Mongiovì *et al. Appl Netw Sci* (2020) 5:86
https://doi.org/10.1007/s41109-020-00323-w

Applied Network Science

RESEARCH Open Access

REDUNET: reducing test suites by integrating set cover and network-based optimization

Misael Mongiovì[1,2*], Andrea Fornaia[2] and Emiliano Tramontana[2]

*Correspondence:
misael.mongiovi@istc.cnr.it
[1] ISTC CNR, Via Gaifami, 18, 95126 Catania, Italy
Full list of author information is available at the end of the article

Abstract

The availability of effective test suites is critical for the development and maintenance of reliable software systems. To increase test effectiveness, software developers tend to employ larger and larger test suites. The recent availability of software tools for automatic test generation makes building large test suites affordable, therefore contributing to accelerating this trend. However, large test suites, though more effective, are resources and time consuming and therefore cannot be executed frequently. Reducing them without decreasing code coverage is a needed compromise between efficiency and effectiveness of the test, hence enabling a more regular check of the software under development. We propose a novel approach, namely REDUNET, to reduce a test suite while keeping the same code coverage. We integrate this approach in a complete framework for the automatic generation of efficient and effective test suites, which includes test suite generation, code coverage analysis, and test suite reduction. Our approach formulates the test suite reduction as a set cover problem and applies integer linear programming and a network-based optimisation, which takes advantage of the properties of the control flow graph. We find the optimal set of test cases that keeps the same code coverage in fractions of seconds on real software projects and test suites generated automatically by Randoop. The results on ten real software systems show that the proposed approach finds the optimal minimisation and achieves up to 90% reduction and more than 50% reduction on all systems under analysis. On the largest project our reduction algorithm performs more than three times faster than both integer linear programming alone and the state-of-the-art heuristic Harrold Gupta Soffa.

Keywords: Test suite reduction, Static analysis, Graph analysis, Network analysis

Introduction

Test cases ensure that a software system is checked against possible defects and let developers assess its correctness. Generally, test developers determine the input values and expected output values for each test case, and then implement a test case that executes a functionality with the chosen input values and compare the received output with the expected output. Since this activity can be time consuming, some tools are available to assist the automatic generation of tests, e.g. for choosing input values and combining them for an execution scenario (Kuhn et al. 2015). Testing frameworks, such as JUnit[1]

[1] https://junit.org/.

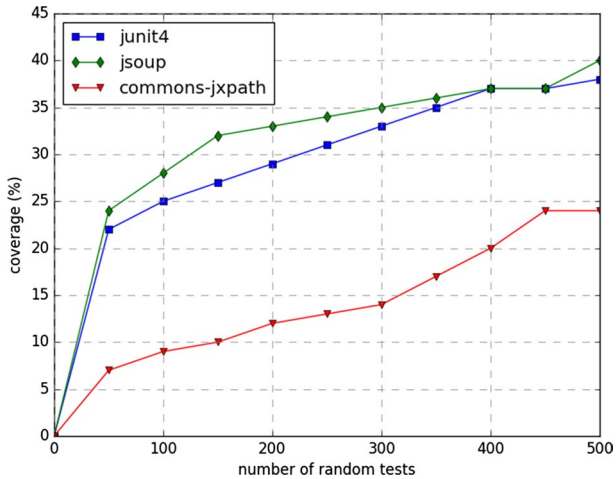

Fig. 1 Code coverage. Number of generated test cases and their related code coverage for some analysed software systems

for Java and their counterparts for other languages, are widely used to make it easier for the developer to implement test cases. Moreover, test generation tools spare developers many coding activities related to the production of test cases.

Many tools generating test code find methods of classes and parameters and produce method calls, with proper values for parameters and the return value. The most prominent of such tools are Randoop (Pacheco and Ernst 2007) and EvoSuite (Fraser and Arcuri 2011, 2012). Randoop produces test cases having a small number of calls to methods, which have been chosen from the analysed code randomly (Pacheco and Ernst 2007). EvoSuite produces test cases consisting of calls to methods, which are selected using evolutionary computation over a population (Fraser and Arcuri 2011). Surely, test generation is useful to check contract violation and for regression tests. However, the number of generated test cases tend to be very large, given the many combinations of calls to methods that can be found. Moreover, for automatic generation of test cases, the percentage of code coverage tends to reach a plateau, where additional test cases generated leave code coverage largely unchanged. As a result the efficacy of additional tests is limited, whereas the time needed to run all tests can be very large and take up many resources (Shamshiri et al. 2015; Kazmi et al. 2017). This effect can be seen when automatically generating test cases for *junit4*, *jsoup* and *commons-jxpath*, having several thousands of lines of code. Figure 1 shows a plot relating the number of automatically generated test cases and the measured code coverage of the systems under test. The executed test cases cover between 24 and 40% of the lines with 500 test cases. The coverage increases slowly when adding test cases, e.g. for *jsoup* when the number of test cases goes from 150 to 450 (a threefold increase), coverage goes from 32 to 37% (about a 15% increase).

Generally, all tests available are executed automatically as soon as developers push new components on a source code repository. When following Agile development processes, all tests could start executing every two hours, hence a capable hardware is needed to have test results in a timely fashion. Considering that the number of

automatically generated test cases could be several thousands, either the hardware resources are very capable or the time for running tests could be not enough for the development practice of continuous integration. Moreover, when test cases are automatically generated, some of the statements on components under test are executed several times because covered by several test cases. Therefore, for the sake of efficiency on hardware resources and running time, test cases should be reduced.

We propose a comprehensive framework for automatic generation and reduction of test suites. While we rely on existing approaches for test generation (Pacheco and Ernst 2007), we focus our effort on test suite reduction, i.e. finding a subset of test cases that cover the same amount of code of the original test suite while minimising its execution time. Although several test suite reduction methods have been designed in the past (Harrold et al. 1993; Tallam and Gupta 2006; Smith and Kapfhammer 2009; Xu et al. 2012; Chen et al. 2008; Jatana et al. 2016; Coviello et al. 2018a) and they could potentially be employed in our framework, such methods either rely on heuristics, which are not able to find the optimal reduction, or employ expensive approaches, which do not scale well on large projects. Moreover, existing methods focus on preserving the coverage of generic requirements, hence they are not specifically optimised for preserving code coverage, and many of them do not specifically focus on minimising the test suite execution time. Unlike the said approaches, our approach REDUNET finds the optimal reduced test suite in terms of execution time in real software projects, and therefore increases the ability to give a quick feedback to the developers. A more extensive comparison with the related work is given in a dedicate section.

The problem of test reduction has been constructed as an optimisation problem aimed at finding the minimum number of test cases that, while keeping the same code coverage, minimises execution time. REDUNET manages to select test cases very quickly, hence it does not impact on used resources.

In our previous contribution (Mongiovì et al. 2019), we have employed Integer Linear Programming (ILP) and a Control Flow Graph (CFG) analysis to reduce the number of test cases while ensuring that code coverage (specifically, *statement coverage*) is kept unchanged, however without considering test execution time. In this work we improved both effectiveness and efficiency of our approach, by taking into account the test suite execution time and enhancing the ability to reduce the computational cost of the optimisation. We also propose a more comprehensive validation of the approach, having performed experiments on ten real software systems. REDUNET is flexible enough to enable its use with other metrics different than test execution time and statement coverage (e.g. *branch coverage*).

Our experiments have shown that REDUNET, while managing to keep the same code coverage, greatly reduces the test suite execution times, in some cases to one-tenth of the original time frame, finding the optimal reduction in all software projects under analysis. The proposed reduction approach is also more than three times faster than ILP alone and than the state-of-the-art heuristic Harrold Gupta Soffa (HGS) (Harrold et al. 1993; Coviello et al. 2018a) on the largest software project, and almost always faster than our previous approach (Mongiovì et al. 2019) thanks to a more careful reduction of the input for the ILP step.

