A Memetic Algorithm for Test Data Generation of Object-Oriented Software

Andrea Arcuri and Xin Yao

***Abstract*— Generating test data for Object-Oriented (OO) software is a hard task. Little work has been done on the subject, and a lot of open problems still need to be investigated. In this paper we focus on container classes. They are used in almost every type of software, hence their reliability is of utmost importance. We present novel techniques to generate test data for container classes in an automatic way. A new representation with novel search operators is described and tested. A way to reduce the search space for OO software is presented. This is achieved by dynamically eliminating the functions that cannot give any further help from the search. Besides, the problem of applying the branch distances of disjunctions and conjunctions to OO software is solved. Finally, Hill Climbing, Genetic Algorithms and Memetic Algorithms are used and compared. Our empirical case study shows that our Memetic Algorithm outperforms the other algorithms.**

I. Introduction

In spite of years of research efforts in software engineering, software is still far from being perfect and still suffers badly from various bugs. Finding and fixing bugs become a very expensive and time consuming issue for the entire software engineering industry. It is impossible to guarantee a bug-free software unless it is trivially small. Various techniques have been proposed to minimise the possibility of bugs, e.g., trying to identify them as early as possible in the software development cycle. One way to try to deal with this issue is to use *Unit Tests* [1]. 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 testing criterion, 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 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 [2], but a system that can generate an optimal set of unit tests for any generic program has not been developed yet. Lots of research has been done on procedural software, but very little on Object-Oriented (OO) software. In this paper we focus on a particular class 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 from the others is the computational cost of operations like insertion, deletion and finding of a data object.

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

Containers are used in almost every type of OO software, so their reliability is important. This paper presents a framework for automatically generating unit tests for container classes. The test criterion is *White Box Testing*. We consider only the *branch coverage*, but the techniques described in this paper can be easily extended also to other coverage criteria. The goal is to find a set of unit tests that, when executed, covers as many branches of the tested class as possible. A branch is covered when it is executed at least once. The task is modelled as a search problem [3]. This paper extends our previous work on this subject [4]. A Memetic Algorithm (MA) is employed and compared to a Hill Climbing (HC) and to a Genetic Algorithm (GA). Novel search operators are presented. They exploit the information that in a container is more important the relative order of the *storing elements* than their actual values. An informal analysis of these operators is given to support the empirical results. Furthermore, a novel Dynamic Search Space Reduction (DSSR) is presented. It is based on the idea that the read-only functions that are already covered can be removed from the search. This technique can be applied to any type of OO software. However, it has sense only when the interest is that of covering all the branches in a class with a single sequence of function calls. Novel branch distances are described for handling disjunctions and conjunctions of predicates. They are a problem in OO software, because the ones used for procedural software can throw exceptions during the monitoring of the execution flow. The novel branch distances are useful in any type of OO software, regardless of whether the branches are individually targeted or not.

Although the used programming language is Java, the techniques described in this paper can be applied to other object oriented languages.

The paper is organised as follows: section II describes the novel search operators. Section III explains the novel search space reduction. The novel branch distances follow in section IV. Section V describes the employed optimisation algorithms. The empirical results are reported in section VI. A brief description of the related work follows in section

VII. Finally, the conclusion of this paper is in section VIII.

II. Representations and Search Operators

The solutions for testing container classes are modelled as sequences S_i of function calls. 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. For example:

```
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 *Class under Test* (CuT), and not one for each of its branches (as usually happens in literature). The aim is to find a single S_i that executes all the branches in the CuT. The order of the FCs in S_i is important, because a FC may change the internal state of the objects on which it is called, and this might influence the behaviour of the FCs that will be called later on the same instance. Each FC has its own input parameters ($input\_list$), with the number of parameters depending on the function. The length of a sequence S_i is defined by the number of its FCs, and it is represented by $len(S_i)$. There are no limits either to the number of parameters that a FC can have or to length of a S_i. For each S_i, only one instance of the CuT is created. All the FCs of S_i are invoked on the same instance.

Only two types of parameters are considered: integer values that represent *indices* (locations in the container) and object references for the *elements* that a container can store or use as *keys*. In the test cluster (see section VI), only 11 public methods on a total of 105 require different types of input parameters.

The elements can be ordered and enumerated according to a *Natural Order*. Although the actual values of the elements are usually not important (only the relative order does), we enumerate the elements by giving them natural values. I.e., the element e_i has position i in the natural order. Elements can be references to any type of object (e.g., network sockets), but it is easier to think of them like *Integer* references. Besides, for testing purposes we need only a relative small set of objects [4]. The value `null` is treated as a special case. The functions under test used for the FCs are only the ones with these types of input parameters, that are *public* and that are implemented in the CuT (i.e., not overwritten methods belonging to any superclass are ignored). However, all branches of all the functions (*private* ones included) and of all the internal classes in the CuT are tried to be covered.

There are no restrictions on the value that an index can have. However, the lowest element in any S_i should be e_1. Besides, there should be no "holes" in the enumeration of the elements. I.e., given that e_m is the highest element in S_i, all the elements with index value in between 1 and m should be present in the sequence. An element e_j can be present in a S_i more than once. The following is a valid sequence:

