Genetic Programming

William H. Hsu
Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall
Manhattan, KS 66506-2302
USA
voice: +1 785-532-6350
fax: +1 785-532-7353
email: bhsu@cis.ksu.edu

Genetic Programming

William H. Hsu, Kansas State University

INTRODUCTION

Genetic programming (**GP**) is a sub-area of evolutionary computation first explored by John Koza (1992) and independently developed by Nichael Lynn Cramer (1985). It is a method for producing computer programs through adaptation according to a user-defined fitness criterion, or objective function.

GP systems and genetic algorithms (GAs) are related but distinct approaches to problem solving by simulated evolution. As in the GA methodology, GP uses a representation related to some computational model, but in GP, fitness is tied to task performance by specific program semantics. Instead of strings or permutations, genetic programs are most commonly represented as variable-sized expression trees in imperative or functional programming languages, as grammars (O'Neill & Ryan, 2001), or as circuits (Koza *et al.*, 1999). GP uses patterns from biological evolution to evolve programs:

- **Crossover** – exchange of genetic material such as program subtrees or grammatical rules
- **Selection** – the application of the fitness criterion to choose which individuals from a population will go on to reproduce
- **Replication** – the propagation of individuals from one generation to the next
- **Mutation** – the structural modification of individuals

To work effectively, GP requires an appropriate set of program operators, variables, and constants. Fitness in GP is typically evaluated over fitness cases. In data mining, this usually means training and validation data, but cases can also be generated dynamically using a simulator or directly sampled from a real-world problem solving environment. GP uses evaluation over these cases to measure performance over the required task, according to the given fitness criterion.

This chapter begins with a survey of the design of GP systems and their applications to data mining problems, such as pattern classification, optimization of representations for inputs and hypotheses in machine learning, grammar-based information extraction, and problem transformation by reinforcement learning. It concludes with a discussion of current issues in GP systems: scalability, human-comprehensibility, code growth and reuse, and incremental learning.

BACKGROUND

Although Cramer (1985) first described the use of crossover, selection, and mutation and tree representations for using genetic algorithms to generate programs, Koza is indisputably the field's most prolific and influential author. (Wikipedia, 2004) In four books, Koza *et al.* (1992, 1994, 1999, 2003) have described GP-based solutions to numerous toy problems and several important real-world problems.

State of the field: To date, GPs have been successfully applied to a few significant problems in machine learning and data mining, most notably symbolic regression and feature construction. The method is very computatinally intensive, however, and it is still an open question in current research whether simpler methods can be used instead. These include supervised inductive learning, deterministic optimization, randomized approximation using non-evolutionary algorithms (such as Markov chain Monte Carlo approaches), or genetic algorithms and evolutionary algorithms. It is postulated by GP researchers that the adaptability of GPs to structural, functional, and structure-generating solutions of unknown form makes them more amenable to solving complex problems. Specifically, Koza *et al.* demonstrate (1999, 2003) that in many domains, GP is capable of "human-competitive" automated discovery of concepts deemed to be innovative through technical review such as patent evaluation.

MAIN THRUST OF THE CHAPTER

The general strengths of genetic programming lie in its ability to produce solutions of variable functional form, reuse partial solutions, solve multi-criterion optimization problems, and explore a large search space of solutions in parallel. Modern GP systems are also able to produce structured, object-oriented, and functional programming solutions involving recursion or iteration, subtyping, and higher-order functions.

A more specific advantage of GPs is their ability to represent procedural, generative solutions to pattern recognition and machine learning problems. Examples of this include image compression and reconstruction (Koza, 1992) and several of the recent applications surveyed below.

GP for Pattern Classification

GP in pattern classification departs from traditional supervised inductive learning in that it evolves solutions whose functional form is not determined in advance, and in some cases can be theoretically arbitrary. Koza (1992, 1994) developed GPs for several pattern reproduction problems such as the multiplexer and symbolic regression problems.

Since then, there has been continuing work on inductive GP for pattern classification (Kishore *et al.*, 2000), prediction (Brameier & Banzhaf, 2001), and numerical curve-fitting (Nikolaev & Iba, 2001, *IEEE Trans. Evol. Comp.*). GP has been used to boost performance in learning polynomial functions (Nikolaev & Iba, 2001, *GP & Evol. Machines*). More recent work on tree-based multi-crossover schemes has produced positive results in GP-based design of classification functions (Muni *et al.*, 2004).

GP for Control of Inductive Bias, Feature Construction, and Feature Extraction

GP approaches to inductive learning face the general problem of optimizing inductive bias: the preference for groups of hypotheses over others on bases other than pure consistency with training data or other fitness cases. Krawiec (2002) approaches this problem by using GP to preserve useful components of representation (features) during an evolutionary run, validating them using the classification data, and reusing them in subsequent generations. This technique is related to the wrapper approach to knowledge discovery in databases (KDD), where validation data is held out and used to select examples for supervised learning, or to construct or select variables given as input to the learning system. Because GP is a generative problem solving approach, feature construction in GP tends to involve production of new variable definitions rather than merely selecting a subset.

