The Evolutionary Induction of Subroutines

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

In this paper[1] we describe a genetic algorithm capable of evolving large programs by exploiting two new genetic operators which construct and deconstruct parameterized subroutines. These subroutines protect useful partial solutions and help to solve the scaling problem for a class of genetic problem solving methods. We demonstrate that our algorithm acquires useful subroutines by evolving a modular program from "scratch" to play and win at Tic-Tac-Toe against a flawed "expert". This work also serves to amplify our previous note (Pollack, 1991) that a phase transition is the principle behind "induction" in dynamical cognitive models.

Introduction

While complex processes of cognition require some form of modularity, learning this modularity has been problematic. It is ignored by simple learning systems (which thus cannot learn complex processes) or built into the architectural "bias" of more complex learning systems (thus begging the origin of such complexity). Thus, the issue of inducing modularity from a complex task in order to perform that task has not been addressed, although a few connectionists are beginning this research (Saunders et al., 1992; Jacobs & Jordan, 1991; Jacobs et al., 1991; Nowlan & Hinton, 1991).

In this paper we describe a genetic algorithm which is capable of evolving large programs by exploiting two new genetic operators which construct and deconstruct parameterized subroutines. These subroutines protect useful partial solutions and help to solve the scaling problem for a class of genetic problem solving methods. After a brief background on genetic algorithms, we show that our system is able to learn how to play and win at Tic-Tac-Toe from "scratch" against an imperfect "expert" player. We discuss the formation and tuning of the subroutines and the reasons why their acquisition addresses the scaling problem within this framework. An analysis of the frequency of subroutine calls shows an exponential growth and decay of subroutine usage as they are induced or expelled from the language, leading us to name this phenomenon *evolutionary induction*, an amplification of our earlier principle of "induction by phase transition" (Pollack, 1991).

1. This research is supported by the Office of Navel Research under contract #N00014-89-J1200.

Genetic Algorithms Background

The genetic algorithm (Holland, 1975; Goldberg 1989a) is a form of problem solving search analogous to natural selection, and is a surprisingly adept search method in even very large ill-formed problem spaces. A simple genetic algorithm typically operates by reproducing and altering a population of fixed-length binary strings. A *fitness function* interprets the strings as task solutions and scores their ability to solve the task. Novel strings are added to the population by a process akin to biological reproduction using a collection of genetic *operators*. One such operator, the *crossover* operator, takes two "parent" strings selected for their fitness and returns a "child" string which is a complementary collection of components from both parents. The *point mutation* operator alters the value of a single position of a single parent string to create offspring.

The schema theorem (Holland, 1975), often called the Fundamental Theorem of Genetic Algorithms, illustrates the power behind these search methods. Holland defines a *schema* to be a class of binary strings which share a collection of subsequences. We use the "#" notation to indicate "don't care" positions in the schema, i.e. positions where a 1 or 0 don't matter. For instance, the string '100101' is a member of the schema '10##01' as is '100001'. The intuition behind schemata is that certain combinations of bits will have a larger contribution to the fitness for a particular string than others. The schema notation allows us to talk about such desirable organizations concisely. A schema's *defining length* is the number of positions at which if we divided the schema into two parts, some of the defined positions (i.e. ones not '#') would be separated. For instance, dividing the schema '#1.0.0##' at any position marked with a '.' will separate some of the defined components giving it a defining length of 2. Similarly, the defining length of '1.0.#.#.0.1' is 5. The schema theorem proves that above average schemata with small defining lengths will be copied an exponential number of times in the generations subsequent to their appearance (Holland, 1975). For a more detailed introduction to genetic algorithms, the schema theorem and its implications see (Goldberg, 1989a).

While simple genetic algorithms can be used to evolve solutions to a wide range of tasks two problems prevent their scaling to more interesting tasks. The first is an inherent limitation due to the fixed-length nature of the string representation. Because of the closed nature of the representation, the maximum complexity needed to solve the task must be anticipated before the search takes place. The second problem is that each bit of the string representation generally stands for the presence or absence of a specific feature of the interpretation. This positional encoding of the binary representation requires that every possible interpretation also be anticipated prior to the search. This amounts to nothing more than using the string as a pointer to a table of pre-defined interpretations.