```
container.insert(e3);  // 1
container.insert(e2);  // 2
container.get(2);      // 3
container.remove(e3);  // 4
container.insert(e1);  // 5
```

The variables `e1, e2`, and `e3` are references to objects, with $e1 < e2 < e3$ and there are no holes between them. For simplicity, we give significant names to variables that represent their values. For example, the variable with name `e1` is a reference to the object e_1.

Two novel search operators are defined for a sequence S_i. They are used to generate a new S_j that is "close" to the original S_i. They are special operators, because they may modify the values of all the input parameters in every FC.

A. Removal of a Function Call

A single FC is removed from S_i. There are $len(S_i)$ neighbour solutions that can be generated with this operation. If the FC has some input elements, its elimination may generate some holes among the elements. Consider the previous example. If line 1 is eliminated, no holes are generated because `e3` is also present in line 4. On the other hand, if line 2 is removed, a hole is generated. When such situation arises, we need to fill the hole. Let e_k be an element of the removed FC that was present in the sequence only once. In order to fill the hole, we can just replace each e_i greater than e_k with e_{i-1}. If more than one unique element is presented in the removed FC, the holes should be filled from the one that has highest position backward to the lowest.

B. Insertion of a New Function Call

A single FC is inserted in the sequence S_i. Both the position and the type of the function need to be chosen. Due to the large search space, the input parameters (if any) are chosen at random. There are $len(S_i)*M$ neighbour solutions that can be generated with this operation, with M being the number of different methods under test.

Although the parameters are chosen at random, they are taken from a restricted set. The indices can be chosen inside the range of the stored elements plus a few outside of it [4]. Assuming that the tested functions can add only one element at a time, the search space for the indices can be bound by the length of S_i. For the elements, a random decision is made: either an element already present in S_i is created with probability ρ, or a new one is created with probability $1-\rho$. The first case is straightforward. In the latter case, given a random e_t with $e_1 \leq e_t \leq e_{m+1}$, before inserting it with its FC in S_i, all e_k with $e_k \geq e_t$ have to be replaced by e_{k+1}. Note that in this way the relative order between the former elements remains unchanged. An example will help to clarify the latter algorithm: consider the previous example sequence, and let `container.remove(e3)` be inserted between position 1 and 2; we will have:

```
container.insert(e4);  // 1
container.remove(e3);  // 2
container.insert(e2);  // 3
```

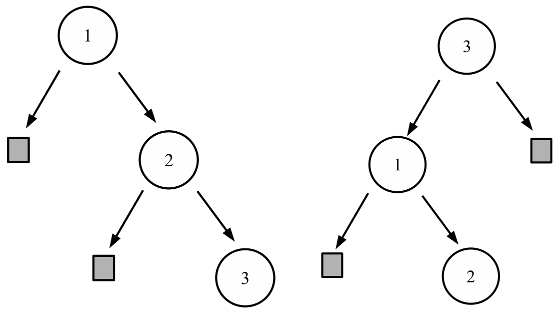

Fig. 1. Heap states of a binary search tree. On the left is represented the state after inserting $< e_1, e_2, e_3 >$. On the right there is the insertion of $< e_3, e_1, e_2 >$.

