Improving Performance of Automatic Program Repair using Learned Heuristics

Liam Schramm
Bard College, United States
ls2181@bard.edu

ABSTRACT

Automatic program repair offers the promise of significant reduction in debugging time, but still faces challenges in making the process efficient, accurate, and generalizable enough for practical application. Recent efforts such as Prophet demonstrate that machine learning can be used to develop heuristics about which patches are likely to be correct, reducing overfitting problems and improving speed of repair. SearchRepair takes a different approach to accuracy, using blocks of human-written code as patches to better constrain repairs and avoid overfitting. This project combines Prophet's learning techniques with SearchRepair's larger block size to create a method that is both fast and accurate, leading to higher-quality repairs. We propose a novel first-pass filter to substantially reduce the number of candidate patches in SearchRepair and demonstrate 85% reduction in runtime over standard SearchRepair on the IntroClass dataset.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; *Search-based software engineering*; • **Computing methodologies →** Machine learning;

KEYWORDS

Automatic program repair, Machine learning, Semantic search

ACM Reference format:
Liam Schramm. 2017. Improving Performance of Automatic Program Repair using Learned Heuristics. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 3 pages.
https://doi.org/10.1145/3106237.3121281

1 INTRODUCTION

The majority of the cost of most software projects is spent on software maintenance, and the majority of the cost of software maintenance is debugging [5]. Automated program repair methods offer the promise of dramatically lowering or even eliminating these costs [3], [4], [5], [7], [8], [9]. For an automated repair tool to be widely useful, it must meet certain criteria of efficiency, accuracy, and generality. For our purposes, we define these as:

- Efficiency: The ability to create and validate a patch for a given bug in a reasonable amount of time. An automated program repair tool must be fast enough to be useful in a real development environment.
- Accuracy: The probability that the patch proposed for a given bug is correct. A common issue in program repair is that of overfitting. If the test suite is used as the sole guide for the creation of patches, any functionality not covered by the test suite is likely to be deleted if any part of it contributes to buggy behavior [10].
- Generality: The range of bugs a repair algorithm is able to address. To reach the level of practical application, a repair method must be able to repair a wide range of bug types.

Several methods exist for automatic program repair, but all of them encounter problems with at least one of the above criteria. One recent approach to solving these problems is SearchRepair [3]. SearchRepair exploits the redundancy of developer code by copying and pasting sections of human-written code as patches. While this approach has shown significant improvements in accuracy over GenProg, RSRepair, and other approachs on the IntroClass Benchmark, it is also much slower [2], [3]. Other work has shown machine learning tools can quickly distinguish between good and bad patches, but with limited accuracy [8].

Our intuition is that is that it is possible to use machine learning to create a first-pass filter for SearchRepair [3]. This filter can eliminate a large number of bad patch candidates from SearchRepair's search space, dramatically reducing its runtime. We show that this improvement reduces runtime by over 85% over the default SearchRepair, with little decrease in generality.

2 RELATED WORK

Machine Learning in Program Repair. Prophet used machine learning to constrain the search patches with similar qualities to human patches and avoid overfitting[8]. This approach has several advantages. Firstly, it does not depend on a test suite, which reduces the threat of overfitting. Secondly, the learned classifier also runs much faster than a test suite, making it as a useful search heuristic. However, even with this approach, Prophet still has limited scope and under 50% accuracy on the GenProg benchmark [8].

SearchRepair and semantic search. One very promising solution to overfitting is semantic search, which is employed by SearchRepair [2], [3]. It cuts and pastes 5-7 line sections of code from other applications as patches (renaming variables as necessary). This approach has three key insights: (1) Due to code redundancy there is usually an existing human-written fix of this particular bug somewhere on GitHub [1], [3]. (2) While a single line may behave very

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3121281

differently out of context, a multiline block of code is less likely to do so (3) The replacement block was not built to the test suite when it was written, so there is less risk of it being overfit

SearchRepair also uses theorem proving to speed up this search [3]. Although theorem proving can be very slow, it is still much faster than running the test suite for every patch.

SearchRepair generates high-quality patches, but has issues with scalability [2], [3], [10]. The scaling problem arises from SearchRepair's use of foreign application's code. Since foreign application code use different variable names, SearchRepair must appropriately rename the variables to insert the code. However, it has no *a priori* way of knowing what the correct variable mapping is, so it must run theorem proving on each possible mapping [2]. Although there are ways to mitigate this expense, it is worst case *O(n!)* even before considering the theorem-proving. This makes a fast heuristic for avoiding bad patches very desirable.

3 METHODOLOGY

One way to avoid the large computational cost associated with SearchRepair's semantic search while maintaining high accuracy is to use a fast heuristic to eliminate poor patch candidates. Many machine learning methods classify instances very quickly, so with a sufficiently accurate classifier, it is possible to eliminate the vast majority of bad patches in a short amount of time. This filter must be independent of variable mappings in order to keep its cost from becoming prohibitively expensive. The theorem prover would then run on these selected candidates, ensuring that each proposed patch is indeed of high quality.

To create such a filter, we begin with the feature extractor proposed by Long and Rinard in Prophet [8]. This method creates a list of atoms (variables or constants) and operations the program uses on them. For instance, if a program checks `if(a > c && a < b)`, then assigns `median = a`, the feature extractor would record `a: lessthan, greaterthan, assign-Right`. The feature vectors of the buggy code and the proposed patch are then concatenated to create a new vector which is fed to the classifier [8]. If the classifier learns a variable-mapping-dependent pattern, then the classifier must be applied for each variable mapping (*n!* times per patch). Thus, we randomize the order in which atoms are encoded for every set of features, preventing the classifier from finding mapping-dependent patterns. This lets us apply the classifier only once per patch.

