Evolutionary Module Acquisition

Peter J. Angeline and Jordan 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 the Proceedings of:

The Second Annual Conference on Evolutionary Programming,

February 25-26, 1993
La Jolla, California

Evolutionary Module Acquisition

Peter J. Angeline and Jordan 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

Evolutionary programming and genetic algorithms share many features, not the least of which is a reliance of an analogy to natural selection over a population as a means of implementing search. With their commonalities come shared problems whose solutions can be investigated at a higher level and applied to both. One such problem is the manipulation of solution parameters whose values encode a desirable sub-solution. In this paper, we define a superset of evolutionary programming and genetic algorithms, called evolutionary algorithms, and demonstrate a method of automatic modularization that protects promising partial solutions and speeds acquisition time.

1. Introduction

Evolutionary programming (EP) (Fogel 1992; Fogel et. al. 1966) and genetic algorithms (GAs) (Holland 1966; Goldberg 1989) have borrowed little from each other. But there are many levels at which EP and GAs are similar. For instance, both employ an analogy to natural selection over a population to search through a space of possibilities. Where these techniques intersect is a profitable place to look for phenomena that reveal deeper truths about the structure of all similar algorithms.

Our research concentrates on the more general set of *evolutionary algorithms* (EAs) (Angeline 1993), which contains both evolutionary programming and genetic algorithms in addition to many other methods that use analogies to evolution for problem solving, search and optimization. One phenomenon that many evolutionary algorithms share is the manipulation of representational components that are necessary for the viability of the individual. We would prefer that once such important representational components are discovered, they are preserved from further manipulation. However, there is no general method that can consistently and definitively identify which components of an individual require no further manipulation. As a result, these components continue to be modified when creating new offspring which slows the search. This problem is exacerbated when the representation is large or dynamic due to combinatorial explosion of the search space.

In this paper, we describe a technique for improving the speed of acquisition for evolutionary algorithms by reducing the manipulation of necessary components of the representation. The selection of which components to "freeze" is done randomly and evaluated by the reproductive advantage it provides to the individual. We demonstrate this technique on an EP control problem and describe an additional variant of the technique that enables higher levels of representational expression to emerge from the evolving solutions. From this discussion we suggest a more general mutation operator for evolutionary programs that can produce self-similar solutions. We begin with a brief discussion of evolutionary algorithms and the inherent empirical power of simulated evolutionary methods.

2. Evolutionary Algorithms

Evolutionary algorithms (EAs) are a set of search and optimization methods that simultaneously manipulate a *population* of search space points. These algorithms differ from parallel implementations of what can be called *single point methods*, e.g. any classical AI search technique (Rich 1983),

```
begin
  i := 0;                          {Initialize generation variable}
  P0 := I();                       {Create initial population}
  P0 := F(P0, i);                  {Evaluate initial population}
  while not H(Pi, i) do            {Do until Halt criterion is true)
    begin
      i := i + 1;                  {Construct a new population}
      for j=1 to L(Pi, i) do       {Next Population length}
        begin
          x := S(Pi-1, i, j);      {Select from last population}
          Pi := Pi + R(x, Pi, i, j); {Add an offspring of x to Pi}
        end;
      Pi := F(Pi, i);              {Evaluate the new population}
    end;
end;
```

Figure 1: *Algorithm template for evolutionary algorithms. P_i is the population at generation i. x and j are temporary variables. The other functions are described in the text.*

since subsequent population members are dependent on more than one member of the previous population. In other words, the presence of a point in population $n+1$ is dependent on several points having appeared in population n. Parallel implementations of single point methods allow each of the solutions inspected at a particular parallel time step to be dependent on at most one solution of the previous time step, if any.

