An Analysis of Hierarchical Genetic Programming

Justinian P. Rosca
rosca@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 566

March 1995

Abstract

Hierarchical genetic programming (HGP) approaches rely on the discovery, modification, and use of new functions to accelerate evolution. This paper provides a qualitative explanation of the improved behavior of HGP, based on an analysis of the evolution process from the dual perspective of diversity and causality. From a static point of view, the use of an HGP approach enables the manipulation of a population of higher diversity programs. Higher diversity increases the exploratory ability of the genetic search process, as demonstrated by theoretical and experimental fitness distributions and expanded structural complexity of individuals. From a dynamic point of view, this report analyzes the causality of the crossover operator. Causality relates changes in the structure of an object with the effect of such changes, i.e. changes in the properties or behavior of the object. The analyses of crossover causality suggests that HGP discovers and exploits useful structures in a bottom-up, hierarchical manner. Diversity and causality are complementary, affecting exploration and exploitation in genetic search. Unlike other machine learning techniques that need extra machinery to control the tradeoff between them, HGP automatically trades off exploration and exploitation.

This material is based on work supported by the National Science Foundation under grant numbered IRI-9406481 and by DARPA research grant no. MDA972-92-J-1012. The government has certain rights in this material.

1 Introduction

The problem of understanding and controlling the mechanism of genetic programming (GP) is challenging especially in the case of GP extensions for the discovery and evolution of functions. Such GP extensions have been designed with the goal of automating the discovery of functions that are beneficial during the search for solutions by exploiting opportunities to parameterize and reuse code. Two such techniques are *automatic definition of functions* (ADF) [Koza, 1992] and *adaptive representation* (AR) [Rosca and Ballard, 1994a]. The former is a GP extension that allows the evolution of reusable subroutines. The latter is based on the discovery of useful building blocks of code. Although ADF and AR approaches implement the ideas of discovery, modification, and use of new functions in different ways, both actually evolve a hierarchy of functions that greatly improve search efficiency. This paper refers to both mechanisms by hierarchical genetic programming (HGP).

No clear mathematical analysis currently exists for how either GP or HGP sample the solution space. The goal of this paper is to analyze the influence of different representational choices on the behavior of GP. This paper analyses several explanations for the improved behavior of HGP due to function discovery and proposes a *bottom-up* HGP evolution scenario: HGP discovers and exploits useful structures in a bottom-up, hierarchical manner.

Two complementary dimensions of genetic search are discussed in the paper: diversity of programs and GP causality. Discovery and use of encapsulated subroutines causes increased population diversity. Experimental evidence outlining increased program size and varied program shape is presented to explain this increased population diversity. The paper compares theoretical and practical distributions of fitness for randomly generated solutions in a test problem characterized by a finite function sample space.

Causality relates changes in the structure of an object with the effect of such changes which represent changes in the properties or behavior of the object. The principle of *strong causality* states that small alterations in the underlying structure of an object, or small departures from the cause determine small changes of the object's behavior, or small changes of the effects, respectively ([Rechenberg, 1994], [Lohmann, 1992]). In GP small alterations of the programs may generate big changes in behavior. From this perspective GP is weakly causal. In this report, the trend of structures called *birth certificates* are presented as evidence for the way HGP inherits useful structures. Birth certificates represent types of crossover in the genealogic tree of a solution and record the evolution trajectory of that solution.

The report outline is as follows. The next section defines the underlying principle of HGP and the resulting change in representation, then briefly presents the two HGP approaches used throughout the experiments and other related work. Section 3 introduces a test case and presents a theoretical analysis of fitness distributions for a uniform probability distribution of solutions. It analyzes the random generation of program trees and compares theoretical distributions of partial solutions with those actually obtained in GP for a varying function set. Changes in representation determine changes in the size, shape, and behavior of program trees. Section 4 presents an analysis of the GP evolution dynamics. The discovery of functions offers a means for expressing, combining, and propagating useful building blocks. Thus, it contributes in an essential way to the exploratory ability of GP. Discovered functions represent an adaptive control mechanism in the exploration-exploitation tradeoff. In conclusion the paper discusses the results and suggests future research.

2 Hierarchical Genetic Programming

Genetic programming departs from the genetic algorithm (GA) paradigm by using trees to represent genotypes ([Cramer, 1985], [Koza, 1992]). Trees provide a flexible representation for creating and manipulating programs. This paper uses the denotations *tree* and *subtree* to refer to the parse tree of a program or a part of it respectively.

Problem representation in GP is defined by a set of problem-dependent primitive functions. Functions of one or more variables label internal nodes of the tree while functions of no arguments, called terminals, label leaves of the tree. The search space for GP is the space of all programs that can be built using these initial primitives. The intuition for hierarchical GP systems is that adapting the composition of these sets dramatically changes the behavior of GP. For example, the inclusion of more complex functions, known to be part of a final solution, will result in less computational effort spent during search and thus will enable a shorter time to finding a final solution.

The HGP approaches presented below, automatic definition of functions (ADF-GP) [Koza, 1992] and adaptive representation (AR-GP) [Rosca and Ballard, 1994a] use the above observation in different ways in order to accelerate search.

2.1 Automatic Definition of Functions

The automatic definition of functions approach (ADF-GP) assumes that parsimonious problem solutions can be specified in terms of a main program and a hierarchical collection of subroutines. The main program invokes a subset of the subroutines to perform the overall computation, while those subroutines may in turn call other subroutines computing partial results.

Genetic programming is used both to search for appropriate subroutines, and to find a way of composing discovered subroutines and primitive functions into a complete solution. In this approach, apparently, GP has to perform a more difficult search task. The problem becomes well defined if the functions and terminals that can be invoked by each subroutine and by the main program are completely specified. During evolution, only the fitness of the complete program is evaluated.

In this approach each individual program has a dual structure. The structure is defined based on a fixed number of components or *branches* to be evolved: several function branches and a main program branch. Each function branch (called ADF_0, ADF_1, etc.) has a fixed number of arguments. The main program branch (*Program-Body*) produces the result. Each branch can be viewed a piece of LISP code built out of specific primitive terminal and function sets, and is subject to genetic operations. The set of function-defining branches, the number of arguments that each of the function possesses and the "alphabet" (function and terminal sets) of each branch define the *architecture* of a program. The references allowed

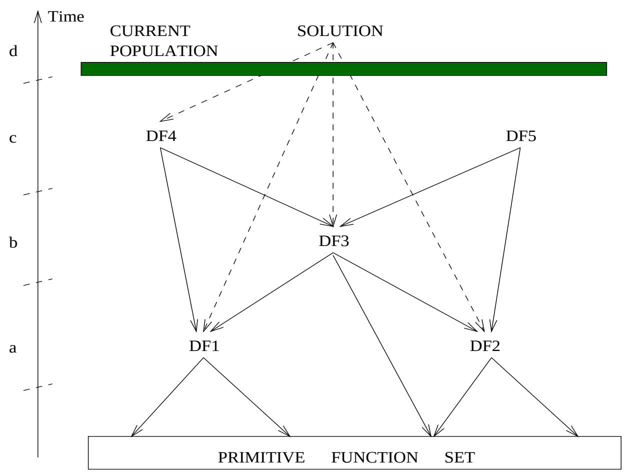

Figure 1: A hypothetical call graph of the extended function set in the AR method. The primitive function set is extended hierarchically with functions ($DF1, DF2$, etc.) discovered at generation numbers a, b, c. A solution is eventually found at generation d.

between function branches determine a hierarchical organization of the set of functions. Although the number and interconnectivity of ADFs are fixed, the definition of ADFs evolve. Genetic operations on ADFs are syntactically constrained by the components on which they can operate. For example, crossover can only be performed between subtrees of the same type, where subtree type depends on the function and terminal symbols used in the definition of that subtree. An example of a simple typing rule for an architecturally uniform population of programs is *branch typing*. Each branch of a program is designated as having a distinct type. In this case the crossover operator can only swap subtrees from analogous branches.

