A Survey of Genetic Improvement Search Spaces

Justyna Petke
Department of Computer Science
University College London
London, UK
j.petke@ucl.ac.uk

Brad Alexander
School of Computer Science
University of Adelaide
Adelaide, Australia
brad@cs.adelaide.edu.au

Earl T. Barr
Department of Computer Science
University College London
London, UK
e.barr@ucl.ac.uk

Alexander E.I. Brownlee
Computing Science and Mathematics
University of Stirling
Stirling, UK
sbr@cs.stir.ac.uk

Markus Wagner
School of Computer Science
University of Adelaide
Adelaide, Australia
markus.wagner@adelaide.edu.au

David R. White
Department of Computer Science
The University of Sheffield
Sheffield, UK
d.r.white@sheffield.ac.uk

ABSTRACT

Genetic Improvement (GI) uses automated search to improve existing software. Most GI work has focused on empirical studies that successfully apply GI to improve software's running time, fix bugs, add new features, etc. There has been little research into *why* GI has been so successful. For example, genetic programming has been the most commonly applied search algorithm in GI. Is genetic programming the best choice for GI? Initial attempts to answer this question have explored GI's mutation search space. This paper summarises the work published on this question to date.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**;

KEYWORDS

Genetic Improvement, Search-based Software Engineering, Program Repair, Search Space, GI, APR, SBSE

ACM Reference Format:
Justyna Petke, Brad Alexander, Earl T. Barr, Alexander E.I. Brownlee, Markus Wagner, and David R. White. 2019. A Survey of Genetic Improvement Search Spaces. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3319619.3326870

1 INTRODUCTION

Genetic Improvement (GI) applies search to improve existing software [29]. GI research emerged from the field of Genetic Programming (GP), and early work on parallelisation [34, 40], runtime and energy optimisation [42], and bug fixing [2, 3] applied tree-based GP directly to abstract syntax tree (AST) representations of software. Later applications by Forrest, Weimer and Le Goues et al. [6, 24, 41] scaled the approach to program repair by using a patch representation. This has also been adopted by Langdon et al. [15, 16], who introduced a line-level patch representation. The results of Langdon's work on improving software's runtime have been adopted by open source projects [19]. More recently, GI has seen commercial deployment in companies, including live bug fixing [8, 10, 45].

Despite these successes, the question — Why and when does GI work well? — remains open. Papers arguing for a more informed choice of various elements of the GI approach are scarce [7, 21, 22] and largely concerned with stating the need for further research in this direction [13, 20, 28, 33, 43, 44]. Almost all GI work to date, for instance, employs Genetic Programming as its key search technique [30]. Insights into GI search spaces would allow us to improve fitness functions and search operators. Recent work has raised this issue: Langdon and Ochoa [20] state that "the global structure of program search spaces is little understood"; Renzullo et al. [33] hypothesise further that "Genetic Improvement of software is more likely to succeed when the search algorithm is well-matched to the fitness landscape"[1].

Modelling the Genetic Improvement search space is not an easy task, as the set of program variants is vast. Consider three basic line-level GI operations: *delete*, *copy* and *replace*. Suppose a program has λ lines of code. Then a single application of the *delete* operator can produce up to λ different program variants. The *copy* operation can produce up to λ^2 program variants, as an existing line of code can be added before any existing line in the program. The *swap* operation can produce up to $(\lambda - 1) * \lambda$ program variants, as an existing line of code can be swapped with any of the remaining $\lambda - 1$ lines. Therefore, the program search space for just one step (i.e., a single mutation) of the Genetic Improvement process is of the size of $2 * \lambda^2$, i.e., $O(\lambda^2)$, and this complexity grows exponentially with the number of mutations.

More generally, the search space for any GI framework can be defined as follows: Let $\Lambda = \{\lambda_1, \cdots, \lambda_i, \cdots \lambda_k\}$ be the set of program locations that can be mutated. Let $M_i = \{\mu_1, \cdots, \mu_j, \cdots, \mu_m\}$ be the list of mutations that can be applied at location λ_i. Then the search space of program variants with just one modification is

[1] For the definition of a *fitness landscape*, see Definition 2.1.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326870

$\sum_i (M_i * \lambda_i)$.[2] Given that each variant can then be further modified by adding mutations, the search space grows exponentially, with the exponent, n, equal to the number of mutations allowed to the code, i.e. $O((\sum_i (M_i * \lambda_i))^n)$.

