Improved Crossover Operators for Genetic Programming for Program Repair

Vinicius Paulo L. Oliveira[1], Eduardo F.D. Souza[1], Claire Le Goues[2], and Celso G. Camilo-Junior[1(✉)]

[1] Instituto de Informatica - Universidade Federal de Goias (UFG), Goiania, GO, Brazil
{viniciusdeoliveira,eduardosouza,celso}@inf.ufg.br
[2] School of Computer Science, Carnegie Mellon University (CMU), Pittsburgh, USA
clegoues@cs.cmu.edu

Abstract. GenProg is a stochastic method based on genetic programming that presents promising results in automatic software repair via patch evolution. GenProg's crossover operates on a patch representation composed of high-granularity edits that indivisibly comprise an edit operation, a faulty location, and a fix statement used in replacement or insertions. Recombination of such high-level minimal units limits the technique's ability to effectively traverse and recombine the repair search spaces. In this work, we propose a reformulation of program repair operators such that they explicitly traverse three subspaces that underlie the search problem: Operator, Fault Space and Fix Space. We leverage this reformulation in the form of new crossover operators that faithfully respect this subspace division, improving search performance. Our experiments on 43 programs validate our insight, and show that the Unif1Space without memorization performed best, improving the fix rate by 34 %.

Keywords: Automatic software repair · Automated program repair · Evolutionary computation · Crossover operator

1 Introduction

Software maintenance is expensive, usually substantially more so than initial development. Maintenance has been estimated to dominate the life cycle cost of software, consuming up to 70 % of those costs [22]. One class of techniques proposed to help mitigate these costs draws on search-based software engineering by applying meta-heuristic search techniques like Genetic Programming [11] to *evolve* program repairs, to improve or mitigate the cost of the bug fixing process [5,20]. The goal is to explore the solution space of potential program improvements, seeking modifications to the input program that, e.g., fix a bug without reducing other functionality, as revealed by test cases.

An important research innovation in this space represents candidate solutions as small *edit programs*, or *patches* to the original program. This is by contrast

F. Sarro and K. Deb (Eds.): SSBSE 2016, LNCS 9962, pp. 112–127, 2016.
DOI: 10.1007/978-3-319-47106-8_8

to earlier work, which adapted more traditional tree-based program representations for repaired program variants [29]. The patch-based representation has significant benefits to both scalability and expressive power in the bug repair domain [17]. It is now commonly used across the domain of Genetic Improvement, a field which treats the program itself as genetic material and attempts to improve it with respect to a variety of functional and quality concerns [26].

Our core contention is that the current formulation of the patch representation overconstrains the search space by conflating its constituent subspaces, resulting in a more difficult to traverse landscape. Consider GenProg [16,29], a well-known program repair method that uses a customized Genetic Programming heuristic to explore the solution space of possible bug fixes represented as patches. The genome consists of a variable-length sequence of tree-based edits to be made to the original program code, with the edits themselves constituting the genes. Each edit takes the following form: `Operation`(*Fault*,*Fix*). `Operation` is the selected edit operator (one of *insert*, *delete*, or *replace*); *Fault* represents the modification point for the edit; and *Fix* captures the statement that will be inserted whenever necessary, such as when `Operation` is a replacement or insertion. That is, each edit contains information along the three subspaces underlying the program repair problem (operator, fault, and fix) [13].

This high gene granularity is considered important to scalability. However, this high granularity for the purposes of crossover limits the search ability to identify, recombine, and propagate the small, low-order building blocks that form the core of a healthy fitness landscape for the purposes of evolutionary computation [10]. Crossover cannot combine partial templates or schema of information along a single subspace, or even two of the three, because the edits themselves are indivisible. We speculate that this is one (though certainly not the only) reason that existing evolutionary program improvement techniques are historically poor at finding multi-edit patches [24].

We therefore propose a novel representation for patch-based evolutionary program improvement, particularly for crossover, to affect a smaller-granularity representation without substantial scalability loss. We instantiate this approach in the GenProg technique for automatic defect repair. Our overall hypothesis is that this new representation and associated crossover operators enable the productive traversal and recombination of information across the actual subspaces of the program improvement problem, and thus can improve performance.

