Cultural Transmission of Information in Genetic Programming
by Lee Spector and Sean Luke

Full citation:
Spector, L., and S. Luke. 1996. Cultural Transmission of Information in Genetic Programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors) *Genetic Programming 1996: Proceedings of the First Annual Conference*, 209-214. Cambridge, MA: The MIT Press.

Cultural Transmission of Information in Genetic Programming

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 shows how the performance of a genetic programming system can be improved through the addition of mechanisms for non-genetic transmission of information between individuals (culture). Teller has previously shown how genetic programming systems can be enhanced through the addition of memory mechanisms for individual programs [Teller 1994]; in this paper we show how Teller's memory mechanism can be changed to allow for communication between individuals within and across generations. We show the effects of indexed memory and culture on the performance of a genetic programming system on a symbolic regression problem, on Koza's Lawnmower problem, and on Wumpus world agent problems. We show that culture can reduce the computational effort required to solve all of these problems. We conclude with a discussion of possible improvements to the technique.

1 Culture and Evolution

In most evolutionary computation systems, individuals are assessed for fitness independently. Each individual is placed into an initialized environment, a fitness test is performed, and the environment is then re-initialized for the next individual. By contrast, in biological populations (and in many Artificial Life systems; see, e.g., [Brooks and Maes 1994]) each individual acts in a shared environment that may bear the marks of contemporaries and of predecessors. Some animals use these marks to their advantage, and some intentionally modify the environment to communicate with others. Modifications to the environment may be ephemeral (e.g., calls, spoken language) or long-lasting (e.g., nests, written language). Through these mechanisms fitness-enhancing information may be transmitted to others, including offspring, by non-genetic means.

We use the term "culture" to refer to any information transmitted between individuals by non-genetic means. Our definition is similar to that of Bonner, who describes his use of the term as follows:

> By culture I mean the transfer of information by behavioral means, most particularly by the process of teaching and learning. It is used in a sense that contrasts with the transmission of genetic information passed by the direct inheritance of genes from one generation to the next. [Bonner 1980, p. 9]

Culture, in this sense, is employed by many animals other than humans. Higher primates provide many examples, but one can also argue that much simpler animals, even insects, make significant use of culture [Bonner 1980].

One can view the evolution of a culture-using species as a complex interaction between two processes, one genetic and one cultural. Dawkins coined the term "meme" to serve the function for cultural transmission that "gene" serves for genetic transmission; that is, a meme is a unit of information transmitted by behavioral means [Dawkins 1976]. The interactions between cultural and genetic transmission, and their combined effects on fitness and adaptation, have been the subject of much discussion in evolutionary biology. For our purposes we may simply note that the difference in mode of transmission leads to a significant difference in the speed with which a new piece of information may spread through a population. Again, quoting Bonner:

> If, for instance, a favorable genetic mutation appears in one individual, even with reasonable positive selection pressures, it will take many life cycles and perhaps hundreds of years for the new gene to be present in an appreciable number of individuals in the population. It is only in some lethal circumstances that the genetic structure of a population can change rapidly.
>
> ... On the other hand a cultural change can be exceedingly rapid. A new fad or dress style may take over a whole nation in a matter of days or weeks. [p. 18]

The difference in speed of transmission leads to a complementary difference in stability; the gene pool is less vulnerable to sudden dangerous fluctuations than is the meme pool. As a result one can expect that a combination of the two transmission modes will be most beneficial for the evolution of certain complex systems.

Several researchers have previously experimented with cultural elements in evolutionary computation systems. For example, Bankes has explored "meme-based" adaptive systems in multiplayer games, although the bulk of this work is concerned with the evolution of memes *within* individuals [Bankes 1995]. Reynolds has developed a framework of dual inheritance "cultural algorithms" in which a "belief space," containing generalizations of the representations of individuals, helps to guide the evolution of a population [Reynolds 1994]. Others have explored related ideas in a range of evolutionary computation contexts (e.g., [Hutchins and Hazlehurst 1993]).