In order to increase the amount of available complexity in simple genetic algorithms, some researchers have devised elaborate interpretation routines (See Belew, McInerney & Schraudolph, 1992 and Dawkins, 1987 for example.) Essentially, this approach removes the complexity from the jurisdiction of the representation and places it into the interpretation. Unfortunately, rather than address the representation of complexity problem in simple genetic algorithms this approach merely shifts the problem to a new component. By placing an undue amount of design into the interpretation of the representation these researchers beg the question of evolving complexity since they have provided the complexity a priori.

Recently, Koza has described an exciting advance in genetic algorithms. In his Genetic Programming Paradigm (GPP), Koza uses a hierarchy of primitive functions rather than a fixed-length string to represent potential solutions (Koza, 1992, Koza, 1990). These hierarchies are interpreted as programs written in a language defined by the primitive functions which when executed compute the solution to the task. Koza's genetic operators exchange subtrees of the hierarchies rather than substrings.

Although Koza's dynamic representation alleviates both the fixed-length and positional encoding limitations of simple genetic algorithms, it also suffers from a malady which prevents its scaling. Consider that a dynamic representation will eventually grow large enough to encompass the complexity necessary to solve the desired problem. At some point in the learning of a very complex task, the structure will be quite large and the chance of breaking up desirable portions of the program with the crossover operator will overwhelm the chance of improving the program. In other words, as the defining length of a desirable schema increases it becomes more likely that we will consistently break it apart rather than improve upon it. We call this the *defining length problem*.[2] As an empirical indication of this problem, we note that the largest evolved program Koza reports is only 48 nodes.

These scaling difficulties call to mind Simon's parable of the two watchmakers Tempus and Hora (Simon, 1969). 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, much like GPP constructs programs. Consequently, if he is interrupted before completing a watch, say by a customer calling on the phone, the intermediate state is lost and he 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 currently being constructed is lost. The lesson from Simon's parable is clear, that in the development of complex systems it is prudent to build incrementally and modularly.

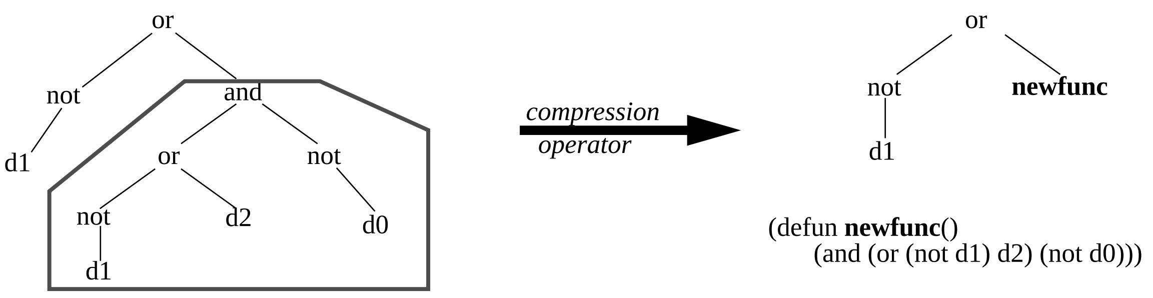
The Genetic Library Builder (GLiB)

The Genetic Library Builder (GLiB) is a genetic algorithm environment based on the ideas forged by Koza in GPP but with provisions for the evolution and evaluation of program subroutines. As in GPP, GLiB uses an expression tree of primitive functions as its representation for potential solution programs. The essential difference between GPP and GLiB is the addition of two new genetic operators. The first operator, called *compression*, creates subroutines from subtrees of individuals in the current population and introduces the subroutines into the "genetic library". This library is simply the collection of subroutines which appear in the programs of population and thus are available for constructing task solutions. Once in the library, the usefulness of a newly constructed subroutine is evaluated by the extent it is used in future generations. The second operator, called *expansion*, replaces compressed subroutines with their original definition. In the following sections we describe these operators and their implementations in detail.

