Genetic Programming with Adaptive Representations

Justinian P. Rosca, Dana H. Ballard

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

Technical Report 489

February 1994

Abstract

Machine learning aims towards the acquisition of knowledge based on either experience from the interaction with the external environment or by analyzing the internal problem-solving traces. Both approaches can be implemented in the Genetic Programming (GP) paradigm. [Hillis, 1990] proves in an ingenious way how the first approach can work. There have not been any significant tests to prove that GP can take advantage of its own search traces. This paper presents an approach to automatic discovery of functions in GP based on the ideas of discovery of useful building blocks by analyzing the evolution trace, generalizing of blocks to define new functions and finally adapting of the problem representation on-the-fly. Adaptation of the representation determines a hierarchical organization of the extended function set which enables a restructuring of the search space so that solutions can be found more easily. Complexity measures of solution trees are defined for an adaptive representation framework and empirical results are presented.

This material is based on work supported by the National Science Foundation under Grant numbered IRI-8903582 by NIH/PHS research grant numbered 1 R24 RR06853-02 and by a Human Science Frontiers Program research grant. The government has certain rights in this material.

1 Introduction

The defining characteristic of a learning system is considered to be the capability of adaptation under the implicit goal of performance task improvement [DeJong, 1988]. The adaptation process subsumes structural changes, discovery and use of new concepts.

Our focus is the Genetic Programming paradigm, as defined in [Koza, 1992]. In GP, evolution can be viewed as structure synthesis, where the format of the structure is in terms of programs. However changing individual program instructions is most often unfruitful, so extensions to GP have tried to randomly define and use larger components such as new functions (*automatic function definition* or ADF in [Koza, 1992]) or *modules* (in [Angeline and Pollack, 1994]). This paper presents an alternative approach to structure synthesis different from either ADF or modules in several respects. We show that efficient structure synthesis can be obtained by discovering useful genetic material and using it globally to adapt the problem representation. The process of discovery is based on an analysis and tracking of substructures called *building blocks* over generations of evolution. We describe how building blocks result from several heuristic criteria for extending the problem representation. These are used to extend the GP paradigm in what we call Adaptive Representation (AR) GP.

In the next section we present an overview of the GP paradigm, and an example of applying its main steps to evolve a population of programs that solve the EVEN-PARITY problem. We also summarize related work and analyze in more detail two extensions to GP, automatic definition of functions and module acquisition. In section 4 the main theoretical concepts of the GA literature (schemata, schemata theorem, implicit parallelism) form the basis for an approach to analyze genetic material in GP. We define the concepts of GP blocks and building blocks and show that the interpretations given provide the foundation of a theory for adapting problem representation and making GP search more efficient. Section 5 presents two alternative methods for analysis of candidate building blocks. The Adaptive Representation (AR) GP algorithm and details on its implementation are presented next. The ideas introduced are analyzed in section 7 using EVEN-PARITY as a test case. Conclusions and future work directions are finally presented.

2 Overview of the GP Paradigm

A concise description of the paradigm, its main parameters and discussions of some advanced topics are presented in [Koza, 1992], [Koza, 1994]. Here we just overview some basic concepts and notations used throughout this paper.

In GP, problem solving is formulated as a search in the space of computer programs, structures of dynamically varying size and shape. Populations of computer programs (individuals) are genetically bred (similarly to the GA approach), based on three genetic operations: Reproduction, crossover and mutation (O_r, O_c and O_m respectively). Each operation is based on a selection of fit individuals.

The fitness of any individual is determined by evaluating the individual program using domain dependent performance/cost functions ($\mathcal{C}$). A useful measure is normalized fitness. If performance of individual i is $c(i) \geq 0$ (also called standardized fitness) then a normalized fitness measure of individual i is $\frac{c(i)}{\sum_i c(i)}$. Normalized fitness can be also defined based on ranking [Winston, 1992] or other ad-hoc methods.

Two selection methods are currently used. A first one, tournament selection, is described in [Angeline and Pollack, 1993a]. We randomly choose a small group of individuals and let them play a "tournament". The winner represents the individual selected. For example, the competition can consist only of fitness comparison: the individual with the best fitness value among the ones in the competing group is declared the winner. A second method is fitness proportionate reproduction: An individual is selected with a probability proportional to its normalized fitness value.

Reproduction is the process of copying individuals according to their fitness value. Crossover is the process in which two new programs result from an exchange of genetic material between two old programs. It consists of three steps. First, two winner individuals are selected from the population. Then a node called *pivot* is randomly chosen in the tree representing each winner. Finally, subtrees rooted at the two pivots are swapped. Mutation is the process in which a newly generated subtree replaces the subtree rooted at a random pivot position in a selected individual.

The population fitness evaluation, reproduction, and genetic recombination steps are repeated until a termination criterion is met (usually when a good enough individual is discovered or a big enough number of generations is consumed).

The initial steps in applying GP to evolve a population of programs for solving a given problem are: (1) define the set of program primitives (initial genetic material), namely the set of terminals $\mathcal{T}$ and the function set $\mathcal{F}_0$; (2) define the evaluation (cost or merit) measures $\mathcal{C}$ and the set of fitness cases $\mathcal{E}$ on which individual programs are tested. $\mathcal{E}$ can represent an evolving population too, as in the parasites metaphor (see [Hillis, 1990]); (3) define the control parameters (population size M, maximum number of generations G, selection method, fractions of individuals on which each genetic operator is applied, etc.); (4) define a termination criterion (new generations of individuals are created until this criterion is satisfied). Once these are determined an extended GP algorithm can be applied. The structure of such an algorithm is given in figure 1. Elements marked (*) are part of our extended algorithm that copes with the automatic adaptation of the problem representation and will be explained later.

2.1 An example: learning the EVEN-n-PARITY function

The problem chosen to test the ideas in this report is learning the boolean even parity function of a number n of arguments (bits), EVEN-N-PARITY. This function appears to be difficult to learn in a GP implementation especially for values of n greater than 5 [Koza, 1992].

Formally, the EVEN-N-PARITY problem is to find a functional program representing a logical composition of primitive boolean functions that computes the sum of input bits over the field of integers modulo 2.

Four primitive boolean functions of two variables make up the function set:

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

GP skeleton

Define problem: $\mathcal{T}, \mathcal{F}_0, \mathcal{C}, \mathcal{E}$, (*)$\mathcal{C}_{BB}$ (block fitness functions). Denote the population at a given generation i by $\mathcal{P}_i$.

1. Initial Generation: $i = 0$
2. Randomly initialize population $\mathcal{P}_0(\mathcal{T}, \mathcal{F}_0)$
3. Repeat until termination criterion is met (generation i)
   (a) Evaluate population($\mathcal{P}_i, \mathcal{E}, \mathcal{C}$)
   (b) Generate a new population $\mathcal{P}_{i+1}$ by reproduction, crossover, mutation of individuals($\mathcal{P}_i$)
      i. Select genetic operation O (O_r, O_c, O_m)
      ii. Select winning individuals $\mathcal{W}$ from current population $(O, \mathcal{P}_i)$
      iii. Generate offspring(O,$\mathcal{W}$,$\mathcal{P}_{i+1}$)
   (c) (*)Adapt representation($\mathcal{P}_i, \mathcal{F}_i, \mathcal{F}_{i+1}$)
   (d) (*)Enrich population $\mathcal{P}_{i+1}$ ($\mathcal{T}, \mathcal{F}_{i+1}$, epoch-replacement-fraction)
   (e) Next generation: $i = i + 1$

Figure 1: Skeleton of a GP algorithm

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

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

For example the tree in figure 2 represents a LISP program implementing a boolean function of 3 variables built on the basis of $\mathcal{F}_0$ and $\mathcal{T} = \{D_0, D_1, D_2\}$.

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, we compare its performance on all possible combinations of boolean values for the input variables against a table defining the EVEN-N-PARITY function. The table represents the set of fitness cases $\mathcal{E}$. Similarly to supervised learning techniques, it tells when the answer given by a program is correct. Each time the program and the EVEN-N-PARITY table give the same value, the program records a *hit*. GP searches (in the space of all possible programs defined by such trees) for a program that has the maximum number (2^n) of hits.

The cost or standardized fitness of a program i having $Hits(i)$ hits, and a size $Size(i)$ is:

$$\mathcal{C}_{raw}(i) = (2^n - Hits(i)) \cdot C1 + Size(i) \cdot C2 \tag{1}$$

where $C1, C2$ are constants. The size of a program represents its structural complexity, and is defined as the number of function points (inner nodes) and terminal points (leaf nodes) in the tree representing the program. The notion of structural complexity will be generalized later for programs that call non-primitive functions.