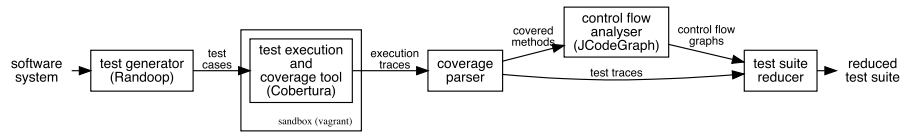

Fig. 2 General framework. The developed tool-chain consisting of test generation, code coverage assessment, control flow graph extraction, and reduced test suite computation

With respect to our previous work (Mongiovì et al. 2019), our contribution is threefold.

- We consider the running time of the test suite as a more appropriate parameter than the number of test cases and adapt our method accordingly;
- We improve the efficiency of our method by using a more effective approach to reduce the size of the data for the optimisation consisting in removing the back edges of the control flow graph;
- We perform an extensive evaluation of the proposed approach on ten real software systems.

In the following, we describe our approach and framework, then we present the problem formulation, the proposed solution, and experimental results. Eventually, we discuss relevant related works and conclude the paper.

A comprehensive approach for the automatic generation and reduction of test suites

Our complete framework includes the automatic generation of test suites and a test reduction process aimed at reducing the resulting test suite execution time without affecting coverage. In the literature, test reduction approaches that preserve coverage are said *adequate*, whereas inadequate ones reduce test suites that partially preserve coverage (Shi et al. 2014; Coviello et al. 2018b). We used *statement coverage* as a code coverage metric. Figure 2 shows the steps performed by the implementation of our complete framework.

First, a test suite is generated for the software system to be analysed, e.g. by using Randoop (Pacheco and Ernst 2007). We used Randoop to generate a large number of test cases that offer as much coverage as possible. Randoop was chosen over EvoSuite (Fraser and Arcuri 2011) since the former is much faster than the latter, and test cases produced by EvoSuite would still be liable of the reduction process. However, manually implemented test cases or test cases generated by another tool could be used too as input for the following steps.

Second, the generated test cases are added to the maven project as Java unit tests.

Third, test cases are executed together with a service tracing code coverage. Tests are executed on a virtual environment that has been created using a Vagrant box.[2] Vagrant lets us configure a virtual environment for execution, hence protecting the real system (the real file system is not accessed, hence avoiding the content of files to be potentially changed by the test execution). For tracing code coverage, we have used Cobertura,[3]

[2] https://www.vagrantup.com.

[3] https://cobertura.github.io/cobertura/.

which gives us the ids of the executed code lines for each test case run. Producing coverage information may introduce up to 30% of time overhead during test execution (Cruciani et al. 2019). Therefore, the coverage information is produced and collected only for the classes of the system under test, i.e. not for the external libraries, since the latter are out of the scope of the developer's testing activities. The outputs of each test case execution are: (1) the lines of code that have been covered, and (2) the execution time.

Fourth, we determine method coverage, i.e. the methods that have been (partially or fully) executed when running a test case. This results from the structure of the software system and its control flow, found by static code analysis, as well as the traces for the executed lines of code (our component JCodeGraph implements the proper analysis). Eventually, control flow graph and test execution traces are given to the test suite reducer, which reduces the number of test cases (later used for regression testing), to achieve a much shorter execution time, while ensuring the same code coverage.

The reduction step is the core of our tool, and is detailed in the following subsections.

Problem formulation and overall test suite reduction method

Formally, a *test suite* is a set of *test cases*, where a test case is represented by a specific executable configuration of the input parameters. Running a test case consumes a certain amount of computational resources, which we quantify in terms of *execution time*.[4] When a test case is run, it produces an *execution trace* (or simply a *trace*), i.e. a sequence of visited instructions (i.e. statements). We say that a test case *covers* an instruction if such an instruction is executed when the test case is run. The set of instructions covered by a set of test cases is named *code coverage* (or simply *coverage*).

Our *test suite reduction* problem can be formulated as follows: given a test suite, find a subset of test cases that maintains *code coverage* while minimising execution time, i.e. whose code coverage is the same as the whole test suite and the running time of its execution is minimum.

Given a test case t, we denote with S_t the set of instructions covered by t, and with c_t the running time (cost) of executing t. Given a test suite $\mathcal{T} = \{t_1, t_2 \ldots, t_n\}$, its coverage is defined as $\mathcal{S}_\mathcal{T} = \bigcup_{t \in \mathcal{T}} S_t$. Formally, our problem is defined as:

$$\begin{aligned} &\min_{\mathcal{T}'} \sum_{t \in \mathcal{T}'} c_t, \\ &\mathcal{S}_{\mathcal{T}'} = \mathcal{S}_\mathcal{T} \\ &\mathcal{T}' \subseteq \mathcal{T} \end{aligned}$$

Table 1 summarises the notation used in this paper.

Our optimisation problem corresponds to a variant of *set cover* called *weighted set cover*. Set cover is a classic computer science problem that, given a family of sets, calls for finding the minimum-size sub-family that covers all elements in the input family. Figure 3 shows an instance of set cover with four sets (S_1, S_2, S_3 and S_4) covering together 16 elements (black dots). In this example, an optimal solution for set cover consists of three sets ($\{S_1, S_2, S_4\}$), since there is no pair of sets that cover all 16 black dots. In weighted set

[4] our approach can be generalised to any additive cost function, e.g. the total CPU time.

Table 1 Summary of the notation used in this paper

Symbol	Name	Description
$\mathcal{T}$	Test suite	The initial set of test cases
$\mathcal{T}'$	Optimised test suite	Resulting set of test cases, which minimise the execution time
S_t	Test case coverage	The set of instructions covered by test case t
$\mathcal{S}_\mathcal{T}$	Test suite coverage	Set of instructions covered by at least one test case in $\mathcal{T}$
c_t	Execution time (cost)	Execution time of test case t
x_t	Test selection variable	Binary variable valued 1 if the test case t is taken, 0 otherwise
G	CFG	The control flow graph of the program under testing
V	Vertices	Set of vertices in a CFG
E	Edges	Set of edges in a CFG
v,u	Vertex	A vertex of a CFG, corresponding to an instruction
e	Edge	An edge of a CFG, showing contiguous instructions or a jump

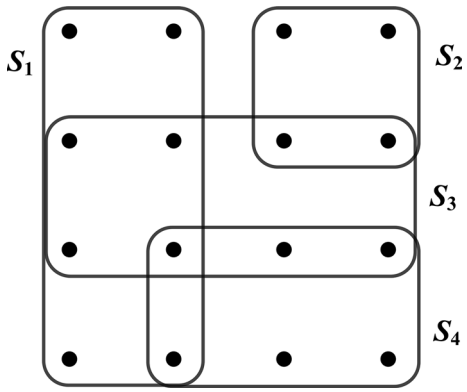

Fig. 3 An instance of set cover. $\{S_1, S_2, S_4\}$ is the minimum-size subfamily that covers all elements of the input family (black dots)

cover, each set is associated to a cost (weight) and the goal is to minimise the total cost of the taken sets.

In our test reduction problem, each test case is associated to its set of covered instructions and its execution time. The family of sets corresponds to the whole test suite. The optimal sub-family corresponds to the resulting test suite, which has the same coverage as the initial one and is minimal, i.e. there is no way to obtain the same coverage with shorter execution time by drawing from the initial test suite.

