Exploring Alternative Operators and Search Strategies in Genetic Programming

Kim Harries and Peter Smith

Department of Computer Science
City University
Northampton Square
London EC1V 0HB
United Kingdom

E-mail: {kim, peters}@soi.city.ac.uk

Tel: +44 171 - 477 8551

ABSTRACT

This paper investigates a number of alternatives to the standard genetic programming operators, along with an alternative to the standard evolutionary search strategy, and compares and analyses their performance. Four test problems with differing properties are used to assess which operators perform best for which strategy, the comparative performance of alternative search strategies, and the extent to which the success of a particular method is dependent upon the problem domain. In all this paper presents the results of 58,100 independent runs spanning fourteen operators for two search strategies on four problem domains.

1 Introduction

Genetic programming (GP) [1] has been successfully applied to a wide range of problem domains [1, 2, 3]. Its success is generally attributed to its use of an evolutionary search technique based on the genetic algorithm (GA) [4] combined with a dynamic, tree-structured representation of the programs. Much recent research has focused on improving the performance of GP by improving its representation [2, 5, 6, 7, 17, 18], and this has proven to be an effective means of enhancing the basic algorithm. In theory the same effect as a change in representation could be achieved by a change to the transformation rules (in GA, the genetic operators). However, some effects may be more easily and practically achieved by one method rather than another. It is the purpose of our current research to explore the effectiveness of alternative transformation rules, both in evolution based GP and alternative approaches. This paper describes our work comparing the performance of GP with several non-standard genetic operators and several varieties of a form of hillclimbing based GP.

Recent research by O'Reilly and Oppacher [8, 9] has focused on a similar analysis. In [8] they compare the performance of GP using three different crossover operators with both hillclimbing (HC) and simulated annealing (SA) methods that use a mutation operator designed by themselves called HVL-Mutate. In [9] they extend this work to look at using crossover-style operators in SA and HC and at combining GP with HC/SA to produce hybrid algorithms. Since they use only a small number of independent runs (~30) in their tests it is difficult for them to obtain statistically significant results where performances do not differ by a substantial margin. In order to allow us to draw conclusions concerning the relative effectiveness of operators that are closer in performance we have used a minimum of 100 runs on each of our tests, with the results from several thousand independent runs per test being used in many cases. This has allowed us to analyze differences that would not otherwise be evident and hopefully gain a greater insight into the various strategies.

The paper is organized as follows. Section 2 gives a brief introduction to genetic programming, including a description of how the standard crossover and mutation operators work in GP. This is followed by a description of several possible types of non-evolution based genetic programming, and the alternative genetic operators that are to be used in the experiments. Section 3 presents the experiments and their results, with a discussion being provided in section 4. Finally, suggestions for further work are supplied in section 5.

2 Genetic Programming Strategies

2.1 Evolution Based Search

The standard genetic programming paradigm uses the evolution-based search strategy of the genetic algorithm, and can be seen as a subcategory of GA. An initial population of individuals (programs in the case of GP) are created by some randomized procedure. These are evaluated using the fitness measure to determine their ability at solving the problem being tackled. Then the population for the next generation are produced by selecting members with a bias towards the fitter ones, and modifying them via the application of various genetic operators to create new individuals. The new individuals are tested for fitness, selected by fitness, and modified to produce the next generation. This process of evaluate-select-modify continues until the termination criterion is met (usually the production of a good enough solution or a predefined limit on the number of generations being reached). The defining feature of a genetic algorithm is the use of a population of individuals to search multiple points in the search space simultaneously in combination with genetic operators that recombine elements from more than one member of the population to produce new individuals. Other operators modifying a single individual are usually used in addition to this. In GP the population consists of individual program trees, which are parse trees in a simple problem dependent language with dynamically determined size and shape. The standard GA operators crossover and mutation have logical analogues in GP as follows. For a full description of GP however see Koza [1].

2.1.1 Crossover

The crossover operator used in GP takes two program trees as parents, and produces two offspring. This is done by selecting a random node in each parent and then exchanging the subtrees for which those nodes are the roots to produce the new trees. It has been noted that due to the rapid expansion of trees with a branching factor greater than one, crossover performed in this manner will spend a large proportion of its time exchanging the leaves of trees (for binary trees approximately half the nodes are leaf nodes). It is therefore common practice (following Koza [1]) to impose a bias such that 90% of crossover points are function nodes. Crossover is normally performed with a high probability (90 - 100%).

