Genetic Algorithms

Introduction

In this paper I will talk about a particular family of algorithms because I think it is very interesting to understand how software that operates on large amounts of data and that are able to "understand" by themselves how to solve a problem are developed. To do this, we need to deepen the field of study relating to artificial intelligence. By doing so, you can find different algorithms that use different methodologies to achieve their purpose and, among them, I was fascinated by genetic algorithms. Initially the question arises, what are these algorithms? A genetic algorithm is a heuristic search and optimization algorithm, inspired by Charles Darwin's principle of natural selection, which regulates biological evolution. Genetic algorithms operate in the field of artificial intelligence and were created with the idea of solving problems that, if solved with a "brute force" algorithm, would take too long. A simple example to illustrate this is the following problem: guess an integer from 1 to 1 billion. A brute force algorithm would try every single possible solution (therefore: one, two, three, ...) until it could solve the question; but what if it were possible to use a method that allowed us to know if we are close to the solution, rather than far or very far? The latter method would certainly be faster since, once the information on the proximity or distance of the number from the solution has been obtained, it would be easier to find another number that is closer to the one sought, until it is identified.

Historical background

The beginnings of these theories are already to be found in the first half of the twentieth century. The keystone, in the case of genetic algorithms, was the incredible development of digital computers in the 1960s. The availability of these machines for universities and laboratories grew exponentially starting from those years, mainly thanks to an ever lower cost of hardware. With this new tool, many research groups were attracted by the idea of being able to use algorithms inspired by biology as a solution to the new complex problems that were being proposed to the new machines. Over the years, three separate groups were distinguished, even geographically, to which today we owe the greatest contribution to the definition of these algorithms.

1. The first is formed at the University of Berlin, where Rechenberg and Schwefel develop their ideas regarding the use of evolutionary algorithms to solve complex parametric optimization problems with real coefficients. From this study a family of algorithms called “Evolution Strategies” will be formed.

2. The second was formed at the University of California, Los Angeles, where Lawrence Fogel, considered a pioneer in this field, hypothesized a new use of these evolutionary techniques for artificial intelligence. Fogel used these algorithms to increase the efficiency of some finite state machines (agents) within an evolutionary framework called "Evolutionary Programming".

3. The third group was formed in the same years at the University of Michigan where Holland saw the latter as powerful tools for implementing particularly robust adaptive systems and above all capable of dealing with a variable environment, with uncertain parameters.

The latter type algorithms will be those discussed and explored in this paper. It should be remembered that the birth of these algorithms is historically attributed to the aforementioned John Henry Holland with the publication, in 1975, of the book "Adaptation in Natural and Artificial Systems", still considered a cornerstone of the entire discipline.

Why are genetic algorithms used?

To simply answer this question, a very significant example will be given. Imagine having a group of monkeys (and an infinite amount of time) casually pressing keys on typewriters, with the intent of composing all of Shakespeare's works. If these monkeys had started such work immediately after the Big Bang explosion, nowadays it would be extremely unlikely to have even the complete draft of "Hamlet"!

Let's do some calculations.
Suppose a monkey has the ability to press one character at a time among the 27 available on the typewriter, including the space bar.
*If you wanted to get the phrase "To be or not to be that is the question", without punctuation, you would have the probability of 1/27 to hit the first letter correctly. The probability of pressing the second key correctly as well would be 1 / (27 * 27) and so on for the entire length of the sentence.*
Therefore, in conclusion, the probability of correctly writing the entire sentence would be 1 in 66,555,937,033,867,822,607,895,549,241,096,482,953,017,615,834,735,226,163, or $(1/27)^{39}$.
Even if the activity carried out by the monkey were simulated by a computer, it would take about 9,719,096,182,010,563,073,125,591,133,903,305,625,605,017 years to achieve the same result. To provide a comparison, scientists believe that the universe has existed for about 13,750,000,000 years.

A procedure of this type can be assimilated to a brute force algorithm, while, to obtain a result in a short time, it would be advisable to use a genetic algorithm.

How a genetic algorithm works?