In this paper we summarise work on the topic of GI search space analysis published to date.

2 DEFINITIONS

In order to reason about large search spaces in evolutionary computation research, the concept of a *fitness landscape* [32] was introduced:

Definition 2.1 (Fitness Landscape [27]). A landscape is a triplet (S, V, f) where S is a set of potential solutions, i.e. a search space, $V : S \rightarrow 2^S$, a neighbourhood structure, is a function that assigns to every $s \in S$ a set of neighbours $V(s)$, and $f : S \rightarrow R$ is a fitness function that can be pictured as the height of the corresponding potential solutions.

Given an optimisation problem, each solution can be assigned a fitness value. Usually, in evolutionary computation, the higher the fitness value, the closer that solution is assumed to be to the optimal one. In many cases *fitness distance correlation* [14] can be used to predict the performance of genetic algorithms on problems with known global maxima. In the fitness landscape representation there will be an edge between two solutions if one can be reached from another using a given transformation operator. Knowing the geometry of a fitness landscape can thus lead to informed decisions about which search algorithm would be best to traverse it. For example, if it has a bell shape, then a hill-climbing algorithm might be the most efficient.

Given the huge search space of the possible program mutations, Ochoa et al. introduced [27] *local optima networks* to visualise fitness landscapes:

Definition 2.2 (Local Optimum [27]). A local optimum is a solution s^* such that $\forall s \in V(s^*)\ f(s) < f(s^*)$.

Definition 2.3 (Basin of attraction [27]). The basin of attraction of a local optimum i is the set $b_i = s \in S \mid LocalSearch(s) = i$. The size of the basin of attraction of a local optima i is the cardinality of b_i.

Definition 2.4 (Local optima network [27]). The local optima network $G = (S^*, E)$ is the graph where the nodes are the local optima, and there is an edge $e_{ij} \in E$ between two local optima i and j if there is at least a pair of direct neighbours (1-bit apart) s_i and s_j, such that $s_i \in b_i$ and $s_j \in b_j$. That is, if there exists a pair of direct neighbours solutions s_i and s_j, one in each basin (b_i and b_j).

The *LocalSearch* algorithm (Algorithm 1), presented below, determines local optima and therefore defines the basins of attraction. It defines a mapping from the search space S to the set of locally optimal solutions S^*.

The above definitions came from the field of evolutionary computation. With the development of GI research, a new concept for reasoning about mutations on software code was introduced, namely *software mutational robustness*:

[2]For example, for a program with 5 mutable lines of code and three classical mutation operators (*delete*, *copy* and *replace*): $|M_i| = 5 + 5 * 5 + 4 * 5 = 50$ for each $1 \leq i \leq 5$.

Algorithm 1 *LocalSearch* [27]

```
Choose initial solution s ∈ S
repeat
  choose s' ∈ V(s) such that f(s') = max_{x∈V(s)} f(x)
  if f(s) < f(s') then
    s ← s'
  end if
until s is a Local optimum
```

Definition 2.5 (Software Mutational Robustness [35]). Given a program P, a set of mutation operators M, and a test suite T such that $T(P) = true$, we define the *software mutational robustness*, written $MutRB(P, T, M)$, to be the fraction of all direct mutants $P' = m(P)$, $\forall m \in M$ which both compile and pass T:

$$MutRB(P, T, M) = \frac{\{P' | m \in M . P' = m(P) \wedge T(P') = true\}}{\{P' | m \in M . P' = m(P)}$$

Software mutation robustness aims to capture with a single value the ratio of code mutations that do not change code functionality, as defined by its test suite. It can be viewed as a measure of neutrality of the search landscape.

GI search spaces are defined by the set of mutations that can be applied to the code. Each point in the search space represents one program variant.

Definition 2.6. In the context of GI, an n-order mutant is a program variant with n mutations applied to it.

The aim of search space analysis is to identify what the fitness landscape looks like. There are, however, multiple ways of defining mutations and evaluating the fitness of a program variant in GI.

The most common mutations in GI [29] are: *delete*, *copy* and *replace* applied at the abstract syntax tree (AST) or line level. A few studies considered more fine-grained, expression level changes, such as mutation of arithmetic expressions [11], or template-based mutations [26]. Furthermore, both in improvement of functional and non-functional program properties, test case result is considered in fitness evaluations. However, an effective mutation (i.e., an individual of better fitness than the original code) in automated program repair, for instance, only needs to pass all tests, while for improvement of runtime such a mutation would not be considered effective if it led to a slowdown with respect to the original code. Moreover, certain studies in non-functional property improvement consider approximate results [5], thus allowing for test failures.

