Strength Through Diversity: Disaggregation and Multi-Objectivisation Approaches for Genetic Programming

Jonathan E. Fieldsend
Computer Science
University of Exeter, UK
J.E.Fieldsend@exeter.ac.uk

Alberto Moraglio
Computer Science
University of Exeter, UK
A.Moraglio@exeter.ac.uk

ABSTRACT

An underlying problem in genetic programming (GP) is how to ensure sufficient useful diversity in the population during search. Having a wide range of diverse (sub)component structures available for recombination and/or mutation is important in preventing premature converge. We propose two new fitness disaggregation approaches that make explicit use of the information in the test cases (i.e., program semantics) to preserve diversity in the population. The first method preserves the best programs which pass each individual test case, the second preserves those which are non-dominated across test cases (multi-objectivisation). We use these in standard GP, and compare them to using standard fitness sharing, and using standard (aggregate) fitness in tournament selection. We also examine the effect of including a simple anti-bloat criterion in the selection mechanism. We find that the non-domination approach, employing anti-bloat, significantly speeds up convergence to the optimum on a range of standard Boolean test problems. Furthermore, its best performance occurs with a considerably smaller population size than typically employed in GP.

CCS Concepts

•Computing methodologies → Genetic programming; •Applied computing → Multi-criterion optimization and decision-making;

Keywords

Genetic programming, optimisation, multi-objectivisation, diversity

1. INTRODUCTION

Interest in semantic methods in Genetic Programming (GP) has increased over the last few years. In GP, a widely-adopted notion of semantics of a program is the vector of the program outputs for all (test) input combinations. The output vector can be also seen as the *disaggregation* of the fitness function in terms of its composite test cases. Recent results clearly indicate that use of semantics in GP considerably increases performance (e.g., [13]).

Several works have focused on using semantics to guide the design of novel mutation and crossover operators (e.g., [14, 10]). Other works have emphasized the importance of preserving semantic diversity in the population. McPhee et al. [12] analysed the impact of subtree crossover in Boolean problems in terms of semantic building blocks. They found that a major problem with this operator is that only 10% of crossover events result in a offspring semantically different from the parents. This may quickly lead to a loss of semantic diversity in the population, hence to stagnation. Beadle and Johnson [3] proposed a crossover operator that explicitly promotes semantic diversity during search obtaining improved performance. Jackson [7] also showed how another form of crossover that promotes semantic diversity results in better performance.

There are some works that use semantics in selection. A traditional work on this subject is fitness sharing in GP [11] which uses information from the individual test cases to determine a new fitness measure obtained by a linear combination of the contributions of the test cases each weighted by its relative frequency in the population. This is a commonly used diversity preservation approach in GP. Much more recent work [5] proposes to select parents for recombination such they are not only fit but also semantically different (in terms of semantic distance on output vectors). This therefore attempts to increase the probability of producing offspring semantically different from their parents.

Multi-objectivisation is the conversion of a uni-objective problem into a multi-objective version to improve search [8, 18]. Bi-objective formulations have regularly been used in GP where the second objective is to minimise bloat (program size) [15]. Recently, in [9], a multi-objectivisation of GP is proposed, based on features describing the intermediate behaviour of programs. Here we examine a multi-objectivisation of GP based on treating the entire output vector of a program (its semantics), i.e., its disaggregated fitness, as a vector of objectives. Disaggregation is appealing because it is a means to ensure that once an *individual* test case has been solved (i.e. passed) by a program in the population this capability is not subsequently lost. Furthermore, non-dominance naturally maintains rich semantic diversity in the population.

In this paper we look empirically at the Boolean domain, however we consider more general formulations at the end.

GECCO '15, July 11 - 15, 2015, Madrid, Spain
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3472-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739480.2754643

Algorithm 1 Simple Boolean GP algorithm.

```
Require: $\mathcal{D}$        Assessment data, with $n$ test cases
Require: $s$                  Population size
Require: $g$                  Number of generations
Require: $k$                  Max. # of nodes permitted in a program
Require: $c$                  Probability of crossover
Require: $\rho$               Probability of node being mutated
 1: $X :=$ initialise_population($s, k$)
 2: for $i := 1$ to $i := s$ do
 3:     $Y_i :=$ evaluate($X_i, \mathcal{D}$)
 4: end for
 5: for $g$ generations do
 6:     for $s$ repitions do
 7:         $p :=$ binary_tournament_selection($X$)
 8:         if Uniform_draw() $< c$ then
 9:             $q :=$ binary_tournament_selection($X \setminus \{X_p\}$)
10:             $\mathbf{x} :=$ crossover($X_p, X_q, k$)
11:         else
12:             $\mathbf{x} :=$ mutate($X_p, \rho$)
13:         end if
14:         $\mathbf{y} :=$ evaluate($\mathbf{x}, \mathcal{D}$)
15:         $i :=$ reverse_binary_tournament_selection(X,Y)
16:         $X_i := \mathbf{x}$
17:         $Y_i := \mathbf{y}$
18:     end for
19: end for
```

The paper proceeds as follows. In section 2 we describe the Boolean GP that is the foundation of the empirical work here, along with traditional fitness sharing in GP. In Section 3 we describe two disaggregation approaches, along with their properties, prior to the empirical comparison in Section 4. In Section 5 we discuss the results, their implications, and directions for future work.

