An Experimental Perspective on Genetic Programming

Una-May O'Reilly and Franz Oppacher
Intelligent Systems Research Group,School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, CANADA

Abstract
Genetic Programming (GP) has recently been introduced by John R. Koza [5] as a method for genetically breeding populations of computer programs to solve problems. We believe GP to constitute a significant extension of the Genetic Algorithm (GA) research paradigm primarily because it generalizes the genetic search techniques: instead of looking for a solution to a specific instance of a problem, GP attempts to evolve a program capable of computing the solutions for any instance of the problem. We have implemented a genetic programming environment, GP*, that is capable of duplicating Koza's experiments. In this paper we describe a specific GP experiment on the evolution of programs to sort vectors, and discuss the issues that must be addressed in any application of GP: the design of fitness functions and test suites, and the selection of program terminals and functions. Our observations point to several previously unnoticed shortcomings of the GP approach. We hypothesize that these shortcomings are due to the fact that GP only uses a hierarchical representation but does not construct its solutions in an explicitly hierarchical manner.

1. Introduction

Genetic Programming (GP) has recently been introduced by John R. Koza [5, 7] as a method for genetically breeding populations of computer programs to solve problems. It is essentially an extension of a Genetic Algorithm (GA) [4, 1, 3] in which the individuals are programs. Koza has shown convincingly that the GP paradigm is general and provides a single, unified approach to many seemingly different problems in an astonishing variety of areas [5–10].

We believe GP to constitute a significant evolutionary extension and enrichment of the GA research paradigm itself; it is not just another application of GA methodology to a new domain. One reason is that GP manipulates variable length solutions which are not encoded as bit strings. String based representations are typically restricted to an apriori selected length. This in advance limits the number of internal states of the system and the complexity of what the system can learn. In GP the maximum length of a solution must be apriori defined to constrain the search space but all solutions up to and including this maximum are considered.

GP makes genetic search techniques more powerful by generalizing them: instead of looking for a single point solution to a specific instance of a problem, GP attempts to evolve a program capable of computing the solutions for any instance of a problem. While there are other non-string based, variable length GA approaches [2, 11] which evolve programs for solving problems, and not solutions to individual problem instances, these programs are limited to simple if-then rules.

The paper proceeds as follows: Section 2 briefly reviews the GP approach. In Section 3 we explain some of the choices that must be made when designing fitness criteria. Section 4 illustrates how the selection of an initial set of functions and variables influences the search space of the algorithm and the ultimate generality of the solution. Section 5 raises questions - suggested by observations made in Section 4 - concerning the precise interpretation of the claim that the results produced by GP are inherently hierarchical [5]. We conclude that GP in its current form is hierarchical only with respect to its representation and not with respect to its process of constructing solutions. This limits the ability of GP to evolve complex programs from simple, general functions, and makes the algorithm strongly dependent on initial human design decisions.

2. A Review of Genetic Programming

GP executes the following two steps:

1. Generate an initial population of computer programs which are random compositions of the functions and terminals of the problem.
2. Repeat:
   a) Execute each program in the population against a set of test cases and assign it a fitness value according to how well it solves the problem.
   b) Create a new population of computer programs by using
      - fitness proportional selection for choosing parents, and
      - genetic crossover for combining randomly chosen parts of two parent programs.

   Until: Termination Criterion is Satisfied.

The major parameters of a GP algorithm are the population size and the number of generations to be run. The termination criterion is satisfied when either a perfect solution has been found or a maximum number of generations has been reached. Other parameters are:

- How many parents are directly propagated by fitness proportional reproduction into the next generation and how many times crossover is performed. Typical values are 10% and 90%, respectively.
- The distribution of selection of crossover points. Typically this is set to a 90% probability of selecting interior nodes (uniformly) and a 10% probability of selecting a leaf node.
- The maximum depth of a program (in tree height) created by crossover and the maximum depth of random programs in the initial population. Typical values are 15 and 4, respectively. For complete details, see [7].

We are using GP* for ongoing experimentation. Here we describe our attempts to derive a program that could sort any linear array.

3. Designing the Fitness Criterion

A crucial first step for evolving any program under the GP methodology is to design a fitness function. In a GA each individual is tested once, e.g., in function optimization the bit string is decoded and the function evaluated. In GP, the fitness function must measure a program's behavior on many test cases. Thus a range of test cases is required and their distribution over the range of desired behaviors must be calculated to enforce an acceptable degree of program generality. For sorting, we clearly want a sort program which can sort an array of any size (given *ArraySize* as a constant).

