The Strength of Random Search on Automated Program Repair

Yuhua Qi[*] , Xiaoguang Mao[†] , Yan Lei, Ziying Dai and Chengsong Wang
College of Computer
National University of Defense Technology, Changsha, China
{yuhua.qi, xgmao, yanlei}@nudt.edu.cn {dzynudt, jameschen186}@gmail.com

ABSTRACT

Automated program repair recently received considerable attentions, and many techniques on this research area have been proposed. Among them, two genetic-programming-based techniques, GenProg and Par, have shown the promising results. In particular, GenProg has been used as the baseline technique to check the repair effectiveness of new techniques in much literature. Although GenProg and Par have shown their strong ability of fixing real-life bugs in nontrivial programs, to what extent GenProg and Par can benefit from genetic programming, used by them to guide the patch search process, is still unknown.

To address the question, we present a new automated repair technique using random search, which is commonly considered much simpler than genetic programming, and implement a prototype tool called RSRepair. Experiment on 7 programs with 24 versions shipping with real-life bugs suggests that RSRepair, in most cases (23/24), outperforms GenProg in terms of both repair effectiveness (requiring fewer patch trials) and efficiency (requiring fewer test case executions), justifying the stronger strength of random search over genetic programming. According to experimental results, we suggest that every proposed technique using optimization algorithm should check its effectiveness by comparing it with random search.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Algorithms, Measurement

[*]The author is currently affiliated with The Institute of Measurement and Communication, Beijing, and can be reached at yuhua.qi@outlook.com.

[†]The corresponding author.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India

Keywords

Automated program repair, random search, genetic programming, search-based software engineering

1. INTRODUCTION

Researchers have devoted considerable attentions to the area of automated program repair [22, 13, 20], aiming at automatically generating patches for failure-aware programs without the manual efforts. With automated program repair, much resource cost can be saved in the software maintenance activity, for which patch generation is an essential task. Recently, researchers have proposed several techniques to automate the process of patch generation [21, 18, 27, 3, 28, 25, 1, 7, 16].

Of these techniques, a particularly effective approach is genetic-programming-based patch generation technique. Both GenProg [40] and Par [18], which won the distinguished awards in ICSE 2009 and ICSE 2013, respectively, showed the very promising results by using the same genetic programming algorithm to guide the patch generation process. To repair a faulty program, GenProg [40], and its extension [23, 21, 38] tries to produce a population of candidate patches in each generation by modifying source code according to mutation and crossover reuse structures, i.e., statement addition, removal and replacement, in other parts of the program. Once candidate patches are available, GenProg has to run fixed size of test cases to evaluate the fitness of each patch to facilitate the next patch generation. GenProg iterates the above steps until some valid patch passing all test cases is obtained, or when some limit (i.e., too much time or too many generations elapse) arrives. Par has the similar patch generation process but generates candidate patches via fix patterns learned from human-written patches, rather than via code reuse used by GenProg.

The two main contributions presented by GenProg and Par are 1) effective mutation operations (i.e., reuse structures for GenProg, and fix patterns for Par), and 2) using genetic programming to guide the patch generation process. Although the promising repair ability has been shown by GenProg and Par, whether the repair ability is got based on the guidance of genetic programming or just because the mutation operations are powerful enough to tolerate the inaction of genetic programming has rarely been studied. As described in [4, 15, 14], we can consider genetic programming to be effective at guiding the repair process, if and only if genetic programming can guide the repair process to search

valid patches much faster, in terms of requiring fewer patch trials[1], than random search.

In addition, similar to other search-based software engineering (SBSE) [15, 14] approaches, genetic programming often suffers from the computationally expensive cost caused by fitness evaluation, a necessary activity used to distinguish between better and worse solutions. In most cases, it is the computation of fitness (by running fixed size of test cases for each candidate patch) that occupies the largest part of the overall repair process [15, 23]. Hence, we should investigate whether genetic programming used by GenProg and Par can guide the patch search fast enough to offset the cost of fitness evaluation and thus speed up the repair process, even if genetic programming has been finally proved to be effective.

Given the above issues, we construct a new repair tool called RSRepair (Random-Search-based Repair), which tries to repair faulty programs with the same mutation operations as GenProg but uses random search, rather than genetic programming, to guide the patch generation process. Unlike genetic programming which requires fitness evaluation in the sense that GenProg has to run fixed size of test cases to compute the fitness of a candidate patch even if GenProg has been aware that the patch is invalid (i.e., the patched program has ever failed to pass some test case), random search has no such constraint. That is, RSRepair immediately discards one candidate patch once the patched program fails to pass some test case. Therefore, for one concrete patch RSRepair can take much fewer test case executions to check its validity. In addition, one second advantage of RSRepair is that RSRepair can speed up the process of early invalid patch identification using classic test case prioritization techniques [43, 30]. It, however, makes little sense to apply the prioritization techniques to GenProg, which still needs to compute the fitness of each patch even if some patches are found to be invalid early.

Experimental results show that RSRepair has much better performance than GenProg. In our experiment, we compared RSRepair with GenProg by running them to repair a set of 7 nontrivial programs with 24 real-world faulty versions. Through the rigorous statistical analysis, we observed that 1) RSRepair, in most cases, has the better repair effectiveness in terms of requiring the smaller *Number of Candidate Patches* generated before a valid patch is found (NCP), which indicates that genetic programming algorithm used by GenProg does not work well and even misleads the search process, confirming the concern presented by Andrea Arcuri and Lionel Briand in their ICSE 2011 paper [4, Page 3] to genetic programming used by GenProg, 2) RSRepair has the higher repair efficiency by requiring much fewer test case executions in the repair process.

In our early work [31], we have presented our insight with limited experiment. In this paper we will introduce our work in detail with the analysis to more experiment size. In short, this paper makes the following contributions:

- Using the simplest randomized algorithm (i.e., random search) to implement a repair tool called RSRepair (Section 3), which mostly has better performance over GenProg, a state-of-the-art repair tool using genetic programming to guide the patch search.
- Analyzing the experimental results and discussing the implications of such results for future research in the area of automated program repair in general (Section 5 and Section 6).
- Presenting a baseline approach based on random search in the promising research area of automated program repair, which is still in its infancy. We suggest that an advanced repair approach should always be compared against at least the baseline approach to check the advantage of the new approach, which is consistent with the declaration presented in [4, 15, 14].

[1]In this paper, we regard the step of checking the validity of one candidate patch in the repair process as a patch trial; in contrast, one concrete repair process can be regarded as one repair trial.

