Meta-Genetic Programming: Co-evolving the Operators of Variation

Bruce Edmonds,
Centre for Policy Modelling,
Manchester Metropolitan University,
Aytoun Building, Aytoun Street, Manchester, M1 3GH, UK.
Tel: +44 161 247 6479 Fax: +44 161 247 6802
E-mail: b.edmonds@mmu.ac.uk URL: http://www.fmb.mmu.ac.uk/~bruce

Abstract

The standard Genetic Programming approach is augmented by co-evolving the genetic operators. To do this the operators are coded as trees of indefinite length. In order for this technique to work, the language that the operators are defined in must be such that it preserves the variation in the base population. This technique can varied by adding further populations of operators and changing which populations act as operators for others, including itself, thus to provide a framework for a whole set of augmented GP techniques. The technique is tested on the parity problem. The pros and cons of the technique are discussed.

Keywords

genetic programming, automatic programming, genetic operators, co-evolution

1. Introduction

Evolutionary algorithms are relatively robust over many problem domains. It is for this reason that they are often chosen for use where there is little domain knowledge. However for particular problem domains their performance can often be improved by tuning the parameters of the algorithm. The most studied example of this is the ideal mix of crossover and mutation in genetic algorithms. One possible compromise in this regard is to let such parameters be adjusted (or even evolve) along with the population of solutions so that the general performance can be somewhat improved without losing much of the generality of the basic algorithm[1]. The most frequently quoted is [4], which investigates the incremental tuning of parameters depending on the success of the operators in initially converged populations.

Such techniques have also been applied in genetic programming (GP)[2]. This usually involved the storing of extra information to guide the algorithm, for example in [7] extra genetic material is introduced to record a weight effecting where the tree-crossover operator will act. As [1] says, "*There is no reason to believe that having only one level of adaption in an evolutionary computation is optimal*" (page 161).

In genetic programming the genetic operators are usually fixed by the programmer. Typically these are a variety of crossover and propagation, but can also include others, for example mutation. A variety of alternative operators have been invented and investigated (e.g. [2]). It is usually found that, at least for some problems these preform better than the 'classic' mix[3] of mostly tree-crossover and some propagation (e.g. [3]).

Meta-genetic programming (MGP) encodes the operators that act on the selected genes as trees. These "act" on other tree structures to produce the next generation. This representation allows the simultaneous evolution of the operators along with the population of solutions. In this technique what is co-evolved is not just a fixed set of parameters or extra genetic material associated with each gene, but operators which are themselves encoded as variable length representations – so their form as well as their preponderance can develop.

This technique introduces extra computational cost, which must be weighed against any advantage gained. Also the technique turns out to be sensitive to biases in the syntax from which the operators are generated, it is thus much less robust.

Iterating this technique can involve populations of operators acting on populations of operators acting on populations etc., and even populations acting on themselves. This allows a range of techniques to be considered within a common framework - even allowing the introduction of deductive operators such as Modus Ponens to be included.

In section 2, I describe the basic technique in more detail. In section 6, I describe the use of possible elaborations of MGP to provide a coherent framework for the analysis of a wide range of such structures. I test a few configurations of MGP on the parity problem at different levels of difficulty in section 4, followed by a general discussion (section 8). I finish with a small sample of the further work that needs to be done on this technique in section 9

2. The Basic Technique

The technique is close to 'classic' GP except that instead of being hard-coded, the operators are represented as tree structures. For simplicity this is an untyped tree structure in the manner of Koza [9] although a strongly typed structure [12] might be more appropriate in general. Two chosen

1. For a survey, which covers such approaches see [16].
2. As well as in evolutionary programming, e.g. [5].
3. i.e. as described in [9].

trees along with randomly chosen nodes in those trees are passed to be operated on by the chosen operator tree. The terminals of the operator tree refer to one or other of these trees at a node related to the one passed. The structure of the operator tree represents transformations on the passed (tree, node) pair until the root node produces the resulting gene to enter the next generation.

Thus there could be the following terminals:

- rand1 - return the first tree, with its randomly chosen node;
- rand2 - return the second tree, with its randomly chosen node;
- bott1 - return the first tree, but with the root node selected;
- bott2 - return the second tree, but with the root node selected;

and the following nodes:

- top - cut the passed tree at the indicated node and return it with the new root node selected;
- up1 - pass the tree as it is but choose the node immediately up the first branch (if there is such a branch, otherwise just pass the tree as it is);
- up2 - pass the tree as it is but choose the node immediately up the second branch (if there is such a branch, otherwise just pass the tree as it is);
- down - pass the tree as it is but choose the node immediately down from that indicated (if possible otherwise just pass the tree as it is);
- down1 - identical to down (so that the operators will not have an inherent bias due to the prevalence of 'up's compared to 'down's);
- subs - pass down a new tree which is made by substituting the subtree in the first argument at its indicated node into the second argument at its indicated node.

For example if the operator, [up2 [top [rand1]]] were passed the following input (tree, node) pairs to act upon: ([B [A [A [B [2]] [1]] [B [3]]]], [1 1]), ([A [1] [2]], [2]) it would return the pair ([A [B [2]], [2])[4] – this process is illustrated below in figure 1.

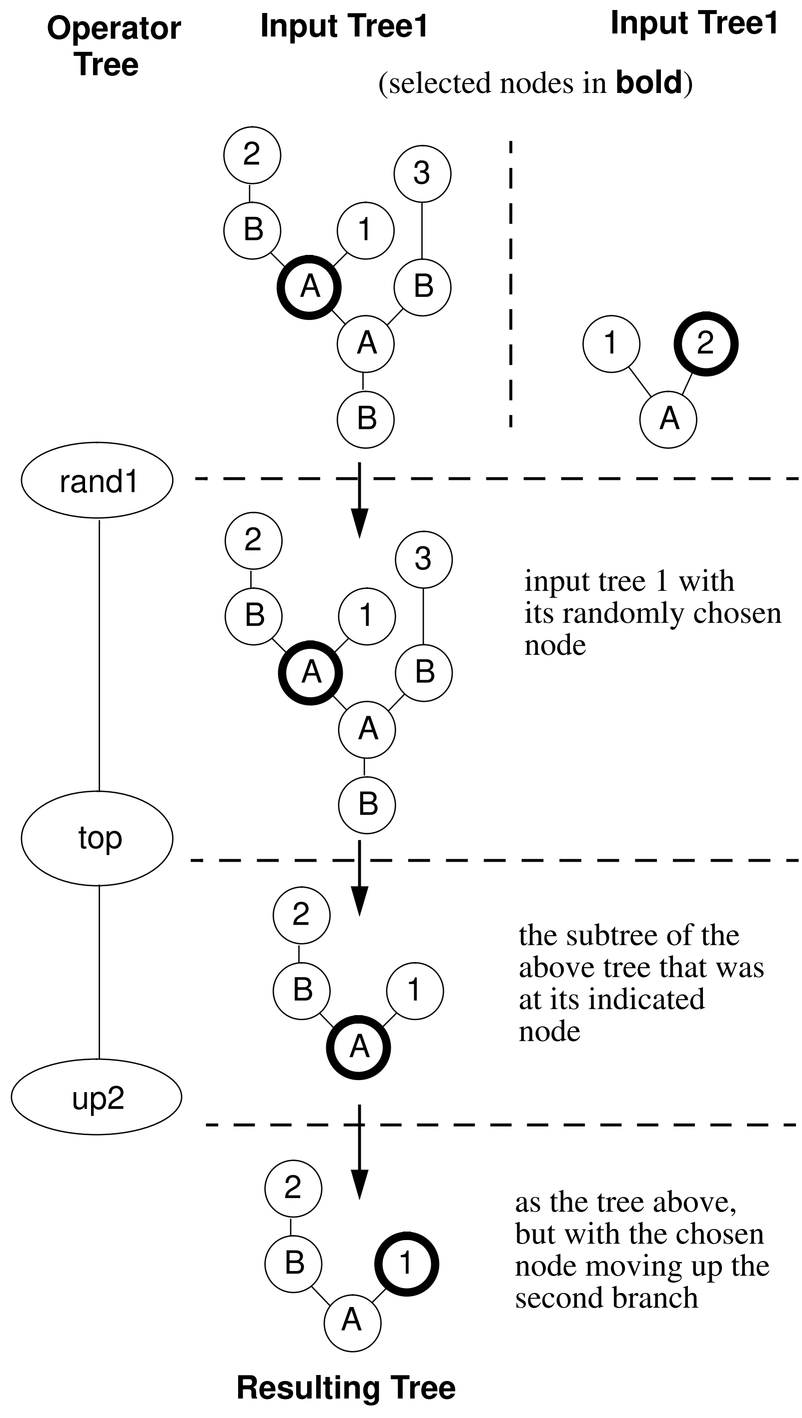

Figure 1: An example action of an operator on a pair of trees with selected nodes

4. Here node positions are indicated by a list, e.g. [1 2] would mean the node at the second branch of the first branch.

A second example is if the operator, [subs [bott2] [rand2]] were passed the same inputs it would return ([A [1] [A [1] [2]]], [2]) – as illustrated in figure 2.