The Block Stacking experiment [7] consisted of a complete random sample containing 166 test cases. It was significantly shortened by consolidating various inherently similar initial configurations and by sampling. In the GP* sort experiment there are 48 different vectors of 5 different sizes. For the smaller sizes every permutation of elements is a test case. Larger sizes are sampled. As yet, it is unclear how many test cases are sufficient to evolve a general sort program: there is a trade-off between finding the one solution exactly fitted to a particular small test suite and spending a lot of time evaluating fitnesses (which is the most computationally expensive aspect of the entire GP methodology).

One subtle aspect of the choice of a test suite is the 'incremental encouragement' it can exert. For example, in the Block Stacking experiment, solving the test case which puts the only block on the table onto the (already ordered) stack is worth as much as solving the test case that involves removing all blocks from the stack and restacking them correctly. Thus, the reward for doing a little task correctly is equal to that of doing a large task. In this manner, solving the first test case could create a 'building block' program for solving harder ones. This suggests that test cases whose solutions are likely to be reused should be given relatively greater weights, and that the eventual solution could be 'coached' along by progressive incentives arising from an interplay between the fitness function and a cleverly chosen test suite. It does seem, however, that while this designer-controlled incremental encouragement may influence the convergence toward a solution it is not as crucial as function selection which we discuss in section 4 below.

In the Block Stacking experiment the distance for a test case was either 1 (blocks incorrectly stacked) or 0 (blocks correctly stacked). One may wonder:

> Under what circumstances should the distance function be sensitive to partially correct program behavior? A program that does something right should be given a proportional chance to contribute this expertise to the next generation. Is such a policy experimentally and theoretically sound, and applicable to all problems?

During early generations, a binary distance function, i.e., one restricted to values of 0 and 1, seems appropriate because programs are small and each relatively fit individual is judged fit in virtue of its entire structure. As generations progress these early fit individuals become the building blocks which are exploited and combined into larger structures. Under a distance function that is sensitive to partial program correctness, a small subprogram will usually be responsible for a program's overall fitness. We do not know whether crossover during early generations favours either approach. In early generations the chance of a complete program being chosen as a crossover point is relatively good because programs are small, but, since there are more subtrees than complete trees, disruption of complete trees is very likely. This fact would seem to favour a sensitive distance function. However, a program which has only one subprogram that earns its fitness can still contribute its other useless subtrees during crossover. Moreover, it is possible that the choice of either a binary or a sensitive distance function is rendered inconsequential by the scale of disruption caused by crossover.

Our experiments reward programs which demonstrate partially correct behavior: every element not in the correct position is scored as 1 point, so that the distance of a program ranges between 0 and the size of the array. To date, we have studied aspects of the overall behavior of the algorithm: histogram data of fitness distribution, the convergence progress, and relations between program size (tree height) and the distribution of fitness and convergence properties. At this point in our investigation we can not distinguish behavior due to the distance function from behavior due to the choice of available functions and terminals. In later experiments we will employ a discrete distance function for comparison.

4. Selecting the Functions and Variables

The design of a GP experiment also necessitates choosing the function set F and the terminal set T (variables and constants). The process is similar to being told to pack a knapsack to travel to a known destination, without being told the means of conveyance or the mode of the voyage. The sets F and T used in the GP* Sort experiment are shown in Table 1.

Table 1
F and T in the GP* Sort Experiment

Functions	Terminals
(*minus1* var)	*j*
(*aref*Array** index)	*i*
(*swap*Array** index1 index2)	1
(*if-lt* index1 index2 true-form)	*ArraySize*
(*do-until-Up-with-*j** startValue endValue true-form)	
(*do-until-Down-with-*i** startValue endValue true-form)	

An examination of well-known sorting algorithms suggested the need for the following constructs: looping, a conditional function, a swap function, and array element referencing. This simple identification does not make further choices clearer concerning the extent to which the 'freedom' of a function should be limited:

- A (relatively) specialized function has one or more of its arguments absorbed into the function name, i.e., has its arguments set to fixed values.

- A (relatively) generalized function has its behavior determined to a large extent by the type and value of its explicit, GP*-manipulatable arguments.

Specialized functions search a smaller space, and are thus more efficient, but limit the shape of the eventual solution in advance (at worst, they may preclude finding a solution). Generalized functions do not restrict the final solution in any way, but, since they have more arguments, they recursively define a larger search space (at worst, they may fail to find a solution in an acceptable time). Consider the simple examples in Table 2. Example A returns the value of a variable decremented by 1, example B accesses an array element, and example C exchanges two values. Recall that in GP the search space continually

Table 2
Examples of Functions

expands as programs evolve! Thus, the question arises:

> Just how tightly can the search space be constrained by using specialized functions before the claim that the evolved form of solution has been biased in advance of the search becomes justified?

