EuroGP-2025. Bing Xue, Luca Manzoni, Illya Bakurov *Eds.*, Trieste. 23-25 April 2025. Springer. *Forthcoming*

# Population Diversity, Information Theory and Genetic Improvement

William B. Langdon, David Clark

W.Langdon@cs.ucl.ac.uk david.clark@ucl.ac.uk
CREST, Department of Computer Science,
UCL, Gower Street, London, WC1E 6BT, UK

**Abstract.** Compression, e.g. gzip, gives algorithmic information theory (Kolmogorov Complexity) based measures of string population diversity. To boost it we use the GI tool Magpie and select programs of average fitness that contribute most to variety, allowing evolution to automatically tailor triangle.c for production speed. We calculate C source code diversity via approximations to the Normalised Compression Distance on Multisets (NCDm) using both Cohen and Vitanyi's $O(n^2)$ approach and our own, $O(n)$ method, finding the cheaper, $O(n)$, is equally good.

**Keywords**
Evolutionary computing, EC, genetic programming, GP, SBSE, NCD, Normalised Information Distance, NID, perf, test set diameter

## 1 Introduction

Diversity plays an important role in optimising finite populations, e.g. in genetic algorithms [1], genetic programming (GP) [2,3] and genetic improvement GI [4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]. In software engineering the widely used application of diversity is to test sets [20][21][22][23][24][25], whilst information theory has been applied to software robustness [26] and security [12,27]. The problem of lack of local gradient, or worse deceptive fitness gradients [28,29] or even fitness plateaux [30] is well known in optimisation and evolutionary computing. Lack of local gradient appears to be important in population based genetic improvement, with software engineering benchmarks such as the triangle program (Section 3 [31,32,33]) having search landscapes [34,35] dominated by large plateaus of equal fitness connected by relatively few improving mutations. Although, using normalised compression distance (NCD), we investigate the usefulness of program source code variability as a measure to decide which individuals to discard and which to select for the next generation (Figure 1), such syntax based population diversity gives mixed results.

Programs are strings but a high quality string diversity measure is a challenging topic. The Rolls Royce measure of string diversity is Vitanyi's Normalised Information Distance [37], which is based on Kolmogorov Complexity. Unfortunately Kolmogorov Complexity is not computable but numerous approximations

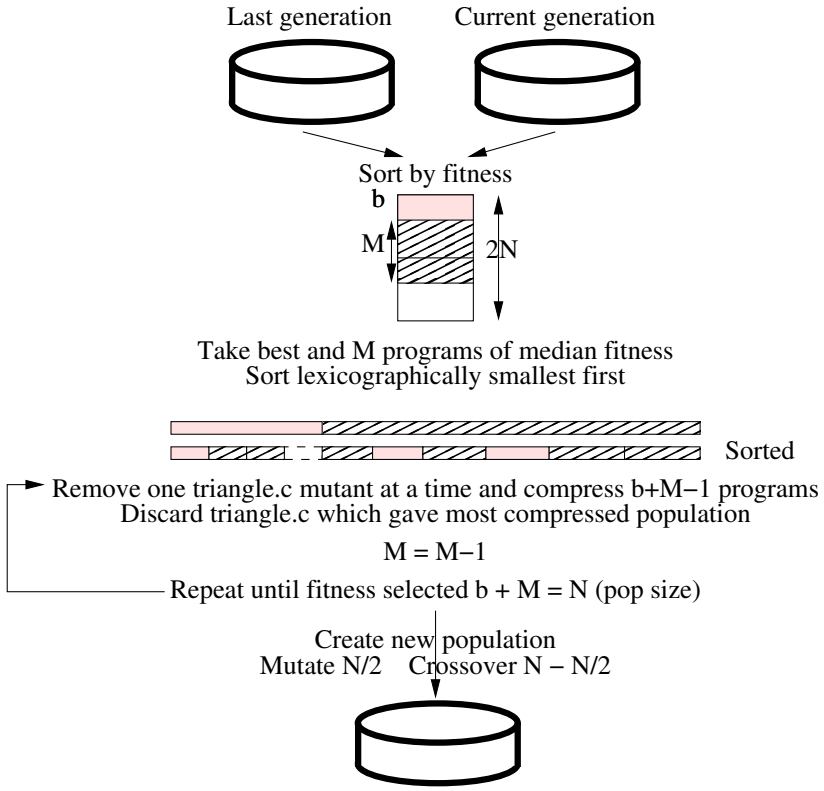


**Fig. 1.** Incorporating Cohen and Vitani's Normalised Compression Distance on Multisets (NCDm) [36] O($n^2$) into genetic algorithms. The GA population can contain duplicates (hence multiset rather than set). Each generation the GA selects from the current and previous generation the best (pink) and those of the average fitness individuals (hatched) which make the population most diverse (i.e. hardest to compress).

are available and these offer trade offs between accuracy and efficiency. Commonly used is Cilibrasi and Vitanyi's compression based approximation, the Normalised Compression Distance (NCD) [38,25]. At the other end of the efficiency scale is the well known but more approximate Levenshtein distance [39,40,41], also satisfying the metric space axioms, and in the middle is dictionary based compression which can be efficient and produce a tight upper Kolmogorov Complexity bound for a known, finite population.

We exploit a variant of NCD known as NCDm, or NCD for multi-sets, that produces a single diversity measure or "diameter" for a multi-set, such as an EC population [21,36]. In addition, we err on the side of efficacy rather than efficiency, using Cohen and Vitanyi's suggested quadratic-in-population-size approximation for NCDm as well as a linear one of our own invention. When improving the execution speed of the triangle program, we find that these diversity

measures are approximately equally useful across our GI runs. Since population size is a confounding variable for diversity, we study a wide range of population sizes from one to 1000. However run time increases quadratically and as no further fitness improvement was found, we limit our experiments to one run per scenario for populations sizes 200–1000.

Section 3 says how we use the popular software engineering triangle program benchmark, whilst Section 4 describes our hybrid genetic programming and GI Magpie system, particularly how it incorporates information based diversity into selection. The experiments (Section 5) and results (Section 6) are followed by a discussion of a 17% improved triangle program (Section 7) and our conclusions (Section 8). But first the next section briefly describes Cohen and Vitanyi's diversity measure NCDm [36], how Feldt et al. [21] use it to measure test suite diversity and then how we have used their approach within genetic improvement to actively select breeding populations of evolving program source code.

## 2 Information Theory applied to Genetic Algorithms

The topic of information theory in genetic algorithms (GAs) and evolutionary computing, e.g. genetic programming [42] and genetic improvement (GI) [43], is vast. We will concentrate upon how we have applied it in population selection in our GI and only note that the approach could be widely used in population based evolutionary computing.

