Variations over Test Suite Reduction

Dennis Güttinger[1], Vitaly Kozyura[2], Dominik Kremer[3], and Sebastian Wieczorek[2]

[1] Goethe Universität, Frankfurt am Main, Germany
guetting@stud.uni-frankfurt.de
[2] SAP AG, Darmstadt, Germany
{v.kozyura,sebastian.wieczorek}@sap.com
[3] Technische Universität Darmstadt, Germany
kremer@mathematik.tu-darmstadt.de

Abstract. This paper deals with the problem of effective test suite reduction. In its original form this problem is equivalent to the set covering problem, which has already been extensively studied and many strategies such as greedy or branch and bound for computation of an approximative optimal solution to this NP-complete problem are known. All of these algorithms only focus on one objective which is the minimization of the number of action calls within the test suite reduction. However, practical experience shows that balancing out the distribution of action calls is another objective which should be considered when choosing an efficient test suite. We will therefore introduce and evaluate different extensions of the standard techniques which incorporate action call distribution. We will see that these adjusted strategies can compute a reduced test suite with a smoother distribution over function calls within an acceptable amount of additional time in comparison to the classic algorithms.

1 Introduction

Automatically generating tests suites from formal specifications as advertised by Model-based Testing (MBT) is regarded as a potential innovation leap in industrial software quality assurance. Most MBT approaches are running in two phases. In the first phase vast amount of test cases are generated for an inserted model until a coverage of model entities is achieved. In the second phase a subset of these test cases is selected with the aim to preserve the targeted coverage and therefore the assumed fault-uncovering capabilities [11]. This activity is called test suite reduction.

The problem of test suite reduction is largely discussed in the literature. There are papers, where the general test suite reduction activity is described [3,9]. Further work on how to apply 0/1-Integer linear programming to the test suite reduction problem [12] or how to improve the Greedy heuristics [5,6,2] can be found. In [1,14] there are approaches using multi-objective optimization functions, whereas in [8] an approach based on genetic algorithms is introduced. Some empirical results for test suite reductions have been reported in [11].

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 149–163, 2013.

In this paper we propose a test suite reduction approach that aims for a smoother test case distribution, as required by our industrial MBT users. The mathematical definition of the problem as well as proposed algorithmic modifications are the main contributions of this paper. Further we present experimental results that demonstrate the applicability and efficiency of the proposed approaches.

The paper is structured as follows. In Section 2 we briefly introduce the standard algorithms for test suite reduction. Section 3 gives a definition of test case distribution, describes its practical relevance and details how we incorporated it in the standard reduction algorithms. In Section 4 we illustrate the impact of our algorithmic modifications on concrete industrial cases and finally discuss our conclusions in Section 5.

2 Test Suite Reduction

In order to describe the test suite reduction problem, we assume a finite set of coverage requirements $\mathcal{R} = \{r_1, \ldots, r_n\}$ which is guiding the test generation. All requirements have to be met by a complete test suite. Each test case tc either satisfies a given requirement ($r_i(\text{tc})$) or does not, that is
$r_i(\text{tc}) \in \{true, false\}$. For convenience we also define

$$\text{cov}(\text{tc}) = \{r_i : r_i(\text{tc}) = \text{True}, 1 \leq i \leq n\}. \tag{1}$$

Now a test suite $\text{TS} = \{\text{tc}_1, \ldots, \text{tc}_m\}$ is *complete*, if $\mathcal{R} = \bigcup_{i=1}^{m} \text{cov}(\text{tc}_i)$ and the test suite reduction problem can be reformulated as follows:

Given: A test suite TS and a set of requirements $\mathcal{R}$, such that TS is complete with respect to these requirements.

Problem: Find a complete test suite $\text{TS}_0 \subseteq \text{TS}$ that is minimal with respect to

$$\text{value}\,(\text{TS}_0) = \sum_{\text{tc} \in \text{TS}_0} |\text{tc}|. \tag{2}$$

In the remainder of this section we present two classical approaches, the *Greedy-* and the *Branch and Bound*-algorithm, to solve this problem. Later we will describe how these algorithms can be modified in order to obtain a better test case distribution.

2.1 Greedy Algorithm

Even though the Greedy algorithm computes an approximation, [4] showed that the result cannot become arbitrarily bad. In fact the upper bound for the error only depends on the number of requirements.

The algorithm stores two objects: An iteratively constructed subset TS_0 of TS, which will be a complete test suite after termination of the algorithm and a set $\mathcal{R}_0$ of all those requirements that are already met by this subset (l. 1–2). While not all requirements are met (l. 3), the algorithm does the following:

It computes the set of all test cases tc, for which the ratio w_{tc} of the number of additionally satisfied requirements and the number of additional action calls is maximal (l. 4–6). Then it picks one of these at random (l. 7). This test case is afterwards added to TS_0 and $\mathcal{R}_0$ is updated appropriately (l. 8–9).

