Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Explicit filtering of building blocks for genetic algorithms

C.H.M. van Kemenade

Computer Science/Department of Software Technology

CS-R9647 1996

Report CS-R9647
ISSN 0169-118X

Explicit Filtering of Building Blocks for Genetic Algorithms

C.H.M. van Kemenade

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
kemenade@cwi.nl

Abstract

Genetic algorithms are often applied to building block problems. We have developed a simple filtering algorithm that can locate building blocks within a bit-string, and does not make assumptions regarding the linkage of the bits. A comparison between the filtering algorithm and genetic algorithms reveals some interesting insights, and we discuss how the filtering algorithm can be used to build a powerful hybrid genetic algorithm.

AMS Subject Classification (1991): 68T20
CR Subject Classification (1991): G.1.7, I.2.8
Keywords & Phrases: genetic algorithms, optimization
Note: This paper was presented at the Parallel Problem Solving from Nature conference PPSN-IV, Berlin, Germany, 1996

1. Introduction

Genetic algorithms (GA's) with bit-based representation are usually regarded as general solvers for bit-coded problems. An interesting class of problems for a GA are the building block based problems. Solutions of such problems can be decomposed in a number of independent building blocks. These building blocks can be discovered separately, and then merged to create a good solution. The schema theorem [4, 7] describes how a canonical GA behaves on this class of problems. A schema is a string over the alphabet $\{\#, 0, 1\}$ of length n, where n is the length of an individual, which encodes a complete solution. A # is a so-called don't-care symbol, which can represent either 0 or 1. During a single generation of a GA 3^n schemata are processed simultaneously. Because each individual in the population is an instance of 2^n schemata, we get the so-called implicit parallelism [4], also called intrinsic parallelism [7]. Recently there has been discussion about the generality of the building block hypothesis, and therefore the schema theorem.

The ultimate goal of the AI-scientist is to create the general problem solver. Such efforts can be expected to fail, as the range of possible problems is too large: see for example the no free lunch theorem for optimization [10, 14]. Until now all efforts to find this general problem solver have failed. Each candidate has a limited class upon which it performs well. Hence for each new problem solver the (probably fuzzy) boundaries of the class upon which it performs well have to be determined. We think it can also be fruitful to take the opposite approach, and first define a broad class of problems. Next a solver is developed which uses knowledge search a solution rapidly. All information which is easily extractable and based on the class definition should be used. A standard GA deviates from this approach, as it only uses fitness of complete individuals to steer the search process. SEARCH, which is an acronym for Search Envisioned As Relation, and Class Hierarchizing, also touches this issue by emphasizing that one has to search for the set of bits that belong to the same building block [8].

In this paper we take the class of building block problems and develop a filtering algorithm using knowledge about this class. This filtering method is shown to perform well on (certain) problem instances. Then we outline how we can combine this method with genetic algorithms.

The rest of this paper is organized as follows. Section 2 discusses some of the reasons why GA's sometimes fail. Section 3 describes and briefly analyses the problems based on the fully deceptive trap functions, which are assumed to be an important representative for a large class of optimization problems. Based on these two sections a new filtering method is developed. This method is compared to GA's and messy GA's in section 5. Finally some conclusions are drawn in section 6.

2. Genetic Algorithms

In this section we discus some of the difficulties for genetic algorithms.

Before applying a GA we have to decide on a representation for solutions to problem we want to handle. An important issue is the linkage. Linkage is said to be tight if bits belonging to the same building block are next to each other on the chromosome, while loose linkage corresponds to a situation where bits belonging to the same building block are scattered over the chromosome. Loose linkage cause problems to GA's using operators that have a positional bias. A positional bias implies that the probability of two bits being taken from the same parent depends upon the (relative) position within the chromosome of these bits [2]. Problems due to linkage have already been studied by Holland [7], and the inversion operator is proposed as a remedy. It has been shown that the inversion operator acts too slow to be useful. Another approach to avoid linkage problems is taken in the (fast) Messy GA's [5, 6, 8], where a different representation without positional bias is introduced. On many (black-box) optimization problems the linkage between bits is not known in advance, so handling loose linkage is of crucial importance for a general optimizer.

Another issue is the number of defined bit-positions of a building block. Even when tight linkage can be assured most genetic operators will introduce a bias. Smaller building blocks are less likely to be disturbed during crossover, and therefore are more likely to be propagated than larger building blocks giving a similar fitness contribution.

