Evolving Reusable Neural Modules

Joseph Reisinger, Kenneth O. Stanley, and Risto Miikkulainen

Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-1188
{joeraii,kstanley,risto}@cs.utexas.edu
http://www.cs.utexas.edu/~{joeraii,kstanley,risto}

Abstract. Topology and Weight Evolving Artificial Neural Networks (TWEANNs) have been shown to be powerful in nonlinear optimization tasks such as double pole-balancing. However, if the input, output, or network structures are high dimensional, the search space may be too large to search efficiently. If the symmetries inherent in many large domains were correctly identified and used to break the problem down into simpler sub-problems (to be solved in parallel), evolution could proceed more efficiently, spending less time solving the same sub-problems more than once. In this paper, a coevolutionary modular neuroevolution method, Modular NeuroEvolution of Augmenting Topologies (Modular NEAT), is developed that automatically performs this decomposition during evolution. By reusing neural substructures in a scalable board game domain, modular solution topologies arise, making evolutionary search more efficient.

1 Introduction

Topology and Weight Evolving Artificial Neural Network (TWEANN) methods become increasingly intractable as the network complexity of the individual solutions (and thus the complexity of the space of solutions) being searched increases. This complexity is known as *dimensionality*, and is influenced primarily by two factors: the number of inputs and outputs a solution network has, and the topological complexity (e.g. amount of hidden neurons) of that network. Many board games are difficult to learn because they have high dimensional input and output spaces (i.e. the board size is large). Even powerful Neuroevolution (NE) methods such as NeuroEvolution of Augmenting Topologies (NEAT) are challenged in these domains because the search space is too large [1]. In this paper, an NE method is proposed to increase the efficiency of high-dimensional search.

The dimensionality problem exists in biological evolution, however nature is able to overcome it and produce exceedingly complex phenotypes through modularization of the genotype-phenotype map [2]. Also, developmental modularity allows for reuse of complex phenotypic structures without correspondingly complex mutations in the genotype. Modularization can be thought of as a decomposition operation that reconstitutes the search space in a way that is more amenable to natural evolution [3].

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 69–81, 2004.
© Springer-Verlag Berlin Heidelberg 2004

The key to tackling the dimensionality problem in artificial evolution, then, is to find ways to perform a similar modular decomposition of the genotypic search space. One way to accomplish this decomposition in NE is to evolve reusable neural substructures (modules) and combine them together.

With the added level of abstraction that modules provide, more complex solutions are searched with fewer operations than standard NE. The trade-off is that the search becomes coarser. For example, mutations represent combinations of modular building blocks instead of weights, making it difficult or even impossible to find some solutions.

In this paper, a coevolutionary NE method Modular NeuroEvolution of Augmenting Topologies (Modular NEAT) is proposed that allows reusing fundamental neural substructures, which encourages modularization. Modular NEAT is tested on an artificial board game domain with scalable input/output complexity, and is shown to be 1) several times more efficient than the standard NEAT method and also 2) generate modular solution networks. This approach allows for the solutions to a class of difficult problems to be approximated more easily than with non-modular NE techniques.

2 Modularity in Natural Evolutionary Systems

Modularity can be intuitively defined as "the integration of functionally related structures and the dissociation of unrelated structures" [4]. In other words the structures are some combination of reusable, interchangeable, and functionally separate. A technical discussion of modularity is given in [5] and also [6]. Independent functions are coded independently using modularity, and as such each function can be optimized separately with little interference with other already optimized functions. Thus modularity not only structures the genotype-phenotype map for mutational stability, but also increases evolvability.

Natural modularity makes a significant class of difficult problems tractable by rearranging the structure of the search space [3]. This has previously been accomplished in NE through gene-duplication [7,8,9], and gene reuse[10,11]. In a gene-duplication scheme, two copies of the gene are made and the resulting two functions can specialize independently. In contrast, modularity can also arise from gene *reuse*, which has not yet been fully explored with regards to NE. In a gene reuse scheme, genes that perform one function are reused wherever that function is needed, and are functionally separate from other genes. Gene reuse can be thought of as a process for generalizing gene function, whereas gene duplication can be thought of as a process for specializing it.

