Handling Test Length Bloat

Gordon Fraser[1] and Andrea Arcuri[2,*]

[1] *University of Sheffield, Department of Computer Science*
Regent Court, 211 Portobello, Sheffield, S1 4DP, UK
[2] *Simula Research Laboratory*
P.O. Box 134, 1325 Lysaker, Norway

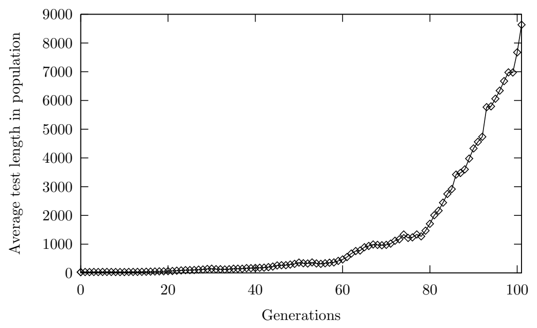
SUMMARY

The length of test cases is a little investigated topic in search-based test generation for object-oriented software, where test cases are sequences of method calls. While intuitively longer tests can achieve higher overall code coverage, there is always the threat of *bloat* – a complex phenomenon in evolutionary computation, where the length abnormally grows over time. In this paper, we show that bloat indeed also occurs in the context of test generation for object-oriented software. We present different techniques to overcome the problem of length bloat, and evaluate all possible combinations of these techniques using different starting lengths for the search. Experiments on a set of difficult search targets, selected from several open source and industrial projects, show that controlling bloat with the appropriate techniques can significantly improve the search performance. Copyright © 2011 John Wiley & Sons, Ltd.

1. Introduction

Deriving test cases for object-oriented software entails generation of sequences of method calls. Search-based techniques have been demonstrated to be a suitable tool for this task [1, 2], but raise important questions such as the choice of which *length* to use for these method sequences when we start the search. Before starting the search, we do not know what is the ideal minimal length for a test sequence that maximizes coverage – this is something that needs to be searched for. Nevertheless, when we start the search with some random test cases we still need to choose a length for them, and allow the search operators (e.g., crossover and mutation) to increase/decrease those test cases. The length, however, is not only an important parameter of the search but also one of its biggest threats: *Bloat* is a phenomenon in evolutionary search where the length of individuals increases to the point of making the search impossible.

*Correspondence to: Gordon Fraser, gordon.fraser@sheffield.ac.uk
Andrea Arcuri, arcuri@simula.no



(a) Bloat occurring during the search for a test case to cover a branch of the `XMLElement` class in NanoXML. As the evolution progresses, the average length of the population increases exponentially.

```
XMLElement xMLElement0 = new XMLElement();
boolean boolean0 = xMLElement0.getBooleanAttribute(
      "", "", "", true);
xMLElement0.setIntAttribute("", 0);
double double0 = xMLElement1.getProperty("F", 0.0);
int int0 = xMLElement0.getProperty("", 0);
int int1 = xMLElement0.getIntProperty("F", (
      Hashtable)null, "F");
XMLElement xMLElement1 = new XMLElement();
xMLElement0.addChild(xMLElement1);
xMLElement1.removeChild(xMLElement0);
xMLElement1.setDoubleAttribute("", 0.0);
xMLElement1.parseString("\\\"R).?>5Xsy");
// ...
```

(b) Excerpt of a test case for `XMLElement` that represents an individual of the search (modified for readability).

Figure 1: Bloat illustrated.

For example, consider Figure 1a, which shows the average length of the test cases in a population across generations computed by a genetic algorithm; Figure 1b shows an excerpt of such a test case. As a typical example of test case generation, the aim of this search is to find a sequence of method calls that will cover a non-trivial branch of the `XMLElement` class in the open source Java project NanoXML. Without any techniques to control bloat, the test cases become longer and longer after each generation of the search, until all the memory is consumed.

Bloat is an extremely complex phenomenon in evolutionary computation, and after many decades of research it is still an open problem whose dynamics and nature are not completely understood [3]. Unfortunately, in the past the issue of length has largely been neglected in the context of test case generation, and so there is no conclusive evidence on what starting length to choose and how to prevent it from being bloated.

This paper extends a previous experiment on the effects of length and bloat in the context of testing object-oriented software [4]. In particular, here we consider two distinct cases: (1) the traditional approach of targeting one branch at a time (e.g., as in [5]), and (2) the alternative approach proposed in [6, 7] of evolving whole test suites.

The evaluation of this paper considers a set of 100 difficult branches selected from six open source projects and an industrial case study. These 100 branches originate from 39 different classes, which we use for the experiments on whole test suite generation. Experiments are performed on 96 different bloat control configurations considering three different starting lengths, repeated with 25 random seeds each. This resulted in a significant amount of data backing up our results that took weeks to compute even when using a large cluster of computers.

The contributions of this paper are as follows:

Bloat: We propose and evaluate a set of different techniques to control bloat, identifying which combinations of techniques work best and should therefore be used in the future.

Length: We analyze the effect of the test case length on the results and on bloat, showing that the length plays a different role based on whether we evolve single test cases or whole test suites.

This paper features a very large empirical study, where the data have been rigorously analyzed with an appropriate statistical process to reduce threats to both internal and external validity. We took particular care in the data analysis, hoping that this paper can serve as a concrete reference example of how data should be collected and analyzed for randomized algorithms in software engineering – an area where there often is a lack of sound empirical evidence [8].

This paper is organized as follows: First, we give an overview of the bloat and length problems as well as previous work in Section 2. In Section 3 we provide details on how the EvoSuite tool automatically generates test cases for object-oriented software. Section 4 describes different techniques that can be applied to control bloat. Section 5 describes the experiments and discusses the results in detail. Finally, Section 6 discusses threats to the validity of our study, and Section 7 concludes the paper.

2. Background

2.1. Test Data Generation Techniques

As the number of possible test cases is usually infinite, a practical solution is to choose a coverage criterion, which represents a finite set of coverage goals. The objective of test generation is to obtain a test suite that, once executed, covers as many as possible of these goals. Unfortunately, for non-trivial software, writing such test suites by hand is a complex and tedious task. Therefore, automated techniques have been developed to address this task. A predominant criterion in the literature on structural testing is branch coverage, but in principle any other coverage criterion (e.g., dataflow based criteria or mutation testing [9]) is amenable to automated test generation.

For some testing goals it can be easy to find test input data to cover them, but for other goals it might be very difficult to find such data. Therefore, a common approach is to use random testing as a first step to cover the easy branches [10, 11]. After this initial phase, more sophisticated techniques are applied in a second phase to target all the remaining uncovered goals. A common approach in the literature is to target one such goal at a time, generating test inputs either symbolically [12] or with a search-based approach [1]. In this paper, we focus our analyses on this second phase: finding test data to cover difficult to reach testing goals, in particular for branch coverage. We further consider an alternative approach [6] we presented together with the EvoSuite tool, where all goals are targeted at the same time, thereby avoiding any problems related to how difficult individual goals are.

Meta-heuristic search techniques have been suggested as a possible solution to automate test case generation [1, 2]. In the context of object-oriented software, test cases are essentially small programs exercising the classes under test. Search-based techniques have been applied to test object-oriented software evolving method sequences with a genetic algorithm [5, 9] and strongly typed genetic programming [13, 14]. A promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [15, 16]), alleviating some of the problems both approaches have.

2.2. Effects of Size in Test Generation

While we aim to obtain the highest achievable coverage, it is also important that the resulting test suites are *small*. In this paper we assume the general case in which no *automated oracle* is available. In such a case, the output of each test case needs to be manually checked (e.g., by writing appropriate assert statements). Therefore, it is not feasible to ask a software tester to manually write assert statements for thousands of test cases. Similarly, long test sequences are intuitively more difficult to analyze and to understand than short sequences. This has led to work in which the goal was still obtaining highest coverage of the desired testing criterion, but with the secondary goal of obtaining a test suite that is as small as possible (e.g., [17–19]).

Effectively, this means that there are two conflicting goals: maximizing coverage C while minimizing the length of the test cases, which can be calculated by counting the number of statements S in them. How to combine these two measures? An approach would be to use a pareto-based multi-objective algorithm [20], but the problem is that the length is less important than coverage. Arcuri and Yao [17] used the following fitness function to maximize coverage: $C + (1/S + 1)$. In this way, in a comparison between two test cases, better coverage is always preferred regardless of length. On the other hand, Andrews *et al.* [19] used $(C \times 1000) - S$, which means that an increase of one point in coverage is better only if it does not result in a test case that is 1000 function calls longer. Baresi *et al.* [18] included the length of test sequences in the fitness function as well, but they do not specify how this was done. Notice that these approaches try to find single test sequences that cover as many testing goals as possible. This can lead to potential problems if there are conflicting goals, such that a single sequence cannot cover all goals at once. Another common approach that does not suffer such a problem of conflicting goals is to target one coverage goal at a time, each one with a different test case which will be combined in a final test suite [1, 5, 9, 21].

Arcuri [22] studied the role of test sequence length on branch coverage. In that work, container classes and an industrial integration testing problem were used as case study. Using *longer* sequences made the testing of these container classes trivial even with naive techniques such as random testing. A simple post processing was very effective to minimize such sequences without compromising their coverage. There has been other related work to shed light on the role of length of test sequences. Andrews *et al.* [23] studied whether for the *fault detection* of random testing it is better to have few long sequences or many short ones. Similar work has been carried out by Fraser and Gargantini [24].

2.3. Bloat in Evolutionary Computation

Evolutionary computation has been used to solve many kinds of scientific/engineering problems [25], where software engineering is just one example among many [26]. Given a problem to solve, and a fitness function to guide the search, the idea is to evolve the solutions through genetically inspired operators such as crossover and mutation. Each candidate solution has a representation (i.e., chromosome) that is dependent on the addressed problem (e.g., sequences of function calls in software testing problems). In some cases, the representation is of variable length, and the right choice for this length is something that needs to be searched for. This is a typical example in Genetic Programming [27], where programs are evolved, and those are usually represented with syntactic trees.

One problem with evolving individuals with variable length is *bloat* [3] where, generation after generation, the individuals become bigger and bigger without any particular improvement in the fitness values. This has several problems, as for example:

- The evolving population might end up consuming all the available RAM of the machine in which the evolutionary algorithm is run, which can crash the algorithm. Even with an upper limit to the maximum length, and by choosing a population size such that the available memory cannot be exceeded, this could result in very small population sizes.
- Longer/bigger individuals are more computationally expensive to run, which leads to less generations for the search given the same amount of time.
- Very long/big individuals will be difficult to analyze and understand for a practitioner. This is a serious issue when, for the given addressed problem, the solutions produced as output by the evolutionary algorithm need such a manual inspection. For example, very long test sequences, even if they find faults in the SUT, might be of little use for debugging because too complex to understand.

Already in the first book on Genetic Programming written by Koza [27], evolved solutions had many subtrees with no impact on the final fitness value (i.e., the so called *introns*, where parts of the genotype are not expressed in the phenotype). Although bloat is a typical problem usually studied in Genetic Programming, it does apply to all evolutionary algorithms in which the evolving individuals have variable size representation. For example, Langdon [28] investigated the bloat effect on the Santa Fe trail problem using Simulated Annealing, two variants of Hill Climbing, and one population-based evolutionary algorithm. Two different kinds of mutations were employed. Bloat always occurred in the population-based evolutionary algorithm, whereas it only occurred in Simulated Annealing and Hill Climbing depending on the mutation operator.

To efficiently solve and avoid bloat, one would need to design novel techniques based on the understanding of why bloat occurs in the first place. Unfortunately, bloat is a very complex phenomenon. After several years of investigation, its dynamics are still not fully understood.