```
container.get(2);       // 4
container.remove(e4);  // 5
container.insert(e1);  // 6
```

C. Simple Representation

To evaluate the performance of the novel solution representation and its operators, they need to be compared with a simpler representation. The container elements will be the same, but there can be holes in their order. Their values are bounded by the length of the current sequence. The removal search operator will not try to fill any hole. The insertion of a FC will have the elements with random values between 1 and the length of the sequence. It will not alter any other FC in the S_i. This representation is the same as the one used in [4].

D. Motivations

In this section, we describe why these novel representation and operators have been developed. Among the most difficult containers to test, there are the ones that behave differently in respect of the order in which the elements are inserted and removed. For example, if we insert $< e_1, e_2, e_3 >$ (i.e., firstly we insert e_1, then e_2 and finally e_3) or $< e_3, e_1, e_2 >$ in a list, its behaviour will be the same. However, if the same sequences are inserted in a *binary search tree*, the two sequences will generate different behaviour. Not only will the shape of the container in the heap be different, but above all, the executed branches of the source code of the container will also be different. In the first case, $< e_1, e_2, e_3 >$, it does not happen that a new node is inserted on the left of a previous one, because the values are inserted in ascending order (i.e., each new inserted element has always the greatest value). Therefore, the code related to the insertion of the left side of a node is never executed. On the other hand, in the case of $< e_3, e_1, e_2 >$, both the code for an insertion on the left and on the right are executed. Figure 1 shows the states of the container in the heap.

In respect of different permutations of the same sequence of the same type of operations, a container might behave differently if in its source code it relies on the *Natural Order* of the elements. In Java, for example, it is easy to know it, because the natural order is defined by the method `compareTo`. Therefore, it is important to exploit this information if we want to improve the performance of the optimisation algorithms. If we have a good sequence, a local search algorithm should look at its neighbourhood to find a better solution. However, without the novel search operators, the neighbourhood would be influenced by the actual values of the elements. For simplicity, just consider the case of an insertion of a FC with in input only one parameter. For example, consider the sequence S_x:

```
container.insert(e1);  // 1
container.remove(e2);  // 2
container.insert(e3);  // 3
```

here we cannot insert a FC between line 1 and 2 such that the new element e_k satisfies $e_1 < e_k < e_2$. On the other hand, consider the following S_y:

