EuroGP-2017, M. Castelli and J. McDermott and L. Sekanina *Eds.*, LNCS, Amsterdam, 19-21 April, Springer. Publisher policy allows this work to be made available in this repository; The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-55696-3_7

Visualising the Search Landscape of the Triangle Program

William B. Langdon and Nadarajen Veerapen and Gabriela Ochoa

CREST, Computer Science, UCL, London, WC1E 6BT, UK
Computing Science and Mathematics, University of Stirling, FK9 4LA, UK

Abstract. High order mutation analysis of a software engineering benchmark, including schema and local optima networks, suggests program improvements may not be as hard to find as is often assumed. 1) Bitwise genetic building blocks are not deceptive and can lead to all global optima. 2) There are many neutral networks, plateaux and local optima, nevertheless in most cases near the human written C source code there are hill climbing routes including neutral moves to solutions.

Keywords genetic improvement, genetic algorithms, genetic programming, software engineering, heuristic methods, test equivalent higher order mutants, fitness landscape, local search

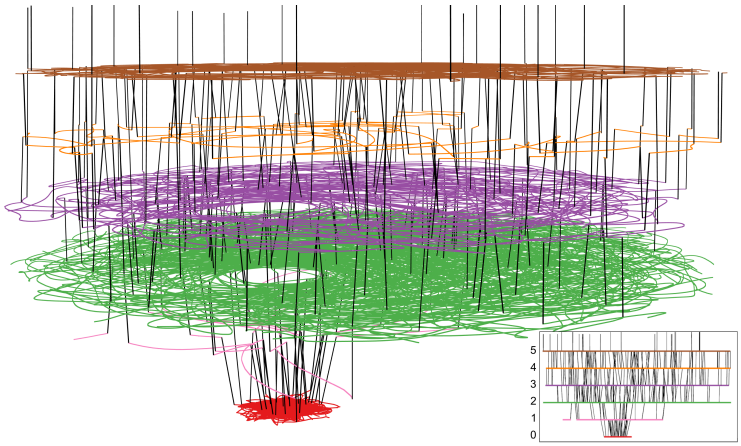

Fig. 1. Local optima network of the Triangle Program using 100 random starts (see Section 4.4). Edges are coloured if they start and end at the same fitness. Insert shows fitness levels edge on. Best (bottom) red 0 (pass all tests), pink 1 (fail only one test), green 2, purple 3, orange 4, brown 5.

1 Genetic Improvement

Genetic Improvement [1,2,3] can be thought of as the use of Search Based Software Engineering (SBSE) [4] techniques, principally genetic programming [5,6,7], to the optimisation of existing (human written) software. GI is often applied

to non-functional properties of software but perhaps it is most famous for improving program's functionality, e.g. by removing bugs [8,9,10,11,12,13,14,15,16] or adding to its abilities [17,18,19,20,21,22]. Non-functional improvements that have been considered or results reported include: faster code [23,24], code which uses less energy [25,26,27,28,29,30,31,32,33,34] or less memory [35], and automatic parallelisation [36,37,38] and automatic porting [39] and embedded systems [40,41,25,42,43,44,45] as well as refactorisation [46], reverse engineering [47,48] and software product lines [49,50]. There is very much a GI flavour in the air with a three-fold increase in GI publications (as measured by GI papers in the genetic programming bibliography) since the first GI workshop [51] was first mooted (October, 7 2014)[1]. Nonetheless there remains a deal of scepticism in software engineering circles.

One criticism of genetic improvement is that the results are empirical and there is little theoretical underpinning [52]. One of the prejudices holding back software engineering is the assumption that software is fragile. It has been shown in a small number of cases that this fear has been overplayed. Whilst many mutations are highly deleterious, in the few cases reported, many others are not [53]. If presented differently, this idea is not news to software engineers: the failure of software engineering to widely adopt mutation testing [54] is in part due to the presence of many equivalent mutants. But an equivalent mutant is simply a mutation which has no effect, which is exactly what GI has reported! Until recently [55] mutation testing rarely considered more than one change to the source code at a time, whereas GI typically allows high order mutations (which make multiple changes simultaneously). We consider up to 17^{th} order mutations.

Mostly genetic improvement considers mutations to source code (typically C, C++ or Java [56]) but similar empirical results have been reported at byte code [57], assembly [58] and indeed machine code [59,60] levels.

Due to the dearth of theoretical genetic improvement analysis, we shall study the GI search landscape [61,62]. Although small (39 lines of C code) we chose the Triangle Program as it is a well known software engineering benchmark and we have used it with mutation testing [55]. Briefly, in mutation testing [54] errors (mutations) like those a human programmer might make are deliberately injected into the program to see if the program's test suite can detect them. Our mutations were to replace numeric comparison operators, e.g., replace `==` with `<=`. In the next section we describe a cut down version where there are only two choices for each of the seventeen comparison sites in the Triangle Program. Section 3 presents a schema [63] analysis of the simplified landscape which shows *none* of the high order schema are deceptive [64]. This suggests that it might be easy for a genetic algorithm (GA) to find solutions. (Section 4.2 adds to this by showing none are strongly deceptive when larger moves are allowed.) We also see (Section 3.3) that there are large plateaux where neighbours have identical fitness.

[1] `http://geneticimprovementofsoftware.com/?page_id=13` (accessed Oct, 9 2016)

Section 4 returns to allowing all six possible C numeric comparison operations and shows that although solutions are extremely rare, the vastly increased search space is still easy for genetic improvement in the sense that a local search hill climbing algorithm can reach a global optima from almost any low order mutation provided neutral moves are allowed. In contrast search from a random point seldom finds a program that can pass all the test cases. This supports the idea that genetic improvement, where search may start near the good part of the search space, is easier than GP, where search may start from a random point.