More formally, we define an evolutionary algorithm to be the 6-tuple $EA = (I, F, R, S, L, H)$ where each component is a function that is independent from the other components of the EA. The component functions are as follows: I is the population initialization function; S is the function that selects members of the population for reproduction; R is the reproduction function; L is a function that determines the size of the population; H is the halting criterion for the algorithm; and F is a function that evaluates the worth of each member of the population, more commonly called a *fitness function*. Figure 1 shows the algorithmic template for an evolutionary algorithm and how each of these components is used. Similar figures have appeared in several previous incarnations for genetic algorithms (Grefensttete 1989; Michalewicz 1993; Davis 1991). Notice in Figure 1 that some of the functions take on several parameters. This acknowledges the diversity of functions available to an evolutionary algorithm and the variety of variables on which these functions can depend. Typically, many of these parameters are ignored by the actual function.

Distinctions between evolutionary algorithms arise through the diversity of characteristics found in their respective component functions. For instance, evolutionary programs and genetic algorithms differ most pointedly in the philosophy of their respective reproduction functions. The reproduction function used in genetic algorithms models evolution at the level of an individual's genetic composition while the reproduction function in evolutionary programming employs a species-oriented model. Both models of evolution are applicable to different classes of problems that require the specific strengths and weaknesses of one model over the other. For a more complete discussion of evolutionary algorithms and their various components see Angeline (1993).

In spite of these differences, the reproduction functions of evolutionary programming and genetic algorithms share a much stronger similarity. Both replicate members of the population based on their fitness relative to the population. The next section discusses the strength of this simple commonality among all evolutionary algorithms.

3. The Empirical Strength of Reproduction

One of the common links between all evolutionary algorithms is the reproduction of current population members to create the subsequent population. This basic operation supplies a strong empirical component to all evolutionary algorithms which has not been fully exploited or explored.

Holland (1975) fitness proportionate reproduction as a major component of his schema theorem. A schema is a set of bit string patterns across the assumed fixed-width binary string representation of the population. Schemata take the form $(1 + 0 + \#)^n$ where n is the length of the binary string representation. A "1" or a "0" at position i in a schema signifies that each string in the set represented by the schema contains that value at that position. A "#" at position i designates strings that contain

either a "1" or a "0" at position i are in the set. Notice that the possible schemata for a given length n do not cover all possible subsets of binary strings of length n. For instance, there is no schema of length 4 that represents the set {0101, 1010}.

We generalize from the concept of a schema to *any* representational feature, μ, of the individual. A representational feature can be any aspect of the representation as long as it is copied from parent to offspring during reproduction. For instance, it could be any subset of possible values for particular positions in the representations or something less tangible like a constraint on the variance for a particular set of real components.

Let $\Lambda(\mu, t)$ be the set of population members at time t which contain the feature μ. Also, let $\sigma(i, t)$ be the probability that population member i will be selected at generation t to be in the next generation. Generally, $\sigma(i, t)$ will be correlated with the fitness of the individual. The expected number of population members at time t+1 which contain feature μ is given by:

$$m(\mu, t+1) = n\,[1 - \varepsilon(\mu, t)] \sum_{i \in \Lambda(\mu, t)} \sigma(i, t) \qquad (1)$$

where n is the population size and $\varepsilon(\mu, t)$ is the probability that the feature will be disrupted during reproduction. We can rewrite equation (1) as follows:

$$m(\mu, t+1) = n\,[1 - \varepsilon(\mu, t)] \frac{m(\mu, t)}{m(\mu, t)} \sum_{i \in \Lambda(\mu, t)} \sigma(i, t) \qquad (2)$$

$$m(\mu, t+1) = n[1 - \varepsilon(\mu, t)]\; m(\mu, t)\; \bar{\sigma}(\mu, t) \qquad (3)$$

where $\bar{\sigma}(\mu, t)$ is the average probability of selection for a member of $\Lambda(\mu, t)$. Notice that when μ is a schema, $\bar{\sigma}$ is defined in accordance with fitness proportionate reproduction and ε is defined to adjust for crossover and point mutation we recover the lower bound expression from the schema theorem.