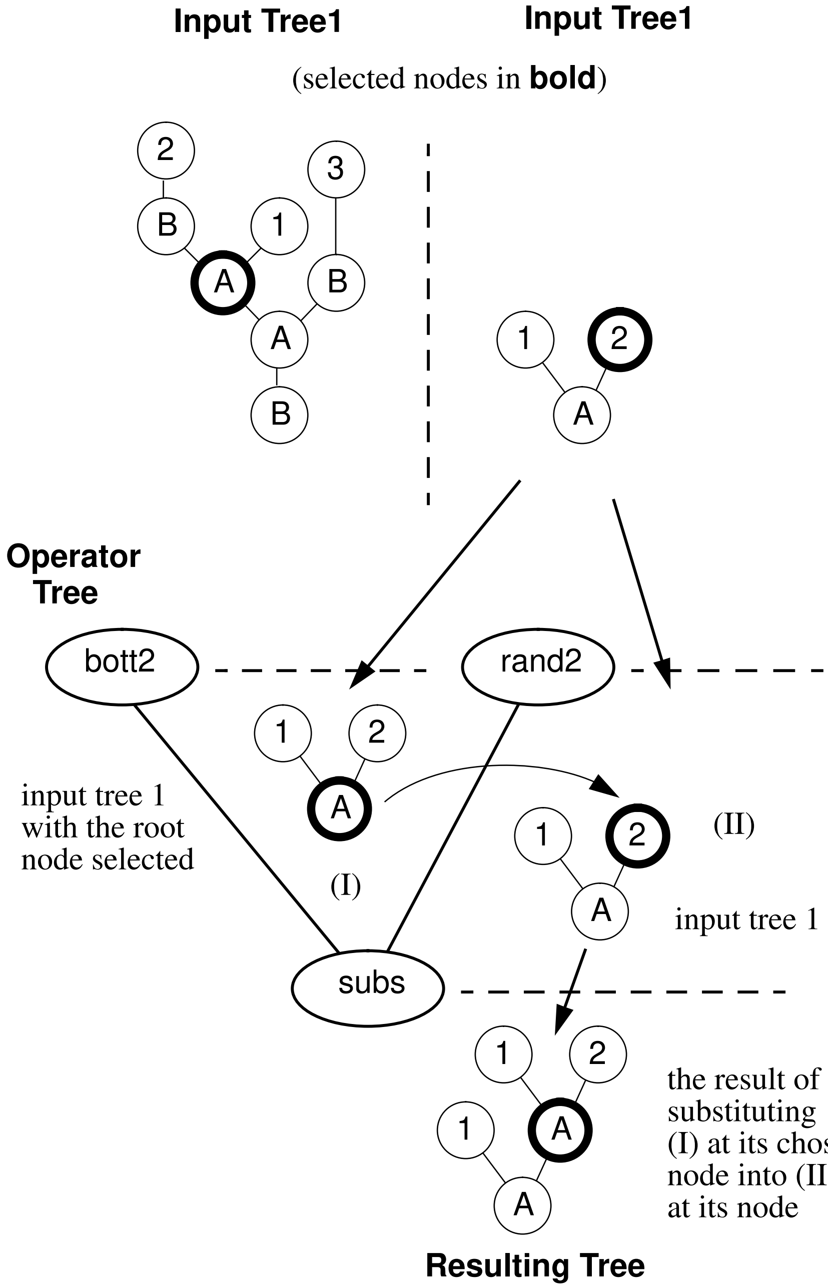

Figure 2: A second example of an operator acting on a pair of trees with selected nodes

Thus a cut&graft operator (one half of a traditional GP crossover) could be represented like this (figure 3).

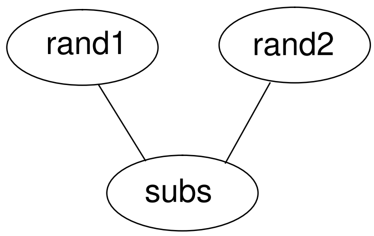

Figure 3: A cut and graft operator encoded as a tree

This takes the first random gene, cuts off the top at the randomly chosen position and then substitutes this for the subtree at the randomly chosen position of the second chosen gene. The operator composed of a single leaf [rand1] would represent simple propagation.

The population of operators is treated just like any other genetic population, being itself operated on by another population of operators. This sequence must eventually stop with a fixed non-evolving population of operators or a population that acts upon itself.

The base population and operator population(s) will have separate and (in general) different fitness functions - the one for the base population one determined by the problem at hand and the operator's fitness function by some measure of success in increasing the fitness of the population they operate on (although as we shall see later this usually needs to be modified).

3. Short-term Operator Success and Population Variation

Initial trials of the technique failed. It was discovered that this was due to the fact that the variation in the base population was destroyed too quickly. To investigate this an experiment was run where a population of 200 randomly generated operators of depth 5 (in the language described above), acted on a population of 10,000 base genes of depth 7 using nodes AND, OR and NOT and terminals input-1,..., input-3. The operators were applied to 50 pairs of input genes each, which were chosen with a probability proportional to their initial fitness (as in standard GP). The fitness of the base population was evaluated on the EVEN PARITY 3 problem before and after the application of the operators. The effect of the operators was evaluated by the average proportionate change in fitness of the operand genes before and after, by comparing the average of the input genes' fitnesses compared to the fitness of the resulting genes.

Not surprisingly, there was only a slight improvement in the population fitness (and marginally worse than what would have been achieved by fitness proportionate selection alone), but the variation of fitnesses and the average maximum depth of base genes significantly diminished, indicating a destruction of the variation in the base population. The overall statistics are shown in table 1, below.

base population	Before	After
Average Fitness	4.01	4.06
SD Fitness	0.64	0.58
Average Maximum Depth	7	6.15

Table 1: Overall Statistics of Experiment

The operators can be divided into those that used parts of both input trees to produce the result (binary operators) and those that, *in effect*, only used one input (unary operators). The binary operators increased the fitness of the population by significantly less than the unary ones. It was not that the unary operators were particularly good at *increasing* fitness, just that, on average, the binary operators were worse than average, significantly decreasing fitness- this is illustrated in figure 4 (operators that were strictly equivalent to simple propagation are excluded to aid the scaling - there were 59 of these – all unary with no effect of fitness, of course).

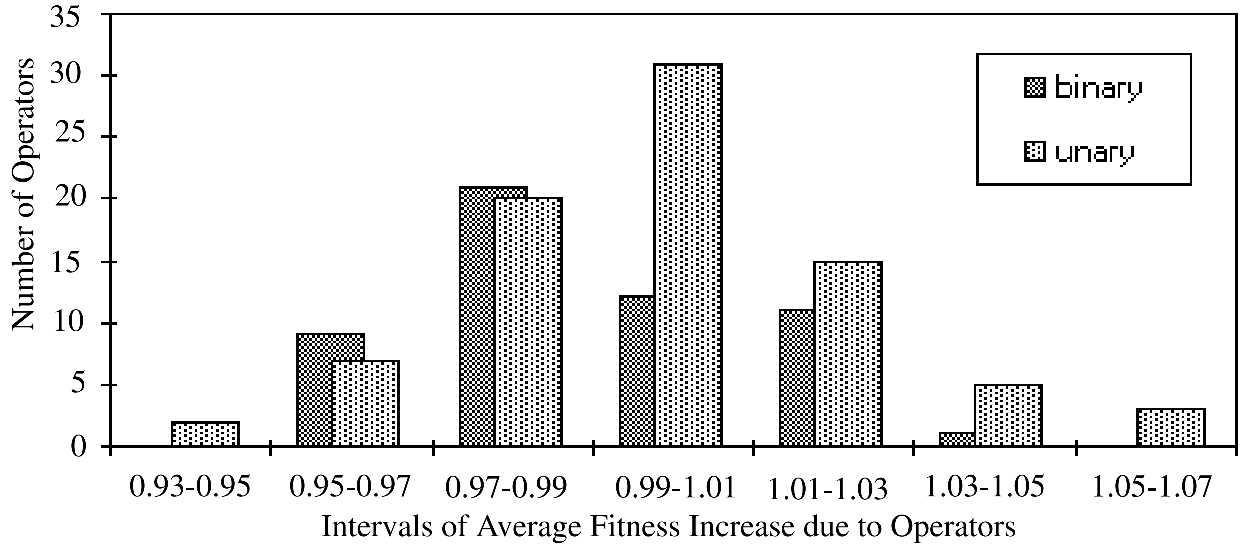

Figure 4: Comparison of Distributions of Fitness Change Caused by Operators

I analysed the operators into those with and without certain nodes and terminals, the results are shown in table 2, in terms of standard deviations of fitness changes from the overall mean fitness change.

