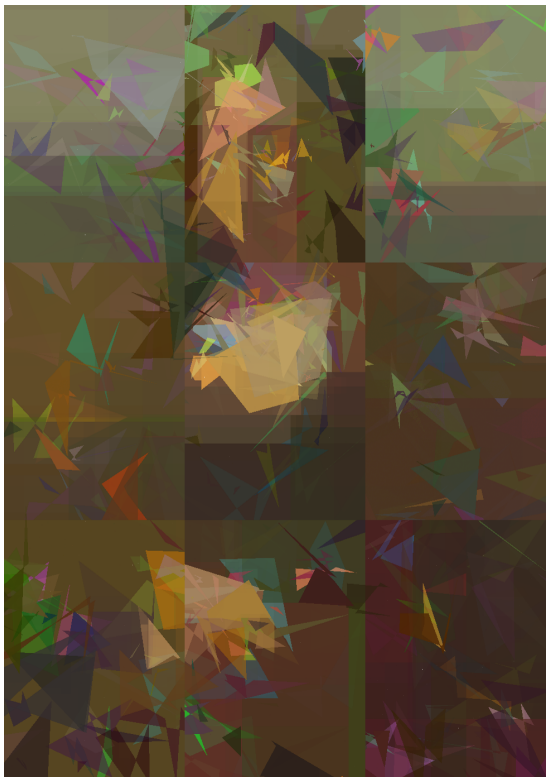
Evolutionary algorithms

Principles, current research status, applications, libraries and experiments

García Fernández, Helios



WSZiB, 2023

1. Introduction - basic ideas

Describe the basic concepts and ideas related to evolutionary algorithms.

The example of citing bibliography [1]. The second example [2].

In this Introduction I would like to lay some groundwork about what is Evolutionary Algorithms and how you can place it in Artificial Intelligence, knowing that artificial intelligence is a vast study in which we try to take advantage of the much knowledge we have about various fields such as biology and integrate it into computing to take advantage of it.

Evolutionary Algorithms (EAs) are a subset of "Evolutionary Computation", a generic population-based metaheuristic optimization algorithm in the field of "Computational Intelligence". Computational Intelligence (CI) is the overarching field that encompasses various techniques for solving complex real-word problems to which mathematical or traditional modelling can be useless due to its partially or completely uncertain nature. Therefore, we refer to CI as a subset of AI where we don't use Crisp logic systems consisting of either including an element in a set, or not (as Hard Computing techniques work, just using a set defined by 0 or 1), whereas Fuzzy logic systems enable elements to be partially in a set (which closer to the way the human brain works by aggregating data to partial truths).

In that way, Evolutionary Computation (EC) is a subfield of CI which approach is to solve global optimization problems using methods inspired by biological conditions. In technical terms, they are a family of population-based trial and error problem solvers with a meta heuristic or stochastic optimization character. An Initial set of candidate solutions is generated and iteratively updated. This field encapsulates a broader of optimization and search techniques, some of which may not strictly adhere to the traditional evolutionary principles, p.e Particle Swarm Optimization (PSO) use the social behaviour of a swarm of particles, the population created at the beginning, to guide the search for the optimal solution to a problem updating the positions of the swarm on each iteration. However, within the range of possibilities that we can find in EC, we will place our sights on the "Evolutionary Algorithms".

Evolutionary Algorithms, in contrast to PSO, ONLY involve techniques implementing mechanisms motivated by the metaphor of natural selection biological evolution, in a similar way Darwin describes the Theory of evolution. In this case, there is a population of individuals (states), in which the fittest (highest value) individuals, produce offspring (successor states) that populate the next generation, a process called recombination (crossover). At the same time, we will find an important step on this process: the mutation; it let the state to have a random "mutation" of there conditions once an offspring has been generated (we will get deeper on the second point about it).

We can find endless forms of evolutionary algorithms depending on: The size of the population, the representation of each individual, the number of individual mixing on the offspring, the selection, the crossover point, the mutation rate and the makeup of the next generation.

2. Evolutionary algorithm operation principle

Describe the principle of operation of the basic version of evolutionary algorithm.

Continuing with what was previously mentioned, we could divide the following 3 fundamental parts of evolutionary algorithms:

1. **Initialization**

   In this step we need to define to principles parameters:

   - Population: First of all, we will need to initialize our "population" of possible solutions to our problem. This population will be made up of a set of randomly generated candidates which will be represented in various ways depending on the structure of the problem.

   - Fitness: The fitness defines the suitability of the candidate. It will decide how the population will evolve depending on how good each of the candidates is based on said value.

2. **Iteration**

   - *Selection*: Now we have our firsts candidates that we would have to select them depending on their fitness: Individuals with higher fitness have a higher change of being **selected** and to take part on the next population imitating the process of natural selection. We need to keep in mind that our population might not decrease, knowing that maybe we will reuse some “parents” more than one time.

   - *Crossover*: After have the selection of our highest candidates, we need to **crossover** there parameters in a probabilistic way to form the offsprings. To do such recombination, a p number of individuals are normally used for mixing. Said p is usually 2, in such a way that "two individuals reproduce and form a child from their 2 parents witch there corresponding random mixture of gens“

   - *Mutation*: In that point, we will have our new population settled. However, to give the chance to don’t get stuck in an optimal local search, there is a random **mutation** of the values of each individual defining a mutation rate which determines how often offspring have random mutations to their representation.

As soon as we finished all these procedures, the next step is replace the last population with the new one recalculating the new fitness due to start again this process.

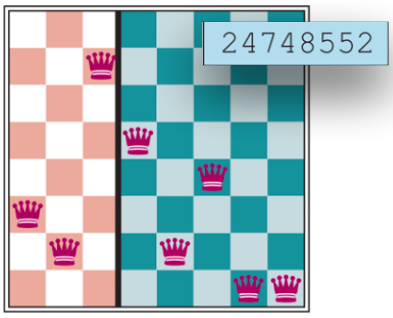
3. **Termination**

   Finally, we will finish this iterative search when we achieve a specific fitness in one of the individuals or establish a limit that allows us to give an approximate solution to the truly optimal solution.

In order to illustrate this procedure more clearly, we will make a small example of a "genetic algorithm" on the 8-queens problem.

Defining our population with a series of individuals called "chromosomes" which will have a series of "genes" as parameters like a string. In order to begin solving our problem we must correctly define our chromosome, its size, and the involvement of each gene. Therefore, we must set these parameters to the needs of the problem. In this case we are looking for a way to place 8 queens on an 8x8 board in such a way that none of them are attacking each other. To define the chromosome in the smallest way possible, we chose to give it a size n equal to the number of queens, where index i-th represents column c-th, so that the possible range of each gene would be between 1 and 8, representing the row in which said queen would be found in said column.

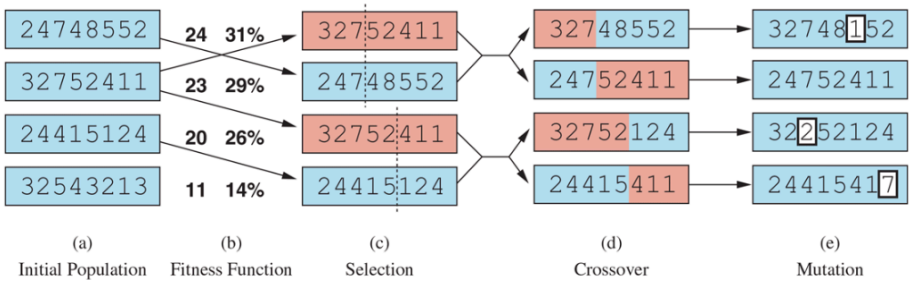
Looking at the next picture, we have an example of the representation of a specific chromosome:



What we would have left to finish the first step, Initialization, we must define our fitness and thus ask if it is a maximization or minimization problem. For the 8- queens problem we use the number of ***nonattacking*** pairs of queens, which has a value of

8 x 7 / 2 = 28 for a solution. Thus, as more fitness we will have in an individual it will be a better solution rather one who has less. This means that we are faced with a **maximization** problem.

We are about to finish the first step, in order to finish we will have to create a population based on what has already been defined. For our example we will use 4 randomly generated chromosomes (as we mentioned initially) and we will calculate their respective fitness.



Using the photo as a guide, we would already have steps a) and b) completed (where in b) the values are normalized by dividing the corresponding fitness by the total of the sum of all the fitness). In (c), two pairs of parents are selected, in accordance with the probabilities in (b) to make the corresponding crossover later (just taking a random subset of the corresponding pairs' genes to swap them). Notice that one individual is selected twice and one not at all because we only select a specific percentage of the best specimens. As we see on d), we have our new population, but we need to take the last step, where each location in each string is subject to random mutation with a small independent probability, as we can observe in e), where only 1 gen of the first, third and fourth chromosomes change.
Now we just have one of the whole amount of iterations that we will need to do to find the solution, so the next step will be to change the old population with our new population and there respective fitness and do the “Iteration” a specific number of times or until we find a fitness equal to 28, which would be the highest possible fitness, and if possible to solve (which it is) the most optimal solution.

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

It is a pseudo code as a reduction of the steps that we have been carrying out previously. To understand it a little more clearly, we must understand that "population" refers to the list of chromosomes, "fitness" refers to the list of the respective fitness as determined values and the function "*WEIGHTED-RANDOM-CHOICES(population, weights, 2)*" is what allows us to select the specific pairs from the selection in section c) to know which are the pair of parents that are going to crossover.

3. Current research status – review of selected current research works

Review selected scientific papers from recent years on research in the field of evolutionary algorithms.

After having seen different studies on evolutionary algorithms, I have been able to verify both a great variability of utilities, and different implementation of the algorithm. Because of this, I wanted to summarise a little some of the things I have been seeing. Since it is such an extensive field I could not deepen its full extent and I have preferred to give a more general view.
In this way, I would like to divide this point into 2 parts:

a)On the one hand I would like to expand certain things discussed in the introduction in a "general" way within the evolutionary algorithms and the theory of optimisation as a field in which said algorithms act to give a result as close to the solution as possible. It must be taken into account that after all the concept of "evolutionary algorithm" is more an abstraction as a model in which it is based on an iterative search with a series of reproduction operators and a certain degree of probability trying to imitate natural evolution. Therefore, being no more than a "skeleton" this algorithm can be seen implementing in very different ways depending on the problem we face. Because of this, sub-specifications have been created that allow you to get closer to a better solution of a smaller group of problems, so it can be a complicated task to select the right approach to each problem.

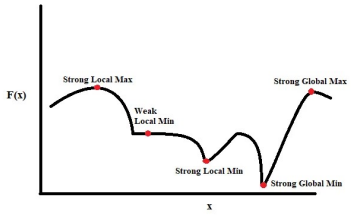
b) After that, I would like to continue with a small look at the five main variants of "evolutionary Algorithms". The reason for this point is to have found different documents of which they seem to speak of different classifications for the specific implementations of evolutionary algorithms. After having found several studies that talk about five general specifications, I thought it appropriate to comment on their main characteristics in a general way.

