MINTS: A General Framework and Tool for Supporting Test-suite Minimization

Hwa-You Hsu and Alessandro Orso
College of Computing - Georgia Institute of Technology
{hsu|orso}@cc.gatech.edu

Abstract

Test-suite minimization techniques aim to eliminate redundant test cases from a test-suite based on some criteria, such as coverage or fault-detection capability. Most existing test-suite minimization techniques have two main limitations: they perform minimization based on a single criterion and produce suboptimal solutions. In this paper, we propose a test-suite minimization framework that overcomes these limitations by allowing testers to (1) easily encode a wide spectrum of test-suite minimization problems, (2) handle problems that involve any number of criteria, and (3) compute optimal solutions by leveraging modern integer linear programming solvers. We implemented our framework in a tool, called MINTS*, that is freely-available and can be interfaced with a number of different state-of-the-art solvers. Our empirical evaluation shows that* MINTS *can be used to instantiate a number of different test-suite minimization problems and efficiently find an optimal solution for such problems using different solvers.*

1 Introduction

When developing and evolving a software system, a common practice is to build and maintain a *regression test suite*, a test suite that can be used to perform regression testing of the software after it is changed. Regression test suites are an important artifact of the software development process and, just like other artifacts, must be maintained throughout the lifetime of a software product. In particular, testers often add to such suites test cases that exercise new behaviors or target newly-discovered faults. As a result, test suites tend to grow in size over time and may become too large to be run in their entirety [10].

In some cases, the size of a test suite is not an issue (*e.g.,* when all test cases can be run quickly and in a fully automated way). In other cases, however, having a large test suite can make regression testing impractical. This is the case, for instance, for large test suites that requires manual work to be run (*e.g.,* to check the outcome of the test cases or setup some machinery).

Researchers have proposed several approaches to address this issue and either select, prioritize, or minimize the test cases in a regression test suite (*e.g.,* [7, 13–18, 20]). *Test-suite minimization*, in particular, aims to reduce the size of a test suite according to some criteria specified by the testers. For example, a tester may want to generate a test suite with maximal coverage, maximal fault detection capability, and minimal setup cost. Due to the computational complexity of multi-criteria minimization, however, most existing techniques target a much simpler version of the problem: generating a test suite that achieves the same coverage as the original test suite with a minimal number of test cases (*e.g.,* [7, 14, 17, 20]).

Although these techniques work well for the simpler problem they address, they are likely to generate test suites that are suboptimal with respect to other criteria. Previous research has shown, for instance, that the error-revealing power of a minimized test suite can be considerably less than that of the original test suite [14, 20]. Furthermore, because even single-criterion versions of the minimization problem are NP-Complete (see Section 2.2), most existing techniques are based on heuristics and approximated algorithms; they trade accuracy for efficiency and compute solutions that are suboptimal even for the simplified version of the minimization problem they target.

To address the shortcomings of existing techniques, we propose a test-suite minimization framework that has three main advantages over previous minimization approaches. *First*, it allows for easily encoding a wide range of test-suite minimization problems. *Second*, it can accommodate minimization problems that involve multiple criteria and, thus, allow testers to encode all of the constraints that they perceive as relevant. *Third*, it can produce solutions for test-suite minimization problems that are optimal with respect to all criteria involved.

Intuitively, our approach is based on (1) providing testers with a flexible way of specifying multi-criteria test-suite minimization problems and (2) encoding such problems and related criteria as binary integer linear programming (ILP) problems [6]. We defined our encoding so that it can be fed directly to one or more ILP solvers. If the solvers are able to compute a solution for the problem, such a solution corresponds to the minimized test suite that satisfies all of the considered criteria. To the best of our knowledge, the only

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

other technique that can handle more than one criterion and compute optimal solutions is the one proposed by Black and colleagues [4]. However, their technique is limited to two criteria and cannot be straightforwardly extended to a larger number of criteria. Moreover, our technique gives testers more expressiveness in defining and combining their minimization criteria.

Our framework and approach are implemented in a tool called MINTS (MINimizer for Test Suites), whose modular architecture allows for plugging in different ILP solvers. More precisely, MINTS can be transparently interfaced with any binary ILP solver that complies with the format used in the Pseudo Boolean Evaluation 2007 [12].

Using MINTS, we performed an empirical evaluation of our approach on a set of real programs (and corresponding test suites) and for a number of test-suite minimization problems; we used MINTS to find an optimal solution for the considered problems leveraging different state-of-the-art solvers, including both SAT-based pseudo-Boolean solvers and simplex-based linear programming solvers, and assessed its performance. The results of our evaluation provide initial evidence that the approach is practical and effective: for all problems considered, MINTS was able to compute an optimal solution in just a few seconds.

This paper provides the following contributions:

- A general test-suite minimization framework that handles minimization problems involving any number of criteria and can produce optimal solutions to such problems.
- A prototype tool that implements the framework, can interface seamlessly with a number of different ILP solvers, and is freely available.
- An empirical study in which we evaluate the approach using a wide range of programs, test cases, minimization problems, and solvers.

2 Motivating scenario

In this section, we introduce a motivating example consisting of a typical test-suite minimization scenario and discuss the limitations of existing approaches in handling such a scenario. We also use the example in the rest of the paper to illustrate our approach.

2.1 Test-suite minimization scenario

Consider a program P and an associated regression test suite $T = \{t_i\}$. Assume that the team in charge of testing P decides that T has become too large and wants to produce a minimized test-suite $MT \subseteq T$. As it is typically the case, we also have a set R of testing requirements for P. Whether the requirements in R are expressed in terms of code coverage, functionality coverage, or coverage of some other entity of relevance for the testing team is inconsequential for our approach. What matters is that T achieves some coverage of R. For this example, we assume to have a single set of requirements expressed in terms of statement coverage.

Assume also that the testing team is interested in three additional aspects of the test suite: (1) the total time required to execute MT; (2) the setup effort involved in running MT (*e.g.,* the number of man-hours required to set up an emulator for missing parts of the system under test); and (3) the (estimated) fault-detection capability of MT. (A common way to compute this value is to associate with each test case a fault-detection index based on historical data, that is, based on how many unique faults were revealed by that test case in previous versions of P.) In our scenario, the testing team's goal is to produce a test suite MT that maintains the same coverage as the original test suite T, minimizes the total time and setup effort, and maximizes the likelihood of fault detection.