In contrast to previous research, this paper describes a cultural mechanism that is a straightforward extension of the memory mechanisms used by individuals. The same mechanisms that are used by individuals to build and maintain state information are extended to allow for the communication of information between individuals within and across generations. The virtues of this scheme include simplicity, a better match to biological notions of culture, and a demonstrable positive impact on the speed of evolution for several test problems.

The remainder of this paper is organized as follows: after a brief digression on performance in genetic programming we describe a simple technique for adding cultural transmission to genetic programming systems. We then show how the addition of culture can reduce the computational effort required to produce solutions for symbolic regression, Lawnmower, and Wumpus world problems. We conclude with a discussion of possible improvements to the technique.

2 Genetic Programming and Performance

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 via 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.

The impact of alternative approaches to genetic programming can only be assessed by measuring performance over a large number of runs. This is because the algorithm includes random choices at several steps; in any particular run the effects of the random choices may obscure the effects of the alternative approaches.

To analyze the performance of a genetic programming system over a large number of runs one can first calculate *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. Given *P*(*M*,*i*) one can calculate *I*(*M*,*i*,*z*), the number of individuals that must be processed to produce a solution by generation *i* with probability greater than *z*.[1] *I*(*M*,*i*,*z*) can be calculated using the following formula:

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

Koza defines the minimum of *I*(*M*,*i*,*z*) as the "computational effort" required to solve the problem with the given system.[2]

3 Memory and Culture in Genetic Programming

Teller has developed a simple mechanism that allows programs produced by genetic programming to make use of memory. The mechanism, called *indexed memory*, consists of a linear array of memory locations and two functions, `READ` and `WRITE`, that are added to the function set. The memory is initialized (e.g., to 0) at the start of each fitness case. `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 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 [Teller 1994]. Others have further examined the utility of indexed memory; for example, Andre 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 [Andre 1995].

We implement "culture" through a simple modification to Teller's indexed memory: all individuals share the same memory, and it is initialized only at the start of a genetic programming run. This means that the contents of memory at the end of one fitness evaluation are available for use in the

[1] For the analyses in this paper a value of *z*=99% is always used.

[2] The *P*(*M*,*i*) and *I*(*M*,*i*,*z*) measures were developed by Koza and are discussed on pages 99 through 103 of [Koza 1994].

next fitness evaluation. If individual programs are subjected to multiple fitness cases, an individual program can use this mechanism to pass information to itself from one fitness case to the next. In any event, programs that are evaluated later may use information left in memory by those that were evaluated earlier. A program may pass information to itself (in later fitness cases), 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; we randomize evaluation order within each generation to prevent systematic exploitation of this effect.

Note that the culture is the collective product of all individuals throughout evolutionary time. On the positive side, this means that a "good idea" developed by any individual in any population may be preserved for use by all other individuals. On the negative side, this means that one destructive individual can destroy a great deal of valuable information. For most of the problems that we have studied so far, the positives outweigh the negatives—the availability of a culture speeds evolution. One avenue for future work, however, is to minimize the effects that destructive individuals can have on the culture (see below).

A program evolved with culture may function correctly only when run with a particular initial memory state. The initial memory state is therefore an intrinsic part of the program, and the genetic programming system should report the appropriate initial memory state along with the best-of-run program. If one later wishes to run such a program outside of the context of the evolutionary process, one should first re-initialize the cultural memory with the reported memory state.

4 Symbolic Regression

To assess the utility of cultural transmission we ran the *lilgp* genetic programming system [Zongker 1995] on a symbolic regression problem. We compared the performance of the system on versions of the problem in which the programs had access to no memory, to ordinary indexed memory, and to culture.

The goal of the symbolic regression problem, as described in [Koza 1992], is to produce a function that fits a provided set of data points. For each element of a set of (*x*, *y*) points, the program should return the correct *y* value when provided with the *x* value. For our experiments we obtained our data points from the equation $y = x^4 + x^3 + x^2 + x$. One can view the task of the genetic programming system as that of "rediscovering" this formula from the data points used as fitness cases. We used 20 fitness cases, 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 value for all 20 cases.

