AppGP: An alternative structural representation for GP

Nicholas Freitag McPhee
Nicholas J. Hopper
Division of Science and Mathematics
University of Minnesota, Morris
Morris, MN USA-56267
(320) 589-6300
{mcphee, hoppernj}@mrs.umn.edu
http://www.mrs.umn.edu/{~mcphee, ~hoppernj}

Abstract- It has been shown that Standard Genetic Programming using standard subtree crossover is prone to a form of structural convergence which makes it extremely difficult to make changes near the root, occasionally causing runs to become trapped in local maxima. Based on these structural limitations we propose a different tree representation, AppGP, which we hope will avoid this problem in some cases. In this paper, we describe this representation, and compare its performance to the performance of Standard GP on a suite of test problems. We find that on all of the test problems, AppGP does no worse than Standard GP, and in several it does considerably better, suggesting that the representation warrants further study.

1 Introduction

Several researchers have shown that standard Genetic Programming (GP) with the basic crossover operator [Koza 1992], is prone to structural convergence in the nodes near the tree root [Gathercole and Ross 1996, Poli and Langdon 1998, Langdon and Poli 1997, McPhee *et al.* 1998*a*, McPhee and Hopper 1999]. Unfortunately, once a run has converged on such a rooted structure, it is extremely difficult to change (via crossover) an "incorrect" node near the root without also changing all the structure below that node. Thus problems that are highly sensitive to the choice of operators near the root (e.g., Gathercole's MAX problem [Gathercole and Ross 1996]) often get "stuck" with "incorrect" choices near the root which are extremely difficult to correct.

In a previous paper [McPhee *et al.* 1998*a*] we compared Standard GP [Koza 1992] (which we'll call SGP) and a variant of GP using the Hindley-Milner type system (HMGP) on a suite of five essentially typeless problems. In these experiments we found that HMGP outperformed SGP on three of the five problems, and essentially broke even on the other two. In the analysis, we speculated that these results were in significant part a result of structural differences between the two representations. We then proposed a simpler representation, which we called AppGP, that we thought might have many of the same structural benefits as HMGP, without requiring a complex type checker and the associated computational overhead.

In this paper we report on a series of experiments comparing the performance of AppGP to SGP on the same suite of five test problems. Our results with AppGP are similar to those with HMGP, with AppGP outperforming SGP on three of the five problems (although there is a question of statistical significance on one), with no appreciable difference on the other two. This suggests that the AppGP representation, which is simple to implement and adds negligible computational overhead, is no worse than SGP on the small test suite, while being better in several cases. While not conclusive, these results, combined with AppGP's ease of implementation, do suggest that the AppGP representation deserves further consideration as an alternative to SGP.

2 Structural and algorithmic differences between SGP and AppGP

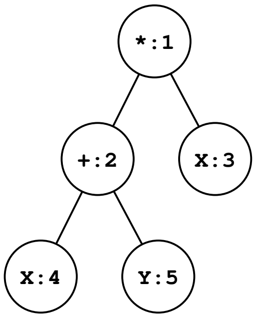
AppGP involves several changes to the basic SGP process, both in the representation of trees and in the algorithms used to manipulate those trees. In this section, we outline the key structural and algorithmic differences necessary to perform GP with the AppGP representation.

2.1 Structural differences

The representation of expression trees in the AppGP system differs in significant ways from the representation of expression trees in SGP. Consider, for example, the expression `(* (+ x y) x)`. The upper tree in Figure 1 shows the standard tree representation of this expression in most GP systems (that is, the SGP representation). Table 1 lists the (sub)expressions represented by each of the nodes, i.e., the expressions available to operators like crossover. Note that the types of all the (sub)expressions are the same, i.e., the closure property holds [Koza 1992].

In the AppGP representation, all non-terminal nodes are application (APP) nodes; these nodes differ in arity according to what function they apply. They always have as their left-most child a function terminal, and the remaining children represent the arguments to the function. The bottom tree in Figure 1 shows the same expression, `(* (+ x y) x)`, represented in AppGP, and Table 2 lists the (sub)expressions represented by each of the nodes in the AppGP representation, i.e., the expressions available to operators like crossover.

