The Journal of Systems & Software 168 (2020) 110632

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An effective formulation of the multi-criteria test suite minimization problem

O. Örsan Özener, Hasan Sözer *

Ozyegin University, Istanbul, Turkey

ARTICLE INFO

Article history:
Received 12 September 2019
Received in revised form 26 April 2020
Accepted 5 May 2020
Available online 8 May 2020

Keywords:
Software testing
Regression testing
Test suite minimization
Integer programming
Multi-objective optimization

ABSTRACT

Test suite minimization problem has been mainly addressed by employing heuristic techniques or integer linear programming focusing on a specific criterion or bi-criteria. These approaches fall short to compute optimal solutions especially when there exists overlap among test cases in terms of various criteria such as code coverage and the set of detected faults. Nonlinear formulations have also been proposed recently to address such cases. However, these formulations require significantly more computational resources compared to linear ones. Moreover, they are also subject to shortcomings that might still lead to sub-optimal solutions. In this paper, we identify such shortcomings and we propose an alternative formulation of the problem. We have empirically evaluated the effectiveness of our approach based on a publicly available dataset and compared it with respect to the state-of-the-art based on the same objective function and the same set of criteria including statement coverage, fault-revealing capability, and test execution time. Results show that our formulation leads to either better results or the same results, when the previously obtained results were already the optimal ones. In addition, our formulation is a linear formulation, which can be solved much more efficiently compared to non-linear formulations.

1. Introduction

Software systems evolve. They have to be maintained over time and usually extended with new features. Regression testing is performed to ensure that maintenance activities and extensions do not introduce bugs or side effects (Yoo and Harman, 2012).

The size of test suites used for regression testing tends to increase over time. It becomes too costly, if not infeasible, to execute the whole test suite for large-scale systems. The testing process might involve millions of test executions (Herzig et al., 2015) and take multiple weeks to complete. For example, it was previously reported (Rothermel et al., 2001) that the entire test suite of an actual product requires 7 weeks to run, although the product comprises around 20,000 lines of code. Test case selection, test case prioritization and test suite minimization approaches have been proposed to address this problem (Yoo and Harman, 2012).

In this paper, we target at test suite minimization, which strives for finding and eliminating redundant test cases to reduce the total number of tests (Yoo and Harman, 2012). Hereby, the problem is to find the minimal subset of the test suite, which satisfies some criteria. Optimization with respect to a single criterion can lead to a test suite that severely compromises its effectiveness in terms of other criteria (Rothermel et al., 1998). Therefore, we focus on, namely, the multi-criteria test suite minimization (MCTSM) problem (Lin et al., 2018) that requires the consideration of multiple criteria such as code coverage, fault detection capability and cost.

The MCTSM problem has been mainly addressed by employing heuristic techniques (Jeffrey and Gupta, 2007; Lin and Huang, 2009) or integer linear programming (Hsu and Orso, 2009; Black et al., 2004; Hao et al., 2012; Li et al., 2014; Jabbarvand et al., 2016; Nardo et al., 2015; Miranda and Bertolino, 2017). These approaches fall short to compute optimal solutions especially when there are dependencies among test cases in terms of various criteria. For instance, simply maximizing the number of detected faults would not necessarily lead to an efficient test suite if the same fault could be detected by multiple test cases. It must be ensured that test cases cover a diverse set of faults. A nonlinear formulation was recently proposed (Lin et al., 2018) based on this observation. Although this formulation significantly improves the results with respect to prior work, we argue that it is still subject to shortcomings. We demonstrate both analytically on small instances and empirically on large test instances that it may fail to identify the optimal solution. We propose an alternative formulation of the problem that addresses the identified shortcomings. Besides, our formulation is a linear formulation, which

* Corresponding author.
E-mail address: hasan.sozer@ozyegin.edu.tr (H. Sözer).

https://doi.org/10.1016/j.jss.2020.110632
0164-1212/

does not require a linear transformation and it can be solved much more efficiently compared to non-linear formulations.

We empirically evaluated the effectiveness of our approach based on a publicly available dataset (Lin et al., 2018). This dataset was previously created based on 5 open-source projects (Henard et al., 2016) and it was used for evaluating solution approaches for the MCTSM problem. We used the same dataset and the same experimental setup to compare our approach with respect to previous studies. That is, we used the same objective function and the same set of criteria including statement coverage, fault-revealing capability, and test execution time. Results show that our formulation leads to either better results or the same results, when the previously obtained results were already the optimal ones. In addition, it leverages better time efficiency as expected.

The contributions of this paper are threefold: *(i)* we identify deficiencies in state-of-the-art MCTSM problem formulations, *(ii)* we propose a novel formulation for optimally solving this problem, and *(iii)* we present an empirical study in which the proposed approach is compared with previous studies based on a publicly available dataset obtained from a set of open-source projects.

The remainder of this paper is organized as follows. In the next section, we briefly explain the MCTSM problem, previously proposed formulations for the problem and their deficiencies. We introduce our formulation for the problem in Section 3. We present an empirical evaluation of our approach and a comparison with other approaches in Section 4. We summarize related studies in Section 5. Finally, we provide our concluding remarks and future research directions in Section 6.

2. Multi-criteria test-suite minimization

There exist 3 basic approaches for increasing the effectiveness of the regression testing process (Yoo and Harman, 2012): *(i)* test suite minimization, *(ii)* test case selection, and *(iii)* test case prioritization. Test case prioritization aims at finding an optimal execution order of the test cases in a test suite according to an objective such as detecting faults as early as possible. Test case selection is performed to select a subset of test cases within a test suite that are associated with the changed parts of the software. The goal of test suite minimization is to identify and then eliminate obsolete or redundant test cases from the test suite. Hence, a subset of the test cases are (de)selected as well. However, test suite minimization can be considered as a more general approach in the sense that the selection criterion is not fixed such as focusing on changed parts of the system only. It is also sometimes called as "test suite reduction" (Rothermel et al., 2002; Zhang et al., 2011; Shi et al., 2014, 2018) to essentially refer to the same type of approach (Yoo and Harman, 2012).

MCTSM is a variant of the test suite minimization problem, where multiple criteria can be considered for selecting and eliminating test cases. It is a multi-objective optimization problem, which can be formalized as follows by adopting the weighted-sums approach (Steuer, 1986).

Given:

- A test suite $T = \{t_1, t_2, t_3, \ldots, t_n\}$
- A set of constraint criteria $C = \{c_1, c_2, c_3, \ldots, c_k\}$
- A set of optimization criteria $O = \{o_1, o_2, o_3, \ldots, o_l\}$
- A set of weights of importance associated with the optimization criteria $W = \{w_1, w_2, w_3, \ldots, w_l\}$

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

- T' satisfies C
- $\forall T''$ that satisfies C, $\sum_{i=1}^{l} w_i \times e_i(T'') \leq \sum_{i=1}^{l} w_i \times e_i(T')$

Hereby, each function e_i provides an evaluation of a given test suite with respect to the optimization criteria o_i. One obvious criterion, by definition, would be to minimize the test suite size. However, there can be others considering, for instance, code coverage and cost. Various combinations of such criteria and constraints lead to an open-ended number of instances of MCTSM. In this paper, we propose a generic approach to formulate this problem. Then, we adapt our formulation and evaluate our approach based on 3 instances (Section 4), which were previously used for evaluating MCTSM approaches (Lin et al., 2018). They are described in the following.

Classic bi-criteria (CB). This problem has two objectives with equal weights: *(i)* minimize the number of test cases, and *(ii)* maximize the fault-detection ability. As a constraint, the statement coverage of the reduced test suite must be equal to the statement coverage of the original test suite.

Variant bi-criteria (VB). This problem employs the same set of objectives and the same constraint as *CB*. In addition, it incorporates an additional constraint based on the observation that some segments of code are more important than others, and as such, they need to be covered multiple times (Jin and Orso, 2012; Jabbarvand et al., 2016). Hereby, importance of a statement is assumed to be proportional to the number of times it is executed by the original test suite. For instance, an additional constraint is specified for each of the top 10% mostly executed statements, where the statement must be executed at least 10% of the number of times it is executed by the original test suite.