Thus, the main contributions of this paper are:

- An explicit consideration of the implications of schema theory on genetic programming for program repair.
- A new representation to use specifically for crossover that provides a traversal and recombination between repair subspaces.
- Six new crossover operators that more effectively explore the search space.
- Experiments demonstrating improvement in fix effectiveness.

The remainder of this paper is organized as follows. Section 2 presents background on genetic programming, and GenProg in particular; Sect. 3 describes our

new representation and operators; Sect. 4 presents experimental setup, results, and discussion. Section 5 discusses related work; we conclude in Sect. 6.

2 Background

Search-based program improvement leverages metaheuristic search strategies, like genetic programming, to automatically evolve new programs or patches to improve an input program.[1] These improvements can be either functional (e.g., bug fixing [13], feature grafting [12]) or quality-oriented (e.g., energy usage [25]). We focus on automatic program repair, GenProg in particular, but anticipate that our innovations for patch representation should naturally generalize. In this section, we provide background on Genetic Programming in general (Sect. 2.1) and its instantiation for repair in GenProg (Sect. 2.2).

2.1 Genetic Programming

Genetic Programming (GP) is a computational method inspired by biological evolution that evolves computer programs. GP maintains a population of program variants, each of which corresponds to a candidate solution to the problem at hand. Each individual in a GP population is evaluated for its *fitness* with respect to a given fitness function, and the individuals with the highest fitness are more likely to be selected to subsequent generations. Domain-specific *mutation* and *crossover* operators modify intermediate variants and recombine partial solutions to produce new candidate solutions, akin to biological DNA mutation and recombination.

In the context of the Evolutionary Algorithms (EA), a *schema* is a template that identifies a subset of strings (in a GA) or trees (in a GP) with similarities at certain positions (gene) [8]. 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 [10], says that short, low-order schemata with above-average fitness increase exponentially in successive generations. The schema theorem informs the *building block hypothesis*, namely that a genetic algorithm seeks optimal performance through the juxtaposition of such short, low-order, high-performance schemata, called *building blocks*. Ideally, crossover combines such schemata into increasingly fit candidate solutions; this is a feature of a healthy adaptive GP algorithm.

2.2 GenProg for Program Repair

GenProg overview. GenProg is a program repair technique predicated on Genetic Programming. 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

[1] We restrict attention to background necessary to understand our contribution; We discuss related work more fully in Sect. 5.

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 [18,19,23]. Although test cases only provide partial specifications of desired behavior, they are commonly available and provide efficient mechanisms for constraining the space and assessing variants. Experimental results demonstrate that GenProg can be scalable and cost-effective for defects in large, real-world open-source software projects [13]. 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.

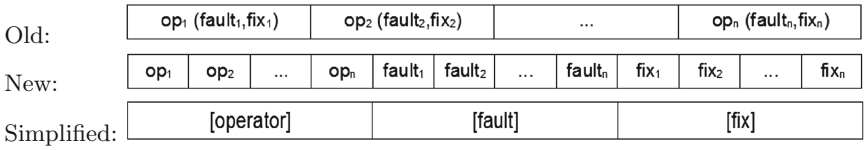

Fig. 1. Old representation (top); New representation (middle); and simplified (bottom).

Search space. The program repair search problem can be formulated along three subspaces: the `Operation`, or the possible modifications that can be applied; the *fault location(s)*, or the set of possibly-faulty locations where the modifications shall be applied; and the *fix code*, or the space of code that can be inserted into the faulty location [14,28]. GenProg constrains this infinite space in several ways: (1) it uses the input test cases to *localize* the defect to a smaller, weighted program slice, (2) it uses coarse-grained perturbation operators at the C statement level (*insert*, *replace*, and *delete*), and (3) it restricts fix code to code within the same program or module, leveraging the *competent programmer hypothesis* while substantially reducing the space of possible fix code.

