Multiobjective Parsimony Enforcement for Superior Generalisation Performance

Yaniv Bernstein, Xiaodong Li, Vic Ciesielski, and Andy Song
School of Computer Science and Information Technology
RMIT University, Melbourne VIC 3001, Australia
Email: {yberstein,xiaodong,vc,asong}@cs.rmit.edu.au

***Abstract*— Program Bloat - the phenomenon of ever-increasing program size during a GP run - is a recognised and widespread problem. Traditional techniques to combat program bloat are program size limitations or parsimony pressure (penalty functions). These techniques suffer from a number of problems, in particular their reliance on parameters whose optimal values it is difficult to *a priori* determine. In this paper we introduce POPE-GP, a system that makes use of the NSGA-II multiobjective evolutionary algorithm as an alternative, parameter-free technique for eliminating program bloat. We test it on a classification problem and find that while vastly reducing program size, it does improve generalisation performance.**

I. Introduction

It is often the case in GP runs that the average size of individuals in the population increases substantially over the course of the run. This phenomenon is known as program bloat. Bloated populations lead to greater load on both CPU and memory over the course of a run, decreased search effectiveness and difficulty in interpreting results. Furthermore, in applications where programs are desired to generalise from a limited training set (and there are many; symbolic regression, machine learning and classification are just a few) it has been observed that smaller solutions tend to generalise better [15], [9].

In this paper we study the performance of POPE-GP, a new algorithm that uses the NSGA-II multiobjective algorithm as the basis for parsimony enforcement. The use of multiobjective techniques for parsimony enforcement has been studied by Bleuler et al. [2] and De Jong et al.[4] and proved extremely successful in producing highly parsimonious solutions that exhibit good performance on the parity problem. Our focus in this paper is somewhat different: we would like to further test the hypothesis that small solutions generalise better than large solutions. To this end, we compare the performance of POPE-GP on a real-world classification problem with that of a GP with more traditional parsimony control and a GP with no control at all, paying particular attention to the performance of solutions on the unseen testing set as a measure of generalisation performance.

Our results not only reconfirm that the multiobjective approach is an excellent way of enforcing parsimony in a population without adversely impacting on fitness, but do also demonstrate an increase in the generalisation ability of the programs generated by POPE-GP.

II. Program Bloat in GPs

There are a number of theories in existence that attempt to explain the causes of bloat; these include *protection against crossover* [13], *removal bias* [14] and *diffusion*[10]. While the exact mechanisms and behaviours differ between the theories, they all echo one fundamental truth: that it is easier to add code to a program than it is to remove it. That is, it is quite difficult to remove code from an effectively functioning program without heavily impacting on that functionality and thus reducing its fitness. On the other hand, adding code to such a program is much less likely to be harmful. As such, there is an inherent bias towards the expansion of code and against its removal.

Code bloat is associated with a number of harmful outcomes:

- **Increased Resource Usage**: Larger programs consume more memory and take longer to evaluate.
- **Decreased Search Effectiveness**: Program bloat interferes with the efficacy of the crossover and mutation operators.
- **Obfuscation**: The operation of a bloated program is usually so opaque that it is nearly impossible for humans to understand its basic functionality.
- **Overfi tting**: This can be classified as both a possible cause and possible effect of bloat. Overfitting occurs when optimisation pressure causes the solution to adhere too closely to the training set, causing degradation in the generalised performance of the solution. An overfitted solution tends to be large because of the complexity of adhering precisely to the often noisy data presented to the learning algorithm. Thus, it is possible that overfitting contributes to program bloat, and conversely that tackling program bloat discourages overfitting and improves a solution's generalisation performance. Tackett [15], Kinnear [9] and Zhang and Mühlenbein [16] all report that smaller solutions tended to have superior generalisation performance in their experiments.

III. Existing Parsimony Enforcement Methods

A. Tree Limitation

One of the simplest ways to manage program bloat is to impose a static maximum upon the depth or size (number of nodes) of program trees within the population. If a new individual is generated (for example by crossover) which breaches these limitations, it is simply rejected. This technique is certainly effective in stopping individuals from reaching an unmanagable size, but is unsatisfactory for a number of reasons.