2.1.2 Mutation

The mutation operator is intended to be analogous to the occasional random corruption of DNA that occurs in biological systems. The standard mutation operator therefore selects a random node within the program tree, deletes it and all its branches, and then replaces it with a randomly created subtree. Mutation is normally performed with a low probability (0 - 1%).

2.2 Alternative Operators

There are many ways in which the standard GP operators could be varied. In this paper however we have restricted ourselves to looking only at a particular class of modifications to the crossover operation.

2.2.1 Depth-based crossover

Recent research has questioned the implicit biases of the standard crossover operator [12, 13, 14]. In particular its bias towards the nodes at the bottom of a tree has caused concern. Therefore we decided to explore the effectiveness of alternative crossover operators that do not exhibit the same bias. A crossover operator that first selects a random depth at which to perform crossover and then selects a subtree at that depth was designed. An immediate question arises when designing this operator however: should the depth for crossover be chosen separately for each parent, or once for both of them? Choosing separately would allow genetic material to move more freely in a vertical manner, which might be useful in preventing some of the problems highlighted by Gathercole and Ross [14], who suggest that the standard crossovers bias towards exchanging material at leaf nodes results in a loss of diversity in the nodes higher up in the program trees. This potentially creates similar problems to those caused by allele loss in normal GA. Using the same depth for each tree however would promote crosses between parts of the program which are more likely to be serving a similar purpose, and thus may be more likely to spread useful subroutines throughout the population. We decided to examine both alternatives and compare their performance. Since the appropriate level of

bias is to be determined we also investigate an operator that half the time works as standard crossover, and half the time uses depth-based crossover. Finally, we examine an operator that uses the standard technique for picking the subtree from one parent, and uses a depth-based system for picking the subtree from the other parent.

Thus we have five different depth-based operators. SameDepths selects a random depth compatible with both parents and then swaps randomly chosen subtrees at that depth from each parent. DiffDepths selects a random depth for each parent independently and then swaps randomly chosen subtrees from those depths. This corresponds to the Height-Fair-XO used by O'Reilly and Oppacher [8]. Std/Same uses Standard Crossover 50% of the time, and SameDepths crossover otherwise. Std/Diff uses Standard Crossover 50% of the time, and DiffDepths otherwise. Half&Half selects a subtree from the first parent by first choosing a random depth then selecting a random subtree at that depth, and selects a subtree from the second parent completely randomly. For comparison we also looked at Standard crossover, which uses a 90% bias towards function nodes, and NoBias crossover which simply selects a crossover point entirely at random.

2.3 Alternative Strategies

Although genetic algorithms have proven successful at solving many complex problems [10] they are still in constant competition with alternative search methods. In particular other stochastic methods often perform well on the same classes of problems for which GA is well suited [16]. It is therefore sensible to explore the possibility that alternative search strategies may provide a viable means to augment the process of automatic program induction.

When initially deciding which algorithms to compare to the traditional evolutionary search we considered briefly the possibility that a blind search such as Iterative Deepening might be able to compete for the sorts of problems to which genetic programming has often been applied. Although it was realized that it would obviously be implausible to fully explore all the program trees of the sizes that are commonly encountered in GP, it seemed possible that since there is generally a far more parsimonious solution available than the ones actually found by GP that a complete search of a restricted search space might prove competitive. This was not found to be the case except for the very simplest problems, and when the size of the search spaces involved is calculated we soon see why. For a problem with two terminals and two functions, each of arity two, the number of program trees of depth three is 192, this rises to 10^{20} different trees at depth six and 10^{166} at depth nine. Clearly the growth in the size of the search space with the depth of the trees is so exponential as to be prohibitive for all but the easiest problems.

A more fruitful alternative is provided by Hill Climbing (HC) or Simulated Annealing (SA). These each proceed by exploring a single point in the search space and then moving to the best of its neighbouring points, except that SA will sometimes choose a successor which is worse than the current point in order to escape local maxima. Both these alternatives are considered by O'Reilly and Oppacher, but we decided to look at only one for reasons given below.