2 Triangle Program Software Engineering Benchmark

The Triangle Program is well studied software engineering benchmark. It can be thought of as a model of unit testing. It classifies triangles as scalene, isosceles, equilateral or not a triangle. We have previously used it to study high order mutation, concentrating particularly on injecting faults which change the numeric comparison operators (`<`, `<=`, `==`, `!=`, `=>` and `>`) [55]. We now consider mutation of all 17 of the comparison operators in the Triangle Program as a genetic algorithm fitness landscape. Taking as our fitness the number of tests [55, Tab. 2] which the modified code fails. A test equivalent mutant is one that passes all the tests and so has the best fitness value, which is zero. We consider all possible simultaneous changes. For the Triangle Program there are $6^{17} - 1 = 16\,926\,659\,444\,735$ mutations, of which 9215 are test equivalent, i.e. pass all the test cases.

3 Binary Representation: Replacing Comparisons with One Alternative

At first, instead of allowing all possible combinations, we study allowing each numerical comparison in the Triangle Program to be replaced by only one other. Table 1 is taken from [55]. It shows hard to detect mutations of the Triangle Program. The source code of the unmutated Triangle Program contains only `<=`, `==` and `>` comparisons. The last three lines of Table 1 gives their replacements in the commonest lower order hard to detect mutations. In mutation testing these mutations are known as the hardest to "kill". Therefore the substitutions given in last three lines of Table 1 are the ones we use. Since there are six comparison operators and 17 potential mutation sites, this reduces the search space from 6^{17} to 2^{17}. (We shall return to the original problem in Section 4.) We evaluate all possible mutants.

3.1 High Order Binary Schema are Not Deceptive

There are 2048 global optima (shown in white in Figure 2). On average each mutant fails only 4.344 $\pm$1.360 tests. The worst mutant only fails six of the 14 tests (Figure 3).

Table 1. Hardest to detect mutations of the Triangle Program [55, Fig. 3]. The first column contains the number of times the individual changes shown appear in 1^{st}, 2^{nd}, 3^{rd} and 4^{th} order test equivalent mutations.

354	== replaced by >=
576	<= replaced by <
708	== replaced by <=
1062	> replaced by !=
1992	<= replaced by ==

Table 2. Mean and standard deviation of number of tests failed for highest order binary schema of the Triangle Program (excluding 22 with average means). Last column is estimated population size needed for a random sample to distinguish between competing pairs of schema.

-4	3.719 ±1.328	1.9
4	4.969 ±1.075	
-5	4.062 ±1.478	4.7
5	4.625 ±1.166	
-6	3.812 ±1.509	2.4
6	4.875 ±0.927	
-11	3.438 ±1.273	1.1
11	5.250 ±0.661	
-14	4.312 ±1.424	43.5
14	4.375 ±1.293	
-16	4.188 ±1.550	8.6
16	4.500 ±1.118	

Of the 34 high order schema[2], 22 have exactly average fitness and contain exactly half the global optima. The other 12 schema either contain no solutions or all of them. In the six schema which contain solutions, on average individuals are better than the average of the whole space. In the other six, the schema average is worse than the average of the whole space. That is, 22 schema have no signal and the remaining 12 are not deceptive. In the best schema (i.e. -11) mutants pass on average 1.813 ±1.015 more tests than its opposite (11) (see also Table 2).

3.2 Binary Schema Predict *All* Solutions of the Triangle Program

As the previous section showed, there are twenty two 16-order schema that have exactly average fitness. These correspond to $22/2 = 11$ variable gene locations. I.e. locations of *s, where either alternative can be used. Together they can be represented as a $17 - 11 = 6^{th}$ order schema by taking their union. This sixth order schema is shown in Table 3. These 11 * (don't cares) give 2048 combinations ($2^{11} = 2048$) each of which is one of the solutions!

An alternative way of looking at this is, once we fix the six mutation sites in the C source code corresponding to the better than average schema in Table 2 we are free to mutate all the others (using our restricted mutation operator, last three rows of Table 1) and the new program will return the correct answer for all of the tests (Table 3).

[2] A 16^{th} order schema has 16 defined positions [64, page 29], and one variable * position (length = 17). Whereas a 1^{st} order mutation is identical to the original except for one change.

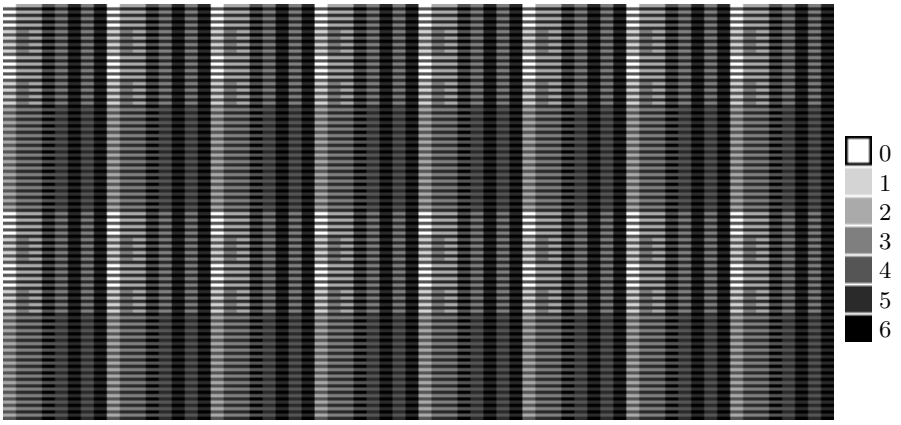

Fig. 2. Fitness landscape of binary comparison improvement of Triangle Program. First 9 bits (512) horizontal, last 8 bits (256) vertical. 2048 test equivalent mutants (fitness=0) in white. The regular pattern of individuals with the same fitness indicates short building blocks. E.g. the vertical strips 8 pixels wide indicates the first three bits do not impact fitness. In contrast the last but one bit divides the figure into four horizontal stripes, two contain 50 176 mutants which fail 4 or more tests (dark pixels) whilst the others hold all the solutions (white). Fitness distance correlation is 0.45

Table 3. 6^{th} order binary schema giving 2048 test equivalent mutations of the Triangle Program. * indicates don't care but a 0 (or 1) means that only the one numeric comparison shown in the next row can be used. The bottom two rows gives the 2^{11} alternatives (for the $17 - 6 = 11$ *s) which pass all the tests.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Schema	*	*	*	0	0	0	*	*	*	*	0	*	*	0	*	0	*
Code	<=	<=	<=	==	==	==	==	<=	<=	<=	>	==	>	==	>	==	>
	==	==	==				<=	==	==	==		<=	!=		!=		!=

