Empir Software Eng (2018) 23:2980–3006
https://doi.org/10.1007/s10664-017-9562-9

Improved representation and genetic operators for linear genetic programming for automated program repair

Vinicius Paulo L. Oliveira[1] · Eduardo Faria de Souza[1] · Claire Le Goues[2] · Celso G. Camilo-Junior[1]

Published online: 25 January 2018

Abstract Genetic improvement for program repair can fix bugs or otherwise improve software via patch evolution. Consider GenProg, a prototypical technique of this type. GenProg's crossover and mutation operators manipulate individuals represented as patches. A patch is composed of high-granularity edits that indivisibly comprise an edit operation, a faulty location, and a fix statement used in replacement or insertions. We observe that recombination and mutation of such high-level units limits the technique's ability to effectively traverse and recombine the repair search spaces. We propose a reformulation of program repair representation, crossover, and mutation operators such that they explicitly traverse the three subspaces that underlie the search problem: the Operator, Fault and Fix Spaces. We provide experimental evidence validating our insight, showing that the operators provide considerable improvement over the baseline repair algorithm in terms of search success rate and efficiency. We also conduct a genotypic distance analysis over the various types of search, providing insight as to the influence of the operators on the program repair search problem.

Communicated by: Martin Monperrus and Westley Weimer

✉ Vinicius Paulo L. Oliveira
viniciusp.comp@gmail.com

Eduardo Faria de Souza
eduardosouza@inf.ufg.br

Claire Le Goues
clegoues@cs.cmu.edu

Celso G. Camilo-Junior
celso@inf.ufg.br

[1] Instituto de Informatica, Universidade Federal de Goias (UFG), Goiânia, Brazil

[2] School of Computer Science, Carnegie Mellon University (CMU), Pittsburgh, PA 15213, USA

Keywords Automatic software repair · Automated program repair · Genetic improvement · Genetic programming · Crossover operator · Mutation operator

1 Introduction

Maintaining high quality software as it evolves is an expensive problem, to the point that it typically dominates software life-cycle cost (Pressman 2001). Software evolution consists of a number of interrelated activities, including bug finding and fixing, and feature implementation. In practice, these activities are performed manually, by expert human developers. However, substantial evidence suggests that the number of person hours required simply to deal with the volume of bugs reported for a particular project is impracticable; more, of course, are required for feature development. As a result, projects ship with both known and unknown defects (Liblit et al. 2005), and software quality remains a pressing practical concern (Britton et al. 2013).

This problem motivates a growing body of recent work in Search-Based Software Engineering (SBSE) (Harman et al. 2012) that applies meta-heuristic techniques like Genetic Programming (Koza 1992) to improve software by *evolving* patches to source code. Among other applications, such techniques have been used to repair defects (Le Goues et al. 2012b; Arcuri and Yao 2008; Arcuri 2011), implement new features by porting functionality from one program to another (Petke et al. 2014; Barr et al. 2015), improve performance characteristics like energy consumption (Bruce et al. 2015; Schulte et al. 2014), or otherwise improve or mitigate the cost of the testing or bug fixing process (Oliveira et al. 2013; Moncao et al. 2013; Machado et al. 2016; Freitas et al. 2016). The core goal in these techniques is to search the solution space of potential program improvements for edits to an input program that, e.g., fix a bug without breaking other functionality, typically as revealed by tests.

A key innovation in this domain is to represent candidate solutions as small *edit programs*, or *patches* to the original program. Early work adapted traditional tree-based program representations from the genetic programming literature to evolve entire programs toward particular quality goals (Weimer et al. 2009; Arcuri and Yao 2008). By contrast, the *patch representation* has significant benefits to scalability (and, albeit to a lesser degree, expressive power) (Le Goues et al. 2012), and genetic improvement techniques that use it have been applied to substantially larger programs than previously considered (e.g., `wireshark`, `php`, `python`, `libtiff` (Le Goues et al. 2015); VLC (Barr et al. 2015)). The patch representation is now commonly used across the domain of Genetic Improvement, a field which treats a program itself as genetic material and attempts to improve it with respect to a variety of functional and quality concerns (Silva and Esparcia-Alcázar 2015).

Our key contention in this article is that, despite its importance to scalability, the patch representation as currently formulated fundamentally overconstrains the program improvement search space by irreducibly conflating its constituent subspaces. This results in a more and needlessly difficult-to-traverse landscape. Consider the patch representation used in GenProg (Weimer et al. 2009; Le Goues et al. 2012b), a well-known program repair method that uses a customized Genetic Programming meta-heuristic to explore the solution space of possible bug fixing patches; its treatment of patches is prototypical, and thus illustrative. The *genome* in the GenProg search approach consists of a variable-length sequence of tree-based edits with respect to the original program code. Each gene is a single edit of the form `Operation`(*Fault*, *Fix*). `Operation` is an edit operator (one of *insert*, *delete*, or *replace*; *swap* has also been investigated); *Fault* captures the modification point for the edit, or the

fault location; and *Fix* represents the statement that will be inserted when necessary.[1] Each gene thus composes information along the three subspaces underlying the program repair search problem (operator, fault, and fix) (Le Goues et al. 2012a; Weimer et al. 2013).

This high gene granularity may unhealthily constrain the search problem, influencing the design of both the mutation and crossover operators. Mutation operators enable search space *exploration* by identifying new beneficial solution properties (Rothlauf 2011). This helps avoid local optima. However, at this granularity level, mutation may only make large changes to individual candidate solutions, via the construction of indivisible edits. Crossover creates new candidate solutions by combining parts of previous candidates (enabling *exploitation* of partial solutions). One feature of a healthy fitness landscape is a set of small, low-order building blocks that the crossover operator can identify, recombine, and propagate over the course of evolution (Holland 1992). For the purposes of program improvement, each subspace may contain independently relevant information, but the crossover operator cannot independently recombine them. That is, partial templates or schema of relevant information from a single subspace cannot be independently reused.

We propose a novel *subpatch representation*, a lower-granularity representation for search-based program improvement that enables a less constrained search strategy without substantial loss to scalability. This representation continues to be linear, but subspace values are manipulable. We speculate that such a representation will allow a meta-heuristic search strategy to more effectively explore the space, by enabling direct traversal and recombination of the constituent subspaces. We assess this representation along several dimensions by instantiating it in the GenProg technique for automated defect repair, in the context of new operators for crossover and mutation, which we assess in terms of their impact on speed search and success. Thus, the main contributions of this article are:

- The subpatch representation, a new representation for genetic program improvement that enables explicit traversal of and recombination between repair subspaces.
- Six new crossover operators that leverage this representation.
- A new mutation operator that manipulates search subspaces individually.
- A novel per-individual memory method to fix "broken" individuals.
- analysis of genotypic distance to surface and identify important characteristics of the novel representations and crossover operators.

Some of these points were previously presented (Oliveira et al. 2016). In our previous work, we proposed and evaluated the new subpatch representation for GenProg specifically in the context of our novel crossover operators. This article extends those contributions to include:

- **Additional quantitative results.** We have increased the number of buggy programs in our evaluation from 43 to 110 examples, increasing to 30 random runs per program per repair effort. This enables stronger claims of statistical significance. We have also added a larger, real world example program to our dataset to substantiate our scalability claims.
- **Additional qualitative results.** We assess patches produced for the IntroClass dataset on the available independent held-out test suites provided with that benchmark, to mitigate and assess the risk of low-quality patches. We also describe several patch examples.

[1] Note that GenProg manipulates C programs at the statement-level, but the formulation generalizes to arbitrary languages and granularity levels.

- **A novel mutation operator.** We propose and evaluation a new mutation method that allows for mutation within the search subspaces.
- **A novel genetic memory method.** Our prior work proposed a method to retain genetic memory to inform the correction of individuals that the new crossover operators could "break." However, our results suggested that the proposed method did not substantially improve search performance. Here, we revisit genetic memory, proposing to use it on an individual- rather than pool-level. We empirically show that this formulation, unlike the prior approach, is effective.
- **Distance analysis.** We perform an analysis of genotypic distance to understand characteristics underlying the improvement provided by the new representation and crossover operators.

The remainder of this paper is organized as follows. Section 2 presents background on genetic programming, in particular (though not exclusively) for program repair. Section 3 describes our new representation, and the new mutation and crossover operators it enables, along with a new technique for per-individual "memory." Section 4 presents experimental setup, results, and discussion. Section 5 discusses related work; we conclude in Section 6.

2 Background

Search-based program improvement leverages meta-heuristic search strategies, like genetic programming, to automatically evolve new programs or patches to improve an input program. These improvements can be either functional (e.g., bug fixing (Le Goues et al. 2012a), feature grafting (Langdon and Harman 2015; Barr et al. 2015) or quality-oriented (e.g., energy usage (Bruce et al. 2015; Schulte et al. 2014)). We focus on automated program repair, GenProg in particular, but anticipate that our innovations for representation should generalize. In this section, we provide background on Genetic Programming in general (Section 2.1) and its instantiation for repair in GenProg (Section 2.2). We discuss related work in depth in Section 5.

