Longer is Better: On the Role of Test Sequence Length in Software Testing

Andrea Arcuri

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

Abstract

In the presence of an internal state, often it is required a sequence of function calls to test software. In fact, to cover a particular branch of the code, a sequence of previous function calls might be required to put the internal state in the appropriate configuration. Internal states are not only present in object-oriented software, but also in procedural software (e.g., static variables in C programs). In literature, there are many techniques to test this type of software. However, to our best knowledge, the properties related to choosing the length of these sequences have received only little attention in literature. In this paper, we analyse the role that the length plays in software testing. We show that on "difficult" software testing benchmarks *longer* test sequences make their testing trivial. Hence, we argue that the choice of the length of the test sequences is very important in software testing.

Keyword: Evolutionary Testing, Object-Oriented Software, State Problem, Search Based Software Engineering, Software Testing

1 Introduction

Software testing for white box criteria (e.g., branch coverage) consists of finding a set of test cases (i.e., the test suite) that maximises those criteria. Often, a test case is a driver that calls the function under test with a particular set of input values. The driver then compares the obtained output against the expected one. Using all possible inputs is unfeasible because their number is infinite. Hence, the automation of software testing in this context consists of automatically finding the smallest set of these inputs such that the testing criterion is maximised.

More problems arise if the the software has an internal state. An internal state can be for example static variables in C programs. In object-oriented (OO) software, most programs have internal states. Internal states are problematic because the coverage of some code structures can depend on the current status of the internal state. To put the internal state in the right configuration, a sequence of function calls is often required.

Figure 1 shows a simple example of a bank account (taken from [6]). To obtain full coverage, the method **withdraw** needs to be called at least ten times to put the variable **numberOfWithdrawals** (i.e., the internal state) in the right configuration (i.e., a value bigger or equal than ten).

On a new unknown testing problem, we would not know how many function calls we need to obtain full coverage. Ideally, the shortest length should be preferred. A short test sequence is not only quicker to compute, but it would also be easier to understand when the oracle (used to obtain the expected output) is a software developers. On one hand, choosing a fixed length has the

disadvantage that it could be too short to cover the target. On the other hand, *searching* for the best length complicates the automation of software testing.

In this paper, we analyse the role that length of test sequences plays in testing software with internal state. In our analysis, we consider search based techniques [5] for testing software. In particular, we compare four different search algorithms: Random Search (RS), Hill Climbing (HC), (1+1) Evolutionary Algorithm (EA) and a Genetic Algorithm (GA). We analyse how the performance of these techniques changes when different lengths for the test sequences are used. We compared searches when different lengths are fixed against searches when the length is an objective to optimise.

We use the *de facto* benchmark for testing software with internal state, i.e. *container classes* [9, 13, 14, 10, 11, 6, 3, 4, 1]. We consider the language Java. The goal is to obtain full branch coverage. Because we are only interested in studying the role of the sequence length, we only consider the most difficult methods [3], i.e. the ones related to the insertion and removal of objects in the containers (and all the methods that can be called through them).

During our analysis of RS, we noticed a very interesting property of this type of software. The ratio of optimal solutions over their total number drastically increases for longer sequences. Therefore, it is much easier to find an optimal solution when the length is longer. A simple post-processing routine can then reduce the length of these optimal solutions. Based on this simple property, even a naive random search can produce full coverage in a fraction of second. The most difficult branches can be covered in average in less than 100 test sequence evaluations.

In literature, the role of the length of test sequences has not received enough attention. In empirical studies often this parameter is not tuned, and its value often is not even specified (e.g., [9, 11]). This paper gives the important contribution of highlighting its importance. Most current techniques in literature can exploit this simple property to improve their performance.

We show that *longer* sequences are *better* for testing containers. Although containers are the de facto benchmark in literature, it is not necessarily true that this property would hold for real-world software. Therefore, we formally analyse the general conditions for which this property holds.

This paper gives the following contributions:

- We give an analysis of how the length of test sequences can influence the final results.
- Based on a simple property that we found in our analyses, we propose a simple technique that can obtain optimal results in a fraction of time compared to the techniques described in literature. This simple property can be exploited by most of these techniques.
- To make our results of more general applicability, we formally analyse the general conditions for which this property holds.
- In the case of search based techniques, we analyse what is the impact of searching for the best sequence length instead of just using a fixed value.

The paper is organised as follows. Section 2 briefly describes the related work. A description of the addressed problem follows in Section 3. The employed search algorithms are discussed in Section 4. Section 5 presents the case study on which we carried out our empirical investigation. A formal theoretical analysis of the role of test sequence length follows in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

Some of the tools in literature do an exhaustive search of the test sequences [13, 14, 10]. They first analyse all the sequences of length 1, then the ones of length 2, and so on. Because the search

```
public class BankAccount {
   private double balance;
   private int numberOfWithdrawals;
   public void deposit(double amount) {
    if(amount> 0.00)
      balance = balance + amount;
   }
   public void withdraw(double amount) {
    if(amount > balance) {
      printError();
      return;
    }
    if(numberOfWithdrawals >= 10) {
      printError();
      return;
    }
    dispense(amount);
    balance = balance - amount;
    numberOfWithdrawals++;
   }
}
```

Figure 1: An example of program with internal state [6].

space is exponential, they use different techniques to prune the search space, like for example state equivalence [13, 10] and subsumption [14, 10]. Because the search is heuristically pruned, these tools cannot guarantee that they would found the shortest solution (as it would be if no pruning was done). These techniques struggle when the required test sequence to find is long.

Evolutionary techniques have been used in the context of testing OO software [9]. A *population* of test sequences is *evolved* using evolutionary inspired operators like *mutation* and *crossover*. A mutation can be for example replacing the value of an input variable. Through a sequence of generations, only the *fittest* individuals survive and procreate. The fitness can be calculated with heuristics based on the execution of the test case. Typical heuristics in literature are for example the *approach level* and the *branch distance* [8]. However, in [9] only the approach level was used. The employed search operators can change the length of test sequences, but there is not attempt to minimise it.