Tri-criteria (TC). This problem has two objectives with equal weights: *(i)* maximize statement coverage, and *(ii)* maximize the fault-detection ability. As a constraint, the total test cost must be less than a given budget. The amount of cost for each test case is provided as input, which might be specified in terms of execution time (Walcott et al., 2006; Zhang et al., 2009). The budget limit is specified as a percentage of the total cost for the original test suite.

In general, MCTSM problem is NP-complete, as it can be reduced from the minimum set-covering problem in polynomial time (Garey and Johnson, 1979). In the following subsections, we introduce previously proposed integer programming formulations of the MCTSM problem. We also introduce adversary examples to demonstrate how these formulations can lead to sub-optimal results. Then, we explain our formulation in the next section.

2.1. Linear formulation

State-of-the-art linear modeling of MCTSM (Black et al., 2004; Hsu and Orso, 2009) adopt a binary integer linear programming formulation of the problem. Hereby, a binary decision variable, d_i is associated with each test case, t_i within the test suite, T. d_i takes the value 1 if t_i is selected as part of the reduced test suite, T', and 0 otherwise. In the following, we present a motivating example adopted from Lin et al. (2018) to illustrate this approach and its shortcomings.

Table 1 lists the statement and fault coverage regarding 3 test cases, t_1, t_2, t_3. There are 3 statements ($S = \{s_1, s_2, s_3\}$) and 4 faults ($F = \{f_1, f_2, f_3, f_4\}$). In the first part of the table, we see that the first statement, s_1 is executed by t_1 and t_3. The second statement, s_2 is executed only by t_2, while s_3 is executed by both t_2 and t_3. In the second part of the table, we see that the first 3 faults are detected by t_2 and t_3, whereas the last fault, f_4 can be detected only by t_1. In this example, there exist only one constraint criterion, c_1, to maintain the same statement coverage as the original test suite. There is also a single optimization criterion, o_1, to maximize the fault detection effectiveness.

Table 1
First adversary example (adopted from Lin et al. (2018)).

Coverage		Test cases		
		t_1	t_2	t_3
S	s_1	1	0	1
	s_2	0	1	0
	s_3	0	1	1
F	f_1	0	1	1
	f_2	0	1	1
	f_3	0	1	1
	f_4	1	0	0

The constraint criterion c_1 is formulated as follows:

$$\sum_{i=1}^{n} \sigma_{ij} \times d_i \geq 1, 1 \leq j \leq |S| \tag{1}$$

Hereby, $|S|$ represents the number of statements and σ_{ij} is a binary variable that takes the value 1 if the statement s_j is executed by the test case t_i, and 0 otherwise. Each decision variable, d_i is associated with the (de)selection of the corresponding test case t_i. The interpretation of Eq. (1) for the example case listed in Table 1 leads to the following concrete constraints:

$$d_1 + d_3 \geq 1$$
$$d_2 \geq 1$$
$$d_2 + d_3 \geq 1$$

The optimization criterion o_1 is formulated as follows:

$$min \sum_{i=1}^{n} \epsilon(t_i) \times d_i \tag{2}$$

This linear objective function tries to minimize the size of the reduced test suite. The function ϵ evaluates the fault detection **incapability** of a given test case. The more faults detected by a test case t_i, the smaller the value gets returned by $\epsilon(t_i)$. Hence, this function is formulated as follows:

$$\epsilon(t_i) = 1 - \frac{\sum_{j=1}^{|F|} v_{ij}}{|F|} \tag{3}$$

Hereby, $|F|$ represents the number of faults and v_{ij} is a binary variable that takes the value 1 if the fault f_j is detected by the test case t_i, and 0 otherwise. The objective function for the example given in Table 1 turns out to be the following:

$$min((1 - \frac{1}{4}) \times d_1 + (1 - \frac{3}{4}) \times d_2 + (1 - \frac{3}{4}) \times d_3)$$

The optimal solution for this formulation turns out to be $\{t_2, t_3\}$ with 1/2 as the value of the objective function. However, this is actually not the optimal solution for the problem. We can see in Table 1 that test cases t_2 and t_3 both detect f_1, f_2 and f_3, while they miss f_4. The optimal solution for this example case is $\{t_1, t_2\}$. These two test cases cover all the statements and detect all the faults although the objective function evaluates to 1. This linear formulation tends to select test cases that detect more number of faults in total; however, it does not maximize the number of distinct faults detected. Although t_1 detects f_4 only, it should have been selected together with t_2 since f_4 can only be detected with t_1 and all the other faults can be detected by t_2 anyhow.

The linear formulation discussed in this subsection can be provided to a linear solver. We refer to this solution approach (Hsu and Orso, 2009) as *LF_LS* (Linear Formulation–Linear Solver) in the rest of this paper. In the following, we explain the state-of-the-art nonlinear formulation approach.

2.2. Nonlinear formulation

The recently proposed nonlinear formulation (Lin et al., 2018) takes test case dependencies into account. That is, faults detected by multiple test cases are also considered to maximize the number of distinct faults detected. In the following, we illustrate an instance of this formulation for the example case provided in the previous subsection, where there is a single constraint criterion (c_1) to maintain the same statement coverage as the original test suite, and there is a single objective (o_1) to maximize the fault detection effectiveness. As a result, the overall objective function turns out to be the same as Eq. (2), except the evaluation function (ϵ), which makes the formulation nonlinear. The refined function is defined as follows.

$$\epsilon(t_i) = 1 - \frac{\sum_{j=1}^{|F|} v_{ij} \times h_{ij}}{|F|}, \tag{4}$$

The evaluation function for the linear formulation was defined in Eq. (3). The only difference between Eq. (3) and Eq. (4) is the multiplying factor, h_{ij}, which is defined as follows.

$$h_{ij} = \prod_{\substack{t=1 \\ t \neq i \wedge v_{tj}=1}}^{n} (1 - d_t) \tag{5}$$

Eq. (5) iterates over all the test cases $(t_1, t_2, \ldots, t_n)$ that detects f_j (i.e., $v_{tj} = 1$), except t_i. If at least one of these test cases is selected, $(1 - d_t)$ becomes 0, which makes the whole equation evaluate to 0. Hence, it becomes not favorable to select t_i since it detects a fault that is already detected by some other selected test case.

The objective function for the example given in Table 1 turns out to be the following:

$$\begin{aligned} \min(& \\ & (1 - \frac{\sum_{j=1}^{4} v_{1j} \times h_{1j}}{4}) \times d_1 + \\ & (1 - \frac{\sum_{j=1}^{4} v_{2j} \times h_{2j}}{4}) \times d_2 + \\ & (1 - \frac{\sum_{j=1}^{4} v_{3j} \times h_{3j}}{4}) \times d_3 \\ &) \end{aligned} \tag{6}$$

The first test case detects only one fault, f_4 that is not detected by any other test case. On the other hand, the second and third test cases both detect f_1, f_2 and f_3. Hence, Eq. (6) can be expanded as follows.

$$\begin{aligned} & \min((1 - 1/4) \times d_1 + \\ & (1 - 1/4 \times ((1 - d_3) + (1 - d_3) + (1 - d_3))) \times d_2 + \\ & (1 - 1/4 \times ((1 - d_2) + (1 - d_2) + (1 - d_2))) \times d_3) \end{aligned}$$

The set of constraints remains the same as defined in Eq. (1). The optimal solution for this formulation turns out to be $\{t_1, t_2\}$ with 1 as the value of the objective function. This is indeed the optimal solution for the example case. The solution selected by the linear formulation ($\{t_2, t_3\}$) leads to value 2 for the revised objective function.

Two alternatives were proposed for solving this nonlinear formulation (Lin et al., 2018). The first alternative is to use a nonlinear solver. This approach is referred as *NF_NS* (Nonlinear Formulation–Nonlinear Solver). The second alternative is to transform this formulation to linear programming and then use a linear solver. We do not explain the employed transformation process and refer the reader to the corresponding publication (Lin et al., 2018) for details. This second approach is referred as *NF_LS* (Nonlinear Formulation–Linear Solver).