Taking $C1 = 1$ and $C2 = 0$ corresponds to a simple fitness function based only on hits. A low value of the standardized fitness function is better than a higher value. According to this measure, we look for individuals with a zero fitness value. In the general case the termination criterion is satisfied if a perfect individual, i.e. one having the maximum number of hits, is found or a maximum generation number is reached.

When $C2 \neq 0$, then for the same number of hits, individuals with a smaller size will have a lower overall fitness value and will be preferred to individuals with a bigger size. In this case the fitness function incorporates a size pressure component.

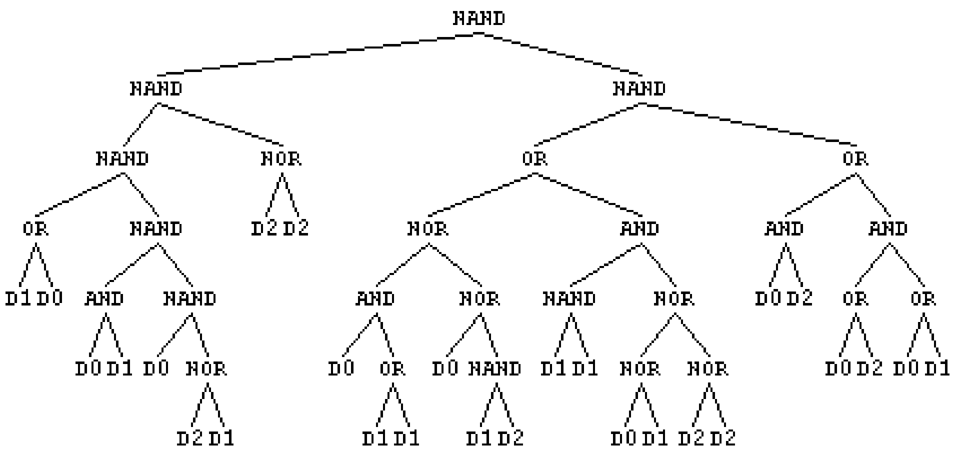

Figure 2: Example of solution to the EVEN-3-PARITY problem

Figure 2 actually represents the solution tree obtained for a run of EVEN-3-PARITY after 23 generations. It implements a boolean function computing parity on 3 input bits.

The size of the search space in GP is equal to the number of program trees of depth at most D (an initial parameter of GP) that can be generated. A lower bound on this value is given by the number of complete binary trees having leaf labels from $\mathcal{T}$ and inner labels from $\mathcal{F}$: $\mid \mathcal{F} \mid^{2^D - 1} \cdot \mid \mathcal{T} \mid^{2^D}$. Regardless of the symmetry or other properties of the problem domain, the number of solution trees is much smaller than this value, fact which explains why searching for solutions is a very hard problem. In addition the sparseness in solutions of the search space increases as the problem is scaled up, as is the case with the results for many problems reported in [Koza, 1994]. The question is then how can we improve the performance of GP for very sparse solution spaces?

3 Related Work

In the next two sections we summarize two extensions to GP that point out the possibility of evolution of new functions [Koza, 1992] or emergence of modules [Angeline and Pollack, 1994] as possible ways to cope with complex problems.

3.1 Automatic definition of functions

Automatic definition of functions (ADF) is an extension of the GP paradigm to cope with the automatic decomposition of a solution function [Koza, 1992]. In ADF-based GP each individual has a fixed number of components: functions to be automatically evolved (having a fixed number of parameters) and result producing branches. Each component is a piece of LISP code built out of specific primitive terminal and function sets, and is subject to genetic operations. Functions represent fragments of code that can be called in the program itself, playing the role of reusable subroutines. Evolution of definitions for the functions as well as of the program body is a result of the fitness pressure on population individuals.

Let us take a program pattern with two automatically defined functions (ADF0 and ADF1) and a result producing branch with one body. Then one has to distinguish between terminal sets and function sets for the body of ADF0, the body of ADF1 and the program body. In the example presented in figure 3 terminals from the initial terminal set are not included in the terminal sets for the function branches.

The primitive function and terminal sets are defined such that the components form a fixed hierarchy. Genetic operations are constrained depending on the components on which they operate. For example crossover can only be performed between components of the same type. Although the hierarchy of components is fixed, it boosts the power of solution discovery especially for problems with regular solutions or decomposable into smaller subproblems.

A generalization of ADF, hierarchical automatic definition of functions, answers a natural question regarding the default structure of the function sets for ADFs: lower order ADFs can be called from higher order ADFs.

GP extended with ADFs (ADF-GP) is theoretically not more powerful than standard GP but it is more efficient. Two main differences are: First ADF-GP develops much more complex programs. The size of the program body would "explode" if we did an inline substitution of ADF down to the basic primitives in the program body (see the definition of the expanded structural complexity notion in section 5). Secondly ADF-GP is able to make larger jumps in the search space. For example a mutation in the lowest ADF level, ADF0, called in higher level ADFs radically changes the behavior of the body of the program. The size of trees is kept reasonably down, while the power of the resulting program is increased.

Note that functions are not shared between individual programs. Functions may have no clear meaning from the point of view of the problem solved, they may not correspond to specific subgoals related to the problem at hand. We do not a priori know what a subproblem is. Functions are not explicitly associated to problem subgoals even in the case when we know what a problem subgoal is. Ultimately, the effort to tune up the architecture may not be negligible.

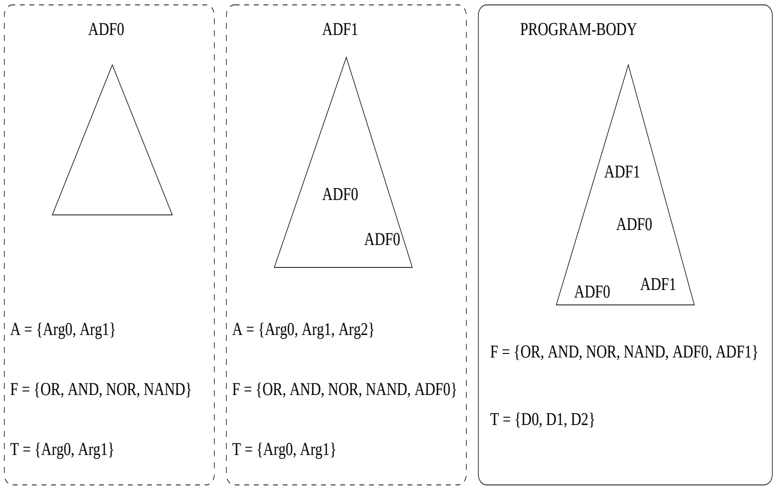

Figure 3: An individual program with two automatically defined functions. It consists of three branches: ADF0, ADF1 and a result producing branch with one body. Each branch has a set of arguments $\mathcal{A}$ (only for ADFs), a function set $\mathcal{F}$ and a terminal set $\mathcal{T}$ which are established in the problem definition

The main advantages of the approach are its generality, flexibility, and performance as proved in many examples ([Koza, 1994], [Kinnear, 1994]). ADF-GP automatically discovers how to decompose a problem into subproblems how to solve the subproblems (ADFs) and how to combine the solutions to subproblems (in higher level ADFs and program body).

3.2 Module acquisition

The *module acquisition* approach [Angeline and Pollack, 1993b] is applied more generally to evolutionary algorithms (an umbrella term covering GP and Evolutionary Programming). The ideas implemented within the genetic programming paradigm are illustrative and will be described shortly here.

This approach is based on the creation and administration of a library of modules which extend the problem representation and on the use of two new genetic operators, compress and expand [Angeline and Pollack, 1994], that control the process of modifying population individuals. 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 function's parameters are determined by the places where the tree is trimmed. The module's name is used to extend the basic language.

The *compress* operator randomly chooses a node in a program tree and extracts a module from it. *Expand*, implements the inverse function of compress. It simply expands a selected module name with the module itself back into the program tree where it was created.

Compression freezes possibly useful genetic material, by protecting it from the destructive effect of other genetic operators [Angeline and Pollack, 1993b]. It also increases the expressiveness of the base language and helps in decreasing the average size of individuals in the population, while maintaining the same power. As a side effect, if used alone, it may generate a loss of diversity in the population. The expansion operator overcomes this problem (see figure 4.)