Representation and mutation. GenProg's patch representation (Fig. 1, top) is composed of a variable-length sequence of high-granularity edit operations. Each edit takes the form: `Operation`(*Fault*,*Fix*), where `Operation` is the edit operator; *Fault* is the modification location; and *Fix* captures the statement that will be inserted when `Operation` is a replacement or insertion. The mutation operation consists of appending a new such edit operation, constructed pseudo-randomly, to the existing (possibly empty) list of edits that describe a given variant.

Crossover. Crossover combines partial solutions and can improve the exploitation of existing solutions and implicit genetic memory. It takes two *parent* individuals from the population to produce two *offspring* individuals. GenProg uses a one-point crossover over the edits composing each of the parents. It selects a random cut point in each individual and then swaps the tails of each list to produce two new offspring that each contain edit operations from each parent. This does not create new edits; this power is currently reserved for mutation. Our illustrative example (Sect. 3.1) indicates the ways that this representation limits the recombination potential offered by crossover (Fig. 2).

3 Approach

Our high-level goal is to enable efficient recombination of genetic information while maintaining the scalability and efficiency of the modern patch representation. The building block hypothesis states, intuitively, that crossover should be able to recombine small schemata into large 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 of past 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 combines information across all three subspaces, and edits are indivisible for the purposes of crossover. 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.

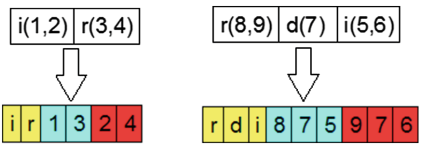

Fig. 2. Example of mapping an individual to the new representation. Each subspace is represented by a color: Yellow is the Operator subspace, blue is the Fault subspace and red is the Fix subspace. The character i = Insert, r = Replace, d = Delete. (Color figure online)

We propose to explicitly conceive of the schemata in this domain as a template of edit operations, where certain operations and their order is necessary to represent key individual information. We instantiate this conception in a new intermediate representation and then new crossover operators that leverage it. We begin with a running example that we will use to illustrate the approach (Sect. 3.1). We propose a new representation and a mapping to it from the existing patch representation (Sect. 3.2). We then present six new crossover operators (Sect. 3.3: OP1Space, Unif1Space, and OPAllS, and then each of these new operators with memorization.

3.1 Illustrative Example

Consider a bug that requires two edits to be repaired:[2] `Insert`(*1,9*) `Delete`(*3,*). Consider also two candidate patches that contain all the genetic material necessary for this repair: (A) `Insert`(*1,2*)`Replace`(*3,4*) and (B) `Replace`(*8,9*) `Delete`(*3,*)`Insert`(*5,6*). The deletion in candidate (B) is correct as is and only

[2] We use integer indices to denote numbered statements taken from a pool of potential faulty locations and candidate fix code, as is standard.

needs to be combined with the appropriate insertion. The current crossover operator can propagate this deletion into subsequent generations.

However, constructing the `Insert`(*1*,*9*) cannot be accomplished through crossover alone, even though the insertion in candidate (A) is only one modification from the solution along the fix space, and (B) contains the correct code in its first replacement. Crossover cannot change the Fix element in (A) from 2 to 9, because the gene is treated as an indivisible unit. The only way to achieve the desired solution is via a combination of edits that compose semantically to the desired solution, or by relying on mutation to produce the insertion from whole-cloth.

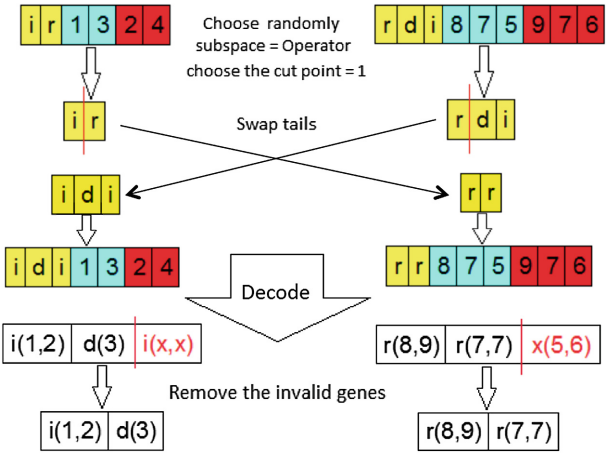

Fig. 3. Example of OP1Space applied to a pair of variants.

