Culture Enhances the Evolvability of Cognition

Lee Spector* †
lspector@hampshire.edu

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

Sean Luke †
seanl@cs.umd.edu

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

Abstract

This paper discusses the role of culture in the evolution of cognitive systems. We define "culture" as any information transmitted between individuals and between generations by non-genetic means. Experiments are presented that use genetic programming systems that include special mechanisms for cultural transmission of information. These systems evolve computer programs that perform cognitive tasks including mathematical function mapping and action selection in a virtual world. The data show that the presence of culture-supporting mechanisms can have a clear beneficial impact on the evolvability of correct programs. The implications that these results may have for cognitive science are briefly discussed.

Introduction

Interactions between cultural and evolutionary processes are discussed in the literatures of several fields, including cognitive science (Donald, 1991), ethology (Bonner, 1980), sociobiology (Lumsden and Wilson, 1981), and primatology (Quiatt and Itani, 1994). This paper reports related work in the field of evolutionary computation, in which problems are solved by use of computational mechanisms that have been derived from evolutionary processes. In particular, results are presented for genetic programming systems, in which executable computer programs are automatically produced through processes of recombination and natural selection (Koza, 1992). Genetic programming systems can automatically produce computer programs for a variety of interesting cognitive tasks including circuit design, grammar induction, block stacking, and action selection (Koza, 1992; Spector, 1994, 1996).

The representations and algorithms employed by artificially evolved cognitive systems may bear little resemblance to those of natural cognitive systems. They nonetheless exemplify *possible* cognitive mechanisms, and as such they may be of interest to cognitive science more broadly. For purposes of cognitive modeling one can also constrain the evolutionary process in various ways. For example, one can restrict the set of functions and data structures out of which programs are constructed to a set that has been deemed cognitively plausible. The overall architecture of evolved programs can be similarly constrained.

One can measure the *computational effort* required to evolve systems under different conditions (e.g., when the systems are architecturally constrained in various ways). By use of such measurements one can assess the *evolvability* of cognitive systems of various sorts given the specified conditions. Because all natural cognitive systems presumably arose through evolutionary processes, this information may be used as evidence for or against hypotheses about cognitive mechanisms in natural systems.

This study examines the effect that *culture* has on the evolvability of cognitive systems. We define "culture" as any information transmitted between individuals and between generations by non-genetic means.[1] Related notions of culture have been explored in other evolutionary computation paradigms (Bankes, 1995; Reynolds, 1994; Hutchins and Hazlehurst, 1993). In this paper experiments are presented in which genetic programming systems evolve programs that perform cognitive tasks (mathematical function mapping and action selection). Special mechanisms are added to support cultural transmission of information, and the resulting impacts on evolvability are observed. The data show that the presence of culture-supporting mechanisms can have a clear beneficial impact on the evolvability of correct programs.

From an engineering perspective, the results show that the efficiency of genetic programming can be improved through the addition of culture-supporting mechanisms; since the mechanisms are also simple and easy to add, practitioners of evolutionary computation should consider their use for other problems. From a cognitive science perspective, the results may shed some light on the relations between evolution and culture more broadly, and they may also suggest new uses of evolutionary computation in cognitive modeling.

The next section describes the experimental method used in this study, with subsections on genetic programming, the measurement of computational effort, indexed memory (the mechanism on which the implementation of culture is based), culture, and two test problems: symbolic regression of $y = x^4 + x^3 + x^2 + x$ and action selection in Wumpus world. Results are then presented in the form of computa-

[0]To appear in the 1996 Cognitive Science (CogSci96) conference proceedings, UC San Diego.

[1]Bonner (1980) provides a similar definition.

tional effort measurements, and the meaning and generality of the results are discussed.

Method

Genetic Programming

Genetic programming is a technique for the automatic generation of computer programs by means of natural selection (Koza, 1992). The genetic programming process starts by creating a large initial population of programs that are random combinations of elements from problem-specific function and terminal sets. Each program in the initial population is then assessed for fitness, and the fitness values are used in producing the next generation of programs by means of a variety of genetic operations including reproduction, crossover, and mutation. After a preestablished number of generations, or after the 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. Details of program representation and algorithms for all genetic operators, along with full source code for genetic programming systems, can be found in (Koza, 1992) and (Koza, 1994).