Input: test suite $\text{TS} = \{\text{tc}_1, \ldots, \text{tc}_m\}$, set of requirements $\mathcal{R}$
Output: approximated minimal test suite $\text{TS}_0 \subseteq \text{TS}$ which is complete
1: $\text{TS}_0 = \emptyset$
2: $\mathcal{R}_0 = \emptyset$
3: **while** $|\mathcal{R}_0| < |\mathcal{R}|$ **do**
4: **for** $\text{tc} \in \text{TS} \setminus \text{TS}_0$ **do**
5: $w_{\text{tc}} = \frac{1}{|\text{tc}|} \cdot |\text{cov}(\text{tc}) \setminus \mathcal{R}_0|$
6: $\text{TS}_0' = \{\text{tc} \in \text{TS} \setminus \text{TS}_0 : w_{\text{tc}} \text{ is maximal}\}$
7: Pick $\text{tc} \in \text{TS}_0'$
8: $\text{TS}_0 = \text{TS}_0 \cup \{\text{tc}\}$
9: $\mathcal{R}_0 = \mathcal{R}_0 \cup \text{cov}(\text{tc})$
10: **return** TS_0

Algorithm 1. Greedy

2.2 Branch and Bound – Algorithm

We use the Branch and Bound variation (Balas-algorithm) described in [7] to compute an optimal result.

The algorithm identifies all possible subsets of $\text{TS} = \{\text{tc}_1, \ldots, \text{tc}_m\}$ with arrays $(n_1, \ldots, n_m)$. Here $n_i = 1$ means, that tc_i is part of the subset, while $n_i = 0$ means, that it is not. To check these arrays systematically, they are organized as a binary tree. At the root node no decisions have been made, as any node on level i represents a certain choice of the first i bits. For simplicity it is also denoted as array $(n_1, \ldots, n_i)$ and identified with the test suite $\{\text{tc}_j : n_j = 1\}$. We denote the level of a node by $\text{level}(n_1, \ldots, n_i) = i$. Now (2) can be extended to nodes by

$$\text{value}(n_1, \ldots, n_i) = \sum_{j=1}^{i} |\text{tc}_j| \cdot n_j. \tag{3}$$

The Branch and Bound algorithm stores two objects: The best solution found so far, n_{res}, and a stack S of nodes that has to be checked. Obviously n_{res} is initialized with the array that represents whole TS and S with the stack that only contains the root node (l. 1–2). As long as additional nodes have to be checked, one of them is popped from S (l. 3–4). Then child nodes n_0 and n_1 are generated, where n_0 rejects the next test case and n_1 includes it. To decide, whether it is necessary to check these as well, the following rules are applied:

- n_0 is expanded if it has a successor which represents a complete test suite (l. 7) and if appending the smallest remaining test case results in a suite that is smaller than the one represented by n_{res} (l. 8). (To be able to efficiently evaluate the second condition, we presume TS to be sorted ascending by length.)

– n_1 is expanded if it is smaller than n_{res} (l. 11), but not if it is complete (l. 12, 15).

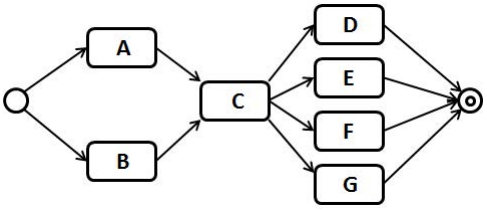
Finally n_{res} is updated, if n_1 is smaller than n_{res} and complete. In this case all nodes bigger than n_1 are removed from S afterwards (l. 11–14).

Input: test suite TS $= \{\text{tc}_1, \ldots, \text{tc}_m\}$ sorted ascending by length, redundant test cases eliminated
Output: exact minimal test suite $\text{TS}_0 \subseteq \text{TS}$
1: $n_{\text{res}} = (1, \ldots, 1)$
2: Stack S $= \{()\}$
3: **while** S $\neq \emptyset$ **do**
4: $n =$ S.pop
5: **if** level$(n) < m - 1$ **then**
6: $n_0 = (n, 0)$
7: **if** $\{\text{tc}_i : (n_0)_i = 1$ **or** $i >$ level$(n_0)\}$ is complete test suite **then**
8: **if** value$(n_0) + |\text{tc}_{i+1}| <$ value(n_{res}) **then**
9: S.push(n_0)
10: $n_1 = (n, 1)$
11: **if** value$(n_1) <$ value(n_{res}) **then**
12: **if** $\{\text{tc}_i : (n_1)_i = 1\}$ is complete test suite **then**
13: $n_{\text{res}} = n_1$
14: S $= \{n' \in S :$ value$(n') <$ value$(n_1)\}$
15: **else if** level$(n_1) < m$ **then**
16: S.push(n_1)
17: **return** $\{\text{tc}_i : (n_{\text{res}})_i = 1\}$

Algorithm 2. Branch and Bound

3 Test Case Distribution

In this section we formulate the test case distribution problem and describe it mathematically by introducing a sequence of functions that measures distribution quality in terms of variances. Using these functions we show how to modify both Greedy and Branch and Bound algorithms in order to improve test case distribution.

