Efficient Computation of Arbitrary Control Dependencies

Jean-Christophe Léchenet[a,b], Nikolai Kosmatov[a,c,*], Pascale Le Gall[b]

[a] *Université Paris-Saclay, CEA, List, Palaiseau, France*
[b] *Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes, CentraleSupélec, Université Paris-Saclay, Gif-sur-Yvette, France*
[c] *Thales Research & Technology, Palaiseau, France*

Abstract

In 2011, Danicic et al. introduced an elegant generalization of the notion of control dependence for any directed graph. They also proposed an algorithm computing the weak control-closure of a subset of graph vertices and performed a paper-and-pencil proof of its correctness. We have performed its machine-checked proof in the Coq proof assistant. This paper also presents a novel, more efficient algorithm called lDFS to compute weak control-closure taking benefit of intermediate propagation results of previous iterations in order to accelerate the following ones. This optimization makes the design of the algorithm more complex and requires subtle loop invariants for its proof. lDFS has been formalized and mechanically proven in the Why3 verification tool. To investigate the impact of several possible optimizations and compare the performances of different versions of the algorithm, we perform experiments on arbitrary generated graphs with up to hundreds of thousands of vertices. They demonstrate that the proposed algorithm remains practical for real-life programs and significantly outperforms all considered versions of Danicic's initial technique.

Keywords: Control dependence, Control-closure, Graph theory, Proof of soundness, Coq, Why3, Program slicing

1. Introduction

Context. Control dependence is a fundamental notion in software engineering and analysis (e.g. [1, 2, 3, 4, 5, 6]). It reflects structural relationships between different program statements and is intensively used in many software analysis techniques and tools, such as compilers, verification tools, test generators, program transformation tools, simulators, debuggers, etc. Along with data dependence, it is one of the key notions used in *program slicing* [2, 7], a program

*Corresponding author.
Email address: `nikolaikosmatov@gmail.com` (Nikolai Kosmatov)

transformation technique allowing to decompose a given program into a simpler one, called a *program slice.*

In 2011, Danicic et al. [8] proposed an elegant generalization of the notions of closure under non-termination insensitive (*weak*) and non-termination sensitive (*strong*) control dependence. They introduced the notions of weak and strong control-closures, that can be defined on any directed graph, and no longer only on control flow graphs. They proved that weak and strong control-closures subsume the closures under all forms of control dependence previously known in the literature. In the present paper, we are interested in the non-termination insensitive form, i.e. *weak control-closure.*

Besides the definition of weak control-closure, Danicic et al. also provided an algorithm computing it for a given set of vertices in a directed graph. This algorithm was proved by paper-and-pencil. Under the assumption that the given graph is a CFG (or more generally, that the maximal out-degree of the graph vertices is bounded), the complexity of the algorithm can be expressed in terms of the number n of vertices of the graph, and was shown to be $O(n^3)$. This can explain why this algorithm was not used until now. Danicic et al. themselves conjectured that it should be possible to improve its complexity.

Motivation. Danicic et al. introduced basic notions used to define weak control-closure and to justify the algorithm, and proved a few lemmas about them. While formalizing these concepts in the Coq proof assistant [9, 10], we have discovered that, strictly speaking, their paper-and-pencil proof of one of them [8, Lemma 53] is inaccurate, whereas the lemma itself is correct. The first motivation of this work is to provide a mechanically verified proof of these results, necessary to avoid any risk of error. Furthermore, Danicic's algorithm does not take advantage of its iterative nature and does not reuse the results of previous iterations in order to speed up the following ones. Our second motivation is thus to optimize Danicic's algorithm.

Goals and Main Results. First, we fully formalize Danicic's algorithm, its correctness proof and the underlying concepts in Coq. Our second objective is to design a more efficient algorithm sharing information between iterations in order to speed up the execution. We call this new algorithm lDFS, since it uses DFS (Depth-First Search) traversals of the graph to update a labeling of the nodes. Since lDFS is carefully optimized and more complex, its correctness proof relies on more subtle arguments than for Danicic's algorithm. To deal with them and to avoid any risk of error, we have proved lDFS correct using a mechanized verification tool once again — this time, the Why3 proof system [11, 12].

Finally, in order to evaluate lDFS with respect to Danicic's initial technique, we have implemented both algorithms in OCaml (using OCamlgraph library [13]) and tested them on a large set of randomly generated graphs with up to hundreds of thousands of vertices. In addition, we considered a couple of optimizations, referred to as **(opt1)** and **(opt2)**, for Danicic's original algorithm, and implemented and evaluated the corresponding versions. A certified implementation extracted from the Coq development was also evaluated and

used as an oracle to check all other implementations. Experiments demonstrate that the proposed algorithm sharing information between iterations turns out to be the most efficient: it remains applicable to large graphs (and thus to CFGs of real-life programs) and significantly outperforms all considered versions of Danicic's original technique.

Contributions. The contributions of this paper include:

- a formalization of Danicic's algorithm and proof of its correctness in Coq;
- a new algorithm lDFS computing weak control-closure and taking benefit from preserving some intermediary results between iterations;
- a mechanized correctness proof of lDFS in the Why3 tool including a formalization of the basic concepts and results of Danicic et al.;
- an implementation of Danicic's initial technique (in several versions, without and with **(opt1)** and **(opt2)**) and our new algorithm in OCaml, their evaluation on randomly generated graphs and a detailed comparison of their performances.

The Coq, Why3 and OCaml implementations are all available in [14]. This paper is an extended version of a previous conference paper [15]. The extensions include careful proofs or proof sketches of all results and a larger evaluation campaign, including optimizations **(opt1)** and **(opt2)** of Danicic's original technique and their comparison with lDFS in order to better understand the impact of possible optimizations with and without sharing information between iterations. In particular, a detailed proof is given for Prop. 3 fixing the minor flaw found in the original proof in [8]. To emphasize all technical difficulties of the underlying argument, we state and prove a new lemma, Lemma 2, and carefully illustrate its main steps by several figures.

Outline. We present our motivation and a running example in Sect. 2. Then, we recall the definitions of some important concepts introduced by [8] in Sect. 3 and state two important lemmas in Sect. 4. Next, we describe Danicic's algorithm in Sect. 5 and lDFS along with a sketch of the proof of its correctness in Sect. 6. Experiments are presented in Sect. 7. Finally, Sect. 8 presents some related work and concludes.

2. Motivation and Running Example

This section informally presents weak control-closure using a running example.

The inputs of our problem are a directed graph $G = (V, E)$ with set of vertices (or nodes) V and set of edges E, and a subset of vertices $V' \subseteq V$. The property of interest of such a subset is called *weakly control-closed* in [8] (cf. Def. 3). V' is said to be *weakly control-closed* if each node reachable from V' is V'-*weakly committing* (cf. Def. 2), i.e. all paths starting at such a node lead

the flow to at most one (first reachable) node in V'. If the property is not true, then, starting from some node u' reachable from V', the flow can diverge at some node u (not in V') and reach V' in two different nodes.

If V' is not weakly control-closed, we often need to build a superset of V' satisfying this property, and more particularly the smallest one, called the *weak control-closure* of V' in G (cf. Def. 5). Intuitively, a weakly control-closed set must contain any node u of V reachable from V' that can *decide* to which node of V' the control will flow. These nodes can also be characterized as the *points of divergence closest to* V'. They are called V'*-weakly deciding*. Formally, vertex u is V'*-weakly deciding* if there exist two non-trivial paths starting from u and reaching V' that have no common vertex except u (cf. Def. 4). To compute the weak control-closure, as it will be proved by Lemma 3, we need to add to V' the V'-weakly deciding vertices that are reachable from V'.

Let us illustrate these ideas on an example graph $G_0 = (V_0, E_0)$ shown in Fig. 1. $V'_0 = \{u_1, u_3\}$ is the subset of interest represented with double circles (u_i) in Fig. 1. Vertex u_5 is reachable from V'_0 and is not V'_0-weakly committing, since it is the origin of two paths u_5, u_6, u_0, u_1 and u_5, u_6, u_0, u_2, u_3 that lead the flow to two different nodes u_1 and u_3 in V'_0. Hence, by definition, V'_0 is not weakly control-closed. We see that these two paths diverge at u_0 and do not have any common node after the divergence point u_0. Thus u_0 decides whether the control flows to u_1 and u_3 in V'_0.

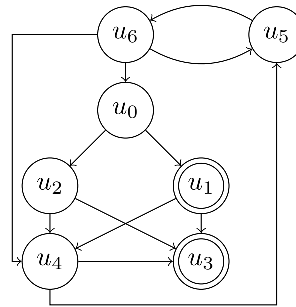
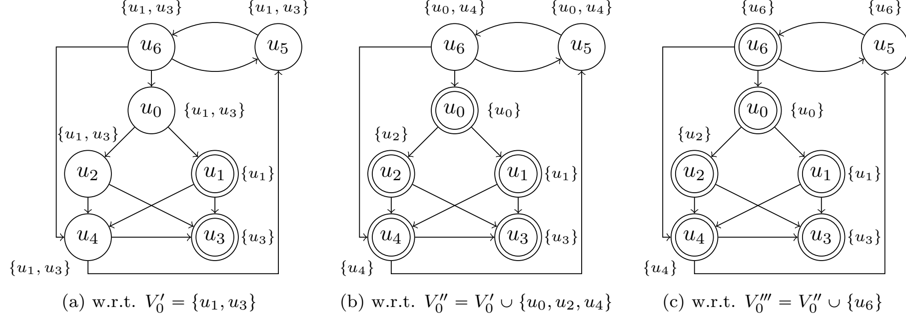


Figure 1: Example graph $G_0 = (V_0, E_0)$, with $V'_0 = \{u_1, u_3\}$

As mentioned above, to build the weak control-closure of V'_0, we need to add to V'_0 all V'_0-weakly deciding nodes reachable from V'_0. Vertex u_0 is such a node. Indeed, it is reachable from V'_0 and we have two non-trivial paths u_0, u_1 and u_0, u_2, u_3 starting from u_0, ending in V'_0 (respectively, in u_1 and u_3) and sharing no other vertex than their origin u_0. Similarly, nodes u_2, u_4 and u_6 are V'_0-weakly deciding and must be added as well. On the contrary, u_5 must not be added, since every non-empty path starting from u_5 has u_6 as second vertex. More generally, a node with only one successor can obviously not be a "divergence point closest to V'" and cannot decide to which node in V' the control will flow, hence it never needs to be added to build the weak control-closure. The weak control-closure of V'_0 in G_0 is thus $\{u_0, u_1, u_2, u_3, u_4, u_6\}$.

Danicic's algorithm to construct the closure — like the one we propose — does not directly try to build such two paths sharing only one node in order to identify a weakly deciding node. Both algorithms identify some V'-weakly deciding vertices relying on a concept called *observable vertex*. Given a vertex $u \in V$, the set of *observable vertices* in V' from u contains all nodes reachable from u in V' without using edges starting in V'. Intuitively, they are the first reachable nodes in V' from u. Figure 2a shows our example graph G_0, where each node is annotated with its set of observables in V'_0.



(a) w.r.t. $V_0' = \{u_1, u_3\}$ (b) w.r.t. $V_0'' = V_0' \cup \{u_0, u_2, u_4\}$ (c) w.r.t. $V_0''' = V_0'' \cup \{u_6\}$

Figure 2: Example graph G_0 annotated with observable sets

The important property about this object underlying the main idea of the algorithms — as it will follow from Lemmas 5 and 3 — is the following: if there exists an edge $(u, v) \in E$ such that (i) u is reachable from V', (ii) u is not in V', (iii) v can reach V', and (iv) there exists a vertex w observable from u but not from v, then u is a V'-weakly deciding node, and it must be added to V' to build its weak control-closure. Indeed, in this case, u decides at least whether the control can flow from u to w or another node (reachable from v) in V'.

