Exploiting Modularity, Hierarchy, and Repetition in Variable-Length Problems

Edwin D. de Jong and Dirk Thierens

Decision Support Systems Group, Universiteit Utrecht, PO Box 80.089
3508 TB Utrecht, The Netherlands
{dejong,dirk.thierens}@cs.uu.nl

Abstract. Current methods for evolutionary computation can reliably address problems for which the dependencies between variables are limited to a small order k. Furthermore, several recent methods can address certain hierarchical problems which feature dependencies between all variables. In addition to modularity and hierarchy, a third problem feature that can be exploited when present is repetition. To enable the study of these problem features in isolation, two test problems for modularity and hierarchy detection by variable length problems are introduced. To explore how a variable length method can exploit these three problem features, a module formation algorithm is investigated. It is found that the algorithm identifies all three forms of problem structure to a substantial degree, leading to significant performance improvements for both the hierarchical and repetitive test problems. The experimental results indicate that the simultaneous exploitation of hierarchy and repetition will require both position-specific module testing and position-independent module use.

Modularity, hierarchy, repetition, SEQ problem, HSEQ problem

1 Introduction

Currently, evolutionary computation can reliably address problems for which the order of the dependencies between variables is limited to a small number k, where two variables are called *dependent* if the fitness contribution of one variable depends on the setting of the other variable and the order of the dependencies is the largest number of interdependent variables. Methods that can address this class of order-k limited problems have been called *competent GA's* [3] and include the fast messy GA [4], the extended compact GA [5], the Bayesian Optimization Algorithm (BOA) [11], LFDA [9], and EBNA [2].

Apart from order-k limited problems, there are certain specific problems with higher-order dependencies that can also be addressed. Specifically, problems with hierarchical structure can feature dependencies up to order $k = n$. These dependencies are limited to specific relations, and by virtue of this hierarchical problems can still be solvable in a scalable manner. Examples of hierarchical problems that have been described so far include H-IFF [15], H-TRAP [10], and

H-XOR [15]. Methods such as SEAM [16] and H-BOA [10] can solve difficult hierarchical problems by exploiting their hierarchical structure.

The class of hierarchical fixed-length problems is of interest because it is the most complex problem class, measured by the order of dependencies between variables, that may still be efficiently addressed by currently known evolutionary algorithms. The class of feasible problems may be further extended however if *variable-length* problems can be addressed.

One reason for employing *variable length* methods is that the length of optimal solutions in a problem may not be known in advance. Furthermore, variable length methods facilitate the use of *translocation* [7], i.e. applying optimized settings from one set of variables to other variables. Translocation exploits the problem feature of *repetition*, and can be used to address increasingly large problem spaces, performing directed exploration of very large search spaces without considering an exponentially increasing number of states; see also [6].

The above suggests that variable length methods exploiting modularity, hierarchy, and repetition would provide a valuable extension of the arsenal of methods currently available. A potential in this direction is demonstrated by the DevRep algorithm [1]; this method was reported to address a 1024-bit version of the HXOR problem. While HXOR [15] features modularity, hierarchy, and repetition, the presence of these multiple features leaves open the question of how these problem features can be exploited *in isolation*. This question will be explored here.

To enable the study of modularity, hierarchy and repetition in isolation, two test problems are introduced: the Sequence problem (SEQ), and the Hierarchical Sequence problem (HSEQ). SEQ features modularity, but no hierarchy or repetition. HSEQ features hierarchy and thereby modularity, but no repetition. To study repetition, we employ the OneMax problem [12]. We investigate how modularity, hierarchy, and repetition can be exploited, and develop a variable length algorithm for module formation. The operation of the algorithm on the SEQ, HSEQ, and OneMax problems is studied in experiments. Control experiments are performed to analyze the necessity of different features of the algorithm.

The structure of the article is as follows. First, a dependency-based classification of problems is provided, discussing modularity and hierarchy. Next, the SEQ and HSEQ problems are introduced. In section 3.1, measures for evaluating the detection of modularity, hierarchy, and repetition are discussed. The module formation that will be employed is described in section 4. Results are provided in section 5, followed by conclusions.

