# A Generator for Hierarchical Problems

Edwin D. de Jong
Institute of Information and Computing Sciences
Utrecht University
PO Box 80.089
3508 TB Utrecht, The Netherlands
dejong@cs.uu.nl

Richard A. Watson
School of Electronics and Computer Science Faculty of Engineering, Science and Mathematics
University of Southampton
SO17 1BJ Southampton
raw@ecs.soton.ac.uk

Dirk Thierens
Institute of Information and Computing Sciences
Utrecht University
PO Box 80.089
3508 TB Utrecht, The Netherlands
dirk@cs.uu.nl

## ABSTRACT

We describe a generator for hierarchical problems called the Hierarchical Problem Generator (HPG). Hierarchical problems are of interest since they constitute a class of problems that can be addressed efficiently, even though high-order dependencies between variables may exist. The generator spans a wide ranges of hierarchical problems, and is limited to producing hierarchical problems. It is therefore expected to be useful in the study of hierarchical methods, as has already been demonstrated in experiments. The generator is freely available for research use.

## Categories and Subject Descriptors

F.2 [**Analysis of algorithms and problem complexity**]: General

## General Terms

Algorithms, Theory, Experimentation, Performance

## Keywords

Modularity, hierarchy, hierarchical problems, hierarchical problem generator, scalability, representation development

## 1. INTRODUCTION

Evolutionary algorithms research aims to address discrete optimization problems efficiently. Without any information or assumptions about the problem that will be addressed, finding a solution in a reasonable amount of time cannot be guaranteed in general, as the size of the search space is exponential in the number of variables.

An important class of problems that can be solved efficiently is that for which the dependencies between variables are limited to some small order $k$. Two variables are interdependent if the fitness contribution of one variable depends on the setting of the other, and independent otherwise. The order of the dependencies is the largest number of interdependent variables.

Problems limited to low order dependencies are efficiently solved by *competent genetic algorithms* (Goldberg, 2002); examples include the fast messy GA (Goldberg, Deb, Kargupta, & Harik, 1993), the gene expression messy GA (Kargupta, 1996), the linkage learning GA (Harik, 1997), the extended compact GA (Harik, 1999), the Bayesian Optimization Algorithm (BOA) (Pelikan, Goldberg, & Cantu-Paz, 1999), LFDA (Mühlenbein & Mahnig, 1999), and EBNA (Etxeberria & Larrañaga, 1999). However, the assumption that all interactions are limited to a small order may not hold in large problems of practical interest. An sensible question therefore is whether certain problems with higher order dependencies can also be addressed.

In the past few years, several *hierarchical* problems have been identified that can be addressed efficiently by methods able to exploit hierarchical structure, such as Hierarchical If and only IF (HIFF) (Watson, Hornby, & Pollack, 1998) and the Hierarchical Trap function (H-Trap) (Pelikan & Goldberg, 2001). Several methods have been introduced that are able to address hierarchical problems: SEAM (Watson & Pollack, 2000), Hiearchical BOA (H-BOA) (Pelikan & Goldberg, 2001), Compact Genetic Codes (Toussaint, 2005), and the Hierarchical Genetic Algorithm (HGA) (De Jong, Thierens, & Watson, 2004; De Jong, Watson, & Thierens, 2005).

In hierarchical problems, many or even all variables may be interdependent, but the dependencies are limited such that efficient problem solving is possible. Hierarchical structure is exploited in many human pursuits such as engineering and the organization of information (Simon, 1968); the ability to address hierarchical problems is therefore expected to bear relevance to significant real-world problems.

The first examples of hierarchical problems already provide an indication of the class of hierarchical problems, and an extensive discussion of the significance of hierarchy can be found in (Watson, 2002). A precise delineation of the class of hierarchical problems, excluding non-hierarchical problems, was first given in (De Jong et al., 2004). The same work gives an analysis of problem features that affect the computational requirements for hierarchical problem solving.

To study the behavior of algorithms on hierarchical problems, and the dependence of this behavior on problem features, it is essential to have a collection of hierarchical problems with different problem features. Two generators exist that can produce hierarchical problems (Watson et al., 1998; Pelikan & Goldberg, 2000), but these generators are not limited to producing hierarchical problems. A generator that can produce hierarchical problems and only produces such problems would therefore be a useful tool in analyzing hierarchical methods. In this paper, we present such a generator.