Our example contains all of the elements of a typical minimization problem: a set of test-related data and a set of minimization criteria defined by the testing team. We list such elements and make the example more concrete by instantiating it with a specific set of values, as follows:

Test-related data

- The test suite to minimize: $T = \{t_1, t_2, t_3, t_4\}$
- The set of requirements: $R = \{r_1, r_2, r_3\}$
- Coverage, cost, and fault-detection data:

	t_1	t_2	t_3	t_4
$stmt_1$	×		×	
$stmt_2$	×	×		
$stmt_3$			×	×
Time to run	22	4	16	2
Setup effort	3	0	11	9
Fault detection	8	4	10	2

Minimization criteria

- Criterion #1: maintain statement coverage
- Criterion #2: minimize time to run
- Criterion #3: minimize setup effort
- Criterion #4: maximize expected fault-detection

As the data shows, T contains four test cases, and R contains three requirements (for the sake of space, we consider a trivial program with three statements only). The table provides information on statement coverage, running time, setup effort, and fault-detection ability for each test case in the test suite. For example, test case t_1 covers statements $stmt_1$ and $stmt_2$, takes 22 seconds to execute, has a setup cost of three (*i.e.,* it takes three man-hours to setup), and has a fault-detection ability of eight (*i.e.,* it revealed eight unique faults in previous versions of P). The four criteria define the constraints for the minimization problem.

2.2 Complexity of minimization problems

Minimization problems are NP-complete because they are in NP and there is a polynomial-time reduction from

the minimum set-cover problem to such problems [17]. We illustrate this point for single-criterion minimization problems—minimization problems that involve only one set of requirements and one criterion. Consider, for instance, a typical minimization problem whose goal is to produce a test suite that contains the minimal number of test cases that achieve the same coverage as the complete test suite. Let us define $cr(t)$ as the set of requirements covered by test case t (*i.e.,* $cr(t) = \{r \in R \mid t \text{ covers } r\}$) and $CovReq$ as the set of all requirements covered by T (*i.e.,* $CovReq = \{r \in R \mid \exists t \in T,\ r \in cr(t)\}$), with $CovReq \subseteq R$. By definition, for each $t \in T$, $cr(t) \subset CovReq$. Therefore, the solution of the test-suite minimization problem is exactly a minimum set cover for $CovReq$—a subset S of $\{cr(t) \mid t \in T\}$ such that (1) every element in $CovReq$ belongs to at least one of the sets in S and (2) $|S|$ is minimal. Multi-criteria minimization problems can be reduced from the set-cover problem in a similar fashion.

As we stated in the Introduction, due to the complexity of the test-suite minimization problem most existing minimization techniques focus exclusively on single-criterion minimization problems (*e.g.,* [7, 14, 17, 20]). By doing so, these techniques force testers to disregard important dimensions of the problem and are likely to generate test suites that are suboptimal with respect to such dimensions. For instance, they may generate test suites that contain a minimum number of test cases but have a longer running time than other possible minimal test suites. Or they may generate test suites that have minimal running time but have a considerably reduced fault detection ability [14, 20]. Moreover, because even single-criterion problems are NP-complete, as we demonstrated above, most of these existing techniques are based on approximated algorithms that make the problem tractable at the cost of computing suboptimal solutions.

To allow testers to compute minimized test suites that are optimal with respect to all of the parameters they consider relevant, we propose a general framework for encoding and solving test-suite minimization problems. In the next two sections, we first discuss the state of the art and then introduce our approach.

3 Related work

For efficiently computing near-optimal solutions to the single-criterion test-suite minimization problem, several heuristics have been proposed. Chavatal [5] proposes the use of a greedy heuristic that selects a test case that covers most yet-to-be-covered requirements until all requirements are satisfied. Harrold and colleagues [7] propose a similar, but improved heuristic that generates solutions that are always as good or better than the ones computed by Chavatal. Agrawal [1] and Marre and Bertolino [11] propose a different approach: they identify the set $R_s \subset R$ such that if every requirement in R_s is covered by a test suite, then every requirement in R will also be covered by that test suite. In [17], Tallam and Gupta classify these latter heuristics as *exploiting implications among coverage requirements (or attribute reductions)*, and Chavatal's and Harrold and colleagues' heuristics as *exploiting implications among test cases (or object reductions)*. They propose another heuristic, called Delay-Greedy, which combines the advantages of both types of heuristics. Delay-Greedy works in three phases: (1) apply object reductions (*i.e.,* remove test cases whose coverage of test requirements is subsumed by other test cases); (2) apply attribute reductions (*i.e.,* remove test requirements that are not in the minimal requirement set); and (3) build a reduced test suite from the remaining test cases using a greedy method. Their experiments show that the reduced test suites generated by Delay-Greedy are at least as good as the ones generated by previous approaches. All of these approaches suffer from both of the shortcomings that we discussed in the previous section: they focus on a single criterion and compute approximated solutions.

Both Rothermel and colleagues [14] and Wong and colleagues [20] empirically investigated the limitations of single-criterion minimization techniques. Specifically, they performed experiments to assess the effectiveness of minimized test suites in terms of their fault-detection ability. Their results show that the minimized test suites generated using a single-criterion technique may detect considerably fewer faults than complete test suites.

The approach by Jeffrey and Gupta [8] addresses the limitations of traditional single-criterion minimization techniques by considering multiple sets of testing requirements (*e.g.,* coverage of different entities) and introducing selective redundancy in minimized test suites. Although their approach improves on existing techniques, it is still heuristic. A better attempt at overcoming the limitations of existing approaches is the technique proposed by Black and colleagues [4], which consists of a two-criteria variant of traditional single-criterion test suite minimization approaches and computes optimal solutions using an integer linear programming solver. Also in this case, the approach can handle only limited kind of minimization problems. Our approach extends and generalizes these existing techniques so as to still be able to compute optimal solutions while letting testers specify (1) any number of minimization criteria and (2) how to combine, weight, or prioritize these criteria.

