Automatic Generation of Adaptive Programs
by Lee Spector and Kilian Stoffel

Full citation:
Spector, L., and K. Stoffel. 1996. Automatic Generation of Adaptive Programs. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB-96).* P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S.W. Wilson (editors). Cambridge, MA: The MIT Press.

Automatic Generation of Adaptive Programs

Lee Spector*†
lspector@hampshire.edu

*School of Cognitive Science and Cultural Studies
Hampshire College
Amherst, MA 01002

Kilian Stoffel †
stoffel@cs.umd.edu

†Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

This paper shows how *ontogenetic programming*, an enhancement to the genetic programming methodology, allows for the automatic generation of adaptive programs. Programs produced by ontogenetic programming may include calls to self-modification operators. By permitting runtime program self-modification, these operators allow evolved programs to further adapt to their environments. In this paper the ontogenetic programming methodology is described and two examples of its use are presented, one for binary sequence prediction and the other for action selection in a virtual world. In both cases the inclusion of self-modification operators has a clear positive impact on the ability of genetic programming to produce successful programs.

1 Introduction

Genetic programming is a methodology for the automatic generation of computer programs by means of biologically-inspired processes of recombination and natural selection (Koza, 1992). Genetic programming systems process populations of computer programs. The initial populations typically consist of programs that are random combinations of elements from problem-specific function and terminal sets. The programs are assessed for fitness, usually by running them on representative sets of inputs, and the resulting fitness values are used in producing the next generation of programs. A variety of genetic operations, including fitness-proportionate reproduction, crossover, and mutation may be employed in constructing programs for the next generation. After a preestablished number of generations, or after the best fitness improves to some preestablished level, the best-of-run individual is designated as the result and is produced as the output from the genetic programming system.

Genetic programming systems are adaptive insofar as they support the adaptation of a population of programs to a particular fitness test throughout the course of evolution. In many problem environments it may also be advantageous for the evolved programs to *themselves* be adaptive. For example, it may be useful for evolved programs to perform on-line learning about complex environments, to adjust their own parameters as dynamic environments change over time, or even to adopt entirely new algorithms on the basis of environmental input. These sorts of runtime adaptive capabilities can be critical for achieving good performance in many real-world environments.

There are two obvious ways in which genetic programming systems could support runtime adaptation. The first involves the runtime manipulation of dynamic data structures upon which program execution may depend. Indexed memory (Teller, 1994) and memory terminals (Iba et al., 1995) both provide the required functionality; programs that use these mechanisms may acquire and store information from their environments at runtime, and they may use this information to guide future behavior.

A second, more direct mechanism is described in this paper: program self-modification operators are included in the set of functions that may be used by evolved programs. By use of these operators, evolved programs can dynamically change their own structure—and thereby their future behavior—during the course of a run. A program's self-modification strategy is itself evolved; it may be arbitrarily complex and it may be conditionalized on runtime environmental inputs.

Biologists refer to the developmental progression of an individual through its life span as *ontogeny*. The inclusion of program self-modification operators in a genetic programming function set allows for the evolutionary production of programs with rich ontogenetic components. For this reason, the technique is called *ontogenetic programming*, and the program self-modification operators are called *ontogenetic operators*.

Some other evolutionary computation frameworks already allow for runtime adaptation. This is generally

