Graph coloring problem

Xavier Querol, Alba Fernández

1 Introduction to the Problem

The graph coloring problem is a well-known challenge in combinatorial optimization, where the aim is to color the nodes of a graph such that no two adjacent nodes share the same color. Solving this problem with the fewest possible colors is of particular interest due to its practical applications in areas such as resource allocation, scheduling, and map coloring.

To approach this challenge, we use Genetic Algorithms (GAs), which are inspired by the process of natural evolution. GAs evolve a population of candidate solutions through operations like selection, crossover, and mutation, iteratively refining these solutions based on a fitness function. This allows them to handle complex, multi-dimensional problems effectively, often finding near-optimal solutions where exact algorithms may be impractical.

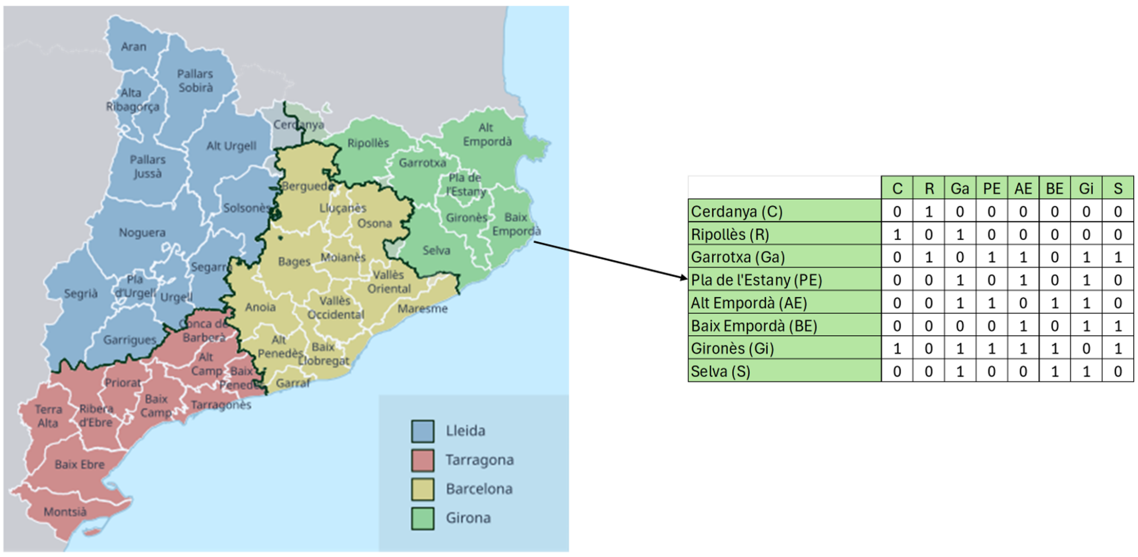
In our specific case, we aim to apply a genetic algorithm to minimize the number of colors needed to color the map of Catalonia, ensuring that no two adjacent regions share the same color. We conduct this experiment on two levels of granularity: first at the county level *(comarca-comarques)*, intermediate geographic divisions that group municipalities for administrative purposes, that will be the main focus of this analysis, and then at the municipality level *(municipi-municipis)* —smaller, local administrative units- treating each as a separate case of the graph coloring problem. Through this approach, we explore the effectiveness of GAs in addressing a real-world problem of map coloring with minimal colors, while considering the unique structure of Catalonia's geographical divisions.

2 Methodology

In this section, we present the approach used to solve the graph coloring problem through Genetic Algorithms (GAs).

Graphs are composed of nodes and edges. In our case, each node represents a polygon in the map, and edges are formed by adjacencies between polygons. The graph coloring problem is addressed by utilizing the adjacency matrix, which captures these relationships. Figure 1 shows the adjacency matrix for the *Província de Girona*, considering only the *comarques* within this province.

With this structure in place, we proceed by defining the representation of candidate solutions, where each solution corresponds to a potential coloring configuration that assigns colors to each region while minimizing conflicts.



	C	R	Ga	PE	AE	BE	Gi	S
Cerdanya (C)	0	1	0	0	0	0	0	0
Ripollès (R)	1	0	1	0	0	0	0	0
Garrotxa (Ga)	0	1	0	1	1	0	1	1
Pla de l'Estany (PE)	0	0	1	0	1	0	1	0
Alt Empordà (AE)	0	0	1	1	0	1	1	0
Baix Empordà (BE)	0	0	0	0	1	0	1	1
Gironès (Gi)	1	0	1	1	1	1	0	1
Selva (S)	0	0	1	0	0	1	1	0

Figure 1: Map of Catalonia with Girona's adjacency matrix.