Computational Effort

Koza has developed a technique for measuring the computational effort required to solve a problem with genetic programming (Koza, 1994). Because the genetic programming algorithm includes random choices at several steps, data is collected from a large number of independent runs. One first calculates *P*(*M*,*i*), the cumulative probability of success by generation *i* using a population of size *M*. For each generation *i* this is simply the total number of runs that succeeded on or before the *i*th generation, divided by the total number of runs conducted. One then calculates *I*(*M*,*i*,*z*), the number of individuals that must be processed to produce a solution by generation *i* with probability greater than z (where z is usually 99%). *I*(*M*,*i*,*z*) is calculated using the formula:

$$I(M,i,z) = M * (i+1) * \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil$$

The minimum of *I*(*M*,*i*,*z*) over the range of *i* is defined as the "computational effort" required to solve the problem with the given system (Koza, 1994).

Indexed Memory

Indexed memory is a mechanism that allows programs developed by genetic programming to make use of runtime memory (Teller, 1994). The mechanism consists of a linear array of memory locations and two functions, `READ` and `WRITE`, that are added to the set of functions from which programs are created. The memory is initialized at the start of each program execution. `READ` takes a single argument and returns the contents of the memory location indexed by that argument. `WRITE` takes two arguments, a memory index and a data item, and stores the data item in the memory at the specified index. `WRITE` returns the *previous* value of the specified memory location. Teller (1994) showed that indexed memory can help to evolve correct programs for certain problems, and that the combination of indexed memory and iteration allows genetic programming to produce any Turing-computable function. Others have further examined the utility of indexed memory; for example, Andre (1995) has experimented with problems that *require* the use of memory, and has explored the ways in which evolved programs use indexed memory in solving these problems.

Culture

"Culture" can be implemented by modifying Teller's indexed memory mechanism to cause all individuals to share the same memory. The memory should be initialized only at the start of a genetic programming run. Subsequent changes to the memory by any individual will persist and be available to other individuals. This makes it possible for a program to pass information to itself (in later executions in the run), to its contemporaries, to its offspring, and to unrelated members of future generations. The order in which the population is evaluated for fitness can clearly have an effect; the evaluation order within each generation can be randomized to prevent systematic exploitation of this effect.

A culture is the collective product of all individuals throughout evolutionary time. This means that any "good idea" developed by any individual may be preserved for use by all other individuals. Unfortunately, it also means that a single destructive individual can erase a great deal of valuable information. For most of the problems studied so far, the positives outweigh the negatives—the availability of a culture speeds evolution.

A program evolved with culture will generally function correctly only when run in an appropriate cultural context. The cultural context within which a program was evolved is therefore an intrinsic part of the program, and the genetic programming system should report the appropriate cultural context (initial cultural memory state) along with the best-of-run program.

Test Problems

Symbolic Regression The goal of the symbolic regression problem, as described by Koza (1992), is to produce a function, in symbolic form, that fits a provided set of data points. For each element of a set of (x, y) points, the function should map the x value onto the appropriate y value. This is the sort of problem faced by a scientist who has obtained a set of experimental data points and suspects that a simple formula will suffice to explain the data. The scientist may further suspect that such a formula can be constructed from a particular set of arithmetic and trigonometric operators. In searching for the correct formula the scientist is attempting to solve a symbolic regression problem. Once the correct formula is found the scientist may use it to map new x values onto their y values.

For the described experiments data points were obtained from the equation $y = x^4 + x^3 + x^2 + x$ (see Figure 1), which is a standard example from the literature (Koza, 1992). One can view the task of the genetic programming system as

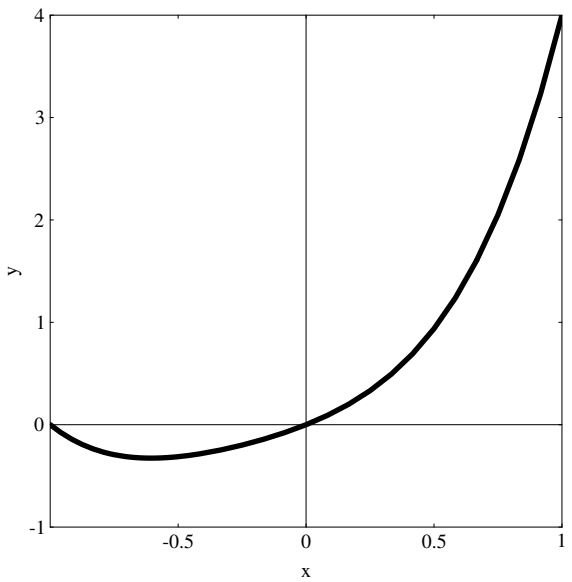

Figure 1: The target function for the symbolic regression problem: $y = x^4 + x^3 + x^2 + x$.