In this paper, automatic modular decomposition is implemented in a cooperative coevolution system with a population of reusable genes (i.e. modules) and a population of blueprints specifying combinations of modules. In such a system,

- Evolution and modularization take place at the same time as modules are discovered.
- The space of a complex fitness function is restructured [12].

– Genes are reused explicitly without the need for a more complex genotype-phenotype map (such as a system with development).

Thus an NE system that takes advantage of modularization can be more efficient than one that does not. The next section presents Modular NEAT, an implementation for modular decomposition with coevolution.

3 Modular NEAT Method

A method that allows for concurrent evolution and modularization of network topologies is developed and tested in this paper. The method is based on NEAT [1], which provides a simple but efficient way to automatically discover the correct level of complexification.

3.1 Standard NEAT Implementation

The standard NEAT implementation has been shown to be a highly effective NE method in several domains [13]. It addresses three problems commonly found in TWEANN systems: 1) how to crossover topologically disparate chromosomes, 2) how to protect new topological innovation, and 3) how to keep topologies as simple as possible throughout evolution [1]. This is accomplished through historical markings, speciation, and incremental complexification.

First, each genome in NEAT includes a list of *connection genes*, each of which refers to two *node genes* being connected. In order to perform crossover, the system must be able to tell which genes match up between *any* two individuals in the population. For this reason, NEAT keeps track of the historical origin of every gene. Two genes that have the same historical origin represent the same structure (although possibly with different weights), since they were both derived from the same ancestral gene from some point in the past. Tracking the historical origins requires very little computation. Whenever a new gene appears (through structural mutation), an *innovation number* is incremented and assigned to that gene. The innovation numbers thus represent a chronology of every gene in the system, and allows crossover of diverse networks without extensive topological analysis. With historical markings the problem of having to match different topologies [14] is avoided.

Second, NEAT networks are speciated so that individuals compete primarily within their own niche. This way, topological innovations are given time to optimize their structure before they have to compete with the entire population. Also, networks share the fitness of their species [15], to prevent one species from taking over the entire population.

Third, unlike other TWEANN methods [7,16], NEAT networks are built from a minimal configuration and complexified incrementally to ensure that solutions of minimal complexity are searched first. This procedure has two advantages: first, it minimizes topology bloat, and second, it improves the efficiency of evolution by complexifying the search space only as needed. On the other hand,

minimal starting is problematic in domains with high input/output dimensionality because "minimal" starting actually represents a huge space of weights. The Modular NEAT method allows for input and output connections to be added during evolution, allowing for even simpler initial topologies.

For more details about NEAT, see Stanley and Miikkulainen [1].

3.2 Modular NEAT Overview

To allow for concurrent evolution and modularization, Modular NEAT reuses arbitrarily complex neural substructures, from a single connection to large subnetworks, to form complete neural networks. This method is similar in vein to the gene-duplication scheme of NE proposed by Calabretta et. al. [9,17], however in Modular NEAT, genes are reused in different spatial locations in the network instead of duplicated. This encourages generalization of function, instead of specialization.

Modular NEAT encodes each reusable substructure (module) as a functionally independent NEAT network from a set of input neurons to a set of output neurons. Since the building blocks are complete neural networks, they can be evolved using standard NEAT.

In addition to the population of NEAT modules, Modular NEAT makes use of blueprints that specify combinations of those modules. Both populations are evolved together symbiotically, as in the SANE method [18]. Good modules must evolve to work well with other modules, and across different spatial locations in the network.

The main difference is that in SANE each neuron can only be used in one place in the resulting network. If a neuron were reused, it would perform the exact same function as before, effectively just reinforcing that particular function. In Modular NEAT, modules can be bound to *different subsets* of input and output neurons without overlap, or with little overlap, so reuse of modules actually has a constructive effect. For example, if the optimal solution network has bilateral symmetry (i.e. the left and right halves are the same network), one module half the size of the target network can be bound on both halves to solve the problem.