For this filter, we train a classifier to decide which patches should be evaluated by the theorem prover. We chose a random forest because it evaluated quickly and had the highest accuracy of all the classifiers we tested. The modified algorithm is as follows. The steps original to this paper are **set in bold font**.

(1) Encode a database of human-written code fragments as satisfiability modulo theories (SMT) [2], [3]
(2) Build a functional description of each fragment that describes its input-output profile
(3) Locate the bug in the buggy program
(4) **Extract features of the buggy code**
(5) **Extract features of each fragment in the database**
(6) **Combine the buggy features and the fragment's features into a single patch feature vector**
(7) **Apply the classifier to each patch feature vector in the database. If the patch feature vector is classified as correct, add the corresponding code fragment to a second database**
(8) Search the new, filtered database for a code fragment that matches the desired input-output profile
(9) Insert the code fragment and run the test suite

4 EVALUATION

First, we performed an evaluation of the filter independent of SearchRepair [3]. This provides a picture of the accuracy of the filter on its own, without the additional complication of the filter's interaction with SearchRepair. Using the proposed feature vectors, we trained a random forest to distinguish between related and unrelated code fragments. These code fragments were sampled from the IntroClass benchmark, which consists of beginner programming assignments from an Introduction to C programming course [6]. There are six sets of programs, each corresponding to an assignment from the class (checksum, digits, grade, median, smallest, syllables). The filter was trained to label pairs of fragments as either "from the same assignment" or "from different assignments".

Using the full dataset, we trained a random forest with 100 trees at 100 iterations and tested it using 10-fold cross-validation.

		predicted label		
		same	**different**	total
actual label	**same**	True positives: 19379 (97.6%)	False negatives: 495 (2.4%)	19871
	different	False positives: 1092 (5.4%)	True negtives: 17690 (94.6%)	18782
	total	20471	18182	

Total accuracy: 96.17%

Figure 1: This confusion matrix illustrates the accuracy of the filter. The labels on the top represent whether the filter predicted a given pair of fragments were from the same assignment, and the labels on the left show whether the pair were actually from the same assignment. Thus, the top left box is the number of pairs that were from the same assignment and were (correctly) classified as same-assignment.

Next, we incorporated the classifier into SearchRepair as a filter and tested it on the IntroClass benchmark. If the classifier labeled the code from the patch and the code from the source as "from the same assignment", then the patch was passed on to SearchRepair to be tested. Otherwise, it was discarded. Although filtered SearchRepair finished the benchmark in just over two days (52.778 hours), there has been substantially more difficulty testing unfiltered SearchRepair. The unfiltered version is slow and resource intensive enough that the authors have not yet gotten it to complete the benchmark without crashing or encountering other difficulties. During its best performance, it ran for roughly two weeks before the server ran into unrelated firmware problems and needed to be

reset. Since this ran 6.83 times longer than the filtered version without finishing, we know that the unfiltered version is at least 6.83 times slower than the filtered version. This translates to a roughly 85% speedup. Because the full unfiltered pipeline has not completed the benchmark, we cannot provide a measure of accuracy as we do not know how many correct patches the filter ruled out. As the false negative rate was only 2.4%, it seems unlikely that a significant number of correct patches were eliminated. The false positive rate was 5.4%, but this is only relevant to speedup, because SearchRepair validates all patches accepted by the filter anyway.

Together, these results show that filtered SearchRepair achieves greater efficiency than unfiltered SearchRepair [3]. The filter is not expected to significantly affect SearchRepair's accuracy. Its goal is only to quickly rule out very unlikely patch candidates that SearchRepair would have rejected anyway, so it is not likely to dramatically change the patches that SearchRepair accepts. Since SearchRepair already had higher accuracy than many of its competitors, this was not considered a problem [2]. Similarly, the filter is unlikely to affect SearchRepair's generality. The filter can be trained on any class of program, so as long as there is access to relevant training data, it should not be the limiting factor of generality.

5 CONCLUSION

Automated program repair holds a great deal of promise, but it is apparent that purely random methods are not sufficient for high-quality patches. Both machine learning approaches and semantic search methods have proven effective, but have drawbacks as well. Combining Prophet's fast feature-based search with SearchRepair's slower, more reliable constraint-solving produces a method that is accurate, general, and significantly more efficient. Additionally, this shows that with sufficiently non-linear learning methods, machine learning can be applied even when there is not a known variable mapping. This is a very useful result for any repair method based on semantic search or non-local code reuse.

ACKNOWLEDGMENTS

This research was funded by NSF research grant 1359275. The author would like to thank Claire Le Goues and Jugal Kalita for their guidance on this project, and Afsoon Afzal for her implementation of SearchRepair and assistance with debugging.

REFERENCES

[1] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Symposium on Foundations of software engineering (FSE 2010)*. ACM, 147–156.

[2] Yalin Ke. 2015. An automated approach to program repair with semantic code search. (2015).

[3] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *Conference on Automated Software Engineering (ASE)* (9–13). 295–306.

[4] X. B. D. Le, D. Lo, and C. Le Goues. 2016. History Driven Program Repair. In *Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224. DOI: http://dx.doi.org/10.1109/SANER.2016.76

[5] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE, 3–13.

[6] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *Transactions on Software Engineering (TSE)* 41, 12 (December 2015), 1236–1256.

[7] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178.

[8] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages (POPL '16)*. 298–312.

[9] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Conference on Software Engineering (ICSE '16)*. 691–701.

[10] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis (ISSTA 2015)*. 24–36.