2. THE BENCHMARK GP OPTIMISER

We employ a Boolean analogue of the symbolic regression TinyGP of [15] as the baseline algorithm, which is outlined in Algorithm 1. The low complexity of this algorithm enables us to focus on the effect of the disaggregation approaches without too many other interactions which may affect the results and therefore cloud the conclusions that may be reasonably drawn.

2.1 Basic algorithm

The algorithm maintains a population of fixed size, s. Each generation s offspring programs are created. These are either generated by crossover of two parents, each selected by binary tournament selection, or via mutation of a single parent selected by binary tournament selection. The quality measure used in the selection is the standard uni-objective aggregation – i.e. how many test cases have the two programs passed (all the test cases in $\mathcal{D}$ being processed). After the generation and evaluation of each child program (lines 6-13) a current population member is selected for replacement by the child. This is accomplished via reverse binary tournament selection. In the baseline algorithm all solutions, except the single best solution in the population, may be replaced in the reverse binary tournament selection, thus making the algorithm *elitest*. We denote the elite subset of the population by X_M and refer to its members as *marked solutions*, so for the benchmark optimiser $X_M = \{X_{best}\}$. By marking and protecting the best solution discovered so far, the best performing solution will improve monotonically.

We make available all 16 possible Boolean binary operators as the function set during the program initialisation (line 1), instead of limiting the set to e.g., AND, OR and NOT. In this fashion we do not bias the search domain in favour of gates that are known to be well-suited to the problems we will be experimenting with. The programs are denoted as trees, and represented internally as arrays of nodes. At initialisation, the root node is always chosen from the function set. All subsequent nodes in the tree have a 50/50 chance of being selected as a random member of the function set, or a random terminal (representing one of the input variables), as in [15]. The maximum permitted depth of the initialised program tree is 10 levels, so nodes at level 10 are always selected as terminals. There is no level constraint for child programs enforced by crossover or mutation. All trees also have the constraint that they must not contain in excess of k nodes in total. We use standard subtree crossover (line 10). If the resultant child contains more than k nodes, the crossover points are drawn again until a child is obtained within the k bound. We use standard point mutation (line 12), so if a terminal node is selected for mutation it is changed at random to a different terminal node; if a function set node is selected it is changed at random to another function set member. We denote this algorithm 'B' for benchmark.

All algorithms in our comparison are modifications of this baseline algorithm. They only differ in the reverse binary tournament selection (i.e. in the way they determine the population of programs that may be selected for replacement). To isolate the effect of the diversity in the population brought by the different replacement mechanisms, we do not change how parent programs are chosen during selection, or any other aspects of the baseline GP.

2.2 Fitness sharing

A standard approach to promote diversity in the population, which uses information from the test cases, is fitness sharing (see [11] for an early example of its use in GP). Here the raw aggregate fitness of a program is modified in light of how many other programs in the current population solve the same test cases from the set $\mathcal{D}$. In algorithm 'B', the aggregate fitness of the ith program in the population is:

$$f_{agg}(X_i, \mathcal{D}) = \sum_{d=1}^{n} f_d(X_i, \mathcal{D}_d) \tag{1}$$

Where $f_d(X_i, \mathcal{D}_d)$ returns 1 if the program is correct on the dth test case, and 0 otherwise. In fitness sharing f_{agg} is replaced by:

$$f_{share}(X_i, \mathcal{D}) = \sum_{d=1}^{n} \frac{f_d(X_i, \mathcal{D}_d)}{\max\left(1, \sum_{j=1}^{|X|} f_d(X_j, \mathcal{D}_d)\right)} \tag{2}$$

This fitness function down-weights the contribution of the test cases the ith program passes, in proportion to the number of programs in the population that also pass these test cases. The max function is used to ensure there are no divisions by zero.

We refer to Algorithm 1 employing fitness sharing in the reverse binary tournament selection subroutine (line 15) as variant 'F' for fitness sharing.

Population X

1	1	17	20				
2	2	1	17	18	19		
3	1	20	4	2	17	19	18
4	2	20	18				
5	7	5	20	18	17		

Test results Y

1	0	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	0	1	0
0	0	1	1	0

Population in tree representation

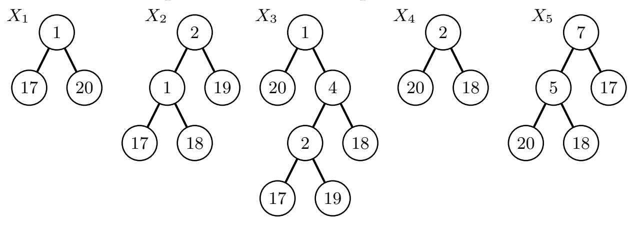

Figure 1: Illustration of the population, X, in array and tree notations, and corresponding test case results, Y. In X, the functional set elements are indicated with a value 1-16, and terminals with 17+. Each program has a corresponding row in Y, denoting the success, 1, or failure, 0, on the test cases. Due to space limitations, we display only five test cases.

3. DISAGGREGATION APPROACHES

A typical elitist approach is to ensure that the best solution in a population does not degrade in quality from one generation to the next. This is achieved by either marking the best solution in a population and ensuring it does not get selected for replacement, or equivalently, by ensuring the reverse binary tournament selection draws *without replacement* when drawing pairs.[1] In the illustration in Figure 1 this would mean X_2 is marked as it has an aggregation fitness of 3 (passing three out of five test cases), which is the highest in the population.