Given the obstacles above, the study of GI search spaces is not a trivial task. Several studies on individual programs were published though. In this paper we summarise results on the topic published to date.

3 METHODOLOGY

In order to gather papers on the subject of analysing GI search spaces, we searched several digital libraries, outlined below, and used the following criterion: *the study must produce and analyse a fixed set of program variants (reachable with a defined set of mutations) regardless of the fitness of the mutated programs.* The reason for enforcing this criterion is that certain papers mention search

landscapes in the context of proposing new search variants or comparing existing approaches, and providing an argument for why they think one approach would be better than the other. Authors of such studies then report on the efficacy of their tools with the number of improved program variants found within a given time limit. We do not consider such work as studies on the analysis of GI search spaces. Typical papers we pick clearly define the search space and report results either for all possible mutation variants within n mutation steps or a fraction of such a search space.

We first gathered papers from Petke et al.'s survey on GI [29]. It covers all GI work to the end of 2015. In order to gather relevant papers published after 2015 we searched the online libraries of three major publishers in evolutionary computation and software engineering, that is, ACM (ACM Digital Library [1]), IEEE (IEEE Xplore [12]) and Springer (SpringerLink [37]). We used "genetic improvement" as the keyword and set the 2016-2019 year range. We also looked through the comprehensive online bibliography on automated program repair [31] in order to find relevant papers.[3]

Table 1: Number of papers on the subject of GI search spaces. Date of search: 3 April 2019.

Source	Found	Relevant
Petke et al.[29]	(citations) 222	4
Published after 2015		
APR online bibliography	106	2
Filters: exact keyword: "genetic improvement" year range: 2016-2019 Full Text and Metadata Computer Science (SpringerLink only)		
IEEE Xplore	42	1
ACM Digital Library	46	3
SpringerLink	42	4
Total (without repetitions)		14

4 SEARCH SPACE ANALYSIS

In the following two subsections, we summarise the results of GI search space analysis in the literature, presented in the papers found through our searches, presented in Table 1. We categorise the papers into two sets: those that deal with improvement of non-functional properties, such as runtime or energy consumption, and those that deal with functional property improvement, such as bug fixing. Unless stated otherwise, the search space considered in the next section is formed by the most commonly used three operators: *delete*, *copy* and *replace*, performed at either line or abstract syntax tree (AST) level.

[3]We found this to be an effective and efficient approach when looking for literature on APR, as phrases such as "fixing", "repair", "automated", "automatic", "software" and "program" have been used interchangeably in the field, thus returning thousands of irrelevant searches from the digital libraries.

4.1 Search space analysis for non-functional improvement

Langdon et al. [18, 21] presented fitness landscapes for first-order GI mutations (see Definition 2.6) applied to three large real-world programs. A subset of the codebase was considered for each piece of software, ranging from a few hundred to a few thousand lines of code. Specialised BNF-like notation was used to prevent compilation errors, further decreasing the number of locations where mutation could be applied. Nevertheless, up to 61,775 possible executable *delete*, *copy* and *replace* line-level mutations were considered. Faithfulness to the original program was established by running a test suite on each of the evolved software variants and comparing the results with the original outputs. The lowest compilation rate was reported in the case of the BWA bioinformatics tool: 23% of the evolved changes led to executable programs. It's worth noting that 89% of those produced no change to the original program. The highest compilation rate was observed for the StereoCamera program: 95% of mutations led to executable programs in this case. Moreover, 62% of the mutations did not change the output, 0.93% of which produced faster software variants. This investigation points to a neutral program landscape of single mutations with relatively few changes leading to faster software variants. It is hard to draw general conclusions, however, as two of the three systems came from the field of bioinformatics and were written in C and C++, while the third system was a CUDA image processing program. Furthermore, restrictions were put on the number of lines considered for mutation imposed by the specialised BNF grammar and initial profiling to determine the most time-consuming parts of code.

