How does mutation occur in the wild?

2018 ACM/IEEE 4th International Genetic Improvement Workshop

How did you choose your mutations for this paper?

Neutrality and Epistasis in Program Space

Joseph Renzullo
Arizona State University
Tempe, Arizona
renzullo@asu.edu

Westley Weimer
University of Michigan
Ann Arbor, Michigan
weimerw@umich.edu

Melanie Moses
University of New Mexico
Albuquerque, New Mexico
melaniem@cs.unm.edu

Stephanie Forrest*
Arizona State University
Tempe, Arizona
stephanie.forrest@asu.edu

ABSTRACT

Neutral networks in biology often contain diverse solutions with equal fitness, which can be useful when environments (requirements) change over time. In this paper, we present a method for studying neutral networks in software. In these networks, we find multiple solutions to held-out test cases (latent bugs), suggesting that neutral software networks also exhibit relevant diversity. We also observe instances of *positive epistasis* between random mutations, i.e. interactions that collectively increase fitness. Positive epistasis is rare as a fraction of the total search space but significant as a fraction of the objective space: 9% of the repairs we found to look (and 4.63% across all programs analyzed) were produced by positive interactions between mutations. Further, the majority (62.50%) of unique repairs are instances of positive epistasis.

CCS CONCEPTS

• **Applied computing** → **Biological networks**; • **Software and its engineering** → **Software evolution**; **Search-based software engineering**;

KEYWORDS

Software evolution, Network science, Biological networks, Software testing and debugging, Automated software engineering

ACM Reference Format:
Joseph Renzullo, Westley Weimer, Melanie Moses, and Stephanie Forrest. 2018. Neutrality and Epistasis in Program Space. In *GI'18: GI'18:IEEE/ACM 4th International Genetic Improvement Workshop , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194810.3194812

*Also at the Santa Fe Institute

GI'18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5753-1/18/06…$15.00
https://doi.org/10.1145/3194810.3194812

1 INTRODUCTION

Genetic Improvement of software is more likely to succeed when the search algorithm is well-matched to the fitness landscape, whether for repairing bugs or improving a nonfunctional property of the program. The fitness landscape integrates information about the problem to be solved, its representation, and the search operators that are used. Neutral spaces are an important component of many fitness landscapes, and we hypothesize that they help determine the dynamics and ultimate success of genetic improvement. The topology of the *neutral space*, i.e. the network of programs (nodes) that are equivalent under the fitness metric and connected by single applications of the search operators (edges) is referred to as a *neutral network*. In this work, fitness is defined to be test-suite performance,[1] and two programs are considered to be neutral if they are test-suite equivalent, even if they behave differently on held-out tests or achieve identical input/output behavior using different implementations. We consider the three most widely used mutations in genetic improvement: delete, copy, and swap. We characterize the topology of the neutral space located around C programs, and we study *epistasis*, or interactions, between multiple neutral mutations (single-edits) to the same programs. Our results on extant C programs illustrate how neutral spaces can be studied in software, which we hope can inform design choices in genetic improvement (e.g., representations, choice of operators and search strategies) and ultimately, provide insights about the nature of software to inform other types of software engineering tools.

We motivate our study by observing that the impressive results achieved to date using genetic methods for improving software leverage the power of evolutionary search to only a limited extent. For example, most approaches rely primarily on mutation [14–16, 23–25, 35], use small population sizes[2] [14, 23, 24], and search for a small number of generations [14, 23–25]. Further, many of the repairs found using these methods can be reduced to a single mutation [18, 24, 25, 35], suggesting that a large part of the successes achieved to date can be reduced to a form of random search.

Tackling larger and more complex problems will require methods that can scale up effectively. This includes supporting larger populations, longer runs, and the productive use of crossover to

[1] Test-suite performance is the pattern of passing and failing test cases (input/output pairs). A program passes a test case if, when given a reference input, it generates an output that is character-for-character identical to the reference output; otherwise, it fails the test case.
[2] Some non-functional property improvement strategies, e.g. energy optimization, are a notable exception.

combine promising partial solutions and advantageous traits which have evolved independently in the population. In addition, we may need to consider more carefully where and when mutations are applied. For example, in biology, different mutation rates have evolved in different types of organisms, and mutation rates can vary widely even within a single chromosome [17] or at different times, as in affinity maturation when mutation rates are increased by up to five orders of magnitude. Baudry's work [2, 6] suggests that both targeting specific locations and varying mutation rates could be advantageous in genetic improvement.

How should we scale up current methods in genetic improvement to tackle much larger codebases and more complex repairs? What is the best way to navigate fitness plateaus (a.k.a. neutral networks), which are common when test cases are used as fitness functions? What portion of a fitness landscape is made up of this plateau? Does this portion change, along with mutational robustness, as we move away from the original program by iterated mutation? Are there more or better repairs near the original program? Or farther away in program-space? How do edits interact, as they accumulate? These motivating questions are not answered here, but in this paper we show how they can be investigated by adapting techniques from evolutionary biology, and we provide preliminary evidence that these methods are fruitful when applied to example programs.