4 Our approach

4.1 Overview

Figure 1 provides a high-level view of our minimization framework as implemented in our MINTS tool. As the figure shows, MINTS takes as input a *test suite*, a set of *test-related data*, and a set of *minimization criteria* and produces a *minimized test suite*—a subset of the initial test suite computed according to the specified criteria.

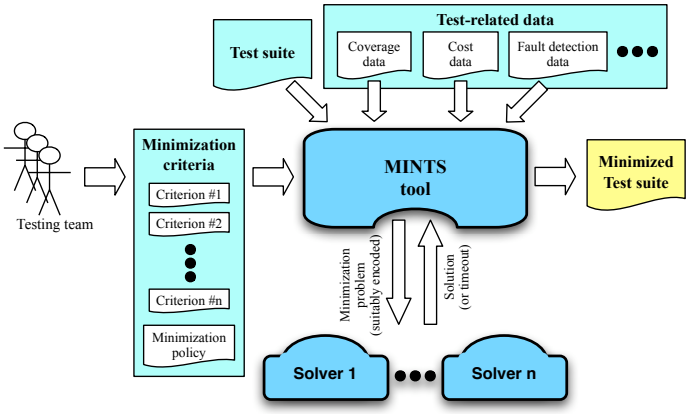

Figure 1. Overall view of our approach.

The set of test-related data can include many different types of data (*e.g.,* coverage data, various cost data, fault-detection data). In general, our framework allows testers to provide any set of data of interest to them. For example, testers may provide data about the last time each test case was run, information that could be used to favor the inclusion in the minimized test suite of test cases that have not been executed recently [9].

The minimization criteria are specified by the testers and consist of two main parts. The first part is a set of one or more criteria, each of which defines either a constraint or a sub-goal for the minimization (*e.g.,* minimizing the test-execution time). The second part is a minimization policy that specifies how the different sub-goals should be combined to find an optimal minimal test suite. Currently, our technique supports three different minimization policies: weighted, prioritized, and hybrid. (We describe these three policies in detail in Section 4.3.)

Our approach takes the set of criteria specified by the testers, combines them according to the associated minimization policy, and transforms them into a binary integer linear programming (ILP) problem. A *binary ILP problem* consists of optimizing a linear objective function in which all unknown variables can only have values 0 or 1 while satisfying a set of linear equality and inequality constraints [19]. Binary ILP problems are also called pseudo-Boolean problems [2]. We discuss our encoding approach in Section 4.4.

After encoding the minimization problem as a binary ILP problem, our approach feeds the resulting problem into one or more back-end ILP solvers. The solvers either return an optimal solution or stop after a given timeout. If an optimal solution is found, our approach reports to the testers the minimized test suite corresponding to that solution. Otherwise, it reports the partial results obtained by the solvers and notifies the user that no optimal solution was found. Although finding a solution to a binary ILP problem is an NP-complete problem, latest-generation ILP solvers have been successful in finding solutions to such problems efficiently, thanks to recent algorithmic advances and implementation improvements [12].

In the rest of this section, we first discuss the different types of minimization criteria supported within our framework; we then present the three minimization policies that our framework provides; finally, we illustrate in detail our approach for modeling such minimization criteria and policies as binary ILP problems and for computing optimal solutions to the minimization problem.

4.2 Minimization criteria

In our framework, each minimization criterion involves one set of test-related data and is one of two kinds: absolute or relative. An *absolute* criterion, such as Criterion #1 in our example, is one that introduces a constraint for the minimization problem. In this case, the set of test-related data involved is statement coverage data, and the constraint is that the minimized test suite must have the same coverage as the complete test suite. An absolute criterion corresponds to a linear constraint in a binary ILP problem. A *relative* criterion introduces an objective rather than a constraint for the minimization problem, such as Criteria #2, #3, and #4 in our example. In the case of Criteria #2, for instance, the set of test-related data involved is test timing data, and the objective is to minimize the testing time. A relative criterion corresponds to an objective function in a binary ILP problem.

Note that each type of test-related data set can be used in both relative and absolute criteria. For example, although coverage data are typically used in absolute criteria, nothing prevents testers from defining a relative criterion that introduces the maximization of coverage as an objective. Analogously, timing data could be used to define an absolute criterion in cases where amount of time that can be allocated to the testing process is limited.

4.3 Minimization policies

When performing multi-criteria test suite minimization, it is typically the case that more than one criterion is a relative criterion and, thus, more than one objective is specified. In these cases, testers can define how the different objectives should be combined by specifying a *minimization policy*. Our framework provides three different minimization policies: weighted, prioritized, and hybrid.

The *weighted* minimization policy allows testers to associate a relative weight to each objective. Such weight defines the extent to which that specific objective will affect the solution and lets testers put different emphasis on different criteria based on their perceived importance. We use our example to illustrate. Assume that, for instance, the testers have very limited man-power and are thus mostly concerned with reducing the setup effort. In such a case, they could assign a weight of .8 to Criterion #3 and a weight of .1 to

Criteria #2 and #4.[1] In this way, the solution will be skewed in favor of Criterion #3. However, test cases that are slightly worse according to Criterion #3, but considerably (an order of magnitude, in this case) better according to Criteria #2 and/or #4 may still be selected.

The *prioritized* minimization policy allows testers to specify in which order the different objectives in a minimization problem will be considered. Unlike the weighted policy, in which all objectives are weighted differently but considered at once, the prioritized policy considers one objective at a time. Intuitively, a prioritized policy first computes the set S_1 of optimal solutions for the objective with the highest priority. Then, if S_1 is not empty, it computes the set of optimal solutions $S_2 \subseteq S_1$ for the objective with the second highest priority. The process continues until all of the objectives have been considered. Considering again our example, a tester may assign Criterion #3 priority one, Criterion #2 priority two, and Criterion #4 priority three. In that case, our technique would first try to compute the set of solutions S_1 that minimize the setup effort while providing the same level of coverage as the complete test suite, then compute subset S_2 of S_1 with minimal testing time, and finally compute the subset S_3 of S_2 with maximal fault detection. In this case, test cases that are worse according to Criterion #3 would never be selected over better test cases for that criterion, regardless of how much better they are according to Criteria #2 and/or #4.