2.1 Candidate solutions

In the graph coloring problem, each candidate solution —referred to as an individual in the Genetic Algorithm (GA)— is represented by a list, where each position corresponds to a node in the graph and the value at that position represents the color assigned to that node. In the specific case of Catalonia, each position in this list represents a *Comarca* (county), resulting in a list length of 43, as there are 43 *comarques* in total.

For instance, a candidate solution for the simplified example in Figure 1 could be represented as: [0, 1, 2, 2, 1, 0, 1, 2]. This list corresponds to the following color assignment: [Cerdanya: 0, Ripollès: 1, Garrotxa: 2, Pla de l'Estany: 2, Alt Empordà: 1, Baix Empordà: 0, Gironès: 1, Selva: 2]. In this encoding, each color (represented by an integer) is assigned to a *comarca*, with the objective of minimizing color conflicts between adjacent *comarques* [1].

This list-based encoding allows the GA to efficiently explore possible color assignments across Catalonia's regions while optimizing for minimal conflicts according to the adjacency matrix constraints.

In genetic algorithms, the order of genes can significantly influence the results, as different arrangements may impact the algorithm's performance and solution quality. To explore this effect, we consider two sorting approaches for the *Comarques*:

1. **Alphabetical Order**: This ordering arranges *Comarques* based purely on lexicographical sequence, without considering geographical or contextual proximity.
2. **Geographical Sorting**: This custom arrangement groups adjacent *Comarques* that are geographically or contextually close.

By analyzing these two sorting strategies, we aim to understand how the ordering of Comarques influences genetic algorithm outcomes and whether proximity-based ordering offers any performance advantages.

2.2 Fitness function

To evaluate how close each individual is to a valid solution, a fitness function is designed to penalize color conflicts between adjacent nodes. Specifically, this function calculates the number of adjacent nodes (representing *comarques*) that share the same color, with the goal of minimizing these conflicts.

Let M be the adjacency matrix, where $M_{ij} = 1$ if nodes i and j are adjacent, and $M_{ij} = 0$ otherwise. Let x_i represent the color assigned to the i-th node in an individual solution. The penalty for each individual solution, indicating the number of conflicts, is defined as:

$$\text{penalty}(i) = \frac{1}{2} \sum_{j=1}^{n} M_{ij} \cdot \delta(x_i, x_j)$$

where:

$$\delta(x_i, x_j) = \begin{cases} 1 & \text{if } x_i = x_j \\ 0 & \text{otherwise} \end{cases}$$

The indicator function $\delta(x_i, x_j)$ is used to check if adjacent nodes i and j are assigned the same color. If $x_i = x_j$, a conflict is counted. The factor of $\frac{1}{2}$ is applied to avoid double-counting each conflict, as each pair of adjacent nodes is considered twice in the summation.

In our implementation, the fitness function calculates the penalty for each individual in the population by multiplying the adjacency matrix M with a comparison of colors between nodes. The Genetic Algorithm aims to minimize this penalty, encouraging solutions with fewer color conflicts between adjacent nodes. An individual with a penalty of zero would represent a valid solution to the graph coloring problem, as no adjacent nodes would share the same color.

2.3 Selection mechanism

In genetic algorithms, selection is the process by which individuals are chosen from the current population to reproduce and form the next generation. It is a crucial step, as it determines which solutions from the current population will have the opportunity to reproduce and contribute their genetic material to future generations. The goal is to select individuals that have demonstrated better performance, meaning those with lower penalties, so their characteristics can be passed on and potentially improve the overall performance of the algorithm.

In this work, selection is performed using a *tournament selection* mechanism. This approach involves pairing individuals from the current population, who then compete in a small tournament. The winner of each match is the individual with the lower penalty, indicating better performance in the context of the graph coloring problem. This process is repeated multiple times to select a set of individuals who will be chosen to reproduce and become the parents of the next generation.

2.4 Crossover operators

Once the individuals have been selected for reproduction, the next step is to create the offspring. This is achieved through the process of crossover, where two parent solutions combine to generate new child solutions. Crossover can be performed in various ways, and in our approach, we test three types: One-Point Crossover, Two-Point Crossover, and Uniform Crossover.