Table 2
Second adversary example.

Coverage		Test cases		
		t_1	t_2	t_3
S	s_1	1	0	0
	s_2	0	1	0
	s_3	1	1	1
F	f_1	1	1	0
	f_2	0	0	1
	f_3	0	1	0
	f_4	0	0	1

Although the proposed nonlinear formulation takes test case dependencies into account, it can still lead to sub-optimal solutions as illustrated in the following.

We introduce a second adversary example, where the set of criteria and constraints are the same as those used in the first example. That is, there exist only one constraint criterion, c_1, to maintain the same statement coverage as the original test suite. There is also a single optimization criterion, o_1, to maximize the fault detection effectiveness. Table 2 lists the statement and fault coverage regarding 3 test cases, t_1, t_2, t_3. There are 3 statements and 4 faults. In the first part of the table, we see that the first statement, s_1 is executed only by t_1. The second statement, s_2 is executed only by t_2, while s_3 is executed by all. In the second part of the table, we see that the first fault is detected by t_1 and t_2, second only by t_3, third only by t_2, and finally fourth only by t_3.

Hence, Eq. (6) for this example can be expanded as follows.

$$\begin{aligned}&\min((1-1/4\times(1-d_2))\times d_1+\\&(1-1/4-1/4\times(1-d_1))\times d_2+\\&(1-2/4)\times d_3)\end{aligned}$$

The statement coverage constraint requires the selection of t_1 and t_2 as s_1 is covered only by t_1 and s_2 is covered only by t_2. On the other hand, t_3 may not be selected at all as it only covers s_3 which is already covered by the first and second test already. The optimal solution for this formulation turns out to be $\{t_1, t_2\}$, where the above equation yields to 1 as the minimum value. However, with the single optimization criterion of maximizing the fault detection effectiveness, the optimal solution to this example should have been selecting all the tests as the last test detects two faults, f_2 and f_4 namely, which cannot be detected by the other tests.

We would stuck at a sub-optimal solution in this example even if we adopted *CB*, where the goal is to minimize the number of test cases as well as maximize the fault-detection ability. Hereby, the last test should still be selected as it covers two faults while increasing the number of test cases selected by one. Hence, we prove that nonlinear formulation may not yield the optimal solution for certain instances of the problem.

Note that the existing linear formulation, *LF_LS* would also select t_1 and t_2 in this example case, for which the objective function can be expanded as follows.

$$\min((1-\frac{1}{4})\times d_1+(1-\frac{2}{4})\times d_2+(1-\frac{2}{4})\times d_3)$$

Hereby, d_1 and d_2 must be equal to 1 since the first two test cases must be selected to be able to cover all the statements. The last decision variable, d_3 will be set to 0 to be able to minimize the objective function value.

Next, we present a third adversary example (Table 3), where the nonlinear formulation fails to identify the optimal solution. In this example, there exists a test, t_1 that covers all the statements but does not detect any fault at all. The other three tests cover one statement and detect two distinct faults each.

Table 3
Third adversary example.

Coverage		Test cases			
		t_1	t_2	t_3	t_4
S	s_1	1	1	0	0
	s_2	1	0	1	0
	s_3	1	0	0	1
F	f_1	0	1	0	0
	f_2	0	1	0	0
	f_3	0	0	1	0
	f_4	0	0	1	0
	f_5	0	0	0	1
	f_6	0	0	0	1

Hence, Eq. (6) for this example can be expanded as follows.

$$\begin{aligned}&\min(d_1+\\&(1-2/6)\times d_2+\\&(1-2/6)\times d_3+\\&(1-2/6)\times d_4)\end{aligned}$$

The statement coverage constraint requires either $\{t_1\}$ or $\{t_2, t_3, t_4\}$ as feasible solutions. The objective function value for these solutions are 1 and $\frac{2}{3}\times 3 = 2$, respectively. Thus, the optimal solution turns out to be $\{t_1\}$ since we are trying to minimize the objective function value. However, with the single optimization criterion of maximizing the fault detection effectiveness, the optimal solution to this example should have been selecting all the tests but the first one as the last three tests detect two faults each whereas the first test detects none. Even for the *CB* problem variant, the last three tests should still be selected as each test covers two faults while increasing the number of test cases selected by one. Hence, we prove that nonlinear formulation does not yield the optimal solution for such instances as well.

Note that the existing linear formulation, *LF_LS* would also select only t_1 in this example case, for which the objective function can be expanded as follows.

$$\min(d_1+(1-\frac{1}{3})\times d_2+(1-\frac{1}{3})\times d_3+(1-\frac{1}{3})\times d_4)$$

The values of this function becomes equal to 1 when d_1 is set to 1 and all the other decision variables are set to 0. The value becomes equal to 2 when d_1 is set to 0 and all the other decision variables are set to 1. Hence, the resulting test suite would be $\{t_1\}$ as the solution with the minimum value.

In the following, we introduce our approach and then later compare it with respect to all the previously proposed alternatives, namely *LF_LS*, *NF_NS*, and *NF_LS*.

3. A novel formulation of the problem

We propose a novel formulation for the MCTSM problem. First of all, our formulation is a linear formulation. Hence, it does not require a linear transformation and it can be solved much more efficiently compared to nonlinear formulations. Second, our formulation can easily handle the adversary cases discussed above for nonlinear formulation and identify the best solution in those cases. In fact, provided that we can find a solution using our formulation,[1] we obtain "the optimal" solution for the problem in consideration. Thus, any other algorithm cannot find a better solution. The same statement does not hold for the nonlinear formulation discussed in the previous section. We demonstrated both analytically on small instances and empirically on large test instances (See Section 4) that it may fail to identify the

[1] Finding the optimal solution may not be always possible due to the size of the instance given that the underlying problem is NP-Hard.

optimal solution. Finally, our model determines the optimal solution in the order of seconds for even larger problems with 58,344 lines of code and 746 tests, while running on a laptop computer (Section 4). Even though the problem itself is NP-Hard, our solution method is computationally efficient, beating the state-of-the-art (Lin et al., 2018) in terms of time efficiency as well.

In our mathematical formulation, instead of devising a "good" objective function that handles all the criteria imposed to the problem, we use a "penalty-based approach" and account for certain issues such as not detecting a fault or not covering a statement in specialized constraints with certain binary indicator variables and penalize those cases in the objective function accordingly by using the corresponding indicator variables. Once again, the resulting model is a binary linear model as opposed to a nonlinear model, which is easier to handle by definition.

We present 3 variants of our formulation, *FCB*, *FVB* and *FTC* for MCTSM problem variants *CB*, *VB* and *TC*, respectively. In the following, we introduce the notation used in these formulations, followed by the definition of *FCB*.

$$x_i = \begin{cases} 1, & \text{if test } i \text{ is performed} \\ 0, & \text{otherwise} \end{cases} \qquad \forall i \in T$$

$$f_j = \begin{cases} 1, & \text{if fault } j \text{ is not detected} \\ 0, & \text{otherwise} \end{cases} \qquad \forall j \in F$$

$$FCB: z = \min \sum_{i \in T} x_i + \sum_{j \in F} f_j, \tag{7}$$

$s.t.$

$$\sum_{i \in T} \sigma_{is} x_i \geq 1, \quad \forall s \in S, \tag{8}$$

$$\sum_{i \in T} v_{ij} x_i + f_j \geq 1, \quad \forall j \in F, \tag{9}$$

$$x_i \in \{0, 1\}, \quad \forall i \in T, \tag{10}$$

$$f_j \in \{0, 1\}, \quad \forall j \in F. \tag{11}$$

The objective is to minimize the sum of the number of test cases performed and the number of faults that cannot be detected by the selected tests. Note that minimizing the number of faults that cannot be detected by the reduced test suite is equivalent to maximizing the number of faults detected by the test suite as the total number of faults is constant. Constraints (8) ensure that the reduced test suite covers all the statements covered by the original test suite. Constraints (9) guarantee that a fault is either detected by the tests in the reduced test suite determined by the model or an indicator binary variable is equal to 1 showing that fault is not detected by the reduced test suite. If the latter is the case, it is accordingly penalized in the objective function. Finally, constraints (10) and constraints (11) are the binary restrictions for the decision variables. The optimal solution to *FCB* yields the best solution for *CB*.