One major difference between these two representations is that the AppGP representation has more nodes than the



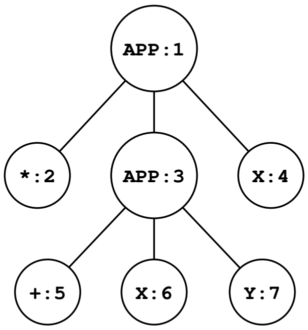


Figure 1: The SGP(top) and AppGP(bottom) representations of `(* (+ x y) x)`. The nodes have been numbered for later reference.

SGP representation, providing more potential points for the application of recombination operators. Unlike the SGP representation, however, AppGP nodes do not satisfy the closure property, with (sub)expressions falling into one of two different classes: Functions and non-functions. The set of non-function subtrees satisfies the closure property, but function leaves like the $+$ and $*$ operators can only be traded for functions having the same arity.

Thus the AppGP representation contains nodes that correspond directly to all of the nodes in the SGP representation (see Table 2), plus several nodes with no direct counterpart in the SGP representation. In particular, in the AppGP representation functions have been moved from internal nodes to leaf nodes. This is potentially significant because it now allows crossover to change an operator without affecting the arguments, a difficult change for crossover in SGP, especially for operators at or near the root of the expression tree. As an example, in the AppGP representation crossover could change the root operator ('*' in our example) to a different operator having a compatible type (such as '+') much more easily than

Node #	Expression
1	`(* (+ x y) x)`
2	`(+ x y)`
3	`x`
4	`x`
5	`y`

Table 1: The (sub)expressions of the SGP expression represented in Figure 1. The Node Numbers refer to the numeric labels in the upper tree in Figure 1.

Node #	Expression	SGP node
1	`(* (+ x y) x)`	1
2	`*`	
3	`(+ x y)`	2
4	`x`	3
5	`+`	
6	`x`	4
7	`y`	5

Table 2: The (sub)expressions of the AppGP expression represented in Figure 1. The Node Numbers refer to the numeric labels in the lower tree in Figure 1. Nodes that correspond to nodes in the untyped representation are listed in "SGP node".

in the SGP representation.

In problems that are highly sensitive to the choice of operators at or near the root, such as Gathercole's MAX problem [Gathercole and Ross 1996], it seems plausible that the AppGP representation might perform quite differently from the SGP representation. Note, however, that it's not obvious that the AppGP representation will in fact be preferable, since the increase in the number of nodes can potentially increase the size of the search space. In SGP, for example, there are 25 possible crossovers of our example expression with itself. In AppGP, there are 29 possible crossovers if we require that the swapped sub-expressions have the same type (25 crossovers of real-valued subtrees and 4 crossovers of real-valued functions.)

2.2 Algorithmic changes for AppGP

The structural changes outlined in Section 2.1 necessitate some changes to the basic creation and crossover algorithms of GP. These differences are outlined here.

2.2.1 Generation of individuals in AppGP

The changes to the creation algorithms described in [Koza 1992] are straightforward. Whenever the SGP representation chooses a terminal, the AppGP algorithm chooses a non-function terminal. When the SGP algorithm chooses a non-terminal, the AppGP algorithm generates an APP node and chooses a function for the leftmost child. The arity of the chosen function then determines the number of remaining children to generate.

2.2.2 Crossover on AppGP individuals

As well as affecting the creation of new individuals, the AppGP representation requires changes to the recombination operators to ensure that they produce semantically valid trees. In the experiments reported here we only use the crossover operator, so we will only discuss changes to the crossover algorithm. Similar changes obviously have to be made to implement other operators such as mutation.

As in most GP systems, crossover acts on two parents P and Q and generates a child C. The process begins by choosing a subtree A from P; in these experiments we restrict the choice to a non-leaf 90% of the time to reduce leaf swapping.

Now, instead of choosing any subtree B from the other parent Q, we need to restrict ourselves to subtrees that have the same type as A (either a value, or function of the same arity). We again restrict the choice of B to be a non-leaf 90% of the time, with the exception that when A is a function (and therefore a leaf), B must also be a function (and therefore also a leaf).