The interest in equation (3) for general feature propagation stems from its characterization of the properties of reproduction as the relative fitness of the population members with and without μ change. As long as $\bar{\sigma}(\mu, t) > 1/(n - n\varepsilon(\mu, t))$, the number of population members with the feature is likely to be larger in the next generation. On the other hand, if $\bar{\sigma}(\mu, t) < 1/(n - n\varepsilon(\mu, t))$ then the number of population members containing μ is likely to decrease. In other words, as long as μ presents a sufficient selection advantage to the subpopulation that contains it, additional population members will tend to acquire the feature. When μ is no longer an advantage, the feature will be removed from the population automatically by the natural dynamics of the evolutionary algorithm.

Equation (3) characterizes the empirical power of the reproductive process used in all evolutionary algorithms. It is this strength that separates EAs from other search and optimization techniques. Of equal importance is the generality and exploitability of this reproductive process. For example, Davis (1991) and Bäck (1991) describe different methods for evolving the parameters for manipulating population members for two different evolutionary algorithms. We wish to tap into this empirical component of evolutionary algorithms to address the unwarranted manipulation of imperative components of an individual.

4. Evolutionary Module Acquisition

Evolutionary module acquisition relies on the empirical strength of reproduction in an evolutionary algorithm to acquire problem specific groupings of the representational components in developing population members. These groupings designate components of the representation which are to be immune from manipulation by the reproductive operators. This forces the grouped components to be copied "as is" into all subsequent offspring.

To identify appropriate modules in the evolving individuals, we add two operators to the reproduction process. The first operator, which we call *compress*, selects a portion of the offspring to preserve from future manipulation. The collection of components that are compressed together we call a

module. The second operator, *expand*, is the opposite of compress. Expand releases a portion of the compressed components so they can once again be manipulated by the reproduction operators. The opposite actions of these operators is important to allow the modularization to be non-linearly adaptable to the changing population.

Because there is no single, general method of identifying what portions of the individual should be compressed, the composition of each module is selected at random. By the arguments of the last section, if a randomly created module protects crucial components of the representation from modification, thus posing a benefit to the reproductive ability of the individual, then this modularization will be passed on to its offspring. Likewise, if a module is detrimental to the member's proliferation, then that module will be selected out of the population. Referring back to equation (3), the components in the module become the feature of interest, μ, and $\varepsilon(\mu, t) = 0$ since the reproductive operators can not modify the contents of the module. Equation (3) then shows that $\bar{\sigma}(\mu, t) > 1/n$ is a sufficient average selection probability to propagate the module through the population.

Exactly how the compress and expand operators modify the individual to signify modules is specific to the representation. The only guideline is that the manipulation should be transparent to the fitness function. In other words, the fitness of an individual before and after any series of compressions and expansions should never change. Compression and expansion perform only a syntactic manipulation to the individual and have no semantic side effects. The side effects of these operators apply only to the reproduction of the individual.

There are two different methods we have identified for the compression and expansion of modules. The first selects any subset of uncompressed components in the individual for compression and any subset of compressed components for expansion. No care is given to ensure that the composition of a compressed module is preserved and uncompressed as a unit. We call this simple form of compression *freezing* since the only effect of a compression is to "freeze" the values of the compressed representational components. Once a component is compressed, no further compress operations will effect it. *Atomization*, a second form of compression, is more true to the operator's name. In this method, the compress operator selects a portion of the representation, freezes it and then treats the entire compressed module as a new component of the representation. Because the composite module is now an atomic component of the representation, it is available for manipulation as a *single* representational unit. This includes additional compressions into other modules. Unlike freezing, this second type of compression creates a hierarchical organization of modules, i.e. modules within modules. In the following sections we discuss the advantages of both methods of compression on specific representations.