2 A Dependency-Based Classification of Problems

Discrete optimization problems can be classified based on the dependencies between the variables in the problem. Two variables are interdependent if and only if the fitness contribution of one variable depends on the setting of the other variable. Below, we discuss a classification of problems based on the dependencies they feature. The notions of modularity and hierarchy employed here are discussed, and correspond with the different problem classes.

The class of problems that is easiest to address is that for which no dependencies are present between variables. Problems in this class can be solved in linear time by optimizing each variable in turn. If dependencies up to a limited order k are present, a variety of modern genetic algorithms can be used to address the problem in a reliable way. This criterion is closely related to our notion of modularity. Our modularity concept is based on the criterion that the number of settings of a module can be reduced, called *decomposability* [14]. A subset $\mathcal{M}$ of the variables in a problem will be called a *module* if the number of settings of $\mathcal{M}$ that maximize fitness for at least one setting of the remaining variables is less than the number of possible settings for $\mathcal{M}$. There is a clear relation between modularity and order k schemata; in both cases, the dependency of the selected variables on the remaining variables is reduced.

Hierarchical problems may have dependencies up to order $k = n$ while still being solvable in an efficient manner. Thus, exploiting hierarchical structure can permit solving difficult problems that cannot be addressed otherwise.

We will now discuss the notion of hierarchy in more detail. Recall that a module is defined as a set of variables whose number of settings can be safely reduced because only some settings occur in optimal solutions. We view hierarchy as the recursive application of this same principle. Thus, a combination of two or more modules can be viewed as a single composite module if and only if this permits a further reduction of the number of possible settings of the variables involved. A clear demonstration of this principle is given by the Hierarchical IF-and-only-iF (H-IFF) function [15], which can be addressed efficiently by the fixed length methods of SEAM [16] and H-BOA [10]. Other examples of hierarchical problems that can be addressed efficiently include H-XOR [15] and H-TRAP [10].

The notions of modularity and hierarchy that have been described are characteristics of a *problem*. In the literature, the concepts of modularity and hierarchy are often used to describe the operation of a *method*. The problem-based notions of modularity and hierarchy provided here are intended to capture the modules that modular and hierarchical methods should ideally find. Thus, an algorithm is expected to have good performance if the modules identified by the method correspond to the modules of the problem.

The modularity and hierarchy concepts provide a strict criterion for determining which combinations of variables may be called modules; since any combination of variables or modules that is called a module must further reduce the number of possible variable settings that must be considered, the identification of modularity and hierarchy effectively reduces the size of the search space that must be visited, and therefore permits a performance gain.

Repetition is viewed as the presence of optimal substrings that occur multiple times within an individual. If a problem features repetition, translocation can in principle confer an advantage. Translocation is normally avoided in genetic algorithms, as the theoretical foundation aimed that explains the operation of the genetic algorithm requires the propagation of schemata at given locations. If the order of the variables on a genome is non-random however, then patterns of adjacent bits may carry information that can be usefully applied in other parts of the string by means of translocation. For example, if a bitstring encodes natural text using four bits for each letter, then the identification of a string in which certain letters are much more frequent than others may benefit from translocation.

3 The SEQ and HSEQ Problems

In this section, two new test problems for the study of modularity and hierarchy in variable length methods are introduced. Several test problems exist that permit testing whether a method not using translocation can identify modularity and hierarchy; examples include HIFF and HXOR [15]. Since these problems feature some degree of repetition however, translocation is expected to be beneficial in addressing these problems. While the combined exploitation of hierarchy and repetition may be very successful on such problems, as found e.g. in [1], our aim is to study how modularity and hierarchy may be identified in isolation by methods permitted to use translocation.

First, we consider the definition of a non-repetitive test problem featuring modularity. The requirement that the problem should not feature repetition can be guaranteed by ensuring that within across all optimal individuals, any combination of two consecutive values may occur in at most one position. This requirement can be only be satisfied for problems of non-trivial size by using an arity that is greater than two.

