Digging into Semantics: Where do search-based software repair methods search?

Hammad Ahmad[1], Padriac Cashin[2], Stephanie Forrest[2], and Westley Weimer[1]

[1] University of Michigan, Ann Arbor MI 48109, USA
{hammada,weimerw}@umich.edu
[2] Arizona State University, Tempe AZ 85281, USA
{pcashin,steph}@asu.edu

Abstract. Search-based methods are a popular approach for automatically repairing software bugs, a field known as automated program repair (APR). There is increasing interest in empirical evaluation and comparison of different APR methods, typically measured as the rate of successful repairs on benchmark sets of buggy programs. Such evaluations, however, fail to explain *why* some approaches succeed and others fail. Because these methods typically use syntactic representations, i.e., source code, we know little about how the different methods explore their semantic spaces, which is relevant for assessing repair quality and understanding search dynamics. We propose an automated method based on program semantics, which provides quantitative and qualitative information about different APR search-based techniques. Our approach requires no manual annotation and produces both mathematical and human-understandable insights. In an empirical evaluation of 4 APR tools and 34 defects, we investigate the relationship between search-space exploration, semantic diversity and repair success, examining both the overall picture and how the tools' search unfolds. Our results suggest that population diversity alone is not sufficient for finding repairs, and that searching in the right place is more important than searching broadly, highlighting future directions for the research community.

Keywords: Semantic search spaces · Program repair · Patch diversity.

1 Introduction

Early works on automatically repairing defects in software demonstrated that evolutionary computation (EC) and related *search-based* approaches can be surprisingly successful in this domain [1, 2, 20, 35, 54]. Since then, there has been an explosion of research into what is now called the *automated program repair* (APR) problem. This research has produced a wide variety of techniques and tools aimed at reducing the manual effort required to repair software bugs or otherwise improve software [23, 31]. These tools typically operate on source code containing one or more bugs, or *defects*, together with a test suite that encodes required functionality and at least one test that exposes the defect. Multiple candidate patches are often generated, which both repair the defect and pass the test suite [1, 20, 37, 38]. The field has standardized on a small number of benchmark test suites to compare the performance of different tools and techniques, often by measuring the fraction of successful repairs [12, 21]. However, we still have little insight into fundamental questions such as: *Why* do some algorithms outperform others? *Which* components of an algorithm are most responsible for its success (or failure)? *How different* are the patches produced by different techniques? *What* kinds of bugs is APR better or worse at solving?

Traditional evaluation approaches are not always helpful for these questions. For example, it can be difficult to determine from a pseudocode description of a new repair algorithm whether it will find a more *diverse* set of repairs than existing ones, or which parts of a search space it will visit [26]. Importantly, today's search-based APR methods use a syntactic representation, i.e., source code, even though repairing bugs involves changing semantics.

Earlier research tackled some of these questions by considering the extent to which proposed repairs are overfit to a test suite [19, 30, 34, 43, 46], nonfunctional properties such as repair readability and maintainability [11, 50], and repair diversity [6, 18, 30, 33, 36, 49, 56]. Within the context of diversity, previous studies examined the search space of a single tool to better understand patch construction [14, 55] and compared the search spaces explored by different tools with respect to high-level program characteristics [30, 32, 52]. However, to the best of our knowledge, this previous work considers only program variants that are *test-suite adequate* [38], or *plausible*, meaning that they repair the bug and pass all required tests. This approach ignores how the search process discovers plausible repairs. In this paper, we propose a method for comparing the semantic search spaces of different APR algorithms, and characterize the program variants generated during the search in addition to the end product.

Insight and approach. Generating candidate variants through syntactic program manipulation is central to search-based APR tools, yet their ultimate value depends on inducing meaningful semantic change. We hypothesize that the effective *search spaces* (the sets of candidate program variants considered or potentially constructed) of different APR tools for a given software defect are distinct but not disjoint. We further hypothesize that lightweight analysis of the run-time semantics of each variant generated, regardless of correctness, can shed light on how different APR tools search for repairs. To analyze the effective search space of a particular tool, we propose to embed its generated variants in a semantic *invariant space*, admitting an approximate notion of *similarity.* Because many individual variants generated during a search are syntactically distinct but semantically equivalent [53], we focus on source-level formal invariants. Since test suites are generally available in this domain, we propose leveraging them for efficient dynamic invariant detection [8], rather than resorting to expensive static or manual approaches. Once each individual variant is characterized by its set of detected invariants, we propose to use a form of weighted vector distance (*Canberra distance* [17]) to assess differences. Because most programs have many invariants, our vector distance approach has a significant scalability advantage over other approaches, such as checking logical implications between invariant sets with a theorem prover. Ultimately, our approach allows both mathematical (i.e., via principal component analysis) and human-understandable (i.e., two-dimensional visualization) analysis of search spaces.

Contributions. The main contributions of this work are as follows:

- A framework for comparing the effective semantic search spaces of APR algorithms.
- An automated analysis of individual program variants to produce a two-dimensional visualization of their semantic diversity.
- An empirical analysis on four established search-based APR tools.
- A discussion of the relationship between syntactic and semantic diversity and implications for APR algorithm design.

2 Background and Contextual Motivation

Automated Program Repair. Automated program repair (APR) methods seek to locate and repair software defects without introducing side effects. Typically, this involves modifying the program's source code to produce a patched version. Most methods rely on a test suite to certify the repaired program's correctness.