3.2 Decoupled Representation

We begin by decoupling the three subspaces in the representation to decrease edit granularity. We map variants to a new representation that imposes independence of subspaces, shown in the middle of Fig. 1. This decoupled representation has fixed positions to improve genetic memory. To simplify presentation, this representation can be further reduced to a one dimensional array by concatenating the three subspace arrays, shown in the bottom of Fig. 1. To simplify subsequent crossover operations while maintaining variant integrity, we add to the `Delete` operator a ghost *Fix* value, equal to its *Fault* value.

Note that we maintain the original patch representation for non-crossover steps because it is beneficial for mutation and because doing so allows us to focus our study on the effects of crossover specifically. A mapping transformation (*encode*) is thus applied to each individual immediately before crossover, which is applied to pairs of individuals selected in the standard way. We then apply a *decode* transformation to the offspring to return them to the canonical

representation for selection and mutation. As will be shown, *Decode* can cause a loss of information. We therefore propose a memorization system to repair broken individuals, which we discuss subsequently.

3.3 New Crossover Operators

We propose six crossover operators to leverage and analyze the proposed representation in search-based program repair. OP1Space and Unif1Space apply to a single subspace, while OPAllS applies to the whole chromosome. These three operators augmented with memorization mechanism result in six total proposed operators.

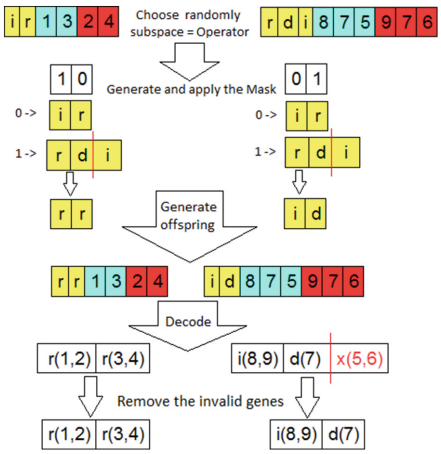

Fig. 4. Example of Unif1Space applied in the Operator subspace.

One Point Crossover on a Single Subspace (OP1Space). OP1Space applies one-point crossover to a single subspace (see Fig. 3 for a visual presentation). It therefore explores new solutions in a single neighborhood, while maintaining potentially important blocks of information in the other subspaces. Given two parent variants *encoded* into the new representation, OP1Space chooses one of the three subspaces uniformly at random, and then randomly selects a cut point. Because the patch representation is of variable length, this number must be bounded by the minimum length of the chosen subspace so as to result in a valid point in both parents. We swap the tails beyond this cut point between parents, generating two offspring. The portion of the individuals relative to the unselected subspaces are unchanged.

Finally, *decode* is applied. *Decode* to unchanged parents is simply the inverse of *encode*. However, this crossover operator can *break* edit operations in offspring when the parents are of different lengths, resulting in either excess or missing data in the unchanged subspaces (e.g., an insert operation without a

corresponding fix statement ID; Fig. 3 provides an example). For this operator, decode simply drops invalid genes.

Uniform Single Subspace (Unif1Space). A *uniform* crossover operator combines a uniform blend of data from each parent [4], promoting greater exploration. However, in certain domains, a uniform operator be problematically destructive [17]. We thus propose a uniform operator along a single subspace, promoting a constrained exploration. As with OP1Space, Unif1Space selects a subspace at random. It then generates a random binary mask of length equal to the smaller of the two subspaces chosen. Genes are swapped between parents to create offspring according to this mask. As with OP1Space, invalid genes are dropped in decode. Figure 4 shows the behavior of this operator on the running example. It can create highly diverse offspring, but may also dissolve many basic blocks.

