https://pubsonline.informs.org/journal/ijoc

INFORMS JOURNAL ON COMPUTING
Vol. 36, No. 1, January–February 2024, pp. 21–38
ISSN 1091-9856 (print), ISSN 1526-5528 (online)

A Set-Covering Approach to Customized Coverage Instrumentation

Carla Michini,[a] Peter Ohmann,[b] Ben Liblit,[c,d] Jeff Linderoth[a,*]

[a] Department of Industrial and Systems Engineering, University of Wisconsin–Madison, Madison, Wisconsin 53706; [b] College of Saint Benedict/Saint John's University, Saint Joseph, Minnesota 56374; [c] Computer Sciences Department, University of Wisconsin–Madison, Madison, Wisconsin 53706; [d] Amazon Web Services, Amazon.com, Inc., Arlington, Virginia 22202

*Corresponding author

Contact: michini@wisc.edu, https://orcid.org/0000-0002-4717-816X (CM); pohmann001@csbsju.edu, https://orcid.org/0000-0002-7670-7374 (PO); liblit@acm.org, https://orcid.org/0000-0002-2245-2839 (BL); linderoth@wisc.edu, https://orcid.org/0000-0003-4442-3059 (JL)

Received: December 20, 2021
Revised: August 25, 2022; February 21, 2023; July 13, 2023; July 18, 2023
Accepted: July 21, 2023
Published Online in Articles in Advance: October 9, 2023

https://doi.org/10.1287/ijoc.2021.0349

Copyright: © 2023 INFORMS

Abstract. Program coverage customization selectively adds instrumentation to a compiled computer program so that a limited amount of directly observed data can be used to infer other program coverage information after a run. A good instrumentation plan can reduce run-time overheads while still giving software developers the information they need. Unfortunately, optimal coverage planning is NP-hard, limiting either the quality of heuristic plans or the sizes of programs that can be instrumented optimally. We exploit the monotonicity property of feasible instrumentations to formulate this problem as an intraprocedural set covering problem. Our formulation has an exponential number of constraints, and we design a polynomial-time separation algorithm to incrementally add the necessary subset of these inequalities. Our approach reduces expected run-time probing costs compared with existing methods, offers a guarantee of the optimality of the instrumentation, and has compilation-time overhead suitable for wide practical use.

History: Accepted by Pascal Van Hentenryck, Area Editor for Computational Modeling: Methods & Analysis.

Funding: This work was supported by the National Science Foundation [Grants CCF-1318489, CCF-1420866, and CCF-1423237] and the Air Force Research Laboratory [Grant FA8750-14-2-0270].

Supplemental Material: The software that supports the findings of this study is available within the paper and its Supplemental Information (https://pubsonline.informs.org/doi/suppl/10.1287/ijoc.2021.0349) as well as from the IJOC GitHub software repository (https://github.com/INFORMSJoC/2021.0349). The complete IJOC Software and Data Repository is available at https://informsjoc.github.io/.

Keywords: debugging • program coverage • set covering

1. Introduction

For a run of a computer program, *program coverage data* indicate which parts of the program were executed on that particular run. Program coverage data are vital for many areas of software development. Classically, developers use program coverage data during program testing, but it has recently also found use in other areas, such as postmortem program analysis (Nishimatsu et al. 1999, Ohmann and Liblit 2017, Ohmann et al. 2017) and fault localization (Santelices et al. 2009, Liblit 2014). Developers and tools gather coverage data for different program points. For example, statement coverage measures which statements ran, whereas function coverage provides the set of functions that executed. Obtaining coverage data for a program requires placing and monitoring probes at locations within the code. These activities incur run-time overhead, so authors have studied efficient mechanisms for placing probes within the code.

In some contexts, developers are interested in *counts* of executed statements. Prior work has optimized probe placement in this scenario, including classic work by Knuth and Stevenson (Knuth and Stevenson 1973, Knuth 1968). Ball and Larus (1994) generalized various problems of optimal *counting* probe placement for statements and/or edges, and more recent work applies similar ideas to other contexts like call tracing (Reiss and Renieris 2001, Dallmeier et al. 2005, Wu et al. 2016).

Coverage data are *binarized* if the data are reported as "covered" or "uncovered" for each point rather than a count of executions for that point. Binarized probes are significantly less expensive in terms of runtime overhead. In the case of binarized coverage, Agrawal (1994, 1999) uses "superblocks" formed from sets of related basic blocks via dominance relations to optimize binarized coverage probe placements. Others (Tikir and Hollingsworth 2002, Misurda et al. 2005, Chilakamarri and Elbaum 2006, Misurda et al. 2011, Kasikci et al. 2014, Pankumhang and

Rutherford 2015) make gathering program coverage more efficient by dynamically removing a probe after it is already observed "covered" in an execution.

Ohmann et al. (2016b) first introduced the *customized coverage probing problem*, which calls for an optimal placement of probes (i.e., observation points) in a program's code to gather binarized statement or branch coverage for a given set of important program points. For customized coverage probing, the focus is on instrumentation for deployed applications, where performance is important, and the software is partially tested. In this scenario, full coverage data are often unnecessary or too expensive to gather. In practice, developers may want coverage data only for untested code (Pavlopoulou and Young 1999, Orso et al. 2002) or for specific program points such as call sites (Nishimatsu et al. 1999, Ohmann and Liblit 2013). This problem statement, originally given by Ohmann et al. (2016b), allows the user to specify both this *desired* set of program points and *instrumentable* points that can be probed. Noninstrumentable points may include, for example, performance- or security-sensitive code regions. An optimal solution to the customized coverage probing problem minimizes the run-time cost of probes while guaranteeing run-time coverage data for the desired set of program points without placing probes in the noninstrumentable regions.

After proving that the customized coverage probing problem is NP-hard, Ohmann et al. (2016b) proposed multiple solutions that vary in their efficiency and optimality guarantee. However, these prior approaches were unsatisfactory in that they were either heuristic with no guarantees of optimality, or optimal in theory but prohibitively inefficient in practice. In this paper, we present a new integer programming approach for computing optimal intraprocedural solutions to the customized coverage probing problem. The key ingredients of the approach are a new set-covering based model combined with branch-and-cut (Mitchell 2002). Compared with existing exact algorithms, our approach finds optimal probe placements orders of magnitude faster. Even compared with previous heuristic methods that do not provide an optimality guarantee, our exact algorithm typically finds probe placements roughly five times faster. With the new ability to determine optimal instrumentation for many programs, we are for the first time able to mathematically demonstrate that previously obtained heuristic optimization plans were of high quality, while in some cases improving on previously best-known coverage plans.

Our primary contributions in this work are the following:

- We formulate the customized coverage probing problem as a set covering problem.
- The integer programming formulation may have an exponential number of constraints, but we develop an efficient separation algorithm that, given a candidate solution, either affirms that the solution satisfies all the original constraints or finds a counterexample. The counterexample is a constraint of the optimization problem that is violated by the candidate solution, which is then added to the model, and the process repeats.
- We formally prove the correctness of the separation algorithm. The proof is based on a characterization of feasible instrumentation plans in terms of forbidden graph structures.
- We show that our separation algorithm's complexity is polynomial in the size of the input program.
- We evaluate our new approach against the prior approaches of Ohmann et al. (2016b). In addition to a guarantee of finding the optimal solution, we find that the optimal solution does indeed reduce expected run-time probing costs in many cases. Furthermore, the compilation-time overhead of our technique makes it suitable for wider practical use than existing methods.

This paper is organized as follows. Section 2 defines the customized coverage probing problem from Ohmann et al. (2016b), provides several examples that emphasize the combinatorial nature of the problem, and briefly recounts prior state-of-the-art approaches implemented by Ohmann et al. (2016b). Section 3 describes our new approach that relies on an integer programming formulation of the problem. We evaluate our techniques in Section 4, especially with respect to the prior results of Ohmann et al. (2016b), and Section 5 concludes.

2. Problem and Background

Notation used for this section and the remainder of the paper is summarized in Table 1 for convenience.

The goal of the customized coverage probing problem is to find an optimal instrumentation plan to gather customized statement coverage of a computer program. A set of desired program points is specified, for which coverage data must be available on any run of the computer program. We are also given a set of locations where we are allowed to insert probes, each having a certain instrumentation cost. An optimal solution is a minimum cost set of probes, placed at a subset of potential locations, such that coverage data for all desired program points can be determined based only on the coverage data at the probe set of locations.

2.1. Input

A *control flow graph* (CFG) is a graph representation of all paths that may be traversed through a computer program during its execution. The nodes of a CFG are *basic blocks*, pieces of computer code that do not contain jumps or

Table 1. Notation Reference

Notation	Meaning
$G = (V, E)$	Directed graph with node set V and arc set E
s	Entry node
T	Set of exit nodes
D	Set of desired nodes
I	Set of instrumentable nodes
c_u	Cost of instrumenting $u \in I$
(u, v)	Arc directed from node u to node v
$\langle u_0, u_1, \dots, u_k \rangle$	A path traversing nodes $u_0, u_1, \dots, u_k$
$V(p)$	Set of nodes (without repetitions) traversed along path p
$A \Delta B = (A \setminus B) \cup (B \setminus A)$	Symmetric difference between sets A and B
$q^1 \circ q^2$	The uw-path obtained by composing the uv-path q^1 and the vw-path q^2
$\mathcal{P}_d^{uv}$	Set of all uv-paths that do not traverse d
V_d^{uv}	Set of all the nodes that belong to some path in $\mathcal{P}_d^{uv}$
$V_d^{\beta T} = \bigcup_{t \in T} V_d^{\beta t}$	Set of all the nodes that belong to some path in $\mathcal{P}_d^{\beta t}$ for all $t \in T$
$G \setminus d$	Subgraph of G obtained by deleting node d from G, Together with all the arcs incident to it
$V_d^+(u)$	Set of all the nodes that can be reached from u in $G \setminus d$

jump targets. Edges of the CFG are used to represent the jumps possible in the control flow logic of the program. The CFG is a key ingredient of many compiler optimization and statis program analysis tools.

An instance of the *customized coverage probing problem* is defined with the following input:

- The CFG of a single function, which is a directed graph $G = (V, E)$. We assume that G has no parallel arcs.
- An *entry* node, $s \in V$, with in-degree 0.
- A set $T \subseteq V$ of *exit* or *termination* nodes of the CFG. The set T may include program locations for premature termination, such as via an assertion failure or program exception.
- A subset $I \subseteq V$ of nodes that can be *probed*, (*instrumented*).
- For each $u \in I$, $c_u > 0$ is the cost of instrumenting u.
- The subset $D \subseteq V$ of *desired* nodes.