3.3 Module Population

Modules are composed of a set of links between input, output and hidden neurons that are specific to the module. These *abstract* neurons are later assigned to real neurons in the solution network, through a process called binding. Each link has a globally unique identifier (for historical marking), a weight, and a start and end neuron. The module population is divided into species based on how similar the individual network topologies are (figure 1) [1]. Topological similarity is determined according to the equation

$$d = k_1 w + k_2 t,$$

where w is the sum of the absolute value of the weight differences for all connections, and t is the number of differing topological innovations (i.e. the number of

links not shared by both individuals). The constants k_1 and k_2 can be changed in order to balance the net effects of the two difference measures.

The module population is speciated for two reasons in addition to those proposed by NEAT: first, it helps maintain specializations in module function, and second, it provides a simple way to determine what other modules could serve as replacements when one is needed in a blueprint (i.e. when another member of the species dies and the blueprints need to be updated).

Module crossover is implemented similarly to crossover in NEAT. The matching historical markings are lined up and crossed over randomly (genes that do not line up are inherited from the most fit parent). Crossover using historical markings ensures that only complete mutations are inherited (i.e. no dangling neurons or unconnected edges will be present in the children). The majority of module crossovers take place with modules of the same species, however it is possible for higher fitness modules of some species to be paired with other modules outside the species.

There are three types of module mutations, two of which increase the dimensionality of the network:

– Perturb an individual weight (Gaussian weight mutation).
– Split a link (adding an additional input, output, or hidden neuron).
– Add a new link between two existing neurons.

Modules begin as a single connection between an abstract input and output neuron, so before new links can be added, the module must mutate to include new input or output neurons. Note that Modular NEAT diverges from NEAT in that new input and output neurons can be added to a module, up to the maximum specified by the solution network topology. This allows modules to "grow" new inputs or outputs as needed.

3.4 Blueprint Population

Blueprints are composed of a fixed-size list of modules paired with mappings from the abstract input/output neurons to real input/output neurons from the final neural network. These mappings are referred to as bindings. For example, module A may specify a link between neuron 0 and 1, however its binding in a certain blueprint may be 0:5, 1:6, 2:7, etc. Thus all references to abstract neuron 0 in the module under this binding actually refer to neuron 5 (in the actual network), etc. These module/binding pairings also have historical markings, in order to track the spread of mutations for data collection (figure 1).

Blueprints are recombined using standard one-point crossover. Cuts in the genetic material can only be made between modules, so blueprint crossover does not contribute to module mutation. The child blueprints inherit the order of the modules and their bindings from the parents.

There are three types of blueprint mutation: toggling a gene (module+binding) expression on or off, changing an entire module and changing part of the bindings on a module. When mutating one module to another, as much of

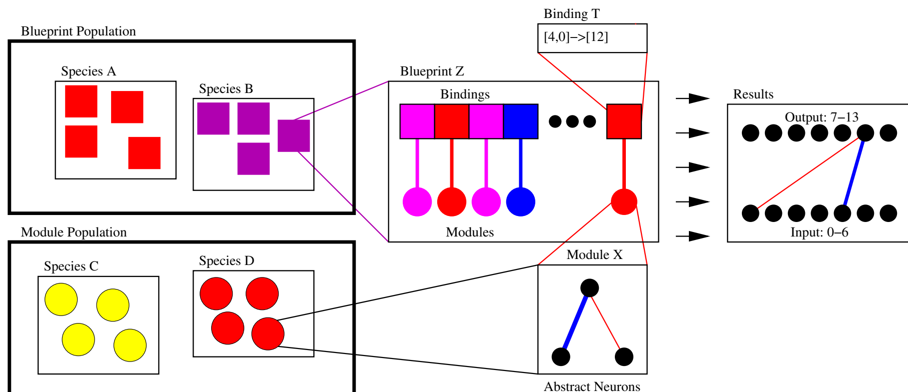

Fig. 1. Graphical indication of how a module is bound into the solution network. An example blueprint from the blueprint population is shown with its binding and module pairs list. The result of applying a binding to a single module is demonstrated in the resulting network.

the original binding is left intact as possible. If the new module has fewer inputs or outputs, the current binding is cut to fit, likewise if the new module is too large, then new neurons are added randomly to the current binding.

