Finding Fix Locations for CFL-Reachability Analyses via Minimum Cuts

Andrei Marian Dan[1](✉), Manu Sridharan[2], Satish Chandra[2], Jean-Baptiste Jeannin[2], and Martin Vechev[1]

[1] Department of Computer Science, ETH Zurich, Zürich, Switzerland
{andrei.dan,martin.vechev}@inf.ethz.ch

[2] Samsung Research America, Mountain View, USA
manu@sridharan.net, schandra@schandra.org, jb.jeannin@gmail.com

Abstract. Static analysis tools are increasingly important for ensuring code quality. Ideally, all warnings from a static analysis would be addressed, but the volume of warnings and false positives usually makes this effort prohibitive. We present techniques for finding *fix locations*, a small set of program locations where fixes can be applied to address all static analysis warnings. We focus on analyses expressible as context-free-language reachability, where a set of fix locations is naturally expressed as a min-cut of the CFL graph. We show, surprisingly, that computing such a CFL min-cut is NP-hard. We then phrase the problem of finding CFL min-cuts as an optimization problem which allows us to trade-off the size of the cut vs. the preservation of computed information. We then show how to solve the optimization problem via a MaxSAT encoding.

Our evaluation shows that we compute fix location sets that are significantly smaller than both the number of warnings and, in the case of a true CFL min-cut, the fix location sets from a normal min-cut.

1 Introduction

Static analysis tools are playing an increasingly important role in ensuring code quality of real-world software. They are able to detect a wide variety of defects, from low-level memory errors to violations of typestate properties. In an ideal setting, code would be made "clean" with respect to these tools: all warnings would be addressed either with a fix of the underlying defect, or a combination of code restructuring and annotations to show the tool no defect exists.

Unfortunately, this ideal is rarely achieved in practice. One issue is that the volume of warnings emitted by these tools can be large, with each issue potentially requiring significant time to inspect and understand. Further, as is inevitable in static analysis, many of the warnings are false positives, and expending significant effort in annotating and restructuring code to avoid false positive reports may overburden the developer.

M. Sridharan—Currently affiliated with Uber.

S. Chandra—Currently affiliated with Facebook.

R. Majumdar and V. Kunčak (Eds.): CAV 2017, Part II, LNCS 10427, pp. 521–541, 2017.
DOI: 10.1007/978-3-319-63390-9_27

This paper presents a technique for computing a small set of *fix locations*, such that code changes at or near those locations can address *all* warnings emitted by a static analysis. Previous work (e.g., Fink et al. [7]) has observed that many static analysis warnings (particularly false positives) can stem from a small amount of buggy or difficult-to-analyze code that leads to cascading warnings. Producing a small set of fix locations would help pinpoint this crucial code, which could significantly ease the process of addressing a large set of warnings. Similarly, recent work provides succinct explanations for type inference errors [14,17,22].

A trend in large companies is to only allow committing code that is warning free. It is deemed acceptable that false positives may need a small refactoring, or an annotation to silence the analyzer. Addressing all warnings results in a program that is error free with respect to the analyser that is used. Note that the user can potentially insert an assumption, instead of modifying the fix location code. This way, the analyzer gains precision and eliminates false positives.

We focus on analyses expressible as context-free-language reachability (CFL-reachability) problems [18]. The CFL-reachability framework can express a wide variety of program analyses, including points-to analysis and interprocedural dataflow analysis [18]. In a CFL-reachability-based analysis, an error report typically corresponds to the existence of corresponding (CFL) paths in a graph representing the program and analysis problem. Hence, the problem of computing fix locations maps naturally to the problem of computing a cut of the graph that removes the paths corresponding to the errors. Intuitively, the fix locations are the program locations corresponding to the edges in this cut.

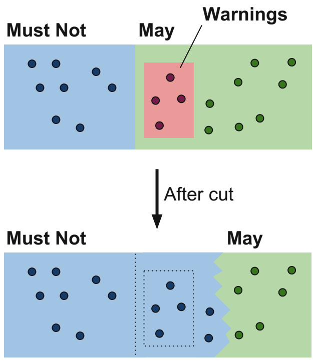


Fig. 1. A cut causes "may" facts that trigger warnings to become "must-not" facts.

Figure 1 gives an overview of our goal. A CFL-reachability analysis typically computes a set of "may" facts that hold for the input program, corresponding to CFL paths in the graph. The absence of a path then corresponds to a complementary "must-not" fact, and a cut of the graph changes some facts from "may" to "must-not" (while preserving "must-not" facts). Certain "may" facts trigger analysis warnings, and our goal is to compute a small cut that turns all such "may" facts into "must-not" facts, thereby eliminating all warnings.

A promising component of computing fix locations would be to compute a *CFL min-cut* of a graph (we will discuss shortly why the CFL min-cut may not be the optimal solution in all cases). A CFL min-cut is distinguished from a standard graph min-cut in that it need only cut CFL paths in the graph. For a CFL-reachability problem, the CFL min-cut may be smaller than a standard min-cut that ignores edge labels (see Sect. 2). As CFL-reachability is computable in polynomial time [18], a natural question is:

Can a CFL min-cut be computed in polynomial time?

A key result of this work is that (perhaps surprisingly) it is not possible to compute a CFL-min-cut in polynomial time for an arbitrary CFL language.[1] Towards that, we prove that computing the CFL min-cut problem for the restricted language of balanced parentheses is NP-hard. Moreover, we prove that for a language with multiple types of parentheses, computing the CFL min-cut on graphs resulting from applying an IFDS null analysis on programs is also NP-hard. We expect that similar techniques will work for other realistic analyses like pointer analysis.

CFL-reachability problems correspond to Datalog programs consisting entirely of chain rules [18]. For such programs, the CFL min-cut problem maps to finding the smallest set of input facts that, when removed, make a specified set of output facts underivable. Our hardness result implies that finding this smallest set for Datalog chain programs is also NP-hard.

Beyond computational hardness, a CFL min-cut may not always provide the most desirable set of fix locations, as it does not consider the degree of unnecessary change to the (abstract) program behavior. Notice that in Fig. 1, the cut caused some "may" facts that did not correspond to warnings to become "must-not" facts. If this set of needlessly-transformed "may" facts becomes too large, the fix locations corresponding to a CFL min-cut may not be desirable (see Sect. 2 for an example). On the other hand, an approach that strictly minimizes the number of transformed "may" facts (while still fixing all warnings) may yield an excessively large set of fix locations. Hence, we require a technique that allows for a tunable tradeoff between these two factors, analogous to tradeoffs between syntactic and semantic change in program repair [6].

To achieve this flexible tradeoff, we first define an *abstract distance* that measures how many of the "may" facts are transformed for a given cut. Second, we formulate the problem of computing the best cut of a CFL graph as an optimization problem. In the process, we show how this formulation can be constructed with a simple instrumentation of an optimized IFDS solver, requiring no changes to analysis clients using the solver. The optimization formulation combines the goal to minimize the cut size (formulated for IFDS problems based on the Reps-Horwitz-Sagiv tabulation algorithm [19]) with the minimization of the abstract distance. Finally, we solve the optimization problem via an encoding to a weighted MaxSAT formula. MaxSAT solvers have improved dramatically in recent years, and the framework of weighted MaxSAT allows us to concisely express the relevant tradeoffs in our problem.

