subsystems are built out of stable components, then 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 [10].

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. A clear definition of building blocks is provided and a general method to discover such components is given based on an extended representation that keeps track of global statistical measures of the substructures the individuals are made of.

Stable blocks become bricks at the basis of future evolution. 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. It clearly exemplifies the effect learning plays in the evolutionary process ([5]). Extraordinary improvements of the effort needed to find a solution have been observed in the examples presented.

This work distinguishes from other approaches to understanding what building blocks are in several major respects. As opposed to [2] where the functional role of a module is identified and the evolution of the representation language is suggested, we select and use blocks in a totally different way. Selection is based on estimated block usefulness determined by an analysis of the GP trace, while use is based on the adaptation of the function set and population enrichment. No need for more genetic operators appears.

The difference from [7], [8] consists in the way functions are discovered (based on "social" usefulness criteria) and in that functions discovered have a global scope. Functions analogous to ADFs but with a clear semantics are created for the benefit of the entire population. There are no syntactic constrains imposed on the functions created.

We have not experimented with deletion of functions previously added to the function set, but our remarks and [2] argue for using the appearance frequency of these functions to decide upon their usefulness. The use of several block fitness functions for evaluating different subproblems or of heuristic block fitness functions represent future directions of work.

The evolution process in the extended paradigm is clearly divided into epochs which are defined by great discoveries (building blocks) that enable multiple step jumps in the fitness of the best-of-generation individual. For problems with a symmetric structure functions solving subproblems are discovered in a hierarchical manner and successfully used to rapidly generate solutions to the problem. Surprisingly, many solutions are generated in one run, proving the effectiveness of the method.

Fast, stepwise evolution influenced by the dynamic function set adaptation appears remarkable. All fit building blocks discovered for a problem have a clear structure and meaning.

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

[1] Peter J. Angeline. Genetic programming and emergent intelligence. In Kim Kinnear, editor, *Advances in Genetic Programming*. MIT Press, 1994.

[2] Peter J. Angeline and Jordan B. Pollack. Coevolving high level representations. In *The Proceedings of Artificial Life III (to appear)*, 1994.

[3] Kenneth DeJong. Learning with genetic algorithms: An overview. *Machine Learning*, 3(2/3):121–138, 1988.

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

[5] Geoffrey E. Hinton and Steven J. Nowlan. How learning can guide evolution. *Complex Systems*, pages 495–502, 1987.

[6] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, second edition, 1992.

[7] John R. Koza. *Genetic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

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

[9] Justinian P. Rosca and Dana H. Ballard. Genetic programming with adaptive representations. Technical Report in preparation, University of Rochester, Computer Science Department, 1994.

[10] 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.

[11] 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.

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

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 5 presents the final call-graph induced in solving one instance of the problem. Different levels in the call graph correspond to different 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.

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.

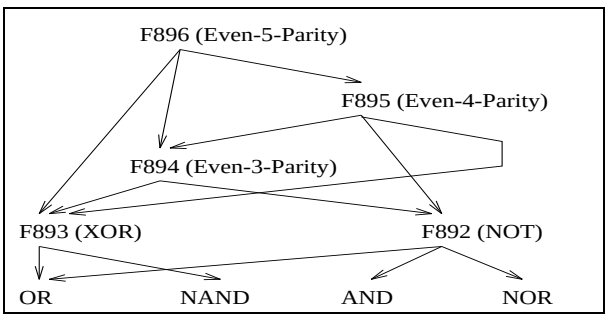

Figure 5: Call graph for the extended function set in the EVEN-5-PARITY example

Method	EVEN-3		EVEN-4		EVEN-5		EVEN-8	
	gen	size	gen	size	gen	size	gen	size
AR-GP	2	17	3	15	5	10	12	17
GP#	5	45	23	113	50	300		
ADF-GP#	3	48	10	60	28	157	24	186

Table 2: Comparison of results: simple GP and AR-GP vs. results reported in [8], marked (#). The results for EVEN-8 are based on one run.

Most importantly to note is 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 to the problem without using the adaptive framework. The computational effort increases considerably as reported in [7]. The adaptive representation method discovers solutions to simpler problems, enabling a problem decomposition into subproblems. The scalability of the problem improves significantly. Row 1 from table 2 shows the number of generations to find a solution with 99% probability in one run and the average size of solution trees obtained for ten runs of AR-GP on EVEN-PARITY problems with $M = 4000$. Rows 2 and 3 present some comparative results taken from [8] for sample runs of GP with the similar parameter values (but $M = 16000$.)