As set cover, weighted set cover is NP-hard and does not have a constant-factor approximation guarantee. Therefore, it is not possible to design an algorithm that solves it with an error within a fixed multiplicative factor of the returned solution. A greedy algorithm for set cover solves it with a factor-$log(n)$ approximation, i.e. with a logarithmic-function error bound (Feige 1998).

Although the greedy algorithm could be applied to reduce test suites, the result would be sub-optimal. This translates to a higher number of test cases to run and therefore a longer running time of the test execution. Computing an optimal solution, on the other hand, would require a high amount of time and computational resources and would be feasible only on sufficiently small problem instances.

We resort to an intermediate solution that is based on *Integer Linear Programming (ILP)* and the properties of the *Control Flow Graph (CFG)* of the software project. An instance of set cover can be easily redefined as an instance of ILP and given to a solver. There are several open and commercial solvers for ILP in the market. They employ a bunch of sophisticated techniques to find an optimal solution or, in the case it cannot be found within a user-defined time limit, an approximated solution with measurable error from the optimal one. This approach makes the computation feasible even on large software projects, but exhibits a decrease in accuracy with the size of the problem instance. Therefore, to maximise the number of times that the optimal solution can be found, we designed a method for reducing the size of the input by performing an accurate analysis of the CFG. In the remainder of this section we describe the ILP reduction and our CFG analysis for reducing the problem instance.

ILP-based test suite optimisation

The problem defined above can be described as an ILP program:

$$\begin{aligned} &\text{minimise} \sum_{t \in \mathcal{T}} c_t \cdot x_t, \\ &\text{subject to} \sum_{t : i \in S_t} x_t \geq 1 \quad \text{for all} \;\; i \in \mathcal{S}_{\mathcal{T}}, \\ &\qquad\qquad x_t \in \{0, 1\} \quad \text{for all} \;\; t \in \mathcal{T} \end{aligned}$$

$\mathcal{T}$ is the input test suite and $\{x_t\}$ are integer binary variables that describe which test cases are taken. Specifically, the value of x_t is 1 if test case t is chosen, 0 otherwise. Solving the ILP problem consists in finding the $\{x_t\}$ variables assignment that minimises the cost function (total execution time of the taken test cases) and satisfies the constraints, which guarantee to have every instruction (of the input test suite coverage) covered at least once.

Existing ILP solvers employ a variety of techniques, including cutting-planes, branch-and-bound and dynamic programming, to find an approximate solution with error within a measurable bound. With small or not computationally hard instances, the error bound can be brought to zero and an exact solution is returned. In this work we employed the open-source solver GLPK.[5] It is based on the branch-and-cut algorithm, a combination of branch and bound and cutting plane, and employs the revised simplex algorithm and the primal-dual interior point method to compute lower bounds (upper bounds for maximisation problems) to the optimal solution and prune the search space by discarding branches that cannot produce optimal solutions. If the time limit expires before finding an optimal solution, the system returns the *gap*, i.e. the relative distance of the best lower bound from the best solution found (in percentage). The gap represents the error since the optimal solution must lie between the best lower bound and the best solution found.

[5] http://www.gnu.org/software/glpk/.

```
1  public class ShoppingCart {
2    double daily_discount = 0.20;
3
4    public double getTotal(int price, int qty, int discount_level) {
5      double total = 0;
6
7      // input validation
8      if (price < 0 || qty < 0
9        || discount_level < 0 || discount_level > 3)
10       return -1;
11
12     if (price == 0) {
13       if (qty > 1) {
14         total = -1;
15       } else {
16         total = 0;
17       }
18     } else {
19       total = price * qty;
20
21       for (int i = 0; i < discount_level; i++) {
22         total *= (1 - 0.10);
23       }
24
25       total += (1 - daily_discount);
26     }
27
28     return total;
29   }
30 }
```

Fig. 4 Sample class ShoppingCart. A sample class having several execution paths

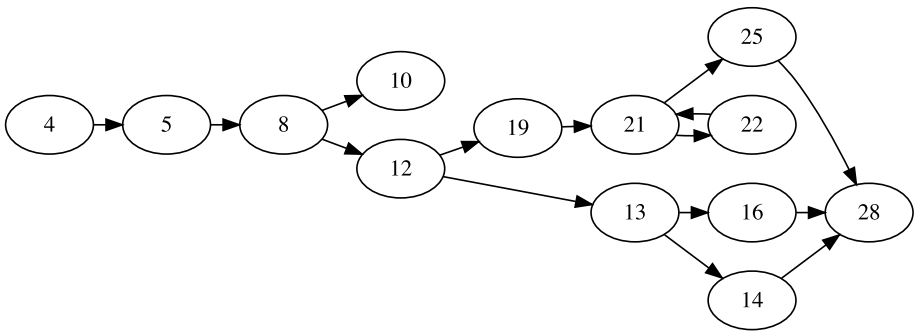

Fig. 5 Control flow graph. Control flow graph for the ShoppingCart class

Increasing efficiency by integrating CFG analysis

A *control flow graph (CFG)* is a static representation of the execution flow of a program. We exploit the properties of the CFG for reducing the input size for the ILP program. Specifically, we show that by CFG analysis we can select a subset of covered instructions for the optimisation problem without affecting correctness.

A CFG is a *graph* $G = (V, E)$, where V is a set of *vertices*, which represent program instructions, and E is a set of (directed) *edges* in the form $e = (u, v)$, where $u, v \in V$ are called *source* and *target*, respectively. We also say that e is an *outgoing* edge from u and an *incoming* edge to v. Two vertices $u, v \in V$ in a CFG are connected by an edge ($(u, v) \in E$) if instructions u and v can be executed subsequently either because

they are contiguous or because of jumps in the control flow (i.e. in the presence of conditional statements, loop statements or method calls). Figure 4 shows class ShoppingCart and Fig. 5 shows its CFG for method `getTotal()`, where vertices are labelled with corresponding instruction line numbers. Vertex 4 corresponds to the first instruction; vertices 8, 12 and 13 represent conditional statements, hence they have more than one outgoing edge. Vertices 21 and 22 represent a *for loop* header and its body, respectively.

Our CFG analysis is aimed at identifying instructions that are guaranteed to be executed even if excluded from the input given to the ILP step. In the program shown in Figs. 4 and 5, if a test t covers instruction 10, we can be sure that it also covers instructions 8, 5 and 4, since the only path from the first instruction through vertex 10 traverses such vertices. In this case we might choose to include vertex 10 and exclude vertices 8, 5 and 4 from the computation without affecting code coverage.

Before applying our algorithm we remove from the CFG instructions that are not covered by the input test suite, since they do not affect the computation. The main idea of our algorithm is based on the observation that for each edge (u, v) of the CFG whose target v has only one incoming edge, we can remove the source u from the ILP optimisation step without loosing its coverage since v can be reached only through u. Note that v might itself be a source of another edge (v, z), hence it might be removed as well, but in this case its coverage would still be guaranteed by the presence of z or one of its descendant.

We can improve the performance of our input size reduction procedure by cleaning the CFG of back edges of the depth first search tree within methods. In short, a back edge is an incoming edge to an ancestor of the depth first search tree. A back edge prevents its target vertex to be removed since it contributes to increase (to at least 2) its number of incoming edges. However, within a method, a back edge occurs only because of a loop statement or a recursive call; in both cases, to reach the source of the back edge we must explore its target, which correspond to the first instruction of the loop (or the method declaration of the recursive call). E.g., in Fig. 5, intuitively vertex 19 can be excluded by the ILP computation since vertex 22 guarantees its coverage (if vertex 22 is reached we can be sure that vertex 19 has been visited). However, without removing back edges, vertex 19 cannot be excluded since its successor, vertex 21, has two incoming edges. If we remove the edge that connects vertex 22 to vertex 21, which is a back edge, vertex 21 is left with just one incoming edge, hence vertex 19 can be excluded.

