Homologous Crossover in Genetic Programming

Frank D. Francone
Register Machine Learning Technologies
360 Grand Avenue Oakland, CA 94610
ffrancone@aimlearning.com
+1-510-834-7191

Markus Conrads
Informatik Centrum Dortmund
44227 Dortmund Germany
conrads@icd.de
+49-2361-185954

Wolfgang Banzhaf
University of Dortmund,
LS11, D-44221, Dortmund, Germany
banzhaf@tarantoga.informatik.uni-dortmund.de
+49-231-970095

Peter Nordin
Chalmers University of Technology
S-41296 Gotenberg, Sweden
nordin@fy.chalmers.se
+46-31-772315

ABSTRACT

In recent years, the genetic programming crossover operator has been criticized on both theoretical and empirical grounds. This paper introduces a new crossover operator for linear genomes that encourages the emergence of positional homology in the population. Preliminary experimental results suggest that this approach is a promising direction for redesign of the mechanism of crossover.

1. Introduction

Crossover in genetic programming has, in recent years, been described as highly destructive [Nordin, 1996] and as performing no better than mutation [Angeline, 1997]. By way of contrast, crossover in nature appears robust and rarely produces lethally defective offspring.

This paper describes and tests a new form of crossover for linear GP. It was inspired by our observation that homology in a population and constraints on crossover appear to be important to the success of crossover in nature. This new operator implements a limited form of forced alignment between genomes during crossover and may encourage the emergence of positional homology in genetic programming populations.

2. The Emergence of Homology

Natural crossover is highly constrained [Banzhaf, et al., 1998]. For example, natural crossover is strongly biased to exchanging genes that are in the same *position* on the chromosome and that express similar *functionality*. That bias arises from two features of crossover:

(1) *Homology*. Crossover almost always occurs between organisms that have nearly identical base pair sequences in their chromosomes. For example, in organisms that reproduce sexually, mating occurs only between members of the same species. Within a species, there is a high degree of similarity (homology) between the genomes of the various members of that species.

(2) *Base Pair Bonding*. Two strands of DNA combine into a single double helix because of base-pair bonding between the bases in the two strands. As a result, there is a strong tendency for two strands of DNA to recombine (crossover) in a manner that exactly matches the complementary base-pair sequences in the other. [Watson, et al., 1987] Indeed, during crossover, "complementary base-pairing between strands unwound from two different chromosomes puts the chromosomes in exact register. Crossing over thus generates homologous recombination; that is, it occurs between two regions of DNA containing identical or nearly identical sequences." [Id]

Homology in nature is, therefore, both *positional* and *functional*. These rigid constraints on crossover suggest that it is an evolved operator—a form of emergent order. Homology permits the meaningful exchange of functionally similar genes through the base-pair bonding mechanism. At the same time, the existence of base pair bonding tends to encourage the emergence of homology. When these biases are applied through many iterations, one can envision the evolution of emergent homology in nature. By this view, sexual reproduction in species represents the evolution of evolvability in nature [Altenberg 1994].

3. Genetic Programming Homology

With few exceptions, crossover in genetic programming is characterized by swapping code between two parents with-

out regard to *position* of the code in the parent programs or the *function* the code performs. This is true regardless whether trees, graphs or linear structures are used to represent the genome [Koza, 1992] [Teller, 1996] [Nordin, 1994].

Furthermore, genetic programming runs have no obvious *homology* in the population, at least at the beginning. After all, the programs in a population are initialized randomly—the opposite of the almost identical genomes shared by members of a sexually reproducing species. This problem is well illustrated by an example we have given elsewhere:

> "Crossing over two programs [in GP] is a little like taking two highly fit word processing programs, Word For Windows and WordPerfect, cutting the executables in half and swapping the two cut segments. Would anyone expect this to work? Of course not." [Banzhaf, et al., 1998, page 158].

The reason the conclusion in the quoted material is so obvious is that there is no homology between Word Perfect and Word for Windows. Accordingly, although these two programs perform the same overall function in a very similar manner, they are in a sense, from different "species." As such, we do not expect them to mate successfully any more than we would expect a zebra to mate successfully with a cow, even though both animals eat grass and have four legs.

But on closer reflection, a type of homology does emerge in genetic programming populations and does so frequently. The term, "introns" in genetic programming has come to refer to nodes or instructions in evolved programs that have no effect on the output of the program, such as *x=x*1* [Banzhaf et al., 1998, p. 186-198] [Nordin et al., 1996] [Soule & Foster, 1996] [Soule & Foster, 1997]. Code bloat (or runaway growth of introns) is a commonly remarked feature of genetic programming runs. When code bloat has come to dominate a population, crossover becomes a matter of swapping sequences of code that have no effect on the fitness of the individual. Such crossover is necessarily non-destructive because it is completely neutral. [Nordin & Banzhaf, 1995].