To do this, I have decided to be guided largely by one of the last publications I found, where Brandon Morgan explains in different topics the Evolutionary computing and the five main variants I commented on. This publication has been the most interesting to me since it allows you to give a general look at a field which I consider very extensive, being able to give you certain ideas for the application of them. In addition, it makes a division of the different main methods of EA along with a series of implementations that allow you to see a direct application of these algorithms. This publication is in turn quite broad, so I have preferred to explain the first topics in section a) to be able to give a greater explanation of the metaheristic in general, and then take a small look at the other topics in section b). Also comment that I use these publications as a guide, although not everything is from that publication. I have tried, as far as possible, to contrast different sources, so that they have allowed me to write and better understand the things I have been learning.

a) Evolutionary Algorithm and Optimisation Theory

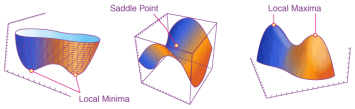
a.1) Critical Points / Optimas

As we discussed in the introduction, evolutionary computing algorithms are an application to solve optimisation problems. These are nothing more than problems in which we seek to find what are the minimum or maximum values of a function, that is, problems that seek the "optimal values" or "critical points". These "extremes" are nothing more than those where the slope of the function is 0. We could classify them into 4:



Weak Extrema: critical points such that the neighbouring points around has the same function value, creating a plateau
Strong Extrema: largest/smallest f(x) value from neighbouring points
Global Extrema: largest/smallest f(x) value from all the strong extrema

Saddle Points: inflexion points between concave up and down trends

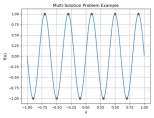


a.2) Types of Optimisation Problems

In addition to different types of minimums and maximums, we can also classify the types of optimisation problems according to their characteristics.

On the one hand we would have the "**Unconstrained problems**" versus the "**Constrained problems**", where the difference lies in being an optimisation problem without any type of restriction, where in this second we will be given a series of conditions that the solution must meet to be a solution.

On the other hand, we could find "**Multi-Solution Problems**" where we would have more than a single global end in the domain of our optimisation problem



Finally we would have the "**Multi-Objective Problems**" or "**MOP**", in which unlike having many possible solutions, we have more than one optimisation problem at the same time. For the resolution of this type of problem we can choose two ways:

-Weighted Aggregation: Creating a final unique optimal function created by aggregations of all the objective functions that are being multiplied by an associated weight value.

-Pareto Optimality: Classifying the possible solutions in a dominance structure, in which domination means a way to distinguish good solutions from bad. There are two types of domination depending on whether they meet one or both conditions, where a decision vector means a possible value from the domain:

A decision vector x_1 strongly dominates another decision vector x_2 if and only if:

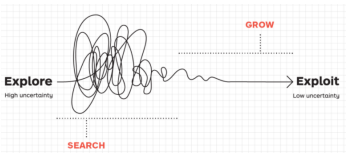
- *x_1 is equal to or better than x_2 in all objectives:* $f_k(x_1) \leq f_k(x_2)\ \forall\ k = 1, \dots, n_k$
- *x_1 is better than x_2 in atleast one objective:* $\exists\ k \mid f_k(x_1) < f_k(x_2)$

In the case where the decision vector meet both conditions it **strongly** dominates the other or the others vectors area as it said in the image. Otherwise, if it only match the second it means that it **weakly** dominates another decision vector.
In this kind of MOP problems we try to find the “Pareto-Front”. It is a set of decision vectors such that each vector weakly dominates each other but strongly dominates all other vectors in the input space.With regard to evolutionary algorithms, after create this Pareto-Front, we can choose the solution with a *tournament_selection*, being the size the Pareto-Front, and choose the most dominant individual to survive. Later we will explain this type of selection.

a.2) Classical Optimization Techniques VS Evolutionary Computation
The reason why EC algorithms are presented is for a series of features that are opposed to classic optimisations and of which it gives us an advantage. ECs present a random search, so it is not assured that for the same initial point the output value is the same (contrast to determinism). In addition, the fact that he works with independent populations allows paralysed search (as opposed to sequential treatment). Finally, they do not necessarily work with differential information, so they are extremely good for non-differentiable problems, in addition to NP hard problems as travelling salesman or scheduling.

a.3) Evolutionary Algorithm Overview
As we mentioned earlier, EAs present an inspiration from biological evolution, describing a process by which individuals become *‘fitter’* in different environments through adaptation, natural selection, and selective breeding.

a.3.1) Structure of EAs
Although we have already previously introduced the structure of the EAs, we will mention it again to delve a little more into this structure that will help us build this algorithm by relying on what this study tells us.
To begin with, it is very important to take into account in these algorithms the terms of **exploitation** and **exploration**. To do this, we must understand that in this type of problem we are navigating within a set of possible values where we try to get as close as possible to the best possible result. In this way, it is as if in a certain way we were approaching that optimal value by observing a portion of the possible values that the population gives us: Equally within this set there is an individual that is very close to a global optimum, or the same said optimal is a local one. This brings us to an uncertainty in which there is always a better value, so we must allow in the problem a certain degree of "exploration" to make populations tend to converge towards the best of the best solutions. This is what these two ideas of "exploitation" and "exploration" are about. On the one hand we must "exploit" the search by tending to a convergence (a solution) without losing the degree of "exploration" that allows us to find an even better solution.



a.3.1.1) Chromosome
As we already know, our chromosome will be the term we use to identify an individual within the problem. This individual will have a series of characteristics or genes that will be reflected in his phenotype. As we will see in the first implementation of my problem, the **genotype** will be reflected as the set of genes defined by a tuple. On the other hand, the **phenotype** will be the set of triangles that these genes define for the construction of the image (Notebook).

a.3.1.2) Initial Population

As a start, we will have the start of the population, where the stochastic factor will be very important when giving a greater degree of exploration. In addition, we will also have to take into account its **size**: the larger the population we will have a greater degree of exploration and diversity, while if that population is smaller it will allow us a greater exploitation by having a lower computational calculation, making it tend to a premature convergence.

a.3.1.3) Fitness Function

Our fitness itself will define the objective function to be optimised. EAs are able to model themselves to all types of optimisation problems that we have previously defined, however there are a some problems that we can find in two of them:

a.3.1.3.1) Constrained Problematics

Because the domain is dynamic in a restricted environment, our algorithm can find unviable solutions. We could solve it in the following ways:

- Physiognomy of the Individual: To make both the generation of new individuals in reproduction and the initial one in such a way that it does not allow restricted values in the problem, by restrict the possibility that the candidate may have some characteristic outside the limitations defined by the problem. Making a preview, in our case a limiting feature could be the margins of an image when recreating a given image. Therefore, in the generation of each individual we will delimit that individual to the size of the image to be recreated.
- Penalty Method: Apply some type of proportional "punishment" to exceed some type of established limitation, where an individual with a very bad fitness becomes better than an individual who at first is very good, but commits some limitation. For this, some type of constant k could be applied as a punishment that at least has the value of the best ideal candidate that the problem could have (failing that, a very large value).

a.3.1.3.2) Multi-Solution Problematics

The fact that we want to find more than one global solution to explore the possibilities we have will pose a problem for these algorithms, since they will tend to converge towards unique solutions or a small group of them. One way to solve this problem could be to apply some type of *penalty* as we saw previously with respect to some *distance* at which a candidate of said solution is already found. In this way, as long as there is a good candidate, but it is close to this optimum, it will be considered a bad solution.

a.3.1.4) Selection

Selection is the part of the algorithm in which after having generated the population we determine which individuals will survive and/or some type of mutation or reproduction will be performed. In this part it is very important to make a compensation between exploitation and exploration, since according to how we select individuals for the next generation we will tend to one thing or another. In practice, sometimes it is better to use both, exploration at the beginning and then exploitation as we get closer to the end.
Among the most popular methods we can find the following:

a.3.1.4.1) Random Selection
As its name suggests, this method is based on a random selection of individuals. In this way it provides us with a good quality of exploration; however, this method alone does not ensure that the best individuals are selected.

a.3.1.4.2) Proportional Selection
Creating a probability distribution of individuals based on their fitness divided by the sum of total fitness ensures a good compensation between exploration and exploitation. However, individuals who from the beginning show high fitness values can dominate the selection process. A solution to this circumstance is to apply a "**Rank Based Selection**", where instead of basing the probability that the individual comes out for his raw fitness, group the individuals in rankings where at the ends of the ranking we have the best or the worst individuals. In this way, we will be able to apply a decreasing probability as we approach a worse ranking. This implementation can give good results, although it makes the algorithm much slower by having to calculate all the fitness of all individuals and classify them.

a.3.1.4.3) Tournament Selection
By selecting a *tournament_size=n*, we randomly choose n individuals from the population to compete with each other by selecting the best among them. This selection can allow us a good degree of exploitation as exploration, because we randomly select a part of the population (giving a certain degree of diversity) and selecting the best of them (increasing the degree of convergence). Depending on whether we seek greater/less exploitation versus exploration, we will increase/decrease *n* respectively.

a.3.1.4.4) Elitism and Hall of Fame Methods
On the one hand we will have the *probability of elitism*, which selects a portion of the population that is better with respect to the entire population. While on the other hand, *Hall of Fame* allows us to save the best value of the population within some selection protocol, ensuring that the best value will not be lost for the next generation since it will be set aside and stored.

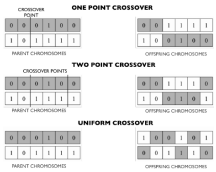
All these methods and more can work together, thus allowing greater diversity based on what the problem requires. Even, in a way, we can see that there is a certain similarity between them (we could say that the selection by tournament is like to use a process of elitism of a single individual after a random selection).

a.3.1.5) Reproduction and Mutation
After having selected the individuals, as we have seen previously, we will apply some method that allows us to create new individuals from those we already had for the next generation. With this we find the **crossover** and the **mutation**.

a.3.1.5.1) Crossover

The crossover is a way to generate new individuals based on certain characteristics of individuals from the previous population that have been selected. This crossing can be *asexual*, *sexual* or *multi-recombination*, depending on whether the offspring will be generated by one, two or more individuals. The fact of having crossover or not can be seen by some type of probabilistic factor. In this way, we also allow the possibility that the genetic material is not lost.

