Search Based Software Testing of Object-Oriented Containers

Andrea Arcuri*and Xin Yao
The Centre of Excellence for Research,
in Computational Intelligence and Applications (CERCIA),
The School of Computer Science,
The University of Birmingham,
Edgbaston, Birmingham B15 2TT, UK.
{A.Arcuri,X.Yao}@cs.bham.ac.uk

Abstract

Automatic software testing tools are still far from ideal for real world Object-Oriented (OO) Software. The use of nature inspired search algorithms for this problem has been investigated recently. Testing complex data structures (e.g., *containers*) is very challenging since testing software with simple states is already hard. Because containers are used in almost every type of software, their reliability is of utmost importance. Hence, this paper focuses on the difficulties of testing container classes with nature inspired search algorithms. We will first describe how input data can be automatically generated for testing Java Containers. Input space reductions and a novel testability transformation are presented to aid the search algorithms. Different search algorithms are then considered and studied in order to understand when and why a search algorithm is effective for a testing problem. In our experiments, these nature inspired search algorithms seem to give better results than the traditional techniques described in literature. Besides, the problem of minimising the length of the test sequences is also addressed. Finally, some open research questions are given.

Keywords: Software Testing, Object-Oriented Software, Containers, Search Algorithms, Nature Inspired Algorithms, Search Based Software Engineering, Testability Transformations, White Box Testing.

1 Introduction

Software Testing is used to find the presence of bugs in computer programs [42]. Even though no bugs are found, testing cannot guarantee that the software is bug-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software developing [7]. This cost is paid because Software Testing is very important. Releasing bug-ridden and non-functional software is in fact an easy way to lose customers. Besides, bugs should be discovered as soon as possible, because fixing them when the software is in a late stage of development is much more expensive than at the beginning. For example, in the U.S.A it is estimated that every year around $20 billion could be saved if better testing was done before releasing new software [47].

One way to try to deal with this issue is to use *Unit Tests* [24, 18]. It consists of writing pieces of code that test as many methods of the project as possible. For example, a method that sums two integers can be called with two particular values (1 and 2 for example), and then the result will be checked against the expected value (3). If they do not match, we can be sure that there is something wrong with the code. Because testing all possible inputs of a method is infeasible, a subset of tests needs to be chosen. How to choose this test subset depends on the type of testing, and it is the problem that this paper addresses.

Writing unit tests requires time and resources, and usually it is a tedious job. Thus, a way to automatically generate unit tests is needed. However, if a formal specification of the system is not given, testing the results against an *Oracle* cannot be automated. I.e., choosing the best inputs can be automated, but checking the result against the expected one (e.g., the value 3 in the

---

*Andrea Arcuri is the corresponding author. Email A.Arcuri@cs.bham.ac.uk. Telephone: +44 (0) 7962615010.

previous example) cannot be automated if the system does not know the semantics of the function. Different approaches have been studied to automatically generate unit tests [33], but a system that can generate an optimal set of unit tests for any generic program has not been developed yet. Lot of research has been done on procedural software, but comparatively little on Object-Oriented (OO) software. The OO languages are based on the interactions of *objects*, which are composed of an internal state and a set of operations (called *methods*) for modifying those states. *Objects* are instances of a *class*. Two objects belonging to a same class share the same set of methods and type of internal state. What might make those two objects different is the actual values of their internal states.

In this paper we focus on a particular type of OO software that is *Containers*. They are data structures like arrays, lists, vectors, trees, etc. They are classes designed to store any arbitrary type of data. Usually, what distinguishes a container class from the others is the computational cost of operations like insertion, deletion and finding of a data object. Containers are used in almost every type of OO software, so their reliability is important.

We present a framework for automatically generating unit tests for container classes. The test type is *White Box Testing* [42]. We analyse different search algorithms, and then we compare them with more traditional techniques. We use a search space reduction that exploits the characteristics of the containers. Without this reduction, the use of search algorithms would have required too much computational time. This paper also presents a novel testability transformation [21] to aid the search algorithms. Although the programming language used is Java, the techniques described in this paper can be applied to other OO languages. Moreover, although we frame our system specifically for containers, some of the techniques described in this paper might also be extended for other types of OO software.

This work gives two important contributions to the current state of the practise of the Search Based Software Engineering (SBSE) [13]:

1. we compare and describe how to apply five different search algorithms. They are well known algorithms that have been employed to solve a wide range of different problems. This type of comparisons are not common in the literature on SBSE, but they are very important [58]. In fact, unexpected results might be found, as the ones we report in section 4.

2. although reducing the size of the test suites is very important [10], the problem of minimising the length of the test sequences has been almost ignored in literature. We addressed it and studied its implications on a set of five search algorithms.

The paper is organised as follows. In section 2 a particular type of software (containers) with its properties is presented. Section 3 describes how to apply five different search algorithms to automatically generate unit tests for containers. Experiments carried out on the proposed algorithms follow in section 4. A short review of the related literature is given in section 5. The conclusions of this work can be found in section 6.

2   Testing of Java Containers

In OO programs, containers hold an important role because they are widely use in almost any type of software. Not only do we need to test novel types of containers and their new optimisations, but also the current libraries need to be tested [32].

There are different types of containers, like arrays, vectors, lists, trees, etc. We usually expect from a container methods like `insert`, `remove` and `find`. Although the interfaces of these methods can be the same, how they are implemented and their computational cost can be very different. Besides, the behaviour of such methods depends on the elements already stored inside the container. The presence of an internal state is a problem for software testing [36, 38, 34], and this is particularly true for the containers.

2.1   Properties of the problem

A solution to the problem is a sequence S_i of Function Calls (FCs) on an instance of the Container under Test (CuT).
A Function Call (FC) can be seen as a triple:

$$< object\_reference, function\_name, input\_list >$$

It is straightforward to map a FC in a command statement. E.g.:

```
ref.function(input[0],...,input[n]);
```

I.e., given an *object_reference* called `ref`, the function with name *function_name* is called on it with the input in *input_list*. There will be only one S_i for the CuT, and not one for each of its branches (as usually happens in literature). Each FC is embedded in a different `try/catch` block. Hence, the paths that throw exceptions do not forbid the execution of the following FCs in the sequence.

Given a coverage criterion (e.g., "path coverage" [28]), we are looking for the shortest sequence that gives the best coverage. For simplicity, we will consider only "branch coverage". However, the discussion can be easily extended to other coverage criteria.

Let S be the set of all possible solutions to the given problem. The function $cov(S_i) : S \to N$ will give the coverage value of the sequence S_i. That function will be upper bounded by the constant NB, that represents the number of branches in the CuT. It is important to remember that the best coverage (global optimum) can be lower than NB, due to the fact that some paths in the execution can be infeasible. The length of sequence is given by $len(S_i) : S \to N$. We prefer a solution S_i to S_j iff the next predicate is true:

$$\begin{cases} cov(S_i) > cov(S_j) & \text{or} \\ cov(S_i) = cov(S_j) \ \wedge \ len(S_i) < len(S_j) \ . \end{cases} \tag{1}$$

The problem can be viewed as a Multi-Objective task [16]. A sequence with a higher coverage will always be preferred regardless of the length. Only if the coverage is the same, we will look at the length of the sequences. Thus, the coverage can be seen as a hard constraint, and the length as a soft constraint. Although there are two different objectives, coverage and length, it is not a good idea to use Pareto Dominance [16] in the search. This is because the length is always less important than the coverage. Thus, the two objectives should be properly combined together in a single fitness function, as for example:

$$f(S_i) = cov(S_i) + \frac{1}{1 + len(S_i)} \ . \tag{2}$$

Containers have some specific properties. Some of them depend on the the actual implementation of the container, thus they cannot be analytically solved without considering the source code. However, our empirical tests show that these properties are true for any examined containers. Let k be the position in S_i of the last FC that improves the coverage of that sequence. The properties are:

- any operations (like insertion, removal and modification of one FC) done on S_i after the position k cannot decrease the coverage of the sequence. This is always true for any software.

