Coevolving High-Level Representations

Peter J. Angeline and Jordan B. Pollack

Laboratory for Artificial Intelligence Research
Computer and Information Science Department
The Ohio State University
Columbus, Ohio 43210
pja@cis.ohio-state.edu
pollack@cis.ohio-state.edu

To Appear in:

Artificial Life III

Coevolving High-Level Representations

Peter J. Angeline and Jordan B. Pollack
Laboratory for Artificial Intelligence Research
Computer and Information Science Department
The Ohio State University
Columbus, Ohio 43210
pja@cis.ohio-state.edu
pollack@cis.ohio-state.edu

Abstract

Several evolutionary simulations allow for a dynamic resizing of the genotype. This is an important alternative to constraining the genotype's maximum size and complexity. In this paper, we add an additional dynamic to simulated evolution with the description of a genetic algorithm that coevolves its representation language with the genotypes. We introduce two mutation operators that permit the acquisition of modules from the genotypes during evolution. These modules form an increasingly high-level representation language specific to the developmental environment. Experimental results illustrating interesting properties of the acquired modules and the evolved languages are provided.

1.0 Introduction

A central theme of artificial life is to construct artifacts that approach the complexity of biological systems. To accomplish this, the benefits and limitations of current methods and tools must be considered. The representation employed in a genetic algorithm, for example, must permit an appropriate level of expression in order for the genotypes to evolve.[22,18] Artificial life researchers favor genetic algorithms both for their proven ability in wide range of environments and for their obvious association with natural evolution. However, in most genetic algorithms, the representation's length is fixed a priori, placing an ad hoc restriction on the developing genotypes.

Other genetic algorithms emphasize a dynamic approach to developing genotypes. Their representations expand and contract according to environmental requirement providing a flexible alternative when simulating evolution. In this paper, such systems are described as *dynamic genetic algorithms*. While these representations allow the size of the genotypes to vary with the environment, their expressiveness remains static.

In this paper, a dynamic genetic algorithm is described that not only alters a genotype's size but also coevolves a high-level representational language tailored to the environment. This is accomplished with the addition of two specialized genetic operators that create stable modules

from portions of the developing genotypes. Each created module serves as an abstract addition to the representational language reflecting some specifics of the environment. We demonstrate this technique and illustrate some of the properties of the evolved modules. The following section provides a review of genetic algorithms theory and related genetic algorithm implementations.

2.0 Simple and Dynamic Genetic Algorithms

A genetic algorithm[17,14] is a search method analogous to natural selection that is surprisingly adept in ill-formed environments. Genetic algorithms evaluate a population of genotypes with respect to a particular environment. The environment includes a *fitness function* that rates the genotype's viability. Genotypes reproduce proportionally to their relative fitness using a variety of genetic operators. One operator, termed *crossover*, uses the recombination of two parents to construct novel genotypes. The *mutation* operator creates new genotypes from a single parent with a probabilistic alteration.

To analyze how genetic algorithms process genotypes, Holland defines a class of genotypes that share specific attributes to be a *schema*.[17] A schema's *defining length* is the number of places where a crossover operator when applied to any member of the schema creates a genotype that is no longer in the schema. The *order* of a schema is similarly the number of positions where the application of a mutation operator removes a member from the schema. Holland's *schema theorem*[17] shows that a schema with a consistently above average fitness, short defining length and low order propagates through the population exponentially fast under ideal conditions. The *building block hypothesis*[14] suggests that the power of genetic algorithms originates in the recombination of good schemata into better schemata. For a complete description of the schema theorem and the building block hypothesis see Goldberg.[14]

2.1 Simple Genetic Algorithms

Simple genetic algorithms use a fixed-length string representation. Two tacit assumptions that accompany simple genetic algorithms limit their evolutionary abilities. First, the representation's fixed length places a strict upper-limit on the number of distinct genotypes and hence the number of phenotypes. A second, more subtle limitation occurs when converting a genotype into its associated phenotype. Typically, the individual positions of the genotype's string are interpreted as distinct features to be included in the phenotype. This *positional encoding* of the genotype reduces its phenotype to a simple concatenation of features, reducing the potential complexity available to simulated evolution.

The limitations of simple genetic algorithms can be overcome by a variety of techniques. Jefferson et al.[18] allows limited variability by defining an extremely long binary string as the genotype that is converted into either a finite state automata (FSA) or a neural network as the phenotype. The content of the fixed-length genotype dictates the actual size of the phenotype. While this interesting idea holds promise, it still dictates a maximum size for both the genotype and the phenotype. Others [5,7] have opted to increase the complexity of the conversion process to construct more interesting phenotypes. But, this approach begs the question of *evolving* a

complex artifact since sufficient representational complexity is provided a priori in the conversion function.