A *path* is a sequence of nodes $\langle u_0, u_1, \dots, u_k \rangle$ such that $e_i = (u_{i-1}, u_i) \in E \ \ \forall i \in \{1, 2, \dots, k\}$. A path may be composed of a single node. A path starting at $u \in V$ and ending at $v \in V$ is called a uv-path. A *cycle* is a closed path, that is, a path $\langle u_0, u_1, \dots, u_k \rangle$ such that $u_0 = u_k$. A path is *simple* if it does not contain repeated nodes, so a simple path contains no cycles. In general, we do not assume paths to be simple. For a path, p, $V(p)$ denotes the set of nodes traversed along p.

Definition 1. Let $t \in T$. Two st-paths p^1 and p^2 are *D-equivalent* if they traverse the same set of desired nodes; that is, $V(p^1) \cap D = V(p^2) \cap D$.

Two D-equivalent st-paths can traverse the nodes in D in different orders or a different number of times.

2.2. Problem Definition

In our setting, two executions of a program are represented as two st paths in the associated CFG, where $t \in T$. We want to instrument a subset C of nodes in I such that, whenever two paths traverse the same nodes in C, they also traverse the same nodes in D.

Definition 2. A *coverage set of D* is a set of nodes $C \subseteq I$ such that, for all $t \in T$ and for all pairs of st paths p^1 and p^2, we have that if $V(p^1) \cap C = V(p^2) \cap C$, then $V(p^1) \cap D = V(p^2) \cap D$, that is, p^1 and p^2 are D-equivalent.

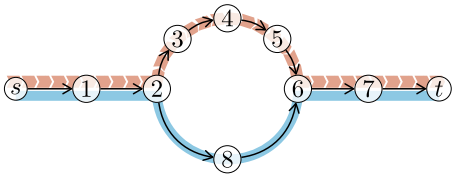
Thus, if $S \subseteq I$ is *not* a coverage set, then there exists $t \in T$ and two st paths, p^1 and p^2, such that $V(p^1) \cap S = V(p^2) \cap S$, but p^1 and p^2 are not D-equivalent.

The *customized coverage probing problem* asks to determine a coverage set of minimum cost. That is, an optimal solution C is a subset of I that is a coverage set of D and where $\sum_{u \in C} c_u$ is minimized.

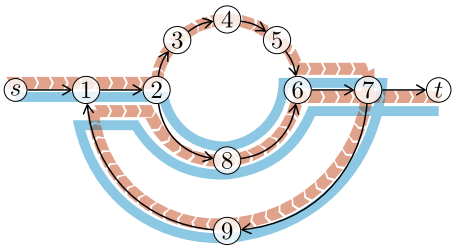
2.3. Examples

We introduce three (running) examples to elucidate the concept of a coverage set. Consider the CFGs in Figure 1. Suppose that, in each graph, the only desired node is 4 (i.e., $D = \{4\}$) and that all nodes can be instrumented (i.e., $I = V$). In each graph, the path p^1 traverses 4, whereas the path p^2 does not. Thus, in each case, p^1 and p^2 are not D-equivalent. The paths p^1 and p^2 need not be simple.

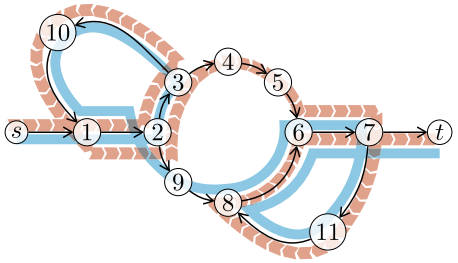
Figure 1. (Color online) CFGs for Examples 1, 2, and 3



(a) CFG for Example 1. $S = \{s, 1, 2, 6, 7, t\}$ is *not* a coverage set, whereas {3}, {4}, {5} and {8} are coverage sets.



(b) CFG for Example 2. $S = \{s, 1, 2, 6, 7, 8, 9, t\}$ is *not* a coverage set, whereas {3}, {4}, and {5} are coverage sets.



(c) CFG for Example 3 $S = \{s, 1, 2, 3, 6, 7, 8, 10, 11, t\}$ is *not* a coverage set, whereas {4}, {5}, and {9} are coverage sets.

Note. Unless otherwise stated, we assume $D = \{4\}$ and $I = V$.

Example 1. In the CFG in Figure 1(a), define the paths p^1 and p^2 as

$$p^1 = \langle s, 1, 2, 3, 4, 5, 6, 7, t\rangle$$
$$p^2 = \langle s, 1, 2, 8, 6, 7, t\rangle,$$

and the set

$$S = \{s, 1, 2, 6, 7, t\}.$$

It is easy to check that S is not a coverage set. In fact, p^1 and p^2 traverse the same nodes of S, but they are not D-equivalent. No subset of S is a coverage set.

Conversely, {3}, {4}, {5} and {8} are coverage sets because any two paths that differ on whether they traverse {4} must also differ on whether they traverse {3}, {4}, {5}, and {8}. Moreover, any superset of {3}, {4}, {5}, or {8} is a coverage set.

Example 2. Consider the CFG in Figure 1(b), and note that it differs from the one in Figure 1(a) only by the presence of node 9 and the arcs $\{(7, 9), (9, 1)\}$. Define the nonsimple paths p^1 and p^2 as

$$p^1 = \langle s, 1, 2, 3, 4, 5, 6, 7, 9, 1, 2, 8, 6, 7, t\rangle$$
$$p^2 = \langle s, 1, 2, 8, 6, 7, 9, 1, 2, 8, 6, 7, t\rangle.$$

In this case, {8} is *not* a coverage set because both p^1 and p^2 traverse {8}. In fact, it can be checked that no subset of

$$S = \{s, 1, 2, 6, 7, 8, 9, t\}$$

is a coverage set.

Conversely, {3}, {4}, and {5} are coverage sets because any two paths that differ on whether they traverse {4} must also differ on whether they traverse {3}, {4} and {5}. Moreover, any superset of {3}, {4}, or {5} is also a coverage set.

Example 3. Consider the CFG in Figure 1(c), and note that it differs from the one in Figure 1(a) only by the presence of nodes {9, 10, 11} and the arcs $\{(7, 11), (11, 8), (3, 10), (10, 1)\}$. Define the paths p^1 and p^2 as

$$p^1 = \langle s, 1, 2, 3, 10, 1, 2, 4, 5, 6, 7, 11, 8, 6, 7, t\rangle$$
$$p^2 = \langle s, 1, 2, 3, 10, 1, 2, 9, 8, 6, 7, 11, 8, 6, 7, t\rangle.$$

In this case, {3} and {8} are *not* coverage sets because both p^1 and p^2 traverse {3,8}. In fact, no subset of

$$S = \{s, 1, 2, 3, 6, 7, 8, 10, 11, t\}$$

is a coverage set.

Conversely, {4}, {5}, and {9} are coverage sets because any two paths that differ on whether they traverse {4} must also differ on whether they traverse {4}, {5}, and {9}, and any superset of {4}, {5}, or {9} is a coverage set.

2.4. Previous Solution Approaches

Ohmann et al. (2016b) establish that the customized coverage probing problem is NP-hard and offer three approaches for its solution. The first approach is a generalization of the approaches described by Agrawal (1999) and Tikir and Hollingsworth (2005) for placing probes to ensure complete program coverage. The approach aggregates program basic blocks into "superblocks" and uses dominator relationships between these superblocks to incrementally place probes so that at completion all nodes in the desired set D are covered. This "bottom-up" approach offers no guarantee about the optimality of the coverage set created, and Ohmann et al. (2016b) show in practice that the coverage sets found by this method may have significantly higher instrumentation costs than coverage sets found by other algorithms. We do not consider or compare against the "bottom-up" approach in this work.

A second "top-down" approach is somewhat more sophisticated, and, empirically, it finds coverage sets of higher quality. Although unstated, the approach relies crucially on the following monotonicity property of coverage sets, which is evident in the examples in Section 2.3.

Property 1 (Monotonicity). *If C is a coverage set and $C \subseteq C' \subseteq I$, then C' is also a coverage set.*

Thus, we have the notion of a *minimal* coverage set.

Definition 3. A coverage set C' is *minimal* if there is no smaller coverage set C with $C \subset C'$. Equivalently, for each $v \in C'$, $C' \setminus \{v\}$ is not a coverage set.

Example 2 (Continued). For the CFG in Figure 1(b), and inputs $T = \{t\}$, $I = V$ and $D = \{4\}$, the minimal coverage sets are {3}, {4}, and {5}. Any strict superset of these sets is a coverage set but not a *minimal* coverage set.

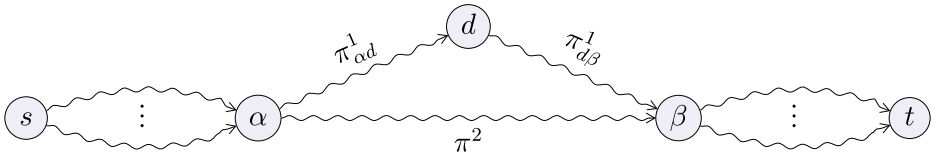
The second approach in Ohmann et al. (2016b) uses the monotonicity property to find a minimal coverage set. The top-down approach begins with the coverage set $S = I$, a coverage set by assumption, and iteratively attempts to remove each node, $s \in S$. If $S \setminus \{s\}$ is still a coverage set of D, a smaller coverage set has been found, and the process is repeated. If $S \setminus \{s\}$ is not a coverage set of D, the method attempts to remove a different node from S. Once the method attempts and fails to remove each node from the current coverage set S, then by Definition 3, S is a minimal coverage set of D.

At each major iteration of this method, the process must check whether $S \setminus \{s\}$ is a coverage set of D for each $s \in S$. To check if a particular set is, in fact, a coverage set of D, Ohmann et al. (2016b) propose an approach based on a characterization of coverage sets in terms of *ambiguous triangles*. The high-level idea is pictured in Figure 2.

In brief, the goal is to find a pair of st paths in G, with $t \in T$, that traverse exactly the same instrumented nodes but such that only one of the paths traverses a desired node $d \in D$, whereas the other does not. If there exists an ambiguous triangle with respect to the current set of instrumented nodes, then the current instrumentation is *not* a coverage set. It is possible to find an ambiguous triangle, or conclude that no such triangle exists, in polynomial time in the size of the graph G. We discuss our own use of ambiguous triangles in Section 3.2.

The notion of a *minimal* coverage set in Definition 3 is with respect to set inclusion. Conversely, the customized coverage probing problem asks to determine a coverage set of *minimum* cost. Although a minimum-cost coverage set must be minimal, the converse is not necessarily true. For example, in the CFG from Figure 1(c), the coverage set {4} is minimal. However, if node costs are based on the expected execution frequency of each basic block, the coverage set {9} may have lower cost, depending on the weighting of the branch at node 2. Thus, although the

Figure 2. (Color online) Ambiguous Triangle



second approach will find a coverage set that is *locally optimal*, it cannot guarantee finding a *globally optimal* coverage set of minimum cost.

