Building block filtering and mixing

C.H.M. van Kemenade

Software Engineering (SEN)

SEN-R9837 December 31, 1998

Report SEN-R9837
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Building Block Filtering and Mixing

Cees H.M. van Kemenade
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
A three-stage evolutionary method, the BBF-GA is introduced. BBF-GA is an acronym for building block filtering genetic algorithm. During the first stage, an ensemble of fast evolutionary algorithms is used to explore the search space. The best individual found by each of these evolutionary algorithms is propagated to the next phase. During the second stage, building block filtering is used to extract the essential parts of each of these local optimal strings, and masks these essential parts. During the third stage, a single evolutionary algorithm is used to find the global optimum by recombining the masked strings. For this purpose we use a special recombination operator that exploits the information stored in the masks. Given an appropriate basis, such that partial solutions can be discovered and evaluated in parallel and be combined afterwards, a recombination-based evolutionary algorithm can be very efficient. Therefore, learning of the structure of problem-spaces is important to make a more efficient recombination possible. The BBF-GA is a first step along this line for binary search spaces and problems that adhere to the building block hypothesis.

1991 Mathematics Subject Classification: 68T05, 68T20
1991 Computing Reviews Classification System: G.1.6, I.2.8
Keywords and Phrases: genetic algorithms, building block filtering, problem solving
Note: Work carried out under theme SEN4 "Evolutionary Computation". Part of this report was published in the proceedings of the IEEE World Congress on Computational Intelligence/IEEE Conference on Evolutionary Computation.

Problems involving high-order building blocks with unknown linkage are difficult to solve. Neither n-point crossover nor uniform crossover can mix high-order building blocks efficiently. Linkage learning methods might help in generating more efficient recombination operators. Even when having an efficient crossover, it might still be difficult to strike the balance between exploration and exploitation. In this report the bbf-GA is developed. This is a hybrid GA that handles building blocks effectively and efficiently. A three-stage approach is used. During the first stage a large number of rapidly converging GA's is used to explore the search-space. During the second stage the best individual of each GA is filtered to locate the (potential) building blocks present in this individual. The third stage consists of a GA that exploits these masked individuals by mixing them to obtain the global optimal solution. The bbf-GA performs well on a set of test-problems, and is able to locate and mix more building blocks than the competitor GA's.

Section 1 presents the basic outline of the bbf-GA. In section 2 the building block filtering method is described. This algorithm takes a bit-string as its input, and produces a corresponding mask that marks the most important parts of the bit-string. The produced pairs of bit-strings and masks can be processed by means of the masked crossover introduced in section 3. In section 4 the bbf-GA is introduced. The test-problems are given in section 5 followed by the experimental results in section 6. Some enhancements for practical applications are suggested in section 7, and conclusions are given in section 8.

1. Outline of bbf-GA

The GA is often assumed to be an efficient method for solving building block oriented problems, because it is able to find building blocks independently of each other, and mix these building blocks afterwards. GA's might sometimes fail to locate optimal solutions. A possible reason for this is that it sometimes is difficult to find an appropriate balance between exploration and exploitation.

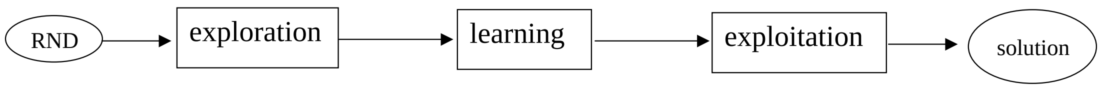


Figure 1: Basic scheme of the algorithm

A GA with emphasis on exploration (searching the complete search-space) is likely to be slow, a GA with emphasis on exploitation (preservation and duplication of the currently best individuals) is likely to converge too rapidly to suboptimal solutions.

The hybrid bbf-GA separates the exploration and exploitation task, and handles building blocks effectively and efficiently. In a traditional GA there is no explicit separation between exploration and exploitation. Both processes are done simultaneously. During the first few generations emphasis is on exploration, and during the final steps the GA is likely to focus on exploitation. There are no GA parameters to balance these two processes explicitly. Most GA parameters influence this balance indirectly, but it is difficult to predict how the balance will turn out. Consequently a lot of hand-tuning of parameters might be required for each new problem. In the bbf-GA the separation between exploration and exploitation is made more explicit. A basic outline of the method we are aiming at is given in Figure 1. Here one differentiates between three subsequent phases:

exploration: find a set of individuals that each contain a few building blocks,

learning: locate the most important bits (and hope that these bits will cover a part/the core of the building block), and

exploitation: mix the individuals (treating the masked parts as a single piece).

The hybrid bbf-GA follows this schematic approach. Exploration and exploitation are performed by means of GA's. Learning is done by a linkage learning algorithm which is called building block filtering.

2. Filtering of building blocks

Standard crossover operators are not always efficient. Problem-specific crossover operators or problem-specific choices of the operators can make the building block processing more efficient. Another approach lies in the usage of an evolutionary algorithm that really is able to learn something about the problem it solves, and that uses this information to steer the recombination process. In this report an evolutionary system that aims at this goal is investigated. To learn the linkage of loci, bit-strings are analysed by measuring the changes in fitness when changing each of the bits of the string, one at a time. The first GA that uses this approach is the GEMGA [Kar96b, Kar96a, BKW98]. Simultaneously, but independently, the building block filtering [vK96] was developed. Both methods estimate marginal fitness contributions of the separate bits. The GEMGA basically uses this information to guide the evolution process by computing a global decomposition of the search space (during the so-called preRecombinationExpression phase). The building block filtering method uses the information to make a decomposition of a bit-string in order to extract the loci that belong to the same building block and their optimal values. So, the building block filtering method aims at extracting local information. In this report a hybrid GA, the bbf-GA, that incorporates the building block filtering method is introduced.