In **Section 2** we discuss a framework for the methods we develop, which draws on results in the field of evolutionary biology. In **Section 3** we apply some of these results, demonstrate a method for building a neutral network for a computer program, and briefly discuss characteristics of interest. In **Section 4** we consider the problem of interacting mutations (a.k.a. higher order mutants). We find that although rare, positive epistasis comprises a significant fraction (9%) of discovered repairs to a latent bug (characterized by a held-out test case). Positive epistasis is also responsible for the majority (62.50%) of unique repairs. We conclude by considering why these methods translate between fields, and how they may be applied in the future.

2 BACKGROUND

We briefly summarize relevant results from the domain of evolutionary biology. Biologists hypothesize that natural selection leverages *neutral networks* to evolve a greater diversity of solutions than would be possible otherwise. We hypothesize that software has acquired some of the same properties that biologists have discovered in organic systems, particularly those that enhance evolvability. We consider whether neutral networks in software can be used to improve existing techniques in genetic improvement - especially in discovering repairs to complex bugs.

2.1 Biological Neutral Networks

Underlying all biological distributed algorithms is the design process that produced them—evolution by natural selection. Evolution has produced many compelling examples of distributed biological computation, including the social insects, brains, immune systems, quorum sensing among microbes, and flocking and herding behaviors among animals. We are interested in evolution as a distributed search process and how it uses *neutral networks* to produce complexity.

Genetic variants that have the same fitness are referred to as *neutral*, and a *neutral network* is a set of equal-fitness individuals related by single mutations. Neutral networks help a population of independent individuals manage the exploration vs. exploitation trade-off, a problem faced by any population-based search. Previous work argues that the topology of neutral networks is key to a population's ability to find high-fitness innovations (exploration) [34], while preserving already-discovered innovations (exploitation), and that neutrality and robustness are key to evolution [12, 13]. In a sense, it is safe for a population to search by spreading out over a neutral network: each member of the population maintains its fitness (because each is neutral) while collectively diversifying the population's genome. This makes it more likely that small mutations will lead to novelty.

Neutral mutations indicate a type of *robustness* in the sense that the mutation is a change that does not affect fitness. The interplay between robustness and evolution has been studied extensively in biology [19, 26, 34], producing many theoretical models, e.g., [5], and an increasing body of experimental results, e.g., [32]. It has been found, e.g. in [1] that large, connected neutral networks are prevalent in the regulatory gene binding sites of multiple species.

To summarize, neutral networks allow evolution to maintain fit phenotypes (external appearance and behavior) while exploring a large genetic search space. In this paper we study neutral networks in a computational context. Specifically, we analyze neutral networks in example computer programs, relate their structure to neutral networks observed in biology, and conjecture that software is evolvable, at least in part, because neutral networks enable stochastic search methods to search for useful innovations (e.g., bug repairs) without damaging existing functionality.

2.2 Mutational Robustness

In biology, *mutational robustness* refers to an organism's ability to preserve its phenotype in the face of internal genetic mutations [34]. We define mutational robustness in software to be the percentage of random mutations to a working program that leaves its behavior unchanged on the program's test suite.

Earlier work showed that mutational robustness is high (> 30%) [30] in a corpus of open-source programs, ranging from small, compact sorting programs to large, open-source implementations. Other authors have confirmed these findings in similar contexts, using similar methods, e.g. [2, 3, 9, 33]. We extend this earlier work to consider the topology of the network formed by these neutral mutations in an example program and find that repairs for bugs (innovations) are clustered in different regions of the network corresponding to different ways of repairing the same bug. We observe that the software neutral network resembles the topology predicted be most amenable to finding innovations in biology [5].

3 SOFTWARE NEUTRALITY

When a random mutation is applied to a working program, we say that the mutation is *neutral* if the mutation does not change the behavior of the program on its test suite. This is similar to, but not the same as, the notion of an equivalent mutant from mutation testing [30]. In mutation testing [8, 10, 28], the goal is to use mutated programs as checks on the completeness and coverage of a test

