Polymorphic Random Building Block Operator for Genetic Algorithms

Ghodrat Moghadampour
VAMK, University of Applied Sciences, Technology and Communication, Wolffintie 30, 65200, Vaasa, Finland

Keywords: Evolutionary Algorithm, Genetic Algorithm, Function Optimization, Mutation Operator, Multipoint Mutation Operator, Polymorphic Random Building Block Operator, Fitness Evaluation and Analysis.

Abstract: Boosting the evolutionary process of genetic algorithms by generating better individuals, avoiding stagnation at local optima and refreshing population in a desirable way is a challenging task. Typically operators are used to achieve these objectives. On the other hand using operators can become a challenging task in itself if applying them requires setting many parameters through human intervention. Therefore, developing operators, which do not require human intervention and at the same time are capable of assisting the evolutionary process, is highly desirable. Most typical genetic operators are mutation and crossover. However, experience has proved that these operators in their classical form are not capable of refining the population efficiently enough. In this work a new dynamic mutation operator called polymorphic random building block operator with variable mutation rate is proposed. This operator does not require any pre-fixed parameter. It randomly selects a section from the binary presentation of the individual, then generates a random bit-string of the same length as the selected section and applies bitwise logical AND, OR and XOR operators between the randomly generated bit-string and the selected section from the individual. In the next step all three newly generated offspring will go through selection procedure and will replace a possibly worse individual in the population. Experimentation with 33 test functions and 11550 test runs proved the superiority of the proposed dynamic mutation operator over single-point mutation operator with 1%, 5% and 8% mutation rates and the multipoint mutation operator with 5%, 8% and 15% mutation rates.

1 INTRODUCTION

Most often genetic algorithms (GAs) have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

A simple GA works as follows: 1) A population of n l-bit strings (chromosomes) is randomly generated, 2) the fitness $f(x)$ of each chromosome x in the population is calculated, 3) chromosomes are selected to go through crossover and mutation operators with p_c and p_m probabilities respectively, 4) the old population is replaced by the new one, 5) the process is continued until the termination conditions are met.

However, more sophisticated genetic algorithms typically include other intelligent operators, which apply to the specific problem. In addition, the whole algorithm is normally implemented in a novel way with user-defined features while for instance measuring and controlling parameters, which affect the behaviour of the algorithm.

1.1 Genetic Operators

For any evolutionary computation an appropriate representation (encoding) of problem variables must be chosen along with the appropriate evolutionary computation operators. Data might be represented in different formats: binary strings, real-valued vectors, permutations, finite-state machines, parse trees and so on.

Decision on what genetic operators to use greatly depends on the encoding strategy of the GA. For each representation, several operators might be employed (Michalewicz, 2000). The most commonly used genetic operators are crossover and mutation.

Moghadampour G..
Polymorphic Random Building Block Operator for Genetic Algorithms.
DOI: 10.5220/0004014103420348
In *Proceedings of the 14th International Conference on Enterprise Information Systems* (ICEIS-2012), pages 342-348
ISBN: 978-989-8565-10-5

1.1.1 Crossover

The simplest form of crossover is single-point: a single crossover position is chosen randomly and the parts of the two parents after the crossover position are exchanged to form two new individuals (offspring). The idea is to recombine building blocks (schemas) on different strings.

In two-point crossover, two positions are chosen at random and the segments between them are exchanged. Two-point crossover reduces positional bias and endpoint effect, it is less likely to disrupt schemas with large defining lengths, and it can combine more schemas than single-point crossover (Mitchell, 1998). Two-point crossover has also its own shortcomings; it cannot combine all schemas.

Multipoint-crossover has also been implemented, e.g. in one method, the number of crossover points for each parent is chosen from a Poisson distribution whose mean is a function of the length of the chromosome. Another method of implementing multipoint-crossover is the "parameterized uniform crossover" in which each bit is exchanged with probability p, typically $0.5 \le p \le 0.8$ (Mitchell, 1998).

In parameterized uniform crossover, any schemas contained at different positions in the parents can potentially be recombined in the offspring; there is no positional bias. This implies that uniform crossover can be highly disruptive of any schema and may prevent coadapted alleles from ever forming in the population (Mitchell, 1998).

