Evolutionary Repair of Faulty Software

Andrea Arcuri

The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK. Email: a.arcuri@cs.bham.ac.uk

Abstract

Testing and fault localization are very expensive software engineering tasks that have been tried to be automated. Although many successful techniques have been designed, the actual change of the code for fixing the discovered faults is still a human-only task. Even in the ideal case in which automated tools could tell us exactly where the location of a fault is, it is not always trivial how to fix the code. In this paper we analyse the possibility of automating the complex task of fixing faults. We propose to model this task as a search problem, and hence to use for example evolutionary algorithms to solve it. We then discuss the potential of this approach and how its current limits can be addressed in the future. This task is extremely challenging and mainly unexplored in literature. Hence, this paper only covers an initial investigation and gives directions for future work. A research prototype called JAFF and a case study are presented to give first validation of this approach.

Keyword: Repair, Fault Localization, Automated Debugging, Genetic Programming, Search Based Software Engineering, Coevolution.

1 Introduction

Software testing is used to reveal the presence of faults in computer programs [50]. Even if no fault is found, testing cannot guarantee that the software is fault-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software development [14]. This is the reason why there has been a lot of effort spent to automate this expensive software engineering task.

Even if an optimal automated system for doing software testing existed, we still need to know *where* the faults are located, that in order to be able to fix them. Automated techniques can help the tester in this task [26, 65, 78].

Although in some cases it is possible to automatically locate the faults, there is still the need to modify the code to remove the faults. *Is it possible to automate the task of fixing faults?* This would be the natural next step if we seek a full automation of software engineering. And it would be particularly helpful in the cases of complex software in which, although the faulty part of code can be identified, it is difficult to provide a patch for the fault. This would also be a step forward to achieve corporate visions like for example IBM's *Autonomic Computing* [40].

There has been work on fixing code automatically (e.g., [63, 61, 68, 25]). Unfortunately, in that work there are heavy constraints on the type of modifications that can be automatically done on the source code. Hence, only limited classes of faults can be addressed. The reason for putting these constraints is that there are infinite ways to do modifications on a program, and checking all of them is impossible.

In this paper we investigate whether it is possible to automatically fix faults in source code without putting any particular restriction on their type. Because the *space* of possible modifications cannot be exhaustively evaluated, we model this task as a *search problem* [30, 18].

The reformulation of software engineering as a search problem (i.e., *Search Based Software Engineering*) has been widely studied in the recent years. Many software engineering tasks have been modelled in this way with successful results (e.g., testing [48]). In many software engineering cases, search algorithms seem to have better performance than more traditional techniques (e.g., [34, 12]). This was our motivation for applying search algorithms on the task of automatically repairing faulty software.

Given as input a faulty program and a set of test cases that reveal the presence of a fault, we want to modify the program to make it able to pass all the given test cases. To decide which modifications to do, we use a search algorithm. Note that we want to correct the source code, and not the state of the computation when it becomes corrupted (as for example in [23]).

The search space of all possible programs is infinite. However, "programmers do not create programs at random" [22]. Therefore, it is reasonable to assume that in most cases the sequences of modifications to repair software would not be very long. This assumption makes the task less difficult.

We discussed the idea of fixing software with search algorithms in a doctoral symposium paper [7], and we then presented very preliminary results on a sorting routine using a limited Lisp-like programming language [11]. In this paper, we present a novel prototype that is able to handle a large sub-set of the Java programming language. The case study is based on realistic Java software.

Different types of search algorithms could be used. In this initial investigation, we consider and compare three search algorithms. We use a random search as baseline. Then we consider a single individual algorithm (i.e., a variant of a Hill Climbing) and a population based algorithm (i.e., Genetic Programming [52]).

To improve the performance of these algorithms, we present a novel search operator that is based on current fault localization techniques. This operator is able to narrow down the search effort to promising sub-areas of the search space. Besides providing an empirical validation, we also theoretically analysed which are the conditions for which this operator is helpful.

The main contributions of this paper are:

- we analyse in detail the task of repairing faulty software in an automatic way, and we propose and describe how to use search algorithms to tackle it.
- we characterise the search space of software repair and we explain for which types of faults our novel approach can scale up.
- to improve the performance, we present a novel search operator. This operator is not limited to the software repairing problem. It can be extended to other applications in which programs are evolved.
- we present a Java prototype called JAFF (Java Automatic Fault Fixer) for validating our automatic approach for repairing faulty software.
- we extend search based software engineering with a new application on an important software engineering problem that has not been addressed before using search algorithms. The use of search algorithms (and in particular evolutionary algorithms) could overcome the difficulties of this problem that have been described in literature.

The paper is organised as follows. Section 2 gives a brief overview of the automation of the debugging activity. Section 3 describes how software repair can be modelled as a search problem.

The analysed search algorithms are described in Section 4. The novel search operator is presented in Section 5. Our research prototype JAFF is presented in Section 6. The case study on which the proposed framework is evaluated follows in Section 7. Section 8 outlines the limitations of repairing software automatically. Future directions of research are discussed in Section 9. Finally, Section 10 concludes the paper.

2 Related Work

Debugging consists of two separated phases. First, we need to *locate* the parts of the code that is responsible for the faults. Then, we need to *repair* the software to make it correct. This means we need to modify the code to fix it. These changes to the code are often called *patch*.

Several different techniques have been proposed to help software developers to debug faulty software. We briefly discuss them. For more details about the early techniques, old surveys were published in 1993 [26] and 1998 [65]. A more updated and comprehensive analysis of the debugging problem can be found in Zeller's book [78] and partially in Jones's thesis [37].

2.1 Fault Localization

One of the first techniques to help to locate faults is *Algorithmic Debugging* [59, 27]. Using a divide-and-conquer strategy, the computation tree is analysed to find which sub-computation is incorrect. This approach has two main limitations. First, an oracle for each sub-computation is required. This is often too expensive to provide. Second, the precision of the technique is too coarse-grained.

A *slice* [70] is a set of code statements that can influence the value of a particular variable at a certain time during the execution of the software. Debugging techniques can exploit these slices to focus on only the parts of the code that can be responsible for the modification of suspicious variables [69, 42, 79].

In *delta debugging* [76, 77, 19, 13] a passing execution is compared against a similar (from the point of view of the execution flow) one that is instead failed. A binary search is done on the memory states of these two executions to narrow down the inspection of suspicious code. The memory states of the failing execution are altered to see whether these alterations remove the failure. This technique is computationally very expensive. Finding two test cases with nearly identical execution path, but one passing and the other failing, can be difficult. If all the provided test cases fail, then this technique cannot be applied.

Software developers often make common mistakes that are practically independent from the semantic of the software. Typical example is opening a stream and then not closing it. Another is sub-classing a method with a new one that has very similar name (doing this has the wrong result of having a new method instead of sub-classing the previous one). Many of these mistakes can be found by statically analysing the source code without running any test case. A set of *bug patterns* can be defined and used to see if a program has any of this known mistakes [33, 56, 66]. One one hand, this technique has the limitation that it can find only faults for which a pattern can be defined. On the other hand, it is a very cheap technique that does not require any test case. It can be easily applied on large real-world software and it can point out many possible sources of faults. This type of static analysis can be improved with data mining techniques applied to real-world source code repositories [71].

In large real-world software, it is common that parts of code result from *copy-and-paste* activities. This has been shown to be very prone to introduce faults, because for example often the developers forget to modify identifiers. If the software does not give any compiling error, then it

is very difficult to find this type of fault. Data mining techniques to identify copy-and-paste faults have been proposed [45].

If the behavioural model of software is available (expressed for example with a finite state machine), one black-box approach is to identify which components of the model are wrongly implemented in the code [74]. Similar to Mutation Testing [22], the idea is to mutate the model with operators that mimic common types of mistakes done by software developers. *Confirming sequences* are then generated from the mutated models and validated against the tested program [74]. The mutated models represent hypothesis about the nature of the faults.

To understand the reason why a fault appears, software developers speculate about the possible reasons. This translates to questions about the code. Tools like *Whyline* [41] automatically present to the user questions about properties of the output, and then they try to give explanations/answers based on the code and the program execution.

Given a set of test cases, coverage criteria can be used as an heuristic to locate faults [55]. One the one hand, parts of code that are executed only by passed test cases cannot be responsible for the faults. On the other hand, code that is executed only by the failing test cases is highly suspicious. Focusing only on this latter type of information gives poor results, because usually most faulty statements are executed by both passing and failing test cases. The *Nearest-Neighbour Queries* technique [55] compares coverage of one passed test case against one failed test case. But in contrast to previous work [5], the two test cases are chosen based on heuristics on their execution flow distance.

Tarantula is a coverage criteria debugging tool that has been quite successful in literature [39, 38, 37]. It is simple to implement, fast to execute and the obtained results are good in average. The idea is to execute all the given test cases, and for each statement s in the code we keep track of how many failed ($failed(s)$) and passed ($passed(s)$) test cases execute them. Using this information, for each statement s we calculate the function $H(s)$:

$$H(s) = 1 - \frac{\frac{passed(s)}{totalpassed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} .$$

For $H(s)$ close to the value 1, it means that the statement s is highly suspicious. On the other hand, for $H(s)$ close to 0 it is likely that s is not responsible for the fault. Using function H, we can rank all the statements in the code. The software testers will hence start to investigate the code from the most suspicious statements.

This function H is just an heuristic. Although it works well in many cases, it does not guarantee to produce good results. Extensions of the Tarantula technique have hence been proposed (e.g., [15, 73]). However, one possible limitation of tools like Tarantula is that they can only identify blocks of suspicious code. Inside a block, they cannot point out which is the particular statement responsible for the fault.

2.2 Software Repair

Compared to fault localization, there has been much less work on software repair. Early attempts to repair software automatically were done by Stumptner and Wotowa [63, 64]. Given a *fault model* that represents a particular type of fault (e.g., wrong left-hand side assignments), then an exhaustive enumeration of all possible programs were made for that fault model.

Buccafurri *et al.* investigated to extend *model checking* with artificial intelligence techniques to repair software [16]. Formal specifications in *computational tree logic* for concurrent systems expressed in Kripke models were considered. Model checking was extended with abductive reasoning and heuristics to narrow down the search space. This line of research has then been extended by for example Zhang and Ding [80].

The use of model checking for software repair has been also studied for *linear temporal logic* specifications by Jobstmann *et al.* [36, 61]. The repair task is modelled as a *game*. Although heuristics to narrow down the search space are presented, the exponential nature of model checking still remains. Similar work has been done for boolean programs [28, 58].

Weimer presented an algorithm based on model checking to repair safety-policy violations [68]. Wang and Cheng considered graphical state transition specifications, and they presented an heuristic to reduce the search space of repairs for model state graphs [67]. He and Gupta presented an algorithm for software repair that, given a formal specification, is based on analysing program execution traces and it uses hypotheses on program states [31]. There has been also work on software repair with artificial intelligence techniques based on proof solving [25, 24].

Although heuristics for decreasing the search space have been proposed, the applicability of these techniques is constrained by their "exhaustive search" nature.

3 Software Repair as a Search Problem

Given as input the code of a faulty program and a set of test cases, the goal is to modify the code to obtain a new version that is able to pass all of these test cases. We model this problem as a *search problem* [30, 18]. Given a set of operators to modify the code, we search for a sequence of modifications that leads to a faultless version.

In a search problem, the number of possible candidate solutions is too high to enumerate all of them. The set of all possible candidate solutions is called the *search space*. A search algorithm can only analyse/evaluate a subset of the search space. The choice of which candidate solutions to evaluate depends on heuristics based on the solutions evaluated so far.

The candidate solutions that solve the problem are called *global optima*. There can be more than one global optimum in the search space. A *local optimum* is a solution that is the best among its neighbourhood in the search space. The neighbourhood definition (i.e., the distance among solutions) is problem and representation dependent. For example, in a bit-string representation, all the solutions that differ of only one bit can be considered as neighbours.

In this section, we first define which search operators can be used. Then we present an heuristic (i.e., the *fitness function*) to guide the search for the repair. Finally, we analyse the properties of the search space.

3.1 Search Operators

In the search problem we are analysing in this paper, search operators consist of modifications of the source code. A modification can be for example removing a statement or modifying the value of a constant.

Given a set of operators, it is important that, for each possible pair of programs, a sequence of operations should exist to transform one of these programs into the other. If this property holds, then each possible fault related to the source code can be addressed. In fact, there would be a sequence of operations to transform the faulty program in a correct one. Unfortunately, the fact that this sequence exists does not mean that it is easy to find it.

