White Paper

Discipulus™ Linear-Genetic-Programming Software: How it Works

F.D.Francone--Chalmers University of Technology and RML Technologies, Inc., Email: ffrancone@aimlearning.com

Introduction

This paper describes the workings of Discipulus Linear-Genetic-Programming software at a high-level. For a detailed, low-level discussion of evolution of machine code, see [20].

Some of the features in Discipulus that contribute to its extraordinary performance [3, 4, 5, 6, 9] are:

- Discipulus implements a Genetic Programming algorithm. This algorithm determines the appropriate functional form and optimizes the parameters of the function. It is an ideal algorithm for complex, noisy, poorly understood domains.
- Discipulus performs Genetic Programming thru direct manipulation of binary machine code. This makes Discipulus about sixty to two-hundred times faster than comparable automated learning approaches [10].
- Discipulus performs *multi-run* Genetic Programming, intelligently adapting its own parameters to the problem at hand.

Each of these capabilities of Discipulus are discussed below.

Genetic Programming

Genetic Programming (GP) is the automatic, computerized creation of computer programs to perform a selected task using Darwinian natural selection. GP developers give their computers examples of how they want the computer to perform a task. GP software then writes a computer program that performs the task described by the examples.

GP is a robust, dynamic, and quickly growing discipline. It has been applied to diverse problems with great success—equaling or exceeding the best human-created solutions to many difficult problems [11,3,4,2]. Good, detailed treatments of Genetic Programming may be found in [2,11].

Discipulus™ [18] is a linear-genetic-programming (LGP) software package that operates directly on machine code. The LGP algorithm in Discipulus is surprisingly simple. It starts with a population of randomly generated computer pro-

grams. These programs are the "primordial soup" on which computerized evolution operates. Then, GP conducts a "tournament" by selecting four programs from the population—also at random—and measures how well each of the four programs performs the task designated by the GP developer. The two programs that perform the task best "win" the tournament.

The GP algorithm then copies the two winner programs and transforms these copies into two new programs via crossover and mutation transformation operators—in short, the winners have "children." These two new child programs are then inserted into the population of programs, replacing the two loser programs from the tournament. GP repeats these simple steps over and over until it has written a program that performs the selected task.

GP creates its "child" programs by transforming the tournament winning programs. The transformations used are inspired by biology. For example, the GP mutation operator transforms a tournament winner by changing it randomly—the mutation operator might change an addition instruction in a tournament winner to a multiplication instruction. Likewise, the GP crossover operator causes instructions from the two tournament winning programs to be swapped—in essence, an exchange of genetic material between the winners. GP crossover is inspired by the exchange of genetic material that occurs in sexual reproduction in biology.

Genetic Programming using Direct Manipulation of Binary Machine Code

Machine-code-based, LGP is the direct evolution of binary machine code through GP techniques [12-17]. Thus, an evolved LGP program is a sequence of binary machine instructions. For example, an evolved LGP program might be comprised of a sequence of four, 32-bit machine instructions. When executed, those four instructions would cause the central processing unit (CPU) to perform operations on the CPU's hardware registers. Here is an example of a simple, four-instruction LGP program that uses three hardware registers:

```
register 2 = register 1 + register 2
register 3 = register 1 - 64
register 3 = register 2 * register 3
register 3 = register 2 / register 3
```

While LGP programs are apparently very simple, it is actually possible to evolve functions of great complexity using only simple arithmetic functions on a register machine [15,17].

After completing a machine-code LGP project, the LGP software decompiles the best evolved models from machine code into Java, ANSI C, or Intel Assembler programs [18]. The resulting decompiled code may be linked to the optimizer and compiled or it may be compiled into a DLL or COM object and called from the optimization routines.

The linear machine code approach to GP has been documented to be between 60 to 200 times faster than comparable interpreting systems [10,12,17]. As will be developed in more detail in the next section, this enhanced speed may be used to conduct a more intensive search of the solution space by performing more and longer runs.

Multiple-Run Genetic Programming

Discipulus is a multiple-run genetic-programming system. That is, it is designed to intelligently perform many runs. While doing so, it intelligently adapts its parameters to the problem at hand.

The importance of multi-run genetic-programming derives from the fact that genetic-programming is a stochastic algorithm. Accordingly, running it over-and-over with the same inputs usually produces a wide range of results, ranging from very bad to very good. For example, Fig. 2 shows the distribution of the results from 30 runs of LGP on the incinerator plant modeling problem mentioned in the introduction—the R^2 value is used to measure the quality of the solution. The solutions ranged from a very poor R^2 of 0.05 to an excellent R^2 of 0.95.

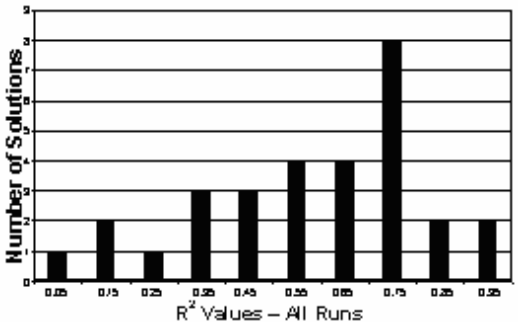

Fig. 1. . Incinerator Control Data. Histogram of Results for 30 LGP Runs

Our investigation to date strongly suggests the typical GP distribution of results from multiple GP runs includes a distributional tail of excellent solutions that is not always duplicated by other learning algorithms. For example, for three separate problem domains, an GP system produced a long tail of outstanding solutions, even though the average GP solution was not necessarily very good. By way of contrast, and in that same study, the distribution of many neural networks runs on the same problems often produced a good average solution, but did not produce a tail of outstanding solutions like GP [8.4].