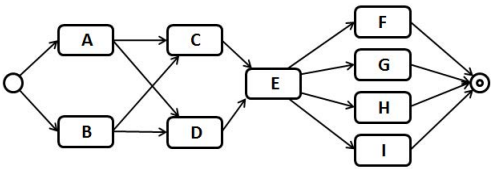


$\text{TS}_1 = \{$(A, C, D), (A, C, E), (**A**, C, F), (B, C, G)$\}$

$\text{TS}_2 = \{$(A, C, D), (A, C, E), (**B**, C, F), (B, C, G)$\}$

Fig. 1. Test Model Example 1

Let us start with the simple example depicted in Figure 1 together with two test suites TS_1 and TS_2. Both test suites are minimal with respect to action coverage and vary only in one action, which is denoted in bold font. Actions A and B are used an equal number of times in TS_2, while they are not in TS_1.



TS_3 = {(A, C, E, F), (A, **C**, E, G), (B, D, E, H), (B, **D**, E, I)}

TS_4 = {(A, C, E, F), (A, **D**, E, G), (B, D, E, H), (B, **C**, E, I)}

Fig. 2. Test Model Example 2

Let us now move on to the more complex example depicted in Figure 2, which again comes with two test suites TS_3 and TS_4. As before, both test suites are minimal and their difference is marked in bold font. However this time all actions are called the same number of times in TS_3 as in TS_4. Nevertheless the second suite can be regarded as having a smoother test case distribution since it includes four different variations for the first two actions, while the first one only includes two. In a similar fashion one could obtain larger examples, where distribution quality only depends on the number of occurrences of even larger test case subsequences.

3.1 Industrial Relevance

In the last few years, development paradigms like lean and agile, propagating the empowerment of developers [10], have found broad adoption in industrial IT organizations. While previously development processes were defined on a global scale and based on an assumed overall efficiency, todays industrial developers have a much bigger degree of freedom in choosing their development approaches and tools. Based on our experience, on individual level of decision making efficiency is often traded for lower learning effort, better user experience and other partially subjective reasons.

Concretely, in a case study [13] that aimed to validate an MBT framework with industrial testers we observed that uneven test case distribution may cause negative assessments on individual level. In interview sessions, most of the case study participants stated that the distribution resulting from standard test suite reduction made them less confident in the effectiveness of the test suite and the correctness of the test generation approach. While no concrete evidence for superior qualities of evenly distributed test suites were found in our experiments, they assumed that a better distribution may be beneficial in the test data assignment, test maintenance and selection of test cases for regression. As a consequence, most participants required smoothly distributed test suites in order to apply MBT in their test routines.

In summary, from our industrial experience a smooth test suite is more desirable as it increases confidence of MBT users. Whether it also has direct effects on the quality of the test suite remains subject of future investigations.

3.2 Formalization

We now want to give a mathematical description of the problem. Therefore we propose a way how to measure the distribution quality of a given test suite $\mathrm{TS}_0 \subseteq \mathrm{TS}$.

Given that a test case is composed of a sequence of action calls, a simple approach is to evaluate their variance in a test suite. For any test case $\mathrm{tc} = (a_1, \ldots, a_k) \in \mathrm{TS}$ we define a counting function $d^{(1)}_{\mathrm{tc}}$, which assigns to each action $a \in \mathcal{A}$ the number of times it is called by tc:

$$d^{(1)}_{\mathrm{tc}} : \mathcal{A} \to \mathbb{N}, \quad a \mapsto \left|\left\{i \in \{1, \ldots, k\} : a_i = a\right\}\right|. \tag{4}$$

Equivalently for any test suite $\mathrm{TS}_0 = \{\mathrm{tc}_1, \ldots, \mathrm{tc}_m\} \subseteq \mathrm{TS}$ we define a counting function $d^{(1)}_{\mathrm{TS}_0}$, which assigns to each action $a \in \mathcal{A}$ the number of times it is called by whole TS_0. This can be formalized as

$$d^{(1)}_{\mathrm{TS}_0} : \mathcal{A} \to \mathbb{N}, \quad a \mapsto \sum_{i=1}^{m} d^{(1)}_{\mathrm{tc}_i}(a). \tag{5}$$

Now the mean number of calls per action in TS_0 is given as

$$\overline{d^{(1)}_{\mathrm{TS}_0}} = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} d^{(1)}_{\mathrm{TS}_0}(a) \quad \left(= \frac{1}{|\mathcal{A}|} \sum_{i=1}^{m} |\mathrm{tc}_i| \right). \tag{6}$$

If TS_0 is well distributed, we would expect $d^{(1)}_{\mathrm{TS}_0}(a)$ not to vary much, but to stay near its mean value for all $a \in \mathcal{A}$. Hence, the *variance* of $d^{(1)}_{\mathrm{TS}_0}$ or the *variance of action calls in* TS_0 will be our first measure for distribution quality:

$$\mathrm{Var}_1(\mathrm{TS}_0) = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \left(d^{(1)}_{\mathrm{TS}_0}(a) - \overline{d^{(1)}_{\mathrm{TS}_0}} \right)^2. \tag{7}$$

