ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
ID: TG-20160212192336

AN OVERVIEW OF THE GENETIC ALGORITHM AND ITS USE FOR FINDING EXTREMA — WITH IMPLEMENTATIONS IN MATLAB

PRIKAZ GENETIČKOG ALGORITMA I NJEGOVA UPORABA ZA NALAŽENJE EKSTREMA — S IMPLEMENTACIJAMA U MATLABU

Jurica Hižak, Robert Logožar

Original scientific paper

Abstract: *The paper outlines the main concepts of the genetic algorithm (GA) in a combined, educational-scientific style. Every step of the GA is first motivated by its biological paragon, then mathematically formalized and explained on simple examples, and finally supported by implementations in MATLAB. Two programs that use GA for the illustrative cases of finding functions' extrema are shown. The authors conclude the paper by presenting the original use of GA in the Stochastic Iterated Prisoner Dilemma, which gave a new insight into this problem.*

Keywords: *genetic algorithm, fitness function, function extrema, stochastic iterative prisoner dilemma.*

Izvoran znanstveni rad

Sažetak: *Ovaj rad predstavlja glavne koncepte genetičkog algoritma (GA) u kombiniranom, edukativno-znanstvenom stilu. Svaki je korak GA isprva motiviran svojim biološkim uzorom, potom matematički formaliziran i objašnjen na jednostavnim primjerima te konačno potkrijepljen primjerima u MATLABU. Predstavljena su dva programa koja koriste GA za ilustrativne slučajeve nalaženja ekstrema funkcija. Autori zaključuju članak prikazom izvorne uporabe GA u stohastičkoj iterativnoj dilemi zatvorenika, što je dalo novi pogled na ovaj problem.*

Ključne riječi: *genetički algoritam, funkcija podobnosti, ekstremi funkcija, stohastička iterativna dilema zatvorenika.*

1. INTRODUCTION

The concept of evolution is omnipresent not only in nature and natural sciences, but can be also found in all other human activities. A good general definition of this notion, with explication in biology is found in [1]: "Evolution is, in effect, a method of searching among an enormous number of possibilities for 'solutions'. In biology, an enormous set of possibilities is the set of possible genetic sequences, and the desired 'solutions' are highly fit organisms—organisms which are capable of surviving and reproducing in their environments."

1.1. Genetics in computation

In computing (computer science), an *evolutionary algorithm* investigates the space of feasible solutions of some quantitative problem and—according to some prescribed criteria—finds the best among the selected ones. This process is then iterated until one of the obtained solutions is "good enough." Therefore, a computational procedure in an evolutionary algorithm is a process analogous to what is happening to live organisms in the natural evolution. The evolutionary algorithms are studied within the field of *evolutionary computing*.

Nowadays the evolutionary computing comprehends three big scientific fields: genetic algorithms, evolutionary strategies, and genetic programming. There are many different evolutionary algorithms, as well as several variations of each of them. However, the underlying idea of all of them is the same: given a *population* of solutions, high-quality solutions are more likely to be selected during the process of selection and reproduction. [2]

Thus, the basic idea and even the terminology of evolutionary algorithms follow the actions of the natural evolution. Of course, one might expect that there are also several technical differences between the two, which may depend on the particular field of the evolutionary computing. Nevertheless, the fundamental functioning of all evolutionary algorithms can be grasped through the following brief insight into the genetic algorithm.

For a given problem, the reduced set of potential solutions is generated in a random fashion. Every solution, or individual—also called (artificial) chromosome—contains artificial genes. Unlike the real chromosomes, which consists of the DNAs built of series of pairs of the four nucleotide bases (A-T, C-G, T-A, G-C), the artificial chromosomes are simple sequences of zeros and ones.[1]

[1] Since the nature is the inspiration for this whole area of computing, here we give a short explication of how the coding is done in the DNA. It should be stressed that this coding is not nearly as straightforward as for the artificial chromosomes. Furthermore, it is highly adjusted to the concrete function it has — to store the information on how ribosomes will form proteins from amino-acids. That information is translated by the messenger RNA. The RNA is a copy of the DNA structure, with the only difference that it has the T (Timine) base replaced with the U (Uracil) base. Of the two helixes, only one—the coding strand—is enough to define the information held in the RNA (the other is determined by the pairwise complements, as in the

From every generation of chromosomes, the best ones are chosen. And while the best individuals in biological populations are selected by complex biological and environmental mechanisms, the selection process in genetic algorithms is governed by the evaluation of the observed population by a fitness function. It assigns a fitness value to every individual in the population. Then, the selection of the best individuals (chromosomes) is done in a stochastic process, in which the probability of selection of an individual is proportional to its fitness value. So, although better individuals have a greater chance of being selected, their selection is not guaranteed. Even the worst individuals have a slight chance for survival. [2]

The stochasticity of the procedure is further enhanced by the processes of recombination and mutation. Their action ensures the variability of solutions, and thus contributes to the exploration of the **whole** search or state space (the set of all possible solutions). This is essential for avoiding the traps of local extrema, in which traditional algorithms get stuck quite often.

Altogether, the combined action of the variation and selection operators results in a very efficient method for improving every next generation of solutions and relatively fast convergence to the desired optimal solution. That is why the evolutionary algorithms are very popular in solving a great variety of problems in both applied and fundamental sciences, such as finding extrema of complex functions, discovering the best game strategies, solving transport problems and modeling complex shape objects.[2] Among those are also the hardest computational problems, which belong to the so-called NP complexity class.[3] Despite some limitations and deficiencies, the evolutionary algorithms tackle all these tough tasks very successfully, thanks to their changed paradigm of operation and the loosened aim. That is, if the accurate solution is so hard-to-find, why not search for a solution that is just "good enough."

1.2. The purpose and the plan of the paper

In this paper, we shall focus our attention to the genetic algorithm (GA). We shall describe the GA in comparison to the corresponding biological processes and give the formalization of its basic steps. Then we shall implement these steps in the widely used mathematical and engineering programming environment MATLAB®.[4] The theoretical presentation is made in an educational, easy-to-follow approach, but mathematically corroborated where needed. For the readers who are already familiar with the field of evolutionary computation on a general level, a few illustrative examples written in MATLAB can make this article interesting as a step further toward the practical use of genetic algorithms—possibly also for their own problems. On the other hand, for the huge group of MATLAB users who are novices to the field, this could be a motivation to learn the topic that was mostly being connected to computer science. The final aim of this endeavor was to show that the genetic algorithm could be effectively programmed in MATLAB, and to describe how this was done in another authors' work, within the field of the game theory.

After the broader outline of the concepts used in evolutionary computation and the genetic algorithm was already given in this introduction, in section 2 we give the basic ideas that underlie the GA more precisely.

In section 3 we deal with the creation of individuals for the initial population. We spend some time to formalize the problem mathematically in a precise and complete manner. However, to keep the reading simple and interesting for the broader audience, we provide several simple examples. And along with them, the formalized steps are immediately implemented in MATLAB.

Section 4 formalizes and then exemplifies the process of selection of individuals for the new population (generation) and discusses the fitness function. The roulette wheel parent selection was illustrated on a simple example and implemented in MATLAB. In section 5 we explain the recombination and mutation, and, as always, cover the theoretical part with examples and the MATLAB code. Then, in section 6 we discuss the behavior of the GA in finding extrema for two illustrative functions. The full MATLAB codes for these programs are given in Appendix A.

In section 7, we discuss the use of GA in the iterated prisoner dilemma problem and show how it can discover unexpected aspects of such nontrivial systems. It should be—as we see it—the main scientific novelty of this work. Finally, section 8 concludes this paper.

2. OUTLINE OF THE GENETIC ALGORITHM

The genetic algorithm was invented by John Holland in 1975. [3] In order to find the function extrema, Holland presented each potential solution (extremum) as a sequence of bits, called the *artificial chromosome*. The bits, containing zeros and ones, or groups of several bits, can be considered as artificial, *binary genes* (Figure 1, confer also footnote number 1).

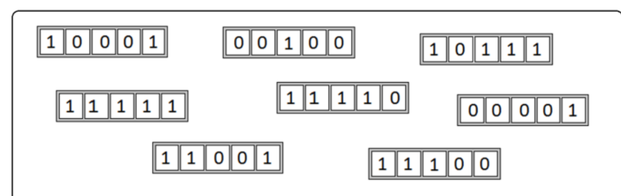

Figure 1. The population of artificial chromosomes, with genes presented by bits.

DNA). In the coding strand, the minimal complete biochemical information is stored in the three-letter words called codons, formed by the letters A, U, C, G. There are $4^3 = 64$ such words. The sequences of these words convey the information necessary for the formation of proteins, which is the basis for the existence and reproduction of all living organisms. [11]

[2] The shape of NASA ST5 spacecraft antenna was modeled by using GA to create the best radiation pattern. [12]

[3] NP stands for *nondeterministic, polynomial time*, meaning that the solution of the NP problem class can be found by the nondeterministic Turing machine (NTM) in the polynomial (favorable) time. The NTM is a theoretical machine that can guess the answers of the hard problems in the polynomial time, even though they require exponential time on normal, deterministic machines (algorithms). A simple comment and a connection of the NP problems to the GA can be found in [9].

[4] MATLAB® is a trademark of the MathWorks Inc.

Obviously, an artificial chromosome can be any form of data that properly describes the features of individuals within some population. Some of the individuals, i.e. chromosomes—as we shall simply call them in the further text—would eventually appear as the solution of the given problem.

2.1. Genetic algorithm

A pseudocode of a common version of the GA is given in Algorithm 1. The input is clear from the previous text. The optional input enables the calculation of the satisfying fitness value that can serve as one of the termination criteria (discussed more in the next subsection).

In step 1, the initial population of solutions (or individuals, or chromosomes) is created by random selection from the whole search space. This is elaborated in more details in section 3. In step 2, the initial population is evaluated by the fitness function, i.e. each chromosome of the population is assigned its fitness value (more on the fitness function is given in §4.1 and §4.2).