Creation of Subroutines in GLiB

The compression operator in GLiB, the sole method of subroutine definition in the system, works as follows. During the construction of each new generation of pro-

2. Because they exploit positional encodings, string-based genetic algorithms do not suffer from this problem.



```
(defun newfunc()
      (and (or (not d1) d2) (not d0)))
```

Figure 1*: Creation of a new subroutine from a randomly selected subtree of an individual in the current population.*

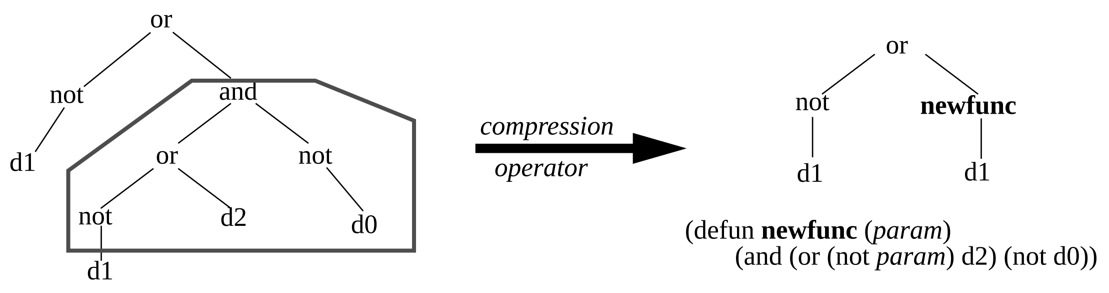
grams, the compression operator is applied to a percentage of the population selected by relative fitness. The compression operator is asexual, like the point mutation operator described above, so only a single "parent" program is selected and copied. The copy serves as the "child" of this parent in the coming generation. A node in the interior of the child's expression tree is then randomly selected and designated the root of the subtree which will become the newly compressed subroutine. Next, a maximum depth for the subtree is randomly selected from a user defined range. When none of the branches of the subtree exceeds this maximum depth we have the instance of subroutine creation depicted in Figure1. Here, the entire subtree is removed from the offspring and used as the body of a new LISP function definition with no parameters. Once the new subroutine is defined, the expression tree of the offspring is altered replacing the extracted subtree by the equivalent LISP function call. This compression of the subtree into the name of the equivalent subroutine call introduces the new subroutine into the genetic library.

Occasionally, some branches of the selected subtree will have a depth greater than the allowed maximum depth for the subroutine being created. In this event, we replace each branch of the subtree at the point where it exceeds the maximum depth with a unique variable. When the LISP function is defined, the variables introduced into the subtree are used as parameters to the new subroutines. When we then compress the expression tree of the child, the portions of the subtree which exceeded the maximum depth are not removed but serve as the values for the parameters in the subroutine call. This instance of modularization in GLiB is depicted in Figure. 2. *Note that invariably when a compression takes place the semantics of the program are not altered, only the manner in which the program is expressed.*

Unfortunately, while the compression operator supplies a method to create subroutines from the population during GLiB's genetic search, it also serves to remove unique subtrees from the population, lowering the diversity of the population. For a genetic search to work, there must be sufficient genetic material in the population so that combinations of promising candidates from the current generation can be recombined into novel organizations. By lowering the diversity of the population and consequently the number of novel combinations, we limit the distance from the current state that a genetic search can look.

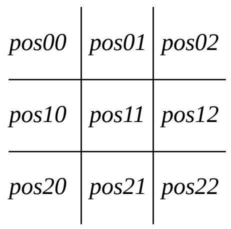
In order to balance the undesirable effects of the compression operator, we have also added an expansion operator which restores the genetic material from the compressed subtrees. This operator searches the offspring's expression tree for a call to an evolved subroutine. If one is found, it is expanded from its atomic reference back into the full subtree and thus replaces the genetic material previously removed.