Operator Type	Comparative Average Fitness Change in standard deviations from the mean
without rand1 or rand2	0.29
without down1 or down2	0.25
without subs	0.21
without up1 or up2	0.14
without top	0.05
with bott1 and bott2	0.02
with up1 and up2	-0.02
with top	-0.04
without bott1 and bott2	-0.04
with down1 and down2	-0.04
with rand1 and rand2	-0.16
with subs	-0.18

Table 2: Table of comparative effect in terms of average fitness change by operator type in the experiment

The presence (or absence) of substitution, random node choice (as opposed to starting at the root) and the lack of movement up or down, were most significant in affecting fitness change. This, however may be problem specific. It is not apparent from the above, but the operators which corresponded in their affect to a standard half of a crossover operator, did *less well* than the average in improving the fitness of the base population (on average they decreased the fitnesses by a factor of 0.98), but *were* good at preserving variety.

This indicates that short-term average increase in fitness may be less important than the occasional discovery of sharply better solutions while preserving variety. This is guessed at via the variation in fitnesses of used and resulting genes from the ground population. This is shown in table 3.

Operator Type	Average Change in Standard Deviation of Fitnesses of Base Population
without top	1.01
without rand1 or rand2	1.01
with bott1 and bott2	0.97
without up1 or up2	0.93
with down1 and down2	0.93
with subs	0.92
with up1 and up2	0.91
without subs	0.90
with top	0.84
with rand1 and rand2	0.84
without down1 or down2	0.81
without bott1 and bott2	0.78

Table 3: Effect on variation of fitness of types of operators in the experiment

Here we can see the significant effect of the presence or otherwise of top, rand and bott type nodes in an operator. It would seem that omitting either top or rand1 & rand2 will have the effect of preserving the variation in the population. I chose to omit both top, rand1 and rand2 as the above table suggested that the effect of this might be to maximise the increase in fitness but *without* the destruction of variety (as it indeed turned out). As we shall see this did have the effect of preserving the base population's variety while allowing the operator population to evolve.

4. Preliminary Test Results

The main practical questions to be resolved with respect to MGP are whether using this technique leads to improved results and whether such improvements are worth the additional computational cost. To do this a straight comparison of the number of evaluations of the base population is unfair as when using MGP one has to evaluate the operator population as well. Thus if one has a base population of 600 in both cases and an operator population of 200 in the second case execution time is increased by a factor of about 1.33. This is a bit rough and ready but seems to give a reasonably fair comparison reflecting computational times in practice.

I chose a problem that was deliberately straight-forward, but not *too* easy for simple GP algorithms: the odd 4-parity problem allowing nodes of AND, OR and NOT only, terminals for the four possible inputs: input-1,..., input-4, and using a base population size of 600. The fitness function was the number of correct outputs the gene produced (out of the total of 16), with a slight penalty (never greater than 1) for depth. The initial population was randomly generated with an even depth of 7.

The MGP algorithm had, in addition, a population of 200 operators using nodes up1, up2, down, down1 and subs with terminals of bott1 and bott2 (as described in section 2 above). This, in turn, was operated on by a population of 9 cut-and-graft operators (as illustrated in figure 3) and one simple propagation operator, all with fixed equal weights (as illustrated in figure 16). This population was also initially randomly generated with an even depth of 6. The fitness function for the operator population was found to be critical to the success of MGP. The basic fitness was the average proportionate change effected in the fitnesses of the base genes it operated upon. There were several elaborations to this basic function: *firstly*, the fitnesses were smoothed by a running average over the last 3 generations; *secondly*, new genes were accorded a fitness of 1; and *thirdly*, genes that had no effect on the fitness of genes (i.e. they were essentially propagation operators) were heavily discounted. It was necessary to do this otherwise these operators came to quickly dominate the population as most ‘active’ operators had a fitness of less than one, due tothe fact that *on average* they decreased fitness.

The MGP version had, in addition, one further modification that was necessary to make it run successfully. In the MGP algorithm fitness differentials were vastly magnified by a ranking mechanism, this was necessary in the operator population because of the extremely small differences in their fitnesses which was viable because otherwise MGP (with the above operator syntax) introduces a higher level of variation into the base population so a stronger selection force can be applied (in the GP algorithm a shifted fitness proportionate system was used as the GP performed worse under a ranking system).

Both algorithms were implemented in the high-level declarative language SDML[5], with some time critical sections implemented in VisualWorks/Smalltalk (which is SDML’s underlying language). The execution times are both for the same Sun 5 Sparcstation. In figure 5 the average fitness and the fitness of the best individual is shown vs. execution time; the dotted lines are plus and minus one standard deviation of the population fitnesses and the crosses indicate when the generations occurred. Note that for figure 6 the MGP algorithm does 800 evaluations per generation while the GP algorithm does only 600 in figure 8 (as discussed above at the start of this section).