Bruce et al. [5] investigated the GI search space for energy consumption optimisation. They were the first to consider synergistic effects between mutations and the first to consider approximate solutions in this context. Bruce et al. used four real-world benchmarks, form the PARSEC benchmark set [4], each consisting of several thousand lines of code. By using the specialised BNF-like notation (developed by Langdon, first used by Langdon and Harman [15]), between 34% and 40% of each benchmark's lines of code were candidates for mutation. Roughly twice as many first-order mutations were randomly generated to investigate the search space (28000 in total), which constituted up to 0.2% of the whole search space. Each of the three types of mutations was selected with equal probability. Only 1.36% of all mutations considered led to improved software variants, with an average of an impressive 33.90% energy consumption reduction. For the investigation of synergistic effects, 15% of all pairwise combinations of effective modifications was chosen at random. Results showed significant interactions between the modifications, with 12% of pairs leading to energy reductions greater than reductions obtained by the sum of each of the two modifications if applied separately, while 38.5% of pairs leading to less energy reductions than the best of the two modifications considered. These results point to a flat search landscape with relatively few local optima.

Haraldsson et al. [11] looked at the search landscape of more fine-grained program modifications for a bioinformatics program. Table 2 lists the applied mutations. Over 4297 mutation points were identified in >8k lines of C and C++ code. The search space was explored by performing a ten-step random walk away from

the original program. This process was repeated 100 times. The authors concluded that execution time generally does not degrade as program variants become more dissimilar from the original, yet the compilation rate drops significantly with no compilable program variants after 9 steps. The authors also sampled 2265 unique first-order mutants from previous experiments and looked at the variation in runtime of those that compiled (1622 in total): most ran to completion in similar time, confirming findings reported by Langdon and Petke [21] and pointing to a neutral search landscape. A related study [9] used similar fine-grained modifications in a random walk for three small Python programs. This focused on the number of test cases passed for the program mutants, rather than execution time. Most first-order mutants of the original program ran and successfully passed their test suites. A large number of first-order mutants caused all tests to fail, with some rare mutations causing only a small subset of tests to fail.

Table 2: Set of program modifications used by Haraldsson et al. [11]. Any member of a given set can be swapped with another member of the same set.

Description	Operations
Numerical constants	increment or decrement by 1
Arithmetic operators	$+, -, *, /, \%$
Arithmetic operators	$+=, -=, *=, /=$
Incremental operators	$++, --$
Relational operators	$<, >, <=, >=, ==, !=$
Bit assignments	$\&=, \mid=$
Bit operators	$\&, \mid$

Sidiroglou et al. [36] presented search spaces for loop perforation of seven C and C++ programs from the PARSEC benchmark test suite [4]. Loop perforation provides a general technique to trade accuracy for performance by transforming loops to execute a subset of their iterations. Even though Sidiroglou et al.'s work was not presented as Genetic Improvement, such modifications could happen in the typical GI framework. Loop perforation rates of 0.25, 0.50, 0.75 and 1 were used. Authors were able to explore the search space exhaustively, with the largest experiment finishing within three days. Graphical representations of all the software variants were created, showing that one can typically achieve a two-fold speed-up, at the cost of around 5% loss of output accuracy. It is not clear, however, how close each of the program variants are in the search space and what percentage led to faster, acceptable solutions.

Published investigations into the GI search space have thus far been restricted to C/C++, CUDA and Python code; little comparison between the impact of programming language on search space structure has been performed.

4.2 Search space analysis for functional improvement

The main focus of search space analysis has been on spaces for automated program repair. For example, the exhaustive first-order mutation search space exploration for the TRIANGLE program published only in 2017 [22]. In all such studies only the pass or failure of a test case on compilable program variants is considered for the purpose of fitness evaluation, in contrast to work on non-functional property improvement.

Schulte et al. [35] considered three mutation operators (*swap*, *delete*, and *replace*), at the AST and assembly language level. They generated 200 first-order mutants of each of their 22 small C benchmark programs and showed that overall 36.8% of mutants are neutral, i.e., still pass all the programs' test cases. Moreover, mutational robustness scores (see Definition 2.5) were all over 21%. Moreover, they generated mutants that are up to 250 neutral steps away from the original program, containing less than 200 lines of code. They also controlled for size to avoid bloat (i.e., only mutants leading to variants that do not increase the original program size were considered). They repeated this process 100 times and reported that mutational robustness of such programs increases with the mutational distance away from the original program. Authors conclude that there are large neutral landscapes around any given program that are easily traversible using iterative mutation. They also considered Haskell, OCaml and C++ implementations for four small sorting programs, showing that high mutational robustness also occurs in those programming languages. Finally, authors generated 5000 neutral variants for 11 small buggy programs and reported that for 9 out of 10 programs there exists a program variant that fixes a bug in such a neutral mutation landscape. This suggests GI methods should be designed to explore neutral networks.

