Bulletin of the *Transilvania* University of Braşov • Vol 10(59), No. 2 - 2017
Series III: Mathematics, Informatics, Physics, 143-154

IMPLEMENTING SMART APPLICATIONS USING GENETIC ALGORITHMS

Alexandra BĂICOIANU [1]

Abstract

The aim of this paper is to present the logic and implementation methods of the solution in view of a number of interesting and practical applications of genetic algorithms. It is beyond any doubt that the fields which allow the application of genetic algorithms are widely varied, ranging from computer gaming to automotive design or robotics. Nevertheless, in this paper we focused our attention on the following list of subjects which involved the use of genetic algorithms: the evolution of a given string, the creation of a logic agent based on neural networks and genetic algorithms for the Pac-man game and the evolution of genetic programming in the case of a target given picture.

Once acquainted with the theory of evolution, the understanding of basic principles of these algorithms is quite easy. When the theoretical mechanisms which make it possible for such an algorithm to function have been understood, passing from the application of the "Hello, World!" type of the evolutionary algorithms to the applications which were of most interest to us was just a step away. Such evolutionary techniques might prove of great use to specialists such as engineers and scientists who work in various fields of knowledge and who might be at their first usage of genetic algorithms in specific applications. At the same time, such applications may be of use to many other individuals who are becoming increasingly acquainted with the topic of genetic algorithms.

2000 *Mathematics Subject Classification:* 92D15, 91A22, 97R40, 68N19, 68U10, 68U15, 68T42.

Key words: genetic algorithms, optimization, evolution, Pac-man agent, image reconstruction.

1 Introduction

Science is the direct product of human curiosity to understand the mechanisms according to which the natural world works and to gain control over them.

[1]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: a.baicoianu@unitbv.ro

The great diversity of the world as we now know it, with numerous species of creatures, with different races of people who have adapted to their specific living environment, characterized by particular forms of ecological balance, is the direct result of an experiment which lasted three billion years, known under the name of evolution. When taking a closer look at the natural world around, we notice that the great complexity and ability to adapt of all creatures living on Earth is the consequence of a process of continuous improvement and combination of the genetic material, over an extended amount of time. Taken on the whole, genetic algorithms act as evolution simulators, regardless of which kind. Nevertheless, in the majority of cases, genetic algorithms are mere methods of probabilistic optimization, based on the guiding principles of evolution.

The setback of the more traditional search and optimization methods is that they are too slow at identifying solutions within a highly complex search area, even when they are implemented in supercomputers. The genetic algorithm, on the other hand, offers the benefit of being a powerful search method which only needs little information in order to perform an effective search in a rather large and complicated search space. In contrast to most search algorithms which take a single focus point, the genetic search progresses through a mass of such focus points.

The genetic algorithm has the particularity of coding parameters of the search space as binary strings of finite length. It uses an entire population of strings which are randomly initialized and which will evolve to the next generation by means of selection, crossover and mutation, namely specific genetic operators. The fitness function acts as an assessment technique for the quality of the string-coded solutions. Selection makes it possible for strings with a higher degree of fitness to achieve a higher degree of probability in the next generation. The function of Crossover is to merge two parents by swapping various parts of their strings, beginning at an arbitrary crossover point. This accounts for new solutions which possess strong qualities from both parents. In order to make sure that the genetic algorithm does not converge too soon, mutation exploits new regions of the search space and flips single bits in a separate string. The genetic algorithm is inclined to choose the fittest solution by focusing the search around the areas which allow access to fitter structures, which finally translate into more accurate solutions to problems proposed.

It is not easy to uncover an optimal parameter setting which works for a given problem. It is of uppermost importance here to determine such factors as robust parameter settings for population size, encoding, criteria used in selection, probabilities with genetic operator and the assessment or fitness techniques for normalization. It was this precise criterion, of searching for a "successful combination of parameters" which prompted us to consider some practical issues in order to find an intelligent solution which involves genetic algorithms.

Next, we will focus on explaining which issues concerned us, how we developed the solution of genetic algorithms and which were the steps we took in developing our applications. In the end, we will comment on the results obtained and will

also debate on the choice of meta-parameters.

2 Case 1: "Hello, World!" - A simple evolutionary algorithm

In this section, we intend to evolve any string from random garbage. This first example serves as evidence that there is nothing which can prevent us from using the technique provided by the genetic algorithms for absolutely classical and basic problems. As it is well known by many, the "Hello, World!" program is traditionally used in order to introduce novice programmers to a programming language. As a consequence, we will evolve even the "Hello, World!" string.