The proposed model is also quite versatile in the sense that *(i)* it can handle different objective functions (i.e., unequal weights of the two criteria corresponds to merely multiplying each summation statement in the objective function value with the desired weights), *(ii)* it can easily incorporate additional constraints imposed to the problem (i.e., limiting the number of faults undetected, limiting the number of tests selected, etc.). These additional limitations does not complicate the solution process. In fact, they might even improve the solution time of our proposed method.

Next, we would like demonstrate how our model handles the last two adversary examples discussed before. For the second adversary example (Table 2), we have the following formulation (excluding the binary restrictions).

$$FCB: z = \min x_1 + x_2 + x_3 + f_1 + f_2 + f_3 + f_4, \tag{12}$$

$s.t.$

$$x_1 \geq 1, \tag{13}$$

$$x_2 \geq 1, \tag{14}$$

$$x_1 + x_2 + x_3 \geq 1, \tag{15}$$

$$x_1 + x_2 + f_1 \geq 1, \tag{16}$$

$$x_3 + f_2 \geq 1, \tag{17}$$

$$x_2 + f_3 \geq 1, \tag{18}$$

$$x_3 + f_4 \geq 1. \tag{19}$$

Due to first two constraints, the first two tests should be selected, $x_1 = x_2 = 1$, and after eliminating the redundant constraints the formulation reduces to:

$$FCB: z = \min 2 + x_3 + f_1 + f_2 + f_3 + f_4, \tag{20}$$

$s.t.$

$$x_3 + f_2 \geq 1, \tag{21}$$

$$x_3 + f_4 \geq 1. \tag{22}$$

Hence, in order to satisfy the remaining two constraints either $x_3 = 1$ or $f_2 = f_4 = 1$, while the former increases the objective function value by one whereas the latter increases by two. As the problem is a minimization problem, the model selects $x_3 = 1$, finding the optimal solution for the *CB* problem.

For the last adversary example (Table 3), we have the following formulation (excluding the binary restrictions).

$$FCB: z = \min x_1 + x_2 + x_3 + x_4 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6, \tag{23}$$

$s.t.$

$$x_1 + x_2 \geq 1, \tag{24}$$

$$x_1 + x_3 \geq 1, \tag{25}$$

$$x_1 + x_4 \geq 1, \tag{26}$$

$$x_2 + f_1 \geq 1, \tag{27}$$

$$x_2 + f_2 \geq 1, \tag{28}$$

$$x_3 + f_3 \geq 1, \tag{29}$$

$$x_3 + f_4 \geq 1, \tag{30}$$

$$x_4 + f_5 \geq 1. \tag{31}$$

$$x_4 + f_6 \geq 1. \tag{32}$$

Hence, due to the last six constraints, we should have either $x_2 = 1$ or $f_1 = f_2 = 1$, either $x_3 = 1$ or $f_3 = f_4 = 1$, and either $x_4 = 1$ or $f_5 = f_6 = 1$. As the objective function value coefficients are the same, we would have $x_2 = x_3 = x_4 = 1$ and $f_1 = f_2 = f_3 = f_4 = f_5 = f_5 = 0$ rather than the other way around. As the problem is a minimization problem, the model selects $x_1 = 0$ as the first three constraints are already satisfied with $x_2 = x_3 = x_4 = 1$, finding the optimal solution for the *CB* problem.

For the second MCTSM variant, *VB*, we propose another mathematical model, *FVB*. Before presenting the formulation of *FVB*, we introduce additional notation used in this formulation.

S_{10} = Set of top 10% mostly executed statements
k_s = Number of times the statement s is executed by the original test suite, $\forall s \in S_{10}$

$$FVB: z = \min \sum_{i \in T} x_i + \sum_{j \in F} f_j, \tag{33}$$

$$s.t.$$

$$\sum_{i \in T} \sigma_{is} x_i \geq 1, \quad \forall s \in S, \tag{34}$$

$$\sum_{i \in T} v_{ij} x_i + f_j \geq 1, \quad \forall j \in F, \tag{35}$$

$$\sum_{i \in T} \sigma_{is} x_i \geq 0.1 k_s, \quad \forall s \in S_{10}, \tag{36}$$

$$x_i \in \{0, 1\}, \quad \forall i \in T, \tag{37}$$

$$f_j \in \{0, 1\}, \quad \forall j \in F. \tag{38}$$

The additional constraints, constraints (36), ensure that each of the top 10% mostly executed statements is executed by the reduced test suite at least 10% of the number of times it is executed by the original test suite. The optimal solution to *FVB* yields the best solution for the *VB* problem. As in *FCB*, *FVB* can also be easily modified to handle additional criteria and restrictions.

Finally, we propose, *FTC*, for the last MCTSM variant, *TC*. Before presenting the formulation of *FTC*, we again introduce additional notation used in this formulation.

B = Budget limit as a percentage of the total cost for the original test suite

β_i = Cost of performing test $i, \forall i \in T$

$$l_s = \begin{cases} 1, & \text{if statement } s \text{ is not covered} \\ 0, & \text{otherwise} \end{cases} \qquad \forall s \in S$$

$$FTC: z = \min \sum_{s \in S} l_s + \sum_{j \in F} f_j, \tag{39}$$

$$s.t.$$

$$\sum_{i \in T} \sigma_{is} x_i + l_s \geq 1, \quad \forall s \in S, \tag{40}$$

$$\sum_{i \in T} v_{ij} x_i + f_j \geq 1, \quad \forall j \in F, \tag{41}$$

$$\sum_{i \in T} \beta_i x_i \leq B, \tag{42}$$

$$x_i \in \{0, 1\}, \quad \forall i \in T, \tag{43}$$

$$f_j \in \{0, 1\}, \quad \forall j \in F, \tag{44}$$

$$l_s \in \{0, 1\}, \quad \forall s \in S. \tag{45}$$

The objective is now to minimize the sum of the number of statements that cannot be covered and the number of faults that cannot be detected by the selected tests. Constraints (40) ensure that either a statement is covered by the reduced test suite or an indicator binary variable, l_s is equal to 1 showing that that particular statement **is not** covered by the reduced test suite. If the latter is the case, it is accordingly penalized in the objective function. Constraints (41) guarantee that a fault is either detected by the tests in the reduced test suite determined by the model or an indicator binary variable is equal to 1 showing that that fault is not detected by the reduced test suite. Again, if the latter is the case, it is accordingly penalized in the objective function. Constraints (42) ensure that the available budget is not exceeded. Finally, constraints (43), (44), and (45) are the binary restrictions for the decision variables. The optimal solution to *FTC* yields the best solution for the *TC* problem.

In the previous section, we introduced adversary cases to prove that existing approaches do not always lead to optimal solutions. In this section, we showed that our approach can find the optimal solution for these cases. However, one might question how prevalent these artificial cases are in reality. In the following section, we perform an empirical evaluation with a benchmark dataset collected from real systems. We show that our approach is able to find better (indeed optimal) solutions with respect to the ones obtained with existing approaches. Therefore, such cases should have been taking place in the dataset as well.

Table 4
Subject programs used for evaluation (Lin et al., 2018).

