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

Regression test suites tend to grow over time as new test cases are added to exercise new functionality or to target newly-discovered faults. When test suites become too large, they can be difficult to manage and expensive to run, especially when they involve complicated machinery or manual effort. Test-suite minimization techniques address this issue by eliminating redundant test cases from a test suite based on some criteria, while trying to maintain the overall effectiveness of the reduced test suite. Most minimization techniques proposed to date have two main limitations: they perform minimization based on a single criterion and produce approximated suboptimal solution. In this paper, we propose a test-suite minimization framework that overcomes these limitations. Our framework allows for (1) easily encoding a wide spectrum of test-suite minimization problems, (2) handling problems that involve any number of criteria, and (3) computing optimal solutions to minimization problems by leveraging modern integer linear programming solvers. We implemented our framework in a tool called MINTS, which is freely-available and can be interfaced with a number of different state-of-the-art solvers. In our empirical evaluation, we show how 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, it is common practice 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, during maintenance, test suites tend to grow in size, to the point that they may become too large to be run in their entirety [17,18]. In some scenarios, the size of a test suite is not an issue. This is the case, for instance, when all test cases can be run quickly and in a completely automated way. In other scenarios, however, having too many test cases to run can make regression testing impractical. For example, for a test suite that requires human intervention (*e.g.,* to check the outcome of the test cases or setup some machinery), executing all test cases could be prohibitively expensive. Another example is the case of cooperative environments where developers run automated regression test suites before committing their changes to a repository. In these cases, reducing the number of test cases to rerun may result in early availability of updated code and improve the overall efficiency of the development process.

To reduce the cost of regression testing, researchers have investigated several directions and proposed various complementary approaches. Regression test selection techniques identify test cases in a regression test suite that need not be rerun on the new version of the software (*e.g.,* [8, 23, 30]). Prioritization techniques, conversely, rank test cases in a test suite based on some criteria, so that test cases that are more effective according to such criteria would be run first (*e.g.,* [25, 28, 31]). Other techniques aim to identify, based on specific knowledge about a system such as historical data or static analysis results, where to focus the testing effort (*e.g.,* [5, 21]). This paper targets another technique proposed to limit the cost of regression testing and to improve the cost-effectiveness of regression test suites: test-suite minimization (also called test-suite reduction).

The goal of *test-suite minimization* techniques is to reduce the size of a test suite according to some criteria. Ideally, these techniques should let testers specify a set of relevant criteria and compute an optimal minimal test suite that satisfies such criteria. For example, a tester may want to generate a test suite with maximal coverage and fault detection capability and minimal running time and 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 the minimal number of test cases (*e.g.,* [12, 24, 29, 34]). 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 [24, 33]. 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. On the one hand, these approaches have the advantage of being practical and fairly efficient. On the other hand, they trade

accuracy for efficiency and compute solutions that are sub-optimal even for the simplified version of the minimization problem they target.

To address these 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, let testers encode all of the constraints that they perceive as relevant. *Third*, it can produce solutions for the test-suite minimization problems that are optimal with respect to all criteria involved, by leveraging Integer Linear Programming (ILP) solvers. Our approach is based on encoding the user-provided minimization problem and related criteria as binary ILP problems. 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 the considered criteria.

To the best of our knowledge, the only existing technique that considers more than one criteria and computes optimal solutions is the one proposed by Black and colleagues [7]. However, their technique is limited to two criteria and could not be easily extended to a larger number of criteria. Moreover, our technique gives testers more expressiveness in defining their minimization criteria and how to combine them.

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 [22].

We used MINTS to perform an empirical evaluation of our approach on a set of real programs (and corresponding test suites) and for several different test-suite minimization problems. In the evaluation, we assessed the performance of MINTS in finding an optimal solution for the considered problems when using different state-of-the-art solvers (including both SAT-based pseudo-boolean solvers and simplex-based linear programming solvers). The results of our study 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, a performance comparable to that of heuristic approaches.
This paper provides the following contributions:

- A test-suite minimization framework that is general, 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 out of the box with a number of ILP solvers, and is freely available.
- An empirical study in which we evaluate the practicality of the approach on a wide range of programs, test cases, minimization problems, and solvers.

The rest of the paper is organized as follows. Section 2 introduces a motivating example and provides background information. Section 3 discusses related work. In Section 4, we present our approach. Section 5 presents and discusses our empirical results. Finally, Section 6 concludes and sketches possible future research directions.

