Multi-Level Random Walk for Software Test Suite Reduction

***Abstract*—**Which test cases should be selected to save the time of software testing? Due to the large time cost of running all test cases, it is necessary to run representative test cases to shorten the software development cycle. Test suite reduction, an NP-hard problem in software engineering, aims to select a subset of test cases to reduce the time cost of test execution in satisfying test requirements. Recently, search based software engineering provides a new direction to test suite reduction by connecting software engineering problems with computational intelligence methods. In this paper, we propose a multi-level optimization algorithm to simplify the original problem instance of test suite reduction. In each level, we search for local optimal solutions with random walk in potential subsets of the test suite. The problem scale is reduced by locking the intersection of local optima and by discarding shielded test cases with no contribution to test requirements. We compare our algorithm with state-of-the-art methods on test suites of ten large-scale open source projects. Experiments show that our algorithm can more efficiently find optima on five out of six projects, in which Integer Linear Programming (ILP) can find optima; for the other four projects that ILP fails to solve, our algorithm provides the best solutions among heuristics in comparison.

Digital Object Identifier 10.1109/MCI.2017.2670460
Date of publication: 10 April 2017

I. Introduction

Software testing is important and time-consuming. A test suite, i.e., a set of test cases, plays a key role in validating the expected program behavior. In modern test-driven development, a test suite pushes the development progress. Software evolves over time; its test suite is executed to detect whether a new code change adds bugs to the existing code. Executing all test cases after each code change is unnecessary and may be impossible due to the limited development cycle. On the one hand, multiple test cases may focus on an identical piece of code; then several test cases cannot detect extra bugs. On the other hand, even executing a test suite once in a large project takes around one hour [1]; frequent code changes require much time for conducting testing. For instance, in Hadoop, a framework of distributed computing, 2,847 version commits are accepted within one year from September 2014 with a peak of 135 commits in one week [2].

Which test cases should be selected to save the testing time? Test suite reduction aims to minimize the number of executed test cases to cover all test requirements. Test requirements are practically viewed as code coverage. For instance, statement coverage, i.e., the ratio of covered statements during test execution, is the most common test requirements in real-world development [3].

Corresponding Author: Jifeng Xuan (jxuan@whu.edu.cn).

1556-603X/17©2017IEEE

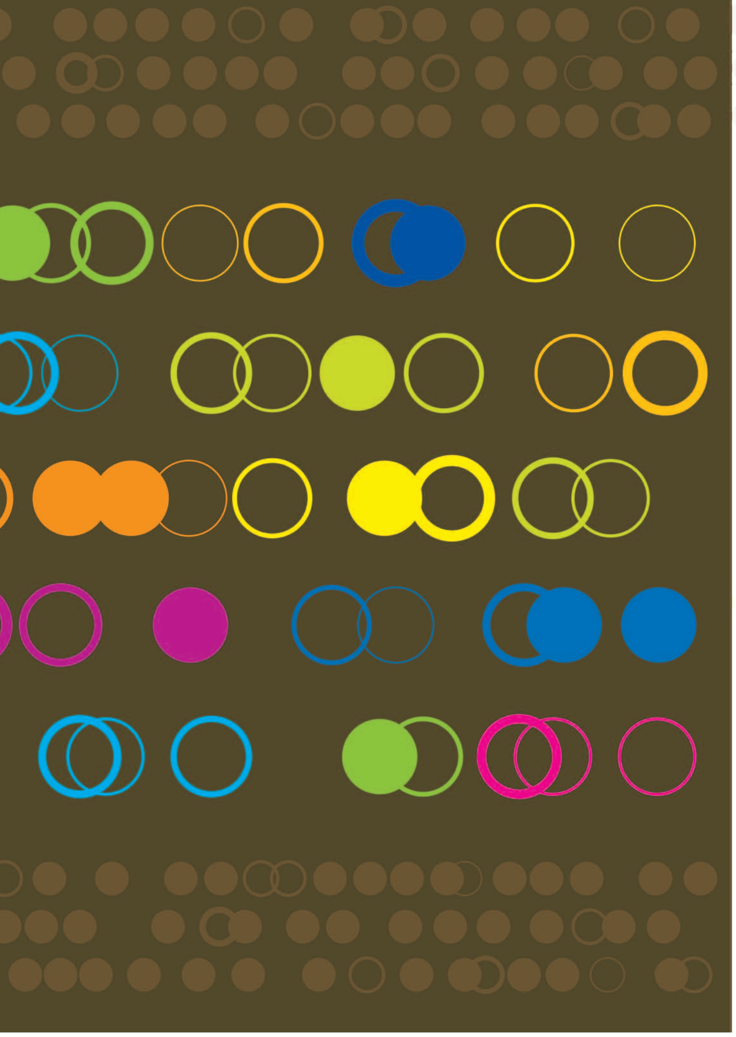


IMAGE LICENSED BY INGRAM PUBLISHING

Zongzheng Chi
School of Software, Dalian University of Technology, CHINA

Jifeng Xuan
State Key Lab of Software Engineering, Wuhan University, CHINA

Zhilei Ren
School of Software, Dalian University of Technology, CHINA

Xiaoyuan Xie
State Key Lab of Software Engineering, Wuhan University, CHINA

He Guo
School of Software, Dalian University of Technology, CHINA

Test suite reduction, also called test suite minimization in some literature [3], is one of the typical fields in search based software engineering [4]. The development of search based software engineering bridges the gap between software engineering problems and computational intelligence methods. Well-designed heuristics in computational intelligence can be employed or transformed to address optimization problems in software engineering. This expands the application of the research in computational intelligence.

Application scenario of test suite reduction. Test suite reduction is inevitable for continuous development and testing. Test suite reduction provides a subset of test suite without losing pre-defined test requirements. After each code change, the reduced test suite is executed to detect potential bugs, i.e., ensuring that no bug is newly involved during continuous code changes.

The test suite reduction problem is formulated as the unicost set covering problem (the "unicost" is omitted for short) [5], which is one of the Karp's 21 NP-complete problems [6]. To find a minimized test suite, many algorithms are designed, including heuristics [7] and exact algorithms like Integer Linear Programming (ILP) [8], [9]. As an exact method for the set covering problem, ILP can find global optima for test suite reduction, but may be expensive in time cost and computing resources. Heuristics can find a near-optimal solution, but there is large room to improve the solution.

In this paper, we propose MultiWalk, a **Multi**-level random **Walk** algorithm for software test suite reduction. MultiWalk leverages the common part of local optima by random walk search to simplify the original problem instance, where the common part (i.e., the intersection) of local optima is referred to as the "backbone". In each level of MultiWalk, we search for local optimal solutions with random walk in potential subsets of the test suite. The problem scale is iteratively reduced by locking the backbone and by discarding shielded test cases, which contribute to no test requirements in the test suite. MultiWalk repeats extracting the backbone and reducing the scale for multiple times until a small problem is achieved. A solution to this small problem is combined with the backbones to refine the final solution to the original problem. Hence, in MultiWalk, the process of finding a minimized test suite is converted into finding backbones and refining the final solution.