3 Experiments

For comparison purposes, we ran several problems using both SGP and AppGP:[1]

- the MAX problem,
- the Artificial Ant problem with the Santa Fe trail,
- 4-bit even parity,
- Symbolic regression with a quartic polynomial as the target function, and
- Sequence induction on a quartic polynomial.

Except where noted, the parameters are those in Table 3, and the function and terminal sets were the same for both SGP and AppGP.[2]

For all but the MAX problem we used lexical fitness, with the tuple components listed in the respective tables. For the non-MAX problems we implemented a parsimony pressure by setting the last element of the fitness tuple to the number of nodes in the individual (with fewer being better). This always gives preference to an individual with better performance on the problem, but selects the more parsimonious among two individuals with equal performance. To discourage early convergence to very small trees, the size was normalized by setting this component to 0 if it was less than a certain threshold arbitrarily determined for each problem; these thresholds are listed in the parameter tables.

Population size	500, steady state
Selection Method	Tournament selection Tournament size t=10
Maximum Generation	50
Generative Method	Ramped half-and-half, max initial depth 4
Maximum created depth	10
Internal crossover rate	90%
Mutation rate	0
Reproduction rate	0
Random # generator	Java 1.1 `java.util.Random`

Table 3: Common parameters for all runs

Function set	{`if-food-ahead`, `begin2`}
Terminal set	{`(move)`, `(turn-left)`, `(turn-right)`}
Fitness measure	(food eaten, size)
Size threshold	50
AppGP runs	50
SGP runs	51

Table 4: Artificial Ant parameters

For the two regression problems (sequence induction and quartic polynomial), we limit the total error accumulated on the set of fitness cases to a maximum of 10^{10}. We also set the total error to that maximum value if some test case yields a NaN value (by, e.g., dividing by 0).

3.1 The MAX problem

The MAX problem was first described in [Gathercole and Ross 1996]. It consists of the set of problems MAX-depth-D-{Function Set}-{Terminal Set} where the depth is restricted to D and the goal is to maximize the tree value. In these experiments, the MAX-depth-7-{*,+}-{0.5} problem was used. The ramped half-and-half generative method was used with the maximum initial depth set to 5. We performed 51 runs using SGP and 50 using AppGP for this problem.

3.2 The Artificial Ant problem

The Artificial Ant problem evolves a program which navigates a near-sighted and motion-constrained artificial ant along an irregular trail of food [Koza 1992]. There are several well-known trails used for this problem; we used the Santa Fe trail, which lies on a 32×32 toroidal grid and contains 89 pieces of food. The ant is constrained to 400 moves, where a move is one of turning left, turning right, or moving straight ahead. For both representations, the available non-terminals were `if-food-ahead` and `begin2`. `if-food-ahead`

---

[1] These runs were performed using the *Sutherland* EC framework. For more information see [McPhee *et al.* 1998*b*] or `http://www.mrs.umn.edu/~mcphee/Research`.

[2] The parameters described here were not tuned for optimal performance on the test problems; they simply provide a convenient reference point when comparing HMGP and SGP.

Population size	1000, steady state
Function set	{and, or, nand, nor}
Terminal set	{a0, a1, a2, a3}
Fitness cases	all 16 possible combinations of 4 bits
Fitness measure	(hits, size)
Size threshold	50
AppGP runs	51
SGP runs	51

Table 5: 4-bit even parity parameters

Function set	{+, -, *, /, sin, cos, exp, log}
Terminal set	{x}
Fitness cases	20 evenly spaced points from the interval [-5, 5)
Fitness measure	(difference, hits, size)
Size normalization threshold	100
HMGP runs	101
SGP runs	101

Table 6: Symbolic regression with quartic polynomial as target

takes two arguments and evaluates the first if there is food ahead and the second otherwise. `begin2` takes two arguments and evaluates them sequentially. The terminals were the side-effecting thunks `(move)`, `(turn-right)`, and `(turn-left)`. For both systems, an individual's program was iterated until all 89 pieces of food were found or the number of moves exceeded 400. The additional problem-specific parameters for the Artificial Ant problem are listed in Table 4.