Code bloat is a form of emergent homology, albeit of a perverse type. Where code has bloated, there is a high degree of homology between almost any two randomly chosen nodes on any two evolved programs in this population. That is, any two nodes chosen for crossover are likely to contain functionally useless code. Because crossover between two such nodes is almost always neutral, the population—dominated by introns and neutral crossover—has become a "species" in the sense that all members can mate with each other with a high probability that the offspring will not be lethally defective. This is, in effect, position independent homology.

4. Encouraging Useful Homology

The homology represented by code bloat is not especially useful or even interesting. Its existence does, however, establish that homology can emerge from genetic programming runs, depending on initial conditions. The issue presented by this paper is whether it is possible to redesign the crossover operator so that it encourages useful homology. In that regard, we need to identify a crossover operator that:

(1) Encourages a genetic programming run to evolve useful homology in the population; and

(2) Exploits the evolved homology usefully.

Such a crossover operator could be expected to be less-and-less destructive as the run continues, tend to discourage code bloat and, perhaps, improve system performance.

4.1 Sticky Crossover

We propose to make our program genomes "sticky," somewhat like DNA, during crossover. We refer to this as a "homologous crossover operator" or, in lighter moments, as "sticky crossover."

In simple terms, the homologous crossover operator only permits instructions at the same position in the genome to be exchanged with other instructions in the same position. More precisely, sticky crossover chooses a sequence of code randomly from one parent program. The sequence of code in the same *position* from the second parent is then chosen. The two segments are then swapped.

4.2 Sticky Crossover Dynamics

Our new operator does not actively seek out functionally equivalent segments of code in evolved programs and cross them over. So in that sense, it does not duplicate the base pair bonding of biological crossover.

Instead, our proposed operator provides a high probability that, say, the second instruction in an evolved program, will be swapped during crossover with the second instruction from another program. This may encourage the evolution of functionally similar code at equivalent positions in the genome.

Why should this be so? In [Nordin & Banzhaf, 1995], we proposed that the probability that an individual will propagate in increasing numbers from one generation to the next depends not just on its own fitness, but on expected value of its offspring's fitness. We referred to this aggregate measure of fitness as *"effective fitness."* Effective fitness describes the notion that even highly fit programs will not long survive if they cannot produce fit offspring. Thus, effective fitness measures a program's ability to be successfully replicated by means of crossover.

The effective fitness of an evolved program may be quite different for different types of crossover. For example, evolved programs containing a high proportion of code that actually affects their output have little defense against destructive ordinary crossover, even if they are highly homologous as to other individuals in the population. This is because ordinary crossover is constantly moving the functional code around in the genome. Each time that happens,

positional homology among members of the population changes. So there is little chance that positional homology will evolve out of functional code when exposed to ordinary crossover.

By way of contrast, we propose that programs containing a substantial proportion of functional code and that are relatively homologous to other programs in the population will have higher effective fitness than those that are not so homologous, when exposed to the sticky crossover operator.

Assume the following about a genetic programming population:

(1) Two programs in the population are more than randomly homologous as between themselves;

(2) The rest of the population is randomly homologous as to all other members of the population; and

(3) The sticky exchange of homologous sequences is less likely to produce low fitness children than the exchange of non-homologous sequences, as appears to be the case in nature.

Given these premises, and all other things being equal, the two relatively homologous programs will be more likely to produce fit offspring when subjected to sticky crossover than other programs in the population. That is, *relatively homologous programs have higher effective fitness*. Because of the reproductive advantage of their children, these two programs will be more likely to propagate their genetic material through generations of evolution than randomly homologous programs.

There is another, and more emergent aspect to this analysis. When two programs that are relatively homologous are crossed over (using sticky homologous crossover), they necessarily produce children that are themselves relatively homologous to the other children *and* to the parents. In other words, the number of relatively homologous programs in the population increases after crossover of two relatively homologous individuals because sticky crossover does not alter homology. As the offspring of two evolved programs that are relatively homologous to each other (and their offspring) spread throughout the population, the descendants' relative reproductive advantage deriving from that homology spreads because there are now more homologous programs in the population derived from the same ancestors. Thus, we would predict that the homology of the overall population should also increase and that homology should appear as emergent order.

But does emerging homology contribute to *useful* fitness from a problem-solving viewpoint? We know from runaway code bloat that emergent homology is not always useful. In this regard, however, sticky, homologous crossover is quite different than standard linear crossover. It respects the position of the swapped code; that is, it respects positional homologies.