- given a random insertion of a FC in S_i, the likelihood that the coverage decreases is low. It will be always zero if the insertion is after k, or if the FC does not change the state of the container.

- following from the previous property, given a random removal of a FC in S_i, the likelihood of the coverage increases is low.

- The behaviour of a FC depends only on the state of the tested container when the FC is executed. Therefore, a FC cannot alter the behaviour of the already executed FCs.

- let S_r be a sequence, and S_t be generated by S_r adding any FC to the tail of S_r . Due to the previous property, we have $cov(S_t) \geq cov(S_r)$. For the same reason, we have $cov(S_r, i) \geq cov(S_r, j)$, where $j \leq i \leq len(S_r)$ and $cov(S_r, i)$ gives the coverage of S_r when only the first i FCs are executed. Given two random sequences generated with the same distribution, therefore, it is easy to understand that we should expect a better coverage from the longest one.

- calculating the coverage for S_i requires the call of $len(S_i)$ methods of the CuT. This can be very expensive from a computational point of view. Thus, the performance in terms of a search algorithm depends heavily on the length of the sequences that it evaluates.

2.2 Search Space Reduction

The solution space of the test sequences for a container is very huge. We have M different methods to test, so there are M^L possible sequences of length L. We do not know a priori which is the best length. Although we can put an upper bound to the max length that we want to consider, it is still an extremely large search space. Besides, each FC can take some parameters as input, and that increases the search space even further.

The parameters as input for the FCs make the testing very difficult. They can be references to instances of objects in the heap. Not only the set of all possible objects is infinite, but a particular instance may need to be put in a particular state by a sequence of FCs on it (they are different from the ones of the container). In the same way, these latter FCs can require objects as input which need sequences of their proper FCs on them.

Fortunately, for testing of Java Containers, we can limit the space of the possible inputs for the FCs. In fact, usually (at least for the containers in the Java API) the methods of a container need as input only the following types:

1. *indices* that refer to a position in the container. A typical method that uses them is `get(int i)`. The indices are always of `int` type. Due to the nature of the containers, we just need to consider values inside the range of the number of elements stored in the CuT. Besides, we also need some few indices outside this range. If we consider that a single FC can add no more than one element to the container (it is generally true), the search space for the indices can be bound by the length of S_i.

2. *elements* are what are stored in a container. Usually they are references to objects. In the source code, the branch statements that depend on the states of elements are only of three types: the *Natural Order* between the elements, the equality of an element to `null` and the belonging of it to a particular Java class. This latter type will be studied only in future work. The natural order between the elements is defined by the methods `equals` and `compareTo`. Given n elements in the CuT, we require that all the orders of these n elements are possible. We can easily do it by defining a set Z, with $|Z| = n$, such that all elements in Z are different between them according to the natural order (i.e., if we call `equals` to any two different elements in Z we should get always false as a result). A search algorithm can be limited to use only Z for the elements as input for the FCs without losing the possibility of reaching the global optimum. Anyway, we should also handle the value `null`.

   The number n of elements in the CuT due to S_i is upper bounded by $len(S_i)$ (we are not considering the possibility that a FC can add more than one element to the CuT). Because the natural order does not depend on the container, we can create Z with $|Z| \geq len(S_i) + 1$ regardless of the CuT. We can as example use `Integer` objects with value between 0 and $|Z|$. It is important to outline that Z is automatically generated: the user does not have to give any information to the system.

   Bounding the range of the variables is a technique that has been widely employed in software testing. However, in this work we give the assumptions for which this reduction does not compromise the results of the search algorithms.

3. *keys* are used in containers such as *Hash Tables*. The considerations about *elements* can be also applied to the *keys*. The difference is that the method `hashCode` can be called on them. Because how the hash code is used inside the source code of the CuT cannot be known a priori, we cannot define for the keys a finite set such Z that guarantees us that the global optimum can be reached. In our experiments, we used the same Z for both *elements* and *keys* with good results. However, for the *keys*, there are no guaranties that Z is big enough.

4. some FCs may need as input a reference to one other container. Such type of FCs are not considered at the moment, and they are excluded from the testing. Future work will be done to cover also them. However, in our test cluster, only 11 methods on a total of 105 require this type of input.

When a random input needs to be chosen, two constants N and P (e.g., -2 and 58) are used to bound the values. An index i is uniformly chosen in $N \leq i \leq P$, and an element/key e is chosen from Z with value in the the same range. However, with probability δ, the element e is used with a `null` value. If after a search operator any sequence S_i has $len(S_i) > P$, the value P is updated to $P_{t+1} = \lceil \alpha P_t \rceil$, where t is the time and $\alpha > 1$. Hence, P is always bigger or equal than the value of the length of any sequence in the search. In fact, any used search operator can only increase a sequence at most by one. There is only one exception (i.e., the *crossover* operator described in section 3.4 can increase a sequence by more than one FC), but that operator cannot make a sequence longer than P in any case (see section 3.4). When a sequence is initialised at random, its length is uniformly chosen in $[0, P]$.

All the search algorithms that will be analysed in this paper will use the space reductions described above, unless otherwise stated.

2.3 Branch Distance

If the only information that a search algorithm can have is about whether a branch is covered or not, the algorithm will have no guidance on how to cover it. In other words, the search space will have big plateaus. In such a case, we can say that the search is "blind". Anyway, if the predicate of that branch statement is not complex, the likelihood of cover both its branches by chance is not so low. It is for that reason that a random search can achieve a quite good coverage. When the predicate is complicated, a blind search is likely to fail. Let's say that we want to cover a *then* branch of an `if` statement (the *else* branch can be studied in the same way). One way to address this problem is to use the *Branch Distance* (BD) of the branch [27]. That is a real value that tells us how far the predicate is to be evaluated as true. If we include BD to the coverage value of the sequence, the search space will become smoother. There can be different ways to define BD. A widely used way can be found in [49].

The BD has an important role in Software Testing, and a lot of research has been done on how to improve its effectiveness [5]. Common issues are: *flags* in the source code [20, 4], especially if they are assigned in loops [3], dependences of the predicate to previous statements [19, 37] and nested branch statements [35]. Anyway, those works are concerning the coverage of only one particular branch at time. On the other hand, the test sequences for Java Containers analysed in this paper try to cover all the branches at the same time. According to [56], it should be expected a better performance if each predicate is target/tested separately. I.e., there will be a different test driver for each predicate in the CuT. However, that work does not consider the necessity of a sequence of function calls to put the program in a helpful state for testing. To our best knowledge, the use of a single sequence of FCs to cover all the branches at the same time with the aid of the branch distances has never been analysed before. Therefore, a comparison of the two different techniques is required.

In detail, we need to use:

$$b(j) = \begin{cases} 0 & \text{if the branch } j \text{ is covered ,} \\ k & \text{if the branch } j \text{ is not covered and its} \\ & \text{opposite branch is covered at least twice ,} \\ 1 & \text{otherwise ,} \end{cases} \tag{3}$$

$$0 < k < 1 , \tag{4}$$

$$B(S_i) = \sum_{j=1}^{NB} \frac{b(j)}{NB} . \tag{5}$$

Where k is the lowest normalised BD for the predicate j during the execution of S_i. The function $b(j)$ defined in (3) describes how close the sequence is to cover that not covered branch j. If its branch statement is never reached, it should be $b(j) = 1$ because we cannot compute the BD for its predicate. Besides, it should also be equal to 1 if the branch statement is reached only once. Otherwise, if j will be covered due to the use of $b(j)$ during the search, necessarily the opposite of j will not be covered any more (we need to reach the branch statement at least twice if we want to cover both of its branches).

