Representation Development from Pareto-Coevolution

Edwin D. de Jong

DSS Group, Utrecht University, The Netherlands
dejong@cs.uu.nl http://www.cs.uu.nl/~dejong/

Abstract. Genetic algorithms generally use a fixed problem representation that maps variables of the search space to variables of the problem, and operators of variation that are fixed over time. This limits their scalability on non-separable problems. To address this issue, methods have been proposed that coevolve explicitly represented modules. An open question is how modules in such coevolutionary setups should be evaluated.
Recently, Pareto-coevolution has provided a theoretical basis for evaluation in coevolution. We define a notion of functional modularity, and objectives for module evaluation based on Pareto-Coevolution. It is shown that optimization of these objectives maximizes functional modularity. The resulting evaluation method is developed into an algorithm for variable length, open ended development of representations called *DevRep*. DevRep successfully identifies large partial solutions and greatly outperforms fixed length and variable length genetic algorithms on several test problems, including the 1024-bit Hierarchical-XOR problem.

Keywords: Development of representations, hierarchical modularity, Pareto-coevolution, Evolutionary Multi-Objective Optimization

In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*. In Press.

1 Introduction

Most genetic algorithms employ a single, fixed representation that is given as part of the problem specification. To apply a genetic algorithm to a problem, a mapping has to be chosen between *genotypes* (sequences of binary or other variables) and the actual individuals they represent, *phenotypes*. We call this mapping the *representation* of the problem. For most genetic algorithms, the representation is chosen once, and does not change during the algorithm's operation. The operators of variation are typically constant too. Thus, the search space and the operators specifying the possible moves in this space cannot be adapted by the algorithm.

In combination, the use of a fixed representation and fixed operators of variation make it unlikely that different combinations of large partial solutions will be explored, see [14]. Partial solutions take the form of schemata, i.e. partial specifications of a genotype. Both forming large partial solutions and mixing

existing genotypes can be achieved in isolation. However, the standard genetic algorithm makes it unlikely for mutually exclusive partial solutions[1] to persist, since no mechanism for protecting mutually exclusive partial solutions is present. Methods that cannot represent information about mutually exclusive partial solutions are unlikely to address non-separable problems[2].

When niching is used, mutually exclusive partial solutions can persist. Still, unless they are explicitly represented, crossover is unlikely to respect the boundaries of such partial solutions. This limits the potential to combine large partial solutions. A solution is to represent partial solutions, *modules*, explicitly. Several methods taking this approach have been investigated so far, including GLiB [1], ADF's [8], ARL [12], ADSN [7], and SEAM [20].

The development of modules allows algorithms to start searching in terms of *combinations* of variables. Thus, such algorithms can be viewed as adapting the *representation* of the problem during search [1, 12, 3]. H-BOA, an apparently quite different approach to address hierarchical problems, also represents partial solutions explicitly, by using decision trees [10].

So far, there has been a lack of theory to guide the development of algorithms forming modules, in particular regarding module evaluation. Methods that simultaneously evolve modules and assemblies are instances of coevolution, as module evaluation is based on interactions with assemblies. Recently, the paradigm of *Pareto-coevolution* has provided a theoretical basis for evaluation in coevolution [6, 19, 2, 4].

Here, we will apply Pareto-coevolution to the question of how modules may be evaluated. The first algorithm to use Pareto-coevolution for module evaluation is Watson's SEAM algorithm [20]. SEAM is designed for fixed length problems. We study the question of how combinations of large partial solutions may be explored for *variable length* problems in an open-ended setup where modules can be combined into new modules recursively and indefinitely. The problems we are interested in are large search problems that have *structure*. A search problem is large if it requires a long solution, in terms of the original variables. If information from part of the search space can be used to predict (better than random) information about other parts of the search space, we will say the problem has structure.