In our example, (u_0, u_1) is an edge such that u_0 is reachable from V_0', u_0 is not in V_0', u_1 can reach V_0' (since $u_1 \in V_0'$), and u_3 is an observable vertex from u_0 in V_0' but not from u_1. Hence u_0 is a node to be added into the weak control-closure. Likewise, considering the edges (u_2, u_3) and (u_4, u_3), we can deduce that u_2 and u_4 must belong to the closure. However, we have seen that u_6 belongs to the closure, but it is not possible to directly deduce it by applying the same reasoning to edges (u_6, u_0), (u_6, u_4) or (u_6, u_5).

As Lemma 4 will establish, the computation technique is actually iterative. We should add to the initial V_0' the nodes that we have already detected and apply this technique again to this new set V_0''. The vertices that will be detected this way will also be in the closure of the initial set V_0'. The observable sets with respect to $V_0'' = V_0' \cup \{u_0, u_2, u_4\}$ are shown in Fig. 2b. This time, applying the aforementioned property to edge (u_6, u_4) or edge (u_6, u_0) allows us to add u_6 to the closure. Applying again the same technique with the augmented set $V_0''' = V_0'' \cup \{u_6\}$ (cf. Fig. 2c) does not reveal new vertices to add. This means that all nodes of the closure have already been found. We obtain the same set as before for the weak control-closure of V_0', i.e. $\{u_0, u_1, u_2, u_3, u_4, u_6\}$.

3. Basic Concepts

This section introduces basic definitions and properties needed to define the notion of weak control-closure. The proofs were inspired by Danicic et al.'s ones [8] and were adapted to give a slightly clearer presentation and to give a rigorous proof for the key property of Prop. 3 whose proof in [8] was inaccurate.

All these notions and proofs have been formalized in Coq [14], including in particular Prop. 3.

From now on, let $G = (V, E)$ denote a directed graph, and V' a subset of V. We define a *path* in G in the usual way. A path is written as the sequence of visited nodes. A non-trivial path has at least two nodes. We write $u \xrightarrow{\text{path}} v$ if there exists a path from u to v. Let $\mathsf{R}_G(V') = \{v \in V \mid \exists u \in V',\ u \xrightarrow{\text{path}} v\}$ be the set of nodes reachable from V'. In our example (cf. Fig. 1), u_6, u_0, u_1, u_3 is a (4-node) path in G_0, u_1 is a trivial one-node path in G_0 from u_1 to itself, and $\mathsf{R}_{G_0}(V'_0) = V_0$.

Definition 1 (V'-disjoint path, V'-path). A path π in G is said to be V'-*disjoint* in G if all the vertices in π, except possibly the last one, are not in V'. A V'*-path* in G is a V'-disjoint path whose last vertex is in V'. In particular, if $u \in V'$, the only V'-path starting from u is the trivial path u.

We write $u \xrightarrow{V'\text{-disjoint}} v$ (resp. $u \xrightarrow{V'\text{-path}} v$) if there exists a V'-disjoint path (resp. a V'-path) from u to v. A non-trivial path π from u to v can also be written in the form $\pi = u\pi'v$ for some (possibly empty) subsequence of nodes π' of the path π.

Example 1. In G_0, u_3; u_2, u_3; u_0, u_1; u_0, u_2, u_3 are V'_0-paths and thus V'_0-disjoint paths. The path u_6, u_0 is a V'_0-disjoint path but not a V'_0-path.

Remark 1. Definition 1 and the following ones used in our formalization are slightly different from [8], where a V'-path must contain at least two vertices and there is no constraint on its first vertex, which can be in V' or not. Our definitions differ from Danicic et al.'s only in the way they deal with the nodes in V'. They lead to the same notion of weak control-closure, but seemed to us to be slightly more natural.

Definition 2 (V'-weakly committing vertex). A vertex u in G is V'*-weakly committing* if all the V'-paths from u have the same end point (in V'). In particular, any vertex $u \in V'$ is V'-weakly committing.

Example 2. In G_0, u_1 and u_3 are the only V'_0-weakly committing nodes.

Definition 3 (Weakly control-closed set). A subset V' of V is *weakly control-closed* in G if every vertex reachable from V' is V'-weakly committing.

Example 3. The empty set $\varnothing$, singletons and the set of all nodes V are trivially weakly control-closed. In our running example G_0, since in particular u_2 is reachable from V'_0 but not V'_0-weakly committing, we see that V'_0 is not weakly control-closed in G_0. Less trivial weakly control-closed sets include $\{u_0, u_1\}$, $\{u_4, u_5, u_6\}$ and $\{u_0, u_1, u_2, u_3, u_4, u_6\}$.

Definition 3 characterizes a weakly control-closed set, but does not explain how to build one. It would be particularly interesting to build the smallest

weakly control-closed set containing a given set V'. The notion of *weakly deciding vertex* will help us to give an explicit expression to that set. Intuitively, as mentioned in Sec. 2, V'-weakly deciding nodes *decide* to which node of V' the control will flow from them.

Definition 4 (V'-weakly deciding vertex). A vertex u is *V'-weakly deciding* if there exist at least two non-trivial V'-paths from u that share no vertex except u. Let $\mathsf{WD}_G(V')$ denote the set of V'-weakly deciding vertices in G.

Property 1. *$V' \cap \mathsf{WD}_G(V') = \varnothing$.*

Proof. Let u be an element of $\mathsf{WD}_G(V')$. By Def. 4, there exist two non-trivial V'-paths from u ending in V'. By Def. 1, since u appears as a non-terminal vertex on such a path, it follows that $u \notin V'$. □

Example 4. In G_0, by Prop. 1, $u_1, u_3 \notin \mathsf{WD}_{G_0}(V'_0)$. We have illustrated the definition to show that $u_0 \in \mathsf{WD}_{G_0}(V'_0)$ and $u_5 \notin \mathsf{WD}_{G_0}(V'_0)$ in Sect. 2. Similarly, it is easily seen that $u_2, u_4, u_6 \in \mathsf{WD}_{G_0}(V'_0)$. Thus we have $\mathsf{WD}_{G_0}(V'_0) = \{u_0, u_2, u_4, u_6\}$.

Lemma 1 (Characterization of a weakly control-closed set). *V' is weakly control-closed in G if and only if there is no V'-weakly deciding vertex in G reachable from V'.*

Proof. Both implications are shown by contradiction. We show first that the condition is necessary. Indeed, if we have a V'-weakly deciding vertex reachable from V', then it is not V'-weakly committing, and therefore is a clear counter-example to V' being weakly control-closed. Assume now that V' is not weakly control-closed. We have a vertex reachable from V' and giving rise to two V'-paths ending in V'. The last vertex of the first V'-path that also occurs on the other V'-path is a V'-weakly deciding node reachable from V'. □

Example 5. In G_0, u_2 is reachable from V'_0 and is V'_0-weakly deciding. By Lemma 1, this gives another argument that V'_0 is not weakly control-closed.

Let us now state two other useful properties of WD_G.

Property 2. *Let V'_1, V'_2 be two subsets of V with $V'_1 \subseteq V'_2$. Then $\mathsf{WD}_G(V'_1) \subseteq V'_2 \cup \mathsf{WD}_G(V'_2)$.*

Proof. Let u be a vertex in $\mathsf{WD}_G(V'_1)$. By Def. 4, there exist two non-trivial V'_1-paths from u ending in V'_1 that share no vertex except u. Since $V'_1 \subseteq V'_2$, these paths also end in V'_2. By considering for each path the shortest prefix ending in V'_2, we obtain two V'_2-paths from u that share no vertex except u. If one of these V'_2-paths is trivial, then $u \in V'_2$ (and the other path is trivial too). Otherwise, $u \in \mathsf{WD}_G(V'_2)$ by Def. 4. □

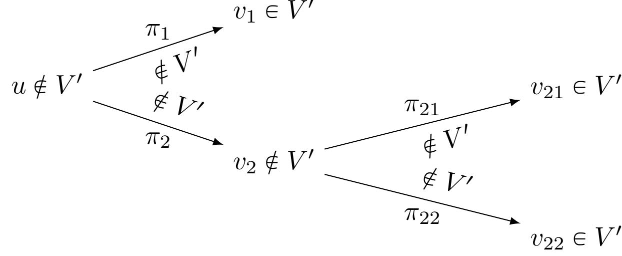


Figure 3: Schematic representation of the configuration of Lemma 2

For the next property, we need the following auxiliary lemma that basically states that if a node u can be the latest point of divergence before reaching (by V'-disjoint paths) a node of V' and a V'-weakly deciding node, then u is V'-weakly deciding itself. Its statement is more general than the corresponding statement proved by Danicic et al.

Lemma 2. *Let $u \notin V'$, $v_1 \in V'$ and $v_2 \in \mathsf{WD}_G(V')$ be three distinct vertices in G, and let $u\pi_1v_1$ and $u\pi_2v_2$ be two paths in G that share no vertex except u. Suppose that π_1 and π_2 have no vertex in V'. Then $u \in \mathsf{WD}_G(V')$.*

Proof. By Def. 4, since $v_2 \in \mathsf{WD}_G(V')$ we have two non-trivial V'-paths $v_2\pi_{21}v_{21}$ and $v_2\pi_{22}v_{22}$, sharing no vertex except v_2 (see Fig. 3). Overall, the lemma is equivalent to the following claim: for any $u \notin V'$ and any three V'-paths $\rho_1 = u\pi_1v_1$, $\rho_2 = u\pi_2v_2\pi_{21}v_{21}$ and $\rho_3 = u\pi_2v_2\pi_{22}v_{22}$, with no intersection between π_1v_1 and π_2v_2 and between $\pi_{21}v_{21}$ and $\pi_{22}v_{22}$, there exist two non-trivial V'-paths from u sharing no vertex except u.

A detailed proof is very technical and is available in the Coq formalization. We give here a sketch of a paper-and-pencil proof of the claim providing the main ideas of the argument without all technical details.

Without loss of generality, we can assume that paths $u\pi_1v_1$ and $u\pi_2v_2$ have no loops (i.e. no node is traversed several times). Indeed, we can eliminate loops and consider shorter loop-free paths of the same form. Similarly, we also assume that $v_2\pi_{21}v_{21}$ is loop-free. Removing loops is illustrated by Figure 4.

Furthermore, we can assume that the whole path $\rho_2 = u\pi_2v_2\pi_{21}v_{21}$ is loop-free as well. Indeed, otherwise, let w_2 be the latest node in subpath $\pi_{21}v_{21}$ that also occurs in subpath $u\pi_2$. Hence, $w_2 \notin V'$, thus w_2 cannot be v_{21}. Removing the loop from w_2 to w_2 (containing v_2) in the whole path ρ_2, we obtain a reduced loop-free path ρ_2' from u to v_{21}. Paths ρ_2' and ρ_3 diverge at w_2, so we can deduce $w_2 \in \mathsf{WD}_G(V')$. If w_2 is u, we are done. If not, w_2 is in π_2, and it is indeed sufficient to prove the claim for paths ρ_1, ρ_2', ρ_3 with a loop-free path ρ_2' from u to v_{21} (instead of ρ_2) and w_2 as a divergence point of ρ_2' and ρ_3 (instead of v_2). Figure 5 shows one example of such a transformation. Similarly, we can assume that the part of ρ_3 between the new divergence point w_2 and v_{22}, as well as the whole path ρ_3, are loop-free too.