Finally, the *hybrid* minimization policy combines the two former policies. Testers can divide objectives into groups, weigh the set of objectives within each group, and assign a priority for each group. In this case, our approach would consider each group of objectives a single objective function, given by the weighted combination of the objectives in the group, and process the different groups in order based on their assigned priority.

4.4 Modeling multi-criteria minimization as binary ILP problems

As we discussed in Section 4.1, our framework encodes test-suite minimization problems in terms of binary ILP problems and then leverages ILP solvers to compute an optimal solution to such problems. Test-suite minimization problems are amenable to being represented as binary ILP problems. (1) A minimized test suite MT for a test suite T can be encoded as a vector of binary values o, of size $|T|$, in which a 1 (resp., 0) at position i in the vector indicates that the i^{th} test case in T is (resp., is not) in MT; (2) minimizing $|MT|$ means minimizing the number of 1s in o and can be expressed as a binary integer linear objective function; and (3) each criterion can be presented as a linear equality or inequality constraint. More precisely, we represent inputs and outputs of our problem as follows.

[1]Without loss of generality, weights are normalized to 1 to make their relative nature explicit.

Test suite: $T = \{t_i\}$

Test-related data: Each type of test-related data set can be represented as a set of values associated with the test cases in T. Therefore, we represent test-related data as an $n \times |T|$ matrix, where n is the number of types of test-related data: $test\text{-}related\ data = \{d_{i,j}\},\ 1 < i \leq n,\ 1 < j \leq\ |T|$. We represent a single type x of test-related data (*i.e.,* a row in the $test\text{-}related\ data$ matrix) as a vector $d_x = \{d_{x,j}\},\ 1 < j \leq\ |T|$

Minimized test suite (output): $OUT = \{o_i\},\ 1< i < |T|$, in which $o_i = 1$ if test case t_i is in the minimized test suite, $o_i = 0$ otherwise.

Minimization criteria (absolute): We express an absolute criterion involving the i^{th} type of test-related data as a constraint in the form $\sum_{j=1}^{|T|} d_{i,j}o_j \oplus const$, in which $\oplus$ is one of the binary operators $<$, $\leq$, $=$, $\geq$, or $>$, and $const$ is a constant value.

Minimization criteria (relative): Similar to what we do for absolute criteria, we express a relative criterion involving the i^{th} type of test-related data as an objective that consists of either maximizing or minimizing the following expression: $\sum_{j=1}^{|T|} norm(d_{i,j})o_j$, in which $norm(d_{i,j})$ is the value of $d_{i,j}$ normalized such that $\sum_{j=1}^{|T|} norm(d_{i,j}) = 1$.

Minimization policies: The encoding of minimization policies is fairly straightforward. A weighted policy is expressed as a set of weights, $\{\alpha_i\}$, one for each relative minimization criterion. A prioritized policy is encoded as a function that maps each relative minimization criterion to an integer representing its priority. Finally, a hybrid policy is encoded as a partition of the relative minimization criteria plus a function that maps each set in the partition to an integer representing its priority.

We now discuss how this encoding lets us model the test-minimization problem as a set of pseudo-Boolean constraints. If the tester defines n relative minimization criteria involving test-related data d_{x_1} to d_{x_n}, specifies m absolute minimization criteria involving test-related data d_{y_1} to d_{y_m}, and uses a weighted policy, the resulting encoding is in the following form:[2]

minimize
$\sum_{i=1}^{n} \alpha_i \sum_{j=1}^{|T|} norm(d_{x_i,j})o_j$
under the constraints
$\sum_{j=1}^{|T|} d_{y_1,j}o_j \oplus const_1$
$\sum_{j=1}^{|T|} d_{y_2,j}o_j \oplus const_2$
...
$\sum_{j=1}^{|T|} d_{y_m,j}o_j \oplus const_m$

[2]Note that in the following formulation, α_i is positive or negative, depending on whether the corresponding criterion involves a minimization or a maximization, respectively.

This formulation expresses the minimization problem as an optimization problem in which the objective function is the expression to be minimized and is defined in terms of OUT—all other values (*i.e.,* $d_{i,j}$, α_i, and $const_i$) are known. This encoding can be fed into a binary ILP solver, which would try to find a solution consisting of a set of assignments of either 0 or 1 values to each $o_i \in OUT$. The set of test cases defined as $\{t_i \mid o_i = 1\}$ would then correspond to the optimal minimized test suite for the initial minimization problem.

In the case of a prioritized policy, the situation would be similar, but the solution would be computed in stages. More precisely, the formulation would consist of a list of objective functions, one for each relative criterion, to be considered in the order specified by the tester.

The first optimization would invoke the solver to **minimize** the first objective function,

$\sum_{j=1}^{|T|} norm(d_{x_1,j})o_j$, **under the constraints**

$\sum_{j=1}^{|T|} d_{y_1,j}o_j \oplus const_1$, ..., $\sum_{j=1}^{|T|} d_{y_m,j}o_j \oplus const_m$.

If the solver found a solution, our technique would then save the (minimal) value of $\sum_{j=1}^{|T|} norm(d_{x_1,j})o_j$, corresponding to the solution val_1.

Our technique would then perform a second invocation of the solver to **minimize** the second objective function,

$\sum_{j=1}^{|T|} norm(d_{x_2,j})o_j$, **under the constraints**

$\sum_{j=1}^{|T|} d_{y_1,j}o_j \oplus const_1$, ..., $\sum_{j=1}^{|T|} d_{y_m,j}o_j \oplus const_m$,
$\sum_{j=1}^{|T|} norm(d_{x_1,j})o_j = val_1$.

Notice how the set of constraints now includes an additional constraint that encodes the result of the first optimization. Intuitively, this corresponds to finding a solution for the second optimization problem only among the possible solutions for the first problem, as we discussed in Section 4.1. Again, our technique would then save the minimal value of the objective function corresponding to the solution found by the solver, if any, and use it to create an additional constraint. Our technique would continue in this way until either the solver cannot find a solution or the last optimization has been performed. At this point, a solution for the last optimization, in terms of values of elements of OUT, would correspond to the minimal test suite for the initial minimization problem.

The computation of a solution in the case of a hybrid policy derives directly from the previous two cases. The solution is computed in stages, as for the prioritized policy, but each objective function corresponds to a set in the partition of relative criteria and involves a set of weights for the relative criteria in the set.