2.2 Dynamic Genetic Algorithms

In dynamic genetic algorithms, the representation of the genotype is a variable size structure, typically a tree or graph. The requirements of the environment dictate the final size and shape of the genotypes. Various dynamic structures have been used to represent genotypes in these genetic algorithms including the linear "assembly code" programs of Tierra,[24] the hierarchical LISP expression trees of the genetic programming paradigm [21,20] and the FSA representations of early evolutionary programming.[12,11,10] In each of these systems, the genotype is expressed as an executable structure with a language of primitives suited for the environment. The phenotype is then the genotype's simulated *behavior.* Typically, the genotype's behavior is of interest in these systems instead of the specification of the behavior altered by the genetic operators. For clarity, we refer to all such executable structures as "programs."

Given that a dynamic genetic algorithm relies on the evaluation of a program for the expression of the phenotype, the design of the primitive language is crucial. First, the syntax of the language should be chosen to minimize the number of programs that are syntactically invalid.[18,24] A good primitive language reduces the probability of constructing non-viable offspring during evolution. Second, the relationship between the primitive language's semantics and the environment determine's how difficult a desired phenotype is to evolve.[18,19] For instance, given a high level language where the individual primitives are tailored to the environment, small programs exhibit sufficient behaviors. With a lower level language, the minimal size of a program that exhibits an equivalent behavior is proportionally larger and thus much more difficult to evolve. Jefferson et al. provides some excellent additional guidelines to consider when evolving dynamic representations.[18]

While the designable aspect to the primitive languages in dynamic genetic algorithms is convenient when constructing applications, it leaves dynamic genetic algorithms susceptible to the same over-design problems as in simple genetic algorithms. A researcher can easily avoid the problem of extracting the complexity from an environment entirely by providing a suitably designed representational language.

3.0 The Genetic Library Builder (GLiB)

The Genetic Library Builder (GLiB) is a dynamic genetic algorithm modeled after Koza's genetic programming paradigm (GPP).[20,21] Primitive languages in both GPP and GLiB employ a LISP expression tree syntax to represent genotypes in the population. Both also use a crossover operator that exchanges randomly selected subtrees between parents to create the novel genotypes. The principal difference between GPP and GLiB is the addition of two novel mutation operators. First, the *compression* operator extracts environment specific additions to the primitive language from the genetic material of the population. Each compressed portion of a genotype, called a *module*, increases the expressiveness of the language and decreases the average size of the genotypes in the population. The usage of the module by subsequent generations provides a

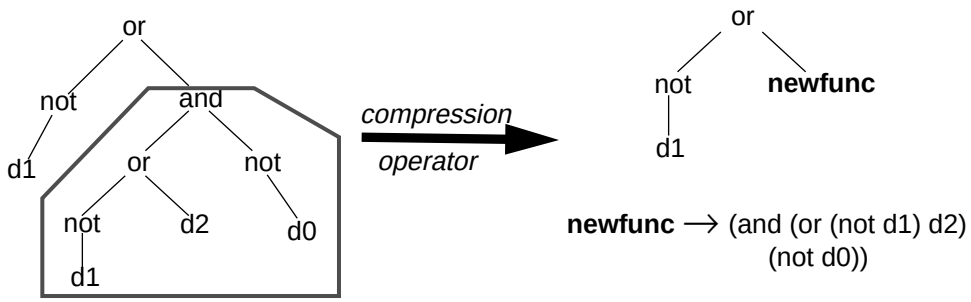

Figure 1: *Creation of a compressed module from a randomly selected subtree of a genotype.* ***Newfunc****'s definition is added to the library.*

measure of a module's worth. A second operator, called *expansion*, replaces a compressed module by its original definition. As with all mutation operators, compression and expansion alter the structure of the genotype. Unlike other mutation operators, they never directly or indirectly alter the phenotype.

3.1 Creation of Environment Specific Modules

A compression operation begins with the selection of a random position in the genotype. This position identifies the root of the subtree to be compressed. When the module is defined, the subtree is extracted from the genotype and replaced with a unique symbolic name. GLiB binds the extracted subtree to the module's symbolic name by entering the pair into the "genetic library." This library is the collection of all created modules and serves only as a symbol reference for GLiB. During either a genotype's execution or the expansion of a compressed module, GLiB retrieves the definitions of symbols from the genetic library.

The *depth compression* method of module creation extracts the subtree beginning at the randomly selected position of the program and clips off any branch exceeding a maximum depth. The maximum depth for each new module is a random number selected uniformly from a user defined range. Figure 1 shows the result of a compression when each branch of the subtree is within the maximum depth. Here, compression removes the entire subtree from the genotype, assigns it a unique module name**,** and defines it as a new LISP function with no parameters. A call to the newly defined module replaces the original subtree as shown in Figure 1. When GLiB evaluates the genotype's fitness, it retrieves the compressed module's definition instead of directly executing the original subtree.