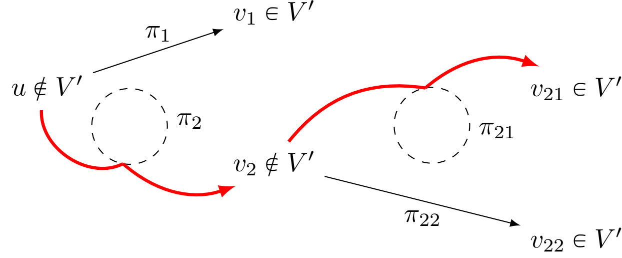


Figure 4: One configuration where cycles in $u\pi_2 v_2$ and $v_2\pi_{21}v_{21}$ are removed. The red thick paths are the loop-free versions of the initial paths. The removed loops are shown dashed.

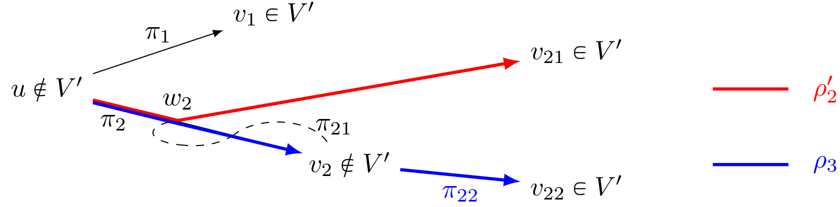


Figure 5: One configuration where a cycle in ρ_2 is removed. In the new configuration, the new path ρ'_2 (in red) and the unchanged path ρ_3 (in blue) diverge at w_2. The removed part of π_{21} is shown dashed.

It remains to show the claim assuming ρ_1, ρ_2, ρ_3 are loop-free. If $\rho_1 = u\pi_1 v_1$ shares no nodes with at least one of the paths ρ_2 or ρ_3 except u, we are done. Otherwise, let w_1 be the first node that ρ_1 has in common with ρ_2 or ρ_3; we can assume (up to renaming) it is common with ρ_2. By assumptions, w_1 must then belong to its subpath $\pi_{21}v_{21}$. Then, the path ρ_3 and the path obtained as the subpath of ρ_1 from u to w_1 followed by the (possibly trivial) subpath of ρ_2 from w_1 to v_{21} can be shown to be two non-trivial V'-paths from u to v_{22} and v_{21} sharing no vertex except u. Figure 6 is an example of the construction of such V'-paths. $\square$

Based on Lemma 2, we can prove the following property. We detail its proof, since it is the one that is inaccurate in Danicic at al.'s work.

Property 3. $\mathsf{WD}_G(V' \cup \mathsf{WD}_G(V')) = \varnothing$.

Proof. Assume there exists an element $u \in \mathsf{WD}_G(V' \cup \mathsf{WD}_G(V'))$. By Property 1, we have then that $u \notin V' \cup \mathsf{WD}_G(V')$. To obtain a contradiction, let us deduce from the assumption that $u \in \mathsf{WD}_G(V')$. By Def. 4, there exist two non-trivial $(V' \cup \mathsf{WD}_G(V'))$-paths from u that share no vertex except u. Either path can end in V' or in $\mathsf{WD}_G(V')$. This gives four cases.

- In the first case, both paths end in V', thus $u \in \mathsf{WD}_G(V')$ by Def. 4.

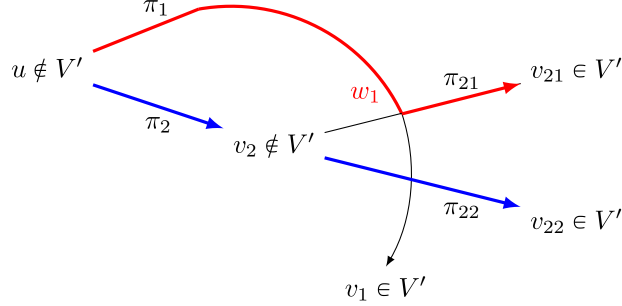


Figure 6: One configuration where $u\pi_1 v_1$ intersects $\pi_{21} v_{21}$. The two non-trivial V'-paths from u sharing no vertex except u are shown with thick lines.

- The second and third cases are symmetrical. One path ends in V', the other one in $\mathsf{WD}_G(V')$. Both paths are $(V' \cup \mathsf{WD}_G(V'))$-disjoint, thus they are in particular V'-disjoint. By Lemma 2, we have $u \in \mathsf{WD}_G(V')$.

- In the fourth case, the two paths are of the form $u\pi_1 v_1$ and $u\pi_2 v_2$ for some $v_1, v_2 \in \mathsf{WD}_G(V')$. By Def. 4, v_1 (resp., v_2) decides if the control can flow to some $v_{11}, v_{12} \in V'$ (resp., some $v_{21}, v_{22} \in V'$). Using the corresponding V'-paths, we obtain four concatenated paths $u\pi_1 v_1 \pi_{11} v_{11}$, $u\pi_1 v_1 \pi_{12} v_{12}$, $u\pi_2 v_2 \pi_{21} v_{21}$ and $u\pi_2 v_2 \pi_{21} v_{22}$. If π_{11} and π_2 had a common vertex, the last vertex in π_{11} that also occurs in π_2 would satisfy the hypotheses of Lemma 2 (for paths to $v_{11} \in V'$ and $v_2 \in \mathsf{WD}_G(V')$), thus would be in $\mathsf{WD}_G(V')$, which is impossible in π_2. Thus π_{11} and π_2 are disjoint. By applying again Lemma 2 for paths $u\pi_1 v_1 \pi_{11} v_{11}$ and $u\pi_2 v_2$, we deduce that $u \in \mathsf{WD}_G(V')$.

The contradiction with $u \notin \mathsf{WD}_G(V')$ finishes the proof. □

Remark 2. Our proof follows the same structure as Danicic et al's proof [8, Lemma 53] . But in the fourth case, instead of relying on Lemma 2 like we do, they refer to the proof of the second and third cases. The hypotheses look the same, there are two V'-disjoint paths sharing no vertex except u, one ending in V', the other ending in $\mathsf{WD}_G(V')$. But actually in the second and third cases, the hypotheses are stronger, the paths are not only V'-disjoint, they are $(V' \cup \mathsf{WD}_G(V'))$-disjoint. And this is a key argument in Danicic et al.'s proof of the second and third cases. Thus it is incorrect to apply that proof in the fourth case. To fix the proof, we proved the auxiliary Lemma 2 whose statement corresponds to the second and third cases but generalized with V'-disjoint paths instead of $(V' \cup \mathsf{WD}_G(V'))$-disjoint paths. That lemma can correctly be applied in the fourth case.

We are ready to prove that adding to a given set V' the V'-weakly deciding nodes that are reachable from V' gives a weakly control-closed set in G. This set is the smallest superset of V' weakly control-closed in G.

Lemma 3 (Existence of the weak control-closure). *Let* $W = \mathsf{WD}_G(V') \cap \mathsf{R}_G(V')$ *be the set of vertices in* $\mathsf{WD}_G(V')$ *that are reachable from* V'. *Then* $V' \cup W$ *is the smallest weakly control-closed set containing* V'.

Proof. We prove that $V' \cup W$ is weakly control-closed in G. By Lemma 1, it is sufficient to show that there is no element of $\mathsf{WD}_G(V' \cup W)$ reachable from $V' \cup W$. Let u be an element of $\mathsf{WD}_G(V' \cup W)$ reachable from $V' \cup W$. By Prop. 2, since $V' \cup W \subseteq V' \cup \mathsf{WD}_G(V')$, we have

$$u \in \mathsf{WD}_G(V' \cup W) \subseteq V' \cup \mathsf{WD}_G(V') \cup \mathsf{WD}_G(V' \cup \mathsf{WD}_G(V')).$$

By Prop. 3, it follows that $u \in V' \cup \mathsf{WD}_G(V')$. Since u is reachable from $V' \cup W$, and all elements of W are reachable from V', u is reachable from V'. We can deduce that $u \in V' \cup W$. By Property 1, it is contradictory with the assumption $u \in \mathsf{WD}_G(V' \cup W)$. Thus $V' \cup W$ is weakly control-closed in G.

We now prove that $V' \cup W$ is included in any weakly control-closed set containing V'. Let X be a weakly control-closed set containing V' and let $u \in V' \cup W$. Let us show that $u \in X$. If $u \in V'$, we have $u \in X$ by assumption. Assume that $u \in W$. In particular, we have $u \in \mathsf{WD}_G(V')$. By Prop. 2, $\mathsf{WD}_G(V') \subseteq X \cup \mathsf{WD}_G(X)$. If u is in X, the proof is done. If $u \in \mathsf{WD}_G(X)$, u is an X-weakly deciding vertex reachable from $V' \cup W$ and thus from X, which contradicts X being weakly control-closed in G by Lemma 1. □

The previous lemma justifies the following definition.

Definition 5 (Weak control-closure). We call *weak control-closure* of V', denoted $\mathsf{WCC}_G(V')$, the smallest weakly control-closed set containing V'.

Property 4. *Let* V', V'_1 *and* V'_2 *be subsets of* V. *Then*

a) $\mathsf{WCC}_G(V') = V' \cup (\mathsf{WD}_G(V') \cap \mathsf{R}_G(V')) = (V' \cup \mathsf{WD}_G(V')) \cap \mathsf{R}_G(V')$.

b) If $V'_1 \subseteq V'_2$, *then* $\mathsf{WCC}_G(V'_1) \subseteq \mathsf{WCC}_G(V'_2)$.

c) If V' *is weakly control-closed, then* $\mathsf{WCC}_G(V') = V'$.

d) $\mathsf{WCC}_G(\mathsf{WCC}_G(V')) = \mathsf{WCC}_G(V')$.

Proof.

a) The equality follows from Lemma 3 and since $V' \subseteq \mathsf{R}_G(V')$.

b) Since $V'_1 \subseteq V'_2 \subseteq \mathsf{WCC}_G(V'_2)$ and $\mathsf{WCC}_G(V'_2)$ is weakly control-closed, we deduce $\mathsf{WCC}_G(V'_1) \subseteq \mathsf{WCC}_G(V'_2)$ by minimality of $\mathsf{WCC}_G(V'_1)$.

c) Obvious by Def. 5.

d) This is a direct consequence of c). □

4. Main Lemmas

This section gives two important lemmas used to justify both Danicic's algorithm and ours.

Lemma 4. *Let V' and W be two subsets of V. If $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$, then $W \cup \mathsf{WD}_G(W) = V' \cup \mathsf{WD}_G(V')$. Moreover, if $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$, then $\mathsf{WCC}_G(W) = \mathsf{WCC}_G(V')$.*

Proof. Assume $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$. Since $V' \subseteq W$, we have by Prop. 2, $\mathsf{WD}_G(V') \subseteq W \cup \mathsf{WD}_G(W)$. We can deduce $V' \cup \mathsf{WD}_G(V') \subseteq W \cup \mathsf{WD}_G(W)$. Furthermore, since $W \subseteq V' \cup \mathsf{WD}_G(V')$, we have by Prop. 2 and 3

$$\mathsf{WD}_G(W) \subseteq V' \cup \mathsf{WD}_G(V') \cup \mathsf{WD}_G(V' \cup \mathsf{WD}_G(V')) = V' \cup \mathsf{WD}_G(V').$$

We deduce $W \cup \mathsf{WD}_G(W) \subseteq V' \cup \mathsf{WD}_G(V')$, and finally the desired equality.

If now $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$, and therefore $\mathsf{R}_G(V') = \mathsf{R}_G(W)$, we deduce $\mathsf{WCC}_G(W) = \mathsf{WCC}_G(V')$ from the previous result by Prop. 4a. □

Lemma 4 allows to design iterative algorithms to compute the closure. Indeed, assume that we have a procedure which, given a set V' that is not weakly control-closed, can return one or more elements of the weak control-closure of V' that are not yet in V'. If we apply such a procedure to V' once, we get a set W that satisfies $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$. By Lemma 4, $\mathsf{WCC}_G(W) = \mathsf{WCC}_G(V')$. To compute the weak control-closure of V', it is thus sufficient to build the weak control-closure of W. We can apply our procedure again, this time to W, and iteratively on all the successively computed sets until the procedure fails to find any new elements (that is, the weak control-closure $\mathsf{WCC}_G(V')$ is found). Since each computed set is a strict superset of the previous one, this iterative process terminates because graph G is finite.