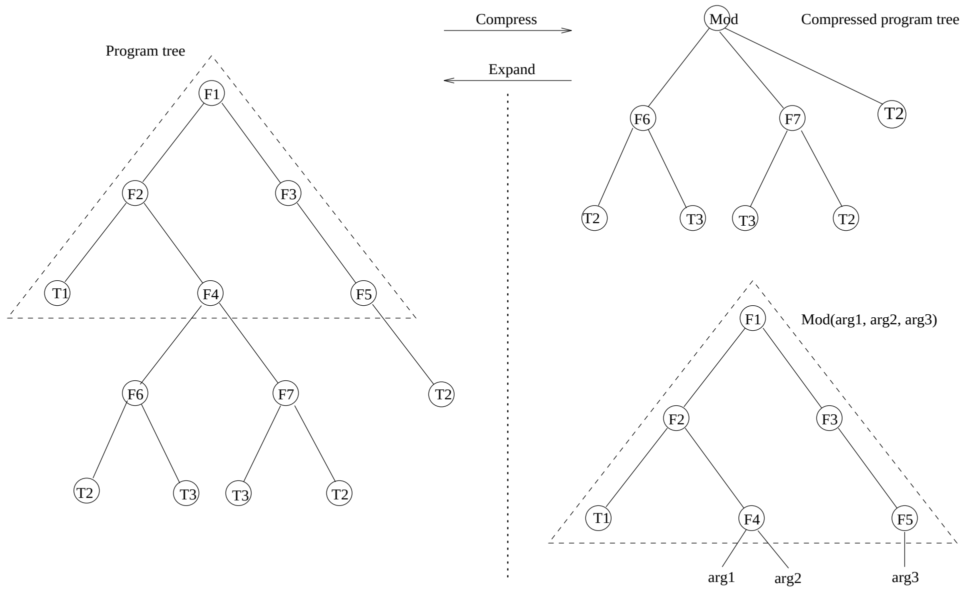

Figure 4: Operators defining the modules approach

It is interesting that in [Angeline and Pollack, 1994], the authors talk about the worth of a module, but they attribute to it a rather passive role. The module's worth is the number of times the module has been used since its birth, in subsequent generations. If a module is not *frequently* used, it means it is not viable in the competition with other individuals. The authors need a way to discard modules, as their number increases tremendously, and module usefulness seems to be an intuitive criteria.

One of the problems with this approach is that the representation is extended with too many modules, most of which are unuseful. Moreover, a library of modules has to keep track of all these modules, increasing memory requirements and overhead. Problem specific modularizations emerge, but they might have no meaning from the application's point of view.

A performance comparison of ADF and module acquisition is presented in a recent paper [Kinnear, 1994]. The author analyzes the performance of the two methods and many other variations of the methods and attributes the better performance of ADF to the repeated use of calls to automatically defined functions and to the multiple use of parameters. An attempt to explain the course of evolution in GP based on an understanding of what building blocks are appears in [Tackett, 1993]. The idea that frequent subtrees in one individual correspond to synthesized features suggests the conclusion that those subtrees comprise "building blocks".

4 Building Blocks

We will cope with some of the problems of these earlier approaches. Most importantly we will give a clear interpretation of what good fragments of genetic material are. GP can take advantage of its own evolutionary trace in order to discover good genetic material and use it to adapt the search process.

4.1 Foundations

In Genetic Algorithms (GA), the schemata theorem [Holland, 1992], [Goldberg, 1989] summarizes the effect of fitness-proportionate reproduction, crossover and mutation. Schemata are template strings representing sets of individuals in the search space. Schemata are defined by strings over the (usually binary) alphabet extended with a *don't care* symbol.

The importance of schemata lies in the characterization of how GAs work, both at the representation and processing levels. The schemata theorem proves that individuals that match short defining length (distance between the first and the last specific string positions) schemata with fitness values above population average will be selected in an exponentially increasing number of samples in the next generations.

Thus at the representation level, schemata outline the role of *building blocks*. A building block is a schema of short defining length whose defined positions tend to improve the behavior of individuals that match it and which describes fragments of a final solution. Individuals in the population provide a partial evaluation of schemata they match. GAs work best when the internal representation encourages the emergence of useful building blocks that can subsequently be combined with each other to improve performance.

At the processing level schemata outline the implicit parallelism of GAs. A population of M individuals actually samples $O(M^3)$ schemata (for binary representations). Individuals in the population provide a partial evaluation of schemata they match.

GAs work best when the internal representation encourages the emergence of useful building blocks that can subsequently be combined with each other to improve performance. [Davidor, 1989] outlines that choosing a representation for a GA is more of an art. The schemata theorem provides only general rules, it does not assess the suitability of a representation. Epistasis (concept derived from the Greek words "epis" - stand, and "stasis" - behind) describes gene interaction, and can be theoretically used as a measure of suitability of a representation. The design of a good representation should reconcile two conflicting goals: allow easy exchange and variation of genetic material and define operators that preserve as much genetic material form the parent strings as possible [Deugo and Oppacher, 1990].

[Antonisse, 1989] argues for the idea that a more expressive alphabet carries more power in an informationally equivalent environment. He uses a different interpretation of schema notation, in which don't care symbols quantify over subsets of symbols that can occupy a given string position. Informational equivalence means that the same number of distinct strings can be specified using each of the different alphabets. The power of an alphabet is given by the number of schemata sampled by each individual in the population.

GP provides a very natural and well understood representation for a program in the form of a parse tree. This is one of the reasons for which LISP appears to be a suitable high level language for implementing the GP paradigm. The epistasis idea can not be easily applied to GP approaches, where the interaction of parts of a program is very hard to decipher. Moreover, no precise definition of a GP schema exists, and no alternative schemata theorem has been proven for GP. One more difficulty is outlined in [Angeline, 1994]. GP individuals represent code, as opposed to the GA representation. There exists a complicated many-to-many mapping between genotypic and phenotypic features.

In the next section we define the concepts of blocks and building blocks and show that the interpretations given provide the basis of a theory for adapting problem representation and making the GP search more efficient.

4.2 Building blocks in GP

An individual in GP is a piece of "code" in a base language (LISP in our experiments). Although the interactions between the different fragments of code may be very entangled, we argue in favor of the natural formation of genotypic features, or GP *building blocks*.

We define blocks as *entire* subtrees of a given maximum height from population individuals. This represents the bottom-up approach in figure 5. (In contrast, the module approach [Angeline and Pollack, 1994] starts with subtrees of any depth and chops off all branches at a given depth.) We will show that the bottom-up approach enables the definition of blocks whose usefulness *can* be evaluated. A key idea is that although it may be important to keep track of blocks of arbitrary size, only monitoring the merit of small blocks (with height less that g_{max}) is feasible. Furthermore useful blocks tend to be small and the process can be applied recursively to discover more and more complex blocks.

A block can be generalized by substituting each occurrence of a terminal of the block by a variable (descriptive generalization in [Michalski, 1983]). This process transforms a block into a parameterized function whose parameters are given by the block variables. Figure 5 presents an example of a function F_b generated in this approach. Steps *3a* and *3b* from figure 6 implement this idea (see section 6).

By analogy with Goldberg's definition, a GP building block corresponds to a parameterized piece of code that, once used as a problem primitive, has noticeable good effect on the behavior of the individuals. Thus, a good building block has the potential to successfully extend the set of function primitives of the problem. For a comparison with GA, note that if we place in the initial population individuals that match a-priori known useful schemata, the convergence of the GA improves drastically. We explain this by the reduction in dimensionality of the search space caused by the "biased" initial population (GA). In GP such an improvement could be explained by the more powerful set of primitives.

The goals we aim at next are the discovery of new functions based on an analysis of blocks that leads to candidate building blocks and the dynamic extension of the function set with these new functions created on-the-fly from promising blocks. We call such a representation an *adaptive representation*, as it is automatically changed while the GP program searches the space of admissible solutions.

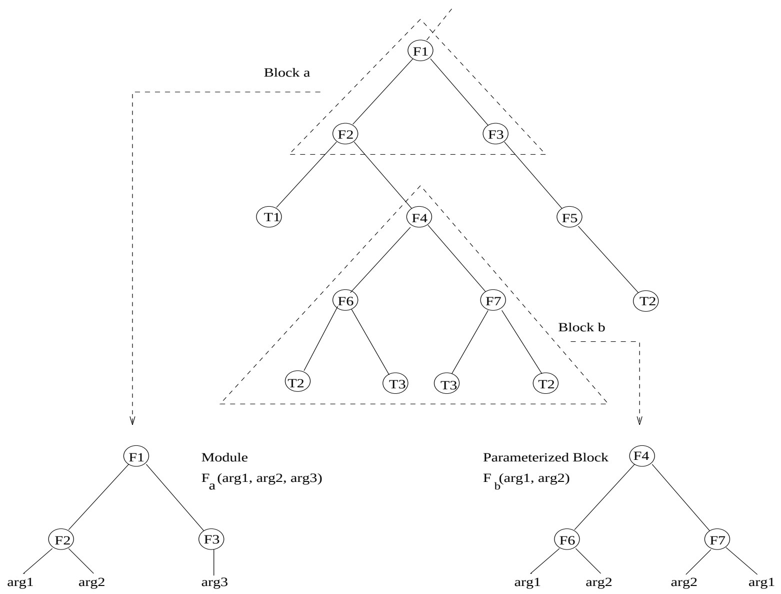

Figure 5: Bottom-up approach to block selection vs. the module approach. The bottom-up approach (right) generalizes the block b to a function $F_b(arg1, arg2)$; The modular approach (left) generalizes an arbitrary portion of the tree.