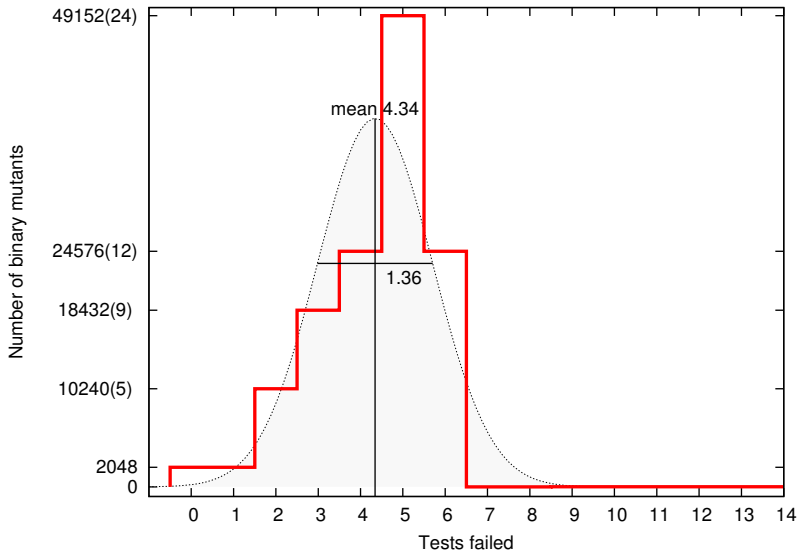

Fig. 3. Fitness distribution in the binary comparison version of the Triangle Program. Gaussian fit to mean and standard deviation shown in background.

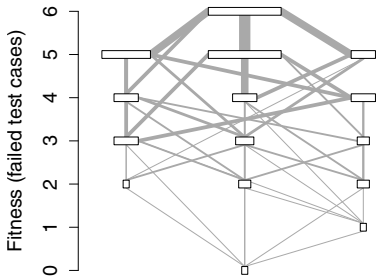

Fig. 4. Plateaux and their connections in the landscape of the binary comparison version of the Triangle Program.

4 Original All Comparisons

Sections 2 and 3 described how in [52] we simplified the Triangle Program Software Engineering benchmark to ease analysis of its schema and fitness landscape. From here on we return to the original formulation [55]. Next we evaluate all possible mutations (Figures 5–8). In Section 4.2 we repeat the binary schema analysis and find when larger moves are allowed there may be deceptive schema. In Section 4.3 we run a local hill climber both, from every part of the search space near the original program, and from samples of higher order mutations, and show that improvements are easy to find. Finally in Section 4.4 we use another local iterated search, which alternates between hillclimbing and random moves, to map the local optima network, see also Figure 1 (page 1).

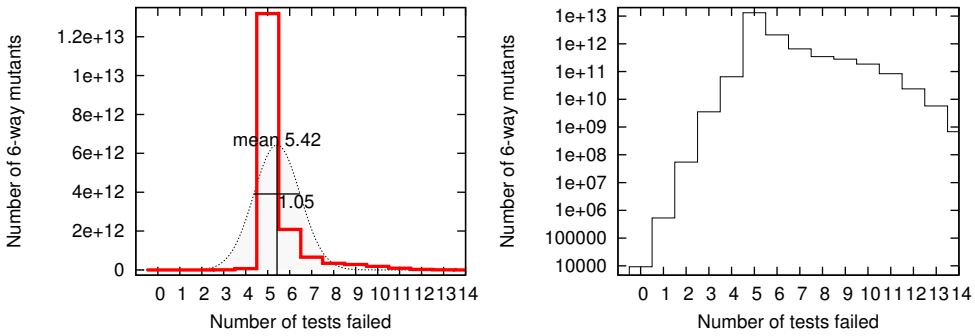

Fig. 5. Fitness distribution of all 16 926 659 444 735 possible comparison mutations of the Triangle Program. Note most (78%) mutants fail five tests which corresponds with the peak in the two options only subspace, Figure 3. Right: same data plotted on log scale. Considering all six comparison operations increases the number of changes which still pass the whole test suite from 2048 to 9215.

4.1 Fitness Space of Triangle Program

We evaluated the whole of the search space. If we compare the results in Figure 5 with the same data for the binary subset (Figure 3) we see:

- Firstly the space is vastly bigger.
- The number of global solutions increases by a factor of 4.5 to 9215. However as a fraction of the total search space it becomes tiny ($5\,10^{-10}$) so random or brute force search are ineffective.
- However the overall shape of the distribution of fitness values of high order mutations remains similar. (The mean increases slightly from 4.34 to 5.42 and the standard deviations are similar, 1.36 v. 1.05 in the full search space.) Note, for example, in both cases there is a large peak of mutated programs that fail exactly five tests. Indeed most mutants still pass most tests.

  As we shall see in Section 4.3, the large fraction of the search space with fitness of exactly five contributes to a huge neutral network. That is, there are many neighbouring programs (i.e. they differ in exactly one comparison) which fail exactly the same number of tests.
- The fraction of mutants which fail more than half the tests remains low (albeit finite rather than zero). Indeed only 40 in a million randomly sampled mutations fail all fourteen tests (right end side Figure 5).
- Considering only first order mutations (Figure 6), 16% pass all the tests. Indeed, as with the smaller search space (shown with dashed line in Figure 6) most programs with only one change fail no more than two tests.

4.2 High Order Schema Analysis

There are $17 \times 6 = 102$ 16-order schema. We estimated their fitness by randomly sampling each one a million times. The results are given in Table 4 and Figure 7. Whereas when considering only two options (Section 3.1, Table 2) in eleven of the

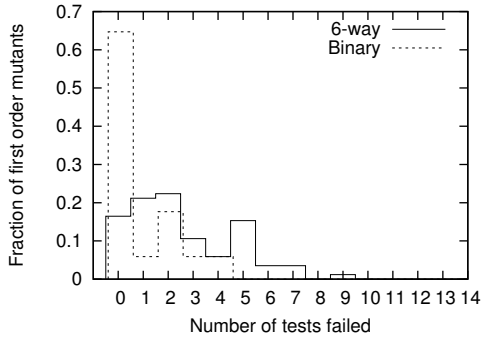

Fig. 6. Fitness distribution of all 85 possible first order comparison mutations of the Triangle Program. (I.e. rescaled leftmost line of Figure 8) Corresponding data for the binary version of the search space shown plotted with dashed line for comparison.