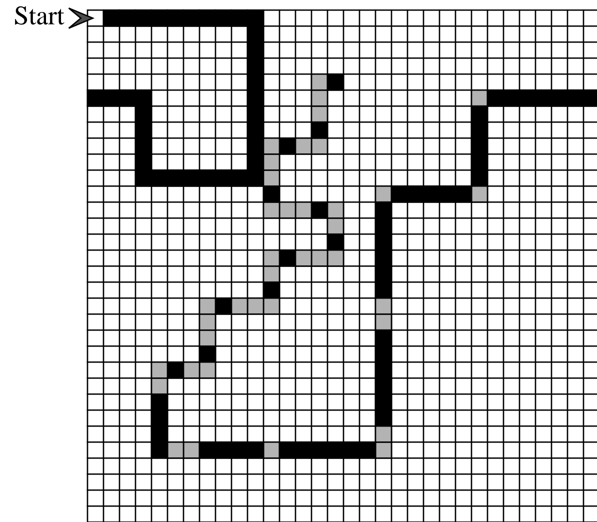

Figure 2: *Path of food on the toroidal grid used in the ant problem. Simulated ant starts at the labeled position facing EAST. Black squares are "food" which disappear after the ant enters that position. Grey squares identify the quickest route through the path and cannot be seen by the ant. There are 89 positions with food in the path.*

5. Freezing Finite State Machines

To illustrate the effects of the *freezing* form of compression, we chose a control problem described in Jefferson et. al. (1992) called the artificial ant problem. The goal of this task is to evolve a controller to guide an artificial ant along the path of food shown in Figure 2 within 200 time steps. The path rests on a 32x32 toroidal grid and contains a total of 89 pieces of food, shown in black in the figure. The ant is equipped with a sin-

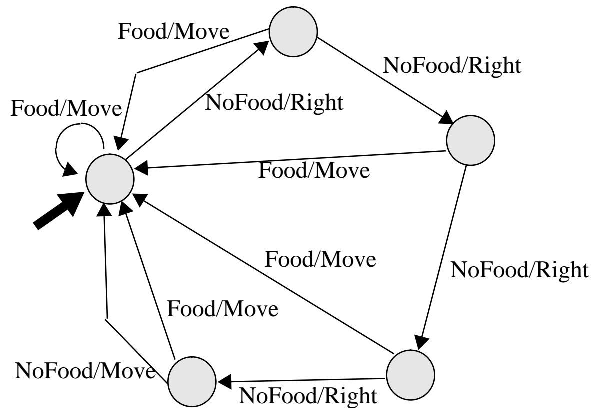

Figure 3: *Simple FSM that traverses the path of food in 314 time steps. The oversized arrow designates the initial state.*

gle sensor that can detect the presence or absence of food in the square directly in front of it. Actuation of the ant is signaled through four possible action commands: move one square forward (MOVE), spin left 90° (LEFT), spin right 90° (RIGHT), or do nothing (NOOP). On each time step, the ant executes an implicit sense/act loop where an input of FOOD or NOFOOD is given to the ant and it executes a single action command. Once the ant enters a position on the grid with food, the food is removed and a point of fitness is awarded.

While this problem appears simple, the criterion of completing the path within 200 time steps makes it rather difficult. For instance, the simplistic path following strategy represented by the finite state machine (FSM) in Figure 3 requires a total of 314 time steps to traverse the path. In order for the ant to receive the maximum fitness, it must induce a controller tailored to the specifics of the path.

Jefferson et. al. (1992) used a genetic algorithm to compare the evolution of bit strings which were interpreted as either finite state machines (FSMs) or recurrent neural networks depending on the experiment. Jefferson et. al. (1992) used a population size of 65,536 and replaced 95% of the population each generation for both representations in both experiments. Evolving an FSM controller for this problem took 52 generations while the neural network controller took 94 generations to emerge. Thus their genetic algorithm searched a total of 3,303,014 FSMs and 5,917,900 recurrent neural networks to solve the ant problem.[1]

In our compression experiments, we evolve FSM controllers for the ant problem using evolutionary programming with and without freezing. To compress an FSM, a single state and up to 5 transitions are selected at random and designated as being "frozen" in the representation. Conversely, expansion "unfreezes" a single randomly selected frozen state and up to 5 frozen links. No effort was made to unfreeze components that were frozen at the same time, each expansion could select any frozen component at anytime. When creating an offspring there was a 10% chance that a compression would be performed and a 20% chance that an expansion would be performed. The higher expansion rate was to ensure that if local minima were reached the number of protected components would decrease and allow components that had been previously protected to be mutable again. All compressions and expansions were done to the offspring prior to the other mutations. At most 75% of the states and 75% of the transitions for any one FSM were allowed to be frozen at a time.