One problem is that it is very difficult to reliably choose a good value for the limit. If a depth limit is made too shallow, there is a risk that good solutions will never be generated because their depth exceeds the limit. Conversely, if the limit is too deep, then program bloat will remain an appreciable problem. As an appropriate depth can rarely be known *a priori*, one is faced with a choice between stunting the generation of good solutions and allowing a substantial degree of bloat.

Another issue is that the process of culling nonconforming individuals actually creates a bias against certain types of information transference within the population. The consequences of this are difficult to determine and are problem dependent but can be adverse in some cases [8].

B. Constant Parsimony Pressure

Constant Parsimony Pressure applies a penalty function to an individual based upon its size. The intuition is that by degrading the fitness of large (and thus possibly bloated) individuals, significant pressure is applied towards brevity within the population. The user defines the value of a parameter α which determines to what degree programs are penalised for their size. A higher value of α results in greater pressure towards parsimony — solutions are more heavily punished for their bulk the higher α is. The selection of an appropriate value for α is critical as it has a significant impact on the search bias of the algorithm.

Note that constant parsimony pressure cannot distinguish between a solution that is bloated and one that is fundamentally large and cannot be represented more compactly. If the value of α is too high, it is quite possible that fit, complex solutions will be rejected in favour of less fit but very simple solutions. Conversely, if α is too low, the parsimony pressure will be weak and bloat can proceed largely unhindered. The problem is that it is impossible to know the ideal value for α *a priori*.

C. Adaptive Parsimony Pressure

Adaptive Parsimony Pressure [16] changes the amount of pressure exerted towards parsimony based upon the circumstances of the population. The pressure parameter α is no longer static but rather a function $\alpha(g)$ that adapts at each generation. There is one required parameter — ϵ, a user-specified error tolerance. While the error of an individual remains worse than ϵ, the parsimony pressure remains low and the main pressure is towards improving performance. When the error is within the specified tolerance, the parsimony pressure becomes far stronger and the main evolutionary pressure is towards the individual reducing its size.

Zhang and Mühlenbein report good results for their technique [16]. Compared to a GP with no growth controls, adaptive parsimony pressure took less time to train and produced substantially smaller individuals with significantly better generalisation performance. Blickle [3] reports mixed results with his experiments on adaptive parsimony pressure, finding it generated small trees but that they had relatively low fitness. Bleuler et al.[2] find that adaptive parsimony pressure is inferior to constant parsimony pressure on the even-parity problem.

IV. Multiobjective Optimisation for Combating Bloat

A. Overview of Multiobjective Optimisation

Many real world problems require optimisation over a number of distinct and often contradictory objectives simultaneously [5]. For example, a structural support may need to be light *and* strong, or we may wish for a computer to be both fast *and* cheap. When there are multiple objectives, it is in most cases no longer possible to have just one optimal solution. For example, which is superior: a very cheap, slow computer; or an expensive, fast computer?

Definition 1: The dominance relation $>_d$ between two solutions i and j, $j >_d i$ holds over the set of objectives Θ if
$\forall\theta \in \Theta(f_\theta(j) \geq f_\theta(i)) \wedge \exists\theta \in \Theta(f_\theta(j) > f_\theta(i))$
where f_θ is the fitness of a solution under objective θ and all objectives are maximisation objectives.

Definition 2: A solution i is said to be nondominated in a solution set $\mathcal{P}$ if
$\neg\exists j \in \mathcal{P}(j >_d i)$.

Definition 3: A solution i is set to be a member of the Pareto Front of a problem if it is nondominated in the set of all possible solutions $\mathcal{S}$.

The goal in multiobjective optimisation is to discover the Pareto Front. Traditional single-objective optimisation techniques are limited in their ability to solve this sort of problem, because they focus on reaching a single globally optimal point. By contrast, multiobjective optimisation algorithms such as NSGA-II [6] and SPEA2 [17] are specifically designed to seek out a set of nondominated solutions as close as possible to the true Pareto Front. They use a variety of different techniques to do so but all algorithms emphasise the gathering of a diverse collection of nondominated individuals rather than a single outstanding solution.