When one or more subtree branches exceed the maximum depth, the defined module has a very different definition. Rather than extracting the entire subtree, only the portions within the selected depth appear in the new module's definition. The branches exceeding the depth are replaced with unique variable names. GLiB then defines the new module with the variables defined as parameters to the module. When GLiB replaces the subtree in the genotype with the call to the module, the sections of the original subtree below the selected depth become the parameter bindings. Figure 2 shows this instance of module creation.

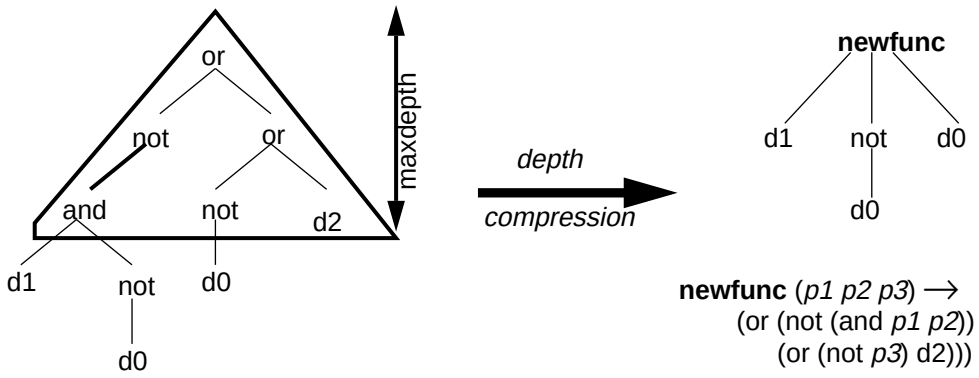

Figure 2: *Depth compression applied to a subtree larger than the maximum depth. The module's definition has parameters as a result.*

Depth compression is the simplest method GLiB currently employs and is the form of compression used in the experiments described in the next section. Other compression methods based on more involved computations are also under investigation. Future research will explore their associated benefits and weaknesses.

GLiB's compression operator introduces a new dynamic into the evolutionary process: the evolution of the representational language. Each compressed module extends the representational language of the genotypes, like a library of functions extends a particular programming language. Since the new modules will be constructed from the primitives and modules already in the language, they will tend to be at a higher level of abstraction. The genotypes and the representational language coevolve until the language is sufficiently expressive to succinctly encode appropriate behaviors.

3.2 Expansion of Compressed Modules

One drawback to the compression operator is that each compression removes genetic material that otherwise would be available to construct novel genotypes in subsequent generations. After only a few generations of moderately applying the compression operator, the available genetic material in the population is insufficient to significantly improve the performance of the population. The loss of genetic diversity in a population leading to premature convergence to a suboptimal solution is a well known problem in genetic algorithms research.[9,14,8] Typical solutions include altering the scaling of the fitness values or altering the frequency of crossovers and mutations.[14] Neither of these solutions are appropriate in this situation.

To offset the loss of diversity in the population due to the application of the compression operator, GLiB provides an additional mutation operator. The *expansion* mutation randomly selects a compressed module from the genotype and expands it back into the original subtree, reintroducing the previously removed genetic material. The expansion process is exactly the reverse of compression. A copy of the module's definition from the genetic library is expanded by replacing all occurrences of parameters with the bindings specified in the genotype's call to the module. GLiB then splices the expanded subtree back into the genotype. One could literally reverse the directions of the arrows in Figures 1 and 2 as illustrations.

The complementary nature of the compression and expansion operators provides an iterative refinement mechanism for the evolving modules. The random selection of a subtree for compression provides no guarantee that a compressed module is beneficial to the developing genotypes in its initial form. It could be that the acquired module contains only a portion of a useful subtree, a useful subtree along with some extraneous additions, or simply nothing of import. The original subtree's replacement provides the chance to capture a better version of the module in a later compression.

3.3 Module Evaluation by the Environment

Randomly compressing subtrees from the genotypes does not guarantee useful additions to the representation language that benefit the evolutionary process. It is not enough to define modules without an appropriate mechanism to judge and discard them if necessary. One appropriate criterion for an evolved module is to examine the number of calls made to it in the subsequent generations.[4] If the population uses a module frequently then it must have provided some consistent beneficial behavior in previous generations. When a module provides such a consistent advantage we say it is *evolutionarily viable*, meaning, literally, the module can survive the evolutionary process. Given the criterion of evolutionary viability as an appropriate evaluation metric, GliB must copy beneficial modules into the offspring of the next generation while inhibiting the proliferation of inconsequential modules. Goldberg's "enlightening" guidelines for genetic algorithms design cautions that one should never be too clever when designing genetic algorithms as a "frontal assault" usually defeats the inherent non-linear interactions.[15]