During the years, several hypotheses have been presented by the research community to explain why bloat occurs. In their review [3], Silva and Costa identify the six main theories regarding the cause of bloat in Genetic Programming. Here we just provide a brief summary, whereas more detailed information can be found in [3].

Hitchhiking: Introns propagate because they adjoin highly fit building blocks [29].

Defense Against Crossover: Standard crossover (swapping of two subtrees) can be quite destructive (particularly in the later stages of the search), where the fitness of offsprings can be much worse than the one of parents. The presence of introns increase the chances of swapping subtrees (i.e., crossover) that have no impact on final fitness [30, 31], and so small modifications made by the mutation operator can better guide the search.

Removal Bias: Removing a subtree in an intron can only result in a reduction of nodes that is equal or lower than the size of the intron itself. On the other hand, we can insert subtrees of any size inside an intron without affecting the fitness value of candidate solution. This results in a bias in which it is more difficult to remove nodes rather than inserting new nodes in an evolving solution [32, 33].

Fitness Causes Bloat: This is the first theory that does not directly involve/rely on the presence of introns. There are several ways to express the same program/solution (i.e., several different genotypes resulting in the same phenotype). But there are many more ways to express the same functionality with large programs than with small programs. When it is difficult to find better individuals, the search tends to reward offsprings that have fitness values equal to the ones of the parents. Because with the same fitness there are many more individuals that are bigger than shorter, this creates a natural drift toward bigger solutions [34].

Modification Point Depth: In the latest stages of the search, it is better to have small modifications than large destructive changes. There is a direct correlation between the depth in the tree in which a modification occurs and its effect on the fitness. For example, a mutation on the root node can completely change the entire behavior of the program, whereas a mutation in a very deep leaf node will have much less impact. Bigger programs will be more deep and have more leaf nodes than small programs, and so they might be less affected by too destructive search operators [35].

Crossover Bias: When standard crossover is applied, the amount of genetic material exchanged between two parents remains constant (i.e., the sum of the sizes of the parents is equal to the sum of sizes of the offsprings). A population that undergoes several crossover operations will end up in having the distribution of its tree sizes converging to a Lagrange distribution of the second kind [36, 37]. In such distributions, small trees are very likely. On average, the generated very small individuals (e.g., a single node) are much less fit than the remaining bigger individuals. These small individuals die out in succeeding generations, and individuals with greater size than the average of the previous generations remain in the population, thus leading to bloat.

Although bloat has been extensively examined in the field of Genetic Programming, the problem has not been explored in detail in the context of test generation. It is of particular relevance for test generation for object-oriented software, where test cases are sequences of statements. The following section takes a closer look at this problem domain.

3. Evolutionary Testing of Object-Oriented Software

Search-based testing uses meta-heuristic search techniques to evolve an initial set of candidate test cases towards satisfying a given test objective, for example to reach a certain branch in the control flow of the software under test. In this section, we describe the techniques commonly used in search-based testing for object-oriented software, which are also those used for experimentation in this paper.

3.1. Genetic Algorithms

A genetic algorithm is a meta-heuristic search technique that tries to imitate the mechanisms of natural adaptation by evolving a population of candidate solutions using genetics-inspired operations. Algorithm 1 shows a commonly used version of such a genetic algorithm: Starting with a randomly generated initial population (*current_population*, Line 1), parents P_1 and P_2 are selected using, for example, rank selection [38] (Line 5), and then crossed over (Line 7) and mutated (Line 10) with a certain probability, resulting in offspring O_1 and O_2. Depending on the fitness values, either the offsprings or the parents are carried over to the next population (Z, Line 11–16). An iteration is done if the next generation Z has reached the same size as the current generation, and then Z becomes the new *current_population* (Line 17). This process is repeated until either an optimal solution has been found, or some other criterion stops the search (e.g., maximum allowed resources spent).

3.2. Fitness Function

The fitness function of a test case generation search depends on the chosen coverage criterion. In this paper, we consider branch coverage, which is also the predominant criterion used in the literature and in practice. A traditional fitness function for branch coverage [1, 2] consists of the *approach level* and the *branch distance*.

The approach level a_k is used to guide the search towards the target branch k. It is determined as the minimal number of control dependent edges in the control dependency graph between the target branch and the control flow represented by the test case. The branch distance b_k (for a target branch k) is a common heuristic to guide the search for input data to solve the constraints in the logical predicates of the branches [1]. The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see [1] for details). For example, for predicate $x \geq 10$ and x having the value 5, the branch distance to the true branch is $10 - 5 + w$, with $w \geq 1$. In practice, to determine the branch distance, each predicate of the software under test is instrumented to evaluate and keep track of the distances for each execution.

Algorithm 1 A genetic algorithm as used for search-based testing.

```
current_population ← generate random population
repeat
    $Z$ ← elite of current_population
    while $|Z| \neq |current\_population|$ do
        $P_1,P_2$ ← rank selection from current_population
        if crossover probability then
            $O_1,O_2$ ← crossover $P_1,P_2$
        else
            $O_1,O_2 \leftarrow P_1,P_2$
        mutate $O_1$ and $O_2$
        $f_P = min(fitness(P_1),fitness(P_2))$
        $f_O = min(fitness(O_1),fitness(O_2))$
        if $f_O \leq f_P$ then
            $Z \leftarrow Z \cup \{O_1,O_2\}$
        else
            $Z \leftarrow Z \cup \{P_1,P_2\}$
    current_population ← $Z$
until solution found or maximum resources spent
```

To avoid that the branch distance dominates the approach level, the branch distance has to be normalized in [0,1]. An appropriate normalizing function $\nu(x)$ is the one suggested by Arcuri [39]: $\nu(x) = x/(x+1)$. The fitness function for test case t and branch coverage goal k can therefore be defined as follows:

$$\text{fitness}(t,k) = a_k + \nu(b_k) \tag{1}$$

An alternative to generating a test case for each individual goal is to generate an entire test suite with respect to all coverage goals [6]. A test suite is represented as a set T of test cases t_i. Given $|T| = n$, we have $T = \{t_1,t_2,\ldots,t_n\}$. The fitness function for a test suite, with respect to how close a test suite is to covering *all* branches of a program, measures how close each branch came to evaluating to true and to false, which can again be estimated using the branch distance. It is important to consider that each predicate has to be executed at least twice so that each branch can be taken. Taking this into account, we define the branch distance $d(k,T)$ for branch k on test suite T as follows:

$$d(k,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(k,T)) & \text{if the predicate has been} \\ & \text{executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

Notice that there is a non-trivial reason behind the choice of $d(k,T) = \nu(d_{min}(k,T))$ applied only when the predicate is executed at least twice [17]. For example, assume the case in which it is always applied. If the predicate is reached, and branch k is not covered, then we would have $d(k,T) > 0$, while the opposite branch k_{opp} would be covered, and so $d(k_{opp},T) = 0$. The search algorithm might be able to follow the gradient given by $d(k,T) > 0$ until k is covered, i.e., $d(k,T) = 0$. However, in that case k_{opp} would not be covered any more, and so its branch distance would increase, i.e., $d(k_{opp},T) > 0$. Now, the search would have a gradient to cover k_{opp} but, if it does cover it, then necessarily k would not be covered any more (the predicate is reached only once) – and so on. Forcing a predicate to be evaluated at least twice, before assigning $\nu(d_{min}(k,T))$ to the distance of the non-covered branch, avoids this kind of circular behavior.

In addition to cover all branches, we require that each method is executed at least once, as some methods may contain no branches. We denote M as the set of methods of the class under test, and M_T as the set of methods executed by test suite T. With B, we denote the set of branch distances b_k. This results in the following fitness function, which the search aims to minimize:

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k,T) \tag{2}$$

3.3. Candidate Solution Representation

In search-based testing for object-oriented software, the candidate solutions are represented by test cases. In genetic algorithm terminology, each test case represents a chromosome. A test case is defined by a sequence of statements [5, 9]. A statement can be a call to a constructor, a method call, a reference to a field or primitive value, or a value assignment. Parameters of method and constructor calls, and source objects of method calls and field accesses have to be objects generated in the same test case at a previous position.

Primitive statements represent numeric variables, e.g.,
`int var0 = 54`, as well as Strings, enumeration variables, and array definitions.

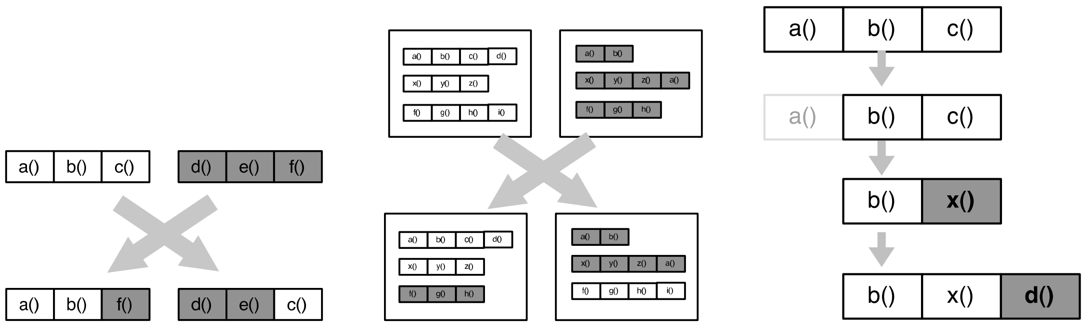
Constructor statements generate new instances of any given class; e.g.,
`XMLElement var1 = new XMLElement()`.

Field statements access public member variables of objects, e.g.,
`int var2 = var1.line_nr`.

Method statements invoke methods on objects or call static methods, e.g.,
`int var3 = var1.countChildren()`.

Assignment statements assign values to array indices or to public member variables of objects, e.g.,
`var2.maxSize = 10`.

A test case is a sequence of such statements, and the *length* of a test case is the number of statements it consists of. A test suite is a set of test cases, where we define the *size* of a test suite as the sum of the lengths of the individual test cases. Note that, given the same size, a test suite could be composed of a different number of test cases (e.g., a few long or many that are small).



(a) Test case crossover (b) Test suite crossover (c) Mutation

Figure 2: Crossover and mutation are the basic operators for the search using a genetic algorithm.

3.4. Crossover

Crossover of test cases creates two offsprings from two parent test cases P_1, P_2. Different flavors of crossover operators have been defined; in evolutionary testing of classes usually a single point crossover is used, meaning that each of the parent chromosomes is split at a single point, and the constituent parts of the parents are merged together (see Figure 2a). Crossover of test suites creates two offspring test suites by merging subsets of the parent test suites (see Figure 2b).

Crossover functions can further vary in how the crossover point is chosen. Tonella [5] chooses a random point in the range of $[1, min(length(P_1), length(P_2))]$. Baresi *et al.* [18] and Fraser and Zeller [9] choose different random positions for each of the parents in the range $[1, length(P_1)]$ and $[1, length(P_2)]$. In this paper, we call this latter crossover operator *Two Point Crossover* (TPX).

In the case of crossing over test cases, statements in the test cases might have dependencies and so it is necessary to try to satisfy these dependencies when attaching two sub-sequences from the parents. For example, if a test case contains a statement `foo.bar(x)`, then the subsequence starting with this statement has dependencies on variables `foo` and `x`. If in the subsequence to which this sequence is attached there are alternative objects of the same type as `foo`, then one of these objects is randomly selected and used to replace `foo`; the same holds for `x`. If there is no object that would satisfy the dependency, then additional statements need to be added to create an alternative instance to replace `foo`. Crossover of test suites does not require such repairing, as the individual test cases are not modified.