We can integrate the normalised BDs of all the branches (5) with the coverage of the sequence S_i in the following way:

$$cb(S_i) = cov(S_i) + (1 - B(S_i)) . \tag{6}$$

It is important to notice that such a function (6) guarantees that

$$cb(S_i) \geq cb(S_j) \Rightarrow cov(S_i) \geq cov(S_j) . \tag{7}$$

Finally, to decide if S_i is better than S_j, we can replace (1) with:

$$\begin{cases} cb(S_i) > cb(S_j) & \text{or} \\ cb(S_i) = cb(S_j) \ \wedge \ len(S_i) < len(S_j) . \end{cases} \tag{8}$$

In such a way, the search landscape gets smoother. Although we can use (8) to aid the search, we still need to use (1) to decide which is the best S_i that should be given as the result. In fact, for the final result, we are only interested in the achieved coverage and length. How close the final sequence is to get a better coverage is not important.

2.4 Testability Transformations

When BDs are used, a common way to improve the performance of a search algorithm is the use of *Testability Transformations* [21]. They are source to source transformations applied to the code of the CuT. Their only requirement is that the test data generated for the transformed program are adequate for the original one. Usually, these transformations are used to handle the *flags* problem. Here we propose a novel testability transformation that is specific for the Java Containers.

Some branch statements in a Java Container source code rely on the natural order of its stored elements. The method `equals` gives no information on how much two elements are different, because it returns only a boolean value (this is like the flag problem). Therefore, we propose to transform all calls to the `equals` method:

```
obj.equals(other)
```

with:

```
(((Comparable)obj).compareTo(other) == 0)
```

We should notice that this transformation is useful only if the stored objects implement the interface `Comparable`. If we use the set Z defined in sec 2.2, we can guarantee it. Although the method `compareTo` says if two objects are different, it does not say how much they are different. Therefore, we need that the objects in Z extend this semantics by sub-classing this method. Because Z can be implemented regardless of the container that will be tested, besides the fact that `compareTo` gives an integer as its output, we can easily do this extension of the semantics. For example, `compareTo` can give as output the difference between the natural order values of the two compared objects. Hence, the search landscape gets smoother, and this fact helps the search algorithms.

It is arguable why we use `compareTo` instead of defining a new method. This choice was made because in such a way we need no testability transformations on the `compareTo` calls. I.e., we need to transform only the calls to `equals` and not the ones to `compareTo`. Otherwise, the `compareTo` method with its original semantics generates plateaus in the search space. However, we cannot know (or it is too difficult to be exactly determined) if all the calls to `equals` belong only to the stored elements. Thus, to be sure of not having any exception raised during the search, we can use this other transformation instead:

```
(((obj instanceof Comparable) ?
 ((Comparable)obj).compareTo(other) :
 (obj.equals(other) ? 0 : 1))
 == 0 )
```

However, this latter transformation introduces two new branches in the program for every call to `equals`. This is a problem, because input test data (in our case a sequence of FCs) for branch coverage for the transformed program is not adequate for the original one. Given a support library `ett` (the name is not important), one possible solution is:

```
(ett.Utils.compare(obj,other)==0)
```

with the function `compare` implemented in this way:

```java
public static int compare(Object obj,Object other)
{
  if(obj instanceof Comparable)
     return ((Comparable)obj).compareTo(other);
  else
     return obj.equals(other) ? 0 : 1;
}
```

It is important that the instrumentation tool (described in 2.5) should be informed to ignore this function (i.e., that function should be considered as a side effect free external function of which there is no access to its source code).

This testability transformation can be done automatically, without any help of the user.

2.5 Instrumentation of the Source Code

To guide a search algorithm, the CuT needs to be executed to get the branch distances of the predicates [27]. To analyse the execution flow, the source code of the CuT needs to be *instrumented*. This means that extra statements will be added inside the original program. To trace the branch distances, we need to add a statement (usually a function call) before every predicate. These statements should not alter the behaviour of the CuT, i.e., they should not have any side effect on the original program. They should only compute the branch distance and inform the testing environment of it. If a constituent of a predicate has a side effect (like the ++ operator or a function call that can change the state of the program), testability transformation should be used to remove the side effects [22]. Otherwise, such side effect could be executed more than once, changing the behaviour of the program.

In the environment that has been developed, it was chosen to use the program *srcML*[14] for translating the source code of the CuT in an XML representation. All instrumentations and testability transformations are done on the XML version. Afterwards, srcML is used to get the modified Java source code from the XML representation. This new source code is used only for testing purposes. It will be discarded after the testing phase is finished. The human tester should not care about this transformed java source code.

3 Search Algorithms

There is not a search algorithm that outperforms all the other algorithms [57]. Therefore, when a new problem needs to be addressed, different algorithms should be compared. This facilitates a deeper understanding of the search problem, so that more appropriate search algorithms can be developed as the next step. In this paper five search algorithms are used: Random Search (RS), Hill Climbing (HC), Simulated Annealing (SA), Genetic Algorithms (GAs) and Memetic Algorithms (MAs). They have been chosen because they represent a good range of different search algorithms. RS is a natural baseline used to understand the effectiveness of the other search algorithms. SA and GAs are very popular global search algorithms that have been applied to successfully solve many tasks. On the other hand, HC is local search algorithm, and it is important because from its performance compared to a global search algorithm we can get more information about the search landscape. Finally, MAs are hybrid algorithms that combine together local and global searches. If for the problem addressed in this paper a local search is not always better than a global search (or vice-versa), combining them might lead to even better results.

Although there are different variants of these search algorithms, in this paper only some simple forms are investigated. In fact, a single parameter of a search algorithm might heavily influence its performance, and many tests need to be carry out to find a good combination of parameters. Otherwise, it would not be fair to compare two search algorithms in which one has a poor choice of its parameters. Therefore, because five different search algorithms are considered at the same time, only limited tests on their variants were possible. Hence, their base forms were preferred because they lead to more fair comparisons among the different search algorithms.

3.1 Using Random Search

Although Random Search is the easiest among the search algorithms, it may give good coverage. The only problem is that we need to define the length of sequences that will be generated during the search. If we do not put any constraint on the length (besides of course the highest value that the variable that stores the length can have), there can be extremely long sequences. Not only the computational time can be too expensive, but such long sequences will be also useless. Therefore, it is better to put an upper bound L to the length. The sequences can still have a random length, but it will be always lower than L. Random search can be implemented in the following way:

1. generate at random a sequence S_i, with $len(S_i) < L$.

2. compare S_i with the best sequence seen so far. For comparison, use formula (1). If S_i is better, store it as the new best.

3. if the search is not finished (due to time limit or number of sequences evaluated), go to step 1.

4. return the best sequence seen so far.

Note that the only parameter that needs to be set is the upper bound L. Exploiting the branch distances is useless in a random search.

7

3.2 Using Hill Climbing

Hill Climbing (HC) is a local search algorithm. It needs that a neighbourhood N of the current sequence S_i is defined. The search will move to a new solution $S_j \in N$ only if S_j is better. If there are no better solutions in N, a local optimum has been reached. In such a case, a restart of the algorithm from a new random solution can be done.

We need to define N. Its size has a big impact on the performance of the algorithm [61]. We can think of three types of operations that we can do on S_i for generating N. Others types of operations will be investigated in future work.

Removal of a single FC from S_i. There will be $len(S_i)$ different neighbour sequences due to this operation.

Insertion of a new FC in S_i. There are $len(S_i) + 1$ different positions in which the insertion can be done. For each position, there are M different methods that can be inserted. Due to the too large search space, the input parameters for the FC will be generated at random.

Modification of the parameters of a FC. All FCs in S_i will be considered, except for the FCs with no parameters. If a parameter is an *index i* (see 2.2), we will have two operations that modified i by ± 1. Otherwise, if the parameter belongs to Z, we will consider two operations which will replace the parameter with the two closest elements in Z according to their natural order.