5 Analysis of Candidate Building Blocks

There exist two obvious choices for candidate building blocks: fit blocks and frequent blocks. A block whose merit (fitness or frequency) exceeds a threshold becomes a candidate.

Frequent blocks are determined by keeping track of how often a block appears in the entire population. Surprisingly, frequent blocks are not necessarily useful building blocks. By analogy to the GA schemata theorem, a good building block spreads rapidly in the population, and this determines a high frequency count for the block. For example, in the even parity problem (see section 7), if we augment the function set with the exclusive OR function (XOR), then XOR (a building block) will soon become dominant in fit program trees. The problem is that the *converse* of the GA schemata theorem is not true. Poor blocks, i.e. blocks that are identities or add no functionality, may be frequent and should not be considered as candidates, although they may have a role of preserving recessive features (introns in [Angeline, 1994]).

In general, if a building block is not discovered early enough, and population diversity is not preserved due to biased operators or poor tuning of the application parameters, other blocks will account for high frequency counts and the search problem will get stuck in a local minimum. A typical histogram plotting the frequency of all the final blocks in one experiment is presented in figure 9. It shows that most of the blocks, even the ones that are part of final solutions, have very low frequency counts.

Some other conclusions obtained by monitoring frequent blocks indicate their unsuitability in estimating block usefulness. Blocks that appear in a final solution may be discovered very late in the search process. They are not necessarily responsible for the evolution process, usually having a low frequency count. Frequent blocks in early generations may become rare in late generations. Simpler and simpler blocks (and highly probable to be less useful) appear in the top of frequent blocks as the number of generations increases.

A much better choice for discovering building blocks is to consider fit blocks. We can incrementally check for new fit blocks instead of relying on expensive statistics in the population.

Several methods can be used to evaluate the fitness of a block. First, one can use the program fitness function to evaluate blocks too. This has the advantage of requiring no more domain knowledge than the knowledge built into the fitness function, but is not a general method. Second, one can use a slightly modified version of the fitness function. For example the block fitness may be measured only on a reduced set of fitness cases, dependent on the variables used in the block. This method actually scales the fitness function down to cope with a smaller size problem of the same type (see the example from section 7). Third, the user can provide one or more *block fitness functions* based on supplementary domain knowledge which is often available and can be successfully used for complex problems.

As expected fit blocks are very useful in dynamically extending the problem representation by means of definition of new global functions. We will see that a highly fit block generated in the initial population spreads in the population very easily and once the population evolves on a right path, good blocks appear among the frequent blocks although frequent blocks themselves would not necessarily help driving evolution on an ascending path, if used to generate new functions.

6 Extended GP Paradigm

6.1 Adaptive representation

Based on an analysis of promising blocks (candidate building blocks) we hypothesize on-the-fly new functions which dynamically extend the function set. Note that the initial members of the function/terminal set are the initial problem building blocks.

The evolution process is naturally split into *epochs*. An epoch is a sequence of consecutive generations in which no candidate building blocks are discovered (have merit above the threshold). Once at least one such block is discovered in a generation, the population undergoes substantial changes. The most fit individuals are retained, while the less fit ones are replaced by new individuals randomly generated using the extended function set. This is the beginning of a new epoch. The proportion of individuals replaced is a parameter of the algorithm called the *epoch-replacement-fraction*. Population enrichment can be done as soon as new candidate blocks are discovered or after waiting for a number of generations.

Our main goal is thus to identify building blocks among the blocks of height less than g_{max} present in the population. To do this we take advantage of the dynamics of the population (as recorded in an execution trace) and search for new blocks, i.e. blocks appeared in the population due to genetic operations in the current generation. All new blocks can be discovered in $O(M)$ time, where M is the population size, by marking the program-tree paths actually affected by a GP operator and by examining only those paths while searching for new blocks. The data structure that records the search control information, called a *super-tree*, is presented below.

The information used in discovering useful blocks may be either global, domain independent information extracted from problem traces or domain dependent knowledge. In the absence of any heuristic rules (block fitness functions) for hypothesizing what good blocks are, the frequency criterion can be useful, provided that we check that functions generated this way bring an improvement of the evolutionary process (for example, an increase in average fitness of individuals that incorporate the function).

The skeleton of the GP algorithm that incorporates the AR component is presented in figure 1. Figure 6 outlines the steps involved in adapting the problem function set $\mathcal{F}_i$ at generation i.

Adapt_representation$(\mathcal{P}_i, \mathcal{F}_i, \mathcal{F}_{i+1})$

1. Discover candidate building blocks $\mathcal{BB}_i$ by evaluating each block's merit$(\mathcal{P}_i)$;
2. Prune the set of candidates$(\mathcal{BB}_i)$;
3. For each block in the candidate set, $b \in \mathcal{BB}_i$, repeat:
   (a) Determine the terminal subset T_b used in the block(b);
   (b) Create a new function f having as parameters the subset of terminals and as body the block(b, T_b);
   (c) Extend the function set $\mathcal{F}_{i+1}$ with the new function$(\mathcal{F}_i, f)$.

Figure 6: GP extension: the Adaptive Representation component

By extending the problem representation, the algorithm naturally builds a hierarchy of functions. A level in this hierarchy corresponds to all functions defined in a given generation. The set of primitive functions that may be used in defining a new function is given by the set of all functions defined in previous generations and the initial function set.

6.2 Incremental discovery of building blocks

In this section we prove that all new blocks can be discovered in O(M) time, where M is the population size. This implies that the AR module can keep track of population blocks in an efficient way.

Whenever a pivot (see section 2) is selected in a program-tree, the path from the root to the pivot position, called *pivot-path*, is recorded in the structure representing the new individual created. The structure representing an individual in the population consists of the following elements:

- program

- fitness value (standardized, normalized, etc.)
- genetic operation used at creation (random initialization, reproduction, crossover or mutation)
- pivot-path
- other book-keeping information

For example, if the pivot point used to generate the individual having the program-tree from figure 7 is e, then the pivot-path is the ordered list of tree nodes $\{a, b, c, d\}$.

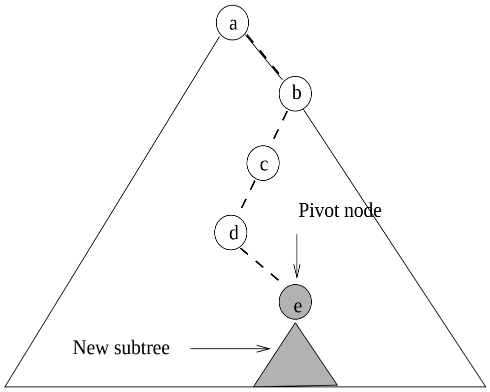

Figure 7: Pivot path of an individual for which the highlighted subtree represents new genetic material pasted as a result of a crossover or mutation operation

The pivot-path and the genetic operation used to create an individual are employed to determine efficiently the new blocks appeared in the population. For crossover and mutation, the pivot-path represents the path to a region of the tree where changes were done. We look for new blocks in a neighborhood of this region. By keeping track of pivots we take advantage of the locality of changes in the individuals of the population.

Simple reproduction creates no new genetic material. In the case of random initialization the whole program tree/subtree has to be searched for useful blocks. This is done just for newly created individuals at the beginning of a new epoch or after a mutation.

In the case of crossover, the search for useful blocks is performed by taking the nodes of the pivot path in reverse order (starting with the last) and trying to find a subtree of height less or equal than g_{max} rooted at the corresponding node of the pivot path. At most

$$max\{0, g_{max} - Height(e)\} \leq D$$

nodes of the pivot path may be considered as new candidate block roots, thus at most $M \cdot D$ steps are needed to determine all the new blocks appeared in the population at generation $i+1$. D is usually a small constant (a typical value is 17) and M is much bigger than D (e.g. 4000 or bigger), thus the time to needed to determine all new blocks is of the order O(M).

The average number of nodes in the pivot-path can be estimated more precisely for a full binary tree of depth D by taking into account that pivots are randomly selected. It is given by

$$Average\ pivot - path\ length = \sum_{i=1}^{D} i \cdot \frac{\text{total nodes of depth } i}{\text{total nodes in the tree}}$$

This can be proved to be

$$Average\ pivot - path\ length = (D-1) + \frac{D+1}{2^{D+1}-1} \leq D$$

Each of the nodes in the program records the most recent generation number when it played a pivot role. Based on this information the system can extract a tree describing the structure of a given program at the end of the run, called a *structure tree*.