Before stating the second lemma, we introduce a key concept. It is called Θ in [8]. We use the name "observable" as in [16].

Definition 6 (Observable). Let $u \in V$. The set of *observable vertices* from u in V', denoted $\mathsf{obs}_G(u, V')$, is the set of vertices u' in V' such that $u \xrightarrow{V'\text{-path}} u'$.

Remark 3. A vertex $u \in V'$ is its unique observable: $\mathsf{obs}_G(u, V') = \{u\}$.

The concept of observable set was illustrated in Fig. 2 (cf. Sect. 2).

Lemma 5 (Sufficient condition for being V'-weakly deciding). *Let (u, v) be an edge in G such that $u \notin V'$, v can reach V' and there exists a vertex u' in V' such that $u' \in \mathsf{obs}_G(u, V')$ and $u' \notin \mathsf{obs}_G(v, V')$. Then $u \in \mathsf{WD}_G(V')$.*

Proof. Since v can reach V', there exists a V'-path π from v to some node $v' \in V'$, possibly trivial if $v \in V'$. We need to exhibit two V'-paths from u that share no vertex except u. We take the V'-path from u to u' as the first one, and the V'-path $u\pi$ from u to v' as the second one. If these V'-paths intersected at a node y different from u, we would have a V'-path from v to u' by concatenating the subpaths from v to y and from y to u', which is contradictory. □

Example 6. In G_0, $\mathsf{obs}_{G_0}(u_0, V_0') = \{u_1, u_3\}$ and $\mathsf{obs}_{G_0}(u_1, V_0') = \{u_1\}$ (cf. Fig. 2a). Since u_1 is a successor of u_0, we can apply Lemma 5, and deduce that u_0 is V_0'-weakly deciding. $\mathsf{obs}_{G_0}(u_5, V_0') = \{u_1, u_3\}$ and $\mathsf{obs}_{G_0}(u_6, V_0') = \{u_1, u_3\}$. We cannot apply Lemma 5 to u_5, and for good reason, since u_5 is not V_0'-weakly deciding. But we cannot apply Lemma 5 to u_6 either, since u_6 and all its successors u_0, u_4 and u_5 have observable sets $\{u_1, u_3\}$ w.r.t. V_0', while u_6 is V_0'-weakly deciding. This shows that with Lemma 5, we have a sufficient condition, but not a necessary one, for proving that a vertex is weakly deciding.

5. Danicic's Algorithm

We present here the algorithm described in [8]. This algorithm and a proof of its correctness have been formalized in Coq [14]. The algorithm is nearly completely justified by a following lemma (Lemma 6, equivalent to [8, Lemma 60]).

We first need to introduce a new concept, which captures edges that are of particular interest when searching for weakly deciding vertices. This concept is taken from [8], where it was not given a name. We call such edges *critical edges.*

Definition 7 (Critical edge). An edge (u, v) in G is called V'*-critical* if:

(1) $|\,\mathsf{obs}_G(u, V')| \geqslant 2$;

(2) $|\,\mathsf{obs}_G(v, V')| = 1$;

(3) u is reachable from V' in G.

Example 7. In G_0, (u_0, u_1), (u_2, u_3) and (u_4, u_3) are the V_0'-critical edges.

Lemma 6. *If V' is not weakly control-closed in G, then there exists a V'-critical edge in G. Moreover, if (u, v) is such a V'-critical edge, then $u \in \mathsf{WD}_G(V') \cap \mathsf{R}_G(V')$, therefore $u \in \mathsf{WCC}_G(V')$.*

Proof. By Lemma 1, there exists a vertex x in $\mathsf{WD}_G(V')$ reachable from V'. Let π be a V'-path from x ending in $x' \in V'$. It follows that $|\,\mathsf{obs}_G(x, V')| \geqslant 2$ and $|\,\mathsf{obs}_G(x', V')| = 1$. Let u be the last vertex on π with at least two observable nodes in V' and v its successor on π. Then (u, v) is a V'-critical edge.

Assume now that (u, v) is a V'-critical edge. From $|\,\mathsf{obs}_G(u, V')| \geqslant 2$ and $|\,\mathsf{obs}_G(v, V')| = 1$, we deduce that $u \notin V'$, v can reach V' and there exists u' in $\mathsf{obs}_G(u, V')$ but not in $\mathsf{obs}_G(v, V')$. By Lemma 5, $u \in \mathsf{WD}_G(V')$ and thus $u \in \mathsf{WCC}_G(V')$. □

Remark 4. We can see in the proof above that we do not need the exact values 2 and 1. We just need strictly more observable vertices for u than for v and at least one observable for v, to satisfy the hypotheses of Lemma 5.

As described in Sect. 4, we can build an iterative algorithm constructing the weak control-closure of V' by searching for critical edges on the intermediate sets built successively. This is the idea of Danicic's algorithm shown as Algorithm 1.

Input: $G = (V, E)$ a directed graph
$V' \subseteq V$
Output: $W \subseteq V$ the weak control-closure of V'
Ensures: $W = \mathsf{WCC}_G(V')$

```
1 begin
2 │ W ← V'
3 │ while there exists a W-critical edge in E do
4 │ │ choose such a W-critical edge (u, v)
5 │ │ W ← W ∪ {u}
6 │ end
7 │ return W
8 end
```

Algorithm 1: Danicic's original algorithm for weak control-closure [8]

Example 8. Let us apply Algorithm 1 to our running example G_0 (cf. Fig. 1). Note that in G_0, every node is reachable from V'_0, thus the reachability conditions always hold and can be ignored in this example. Initially, $W_0 = V'_0 = \{u_1, u_3\}$.

1. $\mathsf{obs}_{G_0}(u_0, W_0) = \{u_1, u_3\}$ and $\mathsf{obs}_{G_0}(u_1, W_0) = \{u_1\}$, therefore (u_0, u_1) is a W_0-critical edge. Set $W_1 = \{u_0, u_1, u_3\}$.

2. $\mathsf{obs}_{G_0}(u_2, W_1) = \{u_0, u_3\}$ and $\mathsf{obs}_{G_0}(u_3, W_1) = \{u_3\}$, therefore (u_2, u_3) is a W_1-critical edge. Set $W_2 = \{u_0, u_1, u_2, u_3\}$.

3. $\mathsf{obs}_{G_0}(u_4, W_2) = \{u_0, u_3\}$ and $\mathsf{obs}_{G_0}(u_3, W_2) = \{u_3\}$, therefore (u_4, u_3) is a W_2-critical edge. Set $W_3 = \{u_0, u_1, u_2, u_3, u_4\}$.

4. $\mathsf{obs}_{G_0}(u_6, W_3) = \{u_0, u_4\}$ and $\mathsf{obs}_{G_0}(u_0, W_3) = \{u_0\}$, therefore (u_6, u_0) is a W_3-critical edge. Set $W_4 = \{u_0, u_1, u_2, u_3, u_4, u_6\}$.

5. There is no W_4-critical edge. $\mathsf{WCC}_{G_0}(V'_0) = W_4 = \{u_0, u_1, u_2, u_3, u_4, u_6\}$.

Proof of Algorithm 1. We denote by W_i the value of W before iteration $i + 1$. To establish the correction of the algorithm, we can prove by induction that W_i satisfies $V' \subseteq W_i \subseteq \mathsf{WCC}_G(V')$ at any step i. If $i = 0$, $W_0 = V'$, and both relations trivially hold. Let $i \geqslant 0$ and assume that $V' \subseteq W_i \subseteq \mathsf{WCC}_G(V')$ and there exists a W_i-critical edge (u, v). We have $W_{i+1} = W_i \cup \{u\}$. The inclusion $V' \subseteq W_{i+1}$ is obvious. By Lemma 6, $u \in \mathsf{WCC}_G(W_i)$. Therefore, by Lemma 4, $u \in \mathsf{WCC}_G(V')$, and thus, $W_{i+1} \subseteq \mathsf{WCC}_G(V')$.

At the end of the algorithm, there is no W-critical edge, therefore W is weakly control-closed by Lemma 6. Since $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$, we have $W = \mathsf{WCC}_G(V')$ by definition of the weak control-closure (Definition 5). Termination of the loop follows from the fact that W strictly increases and is upper-bounded by $\mathsf{WCC}_G(V')$. □

In terms of complexity, [8] shows that, assuming that the degree of each vertex is at most 2 (and thus that $O(|V|) = O(|E|)$), the complexity of the

algorithm is $O(|V|^3)$. Indeed, the main loop of Algorithm 1 is run at most $O(|V|)$ times, and each loop body computes obs in $O(|V|)$ for at most $O(|V|)$ edges.

Remark 5. We propose two optimizations for Algorithm 1:

(opt1) considers, at each step, all critical edges rather than only one (cf. lines 4–5 of Algorithm 1), and adds their origins to W;

(opt2) uses the weaker definition of critical edge suggested in Remark 4.

Example 9. We can replay Algorithm 1 using **(opt1)**. This run corresponds to the steps shown in Fig. 2. Initially, $W_0 = V'_0 = \{u_1, u_3\}$.

1. (u_0, u_1), (u_2, u_3), (u_4, u_3) are W_0-critical edges. Set $W_1 = \{u_0, u_1, u_2, u_3, u_4\}$.
2. (u_6, u_0) is a W_1-critical edge. Set $W_2 = \{u_0, u_1, u_2, u_3, u_4, u_6\}$.
3. There is no W_2-critical edge in G_0. $\mathsf{WCC}_{G_0}(V'_0) = W_2$.

Using **(opt1)**, we can compute the weak control-closure of V'_0 in G_0 in only 2 iterations instead of 4. This run also demonstrates that the algorithm is necessarily iterative: even when considering all V'_0-critical edges in the first step, u_6 is not detected before the second step.

One benefit of formalizing Danicic's algorithm in Coq is that it allows us to extract a certified implementation of the algorithm in the OCaml programming language. We will use such an extraction in our experiments (see Sect. 7).

6. Our New Algorithm: lDFS

6.1. Overview

A potential source of inefficiency in Danicic's algorithm is the fact that no information is shared between the iterations. The observable sets are recomputed at each iteration since the target set W changes. This is the reason why **(opt1)** proposed in Remark 5 is interesting, because it allows to work longer on the same target set and thus to reuse the observable sets.

We propose now to go even further: to store some information about the paths in the graph and reuse it in the *following* iterations. The main idea of the proposed algorithm is to label each processed node u with a node $v \in W$ observable from u in the resulting set W being progressively constructed by the algorithm. Labels survive through iterations and can be reused. As we will see, the labeling is based on iterated reverse DFS traversals, explaining the name we chose for this algorithm, lDFS.

Unlike Danicic's algorithm, ours does not directly compute the weak control-closure. It actually computes the set $W = V' \cup \mathsf{WD}_G(V')$. To obtain the closure $\mathsf{WCC}_G(V') = W \cap \mathsf{R}_G(V')$, W is then simply filtered to keep only vertices reachable from V' (cf. Prop. 4a).

In addition to speeding up the algorithm, the usage of labels brings another benefit: at the end of the algorithm, for each node of G, its label indicates its observable vertex in W (when it exists). Recall that since $\mathsf{WD}_G(W) = \varnothing$ (by Property 3) at the end of the algorithm, each node in the graph has at most one observable vertex in W.