2.2 Adaptive Representation

In contrast to ADF's passive function definition, AR explicitly attempts to discover and use new functions. A hierarchy of automatic functions is created in a bottom-up fashion as the problem is being solved (see figure 1).

At the base of the function hierarchy lie the primitive functions from the initial function set. More complex functions are dynamically built on the primitive functions, and become stable components of the representation. The levels in the hierarchy are discovered by using either heuristic information as conveyed by the environment or statistical information extracted from the population. The heuristics are embedded in block fitness functions which are used to determine fit blocks of code. The hierarchy of functions evolves as a result of several steps:

1. Select candidate building blocks from fit small blocks appearing in the population
2. Generalize candidate blocks to create new functions

3. Extend the representation with the new functions, noticing if progress is made.

In order to control the process of function discovery, AR-GP keeps track of small blocks of code appearing in the population. A key idea is that although one might like to keep track of blocks of arbitrary size, only monitoring the merit of small blocks is feasible. Useful blocks tend to be small and the process can be applied recursively to discover more and more complex useful blocks. Consequently, AR-GP has a *bottom-up approach* to function discovery [Rosca and Ballard, 1994b].

The generation intervals with no function set changes represent evolutionary *epochs*. At the beginning of each new epoch, part of the population is extinguished and replaced with random individuals built using the extended function set [Rosca and Ballard, 1994a]. The extinction step was introduced in order to make use of the newly discovered functions.

The discovery of functions in AR can be guided by domain knowledge. Most generally, the population itself represents a pool of statistical information. Global measures such as the population diversity or local measures such as the differential fitness from parents to offspring can be used to guide the creation of new functions.

2.3 Other related work

Modularization is an approach which addresses the problems of inefficiency and scaling in GP. This issue has generated research efforts towards defining the notion of building block in GP and finding useful ways to manipulate modules of code.

A GP analogy along the lines of GA schemata theory and GA building block hypothesis has been attempted in [O'Reilly and Oppacher, 1994]. The main goal was understanding if GP problems have building block structure and when GP is superior to other search techniques. The approach was to generalize the definition of a GP schema from [Koza, 1992] to a collection of tree fragments, that is a collection of trees possibly having subtrees removed. An individual instantiates a schema in case it "covers" (matches) all the schema fragments, overlappings between fragments not being allowed. The probability of disruption by crossover is estimated based on these definitions. The authors concluded that schema analysis is difficult and does not offer an appropriate perspective for analyzing GP.

A GP structural theory analogous to GA schemata theory fundamentally ignores the functional role of the GP representation. The analysis of building blocks in AR-GP [Rosca and Ballard, 1994a] starts from this hypothesis and takes a functional approach. The ADF approach, presented earlier, is also a method of representing and using modularity in GP. Another method, *module acquisition* ([Angeline, 1994b], [Angeline and Pollack, 1994]) introduced many inspirational ideas. A module is a function with a unique name defined by selecting and chopping off branches of a subtree selected randomly from an individual. The approach uses the *compression* operator to select blocks of code for creating new modules, which are introduced into a genetic library and may be invoked by other programs in the population. Two effects are achieved. First the expressiveness of the base language is increased. Second modules become frozen portions of genetic material, which are not subject to genetic operations unless they are subsequently decompressed.

It has been conjectured that problems whose solutions present symmetry patterns or opportunities to parameterize and reuse code can be solved easier in ADF-GP [Koza, 1994b] but there exists no formal explanation of why ADF-GP works better than standard GP. [Kinnear, 1994] explains why ADF-GP works by introducing the notion of structural regularity. He compares ADF-GP against the module acquisition approach and points out that the module acquisition approach does not directly create structural regularity. Kinnear attributes the better performance of ADF to the repeated use of calls to automatically defined functions and to the multiple use of parameters.

The *lens effect* [Koza, 1994b] is the idea that the tails of the fitness distribution for randomly generated programs are larger for ADF-GP than for standard GP. The effect is attributed to the introduction of new functions into the representation. [Altenberg, 1994] outlines that a similar property should be observed in general in order to make GP search more efficient than random search: the upper tail of the offspring fitness distribution should be wider than that for random search.

The problem of determining the appropriate architectural choices in ADF-GP has generated work on evolution of the GP architecture. The architecture itself can be evolutionarily selected in case the initial population is architecturally diverse and care is taken when crossing over individuals having different architectures [Koza, 1994b]. [Koza, 1994a] introduces six new genetic operations for altering the architecture of an individual program: branch duplication, argument duplication, branch deletion, argument deletion, branch creation and argument creation. These operations are causal in the sense discussed later in this paper.

A rule of thumb in GA literature postulates that population diversity is important for avoiding premature convergence. A comparison of research on this topic is provided in [Ryan, 1994]. Ryan shows that maintaining increased diversity in GP leads to better performance. His algorithm is called "disassortative mating" because it selects parents for crossover from two different lists of individuals. One list of individuals is ranked based on fitness while the other is ranked based on the sum of size and weighted fitness. The goal is to evolve solutions of minimal size that solve the problem. However, by using directly the size constraint the GP algorithm is prevented from finding solutions. The algorithm improves convergence to a better optimum while maintaining speed.

Exploration and exploitation are recurring themes in search and learning problems [Holland, 1992], [Kaelbling, 1993]. Exploitation takes place when search proceeds based on the action prescribed by the current system knowledge. Exploration is usually based on random actions, taken in order to experiment with more situations. For example, in learning classifier systems, roulette wheel action selection is a means of choosing exploratory actions. In the reinforcement phase of the control loop of a classifier system [Wilson, 1994], matching classifiers that do not get activated are weakened. This lowers the chances of choosing unpromising actions in the near future. The weakening magnitude is usually controlled by an explicit parameter, although more elaborate schemes are possible [Wilson, 1994]. In contrast GP is a search technique that implicitly balances exploration and exploitation, as will be showed later.

3 The Role of the GP Representation

One goal of the paper is to analyzes the influence of different representational choices on the behavior of GP both theoretically and experimentally using a standard GP algorithm and HGP. The test case chosen is the parity problem. Parity is an attractive problem for several reasons. First it operates on a finite sample space, the space of Boolean functions with a given number of inputs. This enables the computation of distributions of interest for random choices of an initial population. Second, parity is difficult to learn because every time an input bit is flipped, the output also changes.

The ODD-n-PARITY problem is to find a logical composition of primitive Boolean functions that computes the sum of input bits over the field of integers modulo 2. EVEN-n-PARITY can be defined by flipping the result of ODD-n-PARITY. The ODD-n-PARITY and EVEN-n-PARITY functions appear to be difficult to learn in GP, especially for values of n greater than five [Koza, 1992].

The initial function set for the parity problem in GP is defined by the set of primitive Boolean functions of two variables:

$$\mathcal{F}_0 = \{AND, OR, NAND, NOR\} \tag{1}$$

The terminal set is defined by a set of Boolean variables:

$$\mathcal{T}_0 = \{D_0, D_1, D_2, ..., D_{n-1}\}$$

Any Boolean function of n variables is defined on the set of 2^n combinations of input values. Given a program implementing a Boolean function, its performance is computed on all possible combinations of Boolean values for the input variables and is compared with a table defining the EVEN-n-PARITY function. Each time the program and the EVEN-n-PARITY table give the same result, the program records a *hit*. The task is to discover a program that achieves the maximum number (2^n) of hits.

Theoretical analysis of uniform random sampling

The efficiency of a GP algorithm depends on the computational effort needed to evolve a solution with a given probability. Random search provides an upper bound on the effort needed. The probability of randomly generating a problem solution depends both on the initial function set, and on the method of generating random individuals. Our goal is to understand the influence of the function set composition, and consequently of a function discovery mechanism on this probability.