We have implemented our technique and evaluated it on a realistic null dereference analysis for Java programs. We found that our proposed fix location sets were often smaller than the total number of warnings (up to 11 times smaller). Moreover, we discover that the size of a CFL min-cut is significantly smaller (up to 50%) than the size of a normal min-cut. We also evaluated the tradeoffs between cut vs. abstract distance sizes. To our best knowledge, our system is the first to be able to suggest a minimal number of fix locations for errors found by a static analysis.

[1] Assuming $P \neq NP$.

Contributions. This paper presents the following contributions:

– We prove that computing a CFL min-cut is NP-hard, even for a simple balanced-parentheses language and acyclic graphs (Sect. 3).
– We define the notion of abstract distance and define an optimization problem between the CFL min-cut vs. abstract distance sizes and show how to solve this problem via (MaxSAT Sect. 4).
– We evaluate our approach on an IFDS-based null-pointer analysis on several benchmark programs, showing the benefits of CFL min-cuts and exploring the tradeoffs involved (Sect. 5).

2 Overview

In this section we provide an overview of our approach on a motivating program analysis example, illustrating the usefulness of CFL min-cuts and the tradeoffs involved. We start with some core definitions.

2.1 CFL Reachability

Let us consider the context-free grammar of balanced parenthesis $H = \{\{S\}, \Sigma, R, S\}$, where $\Sigma = \{(,), \epsilon\}$ is a set of symbols; S is the non-terminal starting variable; and $R = \{S \rightarrow SS, S \rightarrow (S), S \rightarrow \epsilon\}$ is the set of production rules. Let $G = (V, E)$ be a graph with vertices V and edges E, each edge labelled with a symbol from Σ; Fig. 2 gives an example.

Given a graph G and a context-free grammar H, CFL reachability from a source vertex u to a destination vertex v is conceptually established as follows. Given a path p, we define word W_p as the concatenation of the edge labels of p. Then, v is CFL-reachable from u iff there exists a path p from u to v s.t. W_p is in the language defined by H. In the graph of Fig. 2, only vertices d and f are CFL-reachable from vertex a.

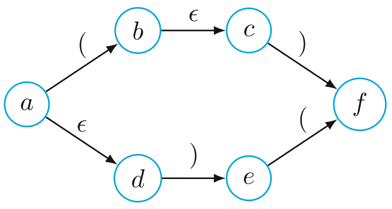

$$\begin{tikzcd}
 & b \arrow[r, "\epsilon"] & c \arrow[dr, ")"] & \\
a \arrow[ur, "("] \arrow[dr, "\epsilon"'] & & & f \\
 & d \arrow[r, ")"] & e \arrow[ur, "("'] &
\end{tikzcd}$$

Fig. 2. Edges of the graph are labelled with symbols from the context-free grammar of balanced parenthesis H.

2.2 CFL Min-Cuts

Min-Cut. A standard min-cut between two nodes a and f of a graph is defined as a minimal set of edges M such that if all edges in M are removed, no path remains from a to f. For Fig. 2, any min-cut between nodes a and f must have size 2, as it must cut the path $a \rightarrow b \rightarrow c \rightarrow f$ and and the path $a \rightarrow d \rightarrow e \rightarrow f$. Computing a min-cut is polynomial in the size of the graph and several algorithms have been developed [8,21].

CFL Min-Cut. Similarly, a CFL min-cut between two nodes a and f of a graph is defined as a minimal-size set of edges that, when removed, ensure no CFL path exists from a to f. Any min-cut is also a CFL cut. However, it is possible that a min-cut is not a CFL min-cut. For Fig. 2, the only CFL path from a to f is $a \rightarrow b \rightarrow c \rightarrow f$. ($a \rightarrow d \rightarrow e \rightarrow f$ is not a CFL path because the word "ϵ)(" has mismatched parentheses.) Therefore, for this example the CFL min-cuts are of size one ($\{ab\}$, $\{bc\}$ or $\{cf\}$). Though computing a standard min-cut has a polynomial time complexity, in Sect. 3 we prove that computing a CFL min-cut is NP-hard even for a simple balanced parentheses language.

2.3 Min-Cuts for Program Analysis

Here, we detail how our technique applies CFL min-cuts to finding fix locations for a static analysis, focusing on an analysis expressed atop the well-known IFDS framework [19].

Figure 3 shows an example program. We use . . . to omit parts of the program, for simplification purposes. The program consists of a class with three static methods (`f`, `g`, `main`) and one static field (`a`).

```
1: class Hello {
2:  static Object a;
3:  static void f() { ... }
4:  static void g() { ... }

5:  public static
      void main(String[] args) {
6:     a = null;
7:     if (...) { f(); }
8:     else { g(); }

9:     a = new Object();
10:    if (...) { a = null; }
11:    ...
12:    if (...) {f(); a.toString();}
13:    else {g(); a.getClass(); }}}
```

Fig. 3. On this example, a null dereference analysis reports 2 warnings at lines 12 and 13. Depending on *if* conditions, these could be bugs or false positives.

Null Dereference IFDS Analysis. A null dereference analysis for Java checks that field accesses and method invocations do not dereference `null`, causing a `NullPointerException`. Such an analysis can be encoded in the IFDS (interprocedural, finite, distributive, subset) framework [19]. IFDS problems can be solved by computing CFL-reachability over an exploded super-graph representation of the program. We briefly describe the technique here; see Reps et al. for further details [19].

Super-Graph. Starting from the input program P, the analysis constructs a *supergraph*, a type of interprocedural control-flow graph. The supergraph nodes for the example in Fig. 3 are the boxes in Fig. 4. In addition to the usual nodes of the intra-procedural flow graphs, each procedure has a distinguished entry and exit node (e.g., s_f and e_f). Each function call is represented by a call and return node at the caller (e.g., n_3 and n_4), and by edges from the call node to the callee entry and from the callee exit to the return node. A detailed description of supergraphs is available in [19].