Our complete algorithm is reported in Algorithm 1. It first initialises $\mathcal{S}$ as the set of covered instructions and cleans the CFG of vertices that are not covered by the test suite and back edges. Then, it visits all edges (u, v) in the CFG and removes sources of edges whose targets have exactly one incoming vertex. Eventually the algorithm returns the family for the ILP computation, after removing elements that are not in $\mathcal{S}$ and empty sets. In the example in Fig. 5, $\mathcal{S}$, initially composed by all vertices, is reduced to $\mathcal{S} = \{10, 25, 22, 16, 14, 28\}$.

Algorithm 1: CFG-based input size reduction for the ILP computation

Result: Return a reduced input for the optimisation of test suite $\mathcal{T}$ based on CFG $G = (V, E)$
$\mathcal{S} \leftarrow \bigcup_{t \in \mathcal{T}} S_t$;
remove vertices that are not in $\mathcal{S}$ and their incident edges from G ;
remove back edges of the depth first search tree from G;
for *every* $(u, v) \in E$ **do**
  if v *has exactly one incoming vertex* **then**
    $\mathcal{S} \leftarrow \mathcal{S} \backslash \{u\}$;
  end
end
return $\{S_t \cap \mathcal{S} : S_t \cap \mathcal{S} \neq \emptyset, t \in \mathcal{T}\}$;

Employing other metrics

The proposed approach is flexible enough to be applied with other metrics different than test execution time and code coverage. E.g., we might want to consider for a test the number of previously detected faults, or the number of mutants detected when adopting mutation testing (Myers et al. 2011; DeMillo et al. 1978). Alternatively, we might want to focus on covering a portion of code that is more likely affected by changes, hence more susceptible to errors, motivated by considerations similar to those found in Panda et al. (2016).

In general, we can substitute code instructions with any other set of elements we aim at covering and the test execution time with any other metrics we aim at minimising. E.g., we could restrict the code instructions to cover a suitably selected subset, or substitute the metric "code instructions covered" with "faults detected" (or "mutants killed" or any other properties of test cases), without changing the other parts of the proposed approach and framework.

When considering the code coverage provided by test cases, CFG analysis has been used in our approach for excluding some instructions before starting the computation of the reduced test suite. Should code coverage be substituted by some other metrics, then our CFG analysis would have to be changed. A similar CFG analysis could be implemented in case a different coverage measure is considered. E.g., when using branch coverage (Godboley et al. 2017) we may reduce the number of branches to be considered according to their control dependencies (e.g. in case of nested branches) (Arcuri 2011).

We might also want to include multiple optimisation criteria, e.g. minimise test execution time and maximise the sum of fault detected (e.g. in a set of mutants) by each test case, while maintaining code coverage. In general, this would require solving a multi-objective optimisation problem, which is out of the scope of this work. However, in some cases the criteria can be combined in a unique linear objective function, hence easily integrated in our approach. E.g., we might consider for each test case the ratio between the execution time and the number of faults detected, hence the following objective function:

$$\min_{\mathcal{T}'} \sum_{t \in \mathcal{T}'} \frac{c_t}{f_t}, \tag{1}$$

where c_t is the execution time of test case t and f_t is its number of detected faults.

Table 2 Sizes of the software systems (number of classes, methods and lines of code) and the related test suites

Software	Classes	Methods	Lines	Test cases	Covered lines
chord-client	11	61	610	73	143
junit4	347	1772	5216	532	2046
jsoup	256	1697	7691	1171	3446
commons-cli	27	291	1213	801	779
commons-codec	102	916	4413	1303	2774
commons-collections	536	4449	13593	1391	3227
commons-jxpath	187	1632	9405	1173	3294
commons-math	917	6287	36603	1345	7066
jackson-core	124	2182	15796	1040	2428
jackson-dataformat-xml	55	520	2509	1031	805

Column "Test cases" gives the number of test cases of the whole test suite; column "Covered lines" reports the number of instructions covered by the whole test suite

Experimental results

We evaluated our approach by executing the implemented tool-chain on ten open source projects implemented in Java. Details about considered versions and other replication data can be found in the *REDUNET* repository.[6] We generated a test suite for each of them by using Randoop (Pacheco and Ernst 2007). For every software system, we set the maximum allowed time to generate the random test suite to 50–150 s overall, consisting of 10–30 runs of Randoop of 5 s each, with a different random seed on each run. This lets us produce a test suite having a size of around 1000 test cases in most cases (a single run for Randoop starting from one random seed gives much less test cases, even when its running time is not bounded). For the smallest system (610 lines of code), *chord-client*, 73 generated test cases gave the maximum instruction code coverage that could be achieved. All the other analysed systems were much bigger, ranging from 1213 lines of code to 36,603 lines of code, and instruction code coverage varied from 15 to 65%.

Table 2 reports the measured sizes for each analysed software system and the related generated test suites, i.e. starting from the left column, the table gives: the name of the software system, the number of classes, the number of methods, the total number of lines of code, the number of generated test cases, the coverage (as the number of code lines that have been executed when running the test suite).

We implemented our tool for test suite optimisation in Java 10. We employed JCodeGraph, a tool of ours based on JavaPDG (Shu et al. 2013) for computing the CFG. We integrated GLPK[7] for solving the ILP program and the library JGraphT[8] for managing graphs. All ILP programs in this evaluation have been solved optimally by GLPK in less than 2 s. Therefore we did not set a time limit for the solver and do not report the error (gap) since it is always zero.

We compared the proposed method with three state-of-the-art approaches. We included a previous version of our system, described in the introduction (Mongiovì

[6] https://github.com/afornaia/redunet.

[7] http://www.gnu.org/software/glpk/.

[8] https://jgrapht.org/.

Table 3 Approaches. Description of approaches compared in this experimental analysis

Approach	Description
Do nothing	No optimisation performed, just the initial test suite is returned
Greedy	Greedy algorithm for set cover, previously adopted by Smith and Kapfhammer (2009)
HGS	Harrold Gupta Soffa (HGS) test-reduction method (Harrold et al. 1993)
CN	Test suite optimisation of our previous tool in Mongiovì et al. (2019), designed to minimise the number of test cases
ILP	ILP-based test suite optimisation as described in the first part of the method section
ILP+CFG	ILP-based test suite optimisation integrated with the CFG analysis, except the back edges removal
REDUNET	Our complete tool, i.e. ILP + CFG + back edges removal