The SEQ problem or Sequence problem is defined as follows. For a string of length n, there are two global optima: the ascending string $A = 0, 1, \ldots, n-1$ and the descending string $D = n-1, n-2, \ldots, 0$. In this n-ary problem, the i^{th} variable provides a fitness contribution of 1 if it equals either A_i or D_i. In addition, any two consecutive variables $2j, 2j+1$ for $0 < j < \frac{n}{2}$ provide an extra fitness contribution of 2 if they equal $A_{2j}A_{2j+1}$ or $D_{2j}D_{2j+1}$. Thus, there are two levels that contribute fitness: the level of single variables and the level of consecutive pairs of variables.

The HSEQ problem or Hierarchical Sequence problem is defined by extending the SEQ problem to a higher number of levels; consecutive pairs of ascending modules form ascending modules at the next level, and likewise for the descending modules. By continuing this principle, the highest level features two modules that equal the global optima A and D. The HSEQ problem is analogous to HXOR and HIFF; all three problems combine pairs of consecutive modules into higher-level modules, thereby reducing the number of optimal settings for the constituent modules from four to two. The HSEQ problem is different however in that it employs an arity of n for a length-n string. As a result, it features a much larger search space; the size of the search space is n^n, and thus grows super-exponentially as a function of the string length n.

It is important to note that to address HSEQ problem effectively, it is necessary to maintain *multiple* settings for each combination of variables (two, in this case) until a global optimum is found; if subsets of the variables are allowed to independently converge to one setting (ascending or descending numbers), it becomes increasingly likely with increasing problem size that different subsets will converge to different choices, thereby ruling out the possibility of finding an optimum. Part of the structure in the problem can already be exploited by identifying pairs of variables, and reducing the number of possible settings from the initial n^2 to two; this would amount to the use of modularity. The resulting number of possibilities that must be maintained in this case is still exponential in the number of pairs of variables. By using hierarchy however, the problem can be addressed efficiently. The full potential for efficiency improvement in this hierarchical problem is exploited by *recursively* applying the principle of identifying the optimal settings for each module, thereby identifying correct settings for modules of exponentially increasing size.

3.1 Measuring Modularity, Hierarchy, and Repetition

In measuring the degree two which the three problem features investigated are identified, two criteria are of interest. First, the number of correct modules identified should be high; the more modules are identified, the higher the computational benefit that can be gained. Second, the number of modules formed that are not modules of the problem should be as low as possible since the construction of modules influences the *exploration distribution* [13], i.e. the distribution of individuals that will be visited.

To measure repetition, a slightly different approach is necessary; if a method uses translocation, then maintaining a single instance of a repetitive element is sufficient. Thus, the number of modules identified does not reflect the degree to which repetition is exploited. Therefore, we measure the extent to which the pattern of interest is repeated *within* the modules formed. Since the length of the repeated pattern equals one in the case of OneMax, this reduces to measuring the frequency of the most frequent bit (zero or one) within modules.

4 A Variable-Length Method Exploiting Modularity, Hierarchy, and Repetition

In the following, we develop a variable-length method designed to exploit modularity, hierarchy, and repetition. The algorithm maintains a population of individuals. Individuals are sequences of modules. Initially, the set of available modules contains the primitives of the problem; $\{0, 1\}$ for a binary problem, or $\{0, 1, \ldots n-1\}$ for the n-variable SEQ or HSEQ problems.

Periodically, a module formation step is performed. The basis for module formation is provided by the notion of hierarchical modularity that has been described. Thus, the aim is to identify combinations of variables for which the number of optimal settings is reduced compared to the number of possible settings. For reasons of computational efficiency, two main restrictions are placed on the modules that can be formed: modules can only consist of consecutive variables, and a module always contains precisely two elements. The main loop of the algorithm is as follows:

```
Module Formation Algorithm()
 1.  pop:=generate_random_individuals(pop_size)
 2.  for generation=1:no_generations{
 3.        if generation mod frequency == 0
 4.              module_formation(pop,front)
 5.        pop:=evolve(pop)
 6.        pop:=local_search(pop)
 7.        if average_length(pop) < size_factor * initial_length
 8.              update_lengths(pop)
 9.        front:=front∪non_dominated(pop)
10.  }
```