Experiments are conducted on ten widely-used and large-scale open source projects in Java. Each project is instrumented to collect runtime traces and to record the statement coverage. We compare MultiWalk with five state-of-the-art heuristics, the exact method ILP, and random walk without the multi-level strategy. Experimental results show that MultiWalk can find optima within less time for five out of six projects that ILP can solve; meanwhile, MultiWalk can obtain better solutions than the state-of-the-art heuristics for projects that ILP fails to solve before timeout. To further understand MultiWalk, we investigate its ability of test suite reduction in three directions, i.e., the similarity between local and global optima, the change of backbone scales with the number of local optima, and the ability of problem scale downgrading with levels.

II. Software Test Suite Reduction Problem

Given a test suite T ($|T| = n$), i.e., a set of test cases, and a set R of test requirements ($|R| = m$), let A be the runtime coverage

A test suite is iteratively developed together with the source code to find software bugs in the early stage. This follows a modern development process, called test-driven development.

between test cases in T and requirements in R. That is $A: T \times R = \{\langle t, r \rangle \mid t \text{ satisfies } r, \text{ where } t \in T, r \in R\}$. A test case is called a test for short. For each $r_i \in R$ and each $t_j \in T$, $a_{ij} = 1$ indicates that r_i is satisfied by t_j; $a_{ij} = 0$ indicates not. A general form of the problem of software test suite reduction is defined as follows,

$$\text{Objective:} \quad \text{minimize} \sum_{j=1}^{n} x_j, x_j \in \{0,1\}$$

$$\text{Constraint:} \bigwedge_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \geq 1 \right), a_{ij} \in \{0,1\}$$

where x_j indicates whether a test $t_j \in T$ is selected in the reduced test suite. This problem has the same model as the set covering problem, which is already proved as NP-complete [6]. The above model of test suite reduction can be specialized via abstracting domain knowledge, such as on-demand requests [10] and abilities of fault localization [11]. In this paper, we address the general model above, whose extension will be discussed in Section VI. For test suite reduction, ILP can exactly solve small-scale instances, while dealing with large-scale instances with ILP may be practically impossible [12].

In practice, test requirements in R are a type of test criterion that indicates software quality. Code coverage, measuring how much code is executed by the reduced test suite, is the most common criterion of test requirements. Informally, the test suite reduction problem is to select a minimized subset of test cases, which cannot lose the statement coverage, comparing with the original test suite. Note that besides statement coverage, fault detection capability and program mutation scores can also serve as test requirements [13]. However, collecting these requirements for large projects is complex and time-consuming. In this paper, we evaluate test suite reduction via statement coverage (i.e., how many statements are executed by test cases), which is widely-used in most of related work of test suite reduction [7], [8].

Ideally, ILP can find optimal solutions to test suite reduction if the computing resource is unlimited. However, for large software projects, only focusing on the optimal solutions with ILP may be infeasible. Hence, many heuristics are developed to efficiently finding sub-optimal solutions. For instance, a pioneering heuristic, GRE by Chen and Lau [14] extends the greedy algorithm by removing redundancy between two unselected test cases; a state-of-the-art meta-heuristic, RAPS by Lan et al. [15] finds solutions via the technique of meta-heuristics for randomized priority search.

Motivated by the trade-off between ILP and heuristics, we design a new algorithm to downgrade the problem scale and to find solutions for small ones. Our algorithm leverages random walk on the selection of test cases; then the common part of solutions by random walk is locked to further reduce the problem scale.

Algorithm 1 **MultiWalk, a multi-level random walk algorithm for test suite reduction.**

```
    Input:
    $\langle T_1, R_1 \rangle$, an original problem instance with a test suite $T_1$
    and a requirement set $R_1$;
    $\varpi$, a local search operator;
    $\alpha$, the maximum number of levels;
    $\beta$, the number of local optima in each level.
    Output:
    $S_1$, a solution to the original instance $\langle T_1, R_1 \rangle$.
 1  for Level $k = 1, \ldots, \alpha$ do
 2      find a set $\Delta_k$ of $\beta$ local optima to the instance $\langle T_k, R_k \rangle$
        by the local search operator $\varpi$;
 3      collect a backbone $\delta_k = \cap \Delta_k$;
 4      collect requirements covered by $\delta_k$, $\tau_k = \text{Cover}(\delta_k)$;
 5      reduce the instance, $T^0_{k+1} = T_k \backslash \delta_k$, $R_{k+1} = R_k \backslash \tau_k$;
 6      collect a shielded test set $\varphi_k = \text{Shield}(T^0_{k+1}, R_{k+1})$;
 7      discard tests in $\varphi_k$, $T_{k+1} = T^0_{k+1} \backslash \varphi_k$;
 8  end
 9  count the actual number $\gamma$ of levels ($\gamma \leq \alpha$);
10  find selected tests $S_{\gamma+1}$ for the smallest instance
    $\langle T_{\gamma+1}, R_{\gamma+1} \rangle$ by $\varpi$;
11  for Level $k = \gamma, \ldots, 1$ do
12      refine selected tests, $S_k = S_{k+1} \cup \delta_k$;
13  end
```

III. Multi-Level Random Walk

To reduce the problem scale of test suite reduction for large projects, we design MultiWalk, a **Multi**-level random **Walk** algorithm. The reduction of problem scales is based on locking the intersection of local optima by random walk search. For the sake of simplification, a solution S of test suite reduction is viewed as a subset of the original test suite, i.e., $S \subseteq T$.

A. Multi-Level Strategy

Since it is hard to directly solve an original problem of test suite reduction, we intend to solve small sub-problems instead. Our idea, motivated by Walshaw [16], is to conduct multiple levels of sub-problems by locking several tests. The multi-level strategy consists of two major stages, the reduction stage that reduces the problem scales and the refinement stage that combines solutions in multiple levels.

To obtain a solution similar to (or even the same as) an optimal solution, an ideal way is to lock tests that belong to the optimal solution. We leverage the concept of "backbone"

[17], [18] that is defined as the common part of optimal solutions. In the context of test suite reduction, a backbone is a subset of the whole test suite. Since it is hard to find optimal solutions of an NP-hard problem, the backbone is practically replaced by the approximate backbone, which is the intersection of local optimal solutions [19], [20]. We use the term "backbone" to denote "approximate backbone" for short. In this paper, random walk search is employed to find local optima of test suite reduction and then collect the backbone based on these local optima.

Algorithm 1 presents the overview of MultiWalk. At Lines 1 to 8, MultiWalk reduces a problem instance for multiple times by locking the backbone based on local search; at Line 9, the actual number of levels is counted; then at Line 10, a solution to the reduced instance is obtained; at Lines 11 to 13, the solution to the small instance and the backbones are combined to form the final solution to the original problem instance.