3.1 Preserving the best solver for each test case

In our first disaggregation approach, for each test case that has been passed (solved) by at least one member of the current population, we preserve one such program by marking it. In the worst case, this results in the marked subset of X, denoted X_M, consisting of n programs (one program for each of the n test cases). For instance, this maximum would occur if all programs only passed at most one test case, and all test cases have been passed by at least one population member. When there are many population members that have solved the same test case, the member of this subset which has solved the most test cases in total is the one marked as the best solver for the test concerned. Often the cardinality of the marked subset will be much smaller than n as the same solution may be selected to represent more than one test (as, if it solves many test cases, it is likely to have a high overall aggregate score, and therefore is likely to be ranked first for many of the test cases it solves). As long as X_M is a proper subset of X, the X_M solutions are not considered for replacement. For the test results in Figure 1 this would mean $X_M = \{X_1, X_2, X_3\}$ or $X_M = \{X_1, X_2, X_4\}$. X_1 is marked as it is the aggregate fittest (and only) solution that solves test 1. X_2 is marked as it is the aggregate fittest solver of tests 2, 4 and 5. Test 3 is solved by X_3 and X_5, which have equal aggregate fitness – in this case the program which was assessed first out of these two will be the marked program.

This approach has two attractive properties:

1. Weakly performing solutions that solve test cases that others do not are preserved, even if in aggregate they may be considered less fit (similar to fitness sharing).

2. The size of the marked set grows only linearly with the number of test cases – meaning with reasonably-sized populations it is unlikely that any of the X_M members will have to be discarded. To be certain of this, we could ensure $s \geq n$.

We denote this variant of Algorithm 1 by 'S' for best solvers.

3.2 Preserving the non-dominated members

Our second disaggregation approach treats passing each individual test as a separate *objective*, recasting the problem as a form of multi-objective optimisation. This means that each program is concerned with simultaneously maximising n objectives: $f_d(\mathbf{x})$, $d = 1, \ldots, n$. A program $\mathbf{x}$ is said to *dominate* another $\mathbf{x}'$ iff $f_d(\mathbf{x}) \geq f_d(\mathbf{x}') \quad \forall d = 1, \ldots, n$ and $\mathbf{f}(\mathbf{x}) \neq \mathbf{f}(\mathbf{x}')$. This is denoted as $\mathbf{x} \succ \mathbf{x}'$. To ensure diversity is maintained in this variant the non-dominated subset of the population is marked for preservation, meaning only dominated members are selected for replacement. In the illustration in Figure 1 this would mean $X_M = \{X_1, X_2, X_3, X_5\}$ as X_4 is the only dominated member of the population (being dominated by X_2).

Unfortunately, even for a simple toy problem like the 4-multiplexer, which has six Boolean inputs (four data signals and two selectors) and a single Boolean output, there are 64 different test cases and therefore effectively 64 different objectives. This results in what is commonly known as a *many*-objective problem in the multi-objective optimisation community [17, 6, 4] (i.e. one with four or more objectives). Traditional Pareto-dominance based approaches scale very badly on these types of problems, and the development of effective optimisers is still very much an open area of research.

However, in this situation *our underlying problem is uni-objective*, and the disaggregation into multiple competing objectives is for the purposes of diversity preservation (and subsequent exploitation). We will often deem a GP run which does not result in a solution satisfying *all* the test cases a failure – whereas it is usual in multi/many-objective optimisation to expect a set of solutions to be returned, from which the end-user will have to select one or more individuals to use. Furthermore, in Boolean problems the properties are rather different to those usually confronted in many-objective optimisation. In the Boolean GP situation the evaluation of each test case (objective) can result in one of only two values: correct or incorrect, rather than a wide range of values as would be the case if, for instance, a symbolic regression problem was being solved. Because of this we may determine, *a priori*, the maximum cardinality of the non-dominated set for any arbitrary Boolean problem (given we do not permit duplicates in quality), as a function

[1]Many implementations, including [15], draw with replacement, so, unless the elite member is not marked in some other fashion, there is no guarantee the best solution in the population will not degrade in quality from one generation to the next.

of the number of test problems, n, namely: $\binom{n}{\lfloor \frac{n}{2} \rfloor}$. In this worst case, the non-dominated set would consist entirely of solutions which solve exactly $\lfloor \frac{n}{2} \rfloor$ tests, with all possible permutations of the objective combinations present. Although this worst-case configuration is unlikely to occur regularly in practice, even approaching it will exceed the capacity of all but the most excessively large GP populations (e.g. $\binom{64}{32} > 10^{18}$).

If X_M (here the non-dominated members of X) is a proper subset of X then only the dominated members of X may be selected for replacement. If however $X_M = X$, then all members are available for replacement. In both cases it is the worst aggregated fitness (the uni-objective fitness) which is used to determine which of the two competing candidates is replaced by a new offspring program. This has the effect that programs with diverse combinations across test cases, which have good aggregated test performance, are preserved. We denote this algorithm variant 'D' for domination-based.

3.3 Preferring shorter equivalent programs

As well as the four algorithms detailed above, B, F, S and D, four variants are also compared here, which include a mechanism to encourage the evolution of smaller programs. Bias towards evolving programs that are much larger than the minimum possible to solve the problem at hand is a well-known issue for GP commonly referred to as program growth or bloat. As mentioned in the introduction, one way of confronting this is to cast the minimisation of the program size (e.g. number of nodes in the program tree) as an additional objective, other approaches to mitigate bloat include using penalty terms, and biasing how programs are evolved – a recent review of approaches is provided at the start of [16].