The iterative process that follows is presented by the two nested while loops. The condition of the outer loop is the negation of the termination criteria, which are explained in §2.2. For now, we assume that the criteria are not met so that the algorithm enters the outer loop.

Within the outer loop, in step 3.1, the GA enters another while-loop, which governs the selection of individuals for the next population. Within the nested loop, in step 3.1.1, a pair of chromosomes is selected randomly from the previous population (generation). The probability for this reselection is proportional to the relative fitness values of the chromosomes (see §4.3), i.e. the more probable chromosomes will be reselected more likely. In general, some of them can be selected more than once, and some of them will not survive the selection process and will not appear in the next generation. In step 3.1.2, the chromosomes from the selected parent pair are subjected to the recombination operator, which first splits them, and then mix the split parts in a specified way. Normally, every pair of parents will produce two descendants, so that every new generation of offspring contains the same number of individuals as the previous generation. In step 3.1.3, the two new children are enlisted as the new chromosomes of the next population. The process is iterated until the new population is filled with the desired number of chromosomes. When this is done, the while-loop 3.1 is completed.

In step 3.2, the descendant chromosomes are exposed to mutation. Depending on the chosen mutation rate, the mutation changes a certain number of genes in the population's gene pool (for further explanations see §5.4).

The processes of mutation and recombination are usually described by the action of the *recombination and mutation operator,* respectively, commonly known as the *variation operators* (more about them in section 5).

In step 3.3, the old population is replaced with the new one and the list of children is deleted. In step 3.4, the new population is evaluated by the fitness function, which is the same action as in step 2 (the duplication of these steps is a sacrifice to achieve the structured, while loop). Now the first iteration of the outer loop is finished, and the termination criteria are checked again.

Algorithm 1. A version of the genetic algorithm (GA).

Input: original function, search space (the set of all individual solutions, or chromosomes), fitness function.
Optional input: good enough solution, from which the good enough fitness value is calculated.

1. **Create initial population** by randomly selecting a certain number of chromosomes from the search space;
2. **Evaluate** the fitness of the current population by calculating every chromosome's fitness value.
3. **While the termination criteria are not met, do:**
[Terminat. criteria: the best fitness value is good enough or the specified number of iterations is done, or …]
   - **3.1 While the new population is not completed, do:**
     - **3.1.1 Select** two parent chromosomes by random selection from the previous population;
     - **3.1.2 Recombine** the parent chromosomes to obtain two children;
     - **3.1.3 Enlist** the two children in the new population;
   - **3.2 Mutate** the chromosomes in the new population;
   - **3.3 Replace** the previous population with the new one;
   - **3.4 Evaluate** the fitness of the current population (0.2).

Output: the best chromosome found (the one with the highest fitness value).

2.2. Termination criteria

If the fitness value of at least one chromosome satisfies some "prescribed criterion," the algorithm is finished. If not, the algorithm enters the next iteration.

A careful reader will probably note that finding and specifying such a criterion pose a fundamental problem not only for GA but also for all other evolutionary algorithms. Namely, GA does not investigate the search space thoroughly, methodologically, or in any "systematic" way. So, there are no sure indicators of when the computation is over. For instance, if the GA is searching for the unknown global extrema of some function, it cannot possibly know how far it is from them. Also, if we didn't search the whole search space, we cannot be sure that somewhere there isn't a more extremal value.

Obviously, in the spirit of our unsystematic and stochastically governed algorithm, we need a similar kind of termination criteria, i.e. (i) something loose but practical, or (ii) something heuristically based.

As for (i), sometimes we might be given a hint of what would be a "desired" or "good enough" solution (e.g. from the knowledge about the observed problem gathered by other means, or by the specifications given by a client or by a boss). By knowing it, the good enough fitness value could be calculated (opt. input) and this would be our first criterion. However, although our client or boss could be satisfied, we as scientists may still wonder how good our solution is and if it can be made better. This will be discussed a bit more in a moment.

And if the good enough solution is not known, then the "heuristic criterion" (ii) boils down to: "stop the iteration after a certain number of iterations," or "check the trends of certain *indicators* and decide about the termination." The indicator most used is the population average

fitness. By tracking it in each iteration, one can draw some conclusions on how the GA is progressing. If the indicator is stalling in its "extremal plateau" for several iterations, the algorithm can be halted. However, this is by no means a sure decision. Namely, although one can expect that with every new iteration of the GA the populations "evolve," i.e. that their average fitness values improve and get closer to a desired extremum, this cannot be taken for granted. The GA's method of searching is stochastic and it often results in unpredictable behavior of the chosen indicators. For instance, the average fitness value of the subsequent populations may stagnate for many iterations, making us believe that it is stable and without a chance for improvements. But after such intervals of stability, GAs can suddenly show the solutions that are worse than previous, or they can surprise us with sudden improvements. This will be illustrated in §6.

Further discussion of the possible indicators for the termination of the GA and their behavior are beyond the scope of this paper. We can simply say that the GA can be run for several times with a different number of iterations or, alternatively, started with different initial populations. Then the obtained results can be compared.

2.3. Use of genetic algorithms

As was already hinted at the end of §1.1, the power of GA is in the great diversification of the intermediary solutions, which ensure that the most of the search space will be properly investigated. This is essential for the functions with several extrema, the so-called *multi-modal functions*, whose search space is *multi-modal search space*. For such functions GA does excel. On the other hand, the classical methods—based on the analytical approach—will very often fail on them. They will tend to go straight toward the nearest local extremum. Another advantage is that the GA does not impose any restrictions on the functions that they investigate. The functions need not be continuous or analytic (derivable).

3. SEARCH SPACE AND INITIAL POPULATION

In the outline of the genetic algorithm in §2, we have already discussed the notion of the artificial chromosome. From there, it is clear that the chromosomes are a suitable representation of individual solutions within the search space of some problem.

3.1. Search space and individual solutions — their presentations and values

In order to illustrate the functioning of GA and the creation of its initial population, we define a common task of this field:

Find the global extremum (minimum or maximum) of the function $f(x)$ of discrete equidistant values x.

To formalize the problem a bit more, we specify that x belongs to the discrete set X,

$$x \in X = \{x_l, x_l + \Delta x, x_l + 2\Delta x, \dots, x_h\}\,, \qquad (1)$$

where x_l is the set's low and x_h its high limit, and Δx is the measure of the set's resolution, $x_l, x_h,\ \Delta x \in \mathbb{R}$. Obviously, X is the set of all possible solutions, i.e. the *search space* of the problem. The extrema of $f(x)$ on this discrete set must be achieved for one of its N_X elements, $= x_{extr}$, where:

$$N_X = \operatorname{card}(X) = \tfrac{x_h - x_l}{\Delta x} + 1\,. \qquad (2)$$

Effectively, we have provided a discrete representation of some closed real interval $[x_l, x_h]$, as is always the case when numerical methods are used on digital computers. This is valid regardless of the details of the number presentation, i.e. it does not matter if they are coded as integers or floating point numbers.[5] So, taking into account the eq. (2), we can choose the b-bit unsigned integer arithmetic without essential loss of generality. For it, the (whole) numbers span from 0 to $2^b - 1$. Thus, the achieved resolution Δx is

$$\Delta x = \frac{x_h - x_l}{2^b - 1}\,, \qquad (3a)$$

and an arbitrary x from the search space X can be presented as the b-bit number n:

$$n = n\langle b\rangle = \Delta x^{-1}(x - x_l) = \tfrac{2^b - 1}{x_h - x_l}(x - x_l). \qquad (3b)$$

Vice-versa, the b-bit number $n\langle b\rangle = n$ represents

$$x = x_l + \Delta x \times n = x_l + \Delta x \tfrac{x_h - x_l}{2^b - 1} n\,. \qquad (3c)$$

The binary presentation $n\langle b\rangle$ can be regarded as a vector,[6] which comes as a nice formal point in the further treatment of chromosomes as individual solutions, and in performing operations on them.

Now, having established the general transformation between the individual solution (x) and its presentation $n\langle b\rangle$ in some b-bit arithmetic, the elaboration can be simplified even further. By taking $\Delta x = 1$, the search space becomes a subset of integers. Furthermore, by taking $x_l = 0$, the lowest individual coincides with the low limit of the unsigned arithmetic and the highest individual coincides with the maximal number of the unsigned arithmetic, $x_h = 2^b - 1$, so that eq. (3c) gives

$$x = n = n\langle b\rangle\,. \qquad (3d)$$

In an undersized example (for the sake of keeping the things simple), let us choose the 8-bit integer arithmetic. It has the search space of $2^8 = 256$ individuals x_i, in the range $0 \le x_i \le 255$. These solutions can be indexed in an appropriate manner. Instead of the programming-style indexing, starting from 0, here we use the mathematical indexation to comply with the notation in the rest of the paper. So the indices i are for 1 greater than values, and the search space is $X = \{x_1, x_2, \dots, x_{256}\}$. Further indexations of the individuals within the populations derived from X will be changed according to the total number of the individuals in those populations (see §3.2).

[5] For instance, let us suppose that the interval of positive numbers $[0, 1]$ is covered by the single precision floating point numbers. The maximal resolution achieved by this arithmetic is given by the minimal Δx accurate across the whole interval. For $x_h = 1$, it is the relative precision of this data type, $\Delta x \approx 10^{-7}$. For the double precision arithmetic, $\Delta x \approx 10^{-14}$.

[6] The chromosomes $x_i = n_i\langle b\rangle$ are vectors in the vector space above the Galois filed GF(2^b).

