FLOWER: Optimal Test Suite Reduction as a Network Maximum Flow

Arnaud Gotlieb and Dusica Marijan
Certus Software Validation & Verification Center
Simula Research Laboratory, Norway
{arnaud,dusica}@simula.no

ABSTRACT

A trend in software testing is reducing the size of a test suite while preserving its overall quality. Given a test suite and a set of requirements covered by the suite, *test suite reduction* aims at selecting a subset of test cases that cover the same set of requirements. Even though this problem has received considerable attention, finding the smallest subset of test cases is still challenging and commonly-used approaches address this problem only with approximated solutions. When executing a single test case requires much manual effort (e.g., hours of preparation), finding the minimal subset is needed to reduce the testing costs.

In this paper, we introduce a radically new approach to test suite reduction, called FLOWER, based on a search among network maximum flows. From a given test suite and the requirements covered by the suite, FLOWER forms a flow network (with specific constraints) that is then traversed to find its maximum flows. FLOWER leverages the Ford-Fulkerson method to compute maximum flows and Constraint Programming techniques to search among optimal flows. FLOWER is an exact method that computes a minimum-sized test suite, preserving the coverage of requirements. The experimental results show that FLOWER outperforms a non-optimized implementation of the Integer Linear Programming approach by $15-3000$ times in terms of the time needed to find an optimal solution, and a simple greedy approach by $5-15\%$ in terms of the size of reduced test suite.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing

General Terms

Theory

Keywords

Automated testing, test-suite reduction, graph theory, maximum flow, Ford-Fulkerson

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
ACM 978-1-4503-2645-2/14/07
http://dx.doi.org/10.1145/2610384.2610416

1. INTRODUCTION

As software evolves over time, test suites or test configurations used to validate the software grow. On the other hand, according to several studies [22, 30, 12], test suite size has a large impact on the overall test cost in the software development lifecycle. Therefore, a trend in software testing is to keep the test suite size as small as possible, while preserving the "quality" of the test suite (e.g., its requirements coverage rate, its overall running time, its expected fault-detection rate or a combination of those). For software testing contexts where executing a single test case requires much manual effort and preparation, finding the smallest set of test cases is essential to keep the test cost as low as possible.

Test suite reduction, also called a test suite minimization, is a general problem cross-cutting several related research areas. In code-based testing where test requirements correspond to code coverage objectives, the goal is to minimize a test suite while preserving code coverage. In functional testing with combinatorial interaction testing, test suite reduction aims at selecting the smallest test suite covering all N-tuples of input parameters. In software product line testing, a challenge is to find the minimal number of test configurations so that all product line features are covered at least once. In regression testing, test suite reduction aims at finding the smallest subset of test cases preserving the error-detection ability of the original test suite. These examples reveal that the main problem addressed in this paper (i.e., test suite reduction) is of a fundamental nature and deserves to be explored in all possible dimensions. Optimal test suite reduction is obtained when the subset is of minimum cardinality. However, the problem of finding a minimum-cardinality subset (not necessarily unique!), known as a Minimum Set Cover, is NP-complete [8, 6]. In the worst case, the time required to compute a minimum-cardinality subset grows exponentially with the size of the problem.

Substantial research work has addressed different variations of this problem. However, most common approaches proposed in the literature (see section 6) exhibit some of the following three drawbacks:

- *Minimum-cardinality test suite is not guaranteed.* Commonly used approaches to compute the minimum cardinality subsets exploit greedy algorithms to find the test cases covering the set of test requirements [24, 29, 16]. For example, they select first the test case that covers the most test requirements, then remove all requirements covered by the selected test case and repeat the process until all requirements are covered. While generally quick, greedy algorithms can only approximate the global-optimum solutions. Once a test case is selected, the choice is usually not reconsidered to find another test case covering less, but different requirements. An exact method, on the contrary, is a method able to compute a minimum-cardinality subset;

- *Tradeoff between test-reduction time and number of test cases.* Exact approaches based on Integer Linear Programming (ILP), such as [2, 4, 12, 9] are able to find a (non-approximated) optimal solution, but at the cost of long run time. On the other hand, greedy heuristics exhibit better runtime, but usually at the cost of larger test suites. Both minimum-cardinality test suite and low runtime are important aspects of test suite reduction in practice and therefore, there is usually a need for a tradeoff;

- *Fault-detection capability or code coverage is not preserved.* When looking only at the size of test suites, while preserving requirements coverage, test suite reduction can compromise other essential characteristics, such as fault revealing capability or code coverage [22, 21, 28, 27]. Some of the recent approaches to test suite reduction have addressed this problem by combining multi-objective optimization methods and search-based test data generation [26] or Integer Linear Programming [2, 12]. However, there is still a challenge to understand how to balance between these criteria and how to select optima in a multi-objective optimization problem.

In this paper, we introduce a new method for test suite reduction (called FLOWER) based on the computation of network maximum flows. From a given test suite and test requirements covered by the suite, FLOWER forms a bipartite graph with special capacity constraints and searches for the maximum flows in its corresponding flow network. FLOWER leverages the Ford-Fulkerson method [7] to compute the maximum flows and constraint programming techniques to search among the optimal flows. According to our knowledge, FLOWER is the first approach that uses network maximum flows to address the test suite reduction problem.

Our approach addresses all three drawbacks of the conventional approaches mentioned above. It is an exact method that finds a true minimum number of test cases covering the same set of requirements as the original test suite. FLOWER is quick for medium-size problems. In particular, for the problems consisting of 3000 requirements and 500 test cases, it computes the minimum-cardinality subset in less than a minute on a standard machine. In addition to test suite reduction retaining requirements coverage, FLOWER can handle multiple criteria in reduction, for example, test execution cost or fault-detection capability.

To evaluate FLOWER, we developed a prototype implementation using the SICStus clpfd library [3] and we conducted a comparative analysis of the time needed to reduce test suites and the size of the reduced suites, for 2005 test suite reduction problems. In this experimental study, FLOWER is compared with a simple greedy model and a non-optimized ILP model, which are implemented using the classical description in the literature [5, 12]. Our results demonstrate that FLOWER outperforms the ILP approach by $15 - 3000$ times in terms of time needed to find solutions, and the simple greedy approach by $5 - 15\%$ in terms of the size of reduced test sets.