In each level of instance reduction, tests in the backbone (Line 5) are selected as a part of the final solution while tests that are shielded by other tests (Line 7) are discarded from the final solution. A shielded test case denotes a non-selected test case, whose covered statements are covered by another test case. We discard shielded test cases since it cannot benefit the statement coverage but adds the size of executed tests.

B. Backbone and Shielded Tests

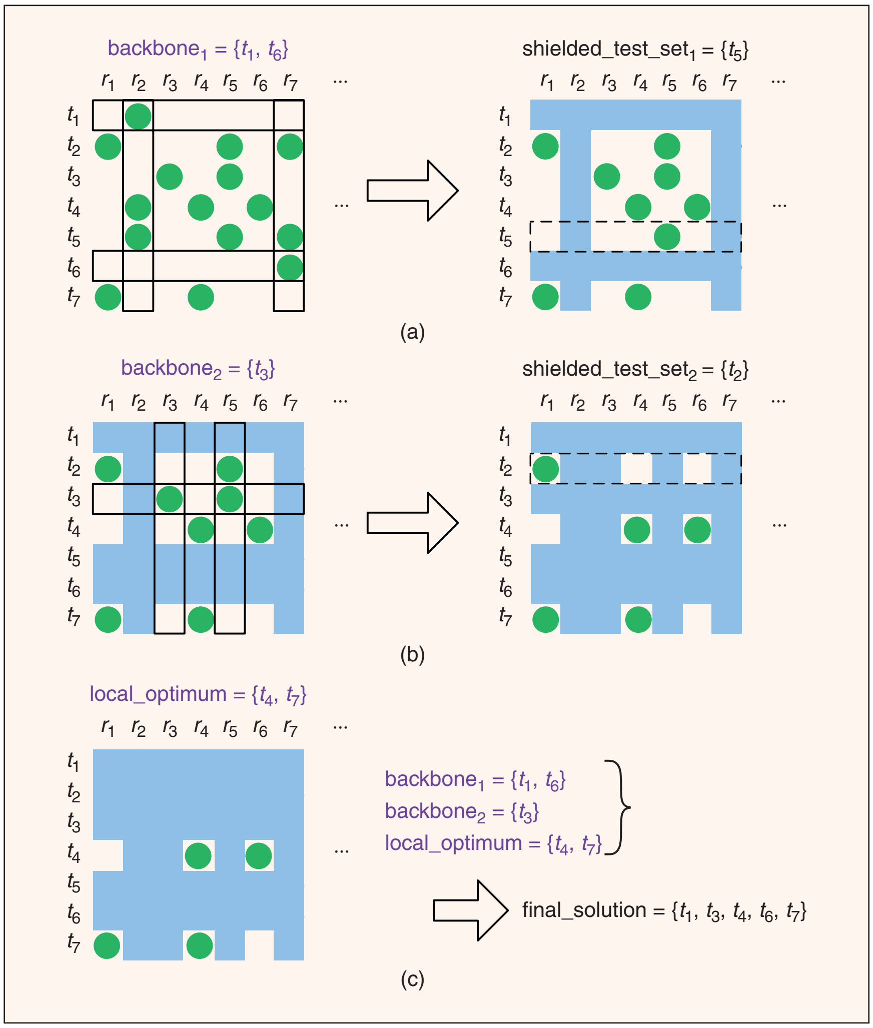
A backbone in test suite reduction is a potential test component, which is possible to be abstracted to improve the design of the test suite [21]. The size of a backbone depends on the number of local optima as well as the ability of the local search operator [22], [23]. In general, any local search technique can be embedded in the multi-level algorithm. An experiment in Section V-B will later show that local optima by WalkTest are similar to the global optimal solution.

In contrast to the backbone that is locked to conduct the final solution, shielded test cases are discarded to avoid the redundancy in solutions. Once a test case is shielded by others, it cannot contribute to adding statement coverage. Since our multi-level strategy reduces the problem instances for several times, shielded test cases can be removed after obtaining the backbone in each level. An experiment in Section V-B will illustrate that discarding shielded test cases will effectively reduce the problem scale. Note that in one level of MultiWalk, an empty backbone or no shielded test does not break the algorithm. MultiWalk can skip this level and directly go to the next level since MultiWalk is an iterative procedure.

Fig. 1 illustrates the roles of backbones and shielded test cases with an example of two-level reduction and refinement in MultiWalk. The original problem instance with seven tests is reduced to two smaller instances; a solution to the reduced instance is combined together with two backbones as the final solution.

C. Random Walk Search, WalkTest

Many local search operators can be embedded into MultiWalk. In this section, we design a new local search based on random walk, which is motivated by the success of random walk techniques in the SAT problem, i.e., WalkSat [24]–[26]. The random walk in WalkTest provides the diversification of



backbone$_1$ = {t_1, t_6}
shielded_test_set$_1$ = {t_5}
r_1 r_2 r_3 r_4 r_5 r_6 r_7 ...
t_1 t_2 t_3 t_4 t_5 t_6 t_7
...
(a)
backbone$_2$ = {t_3}
shielded_test_set$_2$ = {t_2}
(b)
local_optimum = {t_4, t_7}
backbone$_1$ = {t_1, t_6}
backbone$_2$ = {t_3}
local_optimum = {t_4, t_7}
final_solution = {t_1, t_3, t_4, t_6, t_7}
(c)

FIGURE 1 Illustration of test suite reduction on seven test cases and over seven requirements (only seven requirements are shown). Two-level reduction and refinement is conducted before finding the final solution. (a) The first level reduction. A backbone of tests t_1 and t_6 is selected; requirements r_2 and r_7 are satisfied. Then in the reduced instance, a test t_5 is shielded by t_2 or t_3. (b) The second level reduction. In the reduced instance after the first level, a backbone of a test t_3 is selected; requirements r_3 and r_5 are satisfied. Then in the reduced instance, a test t_2 is shielded by t_7. (c) Solving the small instance after the second level and conducting two-level refinement.

solutions; in this case, greedy algorithms are not suitable for collecting backbones.

Fig. 2 illustrates the major steps of our random walk operator, called WalkTest. The nature of random walk is to randomly traverse the search space of an instance; a random walk operator guides such traversal by designing the moves, which are led by a pre-defined probability p. In our case, WalkTest makes greedy moves with p and random moves with $1 - p$; the iterative process in WalkTest is expected to fit the noisy search space [27].

WalkTest starts with randomly generating a solution to test suite reduction. Then a number of tries are conducted to find a local optimum. In each try, WalkTest first collects zero-hurt tests. A zero-hurt test is a selected test, which could be removed without losing statement coverage. That is, this test case is redundant to the reduced test suite. If zero-hurt test exists, one of such tests is removed from the solution; if not, WalkTest proceeds according to a given probability p. With a probability p, a non-selected test that can add the most covered statements will be selected in the solution; with $1 - p$, one of non-selected tests is randomly added to the solution. WalkTest continuously manipulates the solution until the maximum number of tries reaches.