Let us consider the sample space of all functions

$$\mathcal{S} = \{f : \mathcal{B}^n \longrightarrow \mathcal{B}\}$$

where $\mathcal{B} = \{0, 1\}$. Note that $\|\mathcal{S}\| = 2^{2^n}$, thus we can obtain random elements of $\mathcal{S}$ by flipping 2^n distinct fair coins.

Consider the random variable X mapping the finite sample space $\mathcal{S}$ onto the set of positive integer numbers $\mathcal{N}$ defined as follows: X is the number of hits of a randomly generated Boolean function $s \in \mathcal{S}$.

We are interested in analyzing the probability mass function of X.

$$Prob\{X = x\} = \sum_{s \in \mathcal{S}: X(s) = x} Prob\{s\}$$

Consider a random [1] Boolean function with k hits. The k hits are due to i 1-hits and to $(k-i)$ 0-hits. EVEN-n-PARITY takes an equal number (i.e. $\frac{n}{2}$) of 0 and 1 values over the set of input binary strings. Thus, the number of Boolean functions that coincide with EVEN-n-PARITY for a fixed set of k input strings is [2]:

$$\sum_{i=0}^{k} \binom{\frac{n}{2}}{i} \binom{\frac{n}{2}}{k-i} = \binom{n}{k} \tag{2}$$

which implies that X has a binomial distribution, with $p = q = \frac{1}{2}$. It follows that

$$Prob\{X = k\} = \frac{1}{2^n} \cdot \binom{n}{k} \tag{3}$$

The expected value of X is $\frac{n}{2}$ and its variance is $\frac{n}{4}$ and can be computed as follows (or see [Cormen *et al.*, 1990]):

$$E[X] = \sum_{k=0}^{n} k \cdot Prob\{X = k\} = \frac{1}{2^n} \cdot (n 2^{n-1}) = \frac{n}{2} \tag{4}$$

$$\begin{aligned} Var[X] = E[X^2] - (E[X])^2 &= \frac{1}{2^n} \sum_{k=0}^{n} k^2 \binom{n}{k} - \frac{n^2}{4} = \\ &= \frac{1}{2^n} \left[((x+1)^n)''|_{x=1} + \sum_{k=0}^{n} k \binom{n}{k} \right] - \frac{n^2}{4} = \frac{n}{4} \end{aligned} \tag{5}$$

3.1 Program diversity

In order to understand the role of representation and the effect of dynamically changing it we designed a set of experiments for estimating qualitative measures of diversity such as fitness distributions and program size in GP and HGP.

A straightforward definition of diversity in GP is the percentage of structurally distinct individuals at a given generation. Two individuals are structurally distinct if they are not

[1] Here and in turn the term random refers to structures generated randomly according to a uniform probability distribution

[2] In order to prove equality 2 use Newton's binomial on both sides of the identity $(1+x)^n = (1+x)^{\frac{n}{2}}(1+x)^{\frac{n}{2}}$ and identify coefficients.

isomorphic trees. However, such a definition is not practically useful. It is computationally expensive to test for tree isomorphisms. Moreover, associativity of functions is extremely difficult to take into account.

An easily observable type of variation in the population is fitness diversity. Two individuals are different if they score differently.

Another useful qualitative measure of diversity is program size. In HGP, a true measure of the size of an individual is obtained by counting all the nodes in the tree resulting after an "inline" expansion of all the called functions down to the primitive functions. This complexity measure is called "expanded structural complexity" in [Rosca and Ballard, 1994b] and is based on the structural complexity (i.e. the number of tree nodes) of all the functions in the hierarchy which are called directly or indirectly by the individual. The expanded structural complexity of a program F, denoted $IC(F)$, can be computed in a bottom-up manner starting with the lowest functions in the call graph of F. For each subfunction G, called directly or indirectly by F, $IC(G)$ can be defined using a recursive formula (see the appendix).

Three experiments are reported next. First, a uniform random generation of parity tables is compared to a GP random generation of program trees. Second, we vary the composition of the primitive function set and analyze again the fitness distribution of randomly generated GP programs. Third, we analyze the expanded structural complexity of GP and HGP solutions. The method of generating GP individuals in the second experiment, borrowed from [Koza, 1992], is the ramped-half-and-half method. In order to create an initial population of increased diversity this method generates trees of depth varying modulo the initial maximum size (taken to be six) and of either balanced or random shape.

3.2 Diversity experiments

If elements $s \in \mathcal{S}$ are generated uniformly then the probability of generating EVEN-n-PARITY is $\frac{1}{2^{2^n}}$. For the EVEN-3-PARITY problem [Koza, 1994b] reports that no solution is discovered after the random generation of 10 million parity functions. However, the above analysis implies that for $n = 3$ it should be considerably easier (one in 256 trees) to find a solution if the random generation of trees in GP results in a uniform distribution of functions. Unfortunately, even for a uniform distribution of functions, the probability to generate a solution decreases super-exponentially in the problem size. About four billion GP functions would have to be generated in order to find one that computes EVEN-5-PARITY ($n = 5$). We will see that GP with functions does much better than this.

Figure 2 compares the distribution of hits obtained for a population of tables (ideal case), GP functions and ADF-GP functions. The mean and standard deviation of the distribution of randomly generated tables compares closely to the theoretical results outlined above (see relations (4) and (5)) for $n = 5$, although only 16,000 random tables were generated. The distribution of GP functions in the EVEN-5-PARITY problem, with the function set defined in (1), shows that for $h < 12$ or $h > 20$ the probability of having h hits is practically zero. The GP random distribution is much narrower than might be anticipated.

It is worth examining what happens when automatically defined functions are used. Figure 2 shows that a random population of ADF-GP trees generated using the ramped-

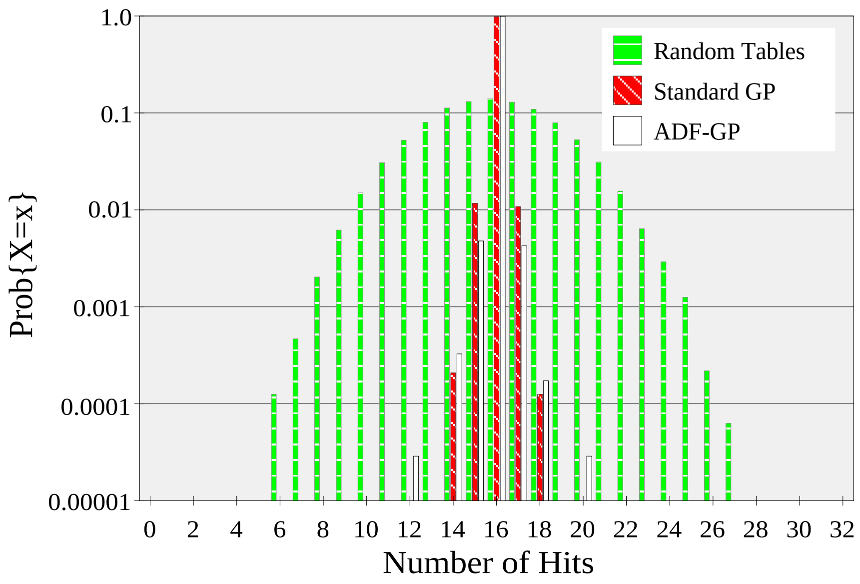

Figure 2: Probability mass function of the random variable X representing the number of hits with the EVEN-5-PARITY function in three cases (a) Random generation of Boolean tables; (b) Random generation of standard GP EVEN-5-PARITY functions; (c) Random generation of ADF-GP EVEN-5-PARITY functions, with two automatically defined functions and two arguments each.

half-and-half method has a wider distribution of hits than standard GP. The effect is called the lens effect in [Koza, 1994b].