Program	Version	LOC	# of tests	# of faults
Grep	2.7	58,344	746	54
Flex	2.5.4	12,366	605	37
Sed	4.2	26,466	324	25
Make	3.80	23,400	158	15
Gzip	1.3	5,682	397	56

4. Empirical evaluation

In this section, we provide an empirical evaluation of our approach. In particular, we compare the effectiveness and time efficiency of our formulation (*FCB*, *FVB* and *FTC*) with respect to previously proposed formulations of the MCTSM problem, namely *LF_LS* (Section 2.1), *NF_NS* and *NF_LS* (Section 2.2). Hence, we investigate the following **research questions**:

> ***RQ1:*** *How do FCB, FVB and FTC compare against LF_LS, NF_NS and NF_LS with respect to effectiveness of the test suite?*
>
> ***RQ2:*** *How do FCB, FVB and FTC compare against LF_LS, NF_NS and NF_LS with respect to time efficiency?*

The metric we used for answering *RQ2* is basically the execution time. We measured the time it takes for each approach to find a solution. The smaller the metric value, the better.

We used two metrics for answering *RQ1*. The first metric is for evaluating the solutions for *CB* and *VB* problems. Hereby, the goal is to minimize both the number of tests and the number of missed faults. These goals have equal priorities. Therefore, the metric is the sum of the number of tests and the number of missed faults. The objective function is to minimize this metric value. *VB* adopts the same objective function as *CB* and introduces a new constraint: each of the top 10% mostly executed statements must be executed at least 10% of the times it is executed by the original test suite. However, the overall goal is still to minimize the sum of the number of tests and the number of missed faults.

The second metric used for answering *RQ1* is regarding the evaluation of solutions for the *TC* problem. The *TC* problem aims at maximizing the coverage of statements and the number of detected faults. These goals have equal priorities as well. Therefore, the metric is the sum of the number of executed statements and the number of detected faults. The objective function is to maximize this metric value. Variants of this problem adopt various budget constraints (i.e., the number of test cases that can be selected); however, they all keep the same objective function.

In the following we give details of our experimental setup towards answering our research questions.

4.1. Experimental setup

We have used an existing, publicly available dataset[2] that was previously created based on 5 open-source projects (Henard et al., 2016). These projects constitute our subject programs as listed in Table 4.

The dataset includes test suites as well as the cost, covered statements and the detected faults for each test within these test suites. These were used for evaluating and comparing solution approaches for the MCTSM problem (Lin et al., 2018). We have used the same dataset for a fair comparison with respect to the previously proposed approaches.

A sample faulty code snippet in one of the subject systems, *Sed*, can be seen in Listing 1. *Sed* stands for *stream editor* and it

[2] https://github.com/jwlin/Nemo/tree/master/subject_programs

can perform many operations on a text file such as searching, replacing, insertion and deletion. Listing 1 shows parts of the function that matches a given address to line(s). This address can be specified simply as the number of the line or a regular expression. One can also specify a range of lines, possibly including only some of the lines within this range based on a step pattern. So, test inputs for this function include numbers, regular expressions (e.g., "[\t]*$" matches all the lines ending with whitespace) and combinations of number ranges with optional step sizes (e.g., "1~3" matches lines 1, 4, 7, etc.). The faulty line in this code snippet is Line 12 and the correct version is commented above it.

We used CPLEX (Anon, 2019) as the solver. We used this solver on NEOS (Czyzyk et al., 1998), which is a free Web service for solving numerical optimization problems. We used the NEOS server to perform a fair performance comparison among the alternative formulations. The same server was utilized by the previous evaluations (Lin et al., 2018) as well.

We used the same set of MCTSM problem variants that was used for previous evaluations. These are *CB*, *VC* and *TC* as described in Section 2. The additional constraint for the *VC* problem is specified for the top 10% mostly executed statements, where each of them must be executed at least 10% of the number of times it is executed by the original test suite. The budget limit for the *TC* problem is specified as a percentage of the total cost for the original test suite. The problem is instantiated with values 5%, 10%, 15%, and 20% used as the budget limit.

4.2. Results and discussion

In the following, we present and discuss results regarding the effectiveness and time efficiency of MCTSM formulations to provide answers for *RQ1* and *RQ2*, respectively.

4.2.1. Effectiveness

Results regarding the *CB* problem are listed in Table 5. Hereby, the alternative formulations are listed in the first column, where the last one is our formulation. The first row lists the 5 subject systems. The second and the third rows list the total number of test cases and the total number of faults for the corresponding system. These are followed by the number of test cases selected by each of the alternative formulations and the number of faults detected by these test cases.

In Table 5, we see that *LF_LS* could not find the optimal solution for any of the subject programs. *NF_NS* and *NF_LS* could find the optimal set of test cases for *Sed* and *Make*. However, this is not the case for the other programs, where only *FCB* reached to the optimal solution. The corresponding cells are highlighted in light gray color on the table. Hereby, please recall that the metric we used for quantifying the quality of a solution for the *CB* problem is the sum of the number of tests and the number of faults missed. This sum should be minimized altogether. For instance, *FCB* detects 53 faults in *Grep* out of 54. So, it misses 1 fault only. The number of test cases is 71. Their sum turns out to be 72. On the other hand, *NF_LS* detects 36 faults out of 54. So, it misses 18 faults. The number of test cases is 59. Their sum turns out to be 77, which is 6.49% worse than the one obtained with *FCB*.

We can see that *NF_NS* coud not obtain any solution for *Grep* since the solver (Czyzyk et al., 1998) timed out after 8 hours (Lin et al., 2018). This turns out to be the case for almost all the subject systems except *Make* for the *VB* problem. Table 6 lists the results regarding this problem. The organization of this table is the same as Table 5. We observe that *LF_LS* could not find the optimal solution for any of the subject programs for this problem as well. *NF_LS* seems to be successful in finding the optimal solution for *Grep*, *Flex Sed* and *Make*. However, this is not the case for *Gzip*, where only *FVB* reached to the optimal solution.

FCB and *FVB* adopt the same objective function. They both aim at minimizing the number of tests as well as the number of missed faults. However, *FVB* finds a solution under an additional constraint: each of the top 10% mostly executed statements must be executed at least 10% of the times it is executed by the original test suite. On the other hand, *FCB* is not subject to any constraint regarding the selection of test cases. Therefore, it can achieve the same fault detection effectiveness with less number of test cases.

Results regarding the instances of the *TC* problem are listed in Table 7. The table is divided into 5 sections. The top section lists the 5 subject systems together the total number of test cases, the total number of statements, and the total number of faults for each. Each of the remaining sections is dedicated to an instance of the problem with a different budget limit, i.e., 5%, 10%, 15%, 20%. The second column lists the alternative formulations for each budget limit. *NF_NS* is not listed here, since it was not possible to obtain a solution due to the size of the model for this problem (Lin et al., 2018). We can see that only *FTC* could find the optimal solution for *Grep*, *Flex*, *Make* and *Gzip*, when the budget limit is 5%. *LF_LS* is again unable to find the optimal solution for any of the subject programs for this instance. The obtained solutions by all the approaches get better as the budget constraint gets relaxed and as such, the number of selected test cases can be increased. Hereby, the quality of solutions is not measured based on solely the number of detected faults but the overall objective function, which is to maximize the number of statements covered as well as the number of faults detected. For instance, tests selected by *FTC* with 5% budget detect 51 faults in *Gzip* and covers 1356 of its statements. Those tests that are selected by *NF_LS* for the same budget is 56. So, all the faults could be detected even when the budget is 5%. However, the number of covered statements is 1343. So, the detection of 5 more faults is compromised by covering 23 less number of statements. This is not considered as a better solution overall since the fault detection effectiveness and statement coverage have equal weights in the specified objective function.

Table 8 lists the amount of improvement achieved by *FTC* over the other approaches regarding the solutions found for the *TC* problem when the budget is 5%. Recall that the metric used for evaluating the objective function is the sum of the number of covered statements and the number of detected faults. Since the number of statements in subject programs are in the order of several thousands, even the detection/coverage of more than a dozen faults/statements is associated with an overall improvement that is less than 1%. Hence, these values do not really reflect the significance of the improvement achieved.

Table 9 lists the objective function values for all the subject systems, all problem instances and all the alternative formulations. The optimum value that is obtained with our formulation is highlighted with light gray color for cases, in which it could not be obtained with any of *LF_LS*, *NF_NS* and *NF_LS*. Hence, we empirically show that previous formulations, even nonlinear ones, cannot always find the optimal solution, whereas our formulation can.