The one-point and uniform crossover methods have been combined by some researchers through extending a chromosomal representation by an additional bit. There has also been some experimentation with other crossovers: segmented crossover and shuffle crossover (Eshelman et al., 1991; Michalewicz, 1996).

Segmented crossover, a variant of the multipoint, allows the number of crossover points to vary. The fixed number of crossover points and segments (obtained after dividing a chromosome into pieces on crossover points) are replaced by a segment switch rate, which specifies the probability that a segment will end at any point in the string.

The shuffle crossover is an auxiliary mechanism, which is independent of the number of the crossover points. It 1) randomly shuffles the bit positions of the two strings in tandem, 2) exchanges segments between crossover points, and 3) unshuffles the string (Michalewicz, 1996). In gene pool recombination, genes are randomly picked from the gene pool defined by the selected parents.

1.1.2 Mutation

The common mutation operator used in canonical genetic algorithms to manipulate binary strings $a = (a_1,...a_\ell) \in I = \{0,1\}^\ell$ of fixed length ℓ was originally introduced by Holland (Holland, 1975) for general finite individual spaces $I = A_1 \times ... A_\ell$, where $A_i = \{\alpha_{i_1},...,\alpha_{i_{k_l}}\}$. By this definition, the mutation operator proceeds by:

i. determining the position $i_1,...,i_h (i_j \in \{1,...,l\})$ to undergo mutation by a uniform random choice, where each position has the same small probability p_m of undergoing mutation, independently of what happens at other position
ii. forming the new vector $a'_i = (a_1,...,a_{i_1-1}, a'_{i_1}, a_{i_1+1},..., a_{i_h-1}, a'_{i_h}, a_{i_h+1},...a_\ell)$, where $a'_i \in A_i$ is drawn uniformly at random from the set of admissible values at position i.

The original value a_i at a position undergoing mutation is not excluded from the random choice of $a'_i \in A_i$. This implies that although the position is chosen for mutation, the corresponding value might not change at all (Bäck et al., 2000).

Mutation rate is usually very small, like 0.001 (Mitchell, 1998). A good starting point for the bit-flip mutation operation in binary encoding is $P_m = 1/L$, where L is the length of the chromosome (Mühlenbein, 1992). Since $1/L$ corresponds to flipping one bit per genome on average, it is used as a lower bound for mutation rate. A mutation rate of range $P_m \in [0.005, 0.01]$ is recommended for binary encoding (Ursem, 2003). For real-value encoding the mutation rate is usually $P_m \in [0.6, 0.9]$ and the crossover rate is $P_m \in [0.7, 1.0]$ (Ursem, 2003).

While recombination involves more than one parent, mutation generally refers to the creation of a new solution from one and only one parent. Given a real-valued representation where each element in a population is an n-dimensional vector $x \in \Re^n$, there are many methods for creating new offspring using mutation. The general form of mutation can be written as:

$$x' = m(x) \quad (1)$$

where x is the parent vector, m is the mutation function and x' is the resulting offspring vector. The more common form of mutation generated offspring vector:

$$x' = x + M \quad (2)$$

where the mutation M is a random variable. M has often zero mean such that

$$E(x') = x \quad (3)$$

the expected difference between the real values of a parent and its offspring is zero (Bäck et al., 2000).

Some forms of evolutionary algorithms apply mutation operators to a population of strings without using recombination, while other algorithms may combine the use of mutation with recombination. Any form of mutation applied to a permutation must yield a string, which also presents a permutation. Most mutation operators for permutations are related to operators, which have also been used in neighbourhood local search strategies (Whitley, 2000). Some other variations of the mutation operator for more specific problems have been introduced in (Bäck et al., 2000). Some new methods and techniques for applying crossover and mutation operators have also been presented in (Moghadampour, 2006).

1.1.3 Other Operators and Mating Strategies

In addition to common crossover and mutation some other operators are used in GAs including inversion, gene doubling and other operators for preserving diversity in the population. For instance, a "crowding" operator has been used in (De Jong, 1975; Mitchell, 1998) to prevent too many similar individuals ("crowds") from being in the population at the same time. This operator replaces an existing individual by a newly formed and most similar offspring.