The structure of the paper is as follows. First, our algorithm is gradually introduced, by discussing structural and functional modularity (2), coevolution of modules and assemblies and Pareto-coevolution (Section 3), evaluation of modules based on Pareto-coevolution (Section 4), and finally the algorithm following from this principle (Section 5). The test problems are described in Section 6. Experimental results are reported in Section 7, followed by discussion and conclusions.

---

[1] Two partial solutions are mutually exclusive if they assign conflicting values to one or more variables.

[2] A problem is *separable* if each variable has a single optimal setting, independent of the other variables [16].

2 Structural versus Functional Modularity

We will distinguish between *structural modularity*, a characteristic of algorithms, and *functional modularity*, the modularity present in a problem. Watson [17] defines notions of structural and functional modularity based on the structure and behavior of a dynamical system. Here, we will say any method that represents partial solutions explicitly features *structural modularity*. While any grouping of elements leads to structural modularity, the value of a modular representation strongly depends on the particular grouping that is chosen. Ideally, the structural modules constructed by an algorithm should correspond to the functional modules present in the problem.

The primitives of a problem are the basic elements that may occur in a genotype, typically numerical values, e.g. $\{0, 1\}$, or actions or operators. A **module** is a sequence of primitives, and has a unique identifier. We will assume that for every primitive there is a module containing only that primitive. Thus, without loss of generality, individuals can be viewed as sequences of modules. Candidate solutions are sequences of modules, and are called **assemblies**.

The **compact** representation of an assembly describes the assembly in terms of the modules of which it consists, using the modules' unique ID's. The assembly's **expressed** form consists of the concatenation of the sequences of primitives represented by its modules. The sizes of the compact and expressed representations of an assembly are called its compact and expressed size. Since an assembly is expressed by concatenating the expressed forms of its modules, the position of a module, and hence of its primitives, is determined by the number and size of the modules that precede it in the assembly.

The functional modularity of a module is considered with respect to some set of assemblies, called the **context set**. We will use the operation of replacing the module at a given position within an assembly by another module. A position used in this way will be called an **insertion point**.

Using the above concepts, a definition for functional modularity in variable length search problems can now be stated. Let $\mathbb{S}$ be a context set and let $\mathbb{C}$ be a set of comparison modules.

Definition 1 (Functional Modularity). *A module A is functionally modular with respect to $\mathbb{S}$, insertion points i_S, and $\mathbb{C}$ iff:*

$$\forall S \in \mathbb{S} : \forall C \in \mathbb{C} : f(S(A, i_S)) \geq f(S(C, i_S))$$

where $S(A, i)$ specifies placing module A at the i^{th} position of S, and $f(S)$ returns the fitness of an assembly S. Functionally modular modules are also simply called functional modules.

3 Coevolution of Modules and Assemblies

We study how the development of functional modules can be achieved in a setup where modules and assemblies coevolve. A coevolutionary approach to

module formation is obtained by using two populations, one containing modules and one containing assemblies. Assemblies are candidate solutions for the problem, and hence their evaluation is given by the fitness function of the problem.

In Evolutionary Multi-Objective Optimization (EMOO [13]), individuals are evaluated on *multiple* objectives instead of a single fitness function, and the values of these objectives are treated separately; for an introduction, see e.g. [5].

The central idea in Pareto-coevolution is to view the outcomes of interactions with other evolving individuals as *objectives*. Treating the outcomes of interactions separately provides more specific information about individuals than a single value such as the average outcome, permiting better-informed methods of selection.

4 Module Evaluation: Assemblies Provide Objectives

This section shows how the Pareto-coevolution view leads to a principle for module evaluation. It will be seen that optimization of the objectives corresponds to the optimization of functional modularity. This provides a connection between Pareto-coevolution and functional modularity.

4.1 Objectives from Pareto-Coevolution

Assemblies provide situations in which modules can perform useful roles, and thereby implicitly define objectives. The maximal set of objectives that can be considered for a module therefore consists of the union of the objectives defined by some set of assemblies, e.g. a subset of the coevolving assembly population. For an individual assembly, the objective of a module in it is to contribute to the assembly's fitness by performing a useful role in it. Assemblies define a number of positions at which modules can perform a useful role. Thus, in an assembly containing n modules, each of the n positions defines an objective. For a module A, the value of the i^{th} objective of assembly S is obtained by using i as an insertion point, and considering the fitness of the assembly resulting from placing A at the insertion point: $f(S(A, i))$.