In genetic algorithms the importance of striking the right balance between exploring to find new good regions of the search space and exploiting the good parts already found has long been known [1]. We present (Figure 1) an information theoretic way of combining fitness based selection and population diversity based on Andrew Cohen and Paul Vitanyi's [36] diversity measure for multisets. Their Normalized Compression Distance (NCD) based multiset distance (NCDm) is very general and has been applied to test set selection [21].

Cohen and Vitanyi's underlying approach is to define the information content of a multiset (which in our case is the population) as using Kolmogorov complexity. The Kolmogorov complexity of a string is the size of the smallest program that can generate the string. However Kolmogorov complexity is not in general computable and so they take their usual NCD approach and approximate it as the length of the compressed string. (Here we will use the size in bytes of the output generated by gzip.) As part of calculating the normalised distance for a collection of strings (technically a multiset, as the collection may contain duplicates) they wish to find the minimum compressed size with all possible orderings and to normalise by dividing by the largest compressed size of the multiset excluding all possible subsets. Since there an exponentially large number of orderings they define an approximation which is still a metric but whose computational complexity is only quadratic in the number of strings $O(n^2)$.

We start with their quadratic algorithm, as used by Robert Feldt et al. [21]'s universal algorithm for measuring the diversity of test suites ("test set diameter"). Our NCD based approach is feasible even for populations of 1000, but as expected

it is slow. Therefore we introduce a further, linear time O($n$) approximation, which can be orders of magnitude faster (Section 6.6) and as effective.

### 2.1 Normalized compression distance (NCD) for Multisets (NCDm)

As there an exponentially large number of orderings, to approximate the smallest compression distance over all possible orderings Cohen and Vitanyi [36] consider only a quadratic number of orderings. To select which ordering, their basic approach is to order the multiset and then concatenate it into a single file which can be compressed (Figure 1). They order the strings (here members of the population) first by size and then alphabetically. By placing similar strings next to each other, there is a good chance the compression algorithm will perform well and give a small compressed output file. They then in order omit one member of the multiset and compress the new (now shorter) concatenated file. They work through the whole multiset one at a time, to find which string contributed least and discard it. This gives a multiset which is one member smaller. They repeat, again removing the string which has least impact on the compression of the new (smaller) multiset, until only 2 strings are left in the multiset. (Notice the algorithm is described as sequential but parts could be run parallel.) We first follow Feldt et al. [21] and in the next section describe how we use this central part of Cohen and Vitanyi's [36] algorithm as part of parent selection in the evolutionary algorithm.

### 2.2 Information Based Parent Selection

To incorporate Feldt et al. [21]'s test case selection algorithm into a fitness based evolutionary algorithm with population size N, we start with the current and previous population (both of size N, total size 2N). From these 2N we select N to be parents of the next generation. These are sorted by fitness (cf. rank based selection [1,44]). Those better than average (median) fitness are automatically selected. Those of worse than median fitness are automatically discarded. We then apply information theory to chose those individuals of average fitness which will contribute most to the breeding population of parents for the next generation. (The number of programs of average fitness is quite variable, but in these experiments it is typically near 10% of the combined population size.) Like Feldt et al. [21] we apply Cohen and Vitanyi's [36] NCDm to the source code of the programs of average fitness but we do not calculate the distance, we merely run the NCDm algorithm (Figure 1) until we have reduced the number of files (here C programs) until it plus the number of better than average fitness members of the combined populations is equal to N, the size of the next population. This becomes our breeding population.

This is not in itself an elitist approach. We can choose to make it elitist by passing one or more members of our breeding population unchanged to the next generation. But we choose to create half the children using mutation and the remainder by crossover. The approach could be readily applied to many evolutionary algorithms which use separate populations.

## 3 Genetic Improvement triangle.c Benchmark

The software engineering triangle benchmark takes three inputs and returns one of four integer values representing the type of the triangle: scalene, isosceles, equilateral or not a triangle. (Versions of the triangle program seem to go back to 1976 and Fortran [45]. We use our C version[1] and test suite[2] [46].) The important function is 40 lines of C source code (1300 bytes) containing 16 comparisons and 8 return statements. The benchmark's test suite is designed to cover all branches. It spends much of its time checking for errors ("not a triangle", 9 of the 14 tests). In the source code most of these error checks are at the start of the code, with another right at the very end.

In our genetic improvement experiment we suppose that the developers of a real system have taken such a heavy error detection approach and later the customer wants the code to be faster for everyday use. That is, in the triangles example, we assume most of the time the code would be presented with three numbers which are indeed the 3 lengths of the side of a triangle. So in our experiment we start with the original code and tests but now weight the tests so important ones score more in the fitness function (see Table 1).

The **fitness** test harness uses the Linux perf utility's API to measure how many computer instructions the mutated code takes on each of the 14 tests and multiplies it by the weighting for that test. The mutant's fitness is the 14 added together (Table 1). Note we minimise fitness scores. If the mutant gives the wrong answer on any test or there is a run time error, its fitness is so poor it will never be selected to be a parent.

Mutations and crossovers are able to re-arrange the existing C code to get better scores by moving code that deals with lower weighted cases to further from the start, allowing important cases to be dealt with more quickly.

Even in a time sharing network desktop, Linux perf's instruction count proved very stable and gave reliable fitness measurements. In contrast measurements of elapsed time taken during fitness testing are very noisy [47,48,49].

## 4 Genetic Programming based on Magpie

Our genetic programming systems is based on Magpie [50][3]. Magpie is a language independent genetic improvement system written in Python. It has many options. We use only its XML mode. Using srcml (version 1.0.0) we convert the mutable source code into a single triangle.c.xml file. To avoid changes to Magpie, the population selection (Section 2.2 and Figure 1) are done externally. Magpie and GP parameters are given in Table 1. Our GP makes use of Magpie in three ways:

1. Magpie was run with triangle.c.xml to generate a pool of all 2535 possible different XML mutations.

[1] `https://github.com/wblangdon/triangle/blob/master/jss/triangle.c`
[2] `https://github.com/wblangdon/triangle/blob/master/jss/testcases_oracle.txt`
[3] `https://github.com/bloa/magpie` downloaded 2 October 2023.

2. To create the initial GP population Magpie is run many times to create one random mutant at a time. We reject mutants which do not compile, give runtime errors or fail one or more fitness test. We keep doing this until we have enough credible mutants to fill the initial population (mutant triangle.c mean size 1320.1 ± 41.8 bytes).
3. As our GP is running, Magpie facilities are used to compile, run, test and calculate fitness of each mutant.