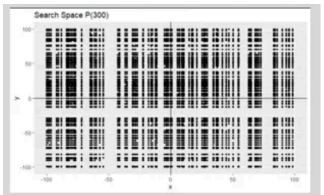
We can also find many variants based on the way in which the genetic material of individuals will be shared.On a chromosome defined by an array of bits it would be quite easy to understand the crossings of *one point*, *two points* and *uniforms*. In the first, we would establish a point from which the part of an individual would be selected, and the rest being that of another (this is the case we put in the introduction). This same rule can be applied to two points, and finally to a uniform distribution. The uniform distribution will allow a more homogeneous distribution of the characteristics of the ancestors, where, for example, each bit of each parent will be randomly selected to introduce it into the new child.

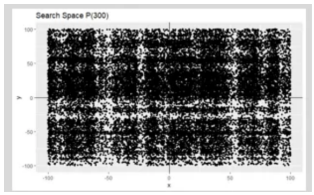


a.3.1.5.2) Mutation

The mutation is a factor that allows us to give a certain "noise" to the population to be able to explore the possible solutions a little more. It is based on altering one or more characteristics of individuals within the generation of the new population. Normally, the mutation factor is usually quite low, since it is usually a fairly random factor that provides greater exploration. The choice of the maximum mutation value depends on the designer, but it is common to maintain about 1% of the total space of the domain for a given variable.

Although it is a factor that will have to be treated delicately, it is also important to observe the power that it could have unified with reproduction. In the left image we would see a case in which only with reproduction it has generated a search for the domain where there is a large unexplored void; while in the right image, thanks to that "noise" produced by the mutation, it has given us a greater range of exploration that alone could not have been possible.





a.3.1.6) Stopping Conditions

To end the iterative search we will have to implement some kind of condition. Although in practice the most used is to propose a **number of generations of stops**, there are other possibilities.

A condition to use could be when we do not find variability in the value of the best fitness. However, this could pose some problems if some kind of elitism is not used since we could find that the value of the best solution fluctuates too much. In addition, if elitism is used, cases could be seen in which that value does not have much variability for several generations before finding a better solution, so we could lose better results.

We could also see the variability of the fitness average. However, if the selection and reproduction operators encourage exploration over exploitation, you could see the case that it never ends. Finally, put an acceptable solution as a stopping point, as we could find in the ANN training when the average quadratic error is below one.

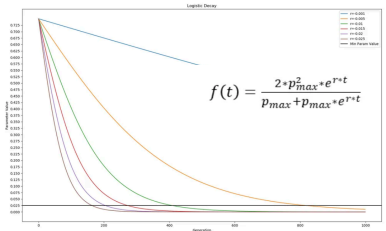
a.3.1.7) Control Parameters

As we have been checking, we will have several options for **hyperparameters** that will directly influence the quality of the problem.The most important are *mutation_probability*, *crossover_probability*, *max_mutation_bound_percentage* and e*litism_percentage*, among others. Depending on how they are implemented, we can consider three types of hyperparameters:

Static hyperparameters are those that are set with a starting value that will remain the same in all generations. Normally, except for elitism for these cases, large static values encourage exploration, while small static values encourage exploitation.

Dynamic hyperparameters decrease, linearly or non-linearly, gradually from large values to small values throughout the execution of the algorithm. For linear functions, a maximum and a minimum will be applied to the hyperparameters, drawing a line that will go from the maximum to the minimum on the number of given generations.

On the other hand, non-linear functions will decrease based on the given nonlinear function, where the most common is "Logistic Decay". Although in this type of function with a logistical nature it is a good practice to put a decrease threshold since the value of the parameter will approach zero, which will lead to stagnation and premature convergence. In this case the threshold is 0.025 (black horizontal size), with *r* the decay ratio, *p_max* being the maximum value for the parameter and *t* the generation.



The problem that could arise in dynamic types is that the values will change regardless of whether they are working well or not. To solve that we will have the S**elf-Adaptive hyperparameters**, which only change depending on how successful they are, if *max_mutation_bound_percentage* is too low, it will self-adapt and increase, or decrease if it is too high.

b) Main types of Evolutionary Algorithms

As I said before, after reading different studies I have perceived some confusion when classifying the different types of evolutionary algorithms. This seems to be due to the appearance of a series of algorithms in parallel in the 60s, which, even when they had been considered individually, they ended up having a certain relationship between them.

It all starts with Turing in 1948, who proposed a "*genetic or evolutionary search*" as a means to provide intelligence to a machine. After that in 1962, Bremermann presented some of the first attempts to apply simulated evolution to numerical optimisation problems in "Optimization through evolution and recombination". With all this and more, in the 60s three different aspects began to appear: ***Evolutionary Programming*** (EP by Fogel, Owens y Walsh), ***Genetic Algorithms*** (GAs by Holland) and ***Evolution Strategies*** (ESs by Bienert, Rechenberg, and Schwefel). These branches were developed independently, but since the early 90s they have been seen as different representations of the same discipline that received the name Evolutionary Computation (EC). With this, two new aspects were added ***Genetic Programming*** (GP by Koza) and ***Differential Evolution*** (DF by Storm y Price) which define the 5 main types of evolutionary algorithms. As we mentioned at the beginning, the EC are somewhat broader than the AE, in addition to mentioning the derivations that can be found based on the specifications used in the AE. In this way, we can define that *metahuristics* that we previously explained as an abstract structure that will allow us to implement in many types of problems.

b.1) Genetic Algorithm (GA)

The genetic algorithm is one of the oldest and most well-known optimization techniques based on nature. This algorithm is normally encoded by means of a binary or floating point representation, where part of the population will be crossed and/or mutated based on a probability to generate the new population. This is intended to preserve genetic information from the parents, since the offspring could get worse. Depending on the type of representation to choose, we will find certain variations in the **crossover** and **mutation techniques**. On the one hand, in the binary crossing techniques we would find the cases of "one point", "two points" and "uniform distribution" that we discussed in the previous section. On the other hand, the crossover present in Floating-Point Representation could be a kind of "uniform crossover" so that each gene belonging to the parents of the offspring could have a probability to appear in the child, and it can be said gene of one or the other (this is what in the publication calls "*Intuitive Crossover*"). In addition, a "*geometric average*" could also be proposed between parents through a linear combination of the components (the proportion of each individual being gamma).

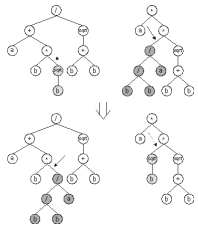
$$\tilde{x}_{ij} = \sum \gamma_p x_{pj}, \ \sum \gamma_p = 1$$

However, crossings alone are not enough. Therefore, we can also find various types of mutation, where as we already know for binary representation it is much more intuitive as would be the fact of randomly flipping bits a certain number of genes; although for floating-point representation we could also implement, for example, adding a uniform random value to each component defined between the limits of each variable.

$$x_j = x_j + U(-x_j^{bound}, x_j^{bound})$$

b.2) Genetic Programming (GP)

This type of algorithm is aimed at the automatic generation of **computer programs** that perform a user-defined **task**. With this, we find the main characteristic in which its phenotype represents a program. These programs are a set of instructions fixed to a series of rules with a certain grammar; where these rules will have to be defined in the problem according to the language with which you are working. In a way, this type of algorithm arises from the mixture between GA and the grammar of the instructions of the programs; thus creating programs of non-fixed length that, when executed, are a candidate solution to the problem

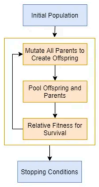


A tree-based structure is usually used as a genotype, which allows us to go from the most abstract tasks/instructions as parent nodes to the sheets as atomic elements of the problem. Therefore, the **Crossover** and **Mutation** processes are usually quite intuitive. On the one hand we could make *crossover_points* with subtrees, so that we cut through some of the branches of the parents and then unify them and form the child. On the other hand, there would be many ways to perform the mutation, such as *exchanging* the value of a node, reducing the tree by *deleting* a node, or in the same way, increasing the size of the tree by *adding* a node (always caring about the grammar possibilities).

b.3) Evolutionary Programming (EP)

Evolutionary programming originated from the basic principles of AI and the intention of this algorithm was to model its behavior. Because the main focus of EPs is the evolution of behavior, in this type of algorithm only **mutations** will be used for the creation of new individuals, where this amount of mutations is known as **behavior**. In addition, raw fitness values will no longer be used to measure based on **relative fitness**, where the fitness of each individual will be represented by how good it is with respect to the rest of the population.

Because the EPs do not have any crossover operator, the mutation must be effective enough to adapt the algorithm to the lack of genetic exchange that crossovers allowed us. Therefore, within the types of hyperparameters that we discuss, this algorithm is that the control parameters of the mutation are *self-adaptive*. To do this, a set of parameters referring to the control of their hyperparameters are included within the set of variables. This will allow to characterize the range of mutation, in this way it is allowed to individualize each individual of the population with respect to their control parameters.



As Brandon comments: "*In evolutionary programming, the 'behavioral' mechanism that is being developed are these strategic parameters, which means that, in addition to the variables of the individual that is evolving, so are its strategic parameters*."

b.4) Evolutionary Strategy (ES)

The generic implementation of evolutionary strategy can become very similar to evolutionary programming. Let's say that the evolutionary strategy could be the union between that advanced mutation operator and what EPs dispense with: **crossover**. In addition, to give it something more of its own character, it presents a couple of its own survival selection operators: **Plus** and **Comma**. After initializing the population, a random selection cycle would be initiated to create offspring by crossover. After that, it uses the advanced *self-adaptive* mutation operator to mutate these descendants (where it will only be accepted if the result is better), to group the offspring and the parents to carry out a type of selection strategy for the next generation.

On the one hand we will have the **Plus strategy**, which generates the same number of descendants or more than the initial generation and then group them with the parents and perform a selection by elitism. On the other hand, in the **Comma strategy** more descendants are generated, they are grouped together with the parents and a selection by elitism is made next to an age value. At the moment an individual reaches his maximum age, that individual is discarded allowing further exploration.

