The Evolution of Evolvability in Genetic Programming

Dr. Lee Altenberg, Research Associate

Department of Civil and Environmental Engineering,
Duke University, Durham, NC 27708-0287 U.S.A.
919/383-5327 Internet: altenber@acpub.duke.edu

Existing theory on the evolutionary dynamics of genetic programming (GP) is simply an extension of Holland's Schema Theorem to subtrees. In a classical fixed-length GA, the only way for a schema to increase in frequency is for the genomes carrying it to increase. In genetic programming, however, subtrees can increase in frequency by spawning multiple copies *within* programs. Such differential proliferation of subtrees within trees has already been observed in practice.

A consideration of the nature of subtrees that would proliferate within programs reveals that GP should naturally exhibit a property not found in classical fixed-length GAs: the evolution of evolvability. The evolvability of a Darwinian system is its ability to generate adaptive variants with non-vanishing probability. In GAs, the rate of generating adaptive variants is fixed at the outset by the relation between the objective function and the representation and operators used. This relation has been called the ‘‘adaptive landscape’’ or the genotype-phenotype map, which concerns the way genetic variability maps to phenotypic variability.

When a new subtree is added to a tree, it adds new degrees of freedom to the tree and to the gene pool of the population. The degrees of freedom change in two ways: the subtree itself may undergo allelic variation, and it is a new source for making additional copies.

Both of these provide means for enhancing the evolvability of the gene pool. Any subtree which is able to increase the fitness of trees it is added to with (relatively) high probability (given the task domain and the current diversity of trees in the population) will be the kind of subtree that proliferates. Thus in time, a growing proportion of subtree additions made by the genetic operator should involve subtrees with a high probability of producing a fitness increase. This is not simply an increase in the adaptedness of the trees in the population, but a change in the fitness effects of genetic variability, i.e. a change in the evolvability of the gene pool.

To analyze GP dynamics, two levels of fitness must be defined on subtrees. The probability that new copies of a subtree will produce fitness increases when added to the trees in the population (which will be referred to as its ‘‘constructional’’ fitness) is a different property from its current average fitness in the population (i.e. its schema fitness). In the population dynamics of genetic programming, the schema fitness will come to reflect the constructional fitness because the time scales of subtree duplication and allelic substitution are the same. However, in population dynamics resembling biological evolution, where the time scale of gene duplications is very much longer than that for allelic substitutions, these two fitness properties become clearly distinguished.

The second consequence of selective subtree addition is that, if there are correlations between the fitness effects of adding a subtree, and the effects of altering it, then the proliferation of constructionally-fit subtrees will also enhance the fitness effects of allelic variation within the subtrees.

These effects are developed into a theory of ‘‘constructional’’ selection for how the evolvability of the genome can increase as a result of genome growth. Several points emerge from this framework:

- In designing GP dynamics to enhance the evolution of evolvability, such as ‘‘automatic function definition’’ and ‘‘subtree compression’’, a distinction needs to be made between the variational and generative aspects of a tree. A selection strategy is suggested to help the dimensions of genetic variability become attuned to the dimensions of environmental variability.
- Theorems are derived in the case of biological dynamics showing the effect of genome growth on the evolvability of the genome.
- Furthermore, a theorem is derived, which is a direct extension of a basic result in theoretical population genetics, showing how the building block hypothesis and the claim for ‘‘implicit parallelism’’ in genetic algorithms require an implicit reduction in the complexity of the search space, through correlations between the fitnesses of parents and their recombinant offspring, applicable to GAs and GP, arbitrary operators, representations, and fitness functions.
- Application of this framework to messy GAs and other systems allowing genome growth is discussed.
- An illustration of constructional selection is given for the evolution of evolvability in Kauffman's ‘‘NK’’ adaptive landscape model under a more biological, rather than GA, evolutionary dynamics.

Genetic Programming and the Emergence of Knowledge

Peter J. Angeline
The Ohio State University
Computer and Information Sciences Department
228 Bolz Hall
2036 Neil Ave
Columbus, OH 43210
pja@cis.ohio-state.edu

It has become popular to incorporate task specific knowledge into the operators and representation of a genetic algorithm in order to improve performance. While it is important to incorporate such knowledge when it is available, to require such knowledge prior to running the genetic algorithm creates several problems. First, such knowledge often comes only at the expense of considerable analysis of the task. Second, it is well known that such knowledge introduces a problem specific bias to the algorithms. Third, the explicit incorporation of knowledge into the operators can lead to further problems. For instance, having a single source of knowledge forces its application to be uniform throughout the evolutionary process. This causes problems since the applicability of portions of the knowledge may change over the course of the run, much like the changing parameters of some GA systems (for instance Davis' OOGA). To compensate for this, knowledge about when given knowledge is applicable would need to be engineered out of the task as well. This leads to the same combinatorial proliferation of knowledge that has contributed to the difficulties found in knowledge-based artificial intelligence systems. It would be better to avoid explicit knowledge if possible in genetic programming.