If it is known which bits within a given individual belong to the same building block, then one can process the corresponding building block easily. If this type of information is not available, then it would be interesting to be able to extract such information. For this purpose the building block filtering method was developed. Given a specific individual this filtering method locates a (small) set of bits that have a relatively large influence on the fitness of the individual. Next, one assumes that this set of bits corresponds to an optimal building block, or at least that a schema corresponding to these bits is a schema that almost covers a single building block. These bits are

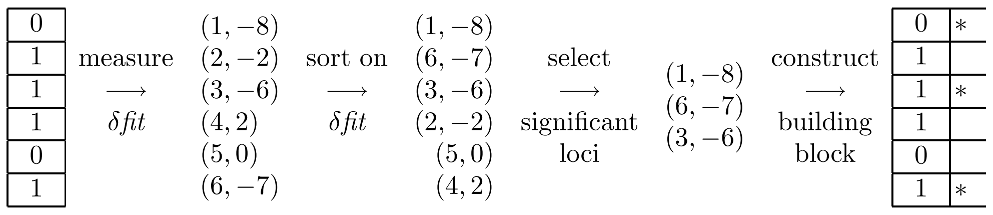


Figure 2: Example of one filtering step

marked and processed as a single unit when applying crossover. In this way the bias introduced by the choice of representation can be reduced. The defining length and the order of the schema that represents the optimal building block is not that important anymore, because the schema is processed in one piece. A detailed description of such a crossover operator is given in section 3.

The building block filtering method is applied in Figure 2. On the left a single individual of length six is given. This individual contains the bit-string 011101. The method uses four steps. During the first step the marginal fitness contribution, *δfit*, of each of the bits is measured, resulting in a set of tuples. A single tuple contains the index of a bit and the *δfit* observed when changing this bit. Next, these tuples are ordered on *δfit*. In the third step a truncation rule is used to select a subset of tuples, and in the fourth step a masked individual is created. The bit-string of this masked individual is exactly the same as the bit-string of the original individual, but a mask is added that indicates the most significant bits.

In the first step the change of fitness, *δfit*, of each of the loci is measured. The measurement for a single locus is performed by changing the value to its complement. So a 1-bit is changed to a 0-bit and a 0-bit to a 1-bit. The *δfit* is the change in fitness due to changing the value of the locus. It is used as a measure for the marginal fitness contribution of the corresponding locus with the context of the complete bit-string. In the example in Figure 2 a bit-string of length 6 is used. Let us assume that the main fitness contribution within this string is coming from a building block containing loci 1, 3 and 6, resulting in a fitness increase of $+7$ when the schema 0#1##1 is present. The *δfit* of each locus is measured by flipping its value, and observing the change in fitness. A set of tuples of type (position, *δfit*) is created. Flipping the bit in loci 1, 3 or 6 breaks schema 0#1##1. As a result the positive fitness contribution $+7$ gets lost. In our example the *δfit*-values of loci 1, 3, and 6 are respectively -8, -6, and -7.

The pseudo-code of this filtering step uses the following tuple:

$$locus \quad = [index, dFit];$$

Here, the *index* denote the position of the locus, and *dFit* denotes a change of fitness. Now, the filtering step is given by the following pseudo-code:

```
FilteringStep(x)
  loci : list of locus;
  loci = {};
  ## Step 1: compute the marginal fitness of all loci
  for $j = 1$ to $l$ do
      $x' = x$;
      $x'_j = 1 - x'_j$;
      $\delta fit$ = Fitness($x'$) - Fitness($x$);
      add $[j, \delta fit]$ to list loci;
  od;
  ## Step 2: sort the list of loci on increasing $\delta fit$
  $sort_{dFit}$(loci);
  ## Step 3: select the most important loci
```

```
    loci = Truncate(loci);
    ## Step 4: construct the masked individual
    ConstructMask(x, loci);
end
```

Here x is the bit-string that has to be filtered, $\{\}$ denotes an empty list, *Fitness*(x) computes the fitness of individual x, and the *add* operation adds a tuple to a list.

During the third step one has to truncate the ordered sequence of tuples in order to select a set of the most influential loci. The simplest approach is to mask a fixed number of tuples. A more complex approach would be to try to estimate the optimal number of bits to select. We estimate this truncation bound by taking a random test-individual, transferring the bit-values of the loci from the filtered individual to the test-individual one by one in the order given by the filtering step, and tracking the change of the fitness of the test-individual. At the moment the building block is transferred completely, a significant increase of the fitness of the test-individual is to be expected. So, in case of the example shown in Figure 2 the bit-values of loci 1, 6, 3, 2, 5, and 4 are transferred in sequence. The bit whose transfer increases the fitness of the test-individual most, and simultaneously results in a new fitness that is larger than the initial fitness of the test-individual, is used as an estimate of the truncation position. Assume that the bit at ranked position r results in the largest increase in fitness. Now, if the filtered individual and the test-individual have exactly the same bit-values from ranked position $r+1$ to $r+s$, then the truncation point is chosen in the middle of this range at position $r+\frac{s}{2}$. An upper bound on r is assumed as one expects to find building blocks of limited size only. Masking half of a bit-string would not make much sense. Application of this procedure with a single test-individual gives only a rough estimate of the optimal truncation point. Therefore, this procedure is repeated a number of times using different random test-individuals; The median of all obtained truncation points is used to select the bits that are masked.

```
Truncate(x, loci)
    truncPoints = list of number;
    for $i = 1$ to numTest do
        fitBlock = list of locus;
        fitBlock = $\{\}$;
        ## generate a random test individual
        y = RandomIndiv();
        fit = Fitness(y);
        ## determine the number of loci to transfer
        for $r = 1$ to $l'$ do
            $k = loci_r.i$;
            if $(x_k \neq y_k)$ then
                $y_k = x_k$;
                dFit = Fitness(y) - fit;
                s = NumSimilarValues(x, y, loci, $r+1$);
                add $[r + \frac{s}{2}$, dFit$]$ to list fitBlock;
            fi;
        od;
        t = MaximalLocus(fitBlock);
        if $(t.dFit > 0)$ then
            add t.index to list truncPoints;
        fi;
    od;
    ## generate the truncated list of loci
    truncationPoint = Median(truncPoints);
    Truncate = TruncateList(loci, truncationPoints);
```

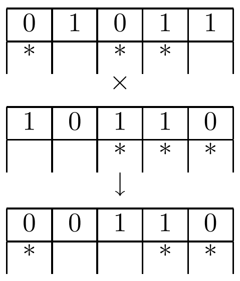


Figure 3: An example of the masked crossover

```
end
```

Here *RandomIndiv*() generates a random test-individual, *NumSimilarValues*(x, y, *loci*, r) determines the number of subsequent ranked loci in x and y that have the same value starting from locus r, *MaximalLocus*(*fitBlock*) extracts the locus of list *fitBlock* that has the highest value of *dFit*, *Median*(m) extracts the median from a list of numbers, and *TruncateList*(*list*, *pos*) truncates a list at location *pos*.