Evolving dimensionally-correct equations on the basis of data is another area where GP has been applied. Keijzer & Babovic (2002) provide a study of how GP formulates its declarative bias and preferential (search-based) bias. In this and related work, it is shown that proper units of measurement (strong typing) approach can capture declarative bias towards correct equations, whereas type coercion can implement even better preferential bias.

Grammar-Based GP for Data Mining

Not all GP-based approaches use expression tree-based representations, nor functional program interpretation as the computational model. Wong and Leung (2000) survey data mining using grammars and formal languages. This general approach has been shown effective for some natural language learning problems, and extension of the approach to procedural information extraction is a topic of current research in the GP community.

GP Software Packages: Functionality and Research Features

A number of GP software packages are publicly and commercially available. General features common to most GP systems for research and development include: a very high-period random number generator such as the Mersenne Twister for random constant generation and GP operations; a variety of selection, crossover, and mutation operations; and trivial parallelism (e.g., through multithreading).

One of the most popular packages for experimentation with GP is *Evolutionary Computation in Java*, or *ECJ* (Luke *et al.*, 2004). *ECJ* implements the above features as well as parsimony, “strongly-typed” GP, migration strategies for exchanging individual subpopulations in island mode GP (a type of GP featuring multiple *demes*, i.e., local populations or breeding groups), vector representations, and reconfigurabity using parameter files.

Other Applications: Optimization, Policy Learning

Like other genetic and evolutionary computation methodologies, GP is driven by fitness and suited to optimization approaches to machine learning and data mining. Its program-based representation makes it good for acquiring policies by reinforcement learning.[i] Many GP problems are "error-driven" or "payoff-driven" (Koza, 1992), including the ant trail problems and foraging problems now explored more heavily by the swarm intelligence and ant colony optimization communities. A few problems use specific information-theoretic criteria such as maximum entropy or sequence randomization.

FUTURE TRENDS

Limitations: Scalability and Solution Comprehensibility

Genetic programming remains a controversial approach due to its high computational cost, scalability issues, and current gaps in fundamental theory for relating its performance to traditional search methods, such as hill climbing. While GP has achieved results in design, optimization, and intelligent control that are as good as and sometimes better than those produced by human engineers, it is not yet widely used as a technique due to these limitations in theory. An additional controversy, stemming from the intelligent systems community, is the role of knowledge in search-driven approaches such as GP. Some proponents of GP view it as a way to generate innovative solutions with little or no domain knowledge, while critics have expressed skepticism over original results due to the lower human-comprehensibility of some results. The crux of this debate is a tradeoff between innovation and originality versus comprehensibility, robustness, and ease of validation. Successes in replicating previously-patented engineering designs such as analog circuits using GP (Koza *et al.*, 2003) have increased its credibility in this regard.

Open Issues: Code Growth, Diversity, Reuse, and Incremental Learning

Some of the most important open problems in GP deal with the proliferation of solution code (called code growth or code bloat), the reuse of previously-evolved partial solutions, and incremental learning. Code growth is an increase in solution size across generations, and generally refers to one that is not matched by a proportionate increase in fitness. It has been studied extensively in the field of GP by many researchers. Luke (2000) provides a survey of known and hypothesized causes of code growth, along with methods for monitoring and controlling growth. Recently Burke *et al.* (2004) explored the relationship between diversity (variation among solutions) and code growth and fitness. Some techniques for controlling code growth include reuse of partial solutions through such mechanisms as automatically-defined functions, or ADFs (Koza, 1994) and incremental learning – that is, learning in stages. One incremental approach in GP is to specify criteria for a simplified problem and then transfer the solutions to a new GP population (Hsu & Gustafson, 2002).

CONCLUSION

Genetic programming (GP) is a search methodology that provides a flexible and complete mechanism for machine learning, automated discovery, and cost-driven optimization. It has been shown to work well in many optimization and policy learning problems, but scaling GP up to most real-world data mining domains is a challenge due to its high computational complexity. More often, GP is used to evolve data transformations by constructing features, or to control the declarative and preferential inductive bias of the machine learning component. Making GP practical poses several key questions dealing with how to scale up; make solutions comprehensible to humans and statistically validate them; control the growth of solutions; reuse partial solutions efficiently; and learn incrementally.

Looking ahead to future opportunities and challenges in data mining, genetic programming provides one of the more general frameworks for machine learning and adaptive problem solving. In data mining, they are likely to be most useful where a generative or procedural solution is desired, or where the exact functional form of the solution – whether a mathematical formula, grammar, or circuit – is not known in advance.

REFERENCES

Brameier, M. & Banzhaf, W. (2001). Evolving Teams of Predictors with Linear Genetic Programming. *Genetic Programming and Evolvable Machines* **2**(4), 381-407.

Burke, E. K., Gustafson, S. & Kendall, G. (2004). Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation* **8**(1), 47-62.

Cramer, Nichael Lynn (1985), "A representation for the Adaptive Generation of Simple Sequential Programs" in *Proceedings of the International Conference on Genetic Algorithms and their Applications* (ICGA), Grefenstette, John J. (*ed*.), Carnegie Mellon University.