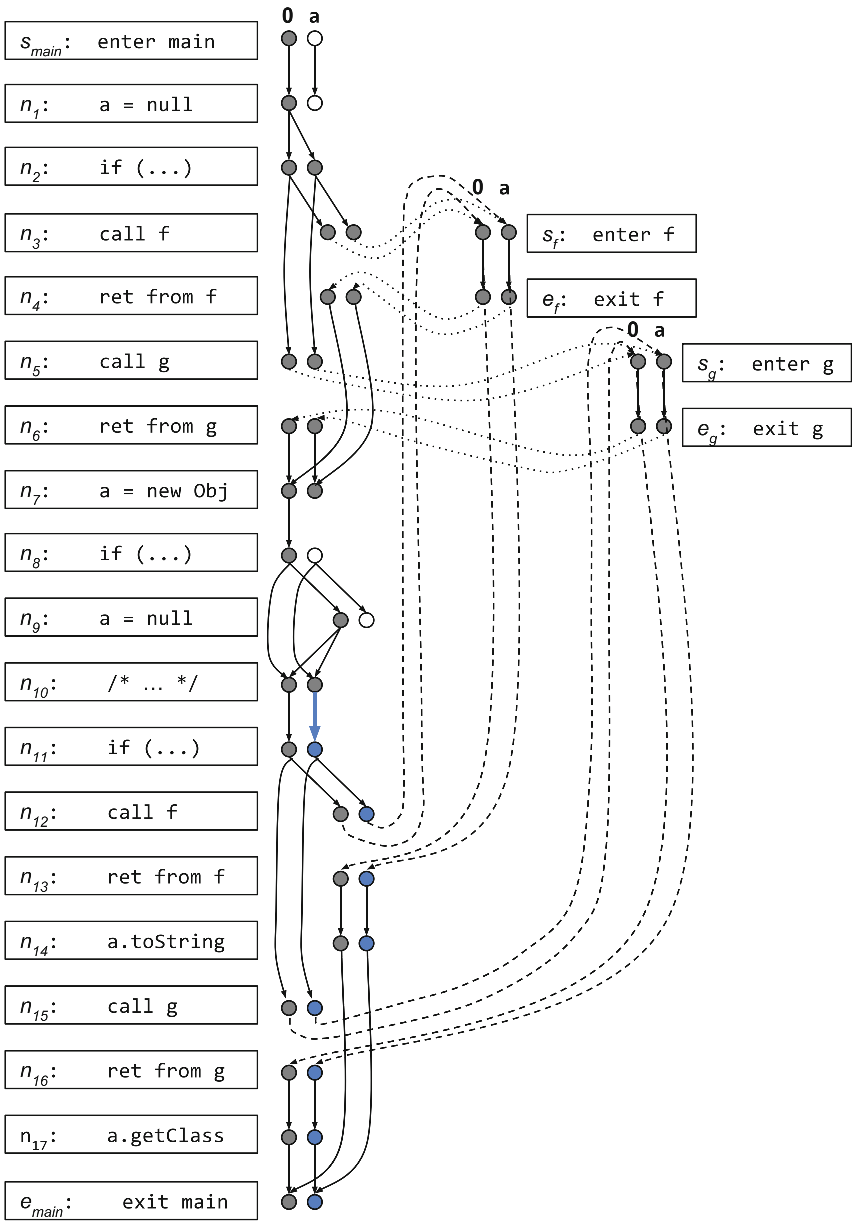


Fig. 4. Exploded super-graph for the program in Fig. 3. The emphasized blue edge represents a CFL min-cut in this graph such that the fact a is not reachable at statements `a.toString` and `a.getClass`. The white circles represent facts that are not reachable from the fact **0** of the main procedure's entry node. The grey and blue circles represent reachable facts from the fact **0** of the main procedure's entry node. The blue nodes become CFL unreachable if we remove the blue edge. A reachable node a means that the variable `a` may be null at that program point. (Color figure online)

Exploded Super-Graph. Given the supergraph, the IFDS analysis constructs an *exploded supergraph*, with nodes representing facts at program points and edges representing control flow and transfer functions. The exploded supergraph for null dereference analysis of Fig. 3 is shown in Fig. 4. Here, the analysis only needs to track nullness of global variable `a`. So, for each node in the supergraph, the exploded supergraph has two nodes: a $\mathbf{0}$ node and an $\mathbf{a}$ node. (The number of nodes depends on the number of facts in the abstract domain.) The $\mathbf{0}$ nodes are required for well-formedness, and the $\mathbf{a}$ node tracks whether the variable `a` *may be null*. The solid edges represent the transfer function for the corresponding program statement. E.g. edges $\langle n_1, \mathbf{0}\rangle \rightarrow \langle n_2, \mathbf{0}\rangle$ and $\langle n_1, \mathbf{0}\rangle \rightarrow \langle n_2, \mathbf{a}\rangle$ show that statement `a = null` "gens" a fact that `a` may be null. The dotted and dashed edges correspond to function calls and returns. We use dotted edges for the first (in program order) calls and return for functions `f` and `g` (e.g., $\langle n_3, \mathbf{0}\rangle \dashrightarrow \langle s_f, \mathbf{0}\rangle$ and $\langle e_f, \mathbf{0}\rangle \dashrightarrow \langle n_4, \mathbf{0}\rangle$) and we use dashed edges for the second calls to these functions. Any *realizable path* through the graph must exit a function call via the same type of edge (dotted or dashed) as it entered. Checking for realizable paths is equivalent to computing CFL reachability with a parenthesis labeling for call and return edges. Each call site gets a unique pair of open and close parentheses, e.g., $(^1_f$ for n_3 to s_f edges and $)^1_f$ for e_f to n_4 edges, and the language ensures any CFL path has matched parentheses.

Warnings as CFL-Reachability. The IFDS analysis computes CFL-reachability of nodes in the exploded super-graph using the tabulation algorithm [19]. In our example, if an $\mathbf{a}$ node is CFL-reachable from node $\langle s_{main}, \mathbf{0}\rangle$, then variable `a` may be null at the corresponding program point. In Fig. 4, all of the grey and blue nodes are CFL-reachable from $\langle s_{main}, \mathbf{0}\rangle$, while the white nodes are not. In particular, the nodes $\langle n_{14}, \mathbf{a}\rangle$ and $\langle n_{17}, \mathbf{a}\rangle$ corresponding to the statements `a.toString` and `a.getClass` are CFL-reachable (blue color). The analysis thus triggers two warnings, indicating possible null dereferences at lines 12 and 13 of Fig. 3. These warnings could be real bugs or false positives, depending on the actual conditions in the if statements.

Fix Locations. To remove all warnings, the corresponding exploded supergraph nodes must be made unreachable from the $\mathbf{0}$ entry node. For our example, making an $\mathbf{a}$ node unreachable corresponds to proving a *must not* be null at the corresponding program point. A CFL min-cut gives a minimal set of edges to make the warning nodes unreachable, and the corresponding program locations are the suggested fix locations. In Fig. 4, the blue solid edge $\langle n_{10}, \mathbf{a}\rangle \rightarrow \langle n_{11}, \mathbf{a}\rangle$ is a CFL min-cut between node $\langle s_{main}, \mathbf{0}\rangle$ and the warning nodes ($\langle n_{14}, \mathbf{a}\rangle$ and $\langle n_{17}, \mathbf{a}\rangle$). If we remove this edge, then all blue nodes become CFL-unreachable from node $\langle s_{main}, \mathbf{0}\rangle$.

Min-Cut vs. CFL Min-Cut. Notice that if we consider regular reachability instead of CFL reachability, then the nodes corresponding to warnings are still reachable from $\langle s_{main}, \mathbf{0}\rangle$ in Fig. 4, even after removing the CFL min-cut (blue edge). For example, node $\langle n_{14}, \mathbf{a}\rangle$ can be reached through the path: $\langle s_{main}, \mathbf{0}\rangle \rightarrow \langle n_1, \mathbf{0}\rangle \rightarrow$

$\langle n_2, \mathbf{a}\rangle \longrightarrow \langle n_3, \mathbf{a}\rangle \cdots\!\!\!\rightarrow \langle s_f, \mathbf{a}\rangle \longrightarrow \langle e_f, \mathbf{a}\rangle \dashrightarrow \langle n_{13}, \mathbf{a}\rangle \longrightarrow \langle n_{14}, \mathbf{a}\rangle$. However, this is not a realizable program path because the call edge $\langle n_3, \mathbf{a}\rangle \cdots\!\!\!\rightarrow \langle s_f, \mathbf{a}\rangle$ does not correspond to the return edge $\langle e_f, \mathbf{a}\rangle \dashrightarrow \langle n_{13}, \mathbf{a}\rangle$. For this example, a regular min-cut has size two.[2] Hence, if only regular min-cuts were considered, the number of fix locations could be unnecessarily higher than a CFL min-cut, leading to needless fixing effort from the programmer.