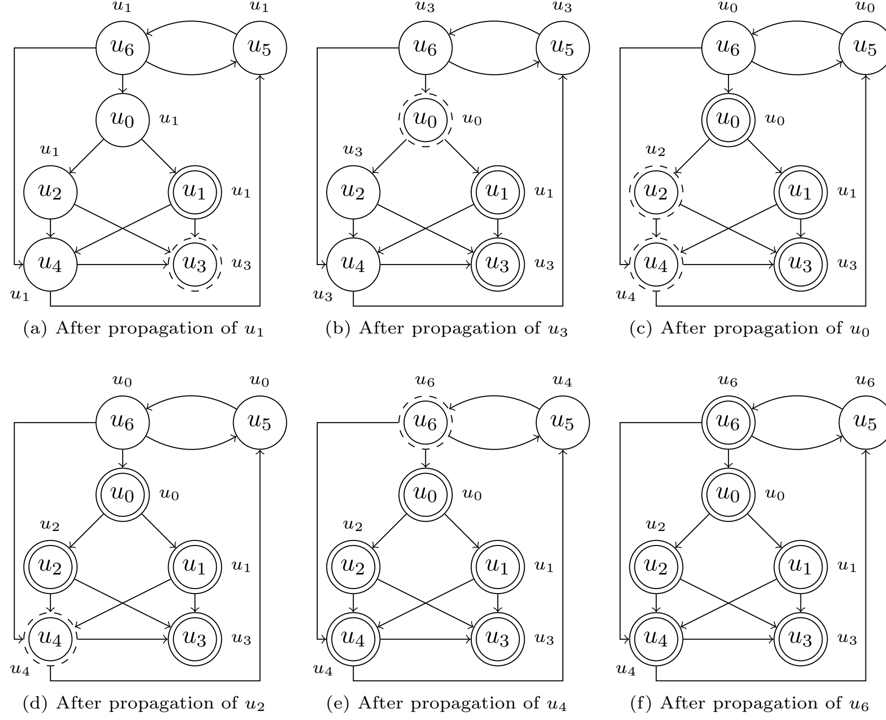
One difficult point with this approach is that the labels of the nodes need to be refreshed with care at each iteration so that they remain up-to-date. Actually, lDFS does not ensure that at each iteration the label of each node is an observable vertex from this node in W. This state is only ensured at the beginning and at the end of the algorithm. Meanwhile, some nodes are still in the worklist and some labels are wrong, but this does not prevent lDFS from working.

6.2. Informal Description

lDFS is given a directed graph G and a subset of vertices V' in G. It manipulates three objects: a set W which is equal to V' initially, which grows during the algorithm and which at the end contains the result, $V' \cup \mathsf{WD}_G(V')$; a partial mapping *obs* associating at most one label $obs[u]$ to each node u in the graph, this label being a vertex in W reachable from this node (and which is the observable from u in $V' \cup \mathsf{WD}_G(V')$ at the end); a worklist L of nodes of the closure not processed yet. Each iteration proceeds as follows. If the worklist is not empty, a vertex u is extracted from it. All the vertices that transitively precede vertex u in the graph and that are not hidden by vertices in W are labeled with u. During the propagation, nodes that are good candidates to be V'-weakly deciding are accumulated. After the propagation, we filter them so that only true V'-weakly deciding nodes are kept. Each of these vertices is associated to itself in *obs*, and is added to W and L. If L is not empty, a new iteration begins. Otherwise, W is equal to $V' \cup \mathsf{WD}_G(V')$ and *obs* associates each node in the graph with its observable vertex in the closure (when it exists).

Note that each iteration consists in two steps: a complete backward propagation in the graph, which collects potential V'-weakly deciding vertices, and a filtering step. The set of predecessors of the propagated node are thus filtered twice: once during the propagation and once afterwards. We can try to filter as much as possible in the first step or, at the opposite, to avoid filtering during the first step and do all the work in the second step. For the sake of simplicity of mechanized proof, the version we chose does only simple filtering during the first step. We accumulate in our candidate V'-weakly deciding nodes all nodes that have at least two successors and a label different from the one currently propagated, and we eliminate the false positives in the second step, once the propagation is done.

Example 10. Let us use our running example (cf. Fig. 1) to illustrate lDFS. The successive steps are represented in Fig. 7. In the different figures, nodes in W already processed (that is, in $W \backslash L$) are represented using a solid double circle (u_i), while nodes in W not already processed (that is, still in worklist L)



(a) After propagation of u_1 (b) After propagation of u_3 (c) After propagation of u_0

(d) After propagation of u_2 (e) After propagation of u_4 (f) After propagation of u_6

Figure 7: lDFS applied on $G = G_0$ and $V' = V'_0 = \{u_1, u_3\}$

are represented using a dashed double circle (u_i). A label u_j next to a node u_i (u_i u_j) means that u_j is associated to u_i, i.e. $obs[u_i] = u_j$. Let us detail the first steps of the algorithm. Initially, $W_0 = V'_0 = \{u_1, u_3\}$ (cf. Fig. 1).

1. u_1 is selected and is propagated backwards from u_1 (cf. Fig. 7a). We find no candidate, the first iteration is finished, $W_1 = \{u_1, u_3\}$.

2. u_3 is selected and is propagated backwards from u_3 (cf. Fig. 7b). u_0, u_2, u_4 and u_6 are candidates. The successors of u_2, u_4 and u_6 (u_3 and u_4, u_3 and u_5, and u_0 and u_4, respectively) are all labeled with u_3, thus those three nodes are filtered out. Only u_0, whose successor u_1 is labeled with $u_1 \neq u_3$, is confirmed as a V'_0-weakly deciding node. It is stored in worklist L and its label is set to u_0. Now $W_2 = \{u_0, u_1, u_3\}$.

3-6. u_0, u_2, u_4 and u_6 are processed similarly (cf. Fig. 7c, 7d, 7e, 7f). At the end, we get $W_6 = \{u_0, u_1, u_2, u_3, u_4, u_6\} = V'_0 \cup \mathsf{WD}_G(V'_0)$.

As all nodes in W_6 are already reachable from V'_0, $W_6 = \mathsf{WCC}_G(V'_0)$.

We can make two remarks on this example. First, as we can see in Fig. 7f, each node is labeled with its observable in W at the end of the algorithm.

Input: $G = (V, E)$ a directed graph
obs : Map(V, V) associating at most one label to each vertex of G
$u, v \in V$ vertices in G
Output: b : *bool*
Ensures: $b = true \iff \exists u', (u, u') \in E \wedge u' \in obs \wedge obs[u'] \neq v$

Algorithm 2: Contract of `confirm` (G, obs, u, v)

Second, in Fig. 7e, we have the case of a node labeled with an obsolete label, since u_5 is labeled u_4 while its only observable node in W is u_6.

6.3. Detailed Description

lDFS is split into three functions:

- `confirm` is used to check if a given node is V'-weakly deciding by trying to find a successor with a different label from its own label given as an argument.
- `propagate` takes a vertex and propagates backwards a label over its predecessors. It returns a set of candidate V'-weakly-deciding nodes.
- `main` calls `propagate` on a node of the closure not yet processed, gets candidate V'-weakly deciding nodes, calls `confirm` to keep only true V'-weakly deciding nodes, adds them to the closure and updates their labels, and loops until no more V'-weakly deciding nodes are found.

Function confirm. A call to `confirm`(G, obs, u, v) takes four arguments: a graph G, a labeling of graph vertices obs, and two vertices u and v. It returns *true* if and only if at least one successor u' of u in G has a label in obs different from v, which can be written $u' \in obs \wedge obs[u'] \neq v$. This simple function is left abstract here for lack of space. The Why3 formalization [14] contains a complete proof. Its contract is given as Algorithm 2.

Function propagate. A call to `propagate`(G, W, obs, u, v) takes five arguments: a graph G, a subset W of nodes of G, a labeling of nodes obs, and two vertices u and v. It traverses G backwards from u (stopping at nodes in W) and updates obs so that all predecessors not hidden by vertices in W have label v at the end of the function. It returns a set of potential V'-weakly deciding vertices. Again, this function is left abstract here but is proved in the Why3 development [14]. Its contract is given as Algorithm 3.

`propagate` requires that, when called, only u is labeled with v (P_1) and that $u \in W$ (P_2). It ensures that, after the call, all the predecessors of u not hidden by a vertex in W are labeled v (Q_1), the labels of the other nodes are unchanged (Q_2), C contains only predecessors of u but not u itself (Q_3), and all the predecessors that had a label before the call (different from v due to P_1) and that have at least two successors are in C (Q_4).

Input: $G = (V, E)$, $W \subseteq V$, $obs : \mathrm{Map}(V, V)$, $u, v \in V$
Output: obs', a new version of obs
$C \subseteq V$ containing candidate W-weakly deciding nodes
Requires: $(\mathbf{P_1})$ $\forall z \in V, obs[z] = v \iff z = u$
Requires: $(\mathbf{P_2})$ $u \in W$
Ensures: $(\mathbf{Q_1})$ $\forall z \in V, z \xrightarrow{W\text{-path}} u \implies obs'[z] = v$
Ensures: $(\mathbf{Q_2})$ $\forall z \in V, \neg(z \xrightarrow{W\text{-path}} u) \implies obs'[z] = obs[z]$
Ensures: $(\mathbf{Q_3})$ $\forall z \in C, z \neq u \wedge z \xrightarrow{W\text{-path}} u$
Ensures: $(\mathbf{Q_4})$ $\forall z \in V, z \neq u \wedge z \xrightarrow{W\text{-path}} u \wedge z \in obs$
$\wedge |\mathsf{succ}_G(z)| > 1 \implies z \in C$
Algorithm 3: Contract of `propagate` (G, W, obs, u, v)

Function main. The main function of our algorithm is given as Algorithm 4. It takes two arguments: a graph G and a subset of vertices V'. It returns $V' \cup \mathsf{WD}_G(V')$ and a labeling associating to each node its observable vertex in this set if it exists. It maintains a worklist L of vertices that must be processed. L is initially set to V', and their labels to themselves (line 2). If L is not empty, a node u is taken from it and `propagate`(G, W, obs, u, u) is called (lines 3–5). It returns a set of candidate V'-weakly deciding nodes (C) that are not added to W yet. They are first filtered using `confirm` (lines 6–10). The confirmed nodes (Δ) are then added to W and to L, and the label of each of them is updated to itself (line 11). The iterations stop when L is empty (cf. lines 3, 13).

6.4. Proof of Correction of lDFS

We opted for Why3 instead of Coq for this proof to take advantage of Why3's automation. Indeed, most of the goals could be discharged in less than a minute using Alt-Ergo, CVC4, Z3 and E. Some of them still needed to be proved manually in Coq, resulting in 330 lines of Coq proof. The Why3 development [14] focuses on the proof of the algorithm, not on the concepts presented in Sect. 3 and 4. Most of the concepts are proved, one of them is assumed in Why3 but was proved in Coq previously. Due to lack of space, we detail here only the main invariants necessary to prove `main` (cf. Algorithm 4). The proofs of I_1, I_2, I_3, I_4 are rather simple. while those of I_5 and I_6 are more complex.

I_1 states that each node in W has itself as a label. It is true initially for all nodes in V' and is preserved by the updates.

I_2 states that all labels are in W. This is true initially since all labels are in V'. The preservation is verified, since all updates are realized using labels in W.

I_3 states that labels in L have not been already propagated. Given a node y in L, y is the only node whose label is y. It is true initially since every vertex in V' has itself as a label. After an update, the new nodes obey the same rule, so I_3 is preserved.