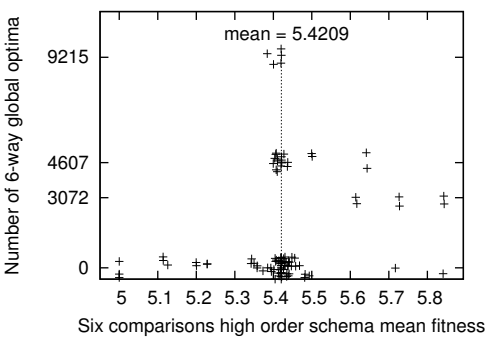

Fig. 7. Fitness of all 102 high order schema and the number of solutions they contain for the Triangle Program with six comparison mutations. Schema contain either none, 1/3, 1/2 or all the solutions. Vertical noise added to separate data. All 16-schema containing a solution have fitness $>$ mean $-$ 0.0368.

17 mutation sites both alternative schema had the same average fitness, in only three locations (4, 5 and 6) might all six schema be said to be within sampling noise of the mean fitness of the whole space. Now that we are considering moves that take us further from the original code, we do find weakly deceptive schema. For locations 11, 12, 14, 15 and 17, although none of the $5 \times 6 = 30$ schema differ strongly from the average, all of the solutions occur in *below* average schema. (See cluster in centre of Figure 7.) Also in three more locations (3, 8 and 9), although one or more schema containing solutions are above average, the strongest schema does not contain any solutions. Only in the first two locations is there a strong signal (i.e. > 0.1) leading to any of the solutions. In the remaining four locations (7, 10, 13 and 16) there are solutions in above average schema, but there isn't a strong signal leading to any of them.

4.3 Local Search for the Triangle Program

Since all first order mutations are by definition one move away from a solution, in all cases it is possible to hillclimb from any 1^{st} order mutation to a solution (Figure 9). In the case of 2^{nd} order mutations, a hill climber can find a program which passes all the test cases in all but two cases (both these local optima fail two tests). For third order there are 133 (0.15% of 3^{rd} order mutations) and for 4^{th} 3623 (0.24%) points in the search space from which a solution cannot be reached by hill climbing. For 5^{th} and 6^{th} its about 4%, after which the fraction of higher order mutations from which a solution can be reached progressively falls towards zero. Even so in most cases a program which fails only one or two test cases can be found by hill climbing. I.e., in almost all cases hill climbing can improve a mutant from failing five test cases to failing two (Figures 8 and 9).

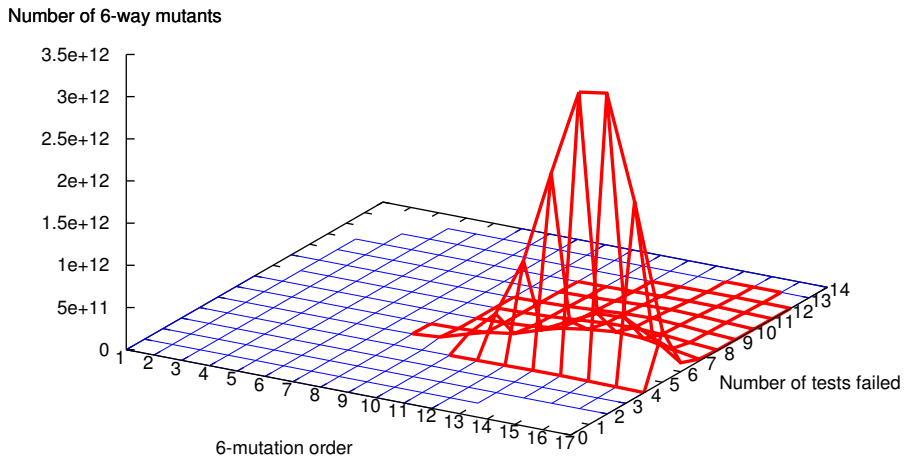

Fig. 8. Fitness distribution of all 16 926 659 444 735 possible comparison mutations of the Triangle Program. (Summary data plotted in Figure 5.) Values above 10^9 plotted in bold. 39% of mutations are either 14^{th} or 15^{th} order which fail five tests, whilst the fitness distance correlation is near zero at -0.070837.

4.4 Local Optima Networks

Local optima networks are a compact representation of fitness landscapes that can be used for analysis and visualisation [66]. A solution is a *local optimum* if none of its neighbours have a better fitness value. A full enumeration of the local optima for the Triangle Program is clearly unmanageable. The networks are therefore based on a sample of high-quality local optima in the search space. This is much more of a practical search algorithm than in the previous section, where we treat finding any hill climbing route to a solution as a problem of searching a directed graph and so we include backing up and trying again if it appears no forward progress is possible.

The sampling algorithm is an Iterated Local Search (ILS) which starts from a locally optimal solution and then alternates between a random mutation and a best-improvement hill-climber. The termination criterion is a fixed number of iterations. At each hill climbing step, only non-worsening local moves are accepted. Both the hill-climber and mutation consider the immediate neighbourhood, i.e, a program which differs only in a single comparison.

Edges are directed and based on the mutation operation. There is an *escape edge* from local optimum i to local optimum j if j is obtained from the mutation of i followed by hill-climbing. The *local optima network* is the graph where the nodes are the local optima and the edges are the escape edges.

To sample the local optima, we ran an ILS, from 1000 random starting points, with a budget of 10 000 iterations. This generated 2 372 805 unique local minima. Figure 10 presents the proportion of those runs that find a solution that passes all test cases. Figure 10 shows that in many cases the landscape can be easily traversed to reach solutions that pass all test cases.