The branch distance should be used carefully. In fact, if (8) is used, the HC can be driven to increase the length to try to cover a particular difficult branch. If it falls in a local optimum without covering that branch, the result sequence could be unnecessarily too long. Because HC finds a local optimum, it cannot decrease the length of the sequence. Hence, our HC starts the search using (8) and then, when a local optimum is reached, it continues the search from that local optimum using (1) instead.

3.3 Using Simulated Annealing

The Simulated Annealing (SA) [26, 52] is a search algorithm that is inspired by a physical property of some materials used in metallurgy. Heating and then cooling the temperature in a controlled way bring to a better atomic structure. In fact, at high temperature the atoms can move freely, and a slow cooling rate makes them to be fixed in suitable positions. In a similar way, a temperature is used in the SA to control the probability of moving to a worse solution in the search space. The temperature is properly decreased during the search.

The use of SA has been investigated. A variant of SA that uses the Metropolis procedure [39] has been employed. The neighbourhood of the current sequence N is defined in the same way as in the Hill Climbing algorithm. It is not easy to define the *energy* of a sequence S_i. According to the Metropolis procedure, the energy is used to compute the probability that a worse sequence will be accepted. Such probability is:

$$W = \exp\left(-\frac{E(S_{i+1}) - E(S_i)}{T}\right) . \tag{9}$$

The problem is that we need to properly combine in a single number two different objectives as the coverage and the length of a given sequence. We need that, if S_i is better than S_j according to (1), than $E(S_i) < E(S_j)$. In formulae:

$$E(S_i) = f(cov(S_i)) + g(len(S_i)) , \tag{10}$$

$$f(NB) = 0 , \tag{11}$$

$$cov(S_i) > cov(S_j) \Rightarrow f(cov(S_i)) < f(cov(S_j)) \ \forall S_i, S_j \in S , \tag{12}$$

$$cov(S_i) > cov(S_j) \Rightarrow E(S_i) < E(S_j) \ \forall S_i, S_j \in S , \tag{13}$$

$$len(S_i) > len(S_j) \Rightarrow g(len(S_i)) > g(len(S_j)) \ \forall S_i, S_j \in S . \tag{14}$$

The function f is used to weigh the coverage, and it can easily be written as:

$$f(S_i) = NB - cov(S_i) . \tag{15}$$

On the other hand, the function g weighs the length. Due to (13) and the fact that the coverage is a positive natural value, we have:

$$0 \leq g(len(S_i)) < 1 \ \forall S_i \in S . \tag{16}$$

It should be less than 1, otherwise it is possible that a sequence can have a lower energy comparing to a longer sequence with higher coverage. One way to do it is :

$$g(len(S_i)) = 1 - \frac{1}{1 + len(S_i)} \; . \tag{17}$$

Using (15) and (17), we can rewrite (10) as:

$$E(S_i) = NB + 1 - \left(cov(S_i) + \frac{1}{1 + len(S_i)} \right) \; . \tag{18}$$

Although the latter formula defines the energy of a sequence in a proper way, it should not be used in the Simulated Annealing algorithm with the neighbourhood described before. This energy can deceive the search letting it looking at sequences always longer at every step. Consider the case Q in which a potential new sequence K is generated by S_i by adding a FC that does not change the coverage, but that increases the length. Thus, $cov(K) = cov(S_i)$ and $len(k) = len(S_i) + 1$. The new sequence K will be accepted as S_{i+1} by the probability (9). Therefore,

$$W_{Q,i} = \exp\left(-\frac{1}{L^2 \cdot T} \right) , \tag{19}$$

$$L^2 = len(S_i)^2 + 3 \cdot len(S_i) + 2 \; . \tag{20}$$

If the temperature T decreases slowly (as it should be), it is possible that $W_{Q,i+1} > W_{Q,i}$ because the length of sequence has increased. Although it should be mathematically proved, this behaviour has a high likelihood to tend to increase the length of the sequence. Empirical tests confirm it.

Instead of (17), we can think to use something like $g(len(S_i)) = \frac{len(S_i)}{M}$. However, this is not reasonable. In fact, we cannot set any finite M such that (13) is always true. Although we can limit M to the maximum length that the actual machine can support/handle, we will have in such a case that the weight of the length has little impact on the energy for any reasonable value of the length. I.e., $g(len(S_i))$ will be very close to zero for any S_i encountered during the search.

For all these reasons, the Simulated Annealing algorithm used is slightly different from the common version. Regardless of any energy, a new sequence K will be always accepted as S_{i+1} if (1) is true. Otherwise, the Metropolis procedure will be used. The energy function will be:

$$E(S_i) = NB - cov(S_i) + \alpha \cdot len(S_i) \; . \tag{21}$$

The constant α has a very important role on the performance of the SA. Before studying what is its best value, we need some assumptions:

- the likelihood that the sequence K will be generated using a *removal* operation on S_i is the same as having an *insertion*. I.e., $P(rem) = P(ins)$.

- the operations used to generate K can increase or decrease the length only by one. I.e., $len(K) - len(S_i) \in \{-1, 0, 1\}$.

- all assumptions in section 2.1 hold.

There is the problem that, due to (21), it is possible that there can be no change in the energy (i.e., $W = 1$) even if the new state is worse according to (1). In such cases, some particular worse sequences will always be accepted regardless of the temperature T. We will firstly analyse the values of α for which that thing does not happen. Afterwards, we will study how the SA behaves when α is not a chosen properly. There are different situations:

1. $len(S_{i+1}) - len(S_i) = 0$. According to (1), we have that $cov(S_{i+1}) \leq cov(S_i)$, otherwise the new sequence would have already been accepted. Therefore, $W = 1$ iff $cov(S_{i+1}) = cov(S_i)$. That means that if there are no changes in both the coverage and length, the new sequence S_{i+1} will always be accepted. This is not a problem for SA, but for other algorithms as Hill Climbing this rule can generate an infinite loop in the search. In this scenario, the value of α has no influence.

2. if $cov(S_{i+1}) = cov(S_i)$, we still need to discuss when $len(S_{i+1}) - len(S_i) = 1$. Note that, due to (1), it cannot be minus one. In that case, we have $W = \exp\left(-\frac{\alpha}{T} \right)$. So we need $\alpha > 0$.

3. $cov(S_{i+1}) < cov(S_i)$ and $len(S_{i+1}) - len(S_i) = 1$. In such a case, we can set $\alpha \geq 0$ to guarantee that $W < 1$.

9

4. the worst case is when $cov(S_{i+1}) < cov(S_i)$ and $len(S_{i+1}) - len(S_i) = -1$. In fact, one objective (the coverage) gets worse, but at the same time the other objective (the length) gets better. For having $W < 1$, we need that $E(S_{i+1}) - E(S_i) = cov(S_i) - cov(S_{i+1}) - \alpha > 0$. Therefore, given

$$M = \min(cov(S_i) - cov(S_{i+1})) \,, \\ \forall S_i \in S \;\; \forall S_{i+1} \in \{S_j | S_j \in N_{S_i}, cov(S_j) < cov(S_i)\} \,, \tag{22}$$

we should have:

$$\alpha < M \,. \tag{23}$$

In our case, we have $M = 1$ because the coverage is always a natural value. Thus, $\alpha < 1$.

The range of value for α such that all the previous conditions are true at the same time is :

$$0 < \alpha < M \,, \tag{24}$$

with $M = 1$. However, it is important to notice that, for the energy defined in (21), we cannot use the *branch distance*. Otherwise, the only lower bound for M will be zero. In such a case, we will have $0 < \alpha < 0$ that has no solution. However, there are no problems to use (8) instead of (1) for deciding if a new sequence is better or not.

If α is chosen not in the range defined in (24), the SA algorithm will be deceived.