The structure of this paper is as follows. First, in Section 2 we briefly describe what we mean by hierarchical problems. Section 3 discusses related work. Section 4 discusses how a generator can be limited to constructing hierarchical problems. In Section 5, we present the generator; an example of the problems it generates is described in Section 6. Finally, Section 7 concludes.

## 2. HIERARCHICAL PROBLEMS

Informally, the idea behind module formation is to identify sets of variables for which only some settings need to be considered, while the remaining settings can be safely disregarded for the remainder of the search. A hierarchical problem is characterized by a hierarchy of modules; each module consists of a number of smaller, non-overlapping modules, and smallest modules at the bottom of the hierarchy are the variables of the problem. By recursively combining small modules into larger modules, a hierarchical problem can be addressed efficiently, even though it can feature high order dependencies.

In the following, we describe the notion of modularity and hierarchy more precisely. For formal definitions and a more detailed discussion, the reader is referred to (De Jong et al., 2004).

*Definition* (**Primitive Module**): Any variable in the problem is a *primitive module*. The *possible settings* for a primitive module are given by the alphabet of the problem, e.g. $\{0,1\}$ for binary problems.

*Definition* (**Context**): The *context* of a module is its complement, i.e. the set of variables that are not part of the module.

*Definition* (**Context-optimal settings**): A setting for a module is *context-optimal* if this setting yields the highest fitness achievable given the setting of the context.

*Definition* (**Possible settings**): The possible settings for a module combination are given by the product of the context-optimal settings of its components. Using this concept, the notion of a composite module is defined recursively as follows.

*Definition* (**Composite Module**): A combination of $k$ modules is a *composite module* if and only if the number of context-optimal settings for the combination is lower than its number of possible settings, and no subset of the module combination establishes such a reduction.

*Definition* (**Modularity**): Both primitive modules and composite modules are modules. A problem is called *modular* if it contains at least one composite module.

*Definition* (**Hierarchy**): A problem is called *hierarchical* if there is at least one composite module that contains another composite module.

### 2.1 Example: HIFF

As an example, we consider a 4-bit instance of the HIFF problem introduced in (Watson et al., 1998), and show it is hierarchical according to the above definitions. HIFF defines

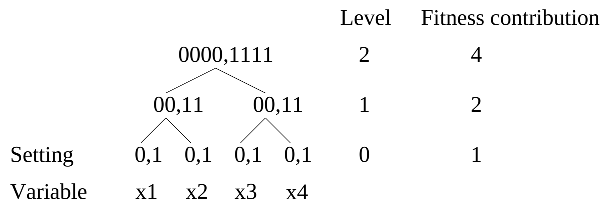

**Figure 1: Settings conferring fitness contributions in HIFF.**

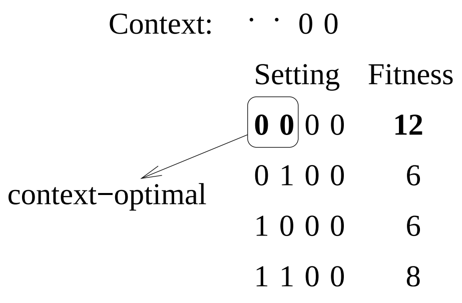

**Figure 2: Context-optimality: a setting is context-optimal if it maximizes fitness given the context.**

a number of blocks, each of which confers a fitness contribution when present in the individual; see Figure 1. As an example, the individual 0011 receives a fitness contribution of 1 for each of the individual variables, and 2 for the 00 and 11 blocks at positions 1,2 and 3,4, but no size-4 blocks are present that confer fitness. This results in a total fitness of 4+2+2=8. The two global optima are 0000 and 1111, both receiving a fitness of 16.

