Improving performance of automatic program repair using learned heuristics

Liam Schramm, Jugal Kalita

***Abstract*— Automatic program repair offers the promise of significant reduction in debugging time. Early generate-and-test systems, such as the genetic programming method GenProg, have had problems creating high-quality patches due to overfitting to the test suite. Recent efforts such as SPR and Prophet demonstrate that including external knowledge about the nature of correct repairs dramatically improves results. SearchRepair demonstrates that including semantic constraints can greatly improve patch quality, but has a high computational cost. Combining Prophet's learning techniques with SearchRepair's semantic constraints allows for a method that is both fast and accurate. This paper proposes a modification to SearchRepair that uses a fast classifier to quickly identify good candidate patches. We use a random forest to quickly classify whether two pieces of code are similar, allowing the search to focus only on the best patch candidates. We report 96% accuracy on this classification task, which is enough to greatly improve search speed. We then train another forest on data of whether or not patches were correct, and use this to filter for patches with a high likelihood of success.**

***Keywords*– Automatic program repair, Machine learning, Semantic search.**

I. INTRODUCTION

Software debugging is a tedious, expensive, and manual process. 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 [3]. Automated program repair offers the promise of dramatically lowering or even eliminating these costs. However, the fledgling field still struggles with making tools that are practical enough to reach the level of popular usage. For an automated repair tool to be successful, it must be meet certain criteria of efficiency, accuracy, and generality.

- 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 that it is cheaper and more convenient to fix the bug automatically than it is to fix it by hand
- Accuracy: The probability that the patch proposed for a given bug is correct. Although we would like to be confident enough in the algorithms recommendations that they would not need human oversight, this goal seems far off. At present, a more reasonable goal is that a correct patch is high enough on the list of recommended patches that a human can easily pick it out.
- Generality: The range of bugs a repair algorithm is able to address. While tools that address specific types of bugs may be adept in their particular field, there are simply too many different types of bugs for addressing them with individual programs to be practical. Methods that use certain preset types of modifications report that no one modification ever accounts for a large fraction of total repairs [10], [12], [4]. For automated program repair to become highly useful in a real-world environment, it must be general enough to fix a wide range of bugs

Several methods exist for automatic program repair, but all of them encounter problems with at least one of the above criteria. Semantic algorithms like DirectFix or Angelix are limited in the bugs they are able to address and sometimes scale poorly [1]. Search-based algorithms such as GenProg and SPR offer the possibility of a wider range of patches, but also are very likely to create patches which pass the test suite, but are ultimately incorrect [1], [4] [3]. One reason that search-based methods have this problem is that the search space is effectively infinite, and within this space, there are many more patches that pass all tests (plausible patches) and are incorrect than there are correct patches [5]. One solution to this problem is to use a fast, learned heuristic to focus the search on patches that look like human-written patches, which are more likely to be correct [2]. Another approach is to introduce semantic constraints and use theorem proving to ensure that undertested functionality is not deleted [8], [13].

In this paper, we combine this fast heuristic search with more rigorous theorem-proving techniques to create a method that is both fast and reliable.

II. BACKGROUND

Automated program repair is the process of patching bugs in a program given a test suite for that program. Currently, there are two common approaches to this problem: generate-and-validate, and semantic analysis. Generate-and-validate solutions create a large number of candidate patches, and then evaluate them one-at-a-time until an optimal candidate has been chosen. This candidate is then tested against the test suite, and if it passes, is accepted. If it is not accepted, a new patch is chosen, and the process is repeated until an acceptable patch is chosen, the search space is exhausted, or the time limit is reached. Semantic analysis works by extracting a repair constraint through symbolic execution. These constraints are then fed to theorem provers that infer a correct patch, which is then inserted.

In 2009, GenProg became the first algorithm to successfully perform automated program repair on large-scale real-world problems. It uses genetic programming to encode possible patches as variations to the source code. Its algorithm works approximately as follows:

1) Randomly generate a population of potential patches.
2) Run a subset of the test suite on each member of the population. The number of tests each member passes serves as the fitness function.
3) Breed the members of the population with the highest fitness.