The most thoroughly studied program for the purpose of exploration of the Genetic Improvement search space is the C implementation of the TRIANGLE program[4] [17, 22, 38]. Given the lengths of three sides of a triangle, the program classifies it as either scelene, isosceles, equilateral, or not a triangle.

Langdon et al. [17] considered the mutation of swapping comparison operators only and investigated mutations up to fourth-order. Given 17 comparisons and 6 comparison operators, over 1.5 million program variants were created. 16% of all first-order mutants passed all the 14 test cases with 21% failing only on one. These numbers dropped rapidly to 0.06% (all pass) and 0.83% (one test case fails) for all program variants with four mutations. It is worth mentioning that the test set was carefully picked to cover all the branches and all Boolean sub-expressions in the IF statements.

Langdon et al. [22] provided the most comprehensive study of the fitness landscape of the TRIANGLE program. Among other experiments, they generated and ran all possible 6^{17} program variants. They discovered 9215 mutants that pass all the test cases, with 78% of them failing five test cases. They found a bell-shaped distribution of the fitness landscape and thus ran a hill climber on every program variant close to the original and a sample of higher-order mutants. They noticed that only 0.24% points in the search space of fourth-order mutants cannot reach a program variant passing all test cases. Moreover, for most programs that fail five test cases the hill climbing algorithm could find program variants failing just two test cases.

Given the huge search space for the possible program mutations, Langdon et al. [22] also used *local optima networks* (see Definition 2.4) to visualise the fitness landscape of the TRIANGLE program.

[4]Full implementation of the trianlge program is provided in [17].

The same approach was used by by Veerapen et al. [38], who additionally analysed the TCAS program. Figure 1 shows the local optima network obtained from 100 runs of the *LocalSearch* Algorithm 1 and their first 1000 iterations for the TRIANGLE program.

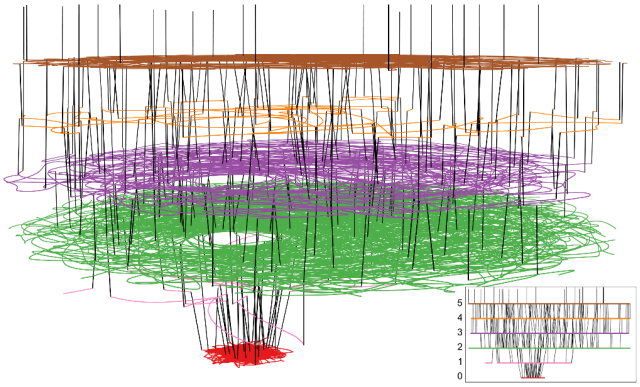

Figure 1: Local optima network for the TRIANGLE program [22]. Red edges connect program variants that pass all test cases, pink fail 1, green 2, purple 3, orange 4, & brown 5.

Veerapen et al. [38, 39] differentiated between mutations of Boolean and comparison operators. They used an iterated local search algorithm that starts from a locally-optimal solution and then alternates between a random mutation and a best-improving hill-climber. The algorithm was run 1000 times (for each set of allowed mutations) and stopped after 10000 steps. All generated local optima networks showed high level of neutrality. In the case of TRIANGLE when both Boolean and comparison mutations were considered, only 31% runs found a global optimum, with most getting stuck on plateau with two failing test cases. However, in all the other cases the success rate was around 90%. This shows that neutrality of the landscapes for the TRIANGLE and TCAS programs does not prevent search to easily find paths to a program variant for which all test cases pass.

Neutrality of Genetic Improvement search spaces has been observed in the work on automated software repair also on larger real-world software systems.

Renzullo et al. [33] looked at Linux utility programs, containing a few hundred lines of C code. They started with a buggy software variant. Next, they selected 10 AST nodes from each program. They generated all possible first-order mutants and all possible pairwise combinations of mutations at the selected AST nodes. The operators used were *delete*, *copy* and *swap*. This process was repeated 4 times, creating 20200 software variants for each program under study. For four out of five pieces of software, over 70% of variants did not compile (only 7% variants of ZUNE failed to compile). This shows the importance of using grammars to restrict the search space for software improvements. Between 3% to 18% of software variants were neutral. Ratio between the neutral and non-neutral variants varied significantly, depending on the program. Moreover, between 0.14% to 0.60% of 202000 software variants led to a repair. Authors also looked at unique repairs. Overall, over twice as many second-order mutants led to a repair than first-order ones. However, most of them were discovered in the LOOK program, where in all 11 cases at least one mutation was a neutral one. This result shows the importance of traversing the neutral (under test-equivalence) program variants in order to find improved software.