Fig. 1. Module Formation Algorithm

The operation of the algorithm will now be detailed. The module_formation procedure first ensures that for any composite modules C=AB present in the module set, occurrences of the modules combination AB in individuals are replaced by a reference to module C, thereby shortening the length of individuals by one element for each occurrence. Modules only replace occurrences of their elements however if the absolute position of the occurrence, measured in terms of the number of preceding primitives (typically bits, but integers here), is identical to the absolute position at which the module itself was located when it was formed. The relative position of a module in the sequence of modules that specifies an individual is simply its index in the sequence.

Next, the module formation set randomly selects two consecutive elements $[AB]$ from a randomly chosen individual that is part of the *front* set, and forms a candidate module $C = AB$. Thus, the pairs of modules selected as candidate modules reflect the distribution of module pairs in the population. If multiple objectives are used, the front accumulates the non-dominated individuals identified over time. In the current experiments, only fitness is used as an objective, and *front* thus accumulates individuals that have improved the maximum fitness achieved at some point.

If the candidate module C does not already exist as a module, it is tested as follows. For each individual x, the primitives represented by C temporarily replace the corresponding primitives of the individual and the individual is evaluated. Next, the same primitives are replaced by alternative settings for the same absolute positions. Specifically, the module C=AB is compared to all alternative modules combinations A* and *B, where the asterisk (*) is filled in by every existing module of the same size. If any of these alternative settings result in an increased evaluation for any objective, the candidate module will not be formed. If this test is passed for all individuals, the module C=AB is formed and added to the module set. Furthermore, the module replacement procedure is invoked for the newly formed module, i.e. occurrences of AB in individuals at the same absolute position are replaced by C. The use of a position-specific module-formation test ensures that a particular combination of values is tried out for the same set of variables in all compared individuals.

After module formation, the algorithm performs a generation of evolution, using Mahfoud's deterministic crowding algorithm [8]. The operators of variation are as follows. The crossover operator used respects the absolute position of the modules that make up the individuals. It does so by considering all module boundaries shared by the two parents (i.e. determining which relative positions correspond in terms of their absolute positions), and selecting a crossover point randomly from these options.

The first mutation operator randomly selects an element in an individual and replaces it with a randomly selected module of the same length. The second mutation operator randomly selects a module and, if possible, replaces the corresponding elements in the individual at the module's original position. Next, optionally, local search is applied. Local search takes each individual, and optimizes each of its elements in a random order. Optimization consists of considering all alternative modules of the same length, and randomly selecting one that achieves the maximum attainable fitness given the remaining variables.

Finally, a mechanism is used to enable the growth of individuals over time. Initially, individuals contain initial_length elements. When elements are replaced by modules, the effective length of the individual remains the same, but the actual length diminishes as a result of the more compact representation facilitated by the use of modules. To allow the effective length of individuals to increase gradually over time, the update_length procedure restores the length of individuals back to the original initial_length by adding random elements. This operation is performed when the average actual length of the population drops below a threshold.

The algorithm that has been described bears a close relation to existing algorithms designed to exploit hierarchy, such as SEAM and DevRep. A main difference with SEAM is the use of a variable length representation, while a main difference with DevRep is in the use of a position-specific module-formation test.

5 Results

In this section, we investigate the ability of the module formation algorithm to identify modularity, hierarchy, and repetition in test problems that feature these characteristics in isolation as far as possible. The experimental settings are as follows. For methods employing module formation, the initial_length parameter is set to 16. The population size is 100. Crossover is used with $P = 0.8$, and both mutation operators with $P = 0.1$. Module formation is performed every 25 generations. The parameter size_factor = 0.5.

5.1 Modularity

The maximum baseline performance for a module formation method that is not able to detect modularity is 50% of correct modules; if module formation randomly selects pairs of consecutive modules, this maximum performance is obtained if the population solely contains global optima.