Let P_1 and P_2 be the parent solutions (arrays) of length n, and let C_1 and C_2 be the resulting child solutions.

Depending on the chosen crossover type, the children are generated using the selected method:

$$C_1, C_2 = \text{crossover}(\textit{crossover\_type}, P_1, P_2)$$

where:

$$\textit{crossover\_type} \in \{\text{uniform}, \text{one-point}, \text{two-point}\}$$

2.4.1 One-Point Crossover

One-Point Crossover is a simple yet effective method in which a random point is selected in the parent solutions, and the offspring are generated by combining parts from each parent.

1. **Random Point:** A random point is chosen within the range $2 \leq \textit{point} \leq n-2$, ensuring that the point is not at the extremes (beginning or end) of the solutions. The point is selected using a uniform distribution:

$$\textit{point} \sim \text{Uniform}(2, n-2)$$

2. **Children Generation:** Once the random point is selected, the two offspring are created by combining the first part of one parent with the second part of the other parent. The first child (C_1) is generated by taking the portion of P_1 before the point and the portion of P_2 after the point. The second child (C_2) is created by taking the portion of P_2 before the point and the portion of P_1 after the point:

$$C_1 = \big[P_1[0], P_1[1], \ldots, P_1[\textit{point}], P_2[\textit{point}+1], \ldots, P_2[n-1]\big]$$

$$C_2 = \big[P_2[0], P_2[1], \ldots, P_2[\textit{point}], P_1[\textit{point}+1], \ldots, P_1[n-1]\big]$$

   This method allows for the exchange of genetic material between the two parents while maintaining a degree of structure from both solutions.

2.4.2 Two-Point Crossover

Two-Point Crossover is a variation of the One-Point Crossover, where two random points are selected within the parent solutions. The offspring are then generated by swapping the segments between these two points. This method introduces more diversity than One-Point Crossover by allowing more significant portions of the parent solutions to be exchanged.

1. **Random Points:** Two random points, $\textit{point}1$ and $\textit{point}2$, are chosen from the range $\{1, \ldots, n-1\}$. These points are sorted such that $\textit{point}1 < \textit{point}2$. The points are selected using the following method:

$$\textit{point}1, \textit{point}2 \sim \text{Sorted}(\text{Sample}(\{1, \ldots, n-1\}, 2))$$

   This ensures that both points are within the bounds of the parent solutions, avoiding positions at the extremes.

2. **Children Generation:** After selecting the two points, the offspring are generated by exchanging the segments between these two points. The first child (C_1) is created by taking the first segment from P_1 up to $\textit{point}1$, followed by the middle segment from P_2

between *point*1 and *point*2, and then the final segment from P_1 after *point*2. The second child (C_2) is created similarly but with the segments from P_2 and P_1 swapped:

$$C_1 = \big[P_1[0], \ldots, P_1[point1-1], P_2[point1], \ldots, P_2[point2-1], P_1[point2], \ldots, P_1[n-1]\big]$$

$$C_2 = \big[P_2[0], \ldots, P_2[point1-1], P_1[point1], \ldots, P_1[point2-1], P_2[point2], \ldots, P_2[n-1]\big]$$

This method allows for a greater degree of mixing between the two parents, increasing the potential for discovering new, effective solutions.

2.4.3 Uniform Crossover

Uniform Crossover differs from the One-Point and Two-Point Crossover methods by selecting the genes for the offspring based on a probability distribution, rather than by using specific points in the parent solutions. This method allows for a more flexible recombination, where each gene in the offspring can come from either parent with a certain probability.

1. **Crossover Probability:** Let p be the crossover probability, which determines the likelihood that a gene from the first parent (P_1) will be inherited by the first child (C_1). The second child (C_2) inherits the complementary genes from P_2 with the same probability distribution.

2. **Children Generation:** The offspring are generated by randomly selecting each gene from either parent based on the crossover probability p. For each gene position i (where $i = 0, 1, \ldots, n-1$), the genes for the children are assigned as follows:

$$C_1[i] = \begin{cases} P_1[i] & \text{with probability } p \\ P_2[i] & \text{with probability } 1-p \end{cases}$$

$$C_2[i] = \begin{cases} P_2[i] & \text{with probability } p \\ P_1[i] & \text{with probability } 1-p \end{cases}$$

   This process is applied independently to each gene position in the parent solutions, leading to more varied offspring combinations. The result is that each child can inherit genes from both parents in a highly flexible manner, depending on the crossover probability p. We have defined this probability at 0.5.