Let

$\mu \rightarrow \#\ of\ \ Parents$

$\lambda \rightarrow \#\ of\ Offspring$

$\kappa \rightarrow Max\ Life\ Span$

Plus Strategies: $(\mu + \lambda)$

$1 \leq \mu \leq \lambda < \infty$

Elitism from Pooled Generation

Comma Strategies: (μ, κ, λ)

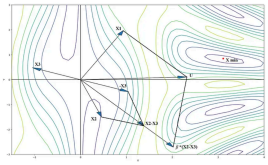
$1 \leq \mu < \lambda < \infty$

Elitism from Pooled Generation, discarding Old Solutions

I find it interesting to comment that the beginnings of this algorithm were like (1+1)ES, where in each iteration, only one new individual with no crossover is created from a one population size. Both evolutionary programming and evolutionary strategy have very close start dates.

b.5) Differential Evolution (DE)

In this paradigm, a set of candidates called agents are presented. These agents are moved around in the search-space by using distance and directional information through unit vectors for reproduction. After the beginning of the population, we will use some selection method for multiparental reproduction, where on the one hand we will have the agents for the **mutation**, and then the **crossing** (all must be different from each other). On the one hand, in the mutation we will select a "target vector" within the selected agents and a set of two or more vectors that will make up the "difference vector" that will be multiplied by a factor ß that will work as a scaling value (values between [0, inf] where smaller values implies greater convergence, and larger values a greater exploration) which is the main strategic parameter (from which that variability between exploitation-exploration will be determined most). With all this, we will form a vector ***U*** of the form:



$$unit = target + scale(difference)$$

$$u_i = x_{i1} + \beta(x_{i2} - x_{i3})$$

where $i \neq i1 \neq i2 \neq i3$, u_i *is unit vector,* x_{i1} *is target vector,* x_{i2} *and* x_{i3} *are selected parents, and* β *is a scaling coefficient*

After calculating the unit vector ***U*** we can enter the crossing, where after having chosen another "main parent" agent (different from all the previous comments) and a crossing is made such as the intuitive crossing and the average geometric crossing commented in the GA. As may have been observed, the main mechanism for the creation of offspring is the creation of the unit vector itself. Therefore, there are different strategies known as DE/x/y/z strategies (*chosen target vector* / *n vectors of difference* / *crossover*), where the defined would be of the type DE/x/1/z since we assume that the target vector is random, we have a single difference and the type of crossing is not yet determined.

4. Application areas

Describe selected fields of application of evolutionary algorithms.

EAs are suitable mainly for optimization, scheduling, planning, design, and management problems. Knowing that these kinds of problems are everywhere (in investments, production, distribution, and so forth) and the increasing use of indeterminist technologist in recent years due to its resolution capacity, we can observe a lot of different uses for this field. Analysing the results obtained from the WoS database, the EAs methods are mainly used in the area such as:

- **Engineering Electrical Electronics**
- **Computer Science Artificial Intelligence**
- **Computer Science Theory Methods**
- **Computer Science Interdisciplinary Applications**
- **Automation Control System**
- **Computer Science Information Systems**
- **Operations research management Science**

To give an overview within the industry field, we will concentrate only on the real-world applications of particular EA methods showing a few of papers that provided solutions using this approach.

a) GENETIC ALGORITHMS

As a universal optimization tool, GAs can solve constrained optimization problems, multimodal optimisation problems, continuous optimization problems, etc. Thus, there is a wide range of applications of GAs. In this case we can see papers such as:

- Solar array layout optimization for Stratospheric Airship using numerical Method. It presents a numerical method based on rotating model of a stratospheric airship to optimize the solar array layout. As results, we can see that the model is helpful in the preparation stage for installing large area flexible solar arrays with significantly improved on the output power of solar panel.

- A genetic algorithm approach for location-inventory-routing problem with perishable products. In this paper, they address a location-inventory-routing model for perishable products. The model determines the number and location of required warehouses, the inventory level at each retailer, and the routes traveled by each vehicle. Noticing the model developed is a NP-hard problem, they decide to develop a GA approach to solve the problem efficiently, achieving high quality near-optimal solutions in reasonable time.

- The evolutionary cost of Baldwin effect in the routing and spectrum allocation problem in elastic optical networks. This paper has two objectives: The main objective in this paper is to investigate the pros and cons of hybridisation on the base of a hard practical and up-to-date problem, covering with it the second objective that is propose an effective optimization method for solving the Routing and Spectrum Allocation of Multicast Flows (RSA/M) problem in Elastic Optical Networks (EONs), taken as the base problem for the first objective. Hybridisation refer on adjust an EA problem by adding a different kind of problem-dependent mechanisms. In this case, they decided to hybidizated it with local search algorithms. Such hybridisation may lead to phenomena called the Baldwin effect, which helps to preserve population diversity, but at the same time might cause significant increase of computation load.

GENETIC PROGRAMMING

- Evolutionary Metric-Learning-Based Recognition Algorithm for Online Isolated Persian/ Arabic Characters, Reconstructed Using Inertial Pen Signals. With the development of sensors appears new tools for human-computer interaction, such as inertial pens. In this paper, analysing the features of the inertial pen character recognition they detect some limitations to apply these methods in languages with complex similar characters. In an attempt to improve this methods they isolate the character recognition for Persian and Arabic characters using a characteristic function which is calculated for each character using a genetic programming algorithm. The experimental results show that the performance of the proposed method is superior to that of one of the state-of-the-art works in terms of recognising those specifics handwriting characters

DIFFERENTIAL EVOLUTION

- Dual-Objective Scheduling of Rescue Vehicles to Distinguish Forest Fires via Differential Evolution and Particle Swarm Optimization Combined Algorithms. This research issue about performing the emergency scheduling of forest fires arises in order to minimise the number of rescue vehicles while minimising the extinguishing time and complete this task giving limited vehicle resources. To do so, they present a novel multi objective scheduling model using a Multiobjective Hybrid model with DE and Particle-Swarm-Optimization algorithm, applying this approach to a real-world emergency scheduling problem of the forest fire in Mt.Daxing'angling (China) and giving Pareto solutions. The experimental results show that the propose approach is able to guide decision-makers in making better decisions in fast way.

As we can observe with these examples, EAs approach can lead a wide range of problems from completely different fields: from a location-inventory-routing model for perishable products to improving the characters recognition of a Inertial Pen.

5. Review of selected programming libraries and tools

Describe selected programming libraries and tools supporting evolutionary algorithms programming.

Because last year I did some work related to genetic algorithms with libraries of my university that worked as a framework in Java, I considered this project as a challenge and start generating the algorithm by hand. In addition, I wanted to use **Python** instead of Java since it is normally used more in the data science industry and to learn and develop new knowledge of Python using libraries that I had never used before. However, it would also have been quite interesting to have used the *LEAP framework*, a general-purpose EC package that would have made things easier for me and achieved better results. Nevertheless, I am really glad I have been building it little by little to understand better how it works.

Libraries and Modules

As our problem is related to the processing of images, we have used the "**NumPy**" library for the treatment of images as arrays; "**Pillow**" to read, generate, operate and save images; "**imageio**" was also used to be able to save the images in the algorithm implemented in the notebook. In addition, "**matplotlib**" for certain graphs and representations.

Regarding specific libraries used for fitness, on the one hand we have used "**scikit-image**" for the *skimage.metrics.structural_similarity* method that calculates the average structural similarity index between two images. On the other hand, the "**colour-science**" library for the *colour.difference.delta_e.delta_E_CIE1976* method in an attempt to use a metric that quantify how the human eye detects different colours.

In relation to the creation of random data we have used the "**random**" Python module that has allowed us to give it the stochastic trait that these algorithms present

In addition, I have used the "**multiprocessing**" library with which we are allowed to paralyse tasks within the implementation.

IDE

For the main implementation, in which I have tried to describe the process step by step, I have used a "*Jupyter Notebook*"; while for the other two implementations I have used "*Visual Studio Code*".

6. The implemented algorithm

Describe in detail the implemented algorithm: solution representation, evolutionary operators (recombination and mutation), selection methods and key parameter values.

The algorithm that I decided to implement is based on the reconstruction of an image with four channels (RGBA) given, in our case the "Gioconda", through the random generation of images formed by polygons. To do this, I have raised the problem as a minimization problem where the fitness function is a specific measure that calculates some type of error between the generated individual, which represents a possible image solution, and the target image. Because it has not been a simple task, this work has been carried out with three different implementations: The implementation proposed in a Notebook called "*AI_Proyect_First_Approach.ipynb*", where the results do not resemble the image so it is a step-by-step explanation how to generate a genetic algorithm model and three different fitness cases are raised with certain conclusions. An implementation with certain changes based on the publication of "*A True Genetic Algorithm for Image Recreation — Painting the Mona Lisa*." And finally, the last implementation where I generate different search cycles by interspersing parallel searches of subdivisions of the problem that allow it to be parallelized.

Taking into account this tour, I would like to comment here on the final result with the modules"***GA_Multi.py***" and "***Individual.py***".

Structure of the Evolutionary Algorithm

Individual.py: Chromosome, Fitness and Individual_Blocks

Chromosome

In this module we define a class called "Individual". This class defines the tools that I have considered necessary for the modeling of individuals. Each individual in the population will be an object of this type, where the properties of "*length*" and "*width*" refer to the size of the frame of the target image to be solved; "*fitness*" to the error defined by the "*get_fitness(target)*" method that given a target image calculates the error produced between one image and the other; "*polygon_n*" refers to the number of polygons as the limit that will be given for the random generation of an individual; and finally the "*image*" and "*array*" properties that represent the image in two different formats which make up the chromosome.

```
class Individual:
    """
    Individual defined by:
        -lenght    -> the lenght of the image target
        -width     -> the width of the image target
        -fitness   -> the error between the image of the individual and the image target
        -poligon_n -> the maximun number of poligons that is going to be used in case where we create a random image from this individual
        -image     -> the image that define the individual as an Image.Image from Pillow
        -array     -> the image that define the individual as an np.ndarray from NumPy

    """
```

```python
def __init__(self, length, width, poligon=6, img=None):
    self.length = length
    self.width = width
    self.fitness = float('inf')
    self.poligon_n = poligon
    if(isinstance(img,Image.Image)): #In cases where it receive an image as an obj Image.Image (Pillow)
        self.image = img
        self.array = np.array(img,dtype=np.uint8)

    elif(isinstance(img,np.ndarray)): #In case where it receive an array as image (Numpy)
        self.image = Image.fromarray(img.astype(np.uint8))
        self.array = img.astype(np.uint8)
    else:   #in cases where there is no img gave
        self.image = None
        self.array = None
        self.create_random_image()
```

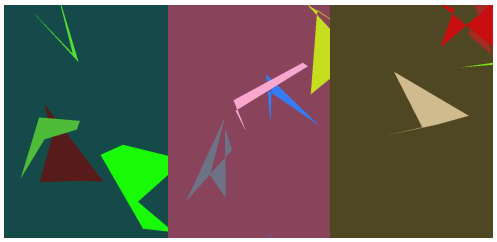
As we can see in the code, the creation of an individual can be in three ways depending on the img value that enters as a parameter: as an **Image** object, as a **numPy** or "*None*" without receiving an image. The first two cases are raised for crossings, where depending on whether a new individual is generated by a method of one library or another, an object of a library or another is returned by assigning its corresponding values to each property of the class. However, in the case in which I don't know anything by the "*img*" parameter, we will be in front of an individual who has been initialized to be randomly generated from scratch. This implies the method "***create_random_image()***".

```python
#Method that create set the properties "image" and "array" generating a random image with a random number of poligons between [3,poligon_n]
def create_random_image(self):
    poligon_num = random.randint(3,self.poligon_n) #Generate from 3 to poligon_n poligons
    region = (self.length+self.width)//8 #With this we will give the chance to create points out of the margin but not so far away
    #this will allow to create more different geometric structures that could be good for the design

    #Generate a new object Image with the size of the problem and
    #  with a random color for the background gived as a String code
    img = Image.new(mode="RGBA", size=(self.length,self.width), color=self.random_string_color())
    for i in range(poligon_num):
        num_points = random.randint(3,6) #from poligon of 3 to 6 points
        x = random.randint(0,self.length)
        y = random.randint(0,self.width)
        #Create a list of tuple_points considering the number of points that conform the poligon
        #The values of the x/y dimension are selected randomly with the margin bounds, and we also
        # let the points be created outside the margin within a given "region" which makes them not stray too far away.
        xy = [(random.randint(x-region,x+region),random.randint(y-region,y+region))
              for _ in range(num_points)]
        #After create the points, giving to the poligon method the list of dots as parameters and the random color
        # that the PIL has to create random colors
        img1 = ImageDraw.Draw(img)
        img1.polygon( xy,fill=self.random_string_color())

    self.image = img
    self.array = np.array(img)
```

```python
@staticmethod
#Static Funtion that give as a random string that
# suite a color codification
def random_string_color():
    return "#"+''.join([random.choice('0123456789ABCDEF')
                        for j in range(6)])
```

