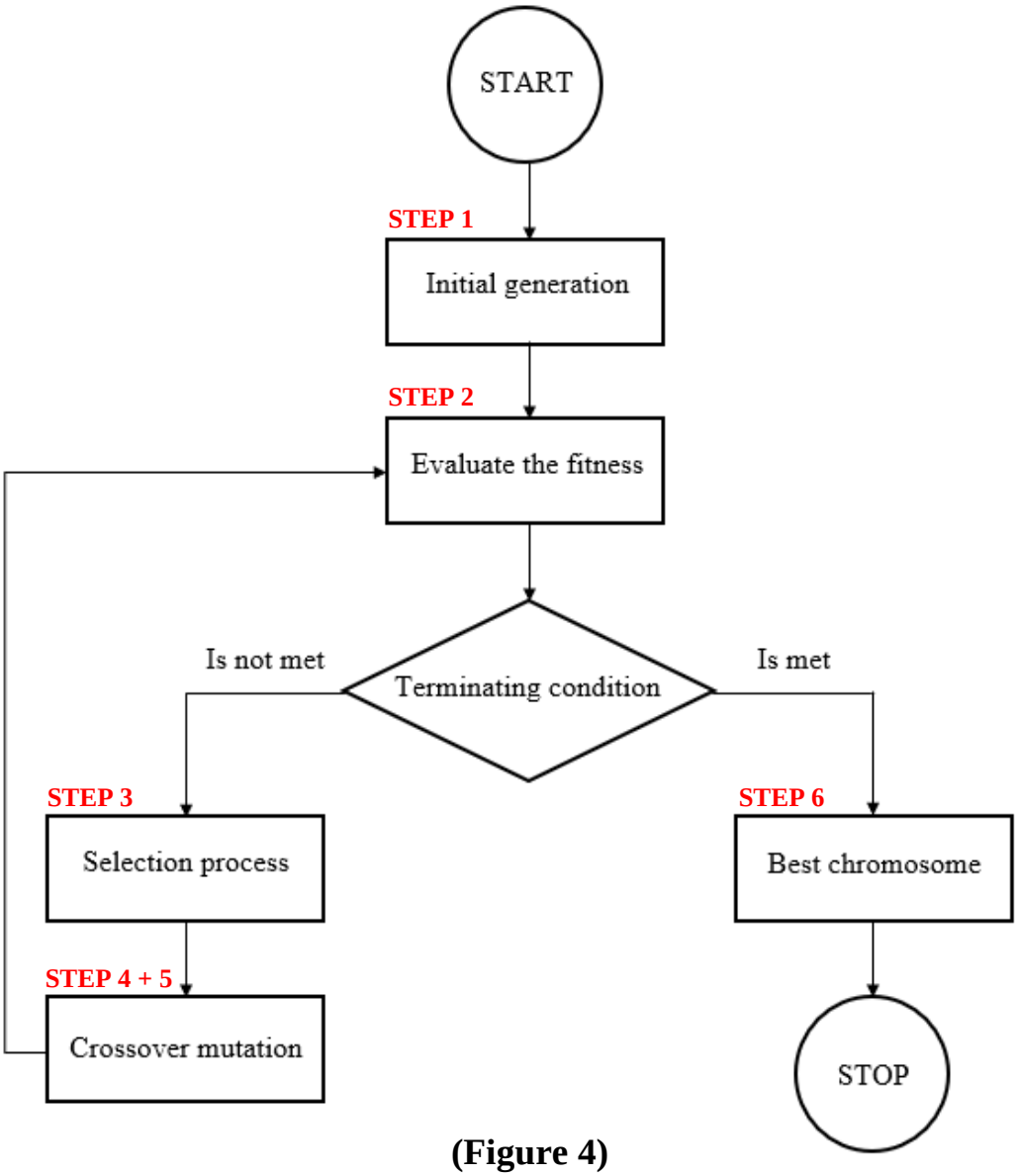
- Sibiu (index = 0)
- Satu-Mare (index = 1)
- Bucuresti (index = 2)
- Cluj (index = 3)
- Constanta (index = 4)

We will also need the location of the hub, as that will be our starting point, and by default we decided to index it with the value with -1. Since we need to come back to the place we started, each one of our chromosomes will have as a starting value and end value -1, in between having a permutation of the destination's indexes.



Steps

- **Step 1** – Initial generation
- **Step 2** – Fitness Evaluation
- **Step 3 –** Selection process
- **Step 4 –** Crossover
- **Step 5 –** Mutation
- **Step 6 –** Selecting the best solution and stopping the execution



(Figure 4)

Step 1 – Initial Generation

For our starting generation, just as in the PSP part of the project, will be randomly generated in order to increase the diversity of our individuals. However, this time we will need to be careful which destinations we have already entered in our current solution, not doing so leading to an infinite loop and ultimately to our algorithm crashing. To avoid this, we will use an auxiliary array with the role of marking each city that has been added to the solution, thus avoiding repeating the same destination twice.

Solution						
-1						-1

Marked				
0	0	0	0	0

Solution						
-1	2					-1

Marked				
0	0	1	0	0

Solution						
-1	2	0				-1

Marked				
1	0	1	0	0

Solution						
-1	2	0	4			-1

Marked				
1	0	1	0	1

Solution						
-1	2	0	4	3		-1

Marked				
1	0	1	1	1

Solution						
-1	2	0	4	3	1	-1

Marked				
1	1	1	1	1

Once again, the size of a generation will play an important role in obtaining a higher diversity amongst our individuals. The bigger it is, the better the chances of generating a good solution while also giving the algorithm room to choose from for the crossover process.

Step 2 – Fitness Evaluation

For each individual, the fitness value will be determined by the total distance the delivery truck will need to travel. In order to calculate the distance between two point we will consider the road as a straight line, so we can use the following formula:

$$d(A,B) = \sqrt{(x_B - x_a)^2 + (y_B - y_A)^2}$$

Since we know how to calculate de distance between to points, we can now use this formula and calculate the distance between two adjacent destinations and the final fitness value will represent the sum of all the distances.

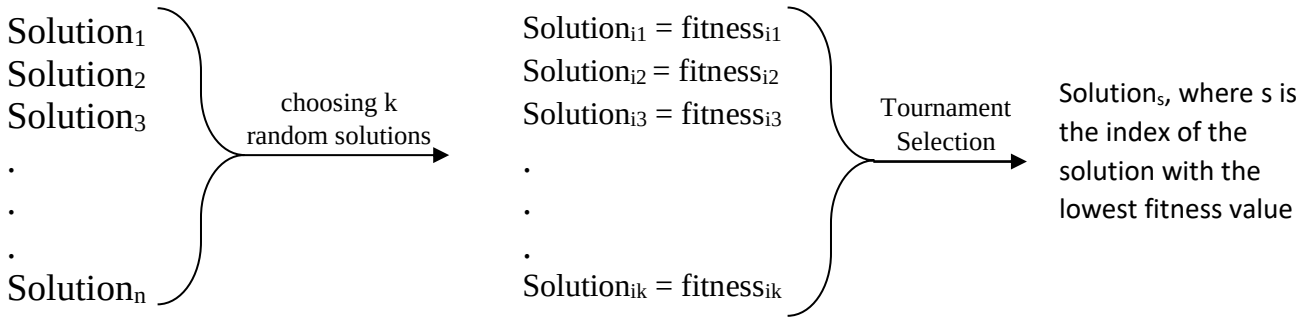
$$\text{fitness} = \sum_{i=0} d(p_i, p_{i+1})$$

where p_i represents the city of index i from our solution.