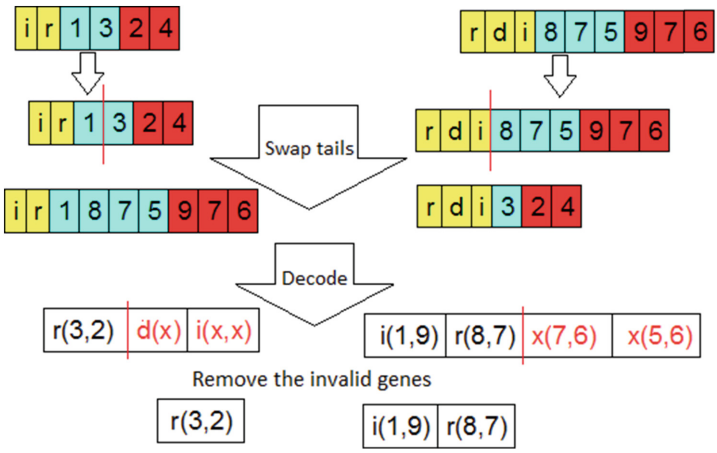

Fig. 5. Example of OPAllS.

One Point Across All Subspaces (OPAllS). OPAllS follows the same rules for cut point selection as OP1Space, but without the restriction to a single subspace. The crossover point is based on the length of the entire individual. It swaps large parts of the entire individual, simultaneously mixing subspaces. This operator can thus maintain larger basic blocks than Unif1Space, with a greater capacity for information exchange than OP1Space. Large blocks containing valuable information within at least one subspace cannot be dissolved, so this operator prevents the destruction of some good information in such subspaces. However, it can completely change certain operations by affecting an entire subspace. For example, this crossover can keep all the original values for "Operator", but, unlike OP1Space, will still change all the Fix values and part of the Fault values. We illustrate with the running example in Fig. 5.

Memorization. As the previous discussion demonstrated, crossover on the decoupled representation presents a possibility of data loss. We therefore propose a

memorization scheme to help reconstruct valid from invalid genes. Memorization maintains, for each individual, a cache of pieces of genes unused after crossover operations, distinguishing between the Operator, Fix and Fault spaces. It then tries to use values from this cache to fix broken genes on demand. This cache is maintained throughout the evolution process.

For example, in Fig. 3, the operation "i" in the first offspring and the fault and fix values 5 and 6 in the second would be stored in the cache for use in subsequent generations. Assuming the existence of data from previous variants in the cache, the memorization algorithm will try to find a Fault and Fix value to repair the first offspring, and an Operator to repair the second. If such values are available, they will be selected between at random, removed from the cache, and inserted into the associated individual.

Thus, we propose the three previously-described crossover operators, as well as the same three operators implemented with memorization, hypothesizing that memorization may decrease data loss and increase the number of solutions through the evolution process.

4 Experiments

In this section, we present experiments that compare the proposed crossover operators to the canonical one-point patch representation crossover operator. We hypothesize that the proposed crossover operators can increase the fix rate.

4.1 Setup

Table 1. Benchmarks, test cases, and buggy versions of each program.

Program	Tests	Versions
gcd	11	1
zune	24	1
checksum	6	7
digits	6	7
grade	7	7
median	7	7
smallest	7	7
syllables	6	6

Benchmarks. Table 1 shows the C programs we use in our evaluation. `gcd` and `zune` have classically appeared in previous assessments of program repair.[3] Both include infinite loop bugs. The other six program classes are drawn from IntroClass [15],[4] 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. We chose 6–7 random programs for each assignment, for a total of 43 defective programs. We use the higher-quality black box tests provided with the benchmark to assess correctness.

Each program version itself is small, but 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, without a prohibitive computational

[3] Both available from the GenProg project: http://genprog.cs.virginia.edu/.
[4] Available at http://repairbenchmarks.cs.umass.edu/.

time. 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, increasing the risk of low-quality patches. We sidestep this issue by evaluating on small but very well-specified programs (as validated by their designers, manually, and experimentally [27]).

Parameters and metrics. We executed 30 random trials for each program version. 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. The parameters used for all runs are: Elitism = 3, Generations = 20, Population size = 15, Crossover rate = 0.5, Mutation rate = 1, Tournament k = 2. 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 Results

Table 2 presents the success rate of experiments for all operators and problems (higher is better). Table 3 presents test suite evaluations, or average fitness evaluations, to repair (lower is better). In the latter table, we omit grade and syllables, as no repairs were found in any run.