2. MOTIVATING SCENARIO

In this section, we introduce a motivating example that corresponds to 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

Let us assume that we have a program P and an associated regression test suite $T = \{t_i\}$, and that the team in charge of testing P decides that T grew too large and wants to minimize it to obtain 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.

Let us also assume that there are three additional parameters of interest for the testing team. The first parameter is the total time required to execute MT. The second parameter is the setup effort involved in running MT (*e.g.,* the number of man-hour required to set up an emulator for missing parts of the system under test). The final parameter of interest is 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.

At this point, 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 3 man-hour 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 test-suite minimization problem.

2.2 Complexity of The Minimization Problem

Minimization problems are NP-complete because they can be reduced to the minimum set-cover problem [10]. We illustrate this point for minimization problems that involve only one set of requirements and one criterion, also called single-criterion minimization problems. Consider, for instance, a typical minimization problem where the goal is to produce a test suite that contains the smallest possible 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.,* $cr(t) = \{r \in R \mid tcovers\ r\}$), 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 $t \in T$, $cr(t)$ is a subset of $CovReq$. 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 to the set-cover problem in a similar fashion.

As we stated in the Introduction, because of the complexity of the test-suite minimization problem, most existing minimization techniques focus exclusively on single-criterion minimization problems (*e.g.,* [12, 24, 29, 34]). By doing so, these techniques 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 are minimal in terms of running time but have a considerably reduced fault detection ability [24, 33]. 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 solutions that are suboptimal even for the simplified version of the minimization problem they target.

Without appropriate support, testers cannot fully leverage the information they possess about their test cases and their testing process for generating optimal minimal test suites. 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

Several heuristics have been proposed for efficiently computing near-optimal solutions to the single-criterion test-suite minimization problem. Chavatal [9] proposes the use of a greedy heuristic that selects test case that covers most yet-to-be-covered requirements, until all requirements are satisfied. When there is a tie between multiple test cases, one test case is randomly selected. Harrold and colleagues [12] 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 [20] propose a different approach. Their approach finds a minimal subset R_s of R, the set of all testing requirements, 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 [29], Tallam and Gupta classify these latter heuristics as *exploiting implications among coverage requirements (or attribute reductions)*, and Chvatal's and Harrold and colleagues' heuristics as *exploiting implications among test cases (or object reductions).* They propose another heuristic, called Delay-Greedy, that 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 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 generating 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.

Two studies performed by Rothermel and colleagues [24] and Wong and colleagues [33] investigated the limitations of single-criterion minimization techniques. Specifically, these studies performed experiments to assess the effectiveness of minimized test suites in terms of fault-detection ability. Their results showed that test suites minimized using a single-criterion minimization technique may be able to detect considerably less faults than complete test suites.

The approach by Jeffrey and Gupta [15] 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 the minimized test suites. Although their approach improves on existing techniques, it is still a heuristic approach. A better attempt at overcoming the limitations of existing approaches is the technique proposed by Black and colleagues [7], 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 is limited in the kinds of minimization problems it can handle. In defining our approach, we extend and generalize the technique proposed by Black and colleagues so as to still be able to compute optimal solutions while letting testers specify any number of minimization criteria and also how to combine, weight, or prioritize these criteria.

4. OPTIMAL TEST-SUITE MINIMIZATION 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.

The set of test related data can include a number of types of data, such as the ones used in our scenario (*e.g.,* coverage data, various cost data, fault-detection data). In general, our framework lets testers provide any set of data that is of interest for them. For example, testers may provide data about the last time each test case was run, which could be used to favor the inclusion in the minimized test suite of test cases that have not been executed recently [16].

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. Our current technique supports three different minimization policies: weighted, prioritized, and hybrid. (We describe these three policies in details 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 [32]. Binary ILP problems are also called pseudo-boolean problems [3]. We discuss our encoding approach in Section 4.4.

After encoding the minimization problem at hand as a binary ILP problem, our approach feeds the resulting problem to one or more back-end 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, last-generation ILP solvers have been successful in finding solutions to such problems efficiently, thanks to recent algorithmic advances and implementation improvements [22].

In the rest of this section, we discuss the different types of minimization criteria supported within our framework, present the three minimization policies that our framework provides, and 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 can be of one of two kinds: absolute or relative. An *absolute* criterion is one that introduces a constraint for the minimization problem, such as Criterion #1 in our example. 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 can be used in both relative and absolute criteria. For example, although coverage data is 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 there is a maximum amount of time that can be allocated to the testing process.