The final approach proposed by Ohmann et al. (2016b) is a mixed integer linear programming (MILP) formulation that can guarantee finding a coverage set of minimum cost. As an exact method that relies on integer linear programming, their approach is related to ours. However, the formulation by Ohmann et al. (2016b) uses a network-flow-based characterization of the paths in the definition of an ambiguous triangle and effectively considers all acyclic paths through G. Instead, as described in Section 3, our formulation crucially relies on the monotonicity property stated previously. We exploit this property to model the customized coverage probing problem as a set covering problem, a well-studied optimization problem with a rich combinatorial structure. We leave out a complete description of this network flow-based integer programming formulation here, and interested readers may consult the technical report Ohmann et al. (2016a) to see the mathematical details. In Section 4, we compare these two exact methods for customized coverage instrumentation.

3. Approach

In this section, we present a new integer programming approach to calculate an exact solution to the customized coverage probing problem. Given two sets A and B, we denote by $A \Delta B$ their *symmetric difference* that is defined as $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

3.1. Set-Covering Formulation

First, we provide an alternative definition of a coverage set that will be useful to derive the constraints of our formulation.

Proposition 1. *The set $C \subseteq I$ is a coverage set of D if and only if for all $t \in T$ and for any two st paths p^1 and p^2, we have that if $(V(p^1)\Delta V(p^2)) \cap D \neq \emptyset$, then $(V(p^1)\Delta V(p^2)) \cap C \neq \emptyset$.*

Proof. Follows directly from the contrapositive of Definition 2: $C \subseteq I$ is a coverage set if and only if for all $t \in T$ and for any two *st* paths p^1 and p^2, we have that if $V(p^1) \cap D \neq V(p^2) \cap D$, then $V(p^1) \cap C \neq V(p^2) \cap C$. The definition of symmetric difference implies that for three sets A, B, and C, we have that $A \cap C = B \cap C$ if and only if $(A \Delta B) \cap C = \emptyset$; thus, the statement follows. □

Proposition 1 implies that if two *st* paths are not D-equivalent, then in the symmetric difference of the paths' vertex sets, we must probe at least one vertex. We can use this insight to formulate the customized coverage probing problem as the following binary integer linear program, denoted by (*SC*).

$$\min \quad \sum_{u \in I} c_u x_u \tag{SC.1}$$

$$\text{s.t.} \quad \sum_{u \in (V(p^1)\Delta V(p^2))} x_u \geq 1 \qquad t \in T, (p^1, p^2) \text{ not } D\text{-equivalent } st\text{-paths}, \tag{SC.2}$$

$$x_u \in \{0,1\} \qquad u \in I. \tag{SC.3}$$

We have one binary decision variable for each node in $u \in I$, and the binary vectors that satisfy all the constraints (SC.2) correspond precisely to the characteristic vectors of the coverage sets of D. Moreover, the value of the objective function (SC.1) in a feasible solution represents the instrumentation cost of the corresponding coverage set, which we wish to minimize. Note that (*SC*) is *infeasible* if and only if I is not a coverage set.

Finally, we remark that the number of constraints (SC.2) can be exponential in $|V|$ and $|E|$. Because it is computationally prohibitive to include all such constraints in (*SC*), we will describe in Section 3.2 a *separation algorithm* for this set of constraints, and the constraints will be added in an iterative manner (Mitchell 2002).

The separation algorithm receives in input a vector $\bar{x} \in [0,1]^I$, and it either states that $\bar{x}$ satisfies all the constraints (SC.2), or it returns a constraint of type (SC.2) that is violated by $\bar{x}$. In particular, if $\bar{x}$ is binary and is the characteristic vector of $S \subseteq I$, then the separation algorithm either states that S is a coverage set, or it returns a constraint of type (SC.2) that is violated by S. Our algorithms initially generates a (limited) pool of constraints of type (SC.2), and it solves a relaxation of (*SC*) including only this subset of constraints. Then, the algorithm iteratively adds more constraint of type (SC.2) to the relaxation by calling the separation algorithm, until all constraints (SC.2) are satisfied by the optimal solution of the relaxation. Computationally, the potential advantage of this approach lies in the fact that, at each iteration, the relaxation of (*SC*) has only few constraints, and thus it can be solved faster. Moreover, we will show that the separation algorithm runs in polynomial time and that it can be implemented by a sequence of shortest path computations.

The optimization problem (*SC*) falls in the class of *set covering problems*, that is, problems of the form

$$\min\{c'\boldsymbol{x} : A\boldsymbol{x} \geq \mathbf{1}, \boldsymbol{x} \in \{0,1\}^n\},$$

where A is a $m \times n$ matrix having entries equal either to zero or one. Some of the constraints (SC.2) might be redundant to define our set covering formulation. In particular, we say that a constraint of the form (SC.2) associated to paths (p^1, p^2) is *dominated* by another constraint of the form (SC.2) associated to paths $(\overline{p}^1, \overline{p}^2)$ if $V(p^1)\Delta V(p^2) \supseteq V(\overline{p}^1)\Delta V(\overline{p}^2)$.

Example 2 (Continued). Consider the CFG in Figure 1(b) and recall that we are assuming $D = \{4\}$. We have $V(p^1)\Delta V(p^2) = \{3,4,5\}$, and all coverage sets must contain some node in $V(p^1)\Delta V(p^2)$. Thus, we impose the constraint

$$x_3 + x_4 + x_5 \geq 1,$$

that is a constraint of the form (SC.2).

If we set $D = \{4,8\}$, the path

$$\overline{p}^2 = \langle s,1,2,3,4,5,6,7,9,1,2,3,4,5,6,7,t\rangle$$

is such that p^1 and $\overline{p}^2$ are *not* D-equivalent. We have $V(p^1)\Delta V(\overline{p}^2) = \{8\}$; thus, all coverage sets must contain node 8. Thus, we also need to impose the constraint $x_8 \geq 1$.

In this case, it can be checked that (*SC*) is

$$\begin{aligned} \min \quad & \sum_{u\in V} c_u x_u \\ \text{s.t.} \quad & x_3 + x_4 + x_5 \geq 1 \\ & x_8 \geq 1 \\ & x_u \in \{0,1\} \qquad u \in V. \end{aligned}$$

Any other constraint of the form (SC.2) is dominated by the two set covering constraints that appear previously.

3.2. Separation

Our goal is to design an algorithm that, given a vector $\overline{\boldsymbol{x}} \in [0,1]^I$, either establishes that $\overline{\boldsymbol{x}}$ satisfies all of constraints (SC.2) or returns an inequality of the form (SC.2) that is violated by $\overline{\boldsymbol{x}}$.

To proceed, we need to introduce some additional notation. If q^1 is a uv path and q^2 is a vw path, we denote by $q^1 \circ q^2$ the uw path obtained by appending q^2 to q^1, and we say that we *compose* the paths q^1 and q^2. For $u, v, d \in V$, we denote by $\mathcal{P}_d^{uv}$ the set of all uv paths that do not traverse d. Moreover, we denote by V_d^{uv} the set of all the nodes that belong to some path in $\mathcal{P}_d^{uv}$, that is, $V_d^{uv} = \bigcup_{p\in\mathcal{P}_d^{uv}} V(p)$. Finally, by considering all possible exit nodes, we define $V_d^{\beta T} = \bigcup_{t\in T} V_d^{\beta t}$.

Definition 4. Let $\overline{\boldsymbol{x}} \in [0,1]^I$; let π^1 and π^2 be two $\alpha\beta$-paths for some $\alpha, \beta \in V$; and let $d \in D$. We say that (π^1, π^2) is an *ambiguous* (α, β, d) *triangle* with respect to $\overline{\boldsymbol{x}}$ if

1. The set $\mathcal{P}_d^{s\alpha} \neq \emptyset$;
2. The set $\mathcal{P}_d^{\beta t} \neq \emptyset$ for some $t \in T$;
3. The path π^1 is obtained as $\pi^1_{\alpha d} \circ \pi^1_{d\beta}$, where $\pi^1_{\alpha d}$ and $\pi^1_{d\beta}$ are an αd path and a $d\beta$ path, respectively;
4. The path π^2 is a path in $\mathcal{P}_d^{\alpha\beta}$; and
5. $\sum_{u\in(V(\pi^1)\Delta V(\pi^2))\setminus Y} \overline{x}_u < 1$, where $Y = V_d^{s\alpha} \cup V_d^{\beta T}$.

An (α, β, d) ambiguous triangle (π^1, π^2) is a *simple ambiguous triangle* if $\pi^1_{\alpha d}$, $\pi^1_{d\beta}$ and π^2 are simple paths.

Condition 1 requires that α can be reached from the entry without traversing d, and Condition 2 requires that an exit can be reached from β without traversing d. Condition 3 asks that π^1 is a path from α to β that *does* traverse d, and Condition 4 asks that π^2 is a path from α to β that *does not* traverses d. Finally, Condition 5 looks at the symmetric difference $V(\pi^1)\Delta V(\pi^2)$, from which we exclude the nodes that are in some path from the entry to α not traversing d and those that are in some path from β to an exit not traversing d. The condition imposes that the total weight of $\overline{\boldsymbol{x}}$ on such set of nodes does not exceed one. When $\overline{\boldsymbol{x}}$ is the characteristic vector of $S \subseteq V$, Condition 5 says that π^1 and π^2 can differ on a node in S only if this node belongs to a path from the entry to α that does not traverse d or to a path from β to an exit that does not traverse d. A pictorial representation of a (α, β, d) ambiguous triangle (π^1, π^2) is given in Figure 2.

Example 2 (Continued). Consider the CFG in Figure 1(b), and suppose that $D = \{4,8\}$ and $I = V$. Let $S = \{s,1,2,6,7,8,9,t\}$ and let $\overline{\boldsymbol{x}} \in \{0,1\}^I$ be the characteristic vector of S. A (2, 6, 4) ambiguous triangle with respect to

$\bar{\boldsymbol{x}}$ is given by $\pi^1 = \langle 2,3,4,5,6\rangle$, and $\pi^2 = \langle 2,8,6\rangle$. To check this, we verify that all the conditions of Definition 4 are satisfied. It can be easily checked that there exists a path from s to 2 that does not traverse 4. Similarly, there exists a path from 6 to t that does not traverse 4. This shows that the first two conditions are satisfied. For the third condition, we set $\pi^1_{24} = \langle 2,3,4\rangle$ and $\pi^1_{46} = \langle 4,5,6\rangle$. Condition 4 is also satisfied because $\pi^2 = \langle 2,8,6\rangle$ does not traverse 4. To verify Condition 5, we note that $V(\pi^1)\Delta V(\pi^2) = \{3,4,5,8\}$ and $S \cap V(\pi^1)\Delta V(\pi^2) = \{8\}$. Because there exists some path from s to 2 that traverses 8 and not 4, we have that $8 \in V^{s2}_4 = \{s,1,2,\ 6,7,8,9\}$. Thus, Condition 5 holds.