2. BACKGROUND AND RELATED WORK

2.1 Automated Program Repair

Generally, automated program repair consists of three steps: fault localization [42, 17], patch generation, and patch validation. When a bug is reported, we first use fault localization techniques to identify suspicious faulty code snippet causing the bug. Once faulty code snippet has been located, many candidate patches can be generated through the modification to that code snippet, according to specific repair rules based on either evolutionary computation [21, 3, 18] or code-based contracts [37]. When a candidate patch has been produced, regression testing, inclusive of negative test cases (reproducing the fault) and positive test cases (characterizing the normal behaviors), is commonly used to validate the correctness of produced candidate patch. The above procedure can be iterated over and over again until some valid patch is found. Any patch passing all these test cases is considered valid.

Automated repair techniques have received considerable recent research attentions. Guided by genetic programming, GenProg has the ability to repair programs without any specification, and GenProg is commonly considered to open a new research area of general automated program repair [26, 20], although there also exists earlier (e.g., [5, 2]) and concurrent work on this topic [6]. AutoFix-E [37] can repair programs but requires for the contracts in terms of pre- and post-conditions. JAFF [3] tries to automatically correct the faulty java programs using an evolutionary approach; the repair effectiveness of JAFF were not reported on real-world software with real bugs. PHPRepair [35] can automatically fix HTML generation Errors in php applications through string constraint solving. According to fix templates learned from human-written patches, Par [18] has fixed successfully many bugs existing in real world Java programs. SemFix [27] tries to repair faulty C programs via semantic analysis based on symbolic execution, constraint solving, and program synthesis; whether SemFix scales well to large-scale programs, however, is still unknown due to the possible expensive computation caused by semantic analysis.

Of these techniques, GenProg and Par, the two award-winning patch generation techniques, presented the very promising results. Both GenProg and Par use the same fault localization technique to locate faulty statements, and genetic programming to guide the patch search, but differ in the concrete mutation operations. Although promising results have been shown in their work, the problem of whether the promising results are caused by genetic programming or just because the used mutation operations are very effective is still not be addressed.

2.2 Genetic Programming and Random Search

Genetic programming, which is a variation of the well-known genetic algorithm, seeks to discover computer programs tailored to a particular task. Similar to traditional genetic algorithm, genetic programming uses genetic operations such as selection, crossover and mutation to evolve its populations to obtain some more adapted solutions [15]. In the research area of automated program repair, GenProg is the promising automatic solution designed to automatically and generically patching bugs in the software maintenance [20]. GenProg uses genetic programming algorithm to guide the patch generation process. As described in [15], GenProg needs to implement two key ingredients before the application of genetic programming: 1) the representation of the solution and 2) the definition of the fitness function. For the representation problem, GenProg represents each candidate patch as the Abstract Syntax Tree (AST) of the patched program. For fitness function, GenProg uses test cases to evaluate the fitness of each patch, and the patches with high fitness passing many test cases are selected for continued evolution in the next generation. That is, for each candidate patch, GenProg has to run fixed size of test cases to compute the fitness.

In contrast, random search, the simplest search algorithm that appears frequently in the SBSE literature, does not use a fitness function and thus does not incur the cost of fitness evaluation. In fact, GenProg produces the first population of candidate patches according to random search algorithm, and genetic programming starts to work on GenProg from the second generation. However, Andrea Arcuri and Lionel Briand found that GenProg often searched valid patches in the random initialization of the first population before the actual evolutionary search even starts to work. They doubted that the promising results may not be brought by genetic programming used by GenProg, because the patch search problem can be easy when random search would have likely yielded similar results. In this paper, however, we plan to further investigate whether genetic programming used by GenProg has the better performance over random search, when the actual evolutionary search has started to work.

2.3 Test Case Prioritization

As described earlier, random search is unguided, and thus requires no fitness evaluation. Specifically for automated repair, for random search one candidate patch can be discarded immediately once the patch is regarded as invalid. Although we can check whether one patch is valid through either regression testing or formal specifications, regression testing is most often used because formal specifications are rarely available in practice. With no need for fitness evaluation, we can speed up the patch validation process by maximizing the rate of invalid-patch detection, which is well-researched in the area of test case prioritization.

As described in [34] by Rothermel et al., test case prioritization problem can be defined as follows:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')\ [f(T') \geq f(T'')]$.

In the above context, PT represents the set of all possible orderings of T; f is a transition function used to evaluate an award value for any ordering of PT. In fact, f describes quantitatively the goal of prioritization, such as increasing the rate of fault detection of a test suite or increasing the coverage of coverable code in the system under test at a faster rate.

Techniques on test case prioritization are relatively mature, and much work has been presented in [43]. According to different coverage criteria a family of prioritization techniques are presented in [34]. Some research work evaluated traditional techniques on test case prioritization in context of time-aware test case prioritization [44]. Furthermore, there also exists much work focusing on different granularity levels such as the function level [9], system model [19], block level, and method level [8]. More details on prioritization techniques have been listed in [43].

Algorithm 1: The GenProg Algorithm

Input : Faulty program P
Input : Test cases T
Input : Mutation operator `Mutate`
Input : Crossover operator `Crossover`
Input : Full fitness predicate `FullFitness`
Input : Sampled fitness `SampleFit`
Input : Parameter $PopSize$
Output: One valid patch pt passing FullFitness

```
1 C_sub ← FaultLocalization(P, T);
2 Pop ← Mutate(PopSize,P, C_sub);
3 repeat
4 |   Fitnesses ← SampleFit(Pop);
5 |   Parents ←
  |   TournSelect(Pop, PopSize, Fitnesses);
6 |   Offsprings ← Crossover(Parents, P);
7 |   Pop ← Mutate(Parents,Offsprings);
8 until ∃ pt∈ Pop. FullFitness(pt) = Passed;
9 return pt;
```

3. IMPLEMENTATION

We have implemented a repair tool called RSRepair, which can automatically generate patches for faulty programs by using purely random search algorithm. RSRepair also comes with an adapted test case prioritization technique to speed up the patch validation process. In this section, we first describe GenProg, a state-of-the-art tool for automated C program repair, on which we built RSRepair. Then, we present the implementation details on RSRepair.