3.5 Evolving Modules and Blueprints

The modular NEAT algorithm consists of the module and blueprint populations undergoing cooperative coevolution. This is implemented in two sequential phases, a module phase and a blueprint phase. Both phases include separate selection, crossover, and mutation steps, after which fitness calculation takes place and the weakest members of the two populations are replaced. This process consists of the following steps:

- Initialize Modules – create an initial population of modules with a static size. These modules begin with minimum complexity.
- Initialize Blueprints – create an initial population of blueprints from the population of modules. Each blueprint has a static size n, i.e. n modules are randomly bound to it. For each blueprint, calculate an initial fitness.
- Until a certain terminating condition:
  - Select modules and perform crossover (binary tournament selection, NEAT multi-point crossover [1]).
  - Select blueprints and perform crossover (tournament selection and one-point crossover).
  - Mutate modules by randomly perturbing weights, adding new neurons, and adding new connections. Modules that change input/output size must be rebound at the blueprint level.
  - Mutate blueprints by changing bindings, and change bound modules.

- Calculate fitness for modules and blueprints.
- Replace modules.
- Replace blueprints.

– Repeat.

During the fitness evaluation step, the fitness in the population is calculated, and the scores are propagated to each of the modules. To help keep modules diverse, the fitness scores a module receives from each blueprint are summed and divided by the number of times that module is used (expressed) in the blueprint population, making sure that modules that are expressed less often have a fair chance to compete against well-established modules.

Also, the blueprint population is allowed several generations to optimize after a change is made in the module population. Kvasnička et. al. presented a similar method in which the genotype and genotype-phenotype map are optimized on different time scales [19]. Applied to Modular NEAT, Kvasnička's procedure allows for more data to be gathered on module fitness resulting in better module evaluations, and also allows the blueprints to better incorporate old modules, before new ones are created. Experimentally, a coevolutionary ratio of 5 blueprint generations to a single module generation is effective.

In the next section, a scalable board game domain is proposed that highlights the types of domains under which Modular NEAT is effective.

4 Experimental Setup

Modular NEAT was tested in an aritificial board game domain where the size of the board can be varied, i.e. the input/output dimensionality. The game itself is relatively simple, so as to focus on modular reuse.

The board is a square with $n \times n$ positions initially with $n/2$ black stones placed about randomly on the left half of the board, and $n/2$ white stones on the right half. The neural network then proceeds to play n stones the board. Stones played on the left half appear white, and stones played on the right half appear black. Play occurs iteratively, with each subsequent board state fed as a three-state (-1, 0, and 1) vector to the neural network, and location of the highest output value taken as the network's next move. To score a game, the neural network is given a point for each of the stones that it places next to one of the initial stones (i.e. "checks" it). Scoring occurs after a fixed number of moves, 5 stones for the 5×5 case, and 10 stones for the 7×7 case.

The domain requires the neural network to learn to differentiate between empty space and the types of stones on the board, and also whether or not it has already played next to a certain stone (otherwise it may end up wasting stones trying to check the same piece more than once). By reversing the color for the two halves of the board (and thus negating the input), modules that are usable on one side of the board are rendered useless on the other half. Having two functionally different regions encourages a modularization that minimizes interactions between the two halves.

A 10×10 board requires a neural network with 100 inputs and 100 outputs, but is still simple enough to be solved without a correspondingly complex topology. Thus this domain tests Modular NEAT's ability to deal with large input/output spaces, complimenting NEAT's ability to deal with topologically complex solutions.

For the experiments described in this paper, the following genetic algorithm parameters were used for the module population: 100 modules, 10% outside species mating, 1% add link, .8% add hidden node, 3% add abstract input/output, and 50% Gaussian weight mutation. For the blueprint population the parameters: 500 blueprints, 10% outside species mating, 1% gene expression mutation, 5% change module mutation, and 10% perturb binding mutation were used. Also, each network was evaluated over 50 random board configurations, and each trial was run for 3000 generations, to ensure that the highest fitness possible is reached. After 3000 generations, little useful evolution takes place.