With standard crossover, the only code that could be the cause of emergent homology is introns (useless code) because that is the only code that is probably homologous *no matter where it appears in the genome*. But with our homologous crossover operator, code that is homologous to other programs in the population need not also be homologous at other positions on the genome because sticky crossover does not move the code around. Obviously, *functional code may exhibit homologous properties when exposed to sticky crossover as long as that code only needs to be homologous in one location on the genome.*

This reasoning does not, of course, prove that useful homology will emerge. Only that we can expect homology with this new operator, to look considerably different than the perverse homology of code bloat, that it will be homology of, at least partially functional code, and that this functional code may well evolve through normal selection to be useful code.

There is one prediction we can make from this reasoning. If useful homology emerges, it will do so because crossover is less destructive when performed between homologous elements of the population. Previous research indicates that intron growth in genetic programming is caused in large part by the destructive effect of the crossover operator. [Nordin et al., 1996] [Soule & Foster, 1997]. If crossover becomes less destructive because of emerging homologies, we would predict that increasing the amount of homologous crossover in the population should decrease the rate of growth of the length of programs in the population.

4.3 Previous Related Work

Our approach is not entirely novel. We are aware of two previous sets of experiments in tree-based genetic programming that added positional dependence to the crossover operator.

D'Haeseleer devised strong context preserving crossover in 1994. In that work, D'Haeseleer permitted crossover to occur only between nodes that occupy exactly the same position in the two parents. D'Haeseleer got modest improvement by combining ordinary crossover with his strong context-preserving crossover [D'Haeseleer, 1994].

Similarly, Poli & Langdon introduced one point crossover in 1997 [Poli & Langdon, 1997]. One point crossover is even more restrictive than strong context preserving crossover because it only permits crossover between nodes that match in position and arity.

But homologous crossover with linear genomes is considerably different than these two prior approaches. These previous works allow very large subtrees and very small subtrees to be exchanged in crossover, as long as the base node of the two subtrees occupies the same position in the genome. By way of contrast, our homologous linear operator requires that the exchanged code sequences are very similar in size *and* that they occupy the same position in the genome. Thus, our linear operator is forced to exchange segments of code that are more likely to be similar (in semantics and, therefore, function) than the code exchanged in these two tree based systems.

5. Implementation

AIMGP stands for **A**utomatic **I**nduction of **M**achine Code, **G**enetic **P**rogramming. AIMGP uses linear genomes made up of native machine code functions. It performs crossover, mutation and fitness evaluations directly on the machine code [Nordin 1994].

Our first experiments with homologous crossover in AIMGP began about two years ago at the University of Dortmund. During that time, we have become increasingly convinced of its beneficial effects.

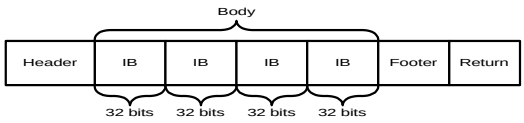
The experiments reported here were run using Discipulus™1.0 software [RML Technologies, Inc., 1998], which is a commercial version of AIMGP written for WINTEL machines running 486DX, Pentium® and Pentium II® processors. It implements over fifty native machine code instructions from the floating-point processor in these systems.

Although we will provide some implementation details here, the implementation and operation of this software is exhaustively documented in [Francone, 1998].

5.1 Program Representation

During evolution, AIMGP systems hold evolved programs as native machine code functions. Each program is comprised of the following parts: header, body, footer and a return instruction as shown in Figure 1.

Figure 1. The Structure of an AIMGP Program. ("IB" Refers to Instruction Blocks).



The body of the evolved programs is where evolution takes place. The crossover and mutation operators operate only in the body and the output of each evolved program is calculated in the body of the program.

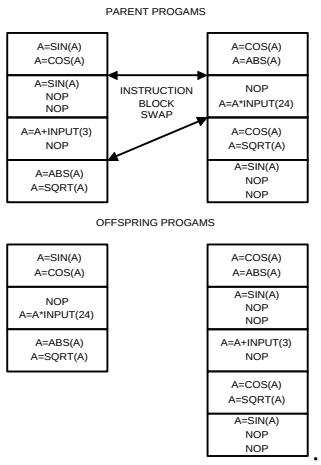
The body of an evolved program is composed of one or more 32 bit "instruction blocks." An instruction block may be comprised of a single 32-bit instruction or any combination of 8, 16, or 24-bit instructions that add up to 32 bits. Crossover occurs only at the boundaries of the instruction blocks and without reference to the content of the instruction blocks.