Appropriately enough, the evolutionary process which propagates and inhibits the reproduction of the genotypes evaluates the worth of every compressed module without any additional intervention. The primary effect of the compression operator is to protect potentially useful schema within the modules. Because the compressed modules are examples of schemata, the genetic algorithm evaluates them automatically. Consider that modules appear in only a single genotype when created. If this genotype has an above average phenotype, then it will be used to create several offspring in the next generation. As stated above, the schema theorem suggests that a schema appearing in consistently above average genotypes with a short defining length and low order will proliferate exponentially through the population under ideal conditions. By definition, a compressed module has a defining length of zero and is of order one, making it a perfect candidate for quick proliferation. After several reproductive cycles, a significant portion of the population will rely on a beneficial module and its number of calls per generation will be high. Likewise, if a module consistently appears in below average genotypes it is removed from the population, resulting in a proportional decrease in its number of calls. When no member of the population contains a call to a particular module, it is no longer in the in the representational language.

4.0 The Emergence of Environment Specific Modules

In this section, we present experiments illustrating some of the properties of GLiB's evolved programs and their associated modules. In the first set of experiments, designed to provide initial insights into the nature of an evolved high-level representation, GLiB created programs to solve the familiar Tower of Hanoi problem. These results permit analysis of the properties of evolved

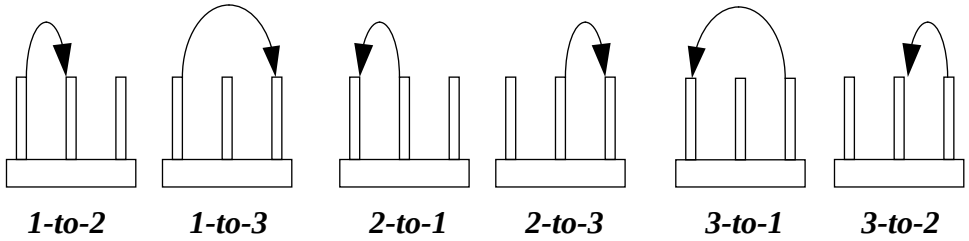

Figure 3: *Primitive language for Tower of Hanoi experiments. Each command moves a single disk between the specified pegs.*

modularity. The second experiment takes a more ambitious track. A general primitive language is used to evolve programs to play and beat an automated Tic-Tac-Toe opponent. The experiments show GLiB preserves important schemata in the created modules while highlighting some additional unexpected features of evolved modular programs.

4.1 Tower of Hanoi Experiments

Our initial experiments with GLiB evolved solutions for the classic Tower of Hanoi problem. This puzzle consists of three pegs, labeled by position, and several uniquely sized disks. The initial state has all disks on the first peg such that no disk is on top of a smaller disk. The goal is to reconstruct the tower of the initial configuration on the third peg by moving the disks, one at a time, such that a larger disk never rests on a smaller disk.

The primitive language for the genotypes included the six possible moves from peg to peg shown in Figure 3. Each primitive moves the top disk from one peg and places it on the other peg. A solution then is a sequence of single disk moves in an order that reconstructs the tower on the third peg. Additionally, we assumed four disks in the initial tower and permitted a maximum of 32 moves to complete the task. This does not imply a limit of the 32 steps in a program, only that the evaluation function stopped executing a program after it had executed 32 primitive operations. The fitness function gave points only for the set of disks an evolved program correctly moved to the third peg. The points awarded for a disk were proportional to the size of the disk. Illegal moves, i.e., ones that would leave a larger disk on top of a smaller disk, were not executed. As a penalty, programs lost one time step for every illegal move executed.

Several trials of this experiment were executed with 10 percent of the population undergoing compressions and expansions each generation. The population size was set to 1000 and 50 generations of decedents were evolved. In each run, GLiB performed as GPP has on similar problems.

A consistent property appearing in the modules evolved during these runs is the ability to improve upon a variety of intermediate problem configurations. As an example, the steps preserved within one evolved module are pictured in Figure 4. Note that step 4 moves no disk in the configuration shown. While this module encodes neither an optimal nor a completely legal subsequence from certain configurations, it is very versatile given the penalty for using illegal or unnecessary commands is so mild, i.e., only the loss of a time step. For instance, given the 27 possible

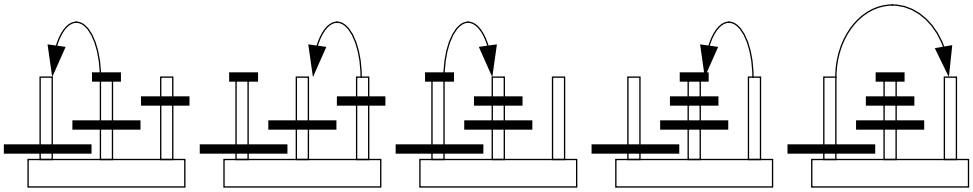

Figure 4: *Sequence of moves implemented by an evolved module. The configuration of the disks in Step 1 is the configuration at which the module is called in the evolved program.*

intermediate configurations of the problem with the largest disk still on the first peg, the evolved module shown in Figure 4 moves the largest disk to the third peg in 4 of them. This module also does well from many other configurations.