Other troubling factors for a GA are genetic hitch-hiking, genetic drift [1], mixing problems [11, 12] and sampling errors due to low order schemata of relative high fitness that are not contained in any of the building blocks. This is for example the case in the fully deceptive trap functions discussed in section 3. Several of these problems are rooted in the iterated character of a GA. A GA continuously applies selection and production to a population which only contains a very small sample of the search-space. A small decision error might easily initiate an avalanche of effects during subsequent iterations.

3. Building Block problems

Many problems involve a search-space which is too large to search it completely. In order to find solutions to large problem instances we have to make some assumptions regarding structures in the search-space, and use these structures to develop a faster optimizer.

An interesting assumption is to assume that a solution is made up of a number of building blocks. If these building blocks can be discovered independently and combined afterwards, we get a tractable problem. A difficult instance of this class can be created by using the parameterized set of fully deceptive trap functions [3]. A fully deceptive trap (sub)function of order k has value [8]

$$f(x) = \begin{cases} k & \text{if } u(x) = k \\ k - u(x) - 1 & \text{otherwise} \end{cases}$$

where $u(x)$ is a function that counts the number of 1-bits in x. The global optimum of this function is the string consisting of k 1-bits resulting in the maximal fitness contribution k. The second best solution is a string consisting of k 0-bits having value $k-1$. As decreasing the number of one

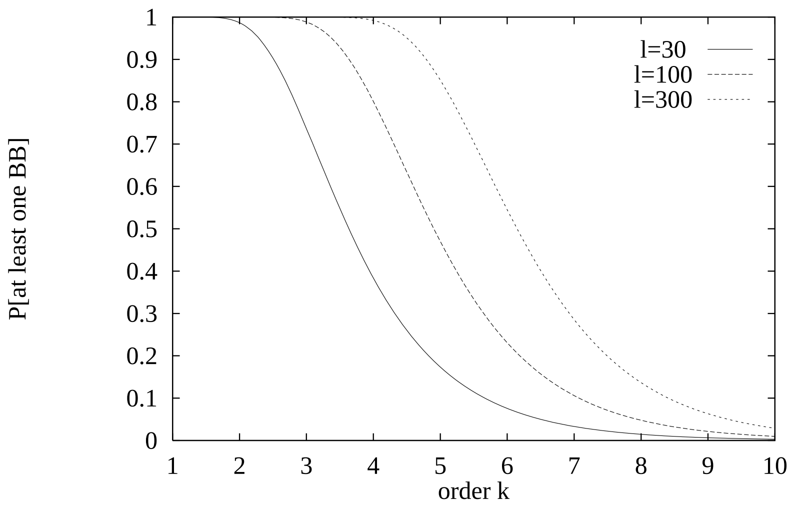


Figure 1: The probability $\mathbb{P}[\textit{at least one BB}]$ for random bit-string coding a solution to a concatenation of fully deceptive trap functions of order k

bits usually increases fitness, except for the optimal string, hill-climbing algorithms will be strongly attracted by the second best optimum.

By concatenating m of these order k subfunctions we create a building block problem, that has a solution which can be represented by a bit-string of length $l = m \times k$. When the bits belonging to the some subfunction are always next to one another we have a building block function with tight linkage. When the bits of a single subfunction are spread over the total bit-string we talk about loose linkage.

Given a random bit-string of length l, the probability that at least one building block is present within this string is

$$\mathbb{P}[\textit{at least one BB}] = 1 - (1 - \frac{1}{2^k})^m \leq \frac{m}{2^k}$$

Figure 1 shows this probability for $l = 30, 100, 300$ as a function of the order of the building blocks k. We see that the probability a building block is present decreases rapidly as k increases. This Figure also shows a number of additional problems for a genetic algorithm. In order to be certain that all building blocks are present in the initial population, a large population is required. Application of mutation will not help us much in this case. A mutation rate of $1/l$, which is commonly used, will concentrate on bit-strings at Hamming distance 1. The only solution seems to be a highly disruptive crossover, such as uniform crossover, which can discover new schemata easily combined with a reasonably high selective pressure in order to prevent the loss of already observed building blocks.

4. Filtering of building blocks

In this section we introduce the filtering algorithm for building block problems.