3.5. Mutation

Mutation introduces local changes into individuals. The mutation operators for test suites and test cases differ. When applying mutation to sequences of method calls, we distinguish three main types of mutation (each one applied with probability 1/3), illustrated in Figure 2c:

Deletion: This mutation operator removes a statement from a test case. As there are dependencies between statements (e.g., a return value might be used as a parameter in another method call), the dependencies need to be resolved, either by recursively deleting dependent statements, or by replacing references with different suitable objects. In a chromosome of length l, each statement is deleted with probability $1/l$.

Change: This mutation operator alters a given statement. For example, Tonella [5] lists a number of different possibilities to change statements. In our experiments, a change replaces a method call with a randomly chosen method call that has the same return type and has all dependencies satisfied at the given position in the test case. Primitive values (e.g., integer numbers) are changed by a random but bounded increase or decrease. Strings are changed in a more complex way, similar to test cases (i.e., each character in a string has a probability of being deleted, changed or there is the insertion of new characters). In a chromosome of length l, each statement is changed with probability $1/l$.

Insertion: In terms of bloat analysis insertion is the most interesting operator, as it is the only mutation operator that contributes to growth of the length. We use the following strategy to insert statements: With probability σ', a new randomly chosen statement is inserted at a random position in the test case. If it is added, then a second statement is added with probability σ'^2, and so on until the ith statement is not inserted. Parameters of new method calls are either satisfied with existing objects, or lead to addition of further statements to create necessary objects.

When generating the initial population of the search, we sample test cases at random, which uses the mutation operators described above. When evolving individual test cases, each individual is a random test case; when evolving test suites each initial individual consists of several randomly generated individuals, where the number of individuals is chosen randomly for each test suite. For each test case, we first choose a value r in $1 \leq r \leq W$ with uniform probability, where W is a value that needs to be set (e.g., $W = 80$). Then, on an empty sequence we repeatedly apply the insertion operator described above until the test case has a length $\geq r$. Because on average we expect $r = W/2$, the value of W should not be set too high, otherwise there is the risk of consuming all the available RAM.

When a test suite T is mutated, each of its test cases is mutated with probability $1/|T|$, such that on average only one test case is mutated. Then, a number of new random test cases is added to T: With probability σ, a test case is added. If it is added, then a second test case is added with probability σ^2, and so on until the ith test case is not added (which happens with probability $1 - \sigma^i$). Given an upper limit limit N on the number of test cases $|T| = n$, test cases are added only if that limit N has not been reached, i.e., if $n < N$.

3.6. Generating Test Suites

Any non-trivial class will have a number of different coverage goals, even for simple coverage criteria. As discussed in Section 2, when one is not targeting all branches at the same time it is common practice to have a first phase of random testing to cover the easy branches. Then, each remaining target can be individually sought with more sophisticated techniques. Some of these remaining coverage goals may be *infeasible*, which means that there exists no test case that would cover them. To avoid that all available resources are wasted on infeasible or difficult coverage goals on which the search fails, it is necessary to limit the resources spent on a single coverage goal. For this, we apply the following strategy:

- For $|B|$ branches to cover and an initial budget of X statements (or fitness evaluations, generations, etc.), the execution limit for the search on each branch is $X/|B|$.
- If a branch is covered, some budget may be left over, and so after the first iteration on all branches there is a remaining budget X'. For the remaining uncovered branches B' a new budget $X'/|B'|$ is calculated and a new iteration is started on these branches.
- This process is continued until the maximum number of statements is reached.

Test cases are only generated for branches that have not been covered previously by other test cases, as a test case can cover more than one branch. In contrast, when evolving test suites with respect to all branches at the same time, one does not need to worry about how to distribute the search budget on the individual branches. In fact, this is a huge advantage as one cannot run into the problem that an unreasonably large amount of the budget is wasted on infeasible branches, such that there is insufficient budget left for the remaining branches. However, in the presence of infeasible branches the search by definition cannot achieve full coverage, and so we need to set a limit on the search as well. However, there is no need of any explicit first phase of random testing to cover the easy branches, as this is directly included in the first population of the GA which is generated at random.

3.7. Test Case Minimization

The type of testing problems addressed in this paper is to target difficult faults for which automated oracles are not available – which is a common situation in practice. Because in these cases the outputs of the test cases have to be verified manually, the generated test suites need to be of manageable size. There are two contrasting objectives: the "quality" of the test suite (e.g., measured in its ability to trigger failures once manual oracles are provided) and its size. The approach we followed in this paper can be summarized as: Satisfy the chosen coverage criterion (e.g., branch coverage) with the smallest possible test suite.

As discussed throughout the paper, allowing the search to dive into longer test cases can significantly improve the coverage [22]. But in the end, we need to output to the user only a smaller test suite with no redundant statement. This means that, when we stop the search (e.g., after a timeout, or after a predefined number of fitness evaluations), we need to *minimize* the test suite to remove unnecessary statements.

A simple minimization technique discussed in [22] is as follow: given a test case of length l, remove one statement, and re-execute this reduced test case. If the coverage decreases, then re-insert that statement, otherwise keep this shorter test case. Repeat this process l times for each statement in the test case.

How long will this minimization process take? For simplicity, let us ignore the case in which a removal of a statement forces the removal of other statements (defining a complete theoretical framework to analyze minimization techniques is not in the scope of this paper). For example, if successive statements have dependency on this removed statement (e.g., it returns a variable that is successively used as input in other function calls), then we can "repair" these statements by using different input variables. When we remove a statement from a sequence of length l, executing this shorter test case would imply executing $l-1$ statements. On one hand, if the test case is already minimized, this minimization process would require to execute $l \times (l-1) = O(l^2)$ statements. On the other hand, if each time we remove a statement the coverage does not decrease, then we would need to execute $(l-1)+(l-2)+\ldots+1 = \sum_{i=1}^{i=l-1} i = l(l-1)/2 = \Omega(l^2)$ statements. Therefore, the complexity of this minimization algorithm would be quadratic in the number of statements, i.e., $\Theta(l^2)$ (tight bound). Notice that, even if more efficient minimization algorithms with lower complexity can be designed, their runtime complexity would still be dependent on the test case length. Consequently, the cost of minimizing a test case might not be negligible if it is too bloated, which is a further argument in favor of bloat control techniques.

4. Bloat Control Techniques

Bloat occurs when small negligible improvements in the fitness value are obtained with larger solutions. This is very typical in classification/regression problems. In software testing the fitness function is often just the obtained coverage, and so we might not expect bloat because the fitness would assume only few possible values. However, as soon as other metrics are introduced with large domains of possible values (e.g., branch distance [1] or mutation impact [9]), bloat might occur.

As discussed in Section 2.3, bloat can be a particularly harmful phenomenon. Longer sequences can consume large amounts of memory and take longer to evaluate, which would lead to less generations in the evolutionary search (within the same amount of time). Furthermore, very long sequences cannot be directly used for testing purposes unless an automated oracle is available, which is usually not the case.

Since the problem of bloat is well known, different techniques have been proposed to keep bloat under control. In this paper, we adapt and tailor bloat control methods from the literature of Genetic Programming [3] to our testing problem to deal with the possibility of bloat. However, these techniques may not apply directly to test case generation.

It is also important to note that there is a difference between the length of the test cases that are given as output after the search is finished and the different length values that the evolving test cases have during the search itself. On one hand, the output sequences should be as short as possible (while optimizing coverage). On the other hand, during the search it can be very useful to have longer sequences [22], because it would make the search able to

Algorithm 2 Adapted genetic algorithm that includes bloat checks, highlighted with gray background.

```
$current\_population \leftarrow$ generate random population
repeat
    $Z \leftarrow$ elite of $current\_population$
    while $|Z| \neq |current\_population|$ do
        $P_1,P_2 \leftarrow$ select with extended rank selection
        if crossover probability then
            $O_1,O_2 \leftarrow$ crossover $P_1,P_2$ with RPX
        else
            $O_1,O_2 \leftarrow P_1,P_2$
        mutate $O_1$ and $O_2$ with size check
        $f_P = min(fitness(P_1),fitness(P_2))$
        $f_O = min(fitness(O_1),fitness(O_2))$
        $l_P = length(P_1) + length(P_2)$
        $l_O = length(O_1) + length(O_2)$
        $T_B$ = best individual of $current\_population$
        if $f_O < f_P \vee (f_O = f_P \wedge l_O \leq l_P)$ then
            for $O$ in $\{O_1,O_2\}$ do
                if $length(O) \leq 2 \times length(T_B)$ then
                    $Z \leftarrow Z \cup \{O\}$
                else
                    $Z \leftarrow Z \cup \{P_1$ or $P_2\}$
        else
            $Z \leftarrow Z \cup \{P_1,P_2\}$
    $current\_population \leftarrow Z$
until solution found or maximum resources spent
```

explore larger areas of the search landscape without being trapped in fitness plateaus. Once the search for maximum coverage is finished, a post-processing can be used to easily remove the unnecessary function calls.

Considering all these differences, the dynamics of bloat in the case of testing object-oriented software might not be the same as the ones in Genetic Programming. Therefore, the methods coming from the literature of Genetic Programming to contrast bloat might need to be adapted. In this paper we want to shed light on this research problem, and to this extent this section describes bloat control techniques that can be applied to test case generation. Algorithm 2 shows an adapted version of the genetic algorithm presented earlier (Algorithm 1). It contains five different techniques to control bloat: First, the rank selection algorithm is updated to take length into account (Line 5); second, the crossover is an important source of bloat and can be adapted (Line 7); third, mutation can increase the length, but a straightforward countermeasure is to introduce a size limit (Line 10); fourth, a common source of length bloat is when offspring has the same fitness as its parents but is longer – this case is handled in

Line 16; finally, rather than just fixing an upper bound it is also possible to restrict the rate of growth dynamically (Line 18). In the following, we discuss these five techniques in detail.

4.1. Integrating Length During Rank Selection

A straight-forward approach to prevent bloat is to penalize the length directly in the fitness function [9, 17–19]. However, as discussed in Section 2, combining two different objectives that have different order of measure is not easy. Furthermore, because the branch distance might obtain any possible continuous value, it would not be possible to combine it with the length such that the length would be less important. The fitness function $C + (1/S + 1)$ (where S is the number of statements of the test case, i.e., its length) discussed in Section 2 works because the coverage C can only assumes integer values.

Instead of combining the length in the fitness function in Equations 1 and 2, we use a different approach. In general, the fitness function is only used to select individuals for reproduction. In this paper, we use *rank selection* [38] (see Line 5 in Algorithm 2). Test cases/suites are ranked based on their fitness value. Individuals with better fitness will receive a better rank, and so will have higher chances of being selected for reproduction. To penalize longer test sequences without penalizing a better fitness value regarding coverage (e.g., branch distance), in case of ties in the ranking (i.e., same fitness value), we resolve the ties by giving a better rank to the test cases/suites that are shorter.

4.2. Relative Position Crossover

One possible source of bloat is the crossover function, in which one of the offsprings can grow in size (length when the crossover is applied between two parent test cases, and number of test cases when it is applied on two parent test suites). If we choose two different splitting points in the parents (e.g., P_1 and P_2) at random using TPX, then the size of the offsprings can be very unbalanced when the splitting points are at the opposite edges of the chromosomes. For example, when crossover is applied on test cases, the length of the offsprings would vary between 0 and $length(P_1) + length(P_2)$, with average value $(length(P_1) + length(P_2))/2$.