The paper is organized as follows: in Section 2 we review the theory underlying our proposed approach to test suite reduction. In Section 3 we describe the problem of test suite reduction, give its graph-based formulation and show how the problem can be solved using a maximum flow in a graph. Section 4 describes the implementation of FLOWER. In Section 5 we evaluate FLOWER compared to the ILP and the greedy approach. Section 6 contains related work, while Section 7 concludes the paper.

2. BACKGROUND AND NOTATIONS

This section revisits the problem of test suite reduction and briefly reviews the concepts of network flow theory.

2.1 Test Suite Reduction

Test suite reduction aims to decrease the overall number of test cases while retaining the coverage of the original test suite. A reduced test suite covering the same test requirements as the original suite, but not necessarily of the smallest possible size, is called a *representative test suite.* An optimal test suite reduction aims to determine the representative test suite consisting only of essential test cases [15], known as a *minimum-cardinality subset.* If any of the test cases from an optimal representative test suite were removed, the test suite would not satisfy all test requirements. In this paper, we will use RTS to refer to a Representative Test Suite and MCS to refer to a Minimum-Cardinality Subset, i.e., an optimal RTS. Note that an MCS is not necessarily unique for a given test suite reduction problem.

Formally, a test suite reduction problem is defined by an initial test suite $T = \{T1, \ldots, Tm\}$, a set of test requirements $R = \{R1, \ldots, Rn\}$ and a function $cov : R \to 2^T$ mapping each requirement to the subset corresponding to its covering test cases. We suppose that each requirement is covered by at least one test case, i.e., $\forall i \in \{1, \ldots, n\}, cov(R_i) \neq \emptyset$. An example with 3 test cases and 7 test requirements is given in Table 1, where 1 denotes that the requirement is covered by the corresponding test case, and 0 denotes the opposite. Given T, R and cov, an RTS T' is a test suite $T' \subseteq T$ such that $\forall i, \exists t \in T'$ and $t \in cov(R_i)$, meaning that each requirement is covered by at least one test case from T'. If $Card(S)$ denotes the cardinality of the set S, an RTS T' such that $card(T')$ is minimal, is an MCS.

Table 1: Coverage of test requirements by test cases

	R1	R2	R3	R4	R5	R6	R7
T1	1	1	0	1	0	0	1
T2	1	0	0	0	0	1	0
T3	0	0	1	1	1	1	0

2.2 Flow Network

A *flow network* is an oriented graph $G = (V, A)$ where each arc $(u, v) \in A$ is associated with a *capacity* $c(u, v) \geq 0$. Every flow network contains two special vertices: the source s and the sink t, and for each $v \in V$ there exists a path from s to t traversing v (connexity). A *feasible flow* (or *flow* in short) in G is a function $f : V \times V \rightarrow \mathbf{R}$ that associates to each arc (u, v) a real value $f(u, v)$ such that the following laws are satisfied:

1. Capacity constraints:
   $\forall (u, v) \in A, 0 \leq f(u, v) \leq c(u, v)$;

2. Flow conservation law:
   $\forall u \in V \setminus \{s, t\}, \sum_{v \in V} f(v, u) = \sum_{w \in V} f(u, w)$.

Provided that there is no incoming arc to the source s, the *value of a flow*, noted $|f|$ is equal to $\sum_{v \in V} f(s, v)$. The *maximum flow problem* aims at finding a flow f^* with a maximum value. In general, there are several flows with maximum value and computing a maximum flow is done using the Ford-Fulkerson method [7]. The Ford-Fulkerson method is based on the residual graph G_f associated with a given flow f, and the computation of augmenting paths [7, 6]. In brief, the method iteratively increases the value of a given feasible flow by finding an augmenting path within the residual graph associated with this flow. The key-point of the method is exhibited by the following procedure:

```
Input: $G, s, t$
Output: Maximum flow $f^*$

Initialize $f$ to 0;
while There exists an augmenting path $p$ in the residual network $G_f$ do
  | Increase flow $f$ along $p$;
end
return $f$
```

There are several variants of this method, e.g., the Edmonds-Karp algorithm, with different complexities. For the approach presented in this paper, any of these algorithms can be employed. Finally, an important result of the network flow theory for our approach is the following: *For any flow network, if the capacities are integers, then any flow value (including the maximum) is an integer [6].* This also applies to the flow values on arcs. This property is crucial when encoding a test suite reduction, because it explains why every computation can be performed on variables of integer type.

2.3 Bipartite Graph

A *bipartite graph* is a graph $G = (V, A)$ where V is partitioned in two subsets: L and R (i.e., $L \cup R = V$ and $L \cap R = \emptyset$), with arcs between L and R only. Figure 1 illustrates a bipartite graph for the test suite reduction problem in Table 1.

3. THE FLOWER APPROACH

In this section we present how to encode a test suite reduction as a network maximum flow problem. We show the equivalence between maximum flows and RTS, and how to compute an MCS for a test suite reduction problem. Further, we review the complexity analysis of the entire approach.

3.1 Encoding Test Suite Reduction with a Flow Network

We encode a test suite reduction with a bipartite graph $G = (V, A)$ augmented with a source node S and a destination node D. Formally speaking, $V = R \cup T \cup \{S, D\}$ and there is an arc $(u, v), u \in R, v \in T$ iff $v \in cov(u)$. For each

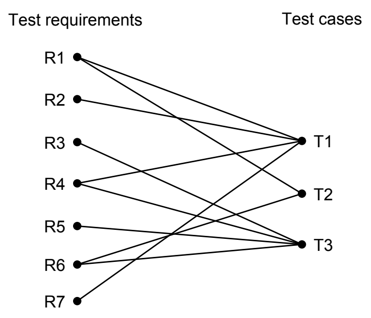

Figure 1: Bipartite graph for test suite reduction

arc (u, v) in the bipartite graph, we add an integer capacity $c(u, v)$ as follows:

1. For any arc (u, v) from S to R_i, we add $c(u, v) = 1$;

2. For any arc (u, v) from R_i to T_j, we add $c(u, v) = 1$;

3. For any arc (u, v) from T_j to D, we add $c(u, v) = \sum_{w \in T} c(w, u)$.

For any maximum flow in G, Item 1 enforces the coverage of all requirements, Item 2 encodes the values of the cov function, while Item 3 encodes the various possible selections of test cases. Figure 2 illustrates the network flow G associated with the test suite reduction problem in Table 1.