Note that the goal of *TC* is to *maximize* both the number of statements exercised and the number of faults detected. Minimizing the number of test cases is not a concern in the problem definition. Therefore, we can see in Table 7 that the number of test cases tends to increase as the amount of budget increases. This is the case for all the formulations including ours and this is expected due to the definition of *TC*. One might find this to be in conflict with the goal of the MCTSM problems in general, i.e., to minimize the test suite. However, we can easily address this concern by extending our approach with an additional optimization step. In this second step, the object function is to

```
1 static bool match_an_address_p(struct addr *,struct input *){
2   switch (addr->addr_type){
3     case ADDR_IS_NULL:
4       return true;
5     case ADDR_IS_REGEX:
6       return match_regex(addr->addr_regex ,... line.length ...);
7     case ADDR_IS_NUM_MOD:
8       return (input->line_number >= addr->addr_number && ...);
9     case ADDR_IS_STEP:
10    case ADDR_IS_STEP_MOD:
11    //return (addr >addr_number > input >line_number);
12      return (addr->addr_number <= input->line_number);
13    case ADDR_IS_LAST:
14      return test_eof(input);
15    ...
16  }
17  return false;
18 }
```

Listing 1: A sample faulty code snippet from *Sed*, which is one of the subject systems used for evaluation.

Table 5
Effectiveness of alternative formulations of the *CB* problem (*# T*: number of test cases, *# F*: number of faults, *N/A*: the solver timed out).

Programs	Grep		Flex		Sed		Make		Gzip	
Original / Formul.	# T	# F	# T	# F	# T	# F	# T	# F	# T	# F
	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	49
NF_NS	*N/A*	*N/A*	44	32	12	25	14	13	45	49
FCB	71	53	48	37	12	25	16	15	49	56
FCB over LF_LS	14.28%		9.43%		25%		5.88%		3.92%	
FCB over NF_LS	6.49%		2.04%		0%		0%		5.77%	
FCB over NF_NS	*N/A*		2.04%		0%		0%		5.77%	

Table 6
Effectiveness of alternative formulations of the *VB* problem (*# T*: number of test cases, *# F*: number of faults, *N/A*: the solver timed out).

Programs	Grep		Flex		Sed		Make		Gzip	
Original / Formul.	# T	# F	# T	# F	# T	# F	# T	# F	# T	# F
	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*
FVB	80	54	66	37	32	25	17	15	60	56
FCB over LF_LS	16.66%		5.71%		8.57%		10.53%		4.76%	
FCB over NF_LS	0%		0%		0%		0%		3.23%	
FCB over NF_NS	*N/A*		*N/A*		*N/A*		0%		*N/A*	

minimize the number of test cases. The constraint is to achieve the same number of covered statements and the same number of detected faults as those obtained in the first step via the original *FTC* formulation. We also tried this two-step approach. Table 10 lists the results for the *TC* instance, where the budget limit is 20%. Hereby, the subject programs are listed in the first column. The following four columns list the number of test cases, the number of statements covered, the number of faults detected and the value of the objective function. Hereby, the number of statements, the number of faults and the objective function value are all the same with those obtained by the original *FTC* formulation (See Tables 7 and 9). However, the number of test cases are reduced significantly. The sixth column lists the number of test cases obtained with the original formulation (See Table 7). The last column lists the amount of reduction in the number of test cases obtained with the extended approach. Our approach can be extended in similar ways to handle additional criteria.

In the following, we share and discuss results regarding the runtime performance of our approach.

4.2.2. Performance

Since our formulation is a linear formulation, it would be expected that its runtime performance is comparable to that of *LF_LS*, better than *NF_LS* and significantly better than *NF_NS*.

Table 7
Effectiveness of alternative formulations of the *TC* problem for various budget limits (*# T*: number of test cases, *# S*: number of statements, *# F*: number of faults, *N/A*: the solver timed out).

Budget	Programs / Original / Formul.	Grep			Flex			Sed			Make			Gzip		
		# T	# S	# F	# T	# S	# F	# T	# S	# F	# T	# S	# F	# T	# S	# F
		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
	FTC	37	1664	41	30	3116	30	16	945	25	8	3789	12	20	1356	51
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
	FTC	75	1695	54	54	3143	37	32	945	25	16	3803	15	40	1401	55
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
	FTC	107	1695	54	74	3143	37	42	945	25	24	3803	15	50	1409	56
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
	FTC	107	1695	54	74	3143	37	42	945	25	27	3803	15	72	1409	56

Table 8
Amount of improvement achieved by *FTC* over the other approaches regarding the solutions found for the *TC* problem when the budget is 5% (*# S*: number of statements, *# F*: number of faults, *N/A*: the solver timed out).

Programs / Original / Formul.	Grep		Flex		Sed		Make		Gzip	
	# S	# F	# S	# F	# S	# F	# S	# F	# S	# F
	1695	54	3143	37	945	25	3803	15	1409	56
LF_LS	1302	29	2093	14	847	18	3665	10	540	25
NF_LS	1635	54	3094	37	945	25	3779	15	1343	56
FTC	1664	41	3116	30	945	25	3789	12	1356	51
FTC over LF_LS	28.09%		49.31%		12.14%		3.43%		149.03%	
FTC over NF_LS	0.95%		0.48%		0%		0.18%		0.57%	

We collected execution times from the results reported by NEOS (Czyzyk et al., 1998), when our approach is used for solving the *CB*, *VB* and *TC* problem instances to confirm this expectation. We compared these with respect to those reported for *LF_LS*, *NF_LS* and *NF_NS* (Lin et al., 2018). The measurements of execution time (in seconds) are depicted with a box plot in Fig. 1. Hereby, note that the *y*-axis that represents the execution time is in logarithmic scale. We can see that the execution time for solving the variants of our formulation never exceeded a second. The runtime performance of *LF_LS* is comparable to our formulation as expected. It takes 1 s to find solutions for given problems. However, results discussed in the previous subsection showed that these solutions were sub-optimal for many of the subject programs. On the other hand, the runtime performance is very divergent for *NF_LS* and *NF_NS*. For instance, *NF_LS* can find a solution within a few seconds for *CB*, *VB*. However, it takes up to several hours for the *TC* problem. *NF_NS* is the worst among all in terms of time performance. In fact, it fails to find any solution at all (before solver times out) for many of the subject programs. Hence, we conclude that the time efficiency of our approach is better than the state-of-the-art nonlinear formulations and comparable to that of previous linear formulations.

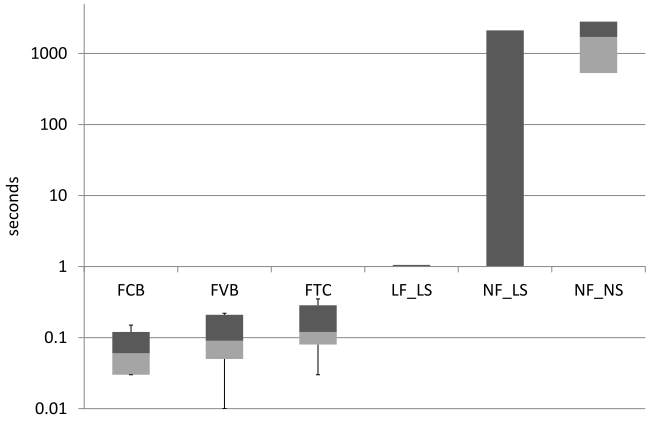

Fig. 1. Runtime performance of variants of our model and the other models on MCTSM problem instances.