The results are shown in figure 2. The graph shows that the number of correct modules is substantially higher than the number of incorrect modules, and thus well exceeds the baseline performance. Thus, according to the measurement criterion employed, the method is able to correctly identify modularity in the problem. Preliminary experiments suggest that the effect of identifying correct modules in SEQ on algorithm performance is limited however; the computational benefit of identifying interdependent sets of variables is expected to be most clear in deceptive problems.

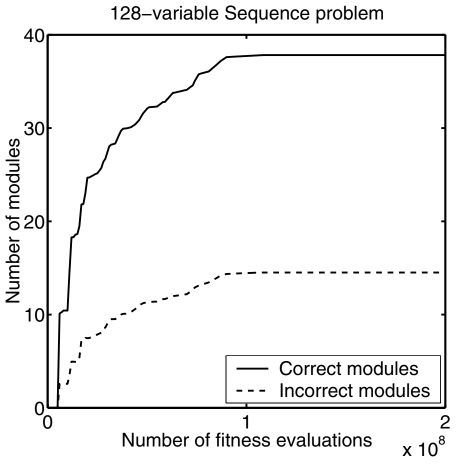
5.2 Hierarchy

To test the ability of the module formation algorithm to identify hierarchy, the method is applied to the HSEQ problem. Again, the evaluation criterion for module formation consists of counting the number of correct and incorrect modules formed. The results are given in Figure 2, and show that the number of correct modules formed grows at a steady pace and then quickly flattens. The number of incorrect modules formed remains low. While it might be expected that the correct modules in HSEQ are more difficult to detect, as the modules range from size 2 to modules representing complete individuals (size 128), the relation between correct and incorrect modules is in fact higher than for the SEQ problem. The total number of correct non-primitive modules for the 128-variable HSEQ problems equals 254.[1] The number of correct modules identified (228) is thus higher, both in absolute and relative terms, than for the SEQ problem; 90% of the modules present in the 128-variable HSEQ problem are identified.

The search space for 128-HSEQ is of size $128^{128} = 2^{896}$, and thus corresponds to a 896-bit problem. In addition to being large, there is very little gradient in this space; in a random string, only 1 in every 64 variables is expected to make a fitness contribution. Therefore, local search is used to speed up the search; this renders the problem more comparable to a 128-bit problem again while maintaining the property that no repetition is present in the problem.

Figure 3 show the performance of the module formation method on the 128-variable HSEQ problem. The method achieves near optimal performance on average. Inspection of the runs showed that 8 out of ten runs reached the global

[1] The 64 pairs for each direction (ascending and descending) form the leaves of a binary tree, which therefore has 63 internal nodes.



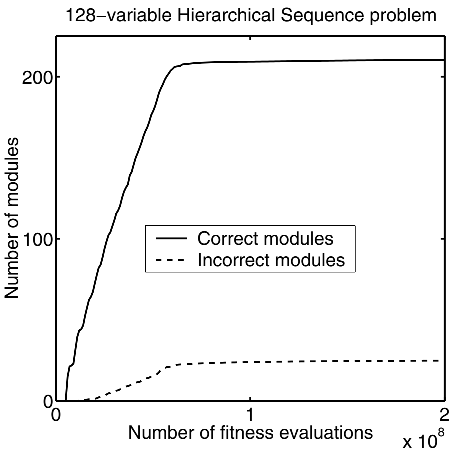


Fig. 2. Number of correct and incorrect modules formed on the SEQ (left) and HSEQ (right) problems.

optimum of 1024. Interestingly, the abrupt change in the module formation graph (Figure 2) corresponds precisely to the moment at which a global optimum is reached on average; this suggests that module formation ends when all correct modules present in the population have been formed.

As a comparison, we apply a genetic algorithm that does not perform module formation to the same task. To focus on the effect of module formation, the algorithm is identical to the module formation algorithm, except that no modules are formed after the module set has been initialized. As a result, the length of individuals cannot increase, and the initial length of individuals is therefore made equal to the length required for the problem, i.e. 128 in this case. Thus, the genetic algorithm receives some prior information about the problem, namely the required length for the problem, which the module formation algorithm does not receive. As figure 3 shows, the performance of the genetic method is significantly less however, and module formation can thus be concluded to have a positive effect on performance.