4.3 Minimization Policies

When performing a multi-criteria test suite minimization, it is typically the case that more than one criterion is a relative criterion and, thus, there is more than one objective 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 lets testers 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. For our example, for instance, the testers may have very limited man power and, therefore, be 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, where 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 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 specify that Criterion #3 has priority one, Criterion #2 has priority two, and Criterion #4 has 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

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

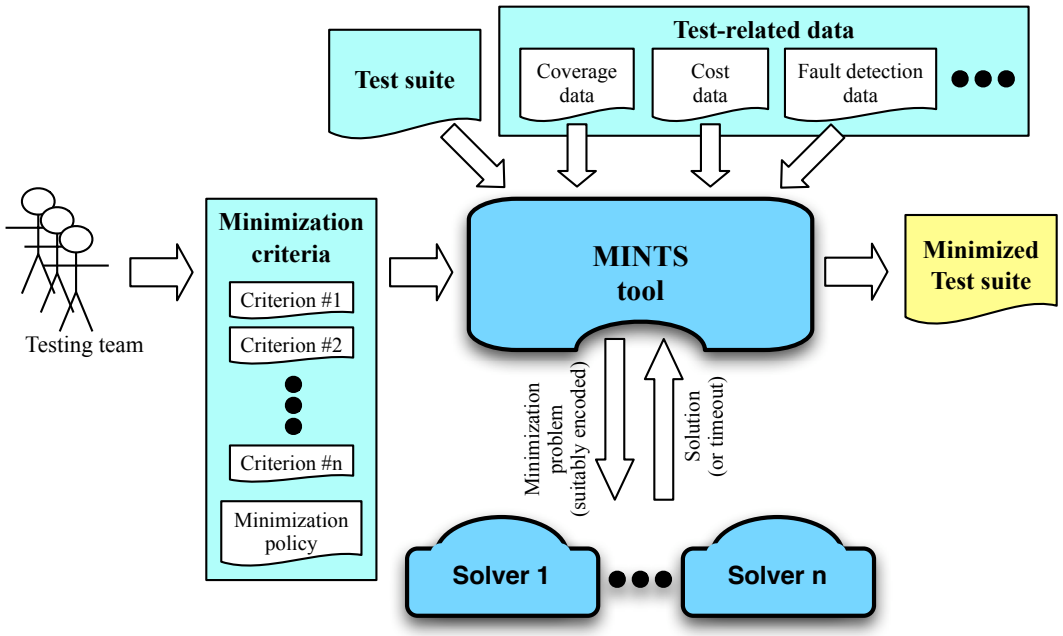

Figure 1: Overall view of our approach.

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, no matter 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 as 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 problem. (1) A minimized test suite MT for a test suite T can be encoded as a vector of binary values o, of size $|T|$, where 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.

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

Test related data: Each type of test-related data 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: $Test\ related\ data = \{d_{i,j}\},\ 1 < i \leq\ n,\ 1 < j \leq\ |T|$

We represent a single type x of test-related data (*i.e.,* a row in the $Test\ related\ data$ matrix) as vector $d_x = \{d_{x,j}\},\ 1 < j \leq\ |T|$

Minimized test suite (output): $OUT = \{o_i\}, 1 < i <\ |T|$, where $o_i = 1$ 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 $\sum_{j=1}^{|T|} d_{i,j}o_j \oplus const$, where $\oplus$ 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 in either maximizing or minimizing the following expression: $\sum_{j=1}^{|T|} norm(d_{i,j})o_j$, where $norm(d_{i,j})$ 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}, 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 form:[2]

[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.

minimize

$\sum_{i=1}^{n} \alpha_i \sum_{j=1}^{|T|} norm(d_{x_i,j})o_j$

under the constraints

$\sum_{j=1}^{|T|} d_{y_1,j}o_j \oplus const_1$
$\sum_{j=1}^{|T|} d_{y_2,j}o_j \oplus const_2$

...

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

This formulation expresses the minimization problem as an optimization problem where 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 to a binary ILP solver, which would try to find a solution consisting in 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 in 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.