Similar work has been done using Genetic Programming [11]. Again, the length of the test sequences could change, but their minimisation is not included in the fitness function.

An hybrid approach has been studied in [6]. On one hand, the structure of the test sequences is sought with an evolutionary algorithm to avoid the problems faced by techniques like [13, 14, 10]. On the other hand, the inputs for the methods are obtained with symbolic execution to overcome the "difficulties" of evolutionary testing. However, the authors consider only the work in [9], which does not exploit any branch distance (which is a fundamental component to search for the method inputs [8]). Like in [9], the test sequence length can change during the evolution but the fitness function does not consider it. The initial population is non-randomly seeded with a constructive heuristic.

In our previous work [3, 1], we analysed different search algorithms for testing OO software. The length of the test sequence was included in the fitness function, but we did not analyse its role.

Table 1: Authors, year of publication and percent of used containers in the case studies of the described related work.

Authors	Year	Containers
[9] Tonella	2004	83%
[13] Xie *et al.*	2004	81%
[14] Xie *et al.*	2005	100%
[10] Visser *et al.*	2006	100%
[11] Wappler and Wegener	2006	100%
[6] Inkumsah and Xie	2008	84%
[3] Arcuri and Yao	2008	100%
[4] Ciupa *et al.*	2008	75%
[1] Arcuri	2009	100%

For example, we did not analyse different length seeding strategies or whether using the length as a second objective could harm the search.

Although it can sound as a naive technique, random testing can achieve good results [4]. Its performance can be further improved when input objects are more evenly sampled according to a specific distance [4]. Again, the issues related to the sequence length are not considered.

The related work presented in this section is a good representative of the current state of the art, although it is not complete by any mean. A survey of the current state of the art of literature is not in the scope of this paper. However, it is interesting to note that container classes are dominant in these case studies. Table 1 summarises their properties. In all this related work, the case studies are composed of containers for at least the 75% of the analysed software.

3 Problem Representation

A test sequence can be arbitrarily complicated. To test a function in OO software, an object of its class needs to be instantiated. Different constructors and super-classes can be available. Each function required in the test sequence could take objects as input. These objects not only need to be instantiated, but they could require their own sequences of function calls to put their internal states in the appropriate configuration.

In this paper we do not deal with these problems. Our goal is to study the role of the length, not the complexity that test sequences can assume. For this reason, in the case of container classes we just use the default constructors. A test sequence is hence a ordered set of method calls on a new container instance. This makes possible to focus on the length of test sequences by removing all the other factors that can influence the results.

We just limit our analysis on methods regarding insertion and removal of objects in a container (and all the methods that can be called through them). Based on our previous analyses, other methods are easier to cover [3], so they are of less interest. Regarding the inputs, we just use integer objects. These are enough to make it possible to obtain full coverage [3]. Some containers (e.g., Hashtable) make use of *keys* when objects are inserted/removed. For simplicity, we just use the same object for both keys and inserted objects. Methods that take as input a key and a value to insert can be just considered as methods with only one input.

A test sequence of length l is hence composed of $2l$ variables. There are l variables indicating the method to use in the sequence. Because we are considering only two possible methods (i.e., insertion and removal), these l variables are binary (i.e., 0 representing an insertion and 1 for

while global optimum not found
    Choose I uniformly at random from $S(l)$.

Figure 2: Pseudo-code of RS

removal). Then, for each method there is a integer variable representing its input.

Unless symbolic execution is used (e.g., like for example in [10]), often constraints are used to limit the range of the input variables with the aim of making the testing easier. For example, integer inputs can be used only in a range $[0,100]$. Because an analysis of this issue is not in the scope of this paper, we just limit the integer inputs in the range $[0,l]$, which is an appropriate for testing containers [3]. For simplicity, we do not consider the case of the insertion of **null** objects.

Given this representation, for length l there are $S(l) = 2^l l^l$ different solutions (there are 2^l possible sequences of the two different methods, and each method can have l different values for input). The search space S is exponential in the length l.

4 Search Algorithms

In this paper we analyse four different search algorithms: RS, HC, (1+1)EA and a GA. We both consider the case of fixed length and variable length representation. For variable length representations, we consider two different cases, i.e. whether the minimisation of the length is sought or not. The fitness function is the commonly used (apart from [9]) approach level plus branch distance (more details can be found in [8]). With L we represent the maximum allowed length for the test sequences.

There exist several search algorithms with different names. In general, a name does not represent a particular algorithm. It rather represents a family of algorithms that share similar properties. In this paper, we analyse four different implementations of four different families of search algorithms. Obviously, the results on a particular implementation cannot be directly extended to its family.

A search algorithm might not find a global optimum in a reasonable amount of time. Therefore, it is common to put premature stopping criteria that are based on the available computational resources. A typical simple example is to put an upper bound to the maximum number of fitness evaluations that are allowed. In this paper, because in the empirical experiments all the search algorithms find global optima in reasonable time, for simplicity we do not put any premature stopping criterion, unless otherwise stated.

4.1 Random Search

RS is the simplest search algorithm. Random solutions are sampled until a global optimum is found. The information given by the fitness function is only used to check whether a global optimum has been sampled. RS is commonly used in literature as a base-line for comparing other search algorithms. What distinguishes among RS algorithms is the probability distribution used for sampling the new solutions. We employ a uniform distribution. Its pseudo-code is shown in Figure 2. For the case RS of variable length, the length of each solutions is randomly chosen in $\{1,\ldots,L\}$.

while global optimum not found
  Choose I uniformly at random from $S(l)$.
  while I not a local optimum in $N(I)$
    Choose I' from $N(I)$ according to strategy δ.
    if $f(I') < f(I)$, **then**
      $I := I'$.