5. For information on SDML see URL: http://www.cpm.mmu.ac.uk/sdml

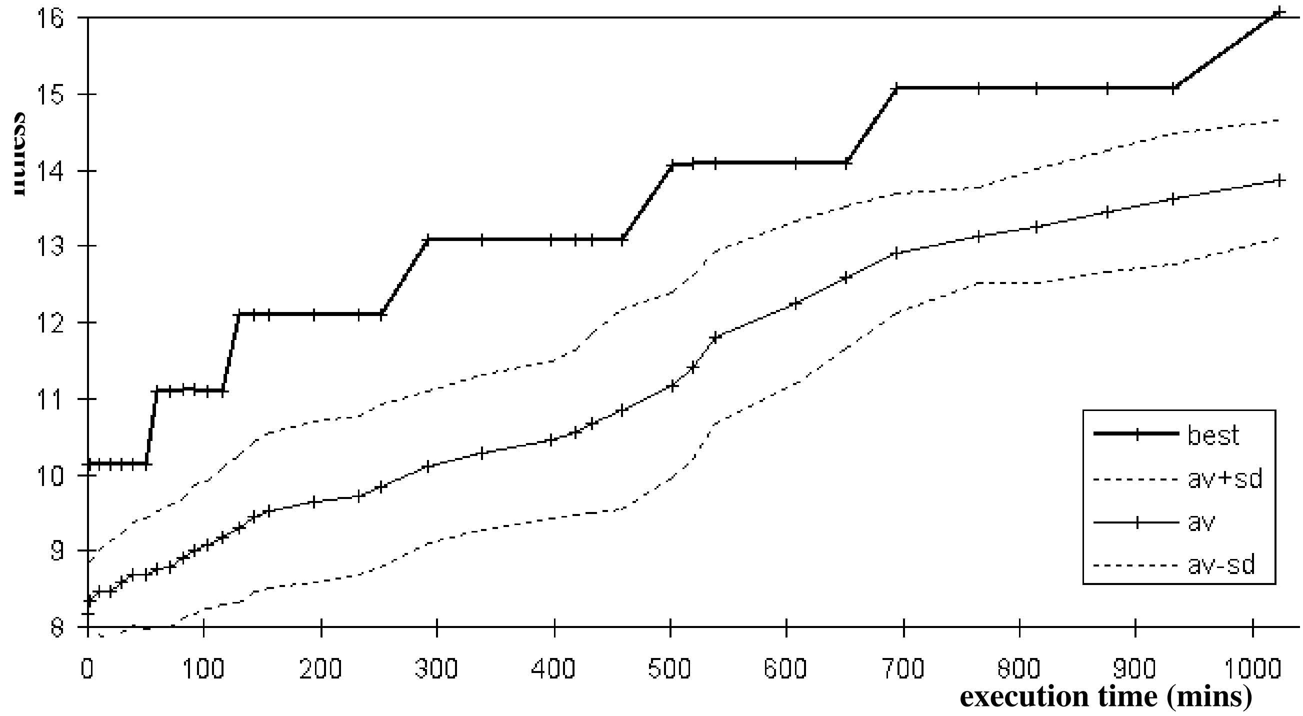

Figure 5: Best and average fitness vs. execution time for MGP run

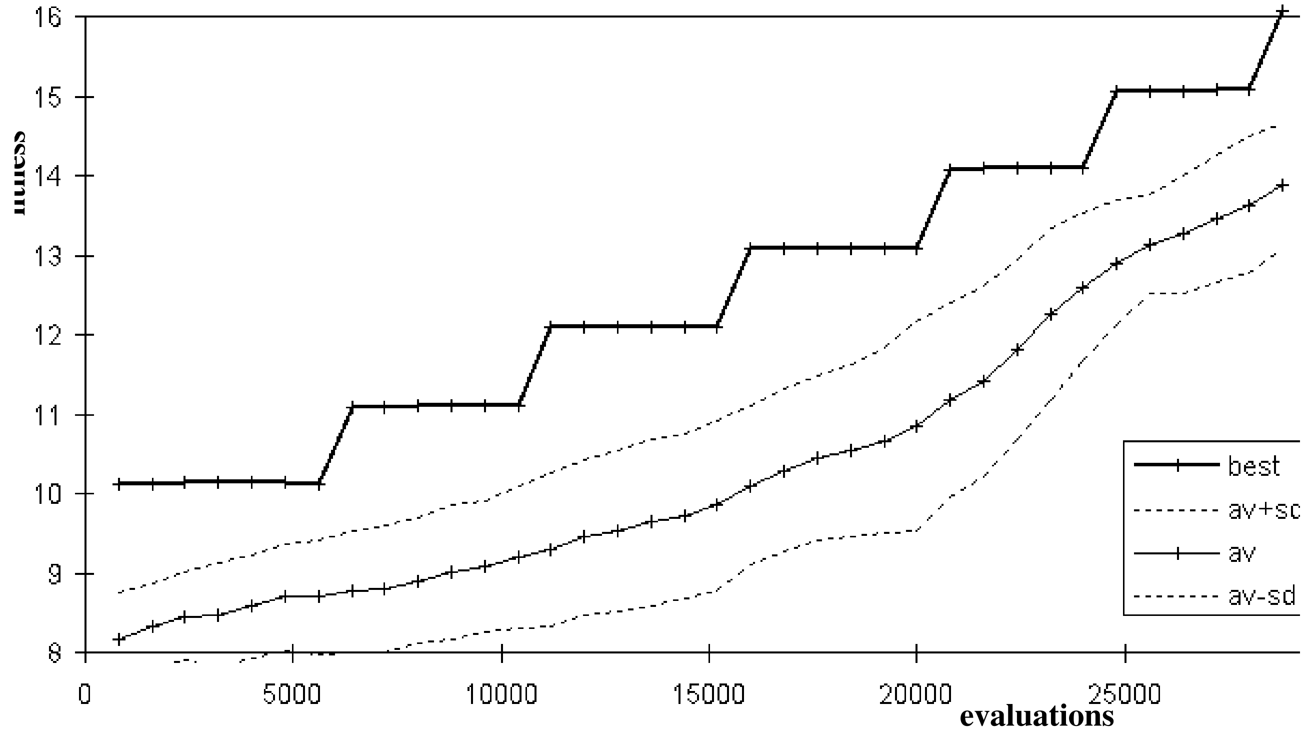

Figure 6: Best and average fitness vs. number of evaluations for MGP run

Thus in the best run of the MGP algorithm it found a perfect solution after 36 generations (28,800 evaluations) and 1050 minutes of execution time. Compare this to the results in figure 7 for the best GP run obtained. This found the perfect solution after 93 generations (55,800 evaluations) and 2380 minutes of execution time. The same number of runs were made of each (6). Although the best run in each case is shown, these results are typical.