5 Results

Several experiments were conducted with Modular NEAT. First, it was compared to NEAT in terms of solution quality and evolutionary efficiency, second, the degree to which modular phenotypes emerged was measured, and third, how useful the evolved modularizations were was tested.

5.1 Comparison to NEAT

Modular NEAT was compared to the standard NEAT implementation of [1]. With 500-member populations over 3000 generations on a 5×5 instance of the game, standard NEAT was able to evolve solutions that played correctly on average 96.66% of the time over 30 trials, and modular NEAT 98.9% (difference is statistically significant, $p < 10^{-23}$). Also, the efficiency of the two algorithms in evolving a 96% accurate solution was measured. It took NEAT on average 1753.87 generations, while Modular NEAT took only 420.73 generations (statistically significant, $p < 10^{-23}$), an effective four-fold increase in efficiency (table 1).

Table 1. Final solution quality in 3000 generations, and number of generations required on average to reach the goal fitness for both NEAT and Modular NEAT. Modular NEAT not only evolves solutions of higher quality, but also evolves them several times faster. All differences are statistically significant ($p < 0.05$), averaged over 30 trials.

	Standard NEAT		Modular NEAT	
	Quality	Efficiency	Quality	Efficiency
5x5	96.66	1753.87	98.9	420.73
7x7	87.5	1981.33	92.5	327.2

In the 7×7 case, the solution quality was on average 87.5% for NEAT and 92.5% for Modular NEAT. In evolving to a fitness goal of 86.6%, NEAT took on average 1981.33 generations, while Modular NEAT took only 327.2 generations, an effective six-fold increase in efficiency (table 1). All differences in means were statistically significant ($p < 0.05$).

These results indicate that Modular NEAT is able to obtain better solution networks than standard NEAT, and find good networks faster. Let us examine how modules contributed to this result next.

5.2 Modularization

Modular NEAT is able to identify and separate the two functions by assigning different module species to either one half or the other. The degree to which modular phenotypes evolved can be measured by counting how often a module takes an input from one half and maps it to an output of the second half, or vice versa. This kind of crosslink does not improve fitness, and should be suppressed if possible.

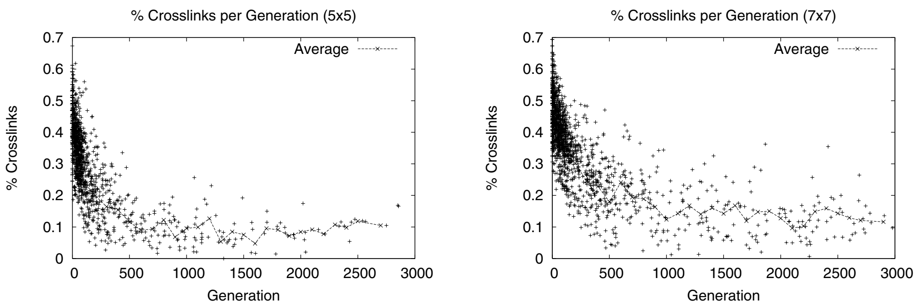

Fig. 2. Crosslink percentage per generation in the 5×5 and 7×7 board game domains. Unlike NEAT, Modular NEAT reduces crosslinks over time, focusing evolution on useful connections.

The expected percentage of crosslinks out of the total links, if no modularization occurs, is 50%. In NEAT, since no modularization occurs, the percentage of crosslinks in the champion genomes indeed stays around 50% during evolution. In Modular NEAT, however, this percentage decreases over time. In the 5×5 case, the percentage of crosslinks decreases to below 10% on average, and in the 7×7 case it decreases to slightly above 10% (figure 2).

This result indicates that the substructures can be arranged in a way that separates them by function, supporting the hypothesis that reusing substructures leads to a modular phenotype. By reusing certain modules on only one half or the other, evolution of both sides can occur simultaneously without interference.

5.3 Module Use

Because modules are reused in many different combinations across many different spatial locations, typical module evaluation is often not consistent, and hence the modules themselves remain relatively generic in function. Two tests are conducted to measure the usefulness of the evolved modules.