3.1 Repair Algorithm for GenProg

GenProg[2] has the ability of fixing bugs in deployed, legacy C programs without formal specifications. With the hypothesis that some missed important functionalities may occur in another position in the same program, GenProg attempts to automatically repair defective program with genetic programming [38]. Algorithm 1 [21] gives the process of patch generation of GenProg. Consider a faulty program P shipping with a set of test cases T, on line 1 GenProg first localizes the faulty code area C_{sub} according to test case coverage information. Then, line 2 initializes the population Pop by independently mutating the $PopSise$ copies of C_{sub} using purely random search. Once Pop is available, the fitness of each patch $pt \in Pop$ is computed via `SampleFit`, resulting in the fitness set $Fitnesses$. The patches with high fitness

[2]Available: http://dijkstra.cs.virginia.edu/genprog/

Algorithm 2: The RSRepair Algorithm

Input : Faulty program P
Input : Test cases T
Input : Mutation operator `Mutate`
Output : One valid patch pt

```
1  index ← 0;                    // Initialize the index value
2  {n_0, t_1, t_2, ..., t_n} ← T;
3  T ← {(n_0, 1)(t_1, index), (t_2, index), ..., (t_n, index)};
4  C_sub ← FaultLocalization(P, T);
5  SuccessFlag ← false;
6  repeat
7      pt ← Mutate(P, C_sub);
8      for i ← 0 to n do
9          //Check that whether pt is valid;
10         (t_index, index) ← GetTestcase(T, i);
11         if PatchValidation(P, pt, t_index) ≠ true then
12             temp ← (t_index, index + 1);
13             T ← Prioritize(T, temp);
14             break;
15         else if i = n then
16             SuccessFlag ← true;
17         else
18             continue;
19         end
20     end
21 until SuccessFlag = true;
22 return pt;
```

have more chances of being selected into the next generation (line 5) to continue evolution using `Crossover` (line 6) and `Mutate` (line 7) operations. Line 4-7 is repeated until either a valid patch is found to pass successfully `FullFitness` or some predetermined limits has elapsed. More details on GenProg can be found in [21].

3.2 Repair Algorithm for RSRepair

Similar to our prior work [30], we design the repair algorithm of RSRepair to generate automatically patches using purely random search algorithm combining with test case prioritization technique to improve the repair efficiency. Algorithm 2 [30] describes the repair algorithm in detail.

Given a faulty program P and test cases T, we reconstruct the test cases T to give T the ability of mapping each test case $t \in T$ to the number of candidate patches having been killed by t. The intuition behind this reconstruction is that the number of candidate patches killed by a test case t indicates, at least partly, the ability of t to detect invalid patches. Hence, we should prioritize test cases having killed more invalid patches to maximize early invalid-patch detection. Note that on line 3 RSRepair initializes the *index* value of n_0, one negative test case, with 1 due to its natural ability of fault reproduction.

To limit the search space for generating patches, we localize the faulty code area C_{sub} of P in the same way like GenProg.

The search begins by generating one patch pt through the `Mutate` operation, which can produce one concrete patch pt by modifying C_{sub}. Note that the patches produced by multiple calls to `Mutate` are probably different due to the application of randomized algorithm. Lines 8-20 run P against T *in order*, and reorder test cases T according to the running result. Given that the ith test case for T, line 10 calls the function `GetTestcase` to get the ith tuple $(t_{index}, index)$ of T; then, the function `PatchValidation` runs P patched by pt against t_{index} (line 11); if a fault is detected by t_{index}, then RSRepair records the fault by updating the tuple $(t_{index}, index)$ with $(t_{index}, index + 1)$ for T (line 12), and reorders each test case $t \in T$ by calling the function `Prioritize` (line 13), which reorders the test cases T in descending order of $index$. (If multiple test cases have the same $index$ value, RSRepair orders them randomly.) If P successfully passes all the test cases T (line 16), RSRepair considers that a valid patch is found (line 16), and terminates the repair process immediately with the output of valid patch (line 22).

3.3 Implementation of RSRepair

According to Algorithm 2, we implemented RSRepair by modifying GenProg, which is written in OCaml language. Specifically, RSRepair uses the same implementation of functions including `FaultLocalization` and `Mutate` in Algorithm 2 as GenProg in Algorithm 1.

RSRepair uses one simple statistical fault localization having been implemented in GenProg to determine fault localizations. With the assumption that a statement visited by negative test cases is more likely to be faulty than other statements, RSRepair computes the suspiciousness value *sp* of each statement in the way like that: a statement never visited by any negative test case has the *sp* value of 0; a statement visited only by negative test case has the high value of 1.0; a statement visited both by positive and negative test cases is given the moderate value of 0.1. RSRepair determines the probability of each statement based on these suspiciousness values; the bigger the suspiciousness value of a statement, the more chances of the statement being selected for mutation in the patch generation process.

Once some statements are selected for mutation, RSRepair will generate one candidate patch by mutating randomly these statements according to `Mutate` operations, i.e., statement addition, removal and replacement. This kind of mutation has been proved to be very effective in terms of repairing successfully 55 of 105 faulty programs in [21].

In fact, RSRepair produces candidate patches in the same way of generating randomly patches in the first generation of GenProg but without fitness guidance and crossover in the subsequent generations. For the process of patch validation, RSRepair validates candidate patches in the way described in Algorithm 2.

4. EXPERIMENTAL DESIGN

To check the performance of RSRepair, we compare it with GenProg, a state-of-the-art tool on automated program repair, on 7 subject programs with 24 faulty versions. We selected GenProg for the reason that GenProg is almost the only state-of-the-art automated repair tool which can fix bugs in real-world, large-scale C programs. Although in the paper [18] Par had better performance than GenProg in Java benchmarks, the paper also acknowledged the performance advantage may not hold in C benchmarks used by GenProg. What is more, unlike Par whose source code is not publicly available, both source code and experimental benchmarks of GenProg are publicly available, facilitating the experiment reproduction and comparison between RSRepair and GenProg. In addition, literature [11] and our previous work [32,

33] also used GenProg as the sole tool to conduct relevant experiments.

4.1 Research Questions

Our experimental evaluation seeks to address the following Research Questions:

RQ1: Whether can GenProg search a valid patch with fewer patch trials, compared to RSRepair?

Intuitively, genetic programming, an advanced search algorithm, should outperform purely random search, the simplest form of search algorithm, because higher quality solutions may be found faster with the aid of a fitness function. The intuition, however, is built on the assumption that fitness function can, at least partly, distinguish between better and worse solutions, and identify how close a candidate solution is to the optimal or near optimal solutions. A poor fitness function cannot benefit the search process, and even has the negative effectiveness. Specifically for GenProg, the fitness of each candidate patch is computed by counting the weighted number of passing test cases; the number is used to measure how close a candidate patch is to the valid patch. Although some promising results for GenProg have been presented in some recent serial papers [40, 23, 21, 38, 10, 22], the problem of whether the promising results are got based on the guidance of genetic programming or just because the mutation operations are powerful enough to tolerate the inaccuracy of used fitness function has never been studied.

Given the problem, RQ1 asks whether genetic programming used by GenProg works well to benefit the generation of valid patches. Andrea Arcuri and Lionel Briand [4] pointed out that a search algorithm should always be compared against at least random search to check how well the search algorithm performs, which can be measured by comparing the amount of effort expended between the search algorithm and random search. This effort, as described by Mark Harman in [15], is commonly measured through counting the number of fitness evaluations (i.e., patch trials) that were performed in the search process; the smaller the number is, the better the search algorithm is. However, in most recent work on GenProg [21] and Par [18] published in ICSE 2012 and ICSE 2013 respectively, we still do not find the effort comparison between genetic programming and random search. In this sense, RQ1 is the supplement to these work.