This method generates a new "*Image*" object with 4 channels and the measures corresponding to those given by the problem (which must be the same as those of the individual) and with a background color randomly generated by the static method "*random_string_color()*" that returns a string of characters that refers to a color. After that, it generates a random number of points between 3 and the "*polygon_n*" to draw polygons by means of the "*ImageDraw.Draw.polygon()*" method, doing as we did in the Notebook with the "*decoder*", along with the random color of said polygons defined by the same function defined above.This generated image is the one that is then stored in the properties of the individual at the end. It is important to note the difference between the first implementation and this one on the chromosome, since in this case apart from working directly with the image (which we will see later in the methods of crossing and mutation), the colours that are given randomly are given as a **string**, the **background** color covers a relevant importance (which was not before) and finally the variable "***region***". This region variable will allow us to increase the variability of the shapes, so that the generated polygons will have a greater definition range within the limits of the image. So, if, for example, a triangle has one of its

points outside the frame of the image, we would now have a rectangle. In this way we would have results like these on the right.



As can be seen, it is not the only difference with the first paradigm, since in this case, for example, the genes are no longer the triangles that make up the image, but the previous *decoder* is now the generator of the chromosome itself, and we work with the image directly.

you can see more examples “GA/experiments/initial_population”

Fitness

As we mentioned earlier, in this module we also have a function called *get_fitness(target)* that will give value to the *fitness* property of each individual.

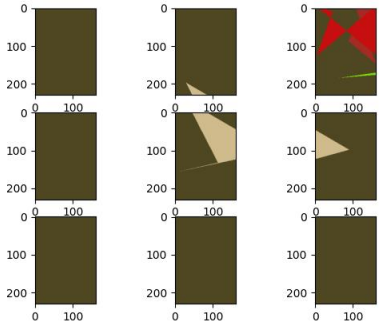
```python
#Method fitness which provide the fitness that the Individual has as an error between the Individual image and the image target
def get_fitness(self,target):self.fitness=np.mean( colour.difference.delta_e.delta_E_CIE1976(target,self.array ) ) # type: ignore
```

In this case we have directly used the given average of the difference ΔE matrix between the image of the individual and the target image, which we will use as a function of errors to minimize trying to store as much as possible at 0.

Individual_Blocks

In this module we have a last method that will be used in the iterative algorithm and it was one of the last implementations made. This method will allow us to divide an individual into equal portions. These portions will be defined by the *next_block_size* parameter to which given a tuple *(x, y)* divides the individual into *x*y* Individuals, each individual being a part of the initial individual.

```python
#Method that return a list of Individuals which are the divisions of this Individual.
# The dimension of the division is defined by the parameter next_block_size which
# define the (width,lenght) proportion that is going to be took from the photo. At the end,
# multiplying this 2 portions given we can know in how many pieces we are diving this Individual.
# The default setting is in 9 pieces.
def div_in_blocks(self,next_block_size = (3,3)):
    w_size, l_size= next_block_size
    img = self.__copy__().array
    blocks = []
    b_w_size, b_l_size = self.width // w_size , self.length // l_size
    for iw in range(w_size):
        for il in range(l_size):
            x = b_w_size * iw
            y = b_l_size * il
            if(isinstance(img,np.ndarray)):
                block = img[int(x):int(x+b_w_size), int(y):int(y+b_l_size)]
                new_Ind = Individual(b_l_size,b_w_size,img=block)
                blocks.append(new_Ind)
    return blocks
```



you can see it in “GA/experiments/initial_population/div_blocks”

GA_Multi.py: Module_Problem, Initialization, Tournament_Selection, Reproduction Strategies,runGA_Multi, Problem_Blocks and Multi_EA

Module_Problem

In this module we create the “GA_Multi" class. This object can be understood abstractly as "the problem" in itself. In this module I have implemented both the initialization of the population, the selection, the reproduction strategies and the evolutionary algorithm.

The main properties it presents are: “*img*"and “target” being the target image as an Image object and numPy object respectively; "*length*" and "*width*" as the length and width of the image; and "*block_w_size*" and "*block_l_size*" as the portion of length and width into which the target image will be divided.

```
"""
GA defined by:
    img -> The target image
    length -> The length of the target image
    width -> The width of the target image
    block_w_size -> The width portion to split the target image
    block_l_size ->  The length portion to split the target image
"""
```

```
def __init__(self, path,block_w_size=3,block_l_size=3):
    if(isinstance(path,str)):
        target = Image.open(path).convert('RGBA')
        target = target.resize((972,1383))
    else:target=path

    self.img = target
    self.target = np.array(target,dtype=np.uint8)
    self.length,self.width = target.size
    #for the splits and concatenation of the problem
    self.block_w_size = block_w_size
    self.block_l_size = block_l_size
```

In order to be able to instantiate an object of the type “*GA_Multi*" we have given the parameters of "*path*", "*block_w_size*" and "*block_l_size*". Similar to how it happened with individuals, to create a *GA_Multy-type* object it can be done in two ways according to the type of value passed by the "**path**" parameter. On the one hand, this value can be a string of characters indicating *where* the target image to be reconstructed *is located* (for our case this will be called the "**General Problem**" since it will be the image to which we want to approximate as a final objective). And on the other hand, we will have the possibility of giving an image as a target image value for the generated object (these instances have been used to generate "**Subproblems**" or "**Secondary Problems**" that will be parts of the general problem that will try to solve in a paralyzed way. The images as a parameter will be portions of an image belonging to a *General Problem* which divides its image into those **portions** defined by "*block_w_size*" and "*block_l_size*" to generate another problems of smaller size and then unify them and solve the *General Problem*. Later we will show how).

Initialization

We would already have the way to form a problem object, so the next thing will be to generate our initial population. This population is generated by means of the *initialization(pop_size,best_first,polygon_n)* method, where given a population size "*pop_size*" a list composed of individuals with the same width and length as the target image as a population is generated. As we can see, by not giving a value to the "img" parameter of the Individuals, a random image will be self-generated with the procedures explained above. In addition, for each individual created, we will call *get_fitness(target)* so that the value of your fitness is defined. In the case in which the "*best_first*" parameter is given to "*True*" it will allow the first value of the population to be the individual with the best fitness of the initial population.

```python
#This Method initialize a random population with a given number
# of members and with the same margin that the problem GA has.
#If the parameter "best_first" is True, then the first value
# gave after create the population will be the best of the random
#population generated.
def initialization(self,pop_size, best_first = False, poligon_n = 6):
    population = []
    for _ in range(pop_size):
        new_Ind = Individual(self.length,self.width,poligon_n)
        new_Ind.get_fitness(self.target)
        if (best_first == True and len(population)!=0):
            if(population[0].fitness >= new_Ind.fitness):
                population.insert(0,new_Ind)
            else: population.append(new_Ind)
        else: population.append(new_Ind)
    return population
```

Tournament_Selection

As a selection method for the selection of parents who are going to be crossed, I have decided to use the "*selection by tournament*". The *tournament_selection(population, tournament_size)* function will select the best individual within a sample of *tournament_size* size taken from the *population* input set.

```python
@staticmethod
#Static Method that generate a tournament selection from a population and a tournament size given.
def tournament_selection(population, tournament_size=6):
    ind = population[-1] #Select a Individual from the population
    #Use there properties to create a new Individual to redifine its fitness
    winner = Individual(ind.length,ind.width,img=ind.image)
    winner.fitness = None
    for _ in range(tournament_size):
        #select a random candidate and ask if its fitness is better than the
        # winner fitness
        cand = random.choice(population)
        if(winner.fitness == None or cand.fitness < winner.fitness):
            #exchange the competitors and give a chance as winner for a better candidate
            winner = cand
    return winner
```

Reproduction Strategies

In this part we will talk about the mutation and crossover strategies that I have proposed. Based on the commented publication, I wanted to give more than one possibility of crossover and mutation based on a probability factor that we will see at the end in the complete construction of the algorithm. For now, let's focus on the different types that I have been implementing:

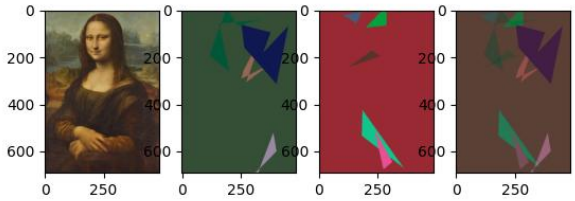
Crossover Strategies

Before starting with the three different types of crossover implemented, I would like to indicate that all of them have a parameter called ***elitism*** which, depending on whether it is "*True*" or "*False*" will determine the return of the candidate or return "*None*" in the case in which said generated crossing is worse than any of the parents for which it has been formed.

- ***Crossover_Blend(parent1, parent2, elitism)***

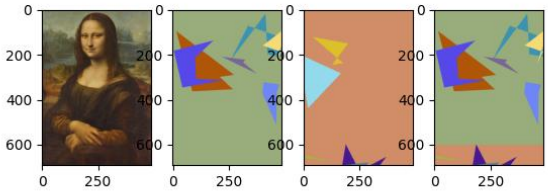
This method blend two individuals given as parameters using the *Image.blend(parent1, parent2, alpha)* method where *alpha* define the *opacity percentage* that will be selected per pair of parents. If *alpha* is 0.0, the child result would be the *parent1*; then, if *alpha* is 1.0 the child would be *parent2*. In our implementation we us "*random.random()*" as *alpha* to give a schocastic feature of how the crossover is going to end up.

```python
def crossover_Blend(self,ind1,ind2,elitism=True):
    child_image = Image.blend(ind1.image,ind2.image,random.random())
    child = Individual(self.length,self.width,img=child_image)
    child.get_fitness(self.target)

    if(child.fitness != min(child.fitness,ind1.fitness,ind2.fitness) and elitism):
        return None
    return child
```



- ***Crossover_Point(parent1, parent2, elitism, horizontal_prob)***

In this approach, first we select randomly If we are going to do the horizontal or vertical crossing by the *horizontal_prob* parameter given. Based on this, a series of columns/rows will be randomly selected to create a matrix of 1s and 0s and its opposite, with the same dimensions and channels as that of individuals. With them, we will multiply one of the parents by one and the other by the opposite, causing the columns/rows corresponding to 1s to keep those *genes* of that parent, while the columns/rows that are 0s will invalidate genes of that parent. In this way, and having done the same with the opposite, these can be added by maintaining the corresponding columns/rows of each one based on that random generation of matrices.