3. Mixing with masked uniform crossover

The filtering method described in section 2 produces a masked individual, where the masked bits are considered to be a potential building block. Next, different building blocks have to be combined in order to find a globally optimal solution to the problem. Masks do not have to represent exactly a building block. It is possible that a mask includes loci that do not belong to the same building block, or it might miss some of the loci that do belong to the building block. Therefore, a genetic algorithm is used with a special crossover operator to perform the mixing task. Next, the usage of the mask during the crossover is described, and it is shown how to transfer the information present in the mask to the offspring.

In the case that the parent individuals have disjunct masks, the crossover is relatively simple. The masked loci of the first parent are transferred to the offspring, next the masked loci of the second parent are added, and then uniform crossover is applied for the remaining loci. If there is an overlap between the masks of the two parents at a certain locus, then two cases have to be distinguished. If both parents have the same bit-value at this locus, then the masked parts are compatible and the same procedure can be applied. If both parents define different values for the locus, then a parent is chosen at random to provide the value for this locus.

The next step involves the creation of a mask for the offspring based on the two masks of the parents. A locus is masked in the offspring when it was masked by one of the parents, or when it was masked by both of the parents and the bit-values of the both parents at the corresponding locus are equal. If both parents masked a locus but define a different value, then the locus is not masked in the offspring. As a result a pruning of the masks can happen.

Figure 3 shows an example of the application of the masked crossover. The first locus and the last locus are masked by exactly one parent, so the offspring inherits the bit-value from the corresponding parents, and the corresponding mask-bits of the offspring are set for these loci. The third locus is masked by both parents, but the parents have non-matching bit-values, so an arbitrary parent is selected, and the mask-bit is not propagated. The fourth locus is also masked by both parents, but this time the bit-values match, so it does not matter from which parent the value of this locus is taken. In this case the corresponding mask-bit in the offspring is set.

4. The hybrid bbf-GA

The bbf-GA uses a three-stage approach, that follows the schematic overview given in Figure 1. During the first stage a large number of rapidly converging GA's is used to explore the search-space. The best individual of each GA is passed to the next stage. In the second stage each

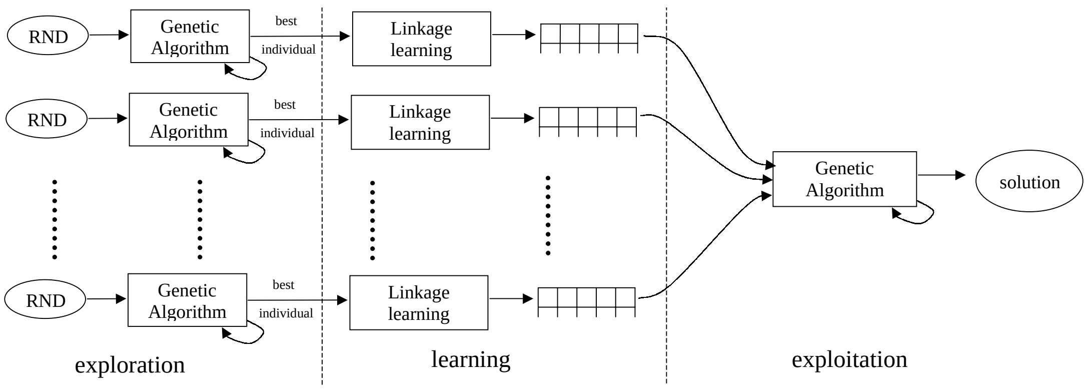


Figure 4: Overview of the algorithm

individual is filtered in order to mask the building blocks present in this individual. The third stage consists of a GA that that exploits these masked individuals by mixing them to obtain the global optimal solution. Figure 4 shows a schematic representation of this algorithm.

Exploration: we use a large set of small-population GA's running in independently to explore the search space. This way cross-competition between building blocks is prevented (as different building blocks can be discovered by different GA's). The initial populations of these GA's are chosen at random. It is important to use small-population GA's during this phase because:

1. GA's with small populations converge fast (i.e. reduction of the computational overhead),
2. GA's with small populations are not very reliable, so different GA's are likely to explore different parts of the search space. When using a reliable GA for exploration the most important building block is discovered repeatedly, and the GA is likely to be too slow.

Learning: we want, given an individual (suboptimal solution), to locate the most important part of it. In fact one tries to learn the linkage of bits within the context of this specific individual (bit-string)

Exploitation: search for the optimal solution by means of a single reliable GA. The initial population of this GA consists of all the masked individuals created during the previous phase. This GA should exploit all information of the different GA's used during the exploration phase and should make use of the linkage learned during the learning phase.

During the exploration phase a GA is needed that converges rapidly, and is likely to converge to different solutions, such that different parts of the search space are explored. Although this GA does not have to be reliable, it is important that the GA really does an exploration of part of the search space and therefore a GA that prevents too much duplication is preferred. The triple-competition selection meets these requirements. The triple-competition selection scheme [vK97b] is a steady-state selection scheme. Figure 5 shows how the next population P_{t+1} is produced from the current population P_t. On the left we see the current population P_t, where each box represents a single individual. The values in the boxes denote the fitness of the corresponding individuals. An intermediate population P_s is generated by doing a random shuffle on P_t. Population P_s is partitioned in a set of triples. Within each set of three individuals the two best performing individuals are allowed to create one offspring. This offspring replaces the third individual. This modified triple is propagated to the next generation. So two-third of the individuals are propagated unmodified to the next generation. The best two individuals are never lost when using this selection scheme. It has been observed that triple-competition selection tends to result in relatively fast convergence of the population.