The numerical value of this objective equals the fitness of the complete assembly, and is not informative by itself. The logic behind this choice of objectives becomes clear when *comparing* the values these objectives assign to different modules. Intuitively, a module A is more valuable than another module B for a given assembly if using A instead of B has a positive effect on the overall fitness of the assembly. This is precisely what is measured when the objective values of two modules A and B are compared; A has a higher objective value than B for position i of an assembly S if $f(S(A, i)) > f(S(B, i))$. This comparison turns out positive for A if A, when replacing B at the i^{th} position of S, results in a higher overall fitness for the assembly.

Using individual assemblies as objectives for the modules they contain allows for the identification of many different specialized roles or tasks. A module can in principle be valuable even if it is only used by a small number of

assemblies, or if only some of the assemblies employing it have high fitness, while an average fitness approach would not detect the value of such a module.

Evaluation by replacing a module with other modules and comparing the overall fitness is a form of *differential fitness comparison*. This principle has been used in various forms, e.g. Cooperative Coevolution [11], COIN's Wonderful Life Utility [15], and SEAM [20].

4.2 Correspondence between Pareto-Coevolution and Functional Modularity

The previous subsection has shown how using Pareto-coevolution, assemblies can provide objectives for modules. An important question is how the resulting objectives relate to the earlier notion of functional modularity. We show that there is a direct correspondence between these.

Let A be a candidate module, chosen from a set of all possible modules $\mathbb{C}$, and let $\mathbb{S}$ be a set of assemblies. Then A is functionally modular with respect to $\mathbb{S}$, insertion points i_S, and $\mathbb{C}$ if and only if:

$$\forall S \in \mathbb{S} : \forall C \in \mathbb{C} : f(S(A, i_S)) \geq f(S(C, i_S)) \tag{1}$$

Now consider the objectives specified by the same assemblies $\mathbb{S}$ and insertion points i_S. As defined in the previous section, these are given by $f(S(A, i_S))$ for all $S \in \mathbb{S}$. A maximizes these objectives simultaneously over $\mathbb{C}$ if and only if:

$$\forall S \in \mathbb{S} : \forall C \in \mathbb{C} : f(S(A, i_S)) \geq f(S(C, i_S)) \tag{2}$$

Equation 1 and 2 are identical. Thus, a module is functionally modular for a set of assemblies and corresponding insertion points if and only if it maximizes the objectives represented by these assemblies and insertion points.

4.3 Practical Issues in Module Evaluation

For a candidate module AB, we can replace all its occurrences in assemblies by a compact representation of the candidate module $X = AB$. After doing so, each occurrence of X in an assembly defines an objective (assembly and insertion point) that is likely to be relevant in evaluating X. For efficiency reasons, we limit module evaluation to these objectives, i.e. the objectives specified by the positions where X actually occurs.

Candidate modules are identified by considering consecutive pairs of modules that occur frequently in assemblies. One of the assemblies in which such a frequent candidate module occurs is selected, and defines an objective value for the candidate module. The candidate module is evaluated by comparing its value for the objective to that of other possible candidate modules in a comparison set $\mathbb{C}$. It is only accepted as a new module if its value for the objective is equal or greater than all alternatives in $\mathbb{C}$, and strictly greater than some alternatives. This condition ensures that no better candidate is available, and that the

candidate is an improvement over alternative combinations of modules. Thus, for given $\mathbb{C}$, S, and insertion point i_S:

$$\forall C \in \mathbb{C} : f(S(A, i_S)) \geq f(S(C, i_S)) \quad \wedge \quad \exists C \in \mathbb{C} : f(S(A, i_S)) > f(S(C, i_S))$$

A possible concern is to what extent evaluation based on a single objective is sufficient. In contrast with most EMOO work, the aim here is to identify modules *maximizing* performance in one or more objectives. Thus, we are only interested in the extremes of the tradeoff front. Since modules are added incrementally rather than evolved, the objectives can at least be optimized independently. To furthermore promote modules that maximize *multiple* objectives, candidate modules are combinations that occur frequently in assemblies.