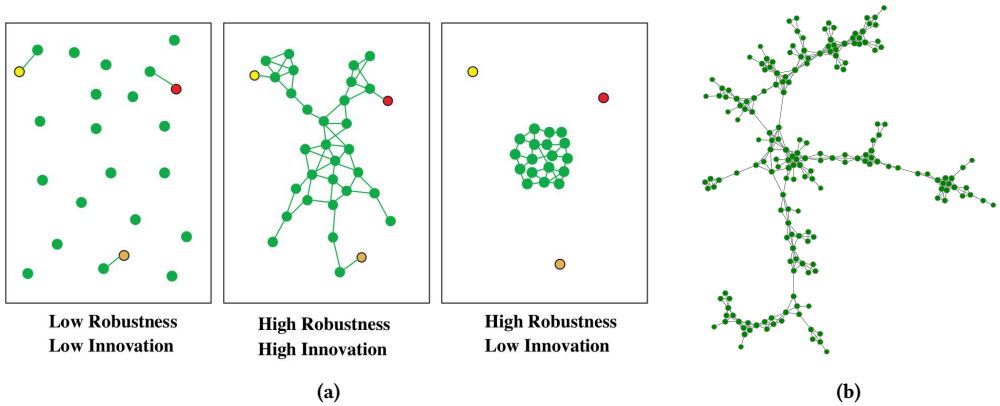

Figure 1: (a): Illustration of the trade-off between *mutational robustness* and *evolvability* (reproduced from Figure 3 in [5]). (b): Example neutral network of the computer program `look`. Each node represents a program variant that is two neutral edits away from the original program. In (a), individuals that are neutral to one another are colored green, and other phenotypes are represented by yellow, orange, and red; nodes are connected by an edge if they are a single mutation apart. A population can span a neutral network like the one in the center panel of (a) without loss of fitness so that it can discover potentially-beneficial innovations that are adjacent to different nodes in the neutral network. This is not possible in the left panel of (a). While there are innovations adjacent to some neutral mutations, there is no network to allow the population to get from one neutral mutation to another. In the right panel of (a), all single-edit mutations exist only in the space close to the current genotype and therefore can not explore to find innovations far from that genotype. A neutral network can create a space in which it is safe to search for innovations if it is of the type in the center panel of (a). In (b) we visualize the neutral network of Ultrix `look`, noting the striking resemblance to the high robustness/high innovation illustration in panel (a).

suite, on the assumption that random mutations should change program behavior. Here, we use test-suite equivalence as a metric of program fitness, and search for mutations which do not change the behavior of the program on the test suite. In mutation testing, semantically-distinct mutants which are not distinguished by test cases indicate a problem to be fixed: either a gap in the test suite or a problem in the program itself; in our work, these mutants are studied as potential avenues of program diversification and improvement.

We refer to positive and negative tests with respect to the original program—tests that a program passes are *positive*, and those that it does not pass are *negative*. We begin with source-level C programs, translate them into the corresponding abstract syntax tree (AST) and apply mutations at this level. Each node in the AST represents a complete C statement, and the mutations therefore manipulate complete statements, sometimes atomic and sometimes compound. Following previous work, e.g. [14], our experiments use three different kinds of mutations:

- **Delete** deletes a randomly selected node (and its subtree if one exists) from the AST.
- **Copy** selects a random node (and its subtree if one exists) in the AST and copies it to another random location.
- **Swap** selects two random nodes (and their subtrees if they exist) in the AST and exchanges their positions.

We require that mutations be applied only to parts of the AST that are executed by at least one test case.

We consider an example neutral network for the `look` utility, a small dictionary lookup program included in many Linux distributions. This example has a latent bug (one untested by the test suite). We generate variants of `look` by applying single mutations, one at a time, to a source program, and iterating according to **Algorithm 1**. We allow for the possibility that the program has been improved (i.e. a previously-failing test case may now pass), but any variants that break functionality from the original program (fail positive tests) are discarded, and we continue generating candidate variants until reaching the target number described in **Algorithm 1**.

Algorithm 1 Generate Variants

1: Let k=5, g=10, $\alpha = \frac{2}{3}$, and $\beta = 1 - \alpha$.
2: **for** 1 **to** g **do**
3: select k nodes at the edge of the graph
4: **for all** selected nodes **do**
5: apply single mutations to generate n children of this node, s.t. $p(\textbf{copy}) = \alpha$, $p(\textbf{delete}) = \beta$, and $\forall_{i \in [0,4]} : p(n = 2^i) \propto \frac{1}{\log_2 2^{i+1}}$
6: **end for**
7: **end for**

Because the neutral network of a computer program is, in principle, infinite, we sample a region near the original program to elucidate the most relevant structure. **Figure 2** shows the results of **Algorithm 1**, which produced a graph of 201 nodes representing