Varying the composition of the primitive function set

Figure 3 shows the hit distributions for GP when the composition of the function set is varied. In the *all-functions* plots, all 16 Boolean functions of two variables are included in $\mathcal{F}$ while in the *some-functions* plots a random selection of half of these 16 functions, including the primitive ones in $\mathcal{F}_0$, are part of the initial function set $\mathcal{F}$. Figure 3 shows the same experiment performed with ADF-GP. Two automatically defined functions have been used, each having two arguments. $ADF0$ has the function set $\mathcal{F}_0$. $ADF1$ can additionally invoke $ADF0$. The program body can additionally invoke both $ADF0$ and $ADF1$.

Expanded structural complexity

Table 1 presents a sample of complexity results obtained with the standard GP algorithm and with ADF-GP. The rows having 0 in the "Generation" column correspond to an initial random generation of programs. The other two rows are the results at the end of successful runs. The ADF-GP rows include the structural complexity values obtained for the two evolved sub-functions (ADF0 and ADF1) and the main program body (Body). The table

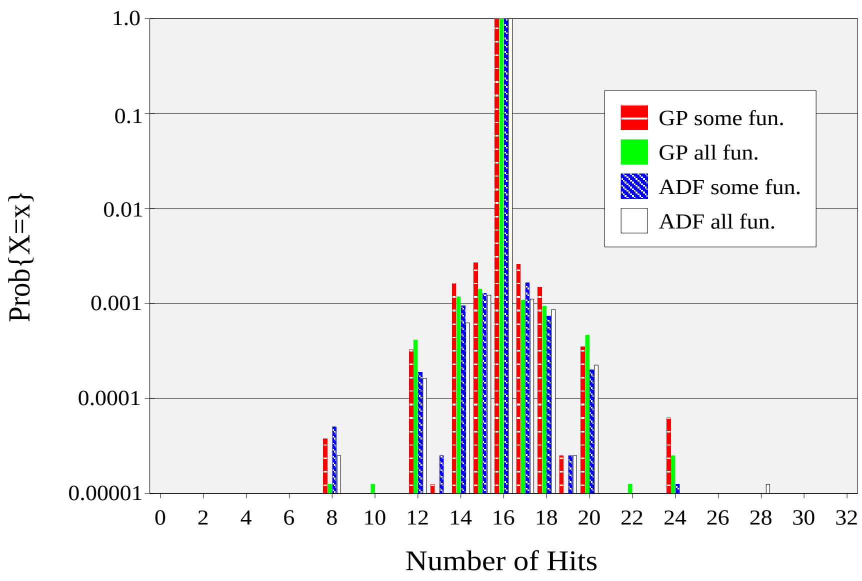

Figure 3: Probability mass function of the number of hits when all or some (a random selection of) Boolean functions of two variables are used in generating EVEN-5-PARITY programs. The primitive set include the primitive functions AND, OR, NAND, NOR.

shows that expanded complexity in ADF-GP is several orders of magnitude higher than the structural complexity of programs in standard GP.

Table 1: Complexity results for EVEN-5-PARITY tested with the standard GP and ADF-GP algorithms. Average values are determined from 12 runs.

Method	Generation	Structural Complexity			Expanded Complexity	
		ADF0	ADF1	Body	Best	Average
Std.GP	0	-	-	-	15	6.53
Std.GP	28	-	-	-	180	241.2
ADF-GP	0	15	15	45	423	439.9
ADF-GP	30	41	13	95	5497	6429.3

3.3 Comparison of results

The narrow GP hit distribution suggests a low population diversity. A solution by means of GP will be difficult to obtain, because it would require more generations, and thus an increased computational effort, to create diverse individuals. Moreover, search may be successful provided that fitness-proportionate selection and the genetic operators used do not narrow the population diversity even more. This change in the hit distribution for HGP is a direct result of the introduction of higher level functions into the representation. It is one of the hypotheses explaining why HGP approaches work better than standard GP.

When the function set is varied an even wider distribution will result (see the *GP-some-functions* and *GP-all-functions* distributions from figure 3). When defined functions are used the hit distribution does not become much wider. However the ADF-GP method still generates larger standard deviations and thus increased diversity. Randomly generated programs with the highest number of hits (28) were obtained using this method. Overall, the distributions of hits look very similar when a larger selection of functions is used.

Note that $\mathcal{F}_0$ is complete in the sense that any Boolean function can be written just using functions from $\mathcal{F}_0$. The effect of apparently non-useful functions, initially included in the function set, is beneficial. All new functions, either ADFs or initial extensions of the function set, are based on the initial primitive functions and terminals. Theoretically, the search space remains the same, the space of all programs that can be built based on $\mathcal{F}_0$ and $\mathcal{T}_0$. The sampling of the search space by means of the crossover operator is changed in ADF-GP. Still, any solution that can be obtained by ADF-GP could theoretically be found by GP although the time to find a solution would be significantly larger. From the static point of view of creating an initial population, using ADFs is equivalent to considering a larger initial function set.

A more formal interpretation of this remarks can be stated by considering the *closure* requirement in GP [Koza, 1992]. Closure requires that any function be well defined for any combination of arguments (terminals or results of other function calls) that it may encounter. Suppose that any subtree returns a value from a domain, call it $\mathcal{D}$, and that the result returned by a program depends on a subset of variables from $\mathcal{T}$ which defines the input space. Define $\mathcal{F}_{total}$ to be the set of functions mapping the input space onto $\mathcal{D}$. Then an ADF is a function from $\mathcal{F}_{total}$. ADF-GP may simply be interpreted as GP over an enlarged function set $\mathcal{F}_{total}$. Over generations, the use of ADFs is equivalent to a dynamic sampling of various functions from this much larger function set.

Naturally, $\mathcal{F} \subset \mathcal{F}_{total}$. It may be difficult to determine the appropriate functions from $\mathcal{F}_{total}$ necessary to solve a given problem. It is unrealistic to consider huge functions sets in either GP or ADF-GP. However, GP can be used to select primitives that can be better combined to yield candidate solution improvements [Koza, 1994b]. In this case, automatic selection of primitives will have its computational cost.

The increased fitness diversity is determined by a larger set of functions that is given to express candidate solutions. Equivalently, the expanded set of functions biases GP search towards regions of the search space containing better individuals.

The increased standard deviation of program hits can be correlated with the increased size and more diverse structure of individuals obtained by using ADF-GP or, similarly, AR-GP. This results in an increased GP exploration of the space of programs. The use of an HGP approach enables the manipulation of a population of higher diversity programs, which positively affects the efficiency of an HGP algorithm for complex problems.

4 HGP Evolution Dynamics

GP evolution dynamics has been very difficult to analyze. The traditional analysis of GAs by Holland [Holland, 1992] focuses on the propagation of schemata from one generation to the next. The building block hypothesis ([Holland, 1992], [Goldberg, 1989]) outlines the importance of small schemata, called building blocks, in the proper functioning of a GA. More recently, crossover has been considered the differentiating feature that gives a GA advantages over other stochastic methods in certain types of problems. For example [Eshelman and Schaffer, 1993] brings evidence that crossover with pair-wise mating helps propagating middle order building blocks.

The arguments presented so far have analyzed a static picture of GP. The wider hit distributions and the increased expanded structural complexity suggested an increased exploration potential of GP with functions. The focus of attention in this section moves to an analysis of the HGP evolution dynamics through the crossover operator. A simple analysis of the effects of the crossover operator suggests that GP structures are highly unstable. However, a more careful analysis of the way HGP discovers useful structures reveals that selection gradually favors changes with small effects on the individual behavior. A hierarchy of functions is essential in order to extend the potential for state space exploitation.

4.1 GP Causality

The main problem in identifying how HGP works is determining the effects of the crossover operation as reflected in the variation of fitness from parents to offspring and in the population composition at a given time.