2.3.1 HCGP

What counts as a neighbour for a particular program will of course depend on how you choose to describe the search space. Whatever transformation rules (genetic operators) are used however it is likely that the number of neighbours for a program of a reasonable size will be large enough that evaluating them each before proceeding will slow down the search process to the point of infeasibility. Therefore we instead decided to use a form of stochastic hillclimbing where a neighbour is chosen at random (by applying the appropriate genetic operators) and may replace the current one depending on its fitness. Initially we decided that the current program should only be replaced if its fitness is exceeded (normally hillclimbing is a greedy algorithm, always proceeding along the steepest incline), but preliminary tests showed that a much better performance was achieved by accepting any new program whose fitness is greater than or equal to the current program. This is presumably because a constantly changing program is less likely to get stuck in local optima. The resulting algorithm is thus very loosely referred to as 'hillclimbing', since it does not in fact evaluate all neighbouring points in the search space and proceed to the fittest.

It was decided not to investigate a separate SA style algorithm because it was considered that in a high dimensionality search space such as this the use of random walk to avoid local optima is likely to be generally as effective as more sophisticated techniques. O'Reilly and Oppacher's HVL-Mutate operator is specifically designed to make the smallest changes possible to a program tree and thus produce an action more similar to the local search of hillclimbing. This is more likely to have problems with local optima and therefore warrants the investigation of an SA style technique. Since we intended to use the mutation operator from standard GP which is capable of making large leaps through the search space (or alternatively can be seen as making the search space of higher dimensionality) this was not such an important issue.

Within this paradigm there are of course a large number of possible ways to implement the specific search strategy.

Our interest here is focused on the effects of different genetic operators, so therefore we chose to look at three alternative operators for HCGP. Firstly, the standard mutation operator from GP. Secondly, a form of crossover called self-crossover used by Kinnear in [11]. Thirdly, a new mutation operator called small-mutation designed to allow the minimum modification possible to be made to a program. The standard mutation operator works as described in 2.1.2. Self-crossover works by performing the standard crossover operator between the program and an identical copy of itself. In these experiments there was no bias towards function nodes used when applying self-crossover. Small-mutation works by first selecting a random node. If this node is a terminal then it is replaced by another randomly chosen terminal (this could result in no change if the same terminal is selected). If it is a function node then it is replaced by a randomly chosen function node, but its arguments are as much as possible kept the same. If the two functions nodes are of the same arity then the arguments are left unchanged. If the replacement node is of a lower arity then any extra arguments are simply ignored. If however the replacement node is of a higher arity then the further arguments are supplied by making copies of the original functions arguments, starting back at its first argument and looping through them.

These three operators were either used on their own, or in combination with one or more of the other operators to effectively form a more complex operation. Thus when for example self-crossover and mutation are used together self-crossover is first applied to the tree, followed by mutation. In this way the 7 combinations consisting of one or more of these operators each constitutes a different transformation rule to be investigated. In future experiments we intend to include O'Reilly and Oppacher's HVL-Mutate operator in our set for comparison purposes. The termination criterion for HCGP was kept the same as GP for each of the problems - either the production of a sufficiently optimal program, or reaching a previously specified number of evaluations (in these experiments 51,000 - which is equivalent to 50 generations of GP with a population size of 1000).

3 The Experiments

Four familiar problems were used as our testbed for evaluating the performance of the alternative strategies: The boolean even-4-parity and even-5-parity problems, the artificial ant problem, and the simple symbolic regression problem for the function $f(x) = x^4 - 3x^3 + 9x^2 - 27x$. The first three are implemented exactly as described in Koza [1]. The regression problem used the function set F = {+, -, *, %} and the terminal set T = {X, $\Re$}, where $\Re$ is an ephemeral random constant in the integer range -10 to +10, and was tested over ten arbitrary values of X between -2.0 and 2.0. A successful program was designated as one achieving ten hits, where a hit is returning a value within 1% of the correct f(x) value. These four were chosen to represent different properties that can be present in GP programs. The even-parity problems have conditional evaluation of the branches of their functions, thus allowing for the easy appearance of introns. It is reasonable to expect that the presence and abundance of unevaluated areas of code may have a large influence on the effectiveness of a particular operator. Some operators may be more likely to produce introns than others, or more likely to have their effects disrupted by them, etc. By comparing the performance between the even-4-parity and even-5-parity problems it will be possible to gain an indication of whether the performance of a particular technique scales well with problem difficulty - a crucial factor for determining its usefulness for more serious problems. The symbolic regression problem on the other hand always has full evaluation, although the results may be ignored in the case of e.g. division by zero. Thus although introns are still possible they are much less likely. It is also the only one of our test suite that uses constants. It is thus possible in this problem to have a program that is structurally correct with just a constant being slightly inaccurate. Such a problem domain would seem to be more likely to benefit from operators which were capable of making less drastic modifications. Finally the artificial ant problem is different because the functionality of the programs is determined by their side effects, not their return value, and because it contains the use of state - the state of the world is affected by the ant and vice-versa. It also has the unusual feature that program execution may be terminated and returned to the beginning of the program (when food is found). These properties may tend to make artificial ant program trees more brittle to modification.