While broad applicability is clearly an advantage in an evolutionary setting, it is unclear whether or not the flexibility found in the evolved modules is a direct reflection of developmental usage or simply a feature that guarantees survival. In either case, it is evident that the modules capture some of the limitations and constraints of the environment.

4.2 Experiments with Tic Tac Toe

The second set of experiments investigate the attributes of modular programs evolved to play Tic-Tac-Toe (TTT) against a player of our own design. Figure 5 shows the primitive language for the TTT genotypes. The primitives ***pos00*** to ***pos22*** are the data points representing the nine positions on the TTT board. For the remaining primitives, the return value is either one of these positions or NIL.[1] For instance, the binary operators ***and*** and ***or*** each take two arguments. When both arguments are non-NIL, ***and*** returns the second. If either argument is NIL then it returns NIL. ***Or*** returns the first non-NIL argument and returns NIL otherwise. The ***play-at*** primitive takes a single argument. If the argument is a position and no player has placed a mark there, then the current player's mark is placed at that position and the turn is halted. Otherwise, ***play-at*** will return whatever it is passed. Finally, the operators ***mine***, ***yours*** and ***open*** take a position and return it when the mark in that position fits the test. Otherwise, they too return NIL.

pos00 .. pos22 - board positions
and - binary LISP "and"
or - binary LISP "or"
if - if <test> then <arg1> else <arg2>
open - returns <arg> if unplayed else NIL
mine - returns <arg> if player's else NIL
yours - returns <arg> if opponent's else NIL
play-at - places player's mark at <arg>

pos00	*pos01*	*pos02*
pos10	*pos11*	*pos12*
pos20	*pos21*	*pos22*

Figure 5: *Primitives used to evolve modular programs to play Tic Tac Toe.*

1. NIL represents FALSE in LISP, GLiB's current implementation language.

This language is general enough to cover any number of games playable on a TTT board. Consequently, there is no guarantee that a random program in this language will observe the rules of TTT or even place a single mark on a TTT board. If the program does not make a valid move, then its turn is forfeited, providing a significant advantage for its opponent. We consider the rules of play to be an additional component of the environment and consequently should be induced by GLiB.

How well a genotype plays a pre-programmed "expert" TTT algorithm determines its viability. The expert is constructed such that it will not lose a game unless forked by an opponent. A *fork* is any board configuration where the addition of a single mark results in more than one possible winning play on the player's next turn. The expert also adapts over a series of consecutive games to individual opponents by selecting positions the opponent frequents when no better move is available. Adaptability of this sort increases the complexity of the expert's strategy and forces the evolving programs to be more robust. Given the generality of the primitives combined with the expert's level of play, this is a difficult environment for learning TTT.

The fitness function used in this experiment averages a genotype's score over a total of four games played against the expert. Individual moves by the genotype receive varying point values. First, because there is only a small possibility of a random program making a legal play, a program receives 1 point for every legal move it makes. It also receives 1 additional point if that move blocks the expert from winning on the next turn. If the game ends in a draw or a win, the scoring function increases the accumulated score by 4 or 12 points respectively. It is important to note that the score for a genotype is a lump sum and provides no indication of which actions are being rewarded. The same results should be achieved if only the final state of the board is scored.

In this experiment, the population contained 1000 programs and the environment consisted of the described expert and scoring method. GLiB applied the compression and expansion operators to 10 percent of the population each generation. All other parameters were as set in Koza's "ant" experiment.[20]

4.3 Results and Discussion

The best evolved program after 200 generations, shown in Figure 6, had an average score of 16.5 points for the four games, a maximum depth of 13, and 15 evolved modules in its top-level. As expected, expanding the definition of these modules back into their original subtrees revealed additional modules in their definitions. In all, this evolved program used a total of 43 distinct modules in 89 calls. The virtual size of the program is 477 nodes with a maximum depth of 39. Two of the modules had a total of nine separate calls each.