Figure 3: Pseudo-code of HC.

4.2 Hill Climbing

HC starts from a search point, and then it looks at *neighbour* solutions. A neighbour solution is structurally *close*, but the notion of distance among solutions is problem dependent. If at least one neighbour solution has better fitness value, HC "moves" to it and it recursively looks at the new neighbourhood. If no better neighbour is found, HC re-starts from a new solution. HC algorithms differ on how the starting points are chosen, on how the neighbourhood is defined and on how the next solution is chosen among better ones in the neighbourhood.

We choose the starting points at random. The neighbourhood N is defined by swapping one single method at one time (i.e., an insertion is replaced by a removal, and vice-versa) without altering the inputs, and by modifying each input by $\pm 1\% l$. Hence the neighbourhood has size $l + 2l$. The strategy δ to visit the neighbourhood starts from left to right in the representation (i.e., the method in the first function call is modified, then the second visited neighbour consists of adding 1 to the input of the first method, etc.). As soon as a better solution is found, HC moves to it. In visiting the new neighbourhoods, instead of starting from the leftmost variable, HC starts from the successive of the last considered (whose modification has led to a new better solution), and it continues toward the rightmost, for then coming back as in a ring to the leftmost and finishing moving right until all neighbours have been visited. The pseudo-code of HC is shown in Figure 3.

In the case of variable length, we extend the definition of neighbourhood. In N_{vl} we also consider as neighbour the l solutions in which one of the l function calls is removed. For each position in the sequence, we also consider the insertion of a new random function call. The starting length of the test sequence is chosen at random in $\{1, \ldots, L\}$. When we want to minimise the sequence length, solutions in N_{vl} that decrease the length without worsening the fitness are always accepted.

4.3 (1+1) Evolutionary Algorithm

(1+1)EA is a single individual evolutionary algorithm. A single offspring is generated at each generation by *mutating* the parent. The offspring never replace their parents if they have worse fitness value. In a binary representation, a mutation consist in flipping bits with a particular probability. Typically, each bit is considered for mutation with probability $1/k$, with k the length of the bit-string. In our case, we have that a test sequence is composed of l function calls. We consider the methods in the test sequences as bits (e.g., 0 to represent an insertion and 1 for a removal), and the method inputs as groups of bits that are mutated together with a special operator. Therefore, in our case we have $k = 2l$. Each method type and input is mutated with probability $1/(2l)$. In case of a mutated input, a new different valid value is chosen with uniform probability. The pseudo-code is shown in Figure 4.

In the case of variable length, like for HC the starting length is chosen at random. In the same way, we consider the removal and insertion of new random function calls. In addition to the previously described mutation, with probability p (e.g., $p = 0.5$) we add/remove a random function

Choose I uniformly at random from $S(l)$
while global optimum not found
  $I' := I$.
  Mutate I'.
  If $f(I') \leq f(I)$
    then $I := I'$.

Figure 4: Pseudo-code of (1+1)EA.

Choose population K uniformly at random from $S(l)$.
while global optimum not found
  Copy best n solutions from K to K'.
  while K' is not completely filled
    Select 2 parents from K according to selection criterion,
    Generate 2 offspring that are copies of their parents,
    Apply crossover on offspring with probability P_{xover},
    Mutate each offspring,
    Copy the 2 new offspring into K'.
  $K = K'$.

Figure 5: Pseudo-code of GA.

call. If that mutation happens, we do again the same type of mutation, but with probability p^2. Again, if this latter happens, we do a third mutation with probability p^3, and so on with probability p^k, where $k - 1$ is the number of mutations done so far on an individual when an offspring is sampled. Similarly to HC, when we want to minimise the sequence length, shorter offspring but with no worse fitness are always accepted.

4.4 Genetic Algorithms

GAs are the most famous meta-heuristic used in the literature of search based software testing, and they are inspired by the Darwinian Evolution theory (as we explained in Section 2). To avoid the possible loss of good solutions, a number of best solutions can be copied directly to the new generation (*elitism*) without any modification. We used a rank-based selection [12] with bias 1.5. The employed crossover operator is a single point crossover. The mutation is done in the same way as for (1+1)EA. Figure 5 shows the pseudo-code of the employed GA. In our experiments, we use a population of 100 individuals. The crossover rate P_{xover} is set to 0.75. Elitism rate is set to 1 individual for generation.

In the case of variable length, the length of the test sequence is chosen at random in $\{1, \ldots, L\}$. Mutation is done in the same way as in (1+1)EA with variable length. For the single point crossover, the splitting point is chosen in the shortest of the two parents. When we want to minimise the sequence length, the individual in the population are ranked also based on their length. Between two individuals with same fitness, the shortest is preferred.

5 Case Study

For the case study, we considered six Java containers that are taken from the Java API and from the case study in [10]. Table 2 briefly summarises the considered methods in these classes. The goal

Table 2: Employed case study.

Container	Methods
TreeMap	put, remove
Hahstable	add, remove
Vector	add, remove
LinkedList	put, remove
BinTree	add, remove
BinomialHeap	insert, delete

is to cover all the possible branches that can be executed in the class (i.e., the considered insertion and removal methods plus all the methods that can be called through them).

5.1 TreeMap

We start our empirical analysis from the most difficult container in our previous work [3, 1], which is TreeMap. We have run each considered search algorithm with different values of the length l, ranging from 7 to 100 on its most difficult branch (an exhaustive search shows that for $l < 7$ there is no solution covering that branch). For each algorithm and each different length, we run $1{,}000$ experiments with different random seeds (i.e., in total $52{,}000$ experiments). Figure 6 shows the average number of fitness evaluations before a global optimum is found. Table 3 compares the performance of the search algorithms for length 7 and length 100. Mann-Whitney U tests confirm that for $l = 7$ (1+1)EA has the best median value, whereas for $l = 100$ it is RS and GA that have the best (no statistically significant difference between these two search algorithms).