To avoid unnecessary material, a candidate module is compared to alternatives of the same or smaller expressed length. An efficiency improvement can be made by viewing the constituents A and B of a module AB as modules. We thus compare a candidate module AB to all modules A* and *B, observing the length requirement, and only accept it if it obtains at least as high fitness as all of these and higher fitness than at least one of these. A final requirement is that its fitness is at least as high as when either or both constituent modules are left out; that is, the module is also compared to A, B, and [].

5 The DevRep Algorithm

DevRep()

```
1.  modules:=primitives;
2.  assemblies:=generate_random_sequences(modules);
3.  while(¬stop_criterion)
4.      modules := create_modules(assemblies, modules);
5.      for i=1:interval
6.          assemblies := evolve_assemblies(assemblies, modules);
7.      end
8.  end
```

Basic cycle of the DevRep algorithm.

The choices that have been made regarding module construction and evaluation lead to an algorithm that Develops a Representation for the problem as part of the search, and is therefore called *DevRep*. This method is based on earlier work presented in [3]. The population of modules is initialized to the set of primitives. The assembly population is initialized to random sequences of these modules of a given length. Next, the following loop is repeated until a stop criterion: pairs of existing modules occurring consecutively in the assemblies are considered for consolidation into new modules, and a generation of evolving the assemblies is performed.

$Create\_modules$ does the following. Let AB be the pair of existing modules consecutively occurring most frequently in the assembly population. One assembly in which AB occurs is selected randomly. Let us write this assembly as $XABY$, where X and Y represent sequences of modules. We now consider all assemblies $XA \cdot Y$ and $X \cdot BY$ in which either A or B has been replaced by some other module, whose expressed length does not exceed that of $XABY$. Then the fitness $f(XABY)$ must be at least as high as that of all of these modified assemblies, and higher than that of at least one of the modified assemblies:

$$\begin{aligned} \forall Z: \quad & f(XABY) \geq f(XAZY) \quad \wedge \quad f(XABY) \geq f(XZBY) \\ \exists Z: \quad & f(XABY) > f(XAZY) \quad \vee \quad f(XABY) > f(XZBY) \end{aligned}$$

If this is the case, we furthermore require that the compact representation of AB contains no unnecessary elements:

$$f(XABY) > f(XAY) \quad \wedge \quad f(XABY) > f(XBY) \quad \wedge \quad f(XABY) > f(XY)$$

If these requirements are met, the new module is given a unique ID, and added to the module population. Furthermore, all occurrences of AB in the current assemblies are replaced by the new module, followed by a null module to maintain the same assembly length.[3] If not, the next `max-modules-to-consider` most frequent pairs of modules are considered in order for consolidation until at most `max-modules-per-gen` new modules are found.

$Evolve\_assemblies$ is based on deterministic crowding [9]. The following cycle is repeated a number of times equal to the assembly population size. Two assemblies are selected randomly to function as parents. Offspring are produced by crossover with probability `pcross`, and by copying otherwise. The resulting assemblies are mutated at each element with probability `pmut`. Mutation replaces a module by a randomly selected element of the module population. Next, the parents are paired up with the offspring, such that the sum of the Hamming distances between the compact representations of the parent-offspring pairs is minimized. Each offspring replaces its matched parent if its fitness is equal or higher than that of its parent.