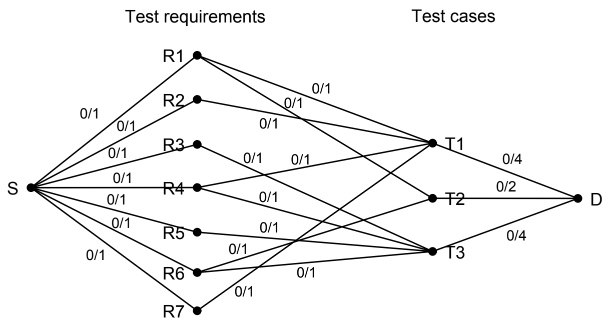

Figure 2: Flow network with 3 test cases covering 7 test requirements. Flow value and capacity are denoted as l/c on each arc. Here, a trivial feasible flow is presented with value $|f| = 0$.

3.2 Finding a Representative Test Suite Using Maximum Flows

The following theorem is essential to link a test suite reduction to a network maximum flow:

Theorem 1 *Let G be the network flow associated with a test suite reduction problem. A representative test suite (RTS) is equivalent to a maximum flow in G.*

To demonstrate the above theorem, we will use the following three properties that we prove equivalent:

1. f is a maximum flow in G;
2. $|f| = n$, where n is the number of requirements;
3. T, composed of the vertices t such that $f(t, D) \neq 0$, is an RTS.

Proof.
(1 $\implies$ 2) For any flow f, $\forall i \in \{1, \ldots, n\}, f(S, R_i) \leq c(S, R_i) = 1$. Therefore, $|f| \leq n$. Proving that there exists a feasible flow f of value n would show that $|f| \geq n$ as well. Consider the maximum flow f where each $\forall u \in R, f(S, u) = 1$. It is a feasible flow because each requirement is covered by at least one test case, and thus the flow conservation law from S to D is preserved, i.e., $\sum_{w \in T} f(w, D) = \sum_{u \in R} f(u, t) = \sum_{u \in R} f(S, u)$. Accordingly, all the capacity constraints are satisfied. Therefore, $|f| = n$.
(2 $\implies$ 3) If $|f| = n$ then $\sum_{w \in T} f(w, D) = n$ and T is the set of test cases t for which $f(t, D) \neq 0$
(3 $\implies$ 1) Consider the flow f associated with the subset T of test cases t for which $f(t, D) \neq 0$. Let us prove that f is actually a maximum flow. As T is an RTS, it means that all requirements are covered. Thus, $\sum_{t \in T} f(r, T) = \sum_{u \in R} f(S, u) \leq \sum_{u \in R} c(S, u) = n$, where n is the number of requirements. As f has reached all the capacity values on arcs from the source S to R, then f is maximum.

As the result, for any maximum flow f, the subset of test cases $T' \subseteq T$ such that $f^*(t, D) \neq 0$, is an RTS, what concludes the proof of Theorem 1.

Figure 2 illustrates a test reduction problem encoded as an augmented bipartite graph G. G is directed from S to D, with l/c values assigned to all arcs. In Figure 2, $l = 0$ for all arcs, meaning that the flow is feasible but not maximum. Maximum flows, as shown in Figure 3, correspond to various solutions of the RTS problem. More precisely, the maximum flow values on arcs give us the RTS, i.e., the subset of a test suite that covers all test requirements. For example, a maximum flow $\langle S-R1-T1-D; S-R2-T1-D; S-R3-T3-D; S-R4-T1-D; S-R5-T3-D; S-R6-T2-D; S-R7-T1-D\rangle$ is associated with the set $\{T1, T2, T3\}$ because none of the flow values on arc from T_i to D are null. This solution is useless as it does not provide any reduction of the original test suite. However, as illustrated in Fig. 3, there are 8 possible maximum flows in G and two of them exhibit optimal reduction, i.e., consist of only two test cases. Our goal is then to search among maximum flows for the flow that maximizes the number of zeros on the arcs from T to D.

3.3 Finding a Minimal-Cardinality Subset

In this section, we prove a theorem that links the Minimum-Cardinality Subset to the search among maximum network flows. We also describe the practical labelling search procedure used to search among maximum flows.

Let the *critical branches* be the final arcs of the bipartite graph, namely the $\langle T_i - D\rangle$ arcs, and *critical values* the values of the flow on these arcs. For any flow f, let $q_1^f, \ldots, q_m^f$ be the critical values associated to f on m critical branches, and let n be the number of requirements. Proved in the previous section, the following property holds: *For any maximum flow f^*, $\sum_{i \in 1..m} q_i^{f^*} = n$.* This property is crucial to prove the following theorem, which contains the essence of the FLOWER approach:

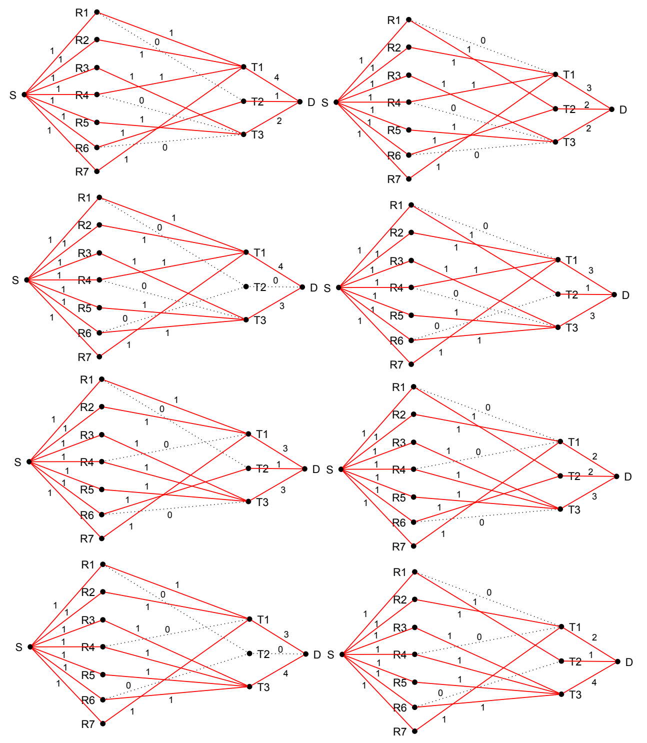

Figure 3: Feasible solutions (maximum flows in a graph) for the MCS problem with 3 test cases covering 7 test requirements.