$\alpha \geq 1$, any neighbour S_j of S_i with worse coverage, shorter length and $cov(S_j) + \alpha \geq cov(S_i)$, will always be accepted as S_{i+1} regardless of the temperature T. Because it is common that any global optimum for a generic container has at least one FC that contributes to the coverage only by one, the SA with $\alpha \geq 1$ cannot be guaranteed to converge. Besides, from empirical tests, the performances of the SA are so poor that even a Random Search performs better than it.

$\alpha \leq 0$, any neighbour S_j that does not reduce the coverage will always be accepted, even if the length increases. Although it does not seem a problem because a S_j with same coverage but shorter length is always accepted due to (1), the SA will tend to move its search to sequences always longer and longer. That can be explained if we consider the probabilities that S_{i+1} has been generated by S_i with an insertion or by removing a FC. Given an operation (op), the probability that applying it to S_i the new sequence will be accepted (acc) is:

$$P(acc|op)_i = \begin{cases} 1 & \text{if } cov(S_{i+1}) \geq cov(S_i) \,, \\ W_i & \text{otherwise} . \end{cases} \tag{25}$$

Due to the assumption in section (2.1), the probability of an accepted insertion (ins) is:

$$P(acc|ins) \approx 1 \,. \tag{26}$$

On the other hand, the probability that a removal (rem) is accepted is:

$$P(acc|rem) \approx P(cov(S_{i+1}) = cov(S_i)|rem) + \\ W_i \cdot P(cov(S_{i+1}) < cov(S_i)|rem) \,. \tag{27}$$

Let R_i be the number of redundant FCs in S_i, i.e. if we remove any of this FCs the coverage does not change. We will have:

$$P(cov(S_{i+1}) = cov(S_i)|rem) = \frac{R_i}{len(S_i)} \,, \tag{28}$$

$$P(cov(S_{i+1}) < cov(S_i)|rem) = \frac{len(S_i) - R_i}{len(S_i)} \,. \tag{29}$$

Therefore, using (28) and (29), we can write (27) as:

$$P(acc|rem) \approx W_i + (1 - W_i) \cdot \frac{R_i}{len(S_i)} \,. \tag{30}$$

Unless we want to change how the neighbourhood is defined, i.e. $P(rem) = P(ins)$, it is more likely that S_{i+1} has been generated from an insertion operation because $R_i < len(S_i)$. I.e. ,

$$P(ins|acc) > P(rem|acc) \,. \tag{31}$$

This is particularly true when S_i is close to a local optimum for the length objective (i.e., when R_i is close to zero). Only for an infinite length the two probabilities are the same, due to:

$$\lim_{len(S_i)\to\infty} \frac{R_i}{len(S_i)} = 1 \ . \tag{32}$$

Therefore, the SA tends to look at sequences that are likely to be always longer than the previous at any step. Besides, such SA diverges from any global optimum, because they are sequences of finite length.

Not only does α need to verify (24), but it should also be chosen carefully. In fact, the lower α is, the higher the average of the length of the sequences S_i will be. This happens because the weight of the length on the energy (21) decreases. Thus, the probability (9) of accepting a new longer sequence gets higher. The higher this average is, the more likely it is that the coverage will be greater. But at the same time the computational cost will increase as well.

In the former description, the SA has been studied with a fixed neighbourhood. However, it has been shown that the SA performs better if the neighbourhood size changes dynamically according to the temperature [61, 62]. The idea is to have a large size at the beginning of the search to boost the *exploration* of the search space. Then, the size should decrease to allow the *exploitation* of the current search region. It can be easily done if we consider, for getting the neighbourhoods of S_i, K_i operations on S_i. I.e., the new S_{i+1} will be generated using K_i operations on S_i instead of only one. Let K_0 be the initial size and N the total number of iterations of the search. We can write the size K_i at the iteration i as:

$$K_i = 1 + (k_0 - 1) \cdot \frac{N - i}{N} \ . \tag{33}$$

In that way, the neighbourhood size starts with a value of K_0, and decreases uniformly until it arrives at 1.

3.4   Using Genetic Algorithms

Among the most used metaheuristics in software testing there are the Genetic Algorithms (GAs) [23, 40]. GAs are a global search metaheuristic inspired by the Darwinian Evolution theory. Different variants of this metaheuristic exist. However, they rely on three basic features: *population*, *crossover* and *mutation*. More than one solution is considered at the same time (population). At each *generation* (i.e., at each step of the algorithm), the solutions in the current population generate *offspring* using the crossover operator. This operator combines parts of the *chromosomes* (i.e., the solution representation) of the offspring's parents. These new offspring solutions will fill the population of the next generation. The mutation operator is applied to make little changes in the chromosomes of the offspring.

To apply a GA for testing containers, we need to discuss:

Encoding, a chromosome can be viewed as sequences of FCs. Each single gene is a FC with its input parameters. The number of parameters depends on the particular method.

Crossover, it is used to generate new offspring combining between them the chromosomes of the parents. The offspring generated by a crossover are always well formatted test sequences. Therefore, no post processing is needed to adjust a sequence. The only particular thing to note is that the parents can have different lengths. In such cases, the new offspring's length will be the average of the parents's length. A single point crossover takes the first K genes from the first parent, and the following others from the tail of the second parent.

Mutations, they change an individual by a little. They are the same operations on a test sequence as described in sec. 3.2.

Fitness, the easiest way to define a fitness for the problem is Eq.(2). Anyway, we need to introduce the branch distance (5) in the fitness, otherwise the search will be blind:

$$f(S_i) = cov(S_i) + \alpha(1 - B(S_i)) + (1 - \alpha)\frac{1}{1 + len(S_i)} \ , \tag{34}$$

where α is in $[0, 1]$ (a reasonable value could be 0.5). Note that, for both the fitness, the following predicate is always true:

$$cov(S_i) > cov(S_j) \Rightarrow f(S_i) > f(S_j) \ \forall S_i, S_j \in S \ . \tag{35}$$

11

Although the latter fitness gives good results, it can deceive the search in the same way as it happens in the Simulated Annealing. I.e., the use of the branch distance can lead to longer sequences without increasing the coverage in the end. To address this problem we can use a rank selection, but in a *stochastic* way [44]. I.e., for the selection phase, we can rank the individuals in the population using randomly either the fitness (2) or (34).

Fitness sharing, it is used to maintain a degree of diversity inside the population. Individuals that have a chromosome close to others in the population will see their fitness decreasing. Common ways to define the distance between two individuals are the Euclidean and Hamming distances. When they are not suitable for a particular problem, a specific new distance measurement can be defined. Although the use of fitness sharing usually gives better results, it is not easy to define a right distance for this problem. In fact, the order of the FCs in the sequence is very important, and a single difference at the beginning of the sequence can completely change its behaviour. Besides, there can be a lot of read-only or redundant FCs that can be modified without altering the behaviour of the sequence. Therefore, in this work we did not used any sort of fitness sharing.

At any rate, distance functions based on the execution traces instead of being based on the chromosomes might be considered. However, choosing which information of an execution should be collected and used is not straightforward. Moreover, keeping track of those execution traces might have a significant overhead cost.

3.5   Using Memetic Algorithms

The Memetic Algorithms (MAs)[41] are a metaheuristic that uses both global and local search (e.g., a GA with a HC). It is inspired by the Cultural Evolution. A *meme* is a *unit of imitation in cultural transmission*. The idea is to mimic the process of the evolution of these memes. From an optimisation point of view, we can approximately describe a MA as a population based metaheuristic in which, whenever an offspring is generated, a local search is applied to it until it reaches a local optimum.

The MA we used in this paper is fairly simple. It is built on our GA, and the only difference is that at each generation on each individual an Hill Climbing is applied until a local optimum is reached. The cost of applying those local searches is high, hence the population size and the total number of generations is lower than in the GA.