Informally, the filtering method tries to locate building blocks in a bit-string s. In order to do so it measures the change in fitness when individual bits of s are flipped. Using this information a set of most influential bits is selected which is likely to contain the building blocks present within s.

$$\begin{array}{|c|}\hline 0\\\hline 1\\\hline 1\\\hline 1\\\hline 0\\\hline 1\\\hline\end{array} \xrightarrow[\textit{dfitness}]{\text{measure}} \begin{array}{l}(1,-8)\\(2,-2)\\(3,-6)\\(4,2)\\(5,0)\\(6,-7)\end{array} \xrightarrow[\textit{dfitness}]{\text{sort on}} \begin{array}{r}(1,-8)\\(6,-7)\\(3,-6)\\(2,-2)\\(5,0)\\(4,2)\end{array} \xrightarrow[\substack{\text{significant}\\\text{bits}}]{\text{select}} \begin{array}{l}(1,-8)\\(6,-7)\\(3,-6)\end{array} \xrightarrow[\substack{\text{building}\\\text{block}}]{\text{construct}} \begin{array}{|c|}\hline 0\\\hline \#\\\hline 1\\\hline \#\\\hline \#\\\hline 1\\\hline\end{array}$$

Figure 2: Example of one filtering step

The actual algorithm is as follows

```
function filter($string$, $partial\_sol$)
  $bb \leftarrow \emptyset$
  for all $i$ where $\neg$defined($i, partial\_sol$)
    $bb \leftarrow bb \cup (i, string_i, dfitness(string, i))$
  sort $bb$ on field $dfitness$
  truncate($bb$)
  return $bb$
```

The parameter *partial_sol* is used to carry information regarding the bits have been determined already. The set bb is filled with tuples $(i, b_i, dfitness(b, i))$, where i is the index of a bit, b_i is a bit-value, and $dfitness(b, i)$ is the change in fitness when the value of bit b_i is flipped within string string b. The set bb is ordered on field *dfitness* after which the set is truncated on the position where the largest change in field *dfitness* between subsequent tuples appears. The rationale for this truncation rule is that the set of bits that makes the largest fitness contribution is selected. By truncating on the largest gradient in *dfitness* we enlarge the probability that important building blocks are completely within the residual set bb, without making assumptions regarding the actual fitness contribution of a building block. This truncation rule does not give any guarantees, but if a bit is removed from set bb, then the current value of this bit is not likely to be necessary to maintain a building block present within set bb.

An example of the application of this filtering procedure is shown in Figure 2. On the left we see a bit-string of length 6. Let us assume that the main fitness contribution within this string is coming from a building block containing bits 1, 3 and 6, resulting in a fitness contribution of +6 when the schema 0#1##1 is present. During the first step the individual fitness contribution, *dfitness* of each bit is measured by flipping this bit and observing the change in fitness, and a set of tuples of type (position, *dfitness*) is created. Flipping bit 1, 3 or 6 will break schema 0#1##1 and therefore result in a relatively large value of *dfitness*. During the second step, these tuples are sorted on *dfitness*. Next the significant tuples are selected by truncating the ordered set of tuples on the position of the largest jump in *dfitness*. In our example the largest jump is between the third and the fourth tuple, where *dfitness* increased from -6 to -2. Based on the remaining tuples a candidate building block can be reconstructed.

The filtering procedure does only a detection of building blocks which are present, so in order to operate, this procedure has to be provided bit-strings which are likely to contain building blocks. In order to test the performance of the algorithm the main loop shown in Figure 3 is used. The loop is entered with an empty partial solution. During each iteration it creates a baseline population of size N_{base}, consisting of bit-strings that have random bit-values for those bits which are not defined by the partial solution. The fitness of each such bit-string is calculated, and the best N_{sel} strings from this

```
$BB \leftarrow \emptyset$
$partial\_sol \leftarrow \emptyset$
repeat
   $Base \leftarrow \emptyset$
   /* create a baseline population */
   for $i \leftarrow 1$ to $N_{base}$ do
      $bstring \leftarrow$ random_string($partial\_sol$)
      $Base \leftarrow Base \cup (bstring, \text{fitness}(bstring))$
   /* select high quality subset and do filtering */
   $Sbase \leftarrow$ best_tuples($N_{sel}, Base$)
   for $a \in Sbase$ do
      $bb \leftarrow$ filter($a, partial\_sol$)
      $BB \leftarrow BB \cup bb$
   /* merge building blocks to partial solution */
   $partial\_sol \leftarrow$ merge_bb($BB$)
until complete($partial\_sol$) $\vee$ no_progress()
```

Figure 3: Pseudo-code of main loop of filtering algorithm

baseline population are selected. For each selected bit-string a filtering process is applied, which tries to locate a building block contained within this string that is responsible for the relative high fitness of this string. These building blocks are added to the set BB. At the end of each iteration a partial solution is created by combining all obtained building blocks. If two building blocks define opposite values for a bit, the value of the bit is taken from the first discovered building block. The main loop is terminated if the obtained partial solution is complete, i.e. specifies a value for each bit, or if no progress is achieved for more than 5 iterations.

