CHAPTER VI

# THE EMERGENCE OF PROBLEM DECOMPOSITIONS AND HIGH-LEVEL REPRESENTATIONS

## 6.0 The Emergence of Problem Decompositions and High-Level Representations

In this chapter, modular programs are induced using genetic program with additional mutation operators designed to allow decompositions to emerge naturally from the problem solving process. Each created module serves as an abstract addition to the representational language that reflects some constraints of the task environment. The evolutionary weak method described in this chapter acquires appropriate modules and abstractions for the task in spite of an absence of explicit knowledge describing how to create modules for the tasks.

## 6.1 Problem Decomposition

Often, a difficult problem can be decomposed into several more simple subproblems, each of which may be more easily solved than the complete problem. In programming, humans build programs by identifying functions that solve reoccurring subproblems and thereby reduce the overall size of the code.

Simon (1969) argues that problem decomposition is an important aspect of problem solving. Simon's (1969) parable of the two watchmakers Tempus and Hora relates another advantage for problem decomposition. In this parable, the two watchmakers build products of similar complexity (1000 parts) using differing design philosophies. Tempus constructs the entire watch directly from the primitive components. Consequently, if he is interrupted before completing a watch, say by a customer calling on the phone, the intermediate state is lost and Tempus must rebuild the entire watch from the individual components. Hora's method of construction, on the other hand, uses stable intermediate modules which are individually created, assembled into larger and larger modules and eventually into the completed product. When Hora is interrupted, only the work for the module cur-

rently being constructed is lost. Simon's moral for computational problem solving is that the identification and solidification of subproblems carries with it benefits of stability and speed.

Fodor (1983) elevates modularity to a property of the mind. Fodor hypothesizes that the mind itself is modular, that the mind can be compartmentalized such that the individual compartments do very little intercommunication. Fodor argues that this compartmentalization need not correspond with the apparent modularity of the brain.

Means-ends analysis (Rich 1982; Winston 1983), a standard AI weak method, embodies a problem decomposition approach to solving problems. In means-ends analysis, problem specific knowledge is used to determine both how to decompose a problem and how to solve each identified sub-problem, possibly by further decomposition. The General Problem Solver (GPS) (Newell and Simon 1963) uses this weak method as its core computational mechanism. As in all weak methods, GPS uses task-specific knowledge, in this case to determine the decomposition of problems and their eventual solutions.

SOAR (Newell 1990; Laird, Rosenbloom and Newell 1986a; Laird, Rosenbloom and Newell 1986b) is in a sense the ultimate version of GPS. It uses task-specific knowledge together with a technique called *chunking* to determine decompositions of the task. Whenever no production rule is available in SOAR to fulfill any of the current outstanding goals, a default production rule creates appropriate subgoals. Whenever SOAR solves a subgoal, the conditions that lead to the subgoal's creation are "chunked" together with the resulting state changes into a new production rule. When a similar set of conditions occur, SOAR uses the new "chunk" rather than recreating the subgoal. Given a particular problem space, SOAR always creates the same decompositions because of its static, task-specific knowledge for when to create subgoals and hence new chunks.

Connectionist researchers are also investigating the benefits of modularity. Work by Jacobs, Jordan and Barto (1990; 1991), Nolan and Hinton (1991) and Jacob, Jordan, Hinton and Nolan (1991) demonstrates techniques for problem decomposition in connectionist networks. For instance, Jacob, Jordan and Barto (1990; 1991) begin with a predetermined architecture that has two parallel subnets and a mechanism for switching between them. Each of the two subnets is appropriate for one portion of their problem. In the end, the network learns to switch to the appropriate network based on which instance

of the problem the input belongs. Jordan and Jacobs (1993) extend this method to a hierarchical organization of subnets and decision components.

In all of the above work, the decompositions of the problem are pre-determined by the programmer either completely or in part. In the case of SOAR, the discovered chunks are merely productions representing the transitive closure over the several levels of the problem space environment. A run of SOAR does not determine an appropriate decomposition but merely benefits from one the programmer provides as explicit knowledge. In the connectionist work, the architectures determine the decomposition of the problem implicitly when the input to the network does not do so explicitly. The networks must only assign appropriate roles to the subnets consistent to their architectural restrictions. While the network training method must determine the computational roles for the individual architectural components, this is far easier than automatically determining a suitable problem decomposition.