4.3. Threats to validity

Our evaluation is subject to *external* validity threats (Wohlin et al., 2012) since it is based on a benchmark dataset that comprises similar types of programs. All of these programs are Unix utilities provided by GNU. This dataset was adopted from a prior study, where the state-of-the-art approach was proposed and evaluated. We used this dataset without any changes. As such, we mitigate *internal* and *construct* validity threats while comparing our approach with respect to the state-of-the-art in terms of the effectiveness of the resulting test suite. Finally, *conclusion* validity is mitigated by adopting the same set of constraints, criteria and weights for test suite minimization. First, we showed flaws in

Table 9
Objective function values obtained with state-of-the-art formalisms and the optimum value obtained with our model (*CB* and *VB* problems aim at minimization, whereas instances of the *TC* problem aim at maximization of the objective function value).

Programs	Problems \ Formul.	LF_LS	NF_LS	NF_NS	The optimum
Grep	*CB*	84	77	*N/A*	72
	VB	96	80	*N/A*	80
	TC with budget 5%	1331	1689	*N/A*	1705
	TC with budget 10%	1331	1749	*N/A*	1749
	TC with budget 15%	1358	1749	*N/A*	1749
	TC with budget 20%	1364	1749	*N/A*	1749
Flex	*CB*	53	49	49	48
	VB	70	66	*N/A*	66
	TC with budget 5%	2107	3131	*N/A*	3146
	TC with budget 10%	2487	3180	*N/A*	3180
	TC with budget 15%	2626	3180	*N/A*	3180
	TC with budget 20%	2719	3180	*N/A*	3180
Sed	*CB*	16	12	12	12
	VB	35	32	*N/A*	32
	TC with budget 5%	865	970	*N/A*	970
	TC with budget 10%	878	970	*N/A*	970
	TC with budget 15%	924	970	*N/A*	970
	TC with budget 20%	925	970	*N/A*	970
Make	*CB*	17	16	16	16
	VB	19	17	17	17
	TC with budget 5%	3675	3794	*N/A*	3801
	TC with budget 10%	3686	3818	*N/A*	3818
	TC with budget 15%	3715	3818	*N/A*	3818
	TC with budget 20%	3715	3818	*N/A*	3818
Gzip	*CB*	51	52	52	49
	VB	63	62	*N/A*	60
	TC with budget 5%	565	1399	*N/A*	1407
	TC with budget 10%	565	1456	*N/A*	1456
	TC with budget 15%	566	1465	*N/A*	1465
	TC with budget 20%	566	1465	*N/A*	1465

existing formulations with small contradictory examples. Then, we showed in our empirical evaluation that our formulation can reach to optimal results for all the subject programs, whereas previous formulations can stuck at sub-optimal results for some of these. We shared both our formulations and the obtained results online[3] for reproducibility of our experimental results. We also shared the source code (in C++) of the program that we developed for automatically generating formulations.

Table 10
Results obtained after minimizing the test suite for the instance of the *TC* problem, where the budget is set as 20% (*# T*: number of test cases, *# S*: number of statements, *# F*: number of faults).

Programs	# T	# S	# F	Obj. function value	# T in FTC solution	Reduction in # T
grep	72	1695	54	1749	107	33%
flex	48	3143	54	3180	74	35%
sed	12	945	25	970	42	71%
make	16	3803	15	3818	27	41%
gzip	49	1409	56	1465	72	32%

5. Related work

Test suite minimization problem has been known to be an NP-complete problem (Yoo and Harman, 2012). Therefore, there have been many solution proposals in the literature that employ heuristics. Some of these approaches (Harrold et al., 1993; Chen and Lau, 1996; Tallam and Gupta, 2005) focus on single-criterion test suite minimization, where greedy heuristics are adopted to find a minimal subset of a test suite that covers all the requirements. It was shown that optimization with respect to such a single criterion can significantly reduce the fault detection effectiveness of the resulting test suite (Rothermel et al., 1998). Consequently, heuristics-based approaches for addressing the MCTSM problem (Jeffrey and Gupta, 2007; Lin and Huang, 2009) were also proposed, where fault detection effectiveness is incorporated as an additional criterion. There exist many other approaches (Sampath et al., 2013) that employ heuristics, machine learning algorithms and greedy algorithms to address the MCTSM problem with hybrid combinations of various criteria

[3] http://srl.ozyegin.edu.tr/tools/mctsm-lp

such as statement coverage, branch coverage and execution time. These approaches generate approximate solutions and do not guarantee an optimal solution.

We employ integer programming as the solution approach. A majority of related studies in this area (Yoo and Harman, 2012; Lin et al., 2018) focus on bi-criteria problems. For instance, Black et al. (2004) considers fault detection ability and the coverage of all uses in data flow as the two criteria for constructing an integer linear programming model. Likewise, another model (Hao et al., 2012) focuses on statement coverage and fault detection ability. There are other approaches (Li et al., 2014; Jabbarvand et al., 2016), which consider statement coverage and energy consumption as the optimization criteria. Similarly, many other types of criteria (Sampath et al., 2013) can be considered for the construction of the integer linear programming model. However, these approaches have two drawbacks. First, they do not guarantee an optimal solution for the MCTSM problem as explained in Section 2. Second, they mainly focus on specific bi-criteria and ignore other types of criteria. MINTS (Hsu and Orso, 2009) was proposed as a generic framework, where an arbitrary number of criteria can be accommodated. However, MINTS does not always provide an optimal solution to the MCTSM problem as well. In fact, the *LF_LS* approach that is compared with respect to ours in Section 4 corresponds to the implementation of MINTS.

Even if the problem is defined based on the same set of concerns, it might be formulated in an open-ended number of ways depending on the definition of constraints, criteria and weights assigned for these criteria. For instance, one can simply aim at minimizing the number of selected tests while covering all the originally covered statements as well as not missing any fault that could be detected by the original test suite (Shi et al., 2014). In this case, there would be only one objective, to minimize the number of tests and two constraints: *(i)* to keep the same statement coverage and *(ii)* to keep the same fault detection capability. One can argue that the solution obtained from the solver is optimal for the given problem definition although it might not be optimal for another definition. However, we argue that existing formulations are subject to flaws that prevent them always reaching optimal solutions even for the intended problem definition. We adapt our formulation for 3 previously defined instances of the problem to prove our argument and to make a fair comparison with the state-of-the-art approaches (Lin et al., 2018). We showed that our formulation can reach to better (indeed optimal) results for the same set of constraints, criteria and weights.

To the best of our knowledge, the most recent, state-of-the-art approach for addressing the MCTSM problem is NEMO (Lin et al., 2018). We facilitate automation for generating linear programming formulations like NEMO does. Our program uses the same set of inputs (fault coverage, line coverage and cost information regarding tests) with the same file format. However, formulations generated by NEMO are different from ours. NEMO formulates MCTSM in the form of an integer nonlinear programming problem. Then, it proposes two alternatives to solve this problem. First, the nonlinear formulation can be supplied directly to a nonlinear solver. The *NF_NS* approach that is compared with respect to ours in Section 4 corresponds to this alternative. Second, the nonlinear formulation can be transformed into a linear one through the use of auxiliary variables. Then, this can be solved using a linear solver. The *NF_LS* approach that is compared with respect to ours in Section 4 corresponds to this second alternative. However, the nonlinear formulation itself is subject to flaws that might lead to sub-optimal solutions. We showed this with adversary examples (Section 2). The linear formulation that is transformed from the nonlinear formulation embodies the same flaws. This becomes apparent in empirical results (Section 4), where the obtained formulation fails to find optimal solutions for some of the subject systems. Our formulation can find the optimal solutions for these as well as all the others. Furthermore, our formulation is a linear one, which makes it unnecessary to adopt an additional transformation process and auxiliary variables.

In real life, one can obtain information regarding only those faults that were detected in previous versions of the system. There have been studies (Zhang et al., 2011; Shi et al., 2014, 2018) to evaluate the effectiveness of MCTSM approaches based on this fact. Hereby, test cases are selected based on faults that have been detected in the past. Then, the selected test cases are executed on the next version of the system to evaluate their capability in detecting new faults. We used an existing dataset to unequivocally show the superiority of our formulation with respect to existing formulations. In our evaluation, we obtain an optimal solution for a given input regarding the test cost, coverage of statements and detection of faults by the test cases. The actual effectiveness of the solution would depend on how much this input reflects the state of the next version of the system. However, evaluation of this correlation is out-of-scope of this paper.