CFL Min-Cut Selection. In general, a graph can have several CFL min-cuts. We choose to select the cut that preserves the most reachability facts in the exploded super-graph. Choosing in this manner converts as few may facts to must-not facts as possible (see Fig. 1), retaining as much of the original safe behavior as possible. The blue edge CFL min-cut in Fig. 4 only makes the eight blue nodes in the graph CFL unreachable. The actual fix can be implemented by simply introducing a line `a = new Object()` (or a more realistic fix) between lines 11 and 12 of Fig. 3. Note that suggesting a concrete *repair* is future work and out of scope for this paper.

Consider another possible CFL min-cut, $\langle n_9, \mathbf{0}\rangle$ to $\langle n_{10}, \mathbf{a}\rangle$. Since node n_{10} can correspond to a large code fragment (that potentially does not write to variable `a`), this alternate cut could correspond to a much more disruptive change. Further trade-offs between preserving semantics and min-cut sizes are presented in Sect. 4. Next, we study the complexity of finding a CFL min-cut.

3 CFL Min-Cut Complexity

The time complexity for CFL-reachability is $O(|\Sigma|^3 n^3)$ when using dynamic programming, where n is the number of graph vertices and $|\Sigma|$ the size of the CFL [16], compared to normal reachability which is $O(n)$ (graph traversal). Computing a normal min-cut has the complexity $O(mn \times log(n^2/m))$ (m is the number of graph edges) [8], and one might expect only a polynomial additional cost for computing a CFL min-cut. But, here we show that computing a CFL min-cut is *NP-hard* even for a restricted version of the problem (Table 1).

Table 1. Time complexity for the reachability and min-cut problems.

	Reachability	Min-cut
Normal	$O(n)$	$O(mn \times log(n^2/m))$ [8]
CFL	$O(\|\Sigma\|^3 n^3)$ [16]	***NP-hard*** (this work)

When considering this problem, we focused on balanced parentheses languages (as defined in Sect. 2.1), as most popular and important CFL-reachability-based program analyses we know of use balanced parentheses. Theorem 1 shows

[2] Note that an edge between two $\mathbf{0}$ nodes (e.g. $\langle s_{main}, \mathbf{0}\rangle \longrightarrow \langle n_1, \mathbf{0}\rangle$) cannot be cut as such an edge is required for the well-formedness of the exploded supergraph [19].

NP-hardness of CFL min-cut for the restricted language of balanced parentheses. Theorem 1 also shows that the hardness result holds *even for acyclic graphs* and it is our best result in terms of the most restricted CFL and graph structure. In practice, when applying a CFL-reachability analysis to graphs from real programs, the resulting graphs may not allow for the structure used in the proof of Theorem 1. Still, most popular CFL-reachability analyses allow for multiple parentheses types and we show an example in Theorem 2 that for graphs resulting from an IFDS null analysis the CFL min-cut problem is NP-hard. We believe that similar proof techniques can be extended to other realistic analyses like pointer analysis.

Preliminaries. Let L be a context-free language over the alphabet Σ and a directed graph $G = (V, E)$ such that the edges in E are labelled with members of Σ ($label : E \rightarrow \Sigma$). Henceforth CFL paths, CFL reachability and CFL min-cuts are understood with respect to the language L. Let $s \in V$ be the source node and $t \in V$ the target node. Each path from s to t in G defines a word over the alphabet Σ. A path is a CFL path if the word corresponding to the path is a member of L. We consider directed graphs. The s, t CFL min-cut problem is defined as finding the smallest subset $E' \subseteq E$ such that there exists no CFL path from s to t in the graph $G' = (V, E \setminus E')$.

Theorem 1. *If L is the language of balanced parentheses, finding the s, t CFL min-cut in acyclic directed graphs with edges labelled by symbols of L is NP-hard.*

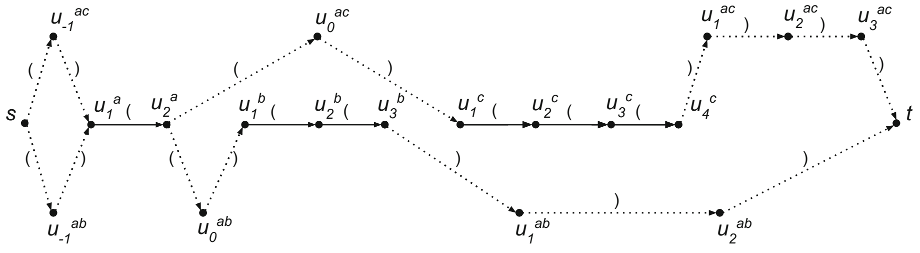


Fig. 5. Acyclic labelled graph for which finding the CFL min-cut between s and t leads to finding the vertex cover for the graph with three vertices and two edges $(\{a, b, c\}, \{ab, ac\})$.

Proof. We reduce the vertex cover problem to s, t CFL min-cut in acyclic graphs. Given an undirected graph $G = (V, E)$, the vertex cover problem [10] computes the smallest set $C \subseteq V$ such that $\forall uv \in E \Rightarrow u \in C \vee v \in C$. We map each element of V to a number in $\{1 \ldots n\}$, where $|V| = n$ ($id : V \rightarrow \{1 \ldots n\}$). Notice that if initially G contains $vv \in E$, where $v \in V$, then v has to be in the vertex cover. Therefore, we add v to the solution and remove all the edges containing v from E. From now we can assume that E does not contain vv edges, $v \in V$.

Starting from the graph (V, E), we construct the new graph (V', E') on which we compute the s, t CFL min-cut. First, we introduce two distinguished vertices $\{s, t\} \subseteq V'$. Then for each vertex $v \in V$ we create $id(v) + 1$ vertices in V': $\{u_1^v, \ldots, u_{id(v)+1}^v\}$. Additionally, we add $id(v)$ edges to E', labelled with the symbol "(": $\{u_1^v u_2^v, \ldots, u_{id(v)}^v u_{id(v)+1}^v\}$.

For each edge $vv' \in E$, ordered (without loss of generality) such that $id(v) < id(v')$, we first create 2 new vertices in V': $\{u_{-1}^{vv'}, u_0^{vv'}\}$, and 4 new edges in E': $su_{-1}^{vv'}$, $u_{id(v)+1}^v u_0^{vv'}$, labelled with "(" and $u_{-1}^{vv'} u_1^v$, $u_0^{vv'} u_1^{v'}$ labelled with ")".