Hsu, W. H. & Gustafson. S. M. (2002). Genetic Programming and Multi-Agent Layered Learning by Reinforcements. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, New York, NY.

Keijzer, M. & Babovic, V. (2002). Declarative and Preferential Bias in GP-based Scientific Discovery. *Genetic Programming and Evolvable Machines,* **3**(1), 41-79.

Kishore, J. K., Patnaik, L. M., Mani, V. & Agrawal, V.K. (2000). Application of genetic programming for multicategory pattern classification. *IEEE Transactions on Evolutionary Computation* **4**(3), 242-258.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press.

Koza, J.R. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press.

Koza, J.R., Bennett, F. H. III, André, D., & Keane, M. A. (1999), *Genetic Programming III: Darwinian Invention and Problem Solving*. San Mateo, CA: Morgan Kaufmann.

Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., & Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. San Mateo, CA: Morgan Kaufmann.

Krawiec, K. (2002). Genetic Programming-based Construction of Features for Machine Learning and Knowledge Discovery Tasks. *Genetic Programming and Evolvable Machines* **3**(4), 329-343.

Kushchu, I. (2002). Genetic programming and evolutionary generalization. *IEEE Transactions on Evolutionary Computation*, **6**(5), 431-442.

Luke, S. (2000). *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, MD.

Luke, S, Panait, L., Skolicki, Z., Bassett, J., Hubley, R., & Chircop, A. (2004). *Evolutionary Computation in Java v*11. Available from URL: http://www.cs.umd.edu/projects/plus/ec/ecj/.

Muni, D. P., Pal, N. R. & Das, J. (2004). A novel approach to design classifiers using genetic programming. *IEEE Transactions on Evolutionary Computation* **8**(2), 183-196.

Nikolaev, N. Y. & Iba, H. (2001). Regularization approach to inductive genetic programming. *IEEE Transactions on Evolutionary Computation* **5**(4), 359-375.

Nikolaev, N. Y. & Iba, H. (2001). Accelerated Genetic Programming of Polynomials. *Genetic Programming and Evolvable Machines* **2**(3), 231-257.

O'Neill, M. & Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*.

Wikipedia (2004). *Genetic Programming*. Available from URL: http://en.wikipedia.org/wiki/Genetic_programming.

Wong, M. L. & Leung, K. S. (2000). *Data Mining Using Grammar Based Genetic Programming and Applications (Genetic Programming Series, Volume 3)*. Norwell, MA: Kluwer.

TERMS AND DEFINITIONS

Automatically-defined function (ADF): Parametric functions that are learned and assigned names for reuse as subroutines. ADFs are related to the concept of macro-operators or macros in speedup learning.

Code growth (code bloat): The proliferation of solution elements (e.g., nodes in a tree-based GP representation) that do not contribute towards the objective function.

Crossover: In biology, a process of sexual recombination, by which two chromosomes are paired up and exchange some portion of their genetic sequence. Crossover in GP is highly stylized and involves structural exchange, typically using subexpressions (subtrees) or production rules in a grammar.

Evolutionary Computation: A solution approach based on simulation models of natural selection, which begins with a set of potential solutions, then iteratively applies algorithms to generate new candidates and select the fittest from this set. The process leads toward a model that has a high proportion of fit individuals.

Generation: The basic unit of progress in genetic and evolutionary computation, a step in which selection is applied over a population. Usually, crossover and mutation are applied once per generation, in strict order.

Individual: A single candidate solution in genetic and evolutionary computation, typically represented using strings (often of fixed length) and permutations in genetic algorithms, or using “problem solver” representations – programs, generative grammars, or circuits – in genetic programming.

Island mode GP: A type of parallel GP where multiple subpopulations (demes) are maintained and evolve independently except during scheduled exchanges of individuals.

Mutation: In biology, a permanent, heritable change to the genetic material of an organism. Mutation in GP involves structural modifications to the elements of a candidate solution. These include changes, insertion, duplication, or deletion of elements (subexpressions, parameters passed to a function, components of a resistor-capacitor-inducer circuit, nonterminals on the right-hand side of a production rule).

Parsimony: An approach in genetic and evolutionary computation, related to "minimum description length", which rewards compact representations by imposing a penalty for individuals in direct proportion to their size (e.g., number of nodes in a GP tree). The rationale for parsimony is that it promotes generalization in supervised inductive learning and produces solutions with less code, which can be more efficient to apply.

Selection: In biology, a mechanism by which the fittest individuals survive to reproduce, and the basis of speciation according to the Darwinian theory of evolution. Selection in GP involves evaluation of a quantitative criterion over a finite set of fitness cases, with the combined evaluation measures being compared in order to choose individuals.

[i] *Reinforcement learning* describes a class of problems in machine learning in which an agent explores an environment, selecting actions based upon perceived state and an internal policy. This policy is updated based upon a (positive or negative) reward that it receives from environment as a result of its actions and seeks to maximize in the expected case.