If we consider parsimony as an additional (pseudo) objective during a GP run, and hence GP as a multiobjective problem, the problem with parsimony pressure techniques becomes clear. We are attempting to combine two different objectives — fitness and size — into a single fitness value. This does not capture the full flavour of what is desired: that individuals do not gain in size without a corresponding increase in fitness.

In other words, we would like individuals to belong to the fitness-size Pareto Front. Once it is recognised that this Pareto front is in fact what we are seeking, multiobjective optimisation algorithms present themselves as an obvious choice for parsimony enforcement.

Bleuler et al.[2] and De Jong et al. [4] have tried this approach of using multiobjective algorithms for reducing bloat before. In both cases a multiobjective algorithm was used to attempt to control bloat and produce superior results on the parity problem. The results were excellent; in particular, tree sizes remained very small compared to all other methods including parsimony pressure. Furthermore, the multiobjective algorithms found better solutions with less computational effort.

V. The POPE-GP Algorithm

The Pseudo-Objective Parsimony Enforcement GP (POPE-GP) uses the NSGA-II multiobjective optimisation algorithm [6] as a base for its operation. The two objectives are defined as being the actual objective of the GP run (the fitness) and the size of the program. Once these objectives have been defined, the NSGA-II algorithm attempts to find the Pareto Front for these two objectives. The operation of the NSGA-II algorithm is described in the following section.

A. The NSGA-II Multiobjective Algorithm

Before we describe the mechanincs of the NSGA-II algorithm, we must define a number of terms.

Definition 4: The **first nondominated front** $\mathcal{N}_1$ of a population $\mathcal{P}$ is the set of individuals that are nondominated in that population.

Using the above definition, we can then recursively define all further nondominated fronts as follows:

Definition 5: The $\boldsymbol{n^{th}}$ **nondominated front** $\mathcal{N}_n$ of a population $\mathcal{P}$ is the first nondominated front of the population $\mathcal{P}'_n = \mathcal{P} - \bigcup_{i=1}^{n-1} \mathcal{N}_i$.

In other words, the n^{th} nondominated front of the population is the first nondominated front of the remaining population when the first $n-1$ nondominated fronts are removed. Thus, the second nondominated front consists of all individuals in the population dominated only by individuals in the first nondominated front, the third nondominated front consists of all individuals dominated only by individuals in the first *and* second fronts, and so on.

We also for convenience define a function $N(i)$ which returns q, the number of the nondominated front of which an individual i is a member:

Definition 6: $N(i) = q$ iff $i \in \mathcal{N}_q$.

The final quantity we need to define is the crowding distance $C(i)$:

Definition 7: The crowding distance $C(i)$ of a particular individual i is equal to the sum of the distance between i's nearest neighbours to either side on its nondominated front for all objectives. If i does not have a neighbour on one side (*ie.* it is on the edge of the front) then its crowding distance is deemed to be infinite.

Using the above definitions we are now able to define the crowded-comparison operator $\prec_n$, which lies at the heart of the operation of the NSGA-II algorithm:

Definition 8: For two individuals i and j we say that $i \prec_n j$ **iff**

- $N(i) < N(j)$, **or**
- $N(i) = N(j)$ *and* $C(i) > C(j)$.

At each generation n the parent population p_n is sorted into nondominated fronts[1] and the crowding distance calculated for each individual. A child population c_n is then created using tournament selection and the user's choice of genetic operators. The tournament selection functions by using the crowded comparison operator $\prec_n$ rather than the usual fitness function. This ensures that solutions on a higher nondominated front are favoured, and within each front, individuals that are less crowded. This creates the necessary pressure for the population to move towards the Pareto Front and to disperse along it.

After the child population is created, the two populations are merged and sorted once again into nondominated order and the crowding distance for each individual once again calculated. The parent population for the next generation p_{n+1} is then created by taking the top half of the sorted, merged population. See Figure 1 for a schematic of the process. Note that individuals within a nondominated front are sorted by decreasing crowding distance. Thus, if a front is split when the new parent population is selected, it is the individuals with higher crowding distance that will be selected. Note that NSGA-II is fully elitist; the next generation is selected from a combination of the parent and child populations. As such, there is no possibility of losing a high quality solution.