The specific implementation of the genetic programming algorithm we used differs from that used by Koza [1] in the following ways. Our algorithm uses steady-state rather than generational replacement, which is increasingly common following an analysis by Syswerda [15], and tournament selection with a tournament size of 2. It also made use of the Hoist operator described by Kinnear in [11], which involves selecting a random subtree which is then used to replace the entire tree. Also, since the programs were stored as arrays a size limit of 1000 nodes per tree was used as a more practical alternative to a limit on the depth of the trees. Population size was set at 1000 for all problems, Crossover was applied with a probability of 1.00, Mutation with a probability of 0.01, and Hoist with a probability of 0.01. This implementation can not be described as canonical but is probably more similar to the algorithms actually used in practice. In particular we feel

that comparing different crossover operators used in isolation may be misleading since how they interact with other operators will be an important part of their effectiveness.

3.1 Results

Table1 below presents the complete results from 58,100 independent runs, over four problem domains for fourteen different search strategies. The first figure in each cell is the percentage of runs that were successful in finding a sufficiently optimal solution for that problem as defined by the success criterion. The value in parentheses is the number of independent runs over which this figure was obtained. The last column shows the average success rate of each strategy over the four problems, and is meant only as a rough guide to performance since its statistical significance is not testable.

The data in this table has then been summarised for easier viewing in eight bar charts showing the comparative performance of the various operators on each of the problem domains.

Table 1: Performance of GP and HC operators on four problem domains

	Even-4-Parity	Even-5-Parity	Regression	Artificial Ant	Average
GP-Standard	72.9% (7000)	09.4% (1000)	37.3% (1000)	37.4% (2000)	39.3%
GP-NoBias	70.9% (1000)	09.9% (1000)	39.1% (1000)	36.7% (1000)	39.2%
GP-SameDepths	35.9% (3000)	00.6% (1000)	41.7% (1000)	35.2% (2000)	28.4%
GP-DiffDepths	73.6% (3000)	11.4% (1000)	22.1% (2000)	26.4% (1000)	33.4%
GP-Std/Same	76.5% (8000)	11.8% (2000)	49.5% (1000)	41.6% (4000)	44.9%
GP-Std/Diff	74.4% (3000)	13.2% (1000)	25.6% (1000)	33.0% (1000)	36.6%
GP-Half&Half	74.0% (3000)	12.5% (1000)	25.8% (1000)	34.7% (1000)	36.8%
HC-Small	00.0% (100)	00.0% (100)	00.0% (100)	00.0% (100)	00.0%
HC-Mut	96.0% (100)	22.7% (300)	41.0% (200)	36.0% (100)	48.9%
HC-Self	01.0% (100)	00.0% (100)	01.0% (100)	12.0% (100)	03.5%
HC-Mut&Small	98.0% (100)	21.3% (300)	10.0% (100)	58.0% (100)	46.8%
HC-Self&Small	27.0% (100)	05.0% (200)	12.0% (200)	27.0% (100)	17.8%
HC-Self&Mut	97.0% (100)	38.3% (300)	19.0% (200)	65.0% (100)	54.8%
HC-All	98.0% (100)	33.0% (300)	09.0% (200)	73.0% (100)	53.3%

Note that the number of independent runs for the hillclimbing searches is in the hundreds rather than the thousands. This is because the different operators being used here tended to have a much more dramatic affect on performance and thus required less runs to produce a statistically significant result.