2.1 Genetic Programming

Genetic Programming Genetic Programming (GP) is a computational method inspired by biological evolution that traditionally evolves computer programs toward particular functionality or quality goals (Holland 1992; Forrest 1993). At a high level, a GP maintains a population of program variants, each of which corresponds to a candidate solution to the problem at hand. The variants are traditionally represented in memory as trees, to which evolutionary operations are applied. Linear Genetic Programming (LGP) is a special case of GP which represents programs as a sequence of imperative instructions, which can be represented in memory linearly (Brameier and Banzhaf 2007). Individuals in the patch representation are sequences of tree-based edits with respect to an original program, which can be considered an instantiation of LGP.

Each individual in a GP/LGP population is evaluated for its *fitness* with respect to a given objective function. Higher-fitness individuals are more likely to be selected to subsequent generations. New candidates are produced via domain-specific *mutation* and *crossover* operators, which modify intermediate variants and recombine partial solutions, akin to biological DNA mutation and recombination. Crossover combines partial solutions, providing for the

exploitation of existing solutions, promoting implicit genetic memory. Mutation enables *exploration*, helping the search avoid local optima (Rothlauf 2011).

Search Landscapes A GP is an instance of an Evolutionary Algorithm (EA). In the context of EAs, a *schema* is a template that identifies a subset of strings (in a GA) or trees (in a GP) with similarities at certain positions (genes) (Goldberg 1989). The fitness of a schema is the average fitness of all individuals that match or include it. Holland's *schema theorem*, also called the fundamental theorem of genetic algorithms (Holland 1992), says that short, low-order schemata with above-average fitness increase exponentially in successive generations. That is, partial solutions propagate and increase in quantity over the course of evolution. The schema theorem informs the *building block hypothesis*, which states that a genetic algorithm seeks optimal performance through the juxtaposition of such short, low-order, high-performance schemata, called *building blocks*. Crossover ideally serves to combine such schemata into increasingly fit candidate solutions. Mutation can disrupt building blocks, but also allows the search to find new beneficial characteristics.

Distance Analysis Evolutionary search spaces can be defined topologically, allowing elements to be measured with respect to one another, e.g., using Manhattan, Euclidean, or Hamming distances (Rothlauf 2011). Distance analyzes look at characteristics of populations in terms of distance between individuals, and are commonly used in search based algorithm research to help clarify and inform improvements to a given problem by highlighting convergence behavior, diversity, complexity, locality or uniformity (Kim and Moon 2004). Distance metrics can measure variation between either phenotypic or genotypic features (DeJong 1975). Genotypic distance compares individuals based on actual genome or representation (such as the patch encoding, in our domain). Phenotypic distance looks at the variant as it is expressed either for or by the fitness function. The latter approach is more common in practice, since genotypic measures can mask important bitwise diversity that manifests very differently phenotypically (Morrison and De Jong 2001). However, there exist domains in which genotypic analysis is informative.

2.2 GenProg for Program Repair

Overview GenProg is a program repair technique predicated on Genetic Programming. GenProg and GenProg-like techniques have been applied to different languages and abstraction levels, like Java (Kim et al. 2013; Orlov and Sipper 2011) and compiled code (Schulte et al. 2010); we focus on GenProg for C (Le Goues et al. 2012a, b; Weimer et al. 2009). GenProg takes as input a program and a set of test cases, at least one of which is initially failing. The search goal is a patch to that input program that leads it to pass all input test cases. Using test cases to define desired behavior and assess fitness is fairly common in research practice, e.g., Long and Rinard (2016), Qi et al. (2014), Mechtaev et al. (2016), Kim et al. (2013), and Le et al. (2016). Although tests only partially specify desired behavior, they are commonly available and provide an efficient mechanism to evaluate intermediate variants and constrain the search space.

Thus, at a high level, the search generates an initial set population of variants; evaluates fitness by applying each candidate patch to the initial program and running the result on the supplied test cases; selects a smaller subset of the population pseudo-randomly, weighted by fitness, and then generates new variants using mutation and crossover (described next). This process iterates until either a solution is found or a pre-specified resource limit is met.

Experimental results demonstrate that GenProg cost-effectively scales to defects in real world software (Le Goues et al. 2015). However, there remain a large proportion of defects that it cannot repair. We focus particularly on the way that GenProg's patch representation results in a suboptimal fitness landscape for the purposes of a healthy adaptive algorithm.

Search Space The program repair search problem can be formulated along three subspaces (Le Goues et al. 2013; Weimer et al. 2013): the `Operation`, or the edits that can be applied; the *Fault location(s)*, or the set of possibly-faulty locations to which edits may be applied; and the *Fix code*, or the space of code that can be inserted into the faulty location. GenProg constrains this trivially infinite space in several ways: (1) it uses the input tests to *localize* the bug to a smaller, weighted program slice (e.g., Jones et al. (2002), Saha et al. (2011), and Wong et al. (2016)), (2) it uses coarse-grained edit operators at the C statement level (*insert*, *replace*, and *delete*; *swap* has also been explored), and (3) it restricts the generation of fix code to the reuse of code from within the same program or module, leveraging the *competent programmer hypothesis* (Martinez and Monperrus 2015; Barr et al. 2014) while substantially reducing the amount of possible fix code that must be considered.

Representation GenProg canonically represents individual solutions as a patch composed of a variable-length sequence of high-granularity edit operations. Figure 1, top, demonstrates pictorially. Each edit takes the form of `Operation`(*Fault*, *Fix*). `Operation` is the edit operator; *Fault* is the modification (or fault) location; and *Fix* is the statement to insert when `Operation` is a *replace* or *insert*.

Mutation A mutation step as applied to an individual variant pseudo-randomly constructs a new edit operation and then appends it to the existing (possibly empty) list of edits that describe that variant. A destination statement s is chosen from the set S of permitted statements weighted by the fault localization strategy. Typically, statements executed exclusively by failing test cases are given a certain weight, while those executed by both failing and passing test cases are given another. Statements that are not executed by a failing test case are excluded from mutation. GenProg chooses between each of the three edit types, typically uniformly at random, and then pseudo-randomly selects *Fix* code to complete the edit type as necessary. Fix code is selected from within the program

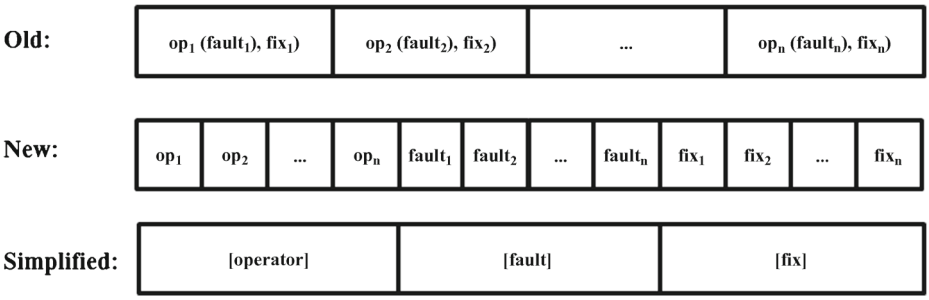


Fig. 1 Old representation (top); Subpatch representation (middle); simplified (bottom)

under repair, subject to certain semantic constraints (i.e., to avoid moving variables out of scope).

Crossover Crossover combines information from two *parent* individuals to produce two *offspring* individuals. GenProg uses a one-point crossover over the edits composing each of the parents. Given two individuals, GenProg selects a random cut point in each, and then swaps the tails of each edit list to produce two new offspring that each contain edit operations inherited from both parents. Crossover does not create new edits.

3 Approach

Our goal is to enable efficient recombination of genetic information while maintaining the scalability and efficiency of the patch representation. The building block hypothesis states, intuitively, that crossover should be able to recombine very small schemata into small schemata of generally increasing fitness. Instead of building high-performance strings by trying every conceivable combination, better solutions are created from the best partial solutions identified over previous generations. We posit that the current patch representation for program repair does not lend itself to the recombination of such small building blocks, because each edit indivisibly combines information across all three subspaces. Thus, partial information about potentially high-fitness features of an individual (e.g., accurate fault localization, a useful edit operator) cannot be propagated or composed between individuals. We conceive of the schemata in this domain as a template of edit operations, where certain operations and their order are necessary to represent key individual information. We instantiate this conception in a new subpatch representation, and then propose new crossover and mutation operators over it.

We first introduce an illustrative running example (Section 3.1). Then we present a lower granularity representation and a mapping to it from the existing patch representation (Section 3.2). We introduce a new mutation operator (Section 3.3), three new crossover operators: OP1SPACE, UNIF1SPACE, and OPALLS (Section 3.4.1), and then each of these new operators with a memory mechanism (Section 3.4.2). Finally, we outline our approach for computing genotypic distance (Section 3.5), for use in our distance analysis.