```
container.insert(e2);  // 1
container.remove(e4);  // 2
container.insert(e6);  // 3
```

in that case we can insert a FC with e_3. The two sequences may seem different because they have different values of the elements, but actually they are exactly equivalent. In fact, the relative order among the elements is the same, so also the behaviour of container is the same. Hence, the local search should consider the relative order of the elements and not their actual values. If we focus on the relative order, we can analyse the shape of the neighbourhood defined by an insertion of a FC (for simplicity with exactly one element in input) on S_i. We want that every possible sequence that holds the same relative order among the former elements of S_i can be reached with just one insertion. There can be up to $len(S_i)$ different elements in S_i. Just call this number D, with $D \in [0, len(S_i)]$. A new element can have the same value of the elements in S_i, or a new one that should be able to be in any position in the natural order relatively to the other elements. I.e., it can be the lowest, the highest or with a value between any two consecutive elements. Therefore, we have $W = D + 1 + 1 + (D - 1) = 2 \cdot D + 1$ different positions for the new element, with $W \leq 2 \cdot len(S_i) + 1$. In the above examples, we have $D = 3$ and $W = 7$, with the neighbourhood of S_y that includes all the $W = 7$ elements for the new FC. On the other hand, for S_x there are only 4 elements available: 3 with same values of the elements in S_x and 1 with a higher value than them.

It can be argued that S_y is better just because it has a wider neighbourhood. Although at a first look it can seem true, it is important to remember that the element values are generated at random. This happens because their search space is too large to be explored in a single step of a local search (remember that a FC can be inserted in $len(S_i) + 1$ different positions and that there are several methods under test). Therefore, the neighbourhood sizes of S_x and S_y are exactly the same. The difference lies on the probabilistic distribution of the neighbours that are reached. For simplicity, consider that the new element e_{new} can have a random value between e_1 and e_7. For S_y, all the $W = 7$ positions are possible, with each of them that has probability $\frac{1}{7}$. Although for S_x the positions relative to the already stored elements can be covered with the same probability, it has a probability $\frac{4}{7}$ of getting the highest position and zero of getting the others.

The difference between *exploration* and *exploitation* of a search space has been widely studied in the literature (e.g., [5]). However, here the situation is particular. In fact, the neighbourhood defined by the novel search operators causes the random choice of neighbours to occur in a wider area, although the averaged *distance* of all these neighbours does not change. We think this is the main point that explains the reason why the novel system may give better results. Therefore, we find that it is easier to explain this concept by adapting this situation to a simpler problem like *OneMax*. Consider the case in which there are two different neighbourhoods: N_1 that flips one bit, and N_2 that flips two bits. In a local search, only k neighbours will be evaluated in these neighbourhoods, with $k < \frac{|N_1|}{2} \wedge k < \frac{|N_2|}{2}$. The sizes of the neighbourhoods will be the same. It is the size of the area from which they pick neighbours that is different. The *Euclidean Distance* is employed. It is easy to understand that the averaged distance of the neighbours in N_2 is higher than the one of N_1. Hence, N_2 gives more emphasis on the *exploration*, instead N_1 gives more emphasis on the *exploitation*. Now consider a neighbourhood N_3 in which, like in N_1, only one bit is flipped, but the k neighbours are always chosen only in the first half of the bit string. It is obvious that a local search that employs N_3 will have poor results. However, the averaged distance of the neighbours in N_3 is equal to the one of N_1. In the search landscape, they both look within the same distance, but N_1 looks in more directions. The basic search operators described in section II-C are like N_3, and the novel ones are like N_1. However, this does not prove that the novel search operators are better. In fact, even if a search strategy looks in fewer directions, the search landscape can have a particular form that permits the optimisation algorithm to reach the same local optima. Besides, it can happen that on average these fewer directions are "better" (that depends of course on the landscape).

The novel search operators can deeply change the representation of a solution in only one step. For example, a removal of a FC with the lowest element may change all the elements along the entire sequence. Anyway, the neighbours that these operators define are "close" to the considered solution. Hence, euclidean or *Hamming* distances are not very significant for this problem, because the relative order among the elements is more important than their actual values. Besides, sub-sequences of consecutive read-only FCs can be randomly sorted without changing at all the behaviour of the sequence. Anyway, a formal definition of a distance for the sequences S_i is beyond the scope of this paper.

In some types of containers, although the relative order of the elements is not important, the behaviour may depend on the equality of a newly inserted element E with an already stored one. In that case, with the new search operators, and in particular by a wise choice of the parameter ρ, the search can be guided to look in the most promising area of the search space. On the other hand, with the simple representation, the probability that E is equal to one of the stored in the container depends on their current values. For example, if E can have a value between 1 and 30, and in the container there are 3 different elements, the probability that E is equal to one of them is $\frac{1}{10}$.