## 6.2 The Genetic Library Builder

The Genetic Library Builder (GLiB) (Angeline and Pollack 1993a; Angeline and Pollack 1992) is an extended genetic program that uses a form of emergent intelligence to construct solutions with task-specific decompositions without relying on explicit knowledge. As with the emergence of task-specific features of the last chapter, the only modification to the evolutionary algorithm is the addition of weak representation-specific operators. No explicit knowledge describing how to extract appropriate modules from the programming language representation is available.

The Genetic Library Builder (GLiB) is a dynamic genetic algorithm using Koza's genetic programming paradigm as the base evolutionary weak method. Individuals of the population in both GPP and GLiB represent programs in a task-specific language that use a LISP expression tree syntax. 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 that allow task decompositions to emerge. These operators, named *compression* and *expansion*, are similar to the *freeze* and *unfreeze* operators of the last chapter. Both are applied probabilistically to evolving individuals, as are all evolutionary operators.

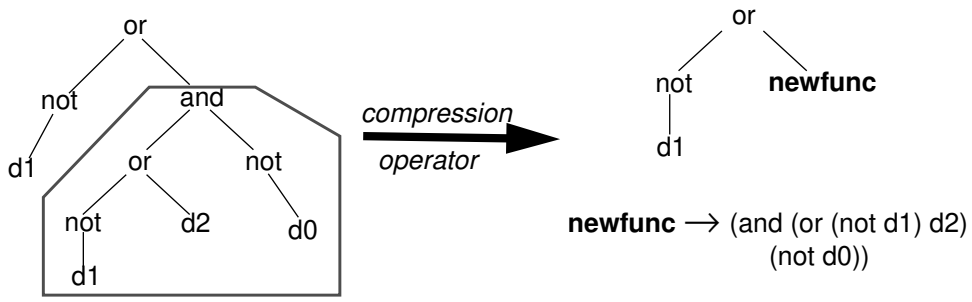


***Figure 24:*** *Creation of a compressed module from a randomly selected subtree of a genotype. Newfunc's definition is added to the library.*

### 6.2.1 Compression of Modules

Compression begins by randomly selecting a subtree from the parent. This subtree becomes the body of a new function, called a *module*, that is added to the *genetic library,* a logical storehouse for the definitions of all modules in the system. This library associates the chosen subtree with a unique identifier that becomes the module's name. The subtree is then extracted from the parent and replaced by the newly defined module's name. When a genetic program with a compressed module is evaluated, the definition of the module is retrieved from the genetic library and executed as though the original subtree were in the genetic program at the point the module's name appears. Thus the compression operator performs only a syntactic modification to the genetic program rather than a semantic one.

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 as shown in Figure 24. 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 table. During a genotype's execution GLiB retrieves the definitions of compressed modules from the genetic library. This form of compression is identical to Koza's *encapsulation* operator (Koza 1992) although Koza takes this operator no further.

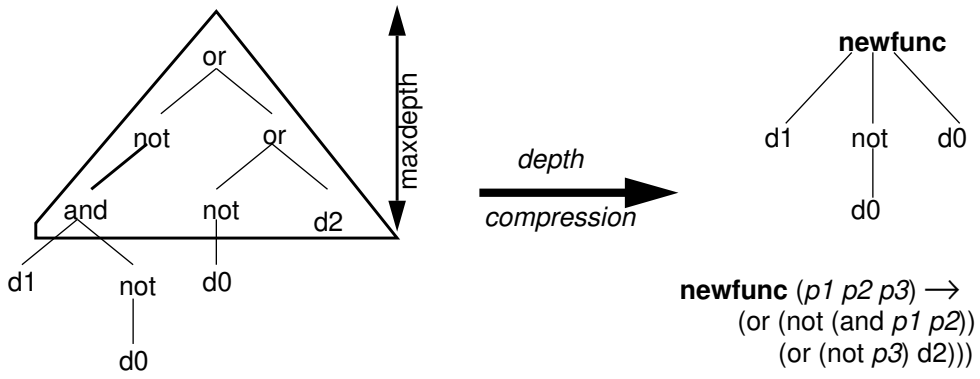


***Figure 25:*** *Depth compression applied to a subtree larger than the maximum depth. The module's definition has parameters as a result.*

Exactly how the randomly selected subtree is pruned to create a module's definition depends on the compression method in use. The *depth compression* method extracts the subtree from its root node and clips off any branch exceeding a randomly selected maximum depth. The maximum depth for each new module is generated by a uniform random variable over a user defined range. Figure 24 shows the result of a compression when each branch of the subtree is within the selected 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 24. 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 25 shows this instance of module creation.