This abstract demonstrates that task specific knowledge can emerge directly from the dynamics inherent in the genetic programming paradigm while evolving solution programs. By relying on emergent techniques, several benefits are won. First, task specific knowledge is not explicitly represented anywhere in the genetic program yet the structures which emerge are consistent with task specific constraints. Because the knowledge is not explicitly represented, there is no combinatorial explosion. The knowledge emerges through the interaction of the dynamics of the genetic program with the provided fitness function.

This concept will be demonstrated through examples from my previous work including the emergence of task specific subroutines described in GLIB and the emergence of goal directed behavior in a competitive GP environment.

Automatic Function Definition as a Means for Automatic Problem Decomposition

John R. Koza

Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305-2140
PHONE: 415-941-0336
FAX: 415-941-9430
Koza@CS.Stanford.Edu

It is often argued that the process of solving complex problems can be automated by first decomposing the problem into subproblems, then solving the presumably simpler subproblems, and then assembling the solutions to the subproblems into an overall solution to the original problem. The overall effort required to solve a problem can potentially be reduced to the extent that the decomposition process uncovers subproblems that are disproportionately easy to solve and to the extent that regularities in the problem environment permit multiple use of the solutions to the subproblems. Sadly, conventional techniques of machine learning and artificial intelligence provide no effective means for automatically executing this alluring three-step problem-solving process on a computer.

It is now clear that this three-step problem-solving process can be implemented by means of genetic programming and automatic function definition. Automatic function definition enables genetic programming to define potentially useful and reusable subroutines dynamically during a run. Recent work involving automatic function problems from the fields of Boolean function learning, symbolic regression, and control will be summarized. In each instance, the problem is automatically decomposed into appropriate subproblems; the subproblems are automatically solved; and the solutions to the subproblems are automatically assembled into a solution to the original problem. Leverage accrues because genetic programming with automatic function definition repeatedly reuses the solutions to the subproblems in the assembly of the solution to the overall problem. In particular, the number of fitness evaluations required to solve each problem is smaller with automatic function definition than without it. Moreover, automatic function definition produces solutions that are smaller in size than the solutions obtained without automatic function definition. Interestingly, the only exception appears to be problems that are so small and trivial that the overhead of solving them with automatic function definition erases the performance advantage of automatic function definition.

Alternatives in Automatic Function Definition: A Comparison of Performance

Kenneth E. Kinnear, Jr.
Sun Microsystems
2 Elizabeth Dr.
Chelmsford, MA 01824
kim.kinnear@sun.com
(508) 442-0318

Certainly one of the most challenging and potentially rewarding areas for research in genetic programming consists of finding ways to successfully apply it to larger and more complex problems. Progress is being made on a number of fronts in this general direction.

When using genetic programming, it quickly becomes clear that the primitives chosen for the problem representation have an enormous influence on how difficult the problem is to solve using GP. While it is frequently possible to choose a problem representation which makes an intractable problem tractable, doing so rarely generalizes into a technique which can be used on a different problem.

What is needed is automatic discovery of problem representations which make a given problem more tractable. This is analogous to the creation of subroutines or sub-functions when done by a human programmer. The same process can also be viewed as finding a way to decompose a problem into smaller, potentially solvable, subproblems.

Several approaches have been proposed to automatically perform these operations, and the work described here compares the performance of two of them on two separate problems: Automatic Function Definition [Koza 1993] and Module Acquisition [Angeline 1993]

Performance comparisons of the Automatic Function Definition (ADF) approach with the Module Acquisition (MA) approach, with each implemented as described by the references, shows that for the even-4-parity function the ADF approach yields significant increases in performance, while the MA approach does not show any improvement. At first glance, both approaches seem fundamentally similar though different in many details, and one might expect them to show similar increases in performance, yet they do not.

Both approaches are then augmented to allow use with iterative and conditional operators, and then applied to the problem of evolving general sorting algorithms. In this case, neither the ADF nor MA approach showed any increase in performance.

Following this comparison, the individual differences between the two approaches are systematically explored in order to find the key factor in the standard ADF approach which increases its performance in the examined problems so substantially over that of the standard MA module acquisition.

The MA approach requires fewer human choices and therefore has the possibility to exploit less domain specific knowledge than the ADF approach, yet this seems inadequate to explain the difference in performance.

One possibility, that the MA approach allows terminals in the modules while the ADF approach can (and does) restrict the appearance of terminals in the automatically defined functions, has been shown not to be significant.

Other differences that are analyzed include the difference between the way that ADF approach continuously evolves the ADF functions while the MA approach essentially freezes the modules it defines, as well as the local definition of ADF functions versus the global definition of MA modules.