5.2 Crossover And Mutation

In these experiments, we blended "ordinary" crossover and sticky, homologous crossover. In this system, some non-homologous crossover is necessary because that is the only way for genomes to change length.

Ordinary Crossover. Ordinary crossover is traditional AIMGP crossover—two-point crossover, with the selected code fragments being chosen randomly from the two parents. Figure 2 represents traditional (non-homologous) crossover in Discipulus™. Each block represents an instruction block. Each instruction block may contain one or more instructions. Crossover occurs at the boundary of instruction blocks.

Figure 2. Non-Homologous Crossover in AIMGP using Instruction Blocks



Homologous Crossover. We implemented homologous crossover as shown in Figure 3. What distinguishes homologous from non-homologous crossover is that in homologous crossover, instruction blocks can only be swapped with an instruction block at the *same position* in the other parent's genome.

Mutation. Mutation makes a random change to a randomly chosen instruction block. There are three kinds of mutation:

- Block Mutation. Random regeneration of an entire instruction block;
- Data Mutation. Randomly pick one instruction in the instruction block and randomly change an operand; and
- Instruction Mutation. Randomly pick one instruction in the instruction block and randomly change its operator.

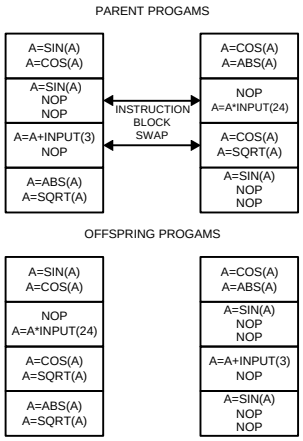
6. Experimental Setup[1]

We chose to test the effect of adding homologous crossover to linear genetic programming on the Gaussian 8D problem. Gaussian 8D is a classification problem comprising two classes and eight independent variable inputs. In addition to the eight 'real' inputs, we added sixteen 'false' inputs—that is, sixteen data series containing random variables are provided to the system in addition to the eight real inputs. This

[1] All data, software, parameters and configuration files necessary to duplicate these experiments in their entirety may be found at [RML Technologies, Inc., 1999].

makes the system engage in variable selection as well as classification.

Figure 3. Homologous Crossover in AIMGP.



The eight 'real' inputs were generated as follows:

- For Class 0 examples, each of the inputs are normally-distributed, random values with a mean of 0 and a standard deviation of 1.
- For Class 1 examples, each of the inputs are normally-distributed, random values with a mean of 0 and a standard deviation of 2.

We divided the data into training and validation sets of one thousand examples each.

We measured the performance of a run by the best classification rate attained on the validation set (in percentage correct classifications).

To test the effect of homologous crossover, we used the parameters described at www.aimlearning.com/gecco. The only parameters varied were the mutation rate and the homologous crossover rate as follows:

- The mutation rate was varied between 5% and 80% (5%, 20%, 50%, and 80%).
- The overall crossover rate was fixed at 100%. The composition of the crossover operator as between homologous (sticky) and non-homologous crossover was varied as follows: 0%, 20%, 40%, 60%, 80%, and 95%. (Any crossover that is not homologous is traditional or non-homologous crossover.)

Altogether we performed twenty-five runs at each parameter setting using a different random seed for each of the twenty-five runs. Because there were twenty-four different parameter combinations tested, we performed 600 runs to obtain the data reported here.

Each run was allowed to continue for two minutes before termination. Although two minutes may seem like a very short run for a difficult problem, this is machine code genetic programming. It is approximately sixty times faster than typical genetic programming systems. Accordingly, a run of two minutes on this system is comparable to a run of about two hours on a typical genetic programming system.

Table 1. Best Percentage Hit Rate on Validation Set by Mutation Rate (horizontal legend) and Homologous Crossover Rate (vertical axis). Each box is average of 25 runs.

	Mutation Rate			
Hom.Crossover Rate	5%	20%	50%	80%
0%	71.1	72.6	72.9	74
20%	69.5	71.4	74.9	73.9
40%	70.6	73.2	73.7	73.4
60%	68	72.4	74.5	74.6
80%	70	74.4	74.9	75.7
95%	70.8	75.3	77.6	77.4

7. Results

We measured overall performance by the percentage of correct classifications attained by the best individual on the validation set.

Table 2. Number of Tournaments before Average Length Equaled or Exceeded 500 bytes, in thousands of tournaments (000), by Mutation Rate (horizontal legend) and Homologous Crossover Rate (vertical axis). Each box is average of 25 runs.