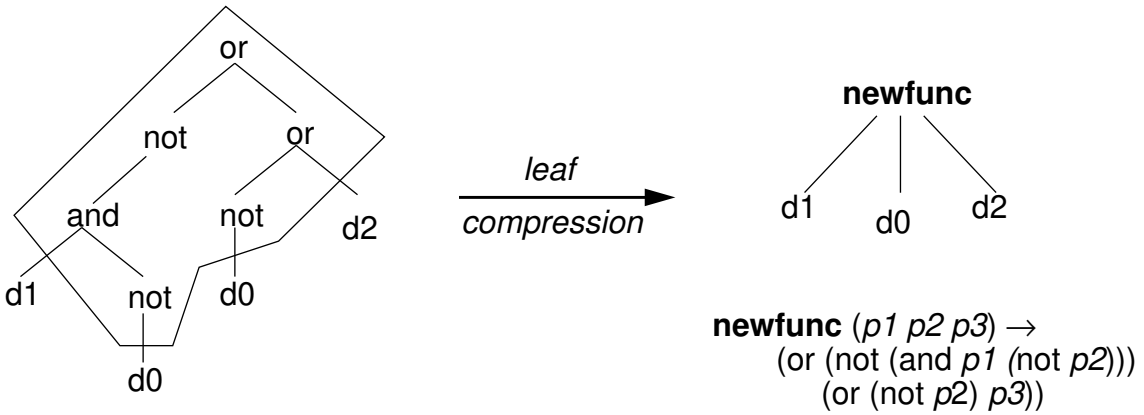


***Figure 26:*** *Leaf compression applied to a selected subtree.ote that identical leaves are represented by the same parameter name in the defined function. Expansion is the reverse of this process.*

Depth compression is but one method for extracting a module from an individual for compression. An example of another modularization method is *leaf compression*, which again selects a random node as the root of the subtree. However, in leaf compression, each unique leaf is replaced with a unique parameter name, as shown in Figure 26. The advantage of leaf compression over depth compression is the reuse of parameters within the defined module. Depth compression always produces a module in which a parameter appears only once. On the other hand, depth compression has the ability to construct modules from partial subtrees while leaf compression always compresses the full subtree. In the experiments that follow, each instance of compression randomly selects between the depth compression and leaf compression methods with equal probability.

Like the components of Hora's watches, the modules formed by compression are stable and protected from dissessembly by other operators. Because the modules are represented by single symbols, no crossover operation can affect its definition. The modules are structurally preserved until an expansion operation replaces the symbol in the program with its definition. Schaffer and Morishima (1987) discribe a different method for evolving appropriate crossover points in a fixed-length representation but with very different properties.

### 6.2.2 Expansion of Compressed Modules

One drawback to the compression operator is that each compression removes genetic material from the population and thus reduces the variability of the population. 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 (De Jong 1975; Goldberg 1989a; Davis 1991). Typical solutions include altering the scaling of the fitness values or altering the frequency of crossovers and mutations. Neither of these solutions are appropriate for GLiB.

To offset the loss of diversity in the population due to the application of the compression operator, GLiB provides an additional mutation operator. When applied, the *expand* operator searches the parent for all modules at the top level and selects one at random. The chosen module name is then replaced by its definition stored in the genetic library. This 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 Figure 24, Figure 25 and Figure 26 as illustrations.Because mutations are executed locally in an individual, other population members which also refer to the module being expanded are not affected.

### 6.2.3 Non-monotonic Formation of Hierarchical Decompositions

A hierarchical decomposition forms in GLiB when a subtree containing an already compressed module is itself compressed. With continual manipulation of the individual, the hierarchy deepens and widens. Because expansion only works on the highest level of modules in an individual, modules further down the hierarchy are protected from being expanded and further manipulated. Thus the problem is decomposed from the bottom of the hierarchy towards the top as a natural consequence of the dynamics of the compression and expansion operators.