We now show that this problem is hierarchical. First, all four variables are primitive modules. For each variable, both settings are context-optimal; for example, for $x1$, the optimal setting in the context ·000 is 0, while the optimal setting for context ·111 is 1. For the module combination x1x2, the possible settings are therefore given by all four possible settings for two binary variables. The context-optimal settings for this module combination however are 00 and 11. Thus, the number of context-optimal settings for this module combination (2) is lower than the number of possible settings (4), which implies the module combination is a module; likewise for the module combination x3x4. For x2x3 however, no similar reduction can be made, as all four combinations are optimal in some context. Finally, by the same principle, the two modules x1x2 and x3x4 form a composite module x1x2x3x4. Since this composite module contains composite modules, the problem is hierarchical.

Figures 2 and 3 exemplify the notions of context-optimality and modularity, based on the example.

## 3. RELATED WORK

In (Watson et al., 1998), a construction that can produce hierarchical problems is defined as follows. A *transform function* accepts a set of module settings, and returns a single scalar value representing the setting for the combination of the modules concerned; thus, the combination can be seen as a new higher order variable. In the HIFF

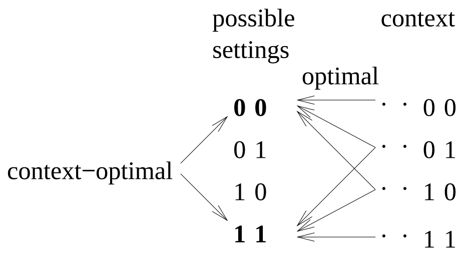

**Figure 3: Modularity: A combination of variables or modules forms a module if the number of context-optimal settings is reduced compared to the number of possible settings.**

example, 00 translates to 0 and 11 to 1, but the remaining, non-context-optimal settings (all other combinations of 0, 1, and '-') translate to a '-'. The fitness function is defined as a function over the module settings thus obtained; only combinations of all zeroes or all ones transform to a 0 or a 1, indicating these settings contribute to the fitness. In (Pelikan & Goldberg, 2000), a similar construction is defined, with the difference that different transformations are permitted at different places, and that modules can differ in size.

While both constructions can be used to specify hierarchical problems, they are not limited to producing hierarchical problems. For example, a one-max problem can be defined by transforming combinations of all ones to a 1, and any other combinations to '-'. For each variable in a one-max problem, the only context-optimal setting is 1. Thus, no further reductions of the number of context-optimal settings is possible by combining variables into modules, and the problem is clearly not hierarchical. In fact, both constructions can generate any given GA problem. An important question therefore is how a generator can be constructed that produces hierarchical problems only. We will consider this question in the following section.

## 4. GENERATING HIERARCHICAL PROBLEMS

A hierarchical problem is defined by a hierarchy of modules, each of which has a number of context-optimal settings that is smaller than its number of possible settings, where the latter is given by the product of the context-optimal settings of the module's components. To generate hierarchical problems therefore, we would like to be able to choose which combinations of variables will form modules in a problem, and which settings are context-optimal for these modules. This encompasses three goals that must be satisfied:

- Every chosen combination of variables forms a module
- No other combination of variables forms a module
- The selected settings for each chosen combination are context-optimal, and no other settings are context-optimal

The first condition can be satisfied by choosing the context-optimal settings for each module such that their number is less than the product of the number of context-optimal settings of the components of the module. Before considering the second condition, we first discuss the third condition.

The chosen settings for a module $M$ at level $l$ in the tree of modules are context-optimal if the following conditions hold:

- When considering all fitness contributions up to level $l$, the selected settings confer a higher fitness contribution than the remaining settings. This can be achieved by assigning sufficiently high fitness contributions to the selected settings of $M$.
- Fitness contributions at levels above $l$ are chosen such that the selected settings are still context-optimal. This can be achieved by ensuring that each chosen setting at level $l$ forms part of at least one context-optimal setting at level $l+1$; in this manner, the selected settings are propagated up to higher levels, and as a result the selected settings at the lowest level are guaranteed to form part of context-optimal settings at the highest level, thus ensuring their context-optimal status is maintained throughout the hierarchy.

The remaining second requirement is to ensure that no unintended combinations of variables should form modules. Thus, for any combination of modules not intended to form a module, all possible settings should be context-optimal. First, we consider the case where the number of components per module $k = 2$ and the number of context-optimal settings per module $M$ equals the arity of the alphabet $s$.