VII. Conclusions

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*. If

ful. This represents a good validation of our idea. 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.)

Taking advice on what blocks seem promising is useful for complex problems. Nonetheless the method does not necessarily rely on domain knowledge. Note that we do not specify what the possible subgoals or subfunctions are. Subgoals correspond to instantiations of building blocks and are considered in an implicit way. The extended paradigm does not impose any ordering of subgoals either. A solution to the problem will point out a decomposition into subproblems, as in [7].

VI. Experimental results

The EVEN-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. It 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.

The results reported herein are based on runs with the AR GP kernel (built on top of the standard GP LISP code described in [7]) with the same set of parameters (among which $M = 4000, G = 50$). 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. 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.

An example of the evolution of most frequent blocks when disabling the adaptation of the function set, is presented in figures 3 and 4. None of the most frequent blocks (MFB) appeared in a final solution. One of them which was the most frequent block in generations 12-14, MFB5, is: (OR (NAND (OR D2 D0) D2) (OR (AND D1 D0) (NOR D1 D0))). It appeared 31 times and had 2 hits. 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.

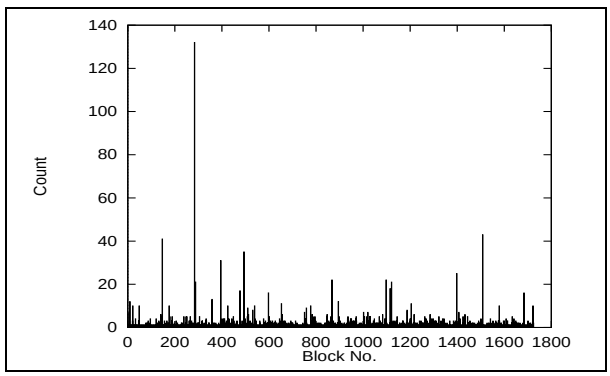

Figure 3: Final distribution of block frequencies in a run of EVEN-3-PARITY

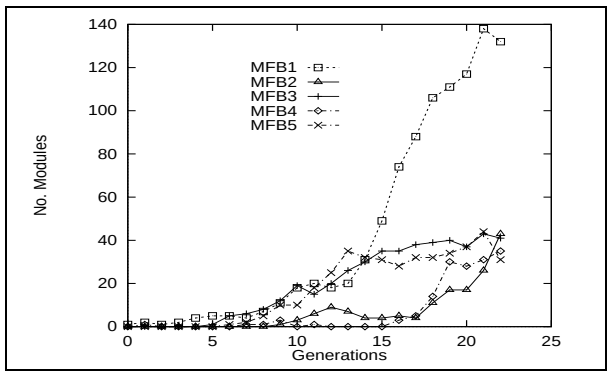

Figure 4: Evolution of most frequent blocks in the EVEN-3-PARITY example

We solved all PARITY problems up to order 11 on a SUN SPARCstation 10 by adapting the representation based on fit blocks. A complete example of a run for EVEN-5-PARITY is given in table 1. With AR all runs of EVEN-5-PARITY need at most 5 generations.

In generation 0 two functions are discovered: F892 (NOT or even parity on 1 bit) and F893 (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

of parsimony pressure and is useful for eliminating redundant or totally unuseful genetic material.

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 3. It shows that most of the blocks, even the ones that are part of final solutions, have very low frequency counts.

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. 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. A high fit block generated in the initial population spreads in the population very easily. Once the population evolves on a right path, good blocks appear among the frequent blocks. Frequent blocks themselves do not necessarily help driving evolution on an ascending path, if used to generate new functions.

V. 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. The new functions are created similarly to modules in ([2]), but will be used differently. Note that the initial members of the function/terminal set are the initial problem building blocks.

In order to enable the future use of newly discovered functions in building new individuals we split the evolution process into *epochs*. An epoch is a sequence of consecutive generations in which no candidate building blocks are discovered. Once at least one such block is discovered in a generation, the population suffers 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 GP paradigm 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 is called a *super-tree* in [9]).

The information used in discovering useful blocks may be either global, domain independent information extracted from problem traces or domain dependent knowledge.

The skeleton of the GP algorithm that incorporates the AR component is presented in figure 1. Figure 2 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 2: GP extension: Adapting Representation

Extending the function set based on fit blocks discovered represents our choice for the current experiments although a combined fit - frequent blocks method can be applied.