To illustrate with a concrete example, we show how our approach would operate for the minimization scenario that we introduced in Section 2.1:

- $T = \{t_1, t_2, t_3, t_4\}$
- $Test\text{-}related\ data =$

1	0	1	0
1	1	0	0
0	0	1	1
22	4	16	2
3	0	11	9
8	4	10	2

- Criterion #1:
  $\sum_{j=1}^{4} d_{1,j}o_j = o_1 + o_3 \geq 1$
  $\sum_{j=1}^{4} d_{2,j}o_j = o_1 + o_2 \geq 1$
  $\sum_{j=1}^{4} d_{3,j}o_j = o_3 + o_4 \geq 1$
- Criterion #2:
  minimize $\sum_{j=1}^{4} norm(d_{3,j})o_j = .5o_1 + .1o_2 + .36o_3 + .04o_4$
- Criterion #3:
  minimize $\sum_{j=1}^{4} norm(d_{4,j})o_j = .13o_1 + .48o_3 + .39o_4$
- Criterion #4:
  maximize $\sum_{j=1}^{4} norm(d_{5,j})o_j = .3o_1 + .17o_2 + .42o_3 + .08o_4$

Given this encoding, if we consider the case of a tester who specifies a weighted minimization policy with weights 0.1, 0.8, and 0.1 for Criteria #2, #3, and #4, respectively, we obtain the following encoding:

minimize
$0.1(.5o_1 + .1o_2 + .36o_3 + .04o_4) + 0.8(.13o_1 + .48o_3 + .39o_4) - 0.1(.3o_1 + .17o_2 + .42o_3 + .08o_4)$
under the constraints
$o_1 + o_3 \geq 1$, $o_1 + o_2 \geq 1$, $o_3 + o_4 \geq 1$

This encoding can be fed into a binary ILP solver, and the solution to the problem, if one is found, would consist of a set of assignments of either 0 or 1 values to o_1, ..., o_4. Such a solution identifies a test suite that solves the minimization problem described in the scenario. The test suite, defined as $\{t_i \mid o_i = 1\}$, would be in this case test suite $\{t_2, t_3\}$.

5 Empirical evaluation

To assess the practicality of our approach, we performed an empirical evaluation involving multiple versions of several software subjects, a number of minimization problems, and several ILP solvers. In our evaluation, we investigated the following research questions:

RQ1: How often can MINTS find an optimal solution for a test-suite minimization problem in a reasonable time?

RQ2: How does the performance of MINTS compare with the performance of a heuristic approach?

RQ3: To what extent does the use of a specific solver affect the performance of the approach?

Sections 5.1 and 5.2 present the software subjects that we used in the study and our experimental setup. Section 5.3 illustrates and discusses our experimental results.

Table 1. Subject programs used in the empirical study.

Subject	Description	LOC	COV	# Test Cases	# Versions
tcas	Aircraft altitude separation monitor	173	72	1608	5
schedule2	Priority queue scheduler	307	146	2700	5
schedule	Priority queue scheduler	412	166	2650	5
tot_info	Information measure	406	136	1052	5
replace	String pattern match and replace	562	263	5542	5
print_tokens	Lexical analyzer	563	194	4130	5
print_tokens2	Lexical analyzer	570	197	4115	5
flex	Fast lexical analyzer generator	12421	567	548	5
LogicBlox	Sales prediction system	570595	29204	393	5
Eclipse	Java IDE	1892226	35903	3621	5

5.1 Experimental subjects

Table 1 provides summary information about our ten subject programs. For each program, the table includes a description, the program size in terms of non-comment lines of code, the number of lines of code covered by the program's test suite, the number of test cases in the program's test suite, and the number of faulty versions of the program that we considered. For all entries in Table 1, if the values differ in different versions of a subject, we use median values.

The first seven subjects, from print_tokens to tot_info, consist of the programs in the Siemens suite, which we downloaded from the software-artifact infrastructure repository (SIR) at UNL (`http://sir.unl.edu/php/index.php`). We selected these programs because they have been widely used in the testing literature and represent an almost de-facto standard benchmark. In addition, and most importantly, they are available together with extensive test suites and multiple versions. For each program, the set of versions includes a golden version and several faulty versions, each containing a single known fault. In our studies, we considered the last five versions of each program.

The programs in the Siemens suite, albeit commonly used, are small programs, ranging from 173 to 570 lines of code. Therefore, to increase the representativeness of our set of subjects, we included three additional programs with real faults: flex, LogicBlox, and Eclipse. Program flex, also available from SIR, contains several real faults that can be individually switched on or off. We chose to seed faults in flex in a way that mimics a realistic scenario, where new faults are introduced by revisions, and not all faults are fixed going from one version to the next. More precisely, we built five faulty versions of flex, *f1* through *f5*, containing ten, seven, five, three and one fault, respectively. Faults in version fn include both new faults and faults already present in version $f(n-1)$, as shown in Table 2. (In the table, faults are identified using a unique fault id.)

LogicBlox (`http://www.logicblox.com`) is a sales prediction system for retailers that consists of one million lines of source code written in different languages: C++, C#, Python, and Java. Developers of LogicBlox built an initial set of unit tests for the program while developing it and added new test cases to the test suite during maintenance.

Table 2. Faults seeded in the versions of flex.

Version	New Faults Id	Existing Faults Id
$f1$	2, 3, 6, 7, 8, 12, 14, 16, 17, 18	
$f2$	11, 15	3, 7, 8, 12, 14
$f3$	1, 4	3, 12, 15
$f4$	5	1 15
$f5$		1

For our experiments, we used five versions of the Java part of LogicBlox together with its developer-provided regression test suite.

Eclipse (`http://www.eclipse.org`) is a widely used and extensible IDE, under which different features are provided by means of plug-ins. Each release of Eclipse includes its core plug-ins and tests suites for such plug-ins. In our study, we used five versions of Eclipse core (from 3.0.1 to 3.1.2) and their test suites.