Finally, for each edge $vv' \in E$, we introduce $id(v) + id(v') - 1$ vertices $\{u_1^{vv'}, \ldots u_{id(v)+id(v')-1}^{vv'}\}$ and $id(v) + id(v')$ edges labelled by ")":

$$\{u_{id(v')+1}^{v'} u_1^{vv'}, u_1^{vv'} u_2^{vv'}, \ldots, u_{id(v)+id(v')-2}^{vv'} u_{id(v)+id(v')-1}^{vv'}, u_{id(v)+id(v')-1}^{vv'} t\}$$

Overall, if $|E| = m$, the number of vertices in V' is $O(n^2 + mn)$ and the number of edges in E' is $O(n^2 + mn)$.

We illustrate the construction for a small graph $(\{a, b, c\}, \{ab, ac\})$ for which we want to find the vertex cover. First, we associate 1 to vertex a, 2 to b and 3 to c. Next, following the steps described above, we construct the graph shown in Fig. 5.

For each edge $vv' \in E$, there exists a corresponding CFL path from s to t in the graph (G', E'). This path contains all the $id(v) + id(v')$ edges corresponding to v and v'. Additionally, the CFL path will contain the sub-CFL-paths of length 2 between s and the first vertex corresponding to v (u_1^v) and from the last vertex corresponding to v ($u_{id(v)+1}^v$) to the first vertex corresponding to v' ($u_1^{v'}$).

There exist additional CFL paths from s to t. Given $vv' \in E$ and $v_1, v_2 \in V$ such that $v_1 v_2 \in E$ and $v_2 v' \in E$ and $id(v_1) + id(v_2) = id(v)$, there is an CFL path from s to t containing the edges added for the vertices v_1, v_2 and v'. An important observation is that any cut for the CFL paths corresponding to edges in E will also be a cut for the additional CFL paths described above. Therefore, the additional CFL paths do not increase the size of the s, t CFL min-cut.

Finding an s, t CFL min-cut in this newly constructed graph is equivalent to finding a vertex cover in the original graph. We show how to obtain the vertex cover given an s, t CFL min-cut $M = \{e_1, \ldots, e_k\}$.

The first step is to transform M such that it contains only edges of type $u_i^v u_{i+1}^v$, where $v \in V$ and $i < id(v) + 1$.

If $e \in M$ is of type $u_i^{vv'} u_{i+1}^{vv'}$, we replace it with $u_{id(v')}^{v'} u_{id(v')+1}^{v'}$. It is impossible that the new edge already exists in M, because all CFL paths that contain $u_i^{vv'} u_{i+1}^{vv'}$ also contain $u_{id(v')}^{v'} u_{id(v')+1}^{v'}$ and this would contradict the minimality of the cut size. Additionally, M remains an s, t CFL cut and has the same size.

If e is of type $su_{-1}^{vv'}$, $u_{-1}^{vv'} u_1^v$, $u_{id(v)+1}^v u_0^{vv'}$ or $u_0^{vv'} u_1^{v'}$ then we replace it with the edge $u_{id(v)}^v u_{id(v)+1}^v$. Similarly to the previous case, all the CFL paths that contain e also contain its replacement ($u_{id(v)}^v u_{id(v)+1}^v$).

Next, to each edge $u_i^v u_{i+1}^v$ in the s, t CFL min-cut corresponds the vertex v in the cover set. For the example in Fig. 5, the CFL min-cut is edge $u_1^a u_2^a$, which corresponds to $\{a\}$ as the result for the vertex cover problem for the graph $(\{a, b, c\}, \{ab, ac\})$.

The s, t CFL min-cut contains at least one edge from each CFL path from s to t. This implies that the vertex cover will contain at least one vertex of each edge in E. The minimality of the cut implies the minimality of the vertex cover. □

Theorem 2. *If L is the language of balanced multiple types of parentheses, finding the s, t CFL min-cut in exploded supergraphs resulted from applying a CFL analysis is NP-hard, considering that the edges between two* **0** *nodes cannot be part of the cut.*

Proof. We prove this theorem by reduction from the vertex cover problem. The proof has two steps. First, given an undirected graph $G = (V, E)$ for which we want to compute the vertex cover, we construct a program P. Second, given a null analysis like the one in Sect. 2, we show that finding a CFL min-cut of the exploded super-graph of program P implies finding a vertex cover for graph G.

Constructing the Program P. Let $n = |V|$ be the number of vertices in the graph G. The program P has a variable x of type Obj and $n + 1$ methods. We assume the class Obj declares a method $f()$. For each vertice $u \in V$ we declare in P a method $m_u()$ that does not modify x. Additionally, we introduce a method *prog* that has a local integer i, initialized to a random value between 1 and m, where $m = |E|$ is the number of edges in G. The method *prog* contains a switch statement that takes as argument i and has m cases, one for each edge in G. For the case corresponding to the edge $uv \in E$, the variable x is set to *null*, then the methods $m_u()$ and $m_v()$ are invoked and the variable x is dereferenced by invoking the function $x.f()$. Finally, each case ends with a *break* statement. For example, given the small graph $(\{a, b, c\}, \{ab, ac\})$, we construct the following program:

Exploded Supergraph Using the Null Analysis. Next, consider the null analysis used in Sect. 2. For each node in the supergraph of program P, the exploded super-graph contains two nodes: a **0** node and an **x** node (meaning x may be null at that program point). Initially, there exists one CFL path from the **0** node at the entry in method *prog* to each dereference $x.f()$ in P (exactly one CFL path for each edge in G). Each path contains a prefix of edges between **0** nodes, and a suffix of edges between **x** nodes. The edge between a **0** and an **x** node corresponds to the statements `x = null` that precede the dereference. Since the cut may not contain edges between two **0** nodes, the CFL min-cut will contain edges that are part of the suffixes of each CFL path that leads to a dereference of x or edges between a **0** and an **x** node. For a CFL min-cut, we replace the edges that are not between **x** nodes inside one of the methods m_u, for $u \in V$, with edges between nodes inside one of the m_u functions. For instance, if the CFL cut contains an edge between a **0** and an **x** node, we will replace it with an edge between two **x** nodes inside the first method that is called after the corresponding `x = null` statement. This does not increase the size of the cut.

```
class P {
 Obj x = new Obj();
 void m_a() { ... }
 void m_b() { ... }
 void m_c() { ... }

 void prog() {
    i = random(2);
    switch(i) {
    case 1: x = null; m_a(); m_b(); x.f(); break;
    case 2: x = null; m_a(); m_c(); x.f(); break; }}
```

Fig. 6. Program for which finding the CFL min-cut in its exploded supergraph for a null analysis leads to finding the vertex cover for the graph with three vertices and two edges $(\{a, b, c\}, \{ab, ac\})$.

Each CFL min-cut has at most one edge from each method m_u. Given a CFL min-cut, we can build the vertex cover by selecting the vertices corresponding to the methods that contain edges of the CFL min-cut. The CFL cut is minimal and there exists exactly one CFL path for each edge in G, therefore the obtained vertex cover is minimal.

Given a vertex cover, we can construct a CFL cut by selecting a cut edge in each method m_u corresponding to a vertex u in the cover.

For the program in Fig. 6, the CFL min-cut is an edge in function `m_a`, which corresponds to the vertex cover $\{a\}$. Intuitively, adding the line `x = new Obj();` in method `m_a` will lead to eliminating all null dereference warnings that the null analysis would trigger.