The combination of the compression and expansion mutations allow for the non-monotonic development of the task decomposition. The random identification of subtrees

to compress 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 by an expansion operation provides the chance to capture a better version of the module in a later compression. For instance, a compressed module can be created and the genetic program can evolve further using the compressed module. Once further refinement is made to the program, the module can be expanded and modified. Such non-monotonic manipulations of hierarchical decompositions may be indispensable when solving complex problems.

### 6.2.4 Evaluation of Modules

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 (Bedau and Packard 1992). If the population uses a module frequently then it must provide some consistent beneficial behavior in previous generations.

The only way that a compressed module can be propagated in GLiB is through reproduction and crossover. A module is not added explicitly to the global primitives of the genetic program since this would increase the number of global primitives greatly and affect performance. Thus any mutation operator that generates a random subtree from the available primitives does not insert a dynamically defined module. Compression and expansion are local manipulations in that they affect only the genetic program being mutated.

As an added benefit, the constrained propagation of compressed modules provides the foundation for their evaluation as a by-product of the dynamics of the evolutionary process. Once a subtree is compressed, no further manipulation of the module's contents by crossover or other mutations is permitted until an expansion restores it. If the compression protects a subtree that is a beneficial decomposition, meaning that it solves some subproblem of the task sufficiently, then the presence of the module contributes to the overall success of the program on the task. If the program's fitness is high compared to the other

population members, it produces children that also contain the module. As long as a compressed module provides an evolutionary advantage to the members that contain references to it, or at least does not inhibit their reproductive success, the module spreads through the population. If, however, a module protects a subtree that is detrimental to solving the problem, it eventually becomes a liability to each offspring that uses it, limiting the fitness of the individual. Eventually, all individuals relying on the malformed module disappear from the population through attrition.

Modules that survive over several generations and become entrenched in the population are termed *evolutionarily viable* (Angeline and Pollack 1992). In this case, a beneficial module is "induced" when more and more of the population relies on its content. Often beneficial modules spread through the population very quickly so that their usage grows dramatically. Such quick proliferations are indicative of phase changes that Pollack (1991) suggests are a form of induction in complex systems including human cognition.

A module is evaluated dynamically by how useful it is to the population members that use it or contain a reference to it. The global worth of a particular compression emerges from the local interactions of a population with the fitness function and each other. Such a usage centered method for evaluation of modules is a result of the opportunistic nature of emergent intelligent techniques.

### 6.2.5 The Emergence of High-Level Representations

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 are constructed from the primitives and modules already in the representation language, they are necessarily at a higher level of abstraction. If they are evolutionarily viable and

One of the most important aspects of GLiB and its emergent modules is that each module is an abstraction away from the primitive language of the genetic program and towards a task-specific language that is more conducive to solving the problem. Much like the progression of computer languages from machine code to high-level languages, the dynamics in GLiB allow increasingly general abstractions to form and be tested for solv-

ing the problem. Because the dynamics of GLiB propagates only those abstractions that do not inhibit the evolution of better solutions to the problem, the resulting high-level representations in GLiB are necessarily task-specific. However, like the solutions themselves, these abstractions are highly opportunistic and consequently difficult to describe.

### 6.2.6 Task-Specific Feature Acquisition

The automatic acquisition of task-specific features is a problem first described by Samuel (1966; 1959). In his classic checker program, Samuel demonstrated how to use a variety of knowledge sources, including books, humans and other programs, to teach a program how to play checkers at a moderate level. In order to represent an evaluation function for the program, Samuel chose several sets of features to describe the positions of the pieces on the board. The evaluation function combined these binary features into a sum with coefficients to designate their relative importance. Samuel (1959; 1966) learned the coefficients of the equation from examples of expert checker play using a simple update rule. Samuel lamented the difficulty of determining sufficient features to adequately represent the checker board and described his desire to acquire them automatically.

The difficulty of determining a best set of features for checkers, or many other complex tasks, is that there may be no objective set of features that are sufficient for evaluating every move of every game. Samuel (1966) understood this and began using multiple evaluation functions: one for moves early in the game, one for the middle game and one for the end game. However, in each evaluation function, the same features are used.