Long and Rinard [25] analysed the search space of patches in the automated program repair field. Although they did not consider typical GI techniques, their results shed some insight into program search space investigations. Long and Rinard considered two tools for automated program repair: Prophet, which uses a fixed set of templates derived from human-written patches, and SPR, which uses a set of transformations, such as introduction of an IF condition to avoid the execution of a faulty statement. To the best of our understanding, they define the search spaces as the total number of possible transformations for the first x statements, as found by the error localiser. They consider the first 100, 200, 300 and 2000 statements. Authors also provided two extensions to SPR and Prophet, that is, condition synthesis extension (which considers certain binary comparison operators) and value replacement extension (which introduces certain variable and constant replacements).[5] These additional transformations further increased the search spaces considered. It is, however, unclear whether all transformations allowed for both systems were considered as the search space. For example, one of the transformations in SPR is statement replacement. With 2000 statements, that is $4 * 10^6$ possible transformations. SPR usually uses prioritisation heuristics to rank most likely transformations. Nevertheless, search space sizes are reported in the paper, with the size of the largest one in the order of 10^5.

Long and Rinard ran SPR and Prophet with these 16 different settings on a set of eight large real-world open source C programs with 69 defects (considering all mutations for the 100, 200, 300 and 2000 statements output by the error localiser). They distinguished plausible and correct patches as follows: plausible patches pass all the test cases, while the correct ones matched the corresponding developer patches. They concluded that correct patches are sparse while plausible ones are orders of magnitude more abundant. They also concluded that larger search spaces (i.e., allowing for more transformations) leads to more plausible patches and less correct patches being found. It is worth noting, however, that there is no universal definition of a correct patch. It is thus difficult to tell whether certain patches that were labelled as "incorrect" truly were so. Le Goues et al. [23] provided examples of correct patches evolved by their system that differed from the developer ones. In that study each automatically generated patch was validated on a held-out test suite, compared with a developer-provided one, and validated manually. For Angelix, for instance, four patches syntactically matched the developer-produced one, whilst manual analysis revealed additional two (50% increase) correct patches that were different from the developer-derived ones. This shows the importance of validating results generated by GI.

Martinez and Monperrus [26] took a more formal approach towards defining the search space for automated program repair tools. In particular, they proposed a probabilistic model for the median number of attempts needed to find a repair shape that fixes a given bug. They analysed two models, one consisting of 41 change actions,

[5] Details of these two extensions can be found in Long and Rinard [25].

such as "statement insertion", and extended it to another model consisting of 173 change actions such as "statement insertion of an IF statement". A repair shape is defined as an unordered tuple of change actions. Authors mined 14 open source Java projects, containing 62000 code changes, called transactions, consisting of 1.2 million changes at the AST level. They considered the space of various fix transactions based on their size. For example, they abstracted repair shapes consisting of single AST changes, 5-AST level changes etc. They consider the search space to be composed as follows: number of statements identified by a fault localisation algorithm × number of repair shapes × number of instantiations of a repair shape. Based on the distribution of repair shapes on one project, they computed a probability distribution model for repair shapes and applied it to the remaining 13 projects, calculating the median number of attempts needed to find the correct repair shape.

Overall, Martinez and Monperrus conclude: using probability distributions over change actions could significantly decrease runtime of tools for automated program repair; the more abstract the repair shape (with fewer choices) the more likely a correct repair shape will be found (therefore, relying on precise repair actions used in program synthesis alone decreases chances of finding the right fixes); and certain repair shapes are impossible to find due to their size (larger than 4 AST nodes for the model with 173 repair actions). Therefore, there is a limit to what can be improved using an automated approach and you need heuristics to combine repair actions as a random template-based approach is unlikely to find correct fixes requiring more than 4 AST changes.

5 SUMMARY

Despite the wide range of successful applications of GI, the question about which search algorithm is most efficient and effective for GI is yet to be answered. We identified 14 published studies on the subject to date, shedding some initial light on how GI search spaces look like. These are usually defined by the most commonly used mutation operators (i.e. *delete*, *copy* and *replace*) on either AST or line-level, with a few considering mutations at the expression level. Furthermore, the definition of effective mutations differs between studies on non-functional and functional property improvement. In the former case the assumption is that all test cases pass in the first instance whilst simultaneously improving the non-functional property of interest. In automated program repair, however, a successful mutation needs to just pass all the test cases. This lack of unified framework and unified benchmark set makes it harder to draw general conclusions.