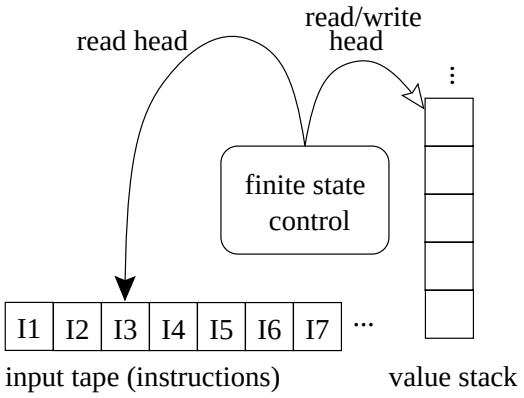

Figure 1: The HiGP virtual stack machine

accomplished by using an underlying adaptive mechanism that is only partially pre-configured by evolutionary processes; additional configuration and adaptation is accomplished by the underlying adaptive mechanism as it confronts its environment. For example, genetic algorithms have been used to evolve the initial parameter vectors for dynamical neural networks that integrate runtime sensor information to modify their own future behavior (Yamauchi and Beer, 1994). In these sorts of systems the runtime adaptive mechanisms (e.g., learning procedures for neural networks) are fixed; by contrast, ontogenetic programming allows for the simultaneous evolution of runtime adaptive mechanisms and the programs that use them.

The remainder of this paper demonstrates the automatic generation of adaptive programs using ontogenetic programming. The HiGP genetic programming system, which operates on linear programs for a stack-based virtual machine, is used for this demonstration. Program modification mechanisms for linear programs are particularly simple, but the technique is not limited to systems with linear programs; see (Spector and Stoffel, 1996) for a discussion of ontogenetic programming with more traditional S-expression-based representations. After the description of the technique two examples are provided: a simple but illustrative binary sequence prediction problem, and a more complex action selection problem. The paper concludes with a brief description of work yet to be done.

2 Ontogenetic HiGP

HiGP is a high-performance genetic programming engine that combines techniques from string-based genetic algorithms, S-expression-based genetic programming systems, and high-performance parallel computing systems (Stoffel and Spector, 1996). It is a fast, flexible, and portable system with an efficient parallel implementation that scales nearly linearly with the number of available processors. HiGP produces and manipulates linear programs for a stack-based virtual machine (as in (Perkis, 1994)), rather than the tree-structured S-expressions used in traditional genetic programming.

The HiGP virtual machine is similar to standard pushdown automata models (see, e.g., (Kain, 1972)) and to the execution model of the FORTH programming language (Brodie, 1981). It consists of an input tape containing a linear program to be executed, a pushdown stack for function arguments and return values, and a finite-state control unit that controls the execution of the program (see Figure 1). The program on the input tape must be composed of words that have been pre-defined as HiGP operators. To execute a program, the finite-state control unit simply reads the program from the input tape and executes the defined function call for each operator. Operators may be defined to perform arbitrary computations and to manipulate the values on the stack. They may also reposition the read head on the input tape, thereby allowing for conditional branch operators and loops. The stack is usually initially empty, although for some applications it may be pre-loaded with inputs for the program. Program execution halts when the last operator on the input tape has been processed, or when a pre-established maximum number of operators have been processed. Program results are read from the top of the stack at the end of program execution. Note that while HiGP evolves programs for the virtual machine, the machine itself never changes—it is simply the execution model for the evolving programs, much as a Lisp-like S-expression evaluator is the execution model for standard genetic programming (Koza, 1992). Further details about HiGP and the HiGP virtual machine can be found in (Stoffel and Spector, 1996).

One normally provides a family of `push` operators that correspond to the terminal set in a traditional genetic programming system; each `push` operator pushes a pre-determined value onto the stack. Because all programs in the system have the same length, and because we do not wish to pre-determine the number of actual problem-solving operators that should appear in solution programs, a `noop` operator is also usually included. With the inclusion of the `noop` operator, which does nothing, the fixed program size becomes a size *limit*, analogous to the depth limits used in S-expression-based genetic programming systems. In other words, "shorter" programs can be encoded by filling in extra program steps with `noop`s.

Additional operators may be easily added to the system. The only restrictions are that they must take their arguments from the stack and that they must push their results back onto the stack. When there are not enough values on the stack to serve as arguments for an opera-

tor it is skipped by the finite-state control unit and the stack remains untouched (as in (Perkis, 1994)).