	Mutation Rate			
Hom.Crossover Rate	5%	20%	50%	80%
0%	10	11	10	12
20%	9	10	11	11
40%	9	11	9	9
60%	11	11	9	10
80%	16	18	12	14
95%	20	24	22	22

Table 1 reports the results of our runs cross-tabulated by mutation rate and homologous crossover rate. As is apparent, runs with high homologous crossover rates tend to perform better than runs with lower homologous crossover rates.

Table 2 shows the average number of tournaments that transpired before the average length of the evolved programs in the population exceeded 500 bytes in length. The more homologous crossover, the longer runs continue before they reach 500 bytes in length. This effect is pronounced at each level of mutation.

7. Discussion

On these data, both of our predictions are confirmed for this problem. Homologous crossover does apparently have a significant and beneficial effect on the fitness of the best individual on the validation data and on code bloat.

Significantly, the emergent aspects of homologous crossover do not become apparent until the operator comprises at least 80% and perhaps 95% of all crossover events. This suggests the mechanism by which the homologous operator works is in fact the emergence of homology within the population.

8. Further Work

The homologous crossover operator introduced here is only the beginning of what we believe should be a concerted effort into improving the crossover operator by analogy to biological systems. For this operator, no attempt was made to guide the evolution of homology within the population other than permitting the genomes to have a degree of stickiness. Further, in our current experimental setup, only one species is able to evolve per population. Additional research is planned into mechanisms that would permit differential speciation within the population. Finally, we expect future work to look more deeply into the actual operation of the crossover operator at run-time rather than viewing only the collective output of a run as a single entity.

Acknowledgments

Peter Nordin gratefully acknowledges support from the Swedish Research Council for Engineering Sciences.

The authors would like to thank Register Machine Learning Technologies for use of the Discipulus™ software.

Bibliography

Altenberg, L. 1994. The evolution of evolvability in genetic programming. In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming*, chapter 3, pages 47-74. MIT Press, Cambridge, MA.

Angeline, P. 1997. Subtree crossover: Building block engine or macromutation. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., and Riolo, R., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13-16, 1997*, pages 9-17, Stanford University, Stanford, CA. Morgan Kaufmann, San Francisco, CA.

Banzhaf, Nordin, Keller & Francone, 1998. *Genetic Programming: An Introduction*. San Francisco, CA, Morgan Kaufmann Publishers, Inc.

D'Haesseleer, P. 1994. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256-261, Orlando, FL, IEEE Press, New York.

Francone, F. 1998. *Discipulus Owner's Manual*. Available for no charge from Register Machine Learning Technology, Inc., at www.aimlearning.com.

Holland, J. 1975. *Adaptation in natural and artificial systems*. Cambridge, MA. The MIT Press.

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

Nordin, P. A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming*, chapter 14, pages 311-331. MIT Press, Cambridge, MA.

Nordin, P. and Banzhaf, W. 1995. Complexity compression and evolution. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference*, pages 310, 317, Pittsburgh, PA. Morgan Kauffmann, San Francisco, CA.

Nordin, P., Francone, F., and Banzhaf, W. 1996. Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P.J. and Kinnear, Jr., K.E. editors, *Advances in Genetic Programming 2*, chapter 6, pages 111-134. MIT Press, Cambridge, MA.

Poli, R. and Langdon, W.B. 1997. A new schema theory for genetic programming with one-point crossover and point mutation. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., and Riolo, R., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13-16, 1997*, pages 278-285, Stanford University, Stanford, CA. Morgan Kaufmann, San Francisco, CA.

RML, Technologies, Inc. 1998. Discipulus™ 1.0. A demonstration version is available for no charge from RML Technologies at www.aimlearning.com.

RML Technologies, Inc. 1999 www.aimlearning.com/gecco.

Soule, T. Foster, J.A. and Dickinson, J. 1996. Code growth in genetic programming. In Koza, J.R., Goldberg, D.E., Fogel, D.B., and Riolo, R.L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215-223, Stanford University, CA. MIT Press, Cambridge MA.

Soule, T. and Foster, J.A. 1997. Code size and depth flows in genetic programming. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., and Riolo, R., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference, July 13-16, 1997*, pages 313-320, Stanford University, Stanford, CA. Morgan Kaufmann, San Francisco, CA.

Teller, A. 1996. Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, P.J. and Kinnear, Jr., K.E., editors, *Advances in Genetic Programming 2*, chapter 3, pages 45-68. MIT Press, Cambridge, MA.

Watson, J.D., Hopkins, N.H., Roberts, J.W., Steitz, J.A., and Weiner, A.M. 1987. *Molecular Biology of the Gene*. Benjamin-Cummings, Menlo Park, CA.