4   Case Study

To validate the techniques described in this paper, the following containers have been used for testing: `Vector`, `Stack`, `LinkedList`, `Hashtable` and `TreeMap` from the Java API 1.4, package `java.util`. On the other hand, `BinTree` and `BinomialHeap` have been taken from the examples in JPF [51]. Table 2 summarises their characteristics. The coverage values are referred to the branch coverage, but they also include the calls to the functions. The achievable coverage is based on the highest coverages ever reached during around a year of several experiments with our framework. Human inspections of the source codes confirmed that all the other non-covered branches seem either unfeasible or not reachable by any test sequence that is framed as we do in this paper. Although these two arguments give strong support on the fact that those coverage values cannot be exceeded by any search algorithm that uses a search space as the one described in this paper, they do not constitute a proof.

4.1   Comparing the Search Algorithms

The different algorithms described in this paper have been tested on the cluster of seven containers previously described. When an algorithm needs that some of its parameters should be set, experiments on their different values had been done. Anyway, these parameters are optimised on the entire cluster, and they remain the same when they are used on the different containers. Although different tests on these values have been carried out, there is no guarantee that the chosen values are the best. Regarding the parameters defined in section 2.2, we used $N = -2$, $P = 58$, $\delta = 0.1$ and $\alpha = 1.3$. Random Search looks to sequences up to length 60. The cool rating in SA is set to 0.999, with geometric temperature reduction and one iteration for temperature. The initial neighbourhood size is 3. The value of α in (21) is 0.5. The GA uses a single point crossover with probability 0.2. Mutation probability of an individual is 0.9. Population size is 64. Rank selection is used with a bias of 1.5. At each generation, the two individuals with highest fitness values are directly copied

to the next population without any modification (i.e., the elitism rate is set to two individuals for generation). The MA uses a single point crossover with probability 0.9. Population size is 8. Rank selection is used with a bias of 1.5. Elitism rate is set to one individual for generation.

Each algorithm has been stopped after evaluating 100, 000 sequences. The machine used for the tests was a Pentium with 3.0 GHz and 1024 M of ram running Linux. The tests have been run under a normal usage of the CPU. Table 3 reports the performances of these algorithms on 100 trials. Using the same data, a Mann Whitney U test [30] has been used to compare the performances of the different algorithms. The performance of a search algorithm is calculated using function (2). The level of significance is set to 0.05. HC and SA are statistically equivalent on `Hashtable`. HC and GA are equivalent on `Stack`, `LinkedList` and `BinomialHeap`. Finally, HC and MA are equivalent on `Stack`, `Vector`, `BinomialHeap` and `BinTree`. If we compare table 3 with the highest achievable coverages (reported in table 2), we will see that only `TreeMap` is difficult to test. Besides, from that table and from the statistical tests the MA seems the best algorithm. A part from `Vector`, it gives the best results on `LinkedList`, `Hashtable`, `TreeMap` and it is among the best algorithms on the other containers. Moreover, the HC seems to have better performance than the GA. This is a very interesting result, because usually local search algorithms are suggested of not being used for generating test data [56]. Although the search spaces for software testing are usually "complex, discontinuous, and non-linear" (J. Wegener et al. [56], 2001), the evidences of our experiments lead us to say that it does not seem true for container classes. However, more tests on a bigger cluster of containers and the use of different search algorithms are required.

Table 4 compares the performance of the MA when it is stopped after 10, 000 and 100, 000 fitness evaluations. A part from `TreeMap`, MA is able to get high quality results in a very short amount of time.

The performances of the RS deserve some comments. In fact, they are sensibly worse than the performances of the other algorithms. Although a reasonable coverage can be achieved, RS poorly fails to obtain it without a long sequence of FCs. That can be explained by the fact that not only the search landscape of the input parameters is large and complex, with only a small subset of it that gives optimal results, but also the search space of the methods has similar characteristics. E.g., some methods need to be called few times (e.g., `isEmpty`) whereas others need to be called many times (e.g., `insert`) to cover a particular branch. In a random search the probabilities of their occurrences in the test sequence are the same, so we will have redundant presences of functions that need to be called only few times. Moreover, the function calls require to be put in a precise order, and obtaining a random sub-sequence with that order might be difficult if the total length is too short.

4.2   Known limits

The system described in this paper is not able to generate input data for covering all the branch statements in the source code of the CuT. This is due to different reasons:

- The system also tries to cover the branches in the `private` methods. Anyway, the generated test sequences do not access directly to the `private` methods of the container. They can be executed only if at least one public method can generate a chain of function calls to them. Although using the Java *reflection* a driver can directly call private methods of the CuT, besides the fact that a driver can be located inside the CuT, it has been preferred to test directly only the `public` methods. It is possible to do assumptions on the semantics of the public methods of a container (see section 2.2), but nothing can be said about the private ones. Thus, the proposed space reduction techniques cannot be applied to them.

- the very few `public` methods with odd input parameters (11 on a total of 105 in our test cluster) cannot be directly called. If they are not called by other methods, they will not be tested.

- some methods can return objects which class is implemented inside the same file of the CuT or even in the same method. For example, the method `keySet` in `TreeMap` returns a `Set` which class is implemented inside `keySet`. Because the system does not call any method on the returned objects of the tested methods, such internal classes cannot be tested.

- for a given test sequence, only one constructor is called. One option to solve this problem might be to use multiple sequences, each one that uses a different constructor.

- some branches can be infeasible. This is a general problem that is not related to the used testing system.

13

4.3 Effects of the Novel Testability Transformation

The use of the testability transformation described in section 2.3 has been investigated. Besides transforming the calls to `equals`, the semantics of `compareTo` is changed too. Table 1 shows the number of such method calls in the cluster of the container that take advantage from the transformation. Note that the used implementations of the `BinTree` and the `BinomialTree` do not handle objects but integer values. This is why the above methods do not appear in them. For `TreeMap`, it has also been considered the calls to the private method `compare`, because it includes and replaces the calls to `compareTo`.

Tests on the different containers have been carried out. However, no particular improvements on the performance of the algorithms have been found. That can be explained by considering that the transformation is applied only to very few branches. Besides, these branches are "easy". I.e., they are covered without any problem even if no guidance has been given to the search algorithms. However, that testability transformation gives no overhead to the computational cost of the search. Therefore, it should always be applied, because there might be containers that have difficult branches that depend on the `equals` and `compareTo` methods.

5 Related work

The previous work on Test Data Generation for container classes can be divided in two main groups: one that includes traditional techniques based for example on symbolic execution, and a second group that uses metaheuristic algorithms like the Genetic Algorithms.

Anyway, doing comparisons with systems developed by other authors is not easy in Software Testing. In fact, there is no common benchmark cluster on which different authors can test and compare their techniques [2]. Although the same classes (e.g., taken from the Java API) can be considered for testing, instrumentations can be done in many different ways. Hence, if there is no access to the instrumented files, no reasonable comparisons can be done, because how the coverage is defined and which parts of the code are actually instrumented might be very different. Even though some systems might have freely available source codes, a lot of effort might be required for adapting a testing environment to handle different types of instrumentations. All these problems make it difficult to evaluate the performance of a novel technique against existing ones.

5.1 Traditional Techniques

There have been different works focused on testing containers. Early works that use *testgraph analysis* can be found in [32], but they require a lot of effort from the tester. In the same way, techniques that exhaustively search a container up to size N [43] cannot be applied to a new container without the help of the tester. In fact, the tester would be responsible for providing both the *generator* and the *test driver*.