Paths π^1 and π^2 are also a (2, 6, 8) ambiguous triangle with the respect to $\boldsymbol{x}'$, the characteristic vector of $S' = \{s,1,2,3,4,5,6,7,9,t\}$. Conditions 1–4 are easily checked. To check Condition 5, we note that $S \cap V(\pi^1)\Delta V(\pi^2) = \{4\}$ and $4 \in V^{s2}_8 = \{s,1,2,3,4,5,6,7,9\}$.

Next, we describe the mechanics of the separation algorithm, we discuss its running time, and we provide some intuition on its structure. The crucial feature of our separation algorithm is that it checks for the existence of an ambiguous triangle with respect to $\bar{\boldsymbol{x}}$: If there exists an ambiguous triangle with respect to $\bar{\boldsymbol{x}}$, then a new inequality of type (SC.2) violated by $\bar{\boldsymbol{x}}$ is generated; if there is no ambiguous triangle with respect to $\bar{\boldsymbol{x}}$, then all constraints (SC.2) are satisfied by $\bar{\boldsymbol{x}}$. In particular, if $\bar{\boldsymbol{x}}$ is binary, and no inequality of (SC.2) is violated by $\bar{\boldsymbol{x}}$, then $\bar{\boldsymbol{x}}$ is the characteristic vector of a coverage set of D.

Algorithm 1 (Separation Algorithm)

Input: $G = (V,E)$, $s \in V$, $T \subseteq V$, $D \subseteq V$, $I \subseteq V$, $\bar{\boldsymbol{x}} \in [0,1]^I$.

Output: inequality of form (SC.2) violated by $\bar{\boldsymbol{x}}$, or certificate that $\bar{\boldsymbol{x}}$ satisfies all constraints (SC.2).

1 **for** *each* $d \in D$ **do**
2 **for** *each node* u *in* $G \setminus d$ **do**
3 compute $V^+_d(u)$ in $G \setminus d$
4 **for** *any two nodes* α *and* β *in* $V \setminus d$ **do**
5 $V^{s\alpha}_d = \{u \mid \alpha \in V^+_d(u) \text{ and } u \in V^+_d(s)\}$
6 $V^{\beta T}_d = \{u \mid u \in V^+_d(\beta) \text{ and } V^+_d(u) \cap T \neq \emptyset\}$
7 **if** $V^{s\alpha}_d \neq \emptyset$ *and* $V^{\beta T}_d \neq \emptyset$ **then**
8 Let $Y = V^{s\alpha}_d \cup V^{\beta T}_d$
9 Define $\boldsymbol{w} \in [0,1]^V$ such that $w_u = \bar{x}_u$ if $u \in I \setminus Y$, and $w_u = 0$ otherwise
10 Let π^1 be a shortest node-weighted $\alpha\beta$-path in $(G, \boldsymbol{w})$ traversing d. If no such path exists go back to line 4 and try a new (α,β) pair
11 Let π^2 be a shortest node-weighted $\alpha\beta$-path in $(G \setminus d, \boldsymbol{w})$. If no such path exists go back to line 4 and try a new (α,β) pair
12 **if** $\sum_{u \in V(\pi^1)\Delta V(\pi^2)} w_u < 1$ **then**
13 **return** $\sum_{u \in (V(\pi^1)\Delta V(\pi^2)) \setminus Y} x_u \geq 1$
14 **return** $\bar{\boldsymbol{x}}$ *satisfies all constraints* (SC.2)

Algorithm 1 shows the pseudocode of the separation algorithm. At line 3, the set $V^+_d(u)$ is computed by a breadth-first search (BFS) in $G \setminus d$. Lines 5 and 6 compute $V^{s\alpha}_d$ and $V^{\beta T}_d$ based on the sets computed at line 3. At line 7, the algorithm verifies if the current (α,β,d) triple satisfies Condition 1 and Condition 2 of Definition 4. If so, at line 8, the set Y of Definition 4 is computed. At line 9, we assign weights to each node of G, according to the current vector $\bar{\boldsymbol{x}}$ and set Y. At line 10, we check if Condition 3 of Definition 4 is satisfied. To find π^1, we first construct an arc-weighted graph G' from the node-weighted graph G. Precisely, we "split" each node $u \in V$ into u^+ and u^-, and we add arc (u^+,u^-) of weight w_u, and arcs $\{(v^-,u^+) : (v,u) \in E\}$, $\{(u^-,v^+) : (u,v) \in E\}$ of weight 0. Then in G', we find a shortest path from α^- to d^+, which we map to a path $\pi^1_{\alpha d}$ in G from α to d. Similarly, in G' we find a shortest path from d^- to β^+, which we map to a path $\pi^1_{d\beta}$ in G from d to β. Finally, we set $\pi^1 = \pi^1_{\alpha d} \circ \pi^1_{d\beta}$. At line 11, we check if Condition 4 of Definition 4 is satisfied. We proceed in a similar way as before to find a shortest node-weighted path π^2 from α to β in $G \setminus d$. If we reach line 13, also Condition 5 of Definition 4 is satisfied; thus, we have found an ambiguous triangle and return the corresponding constraint violated by $\bar{\boldsymbol{x}}$. Only after checking all $d \in D$ and all (α,β,d) triples can we conclude at line 14 that $\bar{\boldsymbol{x}}$ satisfy all constraints (SC.2). The outer loop is executed $O(|V|)$ times, and the two inner loops are executed $O(|V|)$ and $O(|V|^2)$ times, respectively. Each BFS has runtime in $O(|E|)$, each shortest path computation has runtime in $O(|V|^2)$, and computing Y takes $O(|V|)$. It follows that the runtime of the algorithm is $O(|V|(|V||E| + |V|^2(|V| + |V|^2))) = O(|V|^5)$. If the vector $\bar{\boldsymbol{x}}$ in input is binary, then the shortest path computations at line 10 and line 11 can be performed with BFS computations, and the total runtime reduces to $O(|V|^3|E|)$.

The correctness of the separation algorithm is formally proved in Section 3.3. Intuitively, a (α,β,d) ambiguous triangle (π^1,π^2) with respect to $\overline{\boldsymbol{x}}$ effectively yields two *st* paths p^1 and p^2 that are not D-equivalent, which define a constraint of type (SC.2) violated by $\overline{\boldsymbol{x}}$. The two paths p^1 and p^2 can be obtained by composing both π^1 and π^2 with a $s\alpha$ path, with a βt path, for some $t \in T$, and with all cycles in $G \setminus d$ that contain α or β. In our algorithm, we want to generate ambiguous triangles yielding paths p^1 and p^2 that overlap as much as possible. This way, the symmetric difference of p^1 and p^2 is minimal, yielding a constraint that is less likely to be dominated. In fact, each constraint that we generate when fixing a triple (α,β,d) is *not* dominated by any other constraints that could be obtained from a (α,β,d) ambiguous triangle.

3.3. Proof of Separation Algorithm Correctness

In this section, we will prove that the separation algorithm is correct, that is, that either it returns a constraint of the form (SC.2) that is violated by $\boldsymbol{x}$ or that it correctly states that all constraints (SC.2) are satisfied by $\boldsymbol{x}$. The next lemma shows that, when checking for the existence of an ambiguous triangle, we can restrict our search to the simple ones.

Lemma 1. *There exists an ambiguous triangle if and only if there exists a simple ambiguous triangle.*

Proof. One direction is trivial, so we prove that if there is an ambiguous triangle, then there is a simple ambiguous triangle. Suppose that (π^1,π^2) is an ambiguous (α,β,d) triangle with respect to $\boldsymbol{x} \in [0,1]^I$, and let G_1 and G_2 be the node-weighted subgraphs of G containing the arcs and nodes in π^1 and π^2, respectively, such that each node $u \in I$ has weight x_u and all remaining nodes have weight 0. In G_1, we compute a shortest αd path $\overline{\pi}^1_{\alpha d}$ and a shortest $d\beta$ path $\overline{\pi}^1_{d\beta}$, and in G_2, we compute a shortest $\alpha\beta$ path $\overline{\pi}^2$. Because the node weights are nonnegative, $\overline{\pi}^1_{\alpha d}$, $\overline{\pi}^1_{d\beta}$, and $\overline{\pi}^2$ are simple, and they satisfy Conditions 1–5 of Definition 4. □

The next theorem shows that if $\boldsymbol{x} \in [0,1]^I$ satisfies all constraints (SC.2), then there exists no simple ambiguous triangle with respect to $\boldsymbol{x}$. Equivalently, we show that if there is a simple ambiguous triangle with respect to $\boldsymbol{x}$, then $\boldsymbol{x}$ violates a constraint of the form (SC.2).

Theorem 1. *Let (π^1,π^2) be a simple ambiguous (α,β,d) triangle with respect to $\boldsymbol{x} \in [0,1]^I$. Then, for some $t \in T$, there exist two st paths p^1 and p^2 such that $d \in V(p^1)\Delta V(p^2)$ and $\sum_{u\in(V(p^1)\Delta V(p^2))} x_u < 1$. Moreover*

$$V(p^1)\Delta V(p^2) = (V(\pi^1)\Delta V(\pi^2)) \setminus Y, \tag{1}$$

where $Y = V_d^{s\alpha} \cup V_d^{\beta T}$.

Proof. Our proof is algorithmic. We start with $q^1 = \pi^1$ and $q^2 = \pi^2$. At each step, we compose q^1 and q^2 with other paths, so that symmetric difference $V(q^1)\Delta V(q^2)$ reduces but still contains d, until q^1 and q^2 are two *st* paths that satisfy (1). At the end, we set $p^1 = q^1$ and $p^2 = q^2$. Moreover, because (1) holds and because (π^1,π^2) is an ambiguous triangle, we will have that $\sum_{u\in(V(p^1)\Delta V(p^2))} x_u < 1$. In the remainder of the proof, we give the details of the algorithm to construct p^1 and p^2.