In the months after GenProg's publishing, however, it became apparent that the method was not as effective as it had first seemed. Several studies found that GenProg frequently "fixed" bugs by deleting functionality. This is because its fitness function is defined solely by the test suite. Any functionality not covered by the test suite is likely to be deleted if any part of it contributes to buggy behavior [7]. Since the GenProg stops searching once it has found a patch that passes all the tests in the test suite, this pattern of functionality-deletion often contributes to GenProg failing to find patches that are within its search space [5].

This problem can also be viewed as an issue of overfitting. In a sense, GenProg overfit to the test suite by using it as the sole determinant of whether a patch was correct. Thus, it created patches which performed well on the test suite but poorly fit the actual function of the code. Most of the work done in the field since this discovery can be seen as an attempt to avoid overfitting.

Within the generate-and-validate framework, two methods have emerged for improving the quality of patches. The first approach was taken by Prophet and SPR. Although both still create incorrect patches about half the time, they reduce the likelihood of this happening by trying to produce a small number of patches that obey a collection of hand-coded rules rules[4] [2]. Each hand-coded rule, or schema, covers a specific type of repair. SPR and Prophet first use a target value search to check whether each schema is capable of generating a patch for the given bug. If it is not, the schema is discarded. They then generate a search space of patches from the remaining schemas, and test each of those

Since correct patches are sparse in the space of plausible patches, randomly generating solutions is more likely to produce plausible, incorrect patches than correct patches [5]. The authors believe that their careful construction of patches is less likely to produce incorrect patches because it utilizes more information about the program. Prophet also uses machine learning to select for patches with similar qualities to human patches[2]. This has several advantages. Firstly, it allows Prophet to rank patches without using the test suite, reducing the threat of overfitting. Since applying the learner is also much faster than running the test suite, it also serves a a useful search heuristic. This allows Prophet to focus its search significantly by looking for patches that share features with the correct patches.

The other approach, which was taken by SearchRepair, uses an algorithm that combines the search and the semantic analysis paradigms[8].

1) Encode a database of human-written code fragments as satisfiability modulo theories (SMT)
2) Locate the bug
3) Build a functional description of each fragment that describes its input-output profile
4) Search the database for a code fragment that matches the desired input-output profile
5) Insert the code fragment and run the test suite

While still being somewhat similar to its ancestor GenProg, SearchRepair has a few changes that help it combat overfitting. Firstly, it copies and pastes larger sections of code. The intuition is that while a single line may behave very differently out of context, a multiline block of code is less likely to do so. It would replace the entire buggy section with a correctly implemented version written by another human developer. Since this replacement block was not overfit to the test suite when it was written, there is less risk of it being overfit here. In addition, SearchRepair using theorem proving to speed up this search. Although theorem proving is very slow, it is still much faster than running the test suite for every patch.

SearchRepair generated patches with much higher quality than its purely random counterparts, passing 97.3% of tests in an independent test suite as opposed to GenProg's 68.7%. Although SearchRepair did not fix as many bugs as GenProg, RSRepair, or AE, this may be because these other methods might have fixed a large number of bugs simply by deleting functionality [8], [7].

SearchRepair has major issues with scalability [13]. It was almost two orders of magnitude slower than the second slowest method run on the Introclass benchmark, and three orders of magnitude slower than all other methods. One of the main reasons for this is the way its theorem prover works. Because SearchRepair uses pieces of code from other applications as its patches, it must 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 [13]. This means that the theorem prover must run $n!$ times on each patch, where n is the number of variables. Since SMT satisfiability is already an NP-Hard problem, SearchRepair effectively tries to brute-force an NP-Hard problem (variable mapping) that involves solving another NP-Hard problem (SMT satisfiability) on every iteration. This is, needless to say, very inefficient.

Amazingly, while this process is prohibitively expensive for large numbers of patches, the time taken to test a single patch is not terribly high. Since most code fragments used by SearchRepair have no more than 4 or 5 variables, a single patch can often be tested in less than a second. It is only when large databases of code fragments must be searched that problems arise.

III. Methods

As stated earlier, both SearchRepair and SPR/Prophet have drawbacks, either in their scalability or in their accuracy. For automatic program repair to become practically viable, a system must both create high-quality patches and be able to run in a reasonable amount of time.

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 candidates. Using a heuristic to iterate over all the possible patches in a database lets us pick out those with a high likelihood of success, avoiding the majority of the computation previously required. This heuristic must be independent of variable mappings in order 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.