Two variants of the module formation algorithm are compared. The first variant uses position-independent operators by using variants of the crossover and mutation operators that do not respect the alignment of elements of individuals, and can thus translocation. In the second control experiment, the module formation does not test a candidate module at the position where it occurred, but at all positions within the individual.

Figure 3 shows the results of the control experiments. The use of position-independent operators does affect performance, but still permits the formation of useful modules; this algorithm is still very different from the genetic algorithm in figure 2, as individuals have an initial length of 16, and the substantial perfor-

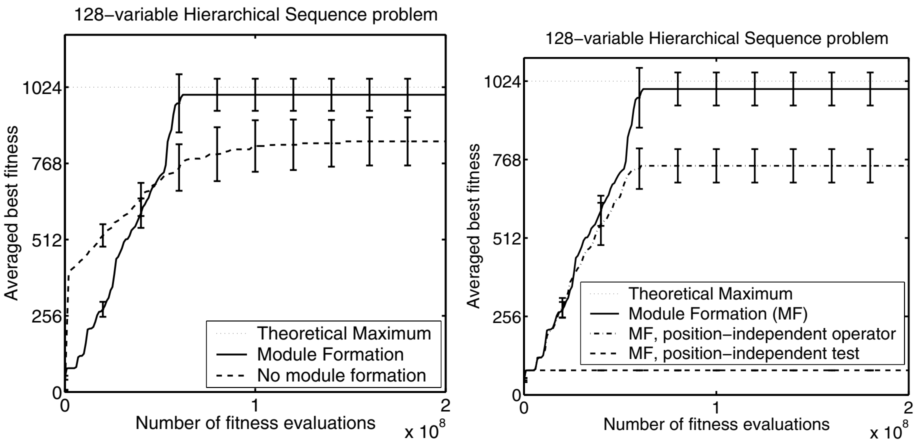


Fig. 3. Performance on the 128-variable HSEQ problem for various methods.

mance increase shows that many modules are still formed. This is different for the control experiment in which the module acceptance test does not respect the positions; for this method, no modules are formed, and performance is therefore limited to a maximum of 80, which is quickly attained but not exceeded.

5.3 Repetition

Finally, we study the ability of the module formation method to identify and exploit repetition. To exploit repetition, translocation is required. Thus, the position-independent operators used in the previous experiments are used again. Since the OneMax problem is binary, we use a larger, 1024-bit version, and local search is not required.

Figure 4 demonstrates the potential benefit of translocation. The standard module formation algorithm already improves substantially over the genetic algorithm not performing module formation. When translocation is employed, maximum performance is reached almost immediately. Inspection of the degree of repetition achieved, as measured by the average relative frequency of the most frequent primitive, showed that the maximum degree repetition, i.e. 1.0, was reached and maintained.

It appears that the recursive combination of strings of ones into larger and larger modules can produce correct solutions to one-max of sizes growing exponentially in the number of generations, but this hypothesis remains to be tested. Likewise, information about the scalability of the method on modular and hierarchical problems is of central interest in further evaluating the potential of the method that has been described.

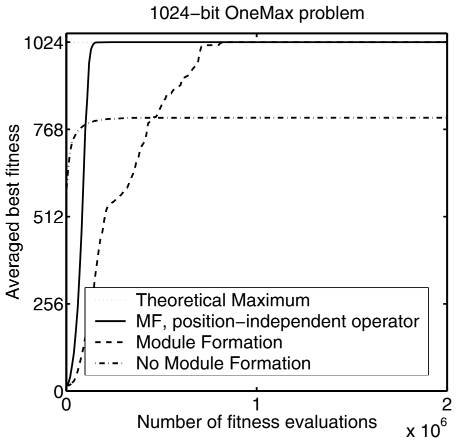


Fig. 4. Performance on the 1024-bit OneMax problem.