We chose _minus1_, _aref*Array*_ and _swap*Array*_ (see Table 2) and added *array* and *arraySize* to the terminal set with reasonable confidence that the generality or the structure of the eventual solution would not be unduly influenced in advance. We did not, however, add the integers in the index range of the array to the terminal set because this would have greatly increased the search space and, more importantly, would have facilitated a highly specialized solution which fits the test suite precisely.

The design of a loop function raised similar issues. The function (_do-until_ loop-test true-form) iterates until loop-test is false but it relies upon the true-form performing a side-effect which changes some variable in loop-test. It uses no loop variable which seems to make it less suited for array sorting.

The function (_do-until_ loopVar startValue endValue varDelta loop-test true-form) has its behaviour completely dictated by the values of its arguments. loopVar is set to startValue and each time through the loop is changed using varDelta. startValue is compared to endValue using the relation loop-test. If the result is true, true-form is executed. true-form could even be a _progn_ of functions but this creates the further unresolved issue of how to reasonably specify the number of functions within the _progn_.

Any one of these arguments could be absorbed into the function to limit its potential behavior. For example, (_do-until-Up-with-*j*_ startValue endValue true-form) uses the variable *j* as implicit loopVar. The function sets *j* to startValue, and it always increments *j* at the end of each repetition, using an implicit varDelta of 1+. It always tests whether $*j* <= endValue$, with an implicit loop-test of <=, to conditionally execute true-form.

Similarly, the function (*do-until-Down-with-*i** startValue endValue true-form) loops downward using *i* as the loop variable. These last two functions require that *i* and *j* be made variables for the program. Note that the loop variables must be different globals to facilitate nested loops that might sweep up and down an array. A valid question remains: does this choice of up and down loops constitute too strong an advance bias? GP design constantly involves this type of second guessing.

To summarize the function and terminal selection process, one can state that it seems impossible to determine on general principles whether a function is too biased or specialized or whether it is chosen with a minimum of bias at an appropriate level of complexity. Our choice was motivated by an intuitive trade-off between a feasibly small search space and a reasonable chance to achieve unanticipated eventual solutions. The GP methodology relies at various crucial junctures on designer expertise to focus its search space.

5. Is GP Completely Hierarchical?

In the GP* Sort experiment we made a deliberate decision NOT to use the following:

(*if-lt-swap* index1 index2).

This function compares the elements of *array* at index1 and index2 and swaps them if the first is less than the second. By linking comparison and exchange, this function is obviously a desirable building block in either an iterative or a recursive solution. But we wanted to discover it rather than start from it. Since *if-lt-swap* does several things, we feel that it is ‘too cooked‘, and hides too much complexity. Shouldn‘t the GP paradigm be capable of evolving a program which has itself discovered (via its ancestors) the conjunction of comparison and swapping in order to sort?[1] The evolving system should identify two smaller ‘building blocks‘, *compare* and *exchange*, as subprograms, and establish a larger building block by combining the subtrees through selection and crossover. While this is more difficult, it is important from the perspective of hierarchical program design, subprogram re-use and the hierarchical construction of solutions from smaller building blocks.

> Does the implicit parallel search on hyperplanes via fitness proportional selection and crossover supply sufficient power for the GP paradigm to hierarchically form complex programs from general and simple (‘not cooked‘) functions?

In Koza‘s experiments the function and terminal sets are expertly designed - not too biased and succinctly defined. Koza‘s experiments make it clear that starting with ‘higher-level‘ or relatively specialized functions will improve the speed of convergence and may yield a simple and elegant solution but they do not answer the above question. For

[1] In fact, GP should also be able to discover the connection between an array and its reference function using an index. That is, (*aref*Array** index1) is itself a building block which is implicitly included in the *if-lt-swap* function.

example, in the Block Stacking experiment there are two functions *move-table* and *move-stack* both of which use an argument which represents a block; alternatively they could have been generalized into the function *move* which took two arguments: the source (table or stack) and the block. The sensor function *top-correct-block* is quite sophisticated: it dynamically specifies the first element of the list which is the longest list of remaining elements of the list STACK that matches a *cdr* of the goal-list. The complexity of these functions yields encapsulated complex building blocks in the search space. Since the encapsulation of a function protects it from disruption during crossover this choice of complex functions influences the rate of convergence.