For any unintended combination of two modules $M1$ and $M2$, let $A$ be the smallest true module containing $M1$ and let $B$ be the smallest module containing $M2$. $A$ consists of two modules, $a1$ and $a2$, and $B$ consists of $b1$ and $b2$. Without loss of generality, we will assume $M1 = a1$ and $M2 = b1$.

Both $a1$ and $a2$ have $m$ context-optimal settings, and these all occur in some context-optimal setting of $A$. Since $A$ itself has only $m$ context-optimal settings, each context-optimal setting of $a2$ is uniquely associated with one context-optimal setting for $a1$, and vice-versa. For any context-optimal setting of $a1$ therefore, we can select a setting for $a2$ such that $a1$'s setting is context-optimal. Likewise, for any setting of $b1$, we can select a setting for $b2$ such that $b1$'s setting is context-optimal. Since the fitness contributions of $A$ and $B$ are independent, this means that for any combination of context-optimal settings for $a1$ and $b1$, a setting for $a2b2$ exists such that $a1b1$'s setting is context-optimal. Thus, the number of context-optimal settings for $a1b1$ equals the number of possible settings, and the combination is not a module. For the lowest level, the above requires that $s = m$.

The above shows that for $k = 2$ and $s = m$, no unintended combination of modules forms a module. For higher values of $k$, ensuring that no unintended modules arise becomes more complicated. For the case of $k = 3$ for example, for any size two subset of an unintended module, all $m^2$ combinations of context-optimal settings must form part of a context-optimal setting. In the binary case ($s = 2$) for example, this means that for all three two-variable combinations in a level one module, all four combinations (00,01,10,11) must form part of context-optimal settings. This requires $m \geq 4$, with the following context-optimal settings as an example: 000, 101, 110, 011. While it is technically possible to ensure that no unintended modules arise, this would

```
HPG()
$level = 0$;
$modules[0] = [x_0, x_1, \ldots, x_n]$
while $|modules[level]| <= kmin$
do
    $todo = modules[level]$
    $level{+}{+}$;
    while $|todo| >= kmin$
    do
        $M = new\ module()$;
        $[M.parts, todo] = choose-parts()$;
        $m = rand(mmin, mmax)$;
        $M.values = use-previous-layer()$;
        if $nosettings < m$
          then
                $options = create-options(M)$;
                $M.values = [M.values\ |$
                   $select-options(options, M)]$;

        $modules[level] = [modules[level] | M]$;

    $modules[level] = [modules[level] | todo]$;

$assign-fitness()$;

```

**Figure 4: Outline of the Hierarchical Problem Generator.**

substantially complicate the design of a problem generation algorithm. Instead, we choose to use the principles for generating modules that have been described. This approach guarantees that the intended modules will indeed be modules, while for certain combinations of parameters additional modules may exist. These do not jeopardize the hierarchical status of the problems that are generated, as reductions at all selected levels of the hierarchy are guaranteed.

## 5. THE HIERARCHICAL PROBLEM GENERATOR

The Hierarchical Problem Generator accepts the following parameters: a range from which the size of each module is randomly drawn $[k_{min}, k_{max}]$, a range from which the number of context-optimal settings of a module is drawn $[m_{min}, m_{max}]$, the arity $s$ of the alphabet, and the number of variables $n$.

The algorithm operates as follows. First, the set of modules is initialized to the set of primitive modules, which equals the set of variables in the problem. This set provides the first layer of modules. The outer loop of the algorithm (see Figure 4) constructs the different levels of modules ascending order. The inner loop constructs the next layer of modules given the current layer. This is done by first combining the modules into module combinations, using the function `choose-parts`, and next choosing the context-optimal settings for these modules.

`choose-parts` randomly chooses $k'$ selected modules, where $k'$ is selected randomly from $[k_{min}, k_{max}]$, and removes these from `todo`. Choosing the context-optimal settings for a module works as follows. First, for each component module, all context-optimal settings must be used at least once.