### 4.1 GP Operations: Mutation and Two Point Crossover

The basic Magpie representation is like linear genetic programming [51,52] and consists of a text based list of genes. Therefore it is easy to extract and insert individual genes from and into Magpie genomes.

Mutation: a parent, selected uniformly at random from the breeding population, is copied and the copy mutated by selecting uniformly at random one gene within it and replacing it with one taken at random from the 2535 possible different XML mutations (see item 1 in previous section).

With crossover: two parents are chosen uniformly at random from the breeding population. The first is copied. Two random cut points are chosen uniformly in the copy and in the second parent. The middle part (i.e. between the cut points) of the copy is replaced by genes copied from the middle of the second parent [53, Fig. 2].

Note mutation does not change the number of genes whereas crossover can but on average neither changes the genome's length.

## 5 Experiments

The GP/Magpie system was run 10 times on populations of 1, 2, 5, 20, 50, 100. Also there were a few runs of 200, 500 and 1000. For each we tried three types of selection (Figure 1 Section 2.2): based on Feldt et al.'s NCDm $O(n^2)$ [21], our linear $O(n)$ approximation to NCDm and finally breaking ties of average fitness at random. The GP representation, fitness and parameters are given in Table 1. The fastest triangle.c mutant on test cases may be found any time up to generation 100.

## 6 Results

### 6.1 Speedup

Figure 2 shows the performance of the best in run for all ten repeated runs. As expected there is variation between runs but typically the population needs to contain at least 20 mutated programs for the search to do well. Indeed, although we did a few runs with larger populations (200, 500 and 1000) there seems to be no advantage in increasing it above 100. There is little difference between the three selection algorithms (plotted with $+$, $\times$ or $\square$). Note the new linear

**Table 1.** Faster triangle.c

| | |
|---|---|
| Representation: | C code converted to XML by srcml. Variable length linear sequence of XML mutations. Mutated XML converted to C code and compiled. |
| Fitness cases: | 14 test cases, each 3 sides of triangle and expected classification. Test suite designed to cover original C code.<br>Test suite weighting to favour important outputs: scalene and equilateral (one test each) weight 81, isosceles (three tests) weight 27, not a triangle (nine tests) weight 1, (Section 3). |
| Selection: | Fitness is the sum of the number of instructions taken by each test multiplied by its weighting<br>fitness = $\sum_{i=1}^{14}$ X86 instructions for test $i \times$ weight $i$<br>If mutant fails to compile, fails at run time, exceeds 2 second time out or gives wrong answer on any test its fitness is so bad it will never have children.<br>$1^{st}$ fitness based rank selection and $2^{nd}$ contribution to population diversity, see Figure 1 and Section 2.2. |
| Population: | Panmictic, non-elitist, generational, size 1, 2, 5, 10 $\cdots$ 1000. |
| initial pop | Every triangle.c is mutated exactly once. All compile and run (page 6 item 2.). Initial fitness 7929–12578 (most as unmutated code 9069). |
| Parameters | |
| Magpie: | Python version 3.10.1, GGC version 10.2.1, compiler options -O3 -DNDEBUG. Magpie defaults except [search] warmup=1. XML edits: StmtReplacement StmtInsertion StmtDeletion ComparisonOperatorSetting ArithmeticOperatorSetting NumericSetting RelativeNumericSetting StmtMoving |
| GP : | 50% subtree XML crossover, 50% subtree XML mutation (Section 4.1). 100 generations. No size limit. |

approximation $\times$ to estimating population diversity does as well as the quadratic approach inspired by Feldt et al.'s Test Set Diameter O($n^2$) [21] + and it is considerably faster (Section 6.6). Except for some runs with a population of only one or two, all runs make progress. If we concentrate on runs with a population of 20 or more, the median speed up is 16%.

### 6.2 Evolution of Performance

The performance of the three types of selection with various population sizes are summarised in Figures 2 and 3. Figure 3 plots the evolution of the fastest (weighted) triangle.c program in the population at each generation for a typical run at each population size. Typically each run does not converge and the populations contain a range of fitness values. As expected performance depends on population size, with larger populations doing better. Runs with a population containing a single program (which will have been created by crossover) typically make no progress (shown by a horizontal line at the top of Figure 3).

In the absence of elitism (Section 2.2 above), even though the best in the population is guaranteed to be part of the breeding population, they have no guarantee that they will be selected from it for either way of making genetic

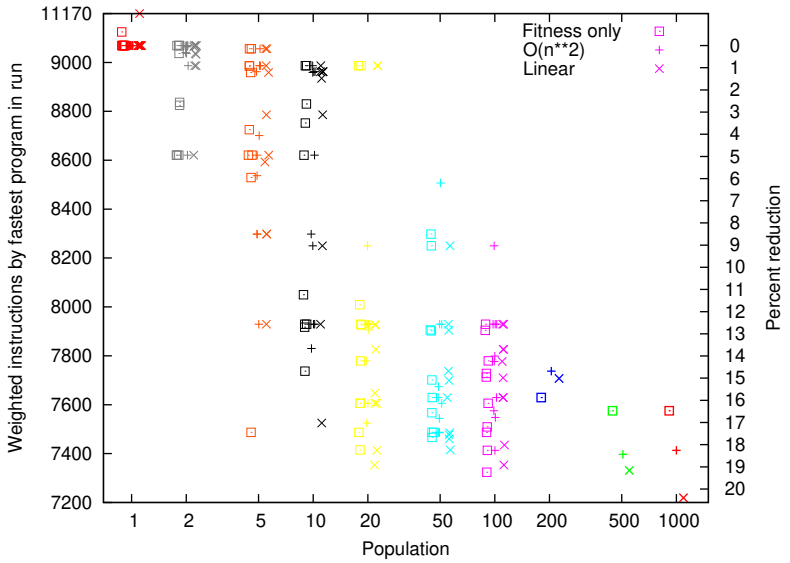


**Fig. 2.** Fastest triangle.c mutant found in ten runs by generation 100 with pop size 1, 2, 5, $\cdots$ up to 1000 (only 1 run 200–1000). Select best from current and previous generations to be parents. (One linear $\times$ run with population 1 got stuck at fitness 11 170.) Small horizontal noise added to spread data. Section 6.1.

changes. And even if selected, their children will be either mutated or created by crossover. Either of which may give a child with worse (or indeed better) fitness. Thus although a downward trend can be seen in Figure 3, fitness does not usually improve monotonically. Indeed, since we are hoping for diverse populations we should not be disappointed that evolution does not lock into the "best seen so far" fitness value.