Here, rather than using program size as an explicit additional objective, or modifying the fitness calculation, or the search operators, we instead use program size as a differentiator in the reverse binary tournament selection when two solutions are otherwise characterised as equal. If the two are equal on their quality measure - the larger program is selected for replacement. Likewise, when offspring solutions are compared to currently marked solutions, if they are equal in quality to the currently marked solution, but shorter, then they will replace them in the marked set. Taking program length into consideration in this fashion ensures that the marked members of the population are the most parsimonious designs discovered so far for the quality performance they represent, and shorter programs are more likely to persist in the population. This means any differences in program structure are more likely to be related to differing performance on test cases. Furthermore, it is well-known (see e.g., [12]) that changes to the program structure by a genetic operator that are far from the root of the tree are likely not to have any effect on the semantics of the program, so producing offspring semantically equivalent to the parent. Preferring shorter individuals tend to lessen this problem, because in shorter individuals structural changes are necessarily closer to the root more often, hence more likely to have an effect on the semantics.

In the illustration in Figure 1, this would mean that in the parsimony preserving variant of the S algorithm, denoted S_P, X_5 would always be chosen over X_3 to be marked for test case 3, as although both programs solve test case 3, and have the same overall fitness, X_5 is the shorter of the two.

3.4 Algorithm differences

A subscript P (for parsimonious) denotes the variant of each algorithm which encourages smaller program sizes as described in Section 3.3. It is worth emphasising at this juncture that for all of the algorithms described above (B, B_P, F, F_P, S, S_P, D and D_P) the only difference between them is the tournament selection used for *replacing* population members (Algorithm 1, line 15). All other aspects of the algorithms are identical (though different storage costs may be incurred). We additionally implement a random optimiser, denoted R, which generates programs at random according to the initialisation approach used in the benchmark optimiser. This algorithm has no population as such, and merely records the aggregate best performing program it has generated so far in a run, until a perfect solver has been located or the limit on the number of function evaluations is reached.

4. EMPIRICAL COMPARISON

We employ a number of commonly used Boolean test problems [19] in the experiments here, specifically: the 6, 7 and 8 input versions of the Boolean even n-Parity problem and the Boolean majority problem (which have 2^n test cases each), the 2, 4 and 8-multiplexer problems[2], which have 2^3, 2^6 and 2^{11} test cases respectively and the 6, 8 and 10 input comparator problems. When a new offspring program is created it is assessed to check if the problem has been completely solved, in which case the algorithm records how many function evaluations were required in total.[3] A maximum of 10^6 function evaluations is permitted on each problem, if the problem remains unsolved after 10^6 evaluations the algorithm terminates.

Each algorithm was run 50 times, and the same 50 random seeds were used for each set of runs across the algorithms. Additionally four different population sizes were employed for each algorithm for each test problem: $s = \{10^1, 10^2, 10^3, 10^4\}$, which enables us to examine how different diversity preservation regimes perform in relation to the number of programs stored (as the population size would be expected to heavily affect the diversity). This results in a total of $8 \times 4 \times 12 \times 50 + 12 \times 50 = 19800$ algorithm runs in our experiments (representing the algorithm configurations across population sizes, problem types, and random seed plus the random optimiser for each problem and random seed). Other algorithm parameters are: crossover rate $c = 0.9$, point mutation probability $\rho = 0.05$, maximum number of nodes $k = 10^4$ (which match those used by the TinyGP of [15]).[4]

We now compare the algorithms based on their median performance on the problems (median evaluations required to find a program passing all test cases), their extreme performance (i.e. their rate of failure to find a zero-error program in 10^6 evaluations), examine the effect of using the parsimony preference, and investigate the dynamics of the marked subset of the population in the different approaches.

[2]These are sometimes referred to as 3, 6, and 11-multiplexer problems. This counts the total number of inputs as opposed to the number of data streams which is the standard naming convention in electronics (which we adhere to here).

[3]A single function evaluation here refers to the complete set of test cases being processed by a program.

[4]Java implementations of all the algorithms used here are available at https://github.com/fieldsend.