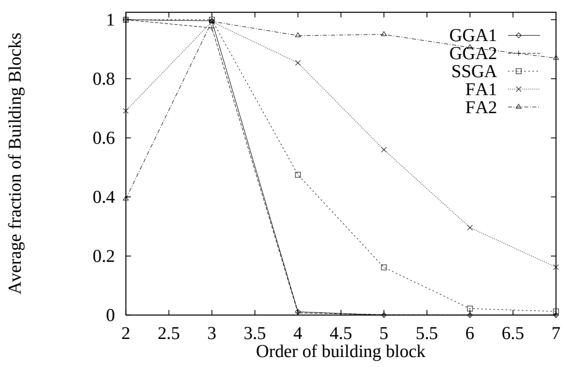
Based on the size of the baseline population N_{base} the expected maximal order of discovered building blocks can be estimated as $\lfloor {}^2\log(N_{base}) \rfloor$. If the filter process produces a large block, then this block mainly consists of noise, or it contains a large number of low order building blocks. Currently we use $\lfloor {}^2\log(N_{base}) \rfloor$ as an upper limit on the size of filtered block. Larger blocks are ignored. Note that this will deteriorate the performance of the method in case the order k of the building blocks is small.

As the solutions of many binary problems are assumed to be decomposable in a set of independent building blocks, the filtering algorithm is a valuable method. It is not necessary to know the linkage between bits in advance. As each sample represents 2^n schemata simultaneously we also have a kind of implicit parallelism. A further advantage of the method is that it yields the actual parts that compose a solution instead of just a complete solution. Such a decomposed solution allows for analysis, which helps in getting a better understanding of the specific problem at hand, and of the behavior of the filtering method on this problem. This is an important advantage over the genetic algorithm, where one usually only gets the (well performing) bit-string, without any knowledge about internal structure of the search-space, or an indication of the confidence one can have in this particular solution.

5. Experiments

A comparison is made between the following algorithms:

GGA1 a generational genetic algorithm with population size 1000, $P_{cross} = 0.7$, $P_{mut} = 1/l$, and tournament selection with tournament size 2,



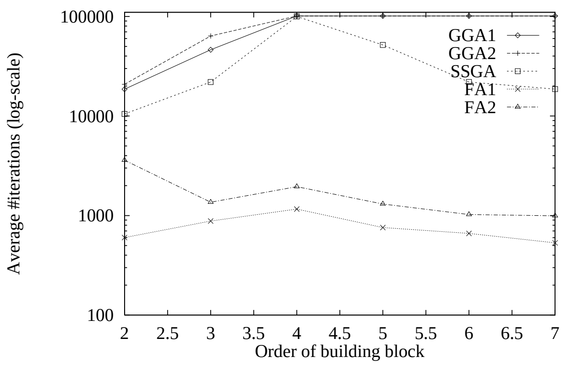


Figure 4: Average fraction of all building blocks found (left) and average number of function evaluations (right) as a function of the order k of the building block ($l \approx 30$).

GGA2 same as GGA1 except for 2-point crossover being used,

SSGA a steady-state genetic algorithm with population size 1000, $P_{cross} = 1.0$, $P_{mut} = 1/l$, uniform selection, and worst fitness deletion [13],

messy GA the messy genetic algorithm [8],

FA1 the destructive building block filtering method with $N_{base} = 200$, and $N_{sel} = 10$, and

FA2 same as FA1 except for having an upper bound of $\lfloor {}^2\log(Nbase) \rfloor$ on the number of bits that can be discovered simultaneously.