Depending on the context, a search operator could make the modified program not possible to be compiled. To avoid this problem, search operators should be based on the grammar of the used programming language.

Valid modifications could lead to programs that never stop. This could happen for example if the predicate in a **while** statement is replaced by **true** constant. A way to address this problem is to give a time limit for the execution of the programs on the test cases. The time limit could be heuristically chosen based on the execution time of the faulty program.

3.2 Fitness Function

The fitness function of a program P is based on how many test cases in the given set T are passed. If it is possible to define a degree for how badly a test case is failed, then this information can be exploited in the fitness function for making the search space smoother.

For example, for each assertion in the test cases, we can calculate a distance d. If an assertion a is passed, then $d(a) = 0$. In case in which a numeric value v is compared against an expected value r, then we can use $d(a) = |v - r|$. In case of booleans, we have $d(a) = 1$ if they do not match. For comparison of string objects, we can calculate for example their edit distance. For other types of predicates, different heuristics could be designed.

Given $T(P)$ the set of assertions in the test cases after executing P, the semantic fitness to minimise is defined as:

$$f_s(P) = \omega(\sum_{a \in T(P)} d(a)) ,$$

where ω is any normalising function in $[0,1]$. In case in which there is any error (e.g., the program P cannot be compiled or its execution exceeds the time limit), then a death penalty is applied (i.e, $f_s(P) = 1$).

A sequence of modifications could make the input program very large. This is a problem that in literature is called *bloat* [46]. Common techniques to fight it are penalising the size of the programs and putting constraints on their maximum allowed size. For simplicity, for the rest of the paper we define the size as the number of nodes in the abstract syntax tree of the program.

For contrasting bloat, in the fitness function we penalise long programs. However, we also need to penalise too short programs. In fact, the original assumption [22] is that the faulty program is not too structurally far from a global optimum. Although a very long program can still be correct (e.g., it might contain a lot of junk code that is not executed), that is not true in general for short programs. Given $N(P)$ the number of the nodes of P, P_{or} the original faulty program, and given the constant δ (e.g., $\delta = 10$), then the node penalisation is defined as:

$$p(P) = \begin{cases} \omega(N(P)) & \text{if } N(P) > N(P_{or}) + \delta \ , \\ 1 & \text{if } N(P) < N(P_{or}) - \delta \ , \\ 0 & \text{otherwise } . \end{cases}$$

Finally, the employed fitness function to minimise is:

$$f(P) = \gamma f_s(P) + p(P) , \tag{1}$$

where γ is a weight to make the semantic score more important (for example we could choose the value $\gamma = 128$, see [46]).

3.3 Search Space

Enumerating the entire space of programs is not feasible because it is infinite. Even if we put constraints to the size of the programs we are looking for, it is still an extremely large search space [43]. However, in the case of fixing faults, we assume that the faulty program is not too distant from a global optimum [22], i.e. with only few modifications we can sample a correct program.

If we limit our search in this neighbourhood of modifications, we would have a search space that is roughly:

$$S = (mN)^k , \tag{2}$$

where N is the number of nodes of the faulty program, m is the number of different modifications that can be done on each node, and finally k is the minimum number of modifications

for reaching a global optimum. Note that Equation 2 is a loose simplification, because the three variables are correlated: the size of the program can change after a code modification, and not all the modifications can be done on all the possible nodes because the modifications might depend on the type of the nodes, etc. At any rate, Equation 2 gives an idea of the size of the restricted search space.

What type of faults can we expect in real-world software? Empirical analyses of large real-world software show that nearly 10% of the faults can be fixed with only one line of code modification [53, 21]). Half of the faults can be corrected by changing up to 10 lines. Most of the faults (i.e., up to 95%) can be fixed with no more than 50 line modifications. Therefore, in many cases the value of needed modifications k is low (e.g., between 1 and 10).

Although the search space increases polynomially in N with exponent k that is supposed to be low , it is still an extremely large search space if we consider programs of millions or even just thousands of lines of code. A first consequence of Equation 2 is that fixing faults in entire software is not feasible, the search space is simply too large even for just evaluating the closest neighbour solutions. However, we can restrict our approach to units of computation (e.g., single functions and classes), in the same way as unit testing is done. In other words, we can use a sort of *unit fault fixing*, in which modules are tested with unit tests and, if they are failed, our framework could be used to fix these units.

Even if we restrict the scope of our application to units of computation, the variable m is still problematic. When we insert new code, for real-world languages (e.g., Java) there might be many possible different instructions (e.g, loops and switches). Although the number of these instructions is a constant depending on the language, that is not true for the possible objects and static functions that can be used (they are infinite, and already too many if we restrict for example to just the Java API). In the search, we could just ignore the classes that are not used in the software under test (e.g., in a sorting algorithm we would not try to add a TCP socket), but that limits the scope of our approach (although it could be argued that it would have little impact on real-world faults).

The empirical study in [21] shows that half of the faults can be fixed by doing modifications in a single method. For simplicity, we call it the *single method* assumption. If we focus on this type of faults, the search space can be further decreased. In fact, we can make a different search for each method that could be the cause of the fault. For each search, only the code of the considered method can be modified. The assumption is that the functions that are called inside the target method are considered correct. Because these searches are independent, they can be run in parallel.

Let l be the average length of the functions. Depending on the programming style, we can reasonably estimate that it would be something like $1 \leq l \leq 100$. Given N the size of the software, we would roughly have N/l methods. A loose estimation of the search space size would hence be:

$$S = (N/l)(ml)^k = m^k l^{k-1} N \ . \tag{3}$$

If we compare Equation 2 with Equation 3, we can see that the single method assumption reduces the search space by the factor $(N/l)^{k-1}$.

Scalability is an important issue that requires to be addressed. At the increase of the size N of the software, we want to know how much more difficult it would be to repair it. With no assumption on the type of faults, the search space is large $\Theta(e^N)$. An exponential search space would make already difficult the repair of tiny toy software. Under the assumption that software is not coded at random, by Equation 2 the search space would be large $\Theta(N^k)$. A polynomial search space could make possible to handle faults that require only few modifications even in large systems, because for example 10% of faults can be repaired with only one line modification. Under the single method assumption, by Equation 3 the search space would be linear $\Theta(N)$.

We cannot expect to be able to automatically repair all the types of faults. However, many real-world faults adhere to some specific assumptions. If these assumptions are exploited, the search space can be drastically reduced.

4 Analysed Search Algorithms

Different search algorithms exist in literature, but none of them is the best on all possible problems [72]. However, for any particular class of problems, there can be search algorithms that perform better on that class. This is one reason why different search algorithms need to be compared and analysed when a new class of problems is tackled.

To our best knowledge, we are aware of no previous work on applying search techniques to software repair. Therefore, in this paper we compare three different types of search algorithm. We use a random search as a baseline to evaluate the other techniques. We then compare a single individual algorithm (i.e., a variant of Hill Climbing) against a search algorithm that uses populations (i.e., Genetic Programming).

These three algorithms are only a small sample of all possible search algorithms used in literature. Other algorithms could be more suited for the task of repairing software. However, these three search algorithms can give first useful validation of the approach of modelling the task of fixing faulty software as a search problem.

To make the comparison more fair, these three algorithms use the same set of program modifications and the same fitness function. Because the employed set of code modifications come from the Genetic Programming literature, without loss of generalisation we call them *mutations*.

4.1 Search Operators

There are several types of Genetic Programming mutations in literature. In our framework, for the the mutation operators we use the one implemented the library ECJ [2]. The choice of the mutation operators has a drastic effect on the final performance. However, a discussion about the proper choice of mutation operators is postponed to Section 9.

Given k a random node, we use the following mutation operators:

- *Point Mutation*: the sub-tree rooted at k is replaced with a new random sub-tree with bounded depth.
- *OneNode Mutation*: k is replaced by a random node with same constraints and arity.
- *AllNodes Mutation*: each node of the sub-tree of k is randomly replaced with a new node, but with same type constraints and arity.
- *Demote Mutation*: a new node m is inserted between k and the parent of k. Hence, k becomes a child of m. The other children of m will be random terminals.
- *Promote Mutation*: the sub-tree rooted at the parent of k will be replaced by the sub-tree rooted in k.
- *Swap Mutation*: two children of k are randomly chosen. The sub-trees rooted at these two nodes are swapped.

In the case of a mutation event, 1 out of these 6 mutation operators is chosen with uniform probability. However, these mutation operators can be too destructive (e.g., a point mutation on the root would generate a completely new program). This is a serious problem, because if the original faulty program does not have a good fitness, then we could quickly converge to very small

and unfit programs (this because smaller programs are rewarded for contrasting bloat). Hence, we changed them such that the number of modified nodes is upper bounded by a relatively small constant (e.g., point mutation can only be applied on sub-trees with at most a depth of 4).

Given a set of mutations, it is important that, for each possible pair of programs, a sequence of mutations should exist to transform one of these programs into the other. If this property holds, then each possible fault related to the source code can be addressed. In fact, there would be a sequence of mutations to transform the faulty program in a correct one. Unfortunately, the fact that this sequence exist does not mean that it is easy to find. Note that it is just enough to have a point mutation to satisfy this property.

When programs are mutated, the search operators can break the syntax of the used language. To avoid this problem, in Strongly Typed Genetic Programming [49] each node has a type and a set of constraints regarding the types of its children. The search operators are such that once applied the constraints still remain satisfied.

Unfortunately, in the case of real-world programming languages, node constraints might depend on their context besides the direct parents and children. For example, the Java compiler checks whether all statements are reachable, and that might depend on the feasibility of the predicates of the previous branches. One way to address this problem would be to use a more general system for defining the constraints (but that might be very challenging to implement). Other option would be to use a sort of "post-processing" for the mapping from genotype to phenotype (i.e., using repairing rules). Finally, syntactically incorrect programs might just have a fitness penalisation. All of these techniques have both benefits and negative sides, and they are common in constraint handling for optimisation problems.

4.2 Random Search

A random program is extremely unlikely that would be a correct implementation of any non-trivial software. We hence consider of little interest comparing search algorithms against a pure random search.

The Random Search (RS) we analyse is based on random mutations of the input program. Let M be the maximum number of allowed mutations. The pseudo-code of the algorithm would be:

1. Check if stopping criterion is satisfied.
2. Randomly choose m in $1 \leq m \leq M$.
3. Apply m mutations to a new copy P' of input program P.
4. If P' is global optimum, return P', otherwise go back to step 1.

4.3 Hill Climbing

Hill Climbing (HC) is a search algorithm that belongs to the class of *local search* algorithms. That means that given a starting point I_0, it looks at neighbour solutions $Z(I_0)$ that are "near" to I_0. If a better solution $I' \in Z(I_0)$ exists, then the next point I_1 will be I'. The same procedure of looking at the neighbour solutions is then repeated on I_1, until a final point I_i is reached, where $\forall I' \in Z(I_i) : f(I') \geq f(I_i)$, assuming we want to minimise function f. This means that no neighbour solution is better, and the algorithm is said to be stuck in either a local or global optimum. If I_i is not a global optimum, then HC can restart from a new different point I_0.

HC is not a single specific algorithm, but a family of algorithms. In fact, we need to define how the neighbourhood Z is generated, the strategy ψ for visiting Z, and finally how to do the restarts.

Applying a common HC in software repair is problematic. In fact, starting the search from a random program (i.e., a random I_0) would be equivalent to the task of generating programs from scratch. Using search algorithms for this latter task is very difficult [10, 54]. Instead of starting from a random program, we can start from the input program P. The neighbour Z would be defined by the mutation operators. However, there would be still the problem of how doing the restarts once HC is stuck in a global optimum.

In our variant of HC, we do not use any restart. We use a dynamic Z that is large enough for not being completely explored during the search. Like for RS, we apply a random number m of mutations each time we sample a new program. Note that this approach makes the algorithms similar to a (1+1) Evolutionary Algorithm. For simplicity, for this variant of HC we just use the name HC instead of inventing a new name.

The pseudo-code of the algorithm would be:

1. P is a copy of the input program.
2. Check if stopping criterion is satisfied.
3. Randomly choose m in $1 \leq m \leq M$.
4. Apply m mutations to a new copy P' of P.
5. If $f(P') < f(P)$, then $P = P'$
6. If P is global optimum, return P, otherwise go back to step 2.