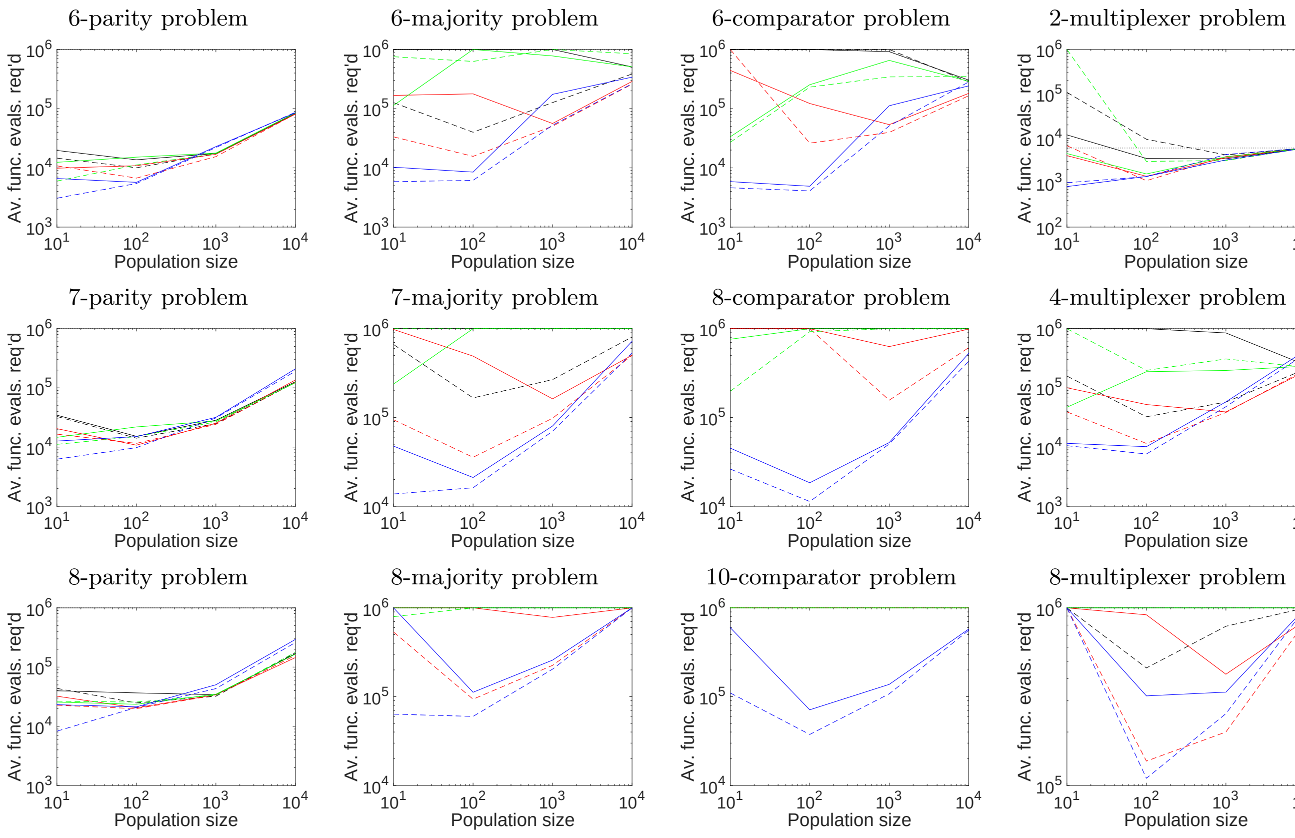

Figure 2: Average (median) number of function evaluations until problem solved, or 10^6 evaluations exhausted, for each population size. Algorithm B denoted with black line, F with a red line, S with green line and D with a blue line. B_P, F_P, S_P and D_P are denoted with the correspondingly coloured dashed lines. Axes are plotted on a log scale. The horizontal dotted line corresponds to random program generation (which does not depend on any population size). Only on the 2-multiplexer problem does the median random generator result solve the task within the 10^6 evaluation limit.

4.1 Perfomance comparison

Figure 2 shows the median performance of each of the algorithms on each test problem, for each of the population sizes. A number of interesting trends are immediately apparent. Algorithms D and D_P are able to solve many of the difficult problems not only with fewer function evaluations than the other approaches, but also with far smaller populations. Conversely maintaining the largest population size ($s = 10000$) is seen to cause relatively worse performance in D and D_P compared to the other algorithms. This is likely due to the preservation of proportionally more solutions that solve relatively few test cases, but are nevertheless non-dominated, meaning convergence to the perfect solution is delayed. This is because a sizeable amount of the search will be expended on spreading out the image of the non-dominated set in the n-dimensional test objective space, rather than pushing it forward. This does not affect the smaller population sizes to the same degree as the aggregate fitness is used to effectively strip weaker solutions from X whenever $X_M = X$.

As the number of test cases increases all algorithms converge more slowly, however B, B_P, S and S_P tend to degrade faster. All approaches tend to do better than random (for where this can be detected, i.e. where over half of the runs

Table 1: Best performing configuration(s) on each problem (written as: *alg*-*s*). The number of times the best is *significantly* better is given in parenthesis (maximum is 32). Assessed using the non-parametric Wilcoxon signed ranks test at the 5% level for all pairs of algorithms modified using Bonferroni correction - i.e. $\alpha = 0.05/33$.

Problem	# tests	Best alg(s)
6-parity	64	D_P-10 (30)
7-parity	128	D_P-10 (28)
8-parity	256	D_P-10 (32)
6-majority	64	D_P-100 (31)
7-majority	128	D_P-100 (30)
8-majority	256	D_P-100, D-100 (30)
6-comparator	64	D_P-100 (30)
8-comparator	256	D_P-100 (32)
10-comparator	1024	D_P-100 (32)
2-multiplexer	8	D-10 (27)
4-multiplexer	64	D_P-100 (30)
8-multiplexer	2048	D_P-100 (31)

