Nemo: Multi-Criteria Test-Suite Minimization with Integer Nonlinear Programming

Jun-Wei Lin
University of California, Irvine
junwel1@uci.edu

Reyhaneh Jabbarvand
University of California, Irvine
jabbarvr@uci.edu

Joshua Garcia
University of California, Irvine
joshug4@uci.edu

Sam Malek
University of California, Irvine
malek@uci.edu

ABSTRACT

Multi-criteria test-suite minimization aims to remove redundant test cases from a test suite based on some criteria such as code coverage, while trying to optimally maintain the capability of the reduced suite based on other criteria such as fault-detection effectiveness. Existing techniques addressing this problem with integer linear programming claim to produce optimal solutions. However, the multi-criteria test-suite minimization problem is inherently nonlinear, due to the fact that test cases are often dependent on each other in terms of test-case criteria. In this paper, we propose a framework that formulates the multi-criteria test-suite minimization problem as an integer nonlinear programming problem. To solve this problem optimally, we programmatically transform this nonlinear problem into a linear one and then solve the problem using modern linear solvers. We have implemented our framework as a tool, called Nemo, that supports a number of modern linear and nonlinear solvers. We have evaluated Nemo with a publicly available dataset and minimization problems involving multiple criteria including statement coverage, fault-revealing capability, and test execution time. The experimental results show that Nemo can be used to efficiently find an optimal solution for multi-criteria test-suite minimization problems with modern solvers, and the optimal solutions outperform the suboptimal ones by up to 164.29% in terms of the criteria considered in the problem.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Test-suite minimization, integer programming

ACM Reference Format:
Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. 2018. Nemo: Multi-Criteria Test-Suite Minimization with Integer Nonlinear Programming. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180174

1 INTRODUCTION

Software testing plays an essential role in software development, providing a means to determine automatically whether a program behaves as expected. To ensure the correctness of a program as it evolves, engineers should perform *regression testing* on it to ensure that modified or introduced code does not break the program's original functionality. To exercise new behaviors or detect newly discovered faults in software, test suites for regression testing are continuously expanded, and hence may become too large to execute in their entirety [29]. For example, a test suite for a system with about 20,000 source lines of code may require seven weeks to run [40]. Moreover, during the development of Microsoft Windows 8.1, more than 30 million test executions were performed [22]. Consequently, large test suites can make regression testing impractical.

To address this problem, several approaches for test-suite maintenance such as test-suite minimization, test-case selection, and test-case prioritization have been proposed [50]. Specifically, test-suite minimization aims to find the minimal subset of the original test suite which satisfies the same testing requirements [20]. Although an existing minimization technique may work well with respect to a single criterion, the capability of a minimized test suite may be severely compromised in terms of other criteria, such as fault-revealing power [39, 49]. As a result, a tester may consider multiple criteria when performing the reduction. For instance, she may want to generate a reduced suite with the same statement coverage and maximal fault-detection capability.

To accommodate multi-criteria test-suite minimization (MCTSM) problems, existing techniques [5, 23] model the problems as binary integer linear programming (ILP) problems. A binary ILP problem optimizes a linear objective function consisting of binary variables under a set of linear constraints [47]. By encoding test-case criteria (e.g., faults or statements covered by tests) as constraints or objective functions of a binary ILP problem, existing techniques claim that the computed solutions are optimal for the minimization problem [23]. However, an ILP formulation actually results in sub-optimal solutions, since the MCTSM problem is inherently nonlinear. The nonlinearity arises due to the fact that test cases are often dependent on each other in terms of test-case criteria. For example, consider the case where the goal of the test-suite minimization is to select test cases in a suite that (1) maximize the fault-detection effectiveness of the reduced suite and (2) maintain the same code coverage as the original unminimized test suite. In this example, simply selecting test cases that reveal *more* faults ignores the possibility of the same fault being revealed by multiple test cases, which our results indicate actually occurs often. To

ensure that test cases cover a *diverse* number of faults, a nonlinear formulation of the test-suite optimization problem is required.

To deal with dependencies among test cases in a MCTSM problem, we formulate the problem as an integer nonlinear programming problem. To solve this problem optimally, we present a novel approach that programmatically transforms this nonlinear problem into a linear problem and then solves the problem using modern ILP solvers. We evaluate our proposed approach using a publicly available dataset of open-source projects [21]. Our experiments for bi-criteria test-suite minimization problems show that modeling the objective functions nonlinearly results in minimized test suites that, on average, cover more faults. For our experiments involving a tri-critera problem, the test suites reduced by a nonlinear formulation always obtain superior statement coverage and fault-detection effectiveness, given an execution-time budget for the test suite.

The contributions of this paper are as follows:

- We propose the first approach for optimally solving MCTSM problems involving dependencies among test-case criteria. Our approach takes into account the inherent nonlinearity of the problem, unlike previous approaches that model the problem linearly.
- We implement a prototype tool, called Nemo (NonlinEar test suite MinimizatiOn), allowing testers to specify MCTSM problems. The tool programmatically transforms nonlinear problems into linear ones so that modern ILP solvers can be leveraged to compute optimal solutions. We also provide a version of Nemo that leverages nonlinear solvers.
- We conducted an empirical study in which the proposed approach is evaluated with a publicly available dataset of open-source projects.

The rest of this paper is organized as follows. Section 2 introduces background on ILP problems, particularly in the context of MCTSM, and provides a motivating example. We describe our novel approach for formulating MCTSM problems nonlinearly in Section 3 and its corresponding implementation in Section 4. We empirically evaluate Nemo in Section 5. Section 6 describes work related to Nemo. Section 7 concludes the paper.

2 BACKGROUND AND EXAMPLE

In this section, we discuss a motivating example to demonstrate that formulating MCTSM problems linearly may result in suboptimal solutions. An MCTSM problem is a multi-objective optimization problem in which the best subset of the original test suite is selected from available alternatives based on some minimization criteria. As a result, integer linear programming (ILP) for mathematical optimization can be adopted to model and solve the problem. Specifically, existing approaches [5, 23] formulate the MCTSM problem as a binary ILP problem. Such a problem tries to find the optimal value of a linear *objective function* consisting of binary *decision variables*, which are restricted to be 0 or 1, while satisfying a set of linear (in)equality constraints, which we refer to as *constraint criteria*. The objective function maximizes or minimizes one or more *optimization criteria*. Although finding a solution for a binary ILP problem is NP-complete, some important subclasses of the problem are efficiently solvable by modern ILP solvers, due to recent algorithmic and implementation advances [47].

To aid in illustrating the encoding of an MCTSM problem into a binary ILP problem, consider the example in Table 1, which depicts a set of three test cases with each test case's corresponding statement and fault coverage. In this example, the problem under consideration involves the following criteria: a constraint criterion c_1, i.e., maintaining the same statement coverage as the original test suite, and an optimization criterion o_1, i.e., maximizing the fault-detection effectiveness of the reduced suite.

Criteria \ Test cases		t_1	t_2	t_3
Statement	$stmt_1$	1	0	1
	$stmt_2$	0	1	0
	$stmt_3$	0	1	1
Fault	f_1	0	1	1
	f_2	0	1	1
	f_3	0	1	1
	f_4	1	0	0