Figure 7 shows the first game played between the evolved program and the expert. There is an interesting point to be made about the apparent strategy of the evolved program. Notice that it establishes a fork on its third move (Figure 7c) but did not win the game until two turns later (Figure 7e). While this seems an odd strategy, recall that the evolved program gets points for every move it makes and extra points when it blocks the expert. Its strategy is to maximize its total point score by forking the expert not once but *twice* in the same game. If the program had ended the game on its fourth move it would have received 3 fewer points. By extending the game

```
(FUNC26677
  (FUNC43031
    (PLAY-AT (PLAY-AT (MINE (MINE (MINE (FUNC42966))))))
    (IF (FUNC20328) (POS20) (POS20))
    (PLAY-AT
      (AND
        (FUNC46557
          (FUNC12959)
          (AND (POS12) (PLAY-AT (OR (POS22) (POS20))))
          (FUNC20917)
          (PLAY-AT
            (YOURS (YOURS (OR (PLAY-AT (OPENP (FUNC16153))) (POS21)))))
          (PLAY-AT
            (PLAY-AT
              (IF (FUNC20917)
                (PLAY-AT (OR (PLAY-AT (FUNC23500)) (FUNC20671)))
                (AND
                  (PLAY-AT (IF (FUNC4076) (MINE (POS01)) (POS10)))
                  (POS20))))))
          (FUNC36830))
        (AND (OR (POS12) (POS12)) (POS12))))
    (FUNC68808)
    (AND (PLAY-AT (FUNC70469 (POS00))) (POS22))))
```

Figure 6: *Best evolved modular program from described experiment. Evolved modules are named FUNC####.*

it increases its score without the possibility of losing. It remains an open research question whether environments can be designed which encourage good sportsmanship.

The analysis of the evolved modules is equally interesting. The compression operator created 16,852 modules during the experiment with only 257 in use during the final generation. Figure 8 shows the number of calls per generation for three of the evolved modules. Each of these has a distinct period where its number of calls increases rapidly. In Figure 8a, an initial increase occurs soon after the module is defined, showing it provided an immediate advantage to the population. This module's usage quickly rises again a short period later, showing its application to additional facets of the environment. Figure 8b shows a module that was extremely useful shortly after its creation but whose use fell off dramatically toward the end, possibly due to a preferred module

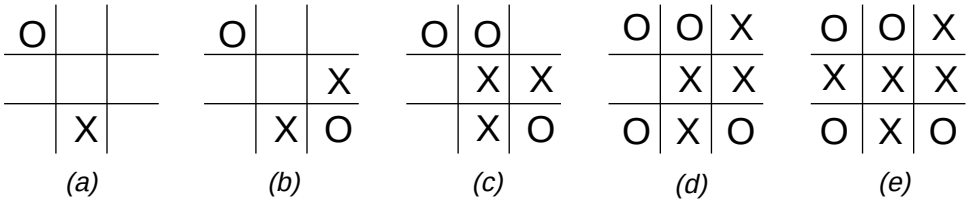

Figure 7: *A sample game from the described run. The evolved program is "X" and is playing first. It sets up the first fork in (b) and completes it in (c). The second fork is completed in (d) and leads to a win in (e). The evolved program received a score of 20 points for this game.*