The complementary nature of the compression and expansion operators implements a form of iterative refinement. The random selection of a subtree for compression provides no guarantee that the selected subtree



```
(defun newfunc (param)
      (and (or (not param) d2) (not d0))
```

Figure 2*: Creation of a new subroutine with parameters replacing branches which are beyond maximum allowed depth.*



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

Figure 3*: Primitives used in evolving modular programs to play Tic Tac Toe.*

will be an above average schema. It is more likely that it will be either a portion of a useful schema or simply of no import at all. By periodically replacing a copy of the compressed subtree back into the population, we provide the chance to capture a better version of the schema at a latter time.

Evaluation of Subroutine Performance

Now that we have a method of extracting potentially useful subroutines from the evolved programs, we need a method for evaluating their contribution. We suggest that an appropriate measure of success for a particular evolved subroutine should be the number of times it is put into use by the population in the course of solving the task. If many members of the population are using the subroutine at some point in the genetic search, then it is likely that the subroutine provided some consistent advantage in earlier generations. When this occurs, we say that the subroutine is *evolutionarily viable*.

Our task now is to insure that good subroutines will be copied generously into subsequent generations while inappropriate ones will be suppressed. The “enlightening” guidelines provided for genetic algorithm design in (Goldberg, 1989b) suggest one should never be too clever when dealing with genetic algorithms as a “frontal assault” to the solution of a design problem usually defeats the inherent non-linear interactions. Thus, one should practice prudence when possible.

Appropriately enough, the genetic search which evolves programs to solve the task, automatically evaluates the worth of the subroutines without any additional intervention. The logic of this is straightforward. Initially, when a new subroutine is created there is only one member of the population which has a reference to it. If this program is comparatively fit, then, by the schema theorem, the call to the subroutine will be copied into several offspring in the next generation. If those individuals are also relatively fit then each of them will have multiple offspring which contain the subroutine call as well. Eventually, the subroutine will spread throughout a significant portion of the population. On the other hand, if the program is comparatively unfit, possibly due to one of its subroutines being more of a hinderance than a help, it will have little or no chance to create offspring. This results in a decrease in the number of calls to the subroutine from generation to generation until there is virtually no member of the population which relies upon it. In other words, if a subroutine presents no advantage to the individuals which use it, it will in time go the way of the human appendix. Once the subroutine is no longer used by the population, it is no longer in the genetic library. Thus the genetic search process at the level of the overall task implicitly determines the fitness of evolved subroutines and allows only those that are useful to be propagated.

Learning to Play Tic-Tac-Toe

In order to illustrate our form of subroutine acquisition at work, we used GLiB to evolve programs to play Tic-Tac-Toe (TTT). The primitive language used for this experiment is shown in Figure 3. The first collection of primitives, ***pos00*** to ***pos22***, are the data points to be used in the program which represent the nine positions on the TTT board. This set of data points serves as the leaves of the expression tree. For the remaining primitives, the return value is either one of the positions or NIL, which represents FALSE in LISP, the current language in which GLiB is implemented. For instance, the binary ***and*** operator takes two arguments and when both are non-NIL returns the second. If either argument is NIL then NIL is returned. 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 their turn is halted. Otherwise, ***play-at*** returns whatever it is passed. Finally, the operators ***mine***, ***yours*** and ***open*** take a position and return that position when the mark on the playing board in that position fits the test. Otherwise, they return NIL. We have purposely made these functions as general as possible to cover any number of games rather than just TTT. *Note 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. We consider legal moves to be apart of the complexity of the task and consequently should be induced by GLiB.

An “expert” TTT algorithm constructed in LISP served as the opponent for all of the evolved programs. This expert was designed in such a way that it could not lose a game unless the opponent it was playing against had forked it, i.e. created a situation where the addition of a single mark by the opponent resulted in more than