Over the past decade, many search-based methods for APR have been proposed, with some more recognizable as Genetic Programming (GP) solutions than others (see Monperrus [31] or Le Goues et al. [23] for comprehensive reviews). In this paper, we evaluate on four established tools that represent different search-based APR techniques. GenProg implements a form of GP to search for repairs [22, 54]. CapGen uses the same mutation operators as GenProg, but allows more granular mutations to sub-elements of statements and mines contextual information to select effective mutations [55]. SimFix mines prior patches, both to construct particular mutations and to guide the selection of operations based on code similarity measurements [14]. TBar is a recent approach that uses 35 different "fix patterns", or templates, to modify the buggy program [25]. Over time these tools have incorporated heuristic information about the software-repair domain to what was originally a pure GP-based approach.

Dynamic Invariant Detection. To capture semantics, we use dynamic invariants (i.e., logical predicates that always hold during the execution of the program) to approximate code functionality. Dynamic invariant detection [8] algorithms trace program state during execution to construct such invariants. These traces contain the state of in-scope variables at specific points in execution, usually before and after function calls. Because they do not rely on program source code to construct invariants (cf. static invariant detection), dynamic approaches are modular and scalable to our problem. However, a finite set of dynamic traces may not capture all possible or relevant future executions and can overfit to the observed traces. Because we are interested in how small regions of code (patches) differ from one another, this issue is less of a concern for our task.

Semantic Search Space. Earlier studies have investigated how well different APR methods explore the search space created by their mutation operators. Typically, the search space is defined as the union of the mutants that can be created by applying n mutations to the original program [18,30,36]. For instance, if an APR tool can only insert one statement before another, then its first-order search space consists of all programs that can be constructed by applying that *insert* operator a single time to the original. This approach has been used to characterize the search space by measuring the density of programs with specific characteristics, such as the number of passing tests or the number of correct patches [36]. By contrast, we define the semantic search space of an APR algorithm in terms of the set of reachable program invariants (via any of its generated mutants) when applying its mutation operations to the original program. Since the goal of the APR process is to construct a semantically correct program, understanding what functionality a given algorithm can construct is crucial to understanding its behavior. Similar to the syntactic search space, the semantic search space is effectively infinite, even with simple operators. Rather than enforcing an n-mutation restriction, as the aforementioned approaches do, we rely on the normal operation of each APR tool, unchanged, to define its semantic search space. This allows us to describe the search spaces of APR tools as they apply in practice.

Contextual Motivation: Does diversity lead to more repairs? Some researchers have suggested that higher population diversity (syntactic or semantic) leads to higher repair rates and better repairs [7, 10, 39, 44], and some tools (e.g., Marriagent [16]) favor high-diversity edits. Other results suggest that high semantic diversity does not necessarily improve repair rates [5,6,51]. If the latter is true, it suggests that researchers should focus less on high diversity mutants, and more on other properties of repair algorithms. If exploring widely in the search space predicts high repair rates, we would expect to observe a correlation between how much of the semantic search space is sampled and an ability to discover repairs. Across the board, however, as we perform quantitative and qualitative analyses to investigate the relationship between semantic diversity and repair rates for APR tools, we find little evidence that this is true (see Section 5). This finding challenges the conventional hypothesis that generating diverse mutants is the key to improving repair rates, and supports recent results arguing otherwise.

3 Technical Approach

Even though most of today's search-based APR methods inherit the concept of mutation from evolutionary computation, such tools do not significantly rely on crossover [24, 37, 38, 53]. We thus focus on the mutation operators of APR tools. We begin with a set of mutants for each APR method. These are mutated variants of the original program, which pass all of the positive (regression) tests and may or may not pass the negative (bug inducing) tests. Given a set of mutants, or candidate patches, we next use Daikon [9] (the state-of-the-art for dynamic invariant detection) to generate a set of invariants, one set for each individual variant, regardless of correctness (Section 3.1), and then apply an efficient heuristic to measure semantic similarity between invariant sets (Section 3.2). Since large invariant sets are challenging to interpret and compare, we also present two visualizations of induced APR search spaces (Section 3.3).

3.1 Sampling APR Search Spaces

We aim to reason qualitatively about the search spaces induced by different APR tools and the techniques they employ (e.g., genetic operator-based vs. template-based mutation). Schulte et al. have previously treated the syntactic representation of each variant generated by an APR tool as a sample of the tool's search space [42]. We hypothesize, however, semantic diversity may be a more relevant consideration for understanding tool effectiveness. Our approach is motivated in part by the fact that syntactic variants often leave functionality unchanged (neutral) [41, 42, 53]. Ultimately, an APR tool's utility relates to its ability to find new functionality that addresses the defect.

We sample the semantic search space in two ways. First, we consider the early phase of a search by selecting the first x variants generated by each tool, reflecting real-world scenarios with scarce computational resources. Our second sampling method provides a broader picture. Some tools might initially search less widely, but focus in later. Thus, we evaluate y mutants selected uniformly at random after each tool completes its search.