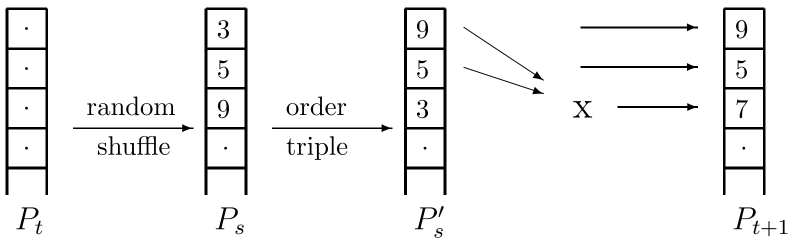


Figure 5: Schematic representation of triple-competition selection

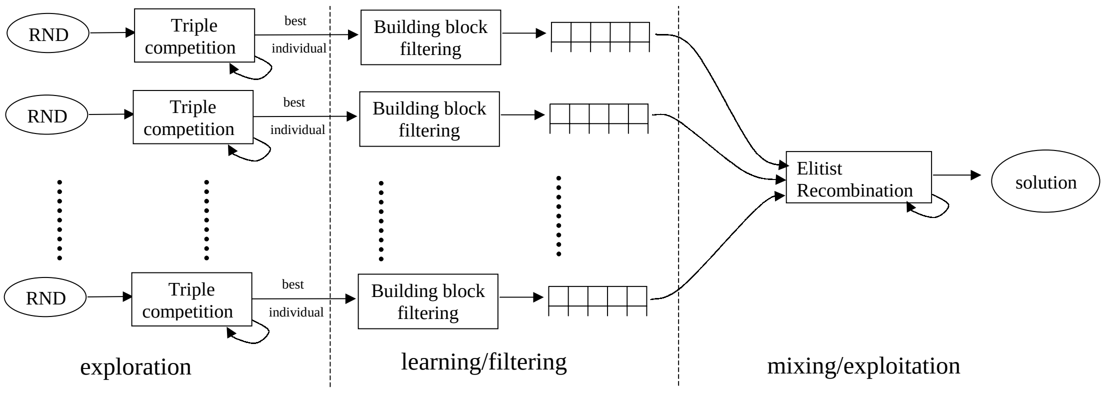


Figure 6: The bbf-GA

Nothing can be assumed about the linkage of bits during exploration; Therefore, the uniform crossover is used.

The linkage learning can be performed by means of the building block filtering described in section 2. The output of this second phase is a set of masked individuals.

During the exploration phase a reliable GA is required that performs well on a mixing task, even in the case that the population is quite small. Therefore elitist recombination is chosen. The Elitist recombination [TG94] uses a local competition between parents and offspring. It

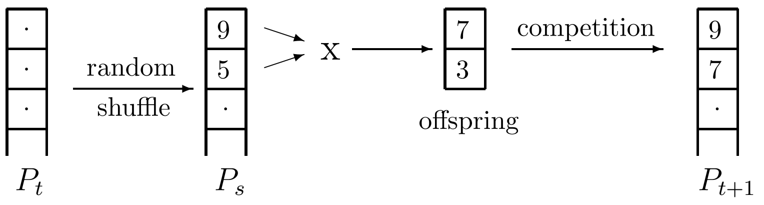


Figure 7: Schematic representation of Elitist recombination

selects parents by creating random pairs of individuals. Because all parents have exactly the same probability of being selected this corresponds to a uniform selection. The sampling error during this selection is reduced because each individual participates exactly once in a competition during a single generation. Figure 7 shows how the next population P_{t+1} is produced from the current population P_t. On the left we see the current population P_t, where each box represents a single individual. The values in the boxes denote the fitness of the corresponding individuals. An intermediate population P_s is generated by doing a random shuffle on P_t. Population P_s is partitioned in a set of adjacent pairs and for each pair the recombination operator is applied to obtain two offspring. Next, a competition is held between the two offspring and their two parents, and the two winners are transferred to the next population P_{t+1}. In the example one parent and one offspring are transferred to P_{t+1}. Elitism is used because parents can survive their own offspring. Elitist recombination has been shown to be more efficient than some other GA's on a problem involving a high-order building block [vK97a].

The information provided by the building block filtering is exploited by using the masked crossover operator. A detailed schematic representation of the bbf-GA is given in Figure 6

5. Test-problems

To study the effectiveness of the filtering and the mixing, two scalable test-problems are used. The number of building blocks m can be adjusted and the size of the optimal building blocks d is adjustable. For both test-problems the global optimum can be partitioned in a set of m order d building blocks.

5.1 *The fully deceptive problem*

The first test-problem is based on the fully deceptive trap-functions [DG93]. The following formula is used to compute the fitness contribution of a single part:

$$f_D(b) = \left\{ \begin{array}{cc} \alpha d & \text{if } u(b) = d \\ (d - u(b))/d & \text{otherwise} \end{array} \right. .$$

Here $\alpha > 1$. The global optimum of the deceptive part contains only 1-bits, which results in a fitness contribution of α.

Based on this building block a scalable test-problem can be constructed by concatenating m of these order d building blocks whose fitness contribution is determined by $f_D(b)$. The fitness of an complete individual is computed as the sum of the contributions of all its parts divided by $m\alpha$, so the fitness ranges from 0 to 1. A solution to this problem can be coded in a straightforward manner in a bit-string of length $l = m \times d$. The actual linkage of the bits belonging to the same part is assumed to be unknown, so we have a problem with loose linkage.