Theorem 2 *A maximum flow f^* such that the number of critical branches with zero flow is maximum, represents a Minimal-Cardinality Subset (MCS), and conversely.*

Proof. Theorem 2 states that if $Card_{i \in 1..m}(\{q_i^{f^*} \text{ s.t.} q_i^{f^*} = 0\})$ is maximum then the RTS corresponding to f^* forms the MCS. According to the above mentioned property, $\sum_{i \in 1..m} q_i^{f^*} = n$. Therefore, by duality, it means that $Card_{i \in 1..m}(\{q_i^{f^*}$ s. t. $q_i^{f^*} \neq 0\})$ is the minimum. Then, considering the RTS T associated to f^*, we get that T is of minimum cardinality, and is thus the MCS. Conversely, there is no smaller set T included in the MCS and therefore, the number of zero flows in the maximum flow f^* is maximized.

Theorem 2 ensures both the correctness and completeness of the FLOWER method by bridging two separate domains, namely Software Testing and Network Flow Theory. However, the Theorem is not constructive, as it says nothing about how to find an MCS. Actually, finding an MCS requires searching among maximum flows. Conveniently, the Ford-Fulkerson method [7] can be combined with a labeling procedure, which is an incremental assignment process of values to variables. The overall search space formed by the various maximum flows can be explored through the assignment one-by-one of the critical values of the bipartite graph. In the example in Figure 3, labelling variables $T1, T2, T3$ permit us to explore all the maximum flows in a flow network. One way to explore the search space is to assign all variables at once, e.g., searching for the solutions with 3 values assigned to $T1, T2, T3$. Another more subtle way is to start by assigning a value to one of the variables and propagating this assignment throughout the flow network, by backtracking until the last assignment, if a contradiction is found, and repeat the assignment process until all variables are assigned. In the example, giving value 0 to $T1$ would lead to contradiction, because the requirement $R2$ cannot be covered any more. In fact, depending on the order of critical branches, the labeling procedure selects a branch and a value, propagates the value through the flow network and evaluates the number of zeros in the remaining critical branches. If the number of zeros is less than the current value, the procedure backtracks to the branch/value selection and makes another choice. Propagating a choice through the flow network means pushing values for the other flows, depending on the initial choice. This process is well-known in Discrete Optimization under the terminology branch-and-bound.

3.4 Multi-objective Test Suite Reduction

In a basic form, test suite reduction aims to retain the requirements coverage of the original test suite while reducing the overall number of test cases. However, there is often a need in practice to consider multiple objectives in test reduction, such as fault detection capabilities, test execution time or code coverage, which can all be considered as alternative priorities. FLOWER supports multi-objective test suite reduction, with test requirements coverage as a primary objective and test fault-detection data as a secondary objective, similarly to the idea that the authors have explored previously [17]. After all MCSs have been computed, secondary objectives are considered to propose the solution with the best fault detection rate. This is handled by exploring the capacity constraints that encode priorities instead of values 1 in the bipartite graph encoding the test suite reduction. Leveraging the cost-based solving procedures [20], MCSs are considered one-by-one to evaluate their fault detection capabilities (or alternatively, test execution time, code coverage, or other priorities relevant for users). In this way, FLOWER provides an MCS that is optimal with respect to test suite size, taking into account fault-detection capability. A drawback of this approach is that well-balanced solutions (not necessarily optimal in terms of test suite size) may not be found by FLOWER. On the other hand, the advantage is that it encodes a simple and efficient priority-based scheme for addressing multi-objective optimization.

3.5 Complexity Analysis

The Ford-Fulkerson method is available in several variations, i.e., algorithms with distinct complexity classes. The algorithm selected in FLOWER is a classical one based on the residual graph and augmented paths. Its asymptotic time-complexity to compute the maximum flow is known to be $O(a*|f|)$ in the worst case, where a is the number of arcs in the bipartite graph and $|f|$ is the flow value. In the context of RTS, we already know that $|f|$ is equal to the number of requirements. Therefore, the complexity is $O(a*n)$ where n is the number of requirements. However, the "costly" part of the FLOWER approach is the search among maximum flows. This procedure is exponential in time as a backtracking procedure may explore all the possible values for critical branches in the worst case. As there are as many critical branches as test cases, the worst-case complexity of this labelling process is $O(n^m)$, where m is the number of test cases. Note that, as the Ford-Fulkerson is launched every time a choice on a critical branch and critical value is made, the overall complexity is $O(a * n^{m+1})$. This result is not surprising as finding an MCS is NP-complete problem.

Although FLOWER may lead to a combinatorial explosion in the worst case, our experiments show that it scales up to the problems of industrial size. Complexity analysis of other approaches used for solving the same problem indicates that the ILP approach in the worst-case is similarly exponential in the number of test cases m [6]. This problem can however be approximated by polynomial algorithms, using greedy approaches. The greedy approach we implemented for the purpose of comparative evaluation is of complexity $O(m * log(m) + 2 * m * n)$. We detail this implementation in the next section.

4. FLOWER IMPLEMENTATION

FLOWER tool is implemented in SICStus Prolog, using the `clpfd` library [3]. This library embeds the constraint solver over finite domains variables, i.e., variables that have a domain of finite integer values. Using this solver allows us to encode the relations between test cases and requirements directly as domain constraints. For example, the bipartite graph in Figure 1 is encoded in FLOWER as: $R1 \in \{1,2\}, R2 = 1, R3 = 3, R4 \in \{1,3\}, R5 = 3, R6 \in \{2,3\}, R7 = 1$. Using this encoding automatically removes test requirements that are covered by a single test case (e.g., $R2$ is assigned to 1) and includes the corresponding test case in the solution set (e.g., T1). In this example, the relation is composed of only 3 requirement variables, because $R2, R3, R5$ and $R7$ are already assigned. This is a simple encoding of the problem using finite domain constraints, but still very efficient.

The Ford-Fulkerson algorithm is interfaced with FLOWER through the filtering procedure of the *global-cardinality constraint* [20]. This constraint is denoted `global_cardinality`$([X1, ..., Xd], [K1 - V1, ..., Kn - Vn])$ or `gcc` in short. In this formulation, the Xi are integers or domain variables, and $[K1 - V1, ..., Kn - Vn]$ is a list of pairs where each key Ki is a unique integer and Vi is a domain variable or an integer. The constraint enforces that Xi is equal to some key and for each pair $Ki - Vi$, exactly Vi elements of $[X1, ..., Xd]$ are equal to Ki. Referring to Section 3, Vi represents a critical value of the critical branch in the corresponding bipartite graph. Therefore, encoding the search for a maximal flow in a bipartite graph is realized by the following call
`gcc`$([R1, ..., Rn], [1 - V1, ...m - Vm])$, where Ri represents the variables associated to the test requirements and Vi represents the critical values in the maximal flow solution set. For example, the problem in Figure 1 is encoded using the following call:
`gcc`$([R1, 1, 3, R4, 3, R6, 1], [1 - V1, 2 - V2, 3 - V3])$ where $V1 \in \{0, .., 4\}, V2 \in \{0, 1, 2\}, V3 \in \{0, .., 4\}$.