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ |
|---|---|---|---|---|---|---|---|---|
| Level 3 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| Level 2 | | 1 | 1 | | 1 | | 1 | |
| | | 0 | 0 | | 0 | | 0 | |
| | 1 | | | 0 | | 1 | | 1 |
| | 0 | | | 1 | | 0 | | 0 |
| Level 1 | | | 1 | | | | 1 | |
| | | | 0 | | | | 0 | |
| | | | | 0 | | 1 | | |
| | | | | 1 | | 0 | | |
| | | 1 | | | 1 | | | |
| | | 0 | | | 0 | | | |
| | 0 | | | | | | | 0 |
| | 1 | | | | | | | 1 |

**Table 1: Modules and their context-optimal settings for an example problem generated by the HPG.**

Therefore, `UsePreviousLayer` randomly selects an unused context-optimal setting from each component module (if possible, and a randomly chosen context-optimal setting otherwise) and combines these into a context-optimal setting for the module combination until all context-optimal settings have been used at least once. $m$ is selected randomly from $[m_{min}, m_{max}]$. In a balanced hierarchical problem with fixed $m$, each component module contains $m$ context-optimal settings, so this will produce precisely $m$ context-optimal settings for the module combination. If $m$ is higher than the $m'$ at the previous level, additional settings may need to be selected. To this end, `create-options` generates *all* combinations of context-optimal setting of the components of the module combination. Next, `select-options` removes from these the settings that were already generated by combining $m'$ randomly selected context-optimal component settings. The remaining $m - m'$ settings are chosen randomly from the remaining settings. Finally, `assign-fitness` assigns a fitness to each module that equals the number of variables in the module.

The above describes the default operation of the generator. Two additional options are available. By default, module combinations are formed by combining randomly selected modules. If the `randomorder` parameter is set to false, the modules are formed by selecting consecutive modules; this can be useful in analyzing the results of a method on a problem. Furthermore, by default, the fitness contribution of a module equals its number of variables. If the `fixedfitnesscontribution` parameter is set to true, the context-optimal module settings are instead assigned a constant value (one).

It is interesting to note that for $s = m$, the problems produced by the generator are equivalent, up to a relabeling of the context-optimal settings, to the H-EQUAL problem described in (Watson & Pollack, 1999), which can be viewed as a generalized version of H-IFF.

## 6. RESULTS

In the following, we describe an example problem produced by the generator; see Tables 1 and 2. The parameters used were $n = 8$ and $k_{min} = k_{max} = m_{min} = m_{max} = s = 2$. At the lowest level, all variables are primitive modules, and have both settings (0 and 1) as context-optimal settings;

| Example | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | Fitness |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 14 |
| 2 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 12 |
| 3 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 10 |
| 4 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 12 |
| 5 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 12 |
| 6 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 10 |
| 7 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 10 |
| 8 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 20 |
| 8 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 8 |
| 10 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 14 |

**Table 2: Example evaluations for the problem shown in Table 1.**

for space considerations, these are not shown. At the first level, four modules exist: $x_1x_8$, $x_2x_5$, $x_4x_6$, and $x_3x_7$. Since the $m$ context-optimal settings of the lowest, single-variable level all feature in precisely one context-optimal setting at this first level, all modules either have optimal settings 00 and 11 or 01 and 10. At the second level, modules are formed by randomly combining two first-level modules. The possible settings for these modules are given by all four combinations of the two context-optimal settings for each constituent level one module. Again, two out of these four combinations are selected to form the context-optimal settings at the second level. Analogously, the third level represents two size eight modules, each contributing an additional fitness value of eight when present in an individual. As an example, table 2 shows fitness evaluations for ten randomly generated individuals.

The use of the HPG is demonstrated in a paper at this conference (De Jong et al., 2005). There, the Hierarchical Genetic Algorithm (HGA) is tested by applying it to different ranges of randomly generated hierarchical problems. The experiments empirically validate the scalability analysis of the algorithm by measuring the effect of the different problem parameters ($k$, $m$, and $m$) on performance. These experiments demonstrate the value of the Hierarchical Problem Generator as a tool in evaluating hierarchical methods.

## 7. CONCLUSIONS

A generator for hierarchical problems has been presented. The generator can produce a wide range of hierarchical problems, and is limited to producing such problems. It is therefore valuable in the study of hierarchical algorithms, as demonstrated in related work (De Jong et al., 2005).