2.5 Mutation Operators

Once the new population (offspring) has been created through crossover, the next step is to introduce genetic diversity by applying mutation operators. Mutation ensures that the population does not converge prematurely to a local optimum and that new areas of the solution space are explored. In the context of the graph coloring problem, mutation involves randomly altering the colors assigned to specific vertices in the individual solutions.

Let individuals be a set of arrays where each array represents a solution, with each element in an array denoting the color assigned to a vertex. Here, p_{mutation} is the mutation probability, and k represents the total number of colors available for the coloring.

1. **Iterate Through Each Individual and Position:** For each individual array in the population and each position i within an individual, check if a mutation will occur at position i with probability p_{mutation}.

2. **Apply Mutation:** If mutation occurs (i.e., if $U(0,1) \leq p_{\text{mutation}}$), assign a new random color to that position, chosen uniformly from the range $\{0, 1, \ldots, k-1\}$:

$$\text{individual}[i] = \begin{cases} \text{Uniform}(0, k-1) & \text{if } U(0,1) \leq p_{\text{mutation}} \\ \text{individual}[i] & \text{otherwise} \end{cases}$$

   where:

   - $U(0,1)$ is a random variable uniformly distributed between 0 and 1, used to decide if mutation occurs at a given position.
   - $\text{Uniform}(0, k-1)$ selects a new color from $\{0, 1, \ldots, k-1\}$, which replaces the original color at position i if mutation occurs.

This mutation process is repeated for each individual in the population, ensuring that the mutation is applied independently to each position with probability p_{mutation}.

2.6 Stopping Criteria

The stopping criteria is essential to determine when the algorithm should terminate. In our approach, the algorithm is composed of two loops. The inner loop continues iterating until either a valid solution is found or a predefined maximum number of generations is reached. The outer loop continues iterating until the maximum generations is reached in the inner loop. So, the stopping criteria are as follows:

1. **Valid Solution Found:** The algorithm aims to find a valid coloring of the graph, meaning that the penalty (the number of conflicting color assignments) for the best individual in the population must reach zero. This indicates that the coloring solution is valid, and no two adjacent nodes share the same color. Once this condition is met, the algorithm attempts to decrement the number of colors by one and searches for a new valid solution. If a valid solution with fewer colors is found, the process repeats until no valid solution can be found with a smaller color set.

2. **Maximum Generations Reached:** The algorithm is also constrained by a maximum number of generations, denoted as *max_generations*. If the number of generations exceeds this limit without finding a valid solution, the algorithm terminates. This ensures that the search does not continue indefinitely, even if a solution is not found in the current generation cycle.

The stopping criteria balance exploration and exploitation by allowing the algorithm to search for optimal solutions while ensuring that it does not run indefinitely. This approach helps in efficiently reaching a solution to the graph coloring problem, while also allowing for an adequate amount of exploration and solution refinement.

2.7 Algorithm

Algorithm 1 Genetic Algorithm for Graph Coloring

```
1: Initialize num_colors, population_size, max_generations, fitness_evaluations,
   total_generations, and results.
2: while num_colors > 0 do
3:   Step 1: Initialize Population
4:   population ← Random integers in range $[0, \text{num\_colors})$ of shape (population_size,
     nodes_graph)
5:   generations ← 0
6:   Step 2: Evaluate Initial Population
7:   penalty ← fitness_function(matrix, population)
8:   Step 3: Evolve Population
9:   while min(penalty) > 0 and generations < max_generations do
10:    individuals_to_reproduce ← tournament_selection(population, matrix)
11:    new_population ← crossover(individuals_to_reproduce, crossover_type, crossover_prob)
12:    new_population ← mutation(new_population, prob_mutation, num_colors)
13:    penalty ← fitness_function(matrix, new_population)
14:    fitness_evaluations ← fitness_evaluations + length of penalty
15:    population ← new_population
16:    generations ← generations + 1
17:  end while
18:  Step 4: Check for a Valid Solution
19:  if min(penalty) = 0 then
20:    num_colors ← num_colors − 1
21:    best_palette ← population[arg min(penalty)]
22:    Print information about the best solution
23:  else
24:    Break {No valid solution with fewer colors}
25:  end if
26: end while
```