In (Mengshoel et al., 2008) a probabilistic crowding niching algorithm in which subpopulations are maintained reliably, is presented. It is argued that like the closely related deterministic crowding approach, probabilistic crowding is fast, simple, and requires no parameters beyond those of classical genetic algorithms.

Diversity in the population can also be promoted by putting restrictions on mating. For instance, distinct "species" tend to be formed if only sufficiently similar individuals are allowed to mate (Mitchell, 1998). Another attempt to keep the entire population as diverse as possible is disallowing mating between too similar individuals, "incest" (Eshelman et al., 1991; Mitchell, 1998).

Another solution is to use a "sexual selection" procedure; allowing mating only between individuals having the same "mating tags" (parts of the chromosome that identify prospective mates to one another). These tags, in principle, would also evolve to implement appropriate restrictions on new prospective mates (Holland, 1975).

Another solution is to restrict mating spatially. The population evolves on a spatial lattice, and individuals are likely to mate only with individuals in their spatial neighborhoods. Such a scheme would help preserve diversity by maintaining spatially isolated species, with innovations largely occurring at the boundaries between species (Mitchell, 1998).

The efficiency of genetic algorithms has also been tried by imposing adaptively, where the algorithm operators are controlled dynamically during runtime (Eiben et al. 2008). These methods can be categorized as deterministic, adaptive, and self-adaptive methods (Eiben & Smitt, 2007; Eiben et al. 2008). Adaptive methods adjust the parameters' values during runtime based on feedback from the algorithm (Eiben et al. 2008), which are mostly based on the quality of the solutions or speed of the algorithm (Smit et al., 2009).

2 THE POLYMORPHIC RANDOM BUILDING BLOCK OPERATOR

The *polymorphic random building block* (PRBB) operator is a new self-adaptive operator proposed here. The random building block (*RBB*) operator was originally presented in (Moghadampour, 2006; Moghadampour, 2011; Moghadampour, 2012), where promising results were also reported.

In this paper we modify the original idea of the operator by applying multiple logical bitwise operators, namely AND, OR and XOR during mutation process in order to produce new offspring. During the classical crossover operation, building blocks of two or more individuals of the population are exchanged in the hope that a better building block from one individual will replace a worse building block in the other individual and improve the individual's fitness value. However, the polymorphic random building block operator involves only one individual.

The polymorphic random building block operator resembles more the multipoint mutation operator, but it lacks the frustrating complexity of such an operator. The reason for this is that the random building block operator does not require any pre-defined parameter value and it automatically

takes into account the length (number of bits) of the individual at hand. In practice, the polymorphic random building block operator selects a section (s_1) of random length (l_s) from the binary presentation of the individual at hand. In the next step the operator produces randomly a binary string (s_2) of the same size (l_s) and then applies AND, OR and XOR bitwise operators between s_1 and s_2 in turn in order to produce three new offspring. In the next step these newly generated offspring go through selection procedure one by one to be either selected or discarded.

This operator can help breaking the possible deadlock when the classic crossover operator fails to improve individuals. It can also refresh the population by injecting better building blocks into individuals, which are not currently found from the population. Figure 1 describes the random building block operator.

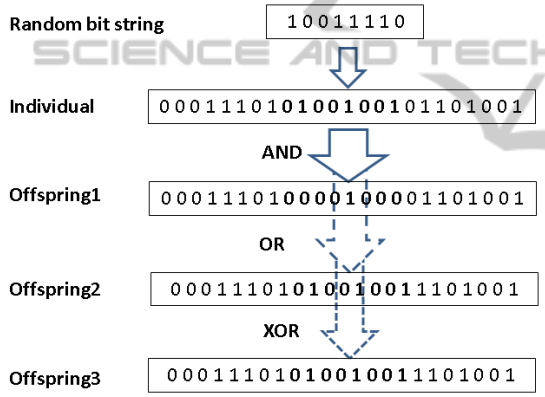

Figure 1: The polymorphic random building block operator. A random building block is generated and is combined with the individual through AND, OR and XOR operators to generate three new offspring.