The end result will represent the total distance that our car will need to travel in order to deliver all its packages to their destinations. Since we are aiming to get the shortest route possible, the lower the fitness value the better that individual will be. In other terms, our algorithm aims to find the global minimum of our function.

Step 3 – Selection Process

The selection process will be done using the Tournament Selection method. This means that each time we want to select a parent for our offspring, we will take a given number of contestants and face them against each other. In the end, the individual with the lowest fitness value (the lowest distance) will come out as the winner. Each of these contestants will be randomly picked from our generation pool.



It is important to know that this time, the crossover process will only yield one offspring. As a result, Elitism, the process of taking the best individual out of the population and placing it in the following generation, will become a mandatory process to ensure our population evolution. While it may seem contradictory to search for the lower fitness value, we have to remember that our problem searches for the minimum possible route, so the lower the fitness, the shorter the traveling distance is.

Last but not least, the reason why we randomize the selection of our contestants is to increase the diversity of our offspring. Choosing only the best ones could possibly get us stuck in a local convergence point.

Step 4 – Crossover

The crossover process is one of the most important parts of our algorithm, as it dictates the creation of our new generation. This time, since we are dealing with connected points that need to be selected exactly once, we will no longer be able to use the k-point crossover, as it could result in the repetition of the same destination place. The solve this issue, we will use a special breeding method called **ordered crossover**.

Ordered crossover consists of randomly choosing a subset of random length ($0 \leq$ length $<$ gene size) from our first parent and fill the remainder of the route with points from the second parent in the order that they appear. In the event that we reach a destination in the second parent that was already introduced into our solution, we will simply skip it. The first and last element of our offspring will have the value of -1 (check the convention made at the beginning of the chapter)

First Parent

-1	2	0	1	4	3	-1

Second Parent

-1	4	2	1	0	3	-1

Child

			1	4	3	

-1	2	0	1	4	3	-1

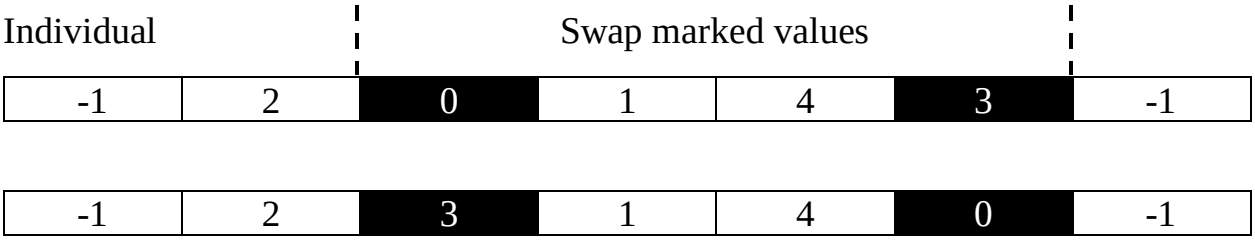
-1	2	0	1	4	3	-1

As in the previous examples, randomizing this process ensure we get better diversity each time we apply this process. The selection and crossover processes are going to run until the new generation's size matches the old one.

Step 5 – Mutation

Mutation, just as crossover, is an essential part of our algorithm, altering an individual's gene in order to add diversity and extend our solution range. This also prevents us from getting stuck in a local convergence point, adding new routes that expand our search space.

If we would've had a chromosome made out 0's and 1's, all we needed to do is assign a low probability for one of these bits to change its value (see PSP mutation). However, since we are dealing with routes that have a certain index and need to be added only once into our solution, we need to find an alternative method. As a result, we will use the **swap mutation** technique. This works by randomly choosing two destinations and swapping their places. It is important to say that the start and end position of our chromosome will be ignored when choosing the 2 cities, as they represent the starting respectively ending point of our route, and changing their place could lead to errors.



Individual after mutation is applied

-1	2	3	1	4	0	-1

Step 6 – Selecting the best solution and stopping the execution

The process of creating and updating a generation (also called an iteration) will run until a stopping condition is met or we have reached the maximum generation count. In our case, we will evaluate the best solutions obtained in the previous generation, the number of generations that we look back at being a given value, and check the best solutions. If there is no improvement, we will consider that the algorithm has reached the optimal solution and therefore we can stop it.

Just like the PSP, we are not guaranteed that the new generations will be better than the previous ones, but now we do not care what the generation size will be, because Elitism is a mandatory process applied to a population. As a result, nu matter what, we will always keep the best solution achieved so far.

After the algorithm is stopped, we will have a series of indexes, representing the order in which our truck will deliver its cargo. The fitness of this solution will represent the total distance our delivery car has to travel in order to cover all our destinations.

Where we stand and what already exists

As the name of this chapter suggests, the problem that this part of our project is based on is the vehicle routing problem (VRP). This problem tackles the question of choosing a series of destination points and ordering them in such a way that the final distance is as short as possible.

The VRP can handle this ordering process for multiple cars at once, but since we are dealing with only one vehicle, we find that our problem has its roots planted deeper in the travelling salesman problem. Both interpretations require the vehicle to return to its starting point, as a result we can say that the vehicle routing problem is a generalization of the traveling salesman problem.

Tests and results

Hardware

The algorithm was tested on an ASUS TUF F15 laptop with the following specification:

Processor: Intel Core i7 – 10870H, 2.20 GHz (8 cores and 16 threads)

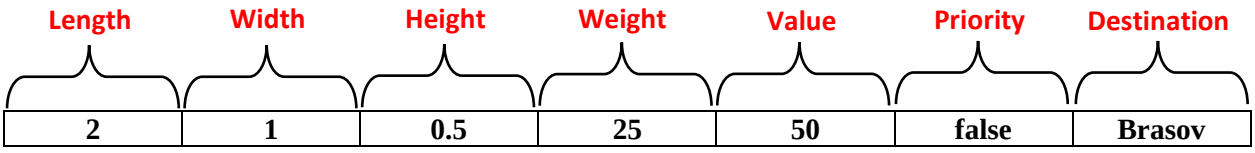
RAM: 16GB of DDR4 memory running at 2933 MHz

Graphics Card: dedicated Nvidia GTX 1660 TI with 6GB DDR5 video memory

Storage: 512GB M.2 2280 Micron SSD

For each test, the algorithm was runed five times to see how the heuristic nature of the GA affects the final result.