As an observation, for acyclic directed graphs with normal cut-sizes of up to 2, the CFL min-cut size is equal to the normal min-cut size. This implies that any min-cut of size at most 2 is also a CFL min-cut.

Proposition 1. *If the s, t min-cut has the size at most 2 in an acyclic graph, then this is also an s, t CFL min-cut.*

Proof. Any s, t min-cut is also an s, t CFL cut because if all paths from s to t are removed, then all the CFL paths are also removed. If the graph G', where we remove all edges that are not part of a CFL path from s to t, has a min cut of size 1, then it is also an s, t CFL min-cut.

If G' has a min-cut of size 2, we show that there cannot exist a smaller s, t CFL min-cut. Assuming an s, t CFL -min-cut of size 1 exists, then all CFL paths from s to t contain the cut edge c. We show that all the non-CFL paths from s to t also contain c. Let p be a non-CFL path from s to t. The first edge of p is either c or comes before c on an CFL path from s to t. It cannot come after c, because we would obtain a cycle in the graph. Similarly, for all the edges of p not equal to c, they must appear before c in an CFL path from s to t. Assume no edge is equal to c, then the last edge of the path, reaching t is before c, which creates a cycle. Since all paths contain c, then the s, t min-cut is of size 1, contradicting the hypothesis.

4 Solving the CFL Min-Cut for IFDS

In this section we present our approach to solve the CFL min-cut problem for an IFDS analysis. First, we instrument the IFDS analysis, recording the relevant information while it computes CFL reachability in the exploded super-graph. Second, based on the recorded information and the warnings found by the analysis, we formulate and solve the CFL min-cut as an optimization problem.

4.1 IFDS Analysis Instrumentation

Types of Edges. The tabulation algorithm [19] solves CFL reachability from the **0** entry node to all nodes in the exploded supergraph. In the process, it derives two types of edges: path edges and summary edges. If the algorithm derives a path edge between nodes n_1 and n_2, it means n_2 is actually CFL reachable from the **0** entry node. Summary edges are derived between fact nodes corresponding to a function call and a matching return from call. For instance, in Fig. 4, a summary edge would be introduced between nodes $\langle n_3, \mathbf{0}\rangle$ and $\langle n_4, \mathbf{0}\rangle$.

Derivation Rules. The tabulation algorithm maintains a worklist of recently derived path edges and applies a set of rules to derive additional path edges. The newly derived path edges are implied by existing path edges, exploded supergraph edges and summary edges. The complete description of the derivation rules can be found in [19]. For the example in Fig. 4, the tabulation algorithm starts with path edge $pathEdge_1$ $\langle s_{main}, \mathbf{0}\rangle \longrightarrow \langle s_{main}, \mathbf{0}\rangle$ in the worklist. Next, based on $pathEdge_1$ and the exploded super-graph edge $\langle s_{main}, \mathbf{0}\rangle \longrightarrow \langle n_1, \mathbf{0}\rangle$ ($graphEdge_1$), the algorithm derives $pathEdge_2$ $\langle s_{main}, \mathbf{0}\rangle \longrightarrow \langle n_1, \mathbf{0}\rangle$, and adds it to the worklist.

Recording All Derivations. During the execution of the tabulation algorithm, we record all derivations (a derivation is an instance of a rule application) and keep track of all path edges (PE), summary edges (SE) and edges of the exploded super-graph (GE) that were used in these derivations. The set of all derivations is D and each derivation is stored as an implication. The following is an example of a derivation:

$$pathEdge_1 \wedge graphEdge_1 \Rightarrow pathEdge_2$$

An important property of our instrumentation is that we record all possible derivations for each path edge and summary edge (in case such an edge can be derived in more than one way). This ensures that we capture all derivations and the CFL min-cut we will compute in the next step is guaranteed to be correct (covers all CFL paths). Let $W \subseteq PE$ be the set of path edges (warnings always correspond to path edges) which corresponds to warnings of the analysis. Given the PE, SE, GE, D and W sets, we proceed to find the CFL min-cut.

4.2 Optimization Objective

Let $Edges = PE \cup SE \cup GE$ and let $\sigma\colon Edges \rightarrow \{true, false\}$ map an edge to *true* or *false*. For a given σ, let $[\![]\!]_\sigma\colon D \rightarrow \{true, false\}$ compute the boolean value of each derivation with respect to σ. That is, the truth value of a derivation (such as the one listed above) is computed by simply applying basic logical rules on the truth values of the edges as defined in σ. We define Q as:

$$Q(D, W) = \{\sigma \;\mid\; \forall d \in D : [\![d]\!]_\sigma \wedge \forall w \in W : \neg\sigma(w)\}$$

Here, $Q(D, W)$ denotes the set of valuations that satisfy all derivations in D and for which all warning edges in W are mapped to *false*.

Let $f_\sigma : Edges \rightarrow \{0, 1\}$ such that:

$$\forall e \in Edges : (\sigma(e) \equiv f_\sigma(e) = 1) \wedge (\neg\sigma(e) \equiv f_\sigma(e) = 0)$$

Using this auxiliary function we can now express the CFL min-cut problem as the following optimization objective:

$$\underset{\sigma \in Q(D,W)}{\text{argmax}} \sum_{p \in GE} f_\sigma(p)$$

The solution of this problem will be a valuation in Q that maps the highest number of graph edges to *true*. The graph edges mapped to *false* are the edges of the CFL min-cut. Note that the optimization problem above can have several possible solutions. We describe next a possible criteria to select a solution.

Minimize Abstract Distance. Given a program P and an IFDS analysis, we consider an abstract program as the exploded super-graph esg_P corresponding to P. Let $nr(esg)$ be the number of nodes of the exploded super-graph esg that are CFL reachable from the **0** entry node. We define the distance between two exploded supergraphs as: $d(esg_1, esg_2) = |nr(esg_1) - nr(esg_2)|$. Given esg_P, one intuitive criteria is to select a CFL min-cut C such that the distance $d(esg_P, esg_P \setminus C)$ is minimal, where $esg_P \setminus C$ is the same as esg_P except we remove the edges in C. As discussed in Sect. 2.3, this criteria leads to a CFL min-cut where we try to preserve as many CFL-reachable nodes in the exploded super-graph as possible. For example, in Fig. 4, the abstract distance between the exploded supergraph before and after removing the cut (blue edge) is 8 (there are 8 nodes, shown in blue, that become CFL unreachable). This criterion can be seen as a proxy for reducing the number of changes to the program (as more changes would likely lead to more changes in the computed abstract facts). We now extend our optimization problem with this criterion.

The goal is to select the valuation σ such that it corresponds to a CFL min-cut and it has a maximal number of path edges mapped to *true*. As mentioned before, each path edge corresponds to an abstract fact that holds at a certain program point. The challenge is that, if we simply add the sum of $f_\sigma(p)$ for all $p \in PE$ to the formula above, it will not be sound. The problem is that the result may have path edges mapped to *true*even if the left hand sides of all their corresponding derivations are *false*. This would lead to incorrect results.