First, we shall define our starting point and end goal. We have an input file "Input.txt" where we have the target, first let us consider a random string: "loerm ipsum dolor sit amet loerm ipsum dolor sit amet loerm ipsum dolor sit amet". Our evolutionary algorithm is implemented in C/C++ like a usual Console Application and it will start with all symbols (32 to 122 in ASCII code), which we can view as the DNA of our "organism". We define this entity like a "candidate" with the structure:

```
struct candidate
{
        double fitness;
        vector <bool> dna;
};
```

It will then randomly mutate some of the DNA, and judge the new mutated string's fitness. The way we determine fitness is probably the most difficult part of any evolutionary algorithm. We considered:

```
{
  double fit_val = 0;
        for (int i = 0; i < source.size(); i++)
        {
                if (source[i] != target[i])
                        fit_val++;
        }
        fit_val = 1 - (fit_val / (double)source.size());
        return fit_val;
}
```

Fortunately, there is an accessible option for achieving this with strings. Thus, it is necessary to record the value of each character in the mutated string and compare it to the same character in the target string. This process is called the distance between two characters. Next, all noticed differences will be added up, leading to a single value which is the fitness of the string. 0 is the perfect fitness and this value indicates that both strings are the same, whereas a fitness of 1 means that a character is off by one. For example, the strings "Hfllo" or "Hdllo" have a fitness of 1. Thus, the greater the fitness value, the weaker the fit is.

All the classical stages of genetic algorithms were implemented as "standard" [1, 2], and for the sake of simplification, we will only present the selection stage, which uses as parameter an initial population of 100 candidates, which is a rank number of 30 (values are obtained by means of multiple tests performed on values

and calibrations). This method returns the index of the chosen parent in view of the selection for the next generation.

```
int selection(candidate *population)
{
        int t = (rank_number*(rank_number + 1)) / 2;
        int p = rand_between(0, t), s = 0;
        for (int i = 0; i < rank_number; i++)
        {
                s += population[i].fitness;
                if (s >= p)
                {
                        return i;
                }
        }
        return 0;
}
```

Other parameters used in developing the other stages of the algorithm are: "stop_percentage = 1" the "perfection" percentage chosen, "mutation_rate = 0.005" the mutation quotient, "total_fitness = 0" the initial value of fitness, "crossover_rate = 0.85" the crossover quotient.

An intermediary stage for the application is presented in Figure 1. It can be noticed that in the 40th generation we already have some "inferred" letters from the string we have proposed. The fitness is calculated for every specific generation and it represents the "quality" of the specific generation of chromosomes. The end

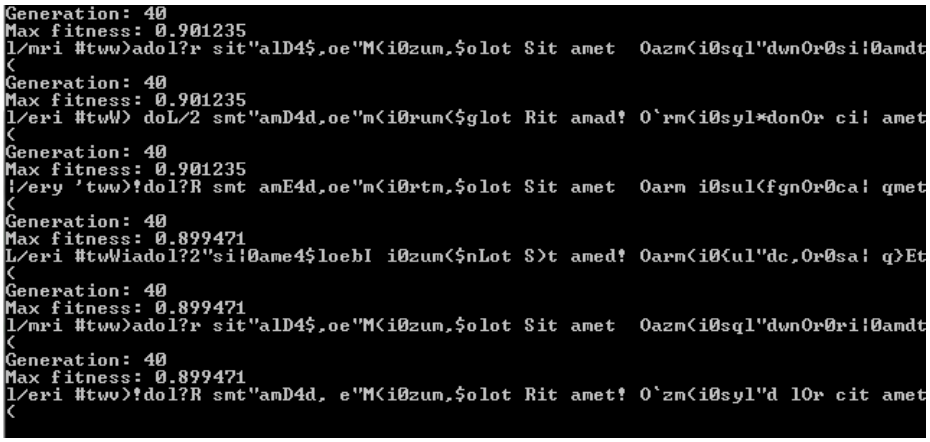

Figure 1: 40th generation - An intermediary stage

of the algorithm is presented in Figure 2. It can be noticed that the required string was entirely generated, and the fitness was balanced at the final iterations close to value 1. The final execution time is of around 90 seconds. If we are to consider the classical string "Hello, World!", the last generations from the evolution are depicted in Figure 3, the execution time being of approximately 2 seconds in this case.