3.1 Illustrative Example

For the purposes of illustration, we use integer indices to denote numbered statements taken from a pool of potential faulty locations and candidate fix code. Consider a bug that requires two edits to be repaired: `Insert`(*5*, *3*) `Delete`(*2,*). Consider two candidate patches that contain all material necessary for this repair:

1. `Insert`(*5*, *6*)`Replace`(*2*, *9*)
2. `Replace`(*4*, *3*)`Delete`(*2,*)`Insert`(*8*,*7*)

Note that the deletion in variant (2) is correct, and only needs to be combined with the appropriate insertion. The current crossover operator can easily propagate it into subsequent generations. However, crossover cannot produce the `Insert`(*5*, *3*), even though the insertion in variant (1) is only one modification away, along the fix axis, and (2) contains the correct fix code in its first replacement. Mutation is also of limited utility, as it cannot modify the 6 to a 3 and must be relied upon to construct the correct insertion from scratch.

Overall, the only way to achieve the desired solution is by combining edits that compose semantically to the desired solution, which is improbable, or relying on mutation to produce the correct insertion from nothing.

3.2 Decoupled Representation

Our novel *subpatch representation* decouples the three subspaces, decreasing edit granularity, as illustrated in the middle of Fig. 1. For simplicity, this representation can be reduced to a single-dimensional array by concatenating the three subspace arrays (Fig. 1, bottom). Note that we add a ghost *Fix* value to the `Delete` operator to maintain consistent subspace lengths, i.e.,, operation `Delete`(*1,*) maps to the array "[d,1,1]".

We maintain the original representation for GP steps outside of mutation and crossover, primarily for ease of implementation, and to focus our study. The translation from the canonical to this new representation is straightforward, as illustrated in Fig. 2. *Decode* from the new representation back to the canonical representation is similarly straightforward in the simple case. However, because the crossover operators applied to the new representation can lead to information loss, decode can sometimes result in "broken" genes, as we discuss subsequently.

3.3 Mutation Operator

GenProg's canonical mutation operator can create large changes, because it always inserts a completely new operation in the patch list that comprises an individual. These modifications can thus be destructive and are thus difficult to compose. This is perhaps one reason that patch-based program improvement often produces patches that reduce to a single edit. That is, many nominally multi-edit patches produced by these techniques contain unnecessary changes that may be removed in post-processing without affecting the patch's bug fixing behavior (Qi et al. 2015).

The subpatch representation allows for the direct mutation of particular subspaces. Our new mutation operator, *Subspace Mutation*, either appends a new edit to the existing genome (as before), or perturbs an existing edit (a low-level perturbation), as illustrated in Fig. 3. With a proportional mutation rate of one, standard in our and previous repair experiments, each individual in a population either receives a new edit, or receives an update to the subspace of an existing edit. Because low-level perturbation does not increase patch size, this operator sometimes maintains individual length rather than increasing it monotonically. This choice is motivated by evidence suggesting human bug fixing patches are typically short (Martinez and Monperrus 2015).

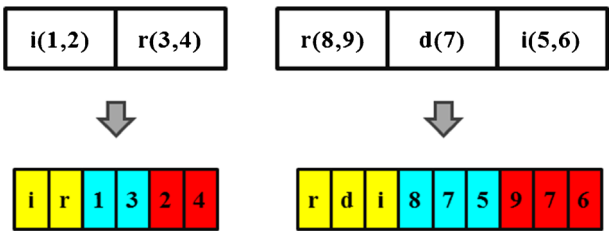


Fig. 2 Mapping an individual to the new representation. Subspaces are distinguished by color: Yellow is the `Operator` subspace; Blue, the `Fault` subspace; Red, the `Fix` subspace. "i" denotes an `Insert` operation; "r", `Replace`; and "d", `Delete`

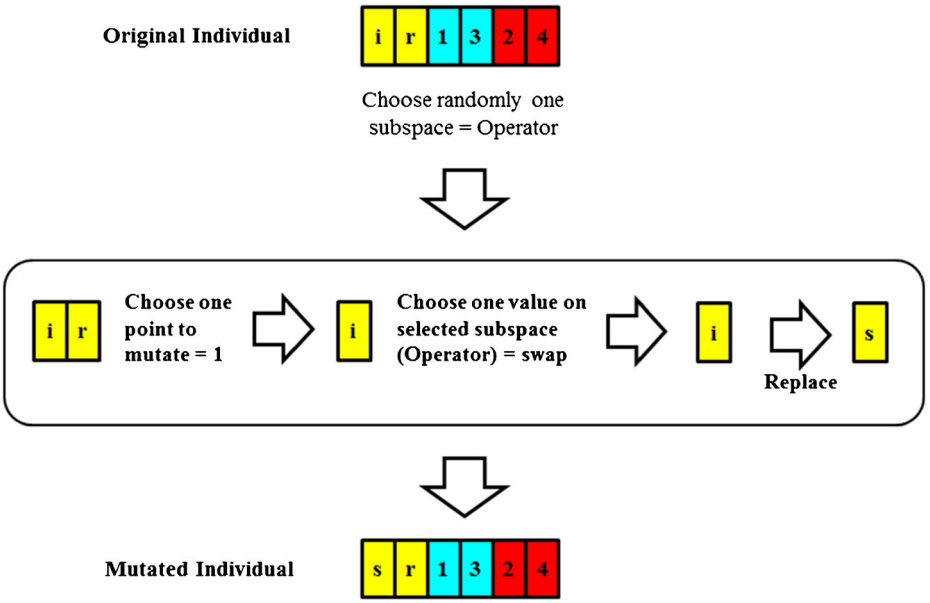


Fig. 3 Example demonstrating the new mutation operator

3.4 Crossover

We propose new crossover operators to leverage and analyze the proposed subpatch representation. OP1SPACE and UNIF1SPACE apply to a single subspace, while OPALLS applies to the whole chromosome. These three operators, alone (Section 3.4.1) and also augmented with a memory mechanism (Section 3.4.2), result in six total new operators.

3.4.1 Three New Crossovers

The crossover process can be divided in four main steps: encode, recombination, decode and remove invalid genes. As the proposed crossover operators are differentiated in the recombination step, see Algorithm 1.

Algorithm 1 High-level crossover procedure; the recombine procedure varies and is parameterized by the operator in question; repairGenes repairs genes left "broken" by the given crossover procedure, either by dropping broken edits, or repairing them using the memory cache (details in text)

```
1 Crossover(p1,p2)
2       let p1' = encode(p1) in
3       let p2' = encode(p2) in
4       let subspace = chooseOneRandom(op, fault, fix)
5       let o1, o2 = recombine(p1', p2')
6       let o1' = repairGenes(o1)
7       let o2' - repairGenes(o2)
8       return o1', o2'
```

The *encode* step translates the canonical to the new representation, as illustrated in Fig. 2; *decode* translates back. The decoding is informed by the structure of the arrays describing a variant, illustrated in Fig. 4.

One Point Crossover on a Single Subspace (OP1SPACE) OP1SPACE applies one-point crossover to a single subspace (see Fig. 5 for a visual presentation). It explores new solutions in a single neighborhood, while maintaining potentially important blocks of information in the other subspaces. For example, an individual may be modifying the appropriate location, but with the wrong edit; this operator allows the location information to be retained.

Given two parent variants, OP1SPACE chooses one of the three subspaces uniformly at random, and then randomly selects a single cut point in the subspace to be used in both parents, bounded by the minimum length of the chosen subspace (so as to result in a valid point in both parents). Tails beyond this cut point are swapped, generating two offspring. The portions of the individuals relative to the unselected subspaces are unchanged.

Uniform Single Subspace (UNIF1SPACE). A *uniform* crossover operator combines a uniform blend of data from each parent (Rawlins 1991), promoting greater exploration. In certain domains, a uniform operator can be problematically destructive (Le Goues et al. 2012). We thus propose a uniform operator along a single subspace, promoting a constrained exploration.

Given two parent variants, UNIF1SPACE chooses one subspace uniformly at random. The larger of the two subspaces is truncated to the length of the shorter. The operator then generates a binary mask of the length of the shorter subspace. This mask is used to choose the source of value in every position in the chosen subspace; one offspring will be the complement of the other, based on the mask. Figure 6 shows an example, choosing the `Operator` subspace.