We next consider how to capture the behavior of a mutant. Since our benchmarks total 357,000 lines of code and have over 20,000 test cases [15], static analysis methods will not scale for our experiments. Instead, we use dynamic analysis and restrict attention to a subset of the test cases. Because we are interested in repairing bugs, we assume that the greatest variation in mutant functionality will be along faulty execution paths, represented by the failing test. Intuitively, since repair algorithms aim to retain required functionality, they are much more likely to agree semantically on regression (positive) tests. We thus collect only traces associated with negative tests, one set for each distinct mutant. The set of invariants represents the most relevant program behavior. To compare variants, we then compute the difference between each pair of invariant sets across all tools in our study using a computational shortcut, which is surprisingly effective.

3.2 Computing Mutant Similarity

Earlier work defined a metric for computing semantic distance between two programs, based on logical implication between their sets of invariants [4]. This metric reflects the content of individual invariants, and as such quantifies difference precisely. Unfortunately, implementations of this approach have $\mathcal{O}(n!)$ time complexity in the worst case. Invariant detectors (e.g., Daikon) often report thousands of invariants for a single complex program. Thus, implication-based distance approaches are too expensive for use in our setting.

Instead, we use an efficient approximation of the semantic distance between two mutants. By treating invariant sets as bit vectors (one dimension for each invariant), we can compute the Canberra distance [17], a numerical measure of the distance between pairs of points in a vector space, between two invariant sets. To do this, we define a canonical ordering of the union of all invariants found across all mutants, and then associate one bit vector with each mutant, where the nth bit is set if and only if the nth invariant was detected for that program. We then compute the Canberra distance between the bit vectors, and use these distances to embed each mutant in an implied semantic vector space. In our setting, candidate patches are mostly identical except for a small number of mutations, and thus, Canberra distance provides a scalable approach that captures invariant differences between programs effectively.

3.3 Visualizing Search Spaces

Simply presenting a raw set of invariants, or even a string difference between two sets of invariants, is not informative to humans [45]. As such, for each defect, we compute the pairwise distance between invariant sets for every mutant, producing one number per pair, regardless of the APR tool that generated it. We use this information to visualize the semantic subspaces generated by each tool by embedding it in a single two-dimensional plot. Since our metric is relative (i.e., we compute the relative distance between the inferred invariant sets for two mutants), we anchor the measurements to two key points: the invariant set for the original defect, and the invariant set for the human-generated repair. Once the distance measurements are computed, our vector distance metric embeds mutants into a human-friendly two-dimensional visualization.

To complement the distance information, we also consider the number of unique semantic invariants introduced by each new mutant. For each tool, we examine the number of new unique invariants inferred for each mutant produced and evaluated. While the 2D embeddings show where each tool is sampling in semantic space, the rate at which unique invariants accumulate shows how much time the tool spends generating mutants with new semantics (and thus new functionality) compared to rediscovering old functionality with new syntax.

These two visualizations decompose our analysis into a spatial and temporal component, both of which are key to understanding the APR search for solutions.

4 Experimental Setup

We now describe our experimental setup for comparing the search spaces of various APR tools. We also make our replication materials publicly available.

Candidate Patch Collection. We gather candidate variants (mutants) from four established tools: CapGen [55], GenProg [22], SimFix [14], and TBar [25]. All four tools use search-based techniques, but each tool uses different mutation operators and search methods. We ran each tool on 34 representative Java defects from Defects4J [15] that all of the tools we consider operate on (see Table 1).

Table 1: Experimental Benchmarks: 34 Java defects selected from Defects4J. ✓ means that the tool produced a repair. Defects not repaired by any tool (omitted for space) comprise Math 7, 9, 12, 16, 17, 18, 19.

	Chart			Lang				Math																			
	8	11	24	6	26	57	59	1	2	3	4	5	8	11	15	20	30	33	53	57	59	63	65	70	75	80	85
CapGen	✓	✓	✓	✓	✓	✓	✓					✓					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GenProg									✓			✓	✓											✓		✓	✓
SimFix								✓				✓	✓			✓		✓	✓	✓	✓	✓		✓	✓	✓	✓
TBar	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓

CapGen reports each generated variant in numeric order, regardless of its correctness. We instrumented the other tools to collect similar information. We note that GenProg caches fitness to increase efficiency, so we record only the variants that would be independently evaluated against the test suite, ignoring duplicates. For all tools, we timestamp and store each variant that is evaluated against the test suite to record how the search proceeds. These modifications account for fewer than 20 lines of code and do not affect search logic.

Invariant Detection. For each program variant in our dataset, we apply the mutations to a clean instance of the Defects4J bug and record a trace of a driver program. Each driver is a small Java program that executes the failing test cases for the mutant. A *trace* is a series of program state observations used to infer program semantics. For each such trace, we then use Daikon to obtain a set of invariants, representing the pre- and post-conditions of executed functions.

We use the invariant sets of the first $x = 600$ mutants generated by each tool to construct a view of the early stages of its search process. We find that the number of semantically unique invariants tapers off at around 300 mutants (Figure 2a, Section 5.1), so we conservatively chose 600 as our cutoff point. We also sample $y = 1000$ mutants uniformly at random from all generated variants (per tool) to provide an overview of the space searched by each tool.