This operation is implemented in the following order: 1) for each individual ind of binary length l in the population a section length l_s proportionate to the number of variables in the problem is randomly generated so that $1 \le l_s \le \frac{l}{number\_of\_variables}$, 2) two crossover points cp_1 and cp_2 are randomly selected so that $l_s = |cp_2 - cp_1|$, 3) a random bit string $bstr$ of length l_s is generated, 4) bits between the crossover points on the individual ind go through bitwise AND, OR and XOR logical operators with the bits on the bit string $bstr$ to generate three new offspring, and 5) each newly generated offspring go through the selection procedure.

2.1 Survivor Selection

After each operator application, new offspring are evaluated and compared to the population individuals. Newly generated offspring will replace the worst individual in the population if they are better than the worst individual. Therefore, the algorithm is a steady state genetic algorithm.

3 EXPERIMENTATION

The random building block operator, three versions of single-point mutation operator (with 1%, 5% and 8% mutation rates) and three versions of multipoint mutation operator (with 5%, 8% and 15% mutation rates) were implemented as prat of a genetic algorithm to solve the following demanding minimization problems: Ackley's ($\forall x_i : -32.768 \le x_i \le 32.768$), Colville's ($\forall x_i : -10 \le x_i \le 10$), Griewank's F1 ($\forall x_i : -600 \le x_i \le 600$), Rastrigin's ($\forall x_i : -5.12 \le x_i \le 5.12$), Rosenbrock's ($\forall x_i : -100 \le x_i \le 100$) and Schaffer's F6 ($\forall x_i : -100 \le x_i \le 100$). Some of these functions have a fixed number of variables and some others are multidimensional in which the number of variables could be determined by the user. For multidimensional problems with an optional number of dimensions (n), the algorithm was tested for $n = 1, 2, 3, 5, 10, 30, 50, 100$. The exception to this was the Rosenbrock's function for which the minimum number of variables is 2. The efficiency of each of the operators in generating better fitness values was studied.

During experimentation only one operator was tested at each time. To simplify the situation and clarify interpretation of experimentation results the operators were not combined with other operators, like crossover.

Single-point mutation operator was implemented so that the total number of mutation points ($total\_mut\_points$) was calculated by multiplying the mutation rate (m_rate) by the binary length of the individual (ind_bin_length) and the population size (pop_size):

$$total\_mut\_points = m\_rate \times ind\_bin\_length \times pop\_size \quad (4)$$

Then during each generation for the total number of mutation points one gene was randomly selected from an individual in the population and mutated. Multipoint mutation operator was implemented so that during each generation for the total number of

mutation points ($total\_mut\_points$) a random number of mutation points (sub_mut_points) from a random number of individuals in the population was selected and mutated. This process was continued until the total number of mutation points was consumed:

$$total\_mut\_points = \sum_{i=1}^{n} sub\_mut\_points_i \quad (5)$$

For each test case the steady-state algorithm was run for 50 times. The population size was set to 9 and the maximum number of function evaluations for each run was set to 10000. The exception to this was the Rosenbrock's function for which the number of function evaluations was set to 100000 in order to get some reasonable results.

The mapping between binary strings into floating-point numbers and vice versa was implemented according to the following well-known steps:

1. The distance between the upper and the lower bounds of variables is divided according to the required precisions, $precision$ (e.g. the precision for 6 digits after the decimal point is $1000000_{(10)}$) in the following way:

$$(upperbound - lowerbound) \times precision \quad (6)$$

2. Then an integer number l is found so that:

$$(upperbound - lowerbound) \times precision \le 2^l \quad (7)$$

Thus, l determines the length of binary representation, which implies that each chromosome in the population is l bits long. Therefore, if we have a binary string x' of length l, in order to convert it to a real value x, we first convert the binary string to its corresponding integer value in base 10, $x'_{(10)}$ and then calculate the corresponding floating-point value x according to the following formula:

$$x = lowerbound + x'_{(10)} \times \frac{upperbound - lowerbound}{2^l - 1} \quad (8)$$

The variable and solution precisions set for different problems were slightly different, but the same variable and solution precisions were set the same for all operators. During each run the best fitness value achieved during each generation was recorded. This made it possible to figure out when the best fitness value of the run was actually found. Later at the end of 50 runs for each test case the average of the best fitness values and the required function evaluations were calculated for comparison. In the following, test results for comparing the efficiency of polymorphic random building block operator with different versions of mutation operator are reported.