Table 4. Average fitness of high order schema and number of solutions they contain for all six comparisons in the Triangle Program. See also Figure 7

Schema		mean	sd	above average		Sol
1	<	5.843567	1.423400	0.4227	±0.0014	3072
1	<=	5.727581	1.349277	0.3067	±0.0013	3071
1	==	5.726547	1.349000	0.3056	±0.0013	3072
1	!=	5.114173	0.318021	-0.3067	±0.0003	0
1	>=	5.000000	0.000000	-0.4209	±0.0000	0
1	>	5.113825	0.317599	-0.3071	±0.0003	0
2	<	5.842667	1.471239	0.4218	±0.0015	3072
2	<=	5.614215	1.319723	0.1933	±0.0013	3071
2	==	5.616813	1.323260	0.1959	±0.0013	3072
2	!=	5.228102	0.473518	-0.1928	±0.0005	0
2	>=	5.000000	0.000000	-0.4209	±0.0000	0
2	>	5.228345	0.473535	-0.1926	±0.0005	0
3	<	5.840933	1.576470	0.4200	±0.0016	0
3	<=	5.499540	1.233664	0.0786	±0.0012	4607
3	==	5.500957	1.236198	0.0801	±0.0012	4608
3	!=	5.343792	0.582205	-0.0771	±0.0006	0
3	>=	5.000000	0.000000	-0.4209	±0.0000	0
3	>	5.342161	0.582014	-0.0787	±0.0006	0
4	<	5.421830	1.059801	0.0009	±0.0011	0
4	<=	5.421768	1.041495	0.0009	±0.0010	0
4	==	5.419896	1.025397	-0.0010	±0.0010	9215
4	!=	5.422214	1.078773	0.0013	±0.0011	0
4	>=	5.421401	1.050374	0.0005	±0.0011	0
4	>	5.420884	1.038998	-0.0000	±0.0010	0
5	<	5.421382	1.060194	0.0005	±0.0011	0
5	<=	5.421160	1.033338	0.0003	±0.0010	0
5	==	5.420832	1.012703	-0.0001	±0.0010	9215
5	!=	5.420481	1.085840	-0.0004	±0.0011	0
5	>=	5.420653	1.051449	-0.0002	±0.0011	0
5	>	5.419979	1.044165	-0.0009	±0.0010	0
6	<	5.419947	1.040180	-0.0010	±0.0010	0
6	<=	5.418500	1.017661	-0.0024	±0.0010	0
6	==	5.420138	1.021691	-0.0008	±0.0010	9215
6	!=	5.421374	1.094645	0.0005	±0.0011	0
6	>=	5.420922	1.058717	0.0000	±0.0011	0
6	>	5.420671	1.048630	-0.0002	±0.0010	0
7	<	5.716858	1.387539	0.2960	±0.0014	0
7	<=	5.641402	1.279383	0.2205	±0.0013	4608
7	==	5.643428	1.283482	0.2225	±0.0013	4607
7	!=	5.200235	0.626614	-0.2207	±0.0006	0
7	>=	5.125798	0.485354	-0.2951	±0.0005	0
7	>	5.199680	0.624128	-0.2212	±0.0006	0
8	<	5.482451	1.101441	0.0616	±0.0011	0
8	<=	5.437518	1.066325	0.0166	±0.0011	4607
8	==	5.436342	1.063603	0.0154	±0.0011	4608
8	!=	5.406059	1.027887	-0.0148	±0.0010	0
8	>=	5.358139	0.993148	-0.0628	±0.0010	0
8	>	5.404727	1.026829	-0.0162	±0.0010	0
9	<	5.467923	1.101697	0.0470	±0.0011	0
9	<=	5.406438	1.051033	-0.0145	±0.0011	4607
9	==	5.419790	1.052137	-0.0011	±0.0011	4608
9	!=	5.420790	1.035139	-0.0001	±0.0010	0
9	>=	5.373288	0.997365	-0.0476	±0.0010	0
9	>	5.437700	1.045860	0.0168	±0.0010	0
10	<	5.481349	1.104830	0.0604	±0.0011	0
10	<=	5.422098	1.055740	0.0012	±0.0011	4607
10	==	5.421187	1.056004	0.0003	±0.0011	4608
10	!=	5.419925	1.032951	-0.0010	±0.0010	0
10	>=	5.357749	0.994258	-0.0632	±0.0010	0
10	>	5.420015	1.033797	-0.0009	±0.0010	0
11	<	5.385125	0.943758	-0.0358	±0.0009	0
11	<=	5.455657	1.102885	0.0348	±0.0011	0
11	==	5.349893	0.897228	-0.0710	±0.0009	0
11	!=	5.493383	1.185560	0.0725	±0.0012	0
11	>=	5.457862	1.142620	0.0370	±0.0011	0
11	>	5.384137	0.979864	-0.0368	±0.0010	9215
12	<	5.401587	1.013621	-0.0193	±0.0010	0
12	<=	5.407264	1.023709	-0.0136	±0.0010	4608
12	==	5.399789	1.014724	-0.0211	±0.0010	4607
12	!=	5.440903	1.078283	0.0200	±0.0011	0
12	>=	5.439726	1.081894	0.0188	±0.0011	0
12	>	5.435198	1.072511	0.0143	±0.0011	0
13	<	5.393600	1.005139	-0.0273	±0.0010	0
13	<=	5.420620	1.052961	-0.0003	±0.0011	0
13	==	5.419422	1.051208	-0.0015	±0.0011	0
13	!=	5.421373	1.044389	0.0005	±0.0010	4608
13	>=	5.446829	1.087792	0.0259	±0.0011	0
13	>	5.421251	1.042247	0.0004	±0.0010	4607
14	<	5.407715	1.022905	-0.0132	±0.0010	0
14	<=	5.413683	1.033827	-0.0072	±0.0010	0
14	==	5.400401	1.016918	-0.0205	±0.0010	9215
14	!=	5.441626	1.079657	0.0207	±0.0011	0
14	>=	5.433470	1.069622	0.0126	±0.0011	0
14	>	5.426515	1.057726	0.0056	±0.0011	0
15	<	5.394988	1.007207	-0.0259	±0.0010	0
15	<=	5.429938	1.064344	0.0090	±0.0011	0
15	==	5.430083	1.064206	0.0092	±0.0011	0
15	!=	5.410615	1.028593	-0.0103	±0.0010	4608
15	>=	5.448421	1.090379	0.0275	±0.0011	0
15	>	5.410818	1.030977	-0.0101	±0.0010	4607
16	<	5.414233	1.035495	-0.0067	±0.0010	0
16	<=	5.426977	1.059495	0.0061	±0.0011	0
16	==	5.407243	1.030172	-0.0137	±0.0010	4607
16	!=	5.434739	1.066421	0.0138	±0.0011	0
16	>=	5.427646	1.060224	0.0067	±0.0011	4608
16	>	5.414871	1.035411	-0.0060	±0.0010	0
17	<	5.404515	1.012176	-0.0164	±0.0010	0
17	<=	5.438065	1.070895	0.0172	±0.0011	0
17	==	5.430419	1.063981	0.0095	±0.0011	0
17	!=	5.410146	1.028519	-0.0108	±0.0010	4608
17	>=	5.439023	1.080056	0.0181	±0.0011	0
17	>	5.403619	1.026794	-0.0173	±0.0010	4607