Nevertheless, from the existing work on the program search spaces in the Genetic Improvement context, the following observations were made: Programs contain large neutral spaces around them that could be explored to improve program's both functional and non-functional properties. Heuristics, such as Genetic Programming, are needed to explore combinations of program mutations. By considering probabilities of occurrence over program mutations, one can speed up the search for good solutions. It is impossible to exhaustively explore the search space of real-world program variants, yet there exist several connections between local optima that the search should explore. Approximation yields more, yet still acceptable, solutions within the program search space.

There are several gaps in the literature. For example, the impact of the programming language on the search space is yet to be explored. In contrast to studies of search landscapes for non-functional improvement, automated program repair literature covered a wide range of programs and programming languages (assembly, Haskell, C++ to name a few). Another issue concerns fitness evaluation that relies on Boolean test case results. This could explain the largely neutral landscapes, rugged around local optima. Therefore, we argue that more fine-grained fitness functions should be considered, to better guide the underlying search algorithms in Genetic Improvement.

ACKNOWLEDGMENTS

The work in this paper was funded by the UK EPSRC [grants EP/P023991/1 and EP/J017515/1]; and Australian Research Council Project DE160100850.

REFERENCES

[1] ACM. 2019. Digital Library. http://dl.acm.org/. (2019).
[2] Andrea Arcuri. 2008. On the automation of fixing software bugs. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 1003–1006. https://doi.org/10.1145/1370175.1370223
[3] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China*. IEEE, 162–168. https://doi.org/10.1109/CEC.2008.4630793
[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, Andreas Moshovos, David Tarditi, and Kunle Olukotun (Eds.). ACM, 72–81. https://doi.org/10.1145/1454115.1454128
[5] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. 2017. Approximate Oracles and Synergy in Software Energy Search Spaces. (2017). Accepted to TSE.
[6] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, Franz Rothlauf (Ed.). ACM, 947–954. https://doi.org/10.1145/1569901.1570031
[7] Saemundur O. Haraldsson. 2017. *Genetic improvement of software: from program landscapes to the automatic improvement of a live system*. Ph.D. Dissertation. University of Stirling, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.725136
[8] Saemundur O. Haraldsson, John R. Woodward, and Alexander E. I. Brownlee. 2017. The Use of Automatic Test Data Generation for Genetic Improvement in a Live System. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. IEEE, 28–31. https://doi.org/10.1109/SBST.2017.10
[9] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and David Cairns. 2017. Exploring Fitness and Edit Distance of Mutated Python Programs. In *Genetic Programming*, James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 19–34.
[10] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1513–1520. https://doi.org/10.1145/3067695.3082517
[11] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, Albert V. Smith, and Vilmundur Gudnason. 2017. Genetic improvement of runtime and its fitness landscape in a bioinformatics application. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1521–1528. https://doi.org/10.1145/3067695.3082526
[12] IEEE Xplore. 2019. Digital Library. http://ieeexplore.ieee.org/Xplore/home.jsp. (2019).
[13] Colin G. Johnson and John R. Woodward. 2015. Fitness as Task-relevant Information Accumulation. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 855–856.

https://doi.org/10.1145/2739482.2768428

[14] Terry Jones and Stephanie Forrest. 1995. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 15-19, 1995*, Larry J. Eshelman (Ed.). Morgan Kaufmann, 184–192.

[15] William B. Langdon and Mark Harman. 2010. Evolving a CUDA kernel from an nVidia template. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*. IEEE, 1–8. https://doi.org/10.1109/CEC.2010.5585922

[16] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Trans. Evolutionary Computation* 19, 1 (2015), 118–135. https://doi.org/10.1109/TEVC.2013.2281544

[17] William B. Langdon, Mark Harman, and Yue Jia. 2010. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software* 83, 12 (2010), 2416–2430. https://doi.org/10.1016/j.jss.2010.07.027

[18] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. 2017. Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines* 18, 1 (2017), 5–44. https://doi.org/10.1007/s10710-016-9273-9

[19] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 1063–1070. https://doi.org/10.1145/2739480.2754652

[20] William B. Langdon and Gabriela Ochoa. 2016. Genetic improvement: A key challenge for evolutionary computation. In *IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016*. IEEE, 3068–3075. https://doi.org/10.1109/CEC.2016.7744177

[21] William B. Langdon and Justyna Petke. 2017. Software is not fragile. In *First Complex Systems Digital Campus World E-Conference 2015*. Springer, 203–211.