Super trees correspond initially to program trees randomly generated. Undergoing evolution, a super tree gathers information on how the individual program has evolved. In summary, here are the main characteristics of representing individuals as super-trees:

1. Keep properties relevant to substructures in the tree attached on the nodes of the tree.
2. Observe an identical behavior with a program tree. Equivalent program trees can be recovered from super trees, and can be evaluated with little overhead.
3. Allow for an efficient search and examination of building blocks.
4. Enable the extraction of a structure tree, a description of how a tree has been formed during the evolutionary process.

Structure trees are an auxiliary means for understanding the way the best of run individual (or any other individual) is created. This kind of information is useful in a visual analysis of the evolution of individuals. For example, the structure tree of the solution obtained in figure 2 is given in figure 8.

6.3 Complexity measures for program trees in adaptive representations

In many GP applications the fitness function is extended, for reasons of efficiency or convergence, with *size pressure* or *execution bound* components. This has been done mostly in an empirical way. For example, individuals with too big a size, number of state-changing operations (as in the LAWN MOWER or ARTIFICIAL ANT problems described in [Koza, 1994]) or "running time" are penalized. It is very important how we measure size or running time. This section outlines the differences between different measures that can be used.

Suppose an individual is represented by a program which calls discovered functions which may call older discovered functions. Nonetheless the call graph based on the caller-callee relation has no cycles. Let $Size(F)$ be the number of nodes in the tree representing a

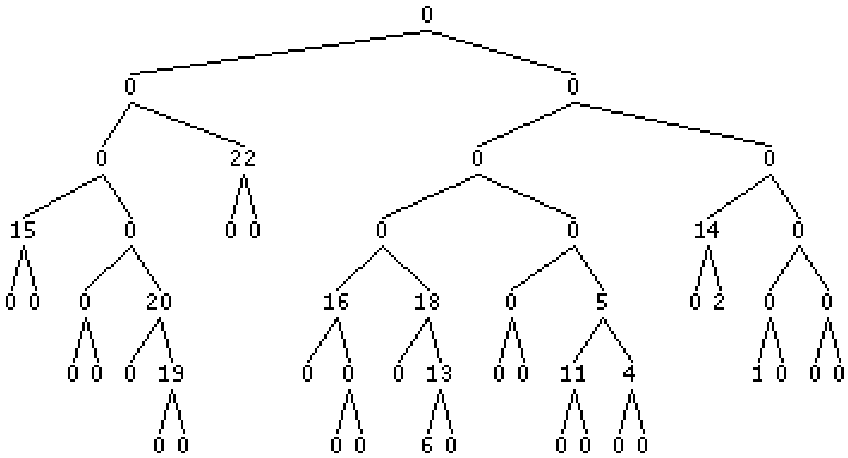

Figure 8: The structure tree for the program tree in figure 1. Node labels represent the generation number when nodes were chosen as pivots or 0 if they remained unchanged

program F. Let F_0 be the program tree representing an individual T_0 which contains direct or indirect calls to $F_1, F_2, ..., F_m$. We define *structural complexity* $SC(F_0)$ and *evaluational complexity* $EC(F_i)$ in the following way:

$$SC(F_0) = \sum_{i \leq j \leq m} Size(F_j) \tag{2}$$

$$EC(F_i) = Size(F_i) + \sum_{i < j \leq m,\ F_j \text{ called in } F_i} EC(F_j) \cdot |Calls(F_i, F_j)| \tag{3}$$

where $|Calls(F_i, F_j)|$ is the number of times F_i calls F_j. In standard GP, where no functions other than the primitive ones are used, the structural and evaluational complexities are equal to the program size $Size(T)$. Assuming that functions from the initial function set are executed in unit time, the evaluational complexity shows how many time units it takes to execute an individual program.

Note that $EC(T)$ is different from the structural complexity of individuals after the "inline" expansion of all the called functions down to the primitive functions, which we call "expanded structural complexity" and represents a true measure of the size of an individual if we had to built it from primitive functions. It corresponds to the notion of size circuit complexity from complexity theory. $EC(T)$ is a lower bound for the expanded structural complexity of T. The two measures are equal when each formal argument of a function is used only once in the definition of the function (as in the case of *modules*).

6.4 Minimum descriptional complexity

We can view the problem of determining a program that explains the set of examples or optimizes a fitness function as one of hypotheses formation: we look for the best program that explains a set of constraints (data) captured by the fitness function. Rissanen's *minimum description length* (MDL) principle offers an answer to approaching the problem. It states that the best theory to explain a set of data is one which minimizes the length of the data description together with the hypothesis description. In general, problems such as the inference of a decision tree that best explains a set of examples [Quinlan and Rivest, 1989], the construction of a finite automaton or the inference of a boolean function that satisfies a set of constraints are all problems that match the described pattern and can be solved using the MDL principle [Li and Vitanyi, 1993]. MDL is also called *stochastic complexity.*

The MDL principle advocates in favor of a hierarchical representation of evolved programs (see appendix A). A more complex behavior can be obtained based on a more complex, hierarchical organization. For such an organization, the size of individuals and discovered functions is kept within reasonable bounds while the structural complexity of individuals is much bigger. The power of GP using automatically discovered functions to solve problems is significantly increased too ([Koza, 1992], [Koza, 1994], [Kinnear, 1994]) (see also section 7).

Moreover, the descriptional complexity can be used as a measure for the fitness of an individual T that would drive GP towards discovering solutions with a smaller descriptional complexity. It balances the requirement of getting a simpler description of a solution tree T and the requirement of minimizing the number of misses. The mechanism for adapting the representation while searching for solutions represents a natural way to generate a hierarchy of more and more complex functions (a hierarchical representation) and to make possible the discovery of a solution of small (or even minimal) description length.

We suggest here that evolution minimizes the stochastic (descriptional) complexity, but increases the structural complexity of individuals. Individuals tend to take a hierarchical organization because this is a way of working towards achieving a minimum descriptional complexity, for a given "power" in explaining the input fitness cases. This is true to certain degree even with simple fitness measures incorporating rewards for hits and penalties for misses as well as a size pressure component, that have been empirically used so far in GP applications.

7 Experimental Results

The results reported herein are based on runs with the AR GP kernel. Its implementation has been done in Allegro Common Lisp, version 4.2b. A graphical interface for representing program trees and structure trees has been implemented in Common Lisp Object System (CLOS) and Common Lisp Interface Manager v2.0 (CLIM) for Sun Systems. The AR kernel has been built built on top of the standard GP LISP code described in [Koza, 1992].

The EVEN-N-PARITY problem (introduced in section 2.1) is solvable by problem decomposition into simpler subproblems and thus represents a good test bench for the discovery of more and more complex building blocks.

We used the following parameter settings. Population size was $M = 4000$. The maximum depth of individuals was $D = 17$, while new individuals may have a depth of maximum 6. The maximum number of generations was $G = 50$. We used both fitness-proportionate and tournament selection (with similar results), no mutation, and a crossover rate of 0.7 at function points and 0.2 at any point. The fitness proportionate reproduction fraction was 0.1. We have not experimented with other values of these parameters, but rather have used the values reported in [Koza, 1992] for result comparability reasons.

The block fitness function was the same as the fitness function, with one slight change. Hits are evaluated on a subset of the set of fitness cases, determined by fixing the values of variables not used in the block to arbitrary values. Only blocks with a maximum number of hits are considered as candidates. No pruning function has been initially considered (step 2 in figure 6). The results show that many different functions with the same number of arguments are discovered. Anyway, all of the functions solve a lower order EVEN-PARITY and are logically equivalent, so only one of them is needed. An epoch-replacement-fraction of $\frac{1}{2}$ gave good results when solving EVEN-N-PARITY with N up to 8. For bigger orders, we chose a smaller value to keep the computational overhead due to adapting the representation lower. In general, the bigger this fraction, the larger the computational effort and memory requirements for a run, so one has to trade off the power obtained with the costs employed.

In the experiments done with AR-GP, we have used structural complexity as a size pressure component by replacing $Size(i)$ with $ST(i)$ in equation (1). Expanded structural complexity was used as an execution time bound component.

We solved all PARITY problems up to order 11 on a SUN SPARCstation 10 by adapting the representation based on fit blocks.

Here we present a discussion of frequent/fit blocks taken from a run for EVEN-3-PARITY, a complete example of a run for EVEN-5-PARITY and an example of a run for EVEN-8-PARITY.