To work, a genetic algorithm needs three techniques: Selection, Inheritance and Variation. The idea is to have elements created randomly, called "population", which will be analyzed by the computer and, through a "fitness" function, will be assigned a score. Here the first technique comes into play, that of Selection. The Selection, as in nature, eliminates the weaker elements, which do not have a high enough fitness score, or which probably will not be able to lead to a solution of the problem. After the Selection has decimated the population, the Inheritance will take place, or the remaining elements will be combined in various ways, trying to obtain "children" elements that will presumably have inherited from the "fathers" the characteristics necessary to contribute to the solution of the problem . The simplest Hereditary technique is that of the "crossover" which, in fact, combines the DNA of the "fathers". Finally, the Variation will be implemented, also called Mutation, which operates, with a very slight probability, the change of some parameters of an element. This will make it possible to have a population made up of elements that, at the end of the

first cycle, are more likely to approach the solution of the problem. This process will have to be repeated until the set goal is achieved.

Fitness function, genotype and phenotype

The fitness function is an algorithm capable of assigning a score to each element of the population based on its quality. The fitness functions are different depending on the problem. Taking as an example the genetic algorithm capable of writing the phrase "To be or not to be that is the question", its fitness function will attribute a positive value to an element when it contains a correct letter in the correct position. The more letters are correct, the higher the fitness value will be. This will allow you to have elements with a high fitness value if and only if they come close to repeating the required phrase. This fitness function, however, is only useful for problems similar to this one, while for other types of questions a different function must be programmed. For example, if the problem involved rockets (SmartRockets) that have to reach a target, the fitness function would be about trajectory accuracy and speed. This means that the rocket with the highest fitness value will be the best in its population and is more likely to be able to pass on its "useful" genes to the next generation.

Within the computer memory, these rockets are encoded in a different way to that shown in the front end. This diversity allows a distinction to be made between genotype and phenotype. The genotype is DNA, or the expression, often binary, which is saved in the computer memory, different for each element of the population. The population is then displayed in a different way to allow a greater understanding of the progress of the algorithm. This is the phenotype.

How to program a genetic algorithm?

To program a genetic algorithm, one must first find a problem in which it can be best exploited. In this thesis the problem related to the phrase “To be or not to be that is the question” will be analyzed. The code was developed in Java on NetBeans software. The editor from which the screenshots were taken is SublimeText; this choice is motivated by better readability.

- **Definition of the class and constructor:**

  We will define a class that we will call DNA; in it we will declare the matrix that will contain the genotype, which will be composed of several arrays of characters. In addition, the other global variables that will be used later by the program must also be declared. Once this is done, it will be appropriate to define the constructor with three parameters, which will be: the quantity of the population, the length of the sentence given in input and the mutation index.

  N.B.: In this example, the genotype and phenotype correspond as the first will be the set of characters on which the algorithm will operate, while the second will be composed of the same set of characters which, however, will be printed on the screen.

  Code:

```java
public class DNA {
    private char[][] genes;
    private int[] fitness;
    private char[] father1;
    private char[] father2;
    public double mutation_value;
    private int chars = 0;
    public boolean end = false;

    public DNA(int quantity, int length, double mut){
        end = false;
        chars = length;
        genes = new char[quantity][length];
        fitness = new int[quantity];
        mutation_value = mut;
        random();
    }

    /*...*/

}
```

The matrix called "`genes`" was designed to hold the data in the following order:

<-- length -->

^ quantity v	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
1	t	o		b	e		o	r		n	o	t		t	o		b	e
2	t		w	s	e		y	r		n	o	g	h	t	o		b	e
3	w	e		t	e		o	r		c	s	t		e	o	y		e
4	t			t	x		q	k			j	t	i	o			i	z

The array of integers called "`fitness`" will be used by the fitness function to assign a value to each sentence.
It can be seen that the constructor has not only the function of initializing the variables, but also that of creating the entire population and assigning the characters to the matrix using the function "`random()`".

Code:

```
21    private void random(){
22        String alphabet = "abcdefghijklmnopqrstuvwxyz ";
23        for(int k = 0; k < genes.length; k++){
24            for(int i = 0; i < genes[0].length; i++){
25                genes[k][i] = alphabet.charAt( (int) (Math.random() * 27));
26            }
27            fitness[k] = 0;
28        }
29    }
```

The above function uses two nested "`for`" loops to scan the array and assign randomly generated characters to its elements. The value 0 is then assigned to each element of the "`fitness`" array.

- **Fitness function:**

The task of the fitness function is to assign a numerical value from 0 to the number corresponding to the length of the sentence given in input, which value will be an indicator of the goodness of an element. Furthermore, it will be the task of this function to stop the execution of the program in the event that an element of the population is successful in the intent of the software, that is to discover the sentence given in input.