Topological Decomposition. How well does Modular NEAT decompose the problem space? One test is to freeze the module bindings and module topologies and randomize the module weights. Re-evolving only the weights from this base gives a good idea as to how well the module topologies and bindings decompose the problem, when compared to a randomly evolved modularization. A random modularization can be created by running the algorithm for a fixed period of time, evaluating blueprints normally and then assigning all modules the same fitness. With this method, the average module complexity can be kept about the same in both the evolved population and the random population.

The evolved population solved the problem on average 1.37 times faster than the random population with 160 generations of setup time, 1.57 times faster with 320 generations of setup time, and 1.65 times faster with 480 generations of setup time (table 2). In all three cases the differences in means were statistically significant ($p < 0.05$). Thus the decomposition encoded in the evolved module topologies and the module bindings is useful even if all the module weights are randomized.

Table 2. Averages of efficiencies (out of 60 trials) for the test and random populations. Setup Generation refers to the number of generations used in setting up the modularization before the module weights are randomized. Speedup is the effective gain in efficiency from the evolved module topologies and bindings combined. The evolved population clearly outperforms the random one.

Setup Generations	160	320	480
Evolved	291.933	333.9	237.898
Random	400.111	524.792	391.8
Speedup	1.371	1.572	1.647

Module Population Decomposition. The results in the previous section indicate that bindings and module topologies together enhance evolution. However, it does not answer how useful the module population itself is, without the set of bindings. The set of modules would have to be generic enough to guide the blueprint evolution quickly to the correct decomposition, and still be specialized enough to represent a better solution to the problem in this configuration, in order to be useful.

One test is to perform evolution for a fixed number of generations, then destroy the blueprint population, and re-evolve from a new random blueprint

population that uses the same set of modules. The re-evolved population should be able to reach the same fitness level than the initial population, but faster.

Two experiments were conducted, one with evolution taking place for 300 generations, and one for 500 generations (table 3). On the 300 generation version, the best solution took on average 255.977 generations to reach, and the re-evolved solution took 236.167 generations, representing an effective speedup of 1.08 times. These means differ by about twenty generations, however the difference is not statistically significant ($p = 0.152$). With 500 generations, the means differed by approximately seventy generations, representing a speedup of 1.19 times ($p < 0.003$).

Table 3. Modules and blueprints are evolved for a fixed number of generations and then the bindings are randomized and evolution proceeds until the highest fitness from the previous run is reached again. The number of generations on average to reach the highest fitness both initially, and after randomization is recorded.

Setup Generations	300	500
Before	255.967	451.433
After	236.167	380.3
Speedup	1.084	1.187

These results indicate that evolved modules are more useful than random ones. However they are not useful enough to warrant eradicating the set of bindings: separating modules from their expression does not significantly speed up evolution.

Taken together, the conclusions in these two experiments suggest that binding information could be specified on the modules themselves, instead of on separate chromosomes. Such an encoding is a promising direction for future research, as will be discussed in the next section.

6 Discussion and Future Work

Modular NEAT is powerful because it can restructure the search space and yield automatic decompositions of the problem without solving parallel sub-tasks more than once. That is, a module that performs a certain function can be reused wherever that function is needed. Although searching through combinations of highly complex modules may prevent the algorithm from finding the optimal solution, the chances are high that it finds a *good* solution. In domains currently intractable due to their dimensionality, finding such a decomposition may be an essential first step in actually finding a solution.

As the experiments in section 5.3 indicate, separating generic modules from their bindings yields a module population only slightly more useful than the initial random one. Even with highly generic modules, modular decompositions

are only useful when the modules and their expression information are coupled. This result makes intuitive sense. If the information is separated, generic modules will be selected more often since modules do not have genetic control over their expressions. What is needed is a system that combines both module topology and expression information into the same chromosome.

In the current version of Modular NEAT, explicit module bindings increase the size of the search space proportional to the number of reuses. However, as long as the overhead of module binding can be kept low and the number of reuses kept high, this overhead is negligible. Compared to systems with developmental elements (i.e. genotype-phenotype maps that are not one-to-one) [7,20,21], explicit module bindings are not efficient. Natural and artificial developmental systems rely on relative targeting [2], which could be used in Modular NEAT to reduce the binding overhead for each module (e.g. modules could be tessellated regularly across the binding space).