7.1 Fit and frequent blocks

An example of the evolution of most frequent blocks when disabling the adaptation of the function set in a run of the EVEN-3-PARITY problem, is presented in figures 9 and 10. None of the most frequent blocks (MFB) appeared in a final solution. The most frequent blocks at the end of generation 22 when a solution is found are:

1. (NAND (OR (NAND D2 D1) (OR D2 D2)) (NAND (AND D0 D1) (NOR D2 D2))) appeared 132 times and has 6 hits;
2. (NAND (NAND (NOR D0 D1) D2) (NAND (AND D0 D1) (NOR D2 D2))) appeared 42 times and has 4 hits;
3. (NOR (NOR D2 D1) (AND D0 D2)) appeared 41 times and has 4 hits;
4. (AND D0 (NOR D0 D2)) appeared 35 times and has 4 hits;
5. (OR (NAND (OR D2 D0) D2) (OR (AND D1 D0) (NOR D1 D0))) appeared 31 times and has 2 hits (MFB5).

Hits were measured for the total number of 8 fitness cases. Note that MFB5 contains a useful sub-block which is nothing but XOR applied to D_0 and D_1 (the right subtree of its root), but its power can not be harnessed unless we detect it and generalize the sub-block's (a fit block) behavior.

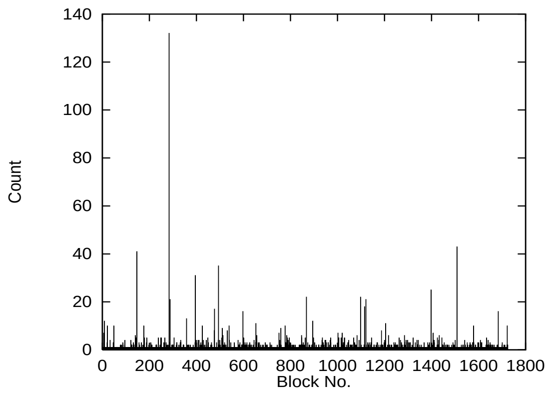

Figure 9: Final distribution of block frequencies in a run of EVEN-3-PARITY

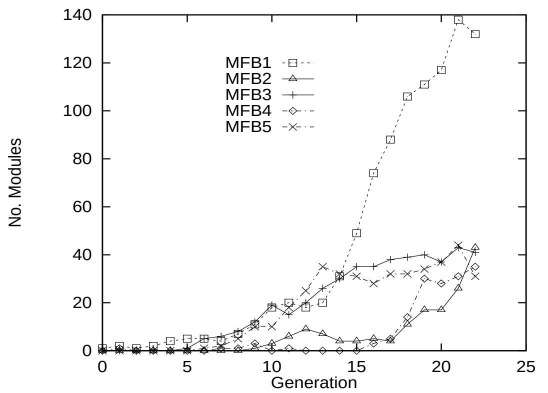

Figure 10: Evolution of most frequent blocks in the EVEN-3-PARITY example

There are 13 blocks with a fitness of 6 at generation 22. One example is (NAND (OR (NAND D2 D1) D0) (NAND (AND D0 D1) (NOR D2 D2))).

By adapting the representation based on fit blocks, solutions are obtained in at most two generations. Performance is improved dramatically in this case as well as in much more complex ones. There is no such improvement if the function is extended based on frequent blocks.

It is interesting to note that a statistical analysis of the frequent blocks and functions used after the function set is extended on the basis of fit blocks, outlines the importance of

Generation 0.

- New functions [F892]: (LAMBDA (D0) (NOR (AND D0 D0) (OR D0 D0))); [F893]: (LAMBDA (D2 D1) (NAND (NAND D2 D1) (OR D1 D2)))
- The best individual in the generation is: (NAND (NOR (NOR D4 D3) (NOR D4 D0)) (AND (OR D1 D2) (NAND D2 D1)))

Generation 1.

- New function [F894]: (LAMBDA (D3 D1 D2) (F893 (F892 (F892 D3)) (F892 (F893 D1 D2))))
- The best individual in the generation is: (OR (NOR (NOR (F892 (NOR D4 D1)) (NAND (NOR D1 D1) (NAND D3 D0))) (NAND (F892 (NAND D0 D4)) (F892 (OR D0 D3)))) (F893 (NAND (F892 (F893 D2 D4)) (OR (OR D0 D3) (F893 (F893 D2 D0) (OR D4 D1)))) (OR (F893 (NAND D1 D0) (F893 D1 D3)) (F892 (F893 D4 D4)))))

Generation 2.

- New function [F895]: (LAMBDA (D4 D3 D0 D2) (F894 (F892 D4) D3 (F893 D2 D0)))
- The best individual in the generation is: (NAND (F892 D1) (F893 (F894 D3 D4 D0) D2))

Generation 3.

- New function [F896]: (LAMBDA (D1 D0 D2 D4 D3) (F893 (F895 D0 D1 D4 D4) (F894 D3 D4 D2)))
- The solution found is: (F892 (F893 (F893 D3 (F894 D4 D0 D1)) (F892 D2)))

Table 1: Evolutionary trace for a run of EVEN-5-PARITY

the new functions created, and thus of the hypothesized building blocks. The new functions rapidly become dominant in the population, if they are useful. We can thus evaluate extensions of the function set. This is of high importance especially when the rules used in establishing the merit of building blocks are heuristics rather than precise ones (as in the case of the EVEN-PARITY examples).

7.2 Emergence of hierarchical representations

We present a complete example on how AR-GP works for a run of EVEN-5-PARITY. A trace of the run is given in table 1 and is explained below. With AR all runs for EVEN-5-PARITY needed at most 5 generations.

In generation 0 two functions are discovered: F892 (NOT) and F893 (negated XOR or even parity on 2 bits). In generation 1 one more function is discovered. It is based on the previous two discovered functions (F894 calls both F892 and F893 and represents even parity on 3 bits). In generation 2 a function of four variables is discovered. It is based on the previous three discovered functions, and solves parity on 4 bits. In the last generation (3) an individual with maximum number of hits (32) is found. Also a function with five parameters is discovered and it represents nothing but the EVEN-5-PARITY function.

It is important to point out the hierarchical structure of the functions dynamically created and their semantics. Figure 11 presents the final call-graph induced in solving one instance of the problem. Different levels in the call graph correspond to higher epochs of the evolutionary process. Functions on the same level do not call one another. Only functions in different epochs may take advantage of the genetic material discovered previously. The foundation of the hierarchy is made of the primitive functions included in $\mathcal{F}_0$. Thus the call-graph for the functions created out of building blocks is extended hierarchically, in a bottom-up fashion. It is remarkable that the whole structure is discovered in only 4 generations.

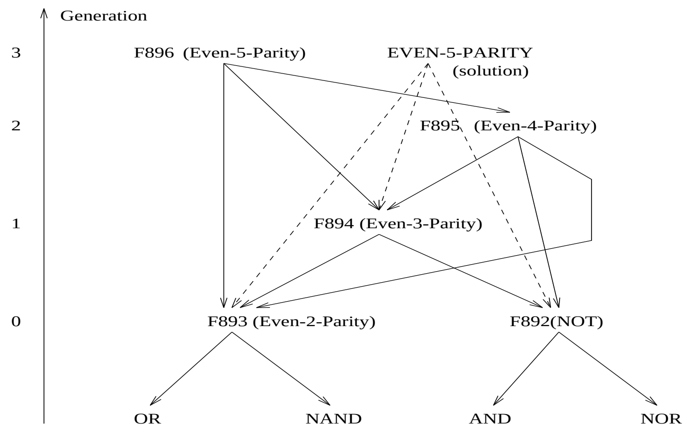

Figure 11: Call graph of the extended function set in the EVEN-5-PARITY example at the end of a run

An analysis of the evolution of the most frequent final blocks is interesting too. The three most frequent blocks are:

1. (F893 D3 (F894 D4 D0 D1)) appears 131 times and has 16 hits, out of 32.
2. (F895 D3 (F894 D2 D1 D3) D3 (OR D1 D1)) appears 102 times and attains 16 hits too.
3. (F894 (F895 D3 (F894 D2 D1 D3) D3 (OR D1 D1)) (F892 (F895 D1 D4 D1 D1)) D0) appears 22 times and has 32 hits (it would be chosen for generating a 5 input subfunction).

The first two blocks are not really very useful. The third one confirms that if the evolution path is a correct one then the population has a high potential to create useful building blocks. The GP algorithm will disseminate it in the population.