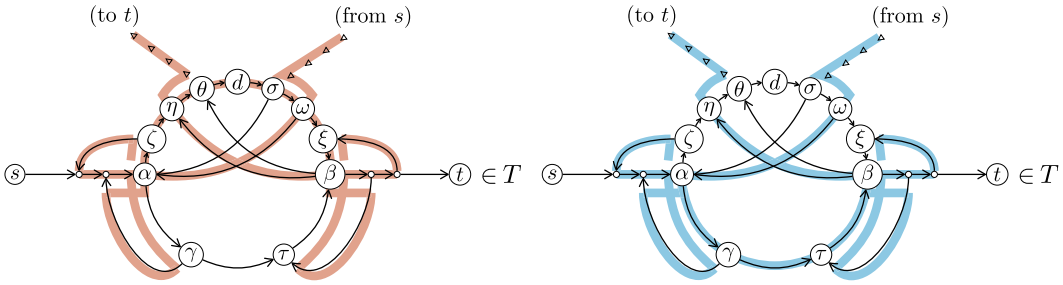
Let $\pi^1 = \pi^1_{\alpha d} \circ \pi^1_{d\beta}$, where $\pi^1_{\alpha d}$ and $\pi^1_{d\beta}$ are the simple αd path and the simple $d\beta$ path of Definition 4. First, we determine the following nodes:

- The first node ξ in $\pi^1_{d\beta}$ such that $\xi \in V_d^{\beta T}$.
- The first node τ in π^2 such that $\tau \in V_d^{\beta T}$.
- The last node ζ in $\pi^1_{\alpha d}$ such that $\zeta \in V_d^{s\alpha}$.
- The last node γ in π^2 such that $\gamma \in V_d^{s\alpha}$.
- The first node η in $\pi^1_{\alpha d}$ such that $\eta \in V_d^{\beta T}$.
- The last node θ in $\pi^1_{\alpha d}$ such that $\theta \in V_d^{\beta T}$.
- The first node σ in $\pi^1_{d\beta}$ such that $\sigma \in V_d^{s\alpha}$.
- The last node ω in $\pi^1_{d\beta}$ such that $\omega \in V_d^{s\alpha}$.

Figure 3 provides a pictorial representation. Note that ξ,τ,ζ,γ are guaranteed to exist because we can have ξ,τ coincide with β and ζ,γ coincide with α. In contrast, η and θ exist if and only if $\pi^1_{\alpha d}$ contains some node in $V_d^{\beta T}$. Similarly, σ and ω exist if and only if $\pi^1_{d\beta}$ contains some node in $V_d^{s\alpha}$.

If $\xi \neq \beta$, then there exists a directed cycle C_ξ containing β and all the nodes in $V(\pi^1_{d\beta}) \cap V_d^{\beta T}$, but not containing d. Furthermore, if $\tau \neq \beta$, then there exists a directed cycle C_τ containing β and all the nodes in $V(\pi^2) \cap V_d^{\beta T}$, but

Figure 3. (Color online) Reference Nodes in the Proof of Theorem 1



Notes. The path displayed on the left is p^1; the path displayed on the right is p^2. By construction, $V(p^1)\Delta V(p^2) = (V(\pi^1)\Delta V(\pi^2)) \setminus Y$.

not containing d. We update

$$q^1 \leftarrow q^1 \circ C_\xi \circ C_\tau$$
$$q^2 \leftarrow q^2 \circ C_\xi \circ C_\tau.$$

Similarly, if $\zeta \neq \alpha$, then there exists a directed cycle C_ζ containing α and all the nodes in $V(\pi^1_{\alpha d}) \cap V_d^{s\alpha}$, but not containing d. Furthermore, if $\gamma \neq \alpha$, then there exists a directed cycle C_γ containing α and all the nodes in $V(\pi^2) \cap V_d^{s\alpha}$, but not containing d. We update

$$q^1 \leftarrow q^1 \circ C_\zeta \circ C_\gamma$$
$$q^2 \leftarrow q^2 \circ C_\zeta \circ C_\gamma.$$

At this point, we have,

$$V(q^1)\Delta(V(q^2) = (V(\pi^1)\Delta V(\pi^2) \setminus Y) \cup Y_1 \cup Y_2,$$

where $Y = V_d^{s\alpha} \cup V_d^{\beta T}$, $Y_1 = V(\pi^1_{d\beta}) \cap (V_d^{s\alpha} \setminus V_d^{\beta T})$ and $Y_2 = V(\pi^1_{\alpha d}) \cap (V_d^{\beta T} \setminus V_d^{s\alpha})$.

If Y_1 is empty, to turn q^1 and q^2 into $s\beta$ paths, we update

$$q^1 \leftarrow q_{s\alpha} \circ q^1$$
$$q^2 \leftarrow q_{s\alpha} \circ q^2,$$

where $q_{s\alpha}$ is a $s\alpha$ path that does not traverse d, which exists by Condition 1 in Definition 4. If Y_1 is nonempty, then σ and ω exist. Moreover σ must occur before ξ in $\pi^1_{d\beta}$. By definition, we have (i) $\sigma \in V(p)$ for some $p \in \mathcal{P}_d^{s\alpha}$ and (ii) $\omega \in V(p')$ for some $p' \in \mathcal{P}_d^{s\alpha}$. Let $q_{s\sigma}$ be a $s\sigma$ path that does not traverse d, which exists by (i) and let $q_{\omega\alpha}$ be a path from ω to α that does not go through d, which exists by (ii). Moreover, let $\pi^1_{\sigma\omega}$ be the subpath of $\pi^1_{d\beta}$ from σ to ω. We update

$$q^1 \leftarrow q_{s\sigma} \circ \pi^1_{\sigma\omega} \circ q_{\omega\alpha} \circ q^1$$
$$q^2 \leftarrow q_{s\sigma} \circ \pi^1_{\sigma\omega} \circ q_{\omega\alpha} \circ q^2.$$

Now q^1 and q^2 are $s\beta$ paths and all nodes in Y_1 belong to both q^1 and q^2.

We proceed similarly with Y_2. If Y_2 is empty, then (1) is satisfied. To turn q^1 and q^2 into st paths, we update

$$q^1 \leftarrow q^1 \circ q_{\beta t}$$
$$q^2 \leftarrow q^2 \circ q_{\beta t},$$

where $q_{\beta t}$ is a βt path with $t \in T$ that does not traverse d, which exists by Condition 2 in Definition 4. If Y_2 is nonempty, then η and θ exist. Moreover θ must occur after ζ in $\pi^1_{\alpha d}$. By definition, we have (i) $\theta \in V(p)$ for some $t \in T$ and $p \in \mathcal{P}_d^{\beta t}$ and (ii) $\eta \in V(p')$ for some $t' \in T$ and $p' \in \mathcal{P}_d^{\beta t'}$. Let $q_{\theta t}$ be a path from θ to $t \in T$ that does not traverse d, which exists by (i) and let $q_{\beta\eta}$ be a path from β to η that does not go through d, which exists by (ii). Moreover, let $\pi^1_{\eta\theta}$ be the subpath of $\pi^1_{\alpha d}$ from η to θ. We update

$$q^1 \leftarrow q_{\theta t} \circ q^1 \circ q_{\beta\eta} \circ \pi_{\eta\theta}$$
$$q^2 \leftarrow q_{\theta t} \circ q^2 \circ q_{\beta\eta} \circ \pi_{\eta\theta}.$$

Now q^1 and q^2 are two st paths, with $t \in T$, and all nodes in Y_2 belong to both q^1 and q^2; thus, (1) is satisfied. □

Let $\boldsymbol{x} \in [0,1]^I$. An immediate consequence of Theorem 2 is that if (π^1, π^2) is an ambiguous (α, β, d) triangle with respect to $\boldsymbol{x}$, then $\boldsymbol{x}$ violates an inequality of form (SC.2). We now prove the converse of Theorem 2, that is, if there

exists no ambiguous triangle with respect to $\boldsymbol{x}$, then $\boldsymbol{x}$ satisfies all the inequalities (SC.2). In particular, if $\boldsymbol{x}$ is binary, then it is the characteristic vector of a coverage set.

In the following, if p is the path defined by $\langle u_0, u_1, \dots, u_k \rangle$ and i, j are two indices such that $0 \le i \le j \le k$, we denote by $p(i,j)$ the subpath of p defined by $\langle u_i, u_{i+1}, \dots, u_j \rangle$.

Theorem 2. *Let $\boldsymbol{x} \in [0,1]^I$. If $\boldsymbol{x}$ violates an inequality of the form (SC.2), then there exists an ambiguous triangle with respect to $\boldsymbol{x}$.*

Proof. Because $\boldsymbol{x}$ violates an inequality of form (SC.2), there exist $t \in T$ and two not D-equivalent st paths p^1, p^2, such that

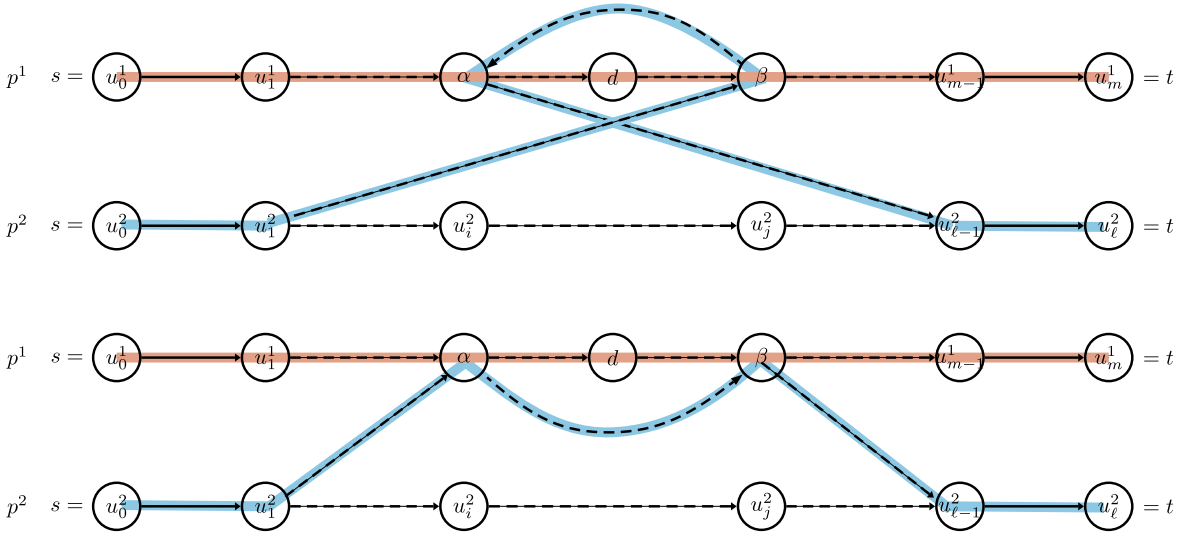
$$\sum_{u \in (V(p^1) \Delta V(p^2))} x_u < 1.$$

Let p^1 be defined by the sequence $\langle s = u_0^1, u_1^1, \dots, u_{m-1}^1, u_m^1 = t \rangle$, and p^2 be defined by the sequence $\langle s = u_0^2, u_1^2, \dots, u_{\ell-1}^2, u_\ell^2 = t \rangle$. Let $d \in D \cap (V(p^1) \Delta V(p^2))$, and suppose without loss of generality $d \in V(p^1) \setminus V(p^2)$, that is, p^1 traverses d and p^2 does not. We consider the first occurrence of d in p^1. Let r be the smallest index in $\{0, \dots, m\}$ such that $u_r^1 = d$. Let $\alpha = u_i^1$ be the first node in $V(p^2)$ that we find traversing p^1 "backward" from u_r^1 to u_0^1. Similarly, let $\beta = u_j^1$ be the first node in $V(p^2)$ that we find traversing p^1 "forward" from u_r^1 to u_m^1.