One Point Across All Subspaces (OPALLS). OPALLS follows the same rules for crossover point selection as OP1SPACE, applied to the entire individual, mixing subspaces. Given two parent variants, OPALLS selects one of the subspaces uniformly at random,

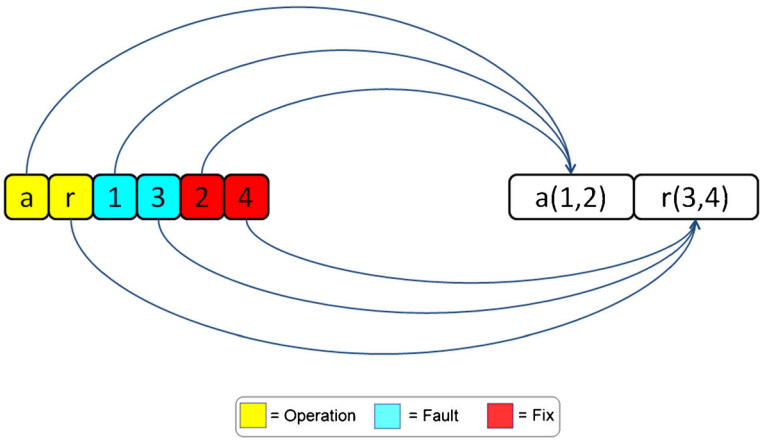


Fig. 4 Example of the decode process

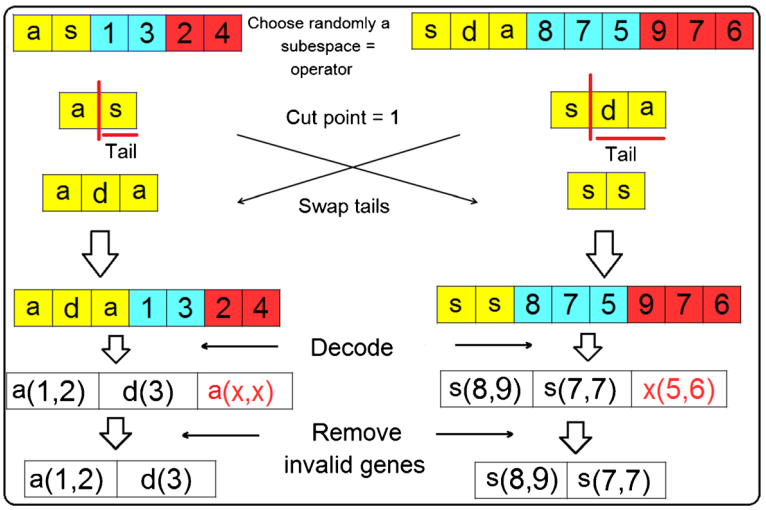


Fig. 5 Example of OP1SPACE applied to a pair of variants

and then a single random cut point in that subspace. OPALLS then does not differentiate between subspaces, instead swapping everything after the cut point to produce offspring. The selected crossover point is bounded by the minimum length of the chosen subspace to avoid mixing the values of different subspaces.

This operator can maintain larger basic blocks than UNIF1SPACE, with a greater capacity for information exchange than OP1SPACE. Large blocks containing valuable information

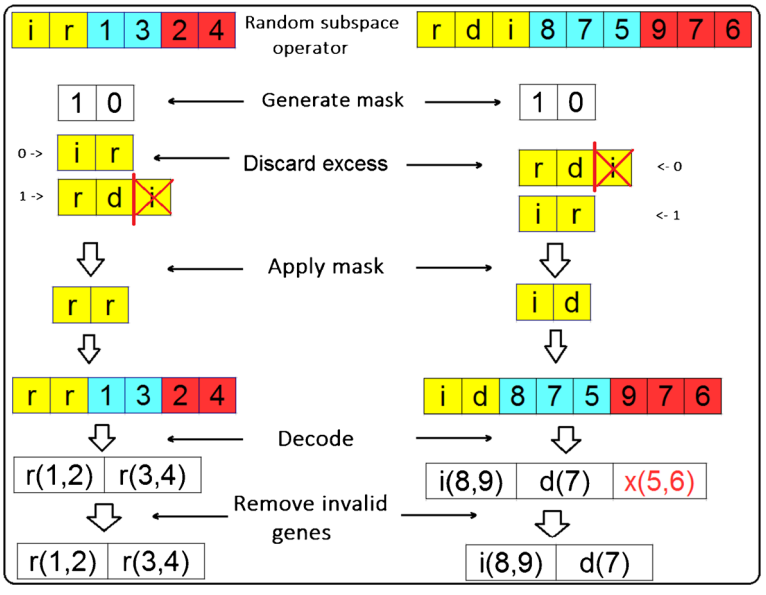


Fig. 6 Example of UNIF1SPACE applied in the `Operator` subspace. Note that this operator can create highly diverse offspring, but may also dissolve basic blocks

within at least one subspace are not dissolved, maintaining certain good information, while completely modifying other edits. For example, this crossover could keep all the original values for `Operator`, while still changing all the `Fix` values and some of the `Fault` values, as illustrated in Fig. 7.

Information Loss Unlike the canonical crossover, our new crossover operators can result in incomplete edit operations in offspring when parents are of different lengths. This can result in either excess or missing data in unchanged subspaces (e.g., an insertion without a corresponding fix statement; Fig. 5 provides an example). Without the memory cache, when converting individuals back to the canonical representation, the *decode* step simply drops invalid genes. This reduces the size of the offspring variants.

3.4.2 Genetic Memory

To mitigate the risk of data loss, we previously proposed a *pool-based memory* scheme to help reconstruct valid from invalid genes (Oliveira et al. 2016). This approach maintains a *pool*, or cache, of unused genes that were dropped from any offspring throughout a given search (distinguishing dropped genes by subspace). This pool is used to fix any broken genes produced by destructive crossover. Such individuals are fixed by selecting elements from the pool at random to replace missing pieces of genes. This process occurs after decode, but before invalid genes are removed. For example, in Fig. 5, the operation "i" in the first offspring, and the fault and fix values (5 and 6) in the second would be stored in the pool for use in subsequent generations. The memory mechanism also checks the cache for fault and fix values to repair the first offspring, and an operator to repair the second, choosing between multiple available options at random. This fix procedure is performed whenever possible, whenever genes are broken by crossover.

However, we found that *pool-based memory* did not improve search efficiency. We speculate that this occurred because the pool-based cache does not preserve relationships between broken individuals and candidate repair genes, and thus individuals can receive

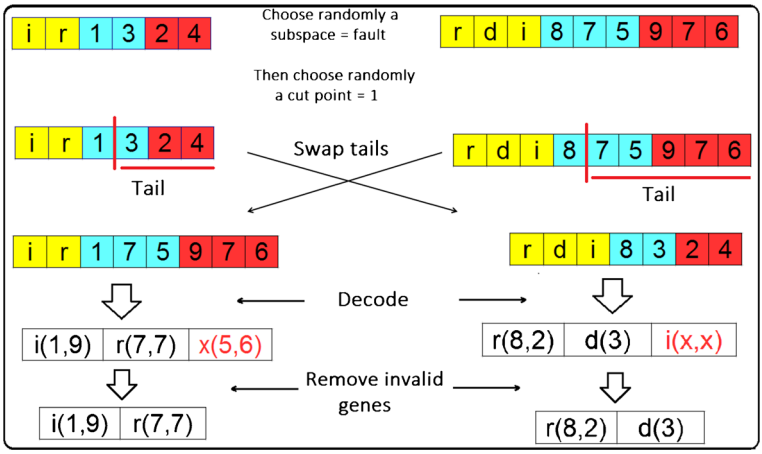


Fig. 7 Example of OPALLS

from the pool unrelated genetic material. This adds noise to the evolutionary process without a commensurate benefit.

We therefore propose here a *per-individual memory scheme* that preserves links between candidate genes and individuals. This approach maintains per-individual caches to store material dropped during crossover over the course of evolution of variant lineages. The scheme otherwise works as before.

3.5 Distance Metric

As discussed in Section 2.1, distance analysis can help clarify and inform improvements to a given search problem (Kim and Moon 2004). Although phenotypic distance is more commonly appropriate (Morrison and De Jong 2001), in this work, we analyze genotypic distance. We do this because phenotypic distance is typically measured by the fitness function. Current genetic improvement search strategies, including GenProg, use test suites for this purpose, which present a highly step-wise function with extensive plateaus (Forrest et al. 2009; Fast et al. 2010). It is therefore minimally informative for the purposes of distance analysis.

We measure distance between individuals using Levenshtein Distance on the individual patches encoded as described above; Levensthein distance generalizes Hamming distance to operate on strings of different lengths (Rothlauf 2011). Summarizing, the distance between the strings A and B is the number of alterations to A necessary to convert it to B. Figure 8 shows an example of distance computation between two individuals of different lengths (s denotes "swap"; i, "insert"; and d, "delete"). We simplify the calculation with a heuristic that pads subspaces with empty spaces, so as to avoid inadvertently comparing values of different subspaces. Equivalent characters between individuals receive a count of 0; different characters, 1 (corresponding to the number of edits required to change one value into another). The distance between two individuals is the sum of these counts. We use this measurement in our analysis in Section 4.5.

4 Experiments

This section presents experiments evaluating our new crossover operators, with and without per-individual genetic memory, and the new mutation operator. We also use distance analysis to surface relevant characteristics of the different operators with respect to the underlying search.