A simple example may help to clarify the operation of the virtual stack machine. Consider the following program:

```
push-x noop push-y * push-x push-z noop - +
noop noop
```

The `noop`s in the program have no effect and the remainder is equivalent to the C expression:

```
(x * y) + (x - z)
```

The ontogenetic version of HiGP results from adding the following program self-modification operators to the function set:

segment-copy copies a part of the program over another part of the program. The function takes three arguments from the stack, all of which are coerced to integers in the appropriate ranges: the start position of the segment to copy (relative to the current instruction), the length of the segment, and the position to which the segment should be copied (relative to the current instruction). If there are not three values on the stack the instruction is skipped.

shift-left rotates the program to the left. The call takes one argument from the stack, which is coerced to a positive integer if necessary: the distance by which the program is to be rotated. If there is no value on the stack the program is rotated by one instruction to the left.

shift-right rotates the program to the right. The call takes one argument from the stack, which is coerced to a positive integer if necessary: the distance by which the program is to be rotated. If there is no value on the stack the program is rotated by one instruction to the right.

The position of the current instruction pointer is not changed by the execution of an ontogenetic operator. For example, if instruction #23 is a `segment-copy`, then after its execution instruction #24 will be executed, regardless of the fact that the *old* instructions #23 and #24 may now have been moved elsewhere.

3 Example 1: Binary Sequence Prediction

The ontogenetic version of HiGP can solve problems that cannot be solved by the ordinary version of HiGP. This section demonstrates the application of ontogenetic programming to a binary sequence prediction problem for which this is the case. Because indexed memory also provides a runtime adaptive capability, one might conjecture that the use of indexed memory would be sufficient for the solution of this problem. But this section shows that this is not the case; the addition of indexed memory to the ordinary version of HiGP is not sufficient to produce solutions to the described problem. In other words, the ontogenetic extensions provide benefits that indexed memory does not.

Consider the *sequential regression* problem, a variant of the symbolic regression problem (Koza, 1992). As in ordinary symbolic regression, the goal in sequential regression is to produce a program that returns the appropriate y value for each x value in a data set. In the sequential regression problem it is additionally stipulated that the x values will be presented in a particular order. Programs are run multiple times, one for each x value in the fitness-testing range, and they always encounter these x values in the same order. Runtime adaptation during the course of a program's progression through the x range may be useful for sequential regression problems because different programs may be most appropriate for different ranges of the target function. Further, transitions from one value to the next may be better mediated by ontogenetic operators than by domain operators.

Consider a binary sequential regression problem with a target function that repeats the values [0 1 0 0 0 1] as x increases. We will call this problem the *binary sequence prediction problem*. The goal is to produce a program that returns the appropriate y value from the sequence for each x (index) value; the program will be run once for each x value, and the x values will be presented in order from 0 to some number n. Several related sequence prediction problems have been described in the literature. In particular, Iba et al. describe a related binary oscillation task (Iba et al., 1995).

For our experiments we assessed fitness by testing each program on the range [0–17]. Positive return values were mapped to 1, and negative return values were mapped to 0; it was therefore sufficient for a program to return either 0 or any negative number in place of each 0, and any positive number in place of each 1. Fitness was calculated as the number of incorrect answers; lower fitness values therefore indicated better programs, and a fitness value of 0 indicated a completely correct program. We used a function set consisting of the 2-argument addition function `+`, the 2-argument subtraction function `-`, the 2-argument multiplication function `*`, the 2-argument protected division function `%` (Koza, 1992), and the 0-argument `push-x` function for the independent variable x.[1] In addition, we included the stack

[1] In previous experiments we also included `push-0` and `push-1` in the function set (Spector and Stoffel, 1996). Surprisingly, their removal had almost no effect on the results.

manipulation function `dup`, which pushes a duplicate of the top element onto the stack, and `noop`. A copy of x was pre-loaded onto the stack prior to each program execution.