We consider two cases. In the first case, p^2 contains a subpath from β to α, that is, $u_h^2 = \beta$, $u_k^2 = \alpha$, for some $0 \le h < k \le \ell$. See the top graph in Figure 4 for reference. Then, there is a directed cycle consisting of the $\alpha\beta$ path $p^1(i,j)$ and of the $\beta\alpha$ path $p^2(h,k)$. We prove that we have a (α, α, d) ambiguous triangle with respect to $\boldsymbol{x}$. To show that $\mathcal{P}_d^{s\alpha} \neq \emptyset$, we recall that $p^1(0,i)$ is a $s\alpha$ path that does not contain d because u_r^1 is the first occurrence of d and $i < r$. To show that $\mathcal{P}_d^{\alpha t} \neq \emptyset$ for some $t \in T$, we recall that $p^2(k,\ell)$ is a αt path that does not contain d because p^2 does not traverse d. Moreover, we define π^2 as the trivial path consisting of the single node α, and we let $\pi^1 = p^1(i,j) \circ p^2(h,k)$. Finally, we show that Condition 5 of Definition 4 is satisfied by proving that $(V(\pi^1) \Delta V(\pi^2)) \setminus V_d^{s\alpha} \subseteq V(p^1) \Delta V(p^2)$. Recall that $V(\pi^2) = \{\alpha\}$; thus, $V(\pi^1) \Delta V(\pi^2) = V(\pi^1) \setminus \{\alpha\}$. First, because $p^2(0,k)$ is a $s\alpha$ path that does not contain d, we have $V(p^2(h,k)) \setminus V_d^{s\alpha} = \emptyset$, and in particular $\beta \in V_d^{s\alpha}$. Moreover, $V(p^1(i,j)) \setminus \{\alpha, \beta\} \subseteq V(p^1) \setminus V(p^2)$ by construction. As a consequence, we obtain $(V(\pi^1) \setminus \{\alpha\}) \setminus V_d^{s\alpha} \subseteq V(p^1) \Delta V(p^2)$.

In the second case, p^2 does not contain any subpath from β to α, that is, if $u_h^2 = \beta$, $u_k^2 = \alpha$, then $h > k$. See the bottom graph in Figure 4 for reference. If p^1 contains multiple occurrences of d, we can consider $p^1(0,j) \circ p^2(h,\ell)$ instead of p^1. Thus, we now assume that p^1 traverses d exactly once. We show that we have an ambiguous (α, β, d) triangle with $\pi^1 = p^1(i,j)$ and $\pi^2 = p^2(k,h)$. As before, we can show $\mathcal{P}_d^{s\alpha} \neq \emptyset$, which implies $\alpha \in V_d^{s\alpha}$. Moreover, $\mathcal{P}_d^{\beta t} \neq \emptyset$ for some $t \in T$ because $p^2(h,\ell)$ is a βt path with $t \in T$, and it does not contain d. This implies $\beta \in V_d^{\beta T}$. Finally, we show that Condition 5 of Definition 4 is satisfied by proving that $(V(\pi^1) \Delta V(\pi^2)) \setminus (V_d^{s\alpha} \cup V_d^{\beta T}) \subseteq$

Figure 4. (Color online) The Two Cases Analyzed in the Proof of Theorem 2



$V(p^1)\Delta V(p^2)$. Because by construction $V(p^1(i,j)) \setminus \{\alpha,\beta\} \subseteq V(p^1) \setminus V(p^2)$, we have $V(\pi^1) \setminus (V_d^{s\alpha} \cup V_d^{\beta T}) \subseteq V(p^1)\Delta V(p^2)$. Moreover, if $w \in V(p^2(k,h) \cap V(p^1)$, then w must be either in $p^1(0,i)$ or in $p^1(j,m)$ based on how we defined α and β; thus, $w \in V_d^{s\alpha} \cup V_d^{\beta T}$. This implies $V(\pi^2) \setminus (V_d^{s\alpha} \cup V_d^{\beta T}) \subseteq V(p^1)\Delta V(p^2)$. As a consequence, we obtain $(V(\pi^1)\Delta V(\pi^2)) \setminus (V_d^{s\alpha} \cup V_d^{\beta T}) \subseteq V(p^1)\Delta V(p^2)$. □

The correctness of the separation algorithm relies on the following corollary.

Corollary 1. *The vector $\boldsymbol{x} \in [0,1]^I$ satisfies all constraints (SC.2) if and only if there exists no ambiguous triangle with respect to $\boldsymbol{x}$.*

Proof. Sufficiency immediately follows from Theorem 2. To prove necessity, by contradiction we suppose that there exists an ambiguous triangle with respect to $\boldsymbol{x}$, and we show that $\boldsymbol{x}$ violates a constraint of form (SC.2). By Lemma 1, we know that there exists a simple ambiguous (α,β,d) triangle with respect to $\boldsymbol{x}$. Our claim then follows by applying Theorem 1 and Proposition 1. □

We are finally ready to prove the main result of this section.

Theorem 3. *The separation algorithm either returns a constraint of type (SC.2) violated by $\boldsymbol{x}$, or it correctly states that all constraints (SC.2) are satisfied by $\boldsymbol{x}$.*

Proof. By Corollary 1, we can equivalently show that the separation algorithm either finds an ambiguous triangle with respect to $\boldsymbol{x}$, or it correctly states that no such ambiguous triangle exists. Clearly, if the algorithm exits at line 13, we have that (π^1,π^2) is a (α,β,d) ambiguous triangle with respect to $\boldsymbol{x}$. We need to prove that if the algorithm exits at line 14, after having checked all $d \in D$ and all (α,β,d) triples, then there is no ambiguous triangle with respect to $\boldsymbol{x}$. By contradiction, suppose that $(\overline{\pi}^1,\overline{\pi}^2)$ is a an (α,β,d) ambiguous triangle with respect to $\boldsymbol{x}$. First, by Lemma 1, we can assume that $(\overline{\pi}^1,\overline{\pi}^2)$ is a simple ambiguous triangle. This implies that $\overline{\pi}^1$ contains only one occurrence of d. Moreover, we can assume without loss of generality that $V(\overline{\pi}^1) \cap V(\overline{\pi}^2) = \{\alpha,\beta\}$, because if not, we can redefine α to be the first node in $V(\overline{\pi}^2)$ that we find traversing $\overline{\pi}^1$ "backward" from d, and we can redefine β to be the first node in $V(\overline{\pi}^2)$ that we find traversing $\overline{\pi}^1$ "forward" from d. Thus, when we consider the triple (α,β,d), we have

$$1 > \sum_{u \in (V(\overline{\pi}^1)\Delta V(\overline{\pi}^2)) \setminus Y} x_u = \sum_{u \in (V(\overline{\pi}^1)\Delta V(\overline{\pi}^2))} w_u = \sum_{u \in V(\overline{\pi}^1)} w_u + \sum_{u \in V(\overline{\pi}^2)} w_u - 2w_\alpha - 2w_\beta. \tag{2}$$

Let π^1 and π^2 be the shortest paths computed at line 10 and line 11, respectively, of the separation algorithm for the triple (α,β,d). We have

$$\begin{aligned}\sum_{u \in (V(\pi^1)\Delta V(\pi^2))} w_u &= \sum_{u \in V(\pi^1)} w_u + \sum_{u \in V(\pi^2)} w_u - 2 \sum_{u \in V(\pi^1) \cap V(\pi^2)} w_u \\ &\le \sum_{u \in V(\pi^1)} w_u + \sum_{u \in V(\pi^2)} w_u - 2w_\alpha - 2w_\beta \\ &\le \sum_{u \in V(\overline{\pi}^1)} w_u + \sum_{u \in V(\overline{\pi}^2)} w_u - 2w_\alpha - 2w_\beta < 1,\end{aligned}$$

where the first inequality follows from the fact that $V(\pi^1) \cap V(\pi^2) \supseteq \{\alpha,\beta\}$, the second inequality follows from the fact that π^1 and π^2 are shortest paths, and the strict inequality follows from (2). This contradicts the fact that the condition at line 12 is not satisfied. □

4. Evaluation

Table 2 shows our evaluated applications. These precisely match those from Ohmann et al. (2016b) and are mostly taken from the Software-Artifact Infrastructure Repository (Do et al. 2005, Rothermel et al. 2006). All applications are written in C, although some applications have multiple versions. These applications were selected because they contain seeded and/or real faults that can be enabled for debugging and program analysis research. In results that follow, we compile only nonfaulty builds to maintain the comparison with prior work by Ohmann et al. (2016b). For each application version, we performed two separate compilations.

First, we used the methods to optimize instrumentation for basic block coverage. In terms of our problem definition from Section 2:

- G is the intraprocedural CFG for each function,
- $T = V$ (the set of all basic blocks, indicating that we could crash or halt at any point),

Table 2. Evaluated Applications, Ordered by Size

Application	Description	Versions	Mean LOC
tcas	Siemens	1	173
schedule2	Siemens	1	373
schedule	Siemens	1	413
replace	Siemens	1	563
tot_info	Siemens	1	564
print_tokens2	Siemens	1	568
print_tokens	Siemens	1	727
ccrypt	Linux utility	1	5,280
gzip	Linux utility	5	8,114
space	ADL interpreter	1	9,563
exif	Linux utility	1	10,611
bc	Linux utility	1	14,292
sed	Linux utility	7	14,314
flex	Linux utility	5	14,946
grep	Linux utility	5	15,460
bash	Linux shell	6	80,443
Gcc	C compiler	1	222,196

- $I = V$ (indicating that we may instrument anywhere), and
- $D = V$ (indicating that we want coverage data for the entire program).

Costs for each block are determined by LLVM's block frequency analysis, which provides a static approximation of each basic block's execution frequency (LLVM Project 2018).

Second, we optimized for coverage at only call sites in aid of program analysis work that uses call-site coverage data (Nishimatsu et al. 1999, Ohmann and Liblit 2013) and for complete comparison with Ohmann et al. (2016b). Formally,

- G, T, and costs (c) are defined as before, but
- I = {blocks with at least one call site}, and
- D = {blocks with at least one call site} (indicating that we only care about coverage data at call sites).