Experimentation results indicated that the polymorphic random building block operator had produced much better results than different versions of the single-point mutation operator in all test cases. The difference in performance seemed to be significant for Colville's function and Ackley's and Griewank's functions when the number of variables increases.

Very low p-values for T-test and F-test indicated that the performance values achieved by Polymorphic Random Building Block operator were significantly smaller than the ones achieved by other operators.

The performance of the polymorphic random building block operator against the single-point mutation operator was also tested on Rastrigin's, Rosenbrock's and Schaffer's F6 functions.

Studying results proved that the polymorphic random building block operator has been able to produce significantly better results in more than 87% of test cases. The results indicated that for Rosenbrock50 and Rosenbrock 100 the polymorphic random building block had on average produced worse results than the single mutation point operator. However, studying the results showed that there are huge differences between the median values (in parentheses) of test results for the benefit of the polymorphic random building block. While the median values for polymorphic random building block operator were less than the average values, the situation was vice versa in all cases for different versions of single point mutation operators. For Rosenbrock50 in 58% of test cases the fitness value achieved by polymorphic random building block operator was less than 351, which is the average of fitness values achieved by single mutation operator with 8% mutation rate. This means that in 58% of test cases polymorphic random building block had a better performance in finding the best fitness value for Rosenbrock's function with 50 variables.

For Rosenbrock100 in 60% of test cases the fitness value achieved by polymorphic random building block operator was less than 342, which is the average of fitness values achieved by single mutation operator with 8% mutation rate. This means that in 60% of test cases polymorphic random building block had a better performance over mutation operator with 1% and 5% mutation rates in finding the best fitness value for Rosenbrock's function with 100 variables.

Very low p-values for T-test and F-test indicated that the performance values achieved by polymorphic random building block operator are significantly smaller than the ones achieved by other operators.

Analysis showed that the differences between average fitness values achieved with different operators wer not significant for Rosenbrock's function with 50 and 100 variables. The superiority of polymorphic random building block operator becomes clear if we recall that it produced in most cases better results than the average values achieved by other operators.

The performance of the polymorphic random building block operator was also compared against the multipoint mutation operator in which several points of the individual were mutated during each mutation operator. As it was earlier mentioned the number of points to be mutated during each mutation operation was randomly determined. Mutation cycles were repeated until total mutation points were utilized. Clearly, the total number of mutation points was determined by the mutation rate, which was 5%, 8% and 15% for different experimentations.

Comparing results proved that the fitness values achieved by the building block operator were better than the ones achieved by different versions of multipoint mutation operator in all cases. Differences between the average fitness values achieved for Ackley's and Griewank's functions with 30, 50 and 100 variables by the polymorphic random building block and different versions of multipoint mutation operator were even more substantial.

Very low p-values for T-test and F-test indicated that the performance values achieved by polymorphic random building block operator were significantly smaller than the ones achieved by other operators.

The performance of the polymorphic random building block operator against the multipoint mutation operator was also tested on Rastrigin's, Rosenbrock's and Schaffer's F6 functions.

Experimentation showed that the polymorphic random building block operator had also outperformed multipoint mutation operator with 5%, 8% and 15% mutation rates. In most cases differences in performance were huge in favour of polymorphic random building block operator.

A small p-value for T-test and very low p-value for F-test indicate that the performance values achieved by polymorphic random building block operator were significantly smaller than the ones achieved by other operators.

4 CONCLUSIONS

In this paper a dynamic mutation operator; polymorphic random building block operator for genetic algorithms was proposed. The operator was tested against single-point mutation operator with 1%, 5% and 8% mutation rates and multipoint mutation operator with 5%, 8% and 15% mutation rates.