In addition to the `gcc` constraint, a branch-and-bound procedure is implemented in `clpfd`. The objective function is the number of zeros in the list $[V1, .., Vm]$, so that maximizing this function reduces to the minimization of the test suite. This procedure is combined with a labeling process over the Vi (i.e., the critical values), which incrementally assigns the values for the Vi in specific order. Several heuristics control which branch and value should be selected first. Each time a Vi is selected for assignment, the objective function is evaluated and its current value recorded. When selecting other values for the same Vi or a different variable, if the number of zeros is lower than the current recorded number, then all the remaining search space to be explored with this value is just discarded. For evaluating the objective function in FLOWER, we maintain a finite domain variable that is bounded to the number of zeros.

This branch-and-bound procedure is time-aware, also known as an anytime algorithm [31], meaning that a given contract of time is allocated by the user beforehand. In fact, most of the time a candidate optimum is proposed very quickly and the rest of the time is spent proving that this is actually the optimum. Therefore, allocating a contract of time permits us to obtain an optimum, and control the time allocated to the proof. This is particularly useful to control the search when several problems have to be solved or very large instances are considered.

For encoding problems to solve, we defined specific input/output formats that are processed in FLOWER by building appropriate data structures. Note that the management of bipartite graph is handled directly by the filtering algorithm of the `gcc` constraint.

5. EXPERIMENTAL EVALUATION

In this section we present the experimental study performed to evaluate FLOWER. First, we study time effectiveness of FLOWER in computing an MCS for a large set of test suite reduction problems. Second, we study how well FLOWER performs in terms of the size of reduced test suites. We compare FLOWER with our own implementations of an ILP model and a simple greedy approach, using descriptions from the literature [2, 5].

The research questions evaluated in this study are:

- **RQ1:** Is FLOWER time-effective in computing an MCS compared to the ILP and greedy approaches?
- **RQ2:** Does FLOWER achieve better test suite reduction rate compared to the ILP and greedy approaches?

5.1 Experiment Subjects

We evaluated FLOWER on a set of 2000 randomly generated TSR problems, specified as a test suite with a set of test requirements and an association defining which test cases satisfy which test requirements. The size of the test suites in the TSR problems ranges from 20 to 1000 test cases, and the number of test requirements from 20 to 5000 requirements. The problems were generated as 20 sets of 100 problems each. The problems in a set have the same test suite size and the same number of test requirements satisfied by the suite. They differ in the number of requirements satisfied by each test case and their assignment to a test case. An important aspect to consider when generating TSR problems is the number of test requirements associated with one test case. This property of a TSR problem affects the time performance of the TSR process. To take into account this factor and evaluate different TSR problems, the number of test requirements associated with one test case in our experimental subjects varies from 1 to the total number of test requirements in a test suite. Table 2 provides summary information about the test subjects used in the experiment.

Supplementary to the evaluation on synthetic subjects, we assessed FLOWER on five subjects representing the test suites used to test industrial video-communication configurable systems. These subjects were selected as the most representative suites from the test database, which contains several tens of suites, mainly similar in size. The test subjects vary from 30 to 59 requirements and from 90 to 107 test cases, with 3 to 11 requirements covered by one test case in average and a standard deviation from 1.1 to 1.5. The suites in a test database are specified manually and as they evolve over time, very often they contain test cases covering the requirements covered by other test cases in the suite. To reduce testing effort and support cost-effective testing, such test suites need to be reduced to their minimal size. Table 3 summarizes information about the real subjects.

5.2 Experiment Setup

The goal of the study is to evaluate the effectiveness of FLOWER in obtaining the MCS for the considered experiment subjects in terms of (1) time to compute the MCS and (2) size of the reduced test suite. To do this, we compare FLOWER with the ILP and the greedy approaches. First, we compare the time taken by FLOWER with the time required by the ILP and greedy. Next we compare the size of the reduced test suite obtained by FLOWER with the size of the suite given by the ILP and greedy.

For the purpose of evaluation, we implemented the ILP model using *clpq* [11], a linear constraint solver over the rationals. The solver is based on classical techniques such as the Fourier elimination, the simplex algorithm and the branch-and-bound procedure for integer problems. The ILP model for optimal test suite reduction encodes the selection of each test case with a $0-1$ variable and each association of a requirement to a test case with a linear inequality. Minimizing the number of test cases requires minimizing the linear sum expressions over all $0-1$ variables. For the same purpose, we implemented a greedy approach as follows: test cases are first sorted by increasing order of requirements coverage using quicksort. Next, an iterative procedure selects one by one a test case until all the requirements are covered at least once.

We conducted our experimental study as follows: First, for each test subject, we ran FLOWER, the ILP and greedy models and measured the time to compute an MCS; Second, we measured the size of the reduced test suites computed by the three approaches. The experiments were run on Intel Core i7-2929XM CPU at 2.5 GHz with 16 GB RAM.

Table 3: Real experiment subjects with an average number of test requirements covered by a test case (rounded to nearest) and a standard deviation.

	Res1	Res2	Res3	Res4	Res5
Num. of test req.	59	59	50	35	30
Num. of test cases	107	90	93	100	100
Coverage	6	7	3	5	11
Deviation	1.2	1.2	1.5	1.1	1.2

5.3 Results and Analysis

This section presents and discusses the results of our experimental study, in order to answer RQ1 and RQ2.

Table 2: Experiment subjects: ES is a set of 100 test suite reduction problems that have the same test suite size and the same number of test requirements satisfied by the suite.