100 runs of the ordinary (non-ontogenetic) version of HiGP were conducted on this problem with a population size of 100, a maximum program size of 30, a crossover rate of 90%, a reproduction rate of 10%, and a maximum of 20 generations per run. No correct solutions were produced by any of the 100 runs. One can conclude that the problem cannot reliably be solved by ordinary HiGP and the given parameters. Small numbers of runs were also conducted with varied parameters (for example, with mutation and with larger populations), but no correct solutions were ever produced.

100 runs of the ontogenetic version of HiGP were then conducted on this problem (adding the `segment-copy`, `shift-right` and `shift-left` functions to the function set). The population size and other parameters were the same as those described above. 12 completely correct solutions were produced by the 100 runs. 10 of these solutions appeared to be general; although fitness was assessed only over the range [0–17], these programs produced correct results over the range [0–39], and they appeared to be correct to any limit. The following is a correct evolved program:

```
+ push-x - noop - shift-left shift-right
push-x segment-copy shift-left shift-right
* + segment-copy + % * * noop noop
shift-right + % shift-left shift-right
noop noop - noop *
```

Because the ontogenetic operators modify programs *as they run*, it is difficult to trace program execution or to get an intuitive feel for program self-transformation strategies. Nonetheless, it is sometimes interesting to look at some of forms through which a program passes. The above program looks as follows after the completion of 9 full executions:

```
- shift-left + % * * noop noop shift-right %
shift-left noop noop - noop * + push-x - noop
- shift-left + % * * noop noop shift-right %
```

At the end of 18 executions it appears more similar to, but still different from, its initial state:

```
+ push-x - noop - shift-left shift-right
segment-copy shift-left * + segment-copy + %
* * noop noop shift-right % shift-left noop
noop - noop * + push-x - noop
```

Although equivalent programs do recur through the 18 executions of the fitness-test range, no obvious alignment of these recurrences to the 6-element sequence value pattern is evident.

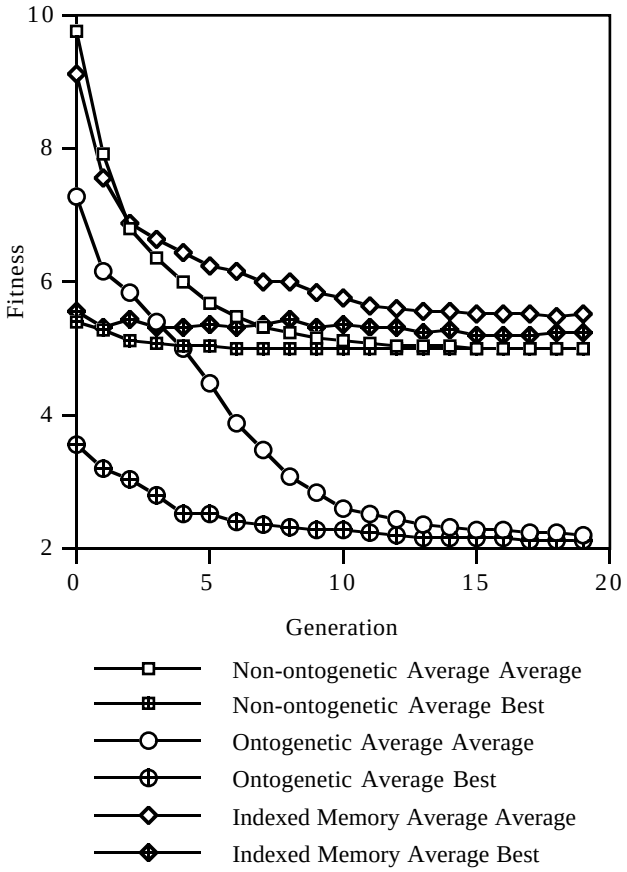

Figure 2: Average and best fitnesses by generation, averaged over the 100 runs for each condition, for the binary sequence prediction problem.