Example



Car details	Max weight: 100 Length: 5 Width: 10 Height: 5
Collet's details	2 1 0.5 25 50 false Brasov 0.5 1 1 10 100 true Timis 0.8 0.3 0.5 15 25 false Cluj 2 0.5 1 30 42 true Satu-Mare 6 3 6 50 18 false Gorj
VRP Settings	Hub: Brasov Generation Size: 10 Tournament Contestants: 5 Crossover Rate: 0.8 Mutation Rate: 0.1 Max Generations: 25 Max Previous: 5
PSP Settings	Generation Size: 20 Max Generations: 25 Items Number: 5 Crossover Rate: 0.8 Mutation Rate: 0.1 Tournament Contestants: 5 Max Previous: 5
Results	The selected items are: 1. Weight: 25.0, Volume: 1.0, Value: 50.0, Destination: Brasov 2. Weight: 10.0, Volume: 0.5, Value: 100.0, Destination: Timis 3. Weight: 15.0, Volume: 0.12, Value: 25.0, Destination: Cluj 4. Weight: 30.0, Volume: 1.0, Value: 42.0, Destination: Satu-Mare **Collet's destinations** The route is Brasov→Brasov→Satu-Mare→Timis→Cluj→Brasov with a total distance of 1059.10km **Hub Location**

Tests and Results

Test no. 1

<table>
<tr><td>Car details</td><td>Max weight: 100
Length: 5
Width: 10
Height: 5</td></tr>
<tr><td>Collets details</td><td>2 1 0.5 25 50 false Brasov
0.5 1 1 10 100 true Timis
0.8 0.3 0.5 15 25 false Cluj
2 0.5 1 30 42 true Satu-Mare
6 3 6 50 18 false Gorj</td></tr>
<tr><td>VRP Settings</td><td>Hub: Brasov
Generation Size: 10
Tournament Contestants: 5
Crossover Rate: 0.8
Mutation Rate: 0.1
Max Generations: 25
Max Previous: 5</td></tr>
<tr><td>PSP Settings</td><td>Generation Size: 20
Max Generations: 25
Items Number: 5
Crossover Rate: 0.8
Mutation Rate: 0.1
Tournament Contestants: 5
Max Previous: 5</td></tr>
<tr><td>Results</td><td>The selected items are:
1. Weight: 25.0, Volume: 1.0, Value: 50.0, Destination: Brasov
2. Weight: 10.0, Volume: 0.5, Value: 100.0, Destination: Timis
3. Weight: 15.0, Volume: 0.12, Value: 25.0, Destination: Cluj
4. Weight: 30.0, Volume: 1.0, Value: 42.0, Destination: Satu-Mare

The route is:
<table>
<tr><th>First run</th><th>Second run</th><th>Third run</th></tr>
<tr><td>Brasov→Brasov→Satu-Mare →Timis→Cluj→Brasov with a total distance of 1059.10km

The running time of the program is 0.0594298 seconds</td><td>Brasov→Timis→Cluj→Satu-Mare →Brasov→Brasov with a total distance of 1084.36km

The running time of the program is 0.0696668 seconds</td><td>Brasov→Satu-Mare→Timis→Cluj →Brasov→Brasov with a total distance of 1059.10km

The running time of the program is 0.0635459 seconds</td></tr>
</table></td></tr>
</table>

The other 2 tests gave the same results as those in the table.

Test no. 2 (same data as first test, but multiply by 10000 the Generation size)

Car details	Max weight: 100 Length: 5 Width: 10 Height: 5
Collets details	2 1 0.5 25 50 false Brasov 0.5 1 1 10 100 true Timis 0.8 0.3 0.5 15 25 false Cluj 2 0.5 1 30 42 true Satu-Mare 6 3 6 50 18 false Gorj
VRP Settings	Hub: Brasov Generation Size: 10**0000** Tournament Contestants: 5 Crossover Rate: 0.8 Mutation Rate: 0.1 Max Generations: 25 Max Previous: 5
PSP Settings	Generation Size: 20**0000** Max Generations: 25 Items Number: 5 Crossover Rate: 0.8 Mutation Rate: 0.1 Tournament Contestants: 5 Max Previous: 5
Results	**The selected items are:** 1. Weight: 25.0, Volume: 1.0, Value: 50.0, Destination: Brasov 2. Weight: 10.0, Volume: 0.5, Value: 100.0, Destination: Timis 3. Weight: 15.0, Volume: 0.12, Value: 25.0, Destination: Cluj 4. Weight: 30.0, Volume: 1.0, Value: 42.0, Destination: Satu-Mare **The route is:**

First run	**Second run**
Brasov→Cluj→Timis→Satu-Mare→Brasov→Brasov with a total distance of **1059.10km** The running time of the program is **2.1210687 seconds**	Brasov→Brasov→Cluj→Timis→Satu-Mare→Brasov with a total distance of **1059.10km** The running time of the program is **2.0741 seconds**

The other 3 tests gave the same results as those in the table.

Test no. 3

<table>
<tr><td>Car details</td><td colspan="5">Max weight: 580
Length: 12
Width: 16
Height: 8</td></tr>
<tr><td>Collets details</td><td colspan="2">2 1 0.5 25 50 false Brasov
0.5 1 1 10 100 true Timis
0.8 0.3 0.5 15 25 false Cluj
2 0.5 1 30 42 true Satu-Mare
6 3 6 50 18 false Gorj
9 5 6 70 650 true Bucuresti</td><td colspan="3">2 3 1 3 20 false Braila
1 2 3 1 30 false Alba
6 2 1 13 4 true Constanta
5 5 5 9 150 false Hunedoara
3 6 9 16 98 false Iasi
1 1 1 5 80 true Sibiu</td></tr>
<tr><td>VRP Settings</td><td colspan="2">Hub: Buzau
Generation Size: 35
Tournament Contestants: 5
Crossover Rate: 0.8</td><td colspan="3">Mutation Rate: 0.1
Max Generations: 25
Max Previous: 5</td></tr>
<tr><td>PSP Settings</td><td colspan="2">Generation Size: 15
Max Generations: 25
Items Number: 12
Crossover Rate: 0.6</td><td colspan="3">Mutation Rate: 0.2
Tournament Contestants: 5
Max Previous: 5</td></tr>
<tr><td rowspan="3">Results</td><td colspan="3">The selected items are:
1. Weight: 25.0, Volume: 1.0, Value: 50.0, Destination: Brasov
2. Weight: 10.0, Volume: 0.5, Value: 100.0, Destination: Timis
3. Weight: 15.0, Volume: 0.12, Value: 25.0, Destination: Cluj
4. Weight: 30.0, Volume: 1.0, Value: 42.0, Destination: Satu-Mare
5. Weight: 50.0, Volume: 108.0, Value: 18.0, Destination: Gorj
6. Weight: 70.0, Volume: 270.0, Value: 650.0, Destination: Bucuresti
7. Weight: 3.0, Volume: 6.0, Value: 20.0, Destination: Braila
8. Weight: 1.0, Volume: 6.0, Value: 30.0, Destination: Alba
9. Weight: 13.0, Volume: 12.0, Value: 4.0, Destination: Constanta
10. Weight: 9.0, Volume: 125.0, Value: 150.0, Destination: Hunedoara
11. Weight: 16.0, Volume: 162.0, Value: 98.0, Destination: Iasi
12. Weight: 5.0, Volume: 1.0, Value: 80.0, Destination: Sibiu
The route is:</td><td colspan="2">The selected items are:
1. Weight: 25.0, Volume: 1.0, Value: 50.0, Destination: Brasov
2. Weight: 10.0, Volume: 0.5, Value: 100.0, Destination: Timis
3. Weight: 15.0, Volume: 0.12, Value: 25.0, Destination: Cluj
4. Weight: 30.0, Volume: 1.0, Value: 42.0, Destination: Satu-Mare
5. Weight: 70.0, Volume: 270.0, Value: 650.0, Destination: Bucuresti
6. Weight: 3.0, Volume: 6.0, Value: 20.0, Destination: Braila
7. Weight: 1.0, Volume: 6.0, Value: 30.0, Destination: Alba
8. Weight: 13.0, Volume: 12.0, Value: 4.0, Destination: Constanta
9. Weight: 9.0, Volume: 125.0, Value: 150.0, Destination: Hunedoara
10. Weight: 16.0, Volume: 162.0, Value: 98.0, Destination: Iasi
11. Weight: 5.0, Volume: 1.0, Value: 80.0, Destination: Sibiu
The route is:</td></tr>
<tr><td>First run</td><td>Second run</td><td>Third run</td><td>Fourth run</td><td>Fifth run</td></tr>
<tr><td>Buzau→ Constanta→ Bucuresti→ Braila→ Iasi→Sibiu→ Alba→ Cluj→ Timis→ Hunedoara→ Gorj→ Brasov→ Satu-Mare→ Buzau with a total distance of 2293.02km