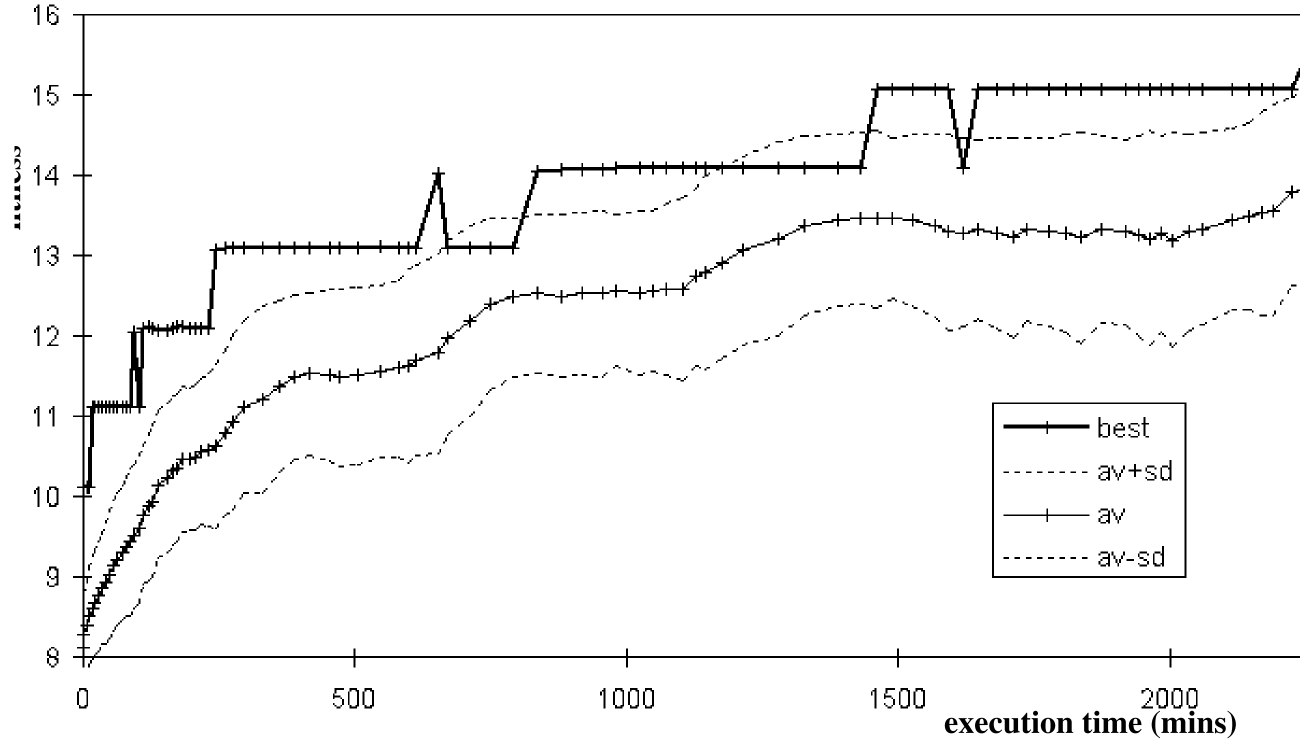

Figure 7: Best and average fitness vs. execution time for GP run

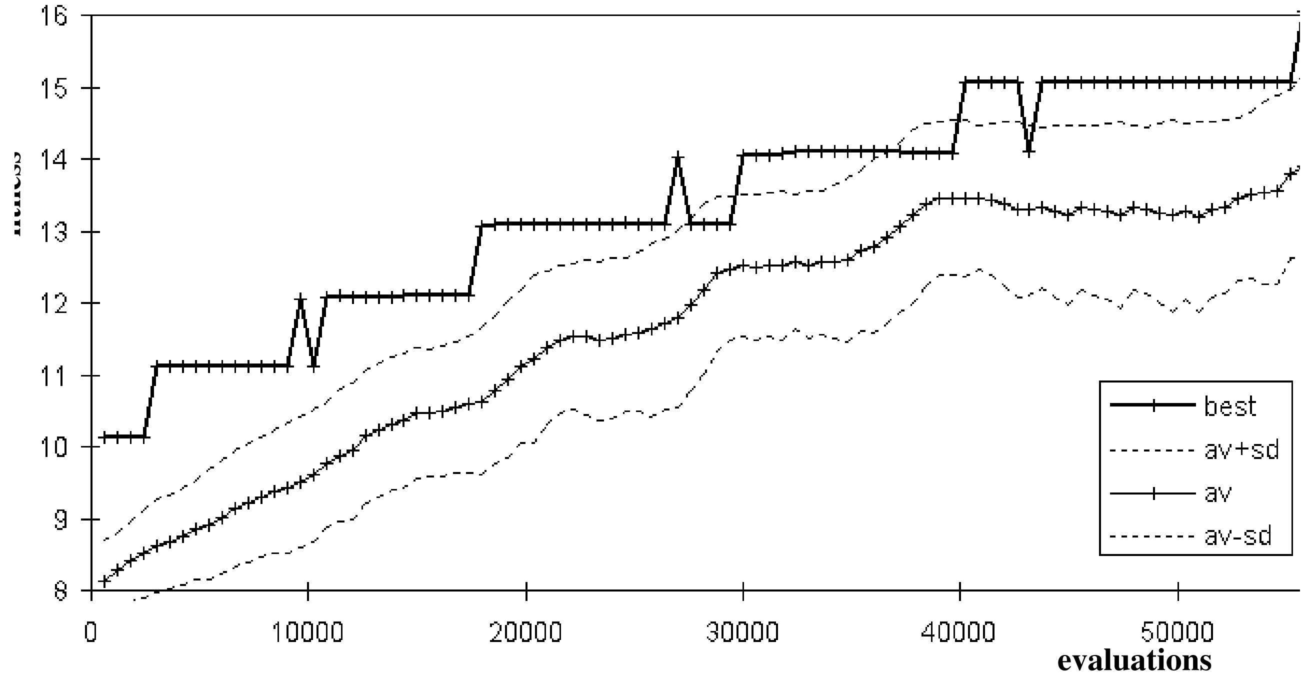

Figure 8: Best and average fitness vs. number of evaluations for GP run

There are several things to note. *Firstly*, that in the MGP algorithm the learning is smoother than in the GP algorithm - this may indicate that the MGP algorithm is acting as a more greedy but less robust search of the space of solutions. *Secondly*, that the MGP algorithm slows down a lot more quickly than the GP algorithm - this is due to the rapid inflation as the MGP algorithm learns to “grow” the genes in the base population using operators with multiple substitution nodes. Such inflation is almost inevitable due to the distribution of solutions of different depths for this problem, as shown in figure 9 - there are just many more better solutions accessible using a greater depth of candidate gene (this cause of bloating is further explored for a different problem in [11]). This is another advantage of GP, as it inflates the population more slowly. *Thirdly*, that while both techniques maintain a good level of variation in their base populations, the GP algorithm not only has a higher level of variation, but also increases the level of variation as it progresses (although it is possible that this is due to the fitness proportionate system of selection compared to the ranking system used in the MGP algorithm). Again this might indicate the greater robustness of standard GP over MGP.

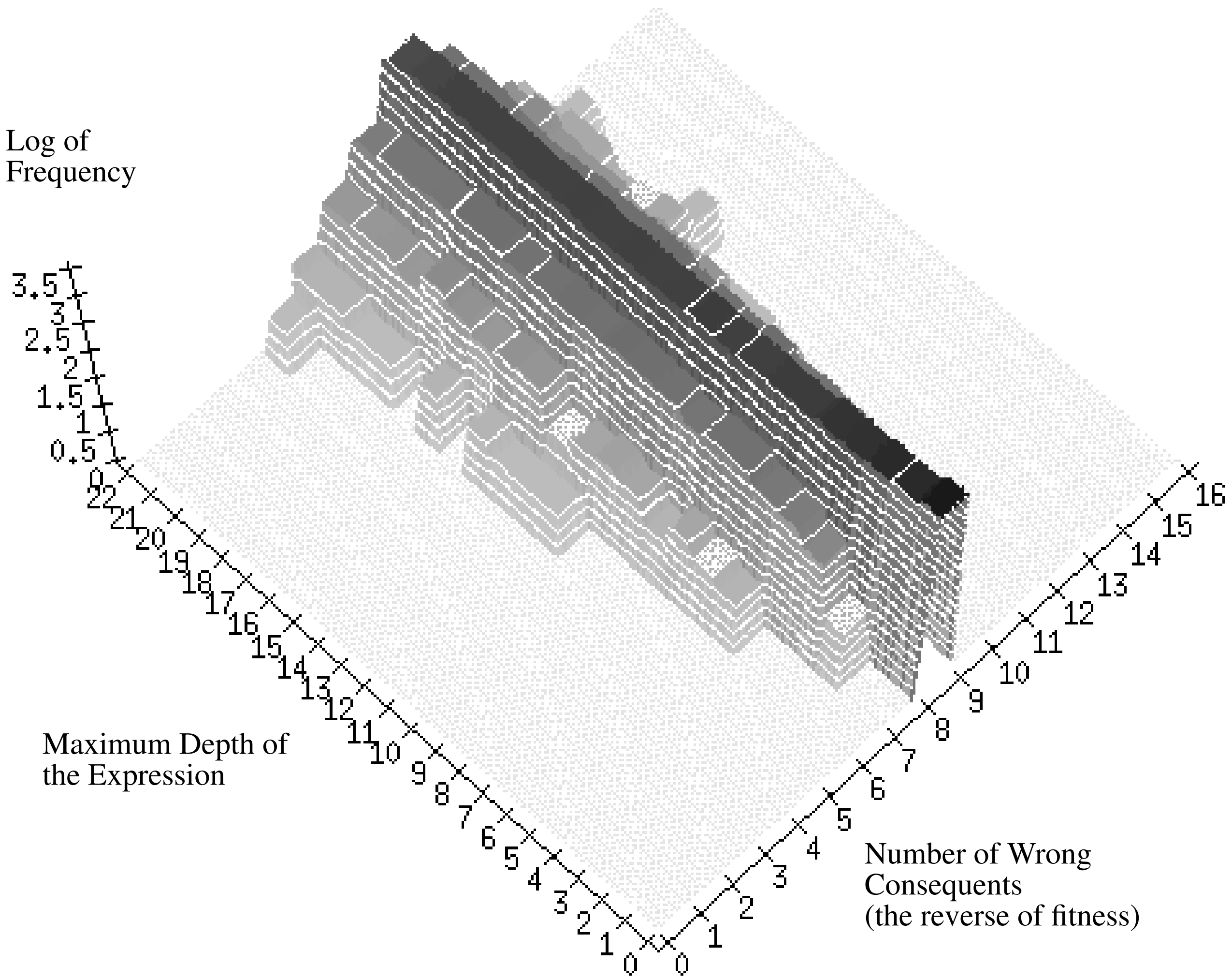

Figure 9: Distribution of 50,000 randomly sampled propositional expressions of different depths in four variables using AND, OR and NOT compared to the parity function

What may not be immediately obvious from the graphs above is the difference in the *shapes* of the curves. In figure 10 we compare the curves of best and average fitnesses on the same axis of execution time. Here we see that, although the GP algorithm did better at the start, both in terms of best and average fitness, the MGP algorithm did not seem to suffer from the characteristic flattening out that occurs in GP algorithms (at least, not to the same extent). It is perhaps this that indicates most clearly the potential of the MGP approach.