5.2 Experimental setup

Test-related data. In our studies, we considered test-related data similar to the ones that we used in our example of Section 2.1: code coverage, running time, and fault-detection ability. We collected these data by executing each version of each subject program against its complete test suite. To measure coverage, we used the GNU utility GCOV, for C programs, and COBERTURA (`http://cobertura.sourceforge.net`), for Java programs. To collect the execution time, we used the UNIX time utility. Finally, we gathered fault-detection data for version n of a program by identifying which test cases revealed at least one fault in version $n-1$ of that program. Because all programs come with a golden version, we could identify failures by simply comparing the output of the golden version with the output of a faulty version when they are executed against the same test case.

Minimization criteria. We consider one absolute minimization criterion and three relative minimization criteria. The absolute minimization criterion corresponds to Criterion #1 in our example: the minimized test suite should achieve the same code coverage as the complete test suite. The three relative minimization criteria consist of (1) minimizing the number of test cases in the test suite, (2) minimizing the execution time of the test suite, and (3) maximizing the number of test cases that are error revealing.

Minimization policies. We considered eight different minimization policies: seven weighted and one prioritized. The weighted policies consist of one policy in which all three relative minimization criteria are assigned the same weight and six policies in which the weights are 0.6, 0.3, and 0.1 and are assigned to the different criteria in turn. The prioritized policy orders the criteria by minimizing the size of test suite first, minimizing the execution time second, and maximizing the fault-detection capability last.

Solvers considered. For our experiments, we interfaced our MINTS tool with six different ILP solvers. Four solvers are SAT-based pseudo-Boolean solvers: BSOLO (http://sat.inesc-id.pt/bsolo), MINISAT+ (http://minisat.se/MiniSat+.html), OPBDP (http://www.mpi-inf.mpg.de/departments/d2/software/opbdp), and PBS4 (http://www.eecs.umich.edu/~faloul/Tools/pbs4). We chose this set of pseudo-Boolean solvers based on their performance in the Pseudo Boolean Evaluation 2007 [12]. The other two solvers, which are not based on a SAT engine, are CPLEX (http://www.ilog.com/products/cplex) and GLPPB (http://www.eecs.umich.edu/~faloul/Tools/pbs4). CPLEX is a generic solver for large linear programming problems that was also used in previous work [4], whereas GLPPB is a pure ILP solver. We ran all solvers except CPLEX on Linux, on a 3 Ghz Pentium 4 machine with 2 GB of RAM running RedHat Enterprise Linux 4. Because we have a Windows-only license for CPLEX, we ran it on a Windows XP machine with a 1.8 GHz Pentium 4 CPU and 1 GB of RAM.

Overall, our experiments involved 400 different minimization problems. For each of the problems, we provided input data to our MINTS tool, which encoded the data as a binary ILP problem (see Section 4.4) and fed the problem to the different solvers. To provide data to the solvers, MINTS used the OPB format [12]. (For CPLEX, which uses a proprietary format, we built a filter that transforms the OPB format into the format used by CPLEX.)

In a normal usage scenario, MINTS would submit the problem to all solvers and return a solution as soon as one of the solvers terminates—because the solvers compute only exact solutions, waiting for additional solvers to terminate would simply result in solutions equivalent to the one already obtained. For our experiments, however, to gather information about the performance of different solvers, we executed all solvers either to completion or until a four-hour time threshold was reached.

5.3 Results and discussion

RQ1 – How often can MINTS find an optimal solution to a test-suite minimization problem in a reasonable time?

To answer RQ1, we first analyzed the data collected in our experiments and determined how many of the 400 minimization problems considered MINTS was able to solve optimally (*i.e.,* at least one of the solvers was able to compute a solution). Next, we measured how long it took MINTS to compute such solutions. Figure 2 shows the results of this analysis using a bar chart with bars of alternating colors to improve readability. The bar chart contains a bar for each of the minimization problems considered, grouped by subject. Within subjects, the entries are in turn grouped by version: the first eight entries for a subject correspond to the results for the seven weighted policies plus the prioritized policy when applied to Version 1 of the subject; the second eight entries correspond to analogous results for Version 2; and so on. The height of a bar represents the amount of time that it took MINTS to compute a solution for the corresponding minimization problem. For example, it took MINTS slightly more than one second to solve the first minimization problem involving subject schedule.

In the figure, we ordered the subjects based on their complexity indicator, where the *complexity indicator* for subject s is computed as s's size multiplied by s's number of test cases. We define the complexity indicator this way because (1) the number of test cases for a subject defines the number of variables involved in the minimization problem, and (2) the size of the subject affects the number of constraints in the problem. Therefore, the product of these two values for a subject can be considered an indicator of the complexity of the minimization problems involving that subject.

As the results in Figure 2 show, MINTS was able to find an optimal solution for all of the minimization problems in at most forty seconds. Moreover, in most cases, the solution was computed in less than two seconds. Even for the most complex minimization problem—the ones involving Eclipse—MINTS computed a solution in less than ten seconds for most minimization problems.

We also observe that, for some subjects, some problems are solved either considerably faster or considerably slower than the other problems involving the same subject. Interestingly, we found that both cases correspond to minimization problems involving a prioritized policy. Our conjecture, partially confirmed by our investigation of a subset of these cases, is that this behavior is due to two conflicting factors. On the one hand, in the case of weighted policies, MINTS combine all criteria and then feeds the resulting combined criterion to the underlying solvers. The solvers are likely to take a longer time to solve this combined, more complex criterion than to solve any of the original single criteria. On the other hand, in the case of prioritized policies, MINTS finds optimal solutions for one criterion at a time, which involves multiple interactions with the underlying solvers (three interactions for the three-criteria minimization problems considered in our study). In other words, weighted policies involve a single optimization of a more complex problem, whereas prioritized policies involve several optimizations of simpler problems. The relative importance of these factors varies depending on the subject, and so prob-