### 6.3 Evolution of Population Diversity

Figure 4 shows the average evolution of information contents of the population in ten runs with a population of 100 for the three selection schemes and Figure 5 presents a summary by selection scheme and population size (note log scales).

Typically only about 10% of the population have average fitness (where information content is used to break ties, Section 2.2). Suggesting, in contrast to typical tournament selection [3, Sect. 2.3], selecting the best of the current and previous generations with 100% (50% crossover + 50% mutation) genetic modification avoids too high a selection pressure and does not drive the population to converge on a single fitness value [54] and instead our GP retains diverse fitness.

As with Figure 2, Figure 5 presents a summary of all the runs for each selection type and population size. As typically the population's information content does not tend to rise to a maximum at the end of the run but often falls

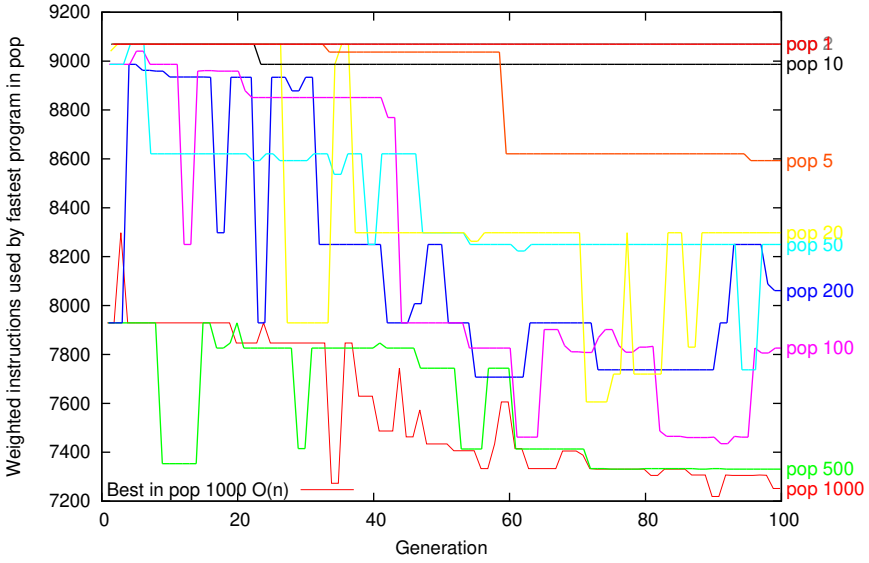


**Fig. 3.** Evolution of best fitness in typical run using linear O($n$) complexity selection. GI populations from 1 to 1000. Runs with O($n^2$) and without complexity selection are similar. (Same run colours as Figure 2.) See Section 6.2.

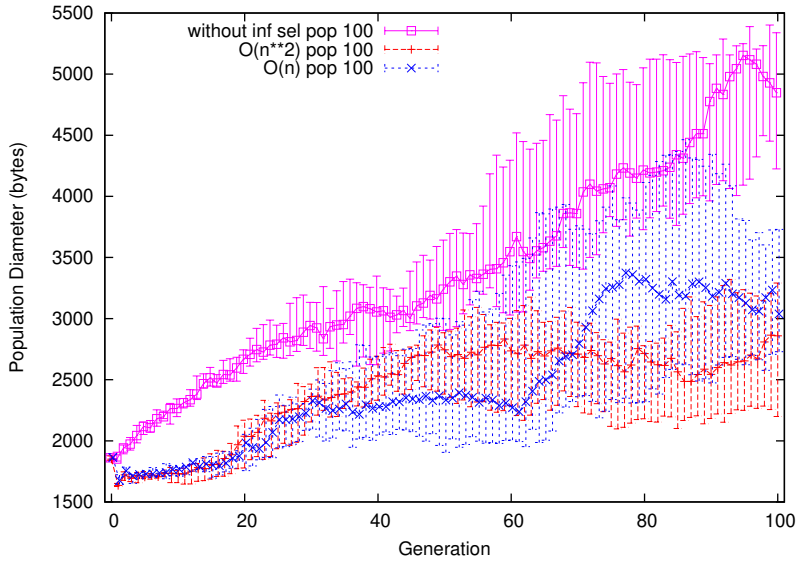


**Fig. 4.** Evolution of median population diameter for ten runs with the three selection schemes. GI populations of 100. The error bars give the interquartile spread across ten runs. Note change in colour scheme.

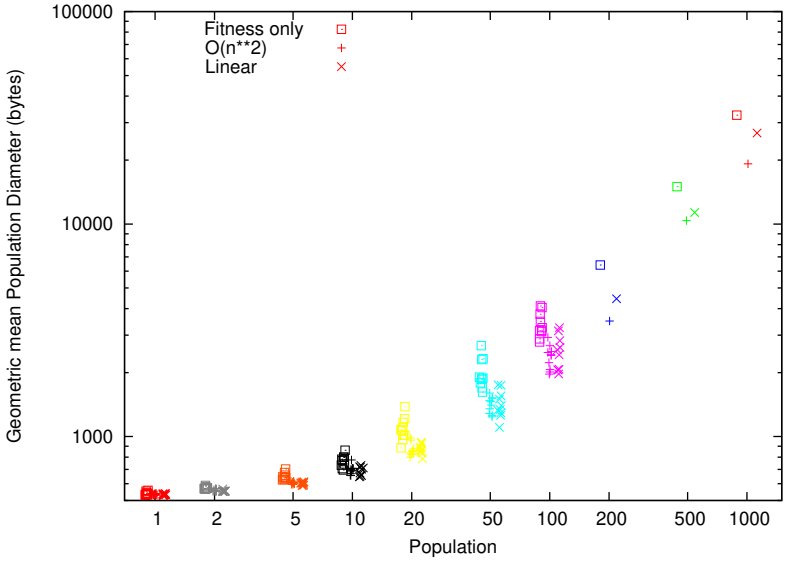


**Fig. 5.** Average compressed population of triangle.c mutants (see Section 6.3) in ten runs up to generation 100 with populations 1, 2, 5, $\cdots$ up to 1000 (only 1 run 200–1000). Small horizontal noise added to spread data.

towards the end, Figure 5 gives the average information content across each run. As expected, Figure 5 shows the higher performing larger populations contain more information than the smaller populations of triangle.c mutants. Whilst, in terms of population information content, our linear O($n$) $\times$ approximation behaves as Cohen and Vitani's O($n^2$) NCDm [36] +. Although the fitness only approach $\square$ gives on average less compressible populations ($p = 4\ 10^{-17}$ two-sided non-parametric Mann-Whitney U test), across the 73 runs of each type the median difference is only 13%.