We choose WalkTest to find the local optima since it is simple in design. WalkTest shares the similar strategy with many existing heuristics. For instance, the design of random walk in WalkTest relates to simulated annealing algorithms [28]. We list major differences between WalkTest and simulated annealing in three categories. First, simulated annealing accepts the best solution during its iteration while WalkTest randomly accepts one of improved solutions. Second, simulated annealing chooses a worse solution based on an acceptance probability while WalkTest updates a worse solution based an acceptance probability. Third, the acceptance probability in simulated annealing decreases during the iteration while the one in WalkTest is constant.

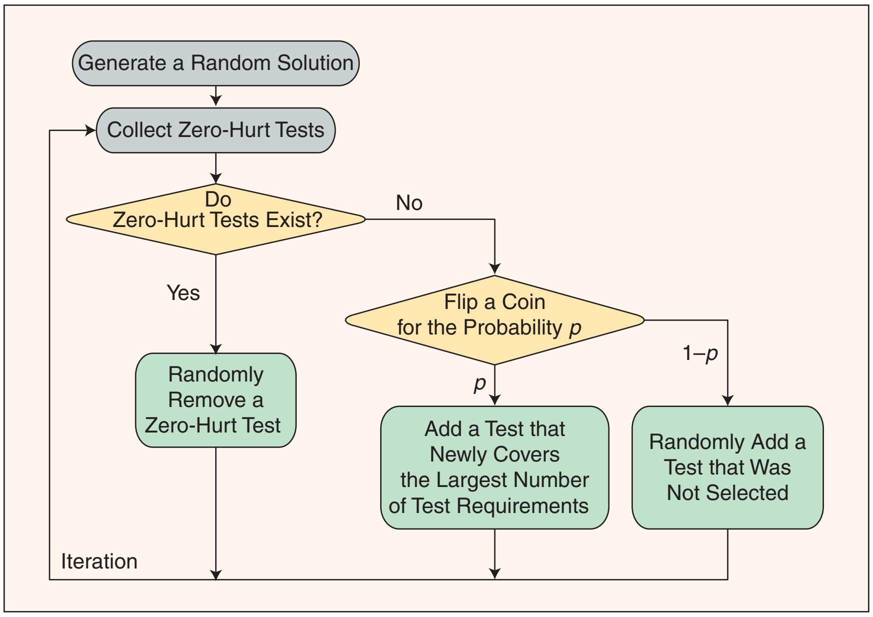

Generate a Random Solution
Collect Zero-Hurt Tests
Do Zero-Hurt Tests Exist?
No
Yes
Flip a Coin for the Probability p
1–p
p
Randomly Remove a Zero-Hurt Test
Add a Test that Newly Covers the Largest Number of Test Requirements
Randomly Add a Test that Was Not Selected
Iteration

FIGURE 2 Flow diagram of main steps in WalkTest.

IV. Experimental Setup

We describe the dataset of test suite reduction in our experiments and algorithms in comparison.

Online Resource for MultiWalk Experiments:
http://cstar.whu.edu.cn/p/multi-walk/
Supplementary materials of experiments and the dataset are publicly available.

A. Data Preparation

Table 1 lists ten large Java open source projects in our study. All these projects are widely-used and original test suites are provided together with the source code. In each project, the test suite is organized with the Java test framework, JUnit, which helps to automate the data collection below. Table 2 shows the number of tests and requirements of these ten projects. The size of test cases in each project ranges from 908 to 6,196 while the median of satisfied requirements per test ranges from 19 to 1,529.

To collect the runtime statement coverage between each test case and each statement, we configure and execute all projects as following steps. First, for each project, we manually configure the project according to its document. Source code and test code are locally compiled and executed to confirm that the whole test suite can pass the execution. Second, we instrument the source code to record whether each statement is executed, i.e., the a_{ij} value for a requirement r_i and a test t_j; meanwhile, we instrument the test code to identify which test case is executed during runtime. Third, based on the instrumentation, we run the project to record the satisfaction between tests and requirements. Before experiments, the above runtime collection is conducted for all projects and the resulted dataset is employed as the input of algorithms in comparison.

B. Algorithms in Comparison

We compare our proposed algorithm MultiWalk with seven other algorithms during experiments, ILP, GRE, HGS, RAPS, RWLS, GA, and WalkTest. As mentioned in Section II, ILP is an exact method, which may fail to find a solution due to the limited time. GRE [14] and HGS [29] are two typical heuristics for solving test suite reduction. GRE is an extended version of the greedy algorithm, which can fast find a near-optimal solution; HGS is a pioneering algorithm of test suite reduction and is designed based on manual analysis of test suites. RAPS [15] and RWLS [30] are two state-of-the-art heuristics for the set covering

TABLE 1 Description of ten large-scale real projects.

PROJECT	FULL NAME WITH VERSION	PROJECT DESCRIPTION
CAMEL	APACHE CAMEL CORE 2.15	A VERSATILE FRAMEWORK BASED ON ENTERPRISE INTEGRATION PATTERNS
ASSERTJ	ASSERTJ CORE †	AN ASSERTION ENHANCED FRAMEWORK FOR JAVA TESTING
CONFIGURATION	APACHE COMMONS CONFIGURATION 1.10	A GENERIC CONFIGURATION INTERFACE OF READING CONFIGURATION DATA
JGIT	JGIT †	A GIT INTERFACE FROM JAVA PROGRAMS
CLOSURE	GOOGLE CLOSURE COMPILER †	A FAST COMPILER FOR JAVASCRIPT
COLLECTIONS	APACHE COMMONS COLLECTIONS 4.0	AN ENHANCED LIBRARY FOR JAVA COLLECTIONS
JFREECHART	JFREECHART 1.0	A LIBRARY FOR PROFESSIONAL QUALITY CHARTS
LANG	APACHE COMMONS LANG 3.4	A LANGUAGE-SUPPORT ENHANCED LIBRARY FOR JAVA
JODATIME	JODA-TIME 2.8.2	AN ENHANCED LIBRARY FOR JAVA TIME AND DATE
MATH	APACHE COMMONS MATH 3.5	A MATH AND ALGORITHM LIBRARY FOR JAVA

†For this project, the source code is not provided with a specific version. Hence, we extract the master branch (on January 31th, 2016) in the version control system for experiments.

problem. RAPS is a meta-heuristic via randomized priority search; RWLS is an efficient search algorithm via weighting rows in the set covering model. GA [31] is a widely-used population-based algorithm in evolutionary computation. In addition, to show the effectiveness of the multi-level strategy in MultiWalk, we also show the results by WalkTest with multi-restart (WalkTest for short).