5.2 *The bb-NK problem*

Typical properties of the fully deceptive problem are that the first-order statistics for the fitness values of each locus are deceptive, and that the different parts can be optimized completely independent of each other. Therefore a second test-problem, the bb-NK problem, is introduced that does not have these properties, but does have building blocks. As a basis for the problem we use NK-landscapes. The NK-landscapes have been introduced by Kauffman [Kau93]. These are quasi-random landscapes that take a bit-string as input and compute the fitness value. The fitness of a string is given by the average fitness contribution over all loci in the bit-string. The fitness contribution of a locus i is determined by a random function $f_i(\cdot)$. This function uses a bit-string of length $(k+1)$ that is obtained by concatenating the value x_i of locus i and the bit-string N_i that contains the values of k other loci that are in a neighbourhood of locus i. Both the random function $f_i(\cdot)$ and the neighbourhood N_i are chosen independently for each of the loci. Two types of NK-landscape exist based on the way the neighbours are selected. The first type uses nearest-neighbour interaction: each locus is dependent on its k nearest other loci. Hence, the expected correlation between loci decreases with distance. The second type of NK-landscape uses a randomly selected neighbourhood. In the case the expected correlation between loci is independent of their relative position on the bit-string and loci that are far apart will usually be stronger correlated than in case of the nearest-neighbour interaction.

The fitness of a NK-landscape is given by the formula

$$f_{\mathrm{NK}}(\vec{x}) = \sum_{i=1}^{l} f_i(x_i \cdot N_i),$$

where f_i is a random function, x_i denotes the value of locus i of individual x, N_i is the bit-string containing the values of the loci that form the neighbourhood of bit i, $\cdot$ is the concatenation operator, and k is a parameter of the landscape that takes a value in the range 0 to $(l-1)$. The function $f_i : I \rightarrow [0,1]$, where I denotes the integer range from 0 to $(2^{k+1}-1)$; So, this is a random function that maps a $(k+1)$-bits integer to a real value between zero and one.

In order to build a test-problem with m building blocks of size d we take an NK-landscape length $l = m \times d$ with a random neighbourhood, and we select a random partition of the bit-strings in m parts. The fitness contribution of a bit is set to one if the part that bit belongs to is completely filled with 1-bits, otherwise the contribution of the bit is determined by the underlying NK-landscape. The fitness of a complete bit-string is computed as the average fitness-contribution of all the bits. The optimal solution is again a bit-string consisting of only 1-bits, each given a fitness-contribution of one. The bb-NK problem does have a set of independent building blocks of order d. The optimal building blocks are completely independent of each other, but if an optimal building block is not present within a certain part, then we have nonlinear interactions that cross the boundaries of the partitions, so this problem is not separable.

6. Experiments

For the exploratory phase of the bbf-GA a population size of 24 is used, which is evolved for 12 generations. Two variants of building block filtering have been investigated. The first variant uses a fixed bound for the number of bits that are going to be selected in the mask. The number of selected bits is set to eight. The second variant uses a flexible number of bits that is determined by the procedure given at the end of section 2. During the third stage elitist recombination with masked crossover is applied for 100 generations on a population of 100 masked individuals.

For the purpose of comparison we also conducted tests using the elitist recombination, the generational GA with tournament selection, and a steady-state GA. Elitist recombination using uniform crossover, is also applied directly to the problem with a population of 300 individuals that is allowed to converge for 100 generations. Furthermore a generational GA with tournament selection is applied with tournament-size 2, population size 300, 100 generations, and crossover is always applied.

For the deceptive problem $\alpha = 1.5$ is used, and for the bb-NK problem a random neighbourhood of size $k = 10$ is taken. All results shown are averaged over 30 independent runs. The bbf-GA with fixed bound for the number of bits to mask, uses 28,000 fitness evaluations per experiment, while the bbf-GA with flexible bounds uses approximately 10,000 additional evaluations to determine the bound during the filtering stage. The generational GA and elitist recombination use 30,000 fitness evaluations per experiment.

A set of 44 different experiments (experiment I) was conducted for varying orders of the building blocks d, and varying numbers of building blocks m. Each experiment was repeated 30 times. The upper graphs of Figures 8 and 9 show the average properties of the overall best individual. In case of the bbf-GA and elitist recombination this individual is present in the last population, but in case of the generational GA this individual might be lost, as the off-line performance is shown. For m the smallest value such that $l = m \times d \geq 60$ is taken, so for $d = 9$ one has $m = 7$ such that the size of an individual becomes 63 bits. The upper graph of Figure 8 shows the average fraction of building blocks present in the best solution for the deceptive problem. The lower graphs shows the off-line best fitness. The upper graph of Figure 9 shows the average fraction of building blocks present in the best solution for the bb-NK problem, and the lower graph shows the corresponding fitness. On both problems the bbf-GA using a flexible bound performs best followed by the bbf-GA with fixed bound, the elitist recombination, and the generational GA in sequence. The flexible bound selection for the bbf-GA performs well on the deceptive problem while on the bb-NK problem the results are only slightly better than for the bbf-GA with a fixed bound. The lower graph of Figure 9 shows the average fitness of the best solution as a function of the order of the optimal building block for the bb-NK problem. It is interesting to see that the generational GA outperforms elitist recombination for $d > 12$ even though it does not find any of the optimal building-blocks. Note that these graphs show the properties of the overall best individual, so it might be the case that the individual is not present in the final population of the generational GA and that the generational GA mainly does a random search.