The size of the triangle.c mutants (phenotype) is determined by the mutations applied (genotype). On average both information based selection schemes increase the C source code by about 10% (to 1430 bytes) while with fitness only selection there is more bloat (24%, 1617 bytes). gzip is very good at compressing the populations. For example with populations 1, 2, 5 and 10, it compresses the whole population into less than half the space of the original program. Even with the larger populations (e.g. 100) gzip gives average compression ratios of 46–74.

### 6.4 Evolution of Genome Size

To illustrate the evolution of the number of genes, Figure 6 shows the growth of the average genome size for ten runs with a population of 100 triangle.c programs. It plots 10 runs with O($n^2$) (solid lines), 10 runs with our linear information based selection (dashed lines) and 10 runs with fitness only selection, where fitness ties

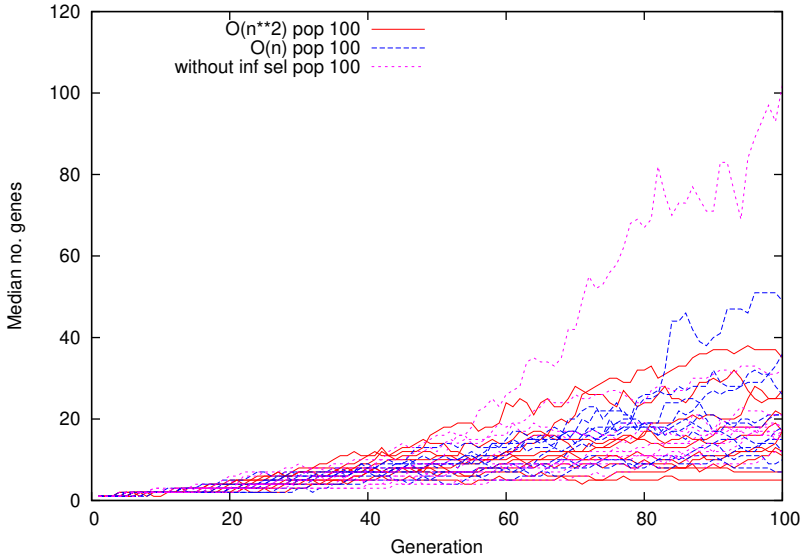


**Fig. 6.** Evolution of genome size for ten runs with the three selection schemes. GI populations of 100. (Same colour scheme as Figure 4.)

are broken randomly (dotted lines), showing the median number of genes rising from 1 initially to $17 \pm 14$ in generation 100. (The instances of triangle.c similarly grow from 1320 on average to $2000 \pm 400$ bytes by generation 100.)

Although there are fluctuations between generations, an upward trend, known as bloat [55], can be seen. Therefore Figure 7 gives genome statistics for the end of each run with each of the population sizes and each of the three selection schemes. Figure 7 shows there is considerable variation between independent runs (note log scales). However there is a trend for larger populations (which tend to contain fitter programs) to contain more genes (i.e. more Magpie mutations of triangle.c, see also Figure 8). In runs with the same population size, all three approaches ($+ \times \Box$) tend to have on average a similar number of mutations.

### 6.5 Size of Best Solutions

Figure 8 shows all the best in run mutant's performance (y-axis) and their number of mutations (x-axis). Figure 8 shows a clear trend for faster (weighted) programs to have had more changes. However a few mutated triangle.c with only one, two or three changes do very well. The three selection schemes do approximately as well as each other.

### 6.6 Time to do Selection

With fitness only selection and our linear $O(n)$ approximation, selection typically takes $< 1$ second per generation on an otherwise unladen multi-core 32GB

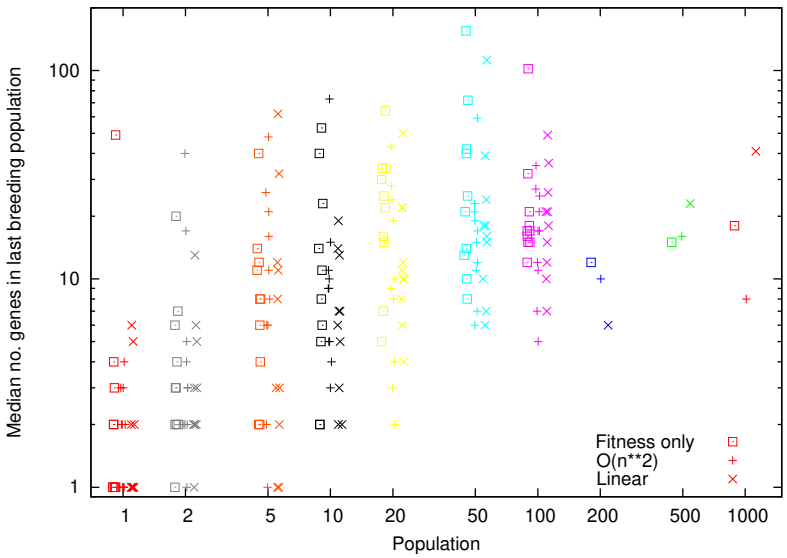


**Fig. 7.** Average number of genes in ten runs at generation 100 showing little difference in the 3 selection schemes. Small horizontal noise added to spread data (same run colours as Figure 2). See Section 6.4.