To address this challenge, we create a new set of derivations, $\hat{D}$, that contains all derivations in D as well as new derivations, described next. Given a path or a summary edge $p \in PE \cup SE$, let $D_p \subseteq D$ be the derivations in D that have the right hand side of the implication equal to p: $D_p = \{d \in D \mid rhs(d) = p\}$. All derivations $d \in D_p$ are of the form $lhs(d) \Rightarrow p$, where lhs represents the left hand side of the implication. For all $p \in PE \cup SE$, we add to the set $\hat{D}$ the derivation: $p \Rightarrow \bigvee_{d \in D_p} lhs(d)$. This avoids valuations that map a path or a summary edge to *true* even if all left hand sides of their corresponding derivations are *false*.

We define $w\colon GE \cup PE \rightarrow \mathbb{R}$ as a function assigning a value to a path or a graph edge. The new optimization objective is:

$$\underset{\sigma \in Q(\hat{D}, W)}{\operatorname{argmax}} \sum_{p \in GE \cup PE} w(p) \times f_\sigma(p)$$

In particular, if $\forall p \in GE : w(p) = |PE| + 1$ and $\forall p \in PE : w(p) = 1$, then the priority of the optimization problem is to find a CFL min-cut, and then select the cut that maximizes the number of path edges. Additionally, we can implement a trade-off between the size of the CFL cut and the number of path edges, if the sum of weights of PE edges is greater than the weight of at least one GE edge. Maximizing the number of path edges corresponds to minimizing the "may" facts from Fig. 1 that are transformed to "must-not" facts.

4.3 Solution via MaxSAT

We solve the above optimization problems via a translation to MaxSAT.

Variables. For each edge in *Edges* we introduce a boolean variable in the MaxSAT formula. Additionally, for each derivation in D, we introduce a new boolean variable. Let B be the set of all the boolean variables of the formula and let $b_e\colon Edges \rightarrow B$ map edges to the correspondin boolean variables and $b_d\colon D \rightarrow B$ map derivations to boolean variables.

Clauses. For each edge $p \in GE \cup PE$, we add to the boolean formula one unit clause - $b_e(p)$, of weight $w(p)$. For each derivation $d \in D$, we add two clauses: (i) a clause contains, for each edge p appearing in $lhs(d)$, the literal $\neg b_e(p)$ and the literal $b_d(d)$, and (ii) a second clause that contains $\neg b_d(d)$ and $b_e(p)$. The weight of these clauses is set to ∞. Further, for each edge $p \in PE \cup SE$, we add a clause containing the literal $\neg b_e(p)$ and the literals $b_d(d)$, for all $d \in D_p$. This clause has the weight ∞. Finally, for each edge $p \in W$, we add the unit clause $\neg b_e(p)$ with weight ∞.

5 Evaluation

In this section, we present an evaluation of the CFL min-cut approach described in Sect. 4, leveraging a null-deference analysis similar to the one in Sect. 2.

Table 2. Benchmarks showing the differences between the number of warnings, CFL and normal min-cut sizes for the null pointer analysis.

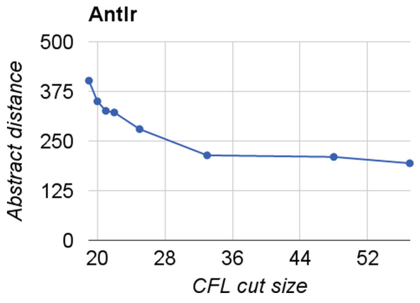
Benchmark	Null pointer analysis		CFL min-cut			Normal min-cut		
	Time (s)	Warnings	Time (s)	Size	Distance	Time (s)	Size	Distance
Antlr	13	114	32	19	401	28	28	588
Eclipse	9	41	13	12	260	11	12	351
Hsqldb	1	22	1	2	120	0.6	2	122
Luindex	11	92	60	9	1519	8	10	1636
Pmd	1	22	2	2	124	0.3	2	126
Xalan	1	40	3	3	232	0.4	3	234
Javasrc-p	12	28	16	11	167	10	14	309
Kawa-c	27	14	34	9	226	23	10	726
Rhino-a	64	59	173	15	2154	87	18	3282
Schroeder-m	2	40	3	5	382	0.8	5	420
Toba-s	1	1	0.4	1	2	0.3	1	30

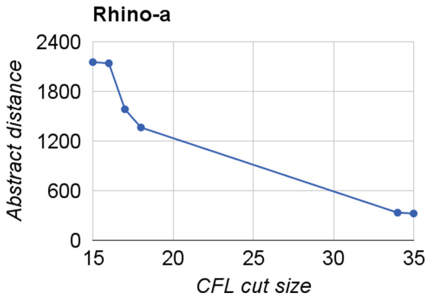
Null Dereference Analysis. We first implemented an IFDS-based analysis for detecting null-pointer dereferences (as described in Sect. 2), leveraging the WALA analysis framework [1]. The analysis runs in the forward direction, tracking access paths that either may be null (if some statement may have written null to the location) or may-not be null (some statement wrote a non-null value into the location; if the may-not be null fact is unreachable, then the variable must be null). A null-pointer error is reported when a variable that may (or must) be null is de-referenced. The analysis includes basic interpretation of branch conditions comparing variables to null. The analysis does not track all aliasing exactly, and hence is unsound—soundness is not required to evaluate our current techniques. In our evaluation, we used the WPM-3-2015.co MaxSAT solver [2].

Benchmarks. We ran the analysis on a suite of 11 open-source Java benchmarks. The set of benchmarks includes programs from the DaCapo suite [3], and also programs used in other recent work on static analysis [7,23].

Results. Table 2 summarizes the results from our experiments. The first column indicates the name of the benchmark program. The next two columns show the running time in seconds of the null dereference analysis and the number of warnings it generates for each program. As described in Sect. 4, the analysis is instrumented and all possible derivations are recorded during its execution. The next three columns present information about the CFL min-cut computation: the running time in seconds, the size of the cut, and the abstract distance between the exploded supergraphs before and after the cut (defined in Sect. 4.2). The final three columns present the same information as before, this time for computing a normal min-cut, instead of a CFL min-cut.

Table 3. Abstract distance vs. CFL cut size trade-off, for Antlr and Rhino-a.





Number of Warnings vs. Number of Fix Locations. For several benchmarks (Hsqldb, Luindex, Pmd, Xalan, Schroeder-m), the size of the min-cut is approximately 10 times smaller than the number of warnings of the analysis. This confirms our hypothesis that a small number of fix locations has the potential to address all warnings—computing a min-cut is beneficial in such cases.

CFL Min-Cut Size. Computing a CFL min-cut can often reduce the size of the cut over a normal min-cut. For benchmarks such as Antlr, Luindex, Javasrc-p, Kawa-c, Rhino-a, the normal min-cut is between 10%–50% larger than the CFL min-cut. This is a non-trivial difference, as each report requires manual effort from the end user.

Program Fixes. We inspected manually the fix locations proposed by our system for several benchmarks and discovered that identifying the concrete fix was straightforward. Adding these fixes removed all the initial warnings. Constructing the fixes can become more complex in the case of other IFDS analyses. We consider that to be an interesting future work item.