We add the fact that each source implementation has unit tests to go along with the source code. For the sake of next comparisons, we also keep track of the number of generations and of the elapsed time.

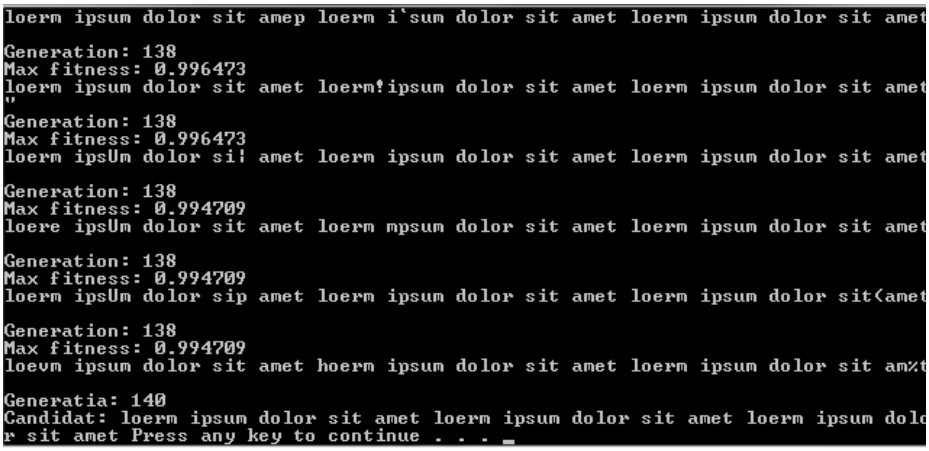

Figure 2: The final iteration, generation 140

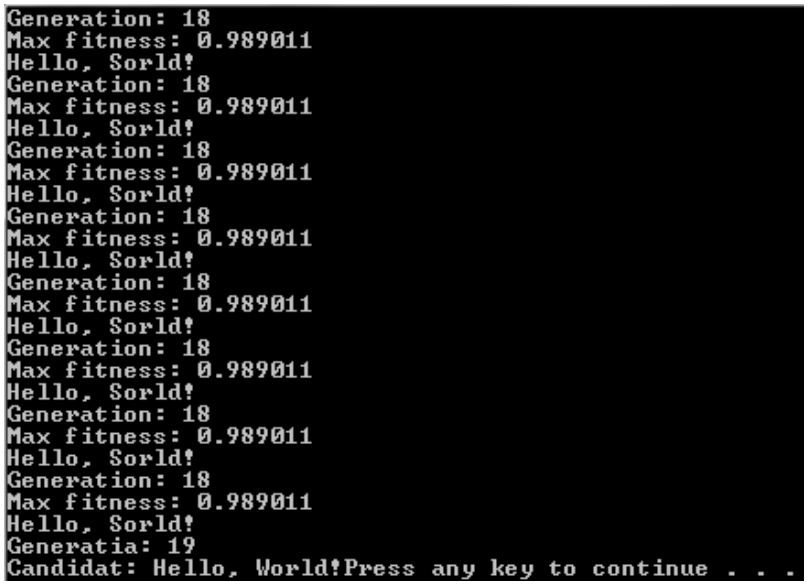

Figure 3: "Hello, World!" string evolution, generation 19

3 Case 2: Genetic programming evolution of a target given picture

The fact of having the basic structure of a genetic algorithm was a step forward toward an even more challenging application: when receiving a "target" image, the algorithm should generate rectangles of different sizes, colors and degrees of transparency, so that when overlapped they form an image which resembles as much as possible the image received as input [4]. The first functional version of the program was immediately generated, but it is quite obvious that this was inefficient and poorly adjusted. When speaking about adjustment we mean both the various constants specific to genetic algorithms, such as: PopulationSize, MutationRate, MutationSize, CrossoverRate and others, and to mutation methods, crossover and calculation function of the fitness. The DNA of a given individual (which is practically an image) from the entire population of a specific generation includes information about various coordinates, the color and transparency of each rectangle which forms the specific individual, leading the starting image (the 1st generation) to look like in Figure 4. After 1000 generations (evolution which