The salient features for representing a solution for a complex task are not necessarily static and objectively definable. Appropriate features are dependent not only on the task but on the current state of solving the task and even the current instance of solving the task. For instance, in a game, the salient features of the board change from the beginning of a game to the end. In backgammon, once all of one player's pieces are in the "home" area, most features of the board that were crucial at the beginning of play are obsolete. In addition, an expert backgammon player may use a different set of features for different players based on perceived ability and past experience. Thus the features required for solving a complex task often require a similar dynamic.

## 6.3 Generality of Emergent Modules

GLiB's form of emergent module definition is general across all evolutionary computations since it relies only on empirical credit assignment that is a natural dynamic in all evolutionary weak methods. For instance, the previous chapter demonstrated that a variant of compression and expansion, called *freeze* and *unfreeze*, speed-up the acquisition of finite state machines (FSMs) in a evolutionary program. Similar methods could be used in genetic algorithms and evolution strategies.

## 6.4 Comparison to Koza's ADFs

Koza (1992) also describes a method for constructing modular genetic programs which he calls *automatically defined functions* (ADFs). In his method, the decomposition of the program is designed prior to evolving the genetic programs, each ADF having a unique primitive language that is specified by the programmer. Each individual contains its own local definition of the designed functions. Unlike GLiB's modules, when using ADFs the individual's functions are initialized to be random and are evolved along with the individual by an occasional crossover operation with another individual's local definition. Eventually, an individual and its ADFs settle into a complete solution to the problem.

While the definition of each individual ADF is emergent, the hierarchy of modularity, including the number of parameters to each module, is necessarily static. This makes ADFs a special case of applying GP at several levels in parallel. If explicit knowledge of the solution's decomposition is available, then it is better to allow only the definition of the functions to emerge as in ADFs. Systems with more emergent components are more general. Therefore, because GLiB allows both the hierarchy and the definitions of the functions to emerge from the dynamics of the systems, GLiB's method of creating task decompositions is more general than ADF. As is always true, if one system is more general than another then the less general algorithm often has certain other advantages. Kinnear (1994b) describes such an advantage for ADFs.

The emergence of high-level representational abstractions is the chief difference between GLiB and Koza's ADFs. Where GLiB is dynamic in its pursuit of all aspects of the hierarchical structure for a particular task, the pre-specification of hierarchical organization of ADFs forces them to create the same modularity each time, although the semantics of the individual functions may be different. While the definitions of ADFs are

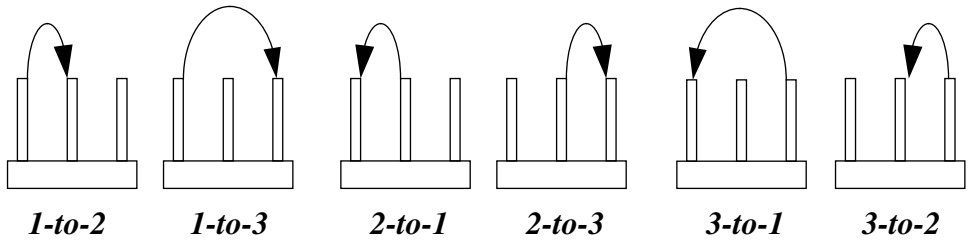


***Figure 27:*** *Primitive language for Tower of Hanoi experiments. Each command moves a single disk between the specified pegs.*

emergent, their organization is not. This leads to the conclusion that while ADFs can allow modular organizations to emerge, they cannot allow higher-level abstractions to emerge. Both the hierarchical organization and the semantics of each level of the hierarchy must be manipulated to avoid limiting the organization of the developing individuals.