VI. Empirical Study: Classification

We compared the generalisation performance of classifier programs generated by the POPE-GP algorithm with those generated by a standard GP with a depth limit of eight and one with no limits at all. We used the Wisconsin Breast

[1]For a description of a fast nondominated sort procedure, see [6].

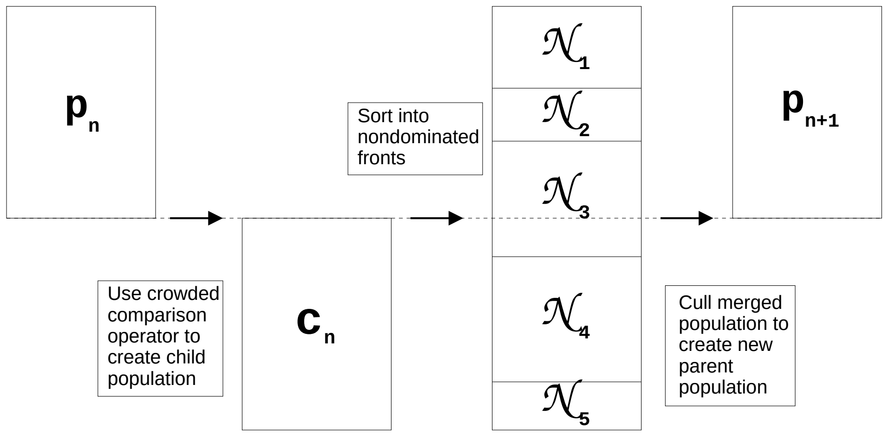

p n
Use crowded comparison operator to create child population
c n
Sort into nondominated fronts
$\mathcal{N}_1$
$\mathcal{N}_2$
$\mathcal{N}_3$
$\mathcal{N}_4$
$\mathcal{N}_5$
Cull merged population to create new parent population
p n+1

Fig. 1. The NSGA-II process.

Cancer Database[2], which has been widely used as a testbed for classification [1], [11]. The dataset consists of 699 instances, each containing nine numerical attributes plus a class attribute. Each instance is in either the malignant or benign class. The dataset contains 16 instances with missing property values. For simplicity, these instances were culled, leaving 683 instances in the dataset.

We divided the data randomly into training and testing sets, so that 70% (479 instances) of the data made up the training set and the remaining 30% (204 instances) constituted the testing set.

We used the RMITGP[3] GP programming library with strongly-typed GP [12]. The root node was required to return a double; if the value returned was below zero, this was taken to mean that the program has classified the instance as benign. Otherwise, it was considered to have classified it as malignant.

The set of functions and terminals used for the classification task is shown in Table I. The set is very simple, with the only terminals being random numbers and classification attributes, and the functions consisting of arithmetic and relational primitives and a conditional operator. Nonetheless, such a set has been shown in the past to be sufficient for creating effective classification programs [11], [7].

The fitness of an individual was taken to be the gross classification error — ie. the number of instances in the training set that are misclassified.

The plan was to have populations of 500 individuals. However, after accidentally setting the population to 50 for a batch of runs on POPE-GP, it was noticed that it performed surprisingly well considering the modest number of evaluations that the algorithm made. Thus as well as having populations of 500 for POPE-GP, the depth-limited GP and the GP with no parsimony enforcement, experiments were also run for POPE-GP and the depth-limited GP using populations of 50. Algorithms were run for 150 generations on the training set, after which the best individual (or the set of nondominated individuals in the case of the POPE-GP) was tested for generalisation performance on the testing set. All algorithms were run 50 times and results averaged.

[2] *http://www.ics.uci.edu/∼mlearn/MLSummary.html*

[3] *http://yallara.cs.rmit.edu.au/∼dylanm/rmitgp.html*

VII. Results and Discussion

Online performance graphs for POPE-GP, the depth-limited GP and the GP with no parsimony control are presented in Figure 2. Note that only the experiments using 500 individuals are plotted: the experiments with 50 individuals were omitted to improve readability.