Since the cost of using theorem proving is much higher than the cost of using the learner, a high false negative rate is much more acceptable than a high false positive rate. If, for instance, there was a false negative of 80%, we could simply try five times as many patches at fairly low cost. However, if we evaluate 1000 patches and have even a 10% false positive rate, this means we must evaluate 100 patches with theorem proving. Given SearchRepair's current speed, this would take about a minute. However, since the learner has been able to evaluate around 50,000 patches per second, it would be more efficient to take a much larger database of patches and use a learner with a very low false positive rate. Even if the learner has a false negative rate of over 50 percent, we can easily replace these these cases by just searching more patches.

To create such a heuristic, we begin with the feature extractor proposed by Long and Rinard in their Prophet paper [2]. This method creates a list of atoms (variables or constants) and what 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, R>`. The fact the `a` was on the right side of the assignment is recorded because assigning and being assigned to are very different operations.

The complete list of features is enumerated below.

1) var: True if the atom is a variable
2) const0: True if the atom is a constant equal to 0
3) constn0: True if the atom is a constant not equal to 0
4) cond: True if the atom is in a conditional
5) iff: True if the atom is in the body or the conditional of an if statement
6) prt: True if the atom is printed
7) loop: True if the atom is in the body or the conditional of a loop
8) EQ: True if the atom is in a statement with a ==
9) NEQ: True if the atom is in a statement with a !=
10) ret: True if the atom is returned
11) plus: True if the atom is in an addition statement
12) times: True if the atom is in a multiplication statement
13) minus_l: True if the atom is on the left side of a subtraction operator
14) divided_l: True if the atom is on the left side of a division operator
15) minus_r: True if the atom is on the right side of a subtraction operator
16) divided_r: True if the atom is on the right side of a division operator
17) increment: True if the atom is incremented
18) decrement: True if the atom is decremented
19) ASSIGN_L: True if a value is assigned to the atom
20) ASSIGN_R: True if the atom is assigned to another variable
21) LESS_THAN: True if the atom is in a statement with a <
22) GREATER_THAN: True if the atom is in a statement with a >
23) LESS_EQ: True if the atom is in a statement with a <=
24) GREATER_EQ: True if the atom is in a statement with a >=

These features are calculated for a given line(C), the three lines above it(P), and the three lines below it(N). The complete feature vector for a line in a program includes the set of features for C, P, and N for each atom in C.

Since the goal is to evaluate whether a given patch will work for given buggy section, we append the feature vectors of the buggy code and the proposed patch to create a new vector, and feed this vector to the learner. However, this gives rise again to the issue of variable mappings. In the Prophet feature extractor, the features of an atom in the buggy program are explicitly encoded alongside the features of the corresponding variable in the patched program. Since encoding this mapping, even implicitly, would defeat the point of a mapping-free heuristic, we randomize the order in which atoms are encoded for every set of features, preventing the learner from finding mapping-dependent patterns.

We train a random forest to classify patches as either correct or incorrect. Then, we use this classifier as a filter for what patches should be evaluated by the theorem prover. The modified algorithm is as follows.

1) Encode a database of human-written code fragments as satisfiability modulo theories (SMT)
2) Locate the bug
3) Build a functional description of each fragment that describes its input-output profile
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 learner 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

The following example serves to demonstrate the new algorithm. Suppose we have the following bug from the IntroClass benchmark.

```
/**/
#include <stdio.h>
#include <math.h>
int main(void)
{
  int int1, int2, int3, med;
```

```
   printf("Please enter 3 numbers separated
       by spaces > ");
   scanf("%d %d %d", &int1, &int2, &int3);

   if (((int1 < int2) && (int1 > int3)) ||
       ((int1 < int2) && (int1 > int3)))
     med = int1;
   else if ((((int2 < int1)) && (int2 >
       int3)) || ((int2 < int3) && (int2 >
       int1)))
       med = int2;
   else if (((int3 < int1) && (int3 > int2))
       || ((int3 < int2) && (int3 > int1)))
      med = int3;

    printf("%d is the median\n", med);
   return 0;
}
```

Since this program uses ¿ and ¡ operators instead of ¿= and ¡= operators, it will fail in the case that two or more of the numbers are equal.