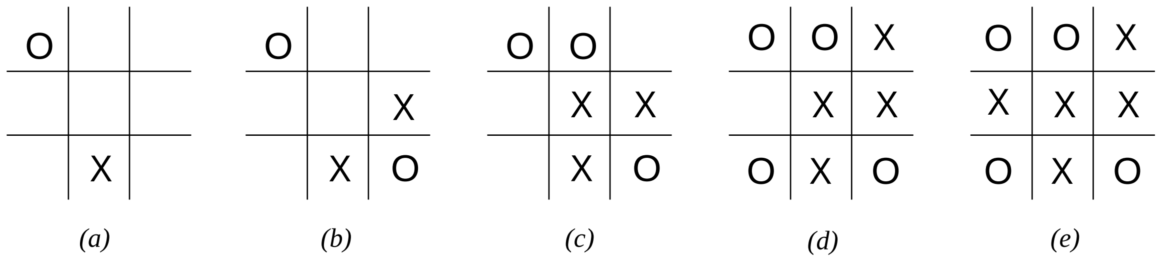


Figure 4: *A sample game from the described run. The evolved program is "X" and is playing first against the expert. Opening moves in (a) lead to first fork setup in (b). Evolved program completes first fork in (c). Evolved program sets up second fork in (d) and wins in (e). The program received a score of 20 points for this win.*

one possible winning play on its next turn. In addition, the expert was slightly adaptive such that when no clear best move was available it would select a position known to be frequented by the program it was currently playing against. The intention of this feature was to increase the apparent complexity of the expert's actions forcing the evolved programs to be more robust. The generality of the primitives combined with the level of play of the expert make this quite a formidable environment for learning TTT.

In order to rate the performance of an evolved program against the expert, a scoring function assigned points for various moves. First, because it is not a given that a program will actually make a legal play, a point was awarded for every legal move made. An additional point was awarded if a move blocked the expert from winning on its next turn. If the game ended in a draw or a win, the accumulated score of the evolved program was increased by 4 or 12 points respectively. It is important to note that the score for the program was a lump sum and provided no indication of which actions were being rewarded. The same results would be achieved if only the final state of the board were scored rather than the individual moves.

We ran GLiB with a population size of 1000 using the described expert and scoring method as the fitness function. In this run we applied the compression operator to 10 percent of the population each generation. All other parameters were as set in (Koza, 1990) for the "ant" experiment. The best evolved program after 200 generations had an average score of 16.5 points for the 4 games it played against the expert to determine its fitness. This score suggests that while the program was able to beat the flawed expert more than once, the best it could do after the expert had adapted to its playing strategy was to get a draw.

The evolved program had 60 nodes, a maximum depth of 13 nodes, and used 15 evolved subroutines at its top-level. As expected, expanding the definition of these subroutines back into their original subtrees revealed additional subroutine calls in their bodies. In all a total of 43 distinct subroutines were used by this evolved program in 89 subroutine calls making the virtual size of the program 477 nodes with a virtual maximum depth of 39. Two of the subroutines had a total of 9 separate calls each in the fully expanded tree. Note that this evolved program is almost 10 times the size of the largest program reported by Koza.

Figure 4 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 was able to establish a fork by its third move (Figure 4c) but did not win the game until 2 turns later (Figure 4e). While this seems an odd strategy, recall that the evolved program gets points for each move it makes and additional points if it blocks the expert. Its strategy, then, is to maximize its total point score by forking the expert not once but *twice* in the same game! If the program had won the game on its fourth move it would have received 3 less points. By extending the game it actually increases its score without the possibility of losing.

The analysis of the created subroutines is equally interesting. Overall in the run there were 16,852 subroutines created by the compression operator with only 257 in use during the final generation. Figure 5 shows the number of calls per generation for three of the evolved subroutines. Each of these has a distinct period during the run where its number of calls per generation rises extremely quickly. In Figure 5a the sharp increase happens relatively soon after the subroutine is defined, showing it posed an immediate advantage. Figure 5b shows an example of a subroutine which was extremely useful shortly after its creation but whose use fell off dramatically. Finally, Figure 5c shows a subroutine which was present in the population for almost 100 generations before being recognized as being useful.