Table 1: An example test suite with coverage and fault detection data

We model the problem based on state-of-the-art formulations provided in previous work [5, 23]. First, we let a binary decision variable t_i represent whether the ith test case is included in the reduced suite, i.e., each t_i takes a 1 if the corresponding test case is selected, and 0 otherwise. Next, we model constraint criterion c_1, as follows, to ensure that every statement covered by the original suite is covered at least once by the reduced suite:

$$\sum_{i=1}^{|T|} \sigma_{ij}\, t_i \geq 1,\ 1 \leq j \leq q \tag{1}$$

$|T|$ is the number of test cases in the test suite, and q is the total number of distinct statements covered by the test suite. σ_{ij} is a binary variable indicating whether statement $stmt_j$ is covered by test case t_i. For the example test suite in Table 1, equation 1 becomes:

$$t_1 + t_3 \geq 1$$
$$t_2 \geq 1$$
$$t_2 + t_3 \geq 1$$

We express the goal to minimize the test suite and optimization criterion o_1 as a linear objective function as follows:

$$min \sum_{i=1}^{|T|} \epsilon(t_i)\, t_i \tag{2}$$

The minimum function tries to select the smallest subset of the original test suite. The function $\epsilon(t_i)$ models the capability of the test case t_i to find faults, specifically the optimization criterion o_1. A test detecting more faults would have a smaller value returned by ϵ, and thus more likely to be selected. ϵ is formulated as follows:

$$\epsilon(t_i) = (1 - w(t_i)),\ w(t_i) = \frac{\sum_{j=1}^{m} v_{ij}}{m} \tag{3}$$

m is the total number of distinct faults covered by the test suite. v_{ij} is a binary variable indicating whether test case t_i reveals fault f_j. In this formulation, v_{ij} is 1 if t_i reveals fault f_j; otherwise, v_{ij} is 0. For the example test suite in Table 1, the values assigned by ϵ are shown in the equation below:

$$min \sum_{i=1}^{3} \epsilon(t_i)\, t_i = (1 - \frac{1}{4})t_1 + (1 - \frac{3}{4})t_2 + (1 - \frac{3}{4})t_3 \tag{4}$$

The formulated problem can then be solved by using an ILP solver. We refer to this approach as *LF_LS (LinearFormulation_LinearSolver)*, which models an MCTSM problem with a linear formulation and solves it with a linear solver. This is the approach that has been followed in prior work [5, 19, 23], including a general tool for test-suite minimization by Hsu and Orso, called MINTS [23].

The optimal solution for this binary ILP problem is $\{t_2, t_3\}$ with a minimal value $\frac{1}{2}$ under the constraints. However, from Table 1, we can see that $\{t_2, t_3\}$ detects only three of the four faults (i.e., it misses fault f_4)—and hence is not the optimal solution for the minimization problem, which is actually $\{t_1, t_2\}$. Note that $\{t_1, t_2\}$ obtains a value of 1 for equation 4. The correct solution can not be computed by LF_LS, because the linear objective function tends to select test cases revealing *more* faults but not necessarily *distinct* faults. For example, once t_2 is selected, no further distinct faults can be revealed by t_3, but t_3 would still be selected because it reveals more faults than t_1 (i.e., three faults instead of just one). In fact, t_1 should be selected for the minimization problem instead of t_3 because t_1 reveals a fault distinct from the faults in t_2 and t_3.

Such dependencies among test cases cannot be encoded with a linear objective function; they must be encoded using a nonlinear objective function. Specifically, the test cases currently selected are dependent on the test cases previously selected. To model such dependencies among test cases, the optimization problem must be formulated in a manner such that the decision variables (i.e., t_is in the previous equations) are multiplied with each other, making them inherently nonlinear. In the next section, we describe this formulation.

3 NONLINEAR PROBLEM FORMULATION

To nonlinearly model the MCTSM problem, we describe the problem more formally, illustrate how our nonlinear formulation models test-case dependencies and multiple objective criteria, and describe the manner in which we utilize linear solvers to optimally solve the nonlinear formulation of the problem.

3.1 Problem definition

To clarify our proposed idea, we formally define the MCTSM problem as follows:

Given:

(1) A test suite $T = \{t_1, t_2, ..., t_n\}$
(2) A set of constraint criteria $C = \{c_1, c_2, ..., c_k\}$ which must be satisfied by T
(3) A set of optimization criteria $O = \{o_1, o_2, ..., o_l\}$, i.e., the criteria to be optimized by an objective function
(4) A non-negative function $\epsilon(t)$ that represents the significance of a test case $t \in T$ with respect to the optimization criteria. $\epsilon(t)$ is a weighted sum of functions indicating the capability of t with respect to each of the optimization criteria. For example, if the optimization criteria are (o_1) maximizing fault detection effectiveness and (o_2) minimizing test execution time, we can define $\epsilon(t)$ as $\epsilon(t) = \alpha_1\epsilon_1(t) + \alpha_2\epsilon_2(t)$. Here, $\epsilon_1(t)$ represents the fault-detection capability of t; $\epsilon_2(t)$ represents the execution time of t. Each factor α is a weight prioritizing a criterion.

Problem: Find a minimum test suite $T' \subseteq T$ such that

(p1) T' satisfies C
(p2) $\forall T''$ satisfying C, $|T'| \leq |T''|$
(p3) $\forall T''$ satisfying C, $\sum_{t_i \in T'} \epsilon(t_i) \geq \sum_{t_k \in T''} \epsilon(t_k)$

This minimization problem is NP-complete, because it is in NP and can be reduced from the minimum set-covering problem in polynomial time [16].

To deal with dependencies among tests over multiple optimization criteria, we must formulate the criteria in a nonlinear fashion. Specifically, when our new formulation computes the capability of a test with respect to each optimization criterion, the formulation needs to consider if the criterion is satisfied by other selected tests. To that end, we alter equations 2-3 to account for (1) multiple optimization criteria and (2) dependencies among test cases over a specific optimization criterion using the following equation:

$$min \sum_{o \in O} \alpha_o \sum_{i=1}^{|T|} \epsilon_o(t_i)\, t_i,\ \epsilon_o(t_i) = (1 - \tilde{w}_o(t_i)) \tag{5}$$

$o \in O$ is an optimization criterion; α_o is the assigned weight for o; $\epsilon_o(t_i)$ is a function computing the capability of a test case t_i in terms of a criterion o. A novelty of our approach is that we consider dependencies among test cases in modeling $\tilde{w}_o(t_i)$, which makes the formulation nonlinear (more details in Section 3.2). $\tilde{w}_o(t_i)$ is a function computing the problem-specific significance of a test case and, thus, can be defined as needed for different criteria and minimization problems. For example, $\tilde{w}_o(t_i)$ can model whether test case t_i identifies faults distinct from previously selected test cases. As another example, $\tilde{w}_o(t_i)$ can model whether test case t_i covers the most frequently executing statements a certain number of times. We next illustrate an instantiation of $\tilde{w}_o(t_i)$ for the fault-detection criterion from the example of Section 2.

3.2 Modeling test-case dependencies

In this subsection, we illustrate an instantiation of the proposed formulation for an MCTSM problem involving the two criteria from our motivating example: a constraint criterion (c_1) that maintains the same statement coverage as the original test suite, and an optimization criterion (o_1) that maximizes the fault-detection effectiveness of the reduced suite. Note that this instantiation involves a single constraint criterion and a single optimization criterion—even though our formulation can handle multiple constraint and optimization criteria—due to space limitations and to maximize readability of our nonlinear formulation. First, for optimization criterion o_1, we set the weight as one (i.e., $\alpha_1 = 1$) and assign the objective function using equation 5 as follows:

$$min \sum_{o \in O} \alpha_o \sum_{i=1}^{|T|} \epsilon_o(t_i)\, t_i \tag{6}$$

$$= 1 \sum_{i=1}^{|T|} \epsilon_{o_1}(t_i)\, t_i \tag{7}$$

$$= \sum_{i=1}^{|T|} (1 - \tilde{w}_{o_1}(t_i))\, t_i \tag{8}$$

Furthermore, to model dependencies among tests over o_1, i.e., fault-detection effectiveness, when calculating how many faults are revealed by a test, we have to consider if the faults are already revealed by selected tests. As a result, we can define $\tilde{w}_{o_1}(t_i)$ as follows:

$$\tilde{w}_{o_1}(t_i) = \frac{1}{|F|} \left(\sum_{j=1}^{|F|} v_{ij} d_{ij}\right) \tag{9}$$

$$d_{ij} = \prod_{t \in T_j} (1 - t),\ t \neq t_i \tag{10}$$

$|F|$ is the number of distinct faults in $F = \{f_1, f_2, ..., f_{|F|}\}$ revealed by T. T_j is the set of test cases that reveal f_j. d_{ij} accounts for dependencies among test cases in terms of each f_j: If at least one of the test cases in T_j is selected, d_{ij} evaluates to zero, which decreases (1) the value of $\tilde{w}_{o_1}(t_i)$ contributed by t_i and f_j, and (2) the likeliness of t_i being selected.