Let us apply this definition to the examples from above. The values obtained for $d^{(1)}_{\mathrm{TS}_i}$ are denoted in table 1. Further calculations show, that for the first example $\mathrm{Var}_1(\mathrm{TS}_1) = 1.35$, while $\mathrm{Var}_1(\mathrm{TS}_2) = 1.06$. As expected TS_2 has a better distribution quality than TS_1. Nevertheless for the second example we get $\mathrm{Var}_1(\mathrm{TS}_3) = \mathrm{Var}_1(\mathrm{TS}_4) = 0.84$, i. e we cannot determine the difference between TS_3 and TS_4 by Var_1. This is not surprising, since the counting functions for both test suites are identical.

To circumvent this problem we generalize our ideas in order to construct a *variance of action-sequence calls* for sequences of a fixed length p: Let $\mathcal{A}^{(p)}$ be the set of all action-sequences of length p that are part of at least one test case in TS. To ease notation we will use the symbol $\mathbf{a}$ to denote such sequences.

Table 1. Values of $d^{(1)}_{\mathrm{TS}_i}$

Example 1							
Suite	A	B	C	D	E	F	G
TS_1	3	1	4	1	1	1	1
TS_2	2	2	4	1	1	1	1

Example 2									
Suite	A	B	C	D	E	F	G	H	I
TS_3	2	2	2	2	4	1	1	1	1
TS_4	2	2	2	2	4	1	1	1	1

First for any test case $\mathrm{tc} = (a_1, \ldots, a_k) \in \mathrm{TS}$ we again define a counting function $d^{(p)}_{\mathrm{tc}}$, which assigns to each action-sequence $\mathbf{a} \in \mathcal{A}^{(p)}$ the number of times it is called by tc:

$$d^{(p)}_{\mathrm{tc}} : \mathcal{A}^{(p)} \to \mathbb{N}, \quad \mathbf{a} \mapsto \left|\left\{i \in \{p, \ldots, k\} : (a_{i-p+1}, \ldots, a_i) = \mathbf{a}\right\}\right|. \tag{8}$$

As before we can use these functions to define a counting function $d^{(p)}_{\mathrm{TS}_0}$ for a whole test suite $\mathrm{TS}_0 = \{\mathrm{tc}_1, \ldots, \mathrm{tc}_m\} \subseteq \mathrm{TS}$, i. e.

$$d^{(p)}_{\mathrm{TS}_0} : \mathcal{A}^{(p)} \to \mathbb{N}, \quad \mathbf{a} \mapsto \sum_{i=1}^{m} d^{(p)}_{\mathrm{tc}_i}(\mathbf{a}). \tag{9}$$

The mean amount of calls per p-action-sequence is given by

$$\overline{d^{(p)}_{\mathrm{TS}_0}} = \frac{1}{|\mathcal{A}^{(p)}|} \sum_{\mathbf{a} \in \mathcal{A}^{(p)}} d^{(p)}_{\mathrm{TS}_0}(\mathbf{a}) \tag{10}$$

and we can finally define the *variance of* $d^{(p)}_{\mathrm{TS}_0}$ or the *variance of p-action-sequences* by

$$\mathrm{Var}_p(\mathrm{TS}_0) = \frac{1}{|\mathcal{A}^{(p)}|} \sum_{\mathbf{a} \in \mathcal{A}^{(p)}} \left(d^{(p)}_{\mathrm{TS}_0}(\mathbf{a}) - \overline{d^{(p)}_{\mathrm{TS}_0}}\right)^2. \tag{11}$$

Let us apply this definition with $p = 2$ to example 2 from above. The values obtained for $d^{(2)}_{\mathrm{TS}_i}$ are denoted in table 2. Further calculation shows, that $\mathrm{Var}_2(\mathrm{TS}_3) = 0.56$, while $\mathrm{Var}_2(\mathrm{TS}_4) = 0.16$. This means that Var_2 rates the second test suite better than the first one, as demanded by the motivation from the beginning of this section.

Table 2. Values of $d^{(2)}_{\mathrm{TS}_i}$

Example 2										
Suite	(A,C)	(A,D)	(B,C)	(B,D)	(C,E)	(D,E)	(E,F)	(E,G)	(E,H)	(E,I)
TS_3	2	0	0	2	2	2	1	1	1	1
TS_4	1	1	1	1	2	2	1	1	1	1

3.3 Lexicographical Approaches

Greedy. We present concrete implementations that respect distribution quality as described in section 3.2. The approaches proposed here optimize with respect to the total number of action calls first and with respect to distribution quality afterwards.