Koza, J. R. (1993) “Hierarchial Automatic Function Definition”, in Foundations of Genetic Programming 2, Whitley, L. D., Ed., San Mateo, CA: Morgan Kaufmann

Angeline, P. A. (1993), Pollack, J. “Coevolving High Level Representations”, in Artificial Life III, Langton, C. G., Ed., Reading, MA: Addison Wesley [to appear]

Scalability, Generalization, and Breeding Schemes in Genetic Programming: The Donut Problem

Walter Alden Tackett

University of Southern California

Hughes Missile Systems Co.

tackett@ipld01.hac.com

Aviram Carmi

California State University at Northridge

Hughes Missile Systems Co

gpc-avc@ipld01.hac.com

In order for Genetic Programming (GP) to be useful as an engineering tool it must work on difficult problems. How does performance of GP change as we make the problem more difficult? Or easier? It is hard to answer these questions when we have no control over the data: indeed, it is important to understand that the notion of difficulty is problem-specific. In the research described here we formulate a problem so that the factors which make it hard are quantifiable and therefore scalable.

The "Donut Problem" requires separating two toroidal distributions (classes) which are "interlocked" like links in a chain. For each class, samples are generated by choosing a point p' from a uniform distribution lying on a circle of radius 1.0. Next, we construct a plane perpendicularly bisecting the circle which passes through it's center and through p'. The final coordinates are chosen by generating a point which lies in this plane whose distance from p' is gaussian distributed with standard deviation "sigma." For "Class A" the circle containing p' lies in the X-Y plane with its center at the origin. For "Class B" the circle lies in the X-Z plane centered at (1, 0, 0). This problem possesses a variety of pathological traits: the mean of each distribution, for example, lies in the densest point of the other.

We use GP to generate functions which create a single output given the (X,Y,Z) coordinates of a sample, with the value being greater than or equal to 0 for samples from Class A, and less than 0 for samples from Class B. Because this is a classification problem, we employ two sets of data. The first set is used for "training." An independent "test set" is used to test the ability of a function to generalize. Generalization refers to the correct classification of new samples which were not used in the training process: this is the critical requirement of a classifier.

The difficulty of this problem is scalable in two different ways: 1) Overlap: as we increase the standard deviation (i.e., "cross section") of the donuts, points from the two distributions "intermingle" so that even a classifier which optimally minimizes misclassification will still make mistakes in classifying "outlying" points; 2) Generalization: by varying the sparseness of the training set, we control the degree to which it is statistically underrepresentative.

We observe the effects of different breeding strategies upon performance under varying conditions, including panmictic vs. distributed evolution ("demes") with and without steady-state elitist population policies. For distributed breeding, we investigate different sizes and configurations for breeding units. In all, over 500 runs are performed, each with a population of 1024 individuals over a period of 50 generations (be sure to read our next paper, "System Resource Loading and Stress Related Disorders in Data Processing Management Professionals").

Preliminary analyses indicate that GP classification performance is directly proportional to class overlap and that there is a low sensitivity to sparseness of training samples, indicating good generalization and scaling properties. Distributed evolution creates markedly better training set performance than panmictic, although generalization (i.e., performance against the test set) is often only slightly improved. While speeding convergence, a steady-state elitist reproduction policy can be detrimental to generalization through early "overfitting" when training data is sparse.

The Evolution of Mental Models

Astro Teller
550 Middlefield Rd.
Palo Alto, CA 94301
(415) 326-6658
teller@cs.stanford.edu

Most interesting problems do not have solutions which are mappings from the inputs to correct outputs. That is to say that purely reactive systems are not sufficient to solve most interesting problems; some kind of internal state or memory is needed to operate well or optimally in these domains. Traditionally, Genetic Programming (GP) has concentrated on solving problems in the functional/reactive arena. This may be due in part to the absence of a natural way to incorporate memory into the paradigm. Memory and state have been used in some successful GP research efforts, but only as a device to make the task possible. Memory has not yet been integrated as a general, malleable concept that can evolve to suit the situation.

This paper proposes a simple addition to the GP paradigm that seamlessly incorporates the evolution of the effective gathering, storage, and retrieval of arbitrarily complicated state information. A new environment is presented in which agents have a limited time in which to explore a grid and push boxes to advantageous locations given only local sensing. The paper then evaluates several benchmarks against this additions to the paradigm in the context of this environment. Experimental results show that the effective productions and use of complex "mental models" can be evolved and that agents using mental models quickly and permanently displace purely reactive and non-deterministic functions.

These mental models appear because, unlike previous uses of state in GP, the proposed addition to the paradigm allows the development of more complicated uses of state. Though there is no clear division between state and mental models, the paper details several behaviors of highly evolved agents based arguably not on simple state changes, but on some model of the environment in which their fitness is evaluated. These results promise not only to aid in future research efforts into the causes and constituents of mental models but to expand the types of problems that can be practically tackled by Genetic Programming.

Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions

Craig W. Reynolds
Electronic Arts
1450 Fashion Island Boulevard
San Mateo, CA 94404
415-513-7442
creynolds@ea.com (or cwr@red.com)

Previous work has shown that the Genetic Programming Paradigm can be used to automatically create programs which enable a simple moving 2D vehicle to avoid collisions with obstacles by mapping sensory input (range data) into motor output (steering action). However, those evolved control programs are extremely "brittle." They solve only the specific problem for which they were evolved. That is, they can only guide the vehicle through the obstacle course(s) used to evaluate their fitness during evolution. They do not solve the general obstacle avoidance problem. They can not guide the vehicle through any other obstacle course. They will not even work in the **same** obstacle course if any of the vehicle's initial conditions (e.g. position or orientation) is slightly perturbed. These brittle controllers are a bit like a "house of cards" which stands only as long as absolutely nothing changes.

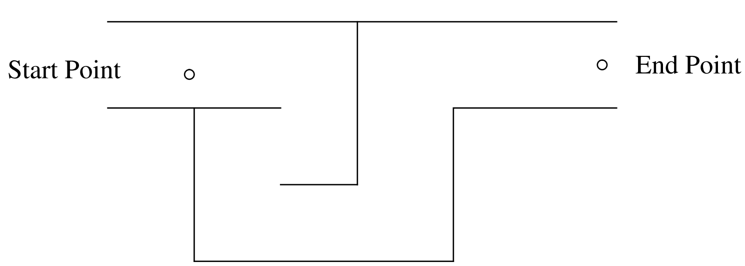
The crux of the problem is that evolution will discover solutions which capitalize on the deterministic, precisely repeatable nature of the fitness tests. Evolved controllers will come to depend on utterly insignificant coincidental properties of the vehicle's sensors, its actuators, and their interaction with the obstacle course.

The current work concerns one approach to avoiding this brittle behavior and seeks to evolve robust, general purpose control programs for the obstacle avoidance problem. The basic idea is to remove determinism from fitness testing by injecting noise into the system. This noise will tend to "jiggle" coincidental relationships between elements of the system and so tend to discourage evolution from capitalizing on them. Now the "house of cards" must be built on a shaky table. Clearly this becomes a much more difficult task, but if it can be accomplished the evolved controller should be a robust solution to the problem.

Noisy fitness tests present new problems. Multiple trials are required to separate noise from underlying fitness. Scores from multiple trials must be combined into an overall score. These experiments have included both taking the **minimum** of the trials (to encourage robustness) and taking the **average** of the trials (to encourage partial solutions).

The GP function set for these experiments include basic arithmetic operations, a sensor using ray-tracing to find the range to the nearest obstacle in a specified direction, and an effector which "steers" by a given angle. The vehicle always moves forward at a constant rate, so steering is its only means of avoiding collision. Raw fitness is simply the number of steps the vehicle moves before colliding with an obstacle.

These experiments have not yet produced a robust controller. But a host of lessons have been learned, and one by one, impediments to a solution have been eliminated. These issues and solutions will be presented along with results related to the interaction of noisy fitness functions with number of trials, methods of combining scores from multiple trials, elitism, and population sizes for Steady State Genetic Programming.



Genetic Programming Using Abstract Syntax Trees

Pawel A. Stefanski
Department of Computer Science
George Mason University
stefan@aic.gmu.edu

There are two general ways of incorporating domain or problem-specific constraints into the genetic algorithms (GA). One method is to constrain the representation used by GA and define genetic operators in such a way that they preserve selected (syntactic or semantic) constraints of the problem. Another is to build the constraints into the fitness function and let the natural selection do the job of eliminating incorrect individuals. An obvious drawback of the second approach is that (at least initially) the GA is asked to search significantly larger space than in the first method (see [De Jong 87]). These considerations become especially important when the structures to be evolved are computer programs or similar complex entities. Original formulation of Genetic Programming approach [Koza] addresses only the syntactic constraints and even this in an indirect way, since genetic operator definitions have to be explicitly modified each time new syntactic construct is introduced. All other types of constraints are left to be enforced by fitness function and natural selection.

We take a different approach, following [Antonisse 87], [Michalewicz 92], and many others. Specifically we argue, that in order to evolve truly complex programs, the genetic operators must be grounded in the basic software engineering knowledge, and directly obey not only syntactic but also (as much as possible) semantic constraints. Since the structures to be evolved are computer programs, they may be conveniently represented as Abstract Syntax Trees (for a discussion of advantages of abstract versus concrete syntax, see [Meyer 90]). ASTs have long been recognized as a convenient way of manipulating programs in compilers, syntax-directed editors, and so on. These ASTs play the role of genotypes, and are mapped to a concrete syntax (phenotype) for evaluation purposes. They also provide a framework, based on which other search-constraining information is defined. This information currently includes static semantics and local redefinitions of genetic operators. Static semantics is "a description of the structural constraints that cannot be adequately captured by syntax descriptions" [Meyer 90]. This usually includes context-sensitive constraints, such as type compatibility of expressions or multiplicity of declarations. It is specified by "decorating" the abstract syntax productions with type information, which is then used by genetic operators. These operators are also associated with productions in the abstract syntax grammar. Their default definitions can be overwritten by associating new definitions with lower-level non-terminal symbols. These new (re-)definitions, if provided, are then used at the recombination/mutation time.