Table 2. Success rate (percentage) over all runs. We aggregate across IntroClass problems for presentation.

Memorization?	Original	OP1Space		Unif1Space		OPAllS	
	N/A	No	Yes	No	Yes	No	Yes
gcd	0.70	**0.80**	0.63	0.67	0.70	0.73	**0.80**
zune	0,66	0.70	0.97	**1.00**	0.97	0.93	0.93
checksum	0.00	0.00	0.01	**0.03**	0.00	0.00	0.00
digits	0.27	0.25	0.31	**0.29**	0.27	0.26	0.28
grade	0.00	0.00	0.00	0.00	0.00	0.00	0.00
median	0.28	**0.50**	0.48	0.48	0.49	**0.50**	0.49
smallest	0.51	0.51	0.58	**0.64**	0.57	**0.64**	0.60
syllables	0.02	0.16	0.16	0.16	0.16	0.16	0.16
Average	0.305	0.365	0.392	**0.408**	0.395	0.402	0.407

Success rate. Unif1Space without memorization presents the best success rate, as can be seen in Table 2. Overall, the Unif1Space was the best operator, producing a 34 % improvement the fix rate over the Original baseline. A Wilcoxon rank-sum test, at $\alpha = 0.05$, establishes that the observed difference in performance between all operators without memorization and the Original crossover

are statistically significant. A Vargha-Delaney test supports the observation that all operators outperformed the Original operator, with effect sizes between 0.532 and 0.564, indicating a small but observable effect size. The effect size is greatest for Unif1Space, as compared to the Original baseline.

Although Unif1Space produced consistently strong results, it is not the best across all problems. At a per-problem level, for the `checksum` problem, Unif1Space without memorization is best, but in general, `checksum` appears to be difficult for all operator. We speculate that this is because most of `checksum` defects require a specific modification that is difficult to produce with the current operators in short programs. On `digits` programs, OP1Space with memorization was the best, followed by Unif1Space without memorization. In the `gcd` problem, all operators produced a high fix rate, but OP1Space without memorization and OPAllS with memorization were best. In the `median` problem, OP1Space and OPAllS without memorization were best, but all proposed operators are comparable. For `zune`, Unif1Space without memorization achieved the maximum fix rate; the other proposed operators were all still better than the original baseline. Finally, for `syllables`, all proposed operators reached the same results and outperformed the original.

Table 3. Test suite evaluations to repair. We aggregate across IntroClass problems for presentation. We omit grade because no repairs were found. N/A is used when no repair was found.

Memorization?	Original	OP1Space		Unif1Space		OPAllS	
	N/A	No	Yes	No	Yes	No	Yes
gcd	10.88	**5.04**	6.20	10.94	7.77	5.91	7.47
zune	3.90	**3.00**	3.30	3.83	3.87	3.22	3.01
checksum	**27.22**	69.50	35.67	47.00	N/A	56.25	N/A
digits	14.53	13.17	17.63	14.90	**9.94**	15.63	13.64
median	**16.07**	37.94	41.48	34.38	40.91	33.30	41.78
smallest	**16.40**	37.59	44.77	65.26	48.33	56.73	46.40
syllables	**20.53**	27.30	27.30	28.97	30.90	25.40	27.13
Average	**15.59**	27.64	25.19	29.32	23.62	28.06	19.91

Efficiency. Table 3 presents the average fitness evaluations to repair for each operator. Overall, the operator with the best success rate was not the most efficient. This is consistent with our expectations: the more difficult problems are harder to solve, and thus succeeding in them (having greater success) can pull up the average time to repair [17]. This behavior may also be explained by the fact the operators that focus on a single subspace, OP1Space and Unif1Space, are less destructive in recombining variants, which may lead to a slower search process as compared to Original and OPAllS. However, overall, the 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, the operator OPAllS with memorization presented a success rate almost the same as Unif1Space without memorization, but presented a considerable smaller time to repair, so it may present a desirable cost-benefit tradeoff.