It turns out that the presentation of individuals x_i in the binary system is both illustrative and practical for their further evaluation and manipulation in the form of artificial chromosomes in GAs. In our example, the binary presentation of the i-th individual is given as:

$$x_i = n_i\langle 8\rangle = \langle d_7 d_6 \dots d_1 d_0\rangle_{i,bin}\,, \quad (4a)$$

where d_j are binary digits, $j = 0,1,\dots,b-1 = 7$, $d_j \in \{0,1\}$. The case of $x = x_{27}$ (where we use the above, "shifted" indexation),

$$x_{27} = n_{27}\langle 8\rangle = 00011010_{bin}\,,$$

is shown in Figure 2 (the subscript bin will be omitted further on where the use of binary numerals is obvious).

Since our simplified choice implies that $x = n\langle b\rangle$ [confer eq. (3c)], we do not need to differentiate between an individual solution (chromosome) and its binary presentation. And of course, if the value V is attributed to every such binary presentation (vector), our simplified case leads back to the initial value of the solution in the search space:

$$V(n_i\langle b\rangle) = \sum_{j=0}^{b-1} d_j \times 2^j = x_i\,. \quad (5a)$$

This is so because the mapping between $n_i\langle b\rangle$ and $V(n_i\langle b\rangle)$ is a bijective function.

The (mathematical) indexation of individuals in the 8-bit search space chosen above, gives $V(n_i\langle b\rangle) = i - 1$.

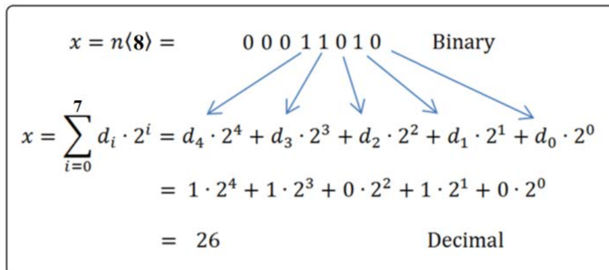

Figure 2. Summary of the notation via an example: individual solution, or chromosome $x_i = x$, its binary presentation $n\langle 8\rangle$ in 8 bits, and the value, $V(n\langle 8\rangle)$.

In the case of finding extrema of a function, the appropriate choice of the fitness function would be either the absolute value as defined above or some simple, linear function of it (see §4).

To generalize the above deliberation, for the function of two variables, $f(x,y)$, the chromosome is defined by the ordered pair (x_i, y_j). In the binary presentation, it would be written as:

$$(x_i, y_j) = \left(\langle d_{b-1} \dots d_1 d_0\rangle_i, \langle d_{b-1} \dots d_1 d_0\rangle_j\right). \quad (6)$$

If the problem is of different nature, the search space and its presentation should be changed appropriately. E.g., if the task is to find the shortest path between the given points in a plane or some higher order space, then the chromosomes should be presented as sequences of points in that space. For k points T_l, $l = 1,2,\dots,k$, and the initial (final) point $T_{l_{init}}$ ($T_{l_{fin}}$), the search space X would be the set of all sequences x_i,

$$x_i = \langle T_{l_{init}}, T_{l_1}, T_{l_2}, \dots, T_{l_{fin}}\rangle\,, \quad (7)$$

such that the following obvious restrictions would imply: (i) there are no repetitions of the same points in the sequence, (ii) the total number of points in the sequence is $\le k$.

We end this formal part with a few notes on the actual sizes of the search spaces in the practical use of GAs. E.g., if the space of 32-bit integers is used as the search space, the total of $4\text{Gi} \approx 4 \times 10^9$ individual solutions (chromosomes) exist. For modern computers, searching over such search space by the "brute force" would be no problem at all. With the computer performance of the order of 100 GIPS, and supposing that the function f can be calculated with about 10^2 instructions, it would be done in the order of a second (here we assume that the program code is written and compiled optimally, and is running much faster than the MATLAB code). However, for a two-variable function with the squared search space [amounting to $(2^{32})^2 = 16\ \text{Ei} \approx 2 \times 10^{19}$], the situation is drastically changed, and computing time would rise to the order of 10^9s. This and similar examples clearly show why the computer science ought to rely on efficient algorithms and not on the sheer power of computers.

In practical applications, the GA can be used for the very large, exponentially growing search spaces, for which the exhaustive searches are inefficient. If the fitness function is not too complex for evaluation, it is now commonly accepted that the GA will produce satisfactory results in acceptable time for most of such problems.

3.2. Creation of the initial population

From the defined search space X, the initial population $X_l = X_0$ is created by choosing a relatively small portion of the possible solutions from the search space,

$$X_0 = \{x_{0,1}, x_{0,2}, \dots, x_{0,N_0}\} \in X\,. \quad (8a)$$

As we have concluded in the previous subsection, the GA is in practice normally used on very large search spaces. Then the cardinality of the initial population is reduced from the cardinality of the whole search space for many orders of magnitude, which we write as

$$\text{card}(X_0) = N_0 \ll \text{card}(X) = N_X\,. \quad (8b)$$

It turns out that the efficiency of GA and the quality of its results do not improve much with enlargement of its population. A typical N_0 for 1-D problems is of the order of magnitude 10^1 to 10^2. For the b-bit chromosome presentations it is often suggested to take $N_0 = b$ (for more details see [9]). Once chosen—as it was already stated in §2—the number $N_l = N_0$ of individuals is normally kept throughout the iterations of the GA.

The selected number of N_0 individuals of the initial population will be chosen randomly from the total of N_X possible solutions of the search space. One way how to do it is described in the next subsection.

3.3. Definition of the search space and creation of the initial population in MATLAB

Example 1. Let the first, simple task be to find the maximum of the function $f(x) = 36x - x^2$. For the sake of making the example as simple as possible, we observe the interval of integers from 0 up to and including 31,

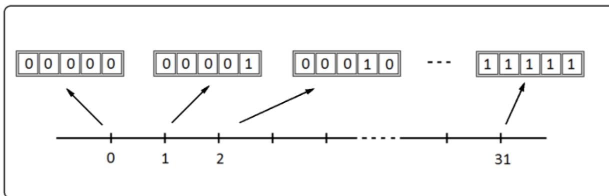

Figure 3. Binary presentation of the set of integers from 0 to 31 — which form the search space of Example 1.

Code Listing 1. Creation of the population of eight (8) 5-bit chromosomes in the MATLAB command prompt. The default orientation of binary numbers in MATLAB is reversed, with the most significant bit to the right.

```
>>popul=round(rand(8,5))

popul=

     0     0     0     1     0
     0     1     1     1     1
     1     1     0     0     1
     1     0     0     0     0
     0     1     0     1     1
     1     0     1     0     0
     1     0     0     0     1
     1     0     1     0     1
```

which implies that $x_l = 0, x_h = 31$, $\Delta x = 1$. The minimal number of bits that will allow the coding of the corresponding chromosomes is $b = 5$ (see Figure 3), leading to the search space

$$X = \{00000, 00001, \ldots, 11110, 11111\}.$$

If we want to create the initial population of $N_0 = 8$ chromosomes from X, this can be done very easily in MATLAB. The simplest way is to create the 8×5 matrix of random numbers in the range $[0, 1)$ by using the function `rand`, and then to round them to 0s and 1s with the `round` function (Code Listing 1).

Getting back—from the binary numbers to their decimal values—is done by the MATLAB function `bi2de`. Thus, the binary presentation of the chromosome $n\langle b\rangle$ (= `n`) is translated to its decimal value x (= `x`) by the following statement (see also Code Listing 2):

```
x = bi2de(n)                                (M1)
```

As a more general case, let us suppose that a search space in the interval $[0, 1]$ must be covered with a resolution of at least 0.05. From eq. (3a) it follows that

$$b \geq \log_2\left(\frac{x_h - x_l}{\Delta x} + 1\right), \tag{9}$$

leading to $b \geq 4.392$, and the minimal $b = 5$. With 5 bits, a slightly better Δx is achieved, $\Delta x = 1/31 = 0.3226$. This is shown in Figure 4.

For this case, the scaling from eq. (3c) gives the following MATLAB statement for conversion of the binary presentation of the chromosome $n\langle b\rangle$ (= `n`) to its decimal value x (= `x`),

```
x = xl + bi2de(n)*(xh-xl)/(2^nbts - 1) (M2)
```

Here the rest of the variables are: `xl` = x_l, `xh` = x_h, `nbts` = b.

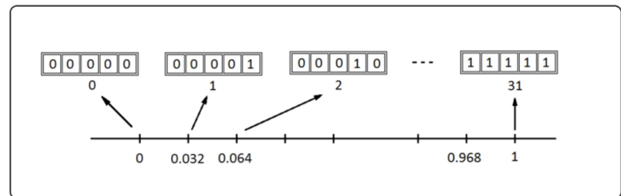

Figure 4. Another example of the 5-bit search space, organized in the real interval $[0,1]$, with $\Delta x = 1/31$.

Code Listing 2. Conversion of the binary presented chromosomes to their decimal values in MATLAB.

```
>>popul_dec= bi2de(popul)

popul_dec=

     8
    30
    10
     1
    26
     5
    17
    21
```

The MATLAB works with matrices, so that it can find the decimal value of the whole set of chromosomes by a single command:

```
popul_dec = xl +                            (M3)
     + bi2de(popul)*(xh-xl)/(2^nbts - 1)
```

4. SELECTION

After the creation of the initial population of solutions, the individual solutions in it must be evaluated to get the estimation of how fit they are for reproduction. As was already mentioned in §1 and §2, the evaluation is done by the fitness function, in literature also known as *quality*, *capability*, *goodness*, or *objective* function.

4.1. Choice of the fitness function

The fitness function F maps the search space to the set of real (rational) or integer nonnegative values, i.e. it assigns the nonnegative *goodness* or *fitness value* $F(x_i)$ to each individual solution, or chromosome x_i,[7]

$$F\colon X \to \mathbb{R}_0^+ \ (\mathbb{Z}_0^+),\ F\colon x_i \mapsto F(x_i) \geq 0. \tag{10}$$

The bigger is the value $F(x_i)$ the better will be the x_i's chance for survival and crossbreeding.