6 Conclusions

A wide range of problems can currently be reliably addressed by methods from evolutionary computation. Furthermore, particular subclasses of the remaining problems can also be addressed efficiently. One such subclass which has received recent attention is that of problems featuring hierarchy. Two other problem features that can improve the efficiency of a search method when exploited are modularity and repetition. We explore how these problem features may be detected and used to benefit by variable length algorithms.

To study the identification and exploitation of modularity, hierarchy, and repetition, two new test-problems have been introduced: the SEQ and HSEQ problems. Existing test problems contain a combination of these features. In contrast, the new test problems, and the existing OneMax problem, enabled the study of these problem features in isolation.

A variable length algorithm employing module formation has been described. In experiments, it was demonstrated that the modules formed by the method correspond to the modules present in the problems, and the method can thus be said to detect modularity, hierarchy, and repetition to a substantial degree. For the HSEQ and OneMax problems, a significant performance gain was achieved as a result of module formation. While translocation was seen to be useful in the presence of repetition and no insurmountable obstacle in the hierarchical HSEQ problem, a position-specific module-acceptance test was found crucial in the latter problem. These findings suggest that successful independent exploitation of hierarchy and repetition requires both position-specific module testing and position-independent module use.

References

1. Edwin D. De Jong. Representation development from Pareto-coevolution. In E. Cantú-Paz et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-03*, pages 262–273, Berlin, 2003. Springer.
2. R. Etxeberria and P. Larrañaga. Global optimization using bayesian networks. In A. Ochoa Rodriguez, M. Soto Ortiz, and R. Santana Hermida, editors, *Proceedings of the Second Symposium on Artificial Intelligence CIMAF*, 1999.
3. David E. Goldberg. *The design of innovation. Lessons from and for competent genetic algorithms.* Kluwer Academic Publishers, 2002.
4. David E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 56–64, 1993.
5. Georges Harik. Linkage learning via probabilistic modeling in the ECGA. Technical Report Illigal report no. 99010, University of Illinois at Urbana-Champain, Urbana, IL, 1999.
6. Inman Harvey. The SAGA cross: the mechanics of recombination for species with variable-length genotypes. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, PPSN-II*, volume 2, pages 269–278, Amsterdam, 1992. North-Holland.
7. John H. Holland. *Adaptation in Natural and Artifical Systems.* University of Michigan Press, Ann Arbor, MI, 1975.
8. Samir W. Mahfoud. *Niching Methods for Genetic Algorithms.* PhD thesis, University of Illinois at Urbana-Champaign, May 1995. IlliGAL Report 95001.
9. Heinz Mühlenbein and Thilo Mahnig. FDA - A scalable evolutionary algorithm for the optimization of additively decomposed functions. *Evolutionary Computation*, 7(4):353–376, 1999.
10. Martin Pelikan and David E. Goldberg. Escaping hierarchical traps with competent genetic algorithms. In L. Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-01*, pages 511–518. Morgan Kaufmann, 2001.
11. Martin Pelikan, David E. Goldberg, and Erick Cantu-Paz. BOA: The bayesian optimization algorithm. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 525–532, San Francisco, CA, 13-17 July 1999. Morgan Kaufmann.
12. J.D. Schaffer and L.J. Eshelman. On crossover as an evolutionary viable strategy. In R.K. Belew and L.B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68, 1991.
13. Marc Toussaint. The structure of evolutionary exploration: On crossover, buildings blocks, and Estimation-Of-Distribution algorithms. In *2003 Genetic and Evolutionary Computation Conference (GECCO 2003)*, Berlin, 2003. Springer.
14. Richard A. Watson. *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis.* PhD thesis, Brandeis University, 2002.
15. Richard A. Watson, Gregory S. Hornby, and Jordan B. Pollack. Modeling building-block interdependency. In A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature, PPSN-V.*, volume 1498 of *LNCS*, pages 97–106, Berlin, 1998. Springer.
16. Richard A. Watson and Jordan B. Pollack. A computational model of symbiotic composition in evolutionary transitions. *Biosystems*, 69(2-3):187–209, May 2003. Special Issue on Evolvability, ed. Nehaniv.