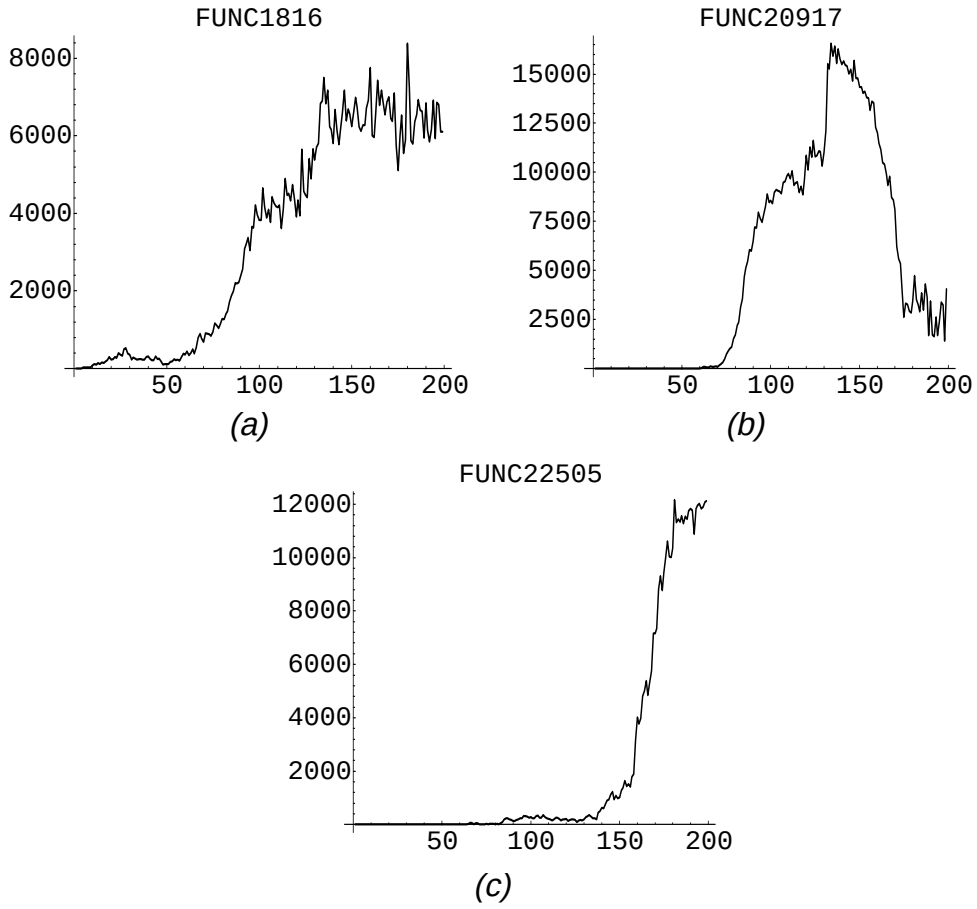

Figure 8: *Graphs showing number of calls (y-axis) per generation (x-axis) for three evolved modules. Note the scales are different in each.*

appearing in the population. Finally, Figure 8c shows a module that was present for almost 100 generations before the population recognized its usefulness, resulting in its explosive proliferation. Such rapid changes are indicative of "phase transitions" which form the basis for inductions in dynamic learning systems.[23, 3]

Discovering general properties of the modules constructed in the Tic-Tac-Toe experiment was not as simple as in the Tower of Hanoi experiments. Obviously, the broad application of the modules seen in the Tower of Hanoi experiment is apparent from the description of the evolved Tic-Tac-Toe program and the high usage of individual modules evident from the graphs of Figure 8. Unfortunately, more specific properties have been extremely difficult to identify. One observation is that the evolved modules do not employ the typical conceptual breakdown for playing Tic-Tac-Toe. For instance, we found no modules that test for either an opportunity to win on the current turn or the possibility of the opponent winning on the subsequent turn. Second, the program modularity that results is not amenable to normal analysis techniques. As expected from an evolutionary process, the usage and semantics of the evolved language is extremely non-standard. For instance, several modules encode numerous side effects in the test position of a conditional. In short, the resulting programs, while containing multiple modules, are not examples of modular

programming but share more commonality with the *distributed representations* of procedural knowledge found in connectionist networks.[16]

5.0 Conclusions

It is apparent from Figure 8 and the statistics of the runs that GLiB was successful in capturing modules advantageous to the construction of complex programs and that reflect the idiosyncracies of the developmental environment. While the number of modules created is large, the computation expended to capture them is insignificant compared to the evolutionary search process. In addition, the emergence of a useful module reduces the size of the genotypes and consequently the number of available crossover and mutation points. This focuses the evolutionary process on improving the evolving programs at a higher level of abstraction rather than destroying previously discovered building blocks. GLiB's construction of useful high-level modules as a by-product of purely local interactions identifies yet another emergent property of genetic algorithms.[13]

The Tower of Hanoi experiments illustrate that the evolved modules are often applicable to multiple environmental situations given the penalty for such flexibility is not too extreme. This flexibility was also evident in the Tic-Tac-Toe experiments. Additionally, the modularity that resulted from the more complex Tic-Tac-Toe environment was significantly more difficult to analyze. This shows our technique is free from the extremely restricted form of programming humans practice. It also suggests a trade-off between complexity of behavior and interpretability of structure when evolving complex modular programs. David Fogel identifies this same trade-off for different reasons. [11, 10]

Extracting subtrees from the programs and placing them into GLiB's genetic library is a convenience rather than a necessity for our technique. The genetic library lends nothing to the development process but is very beneficial in reducing the stored size of the developing genotypes. Exactly the same results would be achieved if the subtrees were not extracted from the genotypes but were still equally protected from alteration.

Although the techniques of compression and expansion encoded in GLiB were illustrated using only one type of dynamic genetic algorithm, they are universally applicable to any method employing a an analogy to natural selection. For instance, simple genetic algorithms could designate protected positions for each individual with parents propagating their protected regions to their offspring. Evolutionary programming, which relies on mutation as its only method of constructing novel population members, could restrict changes to unsettled or unprofitable sections of the representation in a similar manner. Previous experiments have demonstrated that a simplified form of compression produces a speed-up in the acquisition of Finite State Machines by an evolutionary program.[2]

This technique also holds promise for removing the limitations of human designers from the creation of complex structures. We discuss elsewhere how GLiB's operators aid in the construction of arbitrarily complex programs.[3] Similarly, modular designs for artificial organisms in the spirit of Brooks [6] could be evolved rather than hand constructed, possibly leading to wholly new design principles for complex artificial organisms. We are currently exploring this possibility

by extending previous work to evolve modular neural networks for the control of artificial organisms.[25,1]

We feel that an evolutionary process necessarily requires the ability to evolve environment specific abstractions in order to be open-ended or at the very least scalable. Patching up inadequate representations with complex interpretations dodges the issue, as does a representation language that assumes overly powerful primitives. GLiB's technique for coevolving environment specific representations with the population offers a third, universally applicable alternative that preserves the integrity of simulated evolution.