The results in Figure 6 came as a surprise for us. For small lengths, the testing problem is difficult. A RS takes on average 106 thousand steps, whereas a GA is only four times faster than it. The best algorithm however is not GA, it is (1+1)EA. But it still requires on average more than 3 thousand steps before producing an optimal solution. What really surprised us is the behaviour of these algorithms when longer lengths are used. In these cases, a RS can find an optimal solution in less than ten fitness evaluations. The performance of the other algorithms drastically improves as well. GA has the same performance of RS. With a population size of 100 this happens because GA finds a solution even before finishing the evaluation of the first population. In this case, it is exactly equivalent to RS.

Because RS is a very simple algorithm to analyse, we can give the exact reason for its performance. Because the expected runtime of RS can be calculated by dividing the number of all solutions by the number of global optima (see for example [2]), that necessarily means that for longer sequences there are in proportion more optimal solutions. In other words, the ratio of global optima increases with the increment of the sequence length. Unexpectedly, this increase of the ratio is extremely rapid in this case study (Figure 6).

We run the same type of experiment when the length can change during the search. Because longer sequences are computationally more expensive to run, for fairer comparisons we do not consider the number of fitness evaluations. Instead, we consider the average total amount of method calls done during the search. It is equivalent to the number of fitness evaluations time the average sequence length during the search. Figure 7 shows the obtained results out of $52{,}000$ new experiments. Because no minimisation of the length is rewarded in fitness function, we show the average length of the final solutions in Figure 8. A new set of $39{,}000$ experiments has been done for the case when the length is tried to be minimised (note that RS is not considered in this case). Figure 9 shows the average total amount of method calls whereas Figure 10 shows the final average sizes.

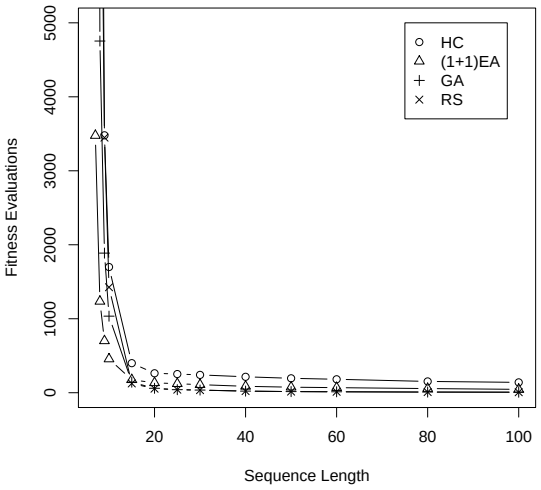

Figure 6: Average number of fitness evaluations before a global optimum is found. Different lengths are analysed. For each configuration, $1{,}000$ runs were used to collect these data.

In both Figures 7 and 9 we can see the same type of behaviour of Figure 6. The first notable difference is in Figure 7. The performance of HC increases with the increase of the length, but after a certain point it decreases. This *could* be related to the fact that the neighbourhood of ± 1 becomes less effective with longer sequences (a more detailed analysis of this behaviour is not in the scope of this paper). A second notable difference is in Figure 9, in which GA seems to have a better performance than $(1+1)EA$ even for small lengths. This *could* happen because in a population of 100 elements it is likely to have sequences whose length is close to the limit L (i.e., they are long). On the other end, if (1+1)EA starts from a very small length (e.g., $l = 1$), it can struggle to increase its length (at least up to 7 to find a global optimum) if that area of the search space is a plateau (i.e., if a mutation that increase the length does not improve the fitness value).

Regarding the final size, Figures 8 and 10 show that GA does not particularly benefit from rewarding shorter test sequences. This *could* be explained by the fact that it is very quick to find a solution. On the other hand, HC and $(1+1)EA$ seem to produce shorter sequences when the length is tried to be minimised. Note that as soon as a best sequence is found, these algorithms stop. Reductions on the length are done before a best solution is found.

5.2 Simple Strategy

Based on the unexpected results shown in Figure 6, we analysed the ratio of local optima in the testing of all the containers in our case study. Because the performance of RS is directly based on this ratio, we only analysed the most difficult (i.e., less global optima) branches in each container.

For each container and for each analysed length, we sampled 100 million random test sequences (for a total of 1.9 billion evaluations for each container). We then show the ratio of the branches that are most difficult (but that are executed at least once). In case a branch in never executed for a particular length, we considered it unfeasible. Although 1.9 billion test sequences for each container and a manual inspection of the code does not constitute a proof, they give strong confidence in claiming that the unreached branches are unfeasible. Results are shown in Figure 11. In many cases, the ratio is higher than 0.01. In these cases, a RS on average would find an

Table 3: Comparisons of performance of search algorithms for length 7 and 100. Data were collected from 1,000 runs for each configuration.