## 6.5 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 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.

### 6.5.1 Tower of Hanoi Experiments

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 27. Each primitive moves the top disk from one peg and places it on

```
(1-to-2
  (1-to-3
   (2-to-3
     (1-to-2
      (3-to-2
        (Func2950
         (1-to-3
           (2-to-3
            (2-to-1
             (2-to-3
              (3-to-2
                (2-to-1
                 (2-to-3
                   (2-to-1 (1-to-2 (Func3280)))))))))))))
```

***Figure 28:*** *Evolved program for Towers of Hanoi experiment. Compressed modules are designated with the name FUNC####.*

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, the experiment 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 leave a larger disk on top of a smaller disk, were not executed by the fitness function. As a penalty, a program 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. Figure 28 shows the resulting

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 the evolved module *Func2950* used in the evolved program shown

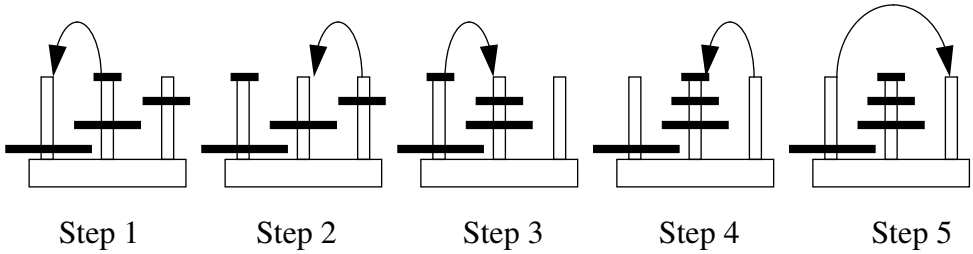


***Figure 29:*** *Sequence of moves implemented by an evolved module. The configuration of the disks in Step 1 is the configuration at which the module is called in the evolved program.*

in Figure 28 are pictured in Figure 29. 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 intermediate configurations of the problem with the largest disk still on the first peg, the evolved module shown in Figure 29 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.

### 6.5.2 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 designed player. Figure 30 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

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

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

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*** returns 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.

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 observes 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. In these experiments, the rules of play are considered 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 to 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.

The best evolved program after 200 generations, shown in Figure 31, 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 evolved modules each had a total of nine separate calls in the fully expanded tree.

Figure 32 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 32c) but does not win the game until two turns later (Figure 32e). 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 three fewer points. By extending the game it increases its score without the possibility of losing.