6.0 Acknowledgments

We owe many thanks to Greg Saunders, John Kolen, Tilmann Wendel and David Fogel for commenting on various drafts of this paper. This research is supported by the Office of Naval Research under contract #N00014-89-J1200.

7.0 References

1. Angeline, P., G. Saunders and J. Pollack. "An Evolutionary Algorithm that Constructs Recurrent Neural Networks." *IEEE Transactions on Neural Networks*, 1993 (To Appear).
2. Angeline, P. and J. Pollack. "Evolutionary Module Acquisition." In *Proceedings of the Second Annual Conference on Evolutionary Programming*, edited by D. Fogel, Morgan Kaufmann Publishers Inc., 1993 (To Appear).
3. Angeline, P. and J. Pollack. "The Evolutionary Induction of Subroutines." In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, 1992.
4. Bedau, M. and N. Packard. "Measurement of Evolutionary Activity, Teleology, and Life." In *Artificial Life II*, edited by C. Langton, C. Taylor, J. Farmer and S. Rasmussen. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.
5. Belew, R., J. McInerney and N. Schraudolph. "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning." In *Artificial Life II*, edited by C. Langton, C. Taylor, J. Farmer and S. Rasmussen. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.
6. Brooks, R. "Intelligence Without Representation." *Artificial Intelligence 47* (1991): 139-159.
7. Dawkins R. *The Blind Watchmaker.* New York, W. W. Norton and Co., 1987.
8. Davis, L. *The Handbook of Genetic Algorithms.* New York, Van Nostrand Reinhold, 1991.
9. DeJong, K. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Doctoral Dissertation, University of Michigan, 1975.
10. Fogel, D. *Evolving Artificial Intelligence*, Doctoral Dissertation, University of California at San Diego, 1992.
11. Fogel, D. "A Brief History of Simulated Evolution." In *Proceedings of the First Annual Conference on Evolutionary Programming*, edited by D. Fogel and J. Atmar, Morgan Kaufmann Publishers Inc., 1992.

12. Fogel, L., A. Owens and M. Walsh. *Artificial Intelligence through Simulated Evolution*, New York: John Wiley & Sons, 1966.

13. Forrest, S. "Emergent Computation: Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks." In *Emergent Computation*, edited by S. Forrest. Cambridge, MA: MIT Press, 1991.

14. Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1989.

15. Goldberg, D. "Zen and the Art of Genetic Algorithms." In *Proceedings of the Third International Conference on Genetic Algorithms,* edited by J. Schaffer. Morgan Kaufmann Publishers, Inc., 1989.

16. Hinton, G., J. McClelland and D. Rumelhart. "Distributed Representations.", In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, Edited by D. Rumelhart and J. McClelland, Cambridge, MA: MIT Press, 1986.

17. Holland, J. *Adaptation in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press, 1975.

18. Jefferson, D., R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. "Evolution as a Theme in Artificial Life: The Genesys/Tracker System." In *Artificial Life II*, edited by C. Langton, C. Taylor, J. Farmer and S. Rasmussen. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.

19. Kolen, J. and J. Pollack. "Apparent Computational Complexity in Physical Systems." In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 1993, (To Appear).

20. Koza, J. "Genetic Evolution and Co-Evolution of Computer Programs." In *Artificial Life II*, edited by C. Langton, C. Taylor, J. Farmer and S. Rasmussen. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.

21. Koza, J. *Genetic Programming*, Cambridge, MA: MIT Press, 1992.

22. Langton, C. "Introduction." In *Artificial Life II*, edited by C. Langton, C. Taylor, J. Farmer and S. Rasmussen. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.

23. Pollack, J. "The Induction of Dynamical Recognizers", *Machine Learning* (7), 227 - 252, 1991.

24. Ray, T. "An Approach to the Synthesis of Life." In *Artificial Life II*, edited by C. Langton, C. Taylor, J. Farmer and S. Rasmussen. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.

25. Saunders, G., J. Kolen, P. Angeline, & J. Pollack. "Additive Modular Learning in Preemptrons", In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Lawrence Erlbaum Associates, Inc., 1992.