The intuition is that most crossover operations have a harmful effect. In particular, offspring of individuals that are already partially adapted to the "environment" and already have a complex structure are more likely to have a worse fitness. This is close to the conclusions on the role of mutation in natural evolution [Wills, 1993]. It is also in agreement with our intuition that a small change in a program may drastically change the program behavior. In addition there is the following simple argument. Consider a partial solution to a hypothesis formation problem obtained using standard GP and represented by a tree T. Consider that T is selected as a parent and it is possible to obtain a solution by modifying T in such a way that a certain subtree T_i is not changed. Consider also that crossover points are chosen with uniform probability over the set of m nodes of T. The probability of choosing a crossover point v that does not lie within T_i is:

$$Prob(\text{Select}(v)|v \notin T_i) = 1 - \frac{Size(T_i)}{Size(T)}$$

The bigger T_i is (and this is true in the case of a hypothetical convergence to a solution) the smaller is the probability of keeping it unchanged. The dynamics of trees shows the phenomenon of *instability* or *poor causality* of GP structures. Next we discuss four important issues that show a more complete picture of the problem.

First, how much code of a parse tree representing an individual is effective? It is well known that GP evolves non parsimonious trees if no size pressure is included in the fitness evaluation, a phenomenon suggestively called "defense against crossover" [Altenberg, 1994] However, the useless regions of code may represent reservoirs of genetic material [Angeline, 1994a]. They either preserve or evolve good fragments of code to be activated later during evolution as a result of crossover. One such example is presented in figure 4. Moreover, redundant regions of code may be created artificially, in analogy with the natural phenomenon of gene duplication, in order to evolve a better program by specializing its treatment of a subclass of inputs [Koza, 1994a].

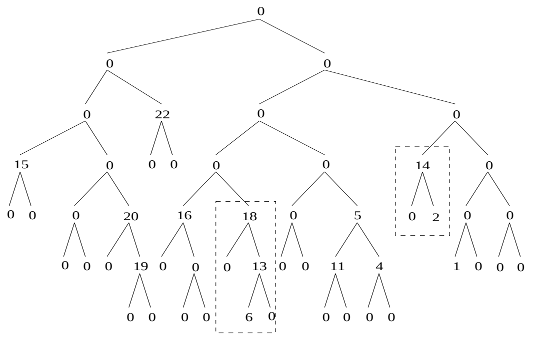

Figure 4: The structure tree for a GP solution to EVEN-5-PARITY. The notion of *structure tree* was introduced in [Rosca and Ballard, 1994a] with the goal of qualitatively analyzing program transformations during evolution. A structure tree has its nodes labeled with the most recent generation number when the node played a pivot role in a crossover operation. Zero labeled nodes remained unchanged from the initial generation. The highlighted subtrees did not play any role in the evaluation of their parents but are important in the final solution.

Second, from a topological point of view, where are most of the crossover changes performed? Equivalently we can ask about the expected height of crossover pivot points. A fundamental remark is that, in current GP practice, crossover nodes are chosen according to a uniform probability distribution. If we additionally assume, for a rough approximation, that trees operated upon are complete binary tress we can compute the expected height of crossover pivot points:

$$E[CrossoverHeight] = \sum_i Prob(i) \cdot Height(i) = 2 - \frac{h}{2^h - 1}$$

where $Prob(i)$ is the probability of choosing node i of height $Height(i)$ as a crossover point, and h is the tree height. The leaf nodes of the complete binary tree all have height 1. This result, although based on an assumption, shows that most of the changes are closer to the tree bottom. The effect can be noticed on the tree in figure 4, which is a typical case.

Third, what is the influence of selecting smaller or bigger subtrees to participate in the crossover operation? We hypothesized that the crossover operator generates non-causal changes in GP. A complete answer to this question would involve an analysis of the properties of the function set. The effect of a small change can be severe in problems of symbolic regression, or less severe in problems of regression of Boolean functions. For example the result of the Boolean function represented in figure 5 will be given by the result of evaluating δ on a given fitness case only if the evaluation of the following Boolean expression is true:

$$\alpha \cdot \neg\beta \cdot \neg\gamma$$

The longer the path to δ the higher will be the probability that δ plays a reduced role in the overall evaluation.

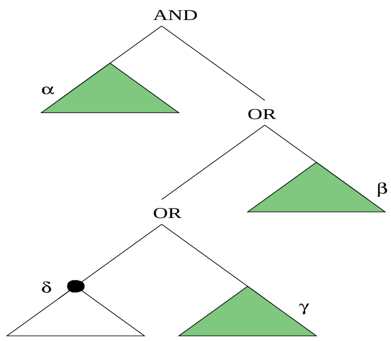

Figure 5: A small change in a Boolean tree will not necessarily determine a sharp change in the program behavior.

Fourth, how does GP *exploit* structures? In contrast to GP crossover, the GA crossover operator is *homologous*, that is it maintains fixed positions for exchanged alleles. GP crossover is non-homologous in the sense that it does not preserve the position of the subtree on which it operates, being allowed to paste a subtree at any tree level. The probability of choosing homologous crossover points in two structurally similar parents in order to transmit the parent functionality to offspring is inversely proportional to the product of the parent sizes, i.e. it is very low. Moreover, if trees grow in size, this probability decreases even more and becomes negligible. This implies that even when the two parents are identical, offspring will most often have a totally different functionality, and most probably they will score less than parents. Selection favors crossover changes that recombine parts of the structure of the parents so as to improve performance, but how? In several problem domains one can observe the superposition of the parent behaviors in the offspring. In an example for the problem of finding an impulse response function, Koza showed that crossover determines an improved offspring performance by improving one parent's performance for one portion of the time domain, and inheriting the behavior of the other parent for the rest of the domain [Koza, 1994b]. Such a behavior has been interpreted as "case splitting": GP refines a partial solution by changing a subtree so that the program treats separately, in a more detailed way, a particular input case. In this case, structures are exploited through the function they have when computing fitness.

In spite of the apparent non-causality, GP evolves better and better solutions. Two explanations could be given to this apparent paradox. A first explanation, which holds mostly in early stages of evolution, is the exploration of the space of programs, as discussed in the previous section. Once the population average fitness increases, it becomes less probable to find above average individuals by pure exploration. Presumably, GP *exploits* the structures preserved in the population. For this, most changes selected for in later stages of evolution are *small* changes, most probably changes at higher tree depths and in subtrees of small heights. Such changes are causal changes because they slightly alter the function of offspring in comparison to parents.

4.2 HGP causality

The above properties and problems are inherited by ADF-GP. Moreover, ADF-GP presents an even higher instability for crossover due to two main reasons: amplification of effects in the subroutine hierarchy and lexical scoping which characterizes subroutine definitions. Note that the ADF approach attacks the search problem at *different structural levels* simultaneously. GP has to discover both the definitions for a fixed set of sub-functions, each with a predefined number of parameters, and how to combine calls to the automatically defined functions within the main body. This corresponds roughly to discovering a way to decompose the problem and solving the subproblems given only the maximum number of subproblems and the general structure of the subproblems (i.e. the number of parameters). Due to the imposed ordering of ADFs we can consider each ADF as a different structural level.

The amplification of effects in the subroutine hierarchy can be illustrated with a simple example. If crossover determines a change in a low ADF in the hierarchy, for instance ADF_0, the change will cause a different behavior for all subroutines which invoke ADF_0 and in all subtrees which invoke the affected subroutines. Thus, a change at the basis of the hierarchy in an individual will drastically change the individual behavior. A change at a higher level in the hierarchy could be amplified too, provided that the subroutine changed is effectively used by other subroutines or the main program more than once.

The lexical scoping problem is illustrated in figure 6. During GP search, modifications are alternatively made at each of the structural levels. A code fragment brought from another individual changes its function entirely if it contains calls to ADFs. For example, consider a piece of code involving calls to lower order ADFs that is pasted in a higher-order function or the main body as a result of a crossover operation, and suppose the definitions of the ADFs in the two parents are entirely different. Lexical scope dictates the definition to be used when computing a call to a sub-function, so that the calls to ADFs from the piece of code transplanted will refer to the ADF definitions in the new scope. The crossover operation will most probably change the function of the receiving parent completely.