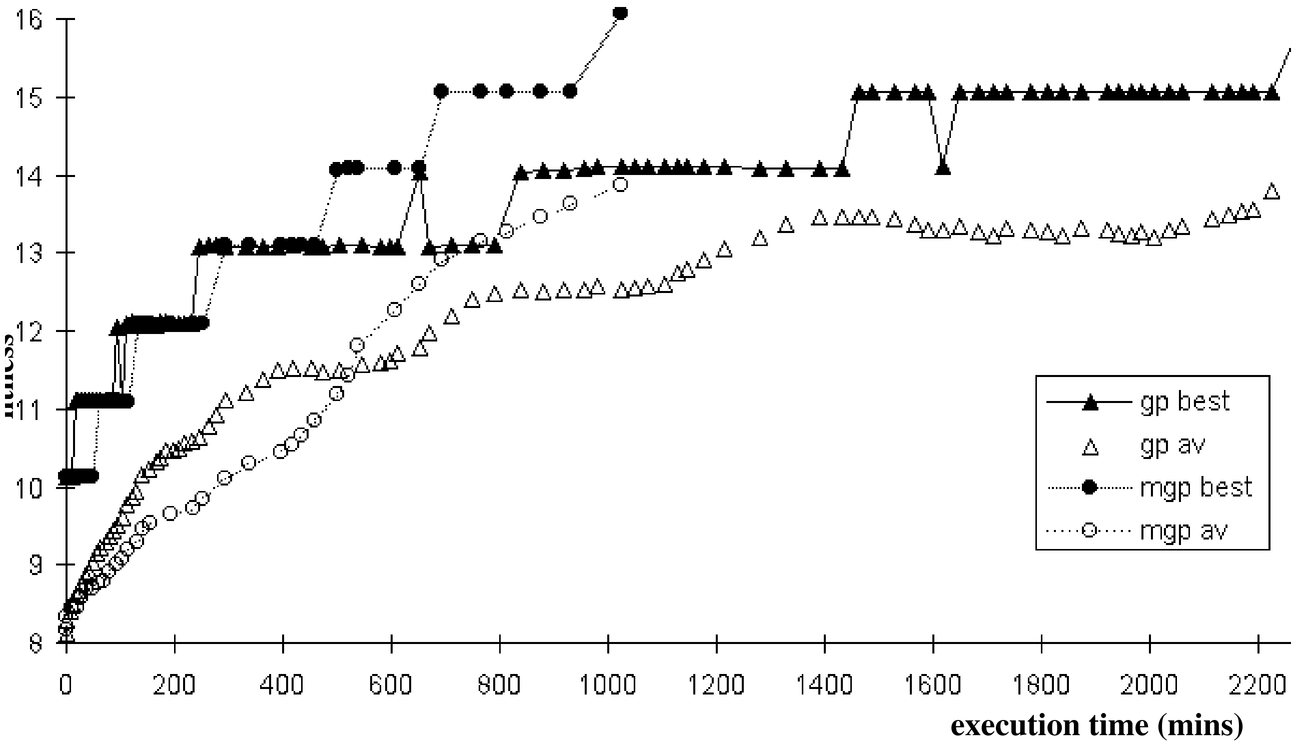

Figure 10: Comparing the MGP and GP runs

However, as mentioned above, the technique is susceptible to hyper-inflation as well as being somewhat brittle to other modifications. For example, if a ceiling is placed on the maximum depth of candidate solutions this can destroy the effectiveness of the MGP algorithm. In figure 11 and figure 12, I show the results of a run where a depth limit of 14 is imposed (where the rest of the set-up is the same as above). It is evident that as soon as the ceiling is reached (after about 40 generation) the algorithm ceases to be effective. This is probably because the depth ceiling disrupts the effective estimation of the operator's effectiveness, encoded in its fitness function.

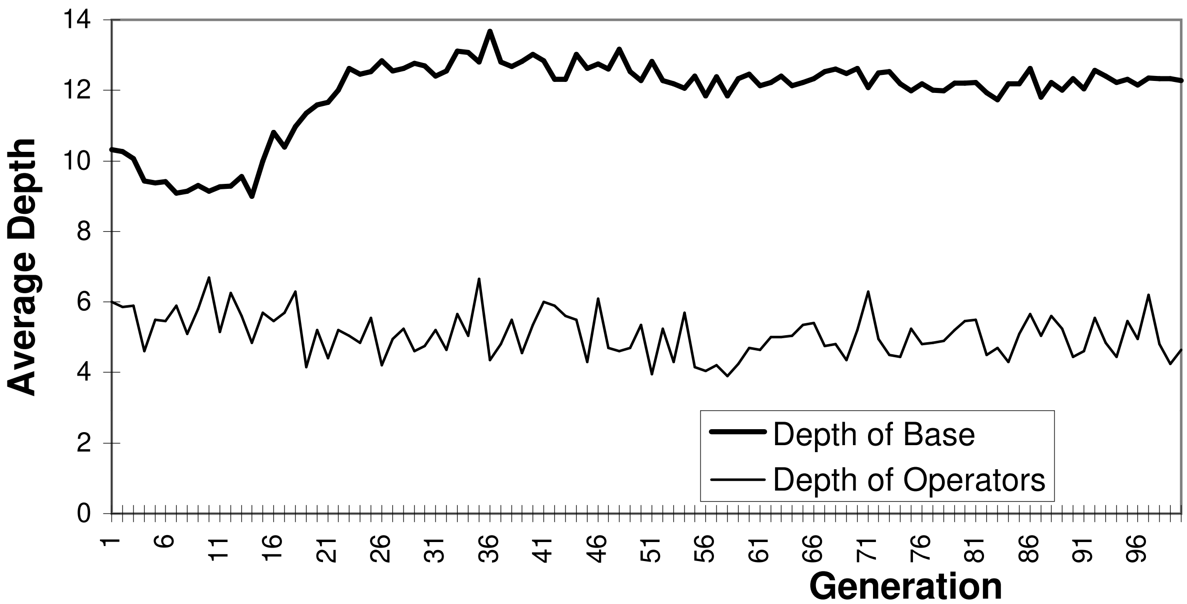

Figure 11: Average Depth in MGP run with a Depth Ceiling of 14

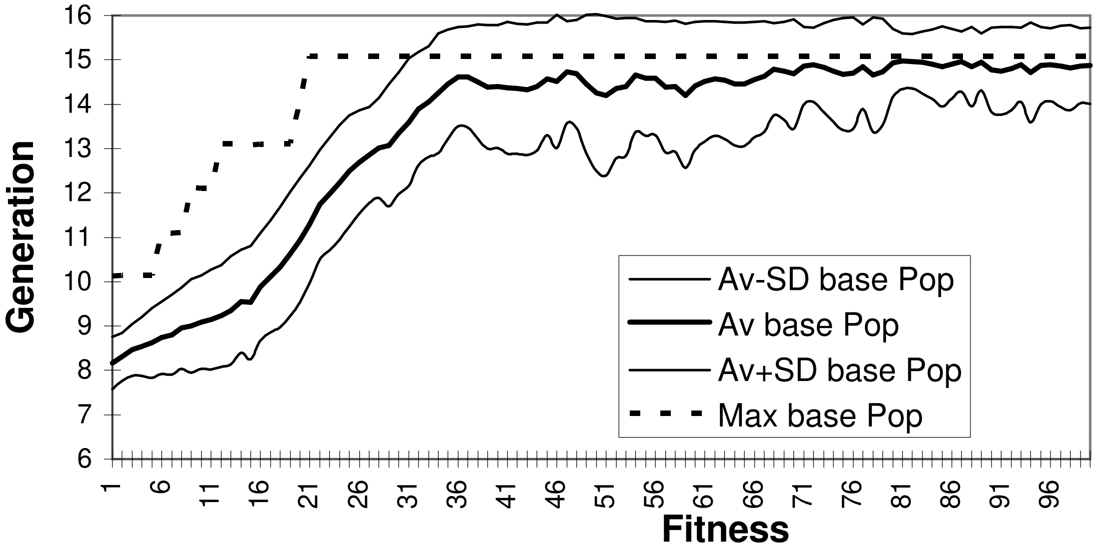

Figure 12: Average Depth in MGP run with a Depth Ceiling of 14

These preliminary explorations indicate a number of possibilities for improving the performance of the MGP algorithm:

- firstly a less greedy approach to gene selection than the ranking system might improve its robustness;
- secondly, a more sophisticated fitness function for operators might include such aspects as the variation it preserves (or introduces), the extent it inflates the base genes and whether it produces occasional sharp increases of fitness rather than merely an average improvement;
- and lastly, the introduction of new terminals into the operator tree syntax to allow mutation operators to be selected for. For example: newRand which would introduce a newly generated tree with a randomly chosen node or newBott which would introduce a newly generated tree but with the root node selected.

5. MGP as a tool for investigating evolutionary algorithms

As well as its possibility as a technique in itself, MGP can be used to investigate how different types of operators might do at different stages of an algorithm and with different problems. To illustrate this I analysed the prevalence of the terminals rand1, rand2, bott1, and bott2 in the operator population during an MGP run. The set-up was as described above but with the following differences: the operand trees were passed to the operator along with a randomly chosen *terminal* (as opposed to any node) – this was to heighten the difference between operating on the leaf-end as compared to the root-end of base trees, the problem was the odd parity-5 problem; and there was a depth ceiling of 18 imposed for the base population.