In order to provide slightly more discriminations between evolved FSMs, we modified the original fitness function to be:

$$food + 0.01\left(1 - \frac{t}{200}\right) \quad (4)$$

where *food* is the amount of food found by the ant within 200 time steps and t is the time step on which the last piece of food was discovered. This fitness function encodes a preference for FSMs that acquire the same amount of food in fewer time steps.

Our method of evolving FSMs is slightly different than the methods described in Fogel et. al. (1966) and Fogel (1992). First, during early experiments with this problem we noticed that the evolutionary process created individuals with larger and larger numbers of states until reaching the allowed maxi-

1. See Angeline et. al. (1993) for an evolutionary program that constructs a recurrent neural network for a variation of this problem.

mum number of 32. In addition, we found that a disproportionate percentage of the population would acquire the same fitness for a considerable amount of time. In these early experiments we were using an equal chance between adding a state, deleting a state or modifying a transition.

Because we knew the ant problem could be solved in far fewer than the 32 state maximum, we tried to determine exactly what was causing the population to consistently acquire the maximum number of states and why so many retained the same fitness. After analyzing a few evolved machines it became apparent that a large percentage of the states were in fact unused. We deduced that the additional states ensure a high percentage of self-replication for the FSMs. By including a large number of superfluous states, whenever a state deletion or manipulation of a transition is performed, there is a better chance that the offspring will retain the ability of its parent. This accounts for the inordinate number of machines with the same fitness and maximum number of states.

To discourage such unproductive manipulations, we altered the mutation of an FSM so that there was an even chance of mutating either a state or a transition. If a state mutation is selected, the chance of deleting a state as opposed to adding a state is given by:

$$P(delete) = \frac{numstates}{maxstates} \qquad (5)$$

where *numstates* is the number of states in the parent FSM and *maxstates* is the maximum number of states allowed for the problem. Thus if the number of states in the parent is less than half that allowed for the problem, there is a greater chance of adding a state than deleting a state. When the number of states in the machine is more than half of the total number allowed, there is a preference for deleting states. While this did not entirely curb the tendency for the runs to approach the maximum number of states, it did allow for a consistently broader distribution of sizes in the population and improved the overall acquisition times.

Run	*Number of FSMs Evaluated (Number of Generations)*		*Speed Up*
	Without Compression	*With Compression*	
1	187,950 (1251)	63,000 (418)	2.99
2	269,850 (1797)	152,100 (1012)	1.77
3	331,200 (2206)	156,600 (1042)	2.12
4	734,850 (4897)	607,950 (4051)	1.20

The number of mutations made to a parent to create an offspring in our experiments is given by the function:

$$1 + \mathrm{round}(\mathrm{abs}\ [N(0, T) \times size]) \qquad (6)$$

where *size* is the number of transitions in the parent FSM and $N(0, T)$ is a gaussian random variable with mean 0 and variance proportional to the fitness "temperature" of the parent. Our method of selection was the competitive method described in Fogel (1992) with the exception that if the two population members being compared had the same fitness, a "winner" was chosen randomly. The number of competitions per individual was 5. The population was sorted by their competitive selection scores with the best half of the population retained and replicated to create the following generation. Each population member created exactly one offspring for the next generation. A population size of 300 machines was used for each run.

To determine the effect of simple compression on the evolution of FSM controllers for the ant problem, eight runs were executed, four with compression and four without. Table 1 shows the total number of FSMs constructed in each run until one FSM in the population guided the ant to all 89 pieces of food within the allotted 200 time steps. The number of generations created for each run is listed in the parentheses. The runs are sorted into

increasing order so that the fastest and slowest of both methods are compared directly. Speed-up, shown in the last column, was computed by dividing the result from the run without compression by the result of the run with compression. Two additional runs, one with compression and one without, did not find a solution within the maximum 5000 generations and have not been included in the table.