Length	Algorithm	min	mean	median	max	var
7	HC	14	51461.4	37587	314456	$2.3 \cdot 10^9$
100	HC	1	140.0	68	1334	$3.2 \cdot 10^4$
7	(1+1)EA	51	3476.7	2501	23189	$1.0 \cdot 10^7$
100	(1+1)EA	1	45.4	27	325	$2.7 \cdot 10^4$
7	GA	46	25749.3	18668	159997	$6.2 \cdot 10^8$
100	GA	1	7.0	5	44	44.2
7	RS	13	106405.2	71789.5	892004	$1.1 \cdot 10^{10}$
100	RS	1	6.822	5	65	41.7

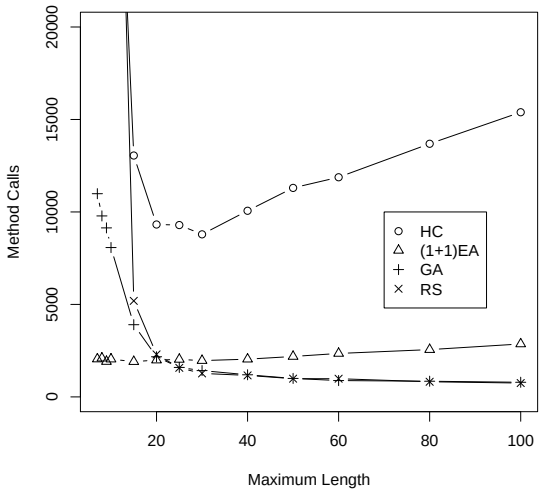

Figure 7: Average number of method calls before a global optimum is found. Different maximum lengths L are analysed. For each configuration, 1,000 runs were used to collect these data.

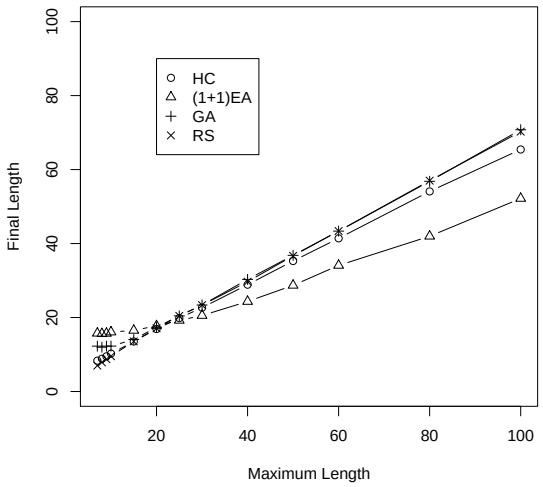

Figure 8: Average length of the final solutions. Different maximum lengths L are analysed. For each configuration, $1{,}000$ runs were used to collect these data.

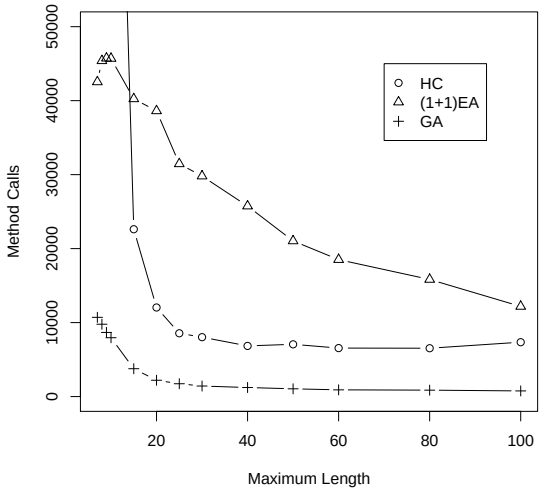

Figure 9: Average number of method calls before a global optimum is found when the length is tried to be minimised. Different maximum lengths L are analysed. For each configuration, $1{,}000$ runs were used to collect these data.

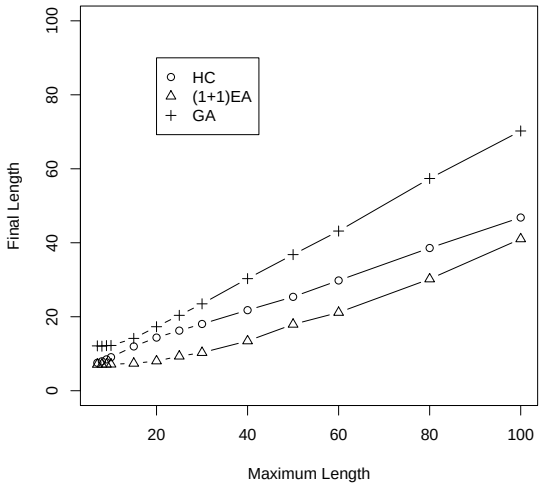

Figure 10: Average length of the final solutions when the length is tried to be minimised. Different maximum lengths L are analysed. For each configuration, $1{,}000$ runs were used to collect these data.

optimal solution in less than 100 iterations (see [2]).

For small values of the length, many branches are unfeasible. Therefore, it is reasonable to see an high variance of the ratio for these small values. However, for higher values of the length, the most difficult branch to cover does not remain the same (this information cannot be derived from Figure 11). Therefore, for each branch that resulted as most difficult for at least one length value, we show its ratio in Figure 12.

Based on these results, we can concluded that using longer sequences is a better strategy when coverage criteria are sought (at least for container classes). For example, for length 25, in all containers the most difficult branch has a ratio that is higher than 0.01. Once an optimal solution is found, a simple heuristic can be used to shorten it. For example, at each step of the post-processing a function call can be removed. If the coverage decreases, then that function call is reintroduced in the sequence. Otherwise, a next function call is removed, and so on until all the current function calls have been tried to be removed.

In Figure 12 we can see only three cases in which the ratio decreases with the increase of the length. However, in these cases their ratio is still so high to make trivial their covering with even a naive RS.

6 Formal Theoretical Analysis

6.1 General Rules

We have carried out empirical analysis only on container classes. Although container classes are the de facto benchmark in the literature of OO software (see Table 1), they are not a complete representation of OO software. Therefore, for real-world software it could happen that longer test sequences do not give better results. To give more general results that can be applied to any software, in this section we formally analyse the conditions for which longer test sequences are more successful.