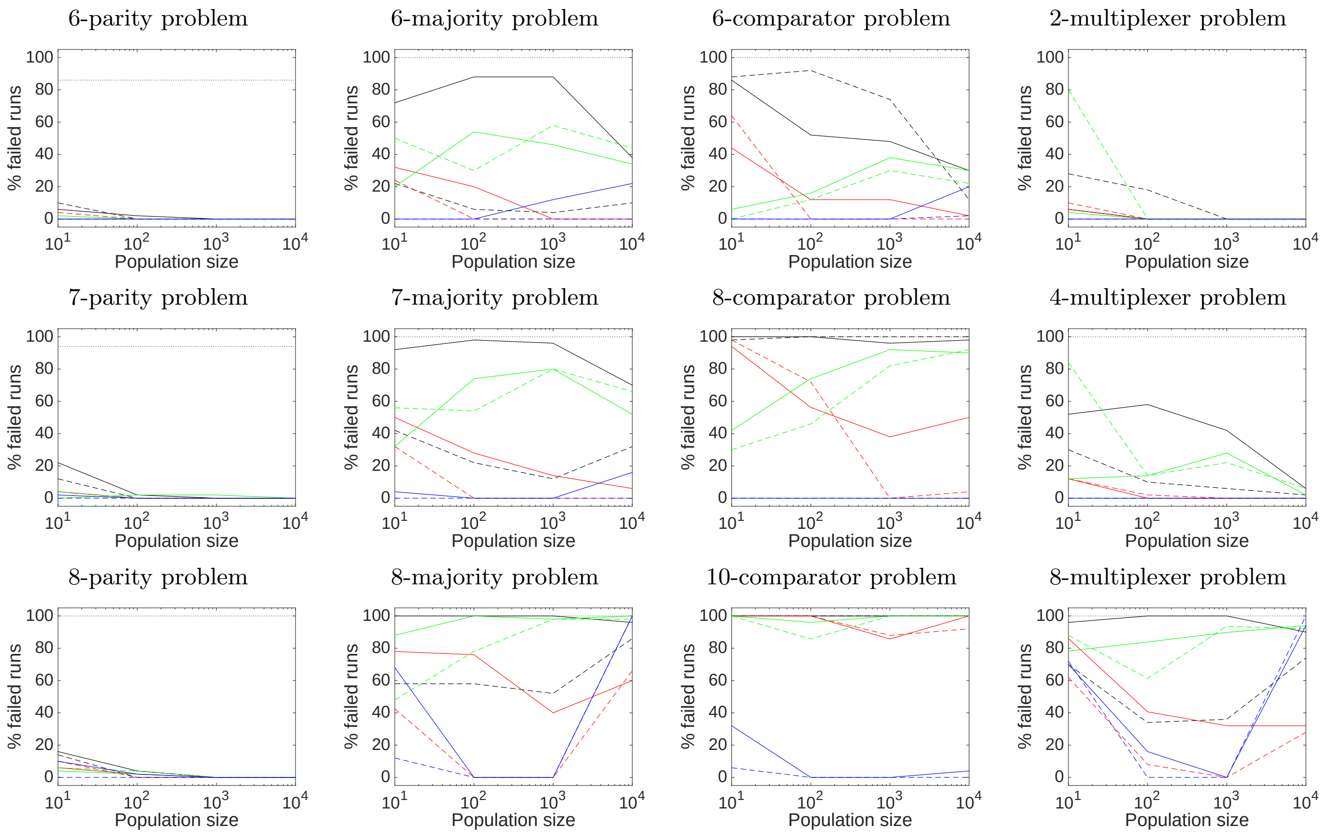

Figure 3: Percentage of runs where an algorithm failed to find a solution within 10^6 function evaluations, for each population size. Algorithm B denoted with black line, F with a red line, S with green line and D with a blue line. B_P, F_P, S_P and D_P are denoted with the correspondingly coloured dashed lines. Axes are plotted on a log scale. The dotted black line corresponds to random program generation (which does not depend on any population size).

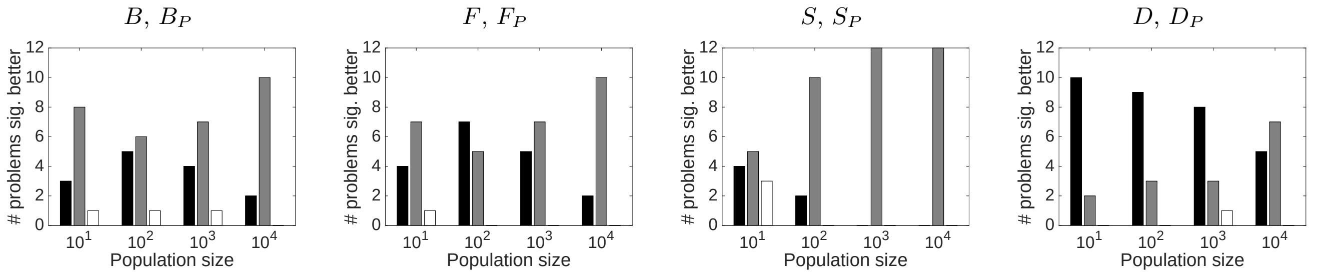

Figure 4: Number of problems where an anti-bloat variant was significantly better than its base algorithm (black bar), where there was no significant difference (grey bar) and where the base algorithm was significantly better than its anti-bloat variant (white bar), for each population size and optimiser.

find a solution within 10^6 function evaluations), except for on the 2-multiplexer problem. Here B, B_P, F_P and S_P when $s = 10$ and B_P when $s = 100$ all perform worse than the random optimiser. On inspection of the runs we find that the populations tend to rapidly fill with single node solutions containing terminals, meaning the population reaches a state where the optimum is no longer reachable given the search operators of TinyGP (as a member of the function set can only be introduced in a child program via recombination of parents which contain a function set member, or via mutation of a parent's existing function set node).

Table 1 presents the pairwise statistical comparison of convergence time of each optimiser with a particular population size, against all other algorithms and population sizes, and the random optimiser. The algorithm that significantly outperforms the most is recorded for each test problem. The D_P algorithm performs best (except on the 2-multiplexer), with the optimal population size being either 10 or 100.