4.4 Genetic Programming

Genetic Programming (GP) [52] is a paradigm for evolving programs to solve for example machine learning tasks. A genetic program is often represented as a tree, in which each node is a function whose inputs are the children of that node. A population of programs is maintained at each generation, where individuals are chosen to fill the next population accordingly to a problem specific fitness function. The programs are modified at each generation by evolutionary inspired operators like crossover and mutation.

Given a set of test cases, if we use GP in its common way, it will be quite difficult to evolve a correct program from scratch [54, 10]. The problem is that we aim to a faultless program that should overfit its training data, because even if one test case is failed, we would know for sure that the program is still faulty.

Because developers do not implement software at random [22], we can exploit the input faulty program for the seeding of the initial population. For example, all the individuals in the first population might be copies of the input program.

Starting from a solution that is close to a global optimum has an impact on the types of the search operations that should be used. For example, in many GP applications crossover is preferred over mutation. But in our case it is the opposite, mostly because for the lack of diversity in the population.

5 Novel Search Operator

To improve the performance of search algorithms, domain knowledge needs to be exploited. In the case of repairing software, if we have some reasons to believe that a fault is generated by a particular area of the code, we can concentrate our search effort in that area.

One way to make this decision is to use fault localization techniques, like for example Tarantula (see Section 2.1). On one hand, the more accurate the technique is the better results we can expect to obtain. On the other hand, because we need to use this fault localization technique each time we need to modify a program, we need that it should be quick to compute.

Given a fault localization technique that ranks the suspiciousness of the statements in the code, let t be the number of nodes (in the syntax tree) that are related to the fault. Let s be the number of nodes that are given the same rank as these t nodes, whereas l is the number of nodes that have lower rank and h is the number of nodes that have higher rank. The total number of nodes in the program is given by $l+s+t+h$. An ideal fault localization technique would have $h=0$ and s as small as possible (remember that tools like Tarantula rank entire blocks of code, so in most cases $s>0$).

The novel search operators we propose is quite simple. When we mutate a program and we need to choose a random node, we randomly pick up n nodes. Then, we apply a tournament selection based on the rank. In other words, we apply the mutation only on the node that has higher rank among these n nodes (i.e., n is the node bias). In case there are several nodes with this highest rank, we randomly choose one of them.

Let δ be the probability of choosing one of the t incriminated nodes in a tournament of size n. The following are obvious properties of δ:

$$\delta(1) = \frac{t}{l+s+t+h} \ ,$$

$$\lim_{n\to\infty} \delta(n) = \begin{cases} 0 & \text{if } h>0 \ , \\ \frac{t}{t+s} & \text{otherwise} \ . \end{cases}$$

For $n=1$, we are actually not using the novel operator. If we are using an ideal fault localization technique (i.e., h is always equal to 0), then it is best to use a tournament size as large as we can. Unfortunately, we cannot assume to have such an ideal tool. For large values of n, we would hence expect a decrease in performance. But, for which values of n can we obtain better results even if $h>0$? In other words, the novel operator is useful only if $\delta(n)>\delta(1)$ even for $h>0$. Of course, the more accurate the fault localization technique is, the better result we can expect. But we want to get better results even if it does not rank perfectly. To answer to this research question, we need to formally calculate the probability δ:

$$\delta(n) = \left(1-\left(1-\frac{t}{l+s+t}\right)^{n}\right)\left(1-\frac{h}{l+s+t+h}\right)^{n} \sum_{i=1}^{n}\sum_{j=0}^{n-i} \left(\left(\frac{i}{i+j}\right)\binom{n}{i}\binom{n-i}{j}\left(\frac{l^{n-i-j}s^{j}t^{i}}{(l+s+t)^{n}-(l+s)^{n}}\right)\right) \quad (4)$$

Formal proof of this Equation 4 is provided in Appendix A. If we are not sure of the quality of the employed fault localization tool, a conservative option would be to use a small n. The smallest value is $n=2$. Under which conditions $\delta(2)$ is better than $\delta(1)$? Their ratio is:

$$\frac{\delta(2)}{\delta(1)} = \frac{l+(l+s+t)}{h+(l+s+t)} \ .$$

Hence, we get an improvement if just $l>h$. Note that the fact of having an improvement is independent of the values s and t. This means that even in case of a high error rate, our novel search operator still gives better results.

Figure 1 shows a 3D plot of the probability $\delta(n)$ when $l=10$, $s=1$, $t=1$, $1\leq n\leq 20$ and $0\leq h\leq 19$. Even for $h>0$, there are values of n for which $\delta(n)$ increases up to a peak that is higher than $\delta(1)$, but then it decreases.

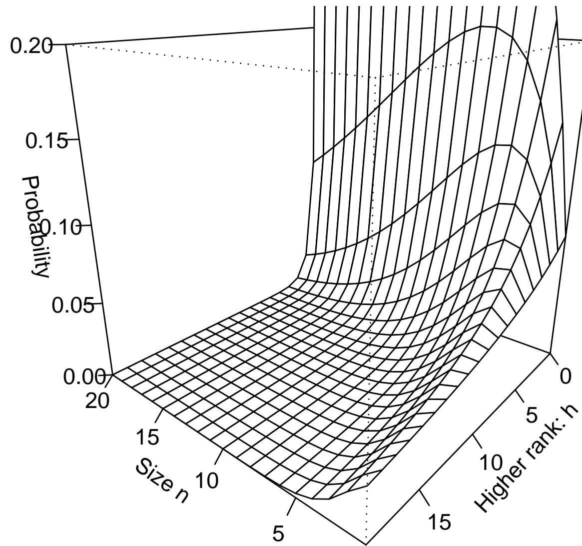

Figure 1: Values of probability $\delta(n)$ when $l = 10$, $s = 1$, $t = 1$, $1 \leq n \leq 20$ and $0 \leq h \leq 19$.

We are using this novel search operator for the task of repairing software. However, it can be easily extended for example to GP applications in which there is a control flow in the evolved programs (i.e., if the employed language uses conditional statements and/or loops).

6 JAFF: Java Automatic Fault Fixer

6.1 The Framework

To validate the approach of automatically repairing faulty software with search algorithms, we developed a framework called JAFF (Java Automatic Fault Fixer). JAFF has been written in Java, and it supports the repair of software written in a sub-set of the Java programming language. Input to the framework is a Java program and a set of test cases. Test cases are written as JUnit tests [3]
.

Input programs are automatically parsed and for each method a configuration file is generated to use the framework. The test cases need to be instrumented to make it possible to inform the framework of their outputs and for handling exceptions. This instrumentation can be automatically done, but our current prototype does not have this feature yet. At the current moment, the framework can be run only by command line. No graphical interface has been developed yet.

For the development of the search algorithms, we used the open source library ECJ [2]. It is a powerful library for evolutionary computation in Java. Unfortunately, the common price of using a general library is loss of efficiency. Moreover, we needed to extend its node constraint system to handle some basic polymorphic types.

Programs are transformed in their syntax trees. Search operators (i.e., the mutations) are applied to the syntax trees. Each time a tree is mutated, to evaluate its fitness value we convert it back to Java source code and then we compile it. This compiled code is run against the instrumented JUnit tests. The compilation of programs is done with the Javassist tool [17].

Our tool features the novel search operator described in Section 5. In particular, for the fault localization technique we use Tarantula [38].

6.2 Technical Problems

Developing a tool for automatically repairing software is a challenging task. Many technical problems need to be addressed. We hence describe them, and we specify which of them our tool does not properly handle yet.

Because in each search a large amount of programs are sampled and tested, the efficiency of how the programs are modified and executed is critical. Unfortunately, this efficiency depends on the programming language. The following discussion about Java might not apply to other languages (e.g., C++).

In many GP systems, programs are executed by interpretation. In other words, these GP systems also provide a virtual machine for executing the GP trees without the need to generate machine code. This approach works well for simple languages and when the programs are not computationally expensive.

Because rewriting a Java virtual machine is not an affordable option, we chose to compile our GP trees directly in Java bytecode, and then to execute them inside a Java virtual machine. Unless the execution of the test cases is comparatively expensive, the efficiency of this compilation process is very important.

A wide set of problems do need to be addressed. Some of them are similar to the problems that are faced in Mutation Testing (see for example [35]).

- We need to compile the code at each fitness evaluation, hence for efficiency we should not touch the file system. In other words, we should not compile a code and then save the results in a file and load/execute it. This means we need to compile directly in memory. For doing this we should not call an external compiler, because it would run on a separate process, and modifying a compiler for making it communicating by process signals (for example) would be too complicated and inefficient.

- Running each program on a different process would be too expensive, particularly in Java because we would need to start a new virtual machine at each fitness evaluation. However, in Java loading and running the programs in the same virtual machine of the framework is not a particular problem, as long as the exceptions are properly handled (some issues would still be there as for example instructions like **System.exit(1)**). This would not be easy to do in languages like C, in which avoiding pointer operations corrupting the state of the framework would not be straightforward.

- We might want to modify the code of a method, but that might be inside a very large class. Hence, we need to be able to recompile single methods and leaving untouched the rest of their classes.

- When we obtain a new version of a class, for executing it we need to load it in the virtual machine. However, all the other classes that depend on it (like for example the classes containing the test cases) would still point on the old implementation. Although it is possible to reload all of them with a different class loader, it would be more efficient to do the modifications directly on the loaded old version (in fact there might be too many test cases

that would need to be reloaded). Unfortunately, this functionality is not directly provided in Java. Another option would be to instrument the software such that to support its dynamic updating (e.g., [51]), but doing this would introduce another set of limitations and problems (see [51] for more details).

- A modified method might enter in an infinite loop. To avoid that, its code can be instrumented such that each loop and recursive call is checked against a global counter. The upper limit of this counter might be estimated on the execution of the faulty program on the set of test cases. Unfortunately, in some cases doing that is not enough. The method could corrupt the internal state of its class and then calling other methods that will loop forever because the state is corrupted. On one hand, we can instrument all the code that can be executed by the analysed method. On the other hand, we can run each program on a separate thread, and then giving a time limit to their execution. Executing new threads and synchronising them might be expensive (remember that we would need to do it at each fitness evaluation), but it would have a lower cost than the compilation of the program and the run of its test cases. Moreover, putting time constraints would help to penalise evolved code that becomes too inefficient.

- We do search operations on the source code and then we compile it. For efficiency, another option would be to directly modify the bytecode. Because reverse engineering on bytecode (we would need it for showing the results to the user) is nowadays not particularly difficult (particularly if no obfuscation technique is employed), we will investigate this option in the future (although it would require quite a lot of re-factoring of our framework). Moreover, it could make easier the implementation of the constraint system for the GP engine.

- Implementing a correct constraint system for the complete Java language is a very time consuming task. Although our current prototype has a sophisticated constraint system, it is possible that legal mutations of syntax trees in our system would end up in programs that cannot be compiled in Java. To mitigate the problem of evolved programs that do not compile, we use a simple post-processing when we translate GP trees back to Java programs. In particular, in the translation we ignore all the statements that come in the same block after **return**, **break** and **continue** commands (because they will result in non-reachable statement compiling errors). In the other cases in which the programs have compiling errors, we just give a death penalty in their fitness value.

To compile Java code we use Javassist [17]. It allows us to compile code directly in memory, and to update single methods directly in the virtual machine. Although its use solves many of the technical issues described before, it unfortunately introduces new ones related to the features of the Java language that are supported. At any rate, such limitations might be solved in its future releases. The description of following limitations are taken from the Javassist documentation:

- The new syntax introduced by J2SE 5.0 (including enums and generics) has not been supported.

- Array initializers, a comma-separated list of expressions enclosed by braces { and }, are not available unless the array dimension is one.

- Inner classes or anonymous classes are not supported.

- Labeled continue and break statements are not supported.

- The compiler does not correctly implement the Java method dispatch algorithm. The compiler may confuse if methods defined in a class have the same name but take different parameter lists.

Table 1: Employed primitives. They are grouped by type. Their name is consistent with their semantics. When the semantics of the primitives could be ambiguous, a short description is provided. More than one primitive can have same name (but different arity and/or constraints).