As an example of a new operator definition let us consider mutation. As originally defined by Koza this operator swaps a randomly selected sub-expression for another one, also generated randomly. This definition is not consistent with the typical "local" character of mutation - and results in significant changes to an individual. We suggest that this global destructiveness is the primary reason for the mutation not being used in any examples presented in [Koza 92]. It is a well-established fact in the GA community, that completely barring the mutation from the evolutionary process introduces the effect of loss of diversity. Our experiments confirmed this also for the GP method. We present quite different definition of a mutation, which is more compatible with the original idea of "local perturbation". It is based on the abstract syntax productions, and in its default form relies on the replacement of a randomly selected sub-tree by another one, generated from the same non-terminal as the original one. Initial experiments (in the Boolean functions domain) suggest that this leads to much improved performance for smaller population sizes (~50-100). This in turn should allow to tackle more complex problems without increasing the population size.

Our future research will concentrate on expanding this paradigm by adding more semantics-based constraints in the form of restricted pre- and post-conditions. Genetic operators will be modified to assure the consistency (or near-consistency) between sub-expressions being transformed.

Selected (sic!) References

Antonisse, J., and Keller, K.S. 1987. Genetic Operators for High- Level Knowledge Representations. ICGA-87.

De Jong, K.A. 1987. On Using Genetic Algorithms to Search Program Spaces. ICGA-87.

Koza, J.R. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, Massachusetts.

Meyer, B. 1990. Introduction to the Theory of Programming Languages. Prentice Hall.

Michalewicz, Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag.

Genetic Programming in C++: Implementation and Design Issues

Mike J. Keith
keithm@icd.ab.com

Martin C. Martin
University of Toronto
Martin@psych.Toronto.Edu

Genetic Programming is a general purpose machine learning paradigm in which a set of programs (called a "population" of "individuals") are mated and modified by simulated Darwinian evolution. As the number of uses for and twists on genetic programming grows, so does the need for a fast and flexible platform. The purpose of the current paper is to present the design and implementation of such a platform in C++, demonstrate it's efficiency and flexibility, and discuss the issues surrounding it.

We first consider the lower level implementation aspects of the platform, specifically, the genome interpreter. The fact that Genetic Programming is a computationally expensive task means that the overall efficiency of the platform in both memory and time is crucial. In particular, the per-node representation is the key part of the implementation in which the overhead will be magnified.

We therefore introduce implementations in which the representation of the program tree is a linear array of nodes as opposed to a traditional (pointer based) tree structure. We introduce a novel prefix/jump-table (PJT) approach which results in a very small per-node overhead in both time and space compared to a traditional tree-structure implementation. In addition to being efficient, our interpreter is also very flexible. We discuss approaches for handling flow control and encapsulation as well as trade-offs with other linear representations, namely postfix and arbitrary positioning of functions and their arguments.

Secondly, we explore the higher level implementation issues surrounding a Genetic Programming Platform and propose a design where each independent mechanism and implementation issue of the GP algorithm is given a separate class and standard interface. This modular approach allows each such mechanism to be dynamically configured, modified, or even totally re-implemented with minimal effect on the other components of the simulation engine. We get this flexibility with no significant loss of runtime efficiency compared to an inflexible, specially designed platform. We have found that our component-based approach allows for maximum extensibility, debugging ease, and experimentation of alternative strategies for Genetic Programming in general.

A Compiling Genetic Programming System that Directly Manipulates the Machine Code

Peter Nordin

DaCapo AB

Chalmers Science PARK

412 88 Goteborg Sweden

Tel: +46 31 7724251

Fax: +46 31 411683

Peter.Nordin@gbg.infolog.se

Most genetic programming approaches uses a technique where a problem specific language is executed through an interpreter. The individual code segments in the population are decoded at run time through a virtual machine. The data structures that constitute the programs often have the form of a tree. This solution gives good flexibility and the ability to customize the language depending on constraints of the problem at hand. The disadvantage of this paradigm is that interpreting the program involves an overhead. There is also often a need to define more complicated genetic operators, which also decreases performance. The more complicated data structures, using pointers, makes execution less efficient as well.

We have evaluated the idea of using the lowest level machine code as the “programs” in the population. Every individual is a piece of machine code that is directly called and manipulated by the genetic operators. There is no interpreting part of the program. The data structures making up the population is directly called as standard C-functions. It’s a “compiling” genetic programming approach.

If you put enough effort into assembler programming and heuristics, you realize that many time critical tasks could be solved using just a hand full of carefully selected machine code instructions, performing for instance a heuristic classification. Such a task could be solved by a small set of low level additions, shifts, subtraction’s and by making a couple of XOR, AND, OR, etc. Replace the assembler “guru” you need, with a standard genetic algorithm and you have the approach we have investigated. Anyway, what does a computer do best, if not execute its own low level native machine instructions? And what are genetic algorithms good at, if not to blindly search a complex problem space.