Another version of the crossover operator generates two offsprings O_1 and O_2 from two parent test cases P_1 and P_2. A random value α is chosen from [0,1]. On one hand, the first offspring O_1 will contain the first $\alpha|P_1|$ genes (i.e., statements or test cases) from the first parent, followed by the last $(1-\alpha)|P_2|$ genes from the second parent. On the other hand, the second offspring O_2 will contain the first $\alpha|P_2|$ genes from the second parent, followed by the last $(1-\alpha)|P_1|$ test cases from the first parent. In this paper, we call this operator *Relative Position Crossover* (RPX), and it is shown in Line 7 in Algorithm 2. In contrast to TPX, in RPX the offsprings will never have greater size than the biggest of the parents.

For crossover applied on test cases, regardless of the crossover operator (i.e., TPX and RPX) the test cases can still grow in size, as additional statements might be added to satisfy dependencies of merged parts of the parents. In addition, test cases can grow as part of the mutation operator.

For RPX applied on test suites, even if no offspring can have more test cases than the "biggest" of the parents, its size (defined as the sum of lengths of each of its test cases, recall

Section 3.3) might significantly increase. This can happen for example if one of the offsprings inherits the longest test cases from both parents, while the other offspring inherits the shortest. In other words, RPX guarantees that the number of test cases in a test suite does not increase, but the total length might increase.

4.3. Fixed Maximum Length

A very common approach to contrast bloat is to put an upper limit L to the length of the test cases, e.g., $L = 100$ function calls. This constraint can be enforced in several ways: First, by having search operators that do not sample offspring test cases that are longer than L (Line 10 in Algorithm 2). For example, an insertion mutation could be avoided if the length already equals L. Second, offsprings that are longer than L (e.g., when we use TPX) can be rejected, and the parents will be copied to the next generation instead of the offsprings. Finally, the limit can be given implicitly by specifying a maximum amount of resources to be spent per individual. For example, one can define a timeout for the execution of test cases.

But how to choose a maximum length L? Should it be equal to 100 or 1,000? Too small a value might make the search very unlikely to succeed. With a large value there might be the risk of running out of memory and being severely affected by bloat. In Genetic Programming, a rule of thumb is to have trees of maximum depth equal 17. In the case of testing object-oriented software, we are aware of no work that tries to analyze and give an answer to this research problem.

As previously mentioned in Section 2.3, bloat is a very complex phenomenon. This is illustrated by the fact that, counterintuitively, using a limit L might actually favor the raise of bloat [37]. As a detailed discussion of this would go beyond the scope of this paper, we refer the interested reader to the literature [3, 37].

4.4. Length Dependent Parent Replacement

A potential source of bloat concerns the relation between the performance of the parents and its offsprings. If one offspring has a fitness value strictly better than the fittest of its parents, then both offsprings will be accepted in the new generation independently of their length (but other bloat control methods might still prevent it). However, in case of equal fitness, the offsprings will be accepted only if they are not longer than their parents; see Line 16 in Algorithm 2. In other words, we accept longer test sequences in the new generations if and only if at least one of the offsprings has strictly better fitness value than both the parents.

4.5. Dynamic Upper Bounds

Choosing a proper value for the upper limit L might not be easy, and there might be side-effects due to the use of a fixed L. Beside L, one further approach discussed by Silva and Costa [3] is to use a dynamic limit based on the best individual T_B in the current generation. For example, an offspring O could be rejected if $length(O) > 2 \times length(T_B)$ (Line 18 in Algorithm 2). In this way, we would not need the burden of fixing a value for L, and would allow a less constrained search of the solution space. For example, if the current best solution has length 10, we would still be able to explore sequences up to length 20. Notice that such a dynamic limit can be used in conjunction with the static limit L.

In whole test suite generation, we can still apply exactly the same dynamic upper bound, where the length of an individual (i.e., a test suite in this context) is the sum of the lengths of its test cases (recall Section 3.3).

5. Experimental Evaluation

To study the effects of both the test case length and the bloat control techniques, we performed a set of experiments. In detail, this evaluation aims to answer the following research questions:

RQ1: How does the maximum starting length W influence the search results?

RQ2: How do the bloat control methods impact the achieved coverage?

RQ3: Which techniques to control bloat are most effective?

RQ4: How do the bloat control methods affect whole test suite evolution?

RQ5: How likely are the presented results to generalize to other case studies?

5.1. Case Study

As subject for our experiments, we selected a set of open source Java libraries: Java Collections (a subset of the java.util library, as used in the Randoop [40] experiments), Apache Commons Collections (version 3.2.1) and Commons Primitives (version 1.0), NanoXML (version 2.2.3 "light"), and a Java translation of the String case study subjects used by Alshraideh and Bottaci [41]. We also use a set of numerical applications used in [42] and a subset of classes from an industrial application [43]. This resulted in a large and variegated case study.

This case study resulted in nearly 1,000 classes and more than 15,000 branches— far too many for an in-depth analysis of bloat control methods. We needed a way to filter out the *easy* branches, and identify the difficult ones. This is also of practical interest: If a testing technique $\mathcal{A}$ is twice as fast as another technique $\mathcal{B}$, then solving an easy problem in one millisecond instead of two milliseconds would be an improvement of no value from a practical stand point. On the other hand, solving a problem in one hour instead of two hours would be of practical interest.

In our case study, we applied the following filtering phase to choose a selection of difficult branches: We applied our test case generation tool with a search limit of 200,000 statements per branch (of which there were more than 15,000) with all bloat control techniques enabled, collecting information for each branch about the number of statements executed until a solution was found (one run per branch). Given this information, we selected the subset of those branches which resulted in a solution (i.e., are feasible), but required between 100,000 and 200,000 statements for this solution (i.e., are non-trivial). This resulted in a set of exactly 100 difficult but feasible branches, which we used for further experimentation.

For experiments with whole test suite generation we require not only individual branches but entire classes, as the optimization targets all branches at the same time. We therefore chose the set of classes that contained the 100 selected branches for experiments with whole test suite generation; these 39 classes together with information on the number of branches and lines of code are listed in Table I. Note that the number of branches can be higher than the number of lines of code as branches are counted on the bytecode, and each compound condition is compiled to several independent branches in the bytecode. It is also interesting to see that many data structures, and in particular many classes related to map data structures, ended up in the final set of classes. However, this choice was not made deliberately, and simply resulted because of the most complex branches, as described above.

5.2. Experimental Setup

For the experiments we consider the five bloat control techniques described Section 4. In particular, we use the following labels to indicate whether a bloat technique is employed:

Bo: the maximum length for the test cases is bounded from above, i.e., if we set an upper limit L. In particular, we chose $L = W$, where $[1,W]$ is the range in which the length of new random test cases is chosen from.

Xo: the crossover RPX is used instead of TPX.

Ra: use the length of the test cases/suites to resolve the ties in the rank selection of the parents for reproduction.

Pa: check length of offsprings against parents' length.

Be: check the length of offsprings against best solution's length in the current population.

5.2.1. Bloat Control Configurations

For all experiments with whole test suite generation, we use the EvoSuite tool [6]. EvoSuite also supports generating individual test cases for individual coverage goals, which we use for the experiments on individual goals. However, we use EvoSuite synonymously for whole test suite generation in the following sections. For the initial length of random test cases, we consider three values for W, specifically $W \in \{20,50,80\}$. For the experiments in this paper, the total number of configurations for the genetic algorithm is hence $2^5 \times 3 = 96$. Because we run the search on each branch independently, this means a total of $96 \times 100 = 9{,}600$ different experiments. In all the experiments, we give a budget of 100,000 statement evaluations (a typical value in the literature, e.g. [21]). The search can finish for two reasons: either a test case that covered the target branch is found (a so called *global optimum*), or the entire execution budget has been consumed.

Table I. Details of the classes used for experimentation with whole test suite generation; the 100 branches used for the first set of experiments is contained in these classes. Lines of non-commenting source code (LOC) in the case study classes were calculated with JavaNCSS (http://javancss.codehaus.org/)

Library	Class	#Branches	LOC
String Casestudy	Costfuns	21	16
String Casestudy	Ordered4	29	11
String Casestudy	Title	43	18
Commons Primitives	adapters.AbstractFloatCollectionCollection	21	46
Commons Primitives	RandomAccessFloatList	81	205
NanoXML	XMLElement	310	661
Java Collections	AbstractList	78	255
Java Collections	AbstractMap	121	190
Java Collections	Arrays	629	843
Java Collections	HashMap	203	394
Java Collections	Hashtable	217	403
Java Collections	IdentityHashMap	208	435
Java Collections	LinkedHashMap	49	120
Java Collections	LinkedList	113	247
Java Collections	SubList	48	110
Java Collections	WeakHashMap	206	395
Java Collections	Collections	509	1,021
Java Collections	TreeMap	402	663
Commons Collections	bidimap.AbstractDualBidiMap	110	289
Commons Collections	bidimap.DualTreeBidiMap	54	149
Commons Collections	buffer.PriorityBuffer	82	168
Commons Collections	collection.CompositeCollection	60	118
Commons Collections	collection.UnmodifiableCollection	9	27
Commons Collections	CursorableLinkedList	290	642
Commons Collections	functors.AnyPredicate	14	31
Commons Collections	functors.ChainedClosure	16	38
Commons Collections	functors.SwitchTransformer	25	57
Commons Collections	iterators.ArrayListIterator	16	42
Commons Collections	iterators.CollatingIterator	56	125
Commons Collections	keyvalue.TiedMapEntry	26	33
Commons Collections	list.AbstractLinkedList	189	438
Commons Collections	list.TreeList	180	357
Commons Collections	map.AbstractHashedMap	243	520
Commons Collections	map.AbstractInputCheckedMapDecorator	21	62
Commons Collections	map.AbstractLinkedMap	94	206
Commons Collections	map.AbstractReferenceMap	163	351
Commons Collections	map.StaticBucketMap	144	261
Commons Collections	map.UnmodifiableEntrySet	24	62
Commons Collections	map.UnmodifiableSortedMap	17	59

5.2.2. Procedure for Individual Test Case Generation

To compare whether a configuration $\mathcal{A}$ is better than another configuration $\mathcal{B}$ on a branch, we apply the following procedure, as described in more detail by Arcuri and Briand [8]. We run the genetic algorithm n times for both configurations on that branch (so $2n$ runs), and we record the number of times a out of n an optimal solution is found with the first configuration $\mathcal{A}$, and the number of times b it is found with the other configuration $\mathcal{B}$. The *success rate* for $\mathcal{A}$ is defined as a/n. If $a > b$, then it would seem that $\mathcal{A}$ is better then $\mathcal{B}$, and the other way round if $a < b$. However, because genetic algorithms are randomized, we need rigorous statistical tests to assess whether there is enough empirical evidence to claim with high confidence that the two success rates are indeed different. We apply a Fisher exact test at significance level $\alpha = 0.05$. If the p-value is above the chosen α level, then there would not be enough evidence to claim a difference in the success rates of $\mathcal{A}$ and $\mathcal{B}$. Still, the performance of the two algorithms can be statistically different, as we will now discuss in more detail.

In case there is no statistical difference in the success rates, we can analyze the *time* an algorithm takes to finding an optimal solution for the runs in which it is successful [8]. For example, assume that $a = b = n$, i.e., for the given budget of statements the algorithm finds an optimal solution in all the $2n$ runs. This would happen if the target branch is easy and/or the given computational budget is very high. In these cases, we might want to know how fast the algorithm finds a solution. This is of practical importance, because we can stop the search as soon as we find an optimal solution. For each run that leads to finding an optimal solution, we can monitor how much computational effort has been spent, measured in the number of statements executed before finding the optimal solution for each run. We can hence compare the computational effort of $\mathcal{A}$ (based on a observations/values) with the effort of $\mathcal{B}$ (based on b observations/values). As discussed in [8], we use a Mann-Whitney U-test (with $\alpha = 0.05$) to asses which configuration requires less computational effort to find optimal solutions.