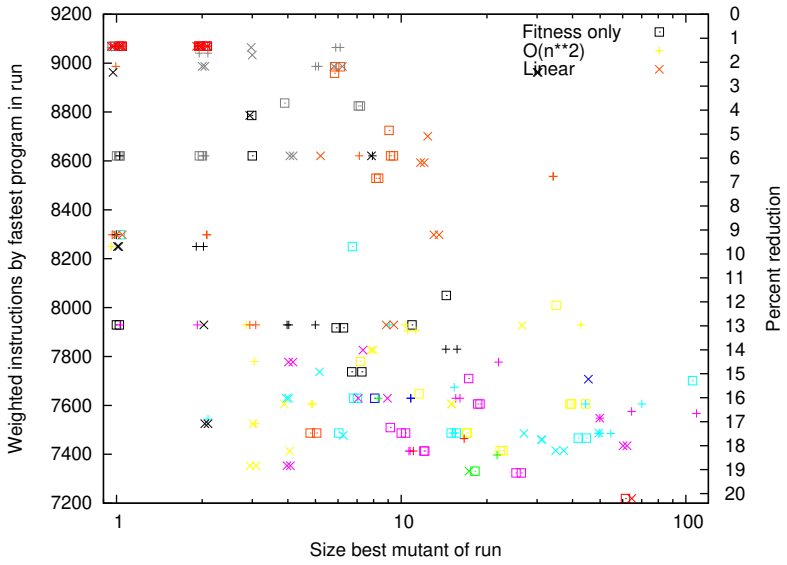


**Fig. 8.** Mutant gnome size v. fitness. Size and fitness of fastest triangle.c mutant found by generation 100 with populations 1, 2, 5, $\cdots$ up to 1000. Small horizontal noise added to spread data (same run colours as Figure 2). See Section 6.5.

```
@@ -14,9 +14,7 @@

      int triang ;

-    if( side1 <= 0 || side2 <= 0 || side3 <= 0){
-        return 4;
-    }
+

     triang = 0;

@@ -27,6 +25,9 @@
         triang = triang + 2;
     }
     if(side2 == side3){
+        if( side1 <= 0 || side2 <= 0 || side3 <= 0){
+            return 4;
+        }
         triang = triang + 3;
     }

@@ -45,6 +46,10 @@
         return 3;
     }
     else if ( triang == 1 && side1 + side2 > side3) {
+        if(side1 + side2 <= side3 ||
+ side2 + side3 <= side1 || side1 + side3 <= side2){
+            return 4;
+        }
         return 2;
     }
     else if (triang == 2 && side1 + side3 > side2){
```

**Fig. 9.** Example Magpie changes to triangle.c which reduces its (weighted) instruction count from 9069 to 7544 (17% improvement). Pink code removed, green inserted. (Initially 1300 bytes, mutant 1438 bytes.) See Section 7.

3.6 GHz Intel i7-4790 desktop. Naturally the O($n^2$) algorithm [36] scales badly with population size and in the worst case the time to select the parents with the largest population (1000) reaches almost two hours.

## 7 Discussion: Example Small High Fitness Mutant

Figure 9 shows a high scoring Magpie mutation as a C source code patch. The example is from run 6 with a population of 50 using O($n^2$) selection (we have deliberately chosen a small example to make explaining it easier). The mutated triangle.c is now larger (and so more difficult to compress). It was discovered in generation 15. As the run continued, evolution found similar mutations with identical scores containing the same genes (some repeated) plus others giving a still larger C source code. The Magpie genome for this mutation contains two

genes `StmtInsertion | StmtMoving`. Both `StmtInsertion` and `StmtMoving` mutations have two components: the XML `stmt` level code to be inserted or moved and the XML location where it is to be placed.

`StmtMoving` is perhaps the easiest to explain, it moves the compound `if` (XML `stmt 1`, C sources lines 17–19, shown with pink shading in Figure 9) and inserts it at XML `_inter_block 19` (before line 30 in triangle.c, central green shading in Figure 9). This means two of the three highly weighted tests for isosceles (return 2) and the even higher weight scalene (return 1) (see Table 1) do not incur the cost of checking for non-positive lengths, which correspond to not a valid triangle (return 4). However it also means risking missing some error conditions, which are not in the test suite.

The first mutation `StmtInsertion` copies another compound `if` from XML stmt 11 (lines 34–36 of triangle.c) and inserts it at XML `_inter_block 30` (before line 48 in triangle.c, last green shaded region in Figure 9). In the benchmark's test suite there are three tests with invalid zeros as input, notice that the now duplicated compound `if` detects them all. Thus, if because the initial tests for zero side length have been removed, execution reaches line 48, the duplicated code will still correctly detect the bad inputs and return 4. So the mutated compiled (with -O3) code gets a higher score by classifying important test cases earlier.

Notice how Magpie's operation on XML, effectively at the compiler AST level, means the mutations can easily operate with compound statements covering multiple lines and (except in a few odd ball cases) the mutant remains valid C code. (Even the indentation is correct.)

## 8 Conclusion

We have demonstrated how evolutionary algorithms can use information theory based population diversity alongside fitness selection. In the case of genetic improvement we used program source code, whereas Genetic Programming might use trees or instructions and Genetic Algorithms would use bit strings. Being compression based Cohen and Vitanyi's Normalised Information Distance (NID) in its single measure for a multiset (population) form (NCDm) would work with GP and GAs as well. We also invented a much faster linear version of Cohen and Vitanyi's quadratic approximation.

Apart from Pareto multi-objective combinations of our information and traditional fitness measures or systematically investigating more and weaker approximations to NID and NCDm in the hopes of a Goldilocks trade off, perhaps the next thing to consider is other ways to exploit this diversity measure. Population size is a confounding random variable for diversity so rather than theory that predicts convergence to optimal on the basis of population size [56] we could predict convergence on the basis of NCDm diversity diameter.

**Acknowledgements**

I am grateful for the assistance of Aymeric Blot, Dan Hoffman and Dan Blackwell.

Example C code in `https://github.com/wblangdon/linux_perf_api` etc.

## References


1. Goldberg, D.E.: Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley (1989)
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992), https://mitpress.mit.edu/9780262527910/genetic-programming/
3. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008), http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)
4. Langdon, W.B.: Genetic improvement of programs. In: Matousek, R. (ed.) 18th International Conference on Soft Computing, MENDEL 2012. Brno University of Technology, Brno, Czech Republic (27-29 Jun 2012), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2012_mendel.pdf, invited keynote
5. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation 19(1), 118–135 (Feb 2015), http://dx.doi.org/10.1109/TEVC.2013.2281544
6. Petke, J., Langdon, W.B., Harman, M.: Applying genetic improvement to MiniSAT. In: Ruhe, G., Yuanyuan Zhang (eds.) Symposium on Search-Based Software Engineering. Lecture Notes in Computer Science, vol. 8084, pp. 257–262. Springer, Leningrad (Aug 24-26 2013), http://dx.doi.org/10.1007/978-3-642-39742-4_21, short Papers
7. Petke, J., et al.: Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolutionary Computation 22(3), 415–432 (Jun 2018), http://dx.doi.org/doi:10.1109/TEVC.2017.2693219
8. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In: Nicolau, M., et al. (eds.) 17th European Conference on Genetic Programming. LNCS, vol. 8599, pp. 87–99. Springer, Granada, Spain (23-25 Apr 2014), http://dx.doi.org/10.1007/978-3-662-44303-3_8
9. Langdon, W.B., et al.: Genetic improvement of GPU software. Genetic Programming and Evolvable Machines 18(1), 5–44 (Mar 2017), http://dx.doi.org/10.1007/s10710-016-9273-9
10. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Transactions on Software Engineering 44(6), 574–594 (Jun 2018), http://dx.doi.org/10.1109/TSE.2017.2702606
11. Blot, A., Petke, J.: Empirical comparison of search heuristics for genetic improvement of software. IEEE Transactions on Evolutionary Computation 25(5), 1001–1011 (Oct 2021), http://dx.doi.org/10.1109/TEVC.2021.3070271
12. Mesecan, I., et al.: HyperGI: Automated detection and repair of information flow leakage. In: Khalajzadeh, H., Schneider, J.G. (eds.) The 36th IEEE/ACM International Conference on Automated Software Engineering, New Ideas and Emerging Results track, ASE NIER 2021. pp. 1358–1362. Melbourne (15-19 Nov 2021), http://dx.doi.org/10.1109/ASE51524.2021.9678758
13. Brownlee, A.E.I., et al.: Enhancing genetic improvement mutations using large language models. In: Arcaini, P., Tao Yue, Fredericks, E. (eds.) SSBSE 2023: Challenge Track. LNCS, vol. 14415, pp. 153–159. Springer, San Francisco, USA (8 Dec 2023), http://dx.doi.org/10.1007/978-3-031-48796-5_13