First notice that all of the runs, both with and without compression, evolved solutions to the ant problem more quickly than Jefferson et. al. (1992). The improvements range between factors of 52 and 4.5. Such improvement is often the case when converting from a genetic algorithm using binary representations to an evolutionary program. This is because the added complication of a function to convert between the genotypic and phenotypic representations in the genetic algorithm is avoided in evolutionary programming. Whether or not the differences between the two evolutionary algorithms is solely responsible for the improved results or if our modified fitness function also contributed to the speed-up is unknown. Regardless, the improvement over Jefferson et. al. (1992) is noteworthy.

Next, notice that each of the runs show speed-up in favor of compression. An explanation for these results is that the freezing process identifies and protects components that are important to the viability of the offspring. Subsequent mutations are forced to alter only less crucial components in the representation. Figure 4 shows the evolved FSM from run 1 with compression. Frozen transitions and states are shown in grey. This FSM guides the ant to all 89 pieces of food in 193 time steps. There are a total of 22 states in this solution only eleven of which are used to solve the problem. Twelve of the states and 23 of the 44 possible transitions are frozen in this solution. It is difficult to deduce any significance for the frozen components in this FSM. The non-determinism of our acquisition method places an emphasis on whatever gets results. One observation is that states 1 through 6 and the transitions between them are largely protected from mutation. This portion of the FSM is responsible for traversing the continuous stretches of food of the path. Mutation of one of these states

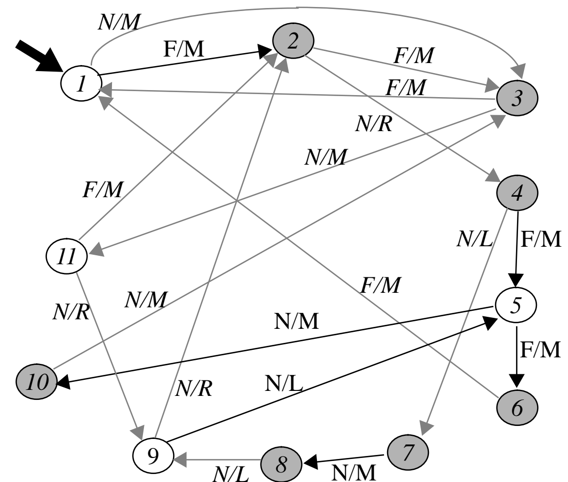

Figure 4: *FSM evolved with compression in 420 generations. Frozen states and transitions are shown in grey. Input symbol set is (F, N) and output symbol set is (M, L, R, N) as described in the text. Extraneous states and transitions are not shown. The initial state is indicated by the oversized arrow.*

or transitions has a high probability of creating an offspring far below optimal.

The variation in the times for the runs using compression illustrate an important point of this technique. How well modularization works on any given run is determined by the types of modules it acquires. In some cases, as in run #1 from the table, the modularization will quickly find a good compression and expedite the discovery of a solution. In other cases, simple freezing will protect a portion of the representation that is in need of mutation and allow it to remain unmodified. Occasionally this inhibits the evolutionary process and cause longer rather than shorter acquisition times. But such inhibition will be rare if the representation is of sufficient flexibility. Assuming this, modifications which “work around” inappropriately frozen components will be discovered.

6. Evolving Modular Programs

A second method we have investigated for compression is *atomization*. In this method we compress the selected components into a module so that they are associated together as a single atomic representational unit. Typically, the components selected for this type of compression are chosen to reflect some naturally exploitable modularity in

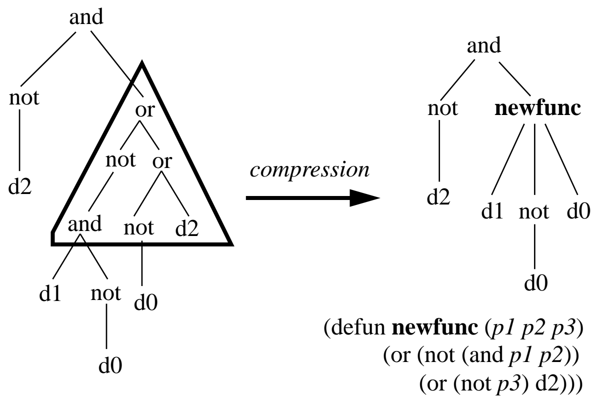

Figure 5: *Compression of tree representation used in genetic programming. The subtree is removed from the individual and replaced by a new function call defined with the removed subtree. The expansion of a compressed function reverses the process by replacing the compressed function name with the original subtree.*