Code:

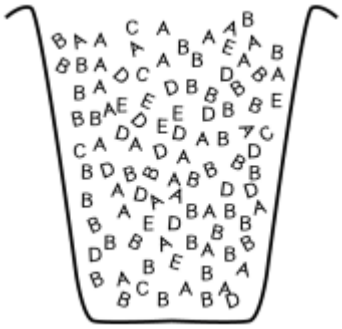
```
32    public void fitness_function(String solution){
33        for(int i = 0; i < genes.length; i++)    fitness[i]=0;
34        for(int k = 0; k < genes.length; k++){
35            for(int i = 0; i < genes[0].length; i++){
36                if(genes[k][i] == solution.charAt(i)){
37                    fitness[k]++;
38                    if(fitness[k] == genes[0].length){
39                        end = true;
40                        return;
41                    }
42                }
43            }
44        }
45    }
```

The above function, before attributing the values to the elements, assigns a value of 0 to the entire "`fitness`" array, since, if this were not done, the fitness values of elements of different generations would be added. Once this is done, the fitness value of an element is increased by 1 every time a letter of the element is right and in the correct place, compared with the sentence given in input.

- **Selection, Inheritance and Mutation functions:**

  These are the functions that make up the "body" of the algorithm as they perform the essential procedures of a genetic algorithm.

  Before analyzing the code, it is necessary to understand the functioning of the "wheel of fortune". The "wheel of fortune" is used to choose which element should father some elements of the new generation. To make this choice, you have to "spin the wheel" and take as a father the element on which our "roulette" will stop. Not all elements, however, will have the same probability of being chosen, since the more an element has a high fitness value, the more times it will have to appear on the wheel. To simplify this concept further, the term "mating pool" is used. This term indicates a swimming pool in which two elements to be coupled are fished. The more times there is a certain element in the pool, the more likely it is to be caught.



Code:

```
47    public void selection(){
48        int ftot = 0, free = 0;
49
50        for(int k = 0; k < genes.length; k++){
51            ftot+= fitness[k];
52        }
53
54        int[]  matingpool = new int[ftot];
55        for(int k = 0; k < genes.length; k++){
56            for(int i = 0; i < fitness[k]; i++, free++){
57                matingpool[free] = k;
58            }
59        }
60
61        for(int i = 0; i < genes.length-1; i+=2){
62            father1 = genes[matingpool[(int) (Math.random() * ftot)]];
63            father2 = genes[matingpool[(int) (Math.random() * ftot)]];
64            while(father1 == father2)
65                father2 = genes[matingpool[(int) (Math.random() * ftot)]];
66            inheritance(i);
67        }
68    }
```

In the above code, the mating pool method is represented by two nested "`for`" loops that insert an element in the "`matingpool`" array as many times as its fitness value is high. In addition, fathers are chosen, who must be different from each other.

Code:

```
71    public void inheritance(int i){
72        int random =  (int) (Math.random() * chars-1);
73        for(int k = 0; k < genes[0].length-1; k++){
74            if(k <= random){
75                genes[i][k] = father1[k];
76                genes[i+1][k] = father2[k];
77            }else{
78                genes[i][k] = father2[k];
79                genes[i+1][k] = father1[k];
80            }
81
82        }
83        mutation(genes[i]);
84        mutation(genes[i+1]);
85    }
```

The inheritance function receives as a parameter an integer i which determines the indices relating to the element of the new generation (`i`) and its next (`i + 1`): these elements will be children of the two chosen fathers. Each pair of fathers will give birth to two elements. This method of reproduction is called “cross over” as, by dividing two fathers in the same index, two children can be obtained. The first will be composed of the first part of the first father and the second part of the second father, while the second son will be composed of the first part of the second father and the second part of the first father.

Fathers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	t	o		b	e		o	r		n	o	g	h	t	o		b	e
2	t		w	s	e		y	r		n	o	t		t	o		b	e

Children

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	t	o		b	e		o	r		n	o	t		t	o		b	e
2	t		w	s	e		y	r		n	o	g	h	t	o		b	e

Once the "cross over" has been carried out, the mutation functions are called which have the purpose of varying a single part of the genotype, that is only one character.

Code:

```
87      public void mutation(char[] gene){
88          if(mutation_value > Math.random()){
89              String alphabet = "abcdefghijklmnopqrstuvwxyz ";
90              gene[(int) (Math.random() * chars)] =  alphabet.charAt( (int) (Math.random() * 27));
91          }
92      }
```

The higher the mutation index, the greater the probability of having a mutation.