5.2.3. Procedure for Whole Test Suite Generation

When we analyze the results for whole test suite generation, we need a slightly different procedure to compare $\mathcal{A}$ with $\mathcal{B}$. An algorithm is better than another if the test suites it generates have higher coverage. Differences in coverage are quantified with the Vargha-Delaney $\hat{A}_{12}$ effect size (see [8, 44]), which indicates the probability that an algorithm (e.g., $\mathcal{A}$) gives higher values (in this case the number of covered branches) than another algorithm (e.g., $\mathcal{B}$). If there is no difference between two algorithms, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ means that, in *all* of the n runs of the analyzed algorithm, we obtained coverage values higher than the ones obtained in *all* of the n runs of the other algorithm. When $\hat{A}_{12} \neq 0.5$, we evaluated whether the effect size is statistically significant with a Mann-Whitney U-test (with $\alpha = 0.05$).

Whenever there is no difference in the obtained coverage (i.e., $\hat{A}_{12} = 0.5$), then in general it is not reasonable to evaluate how long the algorithm took to obtain that coverage. In most cases, there are infeasible testing targets (branches in our context) and the search is stopped after either a predefined number of fitness evaluations or a timeout. Therefore, in general we do not know when an optimal solution has been found, and so stop the search. Even if an algorithm $\mathcal{A}$ is faster than $\mathcal{B}$ to achieve the same coverage, in practical contexts both would be run for the same time anyway. However, in these cases $\mathcal{A}$ and $\mathcal{B}$ can be differentiated based on the size of the generated test suites. If an algorithm generates smaller test suites but with the same coverage, then it can be considered as better. To evaluate whether differences in test suite sizes are different, we used a Mann-Whitney U-test (with $\alpha = 0.05$).

Notice that, when coverage values are different, it does not make much sense to consider the size of the test suites, because in general higher coverage might *require* longer test cases (e.g., some branches could be executed only if the internal state is put in a precise configuration, and if to do that the only way is a particular sequence of function calls). However, it is possible (and we have seen it in the experiments in some cases), that a technique not only achieves higher coverage, but at the same time it also generates smaller test suites.

5.2.4. Search Parameters

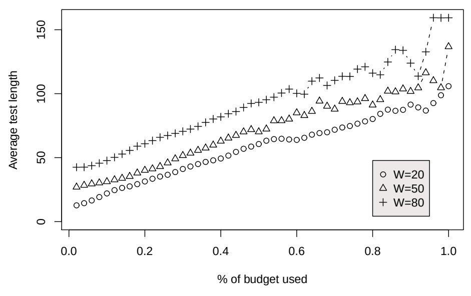
The genetic algorithm was configured with a population size of 100, and a rank bias of 1.7. The crossover probability was set to 0.75, and test case mutation is applied with probability 1/3 each for insertion, deletion, and change. When mutating test suites, the initial probability for test case insertion was set to $\sigma = 0.1$, while the initial insertion probability of inserting statements in test cases σ' was set to 0.5. The maximum number of test cases in a test suite was set to $N = 100$, although their initial number was set randomly in the range of [1,10] test cases for each test suite. These settings are in line with common practice in the literature and our past experience with genetic algorithms. In general, although "default" values as the ones above work well in practice, *parameter tuning* could lead to better results [45].

5.3. Bloat Control Techniques Illustrated

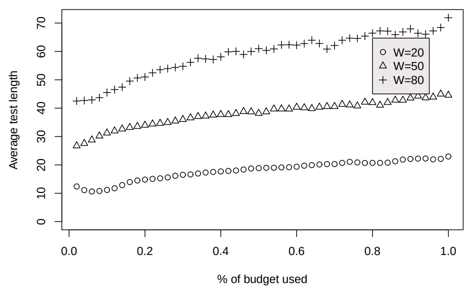
To illustrate the effects of the individual bloat control techniques, we performed a set of experiments on the branch used to generate the plot in Figure 1a. We generated test cases for this branch using 25 different random seeds and a maximum of 100,000 statements, and averaged the results. Figure 3a shows the behavior of the length without any bloat control techniques activated—the length grows, as expected. Figure 3b shows how the average test case length behaves over the evolution of test generation when we use RPX for the same branch. The average size of the test cases increases, but at a much slower rate than with TPX. Figure 3c shows how the average test case length converges when a fixed maximum length is used. Figure 3d shows how the average test case length first shrinks as the long individuals of the initial population are removed, and then grows only slowly. Figure 3e shows how the use of length in the rank reduces the average test case length for the usual example branch. Finally, Figure 3f shows how the average length increases slowly when using the parent check.

5.4. Analysis of Individual Bloat Control Techniques

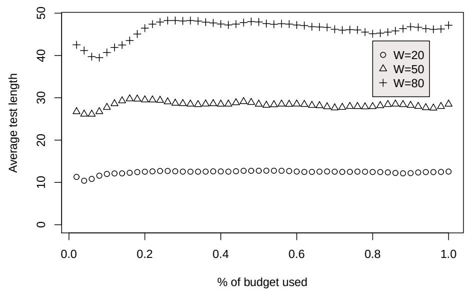
To study the effects of the individual bloat control techniques in detail, we ran a first set of experiments in which, for each $W \in \{20,50,80\}$, we ran a genetic algorithm with no bloat control activated (**No**) and with the five bloat control activated one at a time, for a total of $3 \times (1 + 5) = 18$ configurations for each branch (i.e., a subset of the total 96 configurations).



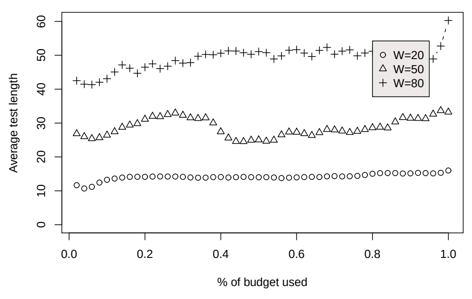
(a) No bloat control (**No**)



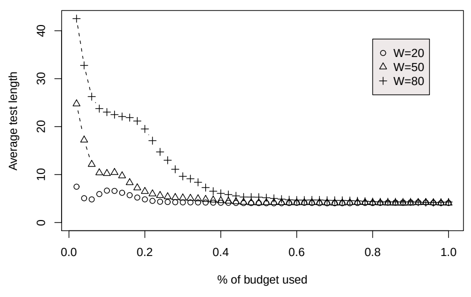
(b) RPX crossover function (**Xo**)



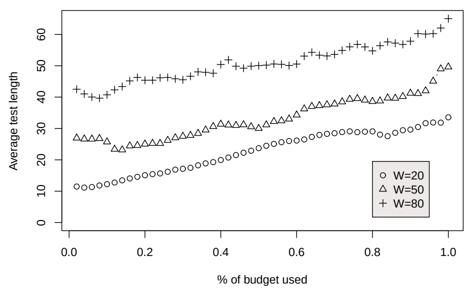
(c) Fixed upper bound (**Bo**)



(d) Check against best (**Be**)



(e) Length in rank (**Ra**)



(f) Check against parents (**Pa**)

Figure 3: Evolution for the same branch as in Figure 1a, using the different bloat control techniques one at a time. Evolution is bounded by 100,000 executed statements for these graphs, results are averaged for 25 runs. Note that y-axis have different scales to make the graphs more readable.

Table II. Comparisons (better, equivalent and worse) of bloat control methods when considering each method in isolation. Each method is compared with the other five, with three different values for W, on all the 100 problem instances. In total we have $5 \times 3 \times 100 = 1500$ comparison results per bloat control technique (i.e., per column).

	No	**Bo**	**Xo**	**Ra**	**Pa**	**Be**
Statistically Better	8	49	15	268	51	29
Statistically Equivalent	1347	1393	1392	1224	1402	1402
Statistically Worse	145	58	93	8	47	69

This first set of experiments is used to assess the implication of each bloat control method in isolation. In fact, the case of multiple combinations of bloat control methods is harder to analyze and visualize.

For each configuration and for each branch, we ran the genetic algorithm $n = 25$ times, for a total of $100 \times 18 \times 25 = 45{,}000$ runs. Figure 4 shows 18 boxplots, one for each analyzed configuration. Each boxplot shows the distribution of success rates on the 100 branches for that configuration. Table II and III summarize the statistical analyses we carried out on these data. In particular, in Table II for each $W \in \{20{,}50{,}80\}$ we report the results of the statistical comparisons of each configuration against the other five (for a total of $100 \times 6 \times 5 \times 3 = 9{,}000$ comparisons). On the other hand, in Table III we report the results of the statistical comparisons regarding the choice of the value $W \in \{20{,}50{,}80\}$. For the five bloat control and no control at all configurations, we compared each choice of W with the other two (hence $3 \times 2 = 6$ combinations), for a total of $100 \times 6 \times 6 = 3{,}600$ comparisons.

As we can see in those tables and figure, all the controlling bloat techniques have a beneficial effect for obtaining higher success rate. In particular, penalizing longer lengths in rank selection (**Ra**) seems to be the most effective technique regardless of the choice of W.

Regarding the choice of W, we do not see any particular trend in the data. Having short starting sequences ($W = 20$) or long ones ($W = 80$) can have an effect, but that is dependent on the chosen bloat control method (see Table III).

5.5. Investigations on All Bloat Control Techniques

There can be subtle interactions within the different bloat control methods when more than one is used at the same time. To study these interactions, we carried out the same type of experiments on the remaining $96 - 18$ configurations, for a total of $100 \times 96 \times 25 = 240{,}000$ runs of the algorithm. This is a very large set of experiments that took several days to complete even when run on a cluster of computers.

5.5.1. Analysis Procedure

To analyze and visualize the results of this large set of data, we used the following procedure: For each branch, we compared the effectiveness of each configuration against all other

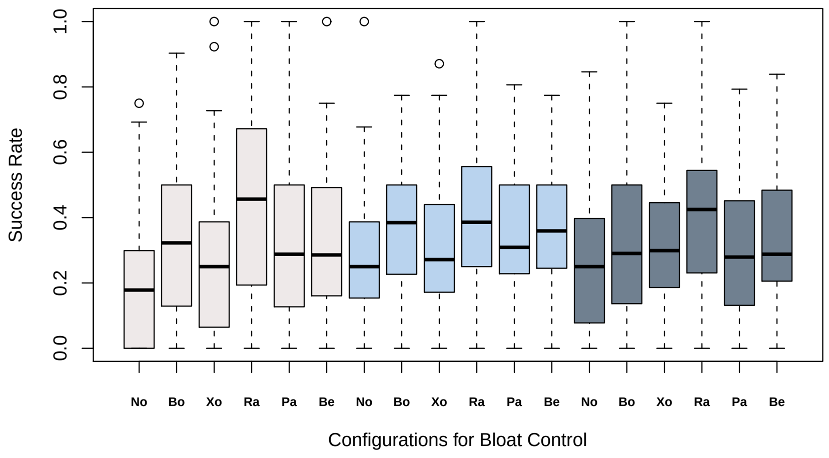


Figure 4: Success rate for 18 configurations, each applied on all the 100 branches. Left six boxplots are for $W = 20$, $W = 50$ in the centre and $W = 80$ for the six boxplots on the right of the figure.

Table III. For each bloat control method in isolation, this table reports the number of times a particular choice of W provides statistically better performance than any of the other two choices. Therefore, each cell can assume values between 0 (never better) and 2×100 (always better on all the 100 problem instances).