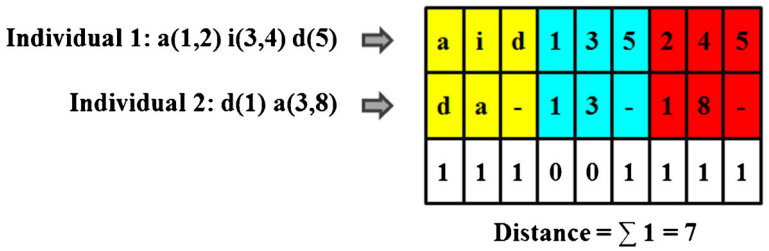


Fig. 8 Example of Levenshtein distance applied to patches

Table 1 Benchmarks; Tests used as input to each repair algorithm ("Repair") and held out, for evaluation of produced patch generalizability ("Held-out"), where available; and buggy versions of each program

Program	Versions	Tests	
gcd	1	11	
zune	1	24	
libtiff	24	1392	
		Repair	*Held-Out*
checksum	8	6	10
digits	18	6	10
grade	16	7	9
median	14	7	6
smallest	14	7	8
syllables	14	6	10

4.1 Setup

Benchmarks Table 1 shows the C programs we use in our evaluation. `gcd` and `zune` have classically appeared in previous assessments of program repair.[2] Both include infinite loop bugs. The other six program classes are drawn from IntroClass (Le Goues et al. 2015), a set of student-written versions of small C programming assignments in an introductory C programming course. IntroClass contains many incorrect student programs corresponding to each problem. Because our experiments are computationally intensive, we randomly subsample 30% of the buggy programs for each assignment. We also include defects from the `libtiff` problem, from the ManyBugs benchmark (Le Goues et al. 2015), which presents bugs in a real-world library consisting of many files and thousand of lines of code.[3]

We use results on the `libtiff` program bugs to validate our claims about scalability and generality to real-world programs. Other than this, we focus on small programs. However, this is important for our evaluation. First, it allows us to run many random trials for more iterations than is typical in program repair evaluations, with acceptable computational cost. Second, our small programs are fully covered/specified by their black box tests, which allows for a separation of concerns with respect to fitness function quality and completeness. That is, the tests provided with real world programs can be *weak proxies* for correctness (Qi et al. 2015), increasing the risk of low-quality patches. We sidestep this issue by evaluating in part on well-specified programs (Smith et al. 2015).

Additionally, the IntroClass programs are associated with two full coverage, test suites that independently exercise functionality. We use the black box tests provided with the benchmark to direct the search. For these programs, we use the second, generated test suite to independently evaluate the degree to which any produced patch generalizes. Note that our primary goal is not to perfectly characterize or measure bug fixing patch quality, an unsolved research problem, but rather to improve the traversal of the program improvement fitness landscape. However, patch quality is a core concern in program repair and its evaluation in a research context, and thus we use these held-out test suites where available to augment our evaluation.

For all observed effects described below, we use Wilcoxon rank-sum tests to assess whether performance distribution between the proposed operators and the original ones are

[2]Both available from the GenProg project, http://genprog.cs.virginia.edu/

[3]IntroClass and ManyBugs are available at http://repairbenchmarks.cs.umass.edu/

statistically significant ($\alpha = 0.05$). The samples for the statistical test are the amount of fixes found in each version of the executed problems.

Parameters and Metrics We executed 30 random trials for each program version, totalling 2820 runs (consistent with recommendations for evaluations of stochastic algorithms (Arcuri and Briand 2011)). The search concludes either when it reaches the generational limit or when it finds a patch that causes the program to pass all provided test cases. As all analyzed algorithms evaluate the same number of individuals per generation, the generation stop criterion supports a controlled comparison.

For simplicity, we omit the post processing patch minimization step based on delta debugging (Zeller 1999). Prior work suggests that this step does not have a statistically significant impact on patch quality (Smith et al. 2015). Our primary concern is effectively traversing the search landscape, to which these post processing measures have no relationship, and thus we omit them for a more focused study.

Our parameters for `gcd`, `zune` and IntroClass problems are: Elitism = 3, Generations = 30, Population size = 30, Crossover rate = 0.5, Mutation rate = 1, Tournament k = 2. The new mutation operator selects between appending a new edit and performing a low-level perturbation with 50-50 probability. Because the `libtiff` problems are larger and take longer to run, we set Generations = 10 and Population size = 40, to render the experiments viable with our compute resources. The evaluation metrics are the success rate and the number of test suite evaluations to repair, a machine- and test suite- independent measure of time.

4.2 Crossover Operator Results

Success Rate Table 2 presents the success rate of experiments for all operators and problems (higher is better). No repairs were found for programs in the `grade` dataset. Prior results show that search-based program repair techniques have a low success rate on `grade` problems. For example, GenProg repairs 2 out of 226 buggy programs (Le Goues et al. 2015). As we consider a random subset of the `grade` problems, it is consistent with these prior results that GenProg might not find any patches, regardless of operator. This is especially true given that the benefits operators provide to success rate and convergence time (discussed subsequently) are modest (and inconsistent with a fundamental change in success rate overall). For the other problems, Table 2 shows that UNIF1SPACE with memory presents the best success rate, increasing the success rate on average by 50.9% over the others.

At a per-problem level, for the `checksum` problem, OP1SPACE with genetic memory was best. On `digits` programs, as well as `zune`, almost all operators are roughly equivalent to the original. In the `gcd` problem, all operators produced a high fix rate, but UNIF1SPACE operators were best. For the `median` problem, the operators that manipulate single subspaces were best (UNIF1SPACE and OP1SPACE), especially without genetic memory. Furthermore, all proposed operators were much better than the original, e.g. UNIF1SPACE without memory was 814.2% better. For `syllables` problems, all proposed operators demonstrated a low fix rate, but outperformed the original. For `smallest` programs, UNIF1SPACE and OP1SPACE were better than OPALLS, and much better than the original operator, e.g. UNIF1SPACE with genetic memory outperformed the original operator by 271.18%. Finally, for `libtiff`, UNIF1SPACE with memory performed the best. With the exception of OP1SPACE without memory and OPALLS with memory, all proposed operators outperformed the original crossover.

Table 2 Success rate (percentage) over all runs

	Original	OP1SPACE		UNIF1SPACE		OPALLS	
Memory?	N/A	No	Yes	No	Yes	No	Yes
gcd	70.00	80.00	**90.00**	**90.00**	**90.00**	86.67	80.00
zune	96.67	**100.00**	**100.00**	83.33	96.67	**100.00**	86.67
checksum	12.50	17.50	**19.50**	17.08	16.25	17.50	19.17
digits	5.37	**5.56**	5.19	**5.56**	**5.56**	**5.56**	**5.56**
grade	0.00	0.00	0.00	0.00	0.00	0.00	0.00
median	5.00	44.76	43.33	**45.71**	41.43	35.71	32.14
smallest	19.05	69.05	65.24	70.24	**70.71**	49.76	49.29
syllables	0.71	6.67	5.24	5.95	**6.90**	**6.90**	6.19
libtiff	40	37.92	41.67	42.08	**48.75**	42.5	44.0
Average	27.7	40.16	41.13	40	**41.8**	38.28	35.89
Total	11.29	27.25	26.67	27.67	**27.76**	22.9	21.30
Significance	–	∗	∗	∗	∗	∗	∗

We aggregate across IntroClass and `libtiff` problems for presentation. Bold text identifies the best results for a problem class. The "∗" denotes results that are statistically significant (p-value ≤ 0.05) with respect to the original operator

Efficiency Table 3 presents test suite evaluations, or average fitness evaluations, to repair (lower is better). We omit `grade`, as no repairs were found in any run. Overall, the operator with the best success rate was not the most efficient. This is consistent with our expectations because more difficult problems are harder to solve, and thus succeeding at them can pull up the average time to repair (Le Goues et al. 2012). Furthermore, increasing the representation granularity commensurately increases the size of the search space. Thus, as expected, the

Table 3 Number of test suite evaluations to find a repair

	Original	OP1SPACE		UNIF1SPACE		OPALLS	
Memory?	N/A	No	Yes	No	Yes	No	Yes
gcd	**21.71**	45.69	31.29	29.52	29.52	45.51	32.27
zune	5.76	4.04	**3.36**	4.50	4.04	3.77	4.60
checksum	**0.65**	20.55	25.58	23.57	21.58	25.97	26.13
digits	1.54	1.12	**1.02**	1.59	1.30	3.73	4.33
median	**37.35**	88.58	90.04	87.85	84.02	83.97	78.5
smallest	**8.39**	27.5	26.6	28.08	24.57	22.26	23.75
syllables	**2.6**	62.8	55.97	40.75	49.15	68.39	53.00
libtiff	**50.7**	61.9	60.8	58.3	72.13	61.21	61.17
Average	**16.08**	39.02	36.83	34.27	35.78	39.35	35.47
Significance	–	∗	∗	∗	∗	∗	∗

We omit `grade` (from Introclass) because no repairs were found. Bold text identifies the best results for a problem class. "∗" denotes results that are statistically significantly (p-value ≤ 0.05) different as compared to the performance of the original operator

search itself requires more iterations to find solutions. Additionally, although the nature of the representation (as tree-based edits over the AST) prevents the construction of any syntactically invalid candidate patches, decoupling the subspaces may result in semantically invalid patches (i.e., by associating fix code to a new candidate fault location, variables may be moved out of scope). The default representation precludes this possibility by using a scope check when selecting fix code to use in a location. Fitness evaluation (on these benchmarks, and generally (Le Goues et al. 2012)) is dominated by test case execution (rather than compilation) time, and so this injection of noise in the search space likely has only modest effects. It is possible to inject a scope check into the decode phase (filtering out invalid edits); we leave this improvement to future work. However, overall, the performance differences are not large, and it may be reasonable to exchange a slight loss of efficiency in favor of a more effective search strategy. On the other hand, memory decreases time to repair for most of programs, especially for OP1SPACE and UNIF1SPACE.