Experimental platform. All the experiments run on a PC with Intel Core 3.6 GHz CPU, 4GB memory, and Ubuntu 12.04. The code instrumentation for the runtime coverage collection [32] is implemented with a Java analysis framework, Spoon 4.0 [33]. Among the algorithms under evaluation, we call ILP with an off-the-shelf linear programming tool, lp_ solve 5.5 [34]. We implement all the other algorithms in Java JDK 1.7.

V. Experimental Results

We evaluate our algorithm on ten large real-world Java open source projects; meanwhile, an exact algorithm, ILP, and six heuristics are employed in comparing the size of tests after test suite reduction and the time cost.

A. Test Suite Reduction on Real-World Projects

Our proposed algorithm, MultiWalk, is compared with seven algorithms in this section. We setup algorithms in use as follows. The timeout of each algorithm is set to two hours (7,200 seconds). All the heuristics are executed for 30 times to obtain the average values; the random seed is the timestamp of the system clock. In WalkTest, the probability p is set to 0.5; 10000 tries are used to obtain local optima. In WalkTest with multi-restart, WalkTest is restarted for 100 times to achieve the best solution. In MultiWalk, WalkTest without restart is used as the embedded local search operator ϖ; the maximum number of levels is set to $\alpha = 20$ and the number of local optima in each level is set to $\beta = 10$. Further experiments in Section V-B will show the sensitivity to α and β in MultiWalk.

Table 3 shows both the test suite size after reduction and the time cost of running algorithms. In these experimental results, MultiWalk can obtain the best-known solutions on nine out of ten projects. One exception is the project Camel, which is the smallest project under consideration. ILP, as well as HGS, RAPS, and RWLS, can obtain the optimal solution on

TABLE 2 Tests and requirements of ten large-scale real projects.

			# SATISFIED REQUIREMENTS PER TEST				
PROJECT	# TESTS	# REQUIREMENTS	MIN	MEDIAN	MAX	AVERAGE	STDEV.
CAMEL	908	21551	1	257	7494	1656.9	2206.2
ASSERTJ	1219	3632	1	66	722	82.1	66.0
CONFIGURATION	1333	9062	2	237	2073	341.7	310.3
JGIT	1448	13678	1	85	4203	158.8	342.2
CLOSURE	1626	14936	1	1529	2586	1316.6	738.8
COLLECTIONS	4882	12512	1	69	591	92.9	87.3
JFREECHART	2247	29846	1	108	3945	293.6	557.6
LANG	2767	11857	1	19	377	42.1	56.0
JODATIME	4118	10567	1	150	1845	209.7	179.9
MATH	6196	44947	1	139	1714	236.3	248.7

The similarity between local and global optima is essential to conduct the instance reduction. An ideal backbone is expected to be one part of the global optimum but is hard to be found in practice.

Camel. Among ten projects, ILP can obtain six optimal solutions while the other four projects are not solved until the timeout of 7,200 seconds. MultiWalk can also obtain the optimal solutions in five out of six projects that ILP solves. RAPS can obtain the best-known solutions on seven out of ten projects, but may lead to more running time than MultiWalk, e.g., on the project, Configuration.

MultiWalk spends less time cost than ILP. In the five projects where both MultiWalk and ILP find optima, the average run times of MultiWalk and ILP are 18.2 and 1353.4 seconds, respectively. That is, MultiWalk is more efficient in the running time than ILP. Note that although our algorithm MultiWalk obtains the optimal solution on these five projects, there is no guarantee that MultiWalk can find the optimal ones (as shown on the project Camel).

Comparing MultiWalk with five heuristics, GRE, HGS, RAPS, RWLS, and GA, MultiWalk is more effective than the other heuristics. Comparing MultiWalk with WalkTest, MultiWalk can find better solutions with less running time. This shows that the multi-level strategy in MultiWalk can save the time of searching potential solutions.

B. Empirical Analysis of MultiWalk

We further analyze three factors of MultiWalk to investigate its ability of test suite reduction, i.e., the similarity between local optima and global optima, the change of backbone scales with the number of local optima, and the ability of reducing problem scales with levels.

Similarity between local and global optima. MultiWalk leverages the backbone, i.e., the intersection of local optima, to lock a subset of tests and then downgrades the problem scale. The backbone is expected to be similar to the optimal solution to find better solutions. Fig. 3 shows the box plots of the similarity between solutions by WalkTest (i.e., local optima) and optima by ILP on six projects (as shown in Table 3). Each box plot is based on 100 solutions by WalkTest.

TABLE 3 Comparison between MultiWalk and six algorithms on ten large projects by measuring both the test suite size after reduction and the running time of algorithms (in seconds).

			ALGORITHMS IN COMPARISON							
PROJECT	METRIC	ORIGINAL TEST SIZE	ILP	GRE	HGS	RAPS	RWLS	GA	WALKTEST	MULTIWALK
CAMEL	SIZE	908	**433.0**	434.4	**433.0**	**433.0**	**433.0**	435.0	434.0	433.3
	TIME		907.0	8.1	4.3	456.4	170.2	182.1	447.7	51.8
ASSERTJ	SIZE	1219	**466.0**	471.5	467.6	**466.0**	479.3	472.3	467.6	**466.0**
	TIME		1219.0	0.2	0.2	3.2	117.9	8.0	36.1	5.1
CONFIGURATION	SIZE	1333	**365.0**	367.0	367.6	**365.0**	384.6	373.0	368.7	**365.0**
	TIME		1333.0	1.1	1.0	14.3	233.3	21.7	110.9	12.7
JGIT	SIZE	1448	**472.0**	477.6	474.0	**472.0**	485.2	478.5	474.0	**472.0**
	TIME		1448.0	0.5	1.3	9.5	38.9	20.3	90.4	10.1
CLOSURE	SIZE	1626	**491.0**	501.0	495.7	**491.0**	495.0	503.7	495.3	**491.0**
	TIME		1626.0	14.1	4.6	289.7	250.4	175.0	508.2	56.7
COLLECTIONS	SIZE	4882	–	949.0	931.6	918.9	923.3	954.5	932.3	**917.6**
	TIME		TIMEOUT	3.5	2.9	10.8	25.7	29.4	125.0	23.3
JFREECHART	SIZE	2247	–	1011.8	997.4	**995.0**	1022.2	1001.3	999.3	**995.0**
	TIME		TIMEOUT	3.7	7.3	39.2	313.7	110.9	322.5	37.9
LANG	SIZE	2767	**1343.0**	1351.0	1349.5	**1343.0**	1365.3	1351.0	1346.0	**1343.0**
	TIME		2767.0	0.4	1.4	5.6	22.7	14.9	50.3	6.4
JODATIME	SIZE	4118	–	660.6	641.3	638.2	673.5	681.5	657.6	**630.8**
	TIME		TIMEOUT	5.4	4.0	25.1	29.3	49.4	133.7	28.5
MATH	SIZE	6196	–	2171.2	2138.8	2128.9	2135.1	2162.7	2143.6	**2125.2**
	TIME		TIMEOUT	12.6	91.0	155.7	170.5	252.9	629.3	72.8