III. Dynamic Search Space Reduction

In some cases, it is convenient to avoid to insert a FC w in S_i that does not change the state of the container. In fact, if all the branches of w and the ones of all of the functions that are called through it (e.g., w calls $f1$, and $f1$ calls $f2$) are already covered, there can be no improvement at all in the quality of S_i. The opposite behaviour will occur and the sequence S_i will get worse because it becomes longer.

For a global search algorithm, it may happen that the insertion of w will later permit the removal of more than one FC without decreasing the coverage. In such a case, the sequence will be better because it would be shorter. However, with local search, we know a priori that the insertion of w will lead to a worse sequence that will be immediately discarded. Therefore, functions like w should be ignored in a local search. In containers like `TreeMap`, the number of read-only functions can be high (14 on a total of 18 in its implementation in the Java API 1.4). If we remove the functions w from the search, the number of evaluations that a local search does can decrease significantly without any loss in the quality of the results. However, there are some problems that need to be solved when we are considering a function t:

- Determining if t is read-only is not straightforward. It requires a syntactic analysis of the source code. Besides, the polymorphism and the the use of the external component, which code is unknown, makes the task extremely difficult[1]. However, there are two cases in which at run-time we can determine for sure if a function is not read-only: if we remove a FC call and the coverage increases or if we insert a FC and the coverage decreases.
  In the case of containers, we can also do an analysis of the name of the function to determine if it is read-only. A database for common function names (e.g., `insert,add,push`) can be used with string matching algorithms. If such a comparison is not able to classify a function, a normal syntactic analysis can still be used.
- The function t can be the only one that calls a particular private function $t1$. Therefore, before removing t from the local search, not only all the branches in t need to be covered, but also all the ones belonging to functions called through t. Determining which those functions are

[1] in this paper we are not considering the idea of an analysis at the byte-code level or the use of reverse engineering.

is not always trivial. Consider the example when t calls a function s in the superclass, and s is the only one that calls the private function u in the CuT. If the code of the superclass is not available, a syntactic analysis of the CuT cannot discover that u can be called only if t is called. A dynamic analysis can show if a call to t leads to the execution of u. However, if u is never called, a dynamic analysis cannot prove that it does not depend on t. For example, it can happen that u is called by s only with a very low probability.
If t has some branches that are infeasible, they will never be marked as covered. Therefore, t will not be removed from the search even if all of its feasible branches are covered, unless the infeasible ones can be recognised. Unfortunately, determining if a branch is infeasible is usually a hard problem.

- These analyses have a computational cost. It is important to guarantee that such overhead is less than the benefits gained by the local search algorithm.

Consider the following predicates:

$R(t)$	the considered function t is read-only. I.e., it does not change the state of the container, and all the functions it calls are read-only as well.
$C(t)$	all the branches in t and all the ones in the functions called by t are covered.

The function t will be removed from the search if the predicate $E(t)$ is true, with:

$$E(t) = R(t) \land C(t)$$

As stated before, it is not always possible to get a precise value for the predicates $R(t)$ and $C(t)$. Therefore, we need to investigate when $E(t)$ has a wrong value:

- we have a *false negative* if the predicate $E(t)$ is estimated as false when it should be true. The function t will not be removed from the search. It happens if at least one of the predicates $R(t)$ or $C(t)$ are wrong. The optimisation will have no positive effects, and the only degradation of the performance will be the overhead caused by the calculation of $E(t)$.
- we have a *false positive* if the predicate $E(t)$ is estimated as true when it should be false. The function t will be removed from the search. It happens if $R(t)$ or $C(t)$ are wrongly evaluated. If t does change the state and it is not completely covered yet, both $R(t)$ and $C(t)$ need to be misclassified to make $E(t)$ wrong. If the classifiers used for $R(t)$ and $C(t)$ are well designed, the probability that both of them misclassify at the same time will be relatively small. However, even if t is removed from the search, it can be later reintroduced if during the search a removal of a FC of t increases the coverage, because that means for sure that t is not read-only. The same can happen if the dynamic analysis of the function calls graph (i.e., the graph of the functions that can be called through t) let the discovering of a new function linked to t (consider the previous example regarding the calls to a superclass function). Anyway, a misclassification of $E(t)$ could generate no problems at all. In fact, there can be cases in which even if t can change the state of the container, these changes could have no influence on the covering of the branches of the other functions. Besides, although it could happen that some branches of the private functions called through t are not covered, those functions could also be called by other functions under test.