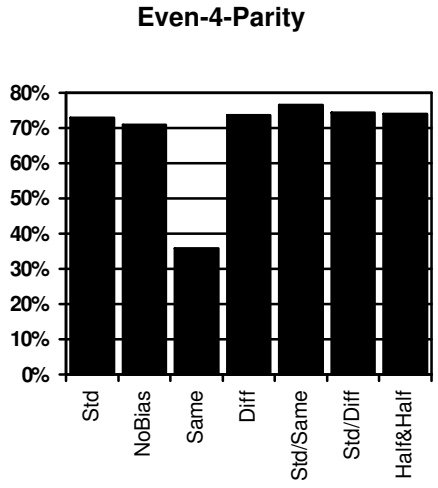

Figure 1: Even-4-Parity problem using GP

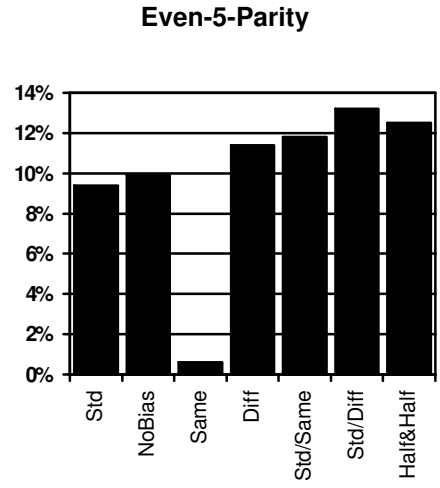

Figure 2: Even-5-Parity problem using GP

Two things to note on the even-4-parity problem: the very poor performance of SameDepths, and the fact that Std/Same outperforms all other strategies.

Again, note the very poor performance of SameDepths. This time Std/Diff has the highest success rate, but due to

the smaller sample size and the lower number of successful this figure is not significantly better than that of Std/Same.

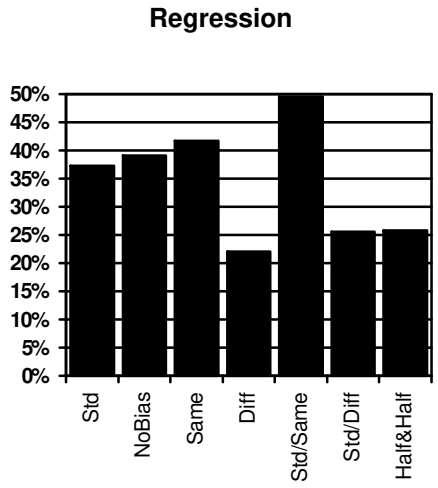

Figure 3: Regression problem using GP

Std/Same is clearly superior here, with the performance of DiffDepths, Std/Diff and Half&Half being particularly poor.

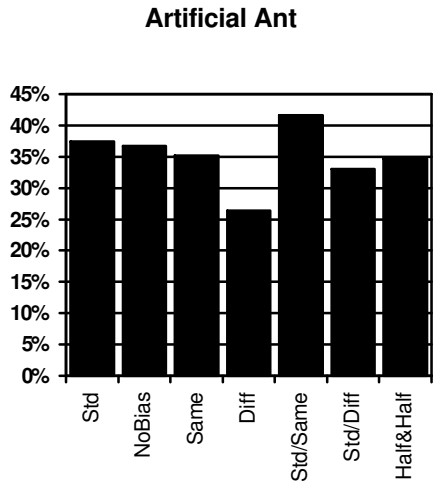

Figure 4: Artificial Ant problem using GP

Again Std/Same comes out best and DiffDepths does badly. The others perform fairly similarly, with Std/Diff perhaps being slightly worse (the difference between it and Standard is only significant at the 5% level).

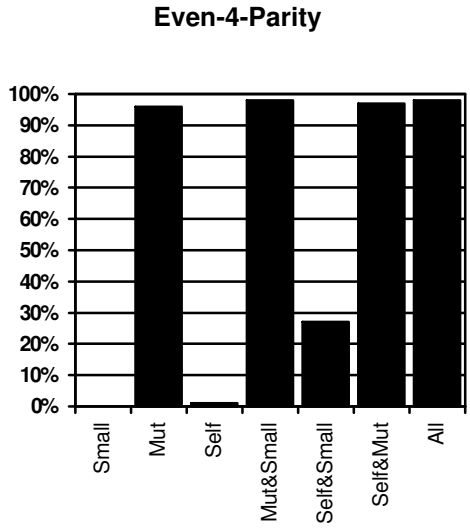

Figure 5: Even-4-Parity problem using HC

Both Small and Self do very poorly by themselves, and still a lot worse than the others when combined only with each other. Any combination that includes Mutation finds this problem very easy.