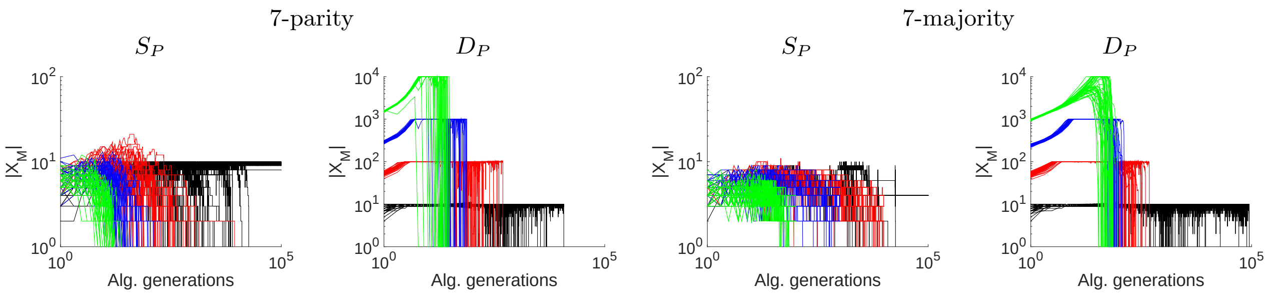

Figure 5: Marked subset sizes for algorithms S_P, and D_P as search progresses on each of the 50 runs for each population size on the Boolean even 7-parity and 7-majority problems. Green denotes $s = 10000$, blue $s = 1000$, red $s = 100$ and black $s = 10$ runs. Note that for D_P the marked set rapidly reaches the limit of $X = X_M$ and oscillates up to this bound until the problem is solved (causing X_M to collapse to a single solution). For S_P, although there are $n = 128$ test cases, $|X_M|$ is rarely above 10 on any of the runs (even those for which $|X|$ substantially exceeds this). Although the dimensionality of the problems is the same, we can see the non-dominated set in D_P takes longer to fill out for the 7-majority problem, highlighting the differences in search landscapes the two problems induce.

Figure 3 shows the percentage of runs where an algorithm failed to find a solution which passed all the test cases within 10^6 program evaluations, for each population size. D_P with both $s = 100$ and $s = 1000$ achieves convergence to a perfect solution within 10^6 function evaluations for *all* test problems for *every* run – this feat is not achieved by any other optimiser or configuration.

4.2 Anti-bloat effects

The use of the anti-bloat mechanism tends to improve the performance of all algorithms, or have no significant effect, as shown in Figure 4. Only for a handful of problems and algorithm configurations does it appear to make performance significantly degrade. Interestingly, approach D seems to benefit from this most. However, it should be highlighted that when the median results of evaluations of both the original algorithm, and its parsimony preserving variant, is 10^6 evaluations, then there will be insufficient evidence to support any significant difference in the distribution of results. As such, a high count of no significant difference comparisons is to be expected given the results shown in Figure 2 for many of the algorithms.

4.3 Marked subset dynamics

In algorithms B, B_P, F and F_P the marked subset only ever has a single member marked. For the disaggregation approaches the marked subset may vary in size throughout the run, though it is limited to $\texttt{min}(n, s)$ in the case of algorithm S and S_P and $\texttt{min}(\binom{n}{\lfloor \frac{n}{2} \rfloor}, s)$ in the case of algorithms D and D_P. Figure 5 illustrates the dynamics of $|X_M|$ for a couple of problems for S_P and D_P. $|X_M|$ in S_P is not greatly influenced by population size for either of these 128-test case problems. $|X_M|$ is however clearly limited by the population size in D_P variants, although the time to saturation does vary between problems.

Although the $|X_M|$ for the D_P algorithm appears to sit on the limit, on close inspection we find that the $|X_M|$ actually oscillates just below and up to the upper bound. This is because when $X_M = X$, then the weakest member of the population in terms of aggregate fitness is removed. Once this occurs there is one of three possibilities: (1) its replacement is itself mutually non-dominated with the remaining X_M, so it is inserted into X_M, which must again be truncated in the next iteration; (2) the child program is dominated, so $X_M \neq X$ and the dominated solution will be replaced in the next iteration; (3) the child dominates one or more members of the current X_M, resulting in X_M shrinking in size overall once the child program is entered. X_M therefore is forced to contain non-dominated solutions which are also in aggregate fit, the degree of aggregate fitness determined by $|X|$ – the larger the $|X|$ the less fit in aggregate a non-dominated solution needs to be in order to ensure its preservation. As such, there is an implicit balance between having a population size sufficient to contain solutions which cover all test cases, but not to such a degree where search is biased toward programs which only solve a small number of test cases (albeit in a combination not otherwise maintained). We can see from Figure 2 that for D_P a population in the range 100-1000 is sufficient for problems where $n = 2$ to $n = 2048$, at least for the range of problem types considered here. This is an encouraging result, as a wide population range covers a wide range of test sizes and types with good consistency.

5. CONCLUSION AND DISCUSSION

We have investigated the effect of a number of different diversity preservation approaches for deciding which population members to replace in GP, and also the effect within these of biasing the replacement of longer programs. Although disaggregating the problem and preserving programs which pass individual tests does not generally give better performance than fitness sharing, multi-objectivisation of the problem, preserving the non-dominated subset of programs and truncating when capacity is reached based on the aggregate fitness is seen consistently to give the best performance. This improvement is both in terms of median time to zero-error program discovery (sometimes over an order of magnitude faster), and in terms of consistency in finding zero-error solutions across repeated runs with different starting random seeds and starting population s.