Because indexed memory also provides a limited runtime adaptive capability, 100 runs of HiGP with indexed memory were also conducted on this problem. A 30-element indexed memory was added, with a 2-argument `write` function that stores the value of one argument in the location indexed by the other, and a 1-argument `read` function that pushes the indexed element of the memory onto the stack. The population size and other parameters were the same as those described above. No correct solutions were produced by any of the 100 runs. One can conclude that indexed memory does not provide sufficient runtime adaptive power for HiGP to reliably solve this problem with the given parameters.

Figure 2 shows graphs of average and best fitnesses by generation, averaged over the 100 runs conducted for each condition (non-ontogenetic, ontogenetic, and indexed memory). From this graph it is clear that the advantage provided by the ontogenetic operators is indeed significant. Note that in all conditions the average average fitness appears to converge to a value close to

the average best fitness. Note also that indexed memory actually makes matters worse; the system performs better without it.

4 Example 2: Action Selection in Wumpus World

Wumpus world (Russell and Norvig, 1995) is an environment in which an agent must select actions to avoid death and to achieve goals. The use of genetic programming for the evolution of Wumpus world agents has been previously described (Spector, 1996). In this section we briefly describe the Wumpus world problem and the results of experiments that use ontogenetic programming to produce Wumpus world agents.

Wumpus world is cave represented as a grid of squares surrounded by walls. The agent's task is to start in a particular square, to move through the world to find and to pick up a piece of gold, to return to the start square, and to climb out of the cave. The cave is also inhabited by a "wumpus"—a beast that will eat anyone who enters its square. The wumpus produces a stench that can be perceived by the agent from adjacent (but not diagonal) squares. The agent has a single arrow that can be used to kill the wumpus. When hit by the arrow the wumpus screams; this can be heard anywhere in the cave. The wumpus still produces a stench when dead, but it is harmless. The cave also contains bottomless pits that will trap unwary agents. Pits produce breezes that can be felt in adjacent (but not diagonal) squares. The agent perceives a bump when it walks into a wall, and a glitter when it is in the same square as the gold.

The wumpus world agent can perform only the following actions in the world: go forward one square; turn left 90°; turn right 90°; grab an object (e.g., the gold) if it is in the same square as the agent; release a grabbed object; shoot the arrow in the direction in which the agent is facing; climb out of the cave if the agent is in the start square.

The agent's program is invoked to select a single action for each time-step of the simulation. The program returns one of the valid actions and the simulator then causes that action, and any secondary effects, to happen in the world. The agent can maintain information between actions by use of a persistent memory system. The agent's program has a single parameter, a "percept" that encodes all of the sensory information available to the agent. The agent's program can refer to the components of the percept arbitrarily many times during its execution.

Agents are assessed on the basis of performance in four worlds.[2] In each world the agent is allowed to perform a maximum of 50 actions, and the agent's score is determined as follows: 100 points are awarded for obtaining the gold, there is a 1-point penalty for each action taken, there is a 100-point penalty for getting killed, and there is a 100-point penalty for each unit of distance between the agent and the gold at the end of the run. Agents are not explicitly rewarded for climbing out of the cave, although less action penalties are accumulated if an agent climbs out and thereby ends the simulation. An agent is considered to have solved the problem if its average score in four worlds is greater than zero. To have obtained such a score an agent must