In order to modify the Greedy algorithm, we have to rate distribution quality not of a complete test suite, but of a partially constructed one. More precisely, we have to determine how well an additional test case $\mathrm{tc} = (a_1, \ldots, a_k)$ would fit in a non-complete test suite TS_0. This is done by the quality functions

$$q^{(p)}_{\mathrm{TS}_0}(\mathrm{tc}) = \sum_{i=p}^{k} d^{(p)}_{\mathrm{TS}_0}(a_{i-p+1}, \ldots, a_i). \tag{12}$$

Using them we can modify the Greedy algorithm in order to achieve better distribution quality by exchanging line 7 with the following steps:

for $i = 1 \to p$ **do**
  $\mathrm{TS}'_i = \left\{ \mathrm{tc} \in \mathrm{TS}'_{(i-1)} : q^{(i)}_{\mathrm{TS}_0}(\mathrm{tc}) \text{ is minimal} \right\}$
Pick $\mathrm{tc} \in \mathrm{TS}'_p$

Branch and Bound. To modify the Branch and Bound algorithm from section 2.2, just some simple modifications have to be made in lines 8, 11 and 13–14, so that the set of all optimal nodes $N^{(0)}$ is returned instead of just one optimal node n_{res}. For example we have to exchange the sharp inequality ($<$) in line 8 with a weak one ($\leq$). We will come back to this particular replacement when discussing experimental results.

More importantly, we have to choose a single node from $N^{(0)}$ when the main loop is finished. This is done such that the according test suite has optimal distribution quality. Therefore we extend definition (11) to nodes $n = (n_1, \ldots, n_i)$ by

$$\mathrm{Var}_p(n) = \mathrm{Var}_p\big(\{\mathrm{tc}_j : n_j = 1\}\big) \tag{13}$$

and insert the following steps between lines 16 and 17:

1: **for** $i = 1 \to p$ **do**
2: $\quad N^{(i)} = \{n \in N^{(i-1)} : \mathrm{Var}_i(n) \text{ is minimal}\}$
3: Pick $n_{\mathrm{res}} \in N^{(p)}$

3.4 Multi-objective Greedy Approach

In this subsection we present another approach to the problem, which is based on the Greedy algorithm from section 2.1. The strategy proposed here optimizes with respect to both objectives (test suite size and distribution quality) simultaneously and allows arbitrary weighting between them.

As discussed in 3.2, the requirement of $\mathrm{TS}_0 \subseteq \mathrm{TS}$ having a good distribution quality is equivalent to the one that its variances are minimal. To simplify matters we will only consider Var_1 here. Therefore this objective can be stated as

$$\mathrm{Var}_1(\mathrm{TS}_0) = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \left(d^{(1)}_{\mathrm{TS}_0}(a) - \overline{d^{(1)}_{\mathrm{TS}_0}} \right)^2 \to \min! \tag{14}$$

To combine it with the goal of minimizing the number of action calls we use a probabilistic approach. The probability of being contained in an optimal test suite with respect to distribution quality is greater for some $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$, if its distribution variance contribution is small. This contribution can be expressed by the value of Var_1 that would result when including tc into TS_0, $\mathrm{Var}_1(\mathrm{TS}_0 \cup \{\mathrm{tc}\})$, in relation to the sum of variances for all elements in $\mathrm{TS} \setminus \mathrm{TS}_0$, namely $\sum_{\tilde{\mathrm{tc}} \in \mathrm{TS} \setminus \mathrm{TS}_0} \mathrm{Var}_1\left(\mathrm{TS}_0 \cup \{\tilde{\mathrm{tc}}\}\right)$.

Thus, we start with an initially empty set of test cases TS_0 and iteratively add test cases $\mathrm{tc} \in \mathrm{TS}$ to TS_0 such that distribution variance increase is minimized. This leads to the following definition of probabilities for each $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$:

$$p_{\mathrm{var}}(\mathrm{tc}) := \frac{1 - \frac{\mathrm{Var}_1(\mathrm{TS}_0 \cup \{\mathrm{tc}\})}{\sum_{\tilde{\mathrm{tc}} \in \mathrm{TS} \setminus \mathrm{TS}_0} \mathrm{Var}_1\left(\mathrm{TS}_0 \cup \{\tilde{\mathrm{tc}}\}\right)}}{|\{\tilde{\mathrm{tc}} \in \mathrm{TS} \setminus \mathrm{TS}_0\}| - 1}. \tag{15}$$

Obviously, $0 \leq p_{\mathrm{var}}(\mathrm{tc}) \leq 1$ for all $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$ by construction and the expressions defined in equation (15) can be interpreted as probabilities.