3 Results

To assess the performance of our genetic algorithm under various parameter configurations, we tested combinations of the following parameters:

- Population size: [100, 200, 300, 500, 800, 1000]
- Mutation probability: [0.05, 0.1, 0.15, 0.2]
- Maximum number of generations: [200, 500, 800]
- Crossover type: [‘one-point’, ‘two-point’, ‘uniform’]

For each parameter combination, we conducted 10 independent runs and calculated the average number of colors required to solve the graph coloring problem. The configuration with the lowest average number of colors was selected as the optimal parameter set, effectively minimizing the number of colors used while maintaining solution validity.

The optimal parameter configuration obtained from these tests was:

- Population size: 500
- Mutation probability: 0.05
- Maximum number of generations: 500
- Crossover type: *two-point*

Table 1 summarizes the mean values of key performance metrics across 10 iterations using the optimal configuration.

Metric	**Mean Value**
Num Colors	4.3
Execution Time (s)	8.32
Generations	209.3
Fitness Function Calls	104650.0

Table 1: Mean values for each performance metric across 10 iterations.

The minimum number of colors required to color the map of Catalonia by *comarques* is 4. In 7 out of 10 runs, the algorithm with the optimal parameters successfully achieved this minimum, while in the remaining 3 runs, it required 5 colors to produce a valid solution. This result suggests that, although the algorithm can reach the desired performance, it does so inconsistently across iterations. Moreover, despite this being the best parameter combination, there is a significant random component due to the heavy dependence on the initial population's configuration.

Given a 70% success rate per run and independent trials, the probability of achieving at least one correct result in 10 runs is:

$$1 - (0.30)^{10} \approx 0.999994$$

Figure 2 shows an example of the map of Catalonia colored using the optimal parameter configuration, achieving the minimum number of colors.

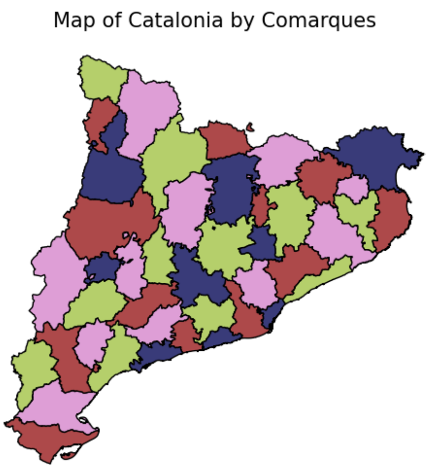


Figure 2: Map of Catalonia colored by Comarques.

Figure 3 shows the evolution of the mean penalty and the penalty corresponding to the lowest 20% of the population across generations in a run of the algorithm using the best parameters found. It can be observed that each drop in penalty corresponds to the reduction of one color needed for the map, eventually reaching the minimum of 4 colors. Once further reduction is no longer possible, the algorithm converges, stabilizing around values that indicate an optimal and valid color assignment.

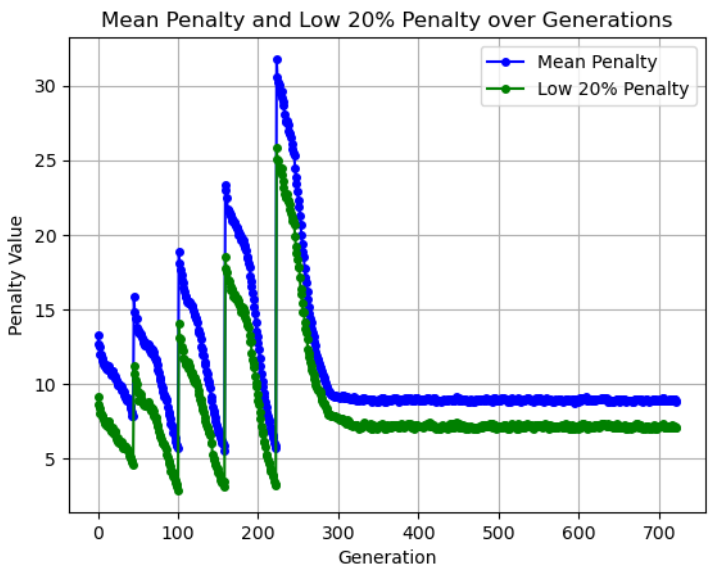


Figure 3: Convergence of Mean and Lowest 20% Penalty to the 4-color solution for Catalonia with optimal parameters.