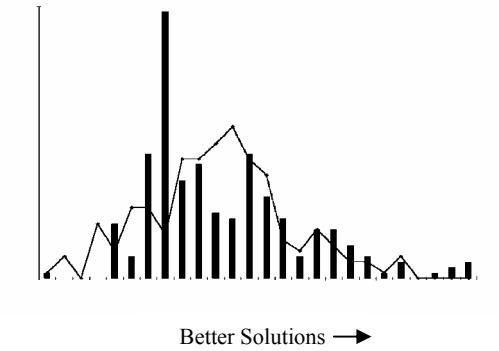

Fig. 2. Typical Comparative Histograms of the Quality of Solutions Produced by LGP Runs (bars) and Neural Network Runs (lines). Discussed in detail in [8]

Figure 3 shows a comparative histogram of LGP results versus neural network results derived from 720 runs of each algorithm on the same problem. Better solutions appear to the right of the chart. Note the tail of good LGP solutions (the bars) that is not duplicated by a comparable tail of good neural network solutions. This same pattern may be found in other problem domains [id].

To locate the tail of best solutions on the right of Figure 3, it is *essential* to perform many runs, regardless whether the researcher is using neural networks or LGP. This is one of the most important reasons why a machine-code approach to GP is preferable to other approaches. It is so much faster than other approaches, that it is possible to complete many runs in realistic time frames on a desktop computer. That makes it more capable of finding the programs in the good tail of the distribution.

References

1. Bäck and Schwefel (1993) Black, T. and Schwefel, H.P., 1993. An Overview of Evolutionary Algorithms for Parameter Optimization, Evolutionary Computation, 1(1): pp. 1-23, 1993.
2. Banzhaf, W., Nordin, P., Keller, R., Francone, F. (1998) Genetic Programming, An Introduction, Morgan Kauffman Publishers, Inc., San Francisco, CA.
3. Deschaine, L.M., (2000) Tackling real-world environmental challenges with linear genetic programming. PCAI Magazine, Volume 15, Number 5, September/October, pp. 35-37.
4. Deschaine, L.M., Patel, J.J., Guthrie, R.G., Grumski, J.T., and Ades, M.J. (2001) "Using Linear Genetic Programming to Develop a C/C++ Simulation Model of a Waste Incinerator," The Society for Modeling and Simulation International: Advanced Simulation Technology Conference, Seattle, WA, USA April, ISBN: 1-56555-238-5, pages 41-48.
5. Deschaine, L.M., Hoover, R.A. Skibinski, J. (2002) "Using Machine Learning to Complement and Extend the Accuracy of UXO Discrimination Beyond the Best Reported

Results at the Jefferson Proving Grounds," (in press), Proceedings of Society for Modeling and Simulation International, April.
6. Fausett, L.V. (2000). A Neural Network Approach to Modeling a Waste Incinerator Facility, Society for Computer Simulation's Advanced Simulation Technology Conference, Washington, DC, USA April.
7. Fogel, D.B. (1992) Evolving Artificial Intelligence. PhD thesis, University of California, San Diego, CA.
8. Francone, F., Nordin, P., and Banzhaf. W. (1996) Benchmarking the Generalization Capabilities of a Compiling Genetic Programming System Using Sparse Data Sets, In Koza et al. Proceedings of the First Annual Conference on Genetic Programming, Stanford, CA.
9. Deschaine, Larry, M. & Francone, F., (2004) White Paper: Comparison of Discipulus™ Linear Genetic Programming Software with Support Vector Machines, Classification Trees, Neural Networks and Human Experts. Available at www.aimlearning.com.
10. Fukunaga, A., Stechert, A., Mutz, D. (1998) A Genome Compiler for High Performance Genetic Programming, in Proceedings of the Third Annual Genetic Programming Conference, pp. 86-94, Morgan Kaufman Publishers, Jet Propulsion Laboratories, California Institute of Technology Pasadena, CA.
11. 14. Koza, J., Bennet, F., Andre, D., and Keane, M. (1999) Genetic Programming III. Morgan Kaufman, San Francisco, CA.
12. Nordin, J.P. (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In Advances in Genetic Programming, K. Kinnear, Jr. (ed.), Cambridge MA: MIT Press.
13. Nordin, J.P. (1999). Evolutionary Program Induction of Binary Machine Code and its Applications, Krehl Verlag.
14. Nordin, J.P., Banzhaf , W. (1995). Complexity Compression and Evolution. In Proceedings of Sixth International Conference of Genetic Algorithms, Morgan Kaufmann Publishers, Inc.
15. Nordin, J.P., Banzhaf, W. (1995). Evolving Turing Complete Programs for a Register Machine with Self Modifying Code. In, Proceedings of Sixth International Conference of Genetic Algorithms, Morgan Kaufmann Publishers, Inc.
16. Nordin, J.P., Francone, F., and Banzhaf, W. (1996) Explicitly Defined Introns and Destructive Crossover in Genetic Programming. Advances in Genetic Programming 2, K. Kinnear, Jr. (Editor), Cambridge MA: MIT Press.
17. Nordin, J.P., Francone, F., and Banzhaf, W. (1998) Efficient Evolution of Machine Code for CISC Architectures Using Blocks and Homologous Crossover. In Advances in Genetic Programming 3, MIT Press, Cambridge MA.
18. Register Machine Learning Technologies, Inc. (2002) Discipulus Users Manual, Version 3.0. Available at www.aimlearning.com.