Type	Name	Description
Arithmetic	$+$, $-$, $*$, $/$, $\%$, $<<$, $>>$, $\&$, $\sim$	Typical arithmetic operators.
Unary Modification	$++$, $--$	Post and pre unary increment/decrement.
Boolean	$\&\&$, $\vert\vert$, $!$, $>$, $\geq$, $==$, $!=$,$<$, $\leq$	Typical operators to handle boolean predicates.
Constant	$true$, $false$, $null$, 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9	Boolean constants, null object and ten integer constants.
Statement	for , $while$, $break$, $continue$, if , $switch$, $case$, $empty_case$, $skip$, $empty_expression$, $conditional_expression$, $=$, $\vert=$, $cast$	Typical statements. $skip$ is the empty statement.
Sequence	$statement_sequence$, $case_sequence$, $expression_sequence$, $update_sequence$	Used to concatenate statements, cases in switch commands, etc.
Variable	$read_variable$, int_tmp , $array_tmp$	Primitive to read variables. Two support variables are used, one of integer type and the other is an array of integers. Based on the program to improve, there are also primitives representing the inputs and the local variables.
Array	$read_array$, new_array , $length$	Primitives to handle arrays.
Primitive Type	int , $boolean$, $char$, $type_wrapper$	Used for casting variables and for defining the type of new generated arrays.
Class	new , $string$	Primitives to use objects. Based on the program to improve, there also primitives for all the types of used objects to handle them (e.g., for calling their methods).

6.3 Supported Language

Our current prototype JAFF does not support yet the entire Java programming language. At any rate, the supported subset is large enough to carry out experiments on realistic software.

Before applying search operations for modifying code, the Java programs are translated in a syntax tree. These trees are composed of nodes. Each node is either a leaf (i.e., no children), or a function (i.e., at least one child node). Note that there is a difference between a function in the tree and a method in the Java language. For example, in $1 + 3$ the operator $+$ is considered in the trees as a node function that takes two sub-trees as input. We used 72 different nodes for representing a large subset of the Java programming language (see Table 1). For each different program, we also added nodes regarding the local variables and method calls. Depending on the program, different types of return statements are used.

The constraint system consists of 12 basic node types and 5 polymorphic types. For the functions and the leaves, there are 44 different types of constraints. For each program, we added as well the constraints regarding local variables and method calls. Although the constraint system is quite accurate, it does not completely represent yet all the possible constraints in the employed subset of the Java language (i.e., a program that satisfies these constraints would not be necessarily compilable in Java).

7 Case Study

7.1 Faulty Programs

To validate our novel prototype JAFF, we applied it to a case study. Ideally, validation should be done on real-world faults. They can be obtained from open source software repositories [21], in which all the versions of the software are stored. Hence, in many cases, it is possible to see which faults are in a particular version, and then checking how they have been fixed in the following versions.

Using real-world software was not possible because our current prototype does not support the entire Java language specification yet. Furthermore, research on software repair is still at an early stage, and in this paper we want to give directions on how to apply search algorithms to tackle it. We are aware of the many difficulties of this task and that more research is still required.

Nevertheless, faults in software in open repositories show only one side of the problem. In general, that type of faults are discovered only after the software has been used for a while. In many

cases, these faults are related to special circumstances that were not considered by the original programmer. But what about the faults that are fixed before submitting a new version of the software? It is not uncommon that a developer write a code, test it, it does not work, and then (s)he spends minutes/hours to fix it, and finally (s)he submits the code only when all the test cases are passed.

This latter type of faults does not usually appear in open source repositories, and it includes for example simple errors (e.g., a $+$ instead of a $-$) that make the program fail on each input. Depending on the complexity of the software, these faults are not necessarily easy to fix.

Given a set of programs written in a subset of Java that our prototype can handle, we had the problem of how to seed faults in them. Doing that by hand would likely end up in a biased case study. We chose to seed the programs with a Mutation Testing [22] tool called muJava [47]. We did it because this type of mutants are actually representative of a range of real errors that developers can occasionally do [6]. Mutation testing has been shown to be very effective to evaluate the quality of test cases. In evaluating test cases, the mutants are more close to real-world faults than faults generated by hand [6]. Furthermore, applying more mutations on the same program gives us a case study with different degrees of difficulty (we can reasonably assume that programs that are mutated more are likely more difficult to fix).

To make our case study as little biased as possible, we chose a set of search operators that is not specialised in fixing faults generated by mutation testing tools. In particular, we just chose all the possible mutation operators in ECJ [2] (we described them in Section 4.1), and we gave the same probability to all of them without any particular bias to any of them. Therefore, our framework can be used to any code level type of faults, although our case study is limited to mutation testing faults.

It might be argued that limiting the case study to faults generated by mutation testing tools is too restrictive. However, repairing software in an automatic way is a very complex task, and a lot of research is still required for having stronger results. Nevertheless, showing its feasibility on a sub-set of realistic types of faults is important to support the first steps in this research field. For example, it has been estimated that 10% of all faults can be fixed with only one line modification [53, 21]).

We test the framework on 7 different Java programs. Among them, 2 are classically used in the literature of software testing of procedural programs: *Triangle Classification* [50, 8] and *Remainder* [62, 57]. *TreeMap* and *Vector* are common in literature of testing object-oriented software [34, 12]. Sorting algorithms as for example *Bubble Sort* [20] are commonly used in literature of GP (e.g., [4]). Finally, *Phase of Moon* is adapted from Apache Ant [1].

Table 2 summarises their properties. Apart from TreeMap and Vector, all the other programs are static functions. Regarding TreeMap, we carried out experiments only on its method *put*. For Vector, we consider the methods *insertElementAt* and *removeElementAt*.

It can be argued that the size of these functions is small, i.e. the longest has only 41 lines of code. However, in this first application of search algorithms on the software repairing task we limit our self to the single method assumption, which in some empirical studies it has been estimated to be valid for half of real-world faults (see Section 3.3).

Note that for TreeMap and Vector there are many private methods that are used inside the analysed three methods, but they are assumed to be correct during the search. Because a priori we would not know which of these methods is faulty, we should do a search in each of these methods in parallel (see Section 3.3). If we ignore the case that a modification in a correct method does fix the fault generated by the faulty method that calls it, we do not need to run these parallel searches in our experiments. Given t steps needed to fix a fault in one of these faulty methods, to estimated the required computational effort we can just multiply this t by the number of involved methods (under the assumption we are not focusing the search in any of them). Note that parallel searches

Table 2: For each program in the case study, it is shown its number of lines of code (LOC), the number of nodes in its syntax tree representation, and finally the type of its inputs.

Name	LOC	GP Nodes	Input
Phase of Moon	8	50	int,int
Vector.insertElementAt	9	53	Object,int
Bubble Sort	10	56	int[]
Vector.removeElementAt	16	62	int
Triangle Classification	25	101	int,int,int
TreeMap.put	36	134	Object,Object
Remainder	41	160	int,int

Table 3: Number of passed assertions in the faulty versions of the programs.

Program	V1	V2	V3	V4	V5
Phase of Moon	0	0	0	0	11
Vector.insertElementAt	262	8	8	8	8
Bubble Sort	32	32	34	32	44
Vector.removeElementAt	180	251	233	250	250
Triangle Classification	86	60	46	44	27
TreeMap.put	86	24	24	24	38
Remainder	83	76	63	55	55

can be done even on a single CPU machine (the parallelism would be simply simulated).

For each program, we generated a set of 100 test cases. Each test case consists of one assert statement, but for TreeMap and Vector there is an assert statement for each insertion operation in the test sequence, and a final assertion on the container size (i.e., around four/five assertions for test case). We call *valid* all the evolved programs that are able to pass all of their 100 test cases. For more validation, we also generated for each program a separated and independent set of 1000 test cases, which are not used during the search. An evolved program that is able to pass all these 1000 test cases is called *robust*. Note that a robust program is not necessarily correct.

All the test cases have been automatically generated based on the fulfilment of the branch coverage criterion. For simplicity, in our experiments we used test generators specialised for our case study. To apply our framework to a new testing problem, the user has to provide the test cases.

For each program, we generated 5 faulty versions. The first is done by a single mutation with muJava, the second by applying a new mutation on the first faulty version (i.e., 2 mutations in total), and so on until the $5th$ that has been generated by applying a new mutation on the $4th$ version (i.e., 5 mutations in total). The mutations were chosen at random, although we replaced the ones that generated equivalent mutants. We used all the method level mutations in muJava (more details about them can be found in [47]).

Table 3 summaries the number of assertions that are passed in each faulty version. Note that a higher number of mutations does not necessarily correspond to fewer passed test cases (see for example the Phase of Moon program). This is a clear example of possible local optima.

7.2 Setting of the Framework

For the employed search algorithms, we used the default values in ECJ [2], unless otherwise specified in the paper. The maximum tree depth is 25. The maximum number of allowed fitness

evaluations is $50{,}000$. The computation is stopped in the case that a program that passes all the test cases is found.

For the GP algorithm, population size is 1000 (hence the maximum number of generations is 50). A tournament selection with size 7 is employed. The elitism rate is set to 1 individual for generation. The main search operator is mutation, that is done with probability 0.9. We still use crossover, but with a very low probability of 0.08. A tree is left unmodified with probability 0.01, whereas with probability 0.01 it is replaced with the original faulty program (this is done for forcing the presence of its genetic material at each generation).

Note that we have not tuned these values. The reason is explained in Section 7.4 after the experiments.

7.3 Experiments

We carried out three different sets of experiments:

1. For each faulty program, we tuned the parameter M (max number of mutations) for RS. Values considered are in range from 1 to 10. Each run has been repeated 100 times with different random seeds. The total number of runs is hence $5 * 7 * 10 * 100 = 35{,}000$.

2. For each faulty program, we tuned the parameter M (max number of mutations) for HC. Values considered are in range from 1 to 10. Each run has been repeated 100 times with different random seeds. The total number of runs is hence $5 * 7 * 10 * 100 = 35{,}000$.

3. For each faulty program, we run the GP algorithm. We tested the novel search operator with values of the node bias ranging from 1 to 10. Each run has been repeated 100 times with different random seeds. The total number of runs is hence $5 * 7 * 10 * 100 = 35{,}000$.

The total number of runs of the framework used for collecting data is $105{,}000$. This is a large number of experiments that can take up a long time to run. A larger case study would necessarily reduce the number of types of experiments and the number of repetitions (with different random seeds) for each experiment.

For these experiments, the number of robust programs that are obtained are shown in Figure 2, Figure 3 and Figure 4. The best value for the parameter M for RS is 4 (first set of experiments, Figure 2), whereas for HC it is 7 (second set of experiments, Figure 3). Tables from 4 to Table 10 compare the tuned RS against the tuned HC and a non-tuned GP. Tables from 11 to Table 17 compare GP when the novel operator is used with tournament size (i.e., node bias) 2.

The event of sampling a valid and/or a robust program can be modelled as a binomial process. Therefore, Fisher's Exact tests can be used to see whether there is any statistical difference between the success rate of two different algorithms or configurations.

7.4 Discussion

The experiments we carried out show that each of the 35 faulty programs can be automatically corrected with our tool JAFF. Of course, depending on the complexity of the software and the faults, these results are achieved with different computational effort.

Not surprisingly, when only few faults are considered (i.e., $V1$ and $V2$), the performance of RS and GP are very similar. But for more complex types of faults (i.e., $V4$ and $V5$) GP clearly stands out from the other considered algorithms (Fisher's exact tests confirm it in many cases). This is one reason why we did not need to tune the parameters of GP. Already with some arbitrarily setting it performs better then tuned RS and HC.