As expected, POPE-GP has been extremely effective at tackling bloat. Figures 2(c) and 2(d) show the size growth of the fittest individual in the population and the population average respectively. The graphs clearly show that without any bloat control, program size grows consistently throughout the run. Setting a depth limit caps the size of the programs eventually, but not before they have grown quite large. Programs evolved with the POPE-GP, however, remain miniscule by comparison. By the 150th generation, the average size of programs evolved with POPE-GP was less than 10% of that of programs created with the depth-limited algorithm and less that 2% of the size of the programs generated by the GP with no parsimony control.

Interestingly, note that although the depth (Figure 2(b)) and size (Figure 2(d)) of the average individual in the uncontrolled populations seemed to grow linearly throughout the GP run, those of the fittest individual (Figure 2(a) & (c)) grow less rapidly than the average and, in the case of size, seemingly sublinearly. This is further evidence in support of the assertion that bloat is harmful to an individual's evolution. Also interesting is that despite their extremely small number of nodes, the depth of the best solutions for POPE-GP are often quite deep (Figure 2(a)). This highlights another problem of depth limitation — that programs are often barred from forming in a shape that is conducive to their function. Limiting depth encourages trees to be shallow and bushy, even if they would do better to be deep and narrow. Admittedly, size limitation does not suffer from this problem and as such should probably be preferred to depth limitation, though it seems to be rarely used.

The classification accuracies (Table II) on the testing data

Name	Return Type	Arity	Argument Types	Functionality
$+$	D	2	{D, D}	Addition
$-$	D	2	{D, D}	Subtraction
$\times$	D	2	{D, D}	Multiplication
$\div$	D	2	{D, D}	Division
If	D	3	{B, D, D}	Conditional operator; returns arg2 if arg1 is true, otherwise arg3
$\leq$	B	2	{D, D}	True if arg1 $\leq$ arg2
$\geq$	B	2	{D, D}	True if arg1 $\geq$ arg2
$=$	B	2	{D, D}	True if arg1 $=$ arg2
Between	B	3	{D, D, D}	True if arg2 $\leq$ arg1 $\leq$ arg3

(a)

Name	Return Type	Description
RandX	D	Randomly assigned constant in the range [0,100]
AttrX	D	Value of randomly assigned attribute

(b)

TABLE I

(A) FUNCTION SET (B) TERMINAL SET *[D=Double, B=Boolean]*

Algorithm	AvDepth	AvFitness	AvSize	BestDepth	BestFitness	BestSize
POPE-GP (500)	6.71	0.9586	18.77	9.40	0.9865	31.50
POPE-GP (50)	5.40	0.9467	13.12	8.00	0.9811	23.20
Depth-Limited (500)	7.99	0.9388	300.79	8.00	0.9845	282.72
Depth-Limited (50)	7.97	0.9175	270.27	7.86	0.9753	261.06
No Parsimony Pressure (500)	33.99	0.9658	1266.01	21.30	0.9858	691.56

(a)

	POPE 500	POPE 50	DL 500	DL 50	No Pressure
Mean Accuracy (%)	95.971	95.932	95.463	94.537	95.151
Standard Deviation	1.065	0.954	1.466	2.442	1.268

(b)

TABLE II

(A) END-OF-RUN AVERAGE VALUES FOR THE ALGORITHMS TESTED. (B) MEAN CLASSIFICATION ACCURACY ON THE TESTING SET.

Algorithm	(1)	(2)	(3)	(4)	(5)
(1) POPE-GP (500)	0	-0.193	**-1.980**	**-3.807**	**-3.500**
(2) POPE-GP (50)	0.193	0	*-1.890*	**-3.763**	**-3.480**
(3) Depth-Limited (500)	**1.980**	*1.890*	0	**-2.300**	-1.139
(4) Depth-Limited (50)	**3.807**	**3.763**	**2.300**	0	1.580
(5) No Parsimony Pressure (50)	**3.500**	**3.480**	1.139	-1.580	0