The arguments above suggest the tendency towards even more non-causal effects of the crossover operation in ADF-GP. The non-causality property of ADF is undesirable in later stages of evolution as it prevents GP from exploiting good structures already incorporated in the population.

In general, GP and ADF-GP exhibit poor causality. It is useful to visualize how the search for a solution may generally proceed in ADF-GP. Each of the ADF functions represents a different subroutine. Consider the last modification imposed on a program tree before it becomes an acceptable solution. It is very unlikely, but not impossible that this last change has been a change with a large influence, for example a change in one of the

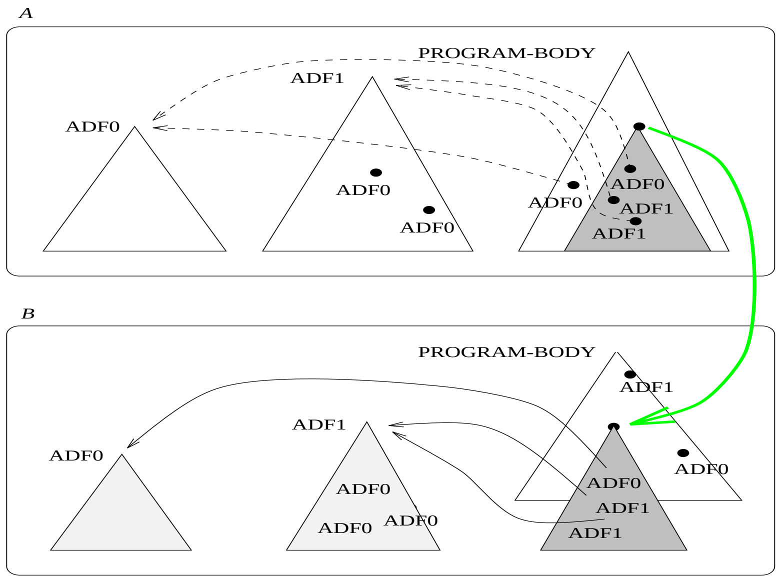

Figure 6: The non-causality of ADF: A fragment of code copied from individual A into individual B finds itself in a new lexical environment, where local definitions for ADFs apply. A represents the donor parent before crossover is applied while B represents the receiving parent after the crossover operation.

functions at the basis of the hierarchy. This situation represents a lucky change. Most probably, though, it was a change at the highest level, in a subtree of small height of the program body.

We hypothesize as a general principle of GP dynamics that selection most often favors small changes. Only such changes, respecting the principle of strong causality, have the highest chance of being successful. The effect of this principle is a stabilization on lower level ADFs that will be useful. The evolutionary process freezes good subroutines at a given hierarchy level and looks for changes at higher levels [Simon, 1973]. Hence:

Hypothesis. Hierarchical genetic programming (and in particular ADF-GP) discovers and exploits useful structures in a *bottom-up* manner.

Note that this hypothesis is the basic idea in the AR-GP extension. In AR-GP, the selection of potentially useful subroutines generalizing blocks of code drives the adaptation of the problem representation.

4.3 Birth certificates

In order to test the above hypothesis we have studied the most recent part of the genealogy tree for EVEN-n-PARITY parity problems. This was done by assigning to each individual a *birth certificate* that specifies its parents and the method of birth (one of ADF0 crossover, ADF1 crossover, main program body crossover or reproduction). We hoped that an analysis of the birth certificates starting with the final solution and tracing its origin backwards, would shed light on the GP dynamics, as hypothesized above. In order to determine the effect of the different types of birth operations we compute a temporally discounted *frequency* factor for a given solution tree T for each type of birth (*birth-type*):

$$frequency(T, birth\text{--}type, d) = \left(\frac{1-\gamma}{1-\gamma^d}\right) \sum_{i=0}^{k_T} \chi_{\{birth-type\}}(i) \cdot \gamma^{depth(T_i)} \tag{6}$$

where k_T is the number of programs in the genealogy tree of T down to a depth d, and $\chi_{\{type\}}(T_i)$ is the characteristic function of ancestor T_i of T, returning 1 if T_i has a birth certificate of type *birth-type* and 0 otherwise. The scaling factor $\frac{1-\gamma}{1-\gamma^d}$ is a normalizing constant that makes each type of discounted frequencies for a fixed tree T add up to 1. We used a discounted formula to reflect the higher importance of crossover operations from more recent generations.

Table 2 presents the results for several successful runs of ADF-GP for EVEN-5-PARITY, with two ADFs and three arguments each. In most cases, the frequency factors are highest for the program-body or clearly decrease from program-body to ADF1 to ADF0. These results support the earlier conclusion that ADF-GP search relies in most cases on changes at higher and higher structural levels which make it possible to exploit good code fragments that already appeared in the population.

Table 2: Statistics of birth certificates in successful runs of EVEN-5-PARITY using ADF-GP with a zero mutation rate. Each certificate of a given type counts one unit and is temporally discounted with a discount factor $\gamma = 0.8$ based on its age. Only certificates at most 8 generations old have been considered.

GP Run	# Individuals Explored	Birth Certificate Frequency			Final Generation
		ADF0	ADF1	Body	
1	123,009	0.295	0.0	0.704	32
2	110,892	0.221	0.472	0.416	32
3	62,699	0.077	0.526	0.397	17
4	35,162	0.447	0.102	0.451	9
5	55,748	0.1	0.214	0.685	15
6	55,438	0.093	0.202	0.704	15

The above numerical results have taken into account only a small time window compared to the entire number of generations. A complete picture of the importance of various types of crossover during the entire GP evolution can be constructed from a more detailed analysis of birth certificates. Such an analysis is depicted in figure 7. It suggests the overall importance of a birth certificate type from generation 0 till the solution is found. While the percentage of program-body changes increases, the percentage of ADF changes decreases.

A similar analysis is depicted in figure 8. Part (a) represents a stacked chart which suggests both the overall importance of a birth certificate type as well as its trend over the entire evolution period, from generation 0 till the solution is found. Part (b) displays the same results in an overlapping fashion to allow for better comparison among the frequency types. The stabilization of changes in the hierarchy occurs bottom-up.

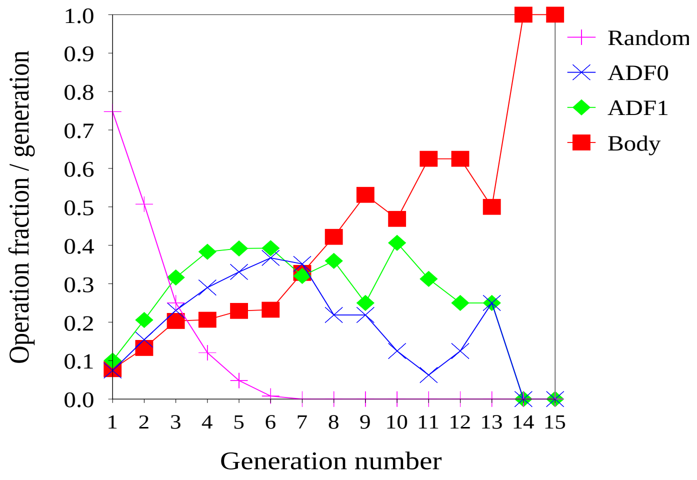

Figure 7: Variation of the fraction of crossover types over generations, while looking for a solution to EVEN-5-PARITY. *Random* indicates the propagation of random individuals from the initial population due to reproduction.

The results of this section support the hypothesis that HGP discovers and exploits useful structures in a bottom-up, hierarchical manner. AR-GP has an explicit policy for a bottom-up *exploitation* of discovered structures for making search more efficient, while ADF-GP neglects it.