What came as a surprise is the performance of HC, which is very poor. One explanation would be that there can be many small modifications that can improve the fitness, but that then drive the

Table 4: Comparison for Phase of Moon

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	95	60	2137.13	1230.0	12482	5740488.0	46	50.63	50.0	57	2.336465
	HC	0	0	50000	50000.0	50000.0	50000	0.0	41	52.28	53.0	61	30.56727
	GP	92	71	1981	7874.73	1993.0	50000	1.80482595E8	40	51.33	50.0	114	49.07182
V2	RS	100	94	16	1203.29	844.0	6708	1657107.0	47	50.97	51.0	56	1.605152
	HC	1	0	1364	49515.62	50000.0	50000	2.365655E7	41	53.19	53.0	61	34.23626
	GP	98	80	1973	4078.29	1993.0	50000	5.5358582E7	46	51.92	51.0	79	12.33697
V3	RS	83	10	406	22460.47	18012.0	50000	3.0671632E8	45	50.4	51.0	66	7.878788
	HC	0	0	50000	50000.0	50000.0	50000	0.0	41	52.72	53.0	61	33.05212
	GP	98	8	2962	6551.3	4959.0	50000	4.7339725E7	41	50.71	51.0	62	11.94535
V4	RS	19	2	2544	45313.18	50000.0	50000	1.28933411E8	45	52.15	52.0	61	5.926768
	HC	2	0	531	49046.58	50000.0	50000	4.5238729E7	42	53.95	55.0	62	34.59343
	GP	88	7	3959	11863.57	5942.0	50000	2.18956904E8	41	52.6	52.0	107	48.78788
V5	RS	0	0	50000	50000.0	50000.0	50000	0.0	47	51.78	51.0	57	3.203636
	HC	0	0	50000	50000.0	50000.0	50000	0.0	44	56.15	57.0	63	34.33081
	GP	8	0	4972	48350.73	50000.0	50000	6.7942853E7	44	60.72	59.0	176	257.3349

Table 5: Comparison for Remainder

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	99	12	4240.81	2655.0	21122	1.9694475E7	156	160.18	160.0	165	1.724848
	HC	19	18	4	42022.78	50000.0	50000	2.96175183E8	152	163.07	161.0	170	24.5102
	GP	100	99	1977	3742.28	2980.0	11886	5275823.0	151	163.69	160.0	189	65.16556
V2	RS	0	0	50000	50000.0	50000.0	50000	0.0	150	158.91	160.0	168	12.16354
	HC	17	17	661	43124.64	50000.0	50000	2.56260183E8	151	163.72	164.5	170	37.39556
	GP	88	87	3971	16349.27	10886.0	50000	2.03742456E8	149	195.33	166.5	311	2888.425
V3	RS	0	0	50000	50000.0	50000.0	50000	0.0	149	158.29	158.5	166	11.15747
	HC	9	9	1382	46693.64	50000.0	50000	1.36677492E8	150	160.9479	160.0	171	30.78673
	GP	91	90	6942	19308.98	14863.0	50000	1.44971481E8	151	183.11	163.5	312	2108.867
V4	RS	0	0	50000	50000.0	50000.0	50000	0.0	149	159.12	160.0	167	9.379394
	HC	0	0	50000	50000.0	50000.0	50000	0.0	150	160.63	160.0	170	29.06374
	GP	42	42	11907	39718.67	50000.0	50000	2.03653224E8	150	172.79	161.0	430	1796.875
V5	RS	0	0	50000	50000.0	50000.0	50000	0.0	147	160.1	161.0	169	10.87879
	HC	1	0	26127	49763.25	50000.0	50000	5700156.0	152	162.78	163.0	171	23.48646
	GP	24	24	12862	43855.15	50000.0	50000	1.52540938E8	145	164.12	161.0	306	467.9248

current solution away from the global optima. Once HC is driven to such a suboptimal region, the use of large jumps (i.e., the number of mutations applied to generate the neighbour solutions) seems to be not enough to escape from them. However, more sophisticated variants of HC could be designed.

Figure 4 clearly shows that the novel search operator is useful in many cases, but for higher values of the node bias n the performance starts to decrease (this is in accordance with Equation 4). When $n = 2$, a closer look at tables from 11 to 17 shows that for simple types of faults (i.e., $V1$ and $V2$), there is not much difference in the performance (whether it is an improvement or a decrease of performance). For more complex faults (i.e., $V4$ and $V5$) it seems that the novel operator gives significantly better results (Fisher's exact tests confirm it for Remainder and TreeMap).

It is interesting to see whether in this particular type of application of GP we would get bloat or no. Unfortunately, bloat does occur. For example, the largest program we obtained is for Remainder (see Table 12). A final size of 430 nodes was obtained from a starting program with size 160.

Most of the time, a valid solution was also robust. That is a very important result, because it means that in general the patches generated by the JAFF are fixing the actual faults (at least in our case study). Given a set of test cases, there is an infinite number of semantically different programs that fit them. However, their distribution in the search space is in general not known, and they might be very far from each other. Fortunately, the experiments show that "near" a correct solution there are only few programs that are valid but not robust.

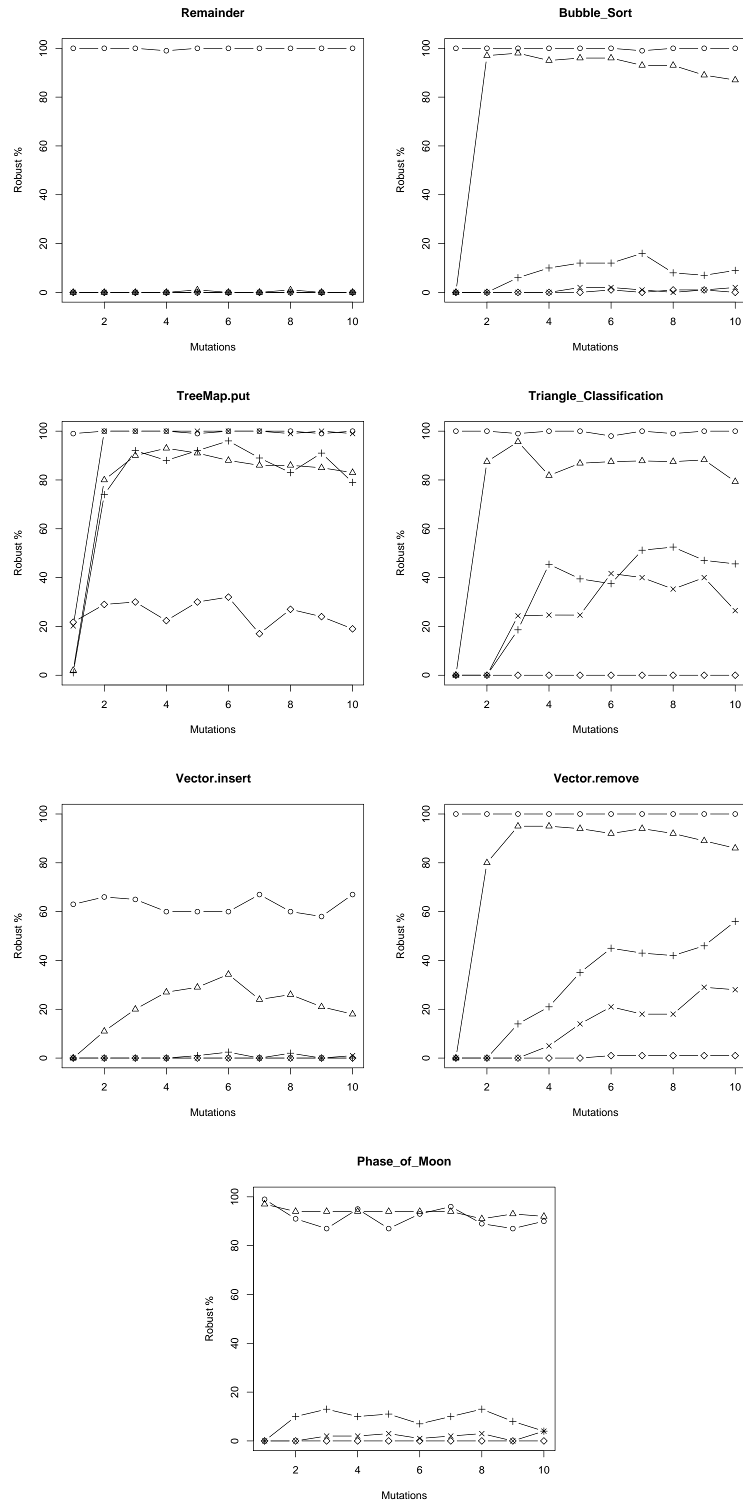

Figure 2: Results of tuning the value of maximum number of mutations for RS. Proportion of obtained robust programs are shown. Data were collected from 100 runs of the framework with different random seeds. There are 7 plots, one for each program in the case study. Each plot contains the results for each of the 5 faulty versions of that program.

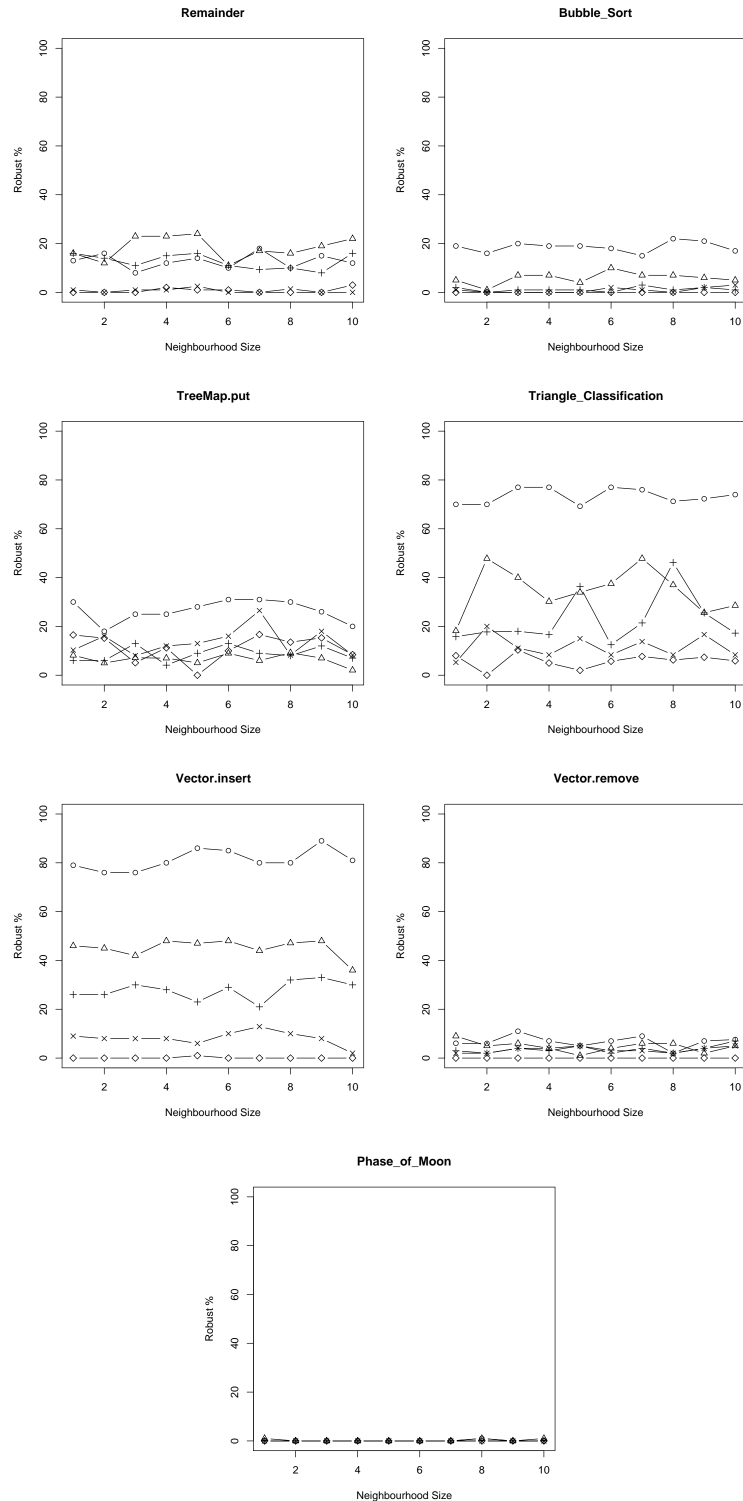

Figure 3: Results of tuning the value of maximum number of mutations (i.e., the neighbourhood size) for HC. Proportion of obtained robust programs are shown. Data were collected from 100 runs of the framework with different random seeds. There are 7 plots, one for each program in the case study. Each plot contains the results for each of the 5 faulty versions of that program.