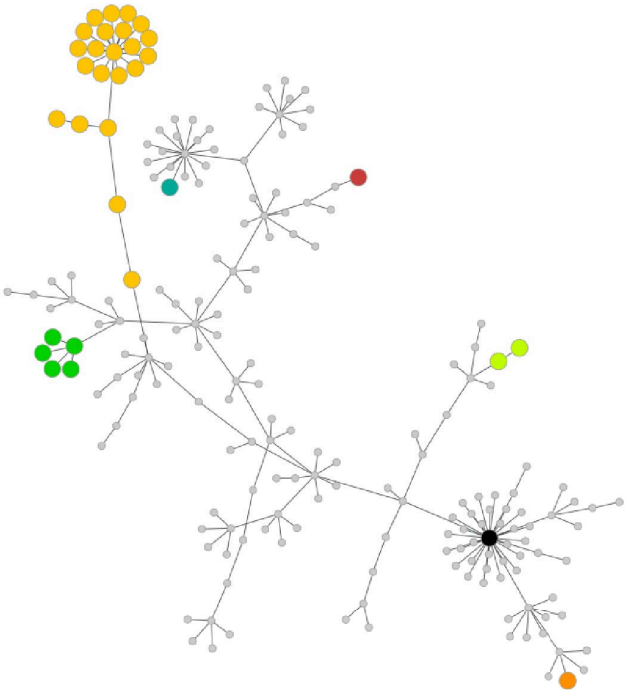

Figure 2: A neutral network of 201 variants of the `look` utility. Each node in this graph represents a mutated variant of `look`. The original program is shown in black. All of the variants depicted pass all of the positive test cases (they retain all originally-tested functionality). Gray nodes are variants which exhibit the exact same behavior on the test suite as the original program. Some variants additionally pass a negative test case which fails for the original program (i.e., they repair a defect)—these are represented by colored nodes. Nodes in the graph which share a color fix the defect in the same way.

200 mutations and the original program (shown in black).[3] The colored nodes of the graph represent variants that repair the defect in the original program. In **Figure 2**, a unique color is assigned to each mutation responsible for repairing the defect. By exploring the region close to the original program, we find diverse mutations which repair the defect, and observe that repaired programs are located in clusters in program-space.

These results demonstrate the existence and traversability of neutral networks in software. Crucially, they show the feasibility of making single edits, iteratively, which maintain working parts of the program while finding repairs to latent bugs. This leaves open the question of how the edits we have already added to the program interact with one another; we explore this topic in the next section. Understanding the extent and topology of neutral space may suggest new methods for finding repairs. For example, it may suggest better operators and search protocols tuned to the problem domain, thereby lowering the bar for repairing complex bugs.

[3]Note: The algorithm used for this experiment, reported in **Figure 2**, allows only for **copy** and **delete** operators - all other experiments reported in this paper also include **swap**.

4 EPISTASIS IN SOFTWARE

Epistasis (interactions between genes or mutations, see **Definition 4.1**) is known to exist in evolved biological systems. It can be a large component of the fitness contribution of a gene and thereby affect which mutations spread through a population. As far as we are aware, epistasis has not been explicitly characterized for bug repair, though some prior work [4] exists for energy optimization. The prevailing view in Genetic Improvement is that mutation-based methods work well for finding repairs that require single-edit patches, and rarely find multi-edit patches. With this in mind, this paper explores the prevalence of 2-edit epistasis for the domain of bug repair in example programs.

We aim to distinguish cases of epistatic interaction parallel to those described in theoretical biology. The prevalence and utility of interactions between single mutations can inform the strategies used to explore neutral networks in software. For instance, if positive interactions between neutral edits are common (i.e. those that provide more than an additive fitness improvement), then it may be optimal to search by combining large numbers of edits previously identified, individually, as neutral [11]. Additionally, if the types of repairs found by interacting mutations are different from the types found by single mutations, then strategies which harness epistasis may provide a better diversity of solutions than strategies which do not. We demonstrate our approach on example programs, and characterize the prevalence of epistasis in local regions around them.

First, we generate a population of single-edit mutations, and evaluate these for neutrality. Next, we generate combinations of these edits and reevaluate the test suite. By comparing the double-mutants to the single-mutants and the original program, we can measure the interactions. Positive epistasis occurs when two mutations that are neutral or deleterious on their own are advantageous when combined, e.g. the whole is greater than the sum of the parts. Negative epistasis is also possible, when mutations interact to weaken functionality of the program that neither on its own damages. An even stronger form of negative epistasis occurs when a combination of individually-safe mutations leads to a program that does not compile. We use the following definition of epistasis, which parallels magnitude epistasis in theoretical biology [1].

Definition 4.1. Epistasis
Let i and j be individual mutations and let P be a program. P_{ij} denotes P with mutations i and j applied to it. Let *f(P)* return the fitness of program P less the fitness of the original program.
Then:
if $f(P_{ij}) > f(P_i)$ and $f(P_{ij}) > f(P_j)$:
Epistasis(P_{ij}) = Positive
if $f(P_{ij}) < f(P_i)$ and $f(P_{ij}) < f(P_j)$:
Epistasis(P_{ij}) = Negative

4.1 Epistasis in `look`

Table 1 summarizes data for 20,200 mutated copies of `look`, across 4 independent, random samples. To generate each sample, we selected

	Sample 1	Sample 2	Sample 3	Sample 4	Average Percent
Does not compile	3901	3547	4304	2794	72.01%
Not Neutral	530	511	375	581	9.89%
Neutral	591	987	339	1618	17.50%
Repair	28	5	32	57	0.60%
Positive Epistasis	3	5	3	0	0.05%
Negative Epistasis	1	3	1	6	0.05%

Table 1: Summary results for four samples, totaling 20,200 mutations to `look` (columns Sample 1 to Sample 4). Average percent is the percent of total mutations that fall into each of the categories. The interactions described as positive epistasis and negative epistasis are segmented from the table in order to indicate that these are labels applied to programs already counted in the section of the table above. Each sample has 5,050 mutants.