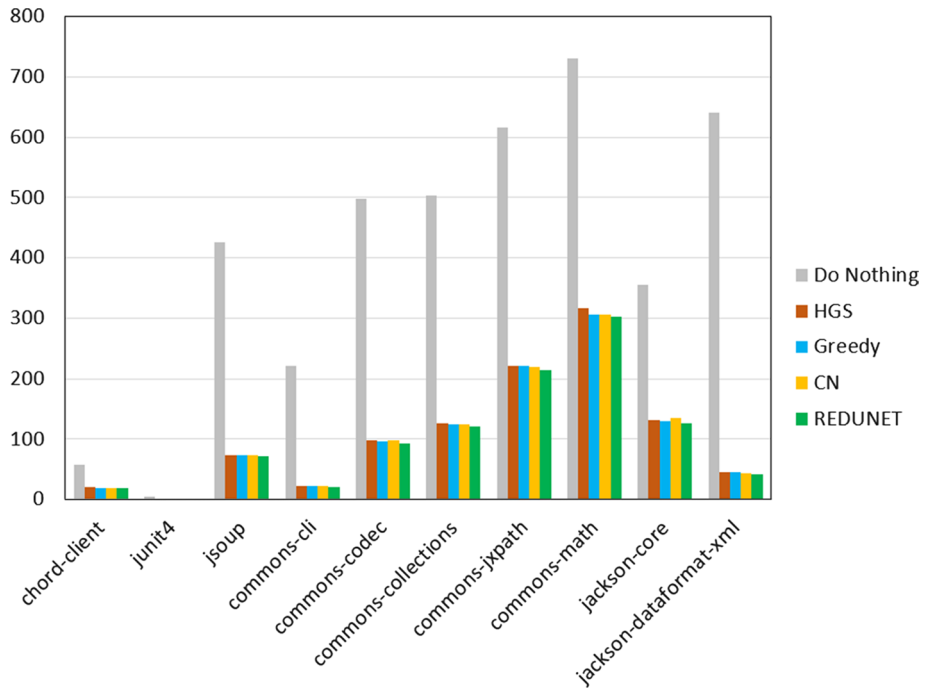

Fig. 6 Effectiveness of our test suite optimisation Running time in seconds of the test suites after the reduction (y-axes). The labels in the x-axis represent the ten software projects under analysis, whose details are reported in Table 2. The series represents the approaches under comparison, whose details are given in Table 3

et al. 2019). We also implemented a greedy algorithm for weighted set cover (Vazirani 2013), a variant of the traditional greedy algorithm for set cover, previously applied for test suite reduction (Smith and Kapfhammer 2009), which aims at minimising the total execution time (we select at each step the test that maximises the ratio between its uncovered elements and its weight, i.e. execution time). Last, we implemented the Harrold Gupta Soffa (HGS) method (Harrold et al. 1993), which has been shown to have superior or similar performances than other heuristics such as Greedy, 2-Optimal, Delayed Greedy, GE and GRE (Coviello et al. 2018a). We also included some variants of our method to assess the impact of including specific techniques. We did not perform tests with methods that do not guarantee code coverage (inadequate

Table 4 Running time in seconds of the test suites after the reduction

Software project	Do Nothing	HGS	Greedy	CN	REDUNET
chord-client	57.1	19.7 (0.0)	18.9 (0.0)	18.8 (0.0)	18.3 (0.0)
junit4	4.51	1.54 (0.7)	1.56 (0.2)	1.55 (0.3)	1.47 (0.3)
jsoup	425.1	73.0 (0.4)	74.2 (0.2)	72.9 (0.4)	71.4 (0.4)
commons-cli	221.4	22.6 (0.0)	22.8 (0.0)	22.0 (0.1)	21.3 (0.1)
commons-codec	498.3	99.0 (0.2)	96.5 (0.0)	97.6 (0.2)	93.3 (0.2)
commons-collections	502.9	126.2 (0.1)	125.3 (0.0)	125.4 (0.3)	120.3 (0.3)
commons-jxpath	616.1	221.1 (0.3)	220.6 (0.0)	219.9 (0.2)	215.2 (0.2)
commons-math	730.2	317.2 (1.7)	306.4 (0.1)	306.2 (0.5)	301.9 (0.5)
jackson-core	355.7	131.9 (0.1)	129.3 (0.0)	134.5 (0.2)	127.1 (0.2)
jackson-dataformat-xml	640.4	45.9 (0.1)	45.0 (0.0)	43.6 (0.1)	41.3 (0.1)

The numbers within brackets represent the time for computing the test suite optimisation (see also Fig. 7)

methods), since they are out of the scope of this work and it has been shown that they perform worse in terms of fault detection ability (Coviello et al. 2018a). Table 3 summarises the approaches included in this evaluation.

All optimisation tests have been run on a Windows 10 PC with Intel Core i5-8250U 1.60 GHz CPU and 8 GB RAM.

The effectiveness of our framework has been assessed by measuring the time necessary for executing the resulting test suite. We run the tests sequentially, then the total execution time is $c_{t_1} + c_{t_2} + \cdots + c_{t_n}$, where c_{t_i} is the running time of a single test. We also evaluated efficiency, in terms of time needed for computing the test suite reduction. We do not consider the time needed to run the test suite, to generate the execution traces, and to build the CFG, since we do not have control on them and we employed tools from third parties, which are not optimised for efficiency. Instead, we focus on the test suite reduction time, since it is more critical for scalability, and include the input size reduction techniques described in the method section.

Figure 6 compares test suites execution times resulting by Greedy, HGS, CN, the current approach REDUNET, and the unfiltered test suite (Do Nothing). ILP and ILP+CFG are not included since their performances are the same as REDUNET. As expected, all approaches reduce the running time of each test suite significantly. For the largest system, *commons-math*, the running time when using REDUNET is less than half of the running time needed for the whole test suite. The best gain is for the system *jackson-dataformat-xml*, for it the resulting running time was reduced more than 10 times. REDUNET performs always better than the other approaches. This is an expected result since REDUNET is specifically designed to minimise execution time and has found the optimally reduced test suite in each case.

To better show the improvement of REDUNET with respect to the other approaches, we report the precise test execution time in Table 4. We also report, within brackets, the optimisation time, which we discuss in the next paragraph. The resulting test suite given by REDUNET has a running time up to 15 s shorter than the one given by HGS (on *commons-math*), up to 5 s shorter than the one given by Greedy (on *commons-jxpath*), and up to 7 s shorter than the one given by CN (on *jackson-core*).

We evaluate the efficiency of our approach by comparing it to previously proposed approaches. We include some variants of our approach in order to assess the impact that

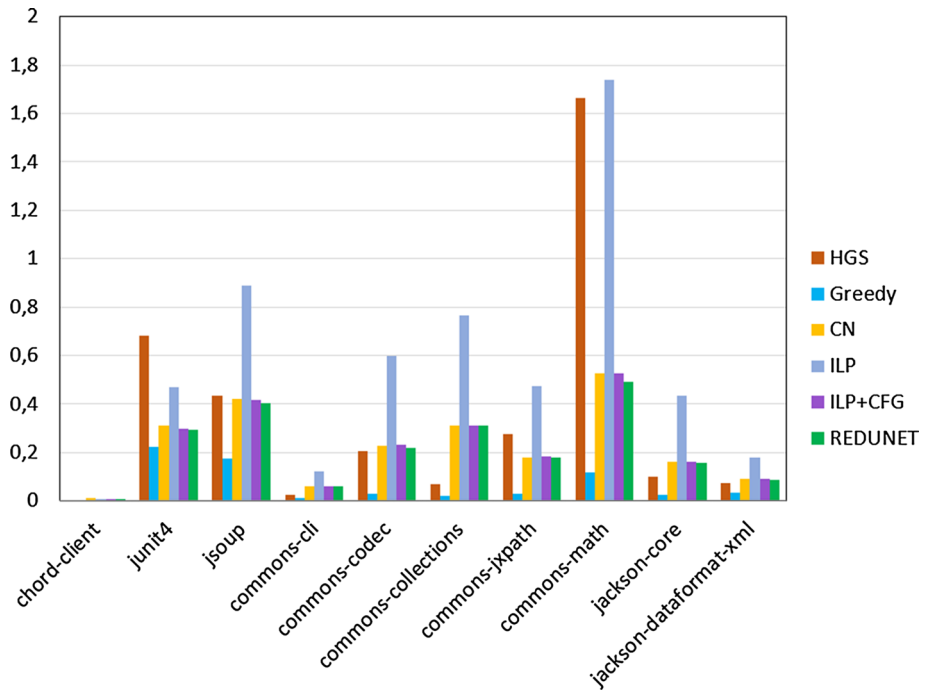

Fig. 7 Efficiency of test suite optimisation Running time in seconds for computing the test suite optimisation (*y*-axes). The labels in the *x*-axis represent the ten software projects under analysis, whose details are reported in Table 2. The series represents the approaches under comparison, whose details are given in Table 3

CFG analysis and the removal of back edges have on REDUNET. To reduce the error due to small fluctuations of the running times, we repeat each run 100 times and consider the average computation time.