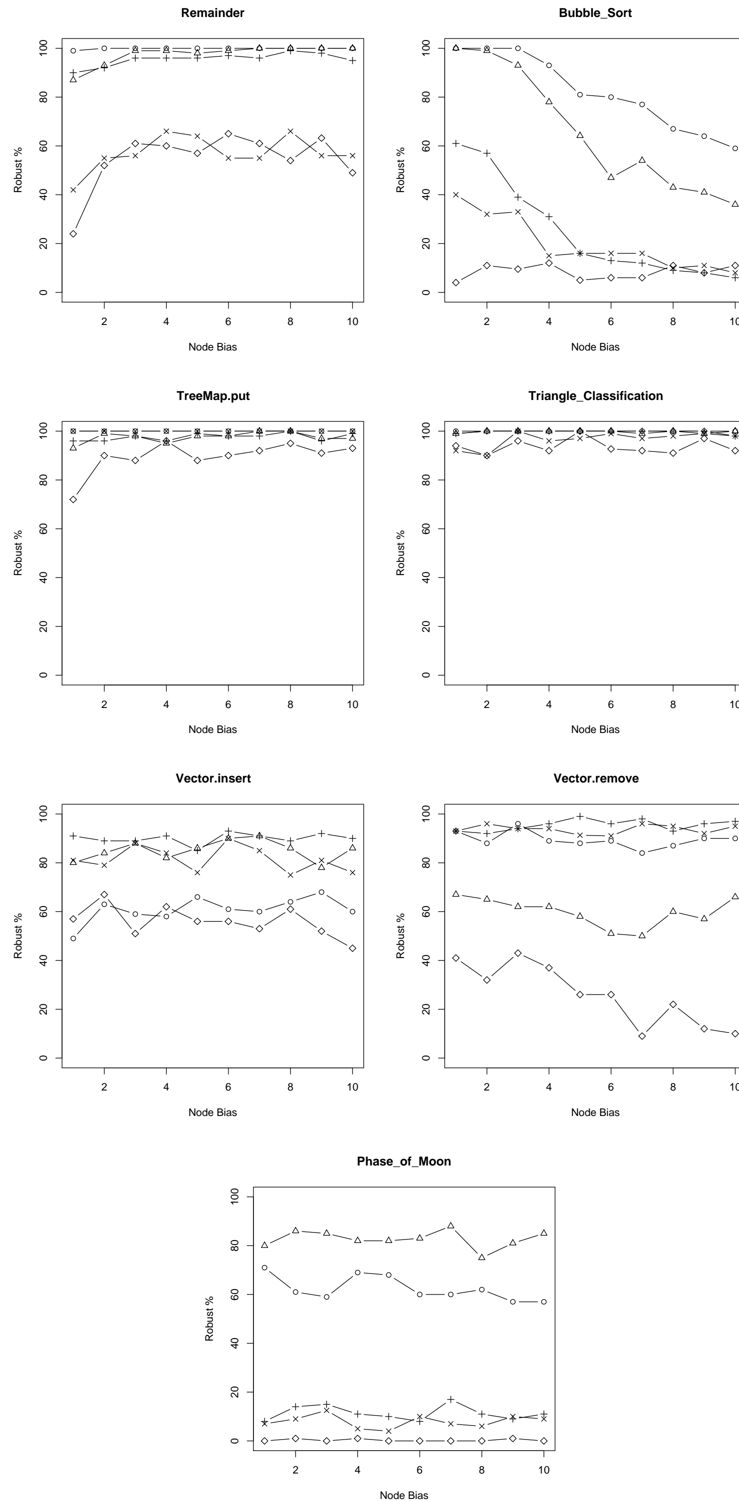

Figure 4: Results of GP using the novel search operator with different values n for the node bias. Proportion of obtained robust programs are shown. Data were collected from 100 runs of the framework with different random seeds. There are 7 plots, one for each program in the case study. Each plot contains the results for each of the 5 faulty versions of that program.

Table 6: Comparison for Bubble Sort

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	100	11	765.78	561.5	5277	577521.9	56	56.3	56.0	61	0.7171717
	HC	15	15	4	43418.5	50000.0	50000	2.56884057E8	48	57.84	56.0	66	15.73172
	GP	100	100	1983	2120.11	1991.0	10888	842416.3	56	56.14	56.0	63	0.7276768
V2	RS	95	95	8	12522.17	7891.0	50000	1.60618102E8	46	57.44	58.0	62	3.25899
	HC	7	7	12	46904.21	50000.0	50000	1.41018839E8	49	59.21	58.0	68	23.29889
	GP	100	100	2966	4314.74	3968.0	12882	3325363.0	56	58.2	58.0	72	6.141414
V3	RS	10	10	14682	47896.13	50000.0	50000	4.9980018E7	54	58.2	58.0	66	3.313131
	HC	4	3	209	48120.73	50000.0	50000	8.606186E7	49	62.74	63.0	69	26.82061
	GP	61	61	9901	34550.55	32167.0	50000	2.07041549E8	51	67.26	65.0	159	189.7095
V4	RS	0	0	50000	50000.0	50000.0	50000	0.0	50	58.33	59.0	67	5.132424
	HC	1	1	2754	49529.52	50000.0	50000	2.2323735E7	50	61.81	60.0	69	27.26657
	GP	40	40	11873	39988.8	50000.0	50000	1.88839409E8	51	65.73	66.0	145	92.54253
V5	RS	0	0	50000	50000.0	50000.0	50000	0.0	51	59.29	61.0	65	12.28879
	HC	0	0	50000	50000.0	50000.0	50000	0.0	61	61.0	61.0	61	0.0
	GP	4	4	13819	49419.68	50000.0	50000	2.7698504E7	51	64.67	65.0	96	33.94051

Table 7: Comparison for TreeMap.put

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	100	7	290.47	215.5	1455	83048.72	124	133.55	134.0	138	6.04798
	HC	33	31	9	33974.3	50000.0	50000	5.34342344E8	109	128.81	129.0	142	38.09485
	GP	100	100	1977	2000.55	1991.0	2986	9930.129	134	134.1	134.0	135	0.0909091
V2	RS	98	93	70	12949.75	7569.0	50000	1.6773158E8	121	134.15	134.0	138	5.421717
	HC	10	6	130	46557.72	50000.0	50000	1.32885341E8	116	130.6	130.0	141	24.10101
	GP	98	93	1991	7233.5	6924.0	50000	4.5553323E7	89	129.78	133.0	223	270.1733
V3	RS	98	88	87	13037.58	9616.0	50000	1.29265796E8	125	134.98	135.0	141	8.807677
	HC	11	9	76	46057.19	50000.0	50000	1.49424541E8	125	131.96	131.0	145	21.45293
	GP	98	96	1993	7074.34	5952.0	50000	4.8643077E7	108	131.67	133.5	223	135.8799
V4	RS	100	100	69	4047.41	2787.0	22842	1.5242408E7	119	130.78	129.0	142	28.51677
	HC	26	26	33	40880.0	50000.0	50000	3.32099098E8	126	133.6471	135.5	140	15.20499
	GP	100	100	2962	3622.98	3959.0	4974	482048.5	125	132.73	131.0	171	47.61323
V5	RS	23	22	1152	43768.76	50000.0	50000	1.76661972E8	126	134.3553	136.0	145	31.91211
	HC	16	16	724	44681.98	50000.0	50000	1.88406817E8	126	133.2143	134.5	142	18.95296
	GP	77	72	4950	23305.71	13344.0	50000	3.24309559E8	107	135.2	134.0	203	184.0404

Table 8: Comparison for Triangle Classification

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	100	9	726.9	451.5	3184	465492.1	79	98.31	101.0	106	24.84232
	HC	78	76	6	12933.87	148.0	50000	4.40213802E8	82	97.42	99.0	110	30.00364
	GP	100	100	1977	2296.69	1993.0	3959	230680.4	89	100.17	101.0	106	7.879899
V2	RS	90	81	1049	11327.27	5309.0	50000	2.19276055E8	93	100.9091	102.0	106	12.09091
	HC	52	47	7	33179.78	49224.0	50000	4.4634373E8	92	100.6957	102.0	111	20.31225
	GP	100	99	2964	4048.86	3971.0	7927	826978.2	89	103.23	101.0	174	215.8961
V3	RS	45	45	1049	38412.36	50000.0	50000	2.69787927E8	90	97.81818	99.0	103	18.76364
	HC	28	21	38	36302.93	50000.0	50000	5.08828403E8	86	97.0	97.0	105	22.30769
	GP	100	99	3957	5067.24	4962.0	6946	570266.1	82	100.54	97.0	193	289.2812
V4	RS	26	24	318	43461.23	50000.0	50000	1.86471282E8	87	98.45205	101.0	107	21.5567
	HC	13	13	88	44285.41	50000.0	50000	2.16766584E8	92	99.93103	101.0	109	18.99507
	GP	100	92	2986	4986.76	4956.0	6956	602590.7	81	102.2	98.0	204	407.596
V5	RS	0	0	50000	50000.0	50000.0	50000	0.0	97	100.4776	101.0	104	4.919946
	HC	7	7	10290	46947.23	50000.0	50000	1.21310996E8	94	100.2308	99.0	108	21.35897
	GP	100	94	4944	7142.54	6935.5	21755	3617325.0	80	106.9	101.0	203	641.9091

Table 9: Comparison for Vector.insertElementAt

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	60	9	415.63	283.5	1834	145470.7	44	53.52	53.0	59	3.969293
	HC	86	80	5	13274.09	1794.5	50000	3.48679583E8	44	52.04	51.0	64	14.38222
	GP	100	49	1977	2001.29	1991.0	2992	10032.29	51	53.72	53.0	57	1.153131
V2	RS	38	27	1409	39237.6	50000.0	50000	2.66236149E8	32	52.07	53.0	60	12.38899
	HC	47	44	97	34150.1	50000.0	50000	3.92340786E8	43	51.81	51.0	64	27.26657
	GP	100	80	2973	7576.95	6935.0	19784	1.2007509E7	36	53.19	53.0	86	34.80192
V3	RS	0	0	50000	50000.0	50000.0	50000	0.0	33	47.61	49.0	61	48.1797
	HC	22	21	104	41622.72	50000.0	50000	3.00193727E8	43	51.05	50.5	63	26.55303
	GP	95	91	7906	16463.11	13885.0	50000	8.5745263E7	31	57.76	55.0	119	235.0125
V4	RS	0	0	50000	50000.0	50000.0	50000	0.0	27	44.06	44.0	60	77.93576
	HC	12	12	23892	47807.71	50000.0	50000	3.9387227E7	34	51.51613	54.0	57	26.65806
	GP	85	81	8817	20418.05	15520.0	50000	1.75634471E8	40	54.53	50.5	104	140.5546
V5	RS	0	0	50000	50000.0	50000.0	50000	0.0	32	52.22	53.5	63	30.13293
	HC	0	0	50000	50000.0	50000.0	50000	0.0	50	54.83	54.0	62	2.506162
	GP	60	57	11893	35015.14	30688.5	50000	2.01166805E8	36	55.37	53.5	114	138.4779

Table 10: Comparison for Vector.removeElementAt

Version	Algorithm	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	RS	100	100	21	1852.94	1446.0	11719	3236402.0	49	61.62	62.0	67	6.94505
	HC	9	9	538	46370.06	50000.0	50000	1.54333325E8	52	60.89	61.0	72	20.09889
	GP	94	93	1981	8528.23	2978.0	50000	1.51724089E8	45	61.87	62.0	70	11.42737
V2	RS	97	95	186	14866.05	11146.0	50000	1.54956132E8	47	60.68	62.0	66	8.866263
	HC	7	6	1174	47277.12	50000.0	50000	1.15798437E8	47	61.28	62.0	71	18.8097
	GP	95	67	2980	11907.98	8927.0	50000	9.5283106E7	42	60.27	60.5	89	48.50212
V3	RS	21	21	7518	44952.85	50000.0	50000	1.33729877E8	46	59.68	60.0	67	16.03798
	HC	4	4	2125	48873.3	50000.0	50000	4.5676465E7	50	60.99	61.0	71	17.94939
	GP	96	93	3963	11335.36	8896.0	50000	8.3188229E7	35	59.59	59.0	108	68.48677
V4	RS	5	5	3153	48750.37	50000.0	50000	4.113287E7	51	59.35	60.0	68	11.05808
	HC	3	3	3670	48854.61	50000.0	50000	4.6501996E7	52	62.56	62.0	71	5.945859
	GP	94	93	3967	16241.65	13856.0	50000	1.10595009E8	37	57.61	57.0	77	39.95747
V5	RS	0	0	50000	50000.0	50000.0	50000	0.0	40	58.56	59.5	69	22.18828
	HC	0	0	50000	50000.0	50000.0	50000	0.0	62	62.0	62.0	62	0.0
	GP	41	41	8926	38049.46	50000.0	50000	2.5916941E8	43	60.77	61.0	72	37.65364