RQ2: Does GenProg find a valid patch much faster than RSRepair in terms of requiring fewer Number of Test Case Executions (NTCE) within a successful repair process?

Like other heuristic algorithm of SBSE, such as hill climbing and simulated annealing, the use of genetic programming has a benefit and a cost. The benefit is fewer patch trials should be needed with the aid of a fitness function (if the used fitness function works well). The cost is that the trial-and-error nature of genetic programming requires a large number of fitness evaluations during the search process; fitness evaluations may turn out to be computationally expensive and occupy the largest part of the overall computational cost of the search process [15]. An effective implementation of genetic programming algorithm should enlarge effectively the benefit to offset the evaluation cost, and further reduce the whole time cost in the search process.

Table 1: Subject Programs

Program	LOC	Test Cases	Version
`lighttpd`	62,000	5	bug-1806-1807
		16	bug-1913-1914
		18	bug-2330-2331
		17	bug-2661-2662
`libtiff`	77,000	73	bug-01209c9-aaf9eb3
		31	bug-0860361d-1ba75257
		73	bug-0fb6cf7-b4158fa
		33	bug-10a4985-5362170
		59	bug-4a24508-cc79c2b
		64	bug-5b02179-3dfb33b
		59	bug-6f9f4d7-73757f3
		77	bug-829d8c4-036d7bb
		35	bug-8f6338a-4c5a9ec
		76	bug-90d136e4-4c66680f
		73	bug-d39db2b-4cd598c
		76	bug-ee2ce5b7-b5691a5a
		60	bug-0661f81-ac6a583
		31	bug-3af26048-72391804
		33	bug-d13be72c-ccadf48a
`gmp`	145,000	144	bug-14166-14167
`python`	407,000	303	bug-69783-69784
`gzip`	491,000	2	bug-3fe0caeada6aa3-39a362ae9d9b00
`php`	1,046,000	4,986	bug-309892-309910
`wireshark`	2,814,000	53	bug-37112-37111

Specifically for GenProg, which computes the fitness of each candidate patch by counting the weighted number of passing test cases, test case executions most often takes the largest part of time cost in the whole repair process[23, Fig.8], especially for safety-critical programs equipping with many test cases. For simplification, we can measure the efficiency of GenProg using the NTCE when a valid patch is found [39]. Compared to random search, genetic programming used by GenProg can be regard as efficient only when the benefit (in terms of early finding a valid patches with fewer number of patch trials), brought by genetic programming, has the ability of balancing the cost of fitness evaluations, caused by genetic programming itself. However, whether the balance can be achieved by genetic programming used by GenProg has still been unknown so far. RQ2 is designed to answer the question.

4.2 Subject Programs

We selected the subject C programs used in the most recent work [21] on GenProg as the experimental benchmarks[3], each of which comes with real-life bugs existing in history versions. We conducted our experiment only on program versions which have ever successfully repaired by GenProg in [21]. For the `fbc` program version, we have the compilation trouble when we try to compile the program. For simplification, we also excluded the faulty versions (including one `lighttpd` version and 2 `libtiff` versions) that can be repaired successfully by modifying not less than two source files (i.e., .c file), because extra work has to be done to make CIL, a tool which can transform C program into AST used by both GenProg and RSRepair, work well to manipulate multi source files.

For `php` programs coming with over 4,000 test cases, validating only one patched program mostly takes several min-

[3]https://church.cs.virginia.edu/genprog/archive/genprog-105-bugs-tarballs/

utes, resulting in time-consuming repair process. Hence, if we conduct experiment on all the `php` program versions, too much time used by experiment evaluation is unavoidable (see [21, Table II]); in the paper [21] Amazon's EC2 cloud computing infrastructure including 10 trials in parallel was used for experiment evaluation. Given the expensive testing computation, we randomly selected one faulty `php` version, although these `php` versions shipping with many test cases, which can cause expensive fitness evaluation cost for GenProg, will give RSRepair more advantages. For `gmp`, `python`, `gzip`, and `wireshark`, there exists only one version having ever been repaired successfully by GenProg in [21]; we selected these versions in our subject programs.

Table 1, in total, describes our 7 subject programs with 24 versions in detail. The LOC (Lines Of Code) column lists the scale of each subject program, and the last two columns give the size of positive test cases and the version information. Note that for all programs in Table 1, although there are more test cases listed in [21, Table I]), in practice the concrete number of test cases used for each bug version is similar but different because not all the test cases work well for every version. In addition, for each subject program we reproduced the bug by executing one negative test case in the repair process.

4.3 Experimental Setup

For the purposes of comparison, we separately ran RSRepair and GenProg to repair all the 24 faulty versions described in Table 1. All the experimental parameters for GenProg in our experiment are similar to those settings in [21]: we limited the size of the population for each generation to 40, and a maximum of 10 generations for each repair process; the global mute rate *mute* is set to 0.01. In fact, for all generations except the first generation, there are another 40 candidate patches generated due to crossover. That is, for one concrete repair process, GenProg can iteratively produce no more than 40+80*9=760 candidate patches. For RSRepair, we also limited the size of the population for each generation to 40, and a maximum of 10 generations for each repair process; for each generation, using random search (without crossover) total 40 candidate patches are produced in the same way of the first generation. Hence, for fair comparison, we considered that RSRepair (GenProg) failed to repair one subject program in one repair process if the valid patch was not found within 40*10=400 candidate patches.

As described in [24, 21], the fitness function (i.e., `SampleFit` function in Algorithm 1) samples a random 10% of the positive tests to compute the fitness when the programs equipping with many test cases: GenProg test one candidate patch against the full suite iff the patch passes all the sampled test cases. Take `python`, which has 303 test cases in Table 1, for instance, GenProg has to run the patched `python` against the total 31 test cases, including 30 sampled positive tests and 1 negative tests, to compute the fitness of one candidate patches; the patched `python` is then tested against the full 303 test cases if it has passed all the above 31 test cases. In our experiment, we ran GenProg with the same fitness evaluation mechanism as described above.

All the experiments ran on an Ubuntu 10.04 machine with 2.33 GHz Intel quad-core CPU and 4 GB of memory. Since randomized algorithm is applied in both RSRepair and GenProg, we statistically analyze the experimental results. Specifically, for RSRepair and GenProg we separately performed 100 trials for each program with the seeds starting from 0 and ending in 99, and logged only the trials leading to successful repairs.