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 33 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 33a, an initial increase occurs soon after the module is defined, showing it provided

```
(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 31:*** *Best evolved modular program from described experiment. Evolved modules are named FUNC####.*

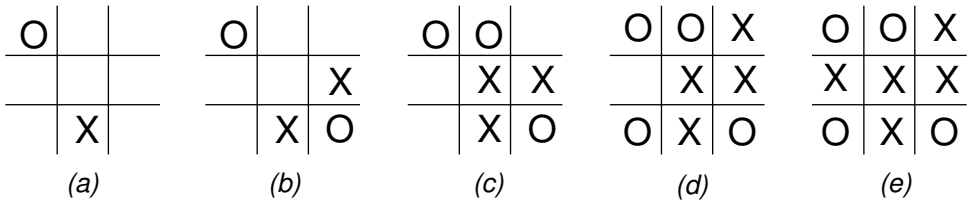


***Figure 32:*** *A sample game from the described run. The evolved program is "X" and is playing first. It sets up the first fork in (b) and completes it in (c). The second fork is completed in (d) and leads to a win in (e). The evolved program received a score of 20 points for this game.*

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 33b 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 appearing in the population. Finally, Figure 33c 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 (Pollack 1991; Angeline and Pollack 1992).

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 33. 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, no modules were found 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*

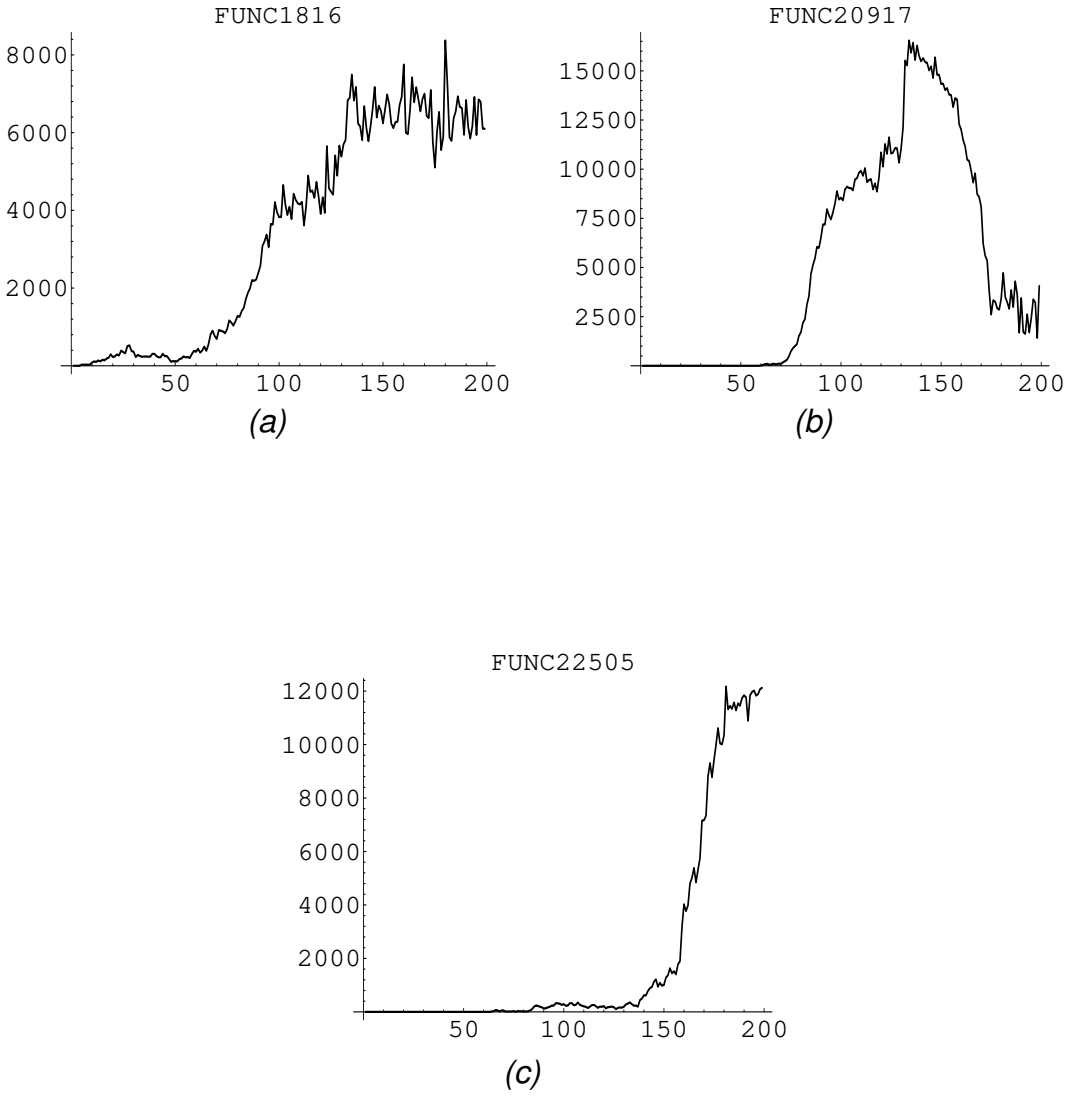


***Figure 33:*** *Graphs showing number of calls (y-axis) per generation (x-axis) for three evolved modules. Note the scales are different in each.*

of procedural knowledge found in connectionist networks (Hinton, McClelland and Rumelhart 1986; Sejnowski and Rosenberg 1987; Smolensky 1988).

## 6.6 Discussion

It is apparent from Figure 33 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.

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 indirectly 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 GLiB's 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. Fogel (1992; 1993b) identifies this same trade-off for different reasons.

Extracting subtrees from the programs and placing them into GLiB's genetic library is a convenience rather than a necessity for this 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.

## 6.7 Conclusion

This chapter demonstrated that the addition of two simple operators to an evolutionary weak method could allow modular programs to be evolved. The validity of a particular decomposition was determined by both the correctness of the semantics and the mutability of the ensuing representation. The identification of a useful module was supported by its

proliferation through the population. Each created useful module, by virtue of its benefits to the proliferation of the population member in solving the task, encodes some regularity from the task environment without explicit knowledge of what makes a good module or how to extract task-specific information being placed in the problem solver. Given that the modules reflect commonalities of the task environment, they also represent levels of abstraction above the original representational level. These abstractions are induced as a natural consequence of the problem solver interacting with the task environment.