At a per-problem level, the original crossover operator outperformed the others for `checksum`, but as the success rate is low, high variability is unsurprising. The second best operator here is UNIF1SPACE without memory. For digits, the OP1SPACE with memory was best, followed by OP1SPACE without. In `gcd` the original outperformed the other operators, but the UNIF1SPACE was the best of the new operators. `zune` presents a low discrepancy within operators, with all search efforts converging rapidly, but OP1SPACE with memory was the most efficient. On `smallest`, `median`, `syllables` and `libtiff`, original performed the best.

Memory The new memory scheme increases success rate on average for OP1SPACE and UNIF1SPACE, suggesting that the mechanism is effective for operators that exchange genetic material between single subspaces. Although crossover operators without memory constrain patch size throughout the search, these results suggest that this benefit is outweighed by the ability to leverage genetic memory.

Subpatch Representation On average, all crossovers using subpatch representation outperformed the standard representation, even when genetic material is lost in decode. This significant improvement of all proposed operators suggests that the low-granularity representation is the main reason for the result. In terms of scalability, the subpatch representation does not use considerably more memory, and the computational cost of transformation is low. Thus, our results suggest that the proposed representation improves GenProg's ability to better exploit the search space, and is promising for other program improvement techniques predicated on patch-based genetic search.

4.3 Mutation Operator Results

Table 4 compares fix rates between canonical GenProg, GenProg with Subspace Mutation, and GenProg with the crossover operator representing the best fix rate for each problem. The new mutation operator is comparable to the original on `gcd`, `zune`, `checksum` and `digits`, but is much better on `median` (362%), `smallest` (74.96%) and `syllables` (504.22%), the problems for which we have the greatest amount of data. This suggests that the problem type may influence search performance.

Overall, the new crossover operators are more important to improved success rate than the new Subspace Mutation operator, though it does improve performance over canonical GenProg. This is consistent with the natural role of mutation, which can not on its own solve all elements of a complex search problem (as it cannot combine partial solutions). However,

Table 4 Success rate (percentage) over all runs, comparing mutation operators with the best-performing crossover Operator (UNIF1SPACE with memory)

Program	Original	Subspace Mutation	UnifSingle Memory
gcd	70.00	70.00	**90.00**
zune	96.67	**100.00**	**100.00**
checksum	12.50	10.83	**16.25**
digits	5.37	4.26	**5.56**
grade	0.00	0.00	0.00
median	5.00	23.10	**45.71**
smallest	19.05	33.33	**70.71**
syllables	0.71	4.29	**6.90**
Average	26.16	30.73	**41.07**
Total	8.26	13.76	**24.61**
Significance	–	*	*

"*" denotes results that are statistically significantly different (p-value ≤ 0.05) as compared to the results for the original mutation operator

this result supports the utility of Subspace Mutation over a subpatch representation, as it improves exploration over the original even with the original crossover.

4.4 Patch Quality

Performance on Held-Out Test Suites Our research goal is to understand, characterize, and more efficiently traverse the search landscape of patch-based program repair/improvement; generated patch quality is thus a largely orthogonal concern. Generated patch quality is an important concern in automated program repair research (Smith et al. 2015; Qi et al. 2015). It is thus important to ascertain the effect, if any, the newly proposed representation and operators have with respect to this concern. The IntroClass problems are released with two high-quality test suites, to assist in evaluations of exactly this type (Le Goues et al. 2015): patches may be generated with respect to one test suite, and then validated against a second, held-out, high-coverage test suite. Table 5 shows results in terms of the proportion of produced patches that generalize fully to the held-out test suites. Results indicate that the majority of produced patches pass all held-out tests. The patches produced for `digits` and `syllables` all do so, regardless of operator. The variation between the operators otherwise is not statistically significant.

Table 5 Proportion of generated patches that generalize to all held-out test cases.

	Original	OP1SPACE		UNIF1SPACE		OPALLS		Submutation
Memory?	N/A	No	Yes	No	Yes	No	Yes	N/A
checksum	100.0	97.6	100.0	100.0	100.0	100.0	100.0	100.0
digits	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
median	91.8	90.3	93.4	94.7	95.1	92.2	94.4	97.9
smallest	98.5	94.2	99.5	98.6	97.1	99.5	97.3	97.9
syllables	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Average	98.1	96.4	98.7	98.7	98.4	98.4	98.4	99.2

Example Patch For illustration, consider the following patched portion of a buggy `checksum` program, produced using UNIF1SPACE with memory (the operator with the best performance):

```
1  // ...
2  while (next != '\n') {
3  scanf("%c", &next);
4  + if (next == '\n') break;
5  sum += next;
6  }
7  sum = sum % 64 + 22;
8  + sum += next;
9  printf("Check sum is %c\n", sum);
10 \\ ...
```

The `checksum` program takes a single-line string as input. It should compute and output the sum of the integer codes of the characters in the string, modulo 64, plus the code for the space character. The original program incorrectly includes the newline in the sum, and adds the wrong ASCII value for space on line 7 (the ASCII code for space is 32, not 22).

The patch first causes the while loop to correctly break *after* reading `next` but *before* adding its result to sum. The insertion on line 8 adds the value of `next` to `sum`. `next` must be newline ("\n") on line 7. The ASCII value for newline is 10 which, when taken with the sum on the preceding line, correctly adds 32 to `sum` modulo 64. A human programmer is, of course, unlikely to produce this patch (a more natural approach would be to change the 22 to 32 directly). However, the operators at GenProg's disposal do not allow for the modification of constants in binary expressions, and instead must work around the problem at a statement-level granularity. This multi-edit patch still achieves the desired functional result.

4.5 Distance Analysis

Convergence Crossover is one of the primary factors in convergence, a desirable GP trait that indicates the maintenance of exploitable information between generations (and distinguishes a GP from a random walk). Offspring that are closer to one another than parents are to one another indicate desirable convergence behavior (Louis and Rawlins 1992; Rothlauf 2011).

Table 6 presents the results of calculating the average distance between parents, offspring, and parent and offspring individuals using each crossover operator, averaged over all generations and all runs. For the original operator, distance between offspring (Offspring1-Offspring2) decreases less as compared to distance between parents than it does with the other operators, at least in part because the selection of crossover point (which can be different for each parent) can highly influence the resulting offspring.

The newly proposed operators' distance between offspring significantly decreases compared to the distance between parents, indicating that the crossover approaches improve the convergence. Note that the new operators can never result in offspring distance greater than parent distance, because the offspring length is always bounded by parent sizes. Consider the subspace operators (OP1SPACE and UNIF1SPACE): Changes in one subspace cannot increase the length of the others, even if a chosen subspace is entirely modified.

Indeed, the effect of the proposed crossovers is more influenced by the parent length difference than the cut point (or mask, for UNIF1SPACE).

Table 6 Levenshtein distance between parents and offspring; numbers represent averages across the multiple experimental runs

	Original	OP1SPACE		UNIF1SPACE		OPALLS	
Memory?	N/A	No	Yes	No	Yes	No	Yes
Parent1-Parent2	8.26	15.37	15.35	13.12	21.13	14.63	20.56
Offspring1-Offspring2	7.87	11.46	11.44	6.42	17.40	9.41	17.40
Parent1-Offspring1	3.30	4.84	4.84	4.84	8.69	9.32	14.25
Parent1-Offspring2	6.27	11.79	11.76	8.26	15.89	5.32	8.69
Parent2-Offspring1	6.27	11.78	11.75	8.28	15.90	5.30	8.68
Parent2-Offspring2	3.30	4.84	4.82	4.85	8.7	9.32	14.25
Average	8.25	10.01	10	7.63	14.62	8.88	14.07

When the biggest distance between an offspring and a parent is smaller than the distance between the parents it also indicates that this crossover provide convergence. The Table 6 shows that all crossover operator provide convergence (Rothlauf 2011).

Figure 9 presents an example illustrating the point: operators that decrease or, at the very least, maintain offspring length as compared to parent variants, are more likely to decrease offspring distance over time, promoting convergence.

Diversity Although convergence is desirable, population diversity is still necessary to support exploration (Gupta and Ghafir 2012). We therefore evaluate the amount of genetic material each offspring inherits from each parent. Ideally, each offspring is equally distant from both parents (Mattfeld 2013).

Table 6 shows that offspring are typically closer to one of the two parent variants (Parent-Offspring), because offspring will always be closer to the longer parent. Although these results show less diversity than desirable, we also observe that the average parent-offspring distance is considerable. The distance between offspring and their more distant parent is almost equal to the distance between offspring within a given iteration. This demonstrates that the crossover operators still promote diversity. At an Operator-specific level, note that UNIF1SPACE and OP1SPACE always provide some recombination, while OPALLS can decrease diversity when the cut point is either at the beginning or end of a parent variant (resulting in identical offspring).