5 Discussion of Results

The main difficulty of solving complex parity problems in GP is that the computational effort increases exponentially with the size of the problem. Each fitness evaluation becomes more expensive as the problem is scaled up. For instance, the number of fitness cases in the EVEN-6-PARITY problem is twice that of the EVEN-5-PARITY problem and it doubles with every unitary increase in the order of the problem. A co-evolutionary approach such as in [Hillis, 1990] could reduce the cost of a fitness evaluation by relying on a subset of fitness cases which evolves dynamically by being controlled in its turn with a genetic algorithm. Also, the number of fitness evaluations necessary to find a solution with high probability increases with problem size [Koza, 1994b] in the standard GP implementation. One explanation of the poor GP convergence is the inability of standard GP to exploit opportunities for code generalization and reuse. In contrast, by using ADFs or adapting the representation as in AR-GP the same problems can be solved more easily ([Koza, 1994b], [Rosca and Ballard, 1994c]). We gave a qualitative explanation of the improved behavior of HGP, based on an analysis of the evolution process on two dimensions: diversity and causality. Next we relate these ideas to the tradeoff between exploration and exploitation.

This paper shows that there exists an implicit bias in the random generation of GP solution encodings which confines population diversity. Diversity increases as a result of

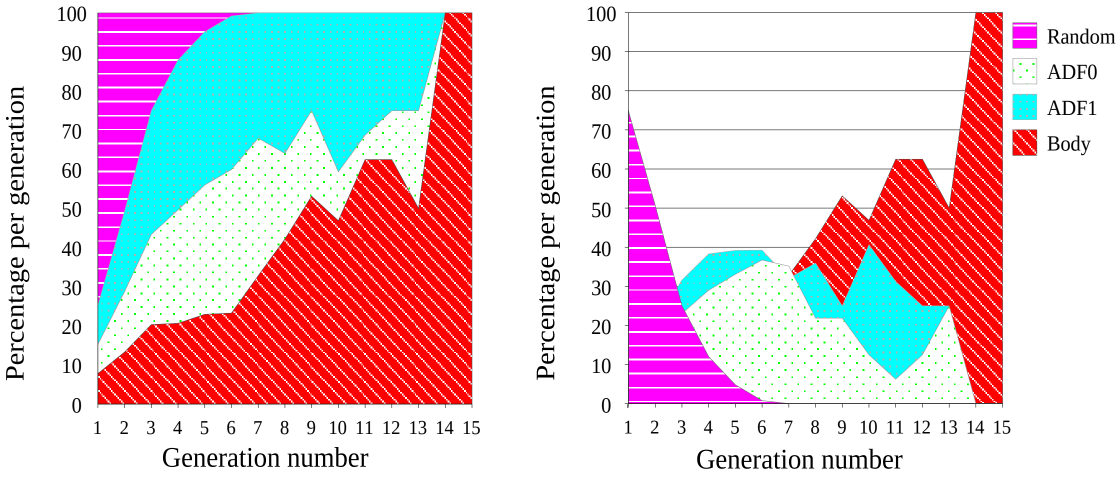

Figure 8: (a) Distribution trend of birth certificates (b) Overlapping distributions show the importance of the different crossover types when looking for a solution to EVEN-5-PARITY that was found in generation 15.

changes in representation. The expanded structural complexity of HGP individuals also increases diversity and highlights the distinctive size and shape of trees as compared to standard GP. Increased diversity is related to an increased exploration of the space of programs.

The results presented confirm that as the population evolves, increasingly causal changes become more important and are selected. Low level crossover changes in the function hierarchy are highly non-causal and have an exploratory role. Exploitative changes are adopted later in the process as the average program fitness increases.

The stabilization of changes in the function hierarchy occurs bottom-up. The GP search process exploits the structures already discovered although it does not avoid spending unnecessary effort with state space exploration. Useful genetic operations are also the more causal ones. Thus, causality is correlated with search space exploitation.

In most learning approaches the system must have an explicit policy of balancing exploration and exploitation. In contrast GP is a search technique that implicitly balances exploration and exploitation, as I argue next.

Discovery and evolution of functions amplifies the exploration ability of the GP search process. However, as the best-of-generation program fitness increases, the probability of falling upon good individuals by exploration decreases substantially. The GP search process exploits the structures already discovered, although it does not avoid spending unnecessary effort with state space exploration. Increased exploitation corresponds to more and more causal effects of the genetic operations (see table 3). This suggests that GP is able to dynamically balance exploration and exploitation.

Another idea suggested by the causality analysis is that the tradeoff can also be adaptively controlled by modifying the rate at which causal or non-causal genetic operations are applied so that GP spend its search effort in a more efficient way. In the following we discuss recent improvements of HGP algorithms from this perspective.

Table 3: Correlation between causality and exploratory ability in GP search.

Time (generation)	Early	Advanced
Crossover changes	Non-causal	Causal
Exploration	High	Low
Exploitation	Low	High

An interesting extension of ADF-GP confirms the importance of the idea of causality in GP. [Koza, 1994a] introduces six new genetic operations for altering the architecture of an individual program: branch duplication, argument duplication, branch deletion, argument deletion, branch creation and argument creation. All addition operations respect the principle of strong causality discussed before. They are performed such that they preserve the behavior of the resulting programs. They merely increase the potential for program refinement and thus they resemble the process of gene duplication in natural evolution [Koza, 1994a]. The duplication of elements of program architecture (branches or arguments) is done in conjunction with a random replacement of the invocations of the corresponding element to the duplicated copy. Such an operation decreases the probability that a future random change will drastically change the behavior of the program. It respects the principle of strong causality while allowing for future behavior improving changes. An analogous conclusion can be drawn for the creation operations. The deletion operations do not possess the nice properties mentioned above. They have the antagonist role of confining the increase in size of the evolved programs.

6 Conclusions

This report presented a unifying view of the two approaches to the discovery of functions, ADF-GP and AR-GP emphasizing the hierarchical structure of the resulting problem representation. It showed that the exploration of the search space in GP depends on the power of the discovery and evolution of functions.

The report also analyzed the causality of the crossover operator in GP and suggested that search control parameters can be adapted for speeding up GP search. Standard GP presents a characteristic instability or poor causality of the structures evolved, which can be varied by changing the probability of selecting crossover points. This effect is amplified in GP with function discovery. Poor causality has been discussed related to the exploration-exploitation tradeoff in search problems.

Arguments for a bottom-up evolutionary thesis of GP were discussed. Early stages of evolution in GP usually discover stable components [Simon, 1973]. Replaying backwards the genealogy tree that resulted in a problem solution shows that most changes in later stages of evolution are performed at the higher hierarchical levels. This suggests that in the unconstrained HGP approach such as ADF-GP there are implicit constraints. Early in the process the changes are focused towards the evolution of more primitive functions. Later in the process the changes are focused towards the evolution of program control structures.

Discovery of functions is also an adaptive mechanism for trading off exploration and exploitation in GP. Most often the control structure of a search algorithm explicitly balances exploration and exploitation by means of control parameters. In contrast, GP is a search technique that implicitly balances exploration and exploitation.

The emergent structure in ADF-GP as an effect of causality is an explicit policy in AR-GP. The bottom-up evolution of HGP discussed in the paper justifies this explicit search for building blocks and the expansion of the problem representation, which was successfully used in AR-GP [Rosca and Ballard, 1994a]. AR-GP uses fit, small blocks to define new functions. AR-GP evolves a variable hierarchy of functions, each having a variable number of arguments. A process of extinction of population individuals accelerates the use of the discovered functions.

Future work aims at an improved version of AR-GP that will additionally allow for evolution of functions. One could use the insights about the dynamics of GP search presented in this paper to come up with a more refined and efficient GP-like system, that involves automatic adaptation of control parameters.

7 Acknowledgments

I am grateful to Dana Ballard for the many inspiring discussions we had on the topics discussed in this paper. Also, I would like to thank Pete Angeline, Jonas Karlsson and Andrew Mccallum for their helpful suggestions on earlier drafts of this document.

Appendix: Expanded structural complexity

We evaluate the expanded structural complexity of function F_i, $IC(F_i)$, after we have evaluated $IC(F_j)$ for all $0 \leq j < i$, by performing inline substitutions of all functions called by F_i with their expanded bodies.