Table 1: Wumpus World Operators

Name	Args	Description
`+`	2	Pushes the sum of the arguments modulo 7 onto the stack.
`-`	2	Pushes the difference of the arguments modulo 7.
`*`	2	Pushes the product of the arguments modulo 7.
`and`	2	Pushes 1 if both arguments are non-zero, or 0 otherwise.
`or`	2	Pushes 1 if either or both arguments are non-zero, or 0 otherwise.
`not`	1	Pushes 1 if the argument is 0, or 0 otherwise.
`ifz`	3	"If zero"—jumps forward the number of instructions specified in the second argument if the first argument is 0; jumps forward the number of instructions specified in the third argument otherwise.
`read`	2	Pushes the contents of the 2-dimensional indexed memory location indexed by the arguments. Memory locations contain 0 if they have not yet been written to.
`write`	3	Pushes the *previous* contents of the memory location indexed by the first two arguments, and then fills the memory location with the third argument.
`noop`	0	Does nothing.
`{0 - 6}`	0	Each pushes the corresponding number.
`rand7`	0	Pushes a random integer between zero and six (inclusive).
`stench`	0	Pushes 1 if the current percept includes a stench (from the wumpus), or 0 otherwise.
`breeze`	0	Pushes 1 if the current percept includes a breeze (from a pit), or 0 otherwise.
`glitter`	0	Pushes 1 if the current percept includes a glitter (from the gold), or 0 otherwise.
`bump`	0	Pushes 1 if the current percept includes a bump (from a wall), or 0 otherwise.
`sound`	0	Pushes 1 if the current percept includes a sound (from the wumpus), or 0 otherwise.

[2] In (Spector, 1996) four new random worlds were generated for each fitness test. This allowed very simple programs that were fortunate enough to be tested on appropriately simple worlds to appear to be quite successful. In the experiments described in this paper four random worlds were generated for each run of the genetic programming system, and the same four worlds were used for each fitness test. This appears to make the problem considerably more difficult, although it is possible that the resulting agents will be overfitted to the particular four worlds and therefore less robust than some of those produced by the previous technique.

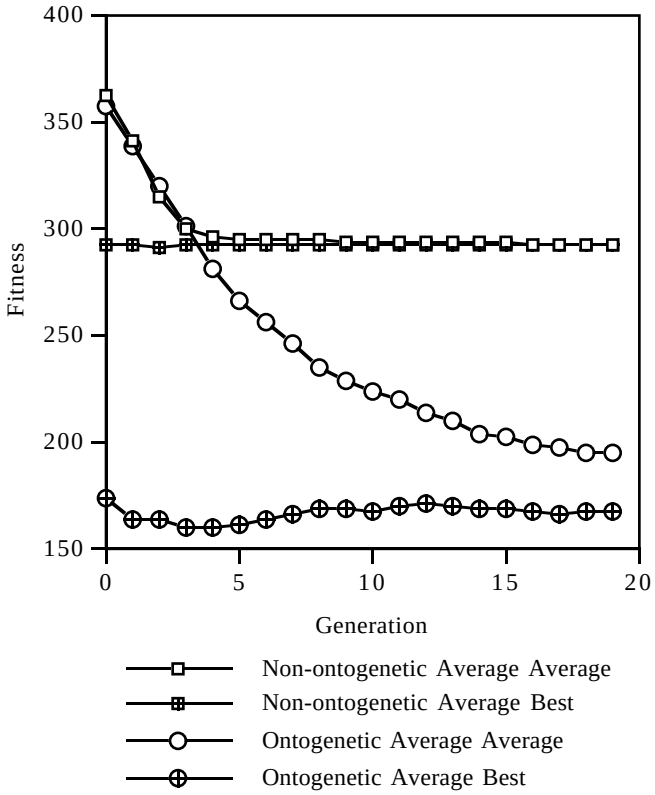

Figure 3: Average and best fitnesses by generation, averaged over the 200 runs for each condition, for the Wumpus world problem.

have grabbed the gold in at least one and usually two or more of the worlds, and it can have died in at most one of the four random worlds. This is difficult; in many cases it is necessary to risk death in order to navigate to the gold, and in some cases the gold may be unobtainable because it is in a pit or in a square surrounded by pits. "Standardized fitness" values (for which lower values are better (Koza, 1992)) are the average of the scores from the four worlds, subtracted from 100.