We ran all experiments on a 24-core, 3.07-GHz Intel Xeon X5675 CPU with 192 GB of RAM. However, as noted in the following sections, all methods use only a single core, and we cap memory use at 32 GB. Our system runs Ubuntu 16.04.5 LTS. We downloaded a copy of prior solvers from Ohmann et al. (2016b), which use the csi-cc instrumenting compiler (Ohmann and Liblit 2013). We built our new set covering implementation into csi-cc and built with Clang/LLVM 4.0 (Lattner and Adve 2004). The network flow–based algorithm from prior work models problem instances using the general algebraic modeling sdystem (GAMS; GAMS Development Corporation 2018) and uses the Gurobi Optimizer (Gurobi Optimization LLC 2018) (v9.1) to solve the generated mixed integer linear program; our new set covering–based method uses the Gurobi Optimizer (v9.1) directly. In all cases, Gurobi is called from within the program compiler. The software implementation and instructions for building and reproducing results are available from the paper's associated Github repository (Michini et al. 2022).

4.1. Compile Time

We began by measuring the amount of time and memory used to compile each application version with each available method. We compared our own method (set cover) to two approaches from Ohmann et al. (2016b): the network-flow-based MILP approach (netflow) and the top-down locally optimal approach (local).

We first examined the number of compilations that exceeded our per-program time (three hours) or memory (32 GB) cutoffs. We selected very generous budgets to allow for as many direct comparisons as possible. Table 3 shows the results as the percentage of compilations for each application that ran out of time or memory. Recall that some applications have multiple versions, per Table 2; percentages between 0% and 100% indicate that some but not all versions ran out of time or memory. The table does not tabulate time and memory failures separately, but all failures for netflow ran out of memory, while all failures for local and set cover ran out of time.

For basic block instrumentation, our new method exceeds the time threshold far less often than prior work, including solutions for seven application versions that ran out of time with the local heuristic. Instrumentation for call sites performs exceptionally well using our approach: We find an optimal instrumentation plan for every program tested, never running out of either time or memory.

In Table 4, we compare the compile times (in seconds) for the netflow and set-cover approaches, including only those applications for which both methods completed within the time or memory limit. These are the only two

Table 3. Percentage of Compilations That Ran Out of Time or Memory

	Basic blocks			Call sites		
Application	Netflow	Local	Set cover	Netflow	Local	Set cover
tcas	0%	0%	0%	0%	0%	0%
schedule2	0%	0%	0%	0%	0%	0%
schedule	100%	0%	0%	0%	0%	0%
replace	100%	0%	0%	0%	0%	0%
totinfo	100%	0%	0%	0%	0%	0%
printtokens2	100%	0%	0%	0%	0%	0%
printtokens	100%	0%	0%	100%	0%	0%
ccrypt	100%	0%	0%	100%	0%	0%
gzip	100%	20%	0%	100%	0%	0%
space	100%	0%	0%	100%	0%	0%
exif	100%	0%	0%	100%	0%	0%
bc	100%	0%	0%	100%	0%	0%
sed	100%	100%	57%	100%	43%	0%
flex	100%	100%	80%	100%	80%	0%
grep	100%	100%	100%	100%	0%	0%
bash	100%	100%	33%	100%	33%	0%
gcc	100%	100%	100%	100%	100%	0%
mean	88%	31%	22%	65%	15%	0%

Notes. Lower is better. Applications are ordered by size.

methods, to our knowledge, that provide provably optimal coverage plans. The set-cover method is clearly more efficient; in the most extreme example, basic block coverage optimization for schedule2 takes 473 seconds (more than seven minutes) with netflow but less than 1 second with set cover. Overall, this table demonstrates that the new set cover–based exact approach we introduce in this work is often hundreds of times faster than the only other known exact approach for customized program coverage.

To compare the compile times of the locally optimal and set cover–based approaches, we computed the average compilation times, treating each timed-out compilation as though it had actually completed in exactly three hours. This saturation study makes the most optimistic possible assumption—that each timed-out compilation was just on the verge of completing. (This is far from the truth; we allowed many "local" compilations to continue for more than 24 hours, and they did not produce a solution.) We plot results as compilation time relative to compiling with no analysis or instrumentation whatsoever for program coverage. The results are shown in Figure 5; we exclude the small Siemens applications as these have trivial compilation times.

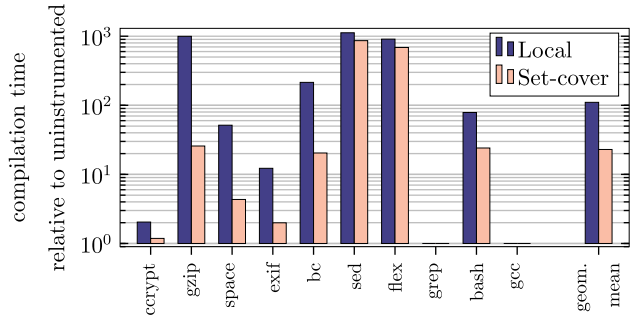
Figure 5(a) shows results for basic block coverage, excluding results for grep and gcc. (Because both these applications did not complete in three hours with either method, there are no results to compare.) For the other applications, we often see significant improvements. For example, gzip times out in some cases with local and saturated averages at almost a 1,000 times increase over noninstrumented build time; with set cover, the increase is 25 times. On average, for all larger applications except grep and gcc, the prior local heuristic is 110 times slower than base, whereas our new method is 23 times.

Figure 5(b) shows call coverage results. Here, we see immense improvements for all applications: sed sees the largest improvement, going from 836 times with local down to just 12 times with set cover. On average, our new method improves compilation overhead from 19 to 4 times. Furthermore, as mentioned earlier, these

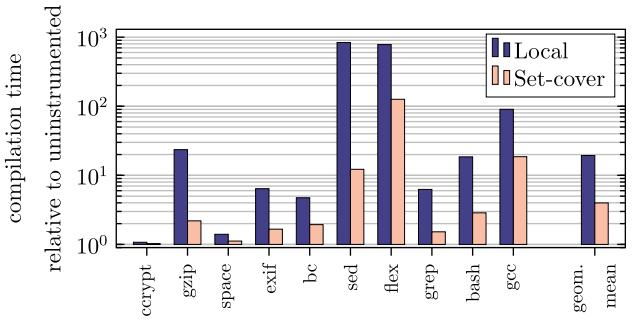
Table 4. Compilation Times (Seconds) for Exact Methods

Application	Instrumentation	Netflow	Set cover	Speedup ×
tcas	Basic blocks	10.28	0.46	22.46
schedule2	Basic blocks	473.00	0.78	604.26
tcas	Call sites	0.47	0.44	1.08
schedule2	Call sites	44.18	0.72	60.99
schedule	Call sites	286.50	0.77	374.47
replace	Call sites	12.48	0.84	14.84
totinfo	Call sites	126.54	0.67	188.60
printtokens2	Call sites	208.02	0.68	305.96

Figure 5. (Color online) Compile-Time Overhead, Saturated to Three Hours



(a) Relative compilation time when instrumenting basic blocks (grep and gcc excluded).



(b) Relative compilation time when instrumenting call sites.

Note. Lower is better. Applications are ordered by size.

improvements are *lower bounds*, in that we assumed that all time-out compilations would have completed immediately at three hours.

4.2. Plan Cost

Recall that our new method provides a guarantee of optimality for any instrumentation plan that it produces. We also examined how often this optimal solution was better than prior approaches (i.e., lower cost, meaning lower expected execution frequency of probes). The prior optimal method (netflow) also produces optimal solutions, so our plans are cost identical. Prior work found that the local heuristic often produces nontrivial improvements over the simple heuristic (Ohmann et al. 2016b). However, the authors did not examine how close this locally optimal solution was to the true optimum.

All results in this section compare expected run-time costs rather than actual running times. We used this model for two reasons. First, prior experiments by Ohmann et al. (2016b) also performed running-time comparisons and established this abstract cost model for comparing performance. Second, this model generalizes beyond the cost of executing individual probes and is especially useful for comparing performance when running-time differences would be hard to measure, as on short-running test cases.

Table 5 shows the count and percentage of functions in each application that received an improved solution (i.e., where local was *not* already fully optimal). Blank entries indicate that no local compilations completed, so results could not be compared. These results clearly show that local was already getting optimal results in most cases. However, some results do improve. For example, gzip with call coverage sees improvements for nine functions, or approximately 3% of the application's codebase. Overall, these improvements demonstrate that our new method is faster than prior approximations while still producing instrumentation plans that are sometimes better (and never worse).

4.3. Integer Programming Implementation Details

The purpose of this section is to provide details on the set covering instances arising from the applications and behavior of our method for solving these instances. The 40 (application, version) combinations tested with both basic-block and customized coverage result in a total of 18,494 functions. The CFG of each function varies in size, and many of the CFGs are quite small, resulting in trivial integer programming instances. To summarize the behavior of the integer programming–based algorithm, we partition the instances based on the number of nodes in

Table 5. Functions Where the Set-Cover Solution Improves on the Local Solution, Reported as Both an Absolute Count of Functions and a Percentage of All Functions in the Application

	Basic blocks		Call sites	
Application	Count	Percent	Count	Percent
tcas	1	11.1%	0	0.0%
schedule2	0	0.0%	1	6.2%
schedule	0	0.0%	0	0.0%
replace	0	0.0%	0	0.0%
totinfo	0	0.0%	0	0.0%
printtokens2	0	0.0%	0	0.0%
printtokens	0	0.0%	0	0.0%
ccrypt	0	0.0%	3	4.0%
gzip	0	0.0%	9	2.9%
space	1	0.7%	1	0.7%
exif	2	1.1%	1	0.6%
bc	0	0.0%	0	0.0%
sed			0	0.0%
flex			0	0.0%
grep			0	0.0%
bash			29	0.4%
gcc				

Notes. Higher is better. Applications are ordered by size.

the CFG for the function. Functions with CFG of 25 nodes or fewer are categorized as *tiny*, functions whose CFG has between 25 and 100 nodes are categorized as *medium*, functions whose CFG has been 101 and 250 nodes are called *large*, and functions whose CFG has more than 250 nodes are called *jumbo*.

In Table 6, for each class of CFG size, we list the number of functions in the class (N), the average number of nodes and arcs in the CFG, the average number of rounds of cut generation, the number of initial ambiguous triangle inequalities (SC.2) added to the integer program (SC) and the total number of inequalities (SC.2) generated by our method, the average total number of branch and bound nodes required to solve the instances, the average time spent solving integer programs, and the average time spent separating inequalities.