Breeze	O Pit	Breeze		Breeze	O Pit
O Pit	Breeze			Breeze	O Pit
Breeze		Breeze			Breeze
	Breeze	O Pit	Breeze Stench		$ Gold
		Breeze Stench	Wumpus	Stench	Breeze
Agent			Stench	Breeze	O Pit

Figure 2: An instance of a 6-by-6 Wumpus world.

that of "rediscovering" or inducing this formula from the data points used as fitness cases. 20 fitness cases were used for the data presented here, with randomly selected x values between -1 and 1. A program was considered to be successful if it returned a value within 0.01 of the target y value for all 20 cases.

The functions that could be used in evolved programs were the 2-argument addition function `+`, the 2-argument subtraction function `-`, the 2-argument multiplication function `*`, the 2-argument protected division function `%`, two 1-argument trigonometric functions `SIN` and `COS`, the 1-argument exponential function `EXP`, and the 1-argument protected logarithm function `RLOG` (as in (Koza, 1992)). Programs could also refer to the independent variable `X`, and to "ephemeral random constants" between -1 and 1 (also as in (Koza, 1992)). Tournament selection was used (tournament size = 5), along with a 90% crossover rate, a 10% reproduction rate, no mutation, a population size of 1000, and a maximum of 51 generations per run. Detailed descriptions of the meanings of these parameters can be found in (Koza, 1992); it suffices here to note that these values are reasonably standard.

For the runs with indexed-memory and for those with culture a 100-element memory array was added, along with `READ` and `WRITE` functions that behave as described above. The "index" arguments of these functions were coerced to the proper range ([0–99]) by multiplying them by 100 and by then taking them modulo 100.

Wumpus World Wumpus world (Russell and Norvig, 1995) is a problem environment within which an agent must select actions to navigate within a dangerous world to achieve goals. The use of genetic programming for the evolution of Wumpus world agents has been described elsewhere (Spector, 1996). This section describes the problem informally; see (Spector, 1996) for a detailed description of the Wumpus world simulator, the function set, and other parameters.

Wumpus world is cave represented as a grid of squares surrounded by walls. A 6-by-6 grid was used for the experiments described here.[2] The agent's task is to start in a particular square, to move through the world to find and to pick up the 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. Figure 2 shows an instance of a 6-by-6 Wumpus world.

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.

[2] Although the experiments described in (Spector, 1996) also used a 6-by-6 grid, the world's "walls" occupied space in the simulator used for those experiments. The actual playing area for those experiments was therefore 4x4, and the results are therefore not directly comparable to those described here.

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 randomly generated worlds (new worlds are generated for each assessment). 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, and there is a 100-point penalty for each unit of distance between the agent and the gold at the end of the run.[3] "Standardized fitness" values (for which lower values are better (Koza, 1992)) are the average of the scores from the four random worlds, subtracted from 100. 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 random worlds is greater than zero. To have obtained such a score an agent must have grabbed the gold in at least one and usually two or more 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.

For the experiments described here, a C-language reimplementation of Russell and Norvig's wumpus world simulator was used for fitness evaluation. The world size was 6-by-6, the population size was 1,000 and the maximum number of generations per run was 21. Tournament selection was used with a tournament size of 7.

Results

Symbolic Regression

100 runs were performed with no memory, 100 with indexed memory, and 100 with culture. Figure 3 shows a graph of $P(M,i)$, the cumulative probability of success by generation. Note that the highest probability was obtained in the runs that had access to culture. Figure 4 shows $I(M,i,z)$, the number of individuals that must be processed to produce a solution with probability greater than z=99%. Note that the lowest number of individuals was obtained using culture. $I(M,i,z)$ values are always very high in early generations; early generations are therefore not shown in graphs of $I(M,i,z)$ so that the detail in the later generations can be seen. The computational effort results (the minima of the $I(M,i,z)$ graphs) are summarized in Table 1.