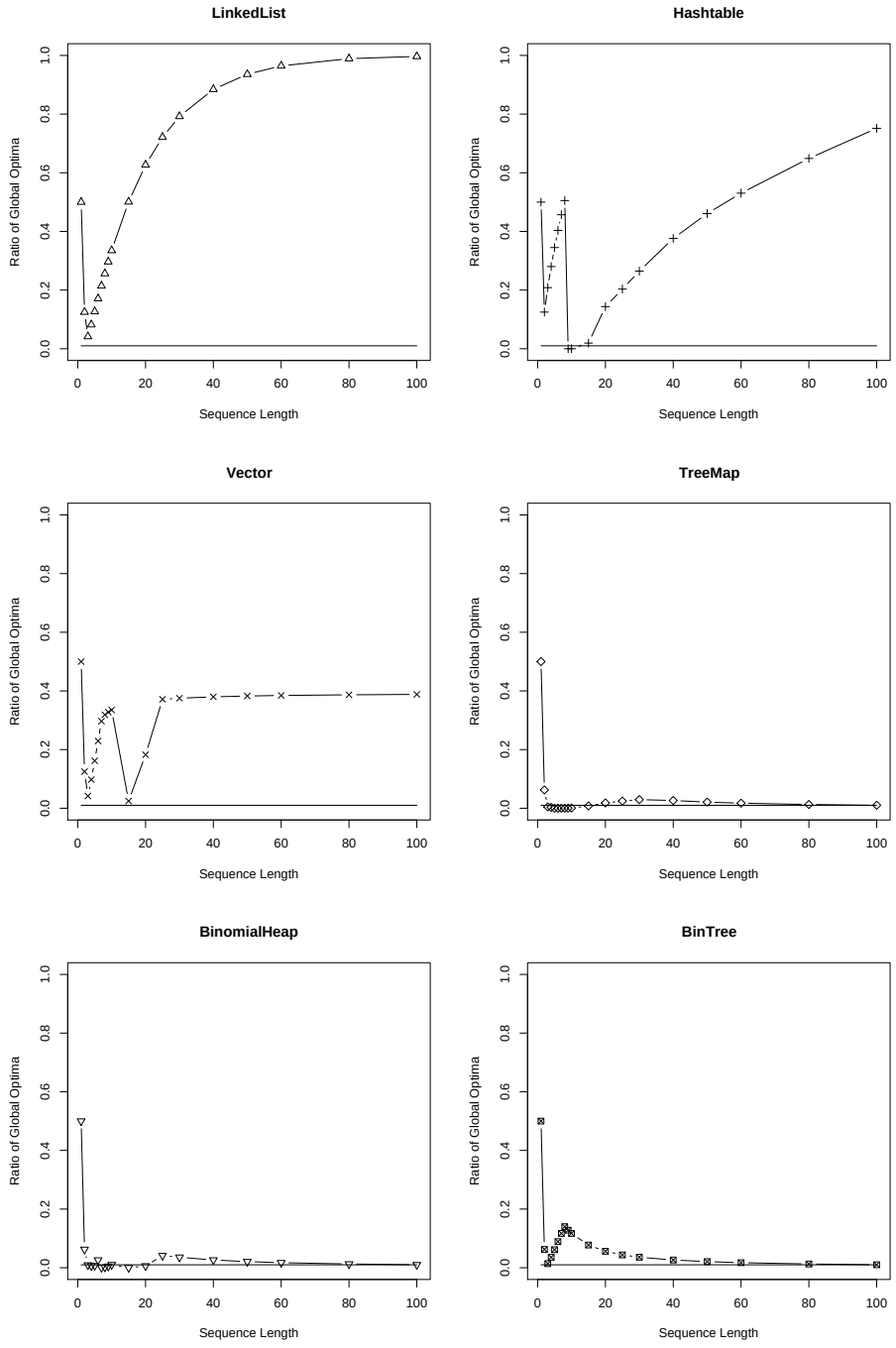

Figure 11: Ratio of global optima over the number of all possible solutions. For each length, the ratio of branch with lowest non-zero number of found optima is shown. A horizontal line shows the threshold 0.01. Data were collected from 100 million random sequences for each length and container.