TABLE III

THE z VALUES FOR THE HYPOTHESIS THAT TWO POPULATIONS HAVE THE SAME CLASSIFICATION ACCURACY. VALUES IN BOLD INDICATE A STATISTICALLY SIGNIFICANT DIFFERENCE BETWEEN POPULATION MEANS FOR CLASSIFICATION ACCURACY (95% CONFIDENCE INTERVAL). VALUES IN ITALICS INDICATE A SOMEWHAT STATISTICALLY SIGNIFICANT DIFFERENCE BETWEEN THE POPULATION MEANS FOR CLASSIFICATION ACCURACY (90% CONFIDENCE INTERVAL). LARGE NEGATIVE VALUES MEAN THE ROW ALGORITHM HAS BETTER CLASSIFICATION ACCURACY THAN THE COLUMN ALGORITHM, WHILE LARGE POSITIVE NUMBERS INDICATE THE CONVERSE.

lend support to the hypothesis that enforcing parsimony does lead to improved generalisation performance. Although the actual difference in mean classification accuracy is not that large, statistical tests confirm that they are significant in most cases (see Table III). In particular the two POPE-GP algorithms (50 and 500 individuals) clearly outperformed all other algorithms in terms of generalisation performance. This is a vindication of the hypothesis that parsimonious solutions tend to generalise better and of the approach of using multiobjective techniques for parsimony enforcement.

The most exciting result was the excellent generalisation performance of POPE-GP with 50 individuals. In fact, the classification accuracy of programs generated by this algorithm on the test data was statistically indistinguishable from the POPE-GP with 500 individuals, and clearly superior to the programs generated by other algorithms. This is an extremely impressive result considering that the algorithm makes only 10% of the evaluations made by the POPE-GP with 500 individuals. Running a standard depth-limited GP with the reduced population produced poor results.

The excellent performance of the POPE-GP with 50 individuals warrants further investigation and analysis, but we do have some clues to why it performed as it did. Firstly, the average size of the best performing individual in this algorithm was substantially smaller — by almost 30% — than the best individuals generated by POPE-GP with 500 individuals. Also, their classification error on the testing data was significantly lower than those produced by the 500 individual algorithm. In other words, the programs were more general and 'fit' the training data less tightly, using only the strongest predictors in the underlying data for classification purposes. The exact reason why this occurred with the smaller population is unclear. The most obvious hypothesis — that overfitting can be reduced simply by cutting the training time — is somewhat contradicted by the poor performance of the depth-limited GP with 50 individuals.

VIII. Further Work and Conclusions

In this paper we introduced the POPE-GP algorithm, a multiobjective parsimony enforcement system based on the NSGA-II algorithm. Using a classic classification problem, we showed that the algorithm is extremely proficient at suppressing the occurrence of code bloat and that solutions generated by POPE-GP generalised better to unseen data when compared to a depth-limited GP and a GP with no parsimony control.

One very interesting result is that a POPE-GP algorithm performed nearly as well when the population size was reduced by 90% – that is, using only one tenth the number of evaluations. A similar reduction in population on a depth-limited population resulted in a significant erosion of performance. Furthermore, the solutions generated by the POPE-GP with the reduced population were even smaller than those generated by the same algorithm with the large population. The exact reasons behind this phenomenon remain unclear and would warrant further investigation given the vary large potential savings in computation time.

While the experiments described in this paper provide good evidence in support of our hypothesis that smaller solutions generalise better, further work is required on a range of problems in which overfitting is known to be an issue before a definitive conclusion can be reached.

References

[1] Bennett, K. P. and Mangasarian, O. L.: Robust linear programming discrimination of two linearly inseparable sets. *Optimization Methods and Software*, (1):23–34, (1992)

[2] Bleuler, S., Brack, M., Thiele, L. and Zitzler, E.: Multiobjective genetic programming: Reducing bloat using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*. IEEE Press (2001) 536–543

[3] Blickle, T.: Evolving compact solutions in genetic programming: A case study. In *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, vol. 1141. Springer-Verlag (1996)564–573

[4] De Jong, E. D., Watson, R. A. and Pollack, J. B.: Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*. Morgan Kaufmann Publishers(2001) 11–18

[5] Deb, K.: *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons (2001)

[6] Deb, K., Pratap, A., Agarwal, S. and Meyarivan,T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197,(2002)