The idea could be used on a generic level and with a wide range of possible implementations and genetic algorithm flavors, but we have narrowed down the possibilities to achieve a fast and simple solution.

Some technicalities are necessary to get a good grip of the basic ideas. This kind of system is most naturally implemented using a compiling language with abilities to make manipulations on a low level, and we use the C-language. The population consists of an array of pointers to functions. The functions are initiated to a dummy function with fixed length and thus a known memory size and known positions of the instructions. (The programs are however not fixed length themselves since we use dummy instructions to limit the actual size.) These functions are then treated as ordinary arrays and the genetic operator acts on this data structure like any other array. To limit the search space and avoid complex control mechanism and thus achieve maximum performance, we have limited the set of machine instructions in a number of ways. We use only machine code instructions that operates internally in the processor and takes one argument from memory and performs an operation on this argument and a processor register and leaves the result in a register. With this kind of instructions it’s possible to touch the maximum processor throughput of instructions per second. An average individual program for instance consisting of a dozen instructions of this type is shown to be executed in a fraction of a micro second on a 486 PC and the task that the genetic programming aim at is performed in the same amount of time. We have also constrained the problems to those who could be solved by a function taking an integer as argument and returning an integer. This does however not limit the problem space to mathematical problems as illustrated below. No control instructions like jump instructions are allowed which means that no loops can be formed. These limitations reduce the complexity and thus execution time of the individual programs. The genetic operators are the simplest possible. A cross-over mechanism acts on the byte level of the arrays/functions and the mutation operator is a random function at the byte level. The mutation function also assures that no illegal instructions will be created. The selection mechanism is simply to let a percentage of the best performing part of the population breed.

We also investigate different methods to make the implementation less machine dependent.

The application of this technique intersects with those problems normally addressed by connectionist approaches. We have investigated and compared the behavior of these two paradigms on an instance problem. This problem is a classification problem and the goal is to find a heuristic way to determine whether a word is noun or not. We have a training set consisting of a couple of thousand short nouns. All the words are coded to a 32 bit integer, using 5 bits per letter, in the genetic programming example. The C-functions in the population take this 32 bit integer, performs a set of integer operation, and returns a value. A non-zero value means that the individual program believes that this word is a noun. The individuals are judged by the number of right classifications in the training set. The preliminary results show that training time is of the same order as with the neural network paradigm, using momentum back propagation. The classifications also have the same success rate, but execution time and storage consumption are a magnitude lower with this genetic programming approach.

Genetic Programming for the Acquisition of Double Auction Market Strategies.

Martin Andrews
mdda@eng.cam.ac.uk

Richard Prager
rwp@eng.cam.ac.uk

Cambridge University Engineering Department
Trumpington Street
Cambridge England

The minute-by-minute trading on futures and commodity exchanges can be modelled as a Double Auction Market. Since 1990, Double Auction Tournaments have been held at the Santa-Fe Institute, Arizona. The competitors are strategies embodied in computer programs written by a variety of economists, computer scientists and mathematicians.

All but one of the computer strategies were explicitly hand-coded.

The automatic acquisition of strategies for Double Auction trading has obvious benefits both theoretically, and financially.

> [Note: Since January 1993, human players have also been competing against machine players over the Internet: The AZTE (AriZona Token Exchange).]

This paper describes how Genetic Programming methods have been used to create strategies superior, in local double auction playoffs, to many of the hand-coded strategies submitted to the earlier tournament.

A comparison is made between GP and Simulated Annealing on the programs, to isolate the contribution that the evolutionary process makes to the search for good strategies.

A comparison is also made between those programs that are only composed of numeric function calls to those that include functions with side-effects.

There is also an investigation into the role of redundant function subtrees, in an attempt to look at the Genetic Programming paradigm from the point of view of the function-subtree genes rather than that of the individual.

Later work will include environments in which GP evolved competitors are allowed to trade against each other whilst the evolutionary process continues.

The Genetic Planner: The Automatic Generation of Plans for a Mobile Robot Via Genetic Programming

Simon Handley
Computer Science Department
Stanford University
Stanford, CA 94305
(415) 723-4096
shandley@cs.stanford.edu

Planning is the creation of programs to control an agent, such as a robot. Traditionally, planners have maintained a logical model of the agents world and planned by reasoning about what plans do to that world. The Genetic Planner uses artificial selection, sexual mixing (recombination) and fitness proportionate reproduction to breed computer programs (i.e., to plan). The Genetic Planner uses a simulation of the world to execute candidate computer programs (i.e., candidate plans). This paper describes The Genetic Planner and shows it at work on a simple problem---a robot on a 2-D grid.