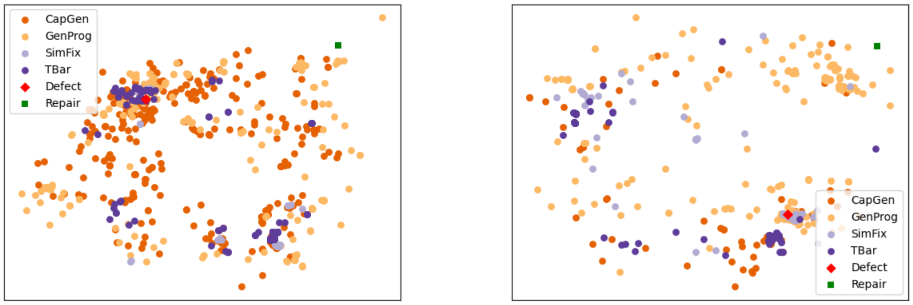
5 Experimental Results

This section presents our results which address the following research questions:

- **RQ1.** Do searches that explore more of semantic space find more repairs?
- **RQ2.** Do different APR tools generate semantically-distinct mutants for a given defect?
- **RQ3.** How does the syntactic diversity of mutants produced by different APR tools relate to their semantic diversity?

5.1 RQ1. APR Search Space Exploration and Repair Rates

We hypothesize that some APR methods sample more widely, that these differences arise from algorithmic decisions, and that these differences lead to dif-



(a) Semantic search space embedding for the first 600 mutants generated by each APR tool for Math 80.

(b) Semantic search space embedding for the randomly sampled 1000 mutants by each APR tool for Math 80.

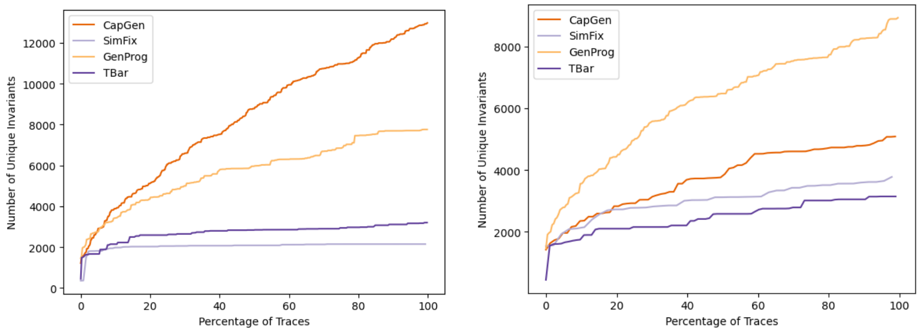
Fig. 1: Search space visualization of the Math 80 defect. Invariant sets for to generated mutants are embedded in 2D space using multidimensional scaling. Green square is the correct repair, while red diamond is the defect. GenProg and CapGen explore more of the search space than TBar and SimFix.

ferential repair rates for each tool depending on the search budget. We studied each tool's search progress on a representative defect from Defects4J, Math 80 (which relates to integer multiplication and Eigen decomposition).

Figure 1 visualizes our results using the two-dimensional embedding, for both the resource-limited early sampling and the final sampling. In the resource-limited cases (panel (a)), GenProg and CapGen explore more broadly (i.e., enclose the largest area) than either SimFix or TBar, which spend most of their evaluations in localized regions, and rarely test radically-different functionality. We conjecture that the heuristics used to order the mutated programs for testing in CapGen lead to a wider range of functionality being explored with relatively few samples. Panel (b), however, shows substantial differences. GenProg samples more broadly than the others, followed by CapGen, even though both use the same *insert*, *delete*, and *swap* mutation operators. TBar and SimFix, by contrast, are more clustered, with jumps between clusters from different repair templates.

The visualizations in Figure 1 show the relative scope of each tool's search, but they do not show the search trajectory. To address this issue, we treat the number of unique invariants as a countable proxy for unique functionality and ask how many unique invariants are explored by each additional individual program mutant that the tool evaluates (Figure 2a). This allows us to visualize both the number of unique invariants that are considered and approximately when they are discovered. The results, shown in Figure 2, indicate that CapGen and GenProg explore more unique functionality early in the search than TBar and SimFix. Both TBar and SimFix plateau early and remain relatively flat for the remainder of the search. We observed similar trends across the 1000-sample datasets (Figure 2b) and across all the defects we studied (data not shown).

These results support the hypothesis that APR searches that explore more widely also sample more semantically unique variants. However, the results do



(a) Each tool's unique invariant accumulation for the first 600 mutants

(b) Each tool's unique invariant accumulation for 1000 randomly sampled mutants

Fig. 2: Unique invariants from each APR tool for Math 80 over time. x-axis is % of traces evaluated, y-axis is the number of unique invariants. Tools that explore more of the search space also find more unique functionality over time.

Table 2: Semantic overlaps among APR tools. Each row reports the % of mutants that are semantically equivalent to at least one mutant from another tool.

	CapGen	GenProg	SimFix	TBar
CapGen	–	29.0%	25.2%	23.8%
GenProg	31.5%	–	10.8%	37.4%
SimFix	20.2%	86.0%	–	81.9%
TBar	38.0%	59.5%	52.6%	–

not predict relative repair rates. Remarkably, the tools that sample the largest extent of semantic space have lower reported repair rates across the entire Defects4J database, and vice versa. For instance, TBar has the best reported repair rates despite having the lowest exploration reach. Similarly, GenProg, which searched most broadly, reports the lowest repair rate. To summarize, we find that the targeted repair operations used by SimFix and TBar appear to outweigh the advantage of a high-diversity search. This surprising result highlights the key role of *representation*, since the implementation of mutation encodes a choice about representation — although we acknowledge that this result could also be related to the nature of the bug scenarios we studied. What remains unknown is how repairs are distributed throughout the search space: when repairs are close to the original program (e.g., defects in popular APR datasets that can be repaired with only one or two code edits), a thorough search of the nearby region will likely succeed more often than an extensive search of a wider region.