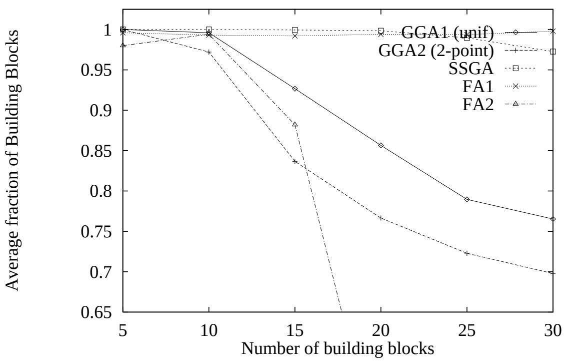
The settings of the GGA1 and GGA2 are the standard ones. SSGA is shown to perform well on a set of numerical optimization problems [13]. For the messy GA we only make a comparison to results from recent literature [8].

During all experiments the building block problems are based on the deceptive trap function. As we are interested in solving problems without any knowledge of the linkage between bits, it seems appropriate to assume a worst-case scenario: loosely coupled building block problems. All the results are averaged over 100 independent runs. The GA's are terminated when the optimum is found, the fitness variance over the population has decreased to zero, or the maximal number of function evaluations is reached.

The first set of experiments investigates scaling of different methods with respect to k. All the problem instances require a bit-string of approximately 30 bits. The exact sizes for m, k, and l are:

k	2	3	4	5	6	7
m	15	10	8	6	5	4
l	30	30	32	30	30	28

Figure 4 shows the average fraction of building blocks in the best solution detected (left) and the average number of function evaluations until termination (right) as a function of the order k of the building blocks. The FA methods outperform the GA's for all problem instances having building blocks of order $k \geq 3$. Amongst the GA's the SSGA method seems to perform best. It finds the optimum more often than the GGA's. An additional advantage of the SSGA is that it is able to terminate if the optimum is not found, which limits the amount of computation (see Figure 4, right). The value $k = 3$ seems to mark a region where the GA's start to get in trouble. A second set of



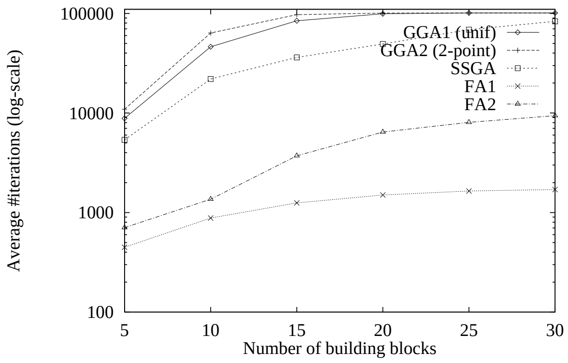


Figure 5: Average fraction of all building blocks found (left) and average number of function evaluations (right) as a function of the number of building blocks m ($k = 3$).

Problem		fast mGA	GGA1	GGA2	SSGA	FA1	FA2
P.90.3	P_{succ}	—	0.0	0.0	0.51	0.94	0.0
	Av. #iter	256,500	100,000	100,000	83,290	1,703	9,398
P.100.5	P_{succ}	—	—	—	0.0	0.06	0.76
	Av. #iter	1,000,500	—	—	30,870	1,825	4,902
P.150.5	P_{succ}	—	—	—	—	0.03	0.63
	Av. #iter	425,000	—	—	—	2,845	11,284
L.30.3	P_{succ}	—	0.84	0.86	0.99	0.01	0.71
	Av. #iter	>120,000	67,270	63,550	24,316	687	2,342

Table 1: Comparison between methods

experiments is performed during which k is set to a fixed value 3, while m takes the successive values 5, 10, 15, 20, 25, and 30. Figure 5 shows the results. The SSGA and FA1 method show comparable performance. The FA2 methods breaks down, as too many order-3 building blocks will be discovered simultaneously, resulting in a violation of the length constraint.

When looking at the graphs to the right in Figures 4 and 5 we see that both FA methods use only a moderate number of function evaluations. During all experiment the FA method uses approximately 10 times less function evaluations than the GA's.

Making a comparison to fast messy GA's is more difficult as our only source of information [8] just contains the outcomes of a single run. Our results are again averaged over 100 independent runs. The results are shown in table 1. Problem P.90.3 consists of a 90 bit problem containing 30 deceptive trap functions of order 3, P.100.5 consist of 20 building blocks of order 5, and P.150.5 contains 30 building blocks. The problem L.30.3 contains 10 building blocks of order 3, with linear scaling of the importance of building blocks. The fitness contribution of building block $\alpha \in [1, 10]$ is multiplied by 10α. Table 1 shows that in all cases where the GA's fail, at least one of the FA methods performs well. Making a comparison to the fast messy GA is more difficult, as we do not have information regarding the probability of convergence of this algorithm. But in all cases the fast messy GA uses at least 10 times more function evaluations than the FA method.