5. EXPERIMENTAL RESULTS

We first present experimental results in Table 2, which reports the summary statistics of the extracted information. The 3rd column and 4th column list the Mean and Median, respectively, of Number of Test Case Executions (NTCE) for each program. The 5th column gives the average time of repairing successfully each program. The 6th column presents the success rate when using one repair tool to fix each program. Recall that in our experiment we separately ran RSRepair and GenProg 100 times on each of the 7 programs with 24 versions. Hence, the success rate is n% if there are n successful trials; all the other statistics in Table 2 are computed according to the n successful trials. The 7th column reports the effect size on the difference between RSRepair and GenProg on NTCE, with the p-value presented in the 8th column.

Next we use our study results to address the two research questions (Section 4.1).

5.1 RQ1

To answer this question, we need to describe the measurement of how well the search process performed, when comparing GenProg and RSRepair. As described in [15], to conduct a fair comparison, it is important to establish the amount of effort expended by each search algorithm to find the optimal or near optimal solutions; "This effort is commonly measured by logging the number of fitness evaluations that were performed". Specifically for GenProg, every candidate patch needs fitness evaluation. Hence, we can compute the number of fitness evaluations by logging the Number of Candidate Patches generated in the repair process (NCP).

For GenProg and RSRepair, the one with smaller NCP (when some valid patch is found) is considered more effective. Computing precisely NCP for GenProg and RSRepair, however, is difficult due to the stochastic nature of randomized algorithms. To mitigate against the effects of random variation, in our experiment both GenProg and RSRepair are repeated 100 times with different seeds. Normally, the NCP of each repair trial should be logged for the subsequent statistic analysis. However, running each repair trial without terminal until some valid patch is found, sometimes, is too computationally expensive when searching a valid patch is a hard problem (in term of requiring lots of NCP), especially when the programs come with many or long-running test cases. For instance, it may take several hours to search a valid patch for `wireshark` in Table 1 with GenProg or RSRepair. As such, according to the work [18, 21], we limited the value of NCP within 400; for each repair trial we consider it fails to repair one program if a valid patch is not found when the limitation arrives. In the experiment, it is unfair comparison if we directly analyze experimental results according to the NCP of only successful repair trials. For example, for two concrete repair trials on the same program, RSRepair finds a valid patch with the NCP value of 399, and for GenProg the NCP value is 430. Although RSRepair has the better repair ability in term of smaller NCP value, the logged value of 399 has the negative effect on the repair ability of RSRepair using statistic analysis, because the value of 430 on GenProg

Table 2: Experimental Results by RSRepair and GenProg

Program	Approach	Mean of NTCE	Median of NTCE	Avg. Time(s) Per Repair	Success Rate	A-test on NTCE	p-value
lighttpd-bug-1806-1807	RSRepair	111	88	290.872	88%	0.616883	0.023741
	GenProg	194	116	268.502	49%		
lighttpd-bug-1913-1914	RSRepair	199	201	513.130	38%	0.597744	0.424466
	GenProg	235	229	352.479	7%		
lighttpd-bug-2330-2331	RSRepair	70	60	168.087	100%	0.709655	0.000001
	GenProg	182	102	270.021	87%		
lighttpd-bug-2661-2662	RSRepair	24	22	19.058	100%	0.794850	0.000000
	GenProg	38	32	24.869	100%		
libtiff-bug-01209c9-aaf9eb3	RSRepair	79	78	18.744	100%	0.940400	0.000000
	GenProg	122	109	25.764	100%		
libtiff-bug-0661f81-ac6a583	RSRepair	102	92	293.806	100%	0.850988	0.000000
	GenProg	304	207	698.341	86%		
libtiff-bug-0860361d-1ba75257	RSRepair	63	57	269.149	100%	0.787165	0.000000
	GenProg	177	119	439.535	97%		
libtiff-bug-0fb6cf7-b4158fa	RSRepair	187	170	242.009	100%	0.813636	0.000000
	GenProg	587	505	465.513	77%		
libtiff-bug-10a4985-5362170	RSRepair	51	47	42.582	100%	0.834946	0.000000
	GenProg	150	89	73.669	93%		
libtiff-bug-3af26048-72391804	RSRepair	55	46	213.590	100%	0.796753	0.000000
	GenProg	135	91	470.891	97%		
libtiff-bug-4a24508-cc79c2b	RSRepair	66	64	55.937	100%	0.926400	0.000000
	GenProg	115	94	88.147	100%		
libtiff-bug-5b02179-3dfb33b	RSRepair	119	103	121.409	100%	0.826053	0.000000
	GenProg	338	218	241.101	95%		
libtiff-bug-6f9f4d7-73757f3	RSRepair	68	65	66.104	100%	0.922800	0.000000
	GenProg	123	101	114.560	100%		
libtiff-bug-829d8c4-036d7bb	RSRepair	154	148	233.548	88%	0.830168	0.000000
	GenProg	540	554	539.218	73%		
libtiff-bug-8f6338a-4c5a9ec	RSRepair	47	43	25.950	100%	0.867400	0.000000
	GenProg	92	75	33.386	100%		
libtiff-bug-90d136e4-4c66680f	RSRepair	100	93	82.677	100%	0.936600	0.000000
	GenProg	349	225	132.086	100%		
libtiff-bug-d13be72c-ccadf48a	RSRepair	89	73	500.578	99%	0.757386	0.000000
	GenProg	265	161	907.452	80%		
libtiff-bug-d39db2b-4cd598c	RSRepair	110	99	46.467	98%	0.883716	0.000000
	GenProg	387	277	96.754	96%		
libtiff-bug-ee2ce5b7-b5691a5a	RSRepair	104	94	99.444	100%	0.926650	0.000000
	GenProg	354	238	131.874	100%		
gmp-bug-14166-14167	RSRepair	312	301	606.511	58%	0.817529	0.000590
	GenProg	663	530	472.828	12%		
python-bug-69783-69784	RSRepair	434	408	452.185	37%	0.990991	0.000000
	GenProg	2572	2318	1409.825	21%		
gzip-bug-3fe0ca...	RSRepair	255	265	246.135	21%	0.285714	0.194481
	GenProg	168	181	161.441	4%		
php-bug-309892-309910	RSRepair	4997	4994	457.486	100%	1.000000	0.000000
	GenProg	20837	16986	1440.835	97%		
wireshark-bug-37112-37111	RSRepair	167	179	2159.907	8%	0.040625	0.000204
	GenProg	630	536	1845.472	20%		

* We separately ran RSRepair and GenProg 100 times on each of the 24 subject programs and only recorded the trials leading to successful repair.