To illustrate the use of equations 8, 9, and 10, we apply it to the example in Table 1. In the example in Table 1, $T = \{t_1, t_2, t_3\}$, and $F = \{f_1, f_2, f_3, f_4\}$. Consequently, the objective function for this scenario is the following:

$$min \sum_{o \in O} \alpha_o \sum_{i=1}^{3} \epsilon_o(t_i)\, t_i \tag{11}$$

$$= 1\,(\epsilon_{o_1}(t_1)\, t_1 + \epsilon_{o_1}(t_2)\, t_2 + \epsilon_{o_1}(t_3)\, t_3) \tag{12}$$

$$= (1 - \tilde{w}_{o_1}(t_1))\, t_1 + (1 - \tilde{w}_{o_1}(t_2))\, t_2 + (1 - \tilde{w}_{o_1}(t_3))\, t_3 \tag{13}$$

$$= \underbrace{(1 - \frac{1}{4}(\sum_{j=1}^{4} v_{1j}\, d_{1j}))\, t_1}_{\text{test case } t_1} + \underbrace{(1 - \frac{1}{4}(\sum_{j=1}^{4} v_{2j}\, d_{2j}))\, t_2}_{\text{test case } t_2} + \underbrace{(1 - \frac{1}{4}(\sum_{j=1}^{4} v_{3j}\, d_{3j}))\, t_3}_{\text{test case } t_3} \tag{14}$$

In the above formulation, test case t_1 reveals fault f_4, and f_4 is not revealed by any other test case. As a result, the set of test cases revealing f_4 (i.e., T_4) is $\{t_1\}$, and the coefficient for t_1 in equation 14 is expanded as follows:

$$\begin{aligned}
&(1 - \frac{1}{4}(\sum_{j=1}^{4} v_{1j}\, d_{1j}))\, t_1 \\
&= (1 - \frac{1}{4}(v_{11}\, d_{11} + v_{12}\, d_{12} + v_{13}\, d_{13} + v_{14}\, d_{14}))\, t_1 \\
&= (1 - \frac{1}{4}(0\, d_{11} + 0\, d_{12} + 0\, d_{13} + 1\, d_{14}))\, t_1 \\
&= (1 - \frac{1}{4}(1\, d_{14}))\, t_1 \\
&= (1 - \frac{1}{4}(1 \times \prod_{t \in T_4} (1 - t)))\, t_1,\ t \neq t_1 \\
&= (1 - \frac{1}{4}(1))\, t_1 \\
&= (1 - \frac{1}{4})\, t_1
\end{aligned}$$

The above equation expresses t_1 as independent of other test cases, since it reveals a fault that no other test case in the suite reveals.

Recall that test cases t_2 and t_3 reveal the same set of faults, i.e., $\{f_1, f_2, f_3\}$. As a result, the set of test cases revealing each of those faults are also the same, i.e., $T_1 = T_2 = T_3 = \{t_2, t_3\}$. Consequently, the coefficient of t_2 in equation 14 is expanded as follows:

$$\begin{aligned}
&(1 - \frac{1}{4}(\sum_{j=1}^{4} v_{2j}\, d_{2j}))\, t_2 \\
&= (1 - \frac{1}{4}(v_{21}\, d_{21} + v_{22}\, d_{22} + v_{23}\, d_{23} + v_{24}\, d_{24}))\, t_2 \\
&= (1 - \frac{1}{4}(1\, d_{21} + 1\, d_{22} + 1\, d_{23} + 0\, d_{24}))\, t_2 \\
&= (1 - \frac{1}{4}(1\, d_{21} + 1\, d_{22} + 1\, d_{23}))\, t_2 \\
&= (1 - \frac{1}{4}(1 \times \prod_{t \in T_1} (1 - t) + 1 \times \prod_{t \in T_2} (1 - t) + 1 \times \prod_{t \in T_3} (1 - t)))\, t_2,\ t \neq t_2 \\
&= (1 - \frac{1}{4}((1 - t_3) + (1 - t_3) + (1 - t_3)))t_2
\end{aligned}$$

Note that the equation above shows an interaction or dependency between test cases t_2 and t_3, as represented by the multiplication of those two decision variables, clearly showing their formulation as nonlinear. Recall that this dependency is expected since both test cases detect the same set of faults.

Finally, due to test case t_3 revealing the same faults as t_2, the t_3 coefficient of equation 14 can be expanded in a manner similar to that of t_2:

$$\begin{aligned}
&(1 - \frac{1}{4}(\sum_{j=1}^{4} v_{3j}\, d_{3j}))\, t_3 \\
&= (1 - \frac{1}{4}(v_{31}\, d_{31} + v_{32}\, d_{32} + v_{33}\, d_{33} + v_{34}\, d_{34}))\, t_3 \\
&= (1 - \frac{1}{4}(1\, d_{31} + 1\, d_{32} + 1\, d_{33} + 0\, d_{34}))\, t_3 \\
&= (1 - \frac{1}{4}(1\, d_{31} + 1\, d_{32} + 1\, d_{33}))\, t_3 \\
&= (1 - \frac{1}{4}(1 \times \prod_{t \in T_1} (1 - t) + 1 \times \prod_{t \in T_2} (1 - t) + 1 \times \prod_{t \in T_3} (1 - t)))\, t_3,\ t \neq t_3 \\
&= (1 - \frac{1}{4}((1 - t_2) + (1 - t_2) + (1 - t_2)))t_3
\end{aligned}$$

For constraint criterion c_1, we adopt constraints similar to those in equation 1 because all requirements of a constraint criterion (e.g., all statements) for c_1 have to be satisfied (e.g. covered) at least once by the reduced suite:

$$\sum_{i=1}^{|T|} \sigma_{ij}\, t_i \geq 1,\ 1 \leq j \leq |c_1|$$

$|c_1|$ is the size of constraint criterion c_1; σ_{ij} is a binary variable indicating whether t_i satisfies the jth requirement of c_1 (e.g., the jth statement).

With the above objective function and constraints, the proposed formulation for Table 1 results in the following assignment for the optimization problem:

minimize:

$$\begin{aligned}
(1 - \frac{1}{4})t_1 + (1 - \frac{1}{4}((1 - t_3) + (1 - t_3) + (1 - t_3)))t_2 \\
+(1 - \frac{1}{4}((1 - t_2) + (1 - t_2) + (1 - t_2)))t_3
\end{aligned}$$

under the constraints:

$$\begin{aligned}
t_1 + t_3 &\geq 1 \\
t_2 &\geq 1 \\
t_2 + t_3 &\geq 1
\end{aligned}$$

With this nonlinear formulation, the optimal solution for integer nonlinear programming (INP) problem is $\{t_1, t_2\}$ with a minimal value 1. This solution is the correct, optimal solution for the minimization problem. Note that the solution selected by the linear objective function in equation 3, i.e., $\{t_2, t_3\}$, obtains a value 2 using the nonlinear objective function; thereby it will not be selected as the final solution.

The above nonlinear formulation can be supplied directly to a nonlinear solver, which is an approach for solving an MCTSM problem that we refer to as *NF_NS (NonlinearFormulation_ NonlinearSolver)*. However, utilizing nonlinear solvers does not guarantee optimal solutions [48], leading us to propose a different approach to solving the nonlinear formulation, as described in the next subsection.

3.3 Transformation to linear programming

There is no known efficient algorithm for solving an INP problem optimally other than trying every possible combination. Furthermore, for problems with non-convex functions (such as MCTSM), nonlinear solvers are not guaranteed to find an optimal solution [48], making NF_NS not necessarily optimal. As a result, instead of directly solving the nonlinear formulation, we investigated how to transform the nonlinear formulation into a linear one. A linear formulation can be solved optimally given the recent advances in ILP solver technology [48]. We refer to this approach of solving the nonlinear formulation using linear solvers as *NF_LS (NonlinearFormulation_LinearSolver)* and describe it in the remainder of this section.