Code coverage has been the mostly adopted criterion for measuring test effectiveness (Zhang et al., 2019) and therefore commonly used for minimizing test suites. However, there exist other coverage metrics as well. One of these is the recently introduced *assertion coverage* (Zhang and Mesbah, 2015), which is also known as *checked coverage* (Schuler and Zeller, 2011). There has been empirical evidence suggesting that assertion coverage has a stronger correlation with mutation score (Zhang and Mesbah, 2015) although it was shown to be the opposite in some other studies (Zhang et al., 2019). A recent test minimization approach (Vahabzadeh et al., 2018) used assertion coverage as the main criterion though it minimizes test suites at the statement level, rather than at the test case level. Our approach is agnostic to the coverage criteria adopted for test suite minimization. It takes as input the list of test cases together with an enumerated list of items covered by each test case. These enumerated items can in principle represent anything including source code statements, faults or assertions.

6. Conclusion and future work

We proposed a novel formulation of the multi-criteria test suite minimization problem based on integer linear programming. We identified shortcomings of the state-of-the-art formulations that might lead to sub-optimal solutions. We demonstrated this with adversary examples and show that our formulation can address the identified shortcomings. We also empirically evaluated the effectiveness of our approach based on a publicly available dataset derived from a set of open-source projects. Results show that our linear formulation leads to better results with respect to the state-of-the-art in terms of the same objective function and the same set of criteria including statement coverage, fault-revealing capability, and test execution time. In addition, it leverages better time performance compared to non-linear formulations, making it more scalable for larger problems.

Formulating the multi-criteria test suite minimization problem as an integer programming problem is effective for reaching the optimal solution. However, this problem is known to be NP-complete and such formulations (even the linear one) do not guarantee to find a solution quickly. This approach might fall short as the problem size grows further. Therefore, a possible future direction would be to extend the evaluation with larger datasets, compare the results with those obtained with heuristics or greedy algorithms, and assess the trade-off between the effectiveness and efficiency of alternative approaches.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Anon, 2019. IBM CPLEX optimizer. https://www.ibm.com/analytics/cplex-optimizer, (Accessed 29 August 2019).

Black, J., Melachrinoudis, E., Kaeli, D., 2004. Bi-criteria models for all-uses test suite reduction. In: Proceedings of the 26th International Conference on Software Engineering, pp. 106–115.

Chen, T.Y., Lau, M.F., 1996. Dividing strategies for the optimization of a test suite. Inform. Process. Lett. 60 (3), 135–141.

Czyzyk, J., Mesnier, M., More, J., 1998. The NEOS server. IEEE Comput. Sci. Eng. 5 (3), 68–75.

Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA.

Hao, D., Zhang, L., Wu, X., Mei, H., Rothermel, G., 2012. On-demand test suite reduction. In: Proceedings of the 34th International Conference on Software Engineering, pp. 738–748.

Harrold, M.J., Gupta, R., Soffa, M.L., 1993. A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. 2 (3), 270–285.

Henard, C., Papadakis, M., Harman, M., Jia, Y., Traon, Y.L., 2016. Comparing white-box and black-box test prioritization. In: Proceedings of the 38th International Conference on Software Engineering, pp. 523–534.

Herzig, K., Greiler, M., Czerwonka, J., Murphy, B., 2015. The art of testing less without sacrificing quality. In: Proceedings of the 37th International Conference on Software Engineering, pp. 483–493.

Hsu, H.Y., Orso, A., 2009. MINTS: A general framework and tool for supporting test-suite minimization. In: Proceedings of the 31st IEEE International Conference on Software Engineering, pp. 419–429.

Jabbarvand, R., Sadeghi, A., Bagheri, H., Malek, S., 2016. Energy-aware test-suite minimization for android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 425–436.

Jeffrey, D., Gupta, N., 2007. Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Trans. Softw. Eng. 33 (2), 108–123.

Jin, W., Orso, A., 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In: Proceedings of the 34th International Conference on Software Engineering, pp. 474–484.

Li, D., Jin, Y., Sahin, C., Clause, J., Halfond, W.G.J., 2014. Integrated energy-directed test suite optimization. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 339–350.

Lin, J.-W., Huang, C.-Y., 2009. Analysis of test suite reduction with enhanced tie-breaking techniques. Inf. Softw. Technol. 51 (4), 679–690.

Lin, J.-W., Jabbarvand, R., Garcia, J., Malek, S., 2018. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1039–1049.

Miranda, B., Bertolino, A., 2017. Scope-aided test prioritization, selection and minimization for software reuse. J. Syst. Softw. 131, 528–549.

Nardo, D.D., Alshahwan, N., Briand, L., Labiche, Y., 2015. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. Softw. Test. Verif. Reliab. 25 (4), 371–396.

Rothermel, G., Harrold, M., J., J.R., Hong, C., 2002. Empirical studies of test suite reduction. Softw. Test. Verif. Reliab. 4 (2), 219–249.

Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C., 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of the 6th International Conference on Software Maintenance, pp. 34–43.

Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27 (10), 929–948.

Sampath, S., Bryce, R., Memon, A.M., 2013. A uniform representation of hybrid criteria for regression testing. IEEE Trans. Softw. Eng. 39 (10), 1326–1344.

Schuler, D., Zeller, A., 2011. Assessing oracle quality with checked coverage. In: Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation, pp. 90–99.

Shi, A., Gyori, A., Gligoric, M., Zaytsev, A., Marinov, D., 2014. Balancing trade-offs in test-suite reduction. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 246–256.

Shi, A., Gyori, A., Mahmood, S., Zhao, P., Marinov, D., 2018. Evaluating test-suite reduction in real-world software evolution. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 84–94.

Steuer, R., 1986. Multi-Criteria Optimization: Theory, Computation, and Application. John Wiley, New York.

Tallam, S., Gupta, N., 2005. A concept analysis inspired greedy algorithm for test suite minimization. SIGSOFT Softw. Eng. Notes 31 (1), 35–42.

Vahabzadeh, A., Stocco, A., Mesbah, A., 2018. Fine-grained test minimization. In: Proceedings of the 40th International Conference on Software Engineering, pp. 210–221.

Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S., 2006. TimeAware test suite prioritization. In: Proceedings of the 15th International Symposium on Software Testing and Analysis, pp. 1–12.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A., 2012. Experimentation in Software Engineering. Springer-Verlag, Berlin, Heidelberg.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.

Zhang, L., Hou, S.-S., Guo, C., Xie, T., Mei, H., 2009. Time-aware test-case prioritization using integer linear programming. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, pp. 213–224.

Zhang, L., Marinov, D., Zhang, L., Khurshid, S., 2011. An empirical study of JUnit test-suite reduction. In: Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering, pp. 170–179.

Zhang, Y., Mesbah, A., 2015. Assertions are strongly correlated with test suite effectiveness. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp. 214–224.

Zhang, J.M., Zhang, L., Hao, D., Wang, M., Zhang, L., 2019. Do pseudo test suites lead to inflated correlation in measuring test effectiveness? In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification, pp. 252–263.

O. Örsan Özener received his B.Sc. (2001) and M.S. (2003) degrees in Industrial Engineering from Middle East Technical University. In 2006, he received his M.S. degree in Operations Research and in 2008 his Ph.D. in Industrial and Systems Engineering from the Georgia Institute of Technology. He worked as a research assistant at Middle East Technical University between 2001 and 2003, and at Georgia Institute of Technology between 2003 and 2008. His research interests are in integer/combinatorial optimization, data-driven optimization, and non-cooperative/cooperative game theory.

Hasan Sözer received his B.Sc. and M.S. degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his Ph.D. degree in 2009 from the University of Twente, The Netherlands. From 2002 until 2005, he worked as a software engineer at Aselsan Inc. in Turkey. From 2009 until 2011, he worked as a postdoctoral researcher at the University of Twente. In 2011, he joined the department of computer science at Ozyegin University, where he is currently working as an associate professor.