The 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 solution 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, use it to create an additional constraint, and continue in this way until either the solver cannot find a solution, or the last optimization has been performed. At this point, the solution for the last optimization, in terms of values of OUT's elements, 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:

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

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 for the minimization problem:

minimize

$0.1(.5o_1 + .1o_2 + .36o_3 + .04o_4) + 0.8(.13o_1 + .48o_3 + .39o_4) - 0.4(.3o_1 + .17o_2 + .42o_3 + .08o_4)$

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 to 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 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 $\{t_2, t_3\}$.

Relation with Traditional Optimization Problems. We defined our three minimization policies based on our experience with and knowledge of test-suite minimization. Interestingly, they correspond to well-known types of multiple criteria optimization problems, as described by Yager [35]. Specifically, our weighted policy corresponds to the generalized Ordered Weight Aggregation (OWA) model, our prioritized policy corresponds to the hierarchically prioritized criteria, and our hybrid policy corresponds to the OWA prioritized criteria aggregation.

5. EMPIRICAL EVALUATION

To assess the practicality of our approach, we performed an extensive 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:

Table 1: Subject programs used in the empirical study.

Subject	Description	LOCs	# Test Cases	# Versions
tcas	Aircraft altitude separation monitor	173	1608	5
schedule2	Priority queue scheduler	307	2700	5
schedule	Priority queue scheduler	412	2650	5
tot_info	Information measure	406	1052	5
replace	String pattern match and replace	562	5542	5
print_tokens	Lexical analyzer	563	4130	5
print_tokens2	Lexical analyzer	570	4115	5
flex	Fast lexical analyzer generator	12421	548	5

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

RQ2: How does MINTS's performance 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.

5.1 Experimental Subjects

For our studies, we used eight subjects. Table 1 provides summary information about our subject programs. For each program, the table shows a description, the program size in terms of non-comment lines of code, the number of test cases available for the program, and the number of versions of the program that we considered.

The first seven subjects, from print_tokens to tot_info, consist of the programs in the Siemens suite [13]. 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 fairly small, with sizes that range from 173 to 570 lines of code. Therefore, to increase the representativeness of our set of subject programs, we included an additional program, flex, which we obtained from the Software-artifact Infrastructure Repository at UNL [27]. Program flex is also available in multiple versions, and each version contains several seeded faults that can be manually switched on or off individually. Because in flex we can easily seed multiple faults in a single version, we chose to seed faults 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. To do this, we built five faulty versions of flex $v2.4.7$, $f1$ through $f5$, containing ten, seven, five, three, and one faults, 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.)

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 data. We gathered these data by collecting measures while running each version of each subject against its complete test suite. To gather coverage data, we used GCOV, a GNU utility that can be used in conjunction with the GCC compiler to perform coverage analysis. We gathered the execution time using the UNIX time utility. Finally, we gathered fault-detection data to be used for a version n by identifying which test cases revealed at least one fault in version $n-1$. Because all programs come with a golden version, failures can be identified by simply comparing the output of the golden version with the output of a faulty version when run against the same test case.

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

Minimization criteria. The only absolute minimization criterion we considered in our experiments involves code coverage, which corresponds to Criterion #1 in our example. We specified that the minimized test suite should achieve the same code coverage as the complete test suite.

We also considered three relative minimization criteria: minimizing the number of test cases in the test suite, minimizing the execution time of the test suite, and maximizing the number of test cases that are error revealing.

Minimization policies. We considered eight different minimization policies: seven weighted policies and one prioritized policy. The weighted policies consist of one where all three relative minimization criteria are assigned the same weight and six where 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 as follows: minimizing the test suite's size first, minimizing execution time second, and maximizing fault-detection capability last.

Solvers considered. For our experiments, we interfaced our MINTS tool with six different ILP solvers. Four of these are SAT-based pseudo-boolean solvers: BSOLO [19], MINISAT+ [11], OPBDP [4], and PBS4 [2]. We chose this set of pseudo-boolean solvers based on their performance in the Pseudo Boolean Evaluation 2007 [22]. The other two solvers, which are not based on a SAT engine, are CPLEX [14] and GLPPB [26]. CPLEX is a generic solver for large linear programming problems that was also used in previous work [7], whereas GLPPB is a pure ILP solver. We ran all pseudo-boolean 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 320 different minimization problems. For each of the minimization problems, we provided the input data to our MINTS tool, which encoded the data as a binary ILP problem, as described in Section 4.4, and fed the problem to the different solvers. To provide data to the solvers, MINTS used the OPB format [22]. (For CPLEX, which uses a proprietary format, we built a filter that transforms the OPB format into CPLEX's format.)