In some cases, the fitness function can be the same as the function whose extrema are being found. However, this is so only if the fitness values are of the same, positive sign or zero for all the individual solutions. That is the case of our first example, in §3.3, where the maximum of $f(x) = 36x - x^2$ is being sought. There, the greater fitness value belongs to the chromosome that is closer to the parabola maximum, so that one can put simply $(x_i) = f(x_i)$.

[7] The reason for non-negativity will become clear soon.

Otherwise, if the negative or both signs of the fitness values appear when finding some function's maximum, a problem arises when the probabilities proportional to the fitness values are to be assigned to the chromosomes. To shift the fitness values to the positive side, a positive constant should be added, so that in general $F(x_i) = f(x_i) + c, c \geq 0$.

The similar problem arises when the minimum is being sought for predominantly positive values of the original function $f(x_i)$. Now the chromosomes closer to the minimal value of $f(x_i)$ should be favored, meaning that $F(x_i)$ must be proportional to $-f(x_i)$. Again, if $-f(x_i)$ produces some negative values, a constant must be added, so that $F(x_i) = -f(x_i) + c,\ c \geq 0$.

Altogether, this leads to the following criterion for the fitness function:

The fitness function must be such that its higher values—to which the higher probabilities are to be assigned—are given to the chromosomes closer to the extremum that is being sought.

The general form of the fitness functions for finding extrema can now be summarized as follows:

$$\text{Finding max. of } f(x_i):\ F(x_i) = f(x_i) + c. \quad (11a)$$
$$\text{Finding min. of } f(x_i):\ F(x_i) = -f(x_i) + c. \quad (11b)$$

The choice of the constant c will be discussed in the next subsection.

For the travelling salesman problem, mentioned at the end of the §3.1, the appropriate value attributed to the x_i and its presentation $\langle T_{l_{Init}}, T_{l_1}, T_{l_2}, \ldots, T_{l_{Init}} \rangle$ would be the total length (λ) of the path defined by that presentation. Then one seeks the solution $x_{i_{min}}$, for which λ is minimal:

$$\text{Finding min. of } \lambda(x_i):\ F(x_i) = -\lambda(x_i) + c. \quad (12)$$

Further comments and an example of the application of GA to this problem can be found e.g. in [9, 10].

Generally, the fitness function should be picked appropriately to the problem. Furthermore, if the system's quality is described by a very complex function, the use of the GA could be questionable. In that case, the corresponding fitness function could impose too high a burden on the algorithm performance.

4.2. The choice of *c* (the problem of negative fitness values)

The role of the constant c is to shift the original function $f(x_i)$ to the nonnegative values. In the case of finding extrema of $f(x_i)$ that has negative values for some x_i, the ideal choice for c would be:

$$\text{Fnd. max. of } f(x_i):\ c = |\min\{f(x_i)\}|,\ f(x_i) < 0; \quad (13a)$$
$$\text{Fnd. min. of } f(x_i):\ c = |\max\{f(x_i)\}|,\ f(x_i) > 0. \quad (13b)$$

However, the $\min\{f(x_i)\}$ and $\max\{f(x_i)\}$ are generally not known. They pose the optimization problem that is just the opposite of the given one.

Another possibility to dissolve the problem of negative fitness values would be to make c so big that it surely lifts the $f(x_i)$ into the first quadrant for all $x_i \in X$. However, quite often this will cause another problem—the loss of details in the original function. That is, a great global shift up can make the chromosomes' fitness values very large positive numbers, and cause a compression of their relative differences. This will in turn make the assigned probabilities less distinct.

The solution is to stick to the first idea but to reduce the pool from which the minimal or maximal value of $f(x_i)$ is being found. In the l-th iteration of the GA, the natural choice is to find these values only among the chromosomes of the previous population, which will be shortly designated as l', $l' = l - 1$, for $l = 1, 2, \ldots \ldots, N_{itr}$, where N_{iter} is the total number of iterations.

Now, the eqs. (13) translate into the following practical formulas for c:

$$\text{Fnd. max. of } f(x_i):\ c = \left|\min\{f(x_{l',i})\}\right|, f(x_{l',i}) < 0; \quad (14a)$$
$$\text{Fnd. min. of } f(x_i):\ c = \left|\max\{f(x_{l',i})\}\right|, f(x_{l',i}) > 0. \quad (14b)$$

After the above deliberation, a careful reader probably noted that the constant c could be applied even if for finding maxima (minima) there are no negative (positive) values of $f(x_i)$. In that case, c would be negative, to shift the fitness values closer to zero and thus expand the relative differences among the chromosomes. To achieve that, the absolute values around the minimum and maximum functions and the following range limits should be simply removed. With that, the formulas for $c \in \mathbb{R}\ (\mathbb{Z})$, now without restrictions, are:

$$\text{Fnd. max. of } f(x_i):\ c = \min\{f(x_{l',i})\}; \quad (15a)$$
$$\text{Fnd. min. of } f(x_i):\ c = \max\{f(x_{l',i})\}. \quad (15b)$$

Of course, if the achieved expanding of the fitness values is not desirable, one should stick to the formulas (12).

Finally, the extra control over the expansion (compression) of the fitness function can be achieved by introduction of the parameter $\gamma \in \mathbb{R}_0^+$, that would multiply the $f(x_i)$ and shrink it for $0 < \gamma < 1$ and expand it for $\gamma > 1$. With that factor, the formulas (11) become:

$$\text{Finding max. of } f(x_i):\ F(x_i) = \gamma f(x_i) + c. \quad (16a)$$
$$\text{Finding min. of } f(x_i):\ F(x_i) = -\gamma f(x_i) + c. \quad (16b)$$

4.3. Population, average and relative fitness values

In the l-th iteration of GA, we calculate the fitness values $F(x_{l',i})$ of the individuals from the $(l-1) = l'$-th population, $X_{l'}$, with generally $N_{l'}$ individuals.[8] From these, the total fitness of the population l' is defined,

$$F(X_{l'}) = \sum_{i=1}^{N_{l'}} F(x_{l',i}), \quad (17)$$

as well as the average fitness of a chromosome within the population $X_{l'}$:

$$\overline{F(x_{l',\iota})} = \overline{F_{l'}} = \frac{F(X_{l'})}{N_{X_{l'}}}. \quad (18)$$

The ratio of the k-th chromosome fitness and the total population fitness gives the *relative fitness* (*goodness*) of the chromosome. This quantity will be chosen as the probability for the crossbreeding of that chromosome,

[8] The numbers of chromosomes, N_l, are kept population-dependent for the sake of generality (see also the discussion at the end of the subsection).

and denoted the same way: [9]

$$\Pr(x_{l',k}) = \frac{F(x_{l',k})}{F(X_{l'})} = \frac{1}{\sum_{i=1}^{N_{l'}} F(x_{l',i})} F(x_{l',k}) . \quad (19)$$

Having defined the probabilities of the chromosomes in the previous population, the set of N_l chromosomes for the next, $(l'+1) = l$-th population is randomly selected as described in the step 3.1 of Algorithm 1. It should be noted that this could produce quite a different set of new individuals. Some of the previous chromosomes (presumably the very probably ones) can appear more than once, and some of them can disappear (presumably those with small probabilities).

We end this subsection with a comment why it could be useful to keep the number N_l of chromosomes general, despite it is standardly fixed to $N_l = N_0$ in all iterations l (as stated in §3.2). Namely, although it is known that the GA's efficiency and quality of results is not very dependent on the population cardinality, one might resort to changing it in some occasions. For instance, if the indirect termination indicators stall for several iterations, one may consider increasing N_l to (re)check the stability of obtained solutions. This idea could be implemented in some enhanced versions of GA.

4.4. The example and the implementation of the selection in MATLAB

We continue our example from §3.3, for finding the maximum of $f(x) = 36x - x^2$. The initial population (X_0) created there consists of $N_0 = 8$ chromosomes.

Now we proceed to the next (first) iteration of the GA. We calculate the fitness values of the previous population by applying the fitness function $F(x_{0,i}) = f(x_{0,i})$, because $f(x_{0,i}) \geq 0$ for each $x_{0,i}$, and also for all x_i. The results are shown in Table 1, with somewhat simplified notation: $x_{0,i} \mapsto x_i$, $F(x_{0,i}) = F(x_i)$. E.g., for $x_1 = 01000$, $F(x_1) = f[V(01000)] = f(8) = 224$.

The total fitness of the initial population is $F(X_0) = 1752$, so that the relative fitness, and in the same time the probability for selection of the chromosome x_1, is

$$\Pr(x_1) = \frac{F(x_{0,k})}{F(X_0)} == \frac{F(x_{0,k})}{\sum_{i=1}^{N_0} F(x_{0,i})} = \frac{224}{1752} \cong 0.128 .$$

On the basis of the chromosome probabilities $Pr(x_i)$, $i = 1,2,\dots,8$, listed in Table 1, two new chromosomes will be selected to form a pair of parents (step 3.1.1 in Algorithm 1), and this will be—together with other actions—iterated four times (within the while-loop 3.1 in Algorithm 1).

The standard procedure for random choice is *roulette wheel parent selection*. It is illustrated in Figure 5 as a roulette on which each chromosome has its own sector, with the area percentage proportional to the chromosome's probability. For the right simulation of the chromosome sectors, the cumulative probabilities must be calculated from the chromosomes' probability distribution. In MATLAB, this is done by the `cumsum` function.

[9] Of course, the relative fitness qualifies for a probability function because $0 \leq \Pr(x_{l,k}) \leq 1$ and $\sum_{i=1}^{N_l} \Pr(x_{l,i}) = 1$.