5.2 RQ2. Similarity of Semantic Search Spaces

The success of TBar suggests that combining multiple operators into a single tool increases the repair rate [25]. To test this, we examined the overlap between variants produced by the different tools in our study. We define overlap to be the total number of times each tool generates a mutant that is identical to one generated by another tool. The degree of overlap between two tools is a proxy for their similarity: we hypothesize that tools with high overlap will also repair a similar set of defects. Table 2 reports these results. CapGen and GenProg have low overlap, ≈26% average, with other tools. SimFix and TBar, on the other hand, are much more similar, as expected. TBar uses repair templates taken from several APR tools, often corresponding directly to the mutation operators of other tools in our study. It is thus unsurprising that TBar has the highest minimum overlap (38%). SimFix uses learned templates mined from human-generated repairs, but these also contain *fix patterns* [25] that mirror approaches found in the other tools.

On our dataset, SimFix and TBar have average *repair overlap* comparable to their semantic overlap rates (raw data not shown for brevity): 63% for SimFix and 50% for TBar. GenProg, however, has a much higher repair overlap (83%) compared to its semantic overlap (26%). Of the GenProg repairs, 67% are shared with CapGen and all are shared with TBar. This result can be explained: TBar incorporates all of GenProg's mutation operators. On average, CapGen has 52% repair overlap, ranging from 21% with GenProg to 84% with TBar.

This experiment reveals similarities among tools that may not be evident from their formal descriptions. It also suggests that the strategy of incorporating methods from earlier tools into a composite approach (e.g., TBar [25], Repairnator [47], and ARJA-p [57]) often succeeds. However, each such addition increases system complexity. An ideal combination would maximize performance and minimize cost and complexity. Search space visualizations (such as Figure 1) support making semantically-guided choices. Finally, focusing only on mutation operators may be misleading, as the tools we studied lack a powerful search heuristic. Even the GenProg family of tools, based on evolutionary computation, searches only in a limited way and relies primarily on mutation.

5.3 RQ3. Syntactic and Semantic Diversity of Mutants

To investigate the relationship between syntactic and semantic diversity for the mutants generated by different APR tools, we compared the rate at which semantically-distinct variants are discovered against the rate of at which unique syntactic variants are discovered. We find that syntactic variants are discovered much more frequently than semantic variants, e.g., between 4 and 20 times greater for Math80, depending on the tool. We observed similar trends for all other defects in our dataset. One explanation for this finding is that many syntactically distinct programs can compile to the same functionality.

Given this disparity, it is natural to ask if a higher semantic discovery ratio (i.e., techniques that find more semantically unique variants per syntactically unique variant) leads to higher overall performance. Our experiments do not support this hypothesis. Instead, we find that high semantic discovery ratios correlate with repair success only 30% of the time. GenProg had the highest ratio (approximately 38%) and the lowest repair rate. Conversely, SimFix had the lowest ratio across 30 of the defects while maintaining a high repair rate. For different defects, TBar and CapGen are typically intermediate between GenProg and Simfix in terms of this ratio, with TBar having the higher ratio of the two.

These results show that repairs are sparse in the search space and that targeting regions of the space where repairs are likely to be found is more effective than randomly sampling a large area of the semantic space. Although each tool finds many more syntactically-unique mutants than semantically-unique ones, it is unclear that this is problematic, given the apparent inverse correlation between semantic reach and repair rates. The success of the search algorithm depends heavily on problem representation, as is well-known in evolutionary computation.

6 Limitations and Threats to Validity

Soundness of invariant detection. Despite being the gold standard for dynamic invariant invariant detection, Daikon can infer invariants that may not hold in some parts of the program. To combat this limitation, we consider only invariants marked "high confidence." Additionally, since our approach is based on relative distances between detected invariants, any consistent detection errors are factored out by the difference operation and are unlikely to affect our results.

Syntactically-invalid patches. Some mutants produced by APR tools fail syntax or type checks, and cannot be analyzed by our approach. We note that other analysis methods also often fail on ill-formed patches [20], and a majority of the patches produced by the tools we consider are included in our analysis.

Generality. The results from our experimental study may not generalize to other APR tools beyond the four tools we examined, posing a threat to external validity. To mitigate this threat, we chose two tools from each of the main subcategories of APR tools that fall under the search-based paradigm (i.e., atomic change operators and template-based change operators [12, Section 6.1]).

7 Related Work

Earlier APR and Genetic Improvement work also considers the search space, typically characterizing it with respect to a specific characteristic, such as patch correctness, energy efficiency, or neutrality [13, 18, 27, 30, 36, 40, 42, 48, 49]. Researchers have characterized *neutral* mutations [13, 42] (mutations do not discernibly change program behavior — also called *sosies* or *safe*) and developed methods to combine them effectively [40]. Similar to neutral mutation work, Veerapen et al. visualized search spaces by considering local searches of the mutation graph [48, 49]. Langdon et al. also completed an exhaustive experiment on the triangle problem [18], concluding that the number of programs that pass all tests is much smaller than the overall search space.