Using Genetic Programming for Decision Tree Induction for Natural Language Processing

Eric Siegel,

Columbia University
145 W. 67th St. 24F
New York, NY 10023
(212) 724-3846
evs@cs.columbia.edu

Decision trees have the advantage that they can be easily understood. Also, they can represent a set of generalizations in a succinct form. We have successfully implemented decision tree induction via genetic programming for the problem of word sense disambiguation. Our work combines GP and GA crossover procedures. Competition (inspired by Hillis' "Coevolving Parasites", AL-II) has proven to be very important in this work.

In natural language, many words have multiple senses. For example, "anyway" can mean "in any case, as in, "I did it anyway." It can also mean, "Let's return to a previous topic", as in "Anyway, what were you saying before?"

We induce decision trees which, for a subclass of words, determine the sense in which a word is being used by looking at its context. In such a decision tree, internal nodes have sets of attributes. The attributes are the token positions -1 through 4 with respect to the word being disambiguated, as well as the word itself. Branches have sets of values. In our domain, values are actual tokens (words and punctuation marks). Leaves contain conclusions -- A symbol representing a word sense. Therefore, for a given occurrence of a word, the tree is traversed deterministically from root to leaf, thus deriving the pertinent sense.

This approach allows us to derive a lower bound for the usefulness of decision trees with such "limited vision".

Note that we allow for a variable number of branches. Also note that sets (of attributes/positions and values/words) are embedded in our tree representation. These sets are represented as a constant-length bit string. Each word in the superset of words has a corresponding position in the string. If a word's position in the string is 1, the word is in the set, else it is not. We intermingle GA style crossover of these bit strings with the usual GP style crossover of the entire trees.

We have 779 manually derived cases, marked by a computational linguist (from Litman, Hirschberg 90). We use half for training, half for testing. There are 34 words in the class we are disambiguating (discourse cue words).

When we first ran the system, it would never induce trees with depth more than 2. This was because, for this domain, there is a 3 node decision tree which attains approximately 75% accuracy. This tree would always dominate the population. Inspired by (Hillis, A-Life III), we introduced competition. Each training case is given a score - the number of times it has foiled a decision tree. When testing a decision tree, two-member tournament selection is used repeatedly to select 350 (possibly duplicate) training cases.

Here is an example:

289/339 Training cases correct = 85.1252%

295/340 Test cases correct = 86.8925%

```
; A - attribute
; V - value
; Discourse/Sentencial - These are the possible word senses
; They appear at the leaves of the tree
A: -1                             ; First look at preceding token
   V: is you ,
   A: word                        ; The word being disambiguated
     V: look or but so say well now
     A: -1
       V: is you ,                ; Note there is some redundancy
       Discourse (16)             ; This leaf denotes "Discoures" sense
```

```
    V: <Anything>                ; Default branch
    Sentential (13)
  V: . and
  Discourse (87)
  V: to of I
  Sentential (17)
  V: <Anything>
  A: word
    V: or but so say now
      A: word
        V: or but so now
        Sentential (34)
        V: <Anything>
        Discourse (10)
    V: <Anything>
    Sentential (112)
```

The numbers in parentheses appearing beside leaf nodes are the number of times the leaf contributed a correct conclusion over the entire training set. To improve the likelihood that a tree encaptures true generalizations, and not idiosyncrasies of the training set, there is a constraint on the minimum value of these numbers (5).

The tree above has automatically derived linguistic data: If the preceding word is “to”, as in “to say”, “say” must be a verb, as in “That is what I heard him say.” (“Sentencial” use.) “Say” can also mean “for example”, as in, “If the system attempts to hold rules, say as an expert database for an expert system, then we expect it not only to hold the rules but to in fact apply them for us in appropriate situations.” (“Discourse” use.)

Genetic Programming of Neural Networks

Frédéric Gruau

Laboratoire de l'informatique du parallélisme,
Ecole Normale Supérieure de Lyon 47 Allé d'Italie 69007 Lyon
gruau@drfmc.ceng.cea.fr
Centre d'étude nucléaire de Grenoble
DRFMC SP2M PSC, BP85X 38041 Grenoble France

Cellular Encoding is a method for encoding families of boolean neural networks having the same structure, that can compute scalable boolean functions. The object that is encoded is a graph grammar. The grammar is encoded as a tree (called *grammar tree*) instead of a set of rules. A Genetic Algorithm is used to search the space of grammar-trees. It can find part of a modular code that yields both architecture and ±1 weights specifying a particular neural network for solving the parity, symmetry decoder boolean function of arbitrary large size.

Koza has shown that the tree data structure is an efficient encoding which can be effectively recombined by genetic operators. *Genetic Programming* has been defined by Koza as a way of evolving computer programs with a genetic algorithm. In Koza's Genetic Programming paradigm the individuals in the population are LISP S-expressions which can be depicted graphically as rooted, point-labeled trees with ordered branches. Since our chromosomes (grammar-trees) have exactly the same structure, the same approach is used for generating and recombining grammar trees. The current study replaces the genetic search of cellular encoding into the Genetic Programming frame using Koza formalism. The particular Genetic Programming scheme used is reported, differences and similarities with Koza's scheme are stressed.