In a normal usage scenario, MINTS would submit the problem to all solvers and return a solution as soon as one of the solvers terminate. Because the solvers only compute exact solutions, waiting for additional solvers to terminate would simply result in solutions that are equivalent to the one already obtained. For our experiments, however, we let all solvers run, either to completion or until a four-hour time threshold was reached, so as to gather information about the performance of the different solvers. We chose four hours as a threshold because we considered a possible scenario where minimization is performed overnight or between shifts.

5.3 Results and Discussion

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

To answer RQ1, we first analyzed the data collected in our experiments and checked for how many of the 320 minimization problems considered MINTS was able to compute an optimal minimized test suite (*i.e.,* at least one of the solvers was able to compute a solution). By looking at the data, we found that MINTS was able to compute a solution for all of the cases considered. Next, we measured how long it took MINTS to compute such a solution. Figure 2 shows the results of this analysis in the form of 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. 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.

Within subjects, the entries are in turn grouped by version, that is: 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. Finally, the subjects are ordered based on a *complexity indicator*, computed as their size multiplied by their 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.

Overall, the results in Figure 2 show that MINTS was able to find an optimal solution for all minimization problems in less than seven seconds, and in most cases the solution was computed in less than two seconds. Although these results may not generalize, they are nevertheless encouraging and show that our minimization approach is practical and efficient.

We also observe that there is a set of problems that 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 combines all criteria together 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 optimizations of a more complex problem, whereas prioritized policies involve several optimization of simpler problems. The relative importance of these factors varies depending on the subject, and so problems involving weighted criteria are solved faster than problems involving prioritized criteria for some subjects (*e.g.,* tcas), and vice versa (*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 flex's section of the bar chart) were all solved in about 0.3 seconds, which is 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 flex's test suite only covered 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 most test cases are equivalent from a coverage standpoint).

As a final observation about the results in Figure 2, we note that there seems to be a correlation between a subject's complexity indicator, 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 in the chart and by noting how the solution time for the subjects grows almost monotonically while going from left to right on the chart. Note, however, that although the cost of the approach grows with the size of the problem, the growth appears to be almost linear, which is encouraging in terms of scalability. For example, the average solution time for tcas, which has 173 lines of code and 1608 test cases, is around 0.5 seconds, whereas the average solution time for replace, which has 564 lines of code and 5542 test cases is around three seconds.

Using lines of code and number of test cases as a measure of complexity is obviously a gross approximation. First of all, 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 characteristic of