In Figure 11, the fitness and Hamming distance of each local minima to the unmutated Triangle Program is presented as a sunflower plot. We can observe that solutions need to maintain some similarity to the original program to pass all test cases. In addition, almost all local optima pass $> 50\%$ of the test cases.

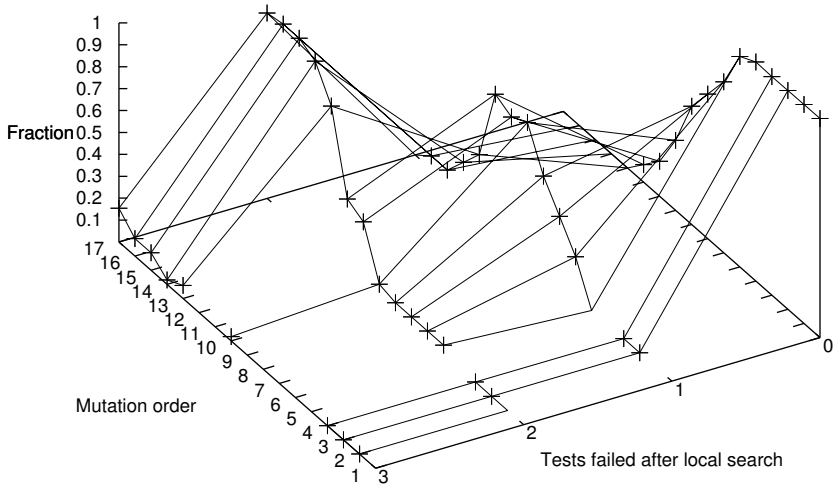

Fig. 9. Best fitness reached by local search hillclimber starting from high order mutations. + non-zero value. Up to 9^{th} order mutations, in most cases a program which passes all the tests can be found. (Data by enumerating 1–4^{th} order mutations and sampling others, more than 1.5 10^{12} fitness evaluations.)

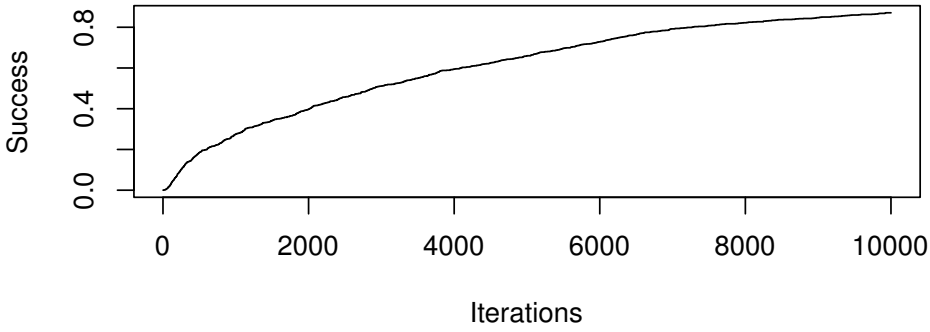

Fig. 10. Proportion of 1000 Iterated Local Search (ILS) runs that find a solution that passes all test cases versus the number of iterations.

Figure 1 (page 1) shows the local optima network obtained from observing a subset of the sampling described before. We only consider 100 ILS runs and their first 1000 iterations. Since this process generated more than 60 000 nodes and edges, only the edges are plotted for the sake of clarity. Edges are coloured if they start and end at the same fitness. Other edges are painted black. The points are positioned in the x-y plane using a force-directed layout algorithm and the fitness is used for the z axis [67]. The fact that many nodes are at the same level does not necessarily mean that they are part of the same plateau. Nevertheless, we can observe that the fitness levels for 2, 3 and 5 failed test cases are densely populated. In addition, the numerous clear paths between fitness levels provide visual evidence of the relative ease with which the network can be traversed.

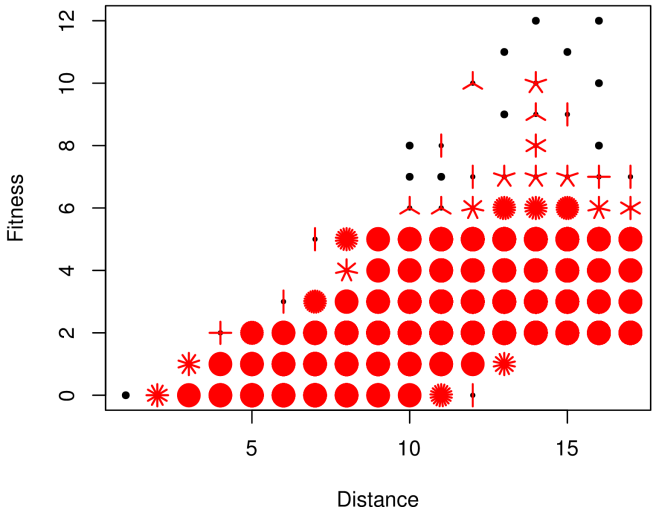

Fig. 11. Fitness vs Hamming distance of 2 372 805 unique local minima presented as a sunflower plot. The number of petals is proportional to the number of points at each coordinate. Note positive correlation between local optima fitness and distance to the original program. Also many mutants that differ from the original program in up to 12 out of 17 mutation points pass all test cases.