Table 11: Node bias for Phase of Moon

Version	Node Bias	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	1	92	71	1981	7874.73	1993.0	50000	1.80482595E8	40	51.33	50.0	114	49.07182
	2	93	61	1981	7608.9	1995.0	50000	1.59994246E8	45	52.81	50.0	111	94.33727
V2	1	98	80	1973	4078.29	1993.0	50000	5.5358582E7	46	51.92	51.0	79	12.33697
	2	97	86	1981	4918.8	1993.0	50000	9.7081664E7	49	52.55	51.0	104	46.08838
V3	1	98	8	2962	6551.3	4959.0	50000	4.7339725E7	41	50.71	51.0	62	11.94535
	2	97	14	2970	6718.12	4956.0	50000	6.7456329E7	45	52.24	51.0	111	85.7398
V4	1	88	7	3959	11863.57	5942.0	50000	2.18956904E8	41	52.6	52.0	107	48.78788
	2	96	9	2977	8152.61	5947.0	50000	8.6429858E7	43	51.74	52.0	82	24.03273
V5	1	8	0	4972	48350.73	50000.0	50000	6.7942853E7	44	60.72	59.0	176	257.3349
	2	14	1	6918	45919.05	50000.0	50000	1.43511683E8	43	57.78	58.0	94	61.95111

Table 12: Node bias for Remainder

Version	Node Bias	Valid	Robust			Steps					Size		
				min	mean	median	max	var	min	mean	median	max	var
V1	1	100	99	1977	3742.28	2980.0	11886	5275823.0	151	163.69	160.0	189	65.16556
	2	100	100	1981	3386.31	1997.0	8926	4389840.0	152	162.36	160.0	183	39.9499
V2	1	88	87	3971	16349.27	10886.0	50000	2.03742456E8	149	195.33	166.5	311	2888.425
	2	94	93	2964	11415.94	8922.0	50000	1.08120192E8	150	189.09	166.0	340	2414.083
V3	1	91	90	6942	19308.98	14863.0	50000	1.44971481E8	151	183.11	163.5	312	2108.867
	2	92	92	4968	15168.0	11876.0	50000	1.33802121E8	149	191.82	162.5	414	3574.129
V4	1	42	42	11907	39718.67	50000.0	50000	2.03653224E8	150	172.79	161.0	430	1796.875
	2	56	55	5949	33459.62	27194.5	50000	2.60867209E8	147	172.44	161.0	383	1573.825
V5	1	24	24	12862	43855.15	50000.0	50000	1.52540938E8	145	164.12	161.0	306	467.9248
	2	55	52	12882	36685.23	41038.5	50000	1.98223693E8	146	177.08	163.0	395	1787.953

Table 13: Node bias for Bubble Sort

Version	Node Bias	Valid	Robust	Steps					Size				
				min	mean	median	max	var	min	mean	median	max	var
V1	1	100	100	1983	2120.11	1991.0	10888	842416.3	56	56.14	56.0	63	0.7276768
	2	100	100	1977	3068.04	1993.0	19797	8760304.0	56	56.47	56.0	64	2.332424
V2	1	100	100	2966	4314.74	3968.0	12882	3325363.0	56	58.2	58.0	72	6.141414
	2	99	99	2971	5511.96	3978.0	50000	2.9593816E7	56	58.92	58.0	85	24.51879
V3	1	61	61	9901	34550.55	32167.0	50000	2.07041549E8	51	67.26	65.0	159	189.7095
	2	58	57	9881	37723.4	40544.0	50000	1.70364011E8	54	69.21	67.0	133	182.6726
V4	1	40	40	11873	39988.8	50000.0	50000	1.88839409E8	51	65.73	66.0	145	92.54253
	2	32	32	12868	43375.54	50000.0	50000	1.40733504E8	54	67.9	66.0	128	143.8687
V5	1	4	4	13819	49419.68	50000.0	50000	2.7698504E7	51	64.67	65.0	96	33.94051
	2	11	11	14850	48615.09	50000.0	50000	3.8011468E7	53	66.61	67.0	117	50.86657

Table 14: Node bias for TreeMap.put

Version	Node Bias	Valid	Robust	Steps					Size				
				min	mean	median	max	var	min	mean	median	max	var
V1	1	100	100	1977	2000.55	1991.0	2986	9930.129	134	134.1	134.0	135	0.0909091
	2	100	100	1975	1990.78	1991.0	2001	26.57737	134	134.11	134.0	135	0.09888889
V2	1	98	93	1991	7233.5	6924.0	50000	4.5553323E7	89	129.78	133.0	223	270.1733
	2	100	99	1991	4820.31	3974.5	9881	3988607.0	103	130.63	134.0	182	79.42737
V3	1	98	96	1993	7074.34	5952.0	50000	4.8643077E7	108	131.67	133.5	223	135.8799
	2	100	96	1993	4533.07	3963.0	7937	3361239.0	116	132.93	135.0	157	40.69202
V4	1	100	100	2962	3622.98	3959.0	4974	482048.5	125	132.73	131.0	171	47.61323
	2	100	100	1987	3157.16	2982.0	3981	185676.7	126	132.78	130.5	231	121.6279
V5	1	77	72	4950	23305.71	13344.0	50000	3.24309559E8	107	135.2	134.0	203	184.0404
	2	91	90	1983	16015.66	8919.0	50000	2.29592935E8	116	136.72	136.0	201	114.8299

Table 15: Node bias for Triangle Classification

Version	Node Bias	Valid	Robust	Steps					Size				
				min	mean	median	max	var	min	mean	median	max	var
V1	1	100	100	1977	2296.69	1993.0	3959	230680.4	89	100.17	101.0	106	7.879899
	2	100	100	1980	2077.368	1991.0	2994	80329.13	91	100.6842	101.0	105	3.398496
V2	1	100	99	2964	4048.86	3971.0	7927	826978.2	89	103.23	101.0	174	215.8961
	2	100	100	2964	3731.582	3963.0	4972	508542.2	88	105.0759	101.0	192	380.9172
V3	1	100	99	3957	5067.24	4962.0	6946	570266.1	82	100.54	97.0	193	289.2812
	2	100	100	3949	4581.29	4940.0	5955	448987.5	84	100.68	97.0	201	389.9976
V4	1	100	92	2986	4986.76	4956.0	6956	602590.7	81	102.2	98.0	204	407.596
	2	100	90	3943	4581.65	4948.0	5957	373174.3	88	101.07	97.5	202	354.0254
V5	1	100	94	4944	7142.54	6935.5	21755	3617325.0	80	106.9	101.0	203	641.9091
	2	100	90	3954	6273.2	5954.0	9899	1144242.0	81	106.96	97.5	202	787.7964

Table 16: Node bias for Vector.insertElementAt

Version	Node Bias	Valid	Robust	Steps					Size				
				min	mean	median	max	var	min	mean	median	max	var
V1	1	100	49	1977	2001.29	1991.0	2992	10032.29	51	53.72	53.0	57	1.153131
	2	100	63	1977	2020.66	1991.0	2986	28935.6	49	53.59	53.0	57	1.436263
V2	1	100	80	2973	7576.95	6935.0	19784	1.2007509E7	36	53.19	53.0	86	34.80192
	2	100	84	2974	7332.65	5957.5	20767	1.4407158E7	38	53.46	53.0	112	74.67515
V3	1	95	91	7906	16463.11	13885.0	50000	8.5745263E7	31	57.76	55.0	119	235.0125
	2	96	89	5947	15653.79	14822.0	50000	6.6893394E7	25	56.76	53.0	109	213.1539
V4	1	85	81	8817	20418.05	15520.0	50000	1.75634471E8	40	54.53	50.5	104	140.5546
	2	86	79	6923	21071.49	15826.5	50000	1.81154133E8	25	54.37	53.0	102	141.3062
V5	1	60	57	11893	35015.14	30688.5	50000	2.01166805E8	36	55.37	53.5	114	138.4779
	2	68	67	11889	33733.53	29624.0	50000	1.89767898E8	28	56.45	55.0	181	254.0682

Table 17: Node bias for Vector.removeElementAt

Version	Node Bias	Valid	Robust	Steps					Size				
				min	mean	median	max	var	min	mean	median	max	var
V1	1	94	93	1981	8528.23	2978.0	50000	1.51724089E8	45	61.87	62.0	70	11.42737
	2	89	88	1981	10108.82	1993.0	50000	2.71177338E8	46	62.15	62.0	74	11.42172
V2	1	95	67	2980	11907.98	8927.0	50000	9.5283106E7	42	60.27	60.5	89	48.50212
	2	94	65	3965	13215.6	9901.0	50000	1.15128681E8	47	61.06	61.0	124	82.11758
V3	1	96	93	3963	11335.36	8896.0	50000	8.3188229E7	35	59.59	59.0	108	68.48677
	2	96	92	3971	11534.18	8422.5	50000	8.8814779E7	46	58.15	57.0	80	42.2904
V4	1	94	93	3967	16241.65	13856.0	50000	1.10595009E8	37	57.61	57.0	77	39.95747
	2	97	96	4966	13323.57	10903.5	50000	6.5809471E7	38	56.57	57.0	94	51.29808
V5	1	41	41	8926	38049.46	50000.0	50000	2.5916941E8	43	60.77	61.0	72	37.65364
	2	32	32	9901	40655.23	50000.0	50000	2.17841782E8	48	60.82	61.0	73	34.69455

8 Limitations

The task of repairing software is very challenging. Regardless of the employed technique, there are serious problems that limit the automation of this task:

- Testing cannot prove that a program is faultless [50]. Therefore, the task of fixing faults cannot be completely automated, regardless of the technique that we use. Although the modified program that we obtain might pass all the given test cases, the introduced modifications might fix the program only for these inputs and they might introduce unwanted side effects. Figure 5 shows a simple example of a program that is able to pass all of its test cases although it is not correct. Hence, it is necessary that the developers check the modifications (i.e., the *patch*) done by the repairing algorithm, and this task cannot be automated (unless a formal way to prove its correctness is given, and that can be done only in trivial cases).

  Even if a patch does not actually fix the fault, it gives useful information to the developers. In fact, that information can be used as a way to locate the area of the code that is related to the manifestation of the fault. If the developer thinks that the proposed patch is not correct, he can provide more test cases for which the program fails and then rerun the framework again.

- A patch can reduce the efficiency of the code, e.g. it can make the software slower. However, the optimisation of non-functional criteria can be included in the search [9].

- When we evaluate a modified program to check if it is correct, the modifications we apply can make the program to enter in an infinite loop. The *Halting Problem* [20] is undecidable. We have to put time limits for the evaluation of modified programs on the test cases. The threshold could be estimated with heuristics based on the run of the faulty program on the test cases. A wrong estimation could severally harm the search.

- The modifications done to a program can be difficult to read. This is a common problem for example in GP. The *readability* of the code can be included in the objective to optimise. A simple heuristic would be to prefer, between two correct modified programs, the one that is more similar to the faulty input program.

- To check if a modified program is correct, we validate it against a set of test cases. Even with an efficient repairing algorithm, still many programs would likely be required to be evaluated during the search. If the execution of the test cases is computationally very expensive (this depends on the type of software), the computational cost of the repairing task would proportionally increase and likely it would become unpractical.

- Unless a formal specification is provided, the efficacy of repairing algorithms depends on the quality of the provided test cases. Quality of a set of test cases can be for example measured with coverage criteria [50]. More and better test cases would result in improved performance of the repairing algorithm. Even with an ideal repairing algorithm, we cannot expect good results if the test cases are too few and of low quality. This is similar to the problem of the choice of test cases for fault localization techniques [75].

9 Future Work

Automatic software repair is a difficult task that this paper addresses with search algorithms. There is still much more research that is required to do before software repair tools can be used in real-world scenarios:

```
<(5,-2,3), 0> // 0 represents 'not triangle'
<(4,3,6) , 1> // 1 represents 'scalene'
<(9,9,16), 2> // 2 represents 'isosceles'
<(3,3,3) , 3> // 3 represents 'equilateral'

function classifyTriangle(a, b, c)
   return  a + b - c;
```

Figure 5: An example of a test cases for the Triangle Classification problem [50] and an incorrect simple program that actually is able to pass all of these test cases.

- First step will be to extend JAFF to handle a larger subset of the Java programming language. This would allow us to use our prototype to more different types of case studies. We do not expect that all types of fault can be fixed in an automatic way. An extension of our framework will help us to better analyse which are the limits of automatic software repair.

- The search operators play a major role in the success of our technique. This operators should be optimised to handle faults that are common in real-world software. A deep analysis of which types of faults actually appear in real-world software is necessary to design proper search operators. One way to obtain this type of knowledge could be to use data mining techniques to software repositories (e.g., [71]).