At a per-problem level, the Original crossover operator outperforms the others for `checksum`, but as the success rate is low, high variability is unsurprising. The second best operator here is OP1Space without memorization. For digits, the Unif1Space with memorization was best, followed by OP1Space without memorization. In `gcd` OP1Space without memorization significantly outperformed Original; OPAllS without memorization performed second best. The `zune` presents a low discrepancy within operators, but OP1Space without memorization was the most efficient. The `smallest` and `median` the Original was much better than others. For `syllables`, Original was best, followed by OPAllS without memorization.

Memorization results. Memorization does not appear to increase success rate, as we can see particularly in the best operators according to this metric (Unif1Space followed by OPAllS, both without memorization). We speculate that the loss of incomplete genes in decode can reduce unnecessary modifications that hinder repair performance. One general lesson is that there may be a benefit to mitigating code bloat throughout the program improvement process. However, in aggregate, comparing each operator with memorization to the same operator without, the version with memorization is more efficient, supporting the general potential of the mechanism. This is particularly true of OPAllS, where memorization provided a significant efficiency benefit. This may suggest that memorization is more beneficial for the more destructive operators, allowing them to avoid large losses of genetic material. As a final note, our maximum generation count was relatively low, reducing the potential utility of a genetic memorization mechanism. We expect the memorization approach may perform better in longer runs.

New representation. In general, on average, crossover using our new representation outperformed the standard representation, even when genes are lost in decode. This indicates the new representation *in particular* has important potential to improving the performance of patch-based program improvement heuristic techniques. In terms of scalability, the new representation does not use considerably more memory over the standard representation, and the computational cost of transforming between them was low. Although we do not directly analyze the progression of schema through the search, our results are affirmed by underlying theory suggesting that the representation improves GenProg's ability to construct and propagate building blocks.

4.3 Threats to Validity

On threat to the validity of our results is that they may not generalize, because 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, which allowed for a more focused study of operator effectiveness; we 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 [27]. 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. Further tests and analysis are required to fully explain the operators' behavior, enabling understanding of why any one operator performed better than another.

5 Related Work

Most innovations in the Genetic Programming (GP) space for program improvement involve new kinds of fitness functions or application domains; there has been less emphasis on novel representations and operators, such as those we explore. However, there are exceptions to this general trend. Orlov and Sipper outline a semantics-preserving crossover operator for Java bytecode [21]. Ackling *et al.* propose a patch-based representation to encode Python rewrite rules [1]; Debroy and Wong investigate alternative mutation operators [6]. Forrest *et al.* quantified operator effectiveness, and compared crossback to traditional crossover [7]. Le Goues *et al.* examined several representation, operator and other choices used for evolutionary program repair [17], 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 in particular. These results corroborate Arcuri's [2] demonstrating 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 and demonstrating the influence of crossover operators on algorithmic performance.

Our results demonstrate that in theory our new representation combined with the crossover operators can improve the creation and propagation of the build blocks, but does not directly investigate the role of schema evolution in this phenomenon; we leave this to future work. For example, Burlacu [3] presents a powerful tool for theoretical investigations on evolutionary algorithm behavior concerning building blocks and fitness.

Informed by the building blocks hypothesis, Harik 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 [9]. He concluded that building blocks can be tightly coded and propagated throughout the population through repeated selection and recombination. His 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 in representing a program in repair problem can be one of the reasons for its complexity.

6 Conclusion

Supported by the Schema Theorem and Building Blocks Hypothesis, our primary contribution in this paper is a new low-granularity patch representation and associated crossover operators to enable better parental recombination in a search-based program improvement algorithm. We also presented a novel memorization process that shows a possibility to repair problematic genes, that even not showing results better than without memorization, it can be useful to develop new ways to solve the broken genes problem. Our objective was to improve the algorithm's ability to traverse the fitness landscape, improving success rate. Our results suggest that this targeted approach is promising: our best new crossover operator, Unif1Space without memorization, demonstrated an increase of 34 % in the success rate over the baseline. However, our results also showed that operator success varied across the different program classes studied. The results 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, motivating future work on such novel evolutionary operators and associated parameters.

