AN IMPLEMENTATION OF BINARY AND FLOATING POINT CHROMOSOME REPRESENTATION IN GENETIC ALGORITHM

Marin Golub
Faculty of Electrical Engineering and Computing, University of Zagreb
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Unska 3, 10000 Zagreb, Croatia
e-mail: golub@zemris.fer.hr

Abstract: *This paper describes the implementation details and compares two methods for optimisation of multi-dimensional cost functions. The implemented genetic algorithm uses two chromosome representations: binary and floating point. In both representations the algorithm is based on steady-state reproduction, roulette-wheel bad individuals selection and has the same parameters.*

Key words: genetic algorithm, chromosome representation, steady-state reproduction

1. Introduction

Genetic algorithms are very practical and robust optimisation methods [1],[3],[6]. The general idea is the simulation of natural evolution in the search for the optimal solution of a given problem. In nature individuals are competing to survive. In this competition, only the fittest individuals survive and reproduce. Therefore, the genes of the fittest survive, while the genes of weaker individuals die out. Genetic material is saved into the chromosomes. Parents' genetic material is mixed during reproduction and the offspring has genes of both parents. Also, an individual's genetic material can be changed by mutation. Mutation is a random change of one or more genes.

Implementation of this natural process is suitable as an optimisation method. Figure 1 shows the structure of a genetic algorithm and an evolution program in general.

```
procedure Evolution program
begin
|   initialize population
|   while termination criterion is not reached
|   begin
|   |   evaluate population
|   |   select individuals for next population
|   |   perform crossover and mutation
|   end
end
```

Figure 1: Structure of the genetic algorithm

When we simulate the natural genetic algorithm, each individual (chromosome) represents a potential solution of a given problem. The GA spends most of the time on evaluating population. Therefore,

the representation of chromosome is very important: it could be an array of bits, a number, an array of numbers, a matrix, a string of characters or any other data structure. A chromosome must satisfy given precision and constraints [2],[3],[5],[6]. And, of course, it has to be suitable for the implementation of genetic operators.

In section 2 the implemented genetic algorithm with steady-state reproduction is described. In sections 3 and 4 chromosome representations are described. Test function and experimental results are described in sections 5 and 6, respectively.

2. Implemented genetic algorithm

In the binary and floating-point chromosome representation the implemented GA is the same and has the same parameters: the size of population, the number of eliminated individuals and the number of mutations per generation. The difference between the two methods is in chromosome representation and specific genetic operators.

The initialisation process is very simple: it creates a population of *POP_SIZE* individuals (chromosomes), where each chromosome is initialised randomly.

Evaluation function *eval* is defined as the difference between function f and the minimal value of function f in current population: $eval(x) = f(x) - \min_i \{ f(x_i) \}$, $i=1,2,...POP\_SIZE$. This procedure is called windowing [1] where minimum is equal to zero (minimum evaluation value is zero). It eliminates negative evaluation values and protects the algorithm from becoming a random search process if all individuals have approximately the same large function values [1].

The selection process saves the best individual. Figure 2 shows the steady-state reproduction with roulette-wheel bad individual selection. Steady-state reproduction replaces only M individuals. On the other hand, the generation replacement technique replaces its entire set of parents by their children. Steady-state reproduction has one parameter: M - the number of new chromosomes to create [1],[5].

```
procedure Genetic algorithm with steady-state reproduction
begin
|   initialize population
|   while termination criterion is not reached
|   begin
|   |   calculate elimination probability for each individual
|   |   turn roulette-wheel M times and each time delete chosen
|   |       individual
|   |   mate survived individuals and substitute M eliminated
|   |       individuals by their children
|   end
end
```

Figure 2: Genetic algorithm with steady-state reproduction

The elimination probability p_e is proportional to individual's elimination fitness: $elimination\_fitness = \max_i \{ eval(x_i) \} - eval(x)$. Because the elimination probability of the best individual is zero, that selection always saves the best individual.

For this particular problem the following parameters are used: population size: *POP_SIZE=10,30,60,100*, number of eliminated individuals: *M=POP_SIZE/2* and number of mutations per generation: *N_M@ POP_SIZE/3*. For the one-dimensional problem, number of populations (iterations) is set to *500*, and for two and ten-dimensional problems it is set to *1000*.

3. The binary implementation

In the binary implementation each chromosome consists of a binary vector that represents real value of the variable x. If the problem is multidimensional, the chromosome consists of an array of binary vectors. The length n of the binary vector depends on the required precision. The domain of the variable x is *[LB,UB]* where *LB,UBÎ Â*. Vector 0...00 (n bits and all bits are set to zero) represents value *LB* and vector 1...11 represents value *UB*. Any other binary vector $b_{n-1}b_{n-2}...b_2b_1b_0$ represents value x with precision Dx:

$$x = LB + \frac{\lambda_{i=0}^{n-1} b_i 2^i}{2^n - 1}(UB - LB), \qquad \Delta x = \frac{UB - LB}{2^n - 1} \qquad (1),(2)$$

The precision of the binary representation can be extended by increasing the number of bits per chromosome, but this considerably slows down the algorithm [5].