[22] William B. Langdon, Nadarajen Veerapen, and Gabriela Ochoa. 2017. Visualising the Search Landscape of the Triangle Program. In *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings (Lecture Notes in Computer Science)*, James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.), Vol. 10196. 96–113. https://doi.org/10.1007/978-3-319-55696-3_7

[23] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. https://doi.org/10.1145/3106237.3106309

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[25] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 702–713. https://doi.org/10.1145/2884781.2884872

[26] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8

[27] Gabriela Ochoa, Marco Tomassini, Sébastien Vérel, and Christian Darabos. 2008. A study of NK landscapes' basins and local optima networks. In *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*, Conor Ryan and Maarten Keijzer (Eds.). ACM, 555–562. https://doi.org/10.1145/1389095.1389204

[28] Justyna Petke. 2017. New operators for non-functional genetic improvement. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1541–1542. https://doi.org/10.1145/3067695.3082520

[29] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, David R. White, Woodward, and John R. Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* (2017). https://doi.org/10.1109/TEVC.2017.2693219

[30] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, David R. White, Woodward, and John R. Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey: Supplemental Material: Core Papers on Genetic Improvement. *IEEE Transactions on Evolutionary Computation* (2017). https://ieeexplore.ieee.org/abstract/document/7911210/media

[31] Program-repair.org. 2019. Online library on automated program repair. http://program-repair.org/bibliography.html. (2019).

[32] Christian M. Reidys and Peter F. Stadler. 2002. Combinatorial Landscapes. *SIAM Rev.* 44, 1 (2002), 3–54. https://doi.org/10.1137/S0036144501395952

[33] Joseph Renzullo, Stephanie Forrest, Westley Weimer, and Melanie Moses. 2018. Neutrality and Epistasis in Program Space. In *Genetic Improvement Workshop, co-located with ICSE 2018*.

[34] Conor Ryan and Paul Walsh. 1997. Paragen II: Evolving Parallel Transformation Rules. In *Computational Intelligence, Theory and Applications, International Conference, 5th Fuzzy Days, Dortmund, Germany, April 28-30, 1997, Proceedings (Lecture Notes in Computer Science)*, Bernd Reusch (Ed.), Vol. 1226. Springer, 573. https://doi.org/10.1007/3-540-62868-1_166

[35] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2014), 281–312. https://doi.org/10.1007/s10710-013-9195-8

[36] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 124–134. https://doi.org/10.1145/2025113.2025133

[37] SpringerLink. 2019. Online search platform. http://link.springer.com/. (2019).

[38] Nadarajen Veerapen, Fabio Daolio, and Gabriela Ochoa. 2017. Modelling genetic improvement landscapes with local optima networks. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1543–1548. https://doi.org/10.1145/3067695.3082518

[39] Nadarajen Veerapen and Gabriela Ochoa. 2018. Visualising the global structure of search landscapes: genetic improvement as a case study. *Genetic Programming and Evolvable Machines* 19, 3 (2018), 317–349. https://doi.org/10.1007/s10710-018-9328-1

[40] Paul Walsh and Conor Ryan. 1995. Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System. In *In Proceedings of ParCo'95*. NorthHolland, 2–2.

[41] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[42] David R. White. 2009. *Genetic Programming for Low-Resource Systems*. Ph.D. Dissertation. University of York.

[43] David R. White. 2016. Guiding Unconstrained Genetic Improvement. In *Genetic and Evolutionary Computation Conference, GECCO 2016, Denver, CO, USA, July 20-24, 2016, Companion Material Proceedings*, Tobias Friedrich, Frank Neumann, and Andrew M. Sutton (Eds.). ACM, 1133–1134. https://doi.org/10.1145/2908961.2931688

[44] David Robert White and Jeremy Singer. 2015. Rethinking Genetic Improvement Programming. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 845–846. https://doi.org/10.1145/2739482.2768426

[45] Yue Jia, Ke Mao, Mark Harman. [n. d.]. Finding and fixing software bugs automatically with SapFix and Sapienz. https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/. ([n. d.]).