[7] Eggermont, J., Eiben, A. E., and van Hemert, J. I.: A comparison of genetic programming variants for data classifi cation. In *Advances in Intelligent Data Analysis, Third International Symposium, IDA-99*, vol: 1642. Springer-Verlag (1999) 281–290

[8] Gathercole, C. and Ross, P.: An adverse interaction between crossover and restricted tree depth in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*. MIT Press (1996) 291–296

[9] Kinnear, Jr., K. E.: Generality and diffi culty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann(1993) 287–294

[10] Langdon, W. B. and Poli, R.: Fitness causes bloat. In *Second Online World Conference on Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London(1997)13–22

[11] Loveard, T. and Ciesielski, V.: Representing classifi cation problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2. IEEE Press (2001)1070–1077

[12] Montana, D. J.: Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, (1995)

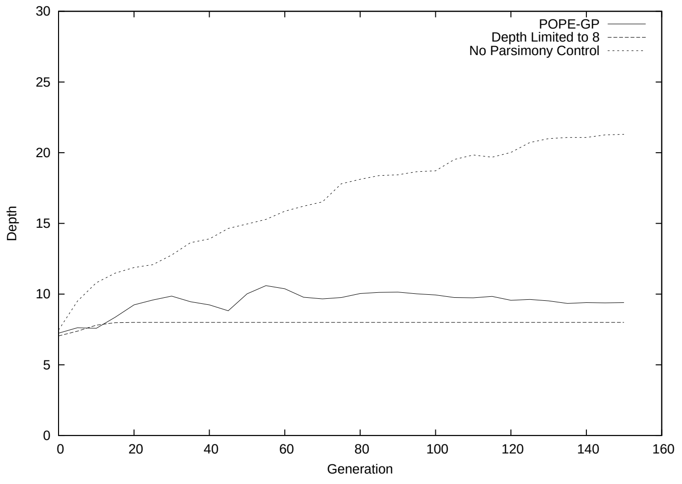
[13] Nordin, P. and Banzhaf, W.: Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*. Morgan Kaufmann(1995) 310–317

[14] Soule, T. and Foster, J. A.: Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, (1998)

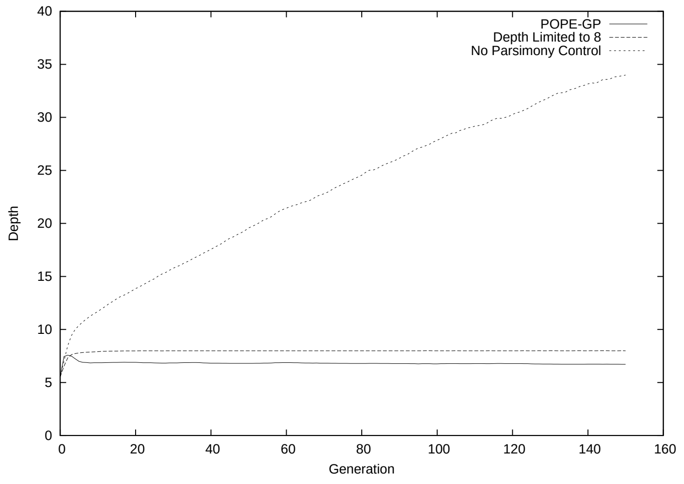
[15] Tackett, W. A.: Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann(1993)303–309

[16] Zhang, B.-T. and M¨uhlenbein,H.: Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, (1995)

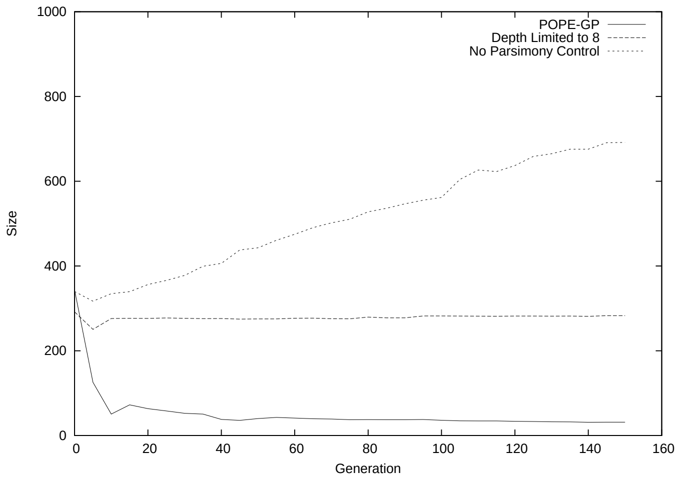
[17] Zitzler, E., Laumanns, M. and Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland(2001)