For our primary objective of minimizing the number of action calls we will construct probabilities in an analogous way by using the same decision criterion as for the Greedy algorithm from section 2.1. Referring to line 5 of the algorithm we denote $w_{\mathrm{tc}} = \frac{1}{|\mathrm{tc}|} \cdot |\mathrm{cov}(\mathrm{tc}) \setminus \mathcal{R}_0|$ as the *weight* for each $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$. The weight of a test case tc is the total number of requirements $r \in \mathcal{R}$ that are satisfied by tc but not yet covered by any test case in TS_0. This value is normalized by the length of tc. It is clear that test cases with a high weight will more probably be contained in a test suite that is optimal with respect to test suite size than test cases with a lower weight. See [4] for more details. We can thus define probabilities for each $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$ as follows:

$$p_{\mathrm{rate}}(\mathrm{tc}) := \frac{w_{\mathrm{tc}}}{\sum_{\tilde{\mathrm{tc}} \in \mathrm{TS} \setminus \mathrm{TS}_0} w_{\tilde{\mathrm{tc}}}}. \tag{16}$$

So, given a rate proportion coefficient $\delta \in [0, 1]$ we can construct a weighted probability distribution by defining

$$p(\mathrm{tc}) := \delta \cdot p_{\mathrm{rate}}(\mathrm{tc}) + (1 - \delta) \cdot p_{\mathrm{var}}(\mathrm{tc}) \tag{17}$$

for each $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$.

A resulting Greedy strategy would be to iteratively sort test cases by their combined probability descending and add the test case with highest probability value to the final solution set. These preliminary considerations yield to a Greedy

algorithm like the one presented in section 2.1, except that lines 4 to 6 have to be replaced by the following steps[1]:

for $\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0$ **do**
  Compute $p(\mathrm{tc})$
$\mathrm{TS}_0' = \{\mathrm{tc} \in \mathrm{TS} \setminus \mathrm{TS}_0 : p(\mathrm{tc}) \text{ is maximal}\}$

4 Experimental Results

In this section we present experimental results for the optimization techniques described above. Due to computational constraints we only considered $p = 2$ for the lexicographical algorithms. All computations were performed on an AMD Opteron (tm) Quad Core with 2.60 GHz and 32 Gigabytes of RAM. As input we derived 13 different transition state machines which were designed on the basis of industrial case studies. To get realistic statements for the context of our work, most of our use cases are small- or intermediate-sized (I-IX). Nevertheless we included some larger models as well (X-XIII). Actually (XII) and (XIII) have proven to be too large to be optimized with algorithms of the Branch and Bound-type.

Table 3. Use cases

#	$\lvert\mathrm{TS}\rvert$	AC	$\mathrm{AC}^{(2)}$	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{A}^{(2)}\rvert$
I	15	84	69	13	26
II	21	123	102	10	15
III	32	128	96	13	28
IV	41	164	123	15	31
V	30	189	159	25	35
VI	36	190	154	31	40
VII	46	317	271	40	52
VIII	45	374	329	23	36
IX	120	600	480	15	33
X	132	1306	1174	26	40
XI	512	6656	6144	30	54
XII	284	1600	1316	140	422
XIII	625	4375	3750	23	86

For detailed information about the use cases consider table 3. It contains the number of test cases ($|\mathrm{TS}|$), the number of action calls (AC), the number of action-pair calls ($\mathrm{AC}^{(2)}$), the number of actions ($|\mathcal{A}|$) and the number of action-pairs ($|\mathcal{A}^{(2)}|$) for each of our use cases. Nevertheless the full model definitions must not be published due to legal reasons.

[1] Note that for $\delta = 1$ this algorithm is equivalent to the one proposed by [4].

4.1 Greedy Based Approaches

In the following we compare experimental results for the original Greedy algorithm from [4] with our proposed extensions. As parameters we choose $p = 2$ for the lexicographical variation and $\delta = 0.5$ for the multi-objective one. Our results are displayed in Table 4.

The second column (AC) contains the number of action calls in the unmodified test suite TS, while columns 4, 8, and 12 (AC_0) in each case contain the number of action calls in the resulting test suite TS_0. Columns 3, 7, and 11 (Time) denote the computation time in seconds needed to run the specific algorithm and columns 5–6, 9–10, and 13–14 present the variance values as defined by equations (7) and (11) for the corresponding reduced suite TS_0.

Table 4. Results for Greedy algorithms

		Original				Lexicographical				Multi-objective			
#	AC	Time	AC_0	Var_1	Var_2	Time	AC_0	Var_1	Var_2	Time	AC_0	Var_1	Var_2
I	84	0.06	16	0.33	0.00	0.08	16	0.33	0.00	0.09	16	0.33	0.00
II	123	0.06	25	2.65	0.86	0.08	25	2.05	0.56	0.09	25	2.05	0.56
III	128	0.08	32	7.17	1.89	0.09	32	4.40	0.16	0.08	32	4.40	0.89
IV	164	0.08	32	4.78	0.76	0.08	32	3.45	0.08	0.08	32	3.45	0.12
V	189	0.08	41	1.83	0.18	0.08	41	1.83	0.18	0.08	41	1.83	0.18
VI	190	0.08	47	0.64	0.15	0.09	48	0.64	0.21	0.09	48	0.64	0.21
VII	317	0.08	69	1.30	0.40	0.09	69	1.30	0.40	0.09	69	1.30	0.40
VIII	374	0.08	47	2.22	0.45	0.09	47	2.22	0.45	0.08	47	2.22	0.45
IX	600	0.09	25	1.56	0.15	0.13	25	1.16	0.19	0.11	25	1.16	0.19
X	1306	0.14	44	1.37	0.30	0.16	44	1.14	0.17	0.17	44	1.14	0.15
XI	6656	0.50	65	1.87	1.00	0.52	65	1.61	0.72	0.52	65	1.61	0.72
XII	1600	0.28	429	92.66	4.63	0.31	425	90.09	4.37	0.30	425	89.95	4.28
XIII	4375	0.36	35	1.82	0.59	0.39	35	1.82	0.59	0.39	35	1.82	0.59