To allow the use of linear solvers for NF_LS, we transform the nonlinear MCTSM problem into a linear one by introducing new "auxiliary" variables [7]. We demonstrate the approach using the instantiation of the MCTSM problem in Section 3.2. We introduce up to $|F| \times |T|$ new decision variables $\bar{v}_{ij}$ defined as follows:

$$\bar{v}_{ij} = v_{ij} d_{ij} t_i$$

$\bar{v}_{ij}$ is a binary variable indicating whether a fault f_j is revealed by t_i or any other previously selected test case: $\bar{v}_{ij}$ is 1 if t_i reveals f_j and is not revealed by the previously selected test cases, and 0 otherwise. Using the newly introduced variables, equation 8 can be rewritten as follows:

$$min \sum_{i=1}^{|T|} (t_i - \frac{1}{|F|} \sum_{j=1}^{|F|} \bar{v_{ij}}) \tag{15}$$

Notice that in this formulation, decision variables t are no longer multiplied. However, the transformation has resulted in the introduction of new decision variables $\bar{v}$. We add an additional set of constraints to the model to avoid having an unselected test case affect the minimized value of a solution. Specifically, $\bar{v}_{ij}$ should be subject to the selection of t_i, i.e., $\bar{v}_{ij}$ matters only when t_i is selected, resulting in the following constraints:

$$\bar{v}_{ij} \leq t_i, \ \forall f_j \text{ revealed by } t_i \tag{16}$$

These constraints illustrate that if a test case t_i is not selected, $\bar{v}_{ij}$ is forced to be 0. However, if t_i is selected, $\bar{v}_{ij}$ could be either 1 or 0, depending on whether f_j is revealed by the previously selected test cases.

In addition, we add constraints to make the selected test cases more diverse in terms of f_j:

$$\sum_{i=1}^{|T|} v_{ij} \bar{v}_{ij} \leq 1, \ 1 \leq j \leq |F| \tag{17}$$

The constraints in (17) model our preference for selecting test cases revealing faults that are not revealed by the previously selected test cases: If a fault f_j is revealed by a selected test case t_i and its $\bar{v}_{ij}$ is set to 1, then for all other test cases revealing f_j, their $\bar{v}_{ij}$ has to be 0. For example, for the constraint $\bar{v}_{21} + \bar{v}_{31} \leq 1$ obtained from equation 17, where $v_{21} = v_{31} = 1$, only t_2 or t_3 can be selected but not both.

To illustrate the proposed transformation (i.e., the objective function in equation 15 and additional constraints in equation 16 and 17), we apply it to the example of Table 1 (i.e., $T = \{t_1, t_2, t_3\}$, $F = \{f_1, f_2, f_3, f_4\}$, and $\alpha_1 = 1$), and transform the nonlinear formulation discussed in Section 3.2 to the following assignment:

minimize:

$$(t_1 - \frac{1}{4}(\bar{v}_{14}))$$
$$+ (t_2 - \frac{1}{4}(\bar{v}_{21} + \bar{v}_{22} + \bar{v}_{23}))$$
$$+ (t_3 - \frac{1}{4}(\bar{v}_{31} + \bar{v}_{32} + \bar{v}_{33}))$$

under the constraints:

$t_1 + t_3 \geq 1$	$(original)$
$t_2 \geq 1$	$(original)$
$t_2 + t_3 \geq 1$	$(original)$
$\bar{v}_{14} \leq t_1$	$(from\,(16))$
$\bar{v}_{21} \leq t_2, \ \bar{v_{22}} \leq t_2, \ \bar{v_{23}} \leq t_2$	$(from\,(16))$
$\bar{v}_{31} \leq t_3, \ \bar{v_{32}} \leq t_3, \ \bar{v_{33}} \leq t_3$	$(from\,(16))$
$\bar{v}_{21} + \bar{v_{31}} \leq 1$	$(from\,(17))$
$\bar{v}_{22} + \bar{v_{32}} \leq 1$	$(from\,(17))$
$\bar{v}_{23} + \bar{v_{33}} \leq 1$	$(from\,(17))$
$\bar{v}_{14} \leq 1$	$(from\,(17))$

Notice that the decision variables are no longer multiplied in the objective function, making the formulation linear, while still solving the nonlinear problem. The optimal solution for this transformed linear problem is $\{t_1, t_2, \bar{v_{14}}, \bar{v_{21}}, \bar{v_{22}}, \bar{v_{23}}\}$ with a minimal value 1. This minimal value is identical to the optimal value for the untransformed nonlinear problem. The computed solution for the minimization problem (i.e., $\{t_1, t_2\}$) is also the optimal solution.

Optimality of NF_LS. NF_LS yields the optimal solution. The usage of auxiliary variables to transform a nonlinear problem formulation to a linear one is well studied in the literature and has been proven to generate an equivalent formulation [7, 48]. After transforming the nonlinear MCTSM problem to an instance of an ILP problem, the optimal solution can be found by leveraging modern solvers. Note that the MCTSM problem is known to be NP-complete, which means that an optimal solution cannot be found in polynomial time. However, that does not preclude modern solvers from finding optimal solutions through utilization of branch and bound and other algorithmic advancements for sizable problems [48]. That is, ILP solvers do not guarantee to return a solution within a time limit, but guarantee that the returned solution is optimal[1].

4 IMPLEMENTATION

Figure 1 depicts our framework for solving the MCTSM problems as implemented in our tool, called Nemo. The tool takes *test-related data* and a *configuration* as input. Test-related data includes the original test suite and the corresponding coverage data such as statement and fault coverage of the test suite. In the configuration file, users can specify the optimization and constraint criteria. Users can also specify which of the three approaches (i.e., LF_LS, NF_NS, and NF_LS) should be used in the minimization of the test suite.

The *Formulator* component takes the input and expresses the minimization problem as an integer programming problem. Formulator is capable of representing the problem as either linear or nonlinear, as well as transforming the nonlinear formulation into a linear form. The output model can then be fed into external solvers

[1] In our experiments, we actually never encountered a problem for which NF_LS could not find a solution.

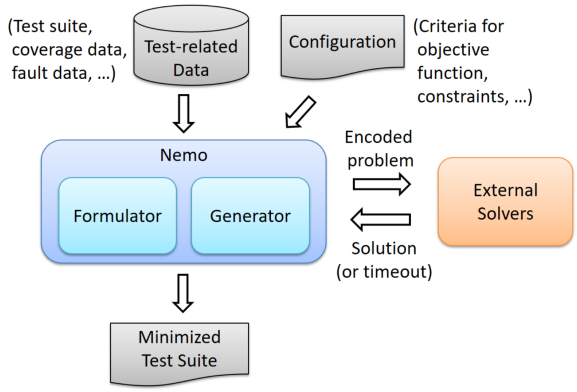

Figure 1: Overview of our approach

to compute the solution. For now our tool supports *lp* and *ampl* formats, which many modern solvers, such as lp_solve [1], CPLEX [13] and Couenne [12] use.

Once the solution for the encoded problem is computed, the *Generator* component produces the minimized test suite in various formats for presentation to the user. Currently, our tool handles solutions generated by the aforementioned solvers and web services using those solvers, such as the NEOS Server [14] and DropSolve [2].

The proposed tool is the only one of its kind to formulate and consider the inherent nonlinearity of the MCTSM problem, and is available for download [3]. In the next section, we evaluate our approach using Nemo to answer a set of research questions.

5 EMPIRICAL EVALUATION

We assess three approaches for solving the MCTSM problem: LF_LS (see Section 2), NF_NS (see Section 3.2), and NF_LS (see Section 3.3). For LF_LS, we reimplemented the technique in a tool called MINTS proposed by Hsu and Orso [23]. We could not use the original implementation of MINTS because the solvers it supports are outdated and not scalable for our dataset. Using these approaches, we investigate the following research questions:

RQ1. How does *NF_LS* compare against *LF_LS* with respect to effectiveness and runtime performance? To assess this research question, we use the size of the reduced test suite and the satisfaction of the minimization criteria to characterize effectiveness of each approach. For performance, we measure the time it takes to solve each approach's formulation of the problem.

RQ2. How does *NF_LS* compare against *NF_NS* with respect to effectiveness and performance? Given that Nemo is the first approach to model general test-suite minimization in a nonlinear manner, we assess which nonlinear formulation is superior, or whether there are empirical tradeoffs between the two versions of Nemo.