TABLE I
CLASSIC DISTANCE-FUNCTION FOR CONJUNCTIONS AND DISJUNCTIONS

Element	Distance
$a \lor b$	$min(dist(a), dist(b))$
$a \land b$	$dist(a) + dist(b)$

IV. BRANCH DISTANCE

The *Branch Distance* is used to guide the optimisation algorithms. A way in which it can be defined can be found in [6]. However, in OO software there can be problems when conjunctions and disjunctions are handled (table I shows their distance functions). Consider the example in which we need to evaluate the branch distance of `if(x!=null && x.foo())`. If the first clause `x!=null` is false, the evaluation of the second will throw an exception. Therefore, we need to handle them in a special way. The problems arise when in an expression like $a\ op\ b$ the predicate b is not computed. This happens when the expression can be evaluated only by considering the first predicate a. In Java, we have this in the case $a\&\&b$ when a is false and when a is true in $a||b$.

If the first clause is true, a disjunction is true (i.e., its distance is 0) and we do not have to evaluate the distance function of the second clause.

The case of the conjunction (e.g., $a \land b$) is more complex. The formula in table I will be used and, if an exception is thrown, it will be caught and the following distance function will be used instead for that branch for the rest of the search:

$$dist(a \land b) = \begin{cases} K \cdot \frac{dist(b)}{1+dist(b)} & \text{if } a \text{ is true },\\ \\ K + dist(a) & \text{otherwise }, \end{cases} \tag{1}$$

with K that can be any arbitrary constant value. In the case in which a is false, b is not used to compute the branch distance. However, to get the best results, all the available information on the predicates should be exploited [7]. Therefore, when a is false, the formula in table I is better because it gives more information. This is why we still use that formula, and only when b throws an exception we use eq.1. When a is true, its contribution on the branch distance is 0. Note that, regardless of b, this new branch distance gives always higher values when a is false than when is true.

V. USED ALGORITHMS

In the experiments, three different optimisation algorithms have been applied: Hill Climbing (HC) with random restarts, a Genetic Algorithm (GA) [8] and a Memetic Algorithm (MA) [9]. In the following, a very brief description of these algorithms is given.

HC is a local search algorithm. It starts from a random point in the search space and looks at the solutions that are "close" to it. If one of its neighbours has a better fitness, the search moves on that solution and starts to look in its neighbourhood for a better one. The algorithms will always move in that way on better solutions until there are no better solutions in the current neighbourhood. In such a case, the algorithm is said to be stuck in a local optimum.

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 consider 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.

MAs 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.

Although a HC and a GA have been already employed in our previous work [4], this paper makes significant contributions by introducing a novel representation and search operators for these algorithms.

The quality of a test sequence S_i is defined by:

$$f(S_i) = coverage(S_i) + \frac{1}{1 + length(S_i)} \ . \tag{2}$$

However, during the search, we need to exploit the information given by the branch distances:

$$\begin{aligned} f(S_i) = & coverage(S_i) + \alpha(1 - B(S_i)) \\ & + (1-\alpha)\frac{1}{1 + length(S_i)} \ , \end{aligned} \tag{3}$$

where $B(S_i)$ is the normalised distance of all the branches [4], and $\alpha \in [0, 1]$. Eq.3 is used only during the search. For evaluating the result of a search algorithm, we need to use eq.2, because at the end of the search only the actual coverage and length are important.

The problem can be described as a Multi-Objective problem. There is one objective to maximise, the coverage, and two to minimise: the branch distance and the length. However, their relations are particular. The coverage is always more important than the length. The optimum for the branch distance (i.e., $B(S_i) = 0$) implies the optimum for the coverage. Although an increment in the coverage always implies a decrease in the branch distance, the opposite is not always true. Besides, the branch distance objective has no importance at the end of the search.

The fitness in eq.3 can be deceptive. The search can be fooled by looking at sequences that get always longer to try to reach a particular branch. That happens when an increase in the length lets the branch distance decrease without covering the branch. Furthermore, a particular small improvement in a branch distance, that will lead to cover the branch, may be discarded because the length increases. There are different ways to address this problem. In the following we describe a useful way that can be applied to a local search. It consists of two phases. In the first, a new neighbour S_n will be judged as better than the current solution S_i if:

$$\begin{cases} g(S_n) > g(S_i) & \text{or} \\ g(S_n) = g(S_i) \ \wedge \ length(S_n) < length(S_i) & , \end{cases} \tag{4}$$

where

$$g(S_j) = coverage(S_j) + (1 - B(S_j)) \ . \tag{5}$$