- **Writing the main function:**
  To make the program work, the constructor and functions must be called in the appropriate order within the main.

  Code:

```
94    public static void main(String args[]){
95        DNA population = new DNA(population_elements, phrase.length(), mutation, attempts);
96        for(int i = 0; i < iterations; i++){
97            population.fitness_function(phrase);
98            if(population.getEnd()){
99                System.out.println("Find! Generation number" + i);
100               break;
101           }
102           population.selection();
103       }
104       System.out.println("End of execution");
105   }
```

  For a correct execution of the program, it is recommended to use a minimum population of 25 elements, a mutation index less than 0.3 and a maximum sentence length of 25 characters.

This example was used as it allows you to easily understand which instructions are executed by the various functions.

Other examples and videos regarding genetic algorithms are:

- SmartRockets:
  http://www.blprnt.com/smartrockets/

- Flexible Muscle-Based Locomotion for Bipedal Creatures:
  https://www.youtube.com/watch?v=pgaEE27nsQw&

- Locomotion Skills for Simulated Quadrupeds:
  https://www.youtube.com/watch?v=dRthdBr46cs&

Applications of genetic algorithms

Although the topics covered so far may seem essentially academic, there are many problems that can be solved through these algorithms. This derives from the generality of the tools used, from the simplicity of the rules used, from the effort to obtain the maximum degree of abstraction of the problem and from the ability that systems then have to evolve based on the imposed algorithms.

Some fields in which genetic algorithms operate are:

- *Planning*, which includes all those problems in which it is required to choose how to use a finite set of resources favoring the lowest cost or highest performance one: this domain includes the route planning of a fleet of vehicles, the problem of transport, the planning of the trajectory of a robot, the planning of the production of an industrial plant, the drawing up of schedules, the determination of the optimal load of a means of transport and so on.

- *Design*, which includes all those problems in which it is required to determine an optimal arrangement of elements (electronic or mechanical components, architectural elements, etc.) in order to satisfy a series of functional, aesthetic and robustness requirements: they fall into this domain, therefore, various problems of designing electronic circuits, engineering structures, designing information systems, and so on.

- • *Simulation and identification*: given a design or model of a system, it must be determined how that system will behave; in some cases this must be done because you are not sure of the behavior of the system, other times the behavior is known but you want to evaluate the accuracy of a model. The systems studied can be chemical (determination of the three-dimensional structure of a protein, the equilibrium of a chemical reaction), economic (simulation of the dynamics of competition in a market economy), medical etc.

- *Control*, which includes all those problems in which it is required to establish a control strategy for a given system.

- *Classification, modeling and automatic learning*, where, starting from a set of observations, it is required to build a model of the underlying phenomenon: depending on the case, this model can consist in the simple determination of whether each observation belongs to one of two or more classes, or in the construction (or machine learning) of a more or less complex model, often to be used for forecasting purposes. Data mining is also part of this domain, which consists in discovering invisible regularities "to the naked eye" in the midst of huge amounts of data.

Of course, the boundaries between these five application domains are not clearly defined and the domains themselves may in some cases overlap to some extent. However, it will be noted that they include a whole series of problems of great economic importance, as well as of enormous difficulty.

Conclusions

This paper was written with the aim of making learning and studying genetic algorithms simple and engaging. The basics of computational thinking related to these algorithms were discussed, touching on both theoretical and practical aspects, but without going into sectoral languages or technicalities, and thus giving a general overview of this sector of artificial intelligence. Some portions of the code were also illustrated and explained, concerning the main functions of the genetic algorithm I programmed and presented. Furthermore, with this paper we want to demonstrate how much artificial intelligence is present in many software used daily.

Bibliography

- **Daniel Shiffma**n – The Nature of Code
- Registration of a university lecture by **Daniel Shiffman**:
  - https://www.youtube.com/watch?v=6l6b78Y4V7Y&t=221s&list=LLWCMkz2zIBbtCDWtoVrcDnA&index=9
- **Carlo Gotti** – Use of genetic algorithms in bioengineering: application to the identification of cardiac models
- **Marco Di Gennaro** – Simulated evolution: artificial life and genetic algorithms
- **Andrea G. B. Tettamanzi** – Evolutionary algorithms: concepts and applications