Suppose then, that the modified SearchRepair's search space includes the following three patches

Patch 1

```
while ((status = scanf("%c", &it)) != EOF &&
    it != '\n')
sum = (sum + (long) it) % 64;
sum = sum + (long) ' ';
printf("Check sum is %c\n", (char) sum);
```

Patch 2

```
for(i=flag1-1; i>=flag2; i--)
{
  if(flag2==1 && i==1)
  printf("-");
  printf("%c\n", digit[i]);
}
```

Patch 3

```
if((a >= b && a <= c) || (a >= c && a <= b))
  median = a;
if((b >= a && b <= c) || (b >= c && b <= a))
  median = b;
else
  median = c;
printf("%d is the median\n", median);
```

Once the database has been constructed, the bug located, and the functional descriptions constructed, the algorithm begins the machine learning section. First, it constructs the feature vector for buggy section of code and for each of the patches. Then, it concatenates the two vectors into a single vector which is fed to the learner. In this case, let's presume that the learner classifies patch 2 and patch 3 as viable, but patch 1 as not viable. Patches 2 and 3 are added to the temporary database. SearchRepair then fetches patch 2 from the database and runs the theorem prover on it. Since it doesn't match the functional specifications given by the test suite, it is discarded. SearchRepair fetches patch 3 from the database and applies the theorem prover to it as well. Since patch 3 fits the semantic constraints, it is accepted, and its variables are replaced with the variables from the buggy program. Finallly it is tested against the test suite. It passes, and the final patch is returned.

```
/**/
#include <stdio.h>
#include <math.h>
int main(void)
{
  int int1, int2, int3;
  printf("Please enter 3 numbers separated
      by spaces > ");
  scanf("%d %d %d", &int1, &int2, &int3);

  if((int1 >= int2 && int1 <= int3) || (int1
      >= int3 && int1 <= int2))
    med = int1;
   if((int2 >= int1 && int2 <= int3) ||
       (int2 >= int3 && int2 <= int1))
      med = int2;
   else
      med = int3;
   printf("%d is the median\n", med);
  return 0;
}
```

IV. Evaluation

Using these feature vectors, we trained a random forest to classify pairs code snippets based on whether they were from programs that were trying to do the same thing. The intuition is that a program with similar functionality is far more likely to provide a correct patch than a program with very different functionality. These code snippets were sampled from the IntroClass benchmark. A pair was counted as a positive case if the two snippets were from programs submitted for the same assignment. Otherwise, they were counted as a negative case. The positive cases were sampled at a higher rate to make up for the asymmetry in the size of the classes.

To evaluate how effective different machine learning methods were for this set of features, we took a subset of the data and tried an array of machine learning techniques to see which produced the best results

Algorithm	Training Time	Accuracy
SGD	13.96 s	51.297%
Naive Bayes	.21 s	55.951%
Bayes Net	.76 s	56.002%
Random Forest	14.07 s	86.165%
J48	27.22 s	75.534%
AdaBoost	6.58 s	56.738%
K-nearest neighbors	n/a	81.587%
SMO	540.51	54.9593

TABLE I: Learner results

Based on these results, we concluded that a Random Forest would be the most effective learner for this application. For this reason, the rest of our experiments are conducted using a random forest.

Using the full dataset, we train a random forest with 100 trees, run for 100 iterations, and test it using 10-fold cross-validation. The forest was correctly able to classify 96.17% of the test cases. The false positive rate (classified as being from the same assignment, but really from different assignments) was 5.4%. The false negative rate (classified as being from different assignments, but really from the same assignment) was 2.4%. For this application, this is a very satisfactory result. Since the vast majority of the patches are from programs with a different purpose than the buggy program, being able to rule out almost 95% of these obviously bad candidates will significantly increase the search speed.

Once it was demonstrated that this method was effective at the code similarity task, it was applied to the patch recognition task as well. For this, we used a random forest and the same training data as Prophet [2]. We classify a patch as correct if and only it it is the developer-written patch. Although this is not necessary correct, very few copy-pasted code segments will produce correct results, whereas copying and pasting the developer patch into the buggy section always will. We did not use actual SearchRepair patches because it would be computationally infeasible to create a large database of patch data using such a method. Thus, pairs of bugs and their respective developer-written patches are considered positive cases, and other pairs are considered negative cases.