6 Test Problems

6.1 Hierarchical Test Problems

Several authors have recently studied the scalability of evolutionary algorithms [14, 20, 10]. As part of this, difficult hierarchical test problem were designed that yet contain structure, such as Hierarchical IF-and-only-iF (H-IFF) [18]. Preliminary experiments showed H-IFF is no longer difficult when modules can be repeated; 64-bit H-IFF was solved within a few generations, or a fraction of a second. We therefore test performance on the analogous H-XOR problem [18], which uses XOR instead of IF and only iF as its basic function, see Table 1. This problem is much more difficult for variable length methods due to its reduced potential for exploiting repetitiveness.

[3] If AB occurs more than once, all occurrences are replaced.

Level	H-IFF		H-XOR	
	A	B	A	B
4	0000000000000000	1111111111111111	0110100110010110	1001011001101001
3	00000000	11111111	01101001	10010110
2	0000	1111	0110	1001
1	00	11	01	10
0	0	1	0	1

Table 1. Target modules for the H-IFF and H-XOR problems. The target modules at each level are composed of those at the previous level, and are each other's inverse. By using XOR instead of IFF, repetition of a single type of module is no longer sufficient for solving the problem.

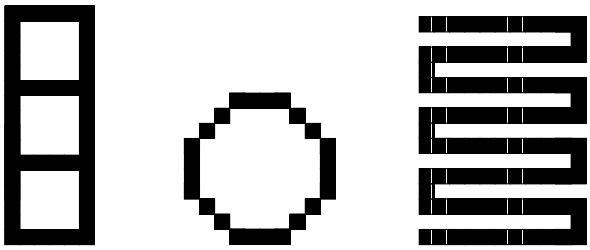

Fig. 1. Target images used in the experiments: squares, octagon, and path.

6.2 Pattern Generation

In the second test problem, the goal is to generate a picture using turtle graphics on a toroidal grid. The primitives for this problem are the following commands: TURN LEFT, TURN RIGHT, MOVE, and PUT PIXEL. The expressed form of an assembly is a sequence of these primitives. The interpretation of a sequence of primitives produces a bitmap, representing the concept specified by the assembly. The starting point for the interpretation of a sequence is the point from which the figure is drawn; thus, locating the target figure on the grid is not part of the task.

The target images are 16x16 bitmaps containing simple line drawings, see Figure 1. The two objectives are the number of black and white pixels correctly produced. Perfect solutions for these four problems require between 70 and 160 primitive operators.

7 Experimental Results

Here, we report experiments with the DevRep algorithm. Assemblies are of length 2 (H-XOR) or 10 (pattern generation). Other parameters are as follows:

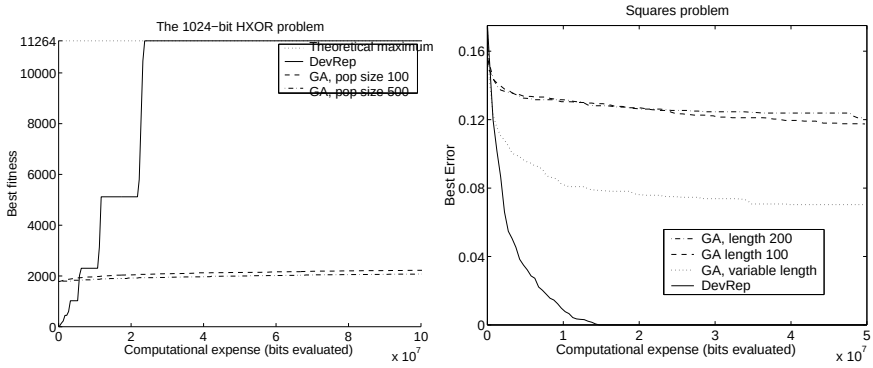

Fig. 2. Performance on the 1024-bit H-XOR problem (left) and the *squares* (right) problem.

`max-modules-to-consider` = 5, `max-modules-per-gen` = 2, `interval` = 50, `pcross` = .9, `pmut` = .1. All curves are averaged over ten runs.

A genetic algorithm variant of DevRep is obtained simply by omitting the module formation procedure. Furthermore, while in DevRep a child replaces its parent when all of its objective values are equal or higher, for the genetic algorithm we employ the standard Pareto-dominance criterion. We employ two genetic algorithm methods using fixed length representations of 100 and 200 primitives, and a variable length method, initialized with size 10 assemblies.