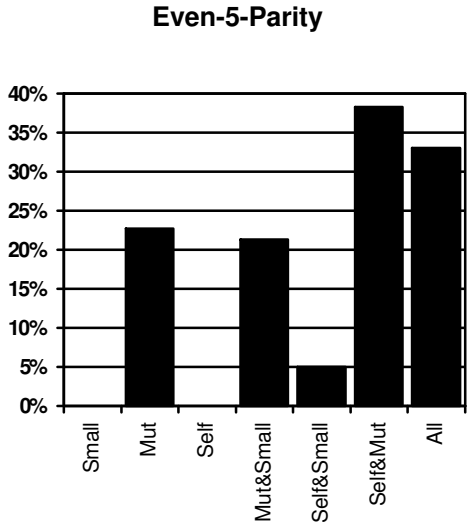

Figure 6: Even-5-Parity problem using HC

Small and Self do equally badly as before. The best combination appears to be Self&Mut, although its advantage over All is not statistically significant.

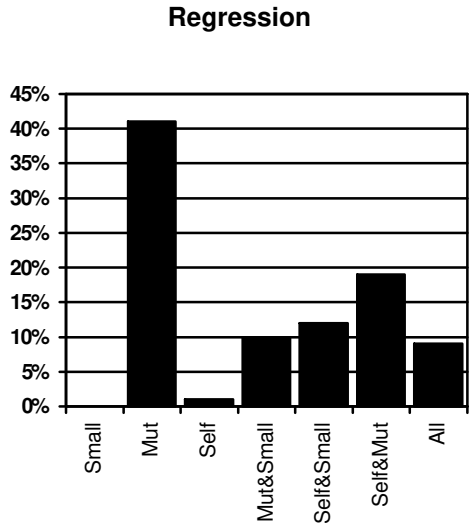

Figure 7: Regression problem using HC

Small and Self provide their routine ineffectualness. Surprisingly the best performer is clearly Mutation when used by itself.

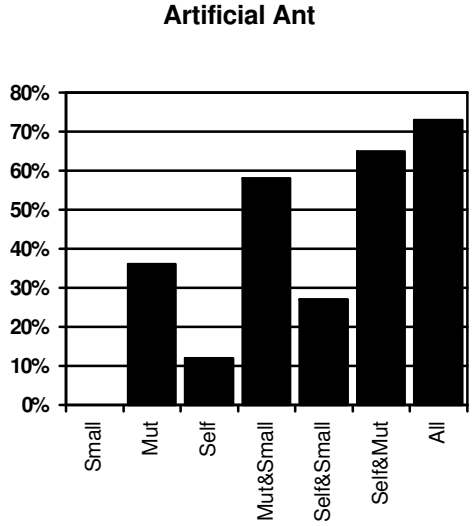

Figure 8: Artificial Ant problem using HC

All combinations involving Mutation perform pretty much the same, but Mutation by itself fairs slightly worse than all of them. This is the only problem where Self obtains a number of successes high enough to not be just flukes. Small fails once more to obtain any successes.

4 Discussion

In the following discussion where reference is made to statistical significance this implies the use of the two-sample z-test for two population proportions [19], and unless otherwise stated a significance level of at least 1% has been used.

There are a number of interesting things to observe from these results. GP-SameDepths performs significantly and substantially worse than the other GP operators on the even-parity problems, whereas on the regression and artificial ant problems its performance is similar to GP-Standard (the difference is not significant). GP-DiffDepths on the other hand does not perform significantly differently to GP-Standard on the parity problems, but is significantly and substantially inferior on the regression and artificial ant problems. Clearly the structural dynamics caused by different problem domains interacts in a critical fashion with the behaviour of these two alternative operators. Examining the details of these runs showed that for most of the operators the size of the best of run program tree was large (500+ nodes) but for the GP-SameDepths runs it was comparatively small (<100 nodes). Obviously only swapping subtrees from the same depth reduces the likelihood of extreme changes in the program size, thus limiting the rate at which the program grows. It's possible that in some cases it may be beneficial to guide more of the search effort towards smaller trees in this fashion but it seems unlikely that this is solely responsible for the differences between GP-SameDepths and GP-DiffDepths since other operators still perform well whilst producing large program trees.