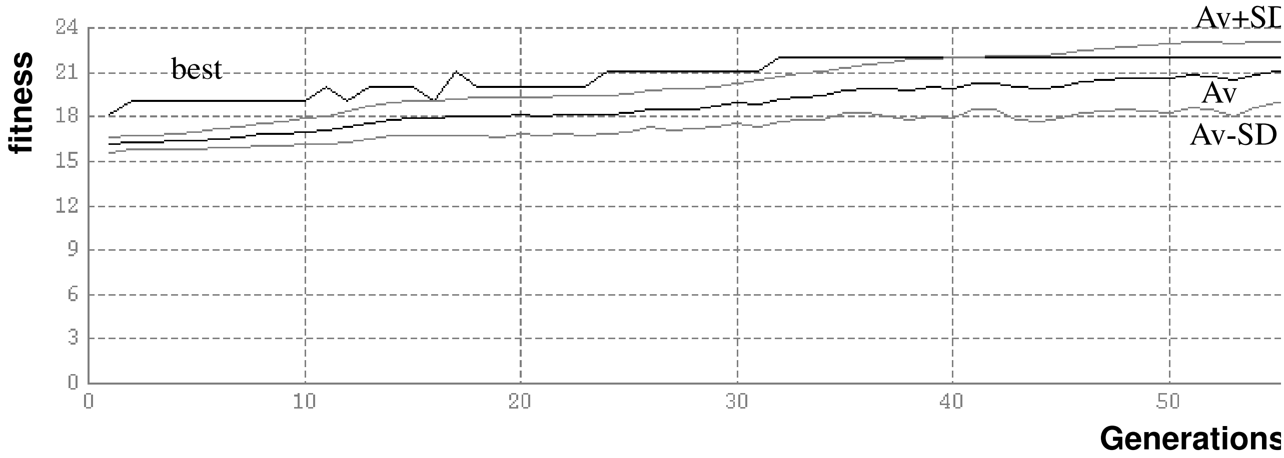

Figure 13: Best, Average+SD, Average and Average-SD of base population

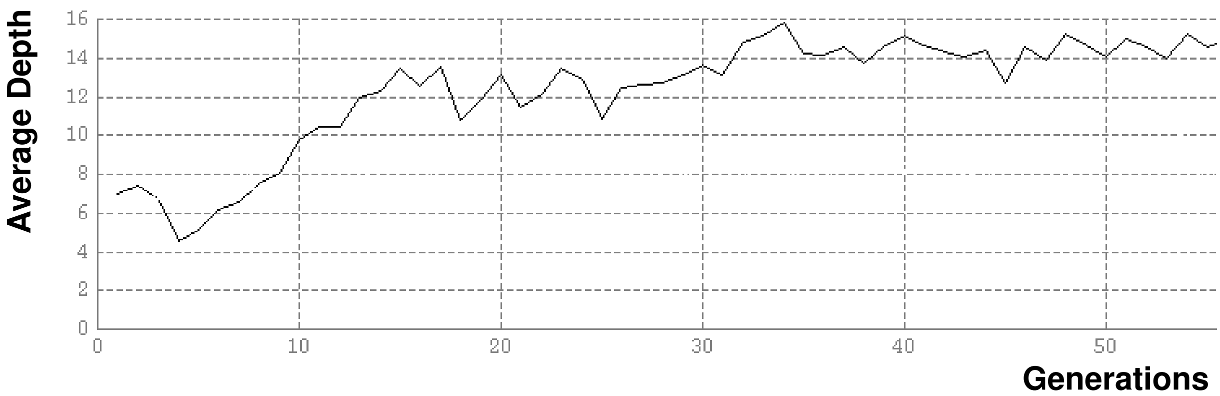

Figure 14: Average maximum depth of base population

As in the above example in section 4, the MGP algorithm ceased to be effective when the depth ceiling for the base population was reached (the operator population did not inflate noticabley). This occurred at around generation 35 as can be seen from figure 13 and figure 14. The prevalence of different terminals in the operator population is shown in figure 15. It is interesting to note how bott1 predominated during the initial period of fast growth and rand1 dropped down when the algorithm ceased to be effective.

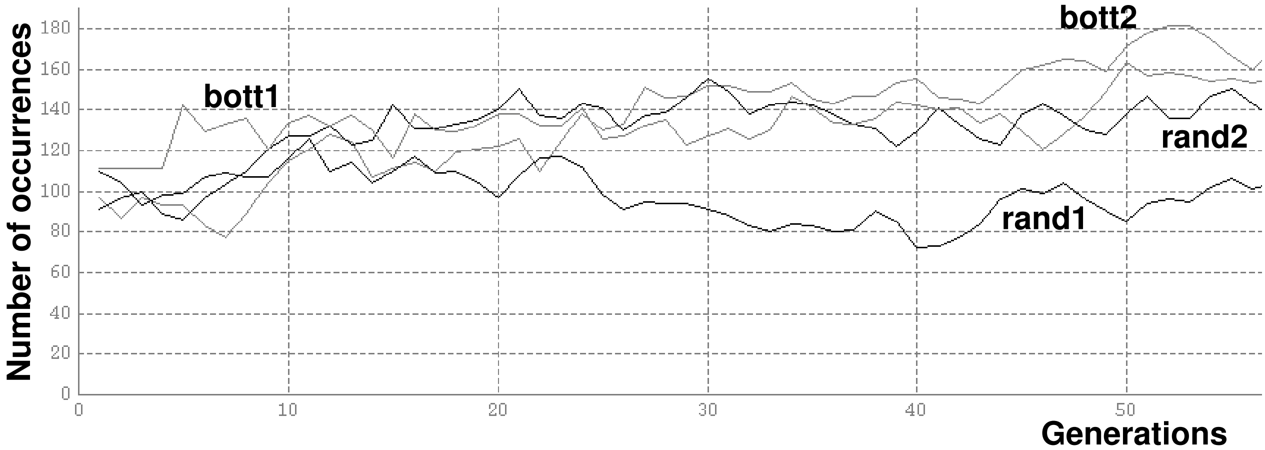

Figure 15: Prevalence of different terminals in operator population

6. MGP as a framework for a set of techniques

The relation of "population A acting as operators on population B" can be represented diagrammatically by an arrow. In figure 16, I illustrate traditional GP and basic MGP setups, with evolving population represented by ellipses and fixed populations by rectangles.

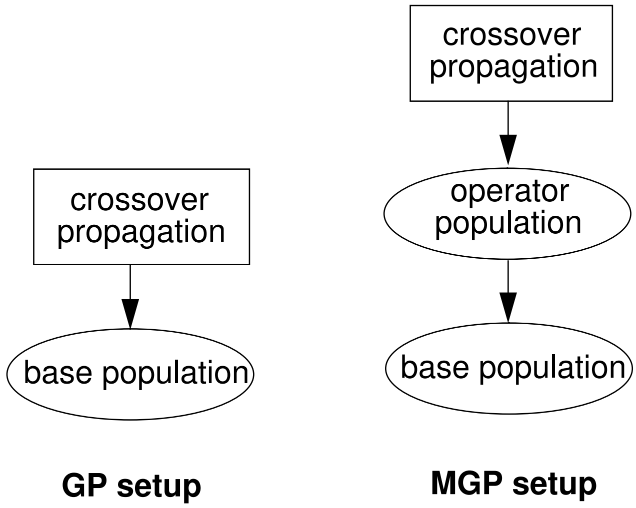

Figure 16: GP and MGP setups

There is no *a priori* reason why many different setups should not be investigated. For example figure 17 shows a simple enhancement of GP where just the proportions of the operators is dynamically changed, implementing a similar technique as that described for GAs in [4], and figure 18 illustrates the possibility of combining a fixed and an evolving population of operators, which might implement a combination of GP robustness and MGP serendipity

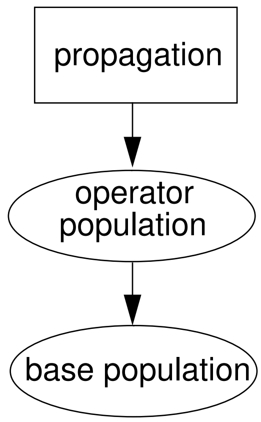

Figure 17: GP with the proportion of operators evolving

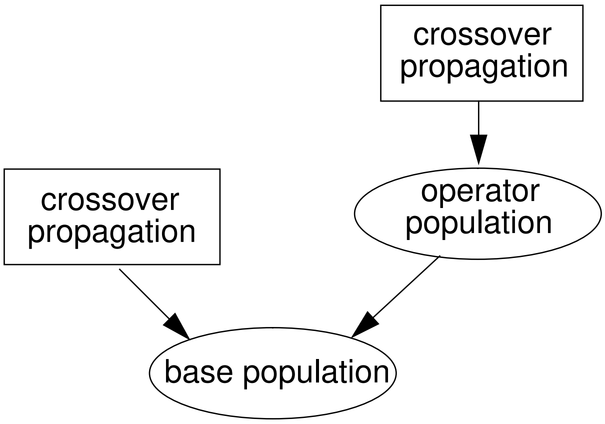

Figure 18: A setup with a mixture of fixed and evolving operators

Finally there is no reason why populations of operators should not act recursively, with populations of operators acting on themselves (as in figure 19) or mutually on each other. One can even imagine situations in which the interpretation (i.e. the fitness) of the base population and operator population was done consistently enough such that you only had *one* population acting on itself in a self-organising loop. In this case great care would need to be taken in the allocation of fitness values, maybe by some mechanism as the bucket-brigade algorithm [6] or similar. Such a structure may allow the implicit decision of the best structure and allow for previously unimaginable hybrid operator-base genes[6]. It does seem unlikely, however, that the same population of operators would be optimal for evolving the operators as well as the base population, due to their different functions.