Long et al. [30] characterized the effect on the search space of different configurations of the SPR and Prophet APR tools [28, 29], and found that increasing the search space generally increased the number of reachable repairs but also made it harder to find repairs. Similarly, we found that increasing the size of the semantic search space was not sufficient to find more repairs. This trade-off regarding choosing the best representation for a repair problem was explicitly addressed by the Genesis tool, which attempted to manage the size of the search space [27]. This prior work, however, does not consider the semantics of the underlying program beyond measuring how many tests passed. In the end, program behavior determines whether a patch correctly repairs a defect. This motivated us to consider mutant semantic similarity based on invariant set similarity.

Population-based repair tools have used semantics to increase initial population diversity [3] or guide exploration [5,6]. In both cases, the authors failed to find conclusive evidence that increasing population diversity leads to better APR performance. Similarly, we find no correlation between methods that consider a semantically-diverse set of programs and their ability to find repairs. However, our approach enables quantitative and qualitative analysis to investigate this relationship in greater detail than any of the previous works.

8 Conclusion

Many APR algorithms have been proposed, but relatively few ways have been proposed to compare them beyond empirical measurements of success at passing test cases or human assessment of patch quality. We add a new dimension to this work by proposing to assess how these methods explore *semantic search spaces*, extending earlier syntax-based analyses. Our automated, scalable approach leverages dynamic invariant detection and an efficient distance calculation to highlight the semantic differences between program variants. Further, our approach can be easily visualized in 2D space, admitting human interpretability.

Our empirical evaluation of four different search-based tools showed that, contrary to expectation, those methods that search most broadly can experience relatively low repair rates. This surprising result suggests that increasing semantic diversity in the search may not be as helpful as is generally believed. Second, tools that explore semantic mutants that are shared with other tools tend to have higher repair rates, providing an explanation for the success of modern composite tools like TBar or ARJA-p. Finally, tools that search extensively for novel semantics do not necessarily find more repairs, suggesting that tools with targeted repair mechanisms may explore important subsets of the search space. Our results suggest several new research directions. For instance, a deeper understanding of how repairs are distributed throughout syntactic and semantic search spaces would refine our understanding of these results. We hope that results like these will lead to a deeper re-examination of how APR tools are studied and compared, ultimately leading to even more improvements in the future.

Acknowledgements. We gratefully acknowledge the partial support of the NSF (CCF 2211749, 2141300, 1763674, 1908633, and CICI 2115075), DARPA (N6600120C4020, FA8750-19C-0003, HR001119S0089-AMP-FP-029), and AFRL (FA8750-19-1-0501).

References

1. Ackling, T., Alexander, B., Grunert, I.: Evolving Patches for Software Repair. In: GECCO 2011. pp. 1427–1434. ACM, Dublin, Ireland (2011). https://doi.org/10.1145/2001576.2001768
2. Arcuri, A.: Evolutionary repair of faulty software. Applied soft computing **11**(4), 3494–3514 (2011)
3. Beadle, L., Johnson, C.G.: Semantic Analysis of Program Initialisation in Genetic Programming. Genetic Programming and Evolvable Machines **10**(3), 307–337 (sep 2009). https://doi.org/10.1007/s10710-009-9082-5, https://link.springer.com/article/10.1007/s10710-009-9082-5
4. Cashin, P., Martinez, C., Weimer, W., Forrest, S.: Understanding Automatically-Generated Patches Through Symbolic Invariant Differences. In: ASE 2019. pp. 411–414. IEEE, San Diego, USA (nov 2019). https://doi.org/10.1109/ASE.2019.00046
5. Ding, Z.Y.: Patch Quality and Diversity of Invariant-Guided Search-Based Program Repair. arXiv (mar 2020), https://arxiv.org/abs/2003.11667v1
6. Ding, Z.Y., Lyu, Y., Timperley, C., Le Goues, C.: Leveraging program invariants to promote population diversity in search-based automatic program repair. In: 2019 IEEE/ACM International Workshop on Genetic Improvement (GI). pp. 2–9. IEEE (2019)
7. Eiben, A.E., Smith, J.E.: Introduction to evolutionary computing, Natural Computing Series, vol. 53. Springer (2003)
8. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: Proceedings of the 22nd international conference on Software engineering. pp. 449–458 (2000)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming **69**(1-3), 35–45 (dec 2007). https://doi.org/10.1016/j.scico.2007.01.015
10. Feldt, R.: Generating diverse software versions with genetic programming: an experimental study. IEE Proceedings-Software **145**(6), 228–236 (1998)
11. Fry, Z.P., Landau, B., Weimer, W.: A Human Study of Patch Maintainability. In: ISSTA 2012. p. 177. ACM, Minneapolis, USA (2012). https://doi.org/10.1145/2338965.2336775, http://dl.acm.org/citation.cfm?doid=2338965.2336775
12. Gazzola, L., Micucci, D., Mariani, L.: Automatic Software Repair: A Survey. IEEE Transactions on Software Engineering **45**(1), 34–67 (2017). https://doi.org/10.1109/TSE.2017.2755013
13. Harrand, N., Allier, S., Rodriguez-Cancio, M., Monperrus, M., Baudry, B.: A Journey Among Java Neutral Program Variants. Genetic Programming and Evolvable Machines **20**(4) (2019)
14. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping Program Repair Space with Existing Patches and Similar Code. In: ISSTA 2018. vol. 18, pp. 298–309. ACM, Amsterdam, Netherlands (jul 2018). https://doi.org/10.1145/3213846.3213871, https://dl.acm.org/doi/10.1145/3213846.3213871
15. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In: ISSTA 2014. pp. 437–440. ACM, San Jose, USA (jul 2014). https://doi.org/10.1145/2610384.2628055, http://dl.acm.org/citation.cfm?doid=2610384.2628055