However, a Java container is a an Object-Oriented (OO) program. Therefore, any tool that claims to automatically generate input data for an OO software should also work for a container. Different experimental tools have been developed to automatically test OO software. The early ASTOOT [17] generates tests from algebraic specifications of the functions. Korat [8] and TestEra [31] use isomorphic generation of data structures, but need predicates that represent constraints on these data structures. Rostra [59] uses bounded exhaustive exploration with concrete values. On the other hand, tools that exploit the *symbolic execution* [25] include for example Symstra[60], Symclat [15], the work of Buy et al. [9] and the model-checker Java PathFinder used for test data generation [50]. Although promising results have been reported, these techniques are unlikely to scale well. Besides, as clearly stated in [15], at the moment they have difficulties in handling non-linear predicates, non-primitive data types, loops, arrays, etc. Although [50] can consider objects as input, it needs to exploit the specification of the functions (in particular the precondition) to initialise such objects. It is important to outline that our system does not need neither any specification or any help from the user at all. It only needs to know that the CuT is a container. Hence, any container from the Java API (for example) can be automatically tested without any particular help from the user.

A work that is specific on container is [51]. Its source codes are freely available. That system is built on Java PathFinder, and uses exhaustive techniques with symbolic execution. To avoid the generation of redundant sequences, it uses *state matching*. During the exhaustive exploration, the abstract shapes of the CuT in the heap are stored. If an abstract shape is encountered more than once, the exploration of that sub-space is pruned. That system has the same problems that we described in section 4.2. Besides, it does not handle object references. E.g., the authors of [51] use

the Java API `TreeMap` implementation for some of their empirical tests, but they had to change all of its code to replace the object "elements" with `int` values. They state that their system could handle objects, but in that case they would have serious scalability issues. A comparison with such system is not trivial, because three important rules for a fair comparison are not satisfied: "competing algorithms would be coded by the same expert programmer and run on the same test problems on the same computer configuration" (Barr et al. [6], 1995). E.g., when comparisons based on time are done, the programming skills of the authors have an important role on the performance of the implemented algorithms. Doing a completely fair comparison between our system and [51] on the testing of the `TreeMap` (for example) was not possible for the following reasons:

- developing a testing environment requires a lot of time, and redeveloping [51] was unreasonable.

- the problem under test is not exactly the same: we use the `TreeMap` version of Java without any modification, and [51] does not. Besides, we consider the branch coverage when they consider basic block and predicate coverage.

- the computer configurations used for the tests are different (although we could have run their experiments again on our machine because their code is freely available).

However, we can highlight that our testing problem is more complex and our techniques obtains competitive results in much less time. For example, in [51] only 2 public methods are tested for the `TreeMap`, whereas we consider 17 public methods. Furthermore, their best predicate coverage is obtained with a length of 20 and required more than four minutes to be obtained. In our problem, we found our best coverage with sequences of length around 50. These sequences do not have any redundant/useless FC inside, because a local search always converges to a local optimum. Because the MA can be run for any arbitrary amount of generation, from table 4 we can see that it already gets reasonable results after few seconds, and competitive results in around half a minute. Due to the exhaustive nature of the techniques in [51] (although some heuristics to prune the search space are employed), we are sceptical of the fact that they might handle problems so complex as the one we address in this paper in a reasonable amount of time. Although the machine we used for the empirical tests is slightly faster than the one used in [51], we are enough confident to claim that meta-heuristic based techniques seem better than the techniques described in this section, because more complex problems are solved in significantly less time. However, more experiments are required to support this claim. Besides, hybrid approaches that take the best of the two worlds might lead to even better results. Another point that we did not discuss is the use of testability transformations. We only employed the one that we presented in section 5.1, but many others could also have been applied, increasing the performances of the metaheuristics even further.

5.2 Metaheuristic Techniques

The use of search based techniques (e.g., GAs) for testing OO programs has started to be investigated in the last few years.

Tonella [48] used GAs for generating unit tests of Java programs. Solutions are modelled as sequences of function calls with their inputs and caller (an object instance or a class if the method is static). Special crossover and mutation operators are proposed to enforce the feasibility of the generated solutions. Similar work with GAs has been done by Wappler and Lammermann [53], but they used standard evolutionary operators. This can cause the generation of infeasible individuals, which will be penalised by the fitness function. Besides, they investigated the idea of separately optimising the parameters, the function calls and the target instanced objects. Strongly Typed Genetic Programming (STGP) has been used by Wappler and Wegener [55] for testing Java programs. They extended their approach by considering the problem of the raised exceptions during the evaluation of a sequence [54]. If an exception is thrown, the fitness will consider how distant the method in which it is thrown is from the target method in the test sequence. In his master's thesis [46], Seesing investigated the use of STGP as well. Liu et al. [29] used a hybrid approach, in which Ant Colony Optimisation is exploited to optimise the sequence of function calls. Multi-agent Genetic Algorithm is used then to optimise the input parameters of those function calls. Also Cheon et al. [12] proposed an evolutionary tool, but they implemented and tested only a random search. They proposed to exploit the specification of the functions that return boolean values to improve the fitness function [11]. It is important to highlight that, even if a testing tool is designed for handling a generic program, container classes are often used as benchmarks. Our system considers only containers and has some specific limitations that the systems described in this section do not have (e.g., there is only one object instance on which the functions can be called on). These limitations were introduced to do fairer comparisons with the traditional techniques

described in section 5.1. Hence, comparing our system with the one described in this section would not be very significant.

In our previous work [1], we proposed a novel representation with novel search operators and a dynamic search space reduction for testing OO containers. The use of Estimation of Distribution Algorithms on this problem has been investigated with a collaboration with Sagarna [45].

6 Conclusions

We presented search based test data generation techniques for containers. Search algorithms like Random Search (RS), Hill Climbing (HC), Simulated Annealing (SA), Genetic Algorithms (GAs) and Memetic Algorithms (MAs) have been applied. The objective of minimising the length of the test sequences has been addressed as well. Because that minimisation can be misleading, discussion on how and why a search algorithm can be deceived has been presented, besides how to avoid it. Search space reductions and a novel testability transformation have also been presented. Although different settings of the algorithm parameters can lead to different results, it has been empirically shown that our MA usually performs better than the other algorithms. Moreover, the HC resulted better than the GA. That can seem strange, because local search algorithms are suggested of not being used for generating test data [56]. Although the MA performs well on all the tested classes, the results on `TreeMap` are not completely satisfactory. That leads us to investigate new algorithms and new ways to improve their performance in the future.

Besides presenting a working system, we emphasise the need of comparing different search algorithms for a given software engineer problem. Too often, in the literature, there is a bias toward GAs. Although GAs have been applied successfully to a wide range of different problems, that does not necessarily mean that they will work well on a new problem [57]. The Search Based Software Engineering field is still too young for deciding what is the best search algorithm that should be applied to it. Regarding testing OO software, nature inspired algorithms seem to be better than the techniques based on symbolic execution and state matching, because they seem able to solve more complex test problems in less time.

The approach of trying to cover every branch at the same time with a single sequence is novel for the search based testing. Therefore, its performance needs to be compared to the traditional approach of testing each branch separately. Future work will try to shed light on which technique is better and why.

The discussed techniques have been applied to testing containers. If they are still valid and how to use them for testing generic OO software is a matter that needs to be investigated.

7 Acknowledgements

References

[1] A. Arcuri and X. Yao. A memetic algorithm for test data generation of object-oriented software. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2048–2055, 2007.

[2] A. Arcuri and X. Yao. On test data generation of object-oriented software. In *Testing: Academic and Industrial Conference, Practice and Research Techniques (TAIC PART)*, pages 72–76, 2007.

[3] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 43–52, 2004.

[4] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 2442–2454, 2003.

[5] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1329–1336, 2002.

[6] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Rescende, and W. R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, 1995.

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

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[9] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 39–48, 2000.

[10] T. Y. Chen and M. F. Lau. On the divide-and-conquer approach towards test suite reduction. *Information Sciences*, 152:89–119, 2003.

[11] Y. Cheon and M. Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1952–1954, 2006.

[12] Y. Cheon, M. Y. Kim, and A. Perumandla. A complete automation of unit testing for java programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290–295, 2005.

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

[14] M. L. Collard. Addressing source code using srcml. In *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC'05)*, 2005.