To interpret the results of the table, it is necessary to explain specific details of our implementation. Crucial to this discussion is the order in which, for each fixed $d \in D$, the node pairs (α, β) are enumerated in Algorithm 1. We search for ambiguous triangles by fixing the maximum distance of α and β from a given desired node $d \in D$. We define the *depth* of a candidate node α as the minimum number of CFG arcs that must be traversed in any path from α to d. Similarly, the depth of β is the minimum number of arcs required to reach β from d. When enumerating (α, β) pairs at step 4 of Algorithm 1, we iterate over (maximum) depth values $\delta = 1, 2, \ldots$, and consider αd and $d\beta$ paths of length at most δ. For each desired node, it is possible to generate multiple ambiguous triangle inequalities with different (α, β) pairs. Our implementation will return up to $\kappa = 7$ violated ambiguous triangles for each desired node d. If we find ambiguous triangle inequalities at depth δ, all ambiguous triangle inequalities from this depth, establishing that the current candidate solution is not a coverage set, are added to the master integer program (SC) and separation is concluded. One benefit of performing enumeration in this manner is that the path π^1 will have length at most 2δ, resulting in constraint (SC.2) having a small support, hopefully dominating other ambiguous triangle inequalities with larger support.

Each integer program (SC) is initialized with a small number of constraints (SC.2). We generate an initial set of inequalities for the integer program by looking for one ambiguous triangle for each potential depth of α and β and for each desired node $d \in D$. As seen in Table 6, this can lead to a large number of initial triangles, but this does not negatively impact the solution time of the integer program.

Table 6. Average IP Performance

Class	N	CFG nodes	CFG arcs	No. of rounds	Initial no. of cuts	Total no. of cuts	B&B nodes	IP time (s)	Sep. time (s)
Tiny	14,474	7.7	10.5	0.4	5.6	6.5	0.4	0.0001	0.0008
Medium	3,176	47.0	74.1	1.7	71.7	80.6	4.4	0.0040	0.1409
Large	651	150.6	239.2	3.7	326.5	366.1	6.2	0.0114	9.0467
Jumbo	193	456.8	759.7	6.4	1,671.7	1,792.6	101.5	0.0902	504.9844

We experimented with an implementation where inequalities (SC.2) were added using Gurobi callback functions in the branch and bound tree (branch-and-cut) or simply adding the inequalities and resolving the integer program from scratch (cut-and-branch). We also experimented with an implementation where inequalities (SC.2) were generated by separating fractional candidate coverage sets $\bar{x}$. Surprisingly, the cut-and-branch implementation, separating only integer-valued candidate solutions $\bar{x}$, was the most efficient. At each iteration, the relaxed integer program (*SC*), with a subset of ambiguous triangle inequalities (SC.2), is solved to optimality, resulting in a candidate coverage set $\bar{x}$. Separation is done as described previously by enumerating potential (α, β) depths δ for each desired node $d \in D$, for fixed depths of $\delta = 1, 2, \ldots$, until inequalities violated by $\bar{x}$ are found or until the maximum depth is reached, at which point we know that $\bar{x}$ is an optimal coverage set. As seen in Table 6, in general, very few rounds of cut generation are necessary.

The most notable aspect of this table is the very small amount of time that is spent on average in solving the integer programs. For the larger instances, nearly all the time is spent in separation. The vast majority of separation time is taken in the final round to prove the candidate coverage solution is in fact feasible. The average CPU times for the last round of cut generation for the four instance families are 0.0006 seconds for tiny, 0.1206 seconds for medium, 8.667 seconds for large, and 496.9752 seconds for jumbo. Speeding up separation is an obvious important future research direction.

4.4. Discussion and Practical Implications

We improve on prior work both in the efficiency of our method and the quality of solutions. As noted in Section 4.1, our algorithm is often orders of magnitude faster than the prior locally optimal heuristic solution. Furthermore, developers using our approach now have the assurance of a certificate of optimality, which prior work could not provide.

When choosing three hours as the time-out cutoff, we do not mean to imply that developers would actually wait this long. Most of the applications we examined compiled in seconds without coverage instrumentation. Thus, although increases on the order of 100 times are unrealistic in many scenarios, increases on the order of 10 times may be tolerable. As made especially evident in Figure 5b, our new method makes optimized instrumentation practical for a much larger set of applications.

Even if for certain applications the compile times for our exact instrumentation method are too long to be used in an online setting, our new approach has significant potential to assess the quality of other methods for the customized coverage probing problem. Specifically, as our method provides the most efficient mechanism for computing optimal coverage plans, it could be used in an offline setting to gauge the performance (in terms of additional overhead) of faster, inexact methods.

5. Conclusion

Program coverage data are pervasive and useful in many areas of computer programming debugging analysis. Many real-world scenarios demand customized coverage instrumentation, where only a portion of a program requires coverage or can be instrumented for program coverage. By using a characterization of a coverage set in terms of subgraphs known as ambiguous triangles, we present a new algorithm for the *customized coverage probing problem*. Using the characterization, we formulate the problem as a set covering problem and leverage state-of-the-art commercial integer programming software to find optimal coverage plans. Our technique produces exact (i.e., optimal) solutions multiple orders of magnitude faster than any previous exact method. In fact, our method produces exact coverage plans faster than previous methods in the literature whose plans come with no guarantee of the coverage plan quality. The new method makes optimized instrumentation practical for a much wider set of applications.

Acknowledgments

Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

Agrawal H (1994) Dominators, super blocks, and program coverage. *Proc. 21st ACM SIGPLAN-SIGACT Sympos. on Principles of Programming Languages* (ACM, New York), 25–34.

Agrawal H (1999) Efficient coverage testing using global dominator graphs. Griswold WG, Horwitz S, eds. *Proc. SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engrg.* (ACM, New York), 11–20.

Ball T, Larus JR (1994) Optimally profiling and tracing programs. *ACM Trans. Programming Language Systems* 16(4):1319–1360.

Chilakamarri K, Elbaum SG (2006) Leveraging disposable instrumentation to reduce coverage collection overhead. *Software Testing Verify Reliability* 16(4):267–288.

Dallmeier V, Lindig C, Zeller A (2005) Lightweight defect localization for Java. Proc. *19th Eur. Conf. of* Object-Oriented Programming (Springer, Berlin), 528–550.

Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engrg.* 10(4):405–435.

GAMS Development Corporation (2018) General algebraic modeling system (GAMS) release 25.1.2. Accessed November 15, 2018, http://www.gams.com/.

Gurobi Optimization LLC (2018) Gurobi optimizer reference manual. http://www.gurobi.com.

Kasikci B, Ball T, Candea G, Erickson J, Musuvathi M (2014) Efficient tracing of cold code via bias-free sampling. *Proc. USENIX Annual Tech. Conf.* (USENIX Association, Berkley, CA), 243–254.

Knuth DE (1968) *The Art of Computer Programming, Volume I: Fundamental Algorithms* (Addison-Wesley, Boston).

Knuth DE, Stevenson FR (1973) Optimal measurement points for program frequency counts. *BIT Numerical Mathematics* 13(3):313–322.

Lattner C, Adve V (2004) LLVM: A compilation framework for lifelong program analysis & transformation. *Proc. Internat. Sympos. on Code Generation and Optim.* (IEEE, Piscataway, NJ).

Liblit B (2014) The cooperative bug isolation project. http://research.cs.wisc.edu/cbi/.

LLVM Project (2018) LLVM block frequency terminology. https://llvm.org/docs/BlockFrequencyTerminology.html.

Michini C, Ohmann P, Liblit B, Linderoth J (2022) A set covering approach to customized coverage instrumentation. http://dx.doi.org/10.1287/ijoc.2021.0349.cd, https://github.com/INFORMSJoC/2021.0349.

Misurda J, Childers BR, Soffa ML (2011) Jazz2: A flexible and extensible framework for structural testing in a Java VM. *Proc. 9th Internat. Conf. on Principles and Practice of Programming in Java* (ACM, New York), 81–90.

Misurda J, Clause JA, Reed JL, Childers BR, Soffa ML (2005) Demand-driven structural testing with dynamic instrumentation. Roman G, Griswold WG, Nuseibeh B, eds. *Proc. 27th Internat. Conf. on Software Engrg.* (ACM, New York), 156–165.

Mitchell JE (2002) Branch-and-cut algorithms for combinatorial optimization problems. Pardalos PM, Resende MGC, eds. *Handbook of Applied Optimization* (Oxford University Press, Oxford, UK), 65–77.

Nishimatsu A, Jihira M, Kusumoto S, Inoue K (1999) Call-mark slicing: An efficient and economical way of reducing slice. *Proc. 21st Internat. Conf. on Software Engrg.* (ACM, New York), 422–431.

Ohmann P, Liblit B (2013) Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *28th IEEE/ACM Internat. Conf. on Automated Software Engrg.* (IEEE, Piscataway, NJ), 378–388.

Ohmann P, Liblit B (2017) Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engrg.* 24(4):865–904.

Ohmann P, Brooks A, D'Antoni L, Liblit B (2017) Control-flow recovery from partial failure reports. Vechev M, ed. *Proc. 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation* (ACM, New York), 390–405.

Ohmann P, Brown DB, Neelakandan N, Linderoth J, Liblit B (2016a) Encoding optimal customized coverage instrumentation. Technical Report TR1836, Department of Computer Sciences, University of Wisconsin-Madison, Madison.

Ohmann P, Brown DB, Neelakandan N, Linderoth J, Liblit B (2016b) Optimizing customized program coverage. *Proc. 31st IEEE/ACM Internat. Conf. on Automated Software Engrg.* (ACM, New York), 27–38.

Orso A, Liang D, Harrold MJ, Lipton R (2002) Gamma system: Continuous evolution of software after deployment. *Proc. ACM SIGSOFT Internat. Sympos. on Software Testing and Analysis* (ACM, New York), 65–69.

Pankumhang T, Rutherford M (2015) Iterative instrumentation for code coverage in time-sensitive systems. *Proc. 8th IEEE Internat. Conf. on Software Testing, Verification and Validation* (IEEE, Piscataway, NJ), 1–10.

Pavlopoulou C, Young M (1999) Residual test coverage monitoring. *Proc. Internat. Conf. on Software Engrg.* (ACM, New York), 277–284.

Reiss SP, Renieris M (2001) Encoding program executions. *Proc. 23rd Internat. Conf. on Software Engrg.* (IEEE, Piscataway, NJ), 221–230.

Rothermel G, Elbaum S, Kinneer A, Do H (2006) Software–artifact infrastructure repository. http://sir.unl.edu/portal/.

Santelices RA, Jones JA, Yu Y, Harrold MJ (2009) Lightweight fault-localization using multiple coverage types. *Proc. 31st Internat. Conf. on Software Engrg.* (IEEE, Piscataway, NJ), 56–66.

Tikir MM, Hollingsworth JK (2002) Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes*, vol. 27, 86–96.

Tikir MM, Hollingsworth JK (2005) Efficient online computation of statement coverage. *J. System Software* 78(2):146–165.

Wu R, Xiao X, Cheung S, Zhang H, Zhang C (2016) Casper: An efficient approach to call trace collection. *Proc. 43rd Annual ACM SIGPLAN-SIGACT Sympos. on Principles of Programming Languages* (ACM, New York), 678–690.