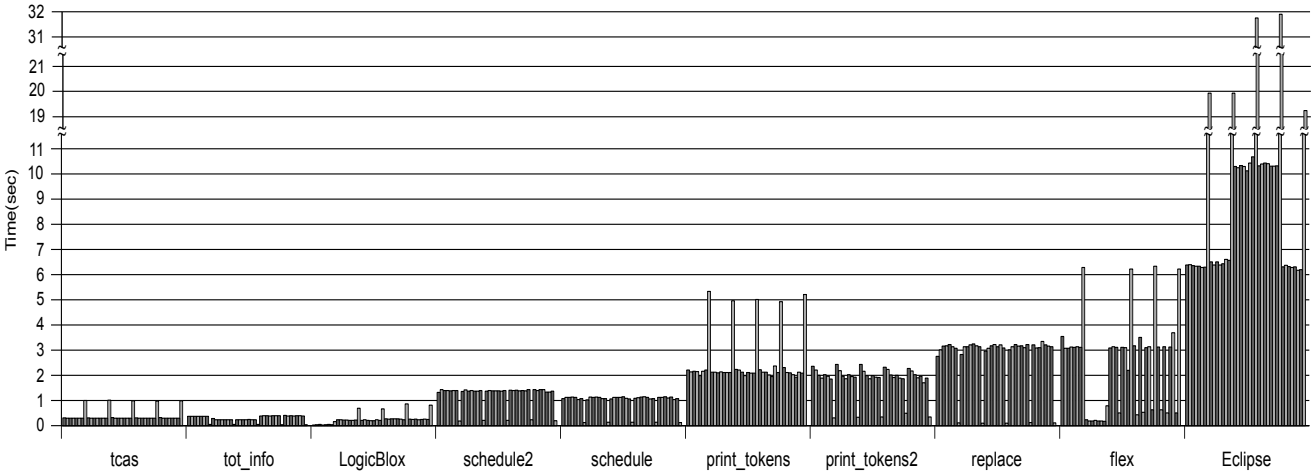

Figure 2. Timing results for MINTS when applied to the 400 minimization problems considered.

lems involving weighted criteria are solved faster than problems involving prioritized criteria for some subjects (*e.g.,* tcas), and slower for others (*e.g.,* tot_info).

Another set of results that deserve further investigation are the ones for subject flex. Whereas the performance of MINTS is fairly similar across the different versions of the seven Siemens programs, for flex we can observe a higher variability. In particular, the eight minimization problems involving the second version of the subject (eight to 15^{th} bars in the section of flex in the bar chart) were all solved in about 0.3 seconds, a much shorter time than that required for most of the other problems involving flex. A more in-depth analysis of the data revealed that the specific combination of faults in that version of flex caused an early termination of the program. Therefore, most test cases in the test suite of flex covered only the same few statements in that version, which resulted in a small number of constraints for the minimization and in an easy-to-find solution (because, from a coverage standpoint, most test cases are equivalent).

Finally, we observe that there seems to be a fairly strong correlation between the complexity of a subject, defined earlier in this section, and the time required to solve minimization problems involving that subject. This correlation can be observed by remembering that the subjects are ordered by increasing complexity index and by noting how the solution time for the subjects grows almost monotonically while going from left to right in the chart. We also note that, although the cost of the approach grows with the size of the problem, such growth appears to be almost linear, which is encouraging in terms of scalability of the approach.

We realize that using lines of code and number of test cases as a measure of complexity is a gross approximation, for several reasons. First, the number of constraints in the minimization problems we consider depends on the number of statements covered by the complete test suite, and not on the total number of statements, as demonstrated by the results for the second version of flex discussed above. Second, the characteristics of the test suites, such as the amount of redundancy among test cases, are likely to have a considerable effect on the results. Finally, the performance of ILP solvers depends on many characteristics of the optimization problem that go beyond the sheer size of the data sets [3]. Nevertheless, our results provide at least initial evidence that the approach can scale. Additional evidence is provided by results of the Pseudo Boolean Evaluation 2007 [12], in which some of the solvers involved were able to compute optimal solutions in a handful of minutes for problems with more than 170,000 constraints (which, in our context, corresponds to the number of requirements) and more than 75,000 variables (number of test cases, in our context).

In summary, although more empirical studies are needed to confirm our results, we believe that such results are promising, show the potential effectiveness and efficiency of our approach, and motivate further research.

RQ2: How does the performance of MINTS compare with the performance of a heuristic approach?

To answer RQ2, we cannot simply apply existing heuristic approaches to the 400 minimization problems considered in our study and evaluate their performance; such approaches, as we previously discussed, cannot handle multi-criteria minimization problems. We therefore first defined a set of simpler minimization problems analogous to the single-criteria ones used in previous work (*e.g.,* [7, 17]). Our problems consisted of computing, for all five versions of each of our subjects, a minimized test suite that maintains the same coverage level as the original test suite.

We then implemented the algorithm by Harrold, Gupta, and Soffa (HGS hereafter) [7] and compared the perfor-

Table 3. Sizes of minimized test suites generated by HGS and MINTS

LogicBlox version	Original test suite size	HGS	MINTS	Difference
v1	255	223	218	5
v2	393	392	390	2
v3	393	392	390	2
v4	395	394	392	2
v5	395	394	392	2
Eclipse version	Original test suite size	HGS93	MINTS	Difference
3.0.1	2460	656	418	238
3.0.2	2467	651	423	228
3.1	3621	851	553	298
3.1.1	3681	833	532	301
3.1.2	3681	656	406	250

mance of HGS and MINTS on this set of single-criterion minimization problems. We chose HGS as a representative of heuristic approaches because it is a well-known and commonly cited algorithm. Moreover, it is fairly simple to implement, which eases experimentation and reduces the risks of introducing errors in its implementation.

To compare the performance of the two techniques, we measured both the time necessary for MINTS and HGS to solve the problems and the size of the resulting minimized test suites. Both techniques were able to solve each problem in fractions of a second, with MINTS being faster than HGS in some cases. Given the imprecision of the UNIX time utility, and the fact that we are using our own implementation of HGS, we can consider the two techniques to have analogous performance with respect to time.

As far as the sizes of the minimized test suites are concerned, MINTS and HGS generate the minimized test suite of the same size for the the Siemens subjects and flex. In other words, the heuristic solutions happen to be optimal for these subject programs. For LogicBlox and Eclipse, however, MINTS always generates minimized test suite with smaller sizes than the ones generated by HGS. The difference is marginal in the case of LogicBlox, but considerable for Eclipse. Table 3 shows the results of the comparison in terms of sizes of the minimized test suites computed by MINTS and HGS for LogicBlox and Eclipse. (We do not report the results for the Siemens programs and flex because, as stated above, they are the same for the two techniques.)