I_4 states that if label z is associated to a node y then there exists a path

Input: $G = (V, E)$, a directed graph
$V' \subseteq V$, the input subset
Output: $W \subseteq V$, the main result
$obs : \mathrm{Map}(V, V)$, the final labeling
Variables: $L \subseteq V$, a worklist of nodes to be treated
$C \subseteq V$, a set of candidate V'-weakly deciding vertices
$\Delta \subseteq V$, a set of new V'-weakly deciding vertices
Ensures: $W = V' \cup \mathsf{WD}_G(V')$
Ensures: $\forall u, v \in V, obs[u] = v \iff v \in \mathsf{obs}_G(u, W)$

```
1  begin
2      W ← V' ; obs|V' ← idV' ; L ← V'                      // initialization
3      while L ≠ ∅ do                                       // main loop
           // invariant:  I1 ∧ I2 ∧ I3 ∧ I4 ∧ I5 ∧ I6
           // variant:  cardinal(L ∪ V\W)
4          u ← choose(L) ; L ← L\{u}
5          C ← propagate (G, W, obs, u, u)                  // propagation
6          Δ ← ∅
7          while C ≠ ∅ do                                   // filtering
8              v ← choose(C) ; C ← C\{v}
9              if confirm (G, obs, v, u) = true then  Δ ← Δ ∪ {v}
10         end
11         W ← W ∪ Δ ; obs|Δ ← idΔ ; L ← L ∪ Δ              // update
12     end
       // assert:  A1 ∧ A2 ∧ A3 ∧ A4
13     return (W, obs)
14 end
```

($\mathbf{I_1}$) $\forall z \in W, obs[z] = z$

($\mathbf{I_2}$) $\forall y, z \in V, obs[y] = z \implies z \in W$

($\mathbf{I_3}$) $\forall y, z \in V, obs[y] = z \wedge z \in L \implies y = z$

($\mathbf{I_4}$) $\forall y, z \in V, obs[y] = z \implies y \xrightarrow{\text{path}} z$

($\mathbf{I_5}$) $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$

($\mathbf{I_6}$) $\forall y, z, z' \in V, y \xrightarrow{W\text{-disjoint}} z \wedge obs[z] = z' \wedge z' \notin L \implies obs[y] = z'$

($\mathbf{A_1}$) $\forall u, v \in V, v \in \mathsf{obs}_G(u, W) \implies obs[u] = v$

($\mathbf{A_2}$) $\mathsf{WD}_G(W) = \varnothing$

($\mathbf{A_3}$) $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$

($\mathbf{A_4}$) $W = V' \cup \mathsf{WD}_G(V')$

Algorithm 4: Function `main` with annotations

between y and z. Initially, there exist trivial paths from each node in V' to itself. When obs is updated, there exists a W-path, thus in particular a path.

I_5 states that W remains between V' and $V' \cup \mathsf{WD}_G(V')$ during the execution of the algorithm. The first part $V' \subseteq W$ is easy to prove, because it is true initially and W is growing. For the second part, we need to prove that after the filtering, $\Delta \subseteq V' \cup \mathsf{WD}_G(V')$. For that, it is sufficient to prove that $\Delta \subseteq \mathsf{WD}_G(W)$ thanks to Lemma 4. Let v be a node in Δ. Since $\Delta \subseteq C$, we know that $v \notin W$ and $u \in \mathsf{obs}_G(v, W)$. Moreover, we have $\texttt{confirm}(G, obs, v, u) = true$, i.e. v has a successor v' such that $v' \in obs$, hence v can reach W by I_2 and I_4, and $obs[v'] \neq u$, hence $u \notin \mathsf{obs}_G(v', W)$. We can apply Lemma 5 and deduce that $v \in \mathsf{WD}_G(W)$.

I_6 is the most complicated invariant. I_6 states that if there is a path between two vertices y and z that does not intersect W, and z has a label already processed, then y and z have the same label. Let us give a sketch of the proof of preservation of I_6 after an iteration of the main loop. Let us note obs' the map at the end of the iteration. Let $y, z, z' \in V$ such that $y \xrightarrow{(W \cup \Delta)\text{-disjoint}} z$, $obs'[z] = z'$ and $z' \notin (L \backslash \{u\}) \cup \Delta$. Let us show that $obs'[y] = z'$. First, observe that neither y nor z can be in Δ. Indeed, if z is in Δ, $obs'[z] = z$ per line 11 of Algorithm 4, thus $z' = z$, whence z' is in Δ, which is contradictory. Likewise, if y is in Δ, the $(W \cup \Delta)$-disjoint path from y to z is empty, thus $z = y$, whence z is in Δ. By the same reasoning we have just followed, we get a contradiction. We examine four cases depending on whether the conditions $z \xrightarrow{W\text{-path}} u$ (H_1) and $y \xrightarrow{W\text{-path}} u$ (H_2) hold.

- $H_1 \wedge H_2$: Both z and y were given the label u during the last iteration, thus $obs'[z] = obs'[y] = u$ as expected.

- $H_1 \wedge (\neg H_2)$: This case is impossible, since $y \xrightarrow{(W \cup \Delta)\text{-disjoint}} z$.

- $(\neg H_1) \wedge (\neg H_2)$: Both z and y have the same label as before the iteration. We can therefore conclude by I_6 at the beginning of the iteration.

- $(\neg H_1) \wedge H_2$:. This is the only complicated case. We show that it is contradictory. For that, we introduce v_1 as the last vertex on the $(W \cup \Delta)$-disjoint path connecting y and z which is also the origin of a W-path to u, and v_2 as its successor on this $(W \cup \Delta)$-disjoint path. We can show that $v_1 \in \Delta$, which contradicts the fact that it lives on a $(W \cup \Delta)$-disjoint path.

We can now prove the assertions A_1, A_2, A_3 and A_4 at the end of `main`. A_1 is a direct consequence of I_6 since at the end $L = \varnothing$. A_1 implies that each vertex u has at most one observable in W: $obs[u]$ if $u \in obs$. A W-weakly deciding vertex would have at least two observables, thus $\mathsf{WD}_G(W) = \varnothing$. A_3 is a direct consequence of I_5. A_4 can be deduced from A_2 and Lemma 4 applied to A_3. This proves that at the end $W = V' \cup \mathsf{WD}_G(V')$. To prove the other post-condition, we must prove that if there are two nodes u, v such that $obs[u] = v$,

then $v \in \mathsf{obs}_G(u, W)$. By I_4, there is a path from u to v. Let w be the first element in W on this path. Then $u \xrightarrow{W\text{-path}} w$. By A_1, $obs[u] = w$. Thus, $w = v$ and $u \xrightarrow{W\text{-path}} v$. This proves the second post-condition. □

7. Experiments

In this section, we compare experimentally lDFS, presented in Sect. 6, and Danicic's algorithm presented in Sect. 5, to measure experimentally whether our proposition is, as we claim, optimized with respect to the latter. In addition to the comparison of both algorithms, we also compare several variants of Danicic's algorithm with each other: the extraction of the Coq formalization of Danicic's algorithm mentioned in Sect. 5 and the main implementation in OCaml, possibly improved by the two optimizations **(opt1)** and **(opt2)** mentioned in Remark 5, and that we recall below:

- **(opt1)** consists in detecting as many critical edges as possible during each iteration;
- **(opt2)** consists in weakening the definition of critical edge.

More precisely, we want to answer the following research questions:

- **(RQ1)** How the OCaml hand-written implementations of Danicic's algorithm compare with the extraction of the Coq certified version?
- **(RQ2)** What are the impacts of **(opt1)** and **(opt2)** on the performance of Danicic's algorithm?
- **(RQ3)** Does lDFS outperform the implementations of Danicic's algorithm?

First, Sect. 7.1 presents the implementations that we used in the experiments. Next, Sect. 7.2 describes the testing procedure. The results obtained are presented and analyzed in Sect. 7.3.

7.1. Implementations

We make general remarks before presenting in detail each implementation.

General remarks. All the implementations are written in the same language: OCaml. The only odd case is the Coq extraction which is not manually written in OCaml. However, the version that is compiled and run is the extraction into OCaml, thus it can also be seen as OCaml code. Admittedly, since the extraction into OCaml produces more complex code than what we would write directly in OCaml, this penalizes the Coq extraction with respect to the other implementations.

All hand-written OCaml implementations, i.e. all but the Coq extraction, are written using the same generic graph library called OCamlgraph (version 1.8.8) [13]. This library provides convenient functions to access the predecessors or the successors of a given node and to traverse the whole graph. It also

comes with high-level operations, such as cloning of a graph, reachability tests and dataflow analyses. This library allows to write the implementations in a concise way, which makes them relatively close to the high-level descriptions given in this article. OCamlgraph allows to choose between multiple graph implementations: imperative, i.e. mutable, or persistent, i.e. immutable; directed or undirected edges; standard (every vertex can easily access its successors, but not its predecessors) or bidirectional (every vertex can easily access both its successors and its predecessors). Initially, we used a standard directed imperative implementation.[1] But since both algorithms are based on backward traversal, we switched to a bidirectional directed imperative implementation, that takes more memory space, but in which accesses to predecessors are claimed to be in constant time instead of being linearly dependent on the size of the graph.

Another fact that we must take into account is that Danicic's algorithm and lDFS do not compute the same set. Indeed, recall that Danicic's algorithm computes from an initial subset V' its weak control-closure $\mathsf{WCC}_G(V') = V' \cup (\mathsf{WD}_G(V') \cap \mathsf{R}_G(V'))$, while lDFS computes the bigger set $V' \cup \mathsf{WD}_G(V')$. To make the comparison fairer, we ensure that the implementation of lDFS also computes the weak control-closure, by adding to it a filtering step at the end that preserves only the vertices reachable from the initial subset V', as suggested in Sect. 6.

In all the implementations of Danicic's algorithm, including the Coq extraction, the computation of the observable vertices is performed as described in [8]. Given a graph G, a subset of vertices V' and a node u in G, the set of observable vertices from u in V' is computed by removing from G all the outgoing edges from nodes in V' and selecting all the nodes in V' that are reachable from u in the resulting graph H. Removing the edges that have their sources in V' guarantees that the nodes in V' that are reachable from u in H are first-reachable from u in V' in graph G, i.e. are observable from u in V' in graph G.

The Coq extraction. The first implementation that we consider is the Coq extraction of Danicic's algorithm. Recall that it is the extraction of the implementation in Coq of Danicic's algorithm based on a prototype graph library presented in [17]. It is a particularly naive implementation. Indeed, as discussed in Sect. 6, the main weakness of Danicic's algorithm is the absence of propagation of information between iterations, and the Coq extraction, as any implementation of Danicic's algorithm, inherits this weakness. What makes it really naive is that it propagates nearly no intermediate information even inside each iteration. For example, during an iteration, the calculations of the observable sets are independent while they could take advantage of each other. It is not completely naive, though, since it implements **(opt1)** (but not **(opt2)**).

OCaml implementations of Danicic's algorithm. We implemented Danicic's algorithm in OCaml, using OCamlgraph. This implementation is much smarter

[1] The experimental results of [15] were obtained using that graph representation.

than the Coq extraction, taking advantage of caching in several places. For example, we compute once at the beginning of the algorithm the set of nodes reachable from V' and use this set for every reachability check needed later in the algorithm. Moreover, at the beginning of an iteration, we compute simultaneously for every node in G its observable set in W using the helper functions provided by OCamlgraph to write a dataflow analysis. This analysis computes the observable set of a node from the observable sets of its successors, which allows to compute the observable information for the whole graph G rather efficiently.

There are four variants of this implementation: without the two optimizations, with **(opt1)** only, with **(opt2)** only, and with both.

OCaml implementation of lDFS. We implemented lDFS directly in OCaml, using OCamlgraph, following closely the presentation of Section 6. Contrary to the implementations of Danicic's algorithm, this implementation uses only the low-level functions of OCamlgraph such as the ones iterating over the successors and the predecessors of a vertex.

7.2. Description of the Testing Procedure

This section presents the methodology we adopted for the experimentation.