Comparing test results revealed that the polymorphic random building block operator was capable of achieving better fitness values within less function evaluations compared to different versions of single-point and multipoint mutation operators. The fascinating feature of polymorphic random building block is that it is dynamic and therefore does not require any pre-set parameter.

However, for mutation operators the mutation rate and the number of mutation points should be set in advance. The polymorphic random building block can be used straight off the shelf without needing to know its best recommended rate. Hence, it lacks frustrating complexity, which is typical for different versions of the mutation operator.

Therefore, it can be claimed that the polymorphic random building block is superior to the mutation operator and capable of improving individuals in the population more efficiently.

4.1 Future Research

The proposed operator can be combined with other operators and applied to new problems and its efficiency in helping the search process can be evaluated more thoroughly with new functions. Moreover, the polymorphic random building block operator can be adopted as part of the genetic algorithm to compete with other state-of-the-art algorithms on solving more problems.

REFERENCES

Bäck, Thomas, David B. Fogel, Darrell Whitely & Peter J. Angeline, 2000. Mutation operators. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

De Jong, K. A., 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph.D. thesis, University of Michigan. Michigan: Ann Arbor.

Eiben, A. and J. Smith, 2007. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2nd edition.

Eiben, G. and M. C. Schut, 2008. *New Ways To Calibrate Evolutionary Algorithms*. In *Advances in Metaheuristics for Hard Optimization*, pages 153–177.

Eshelman, L. J. & J. D. Schaffer, 1991. Preventing premature convergence in genetic algorithms by preventing incest. In *Proceedings of the Fourth International Conference on Genetic Algorithms*. Eds. R. K. Belew & L. B. Booker. San Mateo, CA : Morgan Kaufmann Publishers.

Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor: MI: University of Michigan Press.

Mengshoel, Ole J. & Goldberg, David E., 2008. *The crowding approach to niching in genetic algorithms*. Evolutionary Computation, Volume 16 , Issue 3 (Fall 2008). ISSN:1063-6560.

Michalewicz, Zbigniew (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Third, Revised and Extended Edition. USA: Springer. ISBN 3-540-60676-9.

Michalewicz, Zbigniew, 2000. Introduction to search operators. In *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Mitchell, Melanie, 1998. *An Introducton to Genetic Algorithms*. United States of America: A Bradford Book. First MIT Press Paperback Edition.

Moghadampour, Ghodrat (2006). *Genetic Algorithms, Parameter Control and Function Optimization: A New Approach*. PhD dissertation. ACTA WASAENSIA 160, Vaasa, Finland. ISBN 952-476-140-8.

Moghadampour, Ghodrat (2011). *Random Building Block Operator for Genetic Algorithms*. 13th International Conference on Enterprise Information Systems (ICEIS 2011), 08 - 11 June 2011Beijing – China.

Moghadampour, Ghodrat (2012). *Outperforming Mutation Operator with Random Building Block Operator in Genetic Algorithms*. In *Enterprise Information Systems International Conference, ICEIS 2011* Beijing, China, June 8-11, 2011 Revised Selected Papers. Eds. Runtong Zhang, Zhenji Zhang, Juliang Zhang, Joaquim Filipe and José Cordeiro. Springer-Verlag LNBIP Series book.

Mühlenbein, H., 1992. How genetic algorithms really work: 1. mutation and hill-climbing. In: *Parallel Problem Solving from Nature 2*. Eds R. Männer & B. Manderick. North-Holland.

Smit, S. K. and Eiben, A. E., 2009. *Comparing Parameter Tuning Methods for Evolutionary Algorithms*. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 399–406, May 2009.

Spears, W. M., 1993. Crossover or mutation? In: *Foundations of Genetic Algorithms 2*. Ed. L. D. Whitely. Morgan Kaufmann.

Ursem, Rasmus K., 2003. *Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization (PhD Dissertation)*. A Dissertation Presented to the Faculty of Science of the University of Aarhus in Partial Fulfillment of the Requirements for the PhD Degree. Department of Computer Science, University of Aarhus, Denmark.

Whitley, Darrell, 2000. Permutations. In *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.