Even more interesting is the comparative performance of GP-Std/Same and GP-Std/Diff (where standard crossover and depth-based crossover are used 50% of the time each). In all cases where there is a significant difference GP-Std/Same outperforms all other GP operators. Although on the even-5-parity problem GP-Std/Diff performs significantly better than GP-Standard, the difference with GP-Std/Same is *not* significant. On the even-parity problems the difference between the various operators is small (except for GP-SameDepths which is much worse). On the regression and artificial ant problems GP-Std/Same is significantly better than every other operator. This is only by a small margin for the artificial ant problem but by a fairly substantial amount for the regression problem (25-100% more successful runs). It seems that mixing standard crossover with SameDepth crossover overcomes the latter's weaknesses and improves on its strengths. DiffDepths is likewise improved, though not to the same extent. We hypothesise that GP-SameDepth's weakness and operators involving DiffDepth's comparative strength on the even-parity problems may be due to the fact that these have a very definite hierarchical structure with certain subroutines (e.g. lower order parity functions) being useful at several levels in a successful program. Thus it is important to transport successful code vertically in the tree, allowing repeated

sequences to occur at different points in the hierarchy. This factor would clearly become more important for higher order parity functions and so this theory is supported by the relative improvement in performance of all the operators using some form of DiffDepth crossover on the even-5-parity problem. We are currently testing higher order parity problems to see if this holds true. Conversely on the regression problem it may be the case that swapping code from different depths of the trees is likely to be too destructive, particularly since unlike the other problems where large areas of code are likely to be unevaluated introns in the regression problem all nodes are evaluated and so any change to a tree will probably change its fitness.

The performance of GP-NoBias is never significantly different from that of GP-Standard. This suggests that either the kind of bias that is present in GP-Standard towards function nodes is not the right kind of bias, or perhaps that it is too small a bias to make a significant difference. GP-Half&Half performs in a very similar manner to GP-Std/Diff (there is never a significant difference), which is not surprising since both will have a tendency to swap subtrees from different depths within the program trees with a greater frequency.

Looking at the results for the hillclimbing algorithm a number of points are immediately striking. Its performance on the even-parity problems is impressive, being significantly and substantially better than all of the GP operators except when standard mutation is not used (the other two operators were designed to enhance the use of standard mutation and therefore their poor performance when used in isolation was not altogether surprising). Likewise the performance of HC-All, HC-Mut&Small, and HC-Self&Mut on the artificial ant problem are significantly superior by a reasonable amount to even the best from GP. In contrast to our expectations of this problem being potentially brittle to modifications it appears to be particularly lenient towards them, producing better results the more operators are used. This may simply be because this problem allows so many introns (any IF_FOOD_AHEAD function as the second argument of another will automatically have an intron as its first argument, for instance) that as programs grow much more modification of the trees is necessary to have a chance of a beneficial result.

On the regression problem HC does not perform so well. HC-Mut is significantly worse than GP-Std/Same, though comparable to the next best GP technique. All the other HC operators perform very badly however, being significantly and substantially worse than HC-Mut and the GP operators. This is probably again due to the fact that program trees for this problem are much less likely to contain introns and so HC suffers from the quite destructive standard mutation operator, and this is only made worse by adding more types of mutation. Perhaps an operator like HVL-Mutate designed to make small changes would be more effective for this problem. Comparing the performance between the different HC versions shows that in general the standard mutation operator is necessary to acquire a good performance, though it may be enhanced by the addition of the other operators. Small-mutation by itself proves totally incapable of providing an effective search, as does Self-crossover. Small-mutation operates mainly by substituting one node for another node of the same type (function or terminal). The changes it makes are thus very small indeed, and in particular it is not capable of increasing the depth of a tree. Thus by itself it cannot constitute an effective exploration of the possible program trees. Self-crossover when applied by itself is not capable of introducing any new genetic material into a tree, and is thus very vulnerable to premature allele loss Although together they seem to combine to be more effective it is still clearly better when Standard-mutation is part of the operator set.

Main points:

- GP-Std/Same outperforms the other GP operators, and thus constitutes an improvement to the basic GP mechanism.
- Operator performance is sensitive to the problem domain, with combinations of operators appearing more robust.
- HCGP is a strong competitor to standard GP, although its performance seems to be more sensitive to the interaction between the operators and the problem domain.

5 Further Work

The alternative genetic operators examined in this paper are of course only a very small subset of the infinite number of possible operators that could be designed, and have been tested on only a small subset of the infinite number of possible problems that could be encountered. We shortly intend to expand our testbed of problems to include a richer variety of domains which will provide a greater basis for comparison between operators. To get a better insight into what is really going on however what is required is a detailed analysis of the particular affects of applying certain operators to certain program trees. By recording transformations that are responsible for particularly significant improvements in fitness and examining them it is hoped to gain a superior understanding of the properties of the various operators.