The experimental results are as follows. On the 64-bit H-XOR problem (not shown), all algorithms progressed substantially, while only DevRep reached the maximum score within the given number of bit evaluations and for all runs. On the 1024-bit version of the same problem, see figure 2, the difference in scalability between the methods becomes clear; while the genetic algorithm variants all stall at a low fitness level, DevRep is able to progress by repeatedly forming larger modules, and subsequently searching in terms of these modules. DevRep thereby again achieves maximum performance on the problem for all runs. Inspection of a run showed that the modules formed over time correspond precisely to the target modules for the problem, shown in table 1.

On the pattern generation tasks, fig. 2-3, all genetic algorithm variants performed poorly. Apparently, the biases of the genetic algorithm do not correspond well to those required for these problems, which are characterized by long range dependencies and by sequential structure. DevRep greatly improves over this performance. The average score substantially improved over the genetic algorithm methods, and while the GA methods were unable to find a correct solution for any of the problems, DevRep found perfect solutions for all problems, and in all runs except three of the 'path' problem runs.

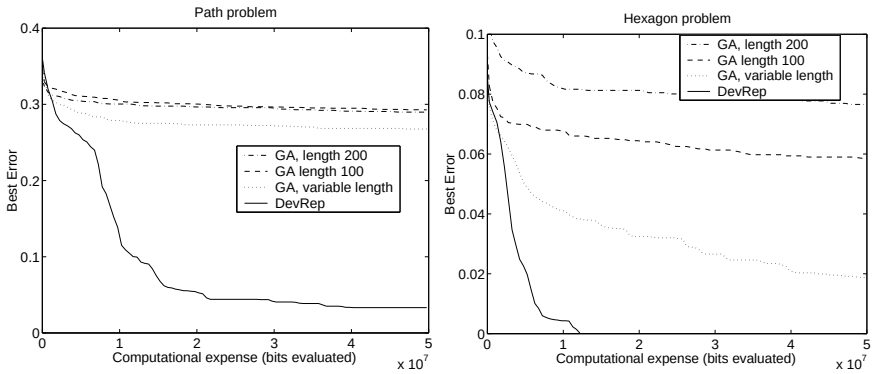

Fig. 3. Performance on the *path* (left) and *octagon* (right) problems.

8 Discussion

The DevRep algorithm shares several features with SEAM [20], most importantly the use of recursive module formation, leading to hierarchy, and the use of Pareto-coevolution for module evaluation; the latter distinguishes these algorithms from other methods for representation development. Compared to SEAM, the main new contribution is the application of this evaluation principle in a variable length setting. A related difference is the use of assemblies evolved on fitness, rather than random assemblies; in additional experiments, fitness based selection was found to be a necessary component.

A crucial idea in defining modularity for variable length problems was to consider modularity relative to specific subsets of all possible assemblies. This possibility is required to address non-separable problems such as H-IFF and H-XOR.

Search algorithms can be characterized by the types of patterns they are able to discover. For DevRep, these include the following:

- **Functional modularity** By maximizing the module objectives, the algorithm searches for modules that are functionally modular, as defined in section 2.
- **Hierarchical modularity** By looking for useful modules recursively, the algorithm searches for modules of hierarchical structure.
- **Repetitive modularity** Modules can be used repeatedly, i.e. a combination of two consecutive primitives or operators can be used multiple times within a single individual, due to the use of position-independent coding.

9 Conclusions

Problems that require long solutions pose difficulties to standard genetic algorithms, due to the size of the associated search spaces. Still, if such problems

have *structure*, they can in principle be addressed. While several authors have investigated the simultaneous development of modules and assemblies, principled evaluation of such partial solutions has long been an open issue. Here, we have derived objectives for module evaluation in variable length problems from Pareto-coevolution.