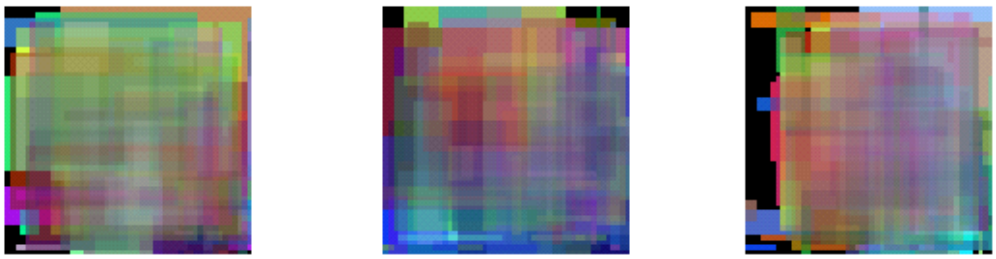

Figure 4: Starting images - 1st generation

occurs after 10-12 seconds), the best individual from the entire population (of 200 individuals) looks like this (left) and holds as a target image (right), Figure 5. In

Figure 5: Images at 1000-nd generation

a left to right sequence : 20, 200, 1000, 10000, etc., see Figure 6.

The selection technique used (the best match for this algorithm) is called the Roulette Wheel Selection Method [1, 5]. The DNA crossover in the two parents

Figure 6: Different stages

selected is accomplished by means of the Uniform Crossover method, which led to the best results and converged much faster than the One-point or Multi-point Crossover methods. For mutation we have chosen the easiest method, by means of which each DNA rectangle was randomly moved, based on the MutationSize and the MutationRate factors, independently from other rectangles (see "mutate" method implementation). For the fitness function, we have calculated the normalized sum of the Euclidean distances between the color of every pixel form the image generated and the color of the corresponding pixel from the target image (see "fitness" method implementation). Here, the most appropriate color space proved to be not RGB but Lab [6], where the difference between the 2 colors should represent the best the perception of the human eye.

```
vector<dreptunghi> mutate(vector<dreptunghi> source)
{
        double p = 0;
        int cul = 255 * mutation_size, alfa = mutation_size, coord_x = w* mutation_size
            , coord_y = h* mutation_size;
        for (int i = 0; i < source.size(); i++)
        {
                p = (double)rand_between(0, 1000);
                if (mutation_rate * 1000.0 >= p)
                {
                        dreptunghi drept = source[i];
                        drept.r = rand_between(max((int)drept.r - cul, 0), min((int)
                            drept.r + cul, 255));
                        /*
                        * the same for g, b
                        */
                        drept.a = rand_between((int)max(drept.a - alfa, 0.2), (int)min(
                            drept.b + alfa, 1.0));
                        drept.pt1.x = rand_between(max((int)drept.pt1.x - coord_x, 0),
                            min((int)drept.pt1.x + coord_x, w - 1));
                        drept.pt1.y = rand_between(max((int)drept.pt1.y - coord_y, 0),
                            min((int)drept.pt1.y + coord_y, h - 1));
                        /*
                        * the same for drept.pt2.x, drept.pt2.y
                        */
                        source[i] = drept;
                }
        }
        return source;
}

double fitness(vector<dreptunghi> source)
{
        double fit_val = 0;
        represent(source);
        double dr, dg, db, dist;
        for (int y = 0; y < h; y++)
        {
                for (int x = 0; x < w; x++)
                {
                        imagine pxs = im[y][x], pxt = target[y][x];
                        dr = pxs.red - pxt.red;
                        dg = pxs.green - pxt.green;
                        db = pxs.blue - pxt.blue;
                        dist = sqrt(dr*dr + db*db + dg*dg);
```

```
                        fit_val += dist;
                }
        }
        fit_val = 1.0 - (fit_val / (442.0*(double)w*(double)h));
        return fit_val;
}
```

We add the fact that each source implementation has unit tests to go along with the source code. All the calibrations of the parameters belong to the author, the same for all the implementations. For genetic algorithms, finding a "suitable" combination of parameters is making the all difference between a very good solution and a moderate one. We used "opencv-3.2.0-vc14" for the complete implementation, the one with the representations of the images. The results that we had on the pictures we named are really closed to the original pictures, thing that can validate the meta-parameters we elect.