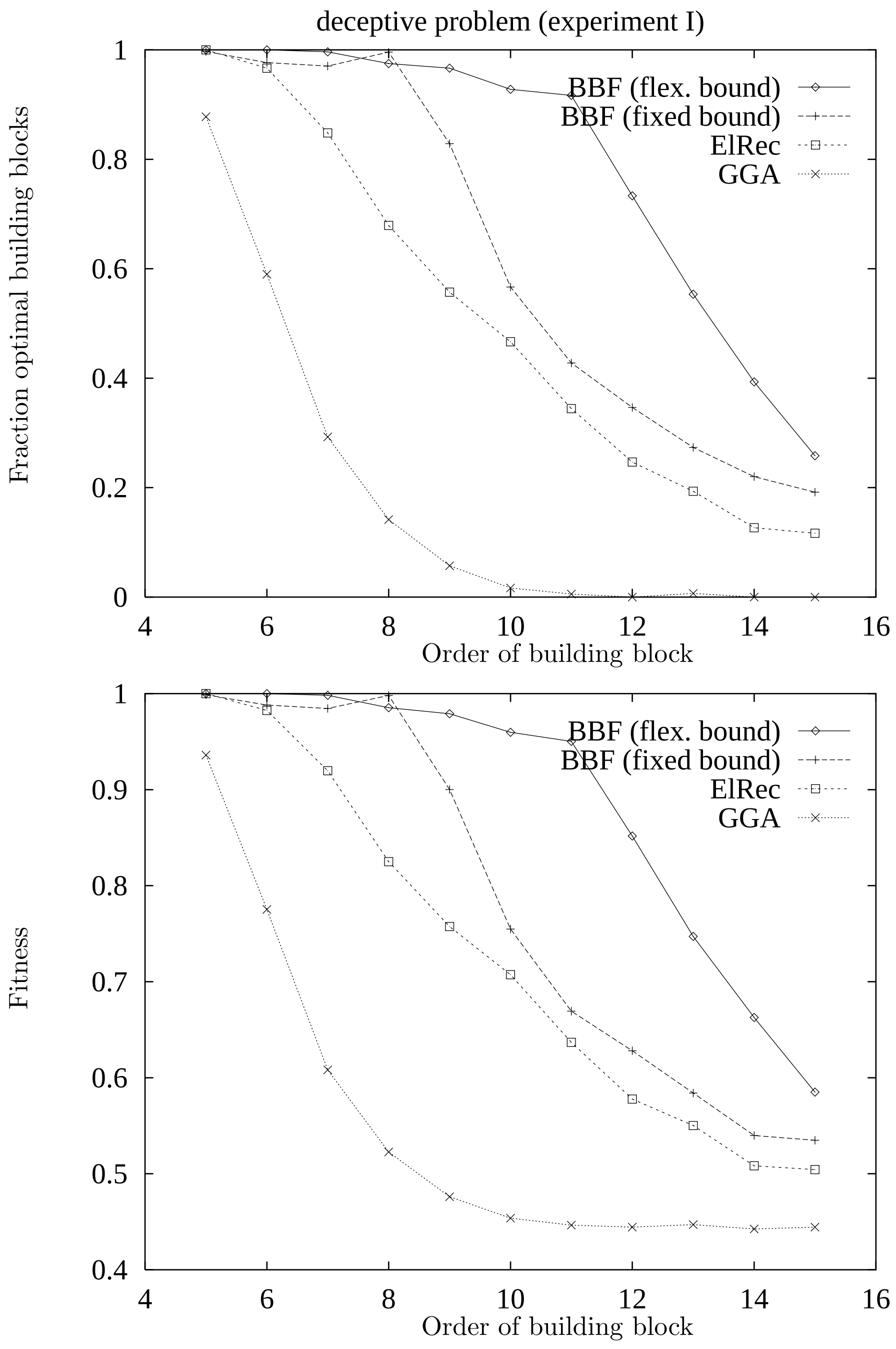
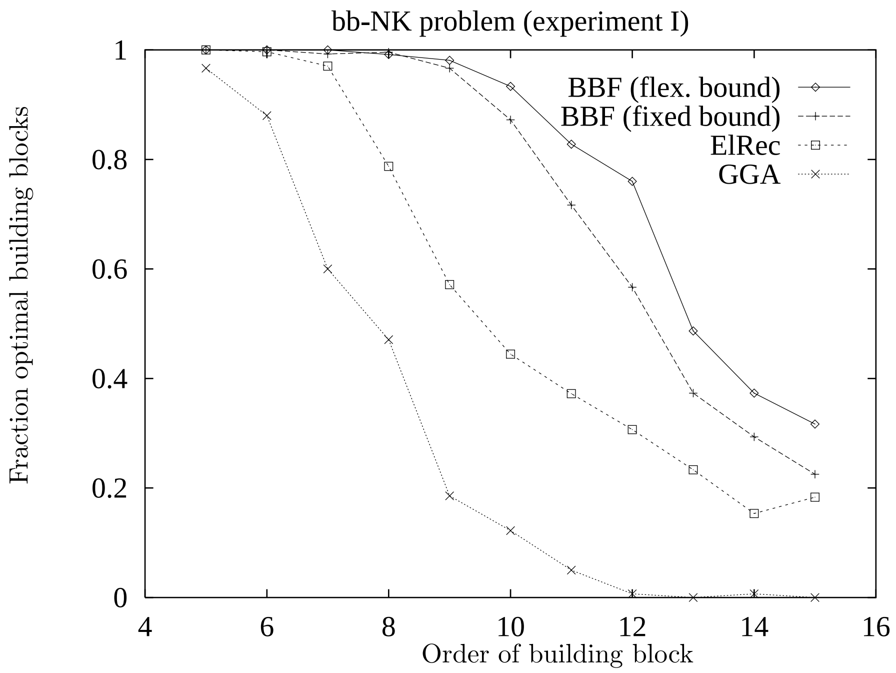


Figure 8: The average number of optimal building blocks in the best solution (top) and the fitness (bottom) when using different optimization methods for the deceptive problem.



