Neutral Networks Enable Distributed Search in Evolution

Joseph Renzullo[1], Stephanie Forrest[1], and Melanie Moses[1]

[1]Department of Computer Science, University of New Mexico
{renzullo, forrest, melaniem}@cs.unm.edu

May 5, 2017

Introduction

Underlying all biological distributed algorithms is the design process that produced them, which is itself a distributed process—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. Here, we focus on evolution as a distributed search process and how it uses *neutral networks* to produce such 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) [11], while preserving already-discovered innovations (exploitation), and that neutrality and *robustness* are key to evolution [3, 4].

Neutral mutations are a form 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 [5, 6, 11], producing many theoretical models, e.g., [1], and an increasing body of experimental results, e.g., [10].

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 an example computer program, relate their structure to neutral networks observed in biology, and conjecture that software is evolvable at least in part because neutral networks enable programmers to search for useful innovations (e.g., bug repairs) without damaging existing functionality.

Software is a compelling and appropriate example for several reasons. First, open-source code repositories contain the complete evolutionary history of a software artifact, including when, where, and why modifications were made. The accessibility of all "mutations" (commits) makes it much easier to conduct large-scale analyses and study the dynamics of search than it is in living systems. Second, our current software bases are enormous (e.g., [2]), complex, and changing quickly in time, which provides a particularly rich domain for studying computational forms of 'evolution.' In spite of this complexity, we can easily measure most genetic (informational) or phenotypic (behavioral) software properties that are relevant.

In biology *mutational robustness* refers to an organism's ability to preserve its phenotype (external appearance and behavior) in the face of internal genetic mutations [11]. 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 ($> 30\%$) [8] in a corpus of open-source programs, ranging from small, compact sorting programs to large, open-source implementations. We extend this work to consider the topology of the network formed by these neutral mutations in an example program (`look`) and find that repairs for bugs (innovations) are clustered in different regions of the network corresponding to different ways of repairing the same bug. This suggests that a population of individuals (whether human programmers or a genetic algorithm) will likely be more successful in finding a repair than a single searcher. Further, we observe that the software neutral network resembles the topology predicted be most amenable to finding innovations in biology [1].

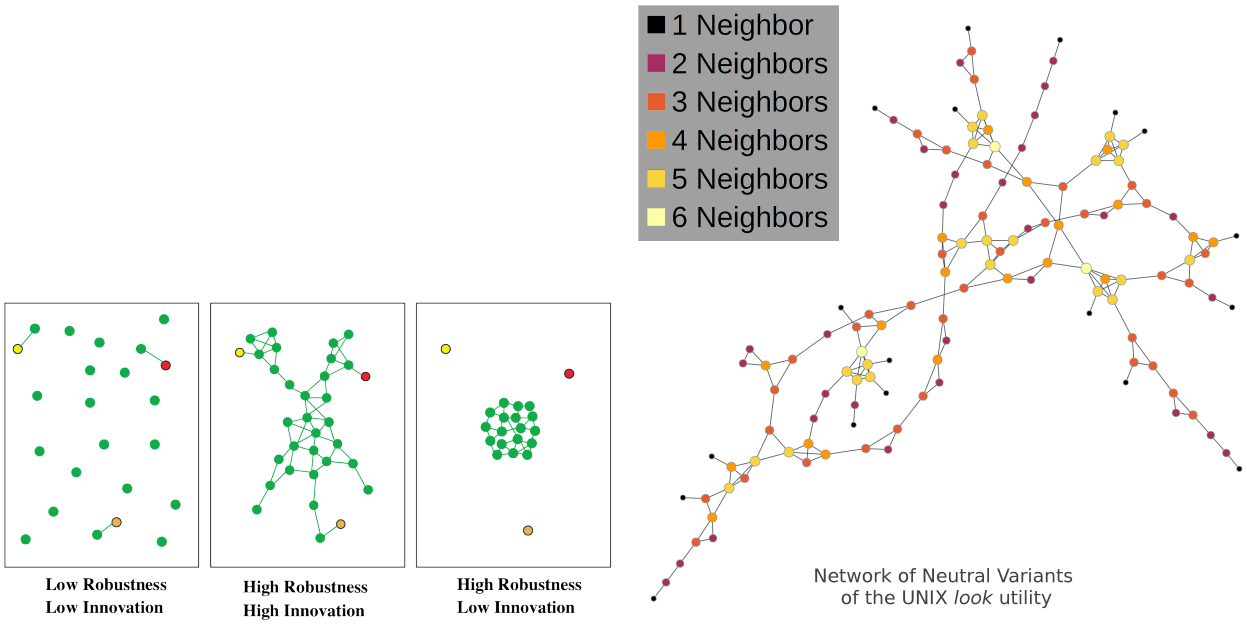

Figure 1: **(a):** Illustration of the trade-off between *mutational robustness* and *evolvability* (reproduced from Figure 3 in [1]). **(b):** Example neutral network of the computer program `look`. Each node represents a program variant that is exactly two neutral edits away from the original program. 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 **Figure 1 (b)** we visualize the neutral network of a computer program, noting the striking resemblance to the high robustness/high innovation illustration in panel (a).