The running time of the program is 0.0763164 seconds</td><td>Buzau→ Braia→ Satu-Mare→ Bucuresti→ Gorj→ Timis→ Cluj→ Hunedoara→ Alba→ Sibiu→ Brasov→ Iasi→ Constanta→ Buzau with a total distance of 2302.89km

The running time of the program is 0.0693379 seconds</td><td>Buzau→ Bucuresti→ Satu-Mare→ Brasov→ Sibiu→ Gorj→ Hunedoara→ Timis→ Alba→ Cluj→ Iasi→ Braila→ Constanta→ Buzau with a total distance of 1997.04km

The running time of the program is 0.0785689 seconds</td><td>Buzau→ Bucuresti→ Brasov→ Alba→ Hunedoara→ Satu-Mare→ Sibiu→ Cluj→Timis→ Gorj→ Iasi→Braila→ Constanta→ Buzau with a total distance of 2642.42km

The running time of the program is 0.0681297 seconds</td><td>Buzau→Iasi→Constanta→Bucuresti→Satu-Mare→Brasov→ Sibiu→Alba→Hunedoara→Timis→Cluj→Braila→Buzau with a total distance of 2210.45km

The running time of the program is 0.0902776 seconds</td></tr>
</table>

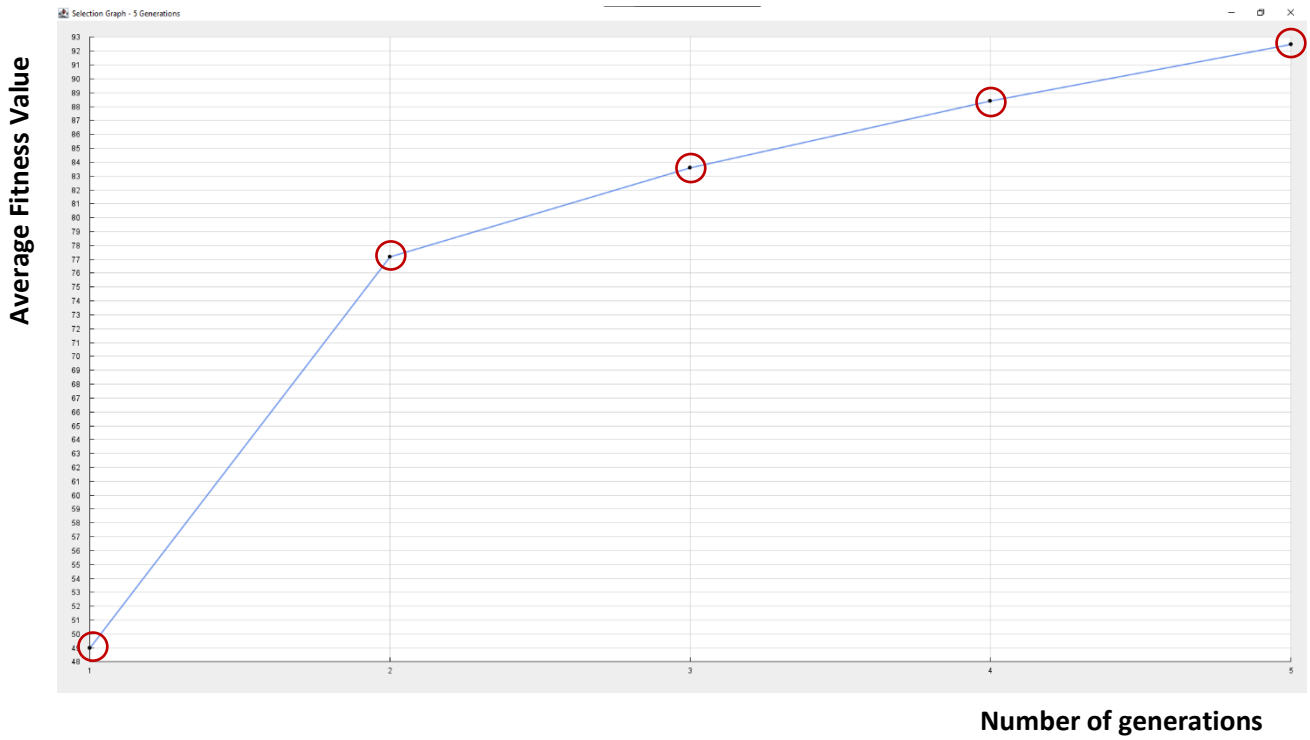
For a better understanding of how the algorithm works, we will present the function graph for each one of the tests. We will have a graph for the genetic algorithm that handles the selections process and one for the routing algorithm.

PSP Graph – was made using the average fitness of each generation

VRP Graph – was made by choosing the best and the worst individual out of each generation