	$ES1$	$ES2$	$ES3$	$ES4$	$ES5$	$ES6$	$ES7$	$ES8$	$ES9$	$ES10$
Number of test requirements	50	250	350	500	200	50	100	20	50	100
Number of test cases	20	20	30	30	40	50	50	100	300	500

	$ES11$	$ES12$	$ES13$	$ES14$	$ES15$	$ES16$	$ES17$	$ES18$	$ES19$	$ES20$
Number of test requirements	400	400	450	500	500	1000	2000	3000	4000	5000
Number of test cases	200	1000	300	250	300	500	200	500	1000	1000

5.3.1 *Time Effectiveness*

To answer RQ1, we analysed the time taken by FLOWER, the ILP model and greedy model to compute the MCS for 2000 experimental subjects in Table 2. The data indicated that FLOWER and the greedy approach were able to find the solution for all considered subjects, while the ILP approach was not able to compute the solution for more complex problems ES9-ES20 in reasonable time. The results showed that the performance of FLOWER and the greedy approach is fairly similar across all subjects. FLOWER took in average between 20% and 30% more time than the greedy approach. However, the ILP model exhibits a significant increase in running time as the size of the problems grows. In particular, for the simplest problems ES1 and ES2, the ILP model required up to 16 seconds in the worst case to find the MCS, while FLOWER took 0.047 seconds in the worst case for the same problems. For the ES6, the worst-case running time for the ILP model was 465 seconds, while for ES7 it went up to 700 seconds, compared to 1.504 seconds in a worst case for FLOWER for the same problems. For the complex test suite reduction problems ES9-ES20, the ILP was not able to compute the solution after more than 700 seconds of running. On the contrary, FLOWER was very quick in solving those problems, it reduced the problems containing 3000 requirements covered by the set of 500 test cases in less than 1 minute on average. Figure 4 shows relative difference between the time taken by FLOWER and the greedy model on average, for 15 sets consisting of 100 randomly generated subjects. The results are in favour of the greedy model, showing that the greedy model consistently outperforms FLOWER in terms of time.

We explain the difference in the performance between FLOWER and the ILP model by the properties of the reduction problem. Our experimental results show that the number of test cases in a suite has the largest impact on the ILP model running time. As this number increases, the overall search space, which is a T-dimension convex polyhedron, where T is the number of test cases, increases as well. On the contrary, FLOWER is less affected by the increase of a number of test cases in a suite, as it encodes the coverage relation using a sparse representation (e.g., R_1 is covered by T_2 and T_5 is encoded in FLOWER as variable R_1 has domain $\{2, 5\}$, while it is encoded as $0 * B_1 + 1 * B_2 + 0 * B_3 + 0 * B_4 + 1 * B_5 + ... \geq 1$ where $B_1, B_2, ...$ are boolean variables in the ILP model). For E4, E5 and E7 subjects, a poor performance of the ILP model is due to the combination of bigger test suites and a larger number of test requirements.

In summary, the results of the experiment show that FLOWER consistently provides faster test suite reduction comparing to the ILP model. For the subjects E1-E8, FLOWER was able to compute the MCS for all subjects in less than 1.504 seconds, while the ILP model took ~700 seconds in the worst case. In particular, FLOWER showed to outperform the ILP model in terms of the time needed to find the solution by $15 - 3000$ times. These results show that FLOWER is time-efficient approach to test suite reduction.

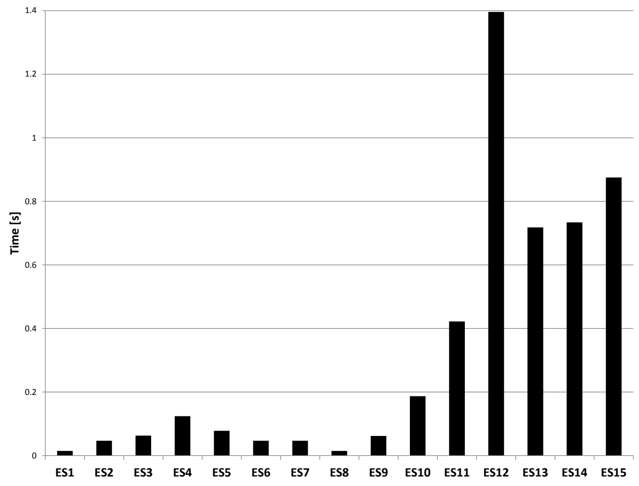

Figure 4: Relative difference in time between FLOWER and the greedy model.

5.3.2 *Size of the Reduced Test Suites*

To answer RQ2, we analysed the reduction rate for FLOWER, the ILP and greedy models, as the number of test cases eliminated from the test suite being reduced. We compared the reduction rate of these three approaches across 2000 experiment subjects (given in Table 2). The results showed that FLOWER and the ILP model achieved identical reduction rate for the considered subjects, while the greedy approach performed worse. Figure 5 presents the distribution of reduction rate in percentage for 7 sets of subjects E1-E7 for FLOWER and the greedy model, in the form of a boxplot. The subjects were grouped in 7 groups by the number of test cases and the number of test requirements per test case. In the boxplot, each ES group is represented with two boxes: the left box illustrates the percentage of reduction rate for FLOWER and the right box illustrates the same for the greedy model. As the results show, the difference in the reduction rate between FLOWER and the greedy approach is consistent across the considered experiment subjects and is within the range of 5% to 15%, to the advantage of FLOWER. This experimental study positively answers the RQ2. It demonstrates that FLOWER is effective in reducing the test suites, producing the size-equivalent solutions as the ILP model and consistently smaller solutions than the greedy model.

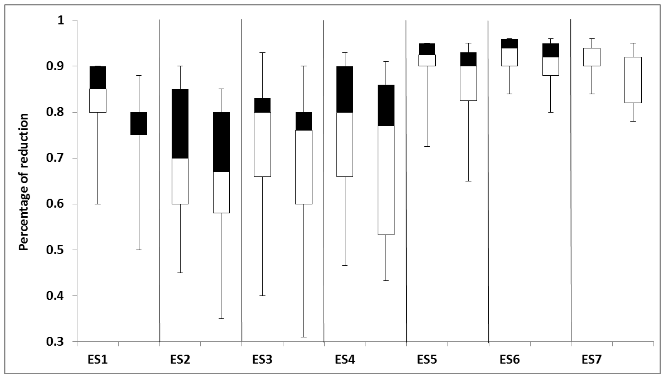

Figure 5: The distribution of the percentage of test suite reduction for FLOWER (left box) and the greedy model (right box).