Bloat Control	$W = 20$	$W = 50$	$W = 80$
No	8	13	24
Bo	20	12	14
Xo	11	17	15
Ra	32	10	18
Pa	18	15	14
Be	16	17	14

configurations, one at a time (so, 96×95 comparisons, which can be reduced by half due to the symmetric property of the comparisons). Initially, we assign a score of 0 to each configuration. For each comparison in which a configuration is statistically better, we increase its score by one, and we reduce it by one in case it is statistically worse. Therefore, in the end each configuration has a score between -95 and 95. The higher the score, the better the configuration is.

After this first phase, we rank these scores, such that the highest score has the best rank, where better ranks have lower values. In case of ties, we average the ranks. For example, if we have five configurations with scores $\{10,0,0,20,-30\}$, then their ranks will be $\{2,3.5,3.5,1,5\}$. We repeat this procedure for all the 100 branches, and we calculate the average of these ranks for each configuration, for a total of $100 \times 96 \times 95/2 = 456{,}000$ statistical comparisons.

This a very large number of comparisons, which can lead to a high probability of Type I error [8] if we consider the hypothesis that *all* tests are significant at the same time. We do not use corrections such as the Bonferroni one, for reasons that are discussed in detail and at length in [8]. The configurations with lower average ranks can be considered better than the others. Table IV shows the performance of all the 96 configurations, ordered by their ranks.

5.5.2. Results

The results in Table IV confirm some of our hypotheses, but also point out some unexpected behaviors. The worst configuration is when no bloat control is activated, and the search starts from small lengths (see last row). This is a particular configuration, with an average success rate 0.190 that is much lower than the 0.464 of the top configuration.

On one hand, regarding the bloat control techniques, the one that has most effect is **Ra**. Activating **Ra** always produces better results, whatever the setting of the other parameters.

On the other hand, it came as a surprise that RPX is actually giving bad results (i.e., **Xo** does not appear in the top rankings). It seems that an unbalanced length crossover such as TPX, which can produce very long as well as very short test cases, is actually beneficial. We can provide a *conjecture* to give a plausible explanation to such an unexpected behavior: In some cases, longer test sequences can have more chances to achieve higher coverage [22], so sampling longer test sequences is beneficial. However, when during the search a fitness plateau is reached, longer test sequences would not have better fitness. Smaller offsprings generated with TPX will likely have the same fitness (plateau), but if **Ra** is activated they will be preferred. The search will hence have a sudden drift toward smaller test sequences. Smaller test sequences are quicker to evaluate, and so more generations would be possible. More generations would lead to a more focused search of the test data in input to those sequences, which might help to find the right input data to escape from the plateau. Although this is a plausible explanation, more research will be required to verify whether that is actual the case, and no other subtle dynamics are involved.

Regarding the other three bloat control methods, they seem beneficial, but they are not as important as **Ra**. When we look at the bottom of the table, it seems that **Pa** is better than **Bo** and **Be**. But at the top, there is not much difference.

The role of W is rather particular: In the best seven configurations, it is set to $W = 50$. But then, for other configurations it does not seem that the choice of W has any particular effect (i.e., we do not see any particular pattern in the data). At the moment, we do not know whether there is any specific reason for why that is the case.

Regarding the first three of our initially posed research questions, this leads us to the following conclusions:

RQ1: How does the maximum starting length W influence the search results?

The results of our experiments show that the choice of the maximum starting length W is not particularly important.

Our results have also clearly shown that all bloat control techniques have a large effect of the achieved coverage. In particular, in our experiments the configuration without bloat control was only successful in 19 out of the 100 branches on average, while the best configuration of bloat control techniques lead to coverage of 46 branches on average.

RQ2: How do the bloat control methods impact the achieved coverage?

Applying bloat control techniques increases coverage significantly.

Among the bloat control methods, **Ra** has a strong beneficial effect, whereas **Xo** decreases the performance. The other three methods are useful, but not as important as **Ra**. Because using only **Ra** alone does not give good enough results (see Table IV), based on our results we can suggest the practitioners to use all bloat control methods but **Xo** at the same time.

RQ3: Which techniques to control bloat are most effective?

Rank selection with length has the best effect, and should be used together with all other techniques but RPX.

5.6. Bloat in Whole Test Suite Evolution

Evolving whole test suites for testing object-oriented software is a alternative approach introduced in [6, 7] with the EvoSuite tool. Obviously, test suites are larger and consume more memory than single test cases. Therefore, it is necessary to study the role of bloat in whole test suite generation as well. In previous sections, we have evaluated different bloat control methods in the common context of targeting branches one at a time. It is hence important to see whether those results carry over to EvoSuite, or if evolving whole test suites requires different mechanisms to control bloat.

Table IV. Performance of the 96 configurations on single target strategy, ordered from top (best performance) to bottom (worst performance). Symbols are used to indicate whether a particular bloat control method is activated.

Bo	**Xo**	**Ra**	**Pa**	**Be**		W		Av. Rank	Av. Success Rate
					20	50	80		
△		⊕	▽	⊞		$\mathcal{W}$		31.475	0.464
△		⊕	▽			$\mathcal{W}$		31.840	0.456
△		⊕		⊞		$\mathcal{W}$		32.595	0.482
		⊕	▽	⊞		$\mathcal{W}$		32.670	0.456
		⊕	▽			$\mathcal{W}$		34.725	0.447
△		⊕				$\mathcal{W}$		35.415	0.448
		⊕		⊞		$\mathcal{W}$		36.070	0.442
△		⊕		⊞	$\mathcal{W}$			37.335	0.423
△	⊠	⊕	▽	⊞		$\mathcal{W}$		37.430	0.430
△		⊕		⊞			$\mathcal{W}$	37.605	0.459
	⊠	⊕		⊞	$\mathcal{W}$			37.615	0.418
△	⊠	⊕		⊞		$\mathcal{W}$		38.080	0.422
	⊠	⊕	▽	⊞		$\mathcal{W}$		39.325	0.419
	⊠	⊕		⊞		$\mathcal{W}$		39.455	0.423
	⊠	⊕	▽			$\mathcal{W}$		39.580	0.413
△		⊕			$\mathcal{W}$			39.790	0.431
		⊕		⊞	$\mathcal{W}$			39.815	0.431
	⊠	⊕			$\mathcal{W}$			40.050	0.414
△		⊕	▽		$\mathcal{W}$			40.140	0.420
△	⊠	⊕	▽			$\mathcal{W}$		40.330	0.425
△		⊕	▽	⊞	$\mathcal{W}$			40.670	0.413
△		⊕	▽	⊞			$\mathcal{W}$	40.700	0.432
△	⊠	⊕		⊞	$\mathcal{W}$			40.835	0.405
		⊕		⊞			$\mathcal{W}$	40.940	0.438
△		⊕	▽				$\mathcal{W}$	41.200	0.455
△	⊠	⊕			$\mathcal{W}$			41.350	0.410
		⊕	▽	⊞			$\mathcal{W}$	41.695	0.423
		⊕	▽	⊞	$\mathcal{W}$			41.890	0.405
		⊕	▽		$\mathcal{W}$			41.925	0.413
	⊠	⊕	▽		$\mathcal{W}$			42.150	0.399
	⊠	⊕	▽	⊞			$\mathcal{W}$	42.195	0.401
	⊠	⊕	▽	⊞	$\mathcal{W}$			42.470	0.388
△	⊠	⊕	▽		$\mathcal{W}$			42.500	0.395
	⊠	⊕		⊞			$\mathcal{W}$	42.800	0.422
		⊕			$\mathcal{W}$			43.075	0.407
	⊠	⊕				$\mathcal{W}$		43.095	0.421
△	⊠	⊕				$\mathcal{W}$		43.255	0.420
△	⊠	⊕	▽	⊞	$\mathcal{W}$			43.635	0.377
		⊕				$\mathcal{W}$		45.160	0.398
	⊠	⊕	▽				$\mathcal{W}$	45.205	0.393
		⊕	▽				$\mathcal{W}$	45.285	0.412
△	⊠	⊕	▽				$\mathcal{W}$	45.450	0.392
△		⊕					$\mathcal{W}$	45.850	0.418
		⊕					$\mathcal{W}$	46.460	0.401
△	⊠	⊕					$\mathcal{W}$	46.625	0.388
△	⊠	⊕		⊞			$\mathcal{W}$	46.700	0.409
△	⊠	⊕	▽	⊞			$\mathcal{W}$	47.760	0.379
	⊠	⊕					$\mathcal{W}$	47.850	0.384
△			▽	⊞		$\mathcal{W}$		48.985	0.342
			▽			$\mathcal{W}$		49.585	0.329
			▽	⊞		$\mathcal{W}$		49.705	0.334
△			▽	⊞	$\mathcal{W}$			49.995	0.369
△	⊠		▽	⊞		$\mathcal{W}$		50.290	0.313
△			▽		$\mathcal{W}$			50.740	0.356
△	⊠		▽			$\mathcal{W}$		51.295	0.313
△			▽			$\mathcal{W}$		51.350	0.340
△				⊞		$\mathcal{W}$		51.570	0.327
△			▽	⊞			$\mathcal{W}$	52.215	0.326
△				⊞	$\mathcal{W}$			52.800	0.330
			▽	⊞			$\mathcal{W}$	53.260	0.330
	⊠		▽	⊞		$\mathcal{W}$		53.610	0.309
△			▽				$\mathcal{W}$	53.845	0.321
	⊠		▽	⊞	$\mathcal{W}$			54.040	0.310
	⊠		▽				$\mathcal{W}$	54.475	0.312
			▽	⊞	$\mathcal{W}$			54.835	0.296
			▽		$\mathcal{W}$			55.080	0.306
				⊞		$\mathcal{W}$		55.290	0.317
	⊠		▽			$\mathcal{W}$		55.390	0.313
	⊠		▽	⊞			$\mathcal{W}$	55.605	0.304
△					$\mathcal{W}$			55.635	0.305
			▽				$\mathcal{W}$	55.695	0.324
△	⊠		▽		$\mathcal{W}$			56.065	0.310
△						$\mathcal{W}$		56.160	0.309
	⊠			⊞		$\mathcal{W}$		56.200	0.304
△	⊠		▽	⊞			$\mathcal{W}$	56.255	0.301
	⊠		▽		$\mathcal{W}$			56.295	0.312
△	⊠		▽	⊞	$\mathcal{W}$			56.655	0.312
△	⊠		▽				$\mathcal{W}$	56.835	0.291
△	⊠					$\mathcal{W}$		57.095	0.279
△	⊠			⊞		$\mathcal{W}$		57.135	0.291
△				⊞			$\mathcal{W}$	57.180	0.319
				⊞			$\mathcal{W}$	57.390	0.306
							$\mathcal{W}$	58.955	0.285
△	⊠			⊞			$\mathcal{W}$	59.085	0.297
				⊞	$\mathcal{W}$			59.190	0.297
△	⊠			⊞	$\mathcal{W}$			59.270	0.285
	⊠					$\mathcal{W}$		59.595	0.279
△							$\mathcal{W}$	59.995	0.300
	⊠			⊞	$\mathcal{W}$			60.145	0.281
	⊠						$\mathcal{W}$	60.150	0.289
△	⊠				$\mathcal{W}$			60.675	0.278
	⊠			⊞			$\mathcal{W}$	60.705	0.289
△	⊠						$\mathcal{W}$	60.975	0.292
						$\mathcal{W}$		61.655	0.267
	⊠				$\mathcal{W}$			65.220	0.238
					$\mathcal{W}$			71.765	0.190

5.6.1. Analysis Procedure

For each of the 39 classes selected as described in Section 5.1, we ran EvoSuite with the same 96 configurations used in the previous experiments in Section 5.5; the data were also analyzed in a similar way. For each class, we compared the effectiveness of each configuration against all other configurations, one at a time, and calculated a performance rank per configuration. These ranks were averaged over the 39 classes. In total, we had $39 \times 96 \times 95/2 = 177{,}840$ statistical comparisons. Data of these analyses are visualized in Table V, in a similar way as previously done in Table IV.