For our experiments we used a function set consisting of the two-argument addition function `+`, the two-argument subtraction function `-`, the two-argument multiplication function `*`, the two-argument protected division function `%`, two one-argument trigonometric functions `SIN` and `COS`, the one-argument exponential function `EXP`, and the one-argument protected logarithm function `RLOG` (as in [Koza 1992]). We used a single terminal for the independent variable `X`, and ephemeral random constants between -1 and 1 (also as in [Koza 1992]). We used tournament selection (tournament size = 5), a 90% crossover rate, a 10% reproduction rate, no mutation, a population size of 1000, and a maximum of 51 generations per run.

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

We performed 100 runs with no memory, 100 runs with indexed memory, and 100 runs with culture. The results can be summarized as follows:

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

These results show that the addition of an ordinary indexed memory *increases* the computational effort required to solve the regression problem. Presumably this is because the search space includes many programs that compute the desired function ($y = x^4 + x^3 + x^2 + x$) without using memory. One might additionally conjecture that the unhelpful functions `READ` and `WRITE` act as noise in the function set, contributing little to success while filling positions that could instead be filled with useful functions. Nevertheless, when the memory is preserved between fitness evaluations 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.

5 The Lawnmower Problem

Similar results were obtained for Koza's 64-square Lawnmower problem [Koza 1994]. The goal in the Lawnmower problem is to find a program for controlling the movement of an autonomous lawnmower on an 8-by-8 torroidal world. All values manipulated by functions for this problem are vectors of integers modulo 8 of the form (i, j), where $0 \leq i, j \leq 7$. The terminal set consists of the 0-argument function `(MOW)`, the 0-argument function `(LEFT)`, and random vector constants modulo 8 ($\Re_{v8}$). `(MOW)` moves the lawnmower in the direction it is currently facing, mows the lawn at the new location, and returns the vector value (0,0).

(LEFT) turns the lawnmower 90° to the left and returns the vector value (0,0). The function set consists of the operators V8A, FROG, and PROGN. V8A is a 2-argument vector addition function that adds vector components modulo 8. FROG is a 1-argument movement operator that jumps the lawnmower to the coordinate produced by adding (modulo 8) its vector argument to the lawnmower's current location. FROG acts as the identity operator on its argument. PROGN is a 2-argument sequencing operator that returns the value of its second argument.

For the runs with indexed-memory and with culture we added a 2-dimensional, vector-indexed, 64-element (8-by-8) memory array along with appropriate READ and WRITE functions.

We performed 100 runs with no memory, 100 runs with indexed memory, and 100 runs with culture. In all cases we used a population size of 500, two automatically-defined functions (as in [Koza 1994]), and a maximum of 51 generations. The results can be summarized as follows:

Condition	Computational Effort
No memory	12,000
Memory	16,000
Culture	10,000

These results show the same pattern as the results for symbolic regression, although the effect is less extreme. Again, memory by itself slows evolution, but evolution proceeds most quickly with culture.

6 Wumpus World

Wumpus world [Russell and Norvig 1995] is a problem environment that is more complex than those described above in several interesting ways. The use of genetic programming for the evolution of Wumpus world agents has been described elsewhere [Spector 1996]. In this section we present only an informal description of the problem and the results of our culture experiments. See [Spector 1996] for more information on the Wumpus world simulator, the function and terminal sets, and other parameters.

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 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.

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 produce 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; that is, sensing is free.

Agents are assessed on the basis of performance in four randomly generated worlds. Four new random worlds are generated for each fitness evaluation; this helps to prevent the production of agents that are overfitted to a particular set of worlds. 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. "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 may be 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.

1709 runs of the lil-gp genetic programming system were performed on the wumpus world problem, 400 without memory, 509 with ordinary indexed memory, and 800 with culture.[3] A C-language re-implementation of Russell and Norvig's wumpus world simulator was used for fitness evaluation. The population size was 1,000 and the maximum number of generations per run was 21. Tournament selec-

[3] Note that 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.

tion was used with a tournament size of 7. The results can be summarized as follows:

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

These results show that when culture is used the computational effort required to produce a successful agent for this version of Wumpus world is reduced to 66% of that required when ordinary indexed memory is used. They also show, surprisingly, that 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 version of Wumpus world used in this experiment differs from that used in previous work [Spector 1996]; while re-implementing the system in C we made other changes as well. In particular, the new version operates on a true 6-by-6 world. In the old version the world's "walls" occupied space so the actual playing area in a "6-by-6" world was only 4-by-4. In addition, the old version charged an agent an explicit 100-point penalty for dying. In the new version the only penalty for death is implicit—after one dies one can no longer get closer to the gold or pick it up.