Functional modularity is defined, and it is shown that optimization of the objectives derived from Pareto-coevolution corresponds to optimization of functional modularity. Based on this evaluation principle, the DevRep algorithm for variable length problems is developed.

DevRep was tested on Hierarchical XOR (H-XOR) up to size 1024, and on pattern generation tasks. It was found to develop large and ideal partial solutions and greatly improve performance compared to a genetic algorithm approach. DevRep is able to exploit structure in certain large search problems, in particular *functional modularity*, *hierarchical modularity*, and *repetitive modularity*. This is achieved by recursively forming modules and searching the space of combinations of such modules, thus forming modules in an open ended way.

We conclude that certain forms of structure in large search problems can be exploited by gradually consolidating learned information. Here the patterns that are detected are templates, but in principle any type of detectable pattern can be considered. According to this view, a challenge for research into large search problems is to identify the patterns present in problems of interest, and to develop corresponding algorithms exploiting those patterns.

Acknowledgements

The author wishes to thank the Agents Systems Group at the Vrije Universiteit and the members of the DEMO Lab at Brandeis, particularly Sevan Ficici, Richard Watson, and Anthony Bucci, who gave valuable feedback on this work. A TALENT-fellowship from the Netherlands Organisation for Scientific Research (NWO) is gratefully acknowledged.

References

1. Peter J. Angeline and Jordan B. Pollack. Coevolving high-level representations. In Christopher G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 55–71, Redwood City, CA, 1994. Addison-Wesley.
2. Anthony Bucci and Jordan B. Pollack. Order-theoretic analysis of coevolution problems: Coevolutionary statics. In *Proceedings of the GECCO-2002 Workshop on Coevolution: Understanding Coevolution*, 2002.
3. Edwin D. De Jong and Tim Oates. A coevolutionary approach to representation development. In E.D. de Jong and T. Oates, editors, *Proceedings of the ICML-2002 Workshop on Development of Representations*, Sydney NSW 2052, 2002. The University of New South Wales. Online proceedings: `http://www.demo.cs.brandeis.edu/icml02ws`.

4. Edwin D. De Jong and Jordan B. Pollack. Learning the ideal evaluation function. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2003*, 2003.
5. Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley & Sons, New York, NY, 2001.
6. Sevan G. Ficici and Jordan B. Pollack. Pareto optimality in coevolutionary learning. In Jozef Kelemen, editor, *Sixth European Conference on Artificial Life*, Berlin, 2001. Springer.
7. Frederic Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, PhD Thesis, Ecole Normale Supérieure de Lyon, 1994.
8. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, Cambridge, MA, May 1994.
9. Samir W. Mahfoud. *Niching Methods for Genetic Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, May 1995. IlliGAL Report 95001.
10. Martin Pelikan and David E. Goldberg. Escaping hierarchical traps with competent genetic algorithms. In L. Spector, E.D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, pages 511–518, San Francisco, CA, 2001. Morgan Kaufmann.
11. Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.
12. Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In P.J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. The MIT Press, Cambridge, MA, 1996.
13. J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 93–100, Hillsdale, NJ, 1985. Lawrence Erlbaum Associates.
14. Dirk Thierens. Scalability problems of simple genetic algorithms. *Evolutionary Computation*, 7(4):331–352, 1999.
15. Kagan Tumer and David Wolpert. Collective intelligence and Braess paradox. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 104–109, Menlo Park, CA, 2000. AAAI Press.
16. Richard A. Watson. *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis*. PhD thesis, Brandeis University, 2002.
17. Richard A. Watson. Modular interdependency in complex dynamical systems. In Bilotta et al., editor, *Workshop Proceedings of the 8th International Conference on the Simulation and Synthesis of Living Systems*. UNSW Australia, 2003.
18. 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.
19. Richard A. Watson and Jordan B. Pollack. Symbiotic combination as an alternative to sexual recombination in genetic algorithms. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. Julian Merelo, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature, PPSN-VI*, volume 1917 of *LNCS*, Berlin, 2000. Springer.
20. 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.