Recall that, when we evolve single test cases to cover specific branches (as done in the experiments discussed in the previous sections), bloat control methods aim to keep the length of these test sequences under control. On the other hand, when we evolve whole test suites, we try to keep under control their size, defined as the sum of the lengths of each of their test cases.

5.6.2. Results

Unexpectedly, the data for EvoSuite in Table V show quite a few differences compared to the previous experiments in Table IV for single test cases. Regarding the obtained coverage averaged across the experiments, there is a large gap from the best configurations (around 87% of coverage) and the worst (around 56%). Although controlling bloat has a positive impact, there is a clear trend in the performance based on the starting length ranges W. For EvoSuite, we have that W is quite important, and this is in clear contrast to the answer we gave for **RQ1** regarding single test cases in Section 5.5. In particular, the performance of EvoSuite is better for shorter lengths.

In contrast with previous experiments, not only now RPX is better than TPX (i.e., **Xo** is activated in *all* the top 29 configurations), but its influence on the performance is even higher than **Bo** and **Be**. Enforcing a check on the parents lengths (**Pa**) has a harmful effect, although small (e.g., **Pa** does not appear in the best configuration, and it is set in the six worst configurations). Finally, although **Ra** is still the most important bloat control method, it does not always give better results (as it was in Table IV).

The search problem addressed in EvoSuite is very complex, and the software used as case study was non-trivial. Therefore, it is not possible to provide *exact* explanations (e.g., formal proofs) for the results presented in Table V. However, a reasonable explanation to interpret some of these data is as follows. As it was originally presented in [6, 7], the only operator in EvoSuite to decrease the length of a test case is through mutation, where a removal happens with probability 1/3, and on average only one statement is removed (see Section 3.5). A random test suite (e.g., in the first generation of the genetic algorithm) would be quite large and time consuming to evaluate. If for example the optimal length for a test case is 15, then starting from long test sequences (e.g., $W = 80$) would require a lot of generations before reducing the length. In all these generations, evaluating so big test suites could consume most of the search budget. Having long test sequences is important, but even more important is it to have mechanisms to increase and decrease them when most appropriate.

Table V. Performance of the 96 configurations for EvoSuite, ordered from top (best performance) to bottom (worst performance). Symbols are used to indicate whether a particular bloat control method is activated.

Bo	**Xo**	**Ra**	**Pa**	**Be**		W		Av. Rank	Av. Coverage
					20	50	80		
△	⊠	⊕		⊞	$\mathcal{W}$			5.987	0.867
△	⊠	⊕	▽	⊞	$\mathcal{W}$			6.833	0.860
△	⊠	⊕	▽		$\mathcal{W}$			6.936	0.860
△	⊠	⊕			$\mathcal{W}$			7.167	0.867
	⊠	⊕		⊞	$\mathcal{W}$			7.487	0.870
	⊠	⊕			$\mathcal{W}$			8.769	0.866
	⊠	⊕	▽	⊞	$\mathcal{W}$			9.423	0.858
	⊠	⊕	▽		$\mathcal{W}$			9.590	0.859
△	⊠	⊕		⊞		$\mathcal{W}$		11.808	0.857
△	⊠	⊕	▽			$\mathcal{W}$		13.679	0.849
	⊠	⊕		⊞		$\mathcal{W}$		13.756	0.855
	⊠	⊕	▽	⊞		$\mathcal{W}$		14.013	0.848
△	⊠	⊕	▽	⊞		$\mathcal{W}$		14.064	0.848
	⊠	⊕				$\mathcal{W}$		14.423	0.855
△	⊠	⊕				$\mathcal{W}$		14.590	0.858
	⊠	⊕	▽			$\mathcal{W}$		15.846	0.848
△	⊠	⊕		⊞			$\mathcal{W}$	23.897	0.837
	⊠	⊕		⊞			$\mathcal{W}$	25.385	0.837
	⊠	⊕	▽	⊞			$\mathcal{W}$	25.679	0.829
△	⊠	⊕					$\mathcal{W}$	25.949	0.838
	⊠	⊕					$\mathcal{W}$	26.154	0.837
△	⊠	⊕	▽				$\mathcal{W}$	26.833	0.831
△	⊠	⊕	▽	⊞			$\mathcal{W}$	27.397	0.828
	⊠	⊕	▽				$\mathcal{W}$	27.987	0.828
△	⊠		▽	⊞	$\mathcal{W}$			33.154	0.835
△	⊠		▽		$\mathcal{W}$			33.397	0.834
	⊠		▽	⊞	$\mathcal{W}$			36.090	0.832
	⊠		▽		$\mathcal{W}$			36.218	0.832
	⊠			⊞	$\mathcal{W}$			37.590	0.847
△		⊕		⊞	$\mathcal{W}$			37.705	0.747
	⊠				$\mathcal{W}$			37.897	0.846
		⊕	▽	⊞	$\mathcal{W}$			38.000	0.746
		⊕		⊞	$\mathcal{W}$			38.641	0.746
△		⊕	▽	⊞	$\mathcal{W}$			39.051	0.746
△	⊠				$\mathcal{W}$			41.551	0.836
△		⊕	▽	⊞		$\mathcal{W}$		41.705	0.742
△	⊠			⊞	$\mathcal{W}$			42.077	0.833
		⊕		⊞		$\mathcal{W}$		43.474	0.741
△	⊠			⊞		$\mathcal{W}$		44.628	0.835
	⊠			⊞		$\mathcal{W}$		44.987	0.833
△	⊠					$\mathcal{W}$		45.218	0.835
		⊕	▽	⊞		$\mathcal{W}$		45.590	0.741
	⊠					$\mathcal{W}$		45.679	0.831
△		⊕	▽		$\mathcal{W}$			45.923	0.719
△		⊕		⊞		$\mathcal{W}$		46.077	0.739
		⊕			$\mathcal{W}$			46.179	0.720
	⊠		▽			$\mathcal{W}$		47.026	0.808
△		⊕			$\mathcal{W}$			47.359	0.721
	⊠		▽	⊞		$\mathcal{W}$		47.628	0.804
△	⊠		▽			$\mathcal{W}$		47.769	0.803
		⊕	▽		$\mathcal{W}$			48.423	0.719
△	⊠		▽	⊞		$\mathcal{W}$		48.513	0.800
△		⊕				$\mathcal{W}$		50.192	0.715
△		⊕	▽	⊞			$\mathcal{W}$	50.756	0.726
		⊕				$\mathcal{W}$		51.038	0.714
△		⊕		⊞			$\mathcal{W}$	51.051	0.729
		⊕		⊞			$\mathcal{W}$	51.192	0.730
△		⊕	▽			$\mathcal{W}$		51.346	0.717
		⊕	▽			$\mathcal{W}$		52.269	0.716
	⊠			⊞			$\mathcal{W}$	53.487	0.818
		⊕	▽	⊞			$\mathcal{W}$	53.551	0.726
△	⊠			⊞			$\mathcal{W}$	54.385	0.818
	⊠						$\mathcal{W}$	54.705	0.816
△	⊠						$\mathcal{W}$	55.128	0.816
△	⊠		▽				$\mathcal{W}$	57.462	0.772
	⊠		▽				$\mathcal{W}$	57.538	0.771
△	⊠		▽	⊞			$\mathcal{W}$	57.654	0.774
	⊠		▽	⊞			$\mathcal{W}$	57.782	0.772
		⊕	▽				$\mathcal{W}$	58.397	0.704
		⊕					$\mathcal{W}$	58.628	0.704
△		⊕					$\mathcal{W}$	59.192	0.702
△		⊕	▽				$\mathcal{W}$	59.218	0.702
				⊞	$\mathcal{W}$			72.974	0.638
△				⊞		$\mathcal{W}$		74.256	0.639
△				⊞	$\mathcal{W}$			74.321	0.634
				⊞		$\mathcal{W}$		74.910	0.640
△				⊞			$\mathcal{W}$	76.795	0.631
				⊞			$\mathcal{W}$	76.897	0.634
			▽	⊞		$\mathcal{W}$		80.718	0.617
			▽	⊞	$\mathcal{W}$			81.859	0.612
△			▽	⊞	$\mathcal{W}$			82.256	0.611
△			▽	⊞		$\mathcal{W}$		82.667	0.617
			▽	⊞			$\mathcal{W}$	83.564	0.611
						$\mathcal{W}$		83.590	0.589
							$\mathcal{W}$	83.692	0.587
					$\mathcal{W}$			83.782	0.584
△					$\mathcal{W}$			84.167	0.578
△						$\mathcal{W}$		84.205	0.587
△			▽	⊞			$\mathcal{W}$	84.231	0.607
△							$\mathcal{W}$	84.679	0.583
			▽			$\mathcal{W}$		88.333	0.578
			▽				$\mathcal{W}$	89.628	0.572
△			▽				$\mathcal{W}$	90.154	0.574
			▽		$\mathcal{W}$			90.269	0.567
△			▽			$\mathcal{W}$		90.449	0.575
△			▽		$\mathcal{W}$			91.423	0.560

The great difference in performance between RPX and TPX at test suite level in comparison with test case level can be explained as follows. An unbalanced crossover operator such as TPX can generate a long test case, and such a *new* test case might cover branches that were uncovered before. On the other hand, a crossover operator at test suite level never produces any new test cases. Even if a crossover operator can produce an offspring test suite that is better than both its parents (i.e., by choosing and combine the right test cases), its importance can decrease during the generations. In other words, when after some generations all the test suites achieve similar coverage, crossover loses its power to improve the coverage of a test suite. In this context, TPX is particularly harmful, as bigger test suites will be more expensive to evaluate (with no possible higher coverage), and too small test suites might have lower coverage.

Why does activating **Pa** lead to worse results? It might be related to the crossover operator properties. The used crossover operator only swaps test cases between the parent test suites, so it does not modify their combined total size. As it does not produce any new test case, its effects on fitness might be minimal in the later stages of the search. Search will be driven only by the mutation operator. But, if **Pa** is activated, small modifications that insert new statements but that do not improve fitness will be rejected. So, the search might be too restricted (i.e., not allowing the exploration of fitness plateaus, which would lead to the search getting stuck in local optima). On the other hand, in the one branch at a time approach, the crossover could make large changes (many new statements inserted) to an individual test case. So, potentially it could improve fitness in one single step in the cases where small steps do not have gradient.

The analysis of these data (Table V compared to Table IV) shows that there can be subtle interactions among the different techniques to control bloat. Regarding EvoSuite, for future work it might be a reasonable idea to use a crossover both at test suite and at test case levels. But in this case, considering all the subtle interactions among parameters, we would need to repeat these experiments (i.e., different combinations of bloat control methods), as new unexpected behaviors could arise.

Another interesting difference between Table V and Table IV is the range of the ranks. In Table IV the average ranks go from 31.4 to 71.7, whereas for Table V the ranges go from 5.9 to 91.4. Although there might be several explanations (e.g., strong variance in the relative performance of the configurations among different case study artifacts), the most likely one is related to the statistical power of the employed statistical tests. In fact, we state that a configuration is better than another one (and so better, lower rank) only if there is enough statistical evidence to claim it (in our case if the obtained p-values are lower than $\alpha = 0.05$). In general, given the same amount of data, a Fisher exact test (used for the analyses in Table IV regarding success rates) produces higher p-values than a Mann-Whitney U-test (used in Table V for comparisons of coverage values).