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. 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. Our experiments use two 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.

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

In this paper, we consider an example neutral network for the UNIX `look` utility, a small dictionary lookup program included in many Linux distributions. We generate variants of `look` by applying single mutations (**copy** or **delete**), 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**.

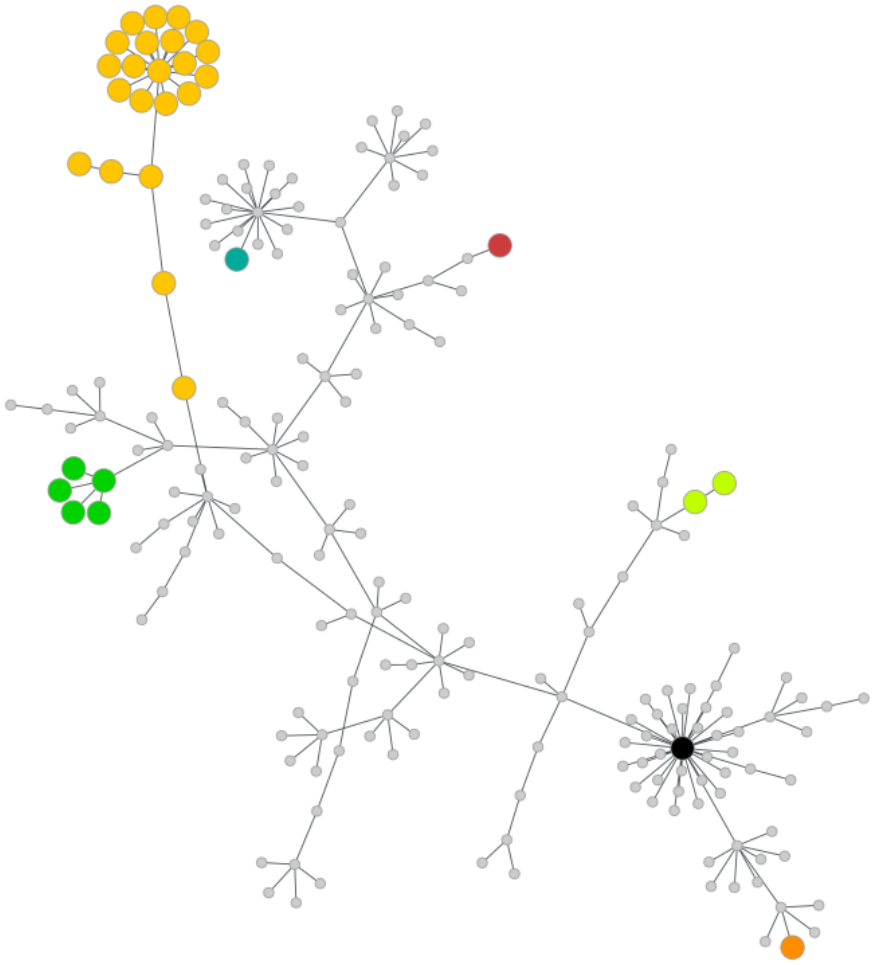

Figure 2: **A neutral network of 201 variants of the UNIX `look` utility.** Each node in this graph represents a mutated variant of UNIX `look`. The original program is shown in black. All of the variants depicted pass all of the positive test cases (they do not break functionality that was originally working). 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 (e.g. 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.

Algorithm 1 Generate Variants

1: initialize the graph by applying the logic in **line 5** to the source program node 5 times
2: **for** 2 **to** 10 **do**
3: select 5 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. $\forall_{i \in [0,4]} : p(n = 2^i) \propto \frac{1}{\log_2 2^{i+1}}$ and $p(mutation = \textbf{copy}) = 2/3$, $p(mutation = \textbf{delete}) = 1/3$
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 (UNIX `look`) to elucidate the most relevant structure. **Figure 2** shows the results of **Algorithm 1**, which produced a graph of 201 nodes representing 200 mutations (gray and various colors) and the original program (black). The colored regions of the graph represent variants that repair the defect in the original program. A unique color is assigned to each mutation responsible for repairing the defect. By exploring only a small region quite 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.

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, we hope to shed light on new methods for enhancing the adaptability of biological distributed algorithms.

We argue that 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. If true, then it should not be surprising that our experiments show that software has acquired properties (mutational robustness and neutral networks) that resemble those of biological systems that were produced by Darwinian evolution. We suggest that this distributed search process confers both robustness and innovation, just as does biological evolution.

Sorting algorithms provide a motivating example of the degeneracy between specification and programs which mirrors the degeneracy between genotype and phenotype. By this we mean that there is an infinite number of programs that can implement any given specification. Our study simply 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 criteria, e.g. functionality, is not necessarily neutral with respect to nonfunctional properties such as run-time or power efficiency [7, 9].

Neutral networks enable the evolutionary search process to both exploit known solutions and explore for innovations through robustness (the ability to make neutral changes to software that do not affect functionality against a suite of positive test cases) and the ability to innovate (the ability to discover variants that pass negative test cases).

References

[1] 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.

[2] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

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

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