It is interesting to note that a statistical analysis of the frequent blocks after the function set is 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 use-

GP skeleton

Define problem: $\mathcal{T}, \mathcal{F}_0, \mathcal{C}, \mathcal{E}$ (set of fitness cases), $(^*)\mathcal{C}_{BB}$ (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) Evaluatey 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

In Genetic Algorithms (GA), the schemata theorem ([6], [4]) summarizes the effect of fitness proportionate reproduction, crossover and mutation, by using strings instead of trees as representation primitives. The fitness of a template string (schemata) is defined as the average over all individuals in the population matching that template. Individuals that match schemata with fitness values above population average will be selected in an (exponentially) increasing number of samples in the next generation. Moreover, schemata play the role of *building blocks*. The creation of the next generation population is based on the heuristic rule that useful discovered building blocks will prove to be useful in similar contexts.

It is not clear what building blocks are in GP. [1] outlines the difficulty created by the complicated many-to-many mapping between genotypic and phenotypic features. An individual in GP is a piece of "code" in a base language. The interactions between the different fragments of code may be very entangled. Nonetheless, we argue in favor of the natural formation of genotypic features, or *building blocks* in GP, and on the possibility of identifying them.

In GP, a 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 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 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), or by the more powerful set of primitives (GP).

Blocks are *entire* subtrees of a given maximum height from population individuals. Although the height of the population trees is bounded, the population is very rich in such blocks. In a full tree of height h there is an exponential number ($O(2^{h-g})$) of subtrees of height $g < h$. We look for building blocks in the set of possible blocks. The 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 complex blocks.

We point out the two challenging goals we aim at: 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 (more generally, control of the problem representation) with these new functions created on-the-fly from promising blocks, possibly candidate building blocks. We will call such a representation an *adaptive representation*, as it is automatically changed while the GP program searches the space of admissible solutions.

IV. 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. In [2], the worth of a module is the number of calls to it in subsequent generations. It is used mainly in order to decide module viability (when to discard modules), as their number increases tremendously. In [11] frequent subtrees in an individual correspond to synthesized features which comprise "building blocks".

Surprisingly, frequent blocks are not necessarily good building blocks. The schemata theorem tells us that a good building block will spread rapidly in the population, and this determines a high frequency count for the block. For example, in the even parity problem, if we augment the function set with the exclusive OR function (XOR), this function (a building block) will soon become dominant in fit program trees. The problem is that the *reciprocal* of the 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 [1]). A slight improvement is noticed in the case a program size component is added to the value computed by the fitness function. This component represents a kind

Learning by Adapting Representations in Genetic Programming

Justinian P. Rosca, Dana H. Ballard
Computer Science Department
University of Rochester
Rochester NY 14627

***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. Genetic Programming (GP) has been effective in learning via interaction but so far there have not been any significant tests to show that GP can take advantage of its own search traces. This paper demonstrates how an analysis of the evolution trace enables the genetic search to discover useful genetic material and to use it in order to accelerate the search process. The key idea is that of genetic material discovery which enables a restructuring of the search space so that solutions can be much more easily found.**

I. Introduction

The defining characteristic of a learning system is considered to be the capability of adaptation under the implicit goal of performance task improvement [3]. The learning process subsumes structural changes, discovery and use of new concepts.

Our focus is the Genetic Programming paradigm, as defined in [7]. 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 recent work has tried to define larger components. Current extensions to GP point out the possibility of evolution of new functions (*automatic function definition* or ADF in [7]) or emergence of viable *modules* (in [2]). 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 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.

The examples given in the paper to illustrate concepts introduced and experiments performed will be taken from the problem of learning the EVEN-PARITY function (see [8] for a complete description).

II. Short overview of the GP paradigm

A concise description of the paradigm, its main parameters and discussions of some advanced topics are presented in [7]. Here we just overview some basic concepts and notations used throughout this paper.

The main steps in applying GP to evolve a population of programs (also called individuals or program trees) 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 fitness (cost) measures $\mathcal{C}$; (3) define the control parameters (population size M, maximum number of generations G, etc.); (4) define a termination criterion (new generations of individuals are created until this criterion is satisfied).

The structure of an extended GP 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.

III. Building Blocks in GP

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. The question is then how we can improve the performance of GP for very sparse solution spaces. 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 [7], [8].