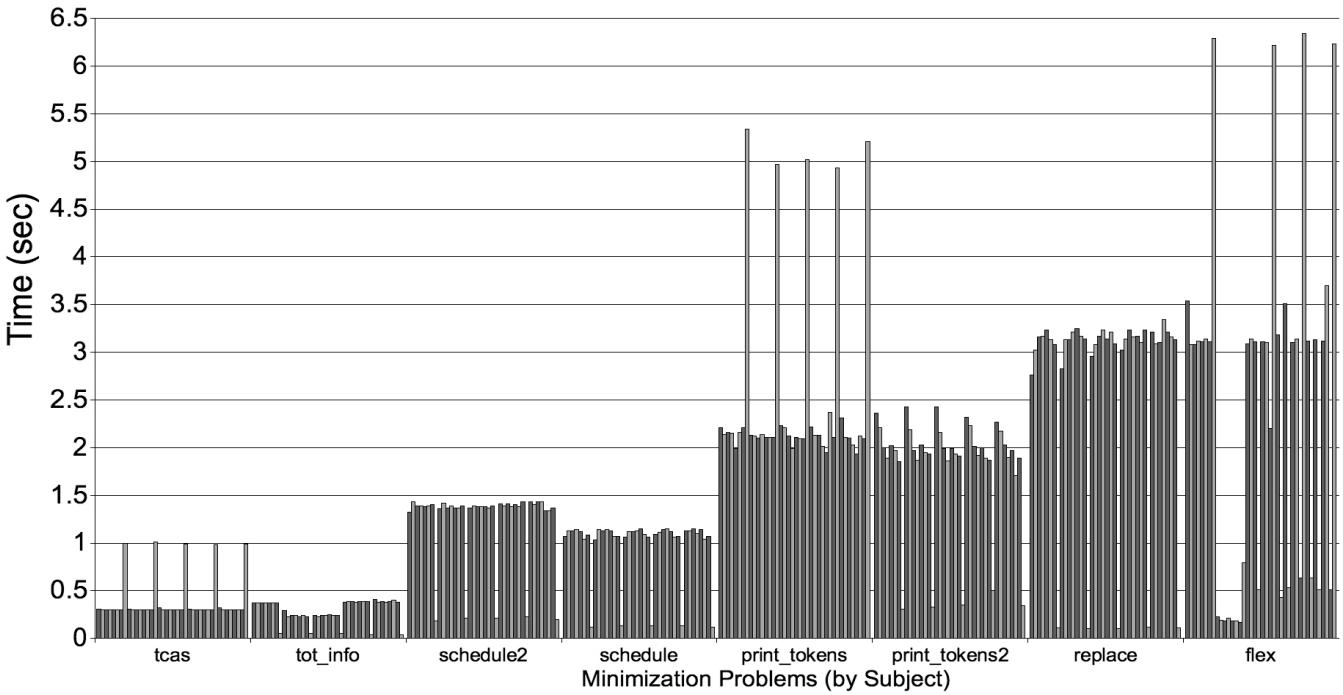

Figure 2: Timing results for mints when applied to the 320 minimization problems considered.

the test suites considered, 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 [6]. Nevertheless, our results provide some initial evidence that the approach can scale. Additional evidence is provided by the results of the Pseudo Boolean Evaluation 2007 [22], where some of the solvers involved where 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 (which in our context corresponds to the number of the test cases). 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.

5.3.2 *RQ2: How does* MINTS*'s performance compare with the performance of a heuristic approach?*

To answer RQ2, we cannot simply evaluate the performance of existing heuristic approaches when applied to the 320 minimization problems that we targeted in our studies; such approaches, as we already discussed extensively, cannot handle multi-criteria minimization problems. To perform at least a partial comparison between our approach and existing ones, we (1) considered a set of (simpler) minimization problems analogous to the single-criteria ones used to evaluate Tallam and Gupta's heuristic approach, Delay-Greedy, and (2) compared our results with the ones they presented [29]. We chose Delay-Greedy as a representative of approximated approaches because it is one of the latest heuristic approaches presented and has been shown to be superior to other existing techniques.

Our set of minimization problems for this study consists of computing, for all five versions of each of the Siemens subjects, a reduced test suite that minimizes the number of test cases while maintaining the same coverage level as the original test suite. We did not consider flex in this study because only the Siemens programs where used by Tallam and Gupta. Moreover, because they did not use the complete test suites available for the Siemens programs, but coverage-adequate subsets of these suites, we also generated similar test suites for this experiment. In particular, we made sure to use initial test suites of the same size as theirs.

Given this setup, we then used MINTS to solve these minimization problems, measured the time necessary for MINTS to compute the solution to the problems, and compared our results with the ones for Delay-Greedy. Because we are not using exactly the same experimental settings used in [29], we cannot perform a precise comparison of our results with the ones presented there. In particular, we cannot compare our results in terms of sizes of minimized test suites. However, we use the same subjects, the same minimization problems, and similar test suites and test data. We can therefore perform at least a qualitative comparison of the performance of the two approaches in terms of time required for the minimization.

Overall, MINTS was able to compute optimal solutions to all 35 problems considered in a time that ranges between 0.003 and 0.07 seconds, with an average time per problem of 0.017 seconds. To compute approximated solution for the set of minimization problems targeted in its evaluation, Delay-Greedy took between 0.003 and 0.027 seconds, with an average time per minimization problem of 0.049 seconds. These results show that, for the subjects and test suites considered (and with the limitations of the comparison discussed above), the performance of our approach is at least compa-

Table 3: Performance of the different ILP solvers.

	MiniSat+	GLPPB	OPBDP	BSOLO	PBS4	CPLEX
# times fastest	28	64	1	65	26	144
# times timed out	79	1	30	22	64	0

rable to, if not better than the performance of a state-of-the art heuristic technique. In addition, our technique can handle a wider range of minimization problems and computes optimal, rather than approximated solutions.

5.3.3 *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 not important 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 insight for improving the approach and for future research. To gather this information, we examined the detailed 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. Table 3 provides this information.

As the table shows, there is a considerable amount of variability in the performance of the different solvers. Interestingly, most solvers produce the fastest solution in a number of cases, but timed out in many other cases. The examination of finer-grained data about the results shows 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 timeout for some of the minimization 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 timeout for minimizations that were completed in a few seconds by other solvers.