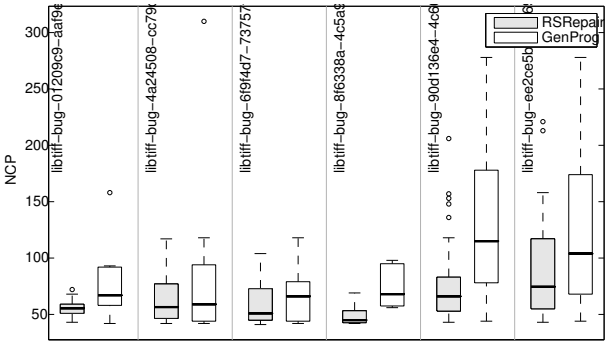
is not logged at all due to the limitation on the maximal NCP.

Given this issue, we conduct the comparison between GenProg and RSRepair as follows. First, we measure how well each repair tool performs using success rate described in Table 2. The intuition behind this measurement is that an effective repair tool, which probably requires smaller NCP when finding a valid patch, should have the higher success rate with the limitation on the maximal NCP in our experiment. Second, if GenProg and RSRepair for one program, such as `libtiff-bug-4a24508-cc79c2b` in Table 2, have the same success rate of 100%, we further compare the repair effectiveness according to NCP measurement using rigorous statistic analysis. Specifically, when repairing one program, GenProg and RSRepair have the completely same patch generation process within the population in the first generation (Recall that RSRepair is the modification version of GenProg with the same patch generation process in the first generation, described in Section 3.3); with the same seed, GenProg and RSRepair should have the same NCP when the NCP value is not bigger than 40, which has been confirmed in our experiment. Given that, if GenProg and RSRepair have the same success rate of 100% for one program, we exclude the repair trials whose NCP are not bigger than 40, the population size of the first generation, because genetic programming used in GenProg just starts to work from the second generation. After the exclusion process, we can check whether genetic programming perform well to guide the patch generation process when it does start to work.

As described in Table 2, in most cases, RSRepair, has the higher (16 of 24) or equal success rate (7 of 24) over GenProg. The higher success rate indicates that RSRepair has the higher repair effectiveness in term of requiring probably fewer NCP in the repair process. The only exception is on `wireshark`, for which GenProg outperforms RSRepair in terms of higher success rate. Having analyzed the repair process, we find that the patched programs are more likely to fail to be compiled in the initial phase of repair process, compared to other programs. Thus, we suspect the reason for the exception to be that GenProg is good at eliminating the bad patches (which fails to be compiled) and can produce more compilation-able candidate patches in the sequent repair process.

To further confirm the advantage of RSRepair, we statistically analyze the NCP on the programs for which GenProg and RSRepair have the success rate of 100%. Statistic results are presented in Figure 1. In the analysis process, we exclude the repair trials whose NCP is not bigger than 40 (note that `lighttpd-bug-2661-2662` is excluded from this table because there exists no repair trials whose NCP is bigger than 40 for this program), and statistically analyze the remaining trials (with the sizes listed in the 3rd list of Figure 1(b)) on NCP. The 4th and 5th column report the mean and median values on NCP, respectively. The 6th column gives the statistical significance by using Mann-Whitney-Wilcoxon test to analyze the difference between the two tools on NCP, with the p value listed in the last column.

Figure 1 suggests that for all the 6 programs RSRepair has the smaller NCP over GenProg even if genetic programming starts to guide the patch generation from the 2th generation. For the last 2 programs in Figure 1(b), the advantage of RSRepair is statistical significance; although there exists no significant difference for the remaining 4 programs due to



(a) Boxplots on NCP of 6 programs.

Program	Approach	Size	Mean	Median	Sig.	p-value
libtiff-bug-012.	RSRepair	10	57	56	0	0.053632
	GenProg	10	77	67		
libtiff-bug-4a2.	RSRepair	8	65	57	0	0.937685
	GenProg	8	93	59		
libtiff-bug-6f9.	RSRepair	17	60	51	0	0.479761
	GenProg	17	68	66		
libtiff-bug-8f6.	RSRepair	5	49	45	0	0.055556
	GenProg	5	75	68		
libtiff-bug-90d.	RSRepair	38	79	66	1	0.000106
	GenProg	38	129	115		
libtiff-bug-ee2.	RSRepair	42	91	75	1	0.031763
	GenProg	42	120	104		

(b) Statistical results.

Figure 1: The statistical comparison on NCP of 6 programs for which GenProg and RSRepair have the success rate 100%.

too small sample sizes (no more than 20 in the "Size" column of Figure 1(b)), RSRepair has the smaller NCP in terms of Mean and Median.

Answer for RQ1: In our experiment, for most programs (23/24), random search used by RSRepair performs better (in terms of requiring fewer patch trials to search a valid patch) than genetic programming used by GenProg, regardless of whether genetic programming really starts to work (see Figure 1) or not. We defer discussing the possible reason to Section 6.

5.2 RQ2

As presented in RQ1, to find a valid patch, GenProg, in most cases, requires not fewer NCP than RSRepair. That is, compared to random search, genetic programming does not bring benefits (in term of fewer NCP in this case) to balance the cost caused by fitness evaluations. Hence, it is not surprising that GenProg, most often, took more time to repair successfully faulty programs, on average, in Table 2. Then, in this subsection we plan to investigate to what extent genetic programming used by GenProg worsens the repair efficiency over random search used by RSRepair.

"Running test cases typically dominated GenProg's runtime" [22], which is also suitable for RSRepair, so we use the measurement of NTCE to compare the repair efficiency between GenProg and RSRepair, which is also consistent with traditional test case prioritization techniques aiming at early finding software bugs with fewer NTCE. An efficient repair tool should find a valid patch with fewer NTCE.

Like the evaluation of NCP in RQ1, it is difficult to compute precisely the value of NTCE due to the stochastic nature of randomized algorithms used by GenProg and RSRepair.