10 AST nodes[4] from the part of the program covered by the test suite, uniformly at random and without replacement. To investigate epistasis across all edits, we do not discard non-neutral variants.[5] Next, we generate all possible edits to these AST nodes using **copy**, **delete**, and **swap**. This leads to a set of 100 single-edit mutations (10 delete, 45 append, and 45 swap). These edits were tested individually, and in all possible two-edit combinations. The small size of the `look` program enables us to perform this exhaustive study which covers a meaningful portion of the higher-order mutants.

We find evidence of both positive and negative epistatic interaction between mutations, though these are rare relative to non-interacting pairs. **Table 1** shows variation across samples in the relative prevalence of edits with different fitness values. The percentage of neutral mutations varies widely between samples. This is due to the particular random combination of AST nodes which were used to construct the sample. However, we find evidence of negative epistasis in every sample, and evidence of positive epistasis in all but sample 4.

This experiment shows high variability in the proportion of edits that do not compile, that are / are not neutral, and that are repairs. This is consistent with previous findings on differential robustness by statement type [2, 6], but the source of the variance in our data is not immediately apparent.

In this experiment, positive epistasis was observed only when edits combined to repair the latent bug, while still passing the test suite. Out of the 11 instances of positive epistatic interaction in **Table 1**, 7 combined a neutral edit with an edit that compiled but was deleterious on its own. The other 4 instances of positive epistasis were combinations of two edits which were individually neutral but collectively constituted a repair.

All 11 repairs modified the same two regions of code with one edit each, although they did so in two different ways. 8 of the repairs inserted or swapped a return statement into a region of code not normally executed - these edits were neutral. The second edit in these 8 cases changed the control flow of the program so that it hit this return when fed the buggy input; these were not always neutral edits, on their own. The remaining 3 repairs were a combination of two edits that both changed control flow, in such a way that the program exited when given the buggy input because it hit an already-existing return statement.

Listing 1: Example Repair

```
594    bot = 0L;
595 -- fseek(dfile, 0L, 2);
596    top = ftell(dfile);
597    while (1) { ... }
...
...
...
710    else {
...
717    tmp___0 = tmp;
718 ++ return (tmp___1);
719    }
```

The code excerpt above contains two edits to `look` that interact. When the program is fed a malformed input, the edit to line 595 changes the control flow of the program, causing the program to execute the return added by the edit to line 718 and exit. Without both of these edits, the program instead experiences a segmentation fault on the buggy input. The pair of edits repairs the bug.

Listing 1 gives an example of positive epistasis in `look`. The edit to line 595 removes a function call; this modifies the flow of the program, because the variable *top* in the next line takes on a new value. On its own, this edit is not neutral - it breaks existing functionality in the program and some previously-passing tests now fail. The edit to line 718 inserts a return statement in a branch of code that is normally not executed at all. On its own, this edit *is* neutral - the behavior of the program on the test cases is unchanged. Taken together, the change to line 595 causes the return to be hit in line 718, and the program exits when given the buggy input, causing it to pass all of the test suite.

[4]Ultrix `look` has exactly 100 AST nodes that are covered by test cases, so this is 10% of the possible AST nodes that could be mutated.

[5]This departs from previous work, which typically discards mutations that cause the program not to compile. It is also distinct from our own previous work, which discards non-neutral mutations, e.g. those that break previously-passing tests.

	ccrypt	look	merge	units	zune
Does not compile	81.91% (16545)	72.01% (14546)	76.35% (15422)	86.58% (17489)	6.75% (1363)
Not Neutral	8.69% (1756)	9.89% (1997)	20.69% (4180)	9.90% (1999)	85.97% (17365)
Neutral	8.83% (1783)	17.50% (3535)	2.96% (598)	3.52% (712)	7.15% (1444)
Repair	0.57% (116)	0.60% (122)	0.00% (0)	0.00% (0)	0.14% (28)
Positive Epistasis	0.00% (0)	0.05% (11)	0.01% (2)	0.00% (0)	0.12% (24)
Negative Epistasis	0.00% (0)	0.05% (11)	0.00% (0)	0.00% (0)	0.00% (0)

Table 2: Each program (`ccrypt`, `look`, `merge`, `units`, and `zune`) was mutated 20,200 times and evaluated against its test suite. The sampling procedure was the same as described in Section 4.1. Data are reported as the percentage of all mutations to that program which fall into the category described by the row labels (with raw counts in parentheses).