Figure 7 reports the results in terms of running times. The most efficient method is Greedy. However, as discussed previously, Greedy is less effective than REDUNET and other ILP-based algorithms, therefore its speed comes at the cost of a more expensive resulting test suite. Note that the running time of the test suite is two order of magnitude longer than the running time of the test suite reduction and the test suite is executed more frequently than the reduction. Therefore, it is worth to spend some fractions of a second more to have a less expensive test suite. The most costly approach is ILP (except on *junit4*, where HGS performs slower). ILP has the advantage of finding the optimal solution, however it does not implement the CFG analysis discussed in the method section. CN, ILP+CFG and REDUNET are significantly quicker than ILP in all projects except *chord-client*. REDUNET is slightly quicker than CN and ILP+CFG in all projects except *chord-client* and *commons-collections*. Note that REDUNET (as well as ILP+CFG) is as effective as ILP, therefore the gain in efficiency comes at no cost. The higher efficiency of REDUNET depends on the ability to reduce the input for the ILP solver. The time necessary for finding an exact solution of an ILP problem grows exponentially with the input size, while the CFG analysis has linear complexity, therefore the advantage of our CFG-based input reduction grows with the size of the input. Interestingly, the only project for which REDUNET performs worse than ILP is *chord-client* (6 ms for REDUNET and 5 ms for ILP, values not shown in the graphics). Due to the very

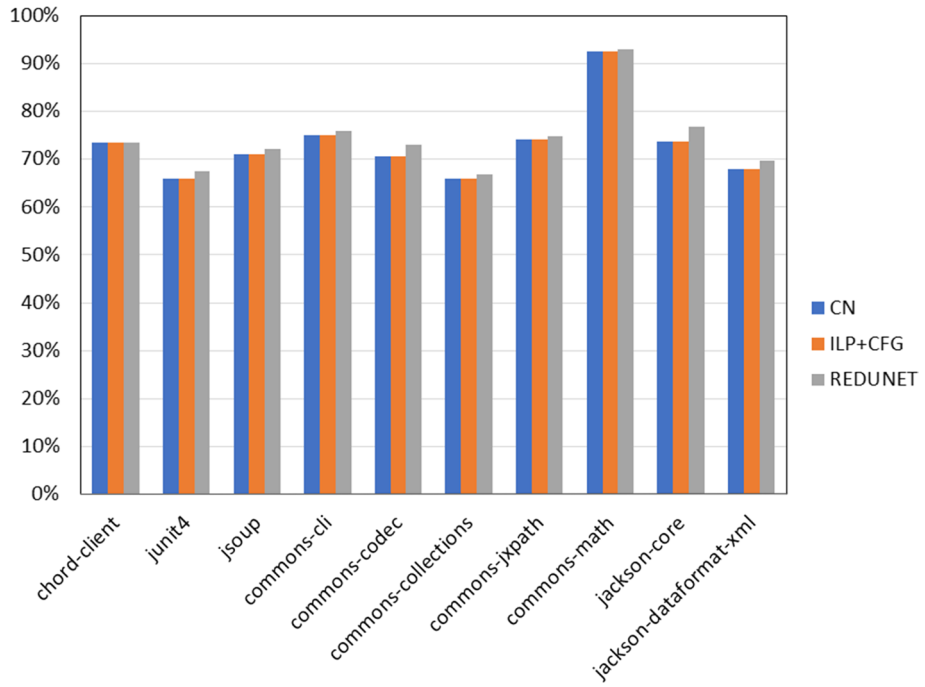

Fig. 8 Input size reduction. Percent of reduction of the input size for the ILP program (*y*-axes). The labels in the x-axis represent the ten software projects under analysis, whose details are reported in Table 2. The series represents the approaches under comparison, whose details are given in Table 3

small size of such a system, the running time of the ILP solver is negligible, hence the time spent by the CFG analyser to provide less data to the ILP solver cannot be counterbalanced when running the ILP solver. HGS shows mixed results in comparison to the other approaches, being less efficient than ILP in one case (on *junit4*) and always less efficient than Greedy. In 6 over 10 projects HGS is faster than REDUNET. However, as shown in Fig. 6, its effectiveness is not superior to Greedy, therefore similar considerations hold.

For small-size software systems the difference in efficiency among CN (or ILP+CFG) and REDUNET is expected to be small, whereas for larger ones the difference is expected to be bigger. This has been confirmed experimentally; indeed the largest analysed system, *commons-math*, is the one in which the results provided by REDUNET shows the largest percent improvement in computation time with respect to CN and ILP+CFG (its value is 6%, value not shown in the plot). On larger projects, the ability to reduce the input size for the ILP program becomes more critical. To assess this ability we compute the percent of reduction of the input size of CN, ILP+CFG and REDUNET with respect to ILP, which performs no input reduction. The *input size* is defined as the sum over all test cases of the number of instructions covered by each test case.

Figure 8 shows the results in terms of input size. The reduction is always significant, in the range between 66 and 93%. REDUNET outperforms CN and ILP+CFG in all projects except *chord-client*, where it shows the same performance. CN and ILP+CFG show the same performances since they use the same reduction algorithm and only differ in the objective function of the ILP optimisation.

Related works

The problems of generating and reducing test suites have been considered by past work both as distinct problems and jointly. We briefly review existing approaches and highlight the differences with our approach.

Automatic generation of test suites

Randoop (Pacheco and Ernst 2007) creates unit tests by chaining sequences of methods randomly selected from the software under test. Code coverage is not considered during test generation, thus several generated test cases could exercise the same code portion. With respect to our contribution it is complementary, since we take the randomly generated test cases and select such test cases that cover the code without much repetition, reducing the test suite size while maintaining the same code coverage.

Test suite reduction, selection and prioritisation

Existing approaches for test suite reduction focus on maintaining generic requirements, which might refer to code coverage or other characteristics of test cases. Although some of these approaches can be applied to maintain statement coverage, on large projects the size of data would make them unfeasible or at least inefficient.

The approach proposed in Tallam and Gupta (2006) considers the requirements exercised by a test, and minimises the number of test cases by excluding a test case when requirements covered are also covered by other test cases. They introduce a solution called delayed-greedy, consisting of some heuristics, such as checking whether a set of requirements is a superset or a subset of another, hence excluding the test cases having subsets of requirements; whether a requirement is covered by only one test case, then including such a test case; and whether the test case covers the maximum number of attributes, then including it.

Harrold et al. (1993) proposed the HGS heuristic to reduce test suites by maintaining a certain set of test requirements, where requirements can refer to code statements or other features of tests. It consists in grouping test requirements into an ordered family of sets, where the i-th set contains requirements that are covered by exactly i tests, and taking tests that satisfy requirements from the 1-st set, then from the 2-nd set and so on, until all requirements are satisfied.

Smith and Kapfhammer (2009) evaluated several heuristics, including HGS (Harrold et al. 1993), the traditional greedy algorithm for set cover and two of its variants, 2-optimal greedy and delayed greedy, for reducing test suites. The greedy algorithm takes test cases one by one considering at each step the test case that maximises the ratio between its number of previously unsatisfied requirements and its cost. 2-optimal greedy is similar but considers pairs of sets in place of single sets. Delayed greedy pre-processes the test suite before applying the greedy algorithm by removing test cases that are subsets of other test cases and including test cases that contain requirements that are not satisfied by other test cases.