When comparing FA1 and FA2 we see that FA2 performs best on all instances having building blocks of order larger than 3. This result is to be expected as the only difference between these two methods is the additional constraint on the order of the obtained block of bits in FA2. As building blocks of low order are easy to find, the bit-strings selected from the baseline population will contain many building blocks having a combined length that violates this additional constraint. FA2 performs well in all those cases where the GA seemed to fail during our experiments.

It is not known yet how the filtering methods will behave on more complex problems containing overlapping building blocks or having building blocks which are not completely independent. On such problems the simple merging rule we used in this paper might be far from optimal. But we are convinced that discovering linkage stays important and therefore that the filtering method is usefull. Powerful solvers can be obtained by combining genetic algorithms with the filtering algorithm. For example, we can use the filtering algorithm as a pre-processing stage to identify the linkage between bits. Based on such linkage-information a specialized set of crossover masks can be constructed, or the genetic algorithm can be used to find the best combination of the actual building blocks discovered by the filtering algorithm. Another approach would be to incorporate the filtering in the GA. This approach is taken in GEMGA, where the a weight is computed for each bit of a chromosome. These weights are used during recombination operations to determine which sets of bits should be determined by the same parent [9].

6. Conclusions

Genetic algorithms were developed to be general problem solvers for arbitrary bit-coded problems based on the evolution principle. Most practical applications incorporate problem-specific knowledge in order to get a competitive algorithm. This deviates from the original idea of the GA as a general problem solver. We propose to incorporate general knowledge instead. One way to do so is to restrict the class of problems, and use the additional knowledge to enhance the genetic algorithm.

In this paper we have restricted ourselves to the class of building block problems. We have defined a filtering algorithm to locate building blocks without making assumptions about the linkage between bits. The results look promising. In this paper we have suggested several ways to combine the filtering algorithm with genetic algorithm in order to construct a fast hybrid genetic algorithm that requires less strong assumptions about the linkage of bits and the defining length of building blocks.

Remarks: Measuring the fitness contribution of individual bits and usage of this information has been developed independently by Hilol Kargupta [9]. Furthermore I would like to thank Joost N. Kok for his useful comments.

References

1. H. Asoh and H. Mühlenbein. On the mean convergence time of evolutionary algorithms without selection and mutation. In *Parallel Problem Solving from Nature III*, pages 88–97, 1994.
2. L.J. Eshelman, R.A. Caruana, and J.D. Schaffer. Biases in the crossover landscape. In *Third International Conference on Genetic Algorithms*, pages 10–19, 1989.
3. D.E. Goldberg. Genetic algorithms and walsh functions: Part II, deception and its analysis. *Complex Systems*, 3:153–171, 1989.
4. D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.
5. D.E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using the fast messy genetic algorithms. In *Fifth International Conference on Genetic Algorithms*, pages 56–64, 1993.
6. D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.
7. J.H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* The university of Michigan Press/Longman Canada, 1975.

8. H. Kargupta. SEARCH, polynomial complexity, and the fast messy genetic algorithm. Technical Report IlliGAL-95008, University of Illinois, October 1995.

9. H. Kargupta. Search, evolution, and the gene expression messy genetic algorithm. Technical Report 96-60, Los Alamos National Laboratory, February 1996.

10. N.J. Radcliffe and P.D. Surry. Fundamental limitations on search algorithms: Evolutionary computing in perspective. In J. van Leeuwen, editor, *Computer Science Today — Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 275–291. Springer–Verlag, 1995.

11. D. Thierens. *Analysis and Design of Genetic Algorithms*. Doctoral dissertation, University of Leuven, Belgium, 1995.

12. D. Thierens and D.E. Goldberg. Mixing in genetic algorithms. In S. Forrest, editor, *Fifth International Conference on Genetic Algorithms*, pages 38–45. Morgan Kaufmann, 1993.

13. C.H.M. van Kemenade, J.N. Kok, and A.E. Eiben. Controlling the convergence of genetic algorithms by tuning the disruptiveness of recombination operators. In *Second IEEE conference on Evolutionary Computation*, pages 345–351. IEEE Service Center, 1995.

14. D.H. Wolpert and W.G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, The Santa Fe Institute, July 1995.