References

1. Ackling, T., Alexander, B., Grunert, I.: Evolving patches for software repair. In: Genetic and Evolutionary Computation, pp. 1427–1434 (2011)
2. Arcuri, A.: Evolutionary repair of faulty software. Appl. Soft Comput. **11**(4), 3494–3514 (2011)
3. Burlacu, B., Affenzeller, M., Winkler, S., Kommenda, M., Kronberger, G.: Methods for genealogy and building block analysis in genetic programming. In: Borowik, G., Chaczko, Z., Jacak, W., Łuba, T. (eds.) Computational Intelligence and Efficiency in Engineering Systems, Part I. SCI, vol. 595, pp. 61–74. Springer, Heidelberg (2015). doi:10.1007/978-3-319-15720-7_5
4. Chawdhry, P.K., Roy, R., Pant, R.K.: Soft Computing in Engineering Design and Manufacturing. Springer, Heidelberg (2012)
5. de Oliveira, A.A.L., Camilo-Junior, C.G., Vincenzi, A.M.R.: A coevolutionary algorithm to automatic test case selection and mutant in mutation testing. In: Congress on Evolutionary Computation, pp. 829–836 (2013)
6. Debroy, V., Eric Wong, E.: Using mutation to automatically suggest fixes for faulty programs. In: International Conference on Software Testing, Verification, and Validation, pp. 65–74 (2010)
7. Forrest, S., Nguyen, T., Weimer, W., Goues, C.L.: A genetic programming approach to automated software repair. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 947–954 (2009)
8. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Reading (1989)

9. Harik, G.R., Lobo, F.G., Goldberg, D.E.: The compact genetic algorithm. IEEE Trans. Evol. Comput. **3**(4), 287–297 (1999)
10. John, H.: Adaptation in natural and artificial systems (1992)
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
12. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknot-sRG for RNA pseudoknot free energy calculation. In: Genetic and Evolutionary Computation Conference, GECCO Companion 2015, pp. 805–810 (2015)
13. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering, pp. 3–13 (2012)
14. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Softw. Qual. J. **21**(3), 421–443 (2013)
15. Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W.: The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans. Softw. Eng. **41**, 1236–1256 (2015)
16. Le Goues, C., Nguyen, T.V., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. (TSE) **38**, 54–72 (2012)
17. Le Goues, C., Weimer, W., Forrest, S.: Representations and operators for improving evolutionary software repair. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 959–966 (2012)
18. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Principles of Programming Languages, POPL 2016, pp. 298–312 (2016)
19. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering, ICSE 2016, pp. 691–701 (2016)
20. Nunes, B., Quijano, E.H.D., Camilo-Junior, C.G., Rodrigues, C.: SBSTFrame: a framework to search-based software testing. In: International Conference on Systems, Man, and Cybernetics (2016)
21. Orlov, M., Sipper, M.: Flight of the FINCH through the Java wilderness. IEEE Trans. Evol. Comput. **15**(2), 166–182 (2011)
22. Pressman, R.S.: Software Engineering: A Practitioners Approach. Palgrave Macmillan, London (2005)
23. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C.: The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, pp. 254–265. ACM (2014)
24. Qi, Z., Long, F., Achour, S., Rinard, M.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: International Symposium on Software Testing and Analysis, pp. 24–36 (2015)
25. Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. In: Architectural Support for Programming Languages and Operating Systems, pp. 639–652 (2014)
26. Silva, S., Esparcia-Alcázar, A.I. (eds.): Genetic and Evolutionary Computation Conference, GECCO 2015, Companion Material Proceedings, Workshop on Genetic Improvement. ACM (2015)
27. Smith, E.K., Barr, E., Goues, C.L., Brun, Y.: 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 (2015)

28. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging program equivalence for adaptive program repair: models and first results. In: Automated Software Engineering (ASE), pp. 356–366 (2013)
29. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: International Conference on Software Engineering (ICSE), pp. 364–374 (2009)