As discussed in [6], 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 simplex-based approaches, we conjecture that such approaches are more suitable to the characteristics of the test-suite minimization problem than approaches based purely on SAT solving used in most other pseudo-boolean solvers.

A deeper analysis of the performance of the various ILP solvers when used in this context is out of the scope of this paper, but would be an interesting direction for 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.

5.4 Threats to Validity

The main threat to the external validity of our results is the fact that we consider only eight applications of limited size. Experiments with additional and larger subjects collected from different sources and with different characteristics may generate different results. However, the applications we use in our studies are real programs, used in many previous studies, and one of them is a widely used program. Moreover, despite the size of the applications, in the studies we use test suites of considerable size. 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 try to 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 can compute optimal minimal solutions for such problems by encoding them as binary ILP problems and leveraging existing ILP solvers. We also presented a tool, MINTS, that implements our approach and is freely available for download.[3] 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 more than 300 minimization problems involving eight different subjects. Our results also show that, for the cases considered, our approach can be as efficient as heuristic approaches.

We are currently considering several directions for future work. First, we will perform additional empirical studies with larger 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.

Second, we will continue to analyze our results to get better insight on the reasons for the large variance in the performance of different ILP solvers. We believe that a deeper understanding of this issue could help us improve our approach and possibly provide interesting data for the developers of such solvers.

Finally, we will investigate the possibility of extending our approach to the test-case prioritization problem, that is, the identification of the ordering of test cases in a test suite that maximizes the likelihood of early detection of faults. In our preliminary investigation we discovered that, because of the NP-hard nature of the minimization problem, straightforward extensions of our approach would not work in this context, and more sophisticated (or alternative) approaches should be investigated.

[3] `http://www.cc.gatech.edu/~orso/software.html`

7. REFERENCES

[1] H. Agrawal. Efficient coverage testing using global dominator graphs. In *Proceedings of the 1999 Workshop on Program analysis for software tools and engineering*, pages 11–20, 1999.

[2] B. Al-Rawi and F. Aloul. Pseudo-Boolean Solver v4.0. `http://www.eecs.umich.edu/~faloul/Tools/pbs4/`, 2008.

[3] 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, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[4] P. Barth. OPBDP. `http://www.mpi-inf.mpg.de/departments/d2/software/opbdp/`, 2008.

[5] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[6] D. L. Berre and A. Parrain. On extending sat solvers for pb problems. In *14th RCRA workshop Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion(RCRA07)*, Rome, July 2007.

[7] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 106–115, Edinburgh, Scotland, United Kingdom, May 2004.

[8] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

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

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

[11] N. Een and N. Sorensson. MiniSat+. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat+.html`, 2008.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering (ICSE 94)*, pages 191–200, 1994.

[14] ILOG. CPLEX v9. `http://www.ilog.com/products/cplex/`, 2008.

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

[16] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 02)*, pages 119–129, 2002.

[17] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of Conference on Software Maintenance '91*, pages 201–208, October 1991.

[18] A. G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 230–240, Montreal, Canada, October 2002.

[19] V. Manquinho and J. Marques-Silva. BSOLO v3. `http://sat.inesc-id.pt/bsolo/`, 2008.

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

[21] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Using static analysis to determine where to focus dynamic testing effort. In *Proceedings of IEE/Workshop on Dynamic Analysis (WODA04)*, 2004.

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

[23] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[24] 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 *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

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

[26] H. Sheini. Pueblo glpPB 0.2. `http://www.eecs.umich.edu/~faloul/Tools/pbs4/`, 2008.

[27] Laboratory for Empirically-based Software Quality Research and Development. Software-artifact Infrastructure Repository. `http://sir.unl.edu/php/index.php`, 2008.

[28] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 97–106, July 2002.

[29] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, September 2005.

[30] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *IFIP TC5 WG5.4 3rd internatinal conference on on Reliability, quality and safety of software-intensive systems*, pages 3–21, May 1997.

[31] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12, Portland, Maine, USA, July 2006.

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

[33] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th IEEE International Symposium on Software Reliability Engineering*, pages 552–528, Albuquerque, New Mexico, USA, November 1997.

[34] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50, April 1995.

[35] R. R. Yager. Modeling prioritized multicriteria decision making. *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, 34(6), December 2004.