Discussion

The dramatic shapes of the calls per generation curves for the subroutines shown in Figure 5 are interesting for two reasons. First, it is apparent that we have been able to capture useful schemata in our subroutines by the exponential-like rises in the subroutine call curves. Second this work amplifies our previous note (Pollack, 1991) that a phase transition is the principle behind

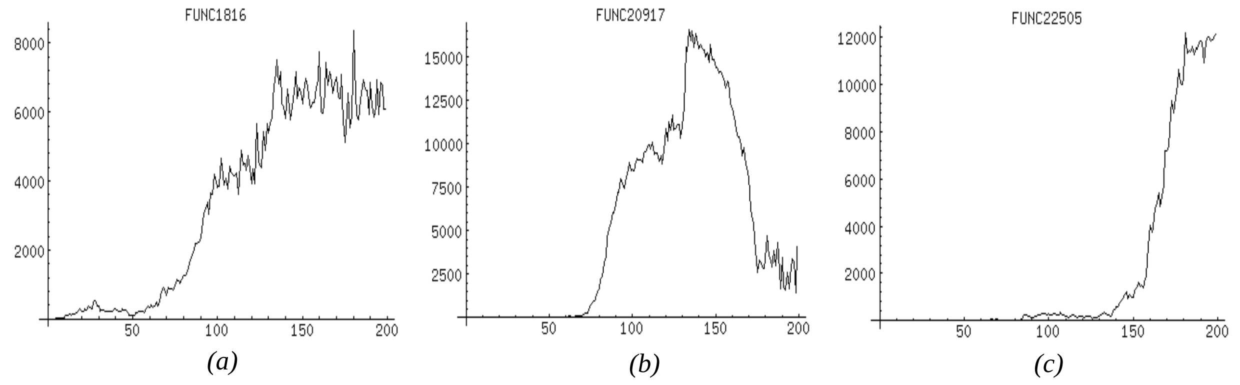


Figure 5: *Graphs showing number of calls (y-axis) per generation (x-axis) for three evolved subroutines. See text for explanation. Note graphs are not equally scaled.*

"induction" in dynamical cognitive models. We call this method of random selection and evolutionary evaluation of subroutines *evolutionary induction*.

But there is more to the story than a simple attachment to Holland's powerful theorem. We also claim that evolutionary induction impacts the defining length problem for dynamic genetic algorithms, although we have no formal verification of this claim. Our reasons are as follows: By compressing random subtrees from the program into representational atoms, we literally reduce the defining length of that subtree to zero. Because we know by the schema theorem that above average schemata will be copied more readily than below average schemata, our chances of compressing a useful schema increases each generation. Once we compress a useful schema reducing its defining length to zero, it can be used to create more complex structures which are still small enough to be propagated intact to future generations. The end result is a complex modular program with nested subroutine calls and an overall structure similar to Hora's watches.

Acknowledgments

Thanks to Greg Saunders, John Kolen and Dave Stucki for reading and commenting on various drafts of this paper.

References

Belew, R., McInerney, J. and Schraudolph, N., 1991, "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning", In *Artificial Life II*, C. Langton, C. Taylor, J. D. Farmer and S. Rasmussen (eds.), Reading, MA: Addison-Wesley Publishing Company, Inc.

Dawkins R., 1987, *The Blind Watchmaker*, New York, W. W. Norton and Co.

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

Goldberg, D., 1989b, "Zen and the Art of Genetic Algorithms", In *Proceedings of the Third International Conference on Genetic Algorithms*, J. Schaffer (ed), Los Altos, CA: Morgan Kaufmann Publishers, Inc.

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

Jacobs, R., Jordan, M., Nowlan, S., & Hinton, G., 1991, "Adaptive Mixtures of Local Experts", *Neural Computation*, 3, 79 - 87.

Jacobs, S., & Jordan, M., 1991, "A Competitive Modular Connectionist Architecture", In *Advances in Neural Information Processing 3*, R. Lippmann, J. Moody, D. Touretzky (eds.), San Mateo CA: Morgan Kaufmann Publications, Inc.

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

Koza, J., 1990, "Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems", Technical Report No. STAN-CS-90-1314, Computer Science Department, Stanford University.

Nowlan, S., & Hinton, G., 1991, "Evaluation of Adaptive Mixtures of Competing Experts", In *Advances in Neural Information Processing 3*, R. Lippmann, J. Moody, D. Touretzky (eds.), San Mateo CA: Morgan Kaufmann Publications, Inc.

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

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

Simon, H. A., 1969, *The Sciences of the Artificial*, Cambridge, MA: MIT Press.