Table 1. Calculations prior the process of selection. For the chromosomes from the previous population, $x_{l-1,i} = x_{0,i} = x_i$, the table shows: $n_i\langle 5\rangle$ = 5-bit binary presentation, $V(n_i)$ = value, $F(x_i)$ = fitness value, $\Pr(x_i)$ = probability of selection in the next population, $\widehat{\Pr}(x_i)$ = cumulative probability.

x_i	$n_i\langle 5\rangle$	$V(n_i)$	$F(x_i)$	$\Pr(x_i)$	$\widehat{\Pr}'(x_i)$
x_1	01000	8	224	0,128	12.8%
x_2	11110	30	180	0,103	23.1%
x_3	01010	10	260	0,148	37.9%
x_4	00001	1	35	0,020	39.9%
x_5	11010	26	260	0,148	54.7%
x_6	00101	5	155	0,088	63.5%
x_7	10001	17	323	0,185	82.0%
x_8	10101	21	315	0,180	100.0%
		Σ:	1752	1.000	

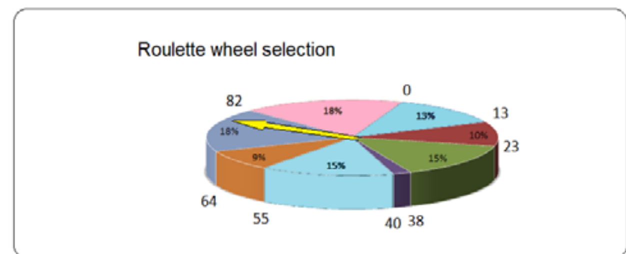

Figure 5. The roulette wheel parent selection for Example 1. To each of the 8 chromosomes of the initial population, a surface proportional to the chromosome probability is attributed.

In our example, the probability distribution `Pr` is defined in Table 1,

```
Pr = [0.128, 0.103, 0.148, 0.020,     (M3)
      0.148, 0.088, 0.185, 0.180]
```

from which the cumulative probabilities are obtained by

```
CumPr =  cumsum(Pr)                   (M4)
```

That gives the cumulative probability distribution:

```
CumPr = [0.128, 0.231, 0.379, 0.399,  (M5)
         0.547, 0.635, 0.820, 1.000]
```

Now, a random number from the interval $[0,1]$ is generated; in MATLAB, by function `rand`. The value of the random number determines the position and the sector on the imaginary roulette, and thus also the belonging chromosome (the random number is compared with the intervals defined by the `CumPr`, and translated accordingly into the index of the corresponding chromosome).

The MATLAB function that does that (taken from [10]) is shown in Code Listing 3. Given the probability distribution `P (= Pr)`, the function `RandChoseN(P,N)` returns the indices of `N` selected chromosomes. At the beginning of the function, the bin edges are defined from `P` by the `cumsum` function, and the vector `roulette` is zeroed. In the `for`-loop, the random number `x` is created. Then the histogram of its incidences (only one!) in the bins is made by function `histc` and stored in `Counts`. The index of `Counts` with the one (and only) hit is stored in `roulette`. The loop is repeated `N` times.

Code Listing 3. The MATLAB function for the roulette wheel parent selection. `RandChooseN(P,N)` gives the indices of `N` randomly selected chromosomes.

```
function roulette=RandChooseN(P,N)
BinEdges=[0, cumsum(P(:)')];
roulette=zeros(1,N);
for i=1:N
    x=rand;
    Counts=histc(x,BinEdges);
    roulette(i)=find(Counts==1);
end
```

4.5. A few comments on the selection process

The above depicted selection—implemented by the roulette method—is called *simple selection*. Another type of selection is *elimination selection*. While the simple selection deletes the old generation only after it is used for the selection process of all pairs of children for the new generation (before the step 3.3 in Algorithm 1), the elimination selection first detects *bad chromosomes* and deletes them prior the crossover and mutation operations. The shortage of chromosomes is then compensated by the additional offspring from the *good chromosomes*. To decrease or avoid the accumulation of duplicate chromosomes, some variants of this method check if the descendants are identical to their parents. If they are, the process of selection is repeated. This sort of selection is called the *elimination selection without duplicates*.

5. RECOMBINATION AND MUTATION

The recombination (crossover) and mutation in biological systems are complex processes of exchange of genetic material that are happening between the pairs of chromosomes (Figure 6). After the *interphase*, in which the chromosomes are duplicated, they align to each other, break in one or more places, and swap their fragments. This mixing results in the enhanced variability of the population.

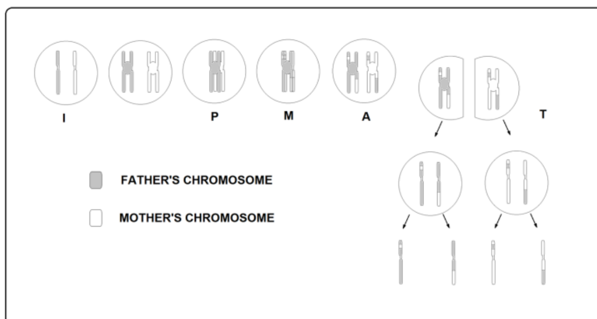

Figure 6. Meiosis, or reduction division. It starts from a somatic cell with a diploid number of chromosomes and ends with four sex cells that have the haploid number of chromosomes. The most important phase is *prophase* (P), in which the recombination and generation of gametes with completely different genetic material occurs.

5.1. Recombination in genetic algorithms

As it was already mentioned in §2.1, the process of recombination in evolutionary algorithms is mathematically abstracted to the action of the crossover operator. This *binary* operator—in the sense that it acts on two operands—exchanges the genes (bits) of two artificial chromosomes. As conceived by Holland in 1975, the crossbreeding performed by the crossover operator is a three-step process. [4] First, two individuals are taken from the parent (previous) population. Second, a point or several points of the chromosome splitting are chosen — the so-called *crossover points*. These points define the segments of genes (bits) between them. The contents of some of those segments are being exchanged, resulting in the generation of two new descendants with generally different chromosome structure (Figure 7).

The simplest way of choosing the crossover points is by random selection. Then some of the formed sections of bits are exchanged and some are not.[10] This is the case of the ordinary crossbreeding, which is governed by the corresponding *ordinary crossover operator*.

In the simple example of Figure 7, the ordinary crossbreeding with single crossover point is applied. It defines the left and the right sections of the two chromosomes, of which the right sections are exchanged. Thus, the probability that the descendant has its corresponding (upper or lower) parent's genes in the left section is one, and in the right section is zero, meaning that the right section has the genes from the other, "crossed" parent.

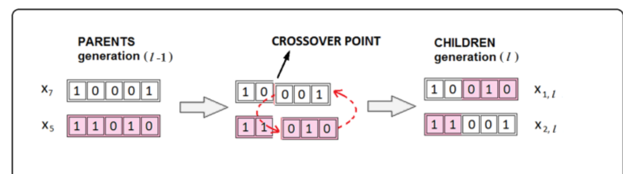

Figure 7. The recombination of chromosomes with one breaking point. From the previous generation ($l' = l - 1$), two parent chromosomes were selected. The crossover operator then splits them between the second and the third bit and exchanges their right parts. Thus, a pair of new chromosomes of the l-the generation is formed.

The above and all the other probability schemes can be generalized by the *crossover probability mask* $\mathbf{m}_C$,

$$\mathbf{m}_C = \big(Pr(d_0), Pr(d_1), \ldots, Pr(d_{b-1})\big). \quad (20a)$$

The mask consists of b-plets of probabilities $Pr(d_j)$, $j = 0,1, \ldots, b-1$, each of which presents the probability that the descendant has the gene—here the binary digit d_j —inherited from its corresponding predecessor. If the mask for the first chromosome in a pair (the upper one in Figure 7) is $\mathbf{m}_C$, the mask applied to its conjugal partner (the lower one in Figure 7) must be $\overline{\mathbf{m}_C}$, with the complementary probabilities, $\overline{Pr(d_j)} = 1 - Pr(d_j)$:

$$\overline{\mathbf{m}_C} = \big(\overline{Pr(d_0)}, \overline{Pr(d_1)}, \ldots, \overline{Pr(d_{b-1})}\big). \quad (20b)$$

With this notation, the probability masks for the example of ordinary crossover in Figure 7 are:

[10] By exchanging all of them, the net result would be the two starting chromosomes with swapped indices.

$$\mathbf{m}_{COrd} = (1, 1, 0,0,0), \; \overline{\mathbf{m}_{COrd}} = (0, 0, 1,1,1).$$

As we have already pointed out, there can be more than one breaking point. For b bits there are maximally $b - 1$ breaking points. The introduced notation enables the specification of such and other cases, by forming the adequate recombination probability masks.

The *uniform crossover* operator allows equal-chance swapping of all bits, meaning that all the probabilities in the mask are equal, $Pr(d_j) = \overline{Pr(d_j)} = 1/2$, leading to

$$\mathbf{m}_{CUnif} = (0.5, 0.5, \dots , 0.5). \qquad (21)$$

In other words, this mask states that all genes from the first parent have the opportunity of $1/2$ to end up in the corresponding (first) descendant chromosome, and that the same is valid for the genes of the second parent and the corresponding (second) descendant chromosome.

Finally, the fully general form of the mask in eq. (20a) allows the fully variant recombination (sometimes misleadingly called the *p-uniform crossover*). We shall call it the *variant crossover* and denote it by *CVar* index. To affirm the meaning of their masks, we give the following example:

$$\mathbf{m}_{CVar} = (0.2, 0.3, 1.0, 1.0, 0.0).$$

Here, the first child will inherit the first gene from her/his first parent with probability 0.2, and the second gene with probability 0.3 (meaning that the probabilities that these genes will be inherited from the second parent are 0.8, and 0.7, respectively). The third and fourth genes will be surely inherited from the parent one (i.e. they stay as they were in the first parent), and the last gene will be surely inherited from the second parent (i.e. it is surely exchanged from what it was in the first parent).