14. Pinna, G., et al.: Enhancing large language models-based code generation by leveraging genetic improvement. In: Giacobini, M., Bing Xue, Manzoni, L. (eds.) EuroGP 2024: Proceedings of the 27th European Conference on Genetic Programming. LNCS, vol. 14631, pp. 108–124. Springer, Aberystwyth (3-5 Apr 2024), http://dx.doi.org/10.1007/978-3-031-56957-9_7
15. Nemeth, Z., Faulkner Rainford, P., Porter, B.: Phenotypic species definitions for genetic improvement of source code. In: Faina, A., et al. (eds.) ALIFE 2024: Proceedings of the 2024 Artificial Life Conference. pp. 530–539. The International Society for Artificial Life, MIT Press, Copenhagen (Jul 22-26 2024), http://dx.doi.org/10.1162/isal_a_00795
16. Guizzo, G., et al.: Speeding up genetic improvement via regression test selection. ACM Transactions on Software Engineering and Methodology 33(8) (Nov 2024), http://dx.doi.org/10.1145/3680466
17. Brownlee, A.E.I., et al.: Large language model based mutations in genetic improvement. Automated Software Engineering 15, article number 15 (2025), http://dx.doi.org/10.1007/s10515-024-00473-6, special Issue on Advances in Search-Based Software
18. Blot, A., Petke, J.: A comprehensive survey of benchmarks for improvement of software's non-functional properties. ACM Computing Surveys (2025), https://discovery.ucl.ac.uk/id/eprint/10203326/1/main.pdf, in press
19. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology 43(14), 833–839 (Dec 2001), http://dx.doi.org/10.1016/S0950-5849(01)00189-6
20. Clark, D., Feldt, R., Poulding, S.M., Shin Yoo: Information transformation: An underpinning theory for software engineering. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. pp. 599–602. IEEE Computer Society (2015), http://dx.doi.org/10.1109/ICSE.2015.202
21. Feldt, R., Poulding, S.M., Clark, D., Shin Yoo: Test set diameter: Quantifying the diversity of sets of test cases. In: IEEE International Conference on Software Testing, Verification and Validation, ICST. pp. 223–233. Chicago, USA (April 11-15 2016), http://dx.doi.org/10.1109/ICST.2016.33
22. Tsong Yueh Chen, Fei-Ching Kuo, Merkel, R.G., Tse, T.H.: Adaptive random testing: The ART of test case diversity. Journal of Systems and Software 83(1), 60–66 (Jan 2010), http://dx.doi.org/10.1016/J.JSS.2009.02.022
23. Arcuri, A., Briand, L.C.: Adaptive random testing: an illusion of effectiveness? In: Dwyer, M.B., Tip, F. (eds.) Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011. pp. 265–275. ACM, Toronto, Canada (July 17-21 2011), http://dx.doi.org/10.1145/2001420.2001452
24. Anand, S., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86(8), 1978–2001 (August 2013), http://dx.doi.org/10.1016/j.jss.2013.02.061
25. Elgendy, I.T., Hierons, R.M., McMinn, P.: Evaluating string distance metrics for reducing automatically generated test suites. In: Lonetti, F., et al. (eds.) Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024). pp. 171–181. Lisbon, Portugal (April 15-16 2024), http://dx.doi.org/10.1145/3644032.3644455
26. Petke, J., Clark, D., Langdon, W.B.: Software robustness: A survey, a theory, and some prospects. In: Avgeriou, P., Dongmei Zhang (eds.) ESEC/FSE 2021, Ideas, Visions and Reflections. pp. 1475–1478. ACM, Athens, Greece (23-28 Aug 2021), http://dx.doi.org/10.1145/3468264.3473133