the representation's syntax. For instance, Figure 5 shows the effect of this compression operator on the labeled tree representation used in the Genetic Library Builder (GLiB) (Angeline and Pollack 1993). GLiB is a genetic algorithm which evolves modular expression trees to solve problems. The expression trees are interpreted as Lisp programs and are executed to produce a behavior in an environment. The behavior of a program in the environment provides is rated by the fitness function, much as in the evaluation of FSMs in evolutionary programming. The expression tree representation for genetic algorithms is thoroughly investigated in Koza (1992).

The compression operator for GLiB selects a subtree of the tree representation, removes it from the tree and defines a new function using the extracted subtree as the definition, as shown in the figure. A call to the new function is placed in the tree at the point where the subtree was removed. Any portion of the subtree that extends below a randomly selected depth is clipped and used as a parameters to the newly defined function. If one of the components of the subtree happens to be a call to a compressed function, then it is also compressed. The syntactic atomicity of the function calls in the representation assure that the extracted subtree will not be altered by mutation or crossover during reproduction. Expansion in GLiB searches the tree representation for compressed subtrees and restores its original structure, thus making it mutable again. Angeline and Pollack (1993) describes several experiments using this form of compression and expansion.

There are two benefits to this more complex form of module acquisition. First, each compressed structure becomes a new atomic element of the representational language. Because the compressed elements contain more primitive components, the composite module forms a language element that is at a higher level of abstraction from components which comprise it. These abstractions emerge directly from the interaction of reproduction with the task environment. This allows "successful" hierarchical abstractions to be motivated by constraints in the environment. Furthermore, since each individual can have a unique collection of compressed modules, multiple abstractions for the problem will be explored simultaneously in the population.

The second benefit of atomization of modules arises once a general abstraction is made. In the modular programs evolved in Angeline and Pollack (1993) it was often the case that modules would be copied by crossing over two individuals with the same abstraction. The additional copies of the modules were applied to other related portions of the task. The ability to copy an evolved abstract module and use it for multiple aspects of a problem is a powerful mechanism for an evolutionary algorithm since it takes direct advantage of the decomposability of a problem into easier subproblems.

Given the mutation only reproductive mechanisms of evolutionary programming, it would be nearly impossible for two copies of a particularly useful abstraction to arise within the same individual. In order for evolutionary programs to take advantage of multiple applications of representational abstractions, an operation that copies a compressed component in the individual is required. One version of such a "split" mutation for an FSM representation is depicted in Figure 6. In the figure, a composite state is split into two copies only one of which retains the original incoming connections. This is much like a general "add state" mutation for the FSM representation except the added state is a composite module.

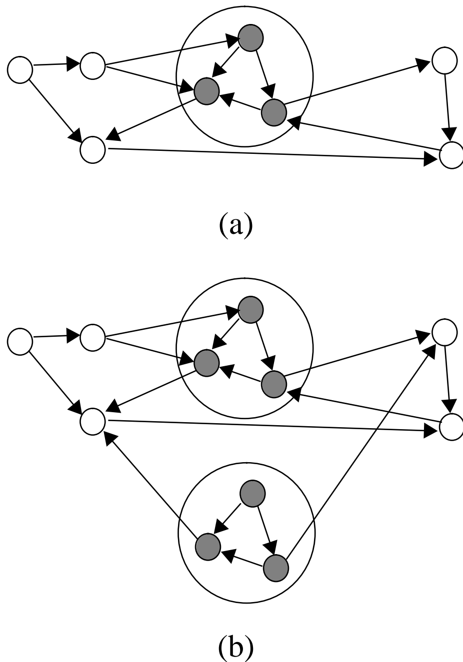

Figure 6: *Illustration of proposed "split" mutation for evolutionary programming. (a) An FSM with a composite module. States and links contained in the composite module are frozen. (b) Result of "split" mutation on composite module. The entire module is copied and the external links are preserved. Links to the new composite module would need to result from subsequent mutations.*