We can see that the computation time for the lexicographical as well as for the multi-objective approach is always higher than the one for the original Greedy algorithm. This is just as expected, since computation and consideration of action call distribution takes additional time. Nevertheless we also note, that the additional time consumption is usually not very significant. The number of action calls is always equal for all algorithms except for the use cases VI and XII. The variances are usually decreased when running a modified algorithm, although for Var_2 this does not always hold. This is reasonable as well, since the lexicographical Greedy optimizes with respect to Var_1 first and the multi-objective Greedy does not consider Var_2 at all.

Another conclusion one can draw from the results is, that the two modifications of Greedy behave quite similarly except for use cases III and IV, where a significant difference in Var_2 can be noticed.

To sum up this discussion, we present the average values over the eleven first use cases in table 5[2]. Here we can see, that the variance values obtained by the original Greedy algorithm can be reduced by almost about 22% on average when using the lexicographical or the multi-objective approach. The values for Var_2 can be even be improved by almost 50% or 37% on average when using the lexicographical or the multi-objective approach, respectively.

Table 5. Comparison of Greedy algorithms

Algorithm	Avg Time	Avg AC_0	Avg Var_1	Avg Var_2
Original	0.121	40.273	2.338	0.558
Lexicographical	0.135	40.364	1.828	0.283
Multi-objective	0.134	40.364	1.828	0.352

4.2 Branch and Bound Based Approaches

Now let us compare the standard Branch and Bound approach from section 2.2 with our extension from 3.3. As parameter we again choose $p = 2$. The use cases are similar to the ones from the last section except that XII and XIII are excluded since the algorithms did not terminate within a reasonable time constraint. Additionally we have to remark that (in difference to the Greedy algorithms) redundant test cases as described in 2.2 were removed from the test suite prior to running the algorithms.

Our results are displayed in table 6. Here the second column (AC) contains the number of action calls in test suite TS after performing the removal of redundant test cases, but before running the Branch and Bound algorithms. The third column (AC_0) contains the number of action calls in the resulting test suite TS_0, i. e after performing Branch and Bound. By construction of the algorithms these numbers are always minimal and thus equal. Columns 4–6 contain further results for the unmodified standard algorithm, while columns 7–9 contain the further results for the modification discussed in section 3.3.

Considering use cases IX and XI it is evident, that the lexicographical approach is totally outperformed by the original algorithm in time. Analyzing this problem leads to the conclusion, that most of the additional time consumption yields from the change in line 8 of the algorithm, where a sharp inequality ($<$) is replaced with a weak one ($\leq$) in order to return all minimal solutions. Hence, we also tried to use another modified version, which uses sharp inequality ($<$) and therefore does not return all minimal solutions, but only a subset of these. Afterwards, the best suite with respect to distribution quality is chosen out of this subset just as in the lexicographical approach. The results for this modified lexicographical algorithm are displayed in columns 10–12 of table 6.

To compare the algorithms with each other we again computed average values, which are presented in table 7. One can see that standard Branch and Bound

[2] The last two examples have been excluded to ensure comparability with Branch and Bound results, see below.

Table 6. Results for Branch and Bound algorithms

			Original			Lexicographical			Lex. Mod.		
#	AC	AC_0	Time	Var_1	Var_2	Time	Var_1	Var_2	Time	Var_1	Var_2
I	78	16	0.06	0.33	0.00	0.09	0.33	0.00	0.09	0.33	0.00
II	123	17	0.06	1.01	0.14	0.09	1.01	0.20	0.11	1.01	0.14
III	128	32	3.66	7.17	1.89	7.55	4.40	0.16	3.78	4.40	0.16
IV	164	32	9.05	5.32	1.25	24.06	3.45	0.08	9.56	3.45	0.08
V	189	38	0.39	1.45	0.16	0.47	1.45	0.16	0.42	1.45	0.16
VI	190	44	10.25	1.08	0.44	11.28	1.08	0.44	10.28	1.08	0.44
VII	317	64	239.61	0.89	0.28	271.66	0.89	0.27	234.02	0.89	0.27
VIII	185	47	0.06	2.22	0.31	0.11	2.22	0.28	0.11	2.22	0.28
IX	600	25	77.49	1.56	0.31	1755.37	1.16	0.05	81.72	1.16	0.10
X	1089	43	198.68	1.38	0.40	230.98	1.15	0.15	198.94	1.38	0.40
XI	4368	52	19.00	1.13	0.52	484.06	1.00	0.37	19.38	1.00	0.37

and the modified lexicographical version have almost equal computation time on average (about 50 seconds). Nevertheless the average variance values for the latter one are considerably better than those for standard Branch and Bound (about 20% for Var_1 and almost 60% for Var_2). The average variance values for the "exact" lexicographical version are of course even smaller, but do not advance very much (only about 2% for Var_1 and 10% for Var_2). However, this benefit comes with the cost of a significant increase in computation time (about 400%).