16. Kou, R., Higo, Y., Kusumoto, S.: A Capable Crossover Technique on Automatic Program Repair. In: IWESEP 2016. pp. 45–50. IEEE, Osaka, Japan (2016). https://doi.org/10.1109/IWESEP.2016.15
17. Lance, G.N., Williams, W.T.: A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems. The Computer Journal **9**(4), 373–380 (1967)
18. Langdon, W.B., Veerapen, N., Ochoa, G.: Visualising the Search Landscape of the Triangle Program. In: European Conference on Genetic Programming. vol. 10196 LNCS, pp. 96–113. Springer (2017). https://doi.org/10.1007/978-3-319-55696-3_7, https://link.springer.com/chapter/10.1007/978-3-319-55696-3_7
19. Le, X.B.D., Thung, F., Lo, D., Le Goues, C.: Overfitting in semantics-based automated program repair. Empirical Software Engineering **23**(5), 3007–3033 (2018)
20. 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: ICSE 2012. pp. 3–13. IEEE, Zürich, Switzerland (2012). https://doi.org/10.1109/ICSE.2012.6227211
21. 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 Transactions on Software Engineering **41**(12), 1236–1256 (2015)
22. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A Genetic Method for Automatic Software Repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (jan 2012). https://doi.org/10.1109/TSE.2011.104
23. Le Goues, C., Pradel, M., Roychoudhury, A.: Automated Program Repair (dec 2019). https://doi.org/10.1145/3318162, https://dl.acm.org/doi/10.1145/3318162
24. Le Goues, C., Weimer, W., Forrest, S.: Representations and operators for improving evolutionary software repair. In: Proceedings of the 14th annual conference on Genetic and evolutionary computation. pp. 959–966 (2012)
25. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: TBAR: Revisiting Template-Based Automated Program Repair. In: ISSTA 2019. pp. 43–54. ACM, Beijing, China (jul 2019). https://doi.org/10.1145/3293882.3330577, https://dl.acm.org/doi/10.1145/3293882.3330577
26. Liu, K., Li, L., Koyuncu, A., Kim, D., Liu, Z., Klein, J., Bissyandé, T.F.: A critical review on the evaluation of automated program repair systems. Journal of Systems and Software **171**, 110817 (2021)
27. Long, F., Amidon, P., Rinard, M.: Automatic Inference of Code Transforms for Patch Generation. In: ESEC/FSE 2017. vol. Part F1301, pp. 727–739. ACM, Paderborn, Germany (aug 2017). https://doi.org/10.1145/3106237.3106253, https://dl.acm.org/doi/10.1145/3106237.3106253
28. Long, F., Rinard, M.: Prophet: Automatic Patch Generation via Learning from Successful Patches. Tech. rep., MIT-CSAIL (jul 2015), www.csail.mit.edu
29. Long, F., Rinard, M.: Staged Program Repair with Condition Synthesis. In: ESEC/FSE 2015. pp. 166–178. ACM, Bergamo, Italy (aug 2015). https://doi.org/10.1145/2786805.2786811, https://dl.acm.org/doi/10.1145/2786805.2786811
30. Long, F., Rinard, M.: An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In: ICSE 2016. vol. 14-22-May-, pp. 702–713. IEEE Computer Society, Austin, Texas (may 2016). https://doi.org/10.1145/2884781.2884872
31. Monperrus, M.: Automatic Software Repair: A Bibliography. ACM Computing Surveys (CSUR) **51**(1), 17 (2018)