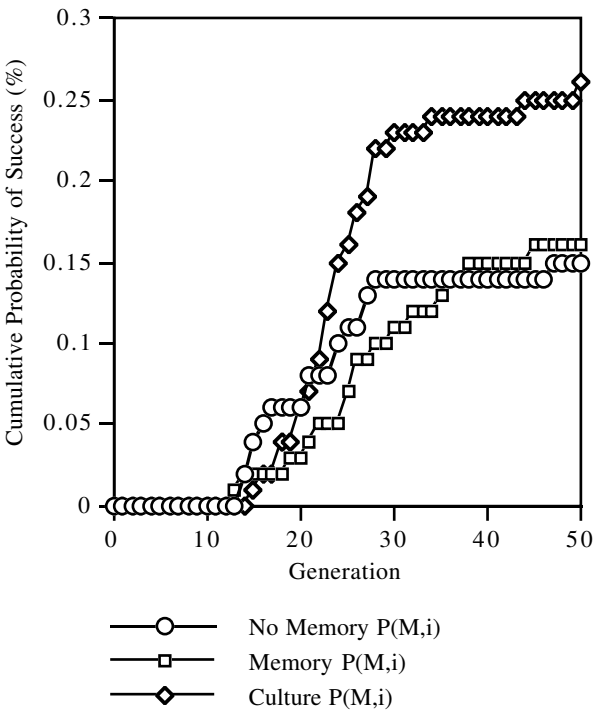

Figure 3: $P(M,i)$ for the symbolic regression problem.

Wumpus World

400 runs were performed with no memory, 509 with ordinary indexed memory, and 800 with culture.[4] Figure 5 shows a graph of $P(M,i)$, the cumulative probability of success by generation. Note that the highest probability was obtained in the runs that had access to culture. Figure 6 shows $I(M,i,z)$, the number of individuals that must be processed to produce a solution with probability greater than z=99%. Note that the lowest number of individuals was obtained using culture. The computational effort results are summarized in Table 2.

Discussion

Conclusions from Computational Effort Results

The symbolic regression results show that the addition of an ordinary indexed memory *increases* the computational effort required to solve the problem. This may be because there are many programs that compute the desired function ($y = x^4 + x^3 + x^2 + x$) without using memory. In addition, the functions `READ` and `WRITE` may act as noise in

[3] In the experiments described in (Spector, 1996) agents were also charged an explicit 100-point penalty for dying. In the version used here the only penalty for death is implicit—after one dies one can no longer get closer to the gold or pick it up.

[4] Computational effort comparisons are indeed valid between sets of different numbers of runs. For these computationally expensive runs we started processes on several machines and stopped them when it was clear that we had obtained sufficient data; the exact numbers of runs are therefore arbitrary.

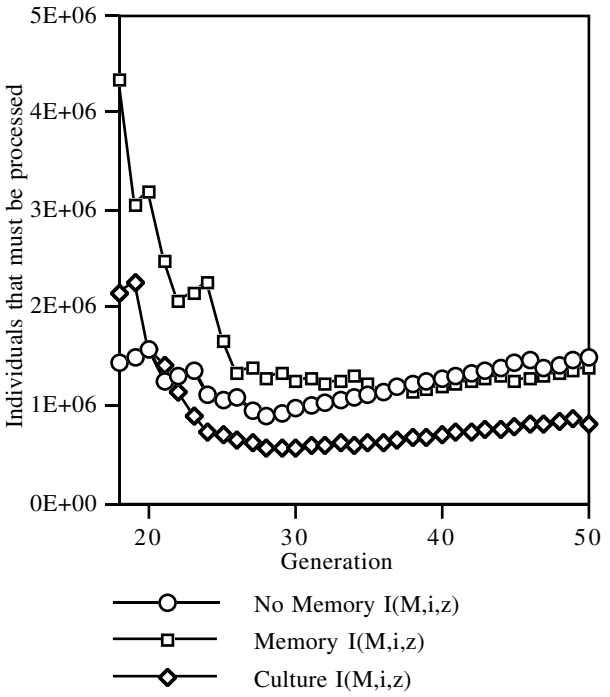

Figure 4: $I(M,i,z)$ for the symbolic regression problem.