The recombination does not have to be done on all individuals within a population. The relative portion of the chromosomes that participate in the recombination—which can be also formulated as the rate of the activation of the crossover operator—is called the *crossover rate* or *crossover probability*. We shall denote is as γ_C, $\gamma_C \in [0, 1]$. If in population (iteration) l the N_l chromosomes were selected, only $N_{l,C}$ of them will be selected for crossbreeding,

$$N_{l,C} = \gamma_C \times N_l n. \qquad (22)$$

The values of γ_C that are reported to give the best behavior of GA are between 0.5 and 1.0. [5]

5.2. Implementation of recombination in MATLAB

A simple implementation of recombination in MATLAB is shown in Code Listing 4. It selects one breaking point by using the function `randi(nbits-1)`, which randomly picks an integer between 1 and $b - 1$. Then the part of the chromosome to the right of the breaking point is exchanged. The exchange of these chromosome parts is performed with the last two statements of the code.

Code Listing 4. Implementation of the recombination with one randomly chosen crossover point in MATLAB. Prior to this code, the vectors of indices for dad and mom are to be defined. They denote the rows of the population matrix that present the corresponding parents.

```
%Calculate the probability of selection:
probs=fitness/sum(fitness);
%Selection of parents:
M=popsize/2;
dad=RandChooseN(probs,M);
mom=RandChooseN(probs,M);
%%CROSSOVER
%Choose the crossover point:
xp=randi(nbits-1)
%Take every 2nd dad's chromosomes
%and exchange genes with mom
popul(1:2:popsize,:)=...
[popul(dad,1:xp) popul(mom,xp+1:nbits)];
popul(2:2:popsize,:)=...
[popul(mom,1:xp) popul(dad,xp+1:nbits)];
```

5.3. Mutation in biology

As a motivation for the implementation of mutation in GA, here we give a short reminder of the influence of mutation on living organisms. Unlike the recombination, the mutation in biology is a random change of genetic material in a somatic or a sex cell. That change can be influenced or stimulated by outer factors, like radiation, and sudden environmental changes. In natural conditions, it is very rare. If it happens to a somatic cell, it can cause its damage or death. It can occasionally trigger uncontrolled division and malign diseases. However, the consequences of such mutations are not inherited by the future generations. So, from the evolutionary point of view, much more significant are mutations that happen in the sex cells, because they are transferred to the next generations. They are manifested in the changes of the organisms' phenotypes, i.e. as new inherent characteristics of the descendants. This makes the mutation the second important cause of the variability of biological populations.

5.4. Mutation in genetic algorithms

In a genetic algorithm, the mutation will be performed by a *mutation operator*. It acts on a single chromosome and is therefore a unary operator. It changes one or more randomly chosen genes (bits) of the artificial chromosome. For instance, the mutation operator can change a chromosome $x_i = 11111$ to its mutant $x_i' = 11101$.

We formalize the action of the mutation operator in a way similar to the one used for the crossover operator. First, we introduce the mutation rate $\gamma_M \in [0,1]$, which decides what percentage of the whole *gene pool* will be influenced by mutation. That is, with N_l chromosomes in a population, of which each has n genes, the gene pool contains a total of $N \cdot n$ genes. Then the number of mutations performed in each iteration of the GA is

$$N_{l,M} = \gamma_M \times N_l n. \qquad (23)$$

Following the example of the natural mutation, γ_M in GA is normally made quite small, typically between 0.001 and 0.01. E.g., with $\gamma_M = 0.01$ and a gene pool with 20 5-bit chromosomes, only one gene (bit) and one chromosome are expected to undergo the mutation.

To further generalize the action of the mutation operator, once can introduce the *mutation probability mask*, $\mathbf{m}_M$, which is fully analogous to the crossover probability mask, defined by eq. (20a).

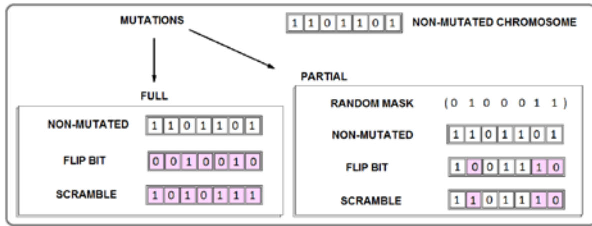

Figure 8. The action of inverting and mixing mutations illustrated on the chromosome 1101101. The inverting (flip) mutation inverts the zeros into ones and vice-versa. The mixing mutation permutes (scrambles) the bits while conserving their parity (the numbers of both zeros and ones stay the same). The partial versions of the mutations operate only on the selected fields of bits (shaded).

In a *simple mutation*, $N_{p,M}$ randomly chosen genes are mutated, meaning that its mask is the same as the uniform crossover mask in eq. (21). Besides that, the *mixing* and *inverting mutations* are used. The mixing mutation permutes the genes within the $N_{p,M}$ randomly chosen chromosomes or within their parts specified by a mutation mask. The parameters of the mutation mask can also be generated randomly. In the *inverting mutation*, the bits specified by the mutation mask are inverted. [6] Both mutations are sketched in Figure 8.

6. EXAMPLES IN MATLAB

Two complete MATLAB programs for finding extrema of the polynomial functions are given in Appendix A. Relying on the provided comments, the readers acquainted with MATLAB basics should follow their code easily.

The Appendix A.1 contains the code for our simple Example 1, introduced in §3.3. The performance of GA for this case is shown by two diagrams of Figure 9. The upper diagram shows that the average fitness value rises to its upper plateau in only about 20 iterations, but keeps oscillating. In fact, it looks like the GA is quite struggling to get to the maximum of this simple 2nd order function. A few iterations prior 30 and 40, it finds that

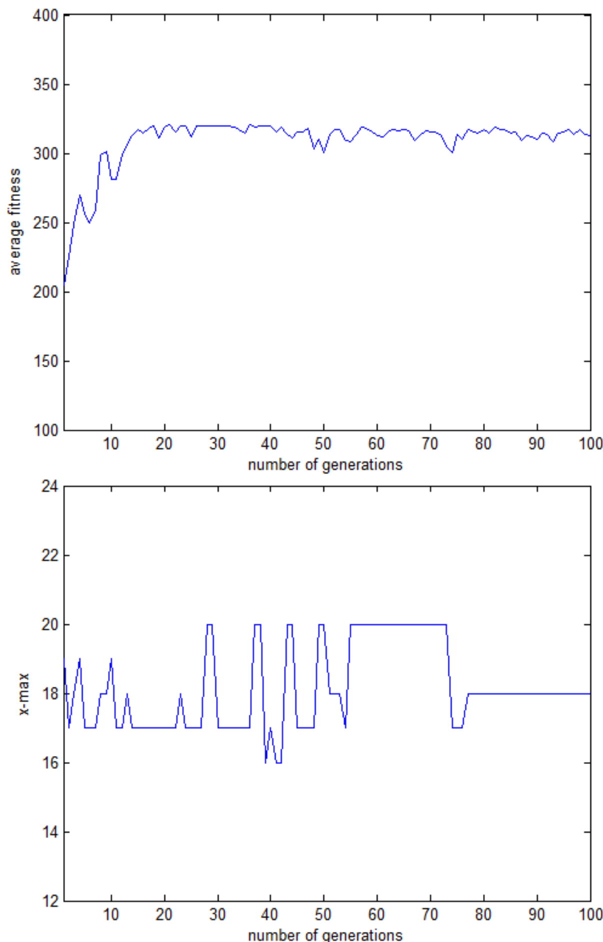

Figure 9. Diagrams of the average fitness (up) and the fittest GA solution (down), for square function of Exmpl. 1. The population average fitness increases quickly, but there is no sure indicator of the optimal solution. During the generations 55-75, the fittest chromosome is $x_{i,\max} = 20$. After 75 generations, GA finally finds the exact solution, $x_{i,\max} = 18$.

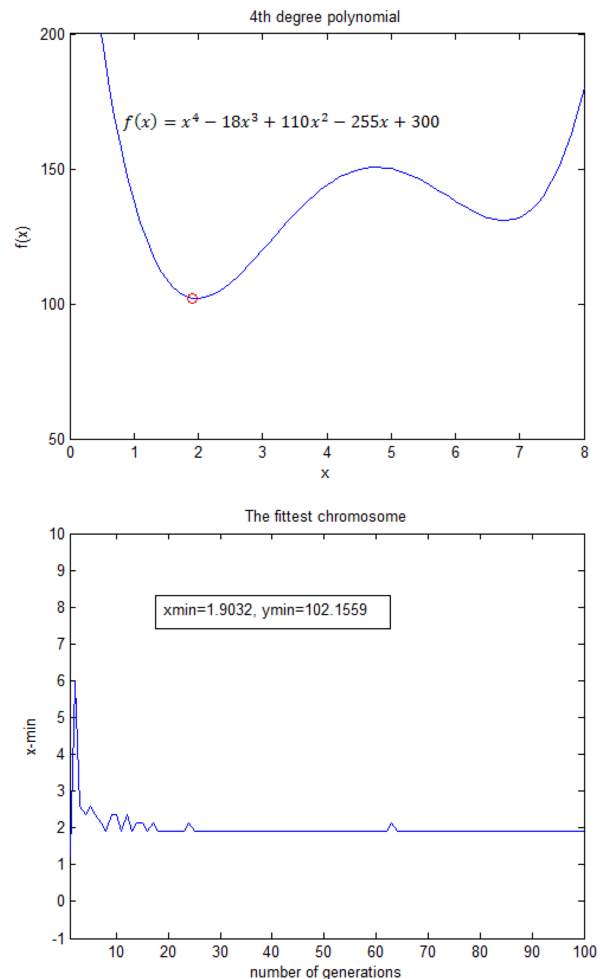

Figure 10. Diagrams of the 4th-degree polynomial (up) and the fittest GA solution. As can be seen by inspecting both diagrams, the algorithm successfully finds the global minimum and gives its accurate coordinates without being trapped in the local minimum (around $x = 6.75$).

the maximum is at $x = 20$, but does not converge immediately to it. Then, from 55th to 75th iteration it stabilizes at that value. However, the average fitness value keeps on oscillating and shows a few dips. The right answer, $x = 18$, is finally found after 75 iterations.