RQ3. How does *NF_LS* scale in relation to the size of subject apps and test suites? *NF_LS* solves the nonlinear problem optimally, which is a computationally expensive process. We investigate how the introduction of auxiliary variables impacts the scalability of *NF_LS*.

5.1 Experimental Setup

Subject programs: Our experimental subjects include the following five open-source C projects: Grep, Flex, Sed, Make, and Gzip, collected from a publicly available dataset [21]. These programs are well-known GNU software and widely used in software testing and debugging research [6, 11, 21, 27, 38]. Each subject program in the original dataset has five versions; we selected the latest version of each program for our experiments. Table 2 depicts the following detailed pieces of information about our subject programs: the selected version, a description of the program, its size (*LOC*), and the number of tests and faults available with each version.

Test suites: Each subject program in [21] comes with a test suite and a set of faults. We augmented available test suites with additional tests, since their coverage of the subject apps' core functions were low (i.e., average coverage was below 48%), and can result in test suites that do not comprehensively test the subject programs. To that end, we used *KLEE* [8] to ensure that subject test suites achieve a coverage higher than 60% on average for the core functions. To measure statement coverage of test suites, we used *gcov* [17].

Faults: The original dataset [21] includes a set of faults for each test suite. These faults are generated by injecting mutants and purified by removing equivalent and duplicate mutants using Trivial Compiler Equivalence [37] (TCE), a scalable and effective approach to find equivalent and duplicate mutants that compares the machine code of compiled mutants. We followed the same technique to generate additional unique mutants for the newly generated tests added to the dataset.

Minimization problems: We compare the minimization techniques using the following three minimization problems: classic bi-criteria, variant bi-criteria, and tri-criteria. A *classic*al *bi-criteria* test-suite minimization problem minimizes the test suite such that statement coverage of the original test suite is maintained, while maximizing its fault-detection ability. The *variant bi-criteria* problem is a classic bi-criteria problem whose segments of code (e.g., specific API calls, methods, or classes) are more important than others, and thereby need to be covered multiple times. Examples of scenarios where covering the same code segment multiple times are important include energy testing [24], performance testing, or field-failure reproduction [27]. The *tri-criteria* problem is a multi-criteria minimization problem where the minimized test suite needs to satisfy a budget constraint, while maximizing the statement coverage and fault-detection ability of the test suite. Since available time for regression testing is often an important constraint [35, 46, 51], we consider execution time as the budget in our evaluation. Note that for these two optimization criteria, i.e., statement coverage and fault detection, NF_NS and NF_LS model dependencies among test cases, while LF_LS does not. The weights for both optimization criteria are the same in our experiments.

Solvers: We used CPLEX [13] and Couenne [12] as linear and nonlinear solvers, respectively. For a fair performance comparison of different solvers, we used these solvers on NEOS [14], a free web service for solving numerical optimization problems.

5.2 RQ1: NF_LS vs. LF_LS

We assess the effectiveness and performance of each approach in terms of the three MCTSM problems described in Section 5.1.

Effectiveness. We measured the effectiveness of each minimization technique in terms of the size of reduced suites (*#T*) and corresponding fault-detection ability, i.e., the number of faults detected (*#F*). For our experiments, the same set of faults for each subject program is used to evaluate the fault-detection ability of the reduced suites, because the main objective of our evaluation is to show that the non-linear formulation of the MCTSM problem can produce

Table 2: Subject programs used in the empirical evaluation.

Program	Version	Description	LOC	# Tests	# Faults
Grep	2.7	Pattern search and matching utility	58,344	746	54
Flex	2.5.4	Lexical analyzer	12,366	605	37
Sed	4.2	Command-line text editor	26,466	324	25
Make	3.80	Executables builder and generator	23,400	158	15
Gzip	1.3	Data compressor	5,682	397	56

Table 3: Effectiveness of different methods modeling the *classic bi-criteria* problem. %: the percentage improved by NF_LS compared to LF_LS and NF_NS

Programs / Methods	Grep		Flex		Sed		Make		Gzip	
	#T	#F	#T	#F	#T	#F	#T	#F	#T	#F
(Original)	746	54	605	37	324	25	158	15	397	56
LF_LS	59	29	44	28	12	21	14	12	45	50
NF_LS	59	36	44	32	12	25	14	13	45	50
NF_NS	n/a	n/a	44	32	12	25	14	13	45	49
% NF_LS over LF_LS	0%	24.14%	0%	14.29%	0%	19.05%	0%	8.33%	0%	0%
% NF_LS over NF_NS	n/a	n/a	0%	0%	0%	0%	0%	0%	0%	2.04%

* n/a: the solver timed out after eight hours

a solution that is superior to its linear formulation. By comparing the solutions produced from approaches using the same exact version of software, faults, and coverage information, we are able to unequivocally show the superiority of non-linear formulation to linear formulation in a controlled setting.

Table3 shows the effectiveness of different minimization techniques for solving the *classic bi-criteria* problem. These results demonstrate that NF_LS achieves an equal or greater fault-detection ability compared to LF_LS and the same test-suite size. More specifically, the minimized suites of NF_LS detected 13% more faults on average than those of LF_LS without compromising test-suite size.

We observed that the improvements achieved by NF_LS vary among different subject programs. Intuitively, for an MCTSM problem, NF_LS works better than LF_LS if test cases for the same constraint criterion (e.g., covering the same statement) satisfy an optimization criterion in a different way (e.g., revealing different faults). For instance, in the example of Table 1, t_1 and t_3 both cover $stmt_1$, but reveal different faults (f_4 for t_1; f_1, f_2, and f_3 for t_3). As a result, this difference in fault-revealing ability can be identified by the nonlinear formulation when it selects a test case for covering $stmt_1$ in the minimization process.

To investigate the aforementioned property in our experimental dataset, we clustered test cases by the statements they covered for each subject program. Then, we calculated average similarity of faults revealed among the tests in the same cluster, using the Jaccard similarity metric [25]. Jaccard formulates the similarity between two sets, A and B, as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In our experiments, A and B are two sets of faults covered by two tests within the same cluster. For example, suppose tests t_1 and t_2 cover the same set of statements, causing the tests to be clustered together. If execution of t_1 reveals $A = \{f_1, f_2\}$, and t_2 reveals $B = \{f_2, f_3\}$, the Jaccard similarity for these two tests are computed as $J(A, B) = \frac{1}{3}$.

For a given a program, effectiveness of the nonlinear formulation will degrade as the average similarity among all clusters in that program approaches 1. This effect occurs since tests covering the same statements also cover increasingly similar faults. As that average similarity approaches 0, the nonlinear formulation performs more effectively, i.e., tests covering the same statements cover increasingly different faults.

Table 4: Average Jaccard Similarity of the fault sets revealed by the test cases covering the same set of statements

Program	Grep	Flex	Sed	Make	Gzip
Similarity	0.8394	0.9762	0.9804	0.9219	1.0000

Table 4 shows the Jaccard similarity calculated for each subject program. To obtain these results, we first calculated the similarity between each pair of test cases in the same cluster. We then computed the average similarity among all pairs within a cluster. For Gzip, the similarity is 1, and the nonlinear formulations obtains no improvement in the number of faults revealed, as shown in Table 3. However, in the case of Grep, the average similarity among all clusters is 0.8394, the lowest similarity among all programs, and also obtains the greatest improvement in the number of faults revealed, as shown in Table 3. Note that the effectiveness of the nonlinear formulation is also influenced by other factors (e.g., total number of faults) and hence cannot be solely predicted by fault similarity.

For the *variant bi-criteria* problem, the goal is to maximize the fault-detection ability of the reduced suite, while maintaining the same statement coverage. Additionally, we selected the top 10% of statements executed most frequently to identify the potentially most important statements to execute. For each of these statements, we forced them to be executed at least 10% of the number of times they were executed by the entire test suite. Table 5 shows the effectiveness of different minimization techniques on the *variant bi-criteria* problem. These results demonstrate that NF_LS's test suites have a superior fault-detection ability than LF_LS's test suites for all subject programs. Specifically, the reduced suites of NF_LS detected 17% more faults on average than the suites of LF_LS without increasing the size of the test suite.