5.3.3 *Real Use Cases*

We observed similar performance of FLOWER compared to the ILP and greedy models when applied in real uses cases, as when applied to the synthetic subjects. In terms of time needed to find the solution, FLOWER consistently takes 30% more time on average than the greedy model and outperforms the ILP model. The worst-case running time for all five experiment subjects for FLOWER is 0.109 seconds, and for the greedy model and the ILP model 0.05 seconds and 2.132 seconds respectively. Detailed experimental results are presented in Figure 6. In terms of ability to reduce the size of test suites, FLOWER performs exactly as the ILP model for all five subjects, while the greedy model gives from 10% to 15% bigger solution test suites than FLOWER and the ILP model. Figure 7 shows the experimental results.

5.4 Threats to Validity

A threat to external validity of our experimental results may be that the considered subjects are not sufficiently representative. To address this threat, we generated the subjects by varying some factors, e.g., the size of the test suite or the number of test requirements. In addition, we used 5 subjects from a real software application. A threat to conclusion validity may be the randomness in the subjects. To address this threat, we used a large number of subjects, which increases our confidence in the results. Another threat to validity of our experimental results relates our own implementations of both the ILP model and the greedy approach, which may not contain optimisation techniques used in other implementations. We intentionally compared FLOWER against the two other models, within a single and totally similar environment (i.e., SICStus Prolog) to allow fair comparison for all used randomly-generated subjects and real cases. In particular, subjects generated at random within two distinct environments, even if they share the same characteristics, can have different requirements coverage properties (different average number of covered requirements by a test case). In our case, the three approaches are evaluated on the same random samples. In addition, we observe that existing implementations of ILP or greedy methods use a combination of different techniques, such as preprocessing and heuristics, what makes a fair comparison with a new proposed technique more difficult. However, even if the results show improvements over our ILP and greedy approach implementations, we consider that these results cannot be generalized without comparison with more specialized and optimized versions of these approaches. More experiments with optimized implementations of ILP and greedy approaches are needed to confirm the results.

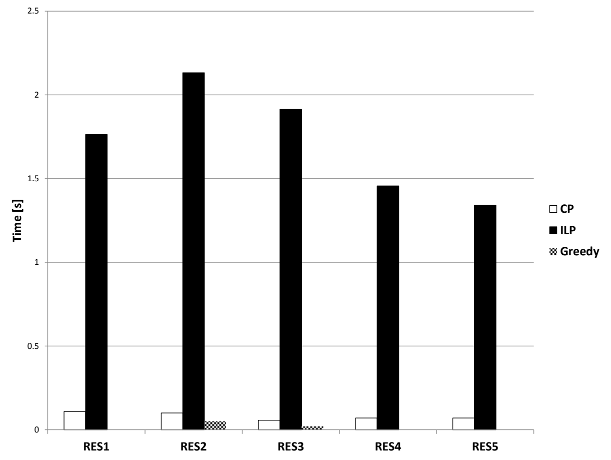

Figure 6: Comparison of time to compute solutions for FLOWER, the ILP and the greedy model.

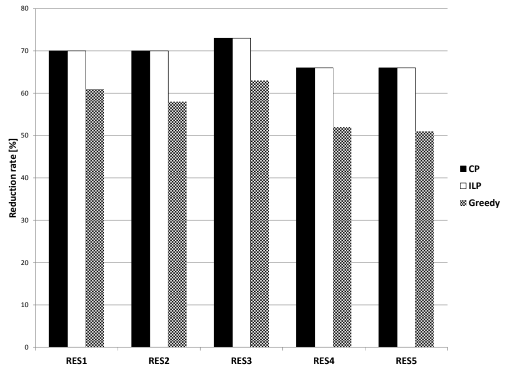

Figure 7: Percentage of a test-suite size reduction for FLOWER, the ILP and the greedy model.

6. RELATED WORK

Substantial research has been done to address the *test suite reduction problem.* In addition to this problem, authors have also considered test suite prioritization techniques in regression testing and experimental studies [10, 19, 24, 22, 30, 25]. Generally speaking, test-suite reduction techniques can be classified in two groups: greedy approaches [24, 13, 14] and exact methods [12, 2, 4, 9].

Greedy approaches. Early work on selecting test cases to cover test requirements is reported by Chvatal [5]. It is a classical approximation approach for the minimum set-cover problem based on a greedy heuristic. The heuristic suggests selecting first the test case that covers the most of uncovered requirements and repeats the process until all requirements are covered at least once. Cormen analysed and reported the upper bound on the distance to the true minimum given by this heuristic in the worst case [6]. The limitation of Chvatal's and similar greedy approaches is that the reduced test suite may contain redundant test cases. Harrold et al. proposed a technique that outperforms [5] in terms of solution size, by approximating the computation of minimum cardinality hitting sets [10]. However, the experimental evaluation confined to less than a few tens of test cases and requirements. A variation of the greedy approach to test suite reduction is presented in [19] where, instead of using a fixed test case ordering, the authors proposed different orderings. Agrawal specialized the technique to handle the coverage of statements and branches in the test suite [1]. Similarly, Marre and Bertolino proposed to formulate test suite minimisation as a problem of finding a spanning set over a graph [18]. In their approach, the SUT structure is represented by a decision-to-decision graph (ddgraph). The results of a data-flow analysis are mapped on the ddgraph for testing requirements and the test suite minimisation is treated as the problem of finding the minimal spanning set. However, these approaches are primarily based on a code-level structural coverage and exploit only the implications among coverage requirements. Tallam and Gupta presented a new test suite reduction greedy approach, called the Delayed-Greedy technique [24], which iteratively exploits implications among test cases and requirements to prune the minimized test suite. Jeffrey and Gupta extended this approach by retaining test cases which improve a fault-detection capability of the test suite [14]. Comparing to [10], the approach produces bigger solutions, but with higher fault detection effectiveness. The shortcoming of all these approaches is that they compute approximated solutions and cannot offer any guaranty on the minimality of the test suite.