27. Kosorukov, I., et al.: Mining for mutation operators for reduction of information flow control violations. In: IEEE/ACM International Conference on Automated Software Engineering, The New Ideas and Emerging Results (ASE-NIER 2024). Sacramento (24 Oct 27-Nov 1 2024), http://dx.doi.org/10.1145/3691620.3695308
28. Goldberg, D.E.: Genetic algorithms and Walsh functions: Part II, deception and its analysis. Complex Systems 3(2), 153–171 (1989), https://www.complex-systems.com/abstracts/v03_i02_a03/
29. Grefenstette, J.J.: Deception considered harmful. In: Whitley, L.D. (ed.) Foundations of Genetic Algorithms 2. pp. 75–91. Morgan Kaufmann, Vail, Colorado, USA (26-29 July 1992), http://dx.doi.org/10.1016/B978-0-08-094832-4.50011-8
30. Ochoa, G., Veerapen, N.: Mapping the global structure of TSP fitness landscapes. Journal of Heuristics 24(3), 265–294 (2018), http://dx.doi.org/10.1007/S10732-017-9334-0
31. Langdon, W.B., Veerapen, N., Ochoa, G.: Visualising the search landscape of the Triangle program. In: Castelli, M., McDermott, J., Sekanina, L. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 96–113. Springer, Amsterdam (19-21 Apr 2017), http://dx.doi.org/10.1007/978-3-319-55696-3_7
32. Veerapen, N., Daolio, F., Ochoa, G.: Modelling genetic improvement landscapes with local optima networks. In: Petke, J., White, D.R., Langdon, W.B., Weimer, W. (eds.) GI-2017. pp. 1543–1548. ACM, Berlin (15-19 Jul 2017), http://dx.doi.org/10.1145/3067695.3082518, best presentation prize
33. Veerapen, N., Ochoa, G.: Visualising the global structure of search landscapes: genetic improvement as a case study. Genetic Programming and Evolvable Machines 19(3), 317–349 (Sep 2018), http://dx.doi.org/10.1007/s10710-018-9328-1, special issue on genetic programming, evolutionary computation and visualization
34. Petke, J., et al.: A survey of genetic improvement search spaces. In: Alexander, B., Haraldsson, S.O., Wagner, M., Woodward, J.R. (eds.) 7th edition of GI @ GECCO 2019. pp. 1715–1721. ACM, Prague, Czech Republic (Jul 13-17 2019), http://dx.doi.org/10.1145/3319619.3326870
35. Langdon, W.B., Bruce, B.R.: The gem5 C++ glibc heap fitness landscape. In: Blot, A., Nowack, V., Faulkner Rainford, P., Krauss, O. (eds.) 14th International Workshop on Genetic Improvement @ICSE 2025. Ottawa (27 Apr 2025), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2025_GI.pdf, forthcoming
36. Cohen, A.R., Vitanyi, P.M.B.: Normalized compression distance of multisets with applications. IEEE Transactions on Pattern Analysis and Machine Intelligence 37(8), 1602–1614 (2015), http://dx.doi.org/10.1109/TPAMI.2014.2375175
37. Vitanyi, P.M.B., Balbach, F.J., Cilibrasi, R.L., Ming Li: Normalized information distance. In: Emmert-Streib, F., Dehmer, M. (eds.) Information Theory and Statistical Learning, chap. 3, pp. 45–82. Springer (2009), http://dx.doi.org/10.1007/978-0-387-84816-7_3
38. Cilibrasi, R., Vitanyi, P.M.B.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (Apr 2005), http://dx.doi.org/10.1109/TIT.2005.844059
39. Sapna, P.G., Mohanty, H.: Automated test scenario selection based on Levenshtein distance. In: Janowski, T., Mohanty, H. (eds.) $6^{th}$ Distributed Computing and Internet Technology (ICDCIT'10), Lecture Notes in Computer Science (LNCS), vol. 5966, pp. 255–266. Springer, Bhubaneswar, India (February 15-17 2010), http://dx.doi.org/10.1007/978-3-642-11659-9_28

40. Sakal, J., Fieldsend, J., Keedwell, E.: Genotype diversity measures for escaping plateau regions in university course timetabling. In: Thomson, S.L., et al. (eds.) Workshop on Landscape-Aware Heuristic Search (LAHS 2022). pp. 2090–2098. GECCO '23, Association for Computing Machinery, Lisbon, Portugal (15-19 July 2023), http://dx.doi.org/10.1145/3583133.3596334
41. Elgendy, I.T., Hierons, R.M., McMinn, P.: A survey of the metrics, uses, and subjects of diversity-based techniques in software testing. ArXiv (16 Nov 2023), https://arxiv.org/abs/2311.09714
42. Johnson, C.G., Woodward, J.R.: Information theory, fitness, and sampling semantics. In: Johnson, C., Krawiec, K., Moraglio, A., O'Neill, M. (eds.) Semantic Methods in Genetic Programming. Ljubljana, Slovenia (13 Sep 2014), https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1bdff27d8e4dbc6321bef2aab06feb13f642b977, workshop at Parallel Problem Solving from Nature 2014 conference
43. Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I.: The use of automatic test data generation for genetic improvement in a live system. In: Galeotti, J.P., Petke, J. (eds.) Search-Based Software Testing. pp. 28–31. IEEE/ACM, Buenos Aires, Argentina (22-23 May 2017), http://dx.doi.org/10.1109/SBST.2017.10
44. Blickle, T., Thiele, L.: A comparison of selection schemes used in evolutionary algorithms. Evolutionary Computation 4(4), 361–394 (Winter 1996), http://dx.doi.org/10.1162/evco.1996.4.4.361
45. Ramamoorthy, C.V., Siu-Bun F. Ho, Chen, W.T.: On the automated generation of program test data. IEEE Transactions on Software Engineering 2(4), 293–300 (December 1976), http://dx.doi.org/10.1109/TSE.1976.233835
46. Langdon, W.B., Harman, M., Yue Jia: Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software 83(12), 2416–2430 (Dec 2010), http://dx.doi.org/10.1016/j.jss.2010.07.027
47. Blot, A., Petke, J.: Comparing genetic programming approaches for non-functional genetic improvement case study: Improvement of MiniSAT's running time. In: Ting Hu, Lourenco, N., Medvet, E. (eds.) EuroGP 2020: Proceedings of the 23rd European Conference on Genetic Programming. LNCS, vol. 12101, pp. 68–83. Springer Verlag, Seville, Spain (15-17 Apr 2020), http://dx.doi.org/10.1007/978-3-030-44094-7_5
48. Blot, A., Petke, J.: Using genetic improvement to optimise optimisation algorithm implementations. In: Hadj-Hamou, K. (ed.) 23ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, ROADEF'2022. INSA Lyon, Villeurbanne - Lyon, France (23–25 Feb 2022), https://hal.archives-ouvertes.fr/hal-03595447
49. Langdon, W.B., Clark, D.: Deep imperative mutations have less impact. Automated Software Engineering 32, article number 6 (2025), http://dx.doi.org/10.1007/s10515-024-00475-4
50. Blot, A., Petke, J.: MAGPIE: Machine automated general performance improvement via evolution of software. arXiv (4 Aug 2022), http://dx.doi.org/10.48550/arxiv.2208.02811
51. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Francisco, CA, USA (Jan 1998), https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X

52. Brameier, M., Banzhaf, W.: Linear Genetic Programming. No. XVI in Genetic and Evolutionary Computation, Springer (2007), http://dx.doi.org/10.1007/978-0-387-31030-5
53. Langdon, W.B., Banzhaf, W.: Repeated sequences in linear genetic programming genomes. Complex Systems 15(4), 285–306 (2005), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_repeat_linear.pdf
54. Langdon, W.B.: Genetic programming convergence. Genetic Programming and Evolvable Machines 23(1), 71–104 (Mar 2022), http://dx.doi.org/10.1007/s10710-021-09405-9
55. Langdon, W.B., Poli, R.: Fitness causes bloat. In: Chawdhry, P.K., Roy, R., Pant, R.K. (eds.) Soft Computing in Engineering Design and Manufacturing. pp. 13–22. Springer-Verlag London (23-27 Jun 1997), http://dx.doi.org/10.1007/978-1-4471-0427-8_2
56. Schmitt, L.M.: Theory of genetic algorithms. Theoretical Computer Science 259(1-2), 1–61 (28 May 2001), http://dx.doi.org/10.1016/S0304-3975(00)00406-0