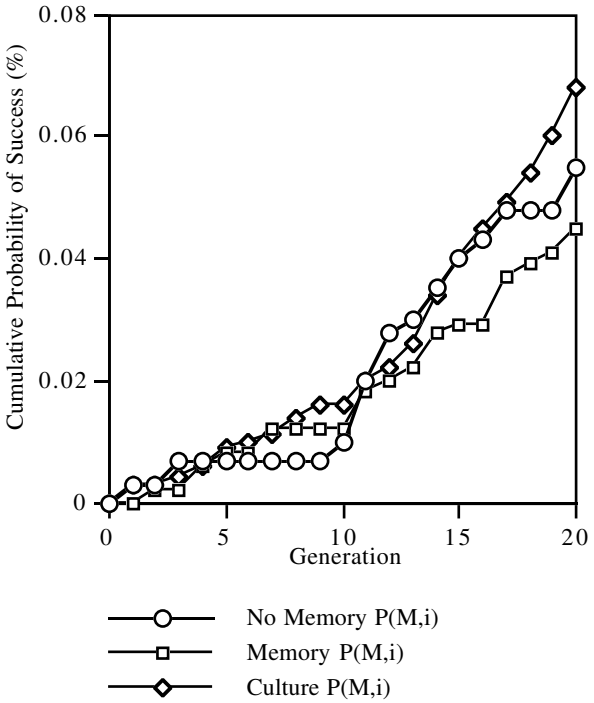

Figure 5: $P(M,i)$ for the Wumpus world problem.

the function set, contributing little to success while filling positions that could instead be filled with useful functions. Nevertheless, when the memory is shared between individuals as culture, the system is able to take advantage of this, and the computational effort is reduced to 61% of that required when no memory is available (49% of that required with ordinary indexed memory).

The Wumpus world results exhibit the same pattern, although the results are noisier and the interpretation is therefore less clear. When culture is used the computational effort required to produce a successful Wumpus world agent is reduced to 81% of that required when no memory is available (66% of that required when ordinary indexed memory is used). Surprisingly, indexed memory once again increases computational effort over the "no memory" condition. It appears that the reactive strategies are at least as useful as knowledge-based strategies for this domain.

The results show that culture-supporting mechanisms may be quite useful within some cognitive system architectures. Their utility is demonstrated by their impact on the computational effort required for evolution; less effort is required to produce successful cognitive systems when the culture-supporting mechanisms are present than is required when they are not. In summary, culture enhances the evolvability of cognition, at least for the tasks and evolutionary mechanisms presented here.

Generality

While culture has been useful in most of the domains to which we have applied it, further work must be conducted

Condition	Computational Effort
No memory	899,000
Memory	1,131,000
Culture	551,000

Table 1: Computational efforts for symbolic regression.

to characterize the relationship between domain characteristics and the expected impact of the technique. For example, the Wumpus world environment can be varied in many ways, and the relative utility of memory and culture may differ in each of these variants. We are currently conducting experiments with several variants and it is clear from the preliminary results that culture is *not always* beneficial in Wumpus world. In particular, for a configuration similar to that of (Spector, 1996) it appears that ordinary indexed memory is best, followed by culture. Here we may gain insights from biological examples; not *all* animals use culture, and there are presumably many niches in which culture confers no adaptive advantage. On the other hand, culture

Condition	Computational Effort
No memory	1,710,000
Memory	2,100,000
Culture	1,386,000

Table 2: Computational efforts for Wumpus world.

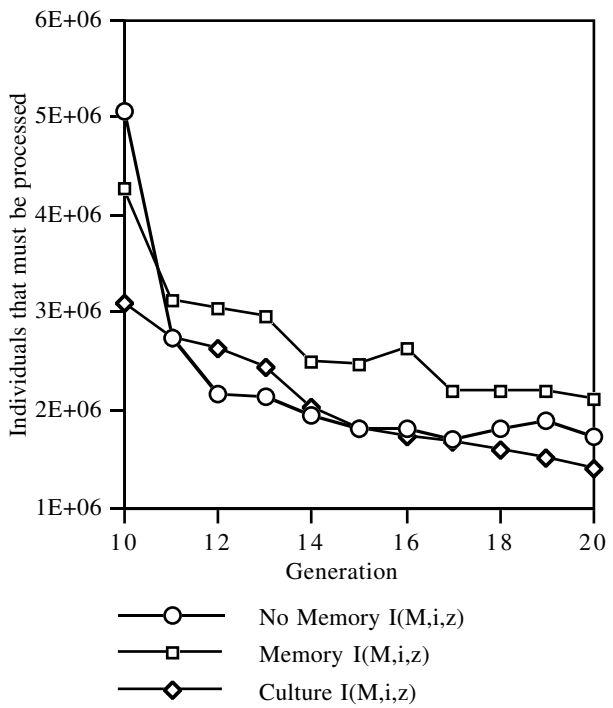

Figure 6: $I(M,i,z)$ for the Wumpus world problem.

appears to be extremely useful in certain situations (e.g., for humans). This suggests that we might expect correspondingly large impacts in certain genetic programming domains.