3.3 4 bit even parity problem

The n-bit even parity problem is the problem of determining whether a set of n input bits has an even number of 1's [Koza 1992]. We tested runs of the 4-bit even parity problem using the parameters listed in Table 5. The fitness component `hits` represents the number of cases correctly classified by the individual.

3.4 Quartic Polynomial

The quartic polynomial $x^4 + x^3 + x^2 + x$ was used as the target for a symbolic regression problem [Koza 1992]. The function set was {+, -, *, /, sin, cos, exp, log}[3]. The terminal set was {x}, where x is the independent variable. Fitness for both systems was the ordered triple `(difference, hits, size)`. `difference` was the sum of the absolute values of the differences between the actual value at each fitness case and the value of the individual at that point. `hits` was the number of fitness cases in which the difference between the actual value and the individual's value was less than 0.05. Other parameters for the Quartic Polynomial problem are listed in Table 6.

[3] Instead of using protected division and log, we take advantage of the fact that Java generates a NaN (Not a Number) value in cases such as division by or log of zero. We then normalize these NaN values to some very poor fitness (e.g., the maximum error).

Function set	$\{+, *, -\}$
Terminal set	$\{j, 0, 1, 2, 3\}$
Fitness cases	$j \in \{0, 1, 2, \ldots, 20\}$
Fitness measure	(difference, hits, size)
Size normalization threshold	100
HMGP runs	256
SGP runs	259

Table 7: Sequence Induction parameters

Prob.	N		Avg.		σ	
	APP	SGP	APP	SGP	APP	SGP
MAX	50	51	2.6E9	1.3E9	2.0E9	2.0E9
Ant	50	51	60.80	60.94	13.68	9.30
Parity	51	51	15.00	14.77	1.25	1.20
QPoly	101	101	402.1	421.3	317.3	348.2
S.I.	256	259	22905	27062	40108	42283

Table 8: Number of runs (N), mean of the best fitness in each run (Avg.), and standard deviation of the best fitness for each run (σ) for each problem (Prob.) using AppGP (APP) and SGP (SGP). For MAX the fitness is the tree value, with larger being better. For Artificial Ant the fitness component listed is `food eaten`, with larger being better. For 4-Bit Even Parity the fitness component listed is `hits`, with larger being better. For Quartic Polynomial (QPoly) and Sequence Induction (S.I.) the fitness component listed is `difference`, with smaller being better.

3.5 Sequence Induction

The sequence induction problem is an instance of the symbolic regression problem in which the domain of the independent variable is constrained to the natural numbers {0,1,2,3...}. We used the sequence $S_j = 5j^4 + 4j^3 + 3j^2 + 2j + 1$, as described in [Koza 1992]. The fitness measure used was the tuple `(difference, hits, size)`. `difference` was the sum of the absolute values of the difference between the individual and the target over the cases $j \in \{0, 1, 2, \ldots, 20\}$. `hits` represented the number of cases where the individual's value exactly matched the value of the target sequence. Other parameters for the sequence induction problem are listed in Table 7.

4 Results

Each of the problems described was run multiple times using both AppGP and SGP, with the results summarized in Table 8.[4] For the MAX problem, the Sequence Induction problem, and the Quartic Polynomial problem the average best fitness over the various runs is substantially better when using AppGP. For the 4-Bit Even Parity problem, AppGP did marginally better (on average less than one-quarter of a case out of 16), and for the Artificial Ant problem, SGP narrowly outperformed AppGP (on average by 0.14 pieces of food out of 89). For data that is approximately normal or uniform, a two-sample T-test is sufficient for determining significance; however, some of our data was highly skewed, so we used a two-sample Mann-Whitney-Wilcoxon rank sum test, verified by a log-scale two-sample T-test.

4.1 The MAX problem

The data for the MAX problem were sufficiently normal that a two-sample T-test sufficed; The T statistic for the MAX problem was 3.13, with an attained significance (1 - p-value) of 0.9977. This indicates that if the populations were identically distributed, we would have a 0.23% chance of randomly producing data as or more disparate than these. Thus the substantial difference observed (more than 1.3×10^9) in performance on the MAX problem was statistically significant and highly unlikely to be caused by random chance.