Overall, these results show that, for the subjects and test suites considered, our approach performed as well or better than state-of-art heuristic technique. In addition, our technique can handle a wider range of minimization problems and computes optimal rather than approximated solutions.

RQ3: To what extent does the use of a specific solver affect the performance of the approach?

Because MINTS can feed each minimization problem to a number of solvers, the performance of the individual solvers is unimportant as long as at least one solver can compute a solution efficiently. However, assessing which ILP solvers are more suitable for test-suite minimization problems may provide useful insights for improving the approach and conducting future research. To gather this information, we examined the data produced by MINTS during the minimization process and identified how many times each solver was the fastest in producing a solution and how many times each solver reached our time threshold without producing a solution at all. This is information is indicated in Table 4.

Table 4. Performance of the different ILP solvers. Note for some problems, multiple solvers report solutions in the same time

	MINISAT+	GLPPB	OPBDP	BSOLO	PBS4	CPLEX
# times fastest	35	64	1	65	26	217
# times timed-out or crashed	119	33	70	22	102	0

As the table shows, the performance of the difference solvers varies considerably. Interestingly, most solvers produced the fastest solution in a number of cases but timed out in many other cases. A study of the finer-grained result data revealed that solvers tend to perform consistently across different versions of the same subject but may behave quite differently across subjects. Some solvers, such as GLPPB and CPLEX, performed extremely well for all problems, with response times often in the single digits. BSOLO also performed fairly well in most cases, although it was not able to complete within the time limit for some of the problems involving a prioritized policy. The performance of MINISAT+, PBS4, and OPBDP was in many cases disappointing in that they did not terminate before the time limit for minimizations that were completed in a few seconds by other solvers. Moreover, OPBDP crashed in some of the minimization problems regarding Eclipse test suite.

As discussed in [3], pseudo-Boolean solvers use different techniques to prune the search space, which can be more or less appropriate for a specific problem. Since the best performing solvers for our problem—GLPPB, CPLEX, and BSOLO—all utilize cutting-planes approaches, we conjecture that such approaches are more suitable to the characteristics of the test-suite minimization problem than the approaches based purely on SAT solving used in most other pseudo-Boolean solvers.

Further analysis of the performance of the various ILP solvers is beyond the scope of this paper and could represent interesting future work. As far as this work is concerned, our results provide evidence that, although the performance of the different solvers varies across subjects, a test-suite minimization approach that relies on ILP solvers can be practical, especially if it can leverage several solvers in parallel, as MINTS does.

Threats to validity. The main threat to the external validity of our results is the fact that we considered only ten applications and related test suites; experiments with additional subjects may generate different results. However, most of the applications we used in our evaluation are real programs, with real test cases, coming from different sources, and used in many previous studies. Threats to internal validity involve possible faults in the implementation of our tool or of the underlying solvers. To mitigate this threat, we spot checked a large number of results and carefully examined results obtained on a set of test programs.

6 Conclusion and future work

Test-suite minimization techniques can reduce the cost of regression testing by eliminating redundant test cases from a test suite based on some criteria. Unfortunately, test-suite minimization is an NP-complete problem, so most existing techniques (1) target simpler versions of the minimization problem and (2) are based on heuristic algorithms that compute approximated, suboptimal solutions. To address these limitations of existing techniques, we proposed a framework that lets testers specify a wide range of multi-criteria test-suite minimization problems and computes optimal solutions for such problems by encoding them as binary ILP problems and leveraging modern ILP solvers. We also presented a tool, MINTS, that implements our approach and is available for download (`http://www.cc.gatech.edu/~orso/software.html`). Finally, we presented a set of empirical results that show that our approach is practical and effective. Using MINTS, we were able to compute optimal solutions for 400 minimization problems involving eight different subjects. Our results also show that, for the cases considered, our approach can be as efficient as heuristic approaches while computing optimal solutions.

We are currently considering several directions for future work. First, we will perform additional empirical studies with more subjects to further assess the scalability of our approach. It is worth noting that, due to the intimate connection between the characteristics of a minimization problem and the performance of a solver on that problem, evaluating our approach using randomly generated large data sets would be unlikely to provide any meaningful information. Instead, we will collect larger programs together with test cases and test-related data and replicate our experiments on such subject programs and data. Our current results are promising in terms of scalability, and we hope to confirm these results in future studies.

We will also further analyze our results to get more insights on the reasons for the variance in the performance of different solvers. We believe that a deeper understanding of this issue may help improve our approach and possibly provide interesting data for the developers of such solvers.

Another direction for future work is the investigation of ways to extend our approach to test-case prioritization—the identification of the ordering of test cases in a test suite that maximizes their likelihood of detecting faults early. In our preliminary investigation we discovered that, because of the NP-hard nature of prioritization problems, straightforward extensions of our approach would not work in this context, and more sophisticated (or alternative) approaches are needed.

Acknowledgments

We thank LogicBlox for providing access to their code base. This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

References

[1] H. Agrawal. Efficient coverage testing using global dominator graphs. In *Proceedings of PASTE 99*, pages 11–20, 1999.

[2] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

[3] D. L. Berre and A. Parrain. On extending SAT solvers for PB problems. In *RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, July 2007.

[4] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE 04*, pages 106–115, May 2004.

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

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

[7] 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, July 1993.

[8] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE TSE*, 33(2):108–123, 2007.

[9] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE 02)*, pages 119–129, 2002.

[10] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *ICSM 91*, pages 201–208, October 1991.

[11] M. Marre and A. Bertolino. Using spanning set for coverage testing. *IEEE TSE*, 29(11):974–984, November 2003.

[12] Pseudo-boolean evaluation 2007. `http://www.cril.univ-artois.fr/PB07/`, 2007.

[13] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, April 1997.

[14] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault-detection capabilities of test suites. In *ICSM 98*, pages 34–43, November 1998.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test Case Prioritization. *IEEE TSE*, 27(10):929–948, October 2001.

[16] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA 02*, pages 97–106, July 2002.

[17] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *PASTE 05*, pages 35–42, September 2005.

[18] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *Int'l Conf on Reliability, Quality and Safety of Software-intensive Systems*, pages 3–21, May 1997.

[19] H. P. Williams. *Model Building in Mathematical Programming*. John Wiley, 1993.

[20] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *ICSE 95*, pages 41–50, April 1995.