For the *tri-criteria* problem, we formulated the problem to constrain the sizes of the reduced suites to 5%, 10%, 15%, and 20%, and to maximize the statement coverage and fault-detection effectiveness with the same weights. Given that even test suites of only 20KLOC

Table 5: Effectiveness of different methods modeling the *variant bi-criteria* problem. %: the percentage improved by NF_LS compared to LF_LS and NF_NS

Methods \ Programs	Grep		Flex		Sed		Make		Gzip	
	#T	#F	#T	#F	#T	#F	#T	#F	#T	#F
(Original)	746	54	605	37	324	25	158	15	397	56
LF_LS	80	38	66	33	32	22	17	13	58	51
NF_LS	80	54	66	37	32	25	17	15	58	52
NF_NS	n/a	n/a	n/a	n/a	n/a	n/a	17	15	n/a	n/a
% NF_LS over LF_LS	0%	42.11%	0%	12.12%	0%	13.64%	0%	15.38%	0%	1.96%
% NF_LS over NF_NS	n/a	n/a	n/a	n/a	n/a	n/a	0%	0%	n/a	n/a

* n/a: the solver timed out after eight hours

Table 6: Effectiveness of different methods modeling the *tri-criteria* problem. %: the percentage improved by NF_LS compared to LF_LS

Size constraint	Methods \ Programs	Grep			Flex			Sed			Make			Gzip		
		#T	#Stmt	#F	#T	#Stmt	#F	#T	#Stmt	#F	#T	#Stmt	#F	#T	#Stmt	#F
	(Original)	746	1695	54	605	3143	37	324	945	25	158	3803	15	397	1409	56
5%	LF_LS	37	1302	29	30	2093	14	16	847	18	8	3665	10	20	540	25
	NF_LS	37	1635	54	30	3094	37	16	945	25	8	3779	15	20	1343	56
	% NF_LS over LF_LS	0%	25.58%	86.21%	0%	47.83%	164.29%	0%	11.57%	38.89%	0%	3.11%	50.00%	0%	148.70%	124.00%
10%	LF_LS	75	1302	29	61	2469	18	32	860	18	16	3676	10	40	540	25
	NF_LS	75	1695	54	61	3143	37	32	945	25	16	3803	15	40	1400	56
	% NF_LS over LF_LS	0%	30.18%	86.21%	0%	27.30%	105.56%	0%	9.88%	38.89%	0%	3.45%	50.00%	0%	159.26%	124.00%
15%	LF_LS	112	1325	33	91	2603	23	49	905	19	24	3703	12	60	541	25
	NF_LS	112	1695	54	91	3143	37	49	945	25	24	3803	15	60	1409	56
	% NF_LS over LF_LS	0%	27.92%	63.64%	0%	20.75%	60.87%	0%	4.42%	31.58%	0%	2.70%	25.00%	0%	160.44%	124.00%
20%	LF_LS	149	1330	34	121	2695	24	65	906	19	32	3703	12	79	541	25
	NF_LS	149	1695	54	121	3143	37	65	945	25	32	3803	15	79	1409	56
	% NF_LS over LF_LS	0%	27.44%	58.82%	0%	16.62%	54.17%	0%	4.30%	31.58%	0%	2.70%	25.00%	0%	160.44%	124.00%

can take weeks to run [40], selecting test suites that are a fraction of the total number of existing test cases is a sensible testing strategy.

The results for this experiment are depicted in Table 6. Our results demonstrate that NF_LS consistently outperforms LF_LS for all subject apps and size constraints. Particularly, test suites reduced by NF_LS executed 45% more statements and 73% more faults than the suites by LF_LS on average. We can see that the improvement achieved by NF_LS in this problem is larger than the previous two problems. This could be attributed to the looser constraints in this problem and hence the larger solution space for all the techniques. For instance, given that the constraint is to reduce the test suite of Grep to 5%, the size of the solution space for this problem is $\binom{746}{37} \approx 5.73 \times 10^{62}$. If there are more constraints, e.g., constraints for statements such as those included in the previous problems, the solution space would be further limited, because some solutions in it do not satisfy the additional constraints. As the solution space grows larger, the effectiveness gap between the solutions returned by NF_LS and LF_LS is expected to increase.

Performance. Tables 7 and 8 showcase the execution times that solvers of different approaches took to solve the *classic bi-criteria* and the *variant bi-criteria* problems, respectively. We retrieved the results from reports generated by NEOS [14]. These results demonstrate that NF_LS can solve these two problems as efficiently as LF_LS does, and the solutions for all subject apps were found within a second.

For the *tri-criteria* problem, we were not able to submit the model files to NEOS due to its size limitations for uploaded files. We thus ran the solver locally, and report the execution times in Table 9. These results indicate that NF_LS takes a longer time than LF_LS to solve the problem. The difference between the performance

Table 7: Performance of different methods modeling the *classic bi-criteria* problem

Methods \ Programs	Solving Time in Seconds				
	Grep	Flex	Sed	Make	Gzip
LF_LS	1	1	1	1	1
NF_LS	1	1	1	1	1
NF_NS	n/a	1708	21854	20	2813

* n/a: the solver timed out after eight hours

Table 8: Performance of different methods modeling the *weighted bi-criteria* problem

Methods \ Programs	Solving Time in Seconds				
	Grep	Flex	Sed	Make	Gzip
LF_LS	1	1	1	1	1
NF_LS	1	1	1	1	1
NF_NS	n/a	n/a	n/a	530	n/a

* n/a: the solver timed out after eight hours

Table 9: Performance of different methods modeling the *tri-criteria* problem with a 10% size constraint

Methods \ Programs	Solving Time in Seconds				
	Grep	Flex	Sed	Make	Gzip
LF_LS	1	1	1	1	1
NF_LS	14099	43608	2976	2673	1565

of NF_LS and LF_LS for the *tri-criteria* problem comes from the higher complexity of the problem compared with the first two minimization problems. While those two problems have a single optimization criterion (i.e., maximizing fault-detection ability), the tri-criteria problem has an additional optimization criterion (i.e., maximizing statement coverage). To solve this significantly more complex problem, auxiliary variables in the range of hundreds of thousands were introduced in NF_LS because there are thousands of

statements for the subject apps. Nevertheless, the optimal solution found by NF_LS for this *tri-criteria* problem, as shown in Table 6, vastly outperforms the solution found by LF_LS. For Gzip under a 10% size constraint, LF_LS solved the problem within a second, while NF_LS took about 26 minutes. However, the suite reduced by NF_LS executed 159% more statements and 124% more faults than the suite reduced by LF_LS.

Although NF_LS takes longer to produce a solution, the approach produces immensely improved solutions. For instance, testers may run NF_LS for a few minutes to hours rather than a few seconds in the case of LF_LS, but they obtain a solution that reveals much more faults with fewer tests which, in turn, can save engineers' time and effort from having to examine and run more tests.

5.3 RQ2: NF_LS vs. NF_NS

Similar to the previous RQ, we compared NF_LS and NF_NS in terms of effectiveness and performance. We further discuss the advantages or disadvantages of selecting among the two nonlinear approaches, based on our empirical results.

Effectiveness. Table 3 shows that, for the *classic bi-criteria* problem, NF_NS was able to find the optimal solutions for only three subject programs—Flex, Sed, and Make. The solver timed out after eight hours for Grep, and returned a suboptimal solution for Gzip. Table 5 further shows that, for the *variant bi-criteria* problem, NF_NS only found the optimal solution for Make, and the solver timed out after eight hours for other subject programs. We do not report the effectiveness of NF_NS for the *tri-criteria* problem, because we were not able to use NEOS to solve it due to the size limitation of NEOS for uploaded files. The reported results indicate that, as expected, the nonlinear solver is not guaranteed to find an optimal solution for the nonlinear formulation if the objective function is non-convex [48]. However, NF_LS can optimally solve MCTSM problems formulated nonlinearly.