Generation 0. New functions [F681]: (LAMBDA (D3) (NOR D3 (AND D3 D3))); [F682]: (LAMBDA (D4 D3) (NAND (OR D3 D4) (NAND D3 D4))) **Generation 1.** New function [F683]: (LAMBDA (D4 D5 D7) (F682 (F681 D4) (F682 D7 D5))) **Generation 3.** New functions [F684]: (LAMBDA (D4 D5 D0 D1 D6) (F683 (F683 D0 D6 D1) (F681 D4) (OR D5 D5))); [F685]: (LAMBDA (D1 D7 D6 D5) (F683 (F681 D1) (AND D7 D7) (F682 D5 D6))) **Generation 7.** The solution found is: (OR (F682 (F682 (F683 D4 D2 D6) (NAND (NAND (AND D6 D1) (F681 D5)) D1)) (F682 (F683 D5 D0 D3) (NOR D7 D2))) D5)

Table 2: Important steps in the evolutionary trace for a run of EVEN-8-PARITY

Table 2 presents the main steps of the trace in a run of EVEN-8-PARITY and figure 12 presents the final call-graph induced in that run. A hierarchy similar to the one in figure 11 is rediscovered from scratch. Figure 12 outlines also the epochs of the evolutionary process. The final solution contains calls to the discovered functions as well as to the initial functions. In this run, the number of arguments of discovered functions is limited to maximum 5. Without this constraint, a hierarchy containing all parity functions is discovered, but the computational effort is higher due to the overhead introduced by generating new individuals in new epochs.

Note that there is no imposed order in which discovered functions call one another. The hierarchy is generated automatically and it observes the problem semantics. All functions on a given level are independent (discovered over one or more generations), and higher levels correspond to higher generation numbers.

For higher order EVEN-PARITY problems it becomes harder and harder to find solutions without using the adaptive framework. The computational effort increases considerably as reported in [Koza, 1992]. The adaptive representation method discovers solutions to harder and harder problems, creating a hierarchy of functions like the one in figure 12 and enabling a problem decomposition into subproblems.

7.3 Evolution of complexity

Figure 13 presents the evaluational complexity, the size of the best of generation individual and the average values over the population. The structural complexity values are bounded from below and above by $Size$ and EC respectively. The best of generation individual becomes simpler and simpler due to size pressure and the variety of useful and more powerful blocks appeared in the population. Starting with generation 3, discovered functions begin to dominate the structure of the best of generation individual as they gradually replace the primitive functions. A statistical analysis of the frequent blocks after the function set is

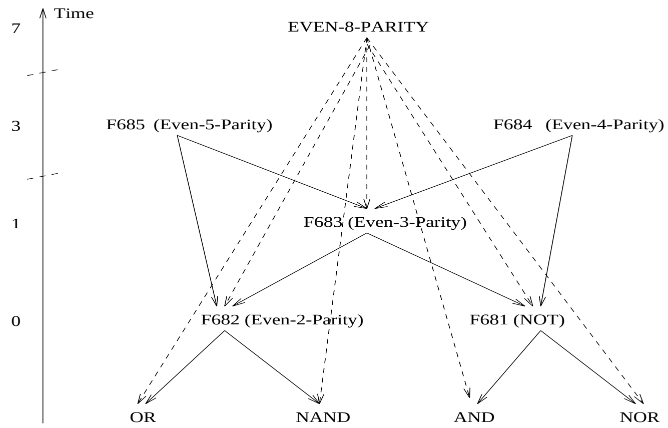

Figure 12: Call graph for the extended function set in the EVEN-8-PARITY example

extended on the basis of fit blocks, outlines the importance of the new functions created, and thus of the hypothesized building blocks. The new functions rapidly become dominant in the population, if they are useful. We can thus evaluate extensions of the function set.

Figure 14 outlines that the complexity of individuals increases considerably for a simpler problem (EVEN-5-PARITY) when standard GP is applied, without bringing noticeable improvements in the standardized fitness. A similar problem appears when the new functions created by AR do not correspond to good building blocks.

A continuation of the evolutionary process after a first solution is found enables the algorithm to find diverse solutions. A fitness measure that includes a structural complexity pressure component determines the selection of better solutions. Figures 15 and 16 synthesize this observation for a run of EVEN-5-PARITY in which the first solution was obtained very early (generation 3).

7.4 Comparison of results

We conclude this section by summarizing the results obtained. A hierarchical representation may become extremely powerful and help solve the problem fast, especially when solutions contain symmetric or repeated patterns. This section has brought clear experimental evidence in this direction. The power of a hierarchical representation is explained by the following remarks.

- Each of the defined functions represents a piece of code that helps in solving the problem, either because it complies with the heuristic criteria set by the block fitness functions or the high frequency heuristic. The functions do not have to be homogeneous. Each may solve a different type of "subproblem". In the EVEN-PARITY examples, functions correspond to subproblems of the same type (parity problems of a lower dimension).

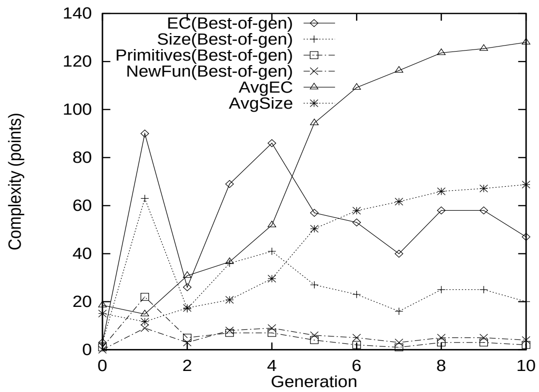

Figure 13: Complexity of best of generation individual and average values over the entire population in the EVEN-8-PARITY example

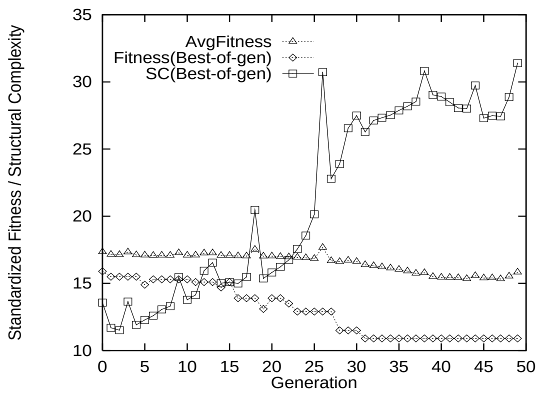

Figure 14: With AR inhibited, an EVEN-5-PARITY run shows permanent increase in structural complexity but a plateau in fitness

- Functions can be reused, so it becomes easier to incorporate and reuse good fragments of genetic material. The effect is a restructuring of the search space [Hinton and Nowlan, 1987]. Many programs that are far from optimal do not become candidates any more simply because it is less probable to generate unfruitful combinations of initial primitives (compare the "NewFunctions" and "Primitives" plots in the figures from section 7.3). In this sense we noted that many diverse solutions can be obtained once the right "ingredients" (hierarchy of functions) are present.

- The expanded structural complexity of individuals increases considerably compared to the complexity (size) of individuals generated from the initial function/terminal sets by plain GP. Evolution is correlated to an increase in complexity up to a given point, so it becomes easier to shortcut slow and long evolution periods. As a consequence, problems can also be solved in a fewer number of generations. Experiments with other

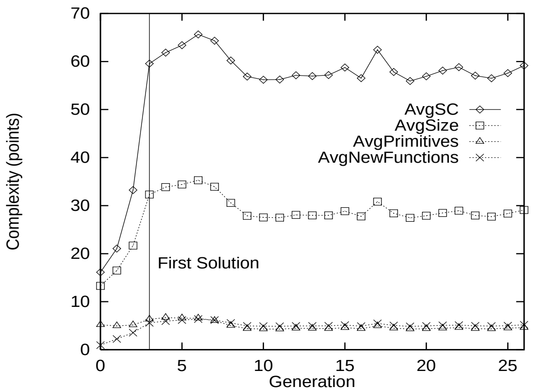

Figure 15: Evolution of average complexity measures for a run of EVEN-5-PARITY after a solution is found

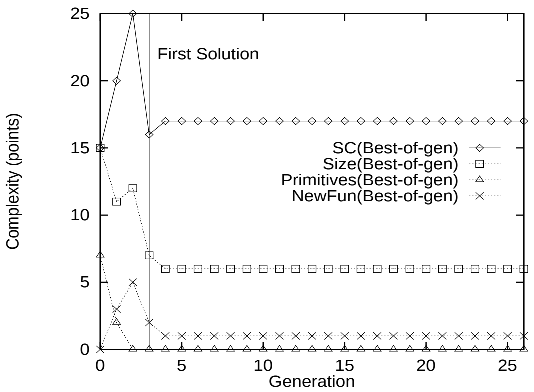

Figure 16: Evolution of best of generation individual complexity measures for a run of EVEN-5-PARITY after a solution is found

problems like LAWN MOWER or ARTIFICIAL ANT also illustrate the relation between program complexity (as defined here) and program quality (how well it solves the problem). Most often we do not accept programs of high structural complexity because of the inherent intractability of very complex structures. The hierarchical approach makes the problems tractable. A shift in representation really transforms a very difficult problem into an easy one.