An advantage of the "split" mutation for evolutionary programming applications is that the complexity of an individual can grow to meet the specifications of the problem more quickly. Such growth should be more manipulatable by the evolutionary program than a randomly generated module of the same size since the behavior of the split composite state will generally be immediately exploitable to some degree. In future work, we plan to investigate the implications of the "split" mutation and composite modular representations in evolutionary programs with both the FSM representation and GNARL, an evolutionary program that constructs recurrent neural networks (Angeline et al. 1993).

7. Conclusions

Advanced computational methods are typically based on analytical solutions to a general class of problems. But for a solution to be analytical, it must be devoid of information about specific problems. Hence, their generality prevents them from exploiting problem specific solutions. Conversely, knowledge-based methods encode problem specific approaches that must be recompiled for each new problem, limiting their applicability to domains that have been previously engineered. Both of these approaches are untenable for consistently determining which components of an evolving individual in an evolutionary algorithm are necessary to the survival of subsequent offspring.

The empirical strength of the reproductive process inherent in evolutionary algorithms can serve as a powerful alternative to analytical and knowledge-based methods. Evolutionary module acquisition relies on this strength of evolutionary algorithms to determine problem specific modularizations of developing representations. The modularization of representational components and their protection from mutation can be viewed as removing unnecessary dimension from the search space on the assumption that the component associated with the dimension is set adequately. The dynamics of compressions and expansions described above remove and introduce search dimensions more or less in accordance with the specific development of each individual. Problem specific modularizations of the representation emerge through the interaction of the evolutionary algorithm directly with the problem. This is the purest form of knowledge acquisition.

Other general features of evolving individuals besides modules should also be acquirable by methods similar to those described above. In the future, we hope to demonstrate that many of the methods employed by artificial intelligence can be approximated with similar emergent methods.

8. Acknowledgments

This work was supported by the Office of Naval Research under contract #N00014-92-J-1195. We thank Greg Saunders for feedback and proof reading assistance. We are also indebted to the members of the Laboratory for Artificial Intelligence Research (LAIR) at The Ohio State University for allowing us to usurp their workstations for indeterminate amounts of time. Finally, thanks to David Fogel for his many clarifications on all things EP.

9. References

Angeline, P. (1993) "An analysis of evolutionary algorithms", Submitted to *International Conference on Genetic Algorithms 1993*.

Angeline, P. and Pollack, J. (1993) "Coevolving high-level representations," *Artificial Life III*, Santa Fe Institute Studies in the Sciences of Complexity. To Appear.

Angeline, P., Saunders, G. and Pollack, J. (1993) "An evolutionary algorithm that constructs recurrent neural networks," LAIR Technical Report #93-PA-GNARLY, Submitted to *IEEE Transactions on Neural Networks Special Issue on Evolutionary Programming*.

Bäck, T., Hoffmeister, F. and Schwefel, H.-P. (1991) "A survey of evolution strategies," *Proceedings of the Fourth International Conference on Genetic Algorithms*, R.K. Belew and L.B. Booker (eds.), Morgan Kaufmann Publishers, San Mateo.

Davis, L. (ed.) (1991) *Handbook of Genetic Algorithms*, New York, Van Nostrand Reinhold.

Fogel, D. (1992) *Evolving Artificial Intelligence*, Doctoral dissertation, University of California, San Diego.

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

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

Grefensttete, J. (1989) "Incorporating problem specific knowledge into genetic algorithms", In *Genetic Algorithms and Simulated Annealing*, L. Davis editor, Morgan Kaufman.

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

Jefferson, D., R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. (1992) "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.

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

Michalewicz, Z. (1993) "A hierarchy of evolution programs: an experimental study", *Evolutionary Computation*, **1** (1), To appear March 1993.

Rich, E. (1983) *Artificial Intelligence*, New York: McGraw Hill.