Wumpus world can be varied in many other ways as well, 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. Further experiments and analysis will be required to discover the reasons for culture's utility in various circumstances.

7 Discussion and Future Work

It is clear from the reported experiments that culture can have a significant beneficial impact on computational effort for certain problems. Fortunately, it is easy to add culture to other genetic programming applications: one simply includes an indexed memory that is shared between all fitness evaluations, and arranges for the appropriate memory state to be returned along with the best-of-run program. If the target application already includes an indexed memory, then simply by sharing this memory between all fitness evaluations the computational effort required to produce a correct program may be decreased. If the application does not already include an indexed memory, then a shared indexed memory can usually be added with little difficulty.

Although we have examined memory and culture as alternatives in this paper, there is no reason why they can't be combined. By providing two versions of the `READ` and `WRITE` functions, one each for ordinary indexed memory and one each for culture, one can give evolving programs access to both mechanisms; for some problems, this may provide the maximum benefit.

Another avenue for future work is to minimize the effects that destructive individuals can have on the culture. One way to do this would be to have each fitness test occur using a copy of the culture in a local memory. The changes to the memory could then be transferred to the culture in a fitness-proportionate way — the greater the fitness of an individual, the greater its impacts on the culture. Unfortunately, it may be the case that low-fitness individuals nonetheless contribute to the culture in important ways; fitness-proportionate impact on the culture would prevent the population from benefiting from these contributions. An alternative scheme for minimizing the effects of destructive individuals would be to use many sub-populations, each of which has its own sub-culture, and to periodically merge sub-populations and sub-cultures. Many variations on this theme exist, and many may be worthy of further exploration.

The possibility also exists that some destructive individuals may be highly fit. Some such individuals may in fact be using destruction to maintain an advantage over others in the population; they could make use of information in the culture and then destroy it so that it would not be available to others.[4] In order to detect this sort of activity one could collect information about access to the culture by individuals of various fitness levels throughout a run. Such information, combined with genealogical audit trails [Koza 1992], would also permit one to explore broader questions about how and why individuals make use of culture.

While culture has been useful in most of the domains to which we have applied it, further work must be conducted to characterize the relationship between domain characteristics and the expected impact of the technique. Here again 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 appears to be extremely useful in certain situations (e.g., for humans), and we may expect correspondingly large impacts in certain genetic programming domains.

8 Conclusions

Culture, the transmission of information between individuals by non-genetic means, can in some cases improve the performance of a genetic programming system. This paper showed how culture can be implemented as a simple modification of Teller's indexed memory mechanism. The paper demonstrated that culture can improve the performance of a genetic programming system for three problems: symbolic regression of $y = x^4 + x^3 + x^2 + x$, the 64-square Lawnmower

[4] Frank D. Francone drew this possibility to our attention.

problem, and a version of Wumpus world. Further research should examine the effects that destructive individuals may have on a shared culture, delineate the cases in which culture is helpful, and examine the ways in which culture is actually employed by successful individuals.

Acknowledgments

The authors wish to thank the reviewers for several helpful comments; in particular, Frank D. Francone raised more interesting issues than we were able to explore in the context of this paper — but they will help to guide our ongoing research on the project. 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.

Bibliography

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*, edited by L.J. Eshelman. 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.

Brooks, R.A., and P. Maes, eds. 1994. *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: The MIT Press.

Dawkins, R. 1976. *The Selfish Gene.* Oxford: Oxford University Press.

Hutchins, E. and B. Hazlehurst. 1993. Learning in the Cultural Process. In *Artificial Life II*, edited by C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen. 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.

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

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

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

Zongker, D., and B. Punch. 1995. *lil-gp 1.0 User's Manual*. WWW: http://isl.cps.msu.edu/GA/software/lil-gp, or via anonymous FTP to isl.cps.msu.edu, in the directory "/pub/GA/lilgp".