In summary, the impact of bloat control methods is significant: In our experiments, the configuration without bloat control achieved an average coverage of only 56%, whereas the best configuration of bloat control techniques lead to 86.7% coverage.

RQ4: How do the bloat control methods affect whole test suite evolution?

Bloat control methods significantly improve performance, but there are subtle interactions among the different methods.

5.7. Generalization

A common problem in empirical research is the generalization of the obtained results. When the proposed techniques are applied to new instances that were not present in the original case study (e.g., a practitioner that uses a research prototype in practice), will these techniques still behave in a similar way? Or are we going to obtain very poor results? In general, to cope with these threats to external validity, case studies should be large and representative of the addressed problem (for more discussions on this topic tailored to software testing, see [2]).

5.7.1. Analysis Procedure

Given an empirical analysis in which several techniques/configurations are compared with the aim of identifying which one is the best, there is the problem of *over-fitting*. In other words, there might be a specific configuration that is too specialized for the case study at hand. It will be the best for the case study, but then it would have poor performance on any other instance. This is a typical problem in machine learning [46] (e.g., in the training of regressors and classifiers). It is also a general problem in search-based software engineering, where *parameter tuning* (e.g., choice of population size and crossover rate in a genetic algorithm) could lead to over-fitted parameters [45]. Because bloat control techniques can be seen as parameters of the search algorithm, we follow the guidelines in [45] to analyze possible issues with over-fitting. This is done to increase our confidence in the external validity (i.e., generalization to other problem instances) of the results discussed in the previous sections.

As discussed in [45], a common approach in machine learning to address generalization issues is *k-fold cross validation* [46]. In our empirical analysis, we have two distinct data sets. The first data set was generated from applying a search for single targets on 100 different branches, whereas the second data set by applying EvoSuite on 39 classes. In both cases, 96 bloat control configurations were analyzed and compared in the previous section. We apply k-fold cross validation on the two data sets separately. Before discussing the results of these analyses, we briefly explain how k-fold validation works. For more information, we refer to machine learning text books such as [46].

To apply k-fold cross validation, we randomly partitioned the case study (e.g., in our two cases, 100 branches and 39 classes) in k non-overlapping subsets (a common value, which we use in this paper, is $k = 10$). We used one of these groups as *test set*, and merged the other $k-1$ subsets to use them as *training set*. We applied the tuning (i.e., identify which combination of bloat control techniques gives best results) *only* on the training set, and then evaluate the performance on the test set (the instances in the test set are not used and play no role in the decision of which configuration is tagged as best). We repeated this process k times, every time with a different subset for the test set, and remaining $k-1$ for the training set. We

Table VI. *K*-fold analysis for one branch at a time approach. For each group in *k*-fold analysis, reported best configuration chosen on training set. Results (average success rate) are calculated on both the training set (Training) and the test set for validation (Validation).

Group	**Bo**	**Xo**	**Ra**	**Pa**	**Be**		W		Training	Validation
						20	50	80		
1	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.446	0.639
2	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.466	0.503
3	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.474	0.444
4	$\triangle$		$\oplus$		$\boxplus$		$\mathcal{W}$		0.500	0.351
5	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.472	0.459
6	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.489	0.216
7	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.476	0.404
8	$\triangle$		$\oplus$	$\triangledown$			$\mathcal{W}$		0.471	0.368
9	$\triangle$		$\oplus$	$\triangledown$	$\boxplus$		$\mathcal{W}$		0.471	0.466
10	$\triangle$		$\oplus$	$\triangledown$			$\mathcal{W}$		0.452	0.594
Average									0.472	0.444

Table VII. *K*-fold analysis for whole test suite approach (i.e., EvoSuite). For each group in *k*-fold analysis, reported best configuration chosen on training set. Results (average coverage) are calculated on both the training set (Training) and the test set for validation (Validation).

Group	**Bo**	**Xo**	**Ra**	**Pa**	**Be**		W		Training	Validation
						20	50	80		
1	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.864	0.896
2	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.862	0.915
3	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.858	0.950
4	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.864	0.898
5	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.866	0.873
6	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.869	0.854
7	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.890	0.671
8	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.866	0.880
9	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.876	0.786
10	$\triangle$	$\boxtimes$	$\oplus$		$\boxplus$	$\mathcal{W}$			0.862	0.965
Average									0.868	0.869

averaged the performance on all the results obtained from all the *k* test sets, which gives a value describing the performance of the algorithm (e.g., success rate for the 100 branches, and coverage for the 39 classes).

These values averaged from the test sets can be used to estimate how well a configuration tuned on the entire data set would perform on new data. In particular, this process is a way to see whether the chosen best configuration suffers of over-fitting problems.

5.7.2. Results

The data in Table VI and Table VII, compared with the first rows (i.e., best configurations) in Table IV and Table V, clearly show that the results are consistent. The choice of the best bloat control methods does not over-fit the data.

RQ5: How likely are the presented results to generalize to other case studies?

The choice of best bloat control methods does not *over-fit the used case study.*

6. Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 25 times, and we followed rigorous statistical procedures to evaluate their results.

The selection of the set of 100 difficult branches (and their respective 39 classes) was based on only one run for practical reasons (total number of branches was more than 15,000). Therefore, it might be possible that some of them would not be difficult on average, and we could have missed some other difficult branches. However, once the set was selected, all experiments on that set were valid and independent from the chosen selection mechanism.

We used both open source projects and industrial software as case studies, for a total of nearly 1,000 classes. We selected different types of applications, such as for example implementations of data structures, complex manipulations of string data and numerical applications. Nevertheless, there is still the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Furthermore, due to the large amount of experiments, only 100 branches and 39 classes were used as case study. To reduce possible issues regarding conclusions that over-fit the data used in the empirical analysis, we employed machine learning techniques such as k-fold cross validation.

7. Conclusions

Evolutionary search with variable size representation is susceptible to *bloat*—that is, a disproportional growth of the length of individuals that quickly uses up all resources and so seriously harming the search. Unfortunately, this also means it applies to search-based testing for object-oriented software, although this has not been sufficiently treated in the literature so far.

In this paper, we performed a set of experiments, using a genetic algorithm, on the properties of test sequence *length* and how to counter the effects of length *bloat* in the context of branch coverage. We studied two different but related testing strategies: generating test sequences to cover specific branches one at a time (as commonly done in the literature), and evolving whole test suites (an alternative promising approach introduced in EvoSuite [6, 7])

Interestingly, our results showed that there are complex subtle interactions among the bloat control methods and, if bloat is not properly taken care of, then there is the danger of running into problems such as using up all memory and drastically increasing execution times. However, our experiments showed that the success rate and coverage for the same amount of resources are *significantly* higher when applying the bloat control techniques described in this paper.

Our experiments clearly point to which bloat control techniques to use and which ones should not be used in practical contexts. In particular:

- In a scenario where coverage goals are targeted one at a time the recommended configuration is to use all presented bloat control techniques except for relative position crossover, while the starting length is of minor importance.
- In a scenario of whole test suite generation, all bloat control techniques except for the parent check are recommended; the starting length in this scenario should be short (e.g., 20).

To support our claims, we carried out a very large empirical study on several types of applications (e.g., data structures, string processing, numerical applications), using both open source and industrial software. A rigorous statistical method (i.e., experiments repeated 25 times, statistical tests and k-fold cross validation) has been employed to verify with high confidence that our results are scientifically sound.

As future work, we plan to repeat our experiments with a larger set of case studies (e.g., the SF100 Corpus [47]), and include further coverage criteria in addition to branch coverage. Furthermore, we will need to consider the effects on length bloat of other parameters such as the population size in genetic algorithms (e.g., to avoid running out of memory, we should study dynamic techniques to reduce population size during search if single individuals get too bloated). As our experiments show that the crossover operator can have a large impact on bloat and subtle interactions with other bloat control methods, further developments in EvoSuite (e.g., crossover operators that work both at test case and test suite levels) will require dedicated analyses on their effects on test length bloat.

To learn more about EvoSuite, visit our Web site:

http://www.evosuite.org/

Acknowledgments. This work has been supported by a Google Focused Research Award on "Test Amplification". Andrea Arcuri is funded by the Norwegian Research Council.

References

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

[2] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)*, 36(6):742–762, 2010.

[3] S. Silva and E. Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.

[4] G. Fraser and A. Arcuri. It is not the length that matters, it is how you control it. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 150 – 159, 2011.

[5] P. Tonella. Evolutionary testing of classes. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[6] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40. IEEE Computer Society, 2011.

[7] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

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

[9] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 147–158. ACM, 2010.

[10] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):438–444, 1984.

[11] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):258–277, 2012.

[12] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19 (7):385–394, 1976.

[13] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1053–1060. ACM, 2005.

[14] José Carlos Bregieiro Ribeiro. Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1819–1822. ACM, 2008.

[15] K. Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE'08: Proc. of the 23rd IEEE/ACM Int. Conference on Automated Software Engineering*, pages 297–306, 2008.

[16] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.

[17] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008. ISSN 0020-0255.

[18] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.

[19] J. H. Andrews, T. Menzies, and F. C.H. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering (TSE)*, 37(1):80–94, 2011.

[20] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms.* John Wiley and Sons, 2001.

[21] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering (TSE)*, 36 (2):226–247, 2010.

[22] A. Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering (TSE)*, 38(3):497–519, 2012.

[23] J. H. Andrews, A. Groce, M. Weston, and R. G. Xu. Random test run length and effectiveness. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 19–28, 2008.

[24] G. Fraser and A. Gargantini. Experiments on the test case length in specification based test case generation. In *International Workshop on Automation in Software Test (AST)*, 2009.

[25] D.B. Fogel. What is evolutionary computation? *Spectrum, IEEE*, 37(2):26–28, 2000.

[26] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.

[27] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[28] W.B. Langdon. The evolution of size in variable length representations. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 633–638. IEEE, 1998.

[29] W.A. Tackett. *Recombination, selection, and the genetic construction of computer programs.* PhD thesis, University of Southern California, 1994.

[30] P.W.H. Smith and K. Harries. Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation*, 6(4):339–360, 1998.

[31] T. Soule. *Code growth in genetic programming.* PhD thesis, University of Idaho, 1998.

[32] WB Langdon, T. Soule, R. Poli, and JA Foster. The evolution of size and shape. *Advances in genetic programming*, 3:163–190, 1999.

[33] T. Soule and J.A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 781–786. IEEE, 1998.

[34] W. Langdon and R. Poli. Fitness causes bloat: Mutation. *Genetic Programming*, pages 37–48, 1998.

[35] S. Luke. Modification point depth and genome growth in genetic programming. *Evolutionary Computation*, 11(1):67–106, 2003.

[36] S. Dignum and R. Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1588–1595. ACM, 2007.

[37] S. Dignum and R. Poli. Crossover, sampling, bloat and the harmful effects of size limits. *Genetic Programming*, pages 158–169, 2008.

[38] D. Whitley. The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, pages 116–121, 1989.

[39] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability (STVR)*, 23(2):119–147, 2013.

[40] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA'07: Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Application*, pages 815–816. ACM, 2007.

[41] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability*, 16(3):175–203, 2006. ISSN 0960-0833. doi: http://dx.doi.org/10.1002/stvr.v16:3.

[42] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 265–275, 2011.

[43] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *IFIP International Conference on Testing Software and Systems (ICTSS)*, pages 95–110, 2010.

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

[45] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 33–47, 2011.

[46] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[47] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.