For the experiments reported here, the seven valid actions were mapped onto the integers from zero to six (inclusive), and a simple function set was used that consists primarily of simple arithmetic operators that manipulate and return numbers in this range.[3] The agent's memory system was implemented as a two-dimensional (7 by 7) indexed memory (Teller, 1994), each element of which could hold a single number. The complete function set is shown in Table 1.[4] For the ontogenetic runs, the above-described `shift-left`, `shift-right`, and `segment-copy` operators were also included. A program's result was computed as the absolute value, modulo 7, of the top element on the stack at the end of program execution.

200 runs of the ordinary (non-ontogenetic) version of HiGP were conducted on this problem with a population size of 200, a maximum program size of 100, a crossover rate of 89.5%, a reproduction rate of 10%, a mutation rate of 0.5%, tournament selection with a tournament size of 3, and a maximum of 20 generations per run. No correct solutions were produced by any of the 200 runs. One can conclude that the problem cannot reliably be solved by ordinary HiGP and the given parameters. Note that the Wumpus world problem always includes an indexed memory, and thereby supports some forms of runtime adaptation even without the use of ontogenetic operators.

200 runs of the ontogenetic version of HiGP were then run on this problem (adding the `segment-copy`, `shift-right` and `shift-left` functions to the function set). The population size and other parameters were the same as those described above. 10 solutions were produced by the 200 runs. The following is one of the evolved programs:

```
noop and 3 write - - not + and + 3 2 noop 6 *
write 5 4 1 ifz 1 + + 6 read 1 and + + 2
shift-right 1 stench breeze + or * 0 breeze +
or 1 2 4 shift-left 3 bump not 1 ifz 0 1 6
read glitter 5 segment-copy not 3 shift-left
shift-right write write * stench - 6 bump
sound - 6 noop bump glitter 0 3 - bump 0 0
sound bump stench 4 * or 1 ifz and 5 2 bump 5
* 5 write 6 and 1 -
```

Figure 3 shows graphs of average and best fitnesses by generation, averaged over the 200 runs conducted for each condition (non-ontogenetic and ontogenetic). This graph shows that with the ontogenetic operators the lowest average best fitness occurs early, around generation 4. Without the ontogenetic operators the average best fitness is considerably worse, it never improves, and it is nearly matched by the average average fitness—that is, the populations appear to have converged on non-solutions. It is therefore unlikely that longer runs (over more generations) would produce solutions without ontogenetic operators. It is clear from the graph that a significant advantage is provided by the ontogenetic operators.

5 Conclusions and Future Work

Ontogenetic programming, an enhancement to the genetic programming methodology, allows for the auto-

[3] The mapping from integers to actions was: 0 = forward, 1 = turn right, 2 = turn left, 3 = shoot, 4 = grab, 5 = release, 6 = climb.

[4] This function set, developed for HiGP, differs slightly from that used in (Spector, 1996), which used S-expression-based genetic programming.

matic generation of programs that adapt to their environments at runtime through the use of program self-modification operators. The ontogenetic programming methodology was described and applied to two examples: binary sequence prediction and action-selection in a virtual world. Runtime adaptation turns out to be useful for both of these problems, and the availability of ontogenetic operators allowed for the evolution of solutions in cases for which ordinary genetic programming failed.

Much additional work must be completed to assess the real value of this technique. Neither of the problems described in this paper are real-world problems, and it is not yet clear that the technique will scale well. It is not even entirely clear what features of an environment are related to the general utility of runtime adaptation. One speculation is that unpredictable dynamism in an environment favors runtime adaptation, but one can make the case that neither of the environments described in this paper have this feature. A program that solves the binary sequence prediction problem must only master a single fixed sequence. Although each program *predicts* the sequence dynamically, one element at a time, the sequence itself does not change throughout a run. Similarly, in the Wumpus world experiments described here, each agent had only to contend with a fixed sequence of four essentially static worlds. But for both of these problems runtime adaptation was nonetheless sufficiently useful to allow for solutions in cases in which no solutions could otherwise be produced. Additional applications of ontogenetic programming to agents in more complex and dynamic real-world environments will help to resolve some of these issues.