32. Motwani, M., Sankaranarayanan, S., Just, R., Brun, Y.: Do Automated Program Repair Techniques Repair Hard and Important Bugs? Empirical Software Engineering **23**(5), 2901–2947 (oct 2018). https://doi.org/10.1007/s10664-017-9550-0, https://link.springer.com/article/10.1007/s10664-017-9550-0
33. Motwani, M., Soto, M., Brun, Y., Just, R., Le Goues, C.: Quality of automated program repair on real-world defects. IEEE Transactions on Software Engineering (2020)
34. Nilizadeh, A., Leavens, G.T., Le, X.B.D., Păsăreanu, C.S., Cok, D.R.: Exploring true test overfitting in dynamic automated program repair using formal methods. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 229–240. IEEE (2021)
35. Orlov, M., Sipper, M.: Genetic programming in the wild: Evolving unrestricted bytecode. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. pp. 1043–1050 (2009)
36. Petke, J., Brownlee, A.E.I., Alexander, B., Wagner, M., Barr, E.T., White, D.R.: A Survey of Genetic Improvement Search Spaces. In: GECCO 2019. pp. 1715–1721. ACM, Prague, Czech Republic (jul 2019). https://doi.org/10.1145/3319619.3326870, https://dl.acm.org/doi/10.1145/3319619.3326870
37. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C.: The Strength of Random Search on Automated Program Repair. In: ICSE 2014. pp. 254–265. ACM, Hyderabad, India (2014). https://doi.org/10.1145/2568225.2568254
38. Qi, Z., Long, F., Achour, S., Rinard, M.: An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In: ISSTA 2015. pp. 24–36. ACM, Baltimore, USA (2015). https://doi.org/10.1145/2771783.2771791
39. Renzullo, J., Weimer, W., Forrest, S.: Multiplicative weights algorithms for parallel automated software repair. In: 35th IEEE International Parallel and Distributed Processing Symposium (2021)
40. Renzullo, J., Weimer, W., Moses, M., Forrest, S.: Neutrality and Epistasis in Program Space. In: ICSE 2018. vol. 18, pp. 1–8. IEEE Computer Society, Gothenburg, Sweden (jun 2018). https://doi.org/10.1145/3194810.3194812, https://dl.acm.org/doi/10.1145/3194810.3194812
41. Schulte, E., Forrest, S., Weimer, W.: Automated Program Repair Through the Evolution of Assembly Code. In: ASE 2010. pp. 313–316. ACM, Antwerp, Belgium (2010). https://doi.org/10.1145/1858996.1859059, http://portal.acm.org/citation.cfm?doid=1858996.1859059
42. Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software Mutational Robustness. Genetic Programming and Evolvable Machines **15**(3), 281–312 (jul 2014). https://doi.org/10.1007/s10710-013-9195-8, https://link.springer.com/article/10.1007/s10710-013-9195-8
43. Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In: ESEC/FSE 2015. pp. 532–543. ACM, Bergamo, Italy (2015). https://doi.org/10.1145/2786805.2786825
44. Soto, M.: Improving patch quality by enhancing key components of automatic program repair. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1230–1233. IEEE (2019)
45. Staats, M., Hong, S., Kim, M., Rothermel, G.: Understanding User Understanding: Determining Correctness of Generated Program Invariants. In: ISSTA 2012. p. 188. ACM, Minneapolis, MN (2012). https://doi.org/10.1145/2338965.2336776, http://dl.acm.org/citation.cfm?doid=2338965.2336776

46. Tan, S.H., Yoshida, H., Prasad, M.R., Roychoudhury, A.: Anti-patterns in search-based program repair. In: ESEC/FSE 2016. vol. 13-18-Nove, pp. 727–738. ACM, New York, NY, USA (nov 2016). https://doi.org/10.1145/2950290.2950295, https://dl.acm.org/doi/10.1145/2950290.2950295
47. Urli, S., Yu, Z., Seinturier, L., Monperrus, M.., Monperrus, M..: How to Design a Program Repair Bot? Insights from the Repairnator Project. ICSE-SEIP 2018 p. 10 (2018). https://doi.org/10.1145/3183519, https://doi.org/10.1145/3183519.
48. Veerapen, N., Daolio, F., Ochoa, G.: Modelling Genetic Improvement Landscapes with Local Optima Networks. In: GECCO 2017. vol. 6, pp. 1543–1548. ACM, New York, NY, USA (jul 2017). https://doi.org/10.1145/3067695.3082518, https://dl.acm.org/doi/10.1145/3067695.3082518
49. Veerapen, N., Ochoa, G.: Visualising the Global Structure of Search Landscapes: Genetic Improvement as a Case Study. Genetic Programming and Evolvable Machines **19**(3), 317–349 (sep 2018). https://doi.org/10.1007/s10710-018-9328-1, https://doi.org/10.1007/s10710-018-9328-1
50. Vessey, I., Weber, R.: Some factors affecting program repair maintenance: an empirical study. Communications of the ACM **26**(2), 128–134 (1983)
51. Villanueva, O.M., Trujillo, L., Hernandez, D.E.: Novelty Search for Automatic Bug Repair. In: GECCO 2020. pp. 1021–1028. ACM, Cancun, Mexico (2020). https://doi.org/10.1145/3377930.3389845, https://dl.acm.org/doi/10.1145/3377930.3389845
52. Wang, S., Wen, M., Lin, B., Wu, H., Qin, Y., Zou, D., Mao, X., Jin, H.: Automated Patch Correctness Assessment: How Far are We? ASE 2020 pp. 968–980 (2020). https://doi.org/10.1145/3324884.3416590
53. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In: ASE 2013. pp. 356–366. IEEE, Silicon Valley, USA (2013). https://doi.org/10.1109/ASE.2013.6693094
54. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically Finding Patches Using Genetic Programming. In: ICSE 2009. pp. 364–367. IEEE, Vancouver, Canada (2009). https://doi.org/10.1109/ICSE.2009.5070536
55. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.C.: Context-aware Patch Generation for Better Automated Program Repair. In: ICSE 2018. vol. 2018-Janua, pp. 1–11. IEEE Computer Society, Pittsburgh, Pennsylvania (2018). https://doi.org/10.1145/3180155.3180233
56. Yang, D., Qi, Y., Mao, X.: Evaluating the strategies of statement selection in automated program repair. In: International Conference on Software Analysis, Testing, and Evolution. pp. 33–48. Springer (2018)
57. Yuan, Y., Banzhaf, W.: Making Better Use of Repair Templates in Automated Program Repair: A Multi-Objective Approach (2020). https://doi.org/10.1007/978-3-030-39831-6_26, https://link.springer.com/chapter/10.1007/978-3-030-39831-6_26