The generator accepts several parameters: the number of variables in the problem, the number of components per module, the number of context-optimal settings per module, and the alphabet size. These parameters span a wide space of problems. As a possible extension, intermediate problems can be generated for which one or more of the latter three properties vary per module.

## Availability

The Hierarchical Problem Generator is freely available for research use from the homepage of the first author: `http://www.cs.uu.nl/~dejong/hpg`.

## References

De Jong, E. D., Thierens, D., & Watson, W. (2004). Hierarchical genetic algorithms. In Yao, X., Burke, E., Lozano, J. A., Smith, J., Merelo-Guervós, J. J., Bullinaria, J. A., Rowe, J., Tiňo, P., Kabán, A., & Schwefel, H.-P. (Eds.), *Parallel Problem Solving from Nature - PPSN VIII*, Vol. 3242 of *LNCS*, pp. 232–241, Birmingham, UK. Springer-Verlag.

De Jong, E. D., Watson, R. A., & Thierens, D. (2005). On the complexity of hierarchical problem solving. In Beyer, H.-G. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-05.*

Etxeberria, R., & Larrañaga, P. (1999). Global optimization using bayesian networks. In Rodriguez, A. O., Ortiz, M. S., & Hermida, R. S. (Eds.), *Proceedings of the Second Symposium on Artificial Intelligence CIMAF.*

Goldberg, D. E. (2002). *The design of innovation. Lessons from and for competent genetic algorithms.* Kluwer Academic Publishers.

Goldberg, D. E., Deb, K., Kargupta, H., & Harik, G. (1993). Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 56–64.

Harik, G. (1997). *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms.* Ph.D. thesis, University of Michigan.

Harik, G. (1999). Linkage learning via probabilistic modeling in the ECGA. Tech. rep. Illigal report no. 99010, University of Illinois at Urbana-Champain, Urbana, IL.

Kargupta, H. (1996). SEARCH, evolution, and the gene expression messy genetic algorithm. Tech. rep. LA-UR 96-60, Los Alamos National Laboratory.

Mühlenbein, H., & Mahnig, T. (1999). FDA - A scalable evolutionary algorithm for the optimization of additively decomposed functions. *Evolutionary Computation*, *7*(4), 353–376.

Pelikan, M., & Goldberg, D. E. (2000). Hierarchical problem solving by the bayesian optimization algorithm. In Whitley et al., D. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pp. 267–274, Las Vegas, Nevada, USA. Morgan Kaufmann.

Pelikan, M., & Goldberg, D. E. (2001). Escaping hierarchical traps with competent genetic algorithms. In Spector et al., L. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-01*, pp. 511–518. Morgan Kaufmann.

Pelikan, M., Goldberg, D. E., & Cantu-Paz, E. (1999). BOA: The bayesian optimization algorithm. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, Vol. 1, pp. 525–532, San Francisco, CA. Morgan Kaufmann.

Simon, H. A. (1968). *The Sciences of the Artificial.* The MIT Press, Cambridge, MA.

Toussaint, M. (2005). *Foundations of Genetic Algorithms 8 (FOGA 2005)*, chap. Compact genetic codes as a search strategy of evolutionary processes.

Watson, R. A. (2002). *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis*. Ph.D. thesis, Brandeis University.

Watson, R. A., Hornby, G. S., & Pollack, J. B. (1998). Modeling building-block interdependency. In Eiben, A., Bäck, T., Schoenauer, M., & Schwefel, H.-P. (Eds.), *Parallel Problem Solving from Nature, PPSN-V.*, Vol. 1498 of *LNCS*, pp. 97–106, Berlin. Springer.

Watson, R. A., & Pollack, J. B. (1999). Hierarchically consistent test problems for genetic algorithms. In Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., & Zalzala, A. (Eds.), *Proceedings of the Congress on Evolutionary Computation*, Vol. 2, pp. 1406–1413, Mayflower Hotel, Washington D.C., USA. IEEE Press.

Watson, R. A., & Pollack, J. B. (2000). Symbiotic combination as an alternative to sexual recombination in genetic algorithms. In Schoenauer et al., M. (Ed.), *Parallel Problem Solving from Nature, PPSN-VI*, Vol. 1917 of *LNCS*, Berlin. Springer.