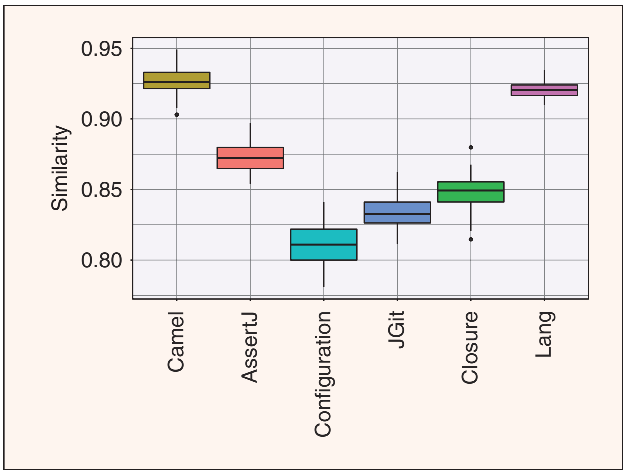


FIGURE 3 Similarity between 100 solutions by WalkTest and optima by ILP on six projects.

Consider a solution S and an optimum O of the original test suite T ($S \subseteq T$ and $O \subseteq T$), the similarity is defined as *similarity* $(S, O) = |S \cap O|/|O|$. As shown in Fig. 3, the median of similarity in five projects is between 0.80 and 0.95. This result indicates that solutions by WalkTest partially contain tests in the optima, which are obtained by ILP. Our proposed algorithm, MultiWalk, benefits from such similarity to conduct final solutions.

Backbone scales with the number of local optima. In MultiWalk, the backbone is extracted based on the intersection of local optima. Fig. 4 shows the change of backbone scales via increasing the number of local optima. For instance, the top curve shows that in the project Lang, the scale of the backbone decreases from 49% to 42% when the number of local optima increases from 1 to 15. In each of these ten projects, when the number of local optima increases from 1 to 3, the scale of backbones decreases fast. When the number of local optima is over 7, the curves tend to be stable; when the number of local optima is from 11 to 15, the change of curves is unnoticeable. Hence, as mentioned before, in MultiWalk, we set the number β of local optima to 10.

Ability of problem scale downgrading in levels. The downgrading of problem scales benefits from locking backbones and discarding shielded tests. In each level of MultiWalk, two opportunities lead to small problem instances. Fig. 5 illustrates the changes of problem scales based on levels. In this illustration, we use four projects that ILP cannot find optimal solutions. As shown in Fig. 5, in the first five levels, all the problem scales decrease fast. From Levels 5 to 15, the problem scale continuously decreases. In each level, locking the backbone results in discarding a number of shielded tests. In the project JFreeChart, locking the first backbone leads to the shielding of 70% tests. In all the four projects under consideration, both the backbones and shielded tests contribute to the scale reduction. In two projects, Collections and Math, 18 levels in Fig. 5 cannot reduce the problem scales to less than 0.1%. For any problem instance after given levels, the local search in MultiWalk will directly run to find a sub-optimal solution. To sum up, the design of the backbone and shielded tests makes MultiWalk find solutions that are similar to or even the same as the optimal solutions.

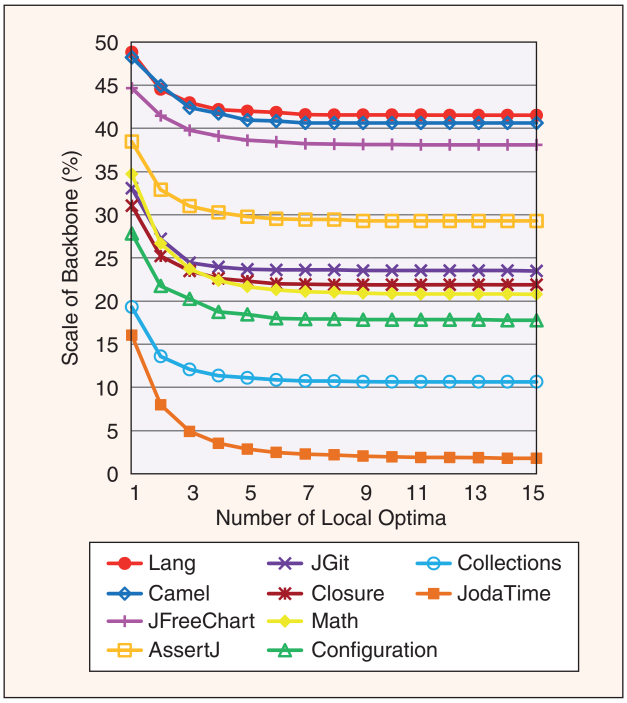


FIGURE 4 Change of backbone scales with the number of local optima on ten projects.

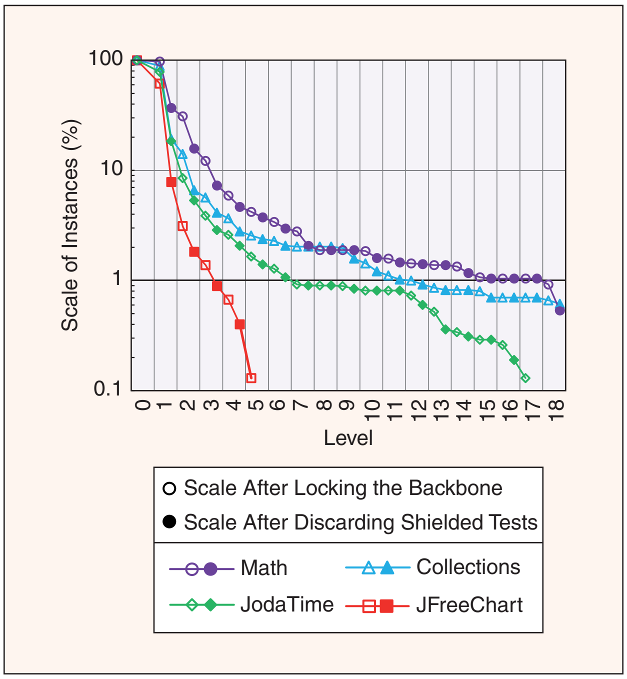


FIGURE 5 Scale of instances by locking backbones and discarding shielded tests on four projects, which ILP cannot solve. Note that the vertical axis is in the logarithmic scale.

The scale of the problem instance can be reduced in each level. The largest project under evaluation is reduced to 0.6% of its original scale within 18 levels.

VI. Discussions

We discuss the potential extension of our method, MultiWalk and the threats to the validity in this section.