When the local search is stuck in a local optimum, the second phase of the search starts. The search will continue from this local optimum using eq.2 as the fitness function for the comparisons. The reason is that in the first phase an improvement of the branch distance is accepted regardless of the length. That can lead to longer sequences without improving the coverage. The second phase is employed to reduce the length of the sequence. Anyway, it is important to notice that it cannot decrease the coverage.

For exploring the neighbourhood of a S_i, the HC uses the search operators defined in section II. The GA uses the same operators for mutating a solution. However, the crossover operator can generate "holes" in the order of the stored elements in the container. These holes will need to be filled by decreasing the values of the elements accordingly. The MA combines the described GA with the HC. All the algorithms use the search space reduction described in section III, unless otherwise stated.

VI. Case Study

The following containers have been used for the testing: `Stack`, `Vector`, `LinkedList`, `Hashtable` and `TreeMap`. They are all taken from the Java API 1.4. Only in `TreeMap` the relative order of elements is important. Table II summarises their characteristics.

The different algorithms described in this paper have been tested on the given cluster of containers. 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.

Table III summarises the performances of the algorithms. The best coverage is the highest coverage ever reached during

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

TABLE II

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	Statistics	Coverage			Length		
		HC	GA	MA	HC	GA	MA
Stack	Mean	10.0000	10.0000	10.0000	6.0000	6.0000	6.0000
	Variance	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	Median	10.0000	10.0000	10.0000	6.0000	6.0000	6.0000
	Best	10	10	10	6	6	6
Vector	Mean	100.0000	99.9530	100.0000	45.3634	47.1121	44.8398
	Variance	0.0000	0.0449	0.0000	0.4400	2.3842	0.4553
	Median	100.0000	100.0000	100.0000	45.0000	47.0000	45.0000
	Best	100	100	100	44	44	44
LinkedList	Mean	83.9240	83.8300	83.9950	35.0870	37.2940	33.3480
	Variance	0.0703	0.2934	0.0050	1.6731	4.4440	0.6015
	Median	84.0000	84.0000	84.0000	35.0000	37.0000	33.0000
	Best	84	84	84	32	33	32
Hashtable	Mean	100.3460	103.1120	98.7430	31.4020	33.7610	30.2380
	Variance	1.6479	7.3488	9.2442	2.3527	5.0289	1.8712
	Median	100.0000	105.0000	100.0000	31.0000	34.0000	30.0000
	Best	106	106	106	36	36	36
TreeMap	Mean	187.6560	184.9740	188.8330	45.0070	41.8820	46.3300
	Variance	0.7084	3.5689	0.7098	5.9109	7.0191	5.8149
	Median	188.0000	185.0000	189.0000	45.0000	42.0000	46.0000
	Best	191	190	191	50	51	46

TABLE III

Comparison of the different optimisation algorithms on the container cluster. Each algorithm has been stopped after evaluating up to $100,000$ solutions. The reported values are calculated on 1000 runs of the test.

the test. The best length is the shortest length for the highest coverage. Note that it would have been better to report the statistics for the length of each different reached coverage, because it is common that for higher coverages you may need longer sequences. Hence, if you have an algorithm that gives shorter sequences, that does not mean that it optimises the length objective in a better way: you also need to control if the coverage is not worse. However, for space limitations we show these statistics without considering the different coverage values.

All of the algorithms use the fitness function defined in eq.3, with $\alpha = 0.5$. The parameter ρ of the novel search operators is set to 0.8. The HC uses a random restart when it reaches a local optimum. The GA uses single point crossover with probability 0.2. Because the parents can have different lengths, the new offspring's length will be the average of the parents's length. A single point crossover takes the genes from the first parent in the usual way, and the following others from the tail of the second parent. The mutation probability of an individual is 0.9. The population size is 64. Rank selection is used with a bias of 1.5. 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 up to $100,000$ sequences. From table III, the MA results the best among the algorithms. Mann Whitney U tests have been carried out with a 0.05 level of significance to support this affirmation. However, the performances of MA on the `Hashtable` are quite poor. That happens because the fitness in eq.3 is deceptive. The results in table IV have been obtained using a MA with eq.4. With such a configuration, the MA has no problems in covering the branches of the `Hashtable`.

Table IV shows the performance of the novel search operators on the MA. Although they do no give the best results on every container, on average they increase the performance of the algorithm.

Table V shows the benefits of using the novel dynamic search space reduction (DSSR). The average number of steps that HC needs to reach a local optimum are reported, both with and without the DSSR. It is important to outline that the quality of the local optima is not influenced by the DSSR.

Container	Statistics	Coverage		Length	
		Base	Novel	Base	Novel
Vector	Mean	100.0000	100.0000	45.8859	45.1191
	Variance	0.0000	0.0000	0.8908	0.5660
	Median	100.0000	100.0000	46.0000	45.0000
	Best	100	100	44	44
LinkedList	Mean	84.0000	84.0000	34.6757	33.9259
	Variance	0.0000	0.0000	1.3837	1.2831
	Median	84.0000	84.0000	35.0000	34.0000
	Best	84	84	32	32
Hashtable	Mean	106.0000	106.0000	35.0000	36.0000
	Variance	0.0000	0.0000	0.0000	0.0000
	Median	106.0000	106.0000	35.0000	36.0000
	Best	106	106	35	36
TreeMap	Mean	189.1461	188.9800	49.7768	46.6947
	Variance	0.4776	0.6148	7.2878	6.1522
	Median	189.0000	189.0000	49.0000	46.0000
	Best	191	191	48	44

TABLE IV

The performances of the MA using the formula 4 are reported. The use of the novel neighbourhood is compared to the base one.