4.2 The Artificial Ant problem

Our data on the Artificial Ant problem was also approximately normal, and produced a T statistic of -0.06, for an attained significance of 0.05; thus there is absolutely no evidence that the difference observed is not caused by random variation. The tiny differences in performance on the Artificial Ant problem are therefore both statistically and practically insignificant.

4.3 The 4-Bit Even Parity problem

The T statistic for our 4-Bit Even parity data was 0.97, yielding an attained significance level of 0.67. Again, this indicates that there is no evidence for a statistically significant difference between the fitnesses of SGP and AppGP on this problem.

4.4 The Quartic Polynomial problem

The data for the Quartic Polynomial problem produced a T statistic of -0.41, yielding an attained significance of 0.32. Thus while AppGP outperformed SGP, we do not have sufficient evidence to conclude that this difference was not caused by random variation.

4.5 The Sequence Induction problem

The data for the Sequence Induction problem were heavily skewed right; thus we performed both a two-sample rank sum test and a log-scale T-test. The log-scale T-test produced a T statistic of -2.09 with 511 degrees of freedom, for an attained significance of 0.963; the rank sum test produced a W-statistic of 62903.5 for an attained significance of 0.94, verifying the log-scale T-test. Thus the difference observed is unlikely to have been caused by random variation.

5 Conclusions

Our tests with AppGP and SGP indicate that AppGP outperforms SGP on several problems of interest, while doing about the same on the other test problems. While it would be nice to conclude that AppGP does as well or better than SGP on all problems, such a conclusion would clearly be premature given the small number of problems tested. The results do indicate that AppGP has potentially significant value, however, and warrants further study.

5.1 When (and why) will AppGP do better?

The obvious question is when will AppGP outperform SGP, and why. The most likely candidate is the ability of AppGP to more easily modify parts of the expression tree that would be near the root in the SGP representation, and therefore extremely difficult to change in SGP. In the remainder of this discussion, we will refer to these as "root" operators.

For the MAX problem, this conjecture is strongly supported by examining the best of run trees on the AppGP and SGP runs of that problem. The optimal SGP tree for this problem has nothing but "*" operators in the first four levels of the tree (starting at the root); "*" and "+" operators are interchangeable on the next level, and all operators on lower levels should be "+". One way, then, of measuring the ability of GP to develop the "root" structure necessary to maximize the tree's value is to determine the size of the largest tree starting at the root (in the SGP representation) containing only "*" operators. For the optimal tree this count should be at least 15, but trees with "+" operators in undesirable positions at or near the root will have lower counts. Out of 50 AppGP runs, 30 best of run individuals had counts of at least 15, i.e., maximal values. Over 51 SGP runs, however, only 16 best of run individuals had counts of at least 15, with 33 having counts no greater than than 7! This is consistent with other research that suggests that premature convergence of the portion of the tree near the root is a significant issue in GP [Gathercole and Ross 1996, Rosca 1997, Poli and Langdon 1998, Langdon and Poli 1997, McPhee and Hopper 1999]

It would be nice if we could easily characterize the desired "root" structure for the other problems; unfortunately this is not entirely straightforward. In the Sequence Induction problem, for example, the inclusion of "−" in the function set introduces the possibility of introns and makes a simple characterization of the resulting trees more difficult.

Note also that the similarity of performance of AppGP and SGP on the Artificial Ant problem further supports the idea that the primary difference between AppGP and SGP is

[4] The raw data these statistics are based on is available at `http://www.mrs.umn.edu/~mcphee/Research`.

the ability to modify root operators. In the SGP representation for the Ant problem, all the active operations (`move`, `turn-left`, `turn-right`) are already leaves, and the two non-leaf operators (`if-food-ahead` and `begin2`) are essentially just "glue" operators that are used to join together sections of code. In this regard, the non-leaf nodes in the SGP representation of the Ant problem are not unlike the APP node in the AppGP representation. The only real difference between SGP and AppGP on this problem, then, is the ability to swap an `if-food-ahead` for a `begin2` and *vice versa*. The similarity of the AppGP and SGP results indicates, however, that adding this ability has no significant effect on performance, suggesting that this problem is not very sensitive to the choices made near the root.