Xu et al. (2012) considered the problem of reducing test suites as a variant of set cover, where costs are associated to sets (test cases) and the goal is to maintain the coverage of requirements by minimising the total cost. The authors apply a greedy algorithm to

solve it. In Hsu and Orso (2009) the authors mapped the set cover problem to a mutation testing problem. By using a greedy approach to find the minimum set of test cases that covers all the mutants killed by the original test suite, they were able to detect the same number of faults.

Chen et al. (2008) adopted an exact approach based on ILP to reduce the test suite by maintaining a set of requirements, and propose some heuristics to increase efficiency. Panda and Mohapatra (2017) proposed a an ILP-based approach for minimising the test suite without loosing coverage of modified sentences and meanwhile minimising a measure of cohesion of code parts covered by each test case. In Jatana et al. (2016) the authors proposed a general ILP-based framework to support test-suite minimisation over multiple criteria, such as code coverage, test execution time and setup cost.

FLOWER (Gotlieb and Marijan 2014) describes a novel Maximum-Flow-based approach for reducing the test suite while maintaining requirement coverage and shows that this method is more effective than greedy approaches, is more efficient than exact ILP-based approaches and admits multi-objective optimisation.

Although the described approaches can take into account statement coverage, the methods are not specifically optimised for it, therefore using them for code coverage guarantees might be unfeasible or ineffective on large programs. Focusing on code coverage, instead of coverage of requirements, has specific advantages and introduces some challenges. Most development practices have only coarse-grained requirements, hence many portions of code are implemented for a requirement. As a result, two (or more) test cases covering the same requirement could exercise different portions of code. Moreover, by automatically deriving the code lines covered by each test case, we need not rely on a manual mapping between test cases and requirements. Focusing on code coverage is harder due to the larger amount of data (code instructions) to handle. Although some heuristics could easily be applied to guarantee code coverage, in place of manually-curated requirements, the accuracy would be penalised to guarantee feasibility. In contrast, we leverage on the properties of the CFG to feasibly solve the problem with high accuracy. Since they are not directly focused on code coverage, none of the previously discussed approaches take advantage of the properties of the CFG for optimising computation.

In Do (2016), several approaches are presented for the reduction, selection, prioritisation of test cases. For test case reduction, in some approaches code coverage was assessed by means of a CFG, and other approaches used ILP for test case selection. We put forward both and we include optimisations (such as the early selection of statements thanks to the CFG) in our proposed solution. For recent approaches the preferred way to assess their benefits is by measuring the reduction rate of tests or their execution time and the coverage ensured by selected test cases. Accordingly, our optimisation method has maximised coverage, while minimising the execution time. Moreover, previous approaches are often scored against a suite of software systems that is sometimes considered controversial (perhaps not representative enough). We have permed experiments on ten recent and large software systems.

In Chen et al. (2010), the authors describe the technique of Adaptive Random Testing and its implications. It is argued that the more evenly spread the test cases the better, as they would allow running several parts of the system. More specifically, it is

recommended that random values for parameters should be chosen to be located away from previously executed test cases that have not revealed failures. Such a technique mainly aims at generating a selection of input values for test cases. It can be argued that for the same method to be tested, the selected values would run different paths. However, for test cases having each a sequence of method calls, it is not straightforward to provide an estimate on the absence of overlapping paths, hence on coverage.

In Cruciani et al. (2019), the authors reduce test suites by computing the similarity among test cases and take the most distant tests as these should be the ones that potentially show faults on several parts of the system. Their reduction approach need not executing tests on the system, hence it performs quickly. Still to determine the code coverage provided by the reduced test suite, test cases have to be executed on the system. Moreover, several reduced test suites could need execution to ensure that a target code coverage has been reached. The comparison of their adequate version with other approaches on a Java dataset has shown no difference in fault detection abilities, whereas the Greedy algorithm achieves the best test suite reduction. REDUNET clearly outperforms Greedy, and this is confirmed empirically by our experimental results.

A study has also compared test suite reduction and test case selection approaches in detecting change-related faults, i.e. faults caused only by some specific changes (e.g. occurred between two code revisions) (Shi et al. 2015). The authors observed that test suite reduction can lose change-related fault-detection capability (of up to 5.93% for killed mutants) with respect to the full test suite, while regression test selection has no loss. This because reduction approaches operate only on one software revision, so they are not aware of changes, and they can remove tests that are deemed redundant for the current revision, irrespective of recent changes (Shi et al. 2014). Test selection approaches are instead specifically designed to select all tests that may be affected by given changes, thus having a better detection of faults caused by them. On the other hand, test reduction approaches are still able to find faults unrelated to those changes.

In Luo et al. (2018) the authors compare eight approaches for Test Case Prioritisation (TCP): given a test suite, test case prioritisation consists in finding a set permutation where test cases are ordered according to a specific performance function (Rothermel et al. 2001), such as the number of synthetic software defects (*mutants*) they are able to detect (*kill*). The authors investigate whether the common practice of evaluating tests performance against mutants would be representative of the performance achieved on real faults, highlighting in which cases this may lead to an over/under estimation of the actual prioritisation effectiveness in terms of fault detection rate. The main difference with our approach is that the test case prioritisation problem differs from the test suite minimisation we solve: the former seeks to order test cases in such a way that early fault detection is maximised, the latter seeks to eliminate redundant test cases in order to reduce the number of tests to run (Yoo and Harman 2012). That is, while the objective of TCP approaches is to prioritise tests while leaving the test suite size unchanged, which is desirable for test cases properly crafted by a software developer, in our proposal we leverage random approaches to automatically generate a considerably higher number of tests, and then reduce the test suite size according to the coverage performance.

The ability of reduced test suites to properly discover faults have been studied in the past. Preliminary research on small C programs have produced mixed results, with e.g.

Wong et al. (1998) showing no significant reduction in fault detection capability, while Rothermel et al. (2002) reported a significant reduction of the ability to detect faults. Their results, however, have been obtained on small programs in C language and might not extend to modern software projects. More recent assessments have reported no significant reduction in fault detection capability of adequate methods (i.e. which maintain code coverage) (Coviello et al. 2018a).

Joint generation and optimisation of test suites

EvoSuite (Fraser and Arcuri 2011, 2012) uses symbolic execution to automatically generate test cases with assertions. It aims at generating a test suite minimising the number of test cases while maximising code coverage. The former characteristic is the one giving EvoSuite more similarities to our approach. However, each sequence of method calls generated by EvoSuite, which represents a population to be selected and bred if considered the most fit, is executed in order to assess code coverage. This is time consuming; risky when having to interact with files, databases, and networks; and since their approach is evolutionary, the number of execution is not bounded nor small (Fraser and Arcuri 2011). Our approach requires only one execution of the test cases, then the selection approach does not require any run.

The authors in Murphy et al. (2013) first generate the test cases to cover all different subpaths in a program, then, since each test case can cover more than one subpath, they reduce the number of test cases by identifying which subpaths are covered by each test case. The test cases selected are found by a greedy solution, which finds in some cases the optimal solution. Compared to our approach, there are similarities on the coverage, though we consider single lines of code, instead of subpaths. As far as the solution is concerned, instead of using a simple greedy approach, we have proposed a solution able to find a guaranteed optimum on all tested software projects and an accurate approximated solution within a measurable error bound in all cases. Moreover our solution leverages on the knowledge of the control flow to reduce the number of code lines to consider, hence searching on a smaller solution space.

To our knowledge no approach uses the CFG properties to optimise the test suite minimisation process.

