Directed Crossover within Genetic Programming

W. B. Langdon[*]

Dept of Computer Science,
University College London.

23 April 1996

Revision history

Revision	Date	Change
17 August 1995		First version. Taken from draft chapter in "Advances in Genetic Programming 2", dated 16 August 1995
1.2	24 Sep 1995	Use RN format. Add Section 2 and improve introduction.
1.3	9 Feb 1996	Add notes on Blickle's edge marking and EDIs
1.4	23 Apr 1996	Add Soft Broad Selection

Abstract

This paper describes in detail a mechanism used to bias the choice of crossover locations when evolving a list data structure using genetic programming. The data structure and it evolution will be described by RN/95/70.

The second section describes current research on biasing the action of reproduction operators within the genetic programming field.

1 Directed Crossover

Often some tasks are easier than others and so evolve more quickly. It is obviously wasteful to perform crossover in code that is working correctly and the directed crossover mechanism described in this paper succeeds in dynamically redistributing crossover locations to code in need of improvement as the population evolves. The current mechanism only considers code at the level of individual operations or adfs but could obviously be refined.

In the evolution of the list abstract data structure each operation (e.g. insert) is allocated a separate GP tree within each individual in the population. Together with the five ADFs (which also have a unique tree each) this makes a total of fiveteen trees per individual. By keeping a record of which trees are executed and with what outcome (i.e. scores on subsequent fitness tests) the current performance of each tree within an individual can be described. This description is used to bias which trees are chosen for crossover. Once a tree has been selected the crossover points within it are chosen in the usual random way.

[1]W.Langdon@cs.ucl.ac.uk

In 90% of crossovers the first parent's fitness and execution path is used to bias the choice of crossover location. The location is chosen to avoid disrupting code that is working, to avoid wasting crossovers by changing code that is never executed and is biased in favour of changing code that appears to be performing poorly. The number of times the tree might be used by parts of the fitness cases which have not be executed is estimated ($unknown$). This is used as a dampening term to avoid emphisising results for trees which have been executed only a few times.

The tree in which crossover is to be performed is selected by choosing three trees at random (with reselection) from the 15. A tree cannot be one of these three if:–

- The tree is believed to be correct.

  This is decided by the fitness tests. If a tree has passed sufficient number of tests, it may be regarded as correct, even if when used in combination with other trees one or more consistency tests fail.

- The tree has never been executed,

- The tree has never been executed in a test sub sequence which subsequently failed a consistency check (nak) and it is anticipated that it would not be executed by any of the fitness test cases that have not been used ($unknown$).

However a tree may be selected as the crossover location immediately if either:–

- The number of times it was used in a test subsequence which subsequently passed its consistency check (ok) is less than nak, or

- both it has never been run successfully and $unknown$ is zero.

Otherwise, the following ratio is calculated:

$$\frac{nak + unknown}{ok + unknown} \tag{1}$$

When ratios for three trees have been calculated, crossover occurs in the tree with the highest ratio.

If 100 trees are examined before three meeting these requirements are found, the tree with the highest ratio found so far is selected. If none have been found, the tree is selected at random from the 15.

To save space, the trace information is tightly packed during which some resolution is lost. An implementation of the above algorithm is available (for non-commercial purposes) via anonymous ftp, node `cs.ucl.ac.uk` directory `genetic/gp-code`

As far as is known, this is the first time program execution paths have been used to guide the choice of GP crossover location. The following section describes other approaches which use program syntax, fitness measures or population performance.

2 Other Approaches to Biasing GP Evolution

Koza [Koz92] and most others use a crude aspect of program syntax (i.e. is the program a terminal or function) to stochastically guide the location of crossover points. This section describes some more sophisticated techniques at guiding the GP evolution.

A number of other papers show (albeit on very different problems) benefits in directing or biasing the operation of the crossover or other genetic operators. For example Rosca [RB94] uses the runtime behavior of each program to define a "fitness" for each subtree within it. These fitnesses are used to bias the choice of which subtrees to protect from crossover.

D'haeseleer [D'h94] describes methods, based upon the syntax of the two parent programs, for biasing the choice of crossover locations.

Gruau [Gru96] argues strongly that GPer should be forthright in using program syntax to guide the GP and shows improved GP performance by using an external grammar to define more tightly the syntax of the evolving programs.

Whigham [Whi95] also uses a grammar to constrain the evolving trees but the grammar itself evolves based on the syntax of previously successful programs (in fact the best of each generation). The grammar does not become more constrictive but instead the rules within it are allocated a fitness which biases (rather than controls) the subsequent evolution of the population.

The work on strongly typed GP ([Mon93], [Mon94], [Mon95], [HWSS95] and [HSW96]) also shows that constraining the GP to search only parts of its search space by forbidding semantically invalid programs improves performance.

Other approaches to protect code from crossover are Peter Angeline's [Ang93] genetic library and Peter Nordin's [NFB96] use of "introns"[2]. More recently Angeline [Ang96] has advocating evolving the probability of crossover occurring at different points in the program along with the program itself. He also suggests multiple crossovers to produce an offspring. The ETL group [Id96] is also believed to be active in this area and other work will be reported in [KGFR96].

Blickle [BT94, Section 4] claims improved performance by marking tree edges when they are evaluated and ensuring crossover avoids unevaluated trees, however the improvement is problem dependent.

The approach in [Tac95] is different in the genetic operator itself is not biased, instead improved offspring are produced by producing multiple offspring per parent pairing and using a (possibly simple) fitness function to ensure only the best are released into the population and so able to bread themselves. [Cre95, Section 2.2.1] uses a similar technique.

References

[Ang93] Peter John Angeline. *Evolutionary Algorithms and Emergent Intelligence*. PhD thesis, Ohio State University, 1993.

[Ang96] Peter J. Angeline. Two self-adaptive crossover operators for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5. MIT Press, Cambridge, MA, USA, 1996.

[2] Blickle has found a performance degradation on adding explicit introns.

[BT94] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[Cre95] Ronald L. Crepeau. Genetic evolution of machine language software. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 121–134, Tahoe City, California, USA, 9 July 1995.

[D'h94] P. D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Gru96] Frederic Gruau. On using syntactic constraints with genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19. MIT Press, Cambridge, MA, USA, 1996.

[HSW96] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 18. MIT Press, Cambridge, MA, USA, 1996.

[HWSS95] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 271–278, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Id96] Hitoshi Iba and Hugo de Garis. Extending genetic programming with recombinative guidance. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 4. MIT Press, Cambridge, MA, USA, 1996.

[KGFR96] John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors. *Genetic Programming 1996 : Proceedings of the First Conference*, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[Mon93] David J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 May 1993.

[Mon94] David J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, March 1994.

[Mon95] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[NFB96] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 6. MIT Press, Cambridge, MA, USA, 1996.

[RB94] J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Tac95] Walter Alden Tackett. Greedy recombination and genetic search on the space of computer programs. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–297, Estes Park, Colorado, USA, 31 July–2 August 1994 1995. Morgan Kaufmann.

[Whi95] P. A. Whigham. Inductive bias and genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 461–466, Sheffield, UK, September 1995. IEE.