We used uniform crossover: each bit of a new child is taken from one of the parents randomly. Figure 3 shows uniform crossover procedure.

```
procedure Uniform crossover
begin
  for i=1 to n
    child[bi]=choose_one_by_equal_chance(parent1[bi],parent2[bi])
end
```

Figure 3: Uniform crossover

The mutation is a random change of one bit. The mutation probability of the best individual is zero and for the other individuals it is:

$$p_m = \frac{N\_M}{(POP\_SIZE - 1) * n}, \qquad (3)$$

where *N_M* is the number of mutations and n is a number of bits.

4. The floating point implementation

In the floating point implementation the chromosome consists of a floating point number. Of course, if the given function has more than one variable, then the chromosome consists of an array of floating point numbers. The precision of such an approach depends on the floating point number implementation.

The crossover operator is defined as: $x_{child} = rand\_number\_in\_range(x_{parent1}, x_{parent2})$.

The simple mutation operator defined by: $x = rand\_number\_in\_range(LB,UB)$ gives bad experimental results in fine local tuning. The reason of such behaviour of the algorithm with simple mutation operator is the very low probability that after mutation an element will fall within a small interval d of the domain range [5]:

$$p_d = \frac{d}{UB - LB}, \tag{4}$$

where $d << (UB-LB)$ T $p_d \approx 0$.

The non-uniform mutation operator depends on the time, that is, on the generation number t. The probability of mutation is constant, but mutation scope isn't and it changes with time. If chromosome x_k is selected for mutation then the new chromosome x_k' is calculated as:

$$x_k' = rand\_number\_in\_range(LBM,UBM) \tag{5}$$

where $LBM = \max\{LB, x_k - (UB - LB)r(t)\}$, $UBM = \min\{UB, x_k + (UB - LB)r(t)\}$, and $r(t) = 1 - s^{\left(1-\frac{t}{T}\right)^b}$, where s is a random number from the interval $[0,1]$, b is a system parameter determining the degree of dependency on iteration number (we used $b=5$) and T is the maximal generation number. This function enables decreasing the search scope and the procedure idea is borrowed from simulated annealing [4]. If t is small value, then the search scope is a random number between 0 and $UB-LB$. At the other hand, at the end of time ($t \to T$) search scope tends to zero (*search scope* $\to 0$) [5].

5. Test function

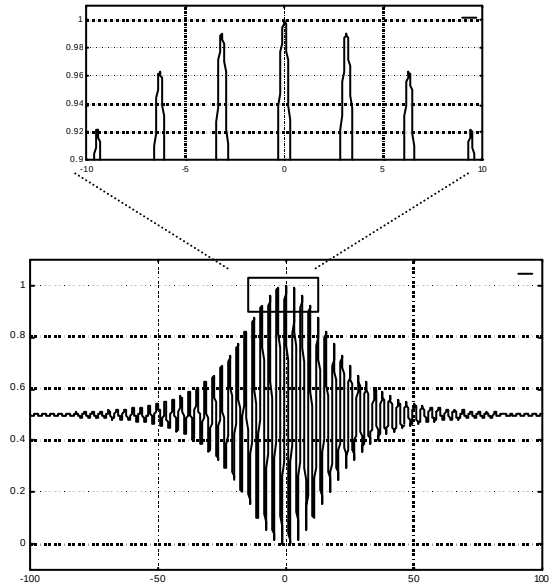

Figure 4: 1D test function

$$f(x) = 0.5 - \frac{\sin^2(x) - 0.5}{\left(1 + 0.001 * x^2\right)^2},$$

$$x \in [-100,100]$$

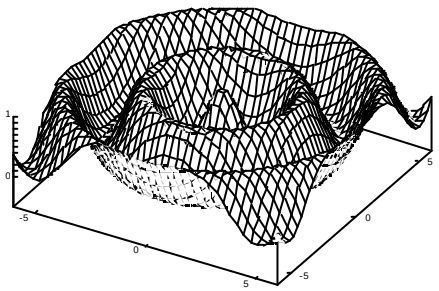

Figure 5: Segment of 2D test function

$$f(x,y) = 0.5 - \frac{\sin^2\sqrt{x^2 + y^2} - 0.5}{\left[1 + 0.001\left(x^2 + y^2\right)\right]^2},$$

$$x, y \in [-100,100]$$

The test function [1] in *N*-dimensional space is given as:

$$f(\vec{x}) = 0.5 - \frac{\sin^2\sqrt{\sum_{i=1}^{N} x_i^2} - 0.5}{\left[1 + 0.001\sum_{i=1}^{N} x_i^2\right]^2}, \qquad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}, \qquad x_1, x_2, \dots, x_N \in [-100,100]$$

The global maximum of the given problem is $f(\vec{0}) = 1$. Figure 4 and Figure 5 show function $f(\vec{x})$ in one and two dimensional space respectively.