4.2 Epistasis in Other Programs

In order to investigate whether the results from our analysis of 20,200 variants of `look` generalized, we repeated the analysis with four other programs: `ccrypt`, `merge`, `units`, and `zune`. The sampling strategy used is identical to the one described in **Section 4.1**. Results are summarized in **Table 2**.

The data produced by the mutants of `zune` was similar to the data for `look` discussed in the previous section. Two of the twenty-eight repairs (7%) were the result of positive epistasis, comparable to the 9% of repairs found for look in this way. `ccrypt` and `units` exhibited no epistatic interaction, and `merge` had two cases of positive epistasis, but no repairs. One salient feature of these results is that they highlight the fact that not all positive epistasis leads to repair; often, two edits may be deleterious on their own but neutral when combined, especially if they counteract one another. A common example we observed was one edit which restored functionality which had been deleted or moved by another edit.

In **Table 2**, no repairs were found for `merge` or `units`, suggesting that not all defects are easily repaired with a small number of evaluations - e.g. the peaks in the fitness landscape for some problems may be rarer (or more distant from the original program) than are those for other problems. In order to investigate this, we ran a very large sample of the fitness landscape for `merge`. **Table 3** shows the results of 1,000,000 mutations applied to `merge`, of which 4,900 are single-edit mutations (an exhaustive sample) and the remainder are double-edit mutations (approximately four percent of all possible two-edit pairs).

We find 106 repairs to the defect in `merge` (an infinite loop on certain classes of input), of which three were single-edit mutations and 103 were double-edit mutations. Interestingly, all but two of the 103 double-edit patches which repair the bug are attributable to the three single-edit mutations, combined with other, neutral mutations. Two of the double-edit patches were the result of epistasis; in both patches, one neutral mutation interacted with one non-neutral mutation to produce a repair. This raised the question of how many of the unique repairs we find belong to each category, which we consider in **Table 4**. These double-edit patches also suggest that traversing a neutral network can help discover unique solutions, since some of the unique solutions we observed contain edits which are individually non-neutral and in combination effect a repair. Methods which evaluate individual mutations in isolation will tend to discard these non-neutral edits, some of which which have the potential to combine productively with other mutations: they are partial solutions.

	single edits	double edits
Does not compile	60.04% (2991)	74.36% (739988)
Not Neutral	26.53% (1300)	23.44% (233290)
Neutral	12.37% (606)	2.18% (21719)
Repair	0.06% (3)	0.01% (103)
Positive Epistasis	N/A	0.00% (30)
Negative Epistasis	N/A	0.00% (13)

Table 3: We report results for 1,000,000 mutants of `merge`. All of the possible single-edit mutations (4,900) and approximately 4% (995,100) of the possible double-edit mutations were evaluated. Data are reported as the percentage of all mutations to that program which fall into the category described by the row labels (with raw counts in parentheses).

	ccrypt	look	merge	zune	total
Single Edit	3	2	3	1	37.50% (9)
Epistasis	0	11	2	2	62.50% (15)

Table 4: Unique repairs to `ccrypt`, `look`, `merge`, and `zune`. Data for `merge` represents 1,000,000 mutants, and `units` was omitted since no repairs were found. Data for the remaining programs represents 20,200 mutants. The rightmost column reports the percent of total unique mutations that were a result of single edits and epistasis.

In order to assess the uniqueness of a repair, we consider the edits to the program that each contains. If a single-edit patch repairs the defect in the program, then it is categorized as a single edit in **Table 4**. Similarly, if a pair of edits repairs a program, but one is neutral and one on its own is a repair, then we classify this as a single edit. This is similar to the approach taken by le Goues et al. [14] for patch minimization. If, however, a patch requires both of its constituent edits in order to repair the defect, then we categorize this as epistasis. The hundreds of candidate repairs that were discovered by our

analysis collapse down by an order of magnitude (from 372 to 24). Most of the discovered single-edit repairs are functional duplicates; they contribute little to the diversity of solutions discovered. 62.50% of the unique repairs we found were the result of epistasis, and most of these (11 of 15) contained at least one non-neutral edit.

Epistatic interaction between multiple edits has been observed in other cases, e.g. [4] and Section 4.4 in [20], and we have found that it is a significant component of the objective space for bug repair. 9% of all of the repairs we found for `look` (and 4.63% of all repairs to all programs described in this paper) were the result of two interacting edits. Moreover, in Sample 2 described in Table 1, *all 5* of the repairs were instances of positive epistasis. Even if rare as a percentage of search space, we believe that epistatic interactions are common with respect to programs of interest for bug repair. Our experiment demonstrates how to measure epistasis in computer programs, and further application of this approach may reveal exploitable patterns in the interactions between mutations.