5.2 Impact of partial functions

One substantial difference between HMGP and AppGP is that the HMGP representation allows for the representation (and swapping via crossover) of partial function applications like `(+ 1)`, a unary function that adds 1 to its argument. Neither the AppGP nor the SGP representations allow for partial function applications, and the question then arises of what effect (if any) partial functions have. The similarity of the HMGP and AppGP results on the majority of the problems tested here suggests that at least on these problems partial functions didn't play a major role. The set of test problems used here is small, however, and it's possible that there may be other problems for which partial functions have greater impact (positive or negative).

5.3 Speed of AppGP vs. SGP

In our previous paper, we showed that HMGP outperformed SGP on a variety of problems. One potential drawback of HMGP, however, is the substantial extra machinery required to implement the Hindley-Milner type system, as well as the various changes to, e.g., crossover to ensure that types are respected. As a result (in large part) of this extra machinery, HMGP also runs significantly slower than SGP.[5]

AppGP, however, avoids these drawbacks while still outperforming SGP on several problems. Implementing AppGP, and making the necessary changes to components like crossover, is straightforward, and AppGP could be easily added to most standard GP systems. As a result of this limited overhead, AppGP runs take essentially the same amount of time as SGP runs.

5.4 In summary…

In this paper we followed up previous work [McPhee *et al.* 1998*a*] by running a series of experiments comparing our new AppGP representation to the SGP representation on a suite of five problems. Our results suggest that AppGP substantially outperforms SGP on three of the five problems. Unfortunately statistically significance at the 95% confidence level is only obtained for two of the three problems where a substantial difference was found. This, and the small size of the test suite, indicate that further experimentation would be necessary before AppGP could be unreservedly recommended, but these results do suggest that AppGP may outperform SGP on some problems of interest while incurring little additional computational overhead.

[5] In our experience HMGP runs took about four times as long as similar SGP runs.

Acknowledgments

Jon Anderson provided invaluable assistance with statistical analysis.

Hopper was supported by the University of Minnesota Undergraduate Research Opportunities Program. A grant from the University of Minnesota Graduate School Grant-in-aid program funded the purchase of computing equipment used in these experiments.

Bibliography

Gathercole, Chris and Peter Ross (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (John R. Koza et al., Eds.). MIT Press. Stanford University, CA, USA. pp. 291–296.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press. Cambridge, MA, USA.

Langdon, W. B. and R. Poli (1997). An analysis of the MAX problem in genetic programming. In: *Genetic Programming 1997: Proceedings of the Second Annual Conference* (John R. Koza et al., Eds.). Morgan Kaufmann. Stanford University, CA, USA. pp. 222–230.

McPhee, Nicholas Freitag and Nicholas J. Hopper (1999). Analysis of genetic diversity through population history. In: *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference* (W. Banzhaf et al., Eds.). Morgan Kauffman. Orlando, Florida USA.

McPhee, Nicholas Freitag, Nicholas J. Hopper and Mitchell L. Reierson (1998*a*). Impact of types on essentially typeless problems in GP. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference* (John R. Koza et al., Eds.). Morgan Kaufmann. University of Wisconsin, Madison, Wisconsin, USA. pp. 232–240.

McPhee, Nicholas Freitag, Nicholas J. Hopper and Mitchell L. Reierson (1998*b*). Sutherland: An extensible object-oriented software framework for evolutionary computation. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference* (John R. Koza et al., Eds.). Morgan Kaufmann. University of Wisconsin, Madison, Wisconsin, USA. p. 241.

Poli, Riccardo and William B. Langdon (1998). On the search properties of different crossover operators in genetic programming. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference* (John R. Koza et al., Eds.). Morgan Kaufmann. University of Wisconsin, Madison, Wisconsin, USA. pp. 293–301.

Rosca, Justinian P. (1997). Hierarchical Learning with Procedural Abstraction Mechanisms. PhD thesis. University of Rochester. Rochester, NY 14627.