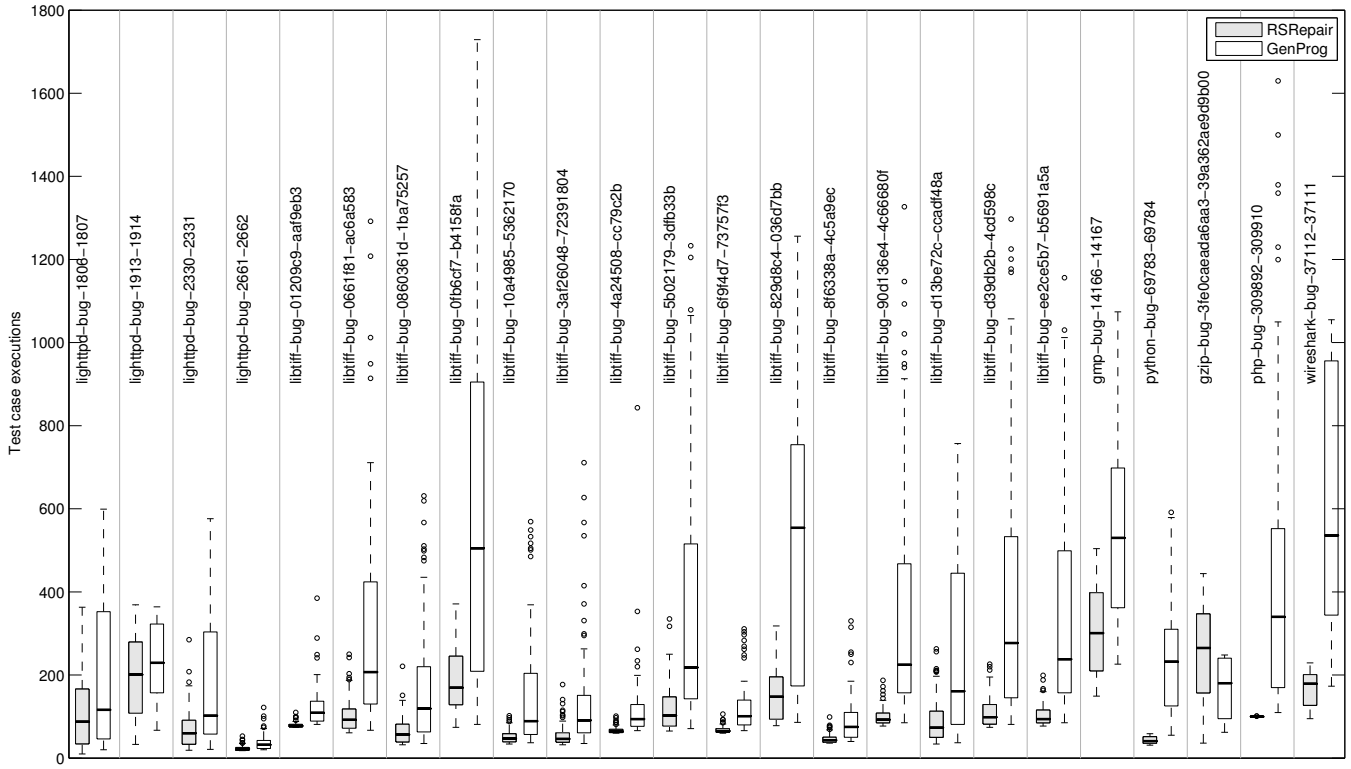


Figure 2: `NTCE` boxplots for experiments. Note that for the `python` and `php`, which give too large `NTCE` over other programs, for the ease of presentation we narrowed down the `NTCE` values with linear scale for the two programs.

As such, we use statistical analysis to compare qualitatively and quantitatively the NTCE between the two tools. We first summarize the NTCE value of each tool using the Mean, because the Mean may perform better over median when the result data are constructed from similarly sized clusters around more than two widely separated values [29]. Then, we use Mann-Whitney-Wilcoxon test [41] and A-test [36], the two nonparametric approaches, to qualitatively and quantitatively analyze the difference significance.

For nonparametric Mann-Whitney-Wilcoxon test, the null hypothesis is that result data from the studied two groups share the same distribution; the alternate hypothesis is that the two group data have different distributions. In this case, we say the difference is statistical significance when we reject the null hypothesis at a 5 percent significance level.

To further assess the difference quantitatively, we use the nonparametric Vargha-Delaney A-test, which is recommended in [4] and was also used in [29, 32], to evaluate the magnitude of the difference by measuring effect size (scientific significance) on NTCE. For A-test, the bigger deviation of A-statistic from the value of 0.5, the greater difference of the two studied groups. In [36] Vargha and Delaney suggest that A-test of greater than 0.64 (or less than 0.36) is indicative of "medium" effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising "large" effect size.

Both Figure 2 and Table 2 suggest that RSRepair has much fewer NTCE, compared to GenProg. Figure 2 presents the boxplots on NTCE when running the two tools to fix bugs existing in the subject programs. Table 2 reports the detailed statistical result on the measurement of NTCE. Obviously, for most programs (23 of 24) in our experiment, NTCE of RSRepair is much smaller than that of GenProg in terms of Mean; the difference can be statistically significance using Mann-Whitney-Wilcoxon test (p-value$<$0.05), in most cases (22 of 23). What is more, it is reasonable to consider that RSRepair reduces the NTCE efficiently in term of often arriving at the promising "large" effect size (A-test$>$0.71 or A-test$<$0.29).

Answer for RQ2: GenProg does not find a valid patch faster than RSRepair. Oppositely, in most cases, GenProg requires much more NTCE to repair faulty programs, leading to the lower repair efficiency than RSRepair.

6. DISCUSSION

Intuitively, genetic programming, which uses fitness values to guide the patch search process, should perform better, at least not worse, compared to random search. Our experiment results, however, suggest that genetic programming used by GenProg, in most cases, does not perform better and even worsens the search process with bigger NCP in the repair process. Then, why does genetic programming, a fitness evaluation directed search, perform worse than a purely random search in our experiment?

We are not surprised for this experimental results. In fact, Andrea Arcuri and Lionel Briand [4, Page 3] has presented their concern to the effectiveness of genetic programming used by GenProg, and considered that genetic programming should be compared against random search to check the ef-

fectiveness, although it is mentioned that for many programs random search is as effective as genetic programming [38].

The reason for the worse repair effectiveness by GenProg may be tracked down from the paper [10]: current fitness functions including that used by GenProg are either overly simplistic or likely to exhibit "all-or-nothing" behavior, and thus are not well correlated with true distance between an individual and the global optimum. In our experiment we found that for GenProg most candidate patches have the same or similar fitness values even if some ones are very close to valid patches. Furthermore, the fact of overwhelming majority of valid patches are generated by the nearest `Mutate` operation, rather than combined action of multi operations in terms of patch evolution, further presents the evidences that genetic programming does not work as well as we think in the patch search process.

Since imprecise fitness functions have the chance of misleading the search process, in our experimental evaluation it should not be surprising that GenProg has the worse repair effectiveness on many programs than RSRepair for which random search is used.

Implication - Before applying some optimization algorithm in search-based research area, we should ensure that the fitness function used by this algorithm should have the ability of computing, at least partly, the distance between candidate solutions and optimal solutions.

7. THREATS TO VALIDITY

The main threats to the validity of our result belong to the internal, external, and construct validity threat categories.

Internal validity threats correspond to the relationship between the independent and dependent variables in the study. One such threat is on the selection of subject programs in our experiment. To save the time resource and simplify the implementation of RSRepair, we excluded the programs for which fixing bugs requires either too many test cases or the modification of multi source files. In fact, the more test cases shipped with subject programs, the more expensive computational resource often required by fitness evaluations, because the fitness values are computed by sampling these test cases according to a fixed rate (10% for GenProg); More expensive evaluations will reduce the benefit brought by genetic programming, because no evidence indicates that fitness evaluation based on more test cases can provide some more accurate guidance for patch search in our experiment. In addition, in the future we will investigate whether genetic programming has the advantage over random search on fixing bugs existing in multi files.