- If a formal specification (e.g., written in either Z [60] or JML [44]) of the software is provided, we can automatically test all the new changes that we are introducing in the faulty program. Programs would evolve to pass the test cases while the test cases would evolve at the same time to find new faults in the evolving programs. This leads to a co-evolution between programs and test cases [32, 10]. This could significantly improve the performance of the framework. We investigated this idea on a toy example [11], and we are planning to extend our prototype JAFF to handle JML.

  Co-evolution could be used even in the case no formal specification is provided. Given a large set of test cases, co-evolution could be used to choose at each generation a subset to employ. This could be useful when there are so many test cases that it would be not efficient to run them all at each fitness evaluation. However, having so many test cases does not happen often.

- The fitness function of the programs is based on how well they pass their test cases. In our framework, we support test cases written as unit tests in JUnit [3]. The classes containing the unit tests need to be automatically instrumented for handling exceptions and for reporting to the framework whether the test cases are passed or not. The assert statements can be easily subclassed for giving more gradient to the search (i.e., they should give a degree of how much an assertion is failed). This is conceptually the same idea of *branch distance* in search based software testing [48]. Therefore, the same type of testability transformations [29] can be used to the instrumented unit test classes. We will investigate the improvement of the results that this technique could bring.

- Hybrid systems that include model checking based tools (see Section 2.2) with search algorithms should be investigated as well.

10 Conclusion

In this paper we have presented JAFF, the first prototype for the novel approach of repairing software in an automatic way with search algorithms. In contrast to the literature on the subject, in our system there is no particular restriction on the type of source code fault that can be fixed. However, exploiting the properties of real-world faults is helpful to reduce the search space.

Automatically repairing software is the natural next step after the automation of software testing and fault localization. It is a very complex task, and this paper gives the contribution of showing a feasible way to address this problem with evolutionary algorithms. Moreover, we analysed in detail the properties of this task, with the aim of finding its critical parts that need to be studied further for improving the performance.

We also presented a novel search operators. We theoretically studied the conditions for which it gives better results. This search operator improved the performance of our framework in our case study. This search operator could be extended to other applications where programs with branches (in the control flow) are tried to be evolved.

11 Acknowledgements

The author would like to thank Mary Jean Harrold and all the members of the program committee of the Doctoral Symposium of the IEEE International Conference on Software Engineering 2008. The author also wishes to thank Xin Yao, Raul Santelices, the Sebase and the Aristotle research groups. This work is supported by an EPSRC grant (EP/D052785/1) and by a IEEE Walter Karplus Summer Research Grant.

A Formal Proofs

Lemma 1. *Let t be the number of nodes (in the syntax tree) that are related to faults. Let s be the number of nodes that are given the same rank as these t nodes, whereas l is the number of nodes that have lower rank and h is the number of nodes that have higher rank. The probability $\delta(n)$ of choosing one of the t faulty nodes using a tournament of size n is given by Equation 4.*

Proof. Given a tournament of n nodes, let ζ be the number of faulty nodes (that are t in total) in n. Let γ be the number of non-faulty nodes in n that have higher rank than ζ (they are h in total) and let ψ the number of nodes with the same rank as ζ.

We need to pick up at least one of the t nodes and none of the h nodes (i.e., $P(\gamma = 0)P(\zeta \geq 1 \mid \gamma = 0)$). Then, among these n nodes, the probability of choosing a faulty node depends on how many of the s nodes are in these n (i.e., $P = \zeta/(\zeta + \psi)$). The probability of $\gamma = 0$ is equal to not picking up any of the h nodes for n times:

$$P(\gamma = 0) = \Big(1 - \frac{h}{l + s + t + h}\Big)^n .$$

The probability of $\zeta \geq 1$ is equal to 1 minus the probability of having $\zeta = 0$. We are assuming $\gamma = 0$, hence:

$$P(\zeta \geq 1 \mid \gamma = 0) = 1 - \Big(1 - \frac{t}{l + s + t}\Big)^n$$

For any possible values of ζ and ψ, $P(\zeta = i \wedge \psi = j \mid \zeta \geq 1 \wedge \gamma = 0)$ is the probability they assume the values i and j respectively. Therefore, we pick up one of the ζ nodes with the

following probability:

$$K(n) = \sum_{i=1}^{n} \sum_{j=0}^{n-i} \left(\Big(\frac{i}{i+j}\Big) P(\zeta = i \wedge \psi = j \mid \zeta \geq 1 \wedge \gamma = 0) \right) .$$

Calculating this probability $P(\zeta = i \wedge \psi = j \mid \zeta \geq 1 \wedge \gamma = 0)$ requires some more passages. Let's consider:

$$T = \frac{t}{l+s+t} \ ,$$

$$S = \frac{s}{l+s+t} \ ,$$

$$L = \frac{l}{l+s+t} \ .$$

Without considering their permutations, we have that the probability of having a set of size n with $\zeta = i \wedge \psi = j \wedge \gamma = 0$ is:

$$Z(i,j) = T^i S^j L^{n-i-j} \ .$$

Using Bayes' theorem, we obtain:

$$Z(i,j \mid i \geq 1) = \big(Z(i,j) - Z(i,j \mid i = 0)(L+S)^n\big)/(1-(L+S)^n) \ .$$

Because we use Z to calculate $K(n)$, and because in $K(n)$ the value i is always different from 0, we have:

$$Z(i,j \mid i = 0)(L+S)^n = 0 \ ,$$

and therefore:

$$Z(i,j \mid i \geq 1) = \frac{Z(i,j)}{(1-(L+S)^n)} = \frac{T^i S^j L^{n-i-j}}{(1-(L+S)^n)} = \frac{l^{n-i-j} s^j t^i}{(l+s+t)^n - (l+s)^n} \ .$$

We still need to calculate the possible permutations of the n nodes, and these are $\binom{n}{i}\binom{n-i}{j}$. Therefore:

$$P(\zeta = i \wedge \psi = j \mid \zeta \geq 1 \wedge \gamma = 0) = \binom{n}{i}\binom{n-i}{j} \frac{l^{n-i-j} s^j t^i}{(l+s+t)^n - (l+s)^n} \ .$$

Finally:

$$\begin{aligned} \delta(n) &= P(\gamma = 0) P(\zeta \geq 1 \mid \gamma = 0) \textstyle\sum_{i=1}^{n} \sum_{j=0}^{n-i} \left(\Big(\frac{i}{i+j}\Big) P(\zeta = i \wedge \psi = j \mid \zeta \geq 1 \wedge \gamma = 0) \right) \\ &= \left(1 - \Big(1 - \tfrac{t}{l+s+t}\Big)^n\right) \Big(1 - \tfrac{h}{l+s+t+h}\Big)^n \textstyle\sum_{i=1}^{n} \sum_{j=0}^{n-i} \left(\Big(\frac{i}{i+j}\Big) \binom{n}{i}\binom{n-i}{j} \left(\frac{l^{n-i-j} s^j t^i}{(l+s+t)^n - (l+s)^n} \right) \right) . \end{aligned}$$

□

References

[1] Apache ant. http://ant.apache.org/.

[2] ECJ: A Java-based Evolutionary Computation Research System. http://www.cs.gmu.edu/eclab/projects/ecj/.

[3] Junit. http://http://junit.sourceforge.net/.

[4] A. Agapitos and S. M. Lucas. Evolving modular recursive sorting algorithms. In *Proceedings of the European Conference on Genetic Programming (EuroGP)*, pages 301–310, 2007.

[5] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of the IEEE Software Reliability Engineering*, pages 143–151, 1995.

[6] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

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

[8] A. Arcuri, P. K. Lehre, and X. Yao. Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem. In *International Workshop on Search-Based Software Testing (SBST)*, pages 161–169, 2008.

[9] A. Arcuri, D. R. White, J. Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *International Conference on Simulated Evolution And Learning (SEAL)*, pages 61–70, 2008.

[10] A. Arcuri and X. Yao. Coevolving programs and unit tests from their specification. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 397–400, 2007.

[11] 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.

[12] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[13] C. Artho. Iterative delta debugging. In *Haifa Verification Conference*, 2008.

[14] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinhold, New York, 1990.

[15] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with tarantula. In *Proceedings of the the IEEE International Symposium on Software Reliability*, pages 137–146, 2007.

[16] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by ai techniques. *Artificial Intelligence*, 112(1-2):57–104, 1999.

[17] S. Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1), 2004.

[18] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[19] H. Cleve and A. Zeller. Locating causes of program failures. In *IEEE International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, second edition, 2001.

[21] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 433–436, 2007.

[22] R. A. DeMillo, R. J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[23] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *IEEE International Conference on Software Engineering (ICSE)*, pages 176–185, 2005.

[24] L. A. Dennis. Program slicing and middle-out reasoning for error location and repair. In *Disproving: Non-Theorems, Non-Validity and Non-Provability*, 2006.

[25] L. A. Dennis, R. Monroy, and P. Nogueira. Proof-directed debugging and repair. In *Seventh Symposium on Trends in Functional Programming 2006*, pages 131–140, 2006.

[26] M. Ducassé. A pragmatic survey of automated debugging. In *International Workshop on Automated and Algorithmic Debugging*, pages 1–15, 1993.

[27] P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri. Generalized algorithmic debugging and testing. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 317–326, 1991.

[28] A. Griesmayer, R. P. Bloem, and C. Byron. Repair of boolean programs with an application to c. In *Computer Aided Verification*, pages 358–371, 2006.

[29] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[30] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.

[31] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental Approaches to Software Engineering*, pages 267–280, 2004.

[32] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In *Artificial Life II: proceedings of the workshop on the Synthesis and Simulation of Living Systems*, pages 313–324, 1992.

[33] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[34] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.

[35] Y. Jia and M. Harman. Milu : A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, pages 94–98, 2008.

[36] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Conference on Computer Aided Verification (CAV)*, pages 226–238, 2005.

[37] J. A. Jones. *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, 2008.

[38] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.

[39] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *IEEE International Conference on Software Engineering (ICSE)*, pages 467–477, 2002.

[40] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[41] A.J. Ko and B.A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *IEEE International Conference on Software Engineering (ICSE)*, pages 301–310, 2008.

[42] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[43] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.

[44] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. 1999.

[45] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[46] S. Luke and L. Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

[47] Y. S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[48] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[49] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[50] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[51] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of java software. In *IEEE International Conference on Software Maintenance (ICSM)*, 2002.

[52] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.

[53] R. Purushothaman and D.E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511– 526, 2005.

[54] M. Reformat, C. Xinwei, and J. Miller. On the possibilities of (pseudo-) software cloning from external interactions. *Soft Computing*, 12(1):29–49, 2007.

[55] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39, 2003.

[56] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *International Symposium on Software Reliability Engineering*, pages 245–256, 2004.

[57] R. Sagarna and J.A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412, 2006.

[58] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In *Formal Methods in Computer-Aided Design*, pages 1–10, 2008.

[59] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.

[60] J. M. Spivey. *The Z Notation, A Reference Manual. Second Edition*. Prentice Hall, 1992.

[61] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 35–49, 2005.

[62] H. Sthamer. *Automatic generation of software test data using genetic algorithms*. PhD thesis, University of Glamorgan, 1996.

[63] M. Stumptner and F. Wotawa. Model-based program debugging and repair. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1996.

[64] M. Stumptner and F. Wotawa. A modelbased approach to software debugging. In *International Workshop on Principles of Diagnosis*, 1996.

[65] M. Stumptner and F. Wotawa. A survey of intelligent debugging. *AI Communications*, 11(1):35–51, 1998.

[66] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 248–257, 2008.

[67] F. Wang and C. H. Cheng. Program repair suggestions from graphical state-transition specifications. In *Proceedings of the International conference on Formal Techniques for Networked and Distributed Systems*, pages 185–200, 2008.

[68] W. Weimer. Patches as better bug reports. In *International conference on Generative programming and component engineering*, pages 181–190, 2006.

[69] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[70] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[71] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[72] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[73] E. Wong, T. Wei, Y. Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 42–51, 2008.

[74] C. Yilmaz and C. Williams. An automated model-based debugging approach. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 174–183, 2007.

[75] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *IEEE International Conference on Software Engineering (ICSE)*, pages 201–210, 2008.

[76] A. Zeller. Automated debugging: Are we close? *IEEE Computer*, pages 26–31, November 2001.

[77] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, 2002.

[78] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.

[79] X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.

[80] Y. Zhang and Y. Ding. Ctl model update for system modifications. *Journal of Artificial Intelligence Research*, 31:113–155, 2008.