5 LIMITATIONS AND FUTURE WORK

5.1 Optimal Traversal of Neutral Spaces

There is existing theoretical work on neutral spaces and how to study them statistically, e.g. [7, 21, 27]. We have not yet formalized our mutation operators in a way that allows us to take advantage of this body of theory. Because we represent programs as ASTs and mutations correspond to actions programmers take when editing code, it is not trivial to make the mapping. For example, reversibility is a common assumption, but deleting a program statement is not naturally reversible. In the future, determining these correspondences could allow us to take advantage of this existing body of theoretical work.

5.2 Large Sample Spaces

The sample space of single edits to a program grows rapidly with program size $O(n^2)$, and for pairs of edits this compounds to $O(n^4)$, where n is the number of AST nodes in the program that are exercised by test cases. For even a small program like `look`, which has 100 such nodes, the space of possible pairs of edits is 10^8. We have not systematically evaluated the most efficient way to sample the space of possible mutations. For tractability, we performed experiments on a small number of relatively small programs; identifying reasonable sampling strategies will allow our approach to be scaled up to larger programs with more expensive test suites.

5.3 Higher-Order Mutants

We present results for single-edit mutants and double-edit mutants. It is unclear whether the rates of interaction we observed between pairs of edits will generalize to additional mutations, or whether higher-order epistatic interactions will be constructive or destructive.

5.4 Evolvability of Software

Our results suggest that some properties of evolved biological systems (neutral networks, mutational robustness, epistasis) are present in extant software to some extent. We conjecture that these same properties are a key reason why tools like GenProg can search effectively for repairs using evolutionary computation. We don't yet understand how or if these properties relate to human evolvability of software, a link which could potentially lead to more human-like and human-readable patches.

6 CONCLUSION

The ability of biological organisms to maintain functionality across a wide range of environments and to adapt to new environments is unmatched by engineered systems, even by those developed using Evolutionary Computation (EC) methods, which seek to mimic the natural evolutionary process. By studying the role of robustness in the context of computational artifacts, demonstrating the existence of neutral networks and their traversability, and translating and applying a method for measuring epistatic interaction between edits, this paper takes an initial step toward understanding and enhancing the use of biological distributed algorithms for genetic improvement.

Section 2 discussed the theoretical foundation for bridging evolutionary biology and software engineering; in particular, we argued that the concepts of *mutational robustness* and *neutrality* apply in both disciplines. **Section 3** described how to study neutral networks of computer programs quantitatively, and observed that they have several properties of interest. **Section 4** characterized the role of epistasis in some bug repairs. Our approach revealed that nearly 1 in 10 of the repairs we found to a latent bug in `look` arose from epistatic interaction between two mutations, some of which individually harm fitness (compensatory mutations). Additionally, our experiments revealed that the majority of unique repairs were epistatic, and that of these epistatic repairs a majority contained non-neutral edits. This reinforces the importance of neutral networks for exploring fitness landscapes like the ones we measure in this paper: accumulating neutral mutations, which interact with other, as-yet-untested mutations allows the process to discover unique solutions which are not accessible otherwise.

As we apply techniques in genetic improvement to larger software bases and more complex bugs, it will become important to combine single mutations into higher-order edits, capable of effecting complex repairs. Although we restricted our attention here to small programs that admit exhaustive searches, we hypothesize that these results will generalize to larger programs, and hope that this work encourages such investigations in the future.

The multiple, roughly-equivalent repairs to the same bug we described in **Section 4** provide a motivating example of the degeneracy between specification and programs, which mirrors the degeneracy between genotype and phenotype in biology. This degeneracy is believed to be the source of mutational robustness in biological systems and, we argue, similarly for computational systems. There are an infinite number of programs that can implement any given specification. Our study highlights this degeneracy for software and the way it enables effective automated search for software repairs. Importantly, software that is neutral with respect to one criterion, e.g. functionality, is not necessarily neutral with respect to nonfunctional properties such as run-time or power efficiency [29, 31].

Neutral networks enable the evolutionary search process to balance *exploitation* of known solutions and *exploration* for innovations. They also allow exploration of partial solutions. Eleven of the unique repairs we found (45.83%) would have been impossible if not discovered in a certain order because, without one half of the partial solution in place, the second half would break functionality and be discarded by existing methods. This well-known balancing act in search processes (e.g., [22]) is a key component of success in finding solutions to complex problems.