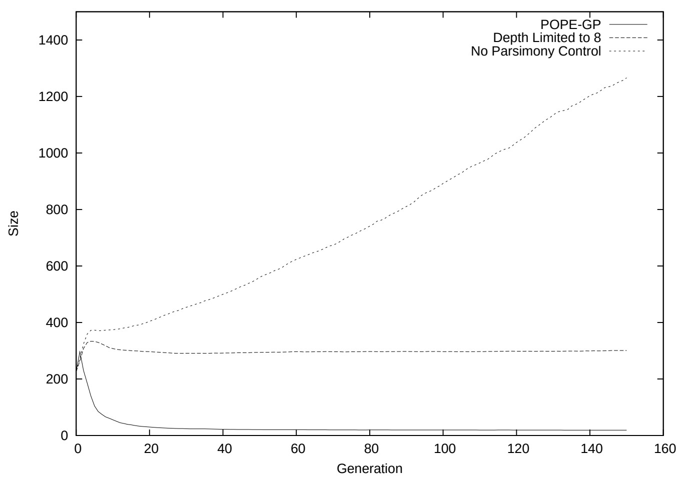
(a) Depth of fittest individual



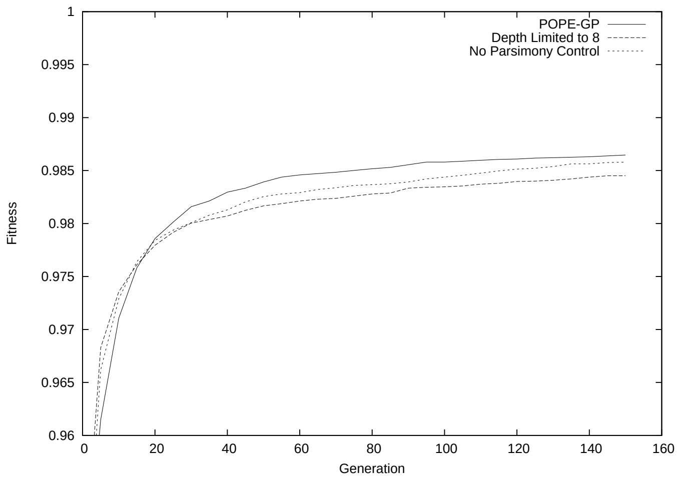
(b) Average depth



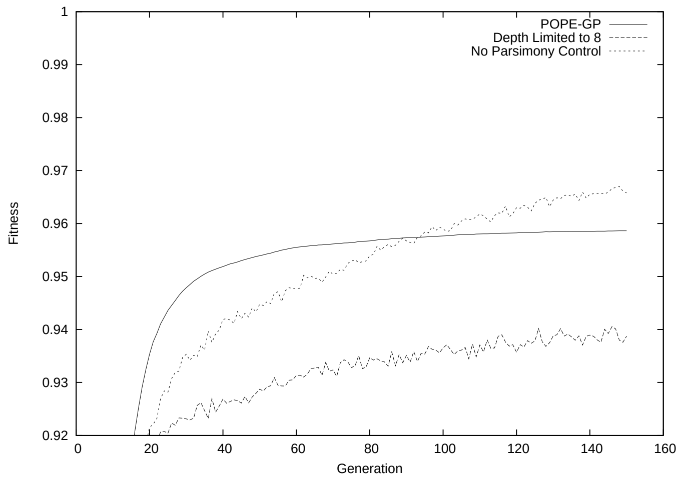
(c) Size of fittest individual



(d) Average size



(e) Fitness of fittest individual



(f) Average Fitness

Fig. 2. Online performance profiles (averaged over 50 runs; population size 500) of depth (a & b), size (c & d) and fitness (e & f). The left column follows the progress of the fittest individual in the population, while the right column shows the movement of the population average.