One goal of the GP* Sort experiment is to see if the current model is sufficiently powerful to start from very general functions or whether additional processing capability is needed in order to achieve a completely hierarchical formation of programs.

In theory, a GP* Sort should be successful starting with a general loop, and (- var value), (*aref* array index) , (*if-lt* x y true-form), and (*swap* x y).[2]

To simplify our task we do not experiment with this most general function set but we do avoid the specific extreme of *if-lt-swap*. Without loss of generality we use (if-lt index1 index2 true-form) to see if the subprogram containing true-form (*swap *Array** index1 index2) can evolve. We have simply eliminated the need to search for (*if-lt* (*aref*Array** index1) (*aref*Array**index2) true-form). Should the experiment find solutions, this specificity could be further 'undone' and the robustness of GP* retested from an even more general starting point. Unfortunately, this has proved unnecessary. Our experiments to date reveal that the GP methodology as implemented in GP* can not robustly find a solution with this function set.

Our experiments use a population size of 300 and set the termination condition to be 51 generations or discovery of a perfect program. All start with an initial random population where the maximum tree height equals four. The test suite is 48 arrays comprising a total of 198 elements (array sizes range from 2 to 6). Thus, the adjusted fitness of a individual is in the range 1 to 199. A hit is the totally correct sorting of an array. Initially the test suite contains 3 correct hits and 48 elements in the correct position. Node crossover is performed with a probability of 90% and leaf crossover with 10%. Crossover is performed on 135 pairs of parents (90%) and 10% of one generation is directly propagated to the subsequent one.

Our best (but still fairly typical) experiment, experiment A, succeeded in placing no more than 141 elements in the correct position but an (adjusted) fitness in this neighbourhood (61% of the elements placed correctly) is often achieved. In experiment A the number of hits was 21, which is only 43% of the test suite. The average fitness of the population equaled 94.0. Figure 1 is a graph plotting the progress of experiment A. The runs of the experiment on the average placed 96 elements correctly. Runs of 51 generations and of 101 generations have shown that the fitness never improves significantly once it has reached the approximate level of 60%.

We do not believe that scaling is responsible for the premature convergence. Though scaling usually limits diversity in later generations there is sufficient diversity in a GP*

[2]Simplifying further, (*swap* x y) should be replaced with (*setf* x y) and a temporary variable should be made available. Thus a swap function would evolve by the following sub-program: (*setf* tmp x) (*setf* x y) (*setf* y tmp).

Figure 1.

Sort population even among the candidates strong enough for selection. In fact, once the 66% neighbourhood has been reached, we see disparity return to the population. On an animated histogram the ‘undulating slinky movement from left to right‘ [7] halts and a downward wave to the left occurs.

We feel that this premature convergence to a non-optimal solution is due to the fact that successful subprograms, once formed, can not be exploited before they are disrupted or lost in the crowd. We suspect that the GP methodology is not as general and powerful as it may appear from the many successful examples described in the literature. It is most often successful if it uses ‘high level‘ or specialized functions which via their encapsulations retain complex structure and are protected from disruption.

Koza states: We believe that premature convergence (or occasional non-convergence for other reasons) should be viewed as an inherent aspect of GA‘s (rather than as a problem to be cured by trying to alter the fundamental nature of the algorithm). Nature can be the guide here. Premature convergence manifests itself in nature in the form of the niche preemption principle. [7].

We disagree with Koza‘s facile attitude toward premature convergence, and we believe that GP suffers from problems of non-convergence because it is not completely hierarchical. GP is only hierarchical in the minimal sense that it exploits the hierarchical representation of a program as a tree. This simple aspect of hierarchy has a strong influence on crossover and thus on the effectiveness of the GP paradigm and even raises doubts as to its consistency with the standard GA theory of implicit parallel search on hyperplanes.

At any rate, GP's process of putting a solution together is not itself hierarchical. Hierarchical solution construction is a process of top-down and bottom-up, level by level, construction of increasingly more efficient pieces of a solution. In a downward direction, decomposition yields primitives which provide more flexibility in expressing a solution. In an upward direction, combination of primitive functions leads to compound functions at a higher level of abstraction. The functions at this level can themselves be considered as primitives and can be combined in turn to form a yet higher level of abstraction. In this way, a hierarchy of function specialization and generalization is built. Hierarchical evolution, as we define it, is guided by these principles and the principles of Darwinian evolution - survival of the fittest and genetic variation.