The Appendix A.2 gives the code for GA that finds the global minimum of the 4th-degree polynomial. The 5-bit chromosome representation is adjusted to the interval $[1, 8]$ by formula (3b). As can be seen in the upper diagram of Figure 10, the polynomial has two minima (of course, with a local maximum between them). The right minimum is only local. On the lower diagram, one can see how the GA hits the global minimum in less than 30 iterations. It is found for the chromosome $n\langle 5\rangle = 00100$, with the value $V(n\langle 5\rangle) = n = 4$, for which eq. (3c) gives $x = x_{\min} = 1.9032$. This abscise value is marked on the polynomial curve as a small circle. It can be seen that the result is very accurate.[11] All in all, it looks like the previous simple function wasn't enough of a challenge for this mighty algorithm!

However, a careful reader will notice that not everything is in a steady state. Slightly below 65 iterations there is an instability, showing slightly higher value for the $x_{\min}$. Such behavior is quite common for genetic algorithms, due to the action of the variation operators.

7. USE OF THE GENETIC ALGORITHM IN THE STOCHASTIC ITERATED PRISONER DILEMMA

After showing the efficiency of the genetic algorithm for finding global extrema of the single-variable functions, here we elaborate on how it was used for a function that depicts cooperation in the heterogeneous population of selfish individuals in the problem known as *Stochastic Iterated Prisoner's Dilemma* (SIPD), which is a version of the *Iterated Prisoner's Dilemma* (IPD).[12]

We shall expose the problem briefly at the expense of accuracy (and hopefully not of the clarity). In the classic form of the IPD, the population usually consists of players represented by deterministic strategies that always follow their algorithms. For example, the ALLC strategy plays cooperatively always, and the strategy ALT plays cooperatively in every 2nd round. The TFT strategy[11] simply copies the moves of the opponent player from the previous round. However, in the real-life situations, both animals and humans make mistakes. This invites for the implementation of the uncertainty into the players' moves. The IPD that includes that is the above mentioned Stochastic IPD. In SIPD, each player is represented by a pair of probability values (p, q), whereby p stands for the probability of cooperation after the opponent's cooperation, and q stands for the probability of cooperation after the opponent's defection.[13] Now suppose that we want to find the best strategy of the SIPD in which the total payoff for a particular player equals to the sum of payoffs gained in the duels with all other players. The payoff as a function of (p, q) depends on various parameters, such as the total number of players, the strategies of the players and the number of the game rounds. Interpreted in another way, the payoff is a multivariable function, whose maximum cannot be found by the classical optimization algorithms.

One possibility to find the extrema of such payoff function is to simulate the natural selection. One can generate a set of strategies and then "leave it to the natural selection." The best strategy will be the one that becomes dominant in a population.

Most of the studies in this field have been done under the assumption that the individuals reproduce asexually; the strategies fight among each other, the generation after generation, with relative frequencies proportional to their payoff in the previous generation. It means that particular strategy could be, and usually is used by more than just one player. The players with copied strategies might be considered as the *clones* of the corresponding original strategies. It should be stressed that in this approach there is no crossbreeding between different strategies.

On the other hand, the GA could be applied to the SIPD if the strategies are suitably encoded in the form of artificial chromosomes. An example of such approach is our model, in which each reactive strategy (p, q) is represented by a 10-bit chromosome. [8] The first five bits in the chromosome encode the probability p, and the remaining five bits encode the probability q (Figure 11).

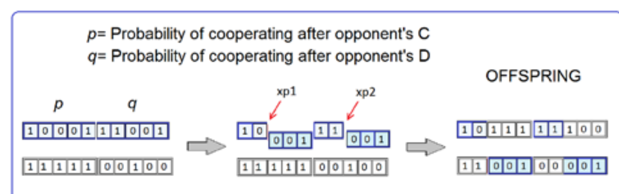

Figure 11. Double-point crossover operator applied to the parent chromosomes which represent a (p, q) SIPD strategy.

[11] The accuracy would be significantly diminished if the interval $[0, 8]$ was chosen. For it, $\Delta x = 0.25806$, and the individual closest to the minimum is 00111, with value 7, giving $x = 1.8065$ (0.0967 lower than $x_{\min}$). The second best is 01000, with value 8, giving $x = 2.0645$ (0,1613 greater than $x_{\min}$). The high dependence of the result on the choice of interval is an obvious consequence of the low resolution of the search space, i.e. of the low number of the chromosome bits.

[12] The Prisoner's Dilemma is a well-known game in which two players can cooperate with or betray each other. The outcome for each player varies according to the combined outcome for them both. It is rational that each player defect (D), even though they would both gain more if they cooperated (C). If there is another encounter of the players, in another round of the game, the cooperation may be really profitable. This kind of game is called *Iterated Prisoner's Dilemma* (IPD). It is considered as the paradigm for the evolution of cooperation among selfish individuals. [7] In the context of biology, the game payoff (food, water, etc.) may be considered as fitness. To find out which IPD strategy is the most effective, the mathematician, political scientist and game theorist Robert Axelrod organized a tournament in 1979, and invited the scientists to submit their solutions. The simplest strategy, known as *Tit for Tat* (TFT), won the tournament. This was a big surprise because it was well known by then that the strategy of defecting on every round (ALLD) is the only strategy that is generally evolutionarily stable (in more details commented in [8]).

[13] This kind of strategy is known as *reactive strategy*.

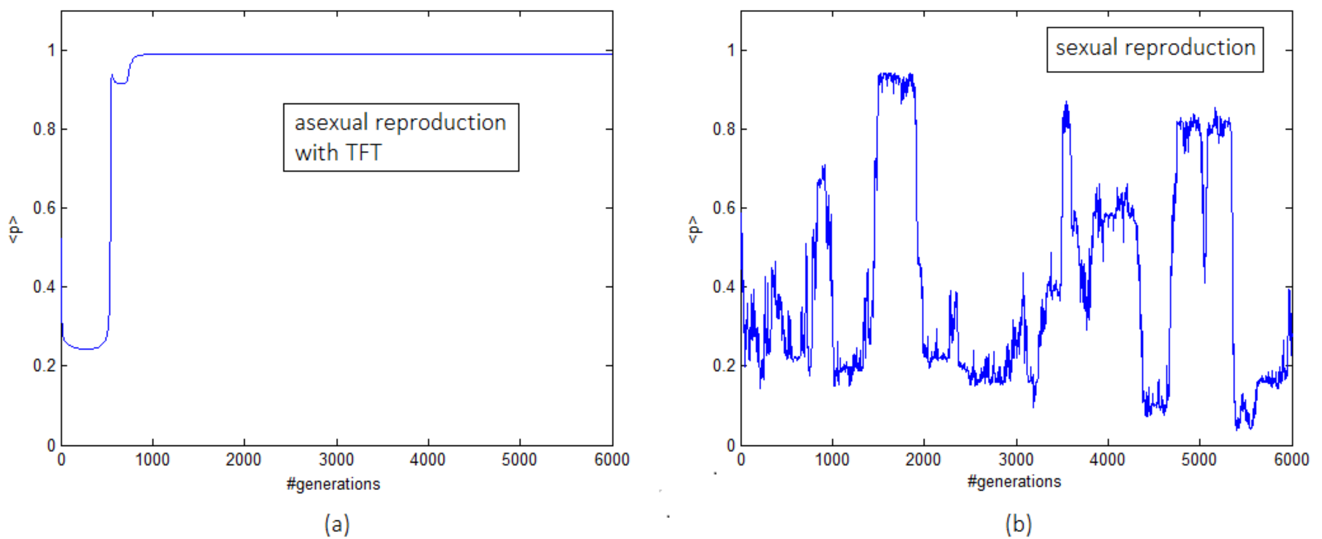

Figure 12. The cooperation coefficients over time in (a) populations that reproduce asexually, (b) populations with sexual reproduction. For the asexual reproduction, the cooperative strategies win after around 1000 generations. The sexual reproduction does not allow the extinction of different strategies and results in their erratic and cyclic exchanges.

The payoff gained in one round serves as a fitness function for the evaluation of strategies — the most successful strategies will have the highest chances for mating. After being selected, every chromosome is subjected to double point crossover operator. One crossover point is randomly selected on the p and another on the q chromosome segment (confer Figure 11).

At the beginning of each iteration (each round of the game), the *cooperation coefficients* are calculated as the averages of the strategy p-probabilities,

$$\bar{p} = \sum_{i=1}^{N_{str}} \omega_i p_i . \tag{24}$$

Here ω_i stands for the relative frequency of the i-th strategy with the cooperation probability p_i, and the summation is over all N_{str} strategies.

This, sexually reproduced SIPD model have shown that the subsequent populations are going through the erratic and cyclic exchanges of cooperation and exploitation. Such result is just the opposite from what the previous models—with asexual reproduction of individuals—are giving. In them, once some strategy is extinct, its restoration is impossible. Figure 12 shows the cooperation coefficients for both of these cases.

The instabilities in our model are the consequence of the GA variation operators. They preserve the genetic variability and diversity of the dominant strategies. In this case, the genetic algorithm does not deliver an optimal solution (the best strategy). On the contrary, it restores the defective strategies and leads to the instability and divergence. There is no dominant strategy that would eliminate the opponent players for good. Both, the cooperators and the exploiters can rule the world by themselves just temporarily.

8. CONCLUSION

When some optimization problem is being solved using a genetic algorithm (GA), it is not guaranteed that it will give a perfect or even a very good solution. But quite often it will deliver a "good enough solution", and almost always it will provide some kind of useful information about the problem.