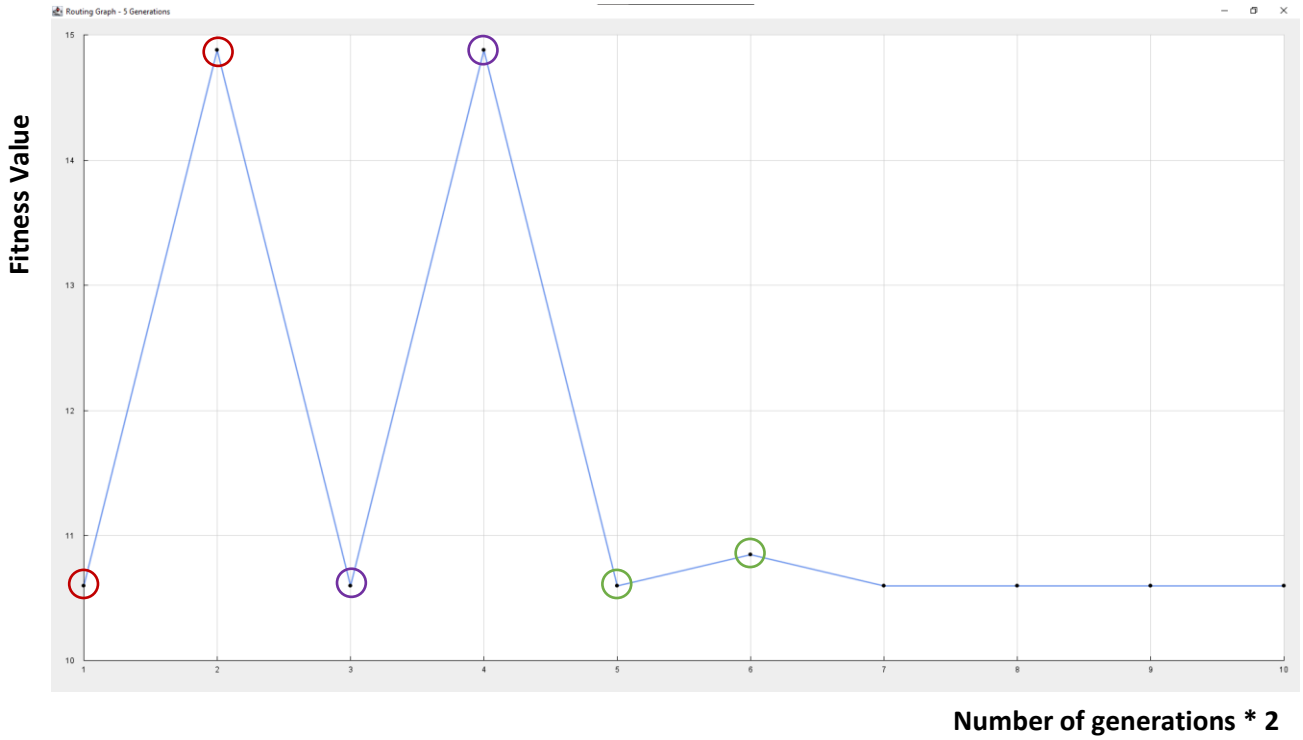
The reason why we decided to add a graph drawing feature to our project is pretty simple, for easier observation of the algorithm. Having a visual representation of how the solutions change over the course of multiple generations can help us understand better the processes that are behind a genetic algorithm and see if we actually improve with every iteration or not. This should be easily visible on the graph (for PSP the graph should ascend, for VRP descend).

PSP Graph - Explanation



Each dot on the X axis represents the number of generations and on the Y axis we have the value of the average fitness from that population.

VRP Graph - Explanation



First ◯ represents the worst-fitness of the current generation and the second one represents the best-fitness value from the same generation. } **1st generation**

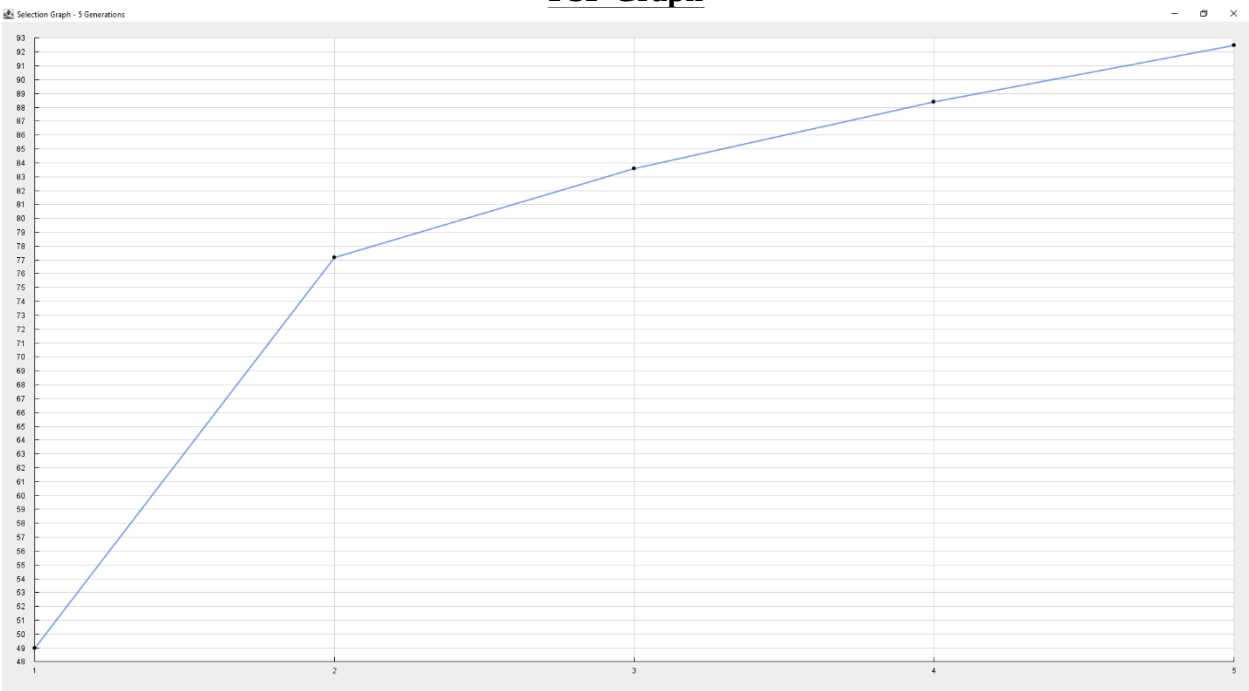
First ◯ represents the worst-fitness of the current generation and the second one represents the best-fitness value from the same generation. } **2nd generation**

First ◯ represents the worst-fitness of the current generation and the second one represents the best-fitness value from the same generation. } **3rd generation**

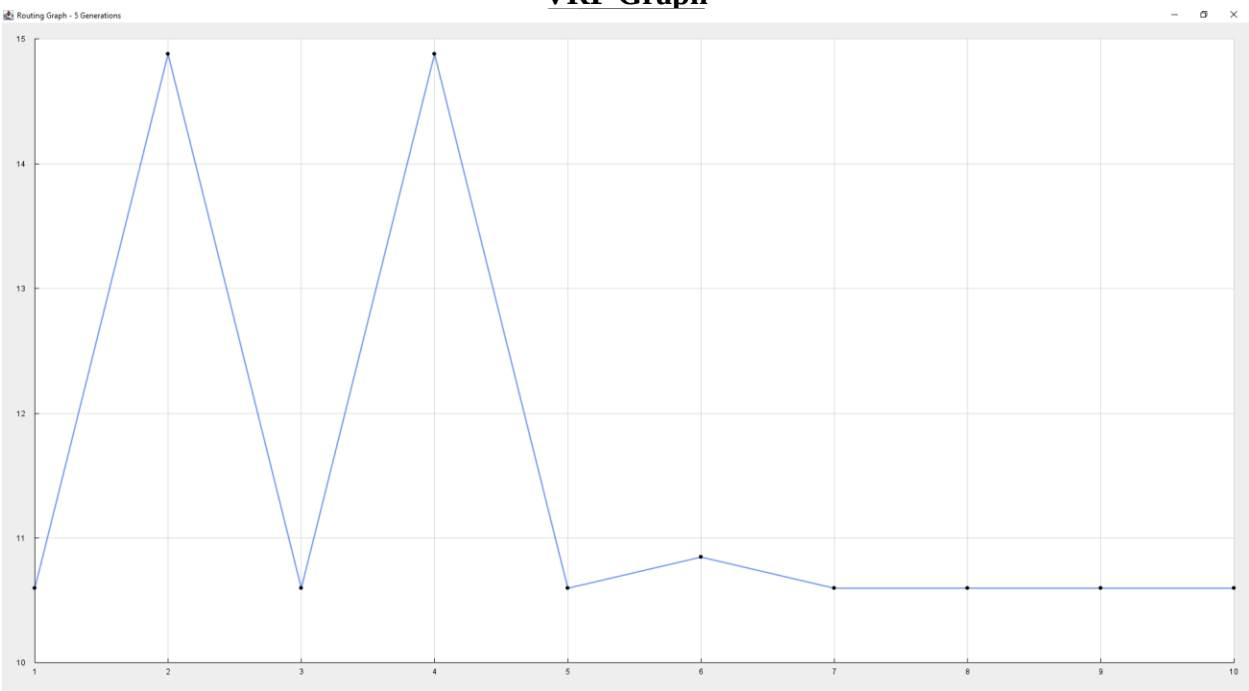
etc…..