Although the evolution-enhancing effects of culture were demonstrated only for genetic programming systems, there are reasons to believe that the effects are more general. Bonner (1980) has described a number of ways in which culture can enhance adaptation and alter the course and speed of evolutionary processes. Several features of organisms and of their environments can contribute to the possible utility of culture, and there is no reason to believe that the genetic programming paradigm exploits these features any more effectively than might other evolutionary computation systems or natural biological evolution. The utility of culture is largely driven by the demands and resources of a task environment; if it contains important regularities that can be *taught* and *learned* then it may be advantageous for individuals to transmit such information behaviorally. One reason is that the encoding of the information in behavior (including "writing" and "teaching" behaviors) may be simpler than the encoding of the same information in the organism's genetic code. Another reason is that information can spread through a population much more quickly by behavioral means than it can by genetic means.

Future Work

From an engineering perspective, the present study suggests several directions for additional research. The generality of the observed effects should be further studied through applications to other problem environments. The combined use of ordinary indexed memory and culture should also be examined. Variations of the technique, for example to limit the destructive effects that unfit individuals can have on the culture, should also be explored. All of these studies may further enhance the utility that culture provides in the genetic programming of problem-solving systems.

From a cognitive science perspective, the present study suggests a new research paradigm that could be applied not only to the further study of interactions between evolution and culture, but also to the study of other aspects of cognition. Computational modeling has a long history in cognitive science, but modeling by means of evolutionary computation is relatively new. Within this framework models are automatically generated by evolutionary processes. One immediate benefit is that the development of models may thereby require comparatively little time and effort. The resulting models may in some cases be difficult to analyze, but several sorts of meaningful data may nonetheless be extracted. For example, the models may be run on new inputs, compared to hand-crafted models in various ways, or "lesioned" to assess their fault-tolerance and failure modes. Alternatively, as in the present study, meaningful data may be extracted from the evolutionary process, rather than from the resulting models. Evidence against a set of hypothesized cognitive mechanisms could be produced, for example, by showing that it is difficult or impossible to evolve the appropriate sorts of cognitive systems using the hypothesized mechanisms as primitives. Similar strategies could be employed to provide evidence for or against general cognitive architectures. Existing techniques can be used to measure the computational effort required to evolve systems under various conditions, and the comparisons can be made "fair" by using the same evolutionary mechanisms for each set of conditions. This contrasts to ordinary computational cognitive modeling, in which different models are often produced by different programming teams. In some of these cases it may not be clear which aspects of the performance of the resulting systems are due to the underlying theories, and which are due to the strengths and weaknesses of the programmers.

Acknowledgments

Some of the ideas in this paper were refined in discussions with Jordan Grafman and Daniel Kimberg. The authors also wish to thank the anonymous reviewers for helpful comments. 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

Andre, D. (1995). The Evolution of Agents that Build Mental Models and Create Simple Plans Using Genetic Programming. In *Proceedings of the Sixth International Conference on Genetic Algorithms* (pp. 248–255). San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Bankes, S. (1995). Evolving Social Structure in Communities of Agents Through Meme Evolution. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press.

Bonner, J.T. (1980). *The Evolution of Culture in Animals.* Princeton, NJ: Princeton University Press.

Donald, M. (1991). *Origins of the Modern Mind: Three Stages in the Evolution of Culture and Cognition*. Cambridge, MA: Harvard University Press.

Hutchins, E. & Hazlehurst, B. (1993). Learning in the Cultural Process. In C.G. Langton, C. Taylor, J.D. Farmer, & S. Rasmussen (Eds.), *Artificial Life II*. Addison-Wesley Publishing 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.

Lumsden, C.J., & Wilson, E.O. (1981). *Genes, Mind, and Culture*. Cambridge, MA: Harvard University Press.

Quiatt, D. & Itani, J. (Eds.). (1994). *Hominid Culture in Primate Perspective*. Niwot, CO: University Press of Colorado.

Reynolds, R.G. (1994). An Introduction to Cultural Algorithms. In *Proceedings of the Third Annual Conference on Evolutionary Programming* (pp. 131–139). River Edge, NJ: World Scientific.

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

Spector, L. (1994). Genetic Programming and AI Planning Systems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 1329–1334). Cambridge, MA: MIT Press.

Spector, L. (1996). Simultaneous Evolution of Programs and their Control Structures. In P. Angeline & K. Kinnear (Eds.), *Advances in Genetic Programming 2*. Cambridge, MA: MIT Press. In press.

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