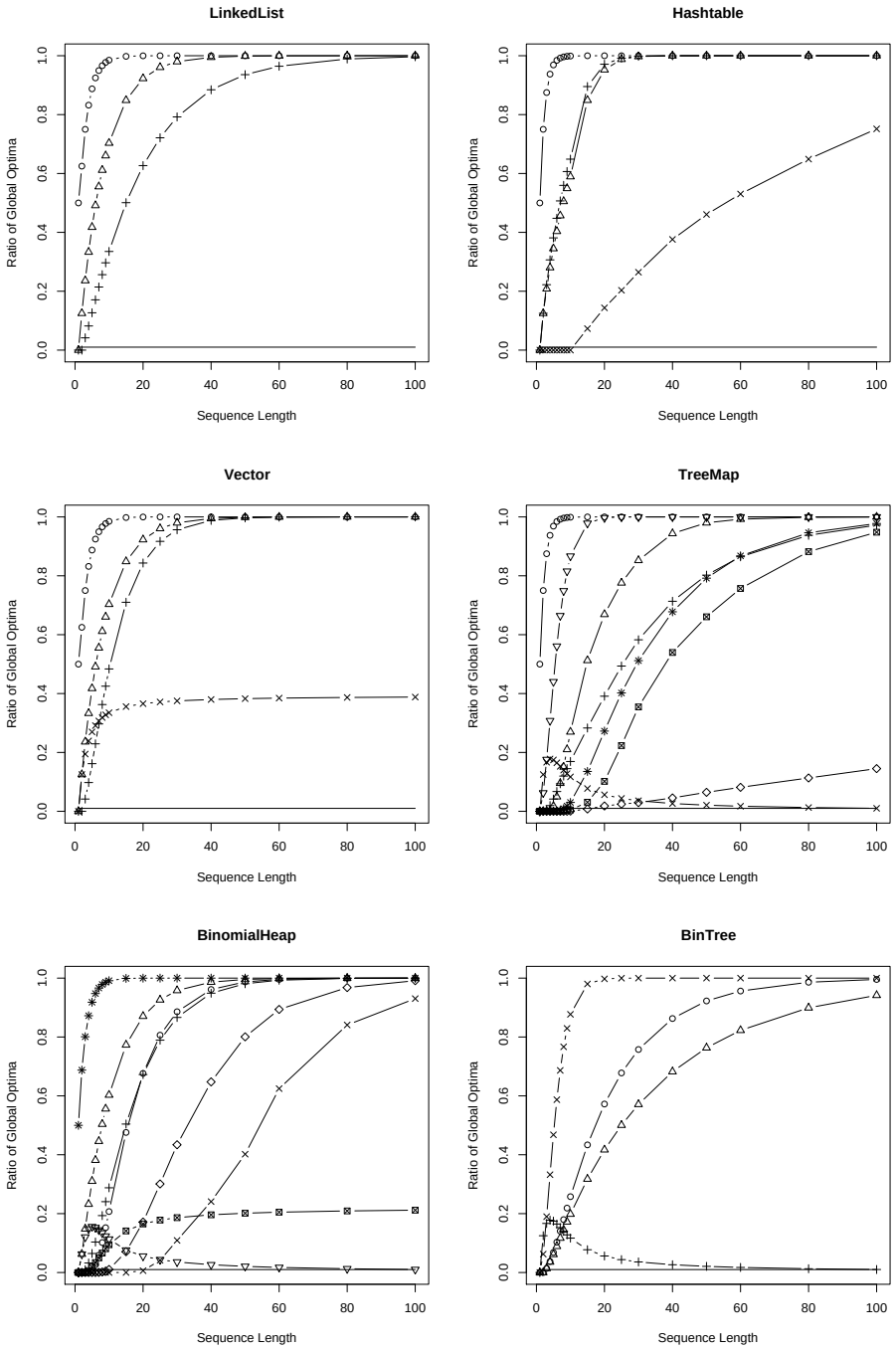

Figure 12: Ratio of global optima over the number of all possible solutions for the most difficult branches. A horizontal line shows the threshold 0.01. Data were collected from 100 million random sequences for each length and container.

To simplify the analysis, we do not consider the initialisation of the object instances used in the test sequences. Let M be the set of different methods. Each method can take as input an element from a set I of infinite cardinality. For simplicity but without loss of generalisation, let consider that I is equal for each method, and each method takes as input only one element. The length of the sequence is l. Constraint for a given length on the variables are represented by choosing a subset $C(l)$ of inputs, i.e. $C(l) \subset I$. Let t be a test sequence for the search space $S(l)$, then $t(k)$ with $k \leq l$ is a test sequence that consider only the first k function calls.

Let $R(l) \subset M \times C(l)$ be the subset of function calls that does not influence the covering of the target branch. For example, in the case of insertion/removal of elements in a container, the call to read-only methods like **size()** does not have any effect.

Let $G(l)$ the set of global optima for the length l. The number of possible sequences is $|S(l)| = |M|^l \cdot |C(l)|^l$. Let $r(l)$ be the ratio of global optima over the number of possible solutions, i.e. $r(l) = |G(l)|/|S(l)|$. We can hence prove the following theorems:

Theorem 1. *If $R(l) \neq \emptyset$, $|G(l)| > 0$ and $|C(k)| = c$ (where c is a positive constant $c > 1$) for all $k \geq l$, then $r(k+1) > r(k)$ for all $k \geq l$.*

Proof. For all the global optima in $G(l)$, adding a function call at the end of the sequence does not change the fact that they are global optima. Therefore, $|G(l+1)| \geq |G(l)| \cdot |M| \cdot c$.

Let choose z among the global optima $G(l)$ such that:

- $z(j-1)$ is not a global optimum, whereas $z(j)$ is a an optimal solution.
- All function calls after position j belong to the set $R(l)$.

At least one element z of this type exists, because given any global optimum we can just replace all the function calls (one at the time) with a new one belonging to $R(l)$ until the new considered sequence is still optimal.

A new sequence z' generated from z by replacing in position j a new function call of type $R(l)$ is not a global optimum. And it will be necessarily different from the $|G(l)| \cdot |M| \cdot c$ optima we described before. At this sequence z', if we append the function call we removed from position j, then this new sequence is necessarily a global optimum. The number of global optima is hence at least $|G(l+1)| \geq |G(l)| \cdot |M| \cdot c + |R(l)|$. Therefore:

$$\begin{aligned} r(l+1) &= \tfrac{|G(l+1)|}{|S(l+1)|} \\ &\geq \tfrac{|G(l)| \cdot |M| \cdot c + |R(l)|}{|M|^{l+1} \cdot c^{l+1}} \\ &= \tfrac{|G(l)|}{|M|^l \cdot c^l} + \tfrac{|R(l)|}{|M|^{l+1} \cdot c^{l+1}} \\ &= \tfrac{|G(l)|}{|S(l)|} + \tfrac{|R(l)|}{|S(l+1)|} \\ &= r(l) + \tfrac{|R(l)|}{|S(l+1)|} \\ &> r(l) \end{aligned}$$

Once proved $r(l+1) > r(l)$, to conclude the proof we can simply use induction to prove $r(k+1) > r(k)$, this because $g(k) \geq g(l) > 0$.

□

Theorem 2. *If $R(l) \neq \emptyset$, $|G(l)| > 0$ and $|C(k+1)| > |C(k)|$ for all $k \geq l$, then it is not necessarily true that $r(k+1) > r(k)$ with $k \geq l$.*

Proof. We just need to find one case for which $r(k+1) \leq r(k)$. Let $|C(l+1)| = c + t$ where $c = |C(l)|$ and $t > 0$. In the infinite space of possible programs, let consider one for which the use

of one of any of these t new inputs makes impossible to cover the target branch. The non-optimal solutions $W(l+1)$ for length $l+1$ are hence at least:

$$|W(l+1)| \geq |S(l+1)| - (|M|^{l+1} \cdot c^{l+1}) \,. \tag{1}$$

This is a very low underestimation of their number, but it is more than enough to prove this theorem.