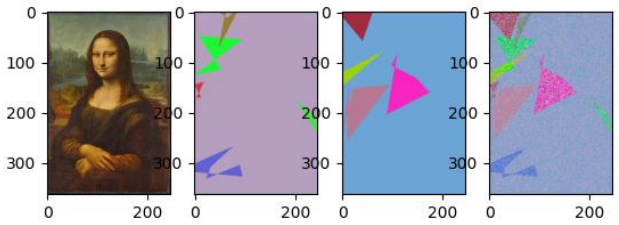



```python
def crossover_Point(self,ind1,ind2,horizontal_prob=0.5,elitism=True):

    #horizontal crossover point
    if random.random() <= horizontal_prob:
        #choose a column from
        split_point = random.randint(1,self.width)
        #create a matrix of ones with the measure of the columns chosen
        ones,zeros = np.ones((split_point,self.length)),np.zeros((self.width-split_point,self.length))
        #then add to this matrix an extension of 0s columns which correspond to the rest of the columns to fit the image's size
        first = np.vstack((ones,zeros))

    #vertical crossover point --> same procedure but now with rows instead of columns
    else:
        #choose a row from
        split_point = random.randint(1,self.length)
        #create a matrix of ones with the measure of the rows chosen
        ones,zeros = np.ones((self.width,split_point)), np.zeros((self.width,self.length-split_point))
        #then add to this matrix an extension of 0s rows which correspond to the rest of the rows to fit the image's size
        first = np.hstack((ones,zeros))
    #this is the way we create the opposite matrix --> where there were 0s now there are 1s and vice versa
    second = 1 - first

    #now we add the differents channels that compound the image with the same values that we generate before
    first_part = np.dstack([first]*self.target.shape[-1])
    second_part = np.dstack([second]*self.target.shape[-1])

    #multiply the matrix with the parents to add the random part which now will correspond to the child
    half_child1,half_child2 = np.multiply(first_part,ind1.array),np.multiply(second_part,ind2.array)
    offspring = np.add(half_child1,half_child2)
    child = Individual(self.length,self.width,img=offspring)
    child.get_fitness(self.target)

    #elitism
    if (child.fitness != min(child.fitness,ind1.fitness,ind2.fitness) and elitism):
        return None
    return child
```

- ***Crossover_Pixel_Wise(parent1, parent2, elitism)***

In this method we do something similar to the previous approach, but instead of choose a number of columns or rows from each parents, we just create a random matrix with 0s and 1s with the shape of the target and its opposite. After that, We would proceed in the same way: multiply one parent by the matrix, the other by the opposite and add them to generate the offspring.

```python
def crossover_Pixel_Wise(self,ind1,ind2,elitism = True):
    # generate a random matrix with the target shape and defined by 0 or 1
    first = np.random.randint(2,size = (self.target.shape))
    second = 1-first #opposite Matrix
    #multiply the matrix with the parents to add the random part which now will correspond to the child
    half_child1, half_child2 = np.multiply(first,ind1.array),np.multiply(second,ind2.array)
    offspring = np.add(half_child1,half_child2) #Mix the parents
    child = Individual(self.length,self.width,img=offspring)
    child.get_fitness(self.target)
    #elitism
    if (child.fitness != min(child.fitness,ind1.fitness,ind2.fitness) and elitism):
        return None
    return child
```

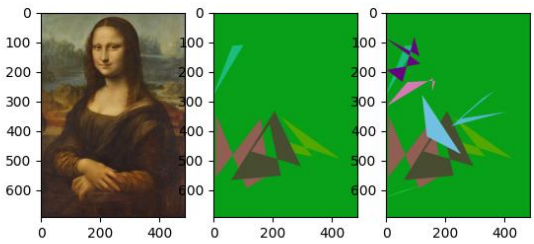


Mutation Strategies

Now we will stop to explain the two types of mutations that I have implemented. Although it will be seen later, it must be taken into account that these mutations will have a very small percentage and will be shared by both.

- ***Mutation_Polygon(ind, n_polygons)***

This method consists of adding a random number of more new polygons between 1 and *n_polygons* to the copy of the individual *ind* given as an argument. In this case, *region* is a little different, since in this case it is chosen between 1 and one-fifth of the sum of the margin measurements. This is with the intention that the possible values have a greater range, at the same time that this does not always happen since it is chosen between 1 and said value (so there will be cases in which even the allowed range will be less than the one proposed for the random generation of the individual)

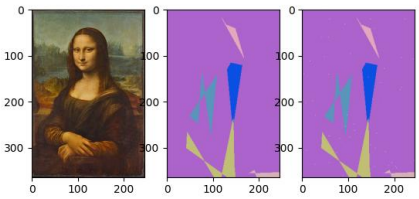



```python
def mutation_poligon(self,ind,n_poligons=6):
    img =  ind.image.copy() #to not really alterate the individual and drop a new one
    poligon_num = random.randint(1,n_poligons)
    region = random.randint(1,(self.width+self.length)//5)
    #SAME as the generation of random individuals
    for i in range(poligon_num):
        num_points = random.randint(3,6)
        x,y = random.randint(0,self.length) , random.randint(0,self.width)
        xy = [(random.randint(x-region,x+region),random.randint(y-region,y+region))
              for _ in range(num_points)]
        #adding more poligons to the image img
        img1 = ImageDraw.Draw(img)
        img1.polygon( xy,fill=ind.random_string_color())

    child = Individual(self.length,self.width,img=img)
    child.get_fitness(self.target)

    return child
```

- ***Mutation_Polygon(ind, pixel, change)***

In this last method we let the chance to change a *pixel* number of pixels chosen randomly by adding a random change between [-*change* and *change*]. Just to make sure that the values of the image not overflows, I use the function "*np.clip(image, lim_min, lim_max)*" to limit the values in the proper bounds (although after some checks there does not seem to be a change). It is true that if we increase the size of the image (as I ended up doing in the last experiments) the mutation is barely noticeable due to the size of the matrix, even more so if the values given as parameters are small. However, I am confident that these small changes can lead to some kind of improvement.

```
def mutation_Pixels(self,ind, pixels=40,change=20):
    indArray = ind.array.copy()
    for _ in range(pixels):
        y= random.randint(0,self.length-1) #random y value
        x=random.randint(0,self.width-1) #random x value
        z= random.randint(0,self.target.shape[-1]-1) #random z value
        indArray[x][y][z] = indArray[x][y][z] + random.randint(-change,change) #change the pixel
    #cast values are in the proper limit for a image
    imgArray = np.clip(indArray,0,255) #not cross the possible range of values that a pixel can have

    child = Individual(self.length,self.width,img=imgArray)
    child.get_fitness(self.target)

    return child
```



runGA_Multi

Now that we have all the ingredients, we can go to the implementation of the evolutionary algorithm. This is handled by the "*runGA_Multi*" function. Because the code is somewhat long, we are going to explain it little by little. Maybe there are some parts that may not be completely clear, because this function is the one that performs the search, but said search will be called by another, and even by different problems at the same time. I'm sure that when we get to that part everything fits better.

```
def runGA_Multi(self,index_procex,return_dict, pop=[]
                ,p_size = 100, generations=15000,CO_B = 0.6, CO_P = 0.30, CO_PW = 0.05, M_R=0.05,
                stop_exploration = 0,stop_random_Hyper = 0, tournament_size = 6, horizontal_prob=0.5,string_block=" "):
    population = pop; size = len(population)
    best_pop_per_print = []
    #in the case the p_size that have to be is more than the pop given we extend the population with more new random values --> More exploration
    if(size < p_size):
        population.extend(self.initialization(p_size-size))
        #to mantain the best candidate at the begining we compare the best from the initialization and the best of the given population
        if(population[p_size-size-1].fitness < population[0].fitness):
            population.insert(0,population.pop(p_size-size-1))
    if(size > p_size):
        population = population[:p_size-1]
    #the list that we will return with the improvement of the generations
    best_population = [population[0] ]
```

To begin with, the algorithm initializes the population by the *pop* parameter. This is because in principle you can give a starting population to the algorithm. If so, it is verified that the *p_size* as "the expected size of the population" have the same size. If not: if it were smaller, the population would be extended with totally new genetic material (increasing the exploration) maintaining that standard that the first value of the population lists is the best individual of that generation; while on the other hand, if that population were larger than expected, it would be cut to the defined limit (*increase in exploitation*).

After that, we keep in the variable "*best_population*" the best result so far, so that later, after each new generation, it is checked whether or not it is better than the best candidate of the new population. As we can see below, the "*new_population*" list must be added to the new population that is being **generated** (now we go into how), and it follows the same procedure that we followed with the "*initialization*" method, where the best individual goes to the beginning. In this way, in "*best_population*" the best candidates per generation will always be saved.

```
    if(len(new_population)==0 or new_population[0].fitness>child.fitness):
        new_population.insert(0,child)
    else: new_population.append(child)

if(new_population[0].fitness < best_population[0].fitness):best_population.insert(0,new_population[0])
```

Continuing with the code we would have the parameters *stop_exploration* and *stop_random_Hyper*, which will affect within the loop of each generation. On the one hand, the first refers to the number of generations in which we would like to remain without an elitist focus on the different types of crossover, changing or not the *elitism* variable. This will allow us a greater exploration or not allowing worse individuals than parents to maintain.

```
        elitism = False #not elitsm aproach
        #initialize CO_Probs
        CO_B_Act = CO_B ;CO_P_Act = CO_P+CO_B_Act; CO_PW_Act=CO_P_Act+CO_P_Act+CO_PW
        i = 0
        while i < generations : #and len(best_population)<p_size YA VEREMOS
            if(i == stop_exploration): elitism=True # elitism aproach
```

While the second refers to the number of generations in which we would like to stay with random crossover ratios. If so, we will find new ratios in each individual. However, in cases where this is not the case, the default values given as parameters must be restored. These values refer to the probability that one of the three crossings will be made.

```
    new_population = []
    while len(new_population) < p_size:
        child = None
        parent1 = self.tournament_selection(population,tournament_size) #tournament size = 6
        parent2 = self.tournament_selection(population,tournament_size)
        if(i < stop_random_Hyper): # i < stop_random_Hyper
            CO_B_Act = random.uniform(0,1)
            CO_P_Act = random.uniform(CO_B_Act,1)
            CO_PW_Act = random.uniform(CO_P_Act,1)
        CO = random.uniform(0,1) #how to make sure that at least have one type of crossover
```