A. Potential Extension

Extension of the general model of test suite reduction. As mentioned in Section II, the addressed model of test suite reduction can be extended due to refined application scenarios. For instance, the abilities of fault localization can be modeled and added as constraints of a test suite reduction instance [35], [36]; the cost of test execution can also be modeled as constraints [37]. Note that multi-objective test suite reduction (e.g., [38]) cannot be addressed by directly extending the model in this paper. Similar to other problems in multi-objective optimization, multi-objective test suite reduction investigates the diversification of solutions, which is not always focused in single-objective ones.

Extension for the non-optima in instances with small scales. Our proposed algorithm MultiWalk can find the best solutions among seven algorithms on nine out of ten projects of test suite reduction. However, MultiWalk fails in finding the best solution on one instance with the small scale. This motivates us to explore the drawback of the design in MultiWalk. A potential solution of solving this problem is to identify the hardness of instances and then employs different algorithms according to instances.

B. Threats to Validity

Construct validity. Our work as well as many existing works measures the size of a reduced test suite to evaluate the algorithm effectiveness. Such evaluation is based on a hypothesis that the executing time of each test is the same. However, this hypothesis does not always hold in practice due to the diversity of tests. In this paper, we consider the code coverage as the major objective; to involve the execution time, a multi-objective method could be designed [38].

Internal validity. To measure the similarity between solutions, we use the global optima by ILP for calculation. However, there is a threat that more than one optimum exists in one instance. The calculation should be based on a set of optima, rather than only one optimum. Due to the complexity of exhausting potential optima, our calculation can be viewed as a trade-off between the accurate similarity and the running time. The technique of fitness landscape analysis may contribute to reduce the risk of similarity calculation with multiple optima [39].

External validity. In our work, tests and requirements are extracted to serve as the datasets for the evaluation. All real-world projects in the experimental setup are Java programs. We have never evaluated our work on projects in other languages. There exists a threat to the generality of the effectiveness. A large amount of experiments on projects in different languages could help to reduce the threat.

VII. Related Work

The general model of test suite reduction reduces the cost of test execution and keeps test requirements satisfied. As mentioned in Section II, existing methods such as HGS, GRE and RAPS, have been empirically evaluated and these methods are effective to reduce the scale of test suites. These methods can be viewed as a method family of search based test suite reduction, which leverages optimization algorithms to find near-optimal and small test suites by reducing test cases.

Search based test suite reduction. Besides the algorithms in the experiment, there exist several works in search based test suite reduction. Zhong et al. [31] have shown that the Genetic Algorithm (GA) can work on the test suite reduction, but is not as effective as HGS. Tallam and Gupta [40] have proposed a concept analysis based on the greedy algorithm. Hao et al. [10] propose an on-demand approach to test suite reduction to balance the global and local greedy reduction methods. Different from direct solving test suite reduction in the above works, MultiWalk leverages the common selected tests and shielded tests to guide the optimization. The original problem is iteratively converted into a small one via analyzing current tests.

Domain knowledge in test suite reduction. Domain knowledge of software testing is involved to extend the above model of test suite reduction. Bengolea et al. [41] propose reduction techniques for bounded exhaustive testing. Shi et al. [42] combine the test suite reduction and test case selection to reduce the test cost. Several existing works have modeled test suite reduction via fault detection [11], [35], [37]. Multi-objective test suite reduction [7], [38] aims to find the trade-off of more than one objective, such as the scale, the execution cost, and the requested resources of test suites. Qian et al. [43] recently prove that Pareto optimization is more efficient than the penalty function method for obtaining the optimal and approximate solutions in the minimum cost coverage problem.

Multi-level search. Multi-level search is a strategy for reducing the search space via transforming the original problem instance. Walshaw [16] has designed multi-level search for the traveling salesman problem. Xuan et al. [44] propose a multi-level method to boost the search process of the next release problem. Jiang et al. [45] develop a new paradigm of search space transformation to reduce and smooth the potential space of high-quality solutions. Mahdavi et al. [46] propose a multiple hill climbing search via identifying building blocks for the software module clustering problem. In MultiWalk, we share the same algorithm framework, multi-level search [16], [44]. In contrast to existing works, the solution in each level is based on backbones and shielded tests, which are obtained via random walk search; backbones and shielded tests could reduce the distance between local optima and global optima.

VIII. Conclusions and Future Work

In this paper, we address the problem of software test suite reduction to support continuous development and testing. We propose MultiWalk, a multi-level random walk algorithm to solve test suite reduction. MultiWalk reduces the scales of problem instances by locking the backbone and discarding shielded tests. Experimental results show that MultiWalk is more effective than state-of-the-art heuristics of test suite reduction. Moreover, MultiWalk can efficiently find optima on five out of six projects, which ILP solves; for large projects that ILP fails to solve, MultiWalk provides the best solutions among algorithms in comparison.

In future work, we aim to improve MultiWalk to handle both large-scale and small-scale instances. Another future work is to further dig the hidden factors that MultiWalk can find optimal solutions in test suite reduction. We also plan to explore the usage of MultiWalk in other types of hard-to-solve problems.

Acknowledgment

This work is partly supported by the National Natural Science Foundation of China (under grants 61502345, 61572375, and 61403057).