Conclusions

Optimising test suites can be a valuable aid for a software developer in establishing agile practices, which require frequent and extensive testing of software projects under development. The proposed approach manages to reduce the size of a test suite used for regression testing considerably, and its execution time is also decreased without losing code coverage.

We have evaluated our test reduction approach on ten open source systems having size between several hundreds to tens of thousands lines of code and their associated test suites generated by Randoop. Our optimisation tool finds the minimum-cost test suite with 100% accuracy in fractions of a second on all software systems under evaluation, and performs an up to ten-fold reduction of the test suite execution time.

Our complete framework, which automates test generation, code coverage computation, and test suite reduction, takes as input a software system and generates a minimal

amount of test cases with the maximum reasonably possible code coverage. The ability of generating effective test suites that execute quickly is beneficial for regression testing since it enables frequent execution of test cases and ensures timely feedback, even when only limited hardware resources are available.

Abbreviations
CFG: Control flow graph; HGS: Harrold Gupta Soffa; ILP: Integer linear programming; TCP: Test case prioritisation.

Acknowledgements
The authors acknowledge the support provided by projects CLARA and TEAMS.

Authors' contributions
ET introduced the problem of reducing test suites, devised the overall solution and designed the abstract framework. MM proposed the problem formulation, designed and described the proposed algorithm, implemented the algorithms for the evaluation, and discussed the results. AF proposed the idea of removing back edges, collected the software projects for the experimental analysis, developed the framework for generating traces and CFGs, and described such a framework. All authors participated in discussions and proposed ideas and suggestions. All authors participated in writing, revising and polishing the manuscript. All authors read and approved the final manuscript.

Funding
This work is supported by project CLARA (CLoud plAtform and smart underground imaging for natural Risk Assessment), SCN 00451, funded by Italian MIUR within the Smart Cities and Communities and Social Innovation initiative, and by project TEAMS–TEchniques to support the Analysis of big data in Medicine, energy and Structures–Piano di incentivi per la ricerca di Ateneo 2020/2022.

Availability of data and materials
The dataset generated and analysed during the current study is available in the ***REDUNET*** GitHub repository (https://github.com/afornaia/redunet).

Competing interests
The authors declare that they have no competing interests.

Author details
[1] ISTC CNR, Via Gaifami, 18, 95126 Catania, Italy. [2] Dipartimento di Matematica e Informatica, University of Catania, Viale A. Doria, 6, 95125 Catania, Italy.

Received: 20 April 2020 Accepted: 9 October 2020
Published online: 02 November 2020

References
Arcuri A (2011) A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. IEEE Trans Softw Eng 38(3):497–519
Chen Z, Zhang X, Xu B (2008) A degraded ILP approach for test suite reduction. In: Proceedings of SEKE
Chen TY, Kuo F-C, Merkel RG, Tse T (2010) Adaptive random testing: the art of test case diversity. J Syst Softw 83(1):60–66
Coviello C, Romano S, Scanniello G (2018a) An empirical study of inadequate and adequate test suite reduction approaches. In: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement, pp 1–10
Coviello C, Romano S, Scanniello G, Marchetto A, Antoniol G, Corazza A (2018b) Clustering support for inadequate test suite reduction. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 95–105
Cruciani E, Miranda B, Verdecchia R, Bertolino A (2019) Scalable approaches for test suite reduction. In: Proceedings of IEEE/ACM international conference on software engineering (ICSE), pp 419–429
DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. Computer 11(4):34–41
Do H (2016) Recent advances in regression testing techniques. In: Memon AM (ed) Advances in computers, vol 103. Elsevier Academic Press, Amsterdam, pp 53–77
Feige U (1998) A threshold of ln n for approximating set cover. J ACM (JACM) 45(4):634–652
Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of ACM SIGSOFT symposium and European conference on foundations of software engineering
Fraser G, Arcuri A (2012) Whole test suite generation. IEEE Trans Softw Eng 39(2):276–291
Godboley S, Panda S, Dutta A, Mohapatra DP (2017) An automated analysis of the branch coverage and energy consumption using concolic testing. Arab J Sci Eng 42(2):619–637
Gotlieb A, Marijan D (2014) Flower: optimal test suite reduction as a network maximum flow. In: Proceedings of ACM international symposium on software testing and analysis
Harrold MJ, Gupta R, Soffa ML (1993) A methodology for controlling the size of a test suite. ACM Trans Softw Eng Methodol (TOSEM) 2(3):270–285

Hsu H-Y, Orso A (2009) Mints: a general framework and tool for supporting test-suite minimization. In: Proceedings of IEEE ICSE
Jatana N, Suri B, Kumar P, Wadhwa B (2016) Test suite reduction by mutation testing mapped to set cover problem. In: Proceedings of ACM international conference on information and communication technology for competitive strategies
Kazmi R, Jawawi DN, Mohamad R, Ghani I (2017) Effective regression test case selection: a systematic literature review. ACM Comput Surv (CSUR) 50(2):1–32
Kuhn DR, Bryce R, Duan F, Ghandehari LS, Lei Y, Kacker RN (2015) Combinatorial testing: theory and practice. Adv Comput 99:1–66
Luo Q, Moran K, Poshyvanyk D, Di Penta M (2018) Assessing test case prioritization on real faults and mutants. In: 2018 IEEE international conference on software maintenance and evolution (ICSME), pp 240–251. IEEE
Mongiovì M, Fornaia A, Tramontana E (2019) A network-based approach for reducing test suites while maintaining code coverage. In: International conference on complex networks and their applications. Springer
Murphy C, Zoomkawalla Z, Narita K (2013) Automatic test case generation and test suite reduction for closed-loop controller software. Technical report, University of Pennsylvania, Department of Computer and Information Science
Myers GJ, Sandler C, Badgett T (2011) The art of software testing. Wiley, Hoboken
Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for java. In: Proceedings of OOPSLA companion
Panda S, Mohapatra DP (2017) Regression test suite minimization using integer linear programming model. Softw Pract Exp 47(11):1539–1560
Panda S, Munjal D, Mohapatra DP (2016) A slice-based change impact analysis for regression test case prioritization of object-oriented programs. Adv Softw Eng. https://doi.org/10.1155/2016/7132404
Rothermel G, Harrold MJ, Von Ronne J, Hong C (2002) Empirical studies of test-suite reduction. Softw Test Verif Reliab 12(4):219–249
Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27(10):929–948
Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In: Proceedings of IEEE/ACM ASE
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 foundations of software engineering, pp 246–256
Shi A, Yung T, Gyori A, Marinov D (2015) Comparing and combining test-suite reduction and regression test selection. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, pp 237–247
Shu G, Sun B, Henderson TA, Podgurski A (2013) Javapdg: a new platform for program dependence analysis. In: Proceedings of IEEE international conference on software testing, verification and validation
Smith AM, Kapfhammer GM (2009) An empirical study of incorporating cost into test suite reduction and prioritization. In: Proceedings of the 2009 ACM symposium on applied computing, pp 461–467
Tallam S, Gupta N (2006) A concept analysis inspired greedy algorithm for test suite minimization. ACM SIGSOFT Softw Eng Notes 31(1):35–42
Vazirani VV (2013) Approximation algorithms. Springer, Berlin
Wong WE, Horgan JR, London S, Mathur AP (1998) Effect of test set minimization on fault detection effectiveness. Softw Pract Exp 28(4):347–369
Xu S, Miao H, Gao H (2012) Test suite reduction using weighted set covering techniques. In: Proceedings of IEEE international conference on software engineering, artificial intelligence, networking and parallel/distributed computing
Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Softw Test Verif Reliab 22(2):67–120

Publisher's Note