Test no.1

PSP Graph

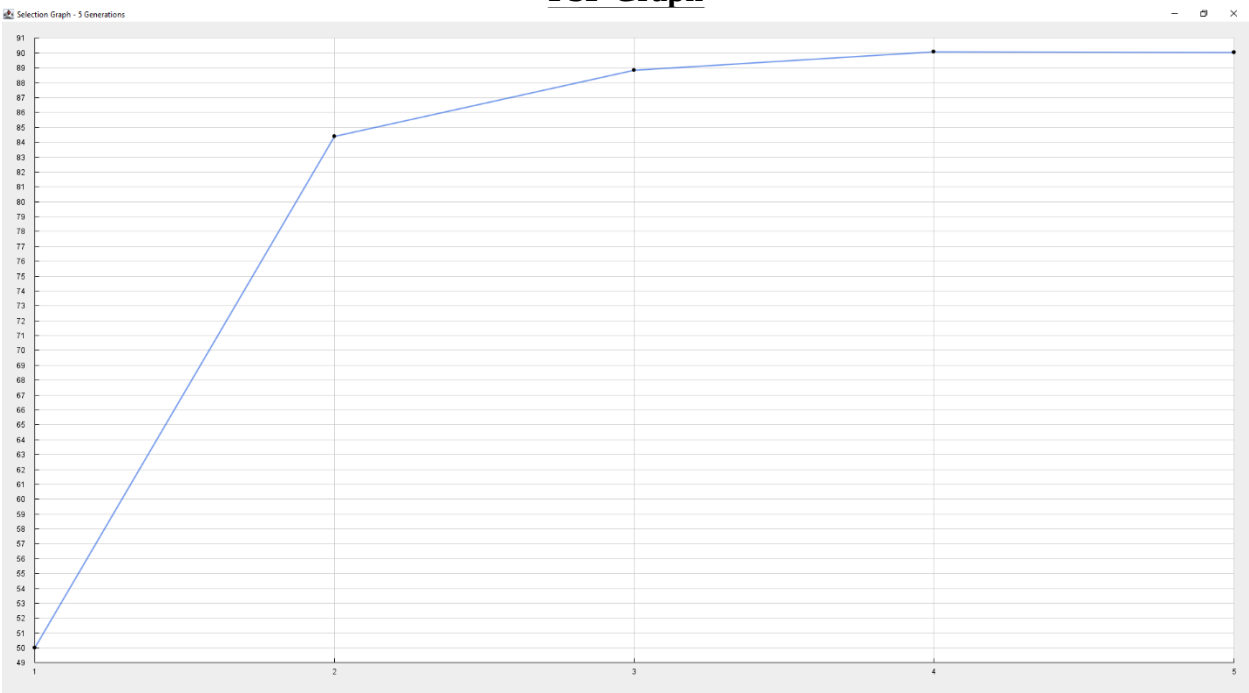


VRP Graph

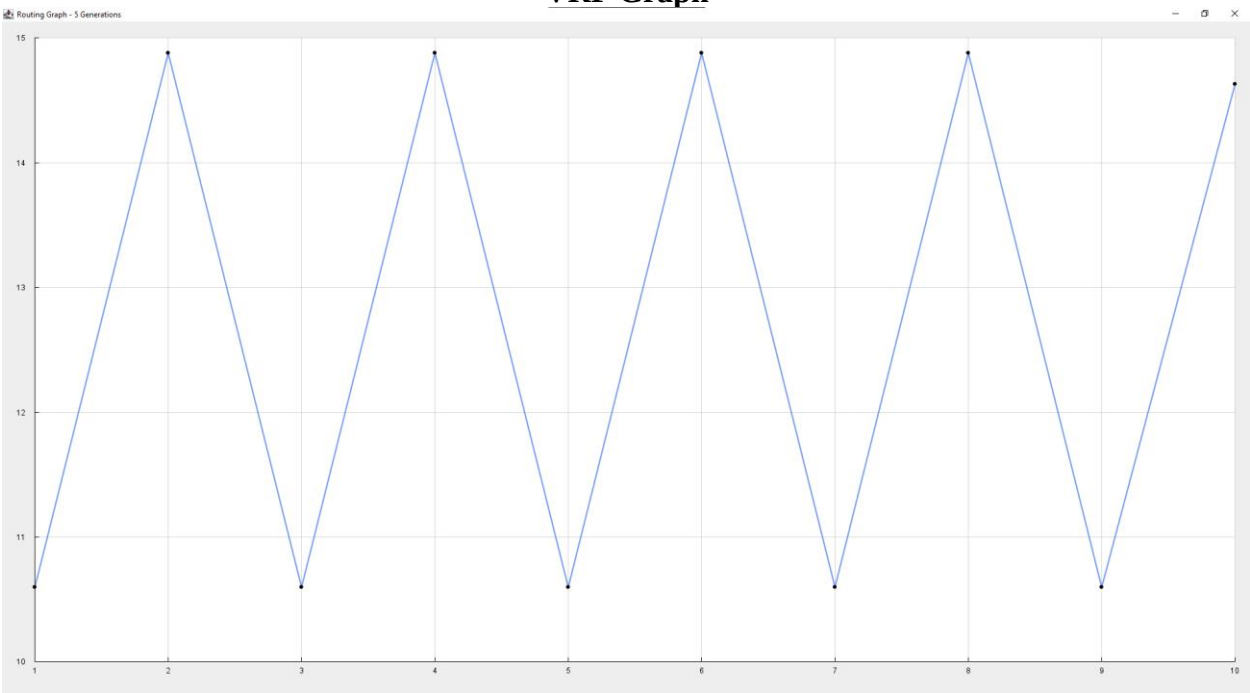


Test no.2

PSP Graph

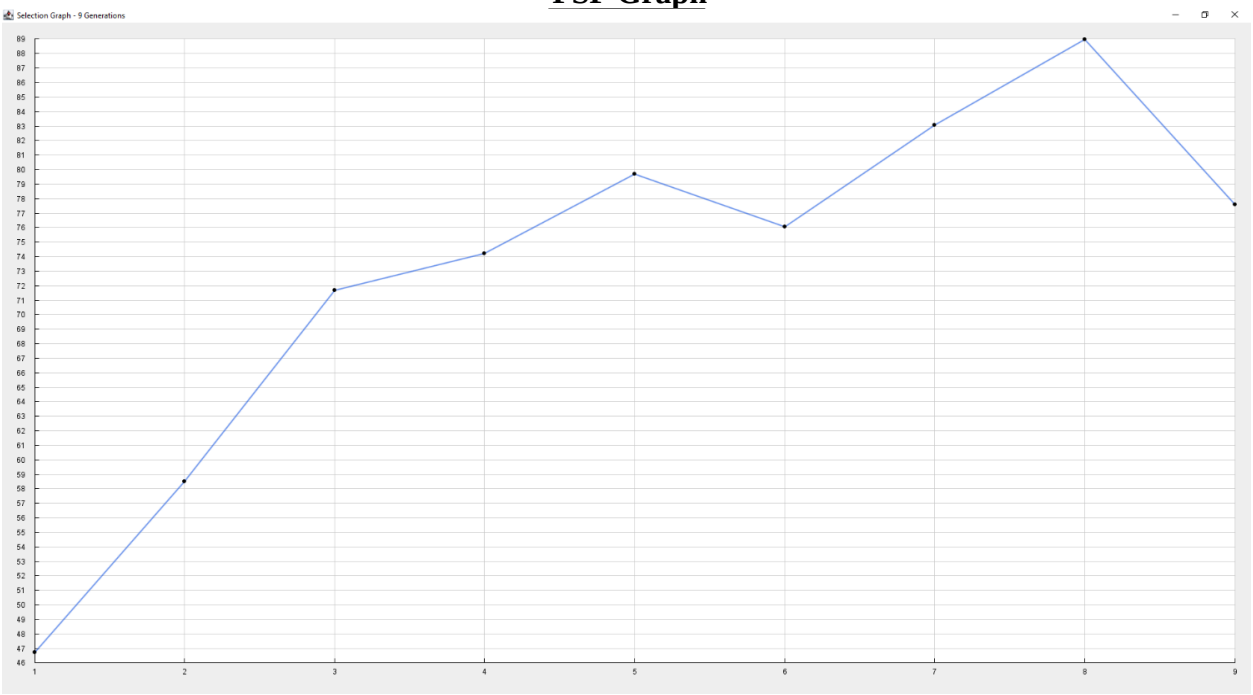


VRP Graph

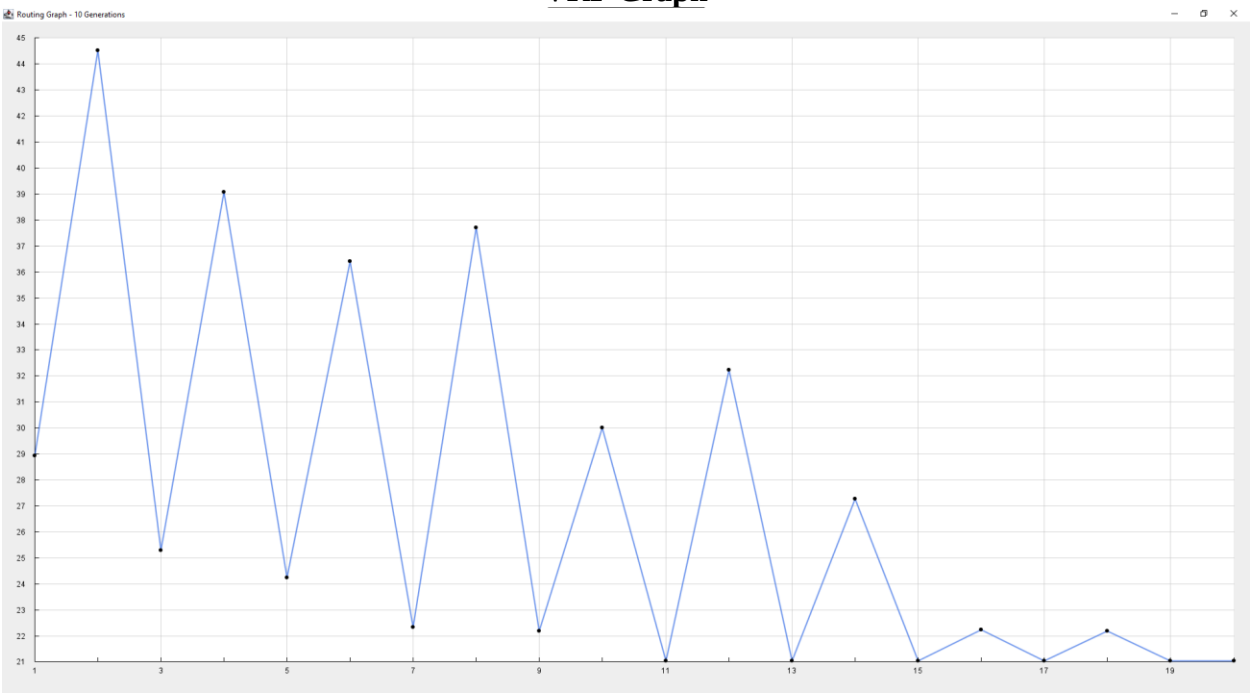


Test no.3

PSP Graph



VRP Graph

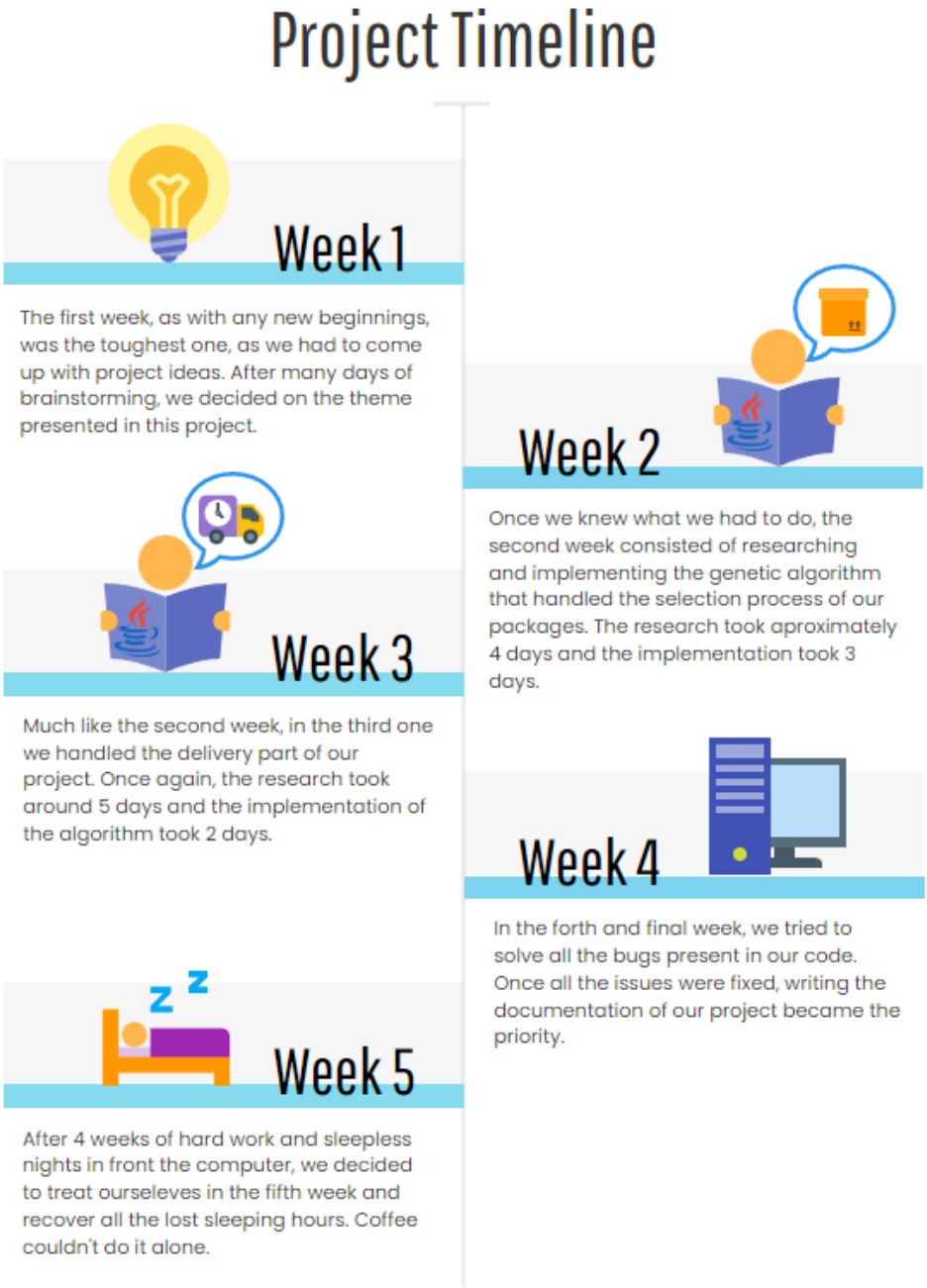


Conclusions and Possible Updates

After each genetic algorithm has finished its execution, we will be granted with two solutions, one telling us the packages that will be fitted inside our delivery truck and the second one showing the shortest route possible to deliver all our cargo. The merge of these solutions will give us the answer to the question asked in the beginning, meaning we successfully found a way to maximize profit and minimize transportation costs.

This isn't a perfect solution however, but rather a generalization on an ideal case. There are many ways in which our idea can be improved and get closer to a real-life scenario, as we didn't take many variables into consideration in an attempt to simplify the problem.

The ways this project can be improved will be discussed in the "Possible Updates" section of this chapter, but first let's look at the timeline regarding the evolution of our algorithm. This should shine a light on how the problem was approached and the way we decided to split the tasks in order to obtain a finished final product.



- Week 1 → 25.11.2021 – 02.12.2021
- Week 2 → 02.12.2021 – 09.12.2021
- Week 3 → 09.12.2021 – 16.12.2021
- Week 4 → 16.12.2021 – 23.12.2021
- Week 5 → 23.12.2021 – 30.12.2021

Possible Updates – Package Selection Problem

While the PSP part of our projects tackles the question of an object's shape impeding it from fitting into our delivery truck, we don't know if there is a configuration our packages need to be placed in order for all of them to fit. This is an important aspect that needs to be treated because while all the packages can fit on their own, we may not be able to stack them all together.

The way we could fix this is by creating an algorithm that given a set of boxes with their respective sizes, by trial and error finds a way to arrange them in such a way that they fit. In the event this can't be done, the least valuable package should be discarded.