Modular NEAT is most suited to domains where there are spatial regularities, such as Go and Othello. Like the test domain, they have rotational symmetry. Another large, self-similar domain is low-level vision processing such as filtering. Although it may be orientation specific, the input is still scalable. Since modules can add new input connections, the controllers can start out using a small subset of the inputs and add more if needed.

7 Conclusions

Modular NEAT has been shown to 1) make evolution significantly more efficient than standard NEAT in a scalable board game domain, and 2) implicitly generate modular solutions. Since there is no explicit selection for modularity built into Modular NEAT, phenotypic modularity must arise from the reuse of fundamental substructures. In nature, this is similar to the modularity that arises from reusing genes. By simultaneous modularization and evolution through cooperative coevolution, Modular NEAT is a principled approach to a significant class of difficult problems, i.e. those with high input/output dimensionality and self-similarity.

References

1. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10** (2002) 99–127
2. Stanley, K.O., Miikkulainen, R.: A taxonomy for artificial embryogeny. Artificial Life **9** (2003) 93–130
3. Hart, W.E., Belew, R.K.: The role of development in genetic algorithms. Technical Report CS94-394, University of California, San Diego, CA (1994)
4. Bongard, J.C.: Evolving modular genetic regulatory networks. In: Proc. of CEC 2002, IEEE Press (2002) 1872–1877
5. Wagner, G., Altenberg, L.: Complex adaptations and the evolution of evolvability. Evolution **50** (1996) 967–976

6. Futuyma, D.J.: Evolutionary biology. Sinauer, (Sunderland, MA)
7. Gruau, F., Whitley, D., Pyeatt, L.: A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) 81–89
8. Nolfi, S.: Using emergent modularity to develop control systems for mobile robots. Adaptive Behavior **5** (1997) 343–363
9. Calabretta, R., Nolfi, S., Parisi, D., Wagner, G.P.: A case study of the evolution of modularity: Towards a bridge between evolutionary biology, artificial life, neuro- and cognitive science. In Adami, C., Belew, R., Kitano, H., Taylor, C., eds.: Proceedings of Artificial Life VI. MIT Press (1998)
10. Watson, R.A., Pollack, J.B.: Symbiotic composition and evolvability. In Kelemen, J., Sosik, P., eds.: Proc. of Advances in Artificial Life, 6th European Conference, (ECAL 2001), Springer (2001) 480–490
11. de Jong, E.D.: Representation development from pareto-coevolution. In: Proc. of GECCO 2003, Morgan-Kaufman (2003)
12. Bentley, P.J., Kumar, S.: Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In: Proc. of GECCO'99. (1999) 35–43
13. Stanley, K.O., Miikkulainen, R.: Competitive coevolution through evolutionary complexification. Journal of Artificial Intelligence Research **21** (2004)
14. Radcliffe, N.J.: Genetic set recombination and its application to neural network topology optimization. Neural computing and applications **1** (1993) 67–90
15. Goldberg, D.E., Richardson, J.: (Genetic algorithms with sharing for multimodal function optimization) 148–154
16. Yao, X.: Evolving artificial neural networks. Proc. IEEE **87** (1999) 1423–1447
17. Calabretta, R., Nolfi, S., Parisi, D., Wagner, G.P.: Duplication of modules facilitates the evolution of functional specialization. Artificial Life **6** (2000) 69–84
18. Moriarty, D.E.: Symbiotic Evolution of Neural Networks in Sequential Decision Tasks. PhD thesis, The University of Texas at Austin (1997)
19. Kvasnička, V., Pospíchal, J.: Emergence of modularity in genotype-phenotype mappings. Artificial Life **8** (2002)
20. Gruau, F.: Automatic definition of modular neural networks. Adaptive Behavior **3** (1994) 151–184
21. Belew, R.K., Kammeyer, T.E.: Evolving aesthetic sorting networks using developmental grammars. In Forrest, S., ed.: Proc. of the 5th International Conference on Genetic Algorithms, San Francisco, CA, Morgan Kaufman (1993)