Table 7. Comparison of Branch and Bound algorithms

Algorithm	Avg Time	Avg AC_0	Avg Var_1	Avg Var_2
Original	50.755	37.273	2.139	0.518
Lexicographical	253.248	37.273	1.648	0.197
Lex. Mod.	50.765	37.273	1.669	0.218

4.3 Comparison of Results

We can see that for the Greedy as well as for the Branch and Bound approaches taking distribution of action calls into account can yield to considerably smaller variances than for the standard versions. Nevertheless except for the lexicographical Branch and Bound algorithm computational effort for the extended approaches is not significantly higher.

When comparing the results for the extended Branch and Bound with the extended Greedy approaches we see that the number of action calls for all Branch and Bound approaches is about 7.5% less on average than the corresponding numbers for the Greedy strategies. Furthermore, the variance of action calls can be decreased by about 10% and the variance of action-pair calls even by 22% up to 43% on average when using an extended Branch and Bound algorithm instead of lexicographical or multi-objective Greedy. However, for the most examples this is dearly bought with a dramatically higher computation time in comparison to the Greedy strategies.

5 Conclusion

In this paper we presented two classical solutions for the test suite reduction problem, namely the Branch and Bound algorithm, which computes an exact solution in exponential time, and the Greedy heuristic, which yields the best approximation possible in polynomial time. Based on these algorithms we introduced modifications to advance distribution quality.

A main contribution of this paper is the formalization of the term "distribution quality" itself. With the variances Var_p at hand a mathematical description of the problem can easily be given. Our modifications of the algorithms introduce simple but effective ways to use this description in order to solve the problem.

Experimental results support these approaches. At no time the result of a modified algorithm was outperformed by the result of its unmodified counterpart in distribution quality. Conversely the variances shrunk in most use cases, at times tremendously. Both variations of the Greedy algorithm performed almost equally and were only marginally slower than the unmodified version. On the other hand it showed that our first modification of Branch and Bound was significantly slower, such that we would not advice to use it. Nevertheless we also introduced a variation that comes with nearly the full advantage of a better distribution quality, but computes insignificant longer compared to standard Branch and Bound.

References

1. Black, J., Melachrinoudis, E., Kaeli, D.: Bi-criteria models for all-uses test suite reduction. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 106–115. IEEE Computer Society, Washington, DC (2004)
2. Chen, T.Y., Lau, M.F.: A new heuristic for test suite reduction 40(5-6), 347–354 (1998)
3. Chen, T.Y., Lau, M.F.: Dividing strategies for the optimization of a test suite. Information Processing Letters 60, 135–141 (1996)
4. Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of Operations Research 4(3), 233–235 (1979)
5. Gupta, R., Soffa, M.L.: Compile-time techniques for improving scalar access performance in parallel memories. IEEE Trans. Parallel Distrib. Syst. 2, 138–148 (1991)
6. Harrold, M.J., Unwersity, C., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2, 270–285 (1993)
7. Chinneck, J.W.: Practical Optimization: A Gentle Introduction (2003), `http://www.sce.carleton.ca/faculty/chinneck/po.html` (Chapter 13)
8. Mansour, N., El-Fakih, K.: Simulated annealing and genetic algorithms for optimal regression testing. Journal of Software Maintenance 11, 19–34 (1999)
9. Offutt, A.J., Pan, J., Voas, J.M.: Procedures for reducing the size of coverage-based test sets. In: Proc. Twelfth Int. Conf. Testing Computer Software, pp. 111–123 (1995)
10. Poppendieck, M., Poppendieck, T.: Lean software development: An agile toolkit. Addison-Wesley Professional (2003)

11. Rothermel, G., Harrold, M.J., von Ronne, J., Hong, C.: Empirical studies of test-suite reduction. Journal of Software Testing, Verification, and Reliability 12, 219–249 (2002)
12. Wang, H.S., Hsu, S.R., Lin, J.C.: A generalized optimal path-selection model for structural program testing. Journal of Systems and Software 10(1), 55–63 (1989)
13. Wieczorek, S., Stefanescu, A.: Improving Testing of Enterprise Systems by Model-Based Testing on Graphical User Interfaces. In: 2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, pp. 352–357. IEEE (2010)
14. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA 2007, pp. 140–150. ACM, New York (2007)