The reason we decided to ignore this step is because the package arrangement process comes with a series of issues on their own. A few of them being by what criteria do we arrange them. This is important because now destination would play a role in their position in the car. We can't place the packages that will be delivered first in the back or vice versa, we need to arrange them in such a way that they will be ordered by their destination so the courier will have easy access to them. This once again generates new problems, as the route is calculated on the final solution's destinations, but to arrange the packages to get the final solution we will need the route.

One possible way we could solve this problem is by rerunning both algorithms each time we can't find a good arrangement, but this would compromise the execution time which we tried to improve.

Possible Updates – Vehicle Routing Problem

Just as in the PSP, in order for our algorithm to work, we had to simply the main problem. In this case, we considered that there is a direct route from any two cities, meaning we can go everywhere from anywhere. This is not how it works in real life, as getting to a certain point could mean going through multiple secondary locations that are not part of our destination pool.

A possible solution that would improve the accuracy of our route-finding algorithm would be to create a weighted graph, that would simulate the roads between each city, an edge between two nodes has the cost equaled to the real-life distance. This will, obviously, create more issues than it would solve. The biggest hit is taken by the gene generation and crossover processes, as now not only we would need to pay attention to the new node we are introducing into the solution (see if it is connected to the previous one or not), but we would also have to find a new way to merge the routes keeping the connections into consideration. The last problem we would have to solve is, in the event that a solution is somehow achieved, what do we do if the destinations are not included. Do we reinitiate the population, do we discard the certain individual? This could lead to a generation with no individuals and as a result crash the algorithm.

These are questions that we have not yet found the answer to, but we have a solid starting point from which we can improve in order to get close to a real-life scenario. It is important to mention that since the theme of this project are Genetic Algorithms, this added some limitations to the way we solve problems, as other approaches could’ve fitted better for our examples.