Our long term goal is of course to develop a set of high-performance operators for GA and/or HC based program induction. Since a transformation rule or operator can be seen as a function or program, we have taken the natural

step of applying program induction techniques to the problem of finding successful genetic operators. This is an especially difficult task for a number of reasons but in particular because the evaluation of any potential operator is so time consuming. The work described in this paper was in part motivated by a desire to test the feasibility of discovering superior genetic operators before proceeding to apply program induction techniques to the problem.

Acknowledgments

We would like to thank Dr Linda Wolstenholme of the City University Department of Actuarial Science and Statistics for her helpful advice. We would also like to thank the reviewers for their comments and criticism which have helped us to improve this paper.

References

[1] Koza, J.R. (1992) *Genetic Programming*. Cambridge, MA: MIT Press.

[2] Koza, J.R. (1994) *Genetic Programming II*. Cambridge, MA: MIT Press.

[3] Kinnear, K.E, Jr. (editor) (1994) *Advances in Genetic Programming*. MIT Press.

[4] Holland, J.H. (1975) *Adaption in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

[5] Angeline, P.J. and Pollack, J.B. (1993) Evolutionary Module Acquisition. *In Proceedings of the Second Annual Conference on Evolutionary Programming*, D.G. Fogel and W. Atmar (eds.), La Jolla, CA: Evolutionary Programming Society, pp. 154-163.

[6] Whigham, P. (1995) Inductive Bias and Genetic Programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, UK:IEE, pp. 461-466.

[7] Koza, J.R. (1994) Scalable Learning in Genetic Programming using Automatic Function Definition. In *Advances in Genetic Programming*, MIT Press, pp 99-118.

[8] O'Reilly, U.-M. and Oppacher, F. (1994) Program Search with a Hierarchical Variable Length Representation: Genetic Programming, Simulated Annealing and Hill Climbing. In *Parallel Problem Solving from Nature - PPSN III, volume 866 of Lecture Notes in Computer Science,* Davidor, Y. Schwefel, H.-P., and Manner, R. (eds.). Berlin Springer Verlag, pp.397-406.

[9] O'Reilly, U.-M. and Oppacher, F. (1996) A Comparative Analysis of Genetic Programming. In *Advances in Genetic Programming 2*, Angeline, P.S. and Kinnear, K.E. Jr. (eds), MIT Press, Cambridge Mass.

[10] Goldberg, D.E. (1989) Genetic Algorithms in Search, Optimisation and Machine Learning. Reading, MA: Addison Wesley, pp 126-129.

[11] Kinnear, K.E. Jr. (1994) Alternatives to Automatically Function Definition. In *Advances in Genetic Programming*, Kinnear, K.E. Jr. (editor), MIT Press, pp 119-141.

[12] Angeline, P.J. (1996) An Investigation into the Sensitivity of Genetic Programming to the Frequency of Leaf Selection During Subtree Crossover. In *Proceedings of the First Annual Conference on Genetic Programming*, Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.), MIT Press, pp21-29.

[13] Soule, T., Foster, J.A., Dickinson, J. (1996) Code Growth in Genetic Programming. In *Proceedings of the First Annual Conference on Genetic Programming*, Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.), MIT Press, pp 215-223.

[14] Gathercole, C., Ross, P. (1996) An Adverse Interaction between Crossover and Restricted Tree Depth in Genetic Programming. In *Proceedings of the First Annual Conference on Genetic Programming*, Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.), MIT Press, pp 291-296.

[15] Syswerda, G. (1991) A Study of Reproduction in Generational and Steady-State Genetic Algorithms. In Foundations of Genetic Algorithms, Rawlins, G.J.E. (editor), San Mateo, CA: Morgan Kaufmann.

[16] Reeves, C.R. (editor) (1995) *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill.

[17] Altenberg, L. (1994) The Evolution of Evolvability in Genetic Programming. In *Advances in Genetic Programming*, MIT Press, pp 47-74.

[18] Gruau, F. (1994) Genetic Micro Programming of Neural Networks. In *Advances in Genetic Programming*, MIT Press, pp 495-518.

[19] Weiss, N.A. (1995) *Introductory Statistics, Fourth Edition*. Addison-Wesley, pp 671-678.