On the other hand, to determine if the sorting method has an effect on the Mean Number of Colors in our genetic algorithm experiment, we performed a paired t-test. This statistical test compares the Mean Number of Colors between the two methods with identical hyperparameters to see if there is a significant difference between them.

The p-value we obtained from this test is 0.30. Since our p-value is greater than a predefined alpha of 0.05, we fail to reject the null hypothesis. This means that, statistically, there is no significant evidence to suggest that the sorting method affects the Mean Number of Colors. In Figure 4 it is reflected the difference between sortings and the t-test result, as the difference it builds a Normal distribution.

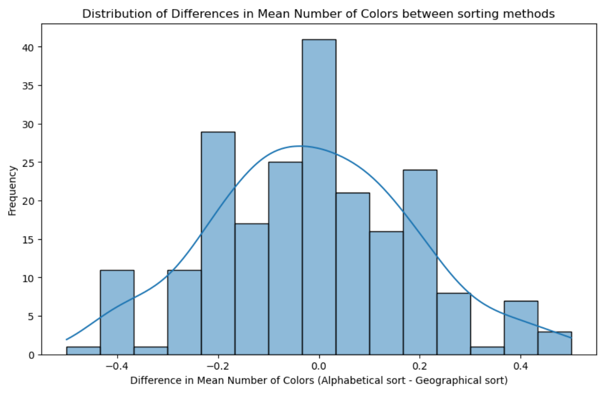


Figure 4: Distribution of differences in mean color count between alphabetical and geographical sorting methods.

The same genetic algorithm can be applied in another graph, i.e in Catalunya *municipis*. In Figure 5 we show the coloring using the 10 colors found by the algorithm.

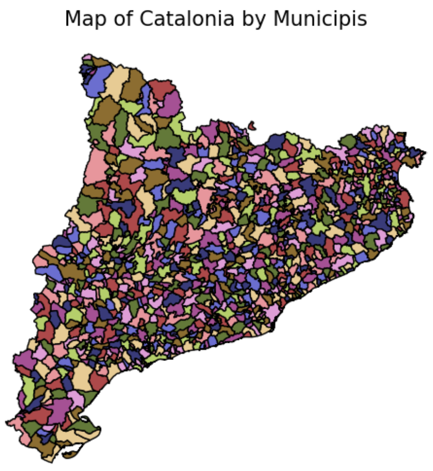


Figure 5: Map of Catalonia colored by Municipis.

4 Conclusions

In this study, we applied a genetic algorithm (GA) to determine the optimal number of colors for coloring the map of Catalonia by *comarques*. Our findings indicate that the GA does not consistently guarantee an optimal solution, as it sometimes fails to identify the minimum number of colors required. Therefore, the algorithm cannot be considered fully optimal since it does not always ensure minimal coloring.

Moreover, finding an optimal solution depends on several stochastic factors, including the initialization of the population, mutation rates, and crossover events, all of which introduce variability that can impact the quality of the results.

With a success probability of 99.9994%, running the algorithm 10 times virtually guarantees at least one correct result. This high probability shows that even with a 30% chance of failure per run, repeated attempts greatly increase the likelihood of success.

Finally, we observed that ordering the *comarques* either alphabetically or geographically did not lead to any improvement in the algorithm's performance. This suggests that the choice of initial ordering does not influence the solution and does not contribute to enhancing the search for optimal coloring.

References

[1] OpenAI. (2024). Response to the prompt "create me an adjacency list for comarques using this dataframe with geographical information" (October 10 version) [GPT-4o mini].

[2] OpenAI. (2024). Response to the prompt "rewrite and syntesise this information" (October 31 version) [GPT-4o mini].

[3] Institut Cartogràfic i Geològic de Catalunya. (n.d.). Mapes de comarques [Maps of regions]. Retrieved October 17, 2024, from https://www.icgc.cat/ca/Ambits-tematics/Recursos-didactics/Mapes-de-comarques

[4] GeeksforGeeks. (2021). Project idea: Genetic algorithms for graph colouring. Retrieved October 2, 2024, from https://www.geeksforgeeks.org/project-idea-genetic-algorithms-for-graph-colouring/