4 Case 3: A logic agent based on neural networks and genetic algorithm for Pac-man game

Pac-man is a very convoluted game even if at first sight it does not seem so. One of the reasons why the game was so popular is that it has many strategies that can be used in order to maximize the score. In this section we attempt to evolve a logic agent that uses a neural network provided with a beginner-level set of information, such as distances to the nearest ghosts and food. A genetic algorithm is used to balance the network's weights. Our idea came from [3] where we had the following entities:

- artificial neural network: 6 (input) - 10 (hidden layer) 1 (output/"decision factor");
- inputs: *the distance to the closest food instance, *the distance to the closest special food, *the distance to the closest ghost, *the state of the closest ghost {1,-1}, *the second closest ghost, *the ghost's status - {1,-1};
- activation function : tanh;
- the weights of the artificial neural network are calculated with genetic algorithms: *100 chromosomes, *50 generations, *mutation rate 0.1, *life span 10 (each entity is taking the control of the game 10 times), *death rate 20 (the number of the entity replaced each time at a new generation), *fitness : average of the scores of the entity.

Our proposed method is with the following characteristics:

- artificial neural network: 4-8-1 (at each step we calculate the decision factor for each of the 4 possible moves);
- inputs: * the distance to the closest food instance, * the distance to the closest special food, * the state of the closest ghost, *the second closest ghost (the distance to the ghosts are multiplied by (-1) if they were "afraid");

- activation function : tanh (here we tried with exponential function, but it did not work well because the negative values are really important);
- the weights of the artificial neural network are calculated with genetic algorithms: *200 chromosomes, *50 generations, *mutation rate 0.03, *crossover rate 0.8 (80%), *fitness : (the score of one entity) , *we used elitism (we kept the best entity from each generation), *for the selection of chromosomes that are chosen in the new generation, we used the Roulette Wheel selection method [2, 1].

The implementation is in C/C++ language and it is a regular Console Application project. The entities that are finishing the game during the learning session are saved in a text file. The artificial neural network is distinguish throw a function (output) that takes two parameters, the forts contains the inputs and the second contains the weights. The activation function is tanh.

```
double output(double *intrari,double*ponderi,const int *arh)
{
        double *c;
        c=new double[arh[1]];
        int p=0;
        for(int i=0;i<arh[1];i++){
                double s=0.0;
                for(int k=0;k<arh[0];k++,p++)
                {
                        s+=intrari[k]*ponderi[i+k*arh[1]];
                }
                c[i]=tanh(s/4);
        }

        double s=0.0;
        for(int i=0;i<arh[1];i++,p++)
                s+=c[i]*ponderi[p];

        return tanh(s/4);
}
```

The entities are represented like the lines of a matrix and in the crossover method the proper weights of an input are changed between the two chromosomes, only if the cross over probability is corresponding. In the "pacmanAlone" method, we choose the right position for Pac-man (we compare the output for all possible moves: up, down, left, right).

```
void pacmanAlone(double *&c,const int *arh)
{
        double sus=-2,dr=-2,jos=-2,st=-2;
        if (mat[pacman.poz_lin-1][pacman.poz_col]>=300)
        {
                int lin=pacman.poz_lin-1;
                int col=pacman.poz_col;
                sus=fct_calcul(c,lin,col,arh);
        }
        /*
         * the same for the other possible moves
         */
        double max=sus;
        if (dr>max) max=dr;
        if (jos>max) max=jos;
        if (st>max) max=st;

        if (max==sus && sus!=-2)
        {
                pacman.poz_lin-=1;
                pacman.graf_1='V';
                pacman.graf_2=179;
        }
        else
                if (max==dr && dr!=-2)
                {
```

```
                        pacman.poz_col+=1;
                        pacman.graf_1=60;
                        pacman.graf_2=45;
                        }
                else
                        /*
                        * ...
                        * the same for the other possible moves (right, left, down)
                        */
}
```

In first generation the agent does not seem to follow a certain pattern, it often gets stuck in a position, either in a corner or even in the middle of the maze. This error of movement happens because an input influence is too big. From the fifth generation the neural network starts to balance and the agent is now fluently moving, even if he still does not know how to maximize the score. These are the steps that can be observed and which the agent is trying to follow:

- First he is learning how to eat the energizer. At this point he is targeting the special food one after another, but unfortunately he will die after there are no more energizers.
- The second step he learns is how to avoid the ghosts that are in chase mode. The combination between targeting special food and running from ghosts gives the agent a medium score of 200 points. Still not enough to finish the game.
- In the end he will learn how to look for simple food and how to eat frightened ghosts if they are nearby. At this point Pac-man will rather sacrifice a life in order to get more points than run away to survive.