To have $r(l+1) > r(l)$, we need enough global optima such that $|G(l+1)| > |S(l+1)| \cdot (|G(l)|/|S(l)|)$. Because $|S(l+1)| = |G(l+1)| + |W(l+1)|$, then we would not have enough global optima if $|W(l+1)| > |S(l+1)| - (|S(l+1)| \cdot (|G(l)|/|S(l)|))$. Therefore, by Disequation 1, we just need to prove:

$$|S(l+1)| - (|M|^{l+1} \cdot c^{l+1}) > |S(l+1)| - \left(|S(l+1)| \cdot \frac{|G(l)|}{|S(l)|}\right) ,$$

which can be reduced to:

$$1 - \frac{1}{(1+\frac{t}{c})^{(l+1)}} > \left(1 - \frac{|G(l)|}{|S(l)|}\right) .$$

The previous disequation is clearly true for $t \to \infty$, but could be false for small values of t like for example $t = 1$. Because the conditions of the theorem just state $|C(k+1)| > |C(k)|$, we hence need to analyse the smallest case $t = 1$. For $t > 1$ we can just follow the same type of reasoning. Instead of analysing the case $r(l+1) < r(l)$, let study the case $r(l+z) < r(l)$. Because $t = 1$ (i.e., $|C(l+1)| = |C(l)| + 1$), then for the same type of discussion done for Disequation 1, we have:

$$|W(l+z)| \geq |S(l+z)| - (|M|^{l+z} \cdot c^{l+z}) \,,$$

this because $|C(l+z)| = |C(l)| + z$ (it easily follows from $|C(l+1)| = |C(l)| + 1$). We would not have enough global optima to guarantee $r(l+z) > r(l)$ if:

$$|S(l+z)| - (|M|^{l+z} \cdot c^{l+z}) > |S(l+z)| - \left(|S(l+z)| \cdot \frac{|G(l)|}{|S(l)|}\right) ,$$

which can be reduced to:

$$1 - \frac{1}{(1+\frac{z}{c})^{(l+z)}} > \left(1 - \frac{|G(l)|}{|S(l)|}\right) .$$

Again, this disequation is clearly true for $z \to \infty$. For proving this theorem we do not need to calculate the smallest value z for which this disequation is true. Therefore, because it does exist at least one value z for which $r(l+z) < r(l)$, there would be necessarily one intermediate value $l \leq k < l+z$ for which $r(k+1) < r(k)$.

□

6.2 Discussion

These two theorems have some interesting consequences. Theorem 1 gives the conditions for which longer test sequences are necessarily better. These conditions are very general, and can be applied to practically most types of software. However, the price of general rules is that they are weaker. In fact, although Theorem 1 states that longer sequences are better, without considering the actual software (or some restricted sets) we can say only little about how much the improvement is. At the current moment, we are not even able to state whether under the same conditions of Theorem 1 we can have $(l \to \infty) \Rightarrow (r(l) = 1)$ (this will be studied in future work).

Intuitively, for longer sequences it is easy to expect more optimal solutions. But at the same time the space of possible solutions increases as well. Theorem 1 formally proves the non-obvious fact that their ratio increases as well.

Theorem 2 gives the conditions for which it is not necessarily true that longer sequences are better. Note that it does not mean that they would be worse for sure. It is important to note that this can happen when we increase the space of available inputs for longer sequences, i.e. $|C(l+1)| > |C(l)|$. This is actually the type of constraints we used in this paper, i.e. for length l we used l different integers as input. In our case study, we obtained that longer sequences are better but in three cases, and this is not in contrast with Theorem 2.

The choice of the input constraints is important for many automated testing techniques (e.g., [9, 4, 3]), but not for the ones that use symbolic execution (e.g., [10, 6]). A formal analysis of the length of test sequences for symbolic execution based techniques will be matter of future work.

7 Conclusion

In this paper we have analysed the role that the length of test sequences plays in software testing. We have empirically shown on common software testing benchmarks that having slightly longer sequences can drastically improve the results. Problems that are considered difficult in literature become trivial to solve. When you can solve a "problem" in a fraction of a second (depending on the efficiency of the tool implementation and the used machine), then this should not be considered as a problem any more.

The role of the test sequence length has received only little attention in literature. This paper gives the important contribution to rigorously support its importance. The findings of this paper can be easily exploited in most the techniques described in literature.

Because any empirical study has the limitation that the results are not necessarily easy to generalise, we also carried out a formal theoretical analysis. We gave the general conditions for which longer sequences are necessarily better and the conditions for which they could be worse. These general conditions could be applied to most types of software.

Future work will follow two directions. First, realistic software for which longer sequences give worse results need to be identified and studied. Hybrid techniques that exploit variable length representation should be hence designed to work well in average. For example, a GA that uses a population of individuals with different lengths could be an appropriate first choice to analyse. Second, additional formal analyses on the role of the length are required to get a better understanding of which are the limitations and difficulties of software testing [7].

8 Acknowledgments

The author is grateful to Per Kristian Lehre for insightful discussions. This work is supported by an EPSRC grant (EP/D052785/1).

References

[1] A. Arcuri. Insight knowledge in search based software testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2009. (to appear).

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

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

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *IEEE International Conference on Software Engineering (ICSE)*, pages 71–80, 2008.

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

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

[7] P. K. Lehre and X. Yao. Runtime analysis of search heuristics on software engineering problems. *Frontiers of Computer Science in China*, 3(1):64–72, 2009.

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

[9] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[10] W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.

[11] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1925–1932, 2006.

[12] 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.

[13] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 196–205, 2004.

[14] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.