The average distance presented on Table 6 can be a convergence indicator, because as closer the population are, less diversity the crossover is providing. Thus, the UNIF1SPACE with memorization presented the highest average distance, except the UNIF1SPACE without

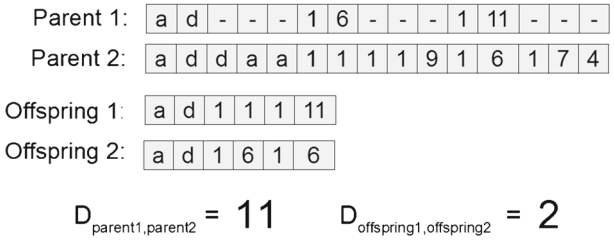


Fig. 9 Example of a crossover using OP1SPACE without memory, with cross point = 1 on the Fault subspace, with parent/offspring distances

memorization, all approaches presented higher average distance, mainly the versions with memorization. Thus, the new approach indicates that improve the diversity of the population. It is expected because the increase of the capacity of recombination of the new representation allow the capacity of create new individuals.

Crossover operators that affect only one subspace at a time produce offspring closer to their parents on average (as expected); those that modify over all subspaces are more disruptive, and consequently increase the population distance. That said, UNIF1SPACE with memory presents average distances comparable to OPALLS with memory, despite having a better fix rate. UNIF1SPACE, with the diversity provided by memory, results in long variants; in OPALLS, the higher distances come from a high mix of genetic material.

We conclude that these operators cannot (and do not) produce perfect diversity and convergence, as the balance between exploration and exploitation is defined by the mix of all operators and parameters (including, e.g., mutation rate and variety) rather than the single crossover component. However, we have characterized the way they contribute to diversity and search space exploitation, providing insight as to how they do so more effectively than the original operator.

4.6 Threats to Validity

One threat to the validity of our results is that our dataset may not be indicative of real-world program improvement tasks. We selected our programs because they allowed us to minimize other types of noise, such as test suite quality, allowing for a more focused study of operator effectiveness and sufficient data to make stronger statistical claims. We thus view this as a necessary tradeoff. Another important concern in program improvement work is output quality, as test-case-driven program improvement can overfit to the objective function or be misled by weak tests. We mitigate this risk by using high-coverage, high-quality test suites (Smith et al. 2015). Nevertheless, we evaluate the patch quality based on held-out test cases. Note that output quality is not our core concern, and the new representation and operators are parametric with respect to fitness functions and mutation operators, and thus should generalize immediately to other patch-based program improvement techniques that produce program improvements. Additionally, although genotypic distance illuminates characteristics of each crossover operator, it may not be sufficient to characterize all the ways the operators impact the search. A deeper analysis using phenotypic distance could highlight additional characteristics of the search, corroborating and extending these results, but would require innovations in the fitness function for program improvement before it can be truly informative.

5 Related Work

Most innovations in the Genetic Programming (GP) space for program improvement involve new kinds of fitness functions or application domains, with less emphasis on novel representations and operators, such as those we explore. However, there are exceptions to this general trend. Orlov and Sipper (2011) propose a semantics-preserving crossover operator for Java bytecode. Ackling et al. (2011) propose a patch-based representation to encode Python rewrite rules; Debroy and Wong (2010) investigate alternative mutation operators. Forrest et al. (2009) quantified operator effectiveness, and compared crossback to traditional crossover. Le Goues et al. (2012) examined several representation, operator and other choices used for evolutionary program repair, quantified the superiority of the patch

representation over the previously-common AST alternative, and demonstrated the importance of crossover to success rate in this domain. Although they do examine the role of crossover, they do not attempt to decompose the representation to improve evolvability, as we do, rather focusing on the effects of representation and parameter weighting. These results, along with our own, corroborate (Arcuri 2011), which demonstrates that parameter and operator choices have tremendous impact on search-based algorithms generally. Our research contributes to this area, presenting a new way to represent and recombine parents evaluating the influence of evolutionary operators on algorithmic performance, and characterizing that performance in terms of population diversity and information.

Our results demonstrate that in theory our proposed representation and novel operators can improve the creation and propagation of genetic building blocks, but we do not directly investigate the role of schema evolution in this phenomenon; we leave this to future work. For example, Burlacu et al. (2015) presents a powerful tool for theoretical investigations on evolutionary algorithm behavior concerning building blocks and fitness.

Informed by the building blocks hypothesis, Harik et al. (1999) proposed a compact genetic algorithm, representing the population as a probability distribution over a solution set, which is operationally equivalent to the order-one behavior of a simple GA with uniform crossover. They conclude that building blocks can be tightly coded and propagated throughout the population through repeated selection and recombination. This theory suggests that knowledge about the problem domain can be inserted into the chromosomal features, and GA can use this partial knowledge to link and build information blocks. The difficulty of representing a program in repair problem may be one of the reasons for its complexity.

Distance correlation has been used to characterize population diversity across various search landscapes (Morrison and De Jong 2001). Kim and Moon (2004) proposed more effective distance measures based on GA context, and show that they can be used as informative metrics. Mattfeld (2013) use distance to analyze crossover uniformity and information preservation. These results suggest several additional phenotypical distance measures we could leverage in this type of work, possibly informing future research on operators and parameters to improve search quality.

Mutation operators are the subject of considerable debate in the field of search-base software engineering. It has been claimed both that mutation may not be necessary (Koza 1994) and that mutation alone is sufficient for many problems, as in Evolutionary Programming. On balance, previous work indicates that mutation is important, but insufficient on its own to address complex search spaces (Rothlauf 2011). We propose one new approach from a broad array of possibilities; on balance, there is clearly significant room to explore mutation in GP for program repair. This is further corroborated by other work more strictly in the space of program repair research, much of which conceptually proposes new template-based mutation operators, informed by, e.g., previous human repairs (Kim et al. 2013; Long and Rinard 2015).

The relative performance of crossover and mutation depends on both the problem and the details of the genetic programming system (Luke and Spector 1997), and although analyzing operators independently is key, it is equally important to analyze their effects on one another. This motivates future study to better understand the benefits and interactive effects of our subpatch representation and operators in the area of search based software repair and genetic improvement more generally.

There exists a considerable body of work in program repair and program improvement, which explicitly use both search-based software engineering strategies (e.g., Kim et al. (2013), Le et al. (2016), Petke et al. (2014), Barr et al. (2015), Weimer et al. (2013), and Debroy and Wong (2010)) as well as many others (e.g., Long and Rinard (2016), Long

and Rinard (2015), Nguyen et al. (2013), Mechtaev et al. (2016), Martinez and Monperrus (2015), and Ke et al. (2015)). We do not propose fundamentally new repair algorithms in this work, and instead focus on the operators that underlie a particular type of search strategy for program repair or other improvements. We anticipate that these advances will be particularly beneficial in GP-based program repair and GI strategies (Silva and Esparcia-Alcázar 2015).

6 Conclusion

We have proposed a new subpatch representation and associated operators to enable better exploration and recombination for search-based program improvement. We further presented a novel individual memory process that effectively repairs broken individuals produced by possibly-destructive operators. Our results are promising: Our best new crossover operator, UNIF1SPACE with memory, demonstrated an increase of 245% in successful rate over the baseline. Several other proposed operators demonstrated positive results as well. e.g., OP1SPACE without memory outperformed the original crossover in terms of success rate by 141%. Our results on the new Subspace Mutation operator further substantiated its potential, supporting the utility of a low-level perturbation strategy for program improvement. Finally, we used a genotypical distance analysis to characterize elements of various operators that contributed to relevant and desirable properties of the search landscape.

These results overall suggest that it may be possible to achieve both the scalability benefits of the patch representation for program improvement as well as more effective recombination over the evolutionary computation. Finally, as is consistent with other work, we see that selected evolutionary operators and parameters can have a tremendous impact on search results, motivating future work on such novel evolutionary operators and associated parameters for this and other software engineering search problems.

Acknowledgements Acknowledgements to be added to support a camera-ready.