More work should also be conducted to examine the self-modification strategies actually employed by successful programs. Many possibilities exist; for example, programs might be copying segments of code into multiple program locations, thereby reusing code modules and obtaining an effect similar to that obtained with automatically defined functions (Koza, 1994) or automatically defined macros (Spector, 1996). It would also be interesting to trace the patterns of recurrence of particular program configurations as a program runs.

The ontogenetic programming technique can be varied in many ways—for example, by changing the set of provided ontogenetic operators. No systematic study of such variations has yet been performed. The use of ontogenetic programming with traditional S-expression-based genetic programming systems has been described (Spector and Stoffel, 1996), but the impact of program representation on the utility of particular ontogenetic operators has not yet been studied.

Considering the importance of adaptation in successful complex systems, it seems reasonable to conjecture that results presented in this paper will generalize in some manner—that genetic programming systems that adaptively generate *adaptive* programs will generally be more useful than those that do not. Ontogenetic programming provides the required capability, although more research must be conducted on the efficient exploitation of this capability.

More broadly, ontogenetic programming systems may present new opportunities for the general study of interactions between evolutionary and developmental processes. Biological systems are adaptive at both evolutionary (phylogenetic) and developmental (ontogenetic) levels; computational models of such systems should be adaptive at both levels as well.

Acknowledgments

Mark Feinstein helped to develop our initial interest in ontogeny, and to refine our understanding of its role in biological systems. Discussions at the 1995 AAAI Fall Symposium on Genetic Programming (Siegel, 1995) helped to further refine our approach to ontogenetic programming. The comments of two anonymous reviewers lead to several improvements in this paper. This research was supported in part by grants from ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), and ARPA contract DAST-95-C0037.

References

Brodie, L. 1981. *Starting FORTH.* Prentice Hall.

Iba, H., T. Sato, and H. de Garis. 1993. Temporal Data Processing Using Genetic Programming. In *Proceedings of the 6th International Conference on Genetic Algorithms, ICGA-95*, edited by Larry J. Eshelman, pp. 279–286. San Francisco: Morgan Kaufmann Publishers, Inc.

Kain, R.Y. 1972. *Automata Theory: Machines and Languages.* New York: McGraw-Hill Book Company.

Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

Perkis, T. 1994. Stack-Based Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pp. 148–153. IEEE Press.

Russell, S.J., and P. Norvig. 1995. *Artificial Intelligence, A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall.

Siegel, E.V., editor. 1995. *Collective Brainstorming at the AAAI Symposium on Genetic Programming*, http://www.cs.columbia.edu/~evs/gpsym95.html.

Spector, L. 1996. Simultaneous Evolution of Programs and their Control Structures. In *Advances in Genetic Programming 2*, edited by P.J. Angeline and K.E. Kinnear, Jr., pp. 137–154. Cambridge, MA: The MIT Press.

Spector, L., and K. Stoffel. 1996. Ontogenetic Programming. In *Proceedings of the Genetic Programming 1996 Conference.* Cambridge, MA: The MIT Press. In press.

Stoffel, K. and L. Spector. 1996. High-Performance Genetic Programming. In *Proceedings of the Genetic Programming 1996 Conference.* Cambridge, MA: The MIT Press. In press.

Teller, A. 1994. The Evolution of Mental Models. In *Advances in Genetic Programming,* edited by Kenneth E. Kinnear, Jr., pp. 199–219. Cambridge, MA: The MIT Press.

Yamauchi, B. and R. Beer. 1994. Integrating Reactive, Sequential, and Learning Behavior using Dynamical Neural Networks. In *From Animals to Animats 3*, edited by D. Cliff, P. Husbands, J. Meyer, and S.W. Wilson, pp. 382–391. Cambridge, MA: The MIT Press.