We believe that extending GP to foster hierarchical evolution is desirable not only because it will lead to improved algorithms but also because it reflects more closely numerous examples of hierarchical evolution in nature. One motivation to form hierarchical structures and to process hierarchically is to achieve optimal needs efficiency in terms of speed. There is an overriding drive to be flexible: i.e. to match the granularity of the semantics to the significant aspects of the problem so that processing is fast. This concurs with the general goal of genetic search and genetic programming.

Thus, the new question is:

How can hierarchical evolution be incorporated into the GP paradigm?

It is implausible to suppose that nature constructs hierarchies at random. Our ongoing work employs a building block operator SPECIALIZE that uses a better subtree selection technique and which, importantly, is used at judicious *system determined times*. This SPECIALIZE operator must be used in conjunction with a DECOMPOSE operator which generalizes the set of functions by disassembling a function so that its components may mix and reenter into other combinations. The SPECIALIZE operator should be applied to a strong subpopulation and the DECOMPOSE operator should be applied to a weak sub-population. SPECIALIZE could choose which subtrees to encapsulate or consolidate by detecting similarities in behavior and program composition in the subpopulation. DECOMPOSE would analogously detect similarities which it would break up by replacing a node with a subtree. Decomposition is exploratory in nature.

Koza has proposed a 'building block operator' [7]: in a program, chosen according to fitness proportional selection, an internal node is randomly selected. The subtree rooted at this node is 'glued' into one function which replaces the subtree as a leaf. Koza's building block operator does accomplish specialization of a function and this prevents a subprogram from being disrupted. However, it selects the subprogram for specialization (within a strong individual) at random.

For example, the function (*aref* array index) could be consolidated as (*aref*Array** index) if the SPECIALIZE operator found that many strong individuals used *aref* with the argument array set to *array* but with different index values. As an example of decomposition, the function (*minus1* var) may often be present in a weak subpopulation. It would then be decomposed into (*minus* var value).

These operators should not be applied every generation because hierarchies require sound foundations. This may be achieved by waiting until a group of stronger individuals emerge to exploit specialization and, when weaker groups frequently recur, generalizations may be tried. Decisions as to when to generalize or specialize can be based on available histogram data.

6. Conclusion

To summarize, through a reconstruction of Koza's Genetic Programming Paradigm in GP*, we have illustrated the subtle aspects of test suite design. We have observed how function selection constrains the search space and strongly biases the structure of the eventual solution. We proposed the question: Does the implicit parallel search on hyperplanes via fitness proportional selection and hierarchical crossover supply sufficient power for the GP paradigm to hierarchically form complex programs from general simple functions? And we know the immediate answer to be negative. However, the evidence of hierarchical processes in nature and the hierarchical structure of planning and program design lead us to believe that extension of GP in this direction is feasible and worthwhile. We have sketched how such an extension might be achieved.

7. References

1 De Jong, K. A., On Using Genetic Algorithms to Search Program Spaces. Proc. 2nd Int'l Conf. on Genetic Algorithms, Lawrence Erlbaum Associates, 1987.

2 Grefenstette, J. J. A System for Learning Control Strategies with Genetic Algorithms. Proc. of 3rd Int'l Conf. on Genetic Algorithms, Morgan Kaufman, San Mateo, CA.,1989.

3 Goldberg, D. E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading MA., 1989.

4 Holland, John H. Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI., 1975.

5 Koza, J.R., Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. Proc. of the 11th Joint Int'l Conf. on Artificial Intelligence, Morgan Kaufman, San Mateo, CA., 1989

6 Koza, J.R., and Keane, Martin A. Cart Centering and Broom Balancing by Genetically Breeding Populations of Control Strategy Programs. Proc. of Int'l Joint Conf. on Neural Networks, 1990.

7 Koza, J.R., Genetic Programming: A Paradigm for Genetically Breeding Populations of Programs, Stanford University TR STAN-CS-90-1314, 1990.

8 Koza, J.R., Evolution and Co-Evolutions of Computer Programs to Control Independently-Acting Agents, in From Animals to Animats: Proc. of the 1st Int'l Conf. on Simulations of Adaptive Behavior, MIT Press, Cambridge, 1991.

9 Koza, J.R., Concept Formation and Decision Tree Inductions Using the Genetic Programming Paradigm, Proc. of Parallel Problem Solving from Nature. Springer Verlag 1991.

10 Koza, J.R., Genetic Evolution and Co-Evolutions of Computer Programs, in Artificial Life II, Editors: Langton, Taylor, Doyne Farmer, Rasmussen. (p 603-629), Addison Wesley, Redwood City, 1991.

11 Schaffer, J.D. Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms. Doctoral Dissertation, Dept. of Electrical and Biomedical Engineering, Vanderbilt University, Nashville, 1984.