References

[1] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proc. Int. Symp. Software Testing and Analysis*, Baltimore, MD, July 12–17, 2015, pp. 211–222.
[2] Hadoop. (2015). Apache Hadoop [Online]. Available: http://github.com/apache/hadoop/graphs/commit-activity.
[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.
[4] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Nov. 2012.
[5] A. Caprara, P. Toth, and M. Fischetti, "Algorithms for the set covering problem," *Ann. Oper. Res.*, vol. 98, no. 1–4, pp. 353–371, Dec. 2000.
[6] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations, The IBM Research Symposia Series*, R. Miller, J. Thatcher, and J. Bohlinger, Eds. New York: Springer-Verlag, 1972, pp. 85–103.
[7] H. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *Proc. 31st Int. Conf. Software Engineering*, Vancouver, Canada, May 16–24, 2009, pp. 419–429.
[8] J. Black, E. Melachrinoudis, and D. R. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proc. 26th Int. Conf. Software Engineering*, Edinburgh, Scotland, May 23–28, 2004, pp. 106–115.
[9] A. Gotlieb and D. Marijan, "FLOWER: Optimal test suite reduction as a network maximum flow," in *Proc. Int. Symp. Software Testing and Analysis*, San Jose, CA, July 21–26, 2014, pp. 171–180.
[10] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *Proc. 34th Int. Conf. Software Engineering*, Zurich, Switzerland, June 2–9, 2012, pp. 738–748.
[11] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 2, pp. 108–123, Feb. 2007.
[12] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *Proc. IEEE 22nd Int. Symp. Software Reliability Engineering*, Hiroshima, Japan, Nov. 29–Dec. 2, 2011, pp. 170–179.
[13] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, Hong Kong, China, Nov. 16–22, 2014, pp. 246–256.
[14] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," *Inform. Softw. Technol.*, vol. 40, no. 5/6, pp. 347–354, July 1998.
[15] G. Lan, G. W. DePuy, and G. E. Whitehouse, "An effective and simple heuristic for the set covering problem," *Eur. J. Oper. Res.*, vol. 176, no. 3, pp. 1387–1403, Feb. 2007.
[16] C. Walshaw, "A multilevel approach to the travelling salesman problem," *Oper. Res.*, vol. 50, no. 5, pp. 862–877, Sept./Oct. 2002.
[17] J. K. Slaney and T. Walsh, "Backbones in optimization and approximation," in *Proc. 17th Int. Joint Conf. Artificial Intelligence*, Barcelona, Spain, July 16–22, 2001, pp. 254–259.
[18] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh, "Backbones and backdoors in satisfiability," in *Proc. 20th National Conf. Artificial Intelligence*, Pittsburgh, PA, July 9–13, 2005, pp. 1368–1373.
[19] W. Zhang, A. Rangan, and M. Looks, "Backbone guided local search for maximum satisfiability," in *Proc. 18th Int. Joint Conf. Artificial Intelligence*, Acapulco, Mexico, Aug. 9–15, 2003, pp. 1179–1186.
[20] P. Kilby, J. K. Slaney, and T. Walsh, "The backbone of the travelling salesperson," in *Proc. 19th Int. Joint Conf. Artificial Intelligence*, Edinburgh, Scotland, July 30/Aug. 5, 2005, pp. 175–180.
[21] M. Greiler, A. van Deursen, and M. A. Storey, "Automated detection of test fixture strategies and smells," in *Proc. IEEE 6th Int. Conf. Software Testing, Verification and Validation*, Luxembourg, Mar. 18–22, 2013, pp. 322–331.
[22] O. Dubois and G. Dequen, "A backbone-search heuristic for efficient solving of hard 3-SAT formulae," in *Proc. 17th Int. Joint Conf. Artificial Intelligence*, Seattle, Washington, Aug. 4–10, 2001, pp. 248–253.
[23] S. Climer and W. Zhang, "Searching for backbones and fat: A limit-crossing approach with applications," in *Proc. 18th National Conf. Artificial Intelligence and 14th Conf. Innovative Applications of Artificial Intelligence*, Edmonton, Canada, July 28/Aug. 1, 2002, pp. 707–712.
[24] H. H. Hoos, "An adaptive noise mechanism for Walksat," in *Proc. 18th National Conf. Artificial Intelligence and 14th Conf. Innovative Applications of Artificial Intelligence*, Edmonton, Canada, July 28/Aug. 1, 2002, pp. 655–660.
[25] S. Cai, K. Su, and C. Luo, "Improving walksat for random k-satisfiability problem with k > 3," in *Proc. 27th AAAI Conf. Artificial Intelligence*, Bellevue, Washington, July 14–18, 2013.
[26] A. Coja-Oghlan and A. M. Frieze, "Analyzing Walksat on random formulas," *SIAM J. Comput.*, vol. 43, no. 4, pp. 1456–1485, July 2014.
[27] C. Qian, Y. Yu, and Z.-H. Zhou, "Analyzing evolutionary optimization in noisy environments," Evol. Comput., 2015.
[28] P. J. M. Laarhoven and E. H. L. Aarts, "Simulated annealing: Theory and applications," *Mathematics & Its Applications Series*, vol. 37. M. Hazewinkel, Ed. Dordrecht, Holland: Springer-Verlag, 1987, pp. 79–83.
[29] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, July 1993.
[30] C. Gao, X. Yao, T. Weise, and J. Li, "An efficient local search heuristic with row weighting for the unicost set covering problem," *Eur. J. Oper. Res.*, vol. 246, no. 3, pp. 750–761, Nov. 2015
[31] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Inform. Softw. Technol.*, vol. 50, no. 6, pp. 534–546, May 2008.
[32] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.
[33] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of Java source code," *Softw. Pract. Exp.*, vol. 46, no. 9, pp. 1155–1179, Sept. 2016.
[34] M. Berkelaar, K. Eikland, and P. Notebaert. (2004). lp_solve 5.5.2.0 [Online]. Available: http://web.mit.edu/lpsolve_ v5520/doc/index.htm.
[35] D. Gong, T. Wang, X. Su, and P. Ma, "A test-suite reduction approach to improving fault-localization effectiveness," *Comput. Lang. Syst. Struct.*, vol. 39, no. 3, pp. 95–108, Oct. 2013.
[36] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "B-Refactoring: Automatic test code refactoring to improve dynamic analysis," *Inform. Softw. Technol.*, vol. 76, pp. 65–80, Aug. 2016.
[37] A. G. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the cost-benefits tradeoffs for regression testing techniques," in *Proc. Int. Conf. Software Maintenance*, Montreal, Canada, Oct. 3–6, 2002, pp. 204–213.
[38] S. Yoo and M. Harman, "Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation," *J. Syst. Softw.*, vol. 83, no. 4, pp. 689–701, Apr. 2010.
[39] P. Merz and B. Freisleben, "Fitness landscape analysis and memetic algorithms for the quadratic assignment problem," *IEEE Trans. Evol. Comput.*, vol. 4, no. 4, pp. 337–352, Dec. 2000.
[40] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Engineering*, Lisbon, Portugal, Sept. 5/6, 2005, pp. 35–42.
[41] V. S. Bengolea, N. Aguirre, D. Marinov, and M. F. Frias, "RepOK-based reduction of bounded exhaustive testing," *Softw. Test. Verif. Reliab.*, vol. 24, no. 8, pp. 629–655, Dec. 2014.
[42] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proc. 10th Joint Meeting Foundations Software Engineering*, Bergamo, Italy, Aug. 30/Sept. 4, 2015, pp. 237–247.
[43] C. Qian, Y. Yu, and Z. Zhou, "On constrained Boolean Pareto optimization," in *Proc. 24th Int. Joint Conf. Artificial Intelligence*, Buenos Aires, Argentina, July 25–31 2015, pp. 389–395.
[44] J. Xuan, H. Jiang, Z. Ren, and Z. Luo, "Solving the large scale next release problem with a backbone-based multilevel algorithm," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1195–1212, Sept./Oct. 2012.
[45] H. Jiang, Z. Ren, X. Li, and X. Lai, "Transformed search based software engineering: A new paradigm of SBSE," in *Proc. 7th Int. Symp. Search-Based Software Engineering*, Bergamo, Italy, Sept. 5–7, 2015, pp. 203–218.
[46] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," in *Proc. 19th Int. Conf. Software Maintenance*, Amsterdam, The Netherlands, Sept. 22–26, 2003, pp. 315–324.