The implementations are run on graphs that are randomly generated using OCamlgraph. More precisely, we use the function `Rand.graph` of OCamlgraph that takes as parameters a number v of vertices and a number e of edges and generates a random graph with v vertices and e edges. For all the experiments, we set the number of edges to twice the number of vertices. The initial set V' consists in three vertices randomly taken in the graph. These choices are not made to get realistic graphs, but to get cases where computing weak control-closure is often not trivial, i.e. cases where the weak control-closure is often not reduced to the initial subset V'. We informally confirmed that fact during our experiments, since in most of the cases the weak control-closure is not reduced to the initial V'. Moreover, in these cases, the resulting closure nearly always represents a significant part of the set of vertices of the graph. The exact choice of 2 as the ratio between the number of vertices and the number of edges is discussed at the end of Section 7.3.

Before even running the first experiments, we wanted to obtain some guarantees that our implementations are correct. Indeed, among our implementations, only the Coq extraction is certified. For that, we ran all of them on random small graphs (typically 30 nodes, and thus 60 edges) and checked that they all computed the same weak control-closure, and in particular the same results as the certified Coq extraction. Moreover, during the experiments, when multiple implementations were run on the same graph, we systematically checked that the computed results were identical. This does not prove that all the implementations are correct, but greatly increases our confidence that they are.

The experimentation consisted in running each implementation on hundreds of random graphs. The parameters used vary on the implementations. For expensive implementations, such as the Coq extraction, the graphs contain a

few dozens of nodes and the step between each graph size tested is 10. For the most efficient implementations, the graphs contain up to hundreds of thousands of nodes and the step used is 10 000. The exact parameters used are given in Sect. 7.3. For each implementation and each size of graph tested, we run the implementation on 10 graphs of this size. As discussed above, for the majority of the 10 graphs, the weak control-closure is rather large. We decided to ignore the cases in which the weak control-closure is equal to the initial V' and thus contains only three vertices. Indeed, in these cases, the execution time is insignificant. The result for that implementation and that size of graph is the average of the running times in the cases where the weak control-closure is not trivial.

In terms of hardware, experiments have been performed on an Intel Core i7 4810MQ with 8 cores at 2.80 GHz and 16 GB RAM.

7.3. Results

This section presents the main results of the experiments.

Implementations are compared pairwise. In each diagram, the ordinate is the execution time in seconds and the abscissa is the number $v = |V|$ of nodes in the graph. Recall that the number of edges e is equal to $2 \times v$, except in the last experiment described below. For the sake of clarity, we adopt the following convention. In each diagram, the implementation that is expected to be slower is represented using red triangles, while the implementation that is expected to be faster is represented using blue plus signs.

Comparison between the Coq Extraction and Danicic Implementation. The first diagram that we present compares the Coq extraction and the implementation of Danicic's algorithm without any optimization. As discussed in Sect. 7.1, the Coq extraction implements **(opt1)**, which gives it an advantage over the implementation of Danicic's algorithm. However, the Coq extraction is written rather naively, in Coq, and using a prototype graph library, which has probably a much bigger negative impact than **(opt1)** has a positive impact on the performance.

Figure 8 shows the results. The Coq extraction was run on graphs of sizes equal to multiples of 10 between 10 and 150. Danicic's implementation was run on graphs of sizes equal to multiples of 10 between 10 and 1 600.

We observe that the Coq extraction explodes for barely more than 100 nodes, while the OCaml implementation can handle graphs with more than 1 000 nodes. This answers **(RQ1)**. As expected, the Coq extraction is particularly inefficient, while the implementation of Danicic's algorithm can handle some non-trivial graphs.

Impact of the Optimizations on Danicic's Implementation. We study the impact of **(opt1)** and **(opt2)** on the performance of the implementation of Danicic's algorithm. Since the implementation shares intermediate results in each iteration, starting a new iteration should be costly in comparison to performing more work in the same iteration. Thus **(opt1)** is expected to be really interesting. The interest of **(opt2)** is less clear.

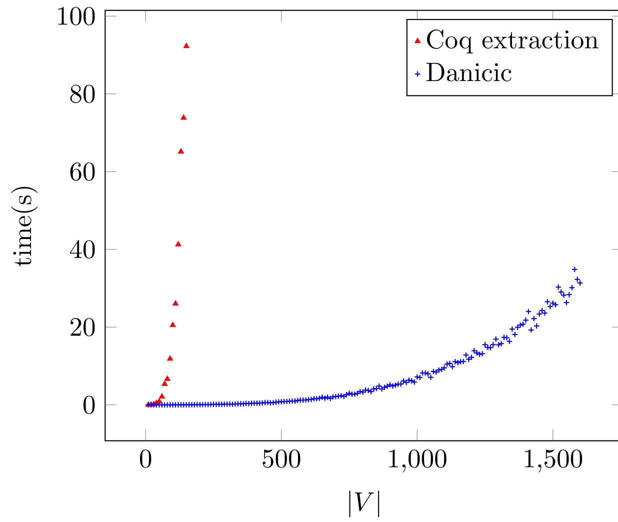


Figure 8: Comparison between the Coq extraction and the implementation in OCaml of Danicic's algorithm

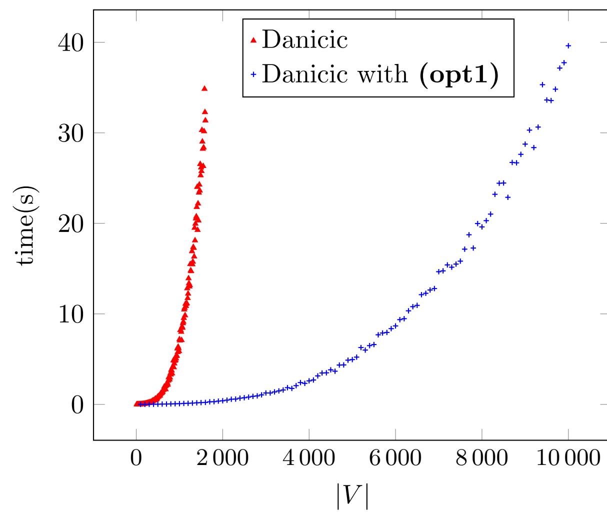


Figure 9: Comparison between two OCaml implementations of Danicic's algorithm, with and without **(opt1)**

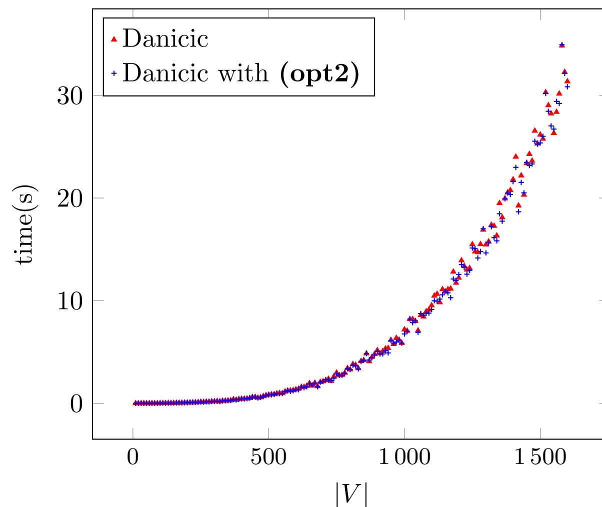


Figure 10: Comparison between two OCaml implementations of Danicic's algorithm, with and without **(opt2)**

Figure 9 shows the comparison of the implementation without any optimization and with **(opt1)**. The implementation without any optimization was run on graphs of sizes equal to multiples of 10 between 10 and and 1 600. The implementation with **(opt1)** was run on graphs of sizes equal to multiples of 100 between 100 and 10 000.

This partially answers **(RQ2)**. As expected, **(opt1)** significantly improves the performance of the implementation. The optimized variant can handle graphs with 10 000 nodes that are five times larger than the graphs on which the unoptimized variant reaches its limits.

Figures 10 and 11 show the impact of **(opt2)**, without and with **(opt1)** respectively. In Figure 10, both algorithms were run of graphs of sizes equal to multiples of 10 between 10 and 1 600. In Figure 11, both algorithms were run of graphs of sizes equal to multiples of 100 between 100 and 10 000.

In both cases, the impact of **(opt2)** is negligible, completing the answer to **(RQ2)**. In combination with **(opt1)**, though, it seems that, for large graphs of at least 6 000 nodes, the version with both optimizations is slightly faster than the version with **(opt1)** only. This can be explained as follows. Alone, **(opt2)** is not a great improvement, since its impact is mainly to change the order in which the nodes are detected. However, in combination with **(opt1)**, it allows to detect more nodes at each iteration, and thus to potentially reduce the number of iterations, resulting in the slight improvement observed in Figure 11.

Comparison between Danicic's algorithm and lDFS. The comparison between the implementation of Danicic's algorithm and the implementation of lDFS is

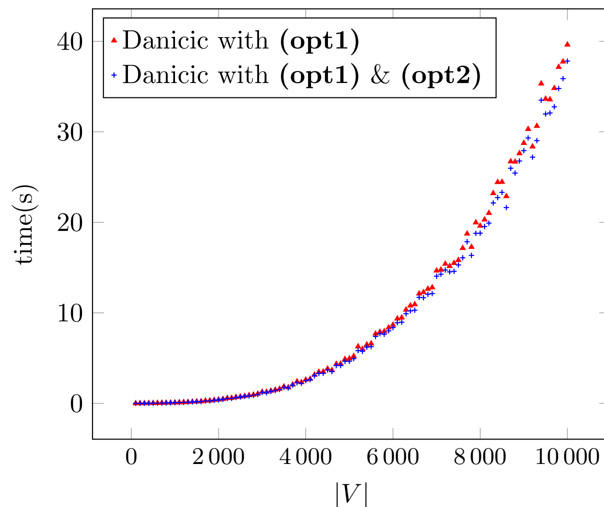


Figure 11: Comparison between two OCaml implementations of Danicic's algorithm, one with **(opt1)** and one with **(opt1)** and **(opt2)**

the main result of the experiments. We choose the OCaml implementation with **(opt1)** and **(opt2)** as implementation of Danicic's algorithm.

Figure 12 shows the results. The implementation of Danicic's algorithm was run on graphs of sizes equal to multiples of 100 between 100 and 10 000, while the implementation of lDFS was run on graphs of sizes equal to multiples of 100 between 100 and 30 000.

As already discussed above, Danicic's algorithm with **(opt1)** and **(opt2)** can handle graphs with 10 000 nodes. But for the largest graphs, it takes 40 s to execute. There is no comparison with lDFS that takes less than 1 s even for graphs with 30 000 nodes. This allows to answer **(RQ3)** positively.

Impact of the Number of Edges on the Experiments. The last experiment that we present does not compare another implementation. It was performed to give confidence in the experimentation. Indeed, the graphs that are used in the experiments are randomly generated by OCamlgraph with twice as many edges as vertices. The results obtained in this setting may not be representative of the general case. For example, the random generator of OCamlgraph may introduce some bias in the results. We did not check whether it is the case. But at least we wanted to question the choice of 2 as the ratio between the number of edges e and the number of vertices v that seems rather arbitrary. Actually, it is somewhat arbitrary, since it has no precise meaning, but was chosen intuitively so that most of the generated graphs give a rather large weak control-closure. To check that the results are not completely different with fewer edges, we reproduced two experiments with the smaller ratio $e/v = 1.5$.