Optimal parameters for the genetic algorithm are a population of 50 individuals and a evolution process over 50 generations (minimum). At the 20-th generation, the medium score is approximately 250 points, and it will continue to rise to about 370 points at the 50-th generation. Because of the non-deterministic way the frightened ghosts run (they run to a random tile), an individual that managed to finish a game in the evolution process, may not finish the game a second time.

Each source implementation has unit tests to go along with the source code. The game has several options like: "play", "play best", "learning session" (a stage from "learning session" in Figure 7). The "play best" variant is the one that is reading all the chromosomes that finished the game, and the "play" variant is the one that is finishing after the first one, "learning session" is the one for learning for the agent.

5 Conclusions

Using all the information above, we can conclude that genetic algorithms are systems based on the supposed functioning mechanisms of life. They are very different from other classical optimization algorithms, because they use e encoding of parameters, instead of parameters themselves, they use fitness functions for optimization and probabilistic transactions functions instead of deterministic ones.

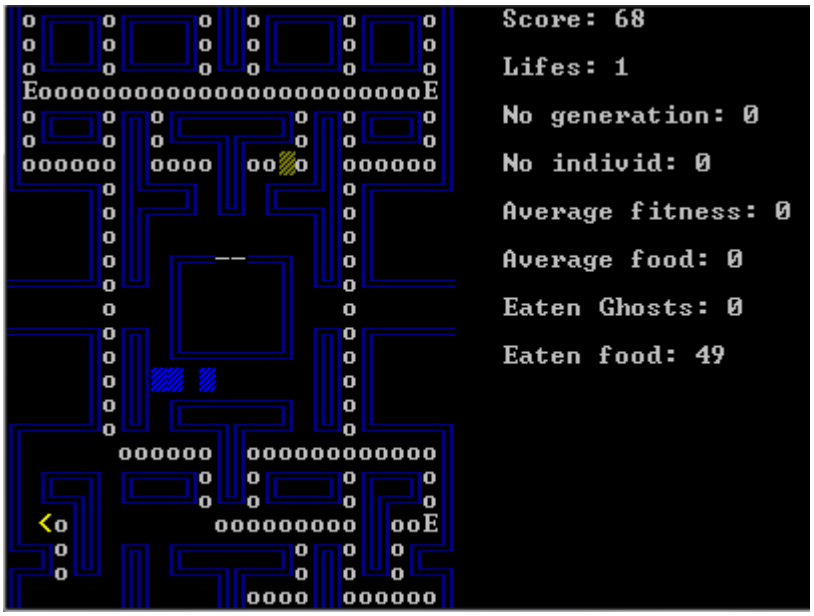

Figure 7: Pac-man in learning session

It is essential, though, to know that the operation of similar algorithms does not guarantee success. Nevertheless, such algorithms have proved to be of great efficiency and are being currently used in areas of interest such as the stock market, the organization of production or the programming process of assembly robots used in the automotive industry.

Since the topic of genetic algorithms can be applied in so many actual and diverse fields of knowledge, our aim is that the applications proposed in this article be integrated in a common framework which should deal with the intelligent resolution of a set of selected problems. Here, we mainly think about Rubik puzzle, $2D$ or $3D$ cutting/packing problems, single and multi-parameter problems, single and multi-objective problems, as well as some new problems of the resource constraint project scheduling type or the generation of a school timetable, based on genetic algorithms.

Characteristics such as the capacity of the algorithm to explore and exploit synchronously an increasing amount of theoretical rationale as well as the successful application to problems encountered in the real world strengthens the conclusion the genetic algorithms represents a powerful optimization technique.

References

[1] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1998.

[2] Mitchell, M., *An Introduction to Genetic Algorithms (Complex Adaptive Systems) Reprint Edition*, The MIT Press, 1998.

[3] Nicolescu, D., *Evolving a Minimum Input Neural Network Based Controller for the Pac-Man Agent*, Annals of the University of Craiova, Mathematics and Computer Science Series, Volume 37(1), 2010, Pages 76-85, ISSN: 1223-6934.

[4] Johansson, R., available at ***, `https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa`.

[5] Russel, S., Norvig, P., *Artificial Intelligence. A Modern Approach*, Prentice Hall, 3rd edition, 2010.

[6] ***, `https://en.wikipedia.org/wiki/Lab_color_space`