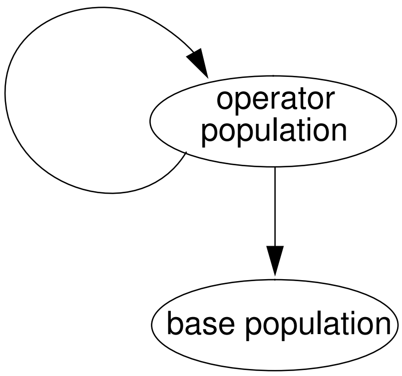

Figure 19: A recursively evolving MGP setup

Lastly, if the language of the operators described above could be extended with the addition of new nodes like “if-then-else”, “equal”, “and” and “root-is-implies”[7] (and some typing to include trees and boolean results), then operators like Modus Ponens (the logical rule that says from $A \rightarrow B$ and A you can infer B) could be encoded (see figure 20). This would mean that if acting on a population of logic like expressions (including a node for ‘implies’) then such an operator would produce child genes which represented logical inferences of those in previous generations. Such an operator would produce deductions on the genes. In a similar way one could allow for a techniques which allow different *mixes* of inferential and evolutionary learning techniques.

6. This is similar to a suggestion by Stuart Kauffman [8] for a computational model of enzymes that act upon each other and themselves
7. This would return ‘true’ if the root node of the passed tree were ‘implies’.

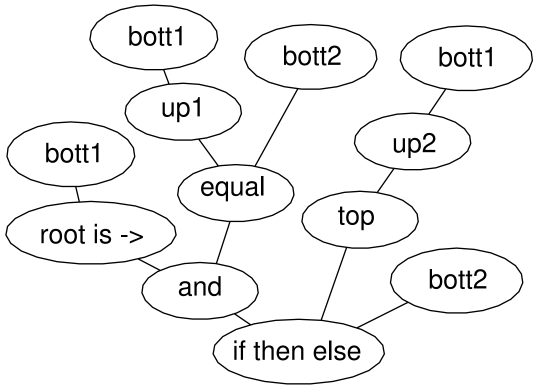

Figure 20: A Modus Ponens operator

In this way the differences between approaches become less marked, allowing a range of mixed learning approaches to tried including: deductive and inductive; GP and MGP. It also provides a rudimentary but common framework for them all to be compared.

7. Related Research

Most of the work done in investigating self-adapting evolutionary algorithms has fallen into one of three camps: those which add extra genetic material to the genome (this includes ADF [10] techniques as well as those such as [2]); those that build up a central library of modules and hence implicitly change the *language* of gene expression (e.g. [14]); and those which adapt a fixed set of parameters, e.g. the rate of mutation against crossover. The work descried here does something different in that the operators are adapted in form as well as propensity – in other words I have moved from co-evolving a fixed block of material to a variable length representation of the evolutionary set-up.

This is not the first time that co-evolving the genetic operators has been proposed. Teller [17] suggests the co-evolution of programs to perform operations of variation on other programs. In that example both the base and operator programs are networks. Part of the rationale for using evolved operators is that there is no natural candidate for a crossover operator and so if they were not evolved the would have had to have been especially designed for each task. Unfortunately this means that it is difficult to compare this with a standard GP algorithm. MGP is more straight-forward extension of the GP paradigm and hence may enable more opportunities for analysis. Furthermore because it is uses a straightforward GP algorithm on the operator population, some of the analysis and heuristics learnt about using GP algorithms can be brought to bear. However, MGP has not proved itself to the extent of Teller's techniques nor are they so readily applicable to network structures.

MGP can be compared to approaches in GAs, such as [15] which augment a standard GA with a series of crossover masks each of which has a weight associated, which changes according to their success, and which affects their later utilisation. In MGP, however, there are a potentially infinite number of such operators

8. Discussion

There are good reasons to suppose that there is no one technique, however clever, recursive, or self-organising that will be optimal for all problem domains [13] – the generality of a technique seems to mean that it tends to do relatively poorly in more specialised domains where domain-knowledge can be applied.

There also seems to be an implicit trade-off between the sophistication of a technique and its computational cost. Much of the power of GP comes from its low computational cost compared to its effectiveness, allowing large populations to successfully evolve solutions where more carefully directed algorithms have failed. Thus the conditions of application of MGP are very important - when it is useful to use and when not. I conjecture that techniques such as MGP might be helpful as a slightly greedy algorithm that is still applicable to difficult problems, but this (like the useful conditions of application of GP itself) is an open question.

A second advantage of GP is its robustness. MGP, through its nature, is likely to be far more brittle (as was illustrated by the various techniques needed to compensate for the inherent biases in the operator language). This is likely to be even more of a problem where more levels of populations acting on each other are involved or where populations act upon themselves.

Lastly MGP might give further insights into other GP algorithms by revealing what kinds of operator are successful at different stages of an algorithm and on different problem domains.

9. Further Research

There is obviously much more research to be done on this technique. In no particular order this includes:

- thorough trials of MGP over a variety of problem domains;
- the optimal tuning of MGP algorithms;
- the biases introduced by the language the operators are expressed in;
- a comparison of MGP and ADF/ADM techniques;
- the merits of combining MGP and ADF/ADM techniques;
- the effectiveness of recursive genetic programming (as illustrated in figure 19) where an evolving population of operators act upon itself.

Acknowledgements

Thanks to Steve Wallis, David Corne and William Langdon for their comments on an early draft of this paper.

SDML has been developed in VisualWorks 2.5.1, the Smalltalk-80 environment produced by ParcPlace-Digitalk. Free distribution of SDML for use in academic research is made possible by the sponsorship of ParcPlace-Digitalk (UK) Ltd.

References

[1] Angeline, P. J. (1995) Adaptive and Self-Adaptive Evolutionary Computations, In M. Palaniswami, et. al. (eds.), *Computational Intelligence: A Dynamic Systems Perspective*, Piscataway, NJ: IEEE Press, pp 152-163.

[2] Angeline, P. (1996). Two Self-adaptive Crossover Operators for Genetic Programming. In Angeline, P. and Kinnear, K. E. (ed.), *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, 89-100.

[3] Angeline, P. (1997). Comparing Subtree Crossover with Macromutation. *Lecture Notes in Computer Science*, 1213:101-111.

[4] Fogarty, T.C. (1989). Varying the probability of mutation in the genetic algorithm. In Schaffer, J. (ed.), Proceedings of the Third INternational Conference on Genetic Algorithms, Morgan Kaufmann, 104-109.

[5] Fogel, D.B., Fogel, L.J. and Atmar, J.W. (1991). Meta-Evolutionary Programming. In Chen, R. (ed.), *Proceedings of the 25th Aslimar Conference on Signals, Systems and Computers*, Maple Press, San jose, CA, 540-545.

[6] Holland, J. H. (1985). Properties of the bucket brigade. In Grefenstette, J. J. (ed.), *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, Lawrence Erlbaum Associates, 1-7.

[7] Iba, H. and de Garis, H. (1996). Extending Genetic Programming with Recombinative Guidance. In Angeline, P. and Kinnear, K. E. (ed.), *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, 69-88.

[8] Kauffman, S. A. (1996). *At Home in the Universe: the search for laws of complexity.* Penguin, London.

[9] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, MA.

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

[11] Langdon, W. B. (1997). Fitness Causes Bloat. WSC2 - 2nd On-Line World Conference on Soft Computing in Engineering Design and Manufacturing, June 1997. Proceedings to be published by Springer-Verlag.

[12] Montana, D.J. (1995). Strongly-typed Genetic Programming. *Evolutionary Computation*, 3:199-230.

[13] Radcliffe, N.J. and Surry, P.D. (1995). Fundamental Limitations on Search Algorithms - Evolutionary Computing in Perspective. *Lecture Notes in Computer Science*, 1000, 275-291.

[14] Rosca, J. R. and Ballard, D. H. (1996). Discovery of Subroutines in Genetic PRogramming. In Angeline, P. and Kinnear, K. E. (ed.), *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, 177-201.

[15] Sebag, M. and Schoenauer, M. (1994). Controlling crossover through inductive learning. In Davidor, Y. (ed.), *Proceedings of the 3rd Conference on Parallel Problem-solving from Nature*, Springer-Verlag, Berlin, 209-218.

[16] Smith, J.E. and Fogarty, T.C. (1997). Operator and Parameter Adaption in Genetic Algorithms. *Soft Computing*, 1:81-87.

[17] Teller, A. (1996). Evolving Programmers: The Co-evolution of Intelligent Recombination Operators. In Angeline, P. and Kinnear, K. E. (ed.), *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, 45-68.