- The the principle of diversity "being different can be as good as being fit" [Winston, 1992] has a higher probability of being satisfied when using an adaptive representation.

With ADF or AR the scalability of the even parity problem improves significantly. We have argued from a theoretical point of view that hierarchical structures are more powerful than structures based on the initial function set. Table 3 brings quantitative experimental evidence.

	EVEN-3		EVEN-4		EVEN-5		EVEN-8	
Method	*gen*	*SC*	*gen*	*SC*	*gen*	*SC*	*gen*	*SC*
AR-GP	2	17	3	15	5	32	10	41
GP#	5	45	23	113	50	300		
ADF-GP#	3	48	10	60	28	157	24	186

Table 3: Comparison of results (rounded figures): AR-GP vs. results reported in [Koza, 1994], marked (#). SC is the structural complexity

Row 1 from table 3 shows the number of generations to find a solution with 99% probability in one run and the average structural complexity of solution trees obtained for ten runs of AR-GP on EVEN-PARITY problems with population size $M = 4000$. Rows 2 and 3 present some comparative results taken from [Koza, 1994] for sample runs of GP with similar parameter values, but $M = 16000$.

8 Conclusions

Automatic discovery of problem representation primitives is a challenging goal that has been tackled in recent research in Machine Learning in general, and Genetic Programming in particular. Its importance comes from the fact that a representation with higher level primitives can transform a hard problem into an easy one. This paper shows that in GP useful genetic material can be discovered and used globally to extend the problem representation in a hierarchical way in order to improve performance.

In building a system it is desirable to distinguish the elementary components of the system from more complex components used in defining subsystems. The functionality of the complex system is based on its elementary parts which are *stable components*. The discovery of stable components when solving a problem could speed the search process, as the time needed for the complex system to evolve based on its subsystems is much shorter than if the system evolves from its elementary parts [Simon, 1973].

We claim that in GP genotypic features (building blocks) emerge that correspond to such stable components. They can be discovered in the population using a process of generalization of substructures (blocks) already incorporated in the individuals. Automatic adaptation of the representation is implemented by extending the function set with fit building blocks discovered. The effect of our approach is a dynamical reshaping of the search space and exemplifies the effect learning, in the form of acquisition of new functions, in the evolutionary process [Hinton and Nowlan, 1987]. Extraordinary improvements of the effort needed to find a solution have been observed for problems with regularity in their solutions.

This work distinguishes from other approaches to understanding what building blocks are in several major respects. As opposed to [Angeline and Pollack, 1994] where the functional role of a module is identified and the evolution of the representation language is suggested, we select, generalize and use blocks in a different way. Selection is based on estimated block usefulness determined by an analysis of the GP trace. Generalization identifies block leaves labeled identically. Use is based on the adaptation of the function set. No need for more genetic operators appears. The difference from [Koza, 1994] consists in the knowledge-driven strategy of defining functions. Functions are created based on the usefulness criteria and thus have a clear semantics. They can be called from any member of the population. There are no syntactic constrains imposed on the functions created, such as the number of arguments or the set of functions they can call. The discovered functions evolve into a hierarchy of functions of increasing complexity that solve subproblems.

We have not experimented with deletion of functions previously added to the function set, but our remarks and [Angeline and Pollack, 1994] argue for using the appearance frequency of these functions to decide upon their usefulness. Developing robust criteria for discovering useful building blocks in general could help in the design of GP applications for complex problems. On the theoretical level, issues such as problem complexity and its relation to problem representation, or defining robust/adaptive measures for implementing the fitness function would represent interesting lines of research. The structural and executional measures of complexity for a hierarchical problem representation help in understanding how complex the problem is and what kind of fitness pressure could create an efficient selective pressure on the GP population.

This extended GP paradigm also suggests a general way in which more domain knowledge can be used to aid in the discovery of useful building blocks and to adapt the problem representation dynamically.

References

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

[Angeline and Pollack, 1993a] Peter J. Angeline and Jordan B. Pollack, "Competitive Environments Evolve Better Solutions for Complex Tasks," In *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc, 1993.

[Angeline and Pollack, 1993b] Peter J. Angeline and Jordan B. Pollack, "Evolutionary Module Acquisition," In *Proceedings of the Second Annual Conference on Evolutionary Programming*, 1993.

[Angeline and Pollack, 1994] Peter J. Angeline and Jordan B. Pollack, "Coevolving High Level Representations," In *The Proceedings of Artificial Life III (to appear)*, 1994.

[Antonisse, 1989] Jim Antonisse, "A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint," In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc, 1989.

[Beckenbach and Bellman, 1965] Edwin F. Beckenbach and Richard Bellman, *Inequalities*, Springer Verlag, 1965.

[Davidor, 1989] Yuval Davidor, "Epistasis Variance - Suitability of a Representation to Genetic Algorithms," Department of Applied Mathematics and Computer Science CS89-25, The Weizmann Institute of Science, 1989.

[DeJong, 1988] Kenneth DeJong, "Learning with Genetic Algorithms: An Overview," *Machine Learning*, 3(2/3):121–138, 1988.

[Deugo and Oppacher, 1990] Dwight Deugo and Franz Oppacher, "Explicitely Schema Based Genetic Algorithms," In *Proceedings of the Canadian Conference on Artificial Intelligence*, 1990.

[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.

[Hinton and Nowlan, 1987] Geoffrey E. Hinton and Steven J. Nowlan, "How Learning Can Guide Evolution," *Complex Systems*, pages 495–502, 1987.

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

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

[Li and Vitanyi, 1993] Ming Li and Paul Vitanyi, *An Introduction to Kolmogorov Complexity and its Applications*, Springer-Verlag, 1993.

[Michalski, 1983] Ryszard S. Michalski, "A Theory and Methodology of Inductive Learning," In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*, pages 83–129. Morgan Kaufmann, 1983.

[Quinlan and Rivest, 1989] J. Ross Quinlan and Ronald L. Rivest, "Inferring Decision Trees Using the Minimum Description Length Principle," *Information and Computation*, pages 227–248, 1989.

[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.

[Tackett, 1993] Walter Alden Tackett, "Genetic Programming for Feature Discovery and Image Discrimination," In *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc, 1993.

[Winston, 1992] Patrick Henry Winston, *Artificial Intelligence, 3rd edition*, Addison-Wesley Pub. Co., 1992.

A Minimum Description Length Principle for Hierarchical Organizations of the Function Set

We can apply the minimum description length principle to interpret how good a hierarchical representation is. We take the simpler case of binary trees, although similar conclusions can be obtained for trees with bounded degree nodes. A MDL encoding of a GP individual is given by a minimum encoding of its program tree T and its misses $Misses(T)$. A tree T of size n can be encoded reasonably as an array of length n. Each element of the array encodes the label of the nodes and the indices of its two children. Labels are taken from a set of size A, thus the array can be encoded using $n \cdot (\log A + 2 \cdot \log n)$ bits. Furthermore, for a fixed total number of fitness cases k, the number of misses of a program tree T can be codified with approximately $Misses(T) \cdot \log k$ bits. A measure of this codification is the *descriptional complexity* of T, $DC(T)$:

$$DC(T) = n \cdot (\log A + 2 \cdot \log n) + Misses(T) \cdot \log k \tag{4}$$

We want to search for trees with no misses, and a minimal descriptional complexity. In the general case, we use the additive property of DC. Let a fixed individual program be represented by a tree F_0 with a size $n_0 > 0$ which calls (directly or indirectly) automatically discovered functions $F_1, F_2, ..., F_m$ having sizes $n_1, n_2, ..., n_m$ with $n_i \geq 0$ for all i. A depends on the function/terminal set size and on m (an upper bound on the number of functions that could have been discovered):

$$A = |\mathcal{F}| + |\mathcal{T}| + m \tag{5}$$

The structural complexity n of T is computed by formula 2. We also consider that the number of misses for the functions is 0. Thus the encoding of T is:

$$DC(T) = \sum_{0 \leq j \leq m} (2 \cdot n_j \cdot \log n_j + n_j \cdot \log A) + Misses(T) \cdot \log k \tag{6}$$

For a fixed total structural complexity $SC(T) = n = \sum_{0 \leq j \leq m} n_j$ and a fixed number of misses, the sum in $DC(T)$ has a maximum if $n_0 = n$ and all $n_j = 0$ and a minimum if all trees have equal size. We can prove this by applying Jensen's inequality [Beckenbach and Bellman, 1965] to the convex function $(\log x + \log A)$.

The case $n_0 = n$ and $n_i = 0$ for all $1 \leq i \leq m$ represents the situation when no (automatically discovered) subfunctions are used. $DC(T)$ is maximum in this case. We obtain a minimum description when a maximum number of automatically discovered functions are used, and the sizes of the trees representing the functions are all equal.