5 Conclusions

Although the Triangle Program is small, the number of possible Triangle Programs is huge. We have fully explored a regular subset of it. We reduced the size of its search space by considering only potential improvements to the existing code made by replacing its comparisons. Firstly we restricted the comparator mutations. This enabled us to analyse a systematic subset of the whole improvement fitness landscape. Solutions in the subset are still solutions in the full problem. There are many solutions all of which are readily found by high order schema analysis. Since we use only the number of tests passed there are few fitness levels and as expected there are large plateaux of neutral moves.

Secondly we returned to allowing all possible comparisons. This greatly increases the size of the search space. By allowing more moves we are further from the human written starting point and now schema analysis suggests global search (e.g. via a genetic algorithm) may be deceived. On the other hand, local search (including neutral moves) remains easy near the start point but its chance of finding a solution falls as we move further from the human code. Nonetheless even from a totally random starting point, hill climbing can improve test based fitness.

These results suggest that the program improvement fitness landscape is not as difficult to search as is often assumed.

Datasets http://www.cs.ucl.ac.uk/staff/W.Langdon/egp2017/triangle/

References

1. Langdon, W.B.: Genetically improved software. In Gandomi, A.H., et al., eds.: Handbook of Genetic Programming Applications. Springer (2015) 181–220
2. Langdon, W.B.: Genetic improvement of software for multiple objectives. In Labiche, Y., Barros, M., eds.: SSBSE. LNCS 9275, Bergamo, Italy, Springer (2015) 12–28 Invited keynote.
3. Petke, J.: Preface to the special issue on genetic improvement. Genetic Programming and Evolvable Machines (2017) Editorial Note.
4. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology **43**(14) (2001) 833–839
5. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT press (1992)
6. 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 (1998)
7. 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) (With contributions by J. R. Koza).
8. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In Wang, J., ed.: 2008 IEEE World Congress on Computational Intelligence, Hong Kong, IEEE (2008) 162–168
9. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In Fickas, S., ed.: International Conference on Software Engineering (ICSE) 2009, Vancouver (2009) 364–374
10. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In Raidl, G., et al., eds.: GECCO, Montreal, ACM (2009) 947–954 Best paper.
11. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Communications of the ACM **53**(5) (2010) 109–116
12. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Glinz, M., ed.: 34th International Conference on Software Engineering (ICSE 2012), Zurich (2012) 3–13
13. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1) (2012) 54–72
14. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Quality Journal **21** (2013) 421–443
15. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing programs with semantic code search. In Grunske, L., Whalen, M., eds.: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Lincoln, Nebraska, USA (2015)
16. Kocsis, Z.A., Drake, J.H., Carson, D., Swan, J.: Automatic improvement of Apache Spark queries using semantics-preserving program reduction. In Petke, J., et al., eds.: Genetic Improvement 2016 Workshop, Denver, ACM (2016) 1141–1146
17. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In Nicolau, M., et al., eds.: EuroGP. LNCS 8599, Granada, Spain, Springer (2014) 137–149

18. Marginean, A., Barr, E.T., Harman, M., Jia, Y.: Automated transplantation of call graph and layout features into Kate. In Labiche, Y., Barros, M., eds.: SSBSE. LNCS 9275, Bergamo, Italy, Springer (2015) 262–268
19. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In Xie, T., Young, M., eds.: International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, Maryland, USA, ACM (2015) 257–269 ACM SIGSOFT Distinguished Paper Award.
20. Harman, M., Jia, Y., Langdon, W.B.: Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In Le Goues, C., Yoo, S., eds.: Proceedings of the 6th International Symposium, on Search-Based Software Engineering, SSBSE 2014. LNCS 8636, Fortaleza, Brazil, Springer (2014) 247–252 Winner SSBSE 2014 Challange Track.
21. Jia, Y., Harman, M., Langdon, W.B., Marginean, A.: Grow and serve: Growing Django citation services using SBSE. In Yoo, S., Minku, L., eds.: SSBSE 2015 Challenge Track. LNCS 9275, Bergamo, Italy (2015) 269–275
22. Langdon, W.B., White, D.R., Harman, M., Jia, Y., Petke, J.: API-constrained genetic improvement. In Sarro, F., Deb, K., eds.: Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE 2016. LNCS 9962, Raleigh, North Carolina, USA, Springer (2016) 224–230
23. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1) (2015) 118–135
24. Langdon, W.B., Lam, B.Y.H., Modat, M., Petke, J., Harman, M.: Genetic improvement of GPU software. Genetic Programming and Evolvable Machines (2017) Online first.
25. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation **15**(4) (2011) 515–538
26. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In Silva, S., et al., eds.: GECCO, Madrid, Spain, ACM, ACM (2015) 1327–1334
27. Bruce, B.R.: Energy optimisation via genetic improvement A SBSE technique for a new era in software development. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 819–820
28. Burles, N., Bowles, E., Brownlee, A.E.I., Kocsis, Z.A., Swan, J., Veerapen, N.: Object-oriented genetic improvement for improved energy consumption in Google Guava. In Labiche, Y., Barros, M., eds.: SSBSE. LNCS 9275, Bergamo, Italy, Springer (2015) 255–261
29. Burles, N., Bowles, E., Bruce, B.R., Srivisut, K.: Specialising Guava's cache to reduce energy consumption. In Labiche, Y., Barros, M., eds.: SSBSE. LNCS 9275, Bergamo, Italy, Springer (2015) 276–281
30. Bokhari, M., Wagner, M.: Optimising energy consumption heuristically on android mobile phones. In Petke, J., et al., eds.: Genetic Improvement 2016 Workshop, Denver, ACM (2016) 1139–1140
31. Haraldsson, S.O., Woodward, J.R.: Genetic improvement of energy usage is only as reliable as the measurements are accurate. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 831–832
32. Langdon, W.B., Petke, J., Bruce, B.R.: Optimising quantisation noise in energy measurement. In Handl, J., et al., eds.: 14th International Conference on Parallel Problem Solving from Nature. LNCS 9921, Edinburgh, Springer (2016) 249–259
33. Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. In: Proceedings of the 19th International