Implementation Details. To simplify the boolean formulas that are generated, we exclude identity edges (propagating the same fact) from the possible cuts. This reduces greatly the burden of the MaxSAT solver, enabling a more scalable implementation. Intuitively this The MaxSAT solver can take advantage of the stratification optimization [2] to compute faster the min-cuts with maximized path edges.

Case Study: Cut Size - Abstract Distance Tradeoffs. We investigated the tradeoff between the CFL cut size and the abstract distance. We kept the weight for all edges $p \in PE$ as $w(p) = 1$ and we set the weight of $p \in GE : w(p) = k$, where k is a constant between 1 and $|PE| + 1$. We illustrate our results on two of the benchmarks, Antlr and Rhino-a. For each benchmark, we show in Table 3 the CFL cut size (on the X axis) and the abstract distance (on the Y axis) for several values of k. (Note that the CFL cuts for different values of k are not subsets of each other.) For both benchmarks we observe that allowing a larger cut size leads to a smaller abstract distance. This makes intuitive sense: as cuts are allowed to grow larger, fix locations can be more "specialized" to warnings, reducing effects on other may facts. In the limit, a cut could include a fix location specific to each warning, minimizing abstract distance.

6 Related Work

Our work bears some similarity to recent work on finding minimal explanations for type inference errors. Pavlinovic et al. leverage MaxSMT for computing small explanations [17], while Loncaric et al. leverage MaxSAT [14]. Zhang and Myers [22] takes a probabilistic approach based on constraint graphs from type inference. These techniques may also benefit from factoring in a notion of abstract distance, rather than purely minimizing the number of fix locations.

Given an error trace that violates an assertion, Jose and Majumdar [9] use MaxSAT to localize the error causes. Their work identifies potential error causes of the concrete test execution that violates an assertion, whereas our approach focuses on warnings of static analyses.

Merlin [12] automatically classifies methods into sources, sinks and sanitizers, for taint analysis. Their sanitizer inference could be viewed as finding ways to "cut" flows from sources to sinks, but our problem differs in that we allow many more graph edges to be cut. Livshits and Chong [11] aim to find a valid sanitization of a given data-flow graph. Their approach leverages static and dynamic analysis, whereas our work is purely static.

The complexity class for the view update problem is studied in Buneman et al. [4]. The paper investigates the complexity of identifying a minimal set tuples in the database whose deletion will eliminate a given tuple from a query view; this problem bears some similarity to the CFL min-cut problem as applied to chain Datalog programs (see Sect. 1). The paper proves the problem is NP-hard for several types of queries.

D'Antoni et al. [6] aim to find repairs of small programs such that both the syntactic and the semantic difference between the original and the fixed programs is minimal, leveraging the Sketch synthesis tool [20]. The semantic difference is the distance between concrete traces of the programs. In contrast, our work focuses on minimizing abstract distances. Moreover, our system suggests fix locations (corresponding to abstract transitions) and runs on large benchmarks, as opposed to computing concrete program fixes for small input programs.

The system in Mangal et al. [15] infers weights for rules of a static analysis in order to classify the warnings of the analysis, based on feedback collected from users on previous analysis runs. It is interesting to apply similar techniques to automatically infer weights for edges in the cut so to explore further the min-cut vs. abstract distance tradeoff. In our work, we directly determine possible fix locations that will address all the warnings. Our experiments show that the number of fix locations can be several times smaller than the number of warnings.

Recent work [5,13] presents a system that infers necessary preconditions (when such a precondition does not hold, the program is guaranteed to be wrong) on large-scale programs. Their work considers non-null, array out of bounds and contracts analyses. Our system can be viewed as inferring a minimal number of sufficient preconditions for IFDS analyses.

7 Conclusion

The CFL min-cut is a fundamental building block for suggesting fix locations for both false positive warnings caused by over-approximations of the analysis and true bugs of the program. In this work, we first proved that computing CFL min-cuts is NP-hard. Next, we phrased the CFL min-cut as an optimization problem and solved it via MaxSAT. Using a null dereference analysis, we experimentally showed that in practice the CFL min-cut frequently yields fewer fix locations and smaller abstract program distances than a normal min-cut. In future work, we plan to apply CFL min-cuts to more program analysis problems and investigate faster CFL min-cut algorithms for common graph structures.

Acknowledgements. We thank Dimitar Dimitrov from ETH Zurich for comments on earlier proofs of the theorems in this paper.

References

1. Watson, T.J.: Libraries for Analysis (WALA). http://wala.sf.net. Accessed 22 Jan 2017
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted maxSAT solvers. In: Milano, M. (ed.) CP 2012. LNCS, pp. 86–101. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33558-7_9
3. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 169–190. ACM, New York (2006)
4. Buneman, P., Khanna, S., Tan, W.-C.: On propagation of deletions and annotations through views. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002, pp. 150–158. ACM, New York (2002)
5. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 128–148. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35873-9_10
6. D'Antoni, L., Samanta, R., Singh, R.: Qlose: program repair with quantitative objectives. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 383–401. Springer, Cham (2016). doi:10.1007/978-3-319-41540-6_21
7. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: Pollock, L.L., Pezzè, M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, 17–20 July 2006, pp. 133–144. ACM (2006)

8. Hao, J., Orlin, J.B.: A faster algorithm for finding the minimum cut in a graph. In: Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1992, pp. 165–174. Society for Industrial and Applied Mathematics, Philadelphia (1992)
9. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 437–446. ACM, New York (2011)
10. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Proceedings of a Symposium on the Complexity of Computer Computations, 20–22 March 1972, IBM Thomas J. Watson Research Center, Yorktown Heights, New York. The IBM Research Symposia Series, pp. 85–103. Plenum Press, New York (1972)
11. Livshits, B., Chong, S.: Towards fully automatic placement of security sanitizers and declassifiers. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, pp. 385–398. ACM, New York (2013)
12. Livshits, B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 75–86. ACM, New York (2009)
13. Logozzo, F., Ball, T.: Modular and verified automatic program repair. In: Leavens, G.T., Dwyer, M.B. (eds.) Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, 21–25 October 2012, pp. 133–146. ACM (2012)
14. Loncaric, C., Chandra, S., Schlesinger, C., Sridharan, M.: A practical framework for type inference error explanation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pp. 781–799. ACM, New York (2016)
15. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 462–473. ACM, New York (2015)
16. Melski, D., Reps, T.W.: Interconvertibility of a class of set constraints and context-free-language reachability. Theoret. Comput. Sci. **248**(1–2), 29–98 (2000)
17. Pavlinovic, Z., King, T., Wies, T.: Practical SMT-based type error localization. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, pp. 412–423. ACM, New York (2015)
18. Reps, T.: Program analysis via graph reachability. In: Proceedings of the 1997 International Symposium on Logic Programming, ILPS 1997, pp. 5–19. MIT Press, Cambridge (1997)
19. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, 23–25 January 1995, pp. 49–61. ACM Press (1995)
20. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pp. 404–415. ACM, New York (2006)

21. Stoer, M., Wagner, F.: A simple min-cut algorithm. J. ACM **44**(4), 585–591 (1997)
22. Zhang, D., Myers, A.C.: Toward general diagnosis of static errors. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 569–581. ACM, New York (2014)
23. Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in datalog. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom - 09–11 June 2014, p. 27 (2014)