Container	DSSR	Steps
Stack	no	103.4179
	yes	85.1809
Vector	no	7319.2860
	yes	4161.0169
LinkedList	no	3718.3870
	yes	2762.9300
Hashtable	no	3143.8719
	yes	1603.6309
TreeMap	no	4946.2600
	yes	3118.2929

TABLE V

The performances of the Dynamic Search Space Reduction (DSSR) are evaluated using the averaged numbers of steps that a HC needs to arrive to a local optimum. The DSSR does not influence the quality of the local optima. The reported values are calculated on 1000 runs of the test.

VII. Related Work

This paper extends our previous work [4] on Test Data Generation for container classes. No other work specifically on containers that uses optimisation algorithms is known to us. That paper explains how to employ four different optimisation algorithms on the problem: Random Search, Hill Climbing, Simulated Annealing and Genetic Algorithms. Their performances are then compared. Besides, a search space reduction that is specific to containers and a novel testability transformation are presented.

Visser et al. [10] used exhaustive techniques with symbolic execution. To avoid the generation of redundant sequences, they used *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.

A container class is an OO software. Therefore, any system that claims to be able to generate test data for OO software should work also on containers. Tonella [11] was the first to develop a system that uses optimisation algorithms. It used a GA with special operators to generate unit tests for Java classes.

A comparison between HC, GAs and MAs was done by Wang and Jeng [12], but they did it on procedural functions. However, their results lead to the same conclusion obtained in our experiments that MAs seem to perform better than HC and GAs, with HC outperforming GAs.

VIII. Conclusion and Future Work

This paper has proposed novel techniques for testing container classes. Besides presenting novel search operators, it presents a comparison of three different optimisation algorithms: Hill Climbing (HC), Genetic Algorithms (GAs) and Memetic Algorithms (MAs). Often, in literature, when a new technique is designed, it is tested only against Random Search (RS). Although RS can give good coverage, it is inappropriate in the context of OO software, because it miserably fails to generate any good sequence with reasonable length.

Our empirical tests show that our MA outperforms the other algorithms. Furthermore, the novel search operators increase its performance. In the software engineering community there is a strong bias toward GAs. Local search algorithms are often ignored, because the search landscape is considered to be too complex and discontinuous. Although in our previous work [4] we already showed that HC outperforms GAs (at least the ones we employed) on testing container classes, the fact that our MA gives the best results leads us to study the landscape of this problem in the future.

A novel search space reduction has been described. It decreases the number of steps that a local search needs to reach a global optimum without altering its quality. However, if the number of read-only functions under test is low, it may be better not to employ this technique, because the over-head can be too big in comparison with the gained benefits. In our case study, this novel technique is arrived up to double the speed of the local search.

A new branch distance has been presented that can be applied to any type of OO software, i.e., it is not specific for containers. It was compulsory to develop it, because in some cases the common branch distance may throw exceptions when it is employed for OO software. We wonder why such a problem has not been addressed before. The only answer we can give is that all the little previous work on OO software was always tested on a very limited number of classes. Hence, we suppose that the case in which exceptions may be thrown was not present in any of those tested classes. This shows one of the problems of the current state-of-art of search based software testing (at least for OO software): the test clusters are too small. This is due to the lack of a public and complete *instrumentator*.

Our future work will focus on studying the landscapes of different typologies of OO software. Exploiting local search seems to give better results than those of traditional GAs. Therefore, we need to analyse the landscape shapes with the aim of developing better local search operators and MAs. That will also help to understand why our MA performed well on most of the test problems used in this paper.

IX. Acknowledgements

This work is supported by an EPSRC grant (EP/D052785/1). The authors wish to thank Ramón Sagarna and Per Kristian Lehre for the useful discussions.

References

[1] 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.

[2] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[3] 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*, vol. 150, no. 3, pp. 161–175, 2003.

[4] A. Arcuri and X. Yao, "Search based testing of containers for object-oriented software," University of Birmingham, Tech. Rep. CSR-07-3, 2007.

[5] X. Yao, "Simulated annealing with extended neighbourhood," *International Journal of Computer Mathematics*, vol. 40, pp. 169–189, 1991.

[6] 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)*, 1998, pp. 285–288.

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

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

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

[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)*, 2006, pp. 37–48.

[11] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[12] H.-C. Wang and B. Jeng, "Structural testing using memetic algorithm," in *Proceedings of the Second Taiwan Conference on Software Engineering*, 2006.