Although we have undertaken nearly 20 000 separate runs across a range of problem types and sizes we have restricted our examination to Boolean problems here. Symbolic regression problems may also be approached using a similar multi-objectivisation approach to preserve diversity, however some additional considerations would need to be taken. Unlike in the Boolean case, it is impossible to derive a theoretical limit on the cardinality of X_M. Additionally it is not immediately obvious that the aggregate fitness will be the best differentiator when truncating non-dominated solutions when the X_M capacity is reached. For Boolean problems, selecting to reject based upon the aggregate fitness is equivalent to selection using the hypervolume, selection using the favour relation, and selection using the average rank. In general these quality measures are not equivalent, although they have all been proposed for use in many-objective optimisation (see e.g. the comparison in [4]), and it may be the case that some are better suited that others for the general multi-objectivisation of GP optimisation. The maintenance of a non-dominated set also has a cost: this part of our code took between 10^{-3} and 10^{-1}ms on average per update (on a 2.66GHz machine). However, the overall cost is marginal for the most effective $s = 100$ variant, taking 0.25–3.13% of the total run time (depending on the problem), and the use of specialised data structures can reduce this further [2].

Partially evaluating programs (only considering a random subset of tests) is often used in GP, giving an approximate fitness but with lower computational cost. Maintaining a non-dominated set for Boolean problems opens the possibility of leveraging such partial program evaluations in a systematic fashion, without introducing additional stochasticity. Given a current X_M, one can identify *a priori* the quality profiles that would be either accepted into X_M, or rejected. It would be possible therefore to undertake a partial evaluation and determine that, irrespective of the performance on the remaining tests, a program would be rejected (for instance if the aggregate will be lower than the least fit program currently in an X_M at capacity), rather than using a random selection of tests. Furthermore, as the overall quality of the X_M members improves, a fewer number of tests would need evaluating before poor solutions may be rejected. We intend to investigate empirically the speed up that might be attained in this fashion. Alternatively multi-objective algorithms for coping with delayed objectives could be exploited [1], as these employ techniques to cope with 'missing' objective values.

6. REFERENCES

[1] R. Allmendinger, J. Handl, and J. Knowles. Multiobjective optimization: When objectives exhibit non-uniform latencies. *European Journal of Operational Research*, 243(2):497–513, 2015.

[2] N. Altwaijry and M. Menai. Data structures in multi-objective evolutionary algorithms. *Journal of Computer Science and Technology*, 27(6):1197–1210, 2012.

[3] L. Beadle and C. Johnson. Semantically driven crossover in genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 111–116, 2008.

[4] D. Corne and J. Knowles. Techniques for Highly Multiobjective Optimisation: Some Nondominated Points are Better than Others. In *Genetic and Evolutionary Computation Conference*, pages 773–780. ACM, 2007.

[5] E. Galvan-Lopez, B. Cody-Kenny, L. Trujillo, and A. Kattan. Using semantics in the selection mechanism in Genetic Programming: A simple method for promoting semantic diversity. In *IEEE Congress on Evolutionary Computation*, pages 2972 – 2979, 2013.

[6] E. J. Huges. Evolutionary many-objective optimisation: many once or one many? In *IEEE Congress on Evolutionary Computation*, volume 1, pages 222–227. IEEE, 2005.

[7] D. Jackson. Promoting phenotypic diversity in genetic programming. In *Parallel Problem Solving from Nature - PPSN XI*, volume 6239 of *Lecture Notes in Computer Science*, pages 472–481. Springer, 2010.

[8] J. Knowles, R. Watson, and D. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *Evolutionary Multi-criterion Optimization, EMO'01*, pages 269–283. Springer, 2001.

[9] K. Krawiec and U.-M. O'Reilly. Behavioral programming: a broader and more detailed take on semantic GP. In *Genetic and Evolutionary Computation Conference*, 2014.

[10] K. Krawiec and T. Pawlak. Locally geometric semantic crossover: a study on the roles of Semantics and homology in recombination operators. *Genetic Programming and Evolvable Machines*, 14:31–63, 2013.

[11] R. I. McKay. Fitness sharing in genetic programming. In *Genetic and Evolutionary Computation Conference*, pages 435–442. Morgan Kaufmann, 2000.

[12] N. F. McPhee, B. Ohs, and T. Hutchison. Semantic building blocks in genetic programming. In *11th European conference on Genetic programming*, pages 134–145. Springer, 2008.

[13] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31. Springer, 2012.

[14] Q. U. Nguyen, X. H. Nguyen, and M. O'Neill. Semantic aware crossover for genetic programming: The case for real-valued function regression. In *12th European Conference on Genetic Programming*, pages 292–302. Springer, 2009.

[15] R. Poli, W. B. Langdon, and N. McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[16] R. Poli and N. McPhee. Parsimony Pressure Made Easy: Solving the Problem of Bloat in GP. In Y. Borenstein and A. Moraglio, editors, *Theory and Principled Methods for the Design of Metaheuristics*, Natural Computing Series, chapter 9, pages 181–204. Springer, 2014.

[17] R. C. Purshouse and P. Fleming. Evolutionary many-objective optimisation: An exploratory analysis. In *IEEE Congress on Evolutionary Computation*, volume 3, pages 2066–2073. IEEE, 2003.

[18] C. Segura, C. A. Coello Coello, G. Miranda, and C. León. Using multi-objective evolutionary algorithms for single-objective optimization. *4OR*, 11(3):201–228, 2013.

[19] D. White, J. McDermott, M. Castelli, L. Manzoni, B. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14:3–29, 2013.