In the event that the sum *CO_B_Act*, *CO_P_Act* and *CO_PW_Act* (the respective portions of "*crossover_Blend*", "*crossover_Point*" and "*crossover_PixelWise*" with different values depending on whether they are random or not) is not 1, it will mean that the remaining value will be the probability that the parents selected by tournament for sexual crossing compete with each other to stay as the best of the two.

```
    if(i == stop_random_Hyper): #random hyperparamiters, not anymore
        CO_B_Act = CO_B
        CO_P_Act = CO_P+CO_B_Act
        CO_PW_Act=CO_P_Act+CO_P_Act+CO_PW
```

```
else:#fight to survive
    child = parent1 if parent1.fitness < parent2.fitness else parent2
```

Finally, we would arrive at how this new population is generated.

```
new_population = []
while len(new_population) < p_size:
```

First, we would select two parents, as we discussed, by means of the "*tournament_selection*" method. After that, it is evaluated which type of crossover has been randomly selected with its corresponding ratios. In the hypothetical case in which the variable "elitism" was "True", the selected crossing method will be repeated as many times as necessary with new parents until it reaches a point where the breeding is better than its descendants.

```
if(CO<CO_B_Act ):#CrossOver_Blend
    child = self.crossover_Blend(parent1,parent2,elitism)
    while(child==None):
        parent1=self.tournament_selection(population,tournament_size)
        parent2=self.tournament_selection(population,tournament_size)
        child = self.crossover_Blend(parent1,parent2,elitism)
elif(CO<CO_P_Act ):#CrossOver Point
    child = self.crossover_Point(parent1,parent2,horizontal_prob,elitism)
    while(child==None):
        parent1=self.tournament_selection(population,tournament_size)
        parent2=self.tournament_selection(population,tournament_size)
        child = self.crossover_Point(parent1,parent2,horizontal_prob,elitism)

elif(CO<=CO_PW_Act): #CrossOver PixelWise
    child = self.crossover_Pixel_Wise(parent1,parent2,elitism)
    while(child==None):
        parent1=self.tournament_selection(population,tournament_size)
        parent2=self.tournament_selection(population,tournament_size)
        child = self.crossover_Pixel_Wise(parent1,parent2,elitism)
else:#fight to survive
    child = parent1 if parent1.fitness < parent2.fitness else parent2
```

After the child is generated in any way, said child may have the possibility of being mutated or not based on a shared *M_R* mutation ratio given as a parameter. In the event of this, both approaches will have the same probability of being selected. After this process, what was previously said about the best individual will be evaluated to introduce it or not at the beginning of the *new_population* list.

```
#Mutation
random_m = random.uniform(0,1)
if(random_m<=M_R):
    mutationType = random.randint(0,1)
    if mutationType < 1:#Mutation Pixel
        child = self.mutation_Pixels(child)
    else:#Mutation Polygon
        child = self.mutation_Polygon(child)
```

After the completion of a generation, *new_population* is exchanged for *population* and 1 is added to *i* to continue with the next generation. For every 100 generations or the last one, the best of the candidates will be selected within *best_population* to be entered along with their generation index in the *best_pop_per_print* list. Finally, this individual will be saved as a png image.

```
if((i%100==0 or i == generations -1) and string_block[0]=="_"):
    candidate=best_population[0]
    if(index_procex!=-1):
        best_pop_per_print.append((i,candidate))

    candidate.image.save("pictures/multi/Colour/"+string_block+"/"+"num_PoP_"+str(p_size)+"_epouch_"+str(i)+" of "+str(generations) +"_fitness_"+str(candidate.fitness)+".png")
```

It is important to note that in order to enter into said conditional it is also necessary that the first value of *string_block* has to be "_". This will be an indication of whether we are facing a *General Problem* or if it is a *Secondary Problem*. *Secondary Problems* will not be saved, since their function in itself is to help the final representation of the *General Problem*.

Finally, it is very important to take into account the last two lines. On the one hand, the *Secondary Problems* will be associated with independent processes, which do not return a value. Therefore, the values are associated with a dictionary that indexes each *Secondary Problem* the last population ("*population"*). However, in cases where we are talking about a *General Problem* it will be possible to extract the return value, where in these cases it will be both *population* and *best_pop_per_print*. This can be useful since we might want to use the best solutions of every 100 generations like a "*Walk of Fame*". This could be interesting from the point of view of being used as another population as an initialization. This will make sense when we explain the "*runGA_Multiprocessing*" function.

```
return_dict[index_procex] = population
return population,best_pop_per_print
```

Problem_Blocks

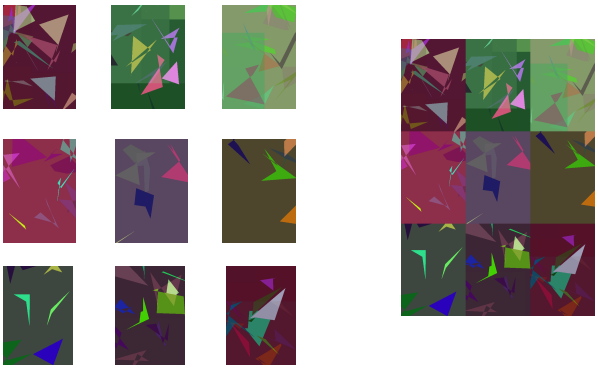
Before reaching the final algorithm, I need to explain these last two methods: *div_image_blocks* and *merge_population(population_subBlocks)*

This part is the newest I've had in the implementation of my code. The approach comes after having seen in certain publications that there are certain specifications of these types of algorithms that use parallel processes. In addition, in one of the parts of Brandon Morgan's publication, in which he talks about the "Island Algorithm", he comments that because the real problems posed in the industry can be very expensive, parallelization can help those times. Because of this, and seeing that this algorithm to obtain relatively decent or "appreciable" results requires a lot of time, I was asked the following: With the same strategy of the "divide and you will win", why not divide the image into smaller pieces turning a problem into smaller problems and that work in different processes. In addition, the fact of having to work with the reproduction of a smaller image implies an advantage, since the work to be done between matrices is much smaller.

- ***Div_image_blocks(next_block_size)***

```python
def div_image_blocks(self,next_block_size = (3,3)):
    w_size=self.block_w_size; l_size= self.block_l_size
    img = self.target.copy()
    blocks = []
    #calculate the new measures of the images (notice that they have to be natural numbers)
    b_w_size, b_l_size = self.width // w_size , self.length // l_size
    for iw in range(w_size):
        for il in range(l_size):
            x = b_w_size * iw
            y = b_l_size * il
            block = img[int(x):int(x+b_w_size), int(y):int(y+b_l_size)]
            #With this portion we create a new object GA_Multi as a new problem
            new_GA = GA_Multi(Image.fromarray(block),next_block_size[0],next_block_size[1])
            #Adding all the "SubProblems" to a list
            blocks.append(new_GA)
    return blocks
```

With this method, using the properties of *block_w_size* and *block_l_size*, it allows us to subdivide the problem into other problems by dividing the image of the problem that calls the method into (*block_w_size* * *block_l_size*) images encapsulated in objects of type GA_Multi



- ***Merge_population(population_subBlocks)***

```python
def merge_population(self,population_subBlocks):
    new_HoF_population = []
    for cl_individual in zip(*population_subBlocks):
        new_HoF_population.append(self.concat_blocks(cl_individual))
    return new_HoF_population
```

This method receive a list of finals populations of Sub Problems called “population_subBlocks”, and return a General Problem merging the populations. To be able to do this, we unpack the “population_subBlocks” and create a zip with them to use the method “concat_blocks” with the list of individuals selected by the zip. The reason of that is because every list of populations refers to a part of the photo, so if we take a individual per population and we chain them in order, we will create a individual of the *General Problem*. This *concat_blocks(blocks)* performs this action. With the list of pieces of the individuals given as an argument, it chains the pieces of them images and then creates an object *Individual* with this new image.

```python
def concat_blocks(self,blocks):
    w_size=self.block_w_size; l_size= self.block_l_size
    rows = []
    for li in range(l_size):
        num_block_row = li*w_size
        row_i = np.concatenate([*map(lambda x: x.array,blocks[num_block_row:num_block_row+w_size])], axis=1)
        rows.append(row_i)
    sol_concat = np.concatenate(rows,axis=0)
    if(sol_concat.shape != self.target.shape): #In the case where there is any problem with the resizment after the merge
        sol_concat = Image.fromarray(sol_concat.astype(np.uint8)).resize((self.length,self.width))
    candidate = Individual(self.length,self.width,img = sol_concat)
    return candidate
#
```

Multi_EA

Finally, after having explained all the parts, we can explain the evolutionary algorithm of this proposal. This algorithm is defined in the "*runGA_Multiprocessing()*" method, where using the "**multiprocessing**" library it generates 4 different processing stages: We start with a "*GA_Multi*" object as *General Problem* which will be divided into nine *Secondary Problems* and they will be assigned to parallel processes the task of performing the "*runGA_Multi()*" method explained above with the parameters we select. Here appears the true utility of the "return_dict", since this dictionary will be associated with a process handler through the "*multiprocessing.Manager().dict()*" function. In this way, we will alter the values of the dictionary to give you the corresponding values, associating the keys to the indexes of the processes (value "i" for the parameter *index_procex*). With *p.start()* we make the processes start, but we can't leave it there. We will put all the processes in a list, to iterate it and make a *p.join()* that will cause it not to continue with the code until the list of processes has finished. As each process ends, we will save the results obtained by the dictionary in a list ("result_b9" in this case)

```
def runGA_Multiprocessing(self):


######################BLOCKS 9 START######################
    blocks_9 = self.div_image_blocks((2,2))
    print("*** "*5 + "blocks_9"+" *** "*5)
    return_dict = multiprocessing.Manager().dict()
    process = []
    result_b9 = []
    for i in range(9):
        p = multiprocessing.Process(target=blocks_9[i].runGA_Multi,args=(i,return_dict,[],70,5000,))
        p.start()
        process.append(p)
    for i,pr in enumerate(process):
        pr.join()
        result_b9.append(return_dict[i])
        print(str(i)+"_b9")
    t=0
    '''
    while(t<9):
        result_b9[t][0].image.save("POP_"+str(t)+" fitness_"+str(result_b9[t][0].fitness)+ " results_b9_EndSolution"+".png")
        t = t+1
    '''

    print("*** "*5 + "blocks_9_END"+" *** "*5)
######################BLOCKS 9 END######################
```