6. Experimental Results

For one and two-dimensional problems both methods give very similar results, however, the GA with floating point representation is 60% faster then the GA with binary representation. The differences between two GA methods are larger if the search space is three or more dimensional. For the ten-dimensional problem GA with floating point implementation gives slightly better solutions in a three times shorter time than the GA with binary implementation.

dimension of problem	1D				2D		10D	
population size	10		30		60		100	
number of iterations	500		500		1000		1000	
number of experiments	1000		1000		100		100	
chromosome representation	Binary	FP	Binary	FP	Binary	FP	Binary	FP
reach global optimum *f(x)=1*	96.7%	98.2%	100%	100%	16%	8%	none	none
reach first local optimum *f(x)@0.99*	3.3%	1.8%	none	none	84%	92%	none	none
reach other local optimum where *f(x)>0.8*	none	none	none	none	none	none	53%	76%
$\overline{d}$ - average deviation	3.61E-4	1.96E-4	1E-5	7.6E-5	0.0083	0.0089	0.217	0.167
σ - standard deviation	0.0018	0.0013	1.9E-4	6.65E-4	0.0036	0.0024	0.093	0.062
CPU time in seconds for 100 experiments	168	103	432	269	2825	1827	23878	7891

Table 1: Experimental optimisation results

Deviation *d* is calculated for each experiment as: $d = \left| f(\vec{0}) - f(\vec{x}_{\max}) \right|$. Last row in the table shows how much CPU-time a SUN SPARCstation 2 (4/75) with 32 MB RAM spends for 100 experiments. The parameters of GA were not optimised. For a particular set of parameters results would be better.

We should expect better results if we shrink the domain range. For example, let it be: $x_1, x_2, \dots, x_N \in [0,100]$. Table 2 shows results of the optimisation of the same test function, but with a smaller domain range. The GA with floating point chromosome representation almost becomes a random search process. The reason for such algorithm behaviour lies in bad crossover operator. If we choose any two chromosomes from the domain range (neither of them is the global optimum), the defined crossover can't produce the global optimum, because the global optimum is on the edge of the domain range.

dimension of problem	1D		2D	
population size	30		60	
number of iterations	500		1000	
number of experiments	100		100	
chromosome representation	Binary	FP	Binary	FP
reach global optimum *f(x)=1*	100%	77%	23%	none
reach first local optimum *f(x)@0.99*	none	23%	77%	74%
reach other local optimum where *f(x)>0.8*	none	none	none	26%
$\overline{d}$ - average deviation	≅0	0.0025	0.0075	0.0135
σ - standard deviation	≅0	0.038	0.004	0.0089

Table 2: Experimental optimisation results with smaller domain range: $x_1, x_2, \ldots, x_N \in [0,100]$

7. Conclusion

The GA with floating point chromosome representation is simpler for implementation and it is faster than GA with binary representation. There are two reasons for that:

1) For the binary implementation, the algorithm must have a conversion mechanism that could convert a bit string (chromosome) to the real value. The floating point implementation does not need such a mechanism, because the chromosome is at the same time a real value.
2) Genetic operators crossover and mutation defined over floating point values are simpler and faster than the uniform crossover and mutation over bit strings.

For multidimensional problem the GA with floating point chromosome representation gives slightly better results then the GA with binary representation (Table 1). The solution of the problem of fine local tuning in floating point representation lies in non uniform mutation.

On the other hand, the GA with floating point chromosome representation almost becomes a random search process, if, for a particular problem, the crossover operator can't produce the optimal solution (see results in the Table 2).

References:

[1] Davis, L. (1991), *Handbook of Genetic Algorithms,* Van Nostrand Reinhold, New York.

[2] Filhio, J.L.R., Treleaven, P.C., Alippi, C. (1994), *"Genetic-Algorithm Programming Environments"*, Computer, Vol. 27-6, pp. 28-43., June 1994.

[3] Goldberg, D.E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley.

[4] Laarhoven, P.J.M., Aarts, E.H.L. (1987), *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Company.

[5] Michalewicz, Z. (1994), *Genetic Algorithms + Data Structures = Evolution Programs,* Springer-Verlag, Berlin.

[6] Srinivas, M., Patnaik, L.M. (1994), *"Genetic Algorithms: A Survey"*, Computer, Vol. 27-6, pp.17-26, June 1994.