References

Ackling T, Alexander B, Grunert I (2011) Evolving patches for software repair. In: Genetic and Evolutionary Computation, pp 1427–1434

Arcuri A (2011) Evolutionary repair of faulty software. Appl Soft Comput 11(4):3494–3514

Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 1–10

Arcuri A, Yao X (2008) A novel co-evolutionary approach to automatic software bug fixing. In: 2008 IEEE Congress on Evolutionary Computation. CEC 2008. (IEEE World Congress on Computational Intelligence), IEEE, pp 162-168

Barr ET, Brun Y, Devanbu P, Harman M, Sarro F (2014) The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp 306–317

Barr ET, Harman M, Jia Y, Marginean A, Petke J (2015) Automated software transplantation. In: International Symposium on Software Testing and Analysis (ISSTA), pp 257–269

Brameier MF, Banzhaf W (2007) Linear genetic programming, 1st edn. Springer, Berlin

Britton T, Jeng L, Carver G, Cheak P, Katzenellenbogen T (2013) Reversible debugging software. Tech rep., University of Cambridge, Judge Business School

Bruce BR, Petke J, Harman M (2015) Reducing energy consumption using genetic improvement. In: Annual Conference on Genetic and Evolutionary Computation (GECCO), pp 1327–1334

Burlacu B, Affenzeller M, Winkler S, Kommenda M, Kronberger G (2015) Methods for genealogy and building block analysis in genetic programming. In: Computational Intelligence and Efficiency in Engineering Systems. Springer, pp 61–74

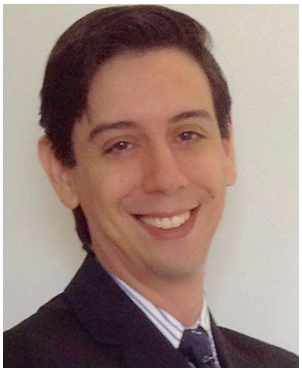
Debroy V, Wong WE (2010) Using mutation to automatically suggest fixes for faulty programs. In: International Conference on Software Testing, Verification, and Validation, pp 65–74
DeJong K (1975) An analysis of the behavior of a class of genetic adaptive systems. Ph D Thesis, University of Michigan
Fast E, Le Goues C, Forrest S, Weimer W (2010) Designing better fitness functions for automated program repair. In: Pelikan M, Branke J (eds) Genetic and Evolutionary Computation Conference (GECCO). ACM, pp 965–972
Forrest S (1993) Genetic algorithms: principles of natural selection applied to computation. Science 261:872–878
Forrest S, Nguyen T, Weimer W, Le Goues C (2009) A genetic programming approach to automated software repair. In: Genetic and evolutionary computation conference (GECCO), pp 947–954
Freitas E, Camilo CG Jr, Vincenzi AMR (2016) SCOUT: a multi-objective method to select components in designing unit testing. In: IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), pp 36–46
Goldberg DE (1989) Genetic algorithms in search, optimization and machine learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc, Reading
Gupta D, Ghafir S (2012) An overview of methods maintaining diversity in genetic algorithms. Int J Emerg Technol Adv Eng 2(5):56–60
Harik GR, Lobo FG, Goldberg DE (1999) The compact genetic algorithm. IEEE Trans Evol Comput 3(4):287–297
Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: trends, techniques and applications. ACM Comput Surv 45(1):11:1–11:61. https://doi.org/10.1145/2379776.2379787
Holland JH (1992) Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence. MIT Press, Cambridge
Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: International conference on software engineering (ICSE), Orlando, FL, USA, pp 467–477. https://doi.org/10.1145/581339.581397
Ke Y, Stolee KT, Le Goues C, Brun Y (2015) Repairing programs with semantic code search. In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 295–306
Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: ACM/IEEE International Conference on Software Engineering (ICSE), San Francisco, CA, USA, pp 802–811
Kim YH, Moon BR (2004) Distance measures in genetic algorithms. In: Genetic and evolutionary computation conference (GECCO). Springer, pp 400–401
Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge
Koza JR (1994) Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge
Langdon WB, Harman M (2015) Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In: Genetic and Evolutionary Computation Conference, GECCO Companion '15, pp 805–810
Le XBD, Lo D, Le Goues C (2016) History driven program repair. In: IEEE 23Rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol 1. IEEE, pp 213–224
Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012a) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering (ICSE), pp 3–13
Le Goues C, Nguyen T, Forrest S, Weimer W (2012b) Genprog: a generic method for automatic software repair. IEEE Transactions on Software Engineering (TSE) 38:54–72. https://doi.org/10.1109/TSE.2011.104
Le Goues C, Weimer W, Forrest S (2012) Representations and operators for improving evolutionary software repair. In: Genetic and evolutionary computation conference (GECCO), pp 959–966
Le Goues C, Forrest S, Weimer W (2013) Current challenges in automatic software repair. Softw Qual J 21(3):421–443. https://doi.org/10.1007/s11219-013-9208-0
Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Transactions on Software Engineering (TSE)
Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI (2005) Scalable statistical bug isolation. SIGPLAN Not 40(6):15–26. https://doi.org/10.1145/1064978.1065014
Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, ESEC/FSE 2015, pp 166–178
Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Principles of Programming Languages, POPL '16, pp 298–312

Louis SJ, Rawlins GJE (1992) Syntactic analysis of convergence in genetic algorithms. In: Foundations of Genetic Algorithms 2, Morgan Kaufmann, pp 141–151
Luke S, Spector L (1997) A comparison of crossover and mutation in genetic programming. Genet Program 97:240–248
Machado BN, Camilo CG Jr, Rodrigues CL, Quijano EHD (2016) Sbstframe: a framework to search-based software testing. In: 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp 004,106–004,111
Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng 20(1):176–205
Mattfeld DC (2013) Evolutionary search and the job shop: Investigations on genetic algorithms for production scheduling. Springer, Berlin
Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering, ICSE '16, pp 691–701
Moncao ACBL, Camilo CG, Queiroz LT, Rodrigues CL, de Sa Leitao P, Vincenzi AMR (2013) Shrinking a database to perform SQL mutation tests using an evolutionary algorithm. In: IEEE Congress on Evolutionary Computation (CEC), pp 2533–2539
Morrison RW, De Jong KA (2001) Measurement of population diversity. In: International Conference on Artificial Evolution (Evolution Artificielle). Springer, pp 31–41
Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) SemFix: program repair via semantic analysis. In: International Conference on Software Engineering (ICSE), pp 772–781
Oliveira AAL, Camilo CG Jr, Vincenzi AMR (2013) A coevolutionary algorithm to automatic test case selection and mutant in mutation testing. In: IEEE Congress on Evolutionary Computation (CEC), pp 829–836
Oliveira VPL, Souza EF, Le Goues C, Camilo CG Jr (2016) Improved crossover operators for genetic programming for program repair. In: International Symposium on Search Based Software Engineering (SSBSE). Springer, pp 112–127
Orlov M, Sipper M (2011) Flight of the FINCH through the Java wilderness. IEEE Trans Evol Comput 15(2):166–182
Petke J, Harman M, Langdon WB, Weimer W (2014) Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Genetic Programming, pp 137–149
Pressman RS (2001) Software engineering: a practitioner's approach, 5th edn. McGraw-Hill Higher Education, Burr Ridge
Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: International Conference on Software Engineering (ICSE), pp 254–265
Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: International Symposium on Software Testing and Analysis (ISSTA), pp 24–36
Rawlins GJE (1991) Foundations of genetic algorithms. Morgan Kaufmann, San Francisco
Rothlauf F (2011) Design of modern heuristics: principles and application. Springer, Berlin
Saha D, Nanda MG, Dhoolia P, Nandivada VK, Sinha V, Chandra S (2011) Fault localization for data-centric programs. In: Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 157–167
Schulte E, Forrest S, Weimer W (2010) Automated program repair through the evolution of assembly code. In: Automated software engineering (ASE), pp 313–316
Schulte E, Dorn J, Harding S, Forrest S, Weimer W (2014) Post-compiler software optimization for reducing energy. SIGARCH Comput Archit News 42(1):639–652
Silva S, Esparcia-Alcázar AI (eds.) (2015) Genetic and evolutionary computation conference companion material proceedings, Workshop on Genetic Improvement, ACM
Smith EK, Barr E, Le Goues C, Brun Y (2015) Is the cure worse than the disease? Overfitting in automated program repair. In: Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 532–543
Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: International Conference on Software Engineering (ICSE), pp 364–374
Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: models and first results. In: Automated Software Engineering (ASE), pp 356–366
Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Transactions on Software Engineering (TSE) 42(8):707–740
Zeller A (1999) Yesterday, my program worked. Today, it does not. Why? In: Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 253–267

Vinícius Paulo L. Oliveira received the BA degree in computer engineering from Federal University of Goiás, the MSc degree in computer science. He is member of the robotics research group Pequi Mecânico, where he works with robotic agent behaviors. He is interested in Artificial Intelligence applyed in robotics, games, human-machine interaction and software problems. More information is available at: www.linkedin.com/in/viniciusp-oliveira.

Eduardo Faria de Souza received the BA degree in Computer Science from Universidade Federal de Goiás, Brazil. He is a MS student at the same university. He develops research in automated program repair and computational intelligence.

Claire Le Goues received the BA degree in computer science from Harvard University and the MS and PhD degrees from the University of Virginia. She is an assistant professor in the School of Computer Science at Carnegie Mellon University, where she is primarily affiliated with the Institute for Software Research. She is interested in how to construct high-quality systems in the face of continuous software evolution, with a particular interest in automatic error repair. More information is available at: http://www.cs.cmu.edu/~clegoues.



Celso G. Camilo-Junior received the BA degree in computer science from Pontifical Catholic University of Goiás, the MSc degree in computer engineering from the Federal University of Goiás and PhD degree in computer engineering from the Federal University of Uberlandia. He is an associate professor in the Informatics Institute at Federal University of Goiás. He is interested in how the Artificial Intelligence can improve the solutions to the real problems in Software, Management and Healthcare domains. More information is available at: http://www.inf.ufg.br/~celso.