Thus, we would already have the first stage completed. Now it would be time to make the unification of the pieces to generate a population that identifies the *General Problem*. So we will use the "*merge_population*" method by giving it as a parameter the generated list of the processes. With this, we could already take out an individual. As we already knew before, the best individual is in the first position on the list of the population, so to see how it turned out we keep that value as an image.

```
######################BLOCKS 9 END######################

    population_f9 = self.merge_population(result_b9)

    candidate = population_f9[0]
    candidate.image.save("pictures/multi/Colour/best_sol/_fitness_"+str(candidate.fitness)+ "Block9_EndSolution"+".png")



#######################FIRST MERGE START#######################
```

As expected, the edges that join the images will be quite noticeable. Therefore, we will proceed to a second stage of search, where in this case we will be trying to solve the *General Problem* directly (so these images will be saved. Also, notice that we now have an initial population to give as an input parameter, in addition to the "*string_block*" that is now "*_b9*", which will imply that the solutions will be saved)

```
########################FIRST MERGE START########################
    print("/// "*5 + "Population1"+" /// "*5)
    population1_index = self.runGA_Multi(0,{},population_f9,80,5000,string_block="_b9")
    print("/// "*5 + "Population1_END"+" /// "*5)

    population1 = population1_index[0]
    candidate = population1[0]
    candidate.image.save("pictures/multi/Colour/best_sol/_fitness_"+str(candidate.fitness)+ "_Pop1_EndSolution"+".png")
########################FIRST MERGE END########################
```

After completing this search, we decided to generate another stage of Secondary Problems, where in this case we divide it into 4 equal parts to try to improve certain parts again separately (which will give better results) but without dividing it so much so that it takes into account a slightly larger dimension and thus try to resemble the structure of the image more.

```
########################BLOCKS 4 START########################
    pop1 = []
    div_pop1 = [*map(lambda  i: i.div_in_blocks((2,2)),population1)]
    for block in zip(*div_pop1):
        pop1.append(block)
    self.block_w_size,self.block_l_size=(2,2)  #resize the blocks measures
    blocks_4 = self.div_image_blocks((2,2))

    print("*** "*5 + "blocks_4"+" *** "*5)

    return_dict = multiprocessing.Manager().dict()
    result_b4 = []
    process = []
    for i in range(4):
        p = multiprocessing.Process(target=blocks_4[i].runGA_Multi,args=(i,return_dict,[*pop1[i]],50,5000,))
        p.start()
        process.append(p)
    for i,pr in enumerate(process):
        pr.join()
        result_b4.append(return_dict[i])
        print(str(i)+"_b4")


    print("*** "*5 + "blocks_4_END"+" *** "*5)
########################BLOCKS 4 END########################


    population_f4 = self.merge_population(result_b4)
    candidate = population_f4[0]
    candidate.image.save("fitness_"+str(candidate.fitness)+ "Block4_EndSolution"+".png")


########################SECOND MERGE START########################
    print("/// "*5 + "Population2"+" /// "*5)
    population2_index = self.runGA_Multi(0,{},population_f4,30,10000,string_block="_b4_b9")
    population2 = population2_index[0]
    print("/// "*5 + "Population2_END"+" /// "*5)

    candidate = population2[0]
    candidate.image.save("pictures/multi/best/fitness_"+str(candidate.fitness)+ "_Pop2_EndSolution"+".png")
########################SECOND MERGE END########################

    bests_per_Generation_9 = population1[1]
    bests_per_Generation_4 = population2[1]
    return population2,bests_per_Generation_9,bests_per_Generation_4
```

Finally, we would have a last stage of *General Problem* where we would finally return the resulting population and the *best_pop_per_print* lists of the second and fourth stages. These lists could also be used as a population that would also allow to bring genetic material from other generations to increase exploration. With this it could allow us a greater variability of layers and changes in the parameters, because we can no longer only give you the last resulting population, but the "*Hall of Fame*" of the previous search.

In the main we would only have to create the object with the path of the image to be made and call the method.

```
if __name__ == '__main__':
    GA_Sol = GA_Multi("gioconda.png",3,3)
    GA_Sol.img.show() # type: ignore
    solution = GA_Sol.runGA_Multiprocessing()
```

7. Results of experiments

Describe in detail the results of your experiments. Use the best, average and worst fitness value charts. The results should be the averages of multiple runs of the experiment.

Because the processing times are very high

Even though we have not been able to do many iterations due to these long times, we have come to obtain really noticeable results. I have been able to verify that the best results are given with the first layer due to the large division that is generated.

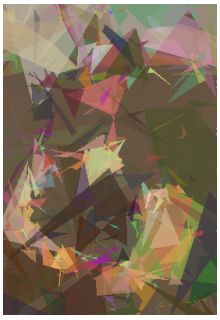
In one of these iterations, the first layer was defined by 9 divisions, with a population size of 70 individuals, and 5000 generations. This layer gave very good results. Comparing the results of the import only with GA, we could say it could be quite promising. I am mainly based on the fact that the image I use is much larger than the images I have been using previously, that my population is 30% smaller than what I usually use (100) and the number of generations is 5000 (which is a third of what I used to use). All this, within a more or less equal or even better processing time.

However, the second layer as a general problem with a population of 80, 5000 generations and using the unification of the previous individuals gave me very low variation. We started with a fitness and 32.71 and ended up at 32.19. This shows us a low convergence with a possible stagnation in the results. However, I think it seems to me a point in favor the fact of comparing results between these small variations in fitness and checking in the same way very small variations in the image. With this we could say that now the model agrees a little more with the given fitness.

After this layer we made another of 4 subdivisions, with a population of 50, a generation of 5000 and the previous population. Observing the beginning of the next layer of General Problem with 30 individuals and 10000 generations (with the intention of improving as much as possible) we find that the first best fitness is 31.58. This implies that there has been a loss of fitness throughout that layer, in addition to not finding an improvement in the image. This may be due to the fact that not always when improving the "subproblems" tend to substantially improve the general problem. Surely during that layer there have been no substantial changes, however they will have improved something to the sub-problem. However, by unifying the first candidate, the fact that they are the

best of each S*ub-Problem* does not imply that the unification of them will make it even better. Maybe those unions between them generate a certain discord in the resulting final structure. However, as we have observed, they do not generate a big change, although they do not generate an improvement after so much waiting time. In this last layer, practically nothing is improved, achieving a result of 31.56. Finally, we would have an average of 32.01 of fitness between the beginnings and ends of layers of general problems, which implies a great improvement taking into account the beginning we had.

To understand the comparison between the GA and the GA_Multi we have an image of the beginning of a GA population with fitness 85.62 and finally, after 15,000 generations with 100 individuals and without the possibility of selection between parents to be a descendant, a fitness of 45.81.

(Images from the folder GA/pictures/pictures1)

Finally, comment that although it seems that the fitness is better adjusted, we have cases that could tell us that it is not so well adjusted to the problem. For example, in one of the iterations I tried in stage 2 we found (as before) one of the best results of this iteration. However, although it does not seem to give bad results, it has a fitness of 32.65. In addition, in its last stage as a result, it returns the image to the right with a fitness of 31.99. However, in the attempt explained above we find that these results are worse, when the image shows many better results. Therefore, we could say that fitness can give a relatively intuitive idea, although a better option would have to be sought to calculate a better approximation to what a human being could understand as a result more similar to the image.


8. Summary

Summary of results and conclusions.

In this project I have tried to take a risk and try to take advantage of things I had never tried before. All this research has allowed me to understand a little more about that extensive field that is Artificial Intelligence and how even such a specific branch is capable of covering so many different fields. In addition, I have been able to learn how to use tools and libraries that I had never used together before, including image processing, array processing, multiprocessing, stochastic libraries along with image and graphic representation. It has also allowed me to understand the great difficulty that there is in an investigation process where at first it is not very clear where to go. In terms of the results, it is true that I would have liked to have been able to find a way to resemble the image even more in smaller times. In addition, I would have liked to have been able to do more tests and have been able to devote more time to this last approach of multiprogramming. It could also be a good way to give it a better solution by trying more alternatives related to ratios, or even implementing some kind of self-adaptive technique. In addition, at the presentation level I would also have liked to have implemented some type of interface in which the different solutions are viewed as I found in a publication called "Genetic Algorithms 2: Painting Vermeer" where they propose a genetic algorithm in which using pygame functions they are able to generate "the girl with a pearl Earring". Finally, it has also made me discover an application related to the selection of neural network models, which I would like to explore very soon.

Bibliography

[1,2] Devin Soni, *Introduction to Evolutionary Algorithms*, https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac, Last accessed: 29.03.2022 , [1,2] https://en.wikipedia.org/wiki/Evolutionary_algorithm, [1,2] https://en.wikipedia.org/wiki/Soft_computing , [1,2] https://en.wikipedia.org/wiki/Computational_intelligence [1,2] https://en.wikipedia.org/wiki/Evolutionary_computation [1,2] https://en.wikipedia.org/wiki/Metaheuristic [1,2] https://blog.oureducation.in/fuzzy-systems/#:~:text=Crisp%20logic%20(crisp)%20is%20the,but%20Ben%20was%20not%20punctual." [1,2] Artificial Intelligence A Modern Approach (4th Edition)

[3,5] Principal: Evolutionary Computation (FULL COURSE) Overview https://towardsdatascience.com/evolutionary-computation-full-course-overview-f4e421e945d9 , [3] https://www.researchgate.net/publication/216300863_A_history_of_evolutionary_computation, [3] https://link.springer.com/article/10.1007/s00521-020-04832-8#Tab1, [3] https://www.unedourense.es/noticias/noticia/algoritmos-irRfx [3] https://link.springer.com/chapter/10.1007/0-387-28356-0_5#:~:text=Genetic programming is a domain,of naturally occurring genetic operations

[3,4] Principal: Evolutionary algorithms and their applications to engineering problems https://link.springer.com/article/10.1007/s00521-020-04832-8 [4]https://www.sciencedirect.com/science/article/pii/S010689041631175X , [4] https://www.sciencedirect.com/science/article/abs/pii/S0278612516300693), [4] https://www.sciencedirect.com/science/article/abs/pii/S1568494616304999), [4] https://ieeexplore.ieee.org/abstract/document/7782360 [4] https://ieeexplore.ieee.org/abstract/document/7378948) [5] Principal; *A True Genetic Algorithm for Image Recreation — Painting the Mona Lisa* https://medium.com/@sebastian.charmot/genetic-algorithm-for-image-recreation-4ca546454aaa [8] https://cosmiccoding.com.au/tutorials/genetic_part_two/ [8] https://www.pygame.org/docs/ref/draw.html#pygame.draw.polygon [8] https://sainivedh.medium.com/optimization-of-cnn-architecture-using-genetic-algorithm-for-image-classification-5c48f25dac9c