External validity is concerned with generalization. Since we conducted our experiment only on programs each of which comes with one bug, conclusions drawn from our paper may not hold when some subject programs with multi bugs are included in our experiment. To our knowledge, there exists little work on fixing automatically faulty programs shipping with multi bugs, because automated program repair is generally considered to be a hard work [13]. When fixing multi bugs in one program, the success rate can be too low, making little sense for statistic analysis. In addition, in this paper we focus only on the comparison between random search and genetic programming, in our future work we plan to study random search with the comparison on other repair techniques such as [12, 5, 28].

Construct validity threats concern the appropriateness of the evaluation measurement. We first use NCP as the measurement to investigate whether genetic programming has a good work at guiding the patch search process with fewer repair trials. Then, we measure the repair efficiency of genetic programming by the comparison between GenProg and RSRepair according to NTCE. For the analysis to experimental results, rigorously statistical analysis including Mann-Whitney-Wilcoxon test and A-test is used.

8. CONCLUSIONS

As an important optimization algorithm in the area of SBSE, genetic programming has been applied successfully to many fields within the general area of software engineering. Recently, in the paper [40] genetic programming is proposed to fix automatically the general bugs, and a prototype tool called GenProg based on this technique is implemented. After that, general automated program repair has gone from being entirely unheard of to having its own multi-paper sessions, such as "Program Repair" session in ICSE 2013, in many top tier conferences [20], and many researchers justify the advantage of their techniques, such as Par and SemFix, via the comparison with GenProg. Furthermore, affected by GenProg, Par also uses genetic programming to guide the patch search in the way like GenProg.

Although GenProg and Par have presented their promising results, to what extent the patch search process benefits from genetic programming is still unknown. Furthermore, the question of whether the benefit brought by genetic programming can balance the cost caused by fitness evaluations is not addressed. In this paper, we try to investigate the two questions via the performance comparison between genetic programming and random search.

Experiment on 7 programs with 24 versions shipping with real-life field failures suggest that 1) random search used by RSRepair, in most cases (23/24), finds valid patches faster (i.e., requiring fewer patch trials to find a valid patch) over genetic programming used by GenProg, and 2) for most programs (23/24), GenProg requires much more test case executions to find a valid patch over RSRepair, which indicates that the benefit brought by genetic programming cannot balance the cost caused by fitness evaluations, and thus worsens the patch search process. Complete experimental results in this paper are available at:

`http://qiyuhua.github.com/projects/rsrepair/`

Based on experimental results, we challenge the research community to develop novel repair techniques using optimization algorithm to defeat the random search algorithm presented in Algorithm 2 in terms of NCP and NTCE.

9. ACKNOWLEDGMENTS

I would like to acknowledge W. Weimer et al. for their noteworthy work on GenProg, which has largely pushed forward the advance of research area on general automated program repair. This research was supported in part by grants from National Natural Science Foundation of China (Nos. 61379054, and 91318301), and National High Technology Research and Development Program of China (No. 2012AA011201).

10. REFERENCES

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving patches for software repair. In *Genetic and Evolutionary Computation (GECCO)*, pages 1427–1434, 2011.

[2] A. Arcuri. On the automation of fixing software bugs. In *International Conference on Software Engineering (ICSE)*, pages 1003–1006, 2008.

[3] A. Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494 – 3514, 2011.

[4] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.

[5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.

[6] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *International Conference on Automated Software Engineering (ASE)*, pages 550–554, 2009.

[7] V. Debroy and W. Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 65 –74, 2010.

[8] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 113 – 124, 2004.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering (TSE)*, 28(2):159 –182, feb 2002.

[10] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation (GECCO)*, pages 965–972, 2010.

[11] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, 2012.

[12] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to c. In *Computer Aided Verification (CAV)*, pages 358–371, 2006.

[13] M. Harman. Automated patching techniques: the fix is in: technical perspective. *Communications of the ACM*, 53(5):108–108, 2010.

[14] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):1–61, Dec. 2012.

[15] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. *Empirical Software Engineering and Verification*, 7007:1–59, 2012.

[16] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation (PLDI)*, pages 389–400, 2011.

[17] W. Jin and A. Orso. F3: fault localization for field failures. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–223, 2013.

[18] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, pages 802–811, 2013.

[19] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of system models in regression test suite prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 247 –256, 2008.

[20] C. Le Goues. *Automatic program repair using genetic programming*. PhD thesis, University of Virginia, 2013.

[21] 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. In *International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.

[22] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[23] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: a generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54 –72, 2012.

[24] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. In *Genetic and Evolutionary Computation (GECCO)*, 2012.

[25] P. Liu and C. Zhang. Axis: automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering (ICSE)*, pages 299–309, 2012.

[26] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": an essay on the problem statement and the evaluation of automatic software repair. In *International Conference on Software Engineering (ICSE)*, 2014 (To appear).

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

[28] Y. Pei, Y. Wei, C. Furia, M. Nordio, and B. Meyer. Code-based automated program fixing. In *International Conference on Automated Software Engineering (ASE)*, pages 392 –395, 2011.

[29] S. Poulding and J. A. Clark. Efficient software verification: Statistical testing using automated search. *IEEE Transactions on Software Engineering (TSE)*, 36(6):763–777, Nov. 2010.

[30] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 180–189, 2013.

[31] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. Does genetic programming work well on automated program repair? In *International Conference on Computational and Information Sciences (ICCIS)*, pages 1875–1878, 2013.

[32] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on*

Software Testing and Analysis (ISSTA), pages 191–201, 2013.

[33] Y. Qi, X. Mao, Y. Wen, Z. Dai, and B. Gu. More efficient automatic repair of large-scale programs using weak recompilation. *Science China Information Sciences*, 55(12):2785–2799, 2012.

[34] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering (TSE)*, 27(10):929 –948, oct 2001.

[35] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in php applications using string constraint solving. In *International Conference on Software Engineering (ICSE)*, pages 277–287, 2012.

[36] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[37] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, 2010.

[38] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[39] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: models and first results. In *International Conference on Automated Software Engineering (ASE)*, pages 356–366, 2013.

[40] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, pages 364–374, 2009.

[41] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80 – 83, 1945.

[42] X. Xie, T. Y. Chen, F.-c. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):Article 31, 2013.

[43] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab*, 22(4):67–120, 2012.

[44] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei. Time-aware test-case prioritization using integer linear programming. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–224, 2009.