The combination of Genetic Programming with neural network using cellular encoding is totally different from the one already proposed by Koza and Rice. Here, it is not a program that simulates a neural network, but neural networks that behave in a program-like manner. Modulo this new kind of combination, Genetic Programming of neural networks allows to combine a symbolic search lead by the GA, with a Connectionist search obtained through learning. It is a natural and elegant solution to the fundamental problem of building neural networks that can process symbols, without an **explicit mapping** from symbols to neurons or neuron-states. Here, the method exploits all the magic of Genetic Programming that builds a different sort of program without using an **explicit mapping** of a target problem to an algorithm. Another interest of the method is to incorporate modularity into neural networks. Modularity is **the property** of programming. It is shown how Genetic Programming of neural networks can build increasingly complex modular neural networks.

Optimizing Confidence of Text Classification by Evolution of Symbolic Expressions

Brij Masand
Thinking Machines Corporation
245 First Street
Cambridge MA 02142 USA
brij@Think.COM

This paper reports some initial experiments in applying genetic algorithms for assessing the confidence of automatically assigned multiple keywords for news stories. Using Memory Based Reasoning (MBR) (a k-nearest neighbor method) to classify the stories, we would like to assign a confidence score per news story, that allows one to refer stories with low classification confidence to a human coder. Using Genetic Algorithms (GA) as used for program evolution by [Koza], we discover and evolve symbolic expressions to compute confidence scores for news stories that allow a higher performance on subsets of the database while referring some stories to human editors. We have earlier reported recall and precision of 81%and 72%, if 100% of the stories are coded automatically [Masand]. Using the evolved confidence measures to refer some stories for manual coding, we can achieve about 80% recall and 80% precision for 92% of the stories. This compares favorably with manually specified confidence functions that could classify 76% of the database with an 80-80% recall-precision requirement.

Meta GA, Desktop Supercomputing, and Object-Oriented GP

Andrew Singleton
Creation Mechanics
RFD 2, Box 144W
Peterborough, NH 03458
(603) 532-4011
p00396@psilink.com

Instead of trying to cover theoretical issues in 10 minutes, which might be impossible, I will make a rapid fire presentation of some practical tools which we have been working on at Creation Mechanics. This will touch on:

- Meta GA - why it is necessary for GP, and what it can do, including primitive selection, parameter tuning, and generalization.
- Desktop supercomputing - A distributed GA algorithm which scales reliably from a single PC to an enterprise LAN to racks of several hundred processors, using the same piece of software.
- An object oriented GP architecture which separates and modularizes into three pieces the GA and population, the GP representation, and the specific problem, fitness function and primitives.
- Future architectures for “feedforward” GP and the GP factory.
- A demo of the “Doodle Garden” GP toy.

A Hierarchical Genetic System For Symbolic Function Identification

Mingda Jiang
Unidata Denver, CO
(303) 394-0800
umj01@unidata.com

Alden H. Wright
Department of Computer Science
University of Montana
Missoula, MT 59812 (406) 243-4790
wright@cs.umt.edu

Abstract Given data in the form of a collection of (x,y) pairs of real numbers, the symbolic function identification problem is to find a functional model of the form y = f(x) that fits the data. This paper describes a system for solution of symbolic function identification problems that combines a genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm. The genetic algorithm uses an expression-tree representation rather than the more usual binary-string representation. Experiments were run with data generated using a wide variety of function models. The system was able to find a function model that closely approximated the data with a very high success rate.

This is the abstract of Mingda Jiang's M.S. thesis done under direction of Alden Wright.

Versions have already been presented at two conferences and will appear in the proceedings of those conferences:

1. "A Hierarchical Genetic System for Symbolic Function Identifications", Proceedings of the 24th Symposium on the Interface: Computing Science and Statistics, College Station, Texas, March 1992

2. "An Adaptive Function Identification System", Proceedings of the IEEE/ACM Int'l Conference on Developing and Managing Intelligent System Projects, Vienna, Virginia, March 1993

Statistical Investigations of Genetic Algorithms and Genetic Programming

Nick Keenan
c/o Creation Mechanics
RFD 2, Box 144W
Peterborough, NH 03458
(603) 532-4011

The statistical properties of each step in a genetic algorithm, roughly classified as selection and reproduction, can be measured experimentally, and the statistical properties of the entire process can be estimated from the properties of the component steps. Knowing the contribution of each step to the fitness mean and variance enables a user to design a process that is appropriate for the problem at hand, to measure the effect of changes in the process, and to make predictions about the efficiency, effectiveness, and requirements of a given search procedure. Questions about population size, number of generations, and chances of success can be addressed. This analysis implies that there is a finite limit to the effectiveness of genetic programming.