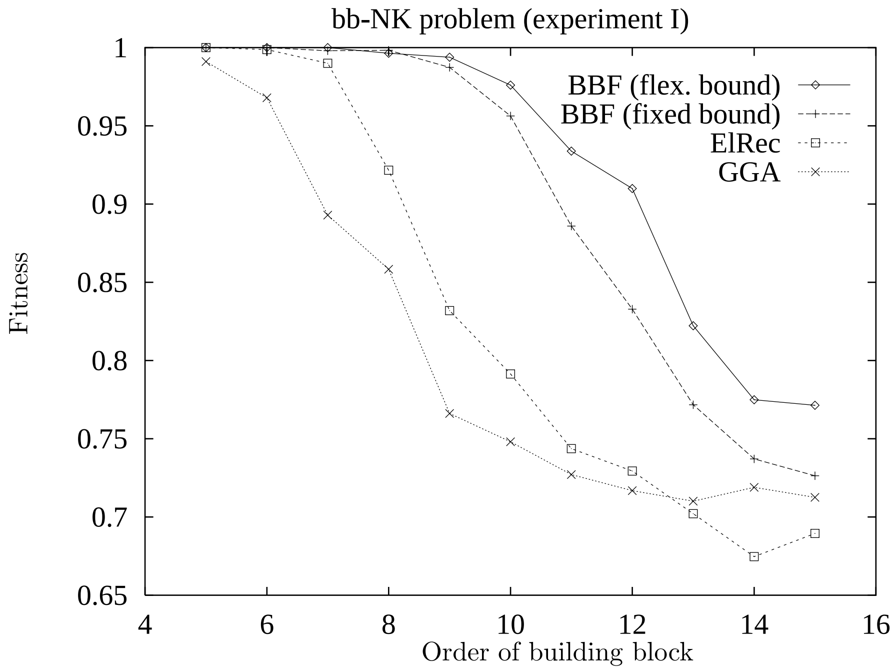


Figure 9: The average number of optimal building blocks in the best solution (top) and the fitness (bottom) when using different optimization methods for the bb-NK problem.

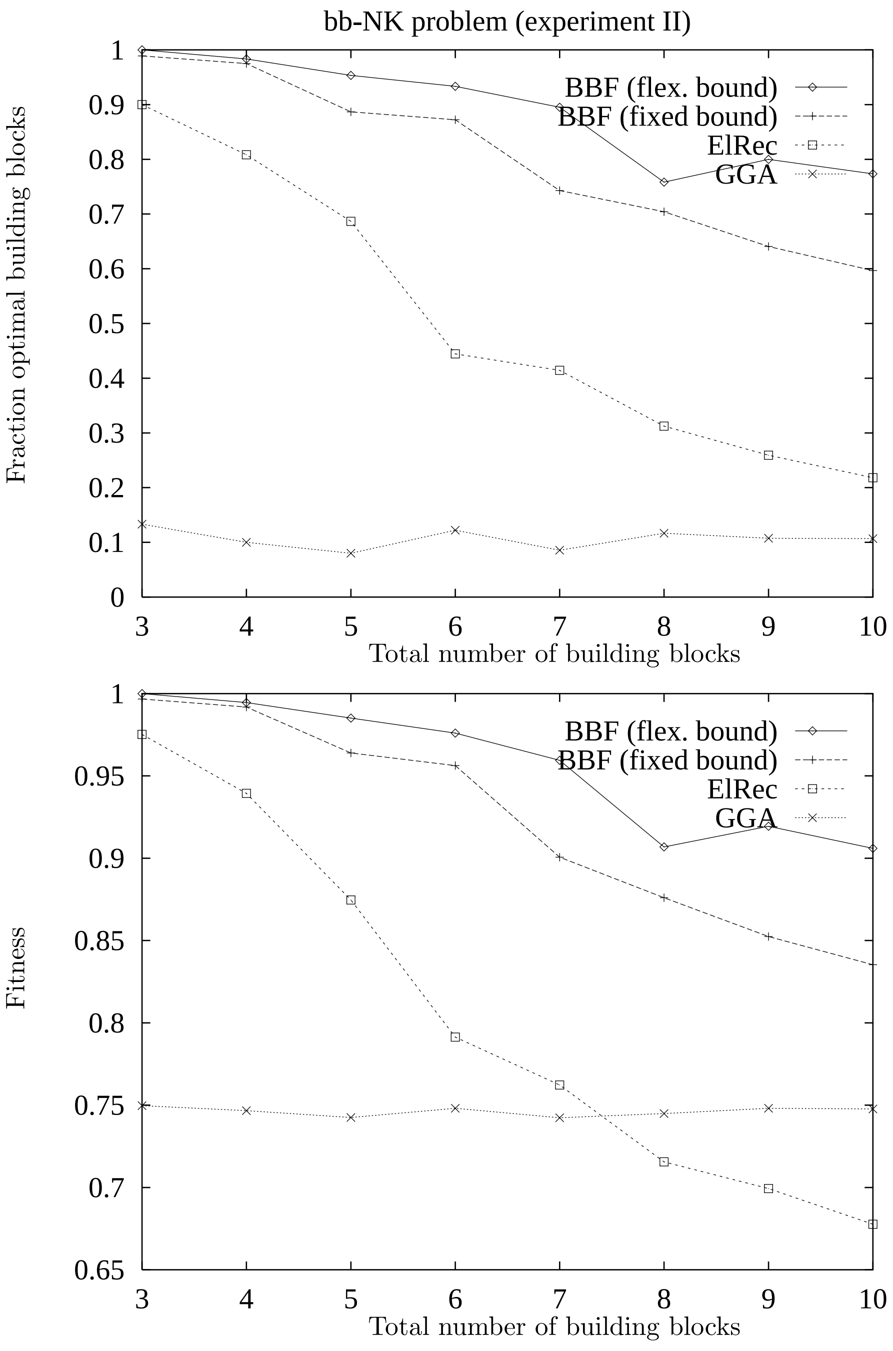


Figure 10: The average number of optimal building blocks in the best solution (top) and the fitness (bottom) when using different optimization methods for the bb-NK problem (Experiment II).

A second set of 32 experiments (experiment II) was conducted. During these experiments the size of the building blocks was fixed at $d = 10$ while the number of building blocks was varied. Again each experiment was repeated 30 times. Figure 10 shows the fraction of optimal building blocks (top) and the fitness (bottom) of the best solution for the bb-NK problem. Again the bbf-GA with flexible bound performs best while the generational GA performs worst. The performance of the BFF-GA's degrades less when increasing the number of building blocks than the elitist recombination. The fraction of building blocks in the best solution is low and nearly constant for the generational GA.

7. Enhancements to the bbf-GA

The test-problems are difficult and the bbf-GA gives good results. However, more extensive tests are required to properly assess the performance of the bbf-GA, and compare it to other algorithms. The test-problems shown in the current report are basically concatenations of functions of unitation. In the bb-NK problem interactions that cross the boundaries of building blocks are present, but these interactions are likely to be relatively small. A number of improvements have been added to the bbf-GA in order to prepare it for broader application. These improvements are discussed in the next subsections.