In the end, what we managed to achieve is a simplified solution to the problem of selecting and choosing an order to transport packages to their destinations. As a result, on certain cases, our algorithm can provide accurate and useful results if used in a real-life scenario.

References

Introduction to Genetic Algorithms, 2017.
Source: towardsdatascience.com
Accessed at: 26th November 2021

Introduction to Optimization with Genetic Algorithm, 2018.
Source: towardsdatascience.com
Accessed at: 26th November 2021

Solving the Knapsack Problem with a Simple Genetic Algorithm, 2017.
Source: www.dataminingapps.com
Accessed at: 27th November 2021

B. L. Miller and D. E. Goldberg, Genetic Algorithms, Tournament Selection, and the Effects of Noise, 1995.
Source: wpmedia.wolfram.com
Accessed at: 27th November 2021

A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri and V. B. S. Prasath, Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach, 2019.
Source: www.mdpi.com
Accessed at: 28th November 2021

V. D. Wester, A Genetic Algorithm for the Vehicle Routing Problem, 1993.
Source: trace.tennessee.edu
Accessed at: 29th November 2021

B. M. Baker, A genetic algorithm for the vehicle routing problem, 2003.
Source: www.sciencedirect.com
Accessed at: 29th November 2021

H. Kurnia, E. G. Wahyuni, E. C. Pembrani, S. T. Gardini and S. K. Aditya, Vehicle Routing Problem Using Genetic Algorithm with Multi Compartment on Vegetable Distribution, 2018.
Source: iopscience.iop.org
Accessed at: 30th November 2021

A. Hussain, Y. S. Muhammad, M. N. Sajid, I. Hussain, A. M. Shoukry and S. Gani, Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator, 2017.
Source: www.hindawi.com
Accessed at: 1st December 2021

Evolution of a salesman: A complete genetic algorithm tutorial for Python, 2018.
Source: towardsdatascience.com
Accessed at: 1st December 2021

J.-Y. Potvin, Genetic algorithms for the traveling salesman problem, 1996.
Source: www.inf.tu-dresden.de
Accessed at: 2nd December 2021

Sushant Shekhar Burnawal, Multi-Objective Vehicle Route Optimization, 2020.
Source: towardsdatascience.com
Accessed at: 8th January 2022

Kie Codes, Genetic Algorithms Explained By Example, 2020.
Source: www.youtube.com
Accessed at: 8th January 2022