The examples presented in this paper have shown that the global extremum of the single-variable functions was found very effectively, especially in the case of the 4th-degree polynomial. Its derivative gives a third order equation that cannot be solved easily. Obviously, the strength and the justification for the use of GA increase with the number of function's local extrema and generally with the degree of the function complexity.

On the other hand, in the case of the Stochastic Iterated Prisoner's Dilemma, the GA did not perform that efficiently. The populations of strategies do not evolve toward any dominant type of strategy. Instead, they diverge and jump from one near-optimal solution to another, showing that there is no clear winning strategy. However, although "the best" strategy wasn't found, the GA revealed erratic and circular patterns in the level of cooperation over time.

Henceforth, we conclude that—besides the determination of the function extrema—the genetic algorithm can occasionally give us something quite different — a new insight into the subject of study.

9. REFERENCES

[1] Mitchell, M.: An Introduction to Genetic Algorithms. Bradford Book MIT Press, Cambridge, Massachusetts, 1996.

[2] Gondra, I.: Parallelizing Genetic Algorithms, in Artificial Intelligence for Advanced Problem Solving Techniques, Vrakas, D. & Vlahavas, I. eds., Hershey, Information Science Reference, 2008.

[3] Negnevitsky, M.: Artificial Intelligence – A Guide to Intelligent Systems, Addison-Wesley, Harlow, 2005.

[4] Booker, L. B.: Recombination, in Handbook of Evolutionary Computation, Kenneth de Jong ed., Oxford University Press, 1997.

[5] Lin, W. Y.; Lee W. Y., Hong, T. P.: Adapting Crossover and Mutation Rates, Journal of Information Science and Engineering, No. 19, 2003, pp. 889-903.

[6] Soni, N.; Kumar, T.: Study of Various Mutation Operators in Genetic Algorithms, International Journal of Computer Science and Information Technologies, Vol. 5, No 3, 2014, pp. 4519-4521.

[7] Nowak, S.: Tit for tat in heterogeneous populations, Nature, Vol 355, 1992, pp. 250-253.

[8] Hižak, J.: Genetic Algorithm Applied to the Stochastic Iterated Prisoner's Dilemma, conference paper, Basoti 2016, Tallinn, Estonia (to be published) https://www.researchgate.net/publication/305768564_Genetic_Algorithm_Applied_to_the_Stochastic_Iterated_Prisoner_Dilemma.

[9] Obitko, M.: Introduction to Genetic Algorithms, Web tutorial, http://www.obitko.com/tutorials/genetic-algorithms/index.php.

[10] Hundley, D. R.: Lectures on Math Modeling, Ch. 4 Genetic Algorithm, Walla Walla: Whitman College, http://people.whitman.edu/~hundledr/courses/M350/Note02.pdf.

[11] Wkp2: Wikipedia articles: Genetic Code, Codons, https://en.wikipedia.org.

[12] Wkp1: Wikipedia article: Space Technology, https://en.wikipedia.org/wiki/Space_Technology_5.

* All cited Web sites and Web pages were accessible and their URLs were correct in December 2016.

Authors' contacts:

Jurica Hižak, M.Sc.
University North, Dpt. of Electrical Engineering
104. brigade 3, HR-42000 Varaždin
jurica.hizak@unin.hr

Robert Logožar, Ph.D.
University North, Dpt. of Multimedia.
104. brigade 3, HR-42000 Varaždin
robert.logozar@unin.hr

APPENDIX A. FINDING EXTREMA OF TWO POLYNOMIAL FUNCTIONS BY THE GENETIC ALGORITHM — MATLAB CODE

A.1. Genetic algorithm for finding the maximum of a square function

```
%%MAXIMUM
%This program finds the x-value for which
%the function f(x)=36x-x^2 reaches the
%maximum value.
%Program written by Jurica Hižak
%University North, Varaždin,2016

%FITNESS FUNCTION:
ff=inline('36.*x-x.^2')
%Parameters:
n=100          %number of generations
popsize=40     %population size
mutrate=0.001  %mutation rate
nbits=5        %number of bits in a
               %chromosome

%CREATE an initial population:
popul=round(rand(popsize,nbits))
%Create a vector that will record near-
%optimal solutions across the generations:
best=zeros(1,n)
%Create a vector that will track the
%average the fitness of the population:
avg_fitness=zeros(1,n)

%EVOLUTION
iter=0
while iter<n
    iter=iter+1
%Calculate the value of each chromosome:
pop_dec= bi2de(popul)
%Calculate the fitness:
f=feval(ff,pop_dec')
%Average fitness of the population:
avg_fitness(iter)=sum(f)/popsize
%Sort fitness vector and find the indices:
[f,ind]=sort(f, 'descend')
%Sort chromosomes according to their
%fitness:
popul=popul(ind,:)
%The x-values:
pop_dec=bi2de(popul)
%The x-value of the chromosome with the
%highest fitness:
best(iter)=pop_dec(1)
%Calculate the probability of selection:
probs=f/sum(f)
%Selection of parents:
M=popsize/2
dad=RandChooseN(probs,M)
mom=RandChooseN(probs,M)

%CROSSOVER
%Choose the crossover point:
xp=randi(nbits-1)
%Take every 2nd dad's chromosomes
%and exchange the genes with mom's:
popul(1:2:popsize,:)=[popul(dad,1:xp)...
popul(mom,xp+1:nbits)]
popul(2:2:popsize,:)=[popul(mom,1:xp)...
popul(dad,xp+1:nbits)]
```

```
%MUTATION
%Number of mutations in the population:
nmut=ceil(popsize*nbits*mutrate)
 for i=1:nmut
    col=randi(nbits)
    row=randi(popsize)
    if popul(row,col)==1
        popul(row,col)=0
    else
        popul(row,col)=1
    end
 end
end

%PLOT the x-value of the best chromosome
%over time:
figure
%Generate an array of integers from first
%to the last generation:
t=1:n
plot(t,best)
axis([1 n 12 24 ])
xlabel('number of generations')
ylabel('x-max')
%Plot the average fitness value of the
%population over time and label the axes:
figure
plot(t,avg_fitness)
axis([1 n 100 400])
xlabel('number of generations')
ylabel('average fitness')
```

A.2. Genetic algorithm for finding the global minimum of the 4-th degree polynomial

```
%%MINIMUM OF THE 4th DEGREE POLYNOMIAL
%The program finds the x-value for which
%the function
%f(x)=x^4-18x^3+110x^2-255x+300
%reaches the minimum.
%Program written by Jurica Hižak
%University North, Varaždin, 2016.

%FITNESS FUNCTION:
clear all
f=inline('x.^4-18*x.^3+110*x.^2-...
255*x+300')
%Plot the function and hold the figure:
x=0:0.1:8;
figure
ymin=feval(f,x);
plot(x,ymin)
axis([0 8 50 200 ])
xlabel('x')
ylabel('f(x)')
hold on
n=100          %number of generations
popsize=40     %population size
mutrate=0.001  %mutation rate
nbits=5        %number of bits in a
               %chromosome

%CREATE an initial population:
popul=round(rand(popsize,nbits))
%Create a vector that will record
%near-optimal solutions across
%the generations:
best=zeros(1,n);
%Create a vector that will track
%the average fitness of the population:
avg_fitness=zeros(1,n);

%EVOLUTION
iter=0
while iter<n
    iter=iter+1
%Having assumed that the solution takes
%place somewhere inside the interval(1,8),
%insert as many real values as possible
%considering the number of bits nbits
%(e.g. 5-bits string allows 32 values).
%Therefore, the decoded value of each
%chromosome is:
pop_dec= 1+bi2de(popul).*(8-1)/(2^nbits-1)

%FITNESS
%First, find the function value fv
%of each chromosome:
fv=feval(f,pop_dec');
%Since we are looking for the MINIMUM,
%the fitness function will be defined in
%respect to "the worst chromosome" whose
%fv-value is actually the highest:
fit=max(fv)-fv;
%Average fitness:
avg_fitness(iter)=sum(fit)/popsize;
%Sort fitness vector and find the indices:
[fit,ind]=sort(fit, 'descend');
%Sort chromosomes according to their
%fitness:
popul=popul(ind,:);
%The x-value of the chromosome with
%the highest fitness:
best(iter)=pop_dec(1);
%Calculate the probability of selection:
probs=fit/sum(fit);
%Selection of parents:
M=popsize/2;
dad=RandChooseN(probs,M);
mom=RandChooseN(probs,M);

%CROSSOVER
%Choose the crossover point:
xp=randi(nbits-1)
%Take every 2nd dad's chromosomes
%and exchange the genes with mom's:
popul(1:2:popsize,:)=[popul(dad,1:xp)...
popul(mom,xp+1:nbits)];
popul(2:2:popsize,:)=[popul(mom,1:xp)...
popul(dad,xp+1:nbits)];

%MUTATION
%Number of mutations in the population:
nmut=ceil(popsize*nbits*mutrate);
 for i=1:nmut;
    col=randi(nbits);
    row=randi(popsize);
    if popul(row,col)==1;
        popul(row,col)=0;
    else
        popul(row,col)=1;
    end
 end
end
```

```
%MINIMUM:
xmin=best(n)
ymin=feval(f,xmin)
%Plot (xmin,ymin) as a red ring on the
%current figure:
plot(xmin,ymin, 'ro')


%PLOT the x-value of the best chromosome
%over time:
figure
%Generate an array of integers from the
%first to the last generation:
t=1:n
plot(t,best)
axis([1 n -1 10 ])
xlabel('number of generations')
ylabel('x-min')
%Show textbox with the output value
%(xmin,ymin)
dim = [.26, .2, .2, .6];
str = ['xmin=',num2str(xmin),...
       'ymin=',num2str(ymin)];
annotation('textbox',dim,'String',...
           str,'FitBoxToText','on')
```