Figure 13 shows the results for the OCaml implementation of Danicic's algorithm. The implementation was run on graphs with $e/v = 2$ of sizes equal

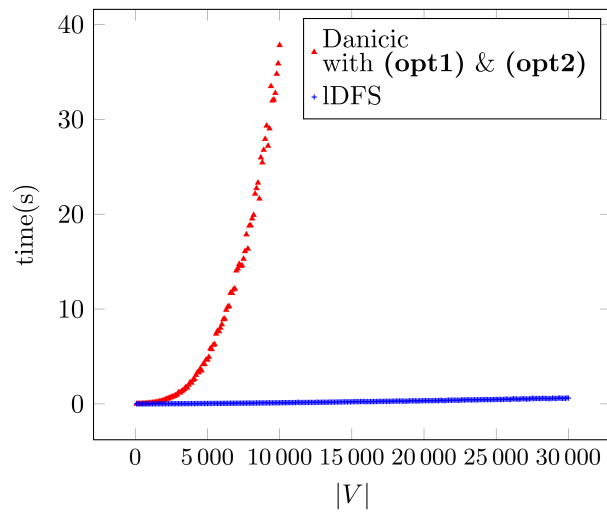


Figure 12: Comparison between the OCaml implementation of Danicic's algorithm with both optimizations and lDFS

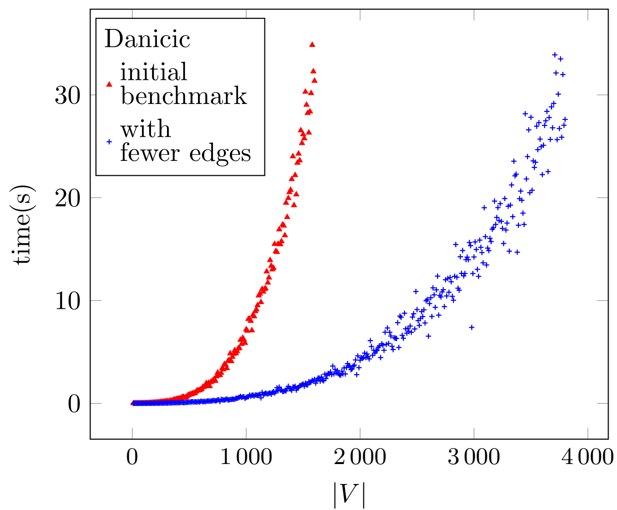


Figure 13: The implementation of Danicic's algorithm on graphs with $e/v = 2$ and $e/v = 1.5$

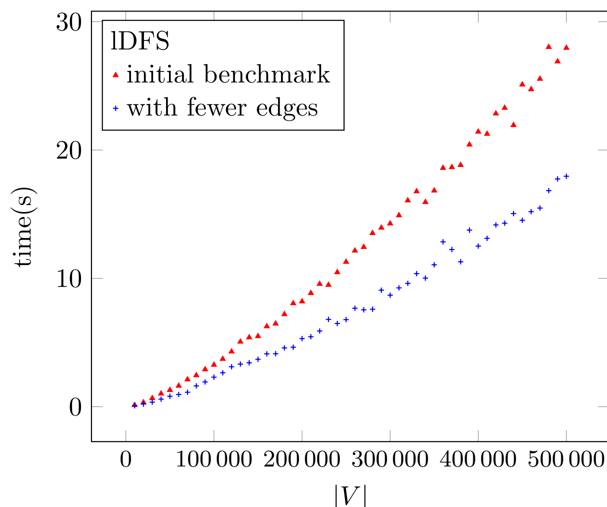


Figure 14: The OCaml implementation of lDFS on graphs with $e/v = 2$ and $e/v = 1.5$

to multiples of 10 between 10 and 1 600, and on graphs with $e/v = 1.5$ of sizes equal to multiples of 10 between 10 and 3 800. Figure 14 shows the results for lDFS. This implementation was run on graphs of sizes equal to multiples of 10 000 between 10 000 and 500 000, first with $e/v = 2$ and then with $e/v = 1.5$.

For both implementations, the running times are noticeably smaller for the graphs with fewer edges. For instance, regarding the implementation of Danicic's algorithm, graphs with around 4 000 nodes and a ratio $e/v = 1.5$ are processed in approximatively the same time as graphs with around 1 500 nodes and a ratio $e/v = 2$.

However, the trends that are revealed are rather similar for both values of e/v. Regarding the implementation of Danicic's algorithm, whether e/v is equal to 2 or 1.5, the implementation reaches its limits for graphs with a few thousand nodes. More importantly, in both cases, the implementation of lDFS is much more efficient than the implementation of Danicic's algorithm.

Summary. The main result of the experiments is the comparison of Danicic's and lDFS that confirms our expectations that lDFS is faster than Danicic's algorithm. Actually, the experimental results show that the difference in terms of running times between both algorithms is substantial.

Another outcome of the experiments is the analysis of the impact of the two optimizations of Danicic's algorithm. While **(opt1)** (detecting at each iteration all the critical edges instead of at most one) is always an interesting optimization, **(opt2)** (relaxing the definition of critical edge) is rarely interesting.

8. Related Work and Conclusion

8.1. Related Work

The last decades have seen various definitions of control dependence given for larger and larger classes of programs [1, 2, 3, 4, 5, 6]. To consider programs with exceptions and potentially infinite loops, Ranganath et al. [18] and then Amtoft [19] introduced non-termination sensitive and non-termination insensitive control dependence on arbitrary program structures. Danicic et al. [8] further generalized control dependence to arbitrary directed graphs, by defining weak and strong control-closure, which subsume the previous non-termination insensitive and sensitive control dependence relations. They also gave a control dependence semantics in terms of projections of paths in the graph, allowing to define new control dependence relations as long as they are compatible with it. This elegant framework was reused for slicing extended finite state machines [20] and probabilistic programs [21]. In both works, an unverified algorithm computing weak control-closure, working differently from ours (more in a breadth-first search way), was designed and integrated in a rather efficient slicing algorithm.

On CFGs with a unique exit point, the standard definition of control dependence in terms of post-dominance can be used. Algorithms computing control dependence can thus take advantage of the efficient algorithms computing the dominance relation [22, 23, 24, 25]. Some of these algorithms are even certified [26], or written with certification in mind [27].

Mechanized verification of control dependence computation was done in formalizations of program slicing. Wasserrab [16] formalized language-independent slicing in Isabelle/HOL, but did not provide an algorithm. Blazy et al. [28] and our previous work [29] formalized control dependence in Coq, respectively, for an intermediate language of the CompCert C compiler [30] and on a WHILE language with possible errors.

8.2. Conclusion and Future Work

Danicic et al. claim that weak control-closure subsumes all other non-termination insensitive variants. It was thus a natural candidate for mechanized formalization. We used the Coq proof assistant to formalize it. A certified implementation of the algorithm can be extracted from the Coq development. During formalization in Coq of the algorithm and its proof, we have detected an inconsistency in a secondary proof, which highlights how useful proof assistants are to detect otherwise overlooked cases. To the best of our knowledge, the present work is the first mechanized formalization of weak control-closure and of an algorithm to compute it. In addition to formalizing Danicic's algorithm in Coq, we have designed, formalized and proved a new one, that we name lDFS. Experiments comparing multiple implementations of Danicic's algorithm and lDFS show that lDFS is faster than Danicic's algorithm, even when the latter benefits of optimizations.

Short-term future work includes considering further optimizations. In particular, we can borrow some ideas from Amtoft et al.'s algorithm [20, 21], since it is also based on Danicic et al.'s work. Long-term future work is to build

a verified generic slicer. Generic control dependence is a first step towards it. Adding data dependence is the next step in this direction.

References

[1] D. E. Denning, P. J. Denning, Certification of programs for secure information flow, Commun. ACM 20 (7) (1977) 504–513.

[2] M. Weiser, Program slicing, in: ICSE 1981, 1981, pp. 439–449.

[3] K. J. Ottenstein, L. M. Ottenstein, The program dependence graph in a software development environment, in: the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1984), ACM Press, 1984, pp. 177–184.

[4] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (3) (1987) 319–349.

[5] A. Podgurski, L. A. Clarke, A formal model of program dependences and its implications for software testing, debugging, and maintenance, IEEE Trans. Software Eng. 16 (9) (1990) 965–979.

[6] G. Bilardi, K. Pingali, Generalized dominance and control dependence, in: PLDI, ACM, 1996, pp. 291–300.

[7] F. Tip, A survey of program slicing techniques, J. Prog. Lang. 3 (3) (1995).

[8] S. Danicic, R. W. Barraclough, M. Harman, J. Howroyd, Á. Kiss, M. R. Laurence, A unifying theory of control dependence and its application to arbitrary program structures, Theor. Comput. Sci. 412 (49) (2011) 6809–6842.

[9] The Coq Development Team, The Coq proof assistant, v8.6, `http://coq.inria.fr/` (2017).

[10] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development, Springer, 2004. doi:10.1007/978-3-662-07964-5.

[11] Why3, a tool for deductive program verification, GNU LGPL 2.1, development version, `http://why3.lri.fr` (January 2018).

[12] J. Filliâtre, A. Paskevich, Why3 - where programs meet provers, in: ESOP, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 125–128.

[13] S. Conchon, J. Filliâtre, J. Signoles, Designing a generic graph library using ML functors, in: Trends in Functional Programming, Vol. 8 of Trends in Functional Programming, Intellect, 2007, pp. 124–140.

[14] J.-C. Léchenet, Formalization of weak control dependence, `https://gitlab.inria.fr/jlechene/ldfs/-/tree/TCS/` (2018).

[15] J.-C. Léchenet, N. Kosmatov, P. Le Gall, Fast Computation of Arbitrary Control Dependencies, in: A. Russo, A. Schürr (Eds.), FASE'18 (Part of ETAPS'18), Springer International Publishing, Cham, 2018, pp. 207–224.

[16] D. Wasserrab, From formal semantics to verified slicing: a modular framework with applications in language based security, Ph.D. thesis, Karlsruhe Inst. of Techn. (2011).

[17] C. Dubois, S. Elloumi, B. Robillard, C. Vincent, Graphes et couplages en Coq, in: Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015), 2015, in French.

[18] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, M. B. Dwyer, A new foundation for control dependence and slicing for modern program structures, ACM Trans. Program. Lang. Syst. 29 (5) (2007).

[19] T. Amtoft, Slicing for modern program structures: a theory for eliminating irrelevant loops, Inf. Process. Lett. 106 (2) (2008) 45–51.

[20] T. Amtoft, K. Androutsopoulos, D. Clark, Correctness of slicing finite state machines, Tech. Rep. RN/13/22, University College London (Dec. 2013).

[21] T. Amtoft, A. Banerjee, A theory of slicing for probabilistic control flow graphs, in: FoSSaCS, Vol. 9634 of Lecture Notes in Computer Science, Springer, 2016, pp. 180–196.

[22] T. Lengauer, R. E. Tarjan, A fast algorithm for finding dominators in a flowgraph, ACM Trans. Program. Lang. Syst. 1 (1) (1979) 121–141.

[23] K. D. Cooper, T. J. Harvey, K. Kennedy, A simple, fast dominance algorithm, Software Practice & Experience 4 (1-10) (2001) 1–8.

[24] L. Georgiadis, R. E. Tarjan, R. F. F. Werneck, Finding dominators in practice, J. Graph Algorithms Appl. 10 (1) (2006) 69–94.

[25] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, J. Westbrook, Linear-time algorithms for dominators and other path-evaluation problems, SIAM J. Comput. 38 (4) (2008) 1533–1573.

[26] S. Blazy, D. Demange, D. Pichardie, Validating dominator trees for a fast, verified dominance test, in: C. Urban, X. Zhang (Eds.), Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings, Vol. 9236 of Lecture Notes in Computer Science, Springer, 2015, pp. 84–99.

[27] L. Georgiadis, R. E. Tarjan, Dominator tree certification and divergent spanning trees, ACM Trans. Algorithms 12 (1) (2016) 11:1–11:42.

[28] S. Blazy, A. Maroneze, D. Pichardie, Verified validation of program slicing, in: CPP 2015, 2015, pp. 109–117.

[29] J.-C. Léchenet, N. Kosmatov, P. Le Gall, Cut branches before looking for bugs: Sound verification on relaxed slices, in: FASE'16 (Part of ETAPS'16), 2016, pp. 179–196.

[30] X. Leroy, Formal verification of a realistic compiler, Commun. ACM 52 (7) (2009) 107–115.