Performance. For the *classic bi-criteria* problem, Table 7 shows NF_NS takes much longer time to solve the problem than NF_LS. While NF_LS solved the problem within a second for all subject apps, the solver used by NF_NS timed out for Grep, and took from 20 seconds to six hours to finish. For the *variant bi-criteria* problem, Table 8 indicates that NF_NS was able to finish only for Make, and the solver timed out for other subjects. The results indicate that, off-the-shelf nonlinear solvers cannot solve MCTSM problems in a timely manner, which confirms the need to transform the nonlinear problem formulation to a linear one. Overall, NF_LS is superior to NF_NS in terms of both effectiveness and performance.

5.4 RQ3: Scalability

Given NF_LS's superior performance over NF_NS, we focus on studying the scalability of NF_LS. Specifically, we investigate the manner in which NF_LS scales with respect to the size of subject programs and test suites. Intuitively, the time NF_LS takes to solve a minimization problem depends on the complexity of the formulated model, in terms of the number of decision variables and constraints. This complexity is determined by the characteristics of the problem (e.g., number of test cases, requirements, and criterion entities).

We conducted a sensitivity analysis of NF_LS on the *tri-criteria* problem, i.e., the most complicated problem in our evaluation. For each program, starting from the size of 20% of the test suite, we gradually increased the size of test suites to 100%, and reported (1) the number of variables in the formulation, and (2) the execution time for solving the problem. We repeated the experiments 30 times and reported the average with a 95% confidence interval.

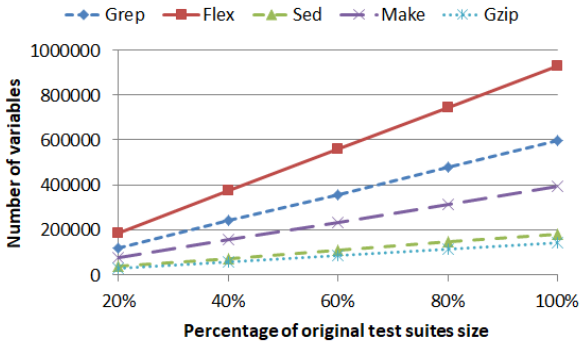

Figure 2: Sensitivity of the test-suite size to number of variables for Nemo on the *tri-criteria* problem

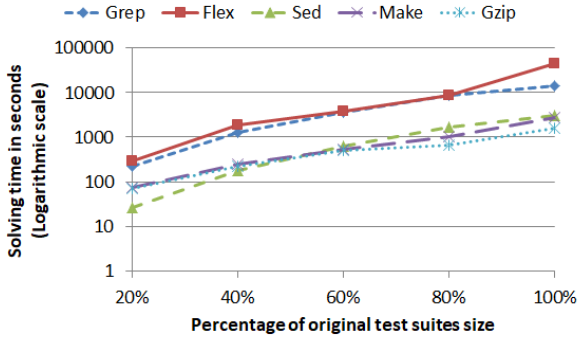

Figure 3: Sensitivity of the test-suite size to solving time for Nemo on the *tri-criteria* problem

Figure 2 depicts the relation between the size of a test suite and the number of variables used in the problem formulation. While the number of variables range in the hundreds of thousands for each problem, the number grows linearly, as does the size of the test suite.

Figure 3 depicts the relation between the size of a test suite and solver execution time. As the size of a test suite grows linearly, the time required to find an optimal solution increases exponentially. Nevertheless, NF_LS is optimal, and can be adopted to compute optimal solutions as efficiently as LF_LS for simpler MCTSM problems, such as the *classic bi-criteria* and *variant bi-criteria* problems, and optimally solves practical, complex problems, such as the *tri-criteria* problem, in a reasonable amount of time.

6 RELATED WORK

Single-criterion test-suite minimization problem. A significant number of previous studies have examined obtaining the minimal subset of the original test suite that satisfies the original test requirements [9, 10, 20, 28, 32, 36, 45]. Chvatal [10] proposes a classical greedy heuristic that iteratively selects test cases covering most unsatisfied requirements until all requirements are covered. Harrold et al. [20] present a greedy heuristic that considers the testing sets that satisfy each test requirement, and repeatedly selects test cases from the testing sets with minimal cardinality. Chen and Lau [9] propose another heuristic that identifies two special kinds of test cases from the original suite: *essential* and *1-to-1 redundant*. The approach iteratively picks essential, removes 1-to-1 redundant,

and greedily selects test cases covering a maximal number of unsatisfied requirements. Offutt et al. [36] propose a heuristic combining different test execution orderings, i.e., forward, reverse, and inside-out, to reduce the size of test suites while maintaining the mutation score or statement coverage achieved by the original suites. Another heuristic by Tallam and Gupta, called *Delayed-Greedy* [45], is based on Formal Concept Analysis and combines both perspectives of test cases and requirements. Their experiments show that Delayed-Greedy can achieve equal or better size reduction than previous heuristics. By formulating the problem as finding a spanning set over a graph, Marre and Bertolino [32] propose a technique to reduce the number of test cases required to satisfy the requirements. Leitner et al. [28] propose a technique combining program slicing and delta debugging to minimize failing test cases in randomized unit test generation, which is focused on reducing individual test cases rather than the sizes of test suites. All of these approaches focus on a single criterion and generate approximate solutions.

Multi-criteria test-suite minimization problem. While previous approaches can be adopted to efficiently find a solution, a major concern of single-criterion minimization is that minimizing a test suite could severely compromise its ability to reveal faults [39, 49]. To account for additional criteria, such as fault-detection capability, several approaches have been developed that consider additional information including hybrid combinations of different coverage criteria [41] in the minimization process and generating a reduced suite which has better effectiveness with respect to various criteria, such as fault detection [4, 5, 15, 18, 19, 26, 31, 33, 34, 42–44] and energy consumption [24, 30].

With respect to improving fault-detection effectiveness of the reduced suite, a few techniques are heuristics-based. Jeffery and Gupta [26] propose a heuristic that selectively picks redundant test cases for the reduced suite by using additional coverage information. A heuristic by Lin and Huang [31] uses an additional testing criterion to break ties in the minimization process.

In addition to the heuristics, ILP is adopted to compute solutions for the problem. Black et al. [5] formulate the problem as a binary ILP model. They directly take the fault-revealing ability of test cases into account, encode it into the objective function, and compute a solution for the problem using an ILP solver. Hao et al. [19] collect statistics on fault-detection loss at the statement level, and encode the information into the constraints of the formulated ILP model to control the fault-detection loss of the reduced suite.

Another set of previous work considers other aspects of minimizing a test suite while maximizing its fault-detection capability. This work includes using different coverage criteria such as call-stack coverage [33], adopting different reduction algorithms [42, 44]—or applying techniques such as delta-debugging [18], non-adequate reduction [4] (i.e., only a certain percentage of the original coverage is retained), or a combination of test reduction and selection [43]. Other work also addresses the trade-offs specific for reused software [34] and an industrial system [15].

With respect to criteria other than fault-revealing capability, Li et al. [30] take energy consumption of test cases into consideration, and adopt an integer programming approach to generate minimized test suites which are energy-efficient for post-deployment testing on embedded systems. Similarly, Jabbarvand et al. [24] propose an integer programming approach as well as a greedy algorithm to minimize test suites while trying to maintain the capability of the reduced suites to reveal energy bugs.

Unlike Nemo, all of these MCTSM approaches focus on specific bi-criteria problems and do not allow testers to specify a wide range of MCTSM problems, cannot compute optimal solutions for them, and cannot deal with dependencies between test cases over any testing criteria.

Hsu and Orso [23] proposed MINTS, a framework for MCTSM, that is able to accommodate an arbitrary number of objectives and provides flexibility for testers to combine, weight, and prioritize their objectives. Recall that LF_LS is a re-implementation of MINTS, since the original implementation uses outdated solvers and does not scale to our experimental dataset (see Section 5). Their approach formulates the problem as one or more ILP problems, in which the test requirements are encoded as constraints, and the objectives can be either associated as weights in objective functions, or prioritized as invocation orders of ILP problems. While their work focuses on a generalized approach for MCTSM, their problem formulation is linear, which, as shown both in theory (Section 2) and empirically (Section 5) in this paper, produces sub-optimal solutions, due to the inability to model dependencies among test cases over specified criteria, which must be modeled nonlinearly.