From a computational point of view, we also have to keep track of the number of times each argument of a function appears in the expanded version of the function, i.e. after the inline substitution of each of the lower order functions have been performed. For example, if F_1 has a call to F_0, the numbers of times each of F_1's arguments appear before and after the inline substitution of F_0 will differ in general. This influences the expanded structural complexity of a function (say F_2) that calls F_1. The general case is considered below.

If F_i has $n_i = j$ arguments, then we represent the number of times each of F_i's arguments appear in F_i's expression by a vector $\vec{\mathbf{x}}^i = \left[\vec{\mathbf{x}}^i_1 \; \vec{\mathbf{x}}^i_2 \; ... \; \vec{\mathbf{x}}^i_j\right]^t$, where the t superscript is the translation vector operator. In the formulae below, T is an arbitrary subtree of F_i. The root label of T, $root(T)$, can be a function from the initial function set $\mathcal{F}_0$, a newly discovered function, a variable of F_i or a leaf which is not a variable. When $root(T)$ is a function of arity k, $T_1, ..., T_k$ are the ordered subtrees of T.

$IC(F_i)$ and $\vec{\mathbf{x}}^i$ are computed in a bottom-up manner, starting with $i = 0$. For each i, $IC(F_i)$ and $\vec{\mathbf{x}}^i = \vec{\mathbf{x}}^i(F_i)$ are defined recursively, starting with the tree representing F_i, $T \equiv F_i$ in the following way:

$$IC(T) = \begin{cases} 1 & \text{if } T \text{ is a leaf} \\ IC(root(T_s)) + \sum_{l=1}^{k} IC(T_i) * \vec{\mathbf{x}}_l^i & \text{if } root(T) \text{ is a function} \end{cases}$$

$IC(f)$ has been computed previously if $f \equiv root(T)$ is a discovered function ($f \in \mathcal{F}_i - \mathcal{F}_0$), or $IC(f) = 1$ for a primitive function ($f \in \mathcal{F}_0$).

$$\vec{\mathbf{x}}^i(T) = \begin{cases} \vec{\mathbf{0}} & \text{if } T \text{ is a non-argument leaf} \\ [0...010...0]^t & \text{if } T \text{ is the } l\text{-th argument of } F_i \\ & \quad \text{(a 1 on the } l\text{-th position)} \\ \sum_{l=1}^{n_m} [\vec{\mathbf{x}}(T_l)]^t \cdot \vec{\mathbf{x}}^m & \text{if } root(T) \equiv F_m,\ m < i \text{ and } n_m \\ & \quad \text{is the number of arguments of } F_m \end{cases}$$

$\vec{\mathbf{x}}(T_l)$ represents the number of appearances of the arguments of F_i in the expanded subtree T_l, and $(\cdot)$ is the scalar vector product. An example is presented in table 4.

Table 4: Example of complexity values for a hierarchy of three Boolean functions F0, F1 and F2.

F0 (DEFUN **F0** (A0 A1) (OR (AND A0 A1) (AND (NAND A0 A0) (NAND A1 A1))))

F1 (DEFUN **F1** (A0 A1) (NAND (OR (AND A0 A1)(**F0** (NAND A0 A0) (NAND A1 A1)))(OR (AND A0 A1) (AND (NAND A0 A0)(NAND A1 A1)))))

F2 (DEFUN **F2** (A0 A1)(**F1** (ADF0 (**F0** D0 D1) (**F0** D2 D3)) D4))

Complexity	**F0**	**F1**	**F2**
Structural (SC)	11	23	43
Executional (EC)	11	34	76
In-line Expanded (IC)	11	39	739

References

[Altenberg, 1994] Lee Altenberg, "The Evolution of Evolvability in Genetic Programming," In Kim Kinnear, editor, *Advances in Genetic Programming.* MIT Press, 1994.

[Angeline, 1994a] Peter J. Angeline, *Evolutionary Algorithms and Emergent Intelligence*, PhD thesis, Computer Science Department, Ohio State University, 1994.

[Angeline, 1994b] Peter J. Angeline, "Genetic Programming and Emergent Intelligence," In Kim Kinnear, editor, *Advances in Genetic Programming.* MIT Press, 1994.

[Angeline and Pollack, 1994] Peter J. Angeline and Jordan B. Pollack, "Coevolving High Level Representations," In Christofer G. Langton, editor, *Artificial Life III, SFI Studies in the Sciences of Complexity*, volume XVII, pages 55–71, Redwood City, CA, 1994. Addison-Wesley.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[Cramer, 1985] Nichael Lynn Cramer, "A Representation for the Adaptive Generation of Simple Sequential Programs," In *Proceedings of the First International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc, 1985.

[Eshelman and Schaffer, 1993] Larry J. Eshelman and J. David Schaffer, "Crossover's Niche," In Stephanie Forrest, editor, *Proceedings of the International Conference on Genetic Algorithms*, pages 9–14. Morgan Kaufmann, 1993.

[Goldberg, 1989] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Hillis, 1990] W. Daniel Hillis, "Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure," In Stephanie Forrest, editor, *Proceedings of the Ninth Annual International Conference of the Center for Nonlinear Studies on Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks*, pages 228–234. North Holland, 1990.

[Holland, 1992] John H. Holland, *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, MIT Press, 2nd edition, 1992.

[Kaelbling, 1993] Leslie Pack Kaelbling, *Learning in Embedded Systems*, MIT Press, 1993.

[Kinnear, 1994] Kim Kinnear, "Alternatives in Automatic Function Definition," In Kim Kinnear, editor, *Advances in Genetic Programming.* MIT Press, 1994.

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

[Koza, 1994a] John R. Koza, "Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming," Computer Science Department STAN-CS-TR-94-1528, Stanford University, 1994.

[Koza, 1994b] John R. Koza, *Genetic Programming II*, MIT Press, Cambridge, Massachusetts, 1994.

[Lohmann, 1992] Reinhard Lohmann, "Structure Evolution and Incomplete Induction," In *Parallel Problem Solving from Nature 2*, pages 175–185. Elsevier Science Publishers, 1992.

[O'Reilly and Oppacher, 1994] Una-May O'Reilly and Franz Oppacher, "The troubling aspects of a building block hypothesis for genetic programming," In *Proceedings of the Third Workshop on Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, Inc, 1994.

[Rechenberg, 1994] Ingo Rechenberg, "Evolution Strategy," In Jacek M. Zurada, Robert J. Marks-II, and Charles J. Robinson, editors, *Computational Intelligence - Imitating Life*, pages 147–159. IEEE Press, 1994.

[Rosca and Ballard, 1994a] Justinian P. Rosca and Dana H. Ballard, "Genetic Programming with Adaptive Representations," Technical Report 489, University of Rochester, Computer Science Department, 1994.

[Rosca and Ballard, 1994b] Justinian P. Rosca and Dana H. Ballard, "Hierarchical Self-Organization in Genetic Programming," In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 251–258. Morgan Kaufmann Publishers, Inc, 1994.

[Rosca and Ballard, 1994c] Justinian P. Rosca and Dana H. Ballard, "Learning by Adapting Representations in Genetic Programming," In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 407–412. IEEE Press, Orlando, 1994.

[Ryan, 1994] Conor. O. Ryan, "Pygmies and Civil Servants," In Kim Kinnear, editor, *Advances in Genetic Programming*. MIT Press, 1994.

[Simon, 1973] Herbert A. Simon, "The Organization of Complex Systems," In G. Braziller Howard H. Pattee, editor, *Hierarchy Theory; The Challenge of Complex Systems*, pages 3–27. New York, 1973.

[Wills, 1993] Christopher Wills, *The Runaway Brain*, BasicBooks, 1993.

[Wilson, 1994] Stewart W. Wilson, "ZCS: A Zeroth Level Classifier System," *Evolutionary Computation*, 2(1):1–18, 1994.