Exact approaches. Black et al. were among the first to propose an exact technique able to compute true optimal solutions to the test suite reduction problem [2]. The approach is based on ILP and a bi-criteria decision making analysis. Similarly, another ILP-based approach was proposed, called MINTS, [12] that considers several criteria in test suite reduction. This technique extends [2] by comparing different heuristics for a multi-criteria ILP formulation, such as weighted sum, prioritized optimisation or hybridation. Nevertheless, the general limitation of ILP is the early exponential time blow-up to determine the true minimum, which exposes the technique to serious limitations even for small problems. In the context of software product lines optimization, an alternative approach has been proposed by Uzuncaova et al. in [25], using SAT or SMT solvers[1]. The general principle of this approach is to encode the test suite reduction problem as a boolean formula that can be evaluated and manipulated by the SAT or SMT solver. The considerable recent improvements made on these solvers makes this approach very promising. Similarly, the approach coined by Salecker et al. [23] encodes the problem with Binary Decision Diagrams and uses traditional transformation algorithms to find an optimal order over the variables to reduce the test suite. To compromise the shortcomings of ILP, Chen et al. proposed a degraded ILP approach, called DILP, that calculates a lower bound of a minimum test suite and then searches a small test suite close to the lower bound [4]. Hao proposes another ILP-based approach to test suite reduction that allows setting upper limits on the loss of fault-detection capability in test suite reduction [9]. In [26], the authors propose using search-based algorithms to find optimal solutions of a multi-objective test suite optimization problem. By comparing several algorithms for 5 different criteria, they observe that random-weighted multi-objective optimization is the most efficient approach. However, this approach assigns weights at random, meaning actually that no priority can be established between the criteria.

The test suite reduction approach presented in this paper is based on Network Flow Theory. To the best of our knowledge, the encoding of an optimal test suite reduction problem as a search problem among maximum flows in a network flow is novel. This is an exact approach computing minimum-size solutions to test suite reduction problems. We have not compared FLOWER with all the previously mentioned models and optimizations, but we compared it with an ILP model and a greedy approach, which are the most relevant techniques.

7. CONCLUSION

This paper introduces FLOWER, a novel approach to test-suite reduction based on network maximum flows. Given a test suite T and a set of test requirements R, FLOWER identifies a minimal set of test cases which maintains the coverage of test requirements. The approach encodes the problem with a bipartite directed graph and computes a minimum cardinality subset of T that covers R as a search among maximum flows, using the classical Ford-Fulkerson algorithm in combination with efficient constraint programming techniques. We evaluated FLOWER on a set of 2005 test suite reduction problems, by comparing it to our own implementations of an Integer Linear Programming (ILP) model and a simple greedy approach. The results demonstrate that FLOWER outperforms ILP by $15 - 3000$ times, in terms of the time needed to find the solution. At the same time, FLOWER obtains the same reduction rate as ILP, because both approaches compute optimal solutions. When compared to the simple greedy approach, FLOWER takes on average 30% more time and produces from 5% to 15% smaller test suites. These initial results show that FLOWER is a promising approach to test suite reduction for practical applications, but also to motivate further research. In future work, we plan to perform extensive empirical studies of FLOWER for multi-objective test suite reduction. We will consider fault detection capability of a test suite, test execution time, and costs associated with test execution setup as reduction objectives. We will perform the evaluation on case studies from industrial applications. We also plan to compare FLOWER to other existing solutions for optimal test suite reduction.

[1]SAT: Boolean Satisfability, SMT: Satisf. Modulo Theories

8. ACKNOWLEDGMENT

This work is supported by the Research Council of Norway (RCN) through the research-based innovation center Certus, under the SFI programme. We are indubitably thankful to Jean-Charles Regin for fruitful discussions on the topic of the paper. His careful review of our constraint model and its results helped us to understand the reasons of good performances of FLOWER with respect to the other approaches. We also thank the anonymous reviewers for their helpful remarks on the preliminary version of this paper.

9. REFERENCES

[1] H. Agrawal. Efficient coverage testing using global dominator graphs. In *Workshop on Program Analysis for Software Tools and Eng. (PASTE'99)*, 1999.

[2] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *26th Int. Conf. on Software Eng.*, pages 106–115, 2004.

[3] M. Carlsson, G. Ottosson, and B. Carlson. An open–ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, 1997.

[4] Z. Chen, X. Zhang, and B. Xu. A degraded ilp approach for test suite reduction. In *20th Int. Conf. on Soft. Eng. and Know. Eng.*, 2008.

[5] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. of Operations Research*, 4(3), 1979.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2 edition, 2001.

[7] L. Ford and D. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., NY, 1990.

[9] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Int. Conference on Software Engineerig*, pages 738–748, 2012.

[10] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM TOSEM*, 2(3):270–285, 1993.

[11] C. Holzbaur. *OEFAI clp(q,r) Manual Rev. 1.3.2.* Austrian Research Institute for Artificial Intelligence, Vienna, AU, 1995. TR-95-09.

[12] H.-Y. Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. In *31st Int. Conf. on Soft. Eng. (ICSE'09)*, pages 419–429, 2009.

[13] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. In *21st International Conference on Software Maintenance*, pages 549–558, 2005.

[14] D. Jeffrey and R. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Trans. on Soft. Eng.*, 33(2):108–123, 2007.

[15] J. Jones and M. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[16] C.-T. Lin, K.-W. Tang, C.-D. Chen, and G. Kapfhammer. Reducing the cost of regression testing by identifying irreplaceable test cases. In *6th International Conference on Genetic and Evolutionary Comp. (ICGEC)*, pages 257–260, 2012.

[17] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 540–543, 2013.

[18] M. Marre and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. on Soft. Eng.*, 29(11):974–984, 2003.

[19] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *12th Int. Conf. on Testing Computer Soft.*, 1995.

[20] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *13th Int. Conf. on Artificial Intelligence (AAAI'96)*, pages 209–215, 1996.

[21] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *International Conference on Software Maintenance (ICSM'98)*, pages 34–43, 1998.

[22] G. Rothermel, M. J. Harrold, J. Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing Verification and Reliability*, 12:219–249, 2002.

[23] E. Salecker, R. Reicherdt, and S. Glesner. Calculating prioritized interaction test sets with constraints using binary decision diagrams. In *Variability-intensive System Testing Workshop (VAST'11)*, 2011.

[24] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *6th Workshop on Program Analysis for Software Tools and Eng. (PASTE'05)*, pages 35–42, 2005.

[25] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309–322, 2010.

[26] S. Wang, S. Ali, and A. Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. In *Genetic and Evolutionary Computation Conference (GECCO'13)*, 2013.

[27] W. Wong, J. Horgan, A. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. In *21st COMPuter Soft. and App. Conf. (COMPSAC '97)*, pages 522–528, 1997.

[28] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th International Conference on Software Engineering (ICSE'95)*, 1995.

[29] M. C. X. Qu and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *International Symphosium on Software Testing and Analysis*, 2008.

[30] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *30th International Conference on Software Engineering (ICSE'08)*, pages 201–210, 2008.

[31] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.