[15] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.

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

[17] R. Doong and P. G. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, pages 101–130, 1994.

[18] M. Ellims, J. Bridges, and D. C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[19] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[20] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1351–1358, 2002.

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

[22] M. Harman, M. Munro, L. Hu, and X. Zhang. Side-effect removal transformation. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension*, pages 310–319, 2001.

[23] J. H. Holland. *Adaptation in Natural and Artificial Systems, second edition*. MIT Press, Cambridge, 1992.

[24] IEEE-Standards-Board. Ieee standard for software unit testing: An american national standard, ansi/ieee std 1008-1987. *IEEE Standards: Software Engineering, Volume Two: Process Standards*, 1999.

[25] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[27] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[28] J. C. Lin and P. L. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64, 2001.

[29] X. Liu, B. Wang, and H. Liu. Evolutionary search in the context of object oriented programs. In *MIC2005: The Sixth Metaheuristics International Conference*, 2005.

[30] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[31] D. Marinov and S. Khurshid. Testera: A novel framework for testing java programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2001.

[32] J. McDonald, D. Hoffman, and P. Strooper. Programmatic testing of the standard template library containers. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 147–156, 1998.

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

[34] P. McMinn. *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD thesis, University of Sheffield, 2005.

[35] P. McMinn and D. Binkley. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop*, pages 165–182, 2005.

[36] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 2488–2500, 2003.

[37] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1363–1374, 2004.

[38] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1013–1020, 2005.

[39] N. Metropolis, A.Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091, 1953.

[40] J. Miller, M. Reformat, and H. Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, 2006.

[41] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech Concurrent Computation Program, C3P Report 826*, 1989.

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

[43] P.Netisopakul, L.White, J. Morris, and D. Hoffman. Data coverage testing of programs for container classes. In *Proceedings 13th International Symposium on Software Reliability Engineering*, pages 183–194, 2002.

[44] T. P. Runarsson and X. Yao. Stochastic ranking for constrained evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 4(3):284–294, 2000.

[45] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 438–444, 2007.

[46] A. Seesing. Evotest: Test case generation using genetic programming and software analysis. Master's thesis, Delft University of Technology, 2006.

[47] G. Tassey. The economic impacts of inadequate infrastructure for software testing, final report. *National Institute of Standards and Technology*, 2002.

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

[49] N. Tracey, J. Clark, K. Mander, and J. A. McDermid. An automated framework for structural test-data generation. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 285–288, 1998.

[50] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[51] 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.

[52] H. Waeselynck, P. T. Fosse, and O. A. Kaddour. Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Software Engineering*, 12(1):35–63, 2006.

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

[54] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 851–858, 2006.

[55] 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.

[56] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

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

[58] M. Xiao, M. El-Attar, M. Reformat, and J. Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.

[59] 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.

[60] 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.

[61] X. Yao. Simulated annealing with extended neighbourhood. *International Journal of Computer Mathematics*, 40:169–189, 1991.

[62] X. Yao. Comparison of different neighbourhood size in simulated annealing. In *Proceedings of the Fourth Australian Conf. on Neural Networks (ACNN'93)*, pages 216–219, 1993.

Container	Transformations
Stack	0
Vector	2
LinkedList	3
Hashtable	5
TreeMap	5
BinTree	0
BinomialHeap	0

Table 1: For each container in the tested cluster, the number of times that the novel transformation can be successfully applied is reported.

Container	LOC	FuT	Achievable Coverage
Stack	118	5	10
Vector	1019	34	100
LinkedList	708	20	84
Hashtable	1060	18	106
TreeMap	1636	17	191
BinomialHeap	355	3	79
BinTree	154	3	37

Table 2: Characteristics of the containers in the test cluster. The lines of code (LOC), the number of the public functions under test (FuT) and the achievable coverages for each container are reported.

Container	Search Algorithms	Coverage			Length		
		Mean	Variance	Median	Mean	Variance	Median
Stack	Random	10.00	0.00	10.00	6.64	0.41	7.00
	Hill Climbing	10.00	0.00	10.00	6.00	0.00	6.00
	Simulated Annealing	10.00	0.00	10.00	6.06	0.06	6.00
	Genetic Algorithm	10.00	0.00	10.00	6.00	0.00	6.00
	Memetic Algorithm	10.00	0.00	10.00	6.00	0.00	6.00
Vector	Random	85.21	1.52	85.00	56.99	7.73	58.00
	Hill Climbing	100.00	0.00	100.00	47.67	1.05	48.00
	Simulated Annealing	99.99	0.01	100.00	45.76	1.11	46.00
	Genetic Algorithm	99.99	0.01	100.00	46.87	1.63	47.00
	Memetic Algorithm	100.00	0.00	100.00	47.89	2.64	48.00
LinkedList	Random	69.96	1.82	70.00	55.27	14.00	56.00
	Hill Climbing	84.00	0.00	84.00	38.48	10.27	38.00
	Simulated Annealing	82.47	2.25	82.50	33.60	5.29	33.50
	Genetic Algorithm	83.83	0.26	84.00	36.66	3.64	36.00
	Memetic Algorithm	84.00	0.00	84.00	36.43	3.58	36.00
Hashtable	Random	92.92	1.17	93.00	54.45	25.97	56.00
	Hill Climbing	106.00	0.00	106.00	35.25	0.19	35.00
	Simulated Annealing	105.84	0.74	106.00	34.98	0.77	35.00
	Genetic Algorithm	101.14	6.50	100.00	31.10	6.31	30.00
	Memetic Algorithm	106.00	0.00	106.00	35.01	0.01	35.00
TreeMap	Random	151.94	5.85	152.00	54.11	26.87	55.00
	Hill Climbing	188.76	0.71	189.00	51.23	10.08	51.00
	Simulated Annealing	184.19	5.75	185.00	40.68	5.88	41.00
	Genetic Algorithm	185.03	3.46	185.00	42.14	8.44	42.00
	Memetic Algorithm	188.86	0.65	189.00	50.55	10.31	50.00
BinomialHeap	Random	77.52	0.29	77.50	47.08	100.24	47.50
	Hill Climbing	77.96	0.48	78.00	24.05	34.86	25.00
	Simulated Annealing	76.41	0.24	76.00	16.02	41.84	14.00
	Genetic Algorithm	77.70	0.92	77.00	19.08	24.54	16.00
	Memetic Algorithm	77.66	0.87	77.00	18.65	24.61	15.00
BinTree	Random	37.00	0.00	37.00	26.86	15.39	27.00
	Hill Climbing	37.00	0.00	37.00	9.02	0.02	9.00
	Simulated Annealing	36.98	0.02	37.00	9.38	0.40	9.00
	Genetic Algorithm	37.00	0.00	37.00	9.21	0.21	9.00
	Memetic Algorithm	37.00	0.00	37.00	9.00	0.00	9.00

Table 3: Comparison of the different search algorithms on the container cluster. Each algorithm has been stopped after evaluating up to 100, 000 solutions. The reported values are calculated on 100 runs of the test. Mann Whitney U tests show that the MA has the best performance on all the containers but Vector.

Container	Fitness Evaluations	Average Coverage	Average Length	Time (seconds)
Stack	10k	10.00	6.00	0.29
	100k	10.00	6.00	2.80
Vector	10k	99.50	73.70	8.89
	100k	100.00	47.89	78.58
LinkedList	10k	83.60	50.60	2.22
	100k	84.00	36.43	17.41
Hashtable	10k	105.90	36.10	1.31
	100k	106.00	35.01	11.75
TreeMap	10k	184.90	47.40	3.00
	100k	188.86	50.55	28.83
BinomialHeap	10k	77.30	19.10	1.87
	100k	77.66	18.65	16.22
BinTree	10k	37.00	9.30	0.18
	100k	37.00	9.00	1.49

Table 4: Performance of the Memetic Algorithm on the test cluster.