We found that the learner had a 66% false negative rate and a 0.11% false positive rate. It was able to evaluate a set of 118,000 proposed patches in four seconds (Not including time to write the file). Of the patches classified as correct by the learner, 20 out of 139, or about one in seven were in fact the developer patches. This is a huge increase in the density of correct patches. The space of potential patches generated for the learner contained about 1.6 correct patches per thousand. Filtering to one in seven is a dramatic improvement. Although most of the correct patches were discarded, the fact that the learner was able to search the space so quickly allows us to increase the total number of correct patches simply by expanding the state space.

V. Future work

Now that the learner has been shown to be effective, the next step would be to make the database adaptive as well. In the current implementation of SearchRepair, the patch database is scavenged more or less at random from a parent application or applications. Some of these patches are likely to be much more successful. For instance, a code snippet with an if statement checking whether or not an integer is equal to zero will likely be a far more useful patch than a section of code implementing a complex mathematical formula. If the formula code is used infrequently enough, then the cost of checking its relevance for every bug will outweigh the benefits of keeping it stored. If such a snippet is repeatedly not used, then it should be deleted from the database and replaced with another snippet with a better likelihood of repairing patches. Similarly, highly redundant snippets are possible in the current implementation. By keeping track of both how often a given snippet fails and how much its fixes overlap with those of other snippets, the database itself can be made adaptive. This would allow the possibility of establishing a general set of patches that cover a wide range of bugs. Other authors have tried similar techniques with good results. For instance, in *History Driven Program Repair*, the authors use a graph-based learner to learn general patch types from developer patches of real programs [12]. This learner is used to guide the creation of future patches, just like here the success of code snippets would guide their future use.

VI. Conclusions

Automated program repair holds a great deal of promise, but it is rapidly becoming apparent that purely random methods are not sufficient for high-quality patches. Both patch synthesis methods like SPR and Prophet, and semantic constraint search-based methods such as SearchRepair have proven effective, but each has drawbacks as well. Combining Prophet's fast feature-based search with SearchRepair's slower, more reliable constraint-solving allows us to produce a method that is both fast and accurate.

Acknowledgement

The authors would like to acknowledge the help of Claire Le Goues and Afsoon Afzal for their implementation of SearchRepair, as well as Kristen Justice for her guidance in this project.

References

[1] S. Mechtaev, Y. Jooyong, and R. Abhik R., Angelix: Scalable multiline program patch synthesis via symbolic analysis, in *Proceedings of the 38th International Conference on Software Engineering.*, New York, NY, 2016, pp. 691-701.

[2] F. Long and M. Rinard, Automatic patch generation by learning correct code, in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 2016, pp. 298-312.

[3] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each. *International Conference on Software Engineering (ICSE).* 2012, pp. 3-13.

[4] F. Long and M. Rinard, Staged program repair with condition synthesis, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 2015, pp. 166-178.

[5] F. Long and M. Rinard, An analysis of the search spaces for generate and validate patch generation systems. *Proceedings of the 38th International Conference on Software Engineering.* 2016, pp.702-713.

[6] C. Le Goues, S. Forrest, and W. Weimer, Current challenges in automatic software repair. *Software Quality Journal* vo. 21, no. 3, pp. 421-443, 2013.

[7] Z. Qi, F. Long, S. Achour, and M. 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.* 2015, pp. 24-36.

[8] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun, Repairing Programs with Semantic Code Search. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference.* 2015, pp. 295-306.

[9] M. Richardson and P. Domingos, Markov logic networks. *Machine learning*, 2006. vo. 62, no.1-2, pp.107-136.

[10] D. Kim, J. Nam, J. Song, and S. Kim, Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering.* 2013, pp. 802-811.

[11] C. Le Goues, N. Holtschulte, E.K. Smith, Y. Brun, P. Devanbu, S. Forrest and W. Weimer, The ManyBugs and IntroClass benchmarks for automated repair of C programs. *Software Engineering, IEEE Transactions*, 2015 vo. 41, no. 12, pp.1236-1256.

[12] X. Le , D. Lo, and C. Le Goues, History Driven Program Repair. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, vo. 1, pp. 213-224.

[13] Y. Ke. "An automated approach to program repair with semantic code search." M.S. thesis, CS Dept., ISU, Ames, IA, 2015.