7 CONCLUSION AND FUTURE WORK

Multi-criteria test-suite minimization techniques help reduce the cost of regression testing by removing redundant tests based on some criteria, while trying to optimally keep the capability of the reduced suite in terms of other criteria. All of the existing approaches suffer from at least one of the two shortcomings discussed in this paper: (1) they either use heuristic algorithms or ignore test-case dependencies among minimization criteria, and hence generate approximate or suboptimal solutions; and (2) they handle minimization problems involving only limited and pre-specified criteria.

In this paper, we proposed a general framework for the multi-criteria test-suite minimization problem. Our approach takes into account the inherent nonlinearity of the problem, and models it using integer nonlinear programming. To solve the nonlinear formulation optimally, we developed a technique that programmatically transforms it to a linear form, so that modern ILP solvers can be leveraged. We have implemented our approach as a tool, called Nemo, and empirically evaluated it. Our experimental results show that Nemo can be used to find optimal solutions for several minimization problems within a reasonable time. Nemo was able to produce reduced test suites that could execute up to 159% more statements and detect 124% more faults than those produced using prior work.

An interesting direction for future work is to investigate applicability of our approach in other test maintenance activities, such as test selection and prioritization. We also plan to conduct experiments involving more complex criteria, such as MC/DC, to assess the effectiveness of a nonlinear approach in satisfying such criteria in test-suite minimization. Finally, we plan to empirically evaluate the fault-detection ability of the reduced test suites when executed on previously unseen faults.

ACKNOWLEDGEMENT

This work was supported in part by awards CCF-1252644, CNS-1629771 and CCF-1618132 from the National Science Foundation, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

REFERENCES

[1] http://lpsolve.sourceforge.net/5.5/
[2] https://dropsolve-oaas.docloud.ibmcloud.com/
[3] http://www.ics.uci.edu/~seal/projects/nemo/index.html
[4] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. 2016. Evaluating Non-adequate Test-case Reduction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 16–26. https://doi.org/10.1145/2970276.2970361
[5] J. Black, E. Melachrinoudis, and D. Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *Proceedings. 26th International Conference on Software Engineering*. 106–115. https://doi.org/10.1109/ICSE.2004.1317433
[6] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 105–115. https://doi.org/10.1145/2610384.2628058
[7] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.
[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756
[9] Tsong Yueh Chen and Man Fai Lau. 1996. Dividing Strategies for the Optimization of a Test Suite. *Inf. Process. Lett.* 60, 3 (Nov. 1996), 135–141. https://doi.org/10.1016/S0020-0190(96)00135-4
[10] V. Chvatal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Math. Oper. Res.* 4, 3 (Aug. 1979), 233–235. https://doi.org/10.1287/moor.4.3.233
[11] Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. 2013. A Learning-based Method for Combining Testing Techniques. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 142–151. http://dl.acm.org/citation.cfm?id=2486788.2486808
[12] Couenne. Retrieved July 27, 2017 from https://projects.coin-or.org/Couenne/
[13] IBM ILOG CPLEX. Retrieved July 27, 2017 from https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[14] J. Czyzyk, M. P. Mesnier, and J. J. More. 1998. The NEOS Server. *IEEE Computational Science and Engineering* 5, 3 (Jul 1998), 68–75. https://doi.org/10.1109/99.714603
[15] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 (2015), 371–396.
[16] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.
[17] Gcov. Retrieved July 27, 2017 from https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[18] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause Reduction: Delta Debugging, Even Without Bugs. *Softw. Test. Verif. Reliab.* 26, 1 (Jan. 2016), 40–68. https://doi.org/10.1002/stvr.1574
[19] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and G. Rothermel. 2012. On-demand test suite reduction. In *2012 34th International Conference on Software Engineering (ICSE)*. 738–748. https://doi.org/10.1109/ICSE.2012.6227144
[20] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (Jul 1993), 270–285. https://doi.org/10.1145/152388.152391
[21] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-box and Black-box Test Prioritization. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 523–534. https://doi.org/10.1145/2884781.2884791
[22] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less Without Sacrificing Quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 483–493. http://dl.acm.org/citation.cfm?id=2818754.2818815
[23] H. Y. Hsu and A. Orso. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *2009 IEEE 31st International Conference on Software Engineering*. 419–429. https://doi.org/10.1109/ICSE.2009.5070541
[24] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware Test-suite Minimization for Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 425–436. https://doi.org/10.1145/2931037.2931067
[25] P. Jaccard. 1901. Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles* 37 (1901), 241–272.
[26] Dennis Jeffrey and Neelam Gupta. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. *IEEE Trans. Softw. Eng.* 33, 2 (Feb 2007), 108–123. https://doi.org/10.1109/TSE.2007.18
[27] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 474–484. http://dl.acm.org/citation.cfm?id=2337223.2337279
[28] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient Unit Test Case Minimization. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 417–420. https://doi.org/10.1145/1321631.1321698
[29] H. K. N. Leung and L. White. 1991. A cost model to compare regression test strategies. In *Proceedings. Conference on Software Maintenance 1991*. 201–208. https://doi.org/10.1109/ICSM.1991.160330
[30] Ding Li, Yuchen Jin, Cagri Sahin, James Clause, and William G. J. Halfond. 2014. Integrated Energy-directed Test Suite Optimization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 339–350. https://doi.org/10.1145/2610384.2610414
[31] Jun-Wei Lin and Chin-Yu Huang. 2009. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology* 51, 4 (Apr 2009), 679–690. https://doi.org/10.1016/j.infsof.2008.11.004
[32] M. Marre and A. Bertolino. 2003. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering* 29, 11 (Nov 2003), 974–984. https://doi.org/10.1109/TSE.2003.1245299
[33] Scott McMaster and Atif M. Memon. 2008. Call-Stack Coverage for GUI Test-Suite Reduction. *IEEE Trans. Softw. Eng.* (2008).
[34] Breno Miranda and Antonia Bertolino. 2017. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software* 131 (2017), 528 – 549. https://doi.org/10.1016/j.jss.2016.06.058
[35] S. Mirarab, S. Akhlaghi, and L. Tahvildari. 2012. Size-Constrained Regression Test Case Selection Using Multicriteria Optimization. *IEEE Transactions on Software Engineering* 38, 4 (Jul 2012), 936–956. https://doi.org/10.1109/TSE.2011.56
[36] A Jefferson Offutt, Jie Pan, and Jeffrey M Voas. 1995. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw.*
[37] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 936–946. http://dl.acm.org/citation.cfm?id=2818754.2818867
[38] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and Early Fault Detection with Lower and Higher Strength Combinatorial Interaction Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/2491411.2491436
[39] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance*. 34–43. https://doi.org/10.1109/ICSM.1998.738487
[40] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct 2001), 929–948. https://doi.org/10.1109/32.962562
[41] S. Sampath, R. Bryce, and A. M. Memon. 2013. A Uniform Representation of Hybrid Criteria for Regression Testing. *IEEE Transactions on Software Engineering* 39, 10 (Oct 2013), 1326–1344. https://doi.org/10.1109/TSE.2013.16
[42] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-offs in Test-suite Reduction. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/2635868.2635921
[43] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-suite Reduction and Regression Test Selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/2786805.2786878
[44] S. Sprenkle, Sreedevi Sampath, E. Gibson, L. Pollock, and A. Souter. 2005. An empirical comparison of test suite reduction techniques for user-session-based testing of Web applications. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 587–596. https://doi.org/10.1109/ICSM.2005.18
[45] Sriraman Tallam and Neelam Gupta. 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 35–42. https://doi.org/10.1145/1108792.1108802
[46] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. TimeAware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1146238.1146240
[47] H Paul Williams. 2013. *Model building in mathematical programming*. John Wiley & Sons.
[48] L. A. Wolsey. 1998. *Integer programming*. Wiley-Interscience, New York, NY, USA.
[49] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *Proceedings of the 17th International Conference on Software Engineering (ICSE '95)*. ACM, 41–50. https://doi.org/10.1145/225014.225018
[50] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (Mar 2012), 67–120. https://doi.org/10.1002/stv.430
[51] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-Aware Test-Case Prioritization using Integer Linear Programming. In *Proc. International Conference on Software Testing and Analysis (ISSTA 2009)*.