Modern software is often developed by multiple people, sometimes geographically and temporally remote from one another. A significant portion of the time and effort in software engineering is focused on combining contributions. Thus, software engineering is at least in part a distributed search (conducted by many programmers) for high-fitness (correct and useful) programs through the processes of inheritance (copying code), mutation (small edits), recombination of successful modules, and selection of the most useful programs. We speculate that this process provides a partial explanation for why software has acquired properties (mutational robustness and neutral networks) that resemble those of biological systems were by Darwinian processes. Finally, we suggest that the human-driven software engineering distributed search process has led to robustness and innovation, just as biological evolution has.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the partial support of NSF (1518878), (1619098) and (1629450), DARPA (FA8750-15-C-0118), AFRL (FA8750-15-2-0075), a James S. McDonnell Foundation Complex Systems Scholar Award, and the Santa Fe Institute. The authors also acknowledge the support of Schloss Dagstuhl, where some of the ideas presented were first conceived.

REFERENCES

[1] José Aguilar-Rodríguez, Joshua L Payne, and Andreas Wagner. A thousand empirical adaptive landscapes and their navigability. *Nature ecology & evolution*, 1(2):0045, 2017.

[2] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. Automatic software diversity in the light of test suites. *arXiv preprint arXiv:1509.00144*, 2015.

[3] Bobby R Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1327–1334. ACM, 2015.

[4] Bobby R Bruce, Justyna Petke, Mark Harman, and Earl T Barr. Approximate oracles and synergy in software energy search spaces. *RN*, 17(01):01, 2017.

[5] Stefano Ciliberti, Olivier C Martin, and Andreas Wagner. Innovation and robustness in complex regulatory gene networks. *Proceedings of the National Academy of Sciences*, 104(34):13591–13596, 2007.

[6] Benjamin Danglot, Philippe Preux, Benoit Baudry, and Martin Monperrus. Correctness attraction: a study of stability of software behavior under runtime perturbation. *Empirical Software Engineering*, pages 1–34, 2017.

[7] Christian A Duncan, Stephen G Kobourov, and VS Kumar. Optimal constrained graph exploration. *ACM Transactions on Algorithms (TALG)*, 2(3):380–402, 2006.

[8] Robert Geist, A. Jefferson Offutt, and Frederick C Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, 1992.

[9] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, Albert V. Smith, and Vilmundur Gudnason. Genetic improvement of runtime and its fitness landscape in a bioinformatics application. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1521–1528, New York, NY, USA, 2017. ACM.

[10] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[11] Martin Kellogg. Combining bug detection and test case generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1124–1126, New York, NY, USA, 2016. ACM.

[12] Motoo Kimura. *The neutral theory of molecular evolution*. Cambridge University Press, 1983.

[13] Motoo Kimura et al. Evolutionary rate at the molecular level. *Nature*, 217(5129):624–626, 1968.

[14] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2012.

[15] Fan Long and Martin Rinard. Prophet: Automatic patch generation via learning from successful patches. 2015.

[16] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[17] Michael Lynch, Way Sung, Krystalynne Morris, Nicole Coffey, Christian R Landry, Erik B Dopman, W Joseph Dickinson, Kazufusa Okamoto, Shilpa Kulkarni, Daniel L Hartl, et al. A genome-wide view of the spectrum of spontaneous mutations in yeast. *Proceedings of the National Academy of Sciences*, 105(27):9272–9277, 2008.

[18] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[19] Joanna Masel and Meredith V Trotter. Robustness and evolvability. *Trends in Genetics*, 26(9):406–414, 2010.

[20] Vinicius Paulo L Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G Camilo-Junior. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering*, pages 1–27, 2018.

[21] Alan Owen and Inman Harvey. Adapting particle swarm optimisation for fitness landscapes with neutrality. In *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, pages 258–265. IEEE, 2007.

[22] Rebecca J. Parsons, Stephanie Forrest, and Christian Burks. Genetic algorithms, operators, and dna fragment assembly. *Machine Learning*, 21(1):11–33, Oct 1995.

[23] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[24] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on*, pages 1875–1878. IEEE, 2013.

[25] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[26] Christian Reidys, Peter F Stadler, and Peter Schuster. Generic properties of combinatory maps: neutral networks of rna secondary structures. *Bulletin of mathematical biology*, 59(2):339–397, 1997.

[27] Christian M. Reidys and Peter F. Stadler. Neutrality in fitness landscapes. *Applied Mathematics and Computation*, 117(2):321 – 350, 2001.

[28] David Schuler and Andreas Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, 23(5):353–374, 2013.

[29] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 639–652. ACM, 2014.

[30] Eric Schulte, Zachary P Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, 2014.

[31] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)*, 30(6):152, 2011.

[32] Nobuhiko Tokuriki and Dan S Tawfik. Chaperonin overexpression promotes genetic variation and enzyme evolution. *Nature*, 459(7247):668–673, 2009.

[33] Nadarajen Veerapen, Fabio Daolio, and Gabriela Ochoa. Modelling genetic improvement landscapes with local optima networks. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1543–1548. ACM, 2017.

[34] Andreas Wagner. *Robustness and evolvability in living systems*. Princeton University Press, 2013.

[35] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.