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14, Salt Lake City, Utah, USA, ACM (2014) 639–652
34. Wagner, M.: Speeding up the proof strategy in formal software verification. In Petke, J., et al., eds.: Genetic Improvement 2016 Workshop, Denver, ACM (2016) 1137–1138
35. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In Silva, S., et al., eds.: GECCO, Madrid, ACM (2015) 1375–1382
36. Walsh, P., Ryan, C.: Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In D'Hollander, E.H., et al., eds.: Proceedings of ParCo'95. Volume 11 of Advances in Parallel Computing., Gent, Belgium, Elsevier (1995) 415–422
37. Williams, K.P.: Evolutionary Algorithms for Automatic Parallelization. PhD thesis, Department of Computer Science, University of Reading, Whiteknights Campus, Reading, UK (1998)
38. Williams, K.P., Williams, S.A.: Genetic compilers: A new technique for automatic parallelisation. In: 2nd European School of Parallel Programming Environments (ESPPE'96), L'Alpe d'Hoez, France (1996) 27–30
39. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In Sobrevilla, P., ed.: 2010 IEEE World Congress on Computational Intelligence, Barcelona, IEEE (2010) 2376–2383
40. White, D.R., Clark, J., Jacob, J., Poulding, S.M.: Searching for resource-efficient programs: low-power pseudorandom number generators. In Keijzer, M., et al., eds.: GECCO, Atlanta, GA, USA, ACM (2008) 1775–1782
41. White, D.R.: Genetic Programming for Low-Resource Systems. PhD thesis, Department of Computer Science, University of York, UK (2009)
42. Yeboah-Antwi, K., Baudry, B.: Embedding adaptivity in software systems using the ECSELR framework. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 839–844
43. Mrazek, V., Vasicek, Z., Sekanina, L.: Evolutionary approximation of software for embedded systems: Median function. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 795–801
44. Burles, N., Swan, J., Bowles, E., Brownlee, A.E.I., Kocsis, Z.A., Veerapen, N.: Embedded dynamic improvement. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 831–832
45. Vasicek, Z., Mrazek, V.: Trading between quality and non-functional properties of median filter in embedded systems. Genetic Programming and Evolvable Machines (2017) Online first.
46. Petke, J.: Genetic improvement for code obfuscation. In Petke, J., et al., eds.: Genetic Improvement 2016 Workshop, Denver, ACM (2016) 1135–1136
47. Harman, M., Jia, Y., Langdon, W.B., Petke, J., Moghadam, I.H., Yoo, S., Wu, F.: Genetic improvement for adaptive software engineering. In Engels, G., ed.: 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14), Hyderabad, India, ACM (2014) 1–4 Keynote.
48. Landsborough, J., Harding, S., Fugate, S.: Removing the kitchen sink from software. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 833–838
49. Harman, M., Jia, Y., Krinke, J., Langdon, W.B., Petke, J., Zhang, Y.: Search based software engineering for software product line engineering: a survey and directions for future work. In: 18th International Software Product Line, SPLC 2014, Florence, Italy (2014) 5–18 Invited keynote.

50. Lopez-Herrejon, R.E., Linsbauer, L., Assuncao, W.K.G., Fischer, S., Vergilio, S.R., Egyed, A.: Genetic improvement for software product lines: An overview and a roadmap. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 823–830
51. Langdon, W.B., Petke, J., White, D.R.: Genetic improvement 2015 chairs' welcome. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 791–792
52. Langdon, W.B., Harman, M.: Fitness landscape of the triangle program. In Veerapen, N., Ochoa, G., eds.: PPSN-2016 Workshop on Landscape-Aware Heuristic Search, Edinburgh (2016) Also available as UCL RN/16/05.
53. Langdon, W.B., Petke, J.: Software is not fragile. In Parrend, P., et al., eds.: Complex Systems Digital Campus E-conference, CS-DC'15. Proceedings in Complexity, Springer (2015) 203–211 Invited talk.
54. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5) (2011) 649–678
55. Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software **83**(12) (2010) 2416–2430
56. Cody-Kenny, B., Lopez, E.G., Barrett, S.: locoGP: improving performance by genetic programming Java source code. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 811–818
57. Orlov, M., Sipper, M.: Flight of the FINCH through the Java wilderness. IEEE Transactions on Evolutionary Computation **15**(2) (2011) 166–182
58. Schulte, E., Forrest, S., Weimer, W.: Automated program repair through the evolution of assembly code. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, ACM (2010) 313–316
59. Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software mutational robustness. Genetic Programming and Evolvable Machines **15**(3) (2014) 281–312
60. Schulte, E., Weimer, W., Forrest, S.: Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In Langdon, W.B., et al., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 847–854 Best Paper.
61. Wright, S.: The roles of mutation, inbreeding, crossbreeding and selection in evolution. In: Proceedings of the Sixth Annual Congress of Genetics. (1932) 356–366
62. Reidys, C.M., Stadler, P.F.: Combinatorial landscapes. SIAM Review **44**(1) (2002) 3–54
63. Holland, J.H.: Genetic algorithms and the optimal allocation of trials. SIAM Journal on Computation **2** (1973) 88–105
64. Goldberg, D.E.: Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley (1989)
65. Daolio, F., Tomassini, M., Verel, S., Ochoa, G.: Communities of minima in local optima networks of combinatorial spaces. Physica A: Statistical Mechanics and its Applications **390**(9) (2011) 1684–1694
66. Ochoa, G., Verel, S., Daolio, F., Tomassini, M.: Local optima networks: A new model of combinatorial fitness landscapes. In Richter, H., Engelbrecht, A., eds.: Recent Advances in the Theory and Application of Fitness Landscapes. Springer (2014) 233–262
67. Ochoa, G., Veerapen, N.: Additional dimensions to the study of funnels in combinatorial landscapes. In: GECCO, ACM (2016) 373–380