7.1 Filtering step

Two changes have been applied to the filtering step. The first change involves the correction for the average marginal fitness of a locus. This average marginal fitness is determined over the set of all individuals that are used during the learning phase. During the filtering step the marginal fitness of locus i in function FilterStep(x) is now computed by the formula

$$\delta fit_i' = \mathrm{Fitness}(x') - \mathrm{Fitness}(x) - \mathrm{Av}(\delta fit_i, x_i);$$

where x' and x only differ at locus i, δfit_i is given in the first definition of the filtering step, and $\mathrm{Av}_i(\delta fit_i, x_i)$ denotes the average change in fitness when changing the value of locus i from x_i to the complementary value. This average change in fitness is computed over the set of all individuals that are filtered. This adjustment helps in getting an appropriate ordering of the loci. The original rule performed well on functions of unitation, but did not always perform well in case that building blocks overlap.

The function Truncation has been reformulated based on the definition of a building block, where a building block is defined as a schema such that schema fitness of the complete schema is strictly larger than the sum of the schema fitnesses of an arbitrary partition. This reflects the fact that a building block has a larger fitness contribution than one would expect, when accumulating the fitnesses of its parts. Given this definition of a building block one can locate the truncation point by comparing the fitness after transferring the values of k loci, to the fitness after transferring $(k-1)$ loci plus the fitness of transferring only ranked locus k. If the transfer of k loci results in a larger fitness contribution, and results in a positive overall fitness contribution, then this locus is marked as a possible truncation point. Furthermore all subsequent loci where the two individuals match are marked as possible truncation points. After applying this procedure to all test-individuals, the locus with the maximal number of marks is selected. Furthermore a minimal value is imposed on the number of loci transferred.

7.2 Masked crossover

In the masked crossover operator defined in section 3 a single mask is generated for the offspring. When building blocks can overlap, it is less convenient to merge the masks of both parents in a single mask for the offspring. Therefore a representation is chosen where each individual carries a list of masks. During crossover the parents are allowed to take turns in propagating a masked set of bit-values to the offspring. The masked bits are transferred in a random order, and a set of masked bits is only transferred when these bits are compatible with the bits already present in the offspring, and the mask defines at least one bit that is not present in the offspring. After all masks of both parents have been processed, the remaining loci in the offspring are determined by a uniform crossover of both parents. No pruning of masks is performed, and the offspring contains a list of all masks that were transferred successfully from one of the parents to this offspring. Exact duplicates of masks are removed. A further advantage of this approach is that the individuals in the final population also contain information on the decomposition of these individuals by means of the mask-list these individuals carry with them.

7.3 Deterministic crowding

The elitist recombination has been replaced by deterministic crowding. Deterministic crowding is almost similar to elitist recombination. The only difference between elitist recombination and deterministic crowding is that in deterministic crowding each parent competes with only one offspring; The parents and offspring are paired such that the similarity between the parents and the offspring is maximized. Due to the masked crossover, the same parts of an individual are transferred time after time. This can easily lead to a rapid duplication of the best few masked individuals. Deterministic crowding performs better at preventing this type of duplication.

7.4 Population size during exploitation

In the explorative phase some loci might always converge to the same value. If that case, such a locus can never get the opposite value during the exploitation phase. To prevent this the population size during exploitation is doubled, and the second half of the population is filled with random individuals. These individuals provide the bit-values that might have been lost. A further advantage is that all masked individuals on average get one opportunity to duplicate their masked part before encountering another masked individual.

8. Conclusions

The proposed building block filtering method is able to discover parts of high-order building blocks even in quasi-random landscapes. Using this filtering method we create masked individuals which are processed with a special crossover operator. This crossover operator uses the mask to process a set of bits, assumed to be relatively important, in one piece while the non-masked bits are processed by a uniform crossover. The masked crossover also computes a new mask for the offspring based on the masks and the bit-values of both of the parents.

The bbf-GA is a hybrid three-stage GA. During the first stage an exploration of the search space is performed. The second stage locates a set of potential building blocks, and the third stage tries to exploit the the building blocks by mixing them in order to generate the optimal solution. The bbf-GA outperformed its competitors and scales better when increasing either the size, or the number of building blocks on our test-problems.

References

[BKW98] S. Bandyopadhyay, H. Kargupta, and G. Wang. Revisiting the GEMGA: Scalable evolutionary optimization through linkage learning. In *Proceedings of the IEEE World Congress on Computational Intelligence/fifth IEEE Conference on Evolutionary Computation (Vol. 1)*, pages 603–608. IEEE Press, 1998.

[DG93] K. Deb and D.E. Goldberg. Analyzing deception in trap functions. In G. Rawlins, editor, *Foundations of Genetic Algorithms-2*, pages 93–108. Morgan Kaufmann, 1993.

[Kar96a] H. Kargupta. Gene expression messy genetic algorithm. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pages 814–819. IEEE Press, 1996.

[Kar96b] H. Kargupta. Search, evolution, and the gene expression messy genetic algorithm. Technical Report 96-60, Los Alamos National Laboratory, February 1996.

[Kau93] S.A. Kauffman. *The origins of order.* Oxford University press, New York/Oxford, 1993.

[TG94] D. Thierens and D.E. Goldberg. Elitist recombination: an integrated selection recombination GA. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 508–512. IEEE Press, 1994.

[vK96] C.H.M. van Kemenade. Explicit filtering of building blocks for genetic algorithms. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 494–503. Springer, Berlin, 1996.

[vK97a] C.H.M. van Kemenade. Cross-competition between building blocks, propagating information to subsequent generations. In Th. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 1–8. Morgan Kaufmann, 1997.

[vK97b] C.H.M. van Kemenade. Modeling elitist genetic algorithms with a finite population. In *Proceedings of the Third Nordic Workshop on Genetic Algorithms*, pages 1–10, 1997.