Efficient Algorithms for Computing the Weak and Strong Control Closure

Master Thesis of

Hinkel Manuel

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science
and
Chair of Software Engineering I

Reviewers:	Prof. Dr. Ignaz Rutter
	Prof. Dr. Christian Hammer
Advisors:	Prof. Dr. Ignaz Rutter
	Julian Beckenbauer

Time Period: 29th January 2026 – 29th July 2026

Statement of Authorship

I hereby declare

1. that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text, and has not yet been submitted to any other examination authority.
2. that I observed the statutes of the University of Passau on safeguarding good scientific practice in their currently valid version.
3. that I grant the University of Passau a simple right of use for the purpose of checking, in anonymized form, using plagiarism detection software.

Hinkel Manuel

Passau, July 20, 2026

I hereby declare that I fully documented in the following table how generative AI-tool tools were used to create the thesis.

Hinkel Manuel

Passau, July 20, 2026

Chapter	AI-Tool	Version	Prompt	Description
Chapter 6	ChatGPT	5.5	Given a .csv file that contains the running time of different algorithms on randomly generated graphs with parameters n, p and p', how can I render a matrix of plots that depict the running time on the y-axis and n on the x-axis using the seaborn library for each combination of (p,p') values.	The generated output was used to render the runtime-plots.
Appendix	ChatGPT	5.5	How can I depict C-code nicely in LaTeX?	The package mentioned in the response was used to render Figure A.1.

Table 1: AI-tools used to create the thesis.

Abstract

Control dependence is an important concept in software analysis and forms the basis of many techniques such as program slicing. It captures the influence that the execution of one instruction has on the execution of another. In the literature there have been proposed many different notions of control dependence. As such in 2011 these different characterizations were unified into the non-termination insensitive weak control closure and the non-termination sensitive strong control closure. In this thesis we will present two new algorithms that compute the weak and strong control closure using a new approach based on the utilization of strongly connected components. For a control flow graph $G = (V, E)$ where every vertex has at most two outgoing edges both algorithms achieve a total running time of $\mathcal{O}(|V|^2)$, and if G is acyclic even only $\mathcal{O}(|V|)$. For computing the weak control closure this matches the state-of-the-art algorithm, and for computing the strong control closure this is an improvement over existing algorithms. By using a more involved data structure to maintain the strongly connected components under edge deletions the algorithms achieve an expected total running time of $\mathcal{O}(|V| \log(|V|)^4)$, which is an improvement over the existing algorithms. Both algorithms can also be applied to compute the weak and strong control closure for general graphs, where the number of outgoing edges a vertex is allowed to have is not restricted, with a total expected running time of $\mathcal{O}(|E| \log(|V|)^4)$. We also implemented the algorithms in Java and compared them to implementations of existing algorithms. Additionally, we show that for certain control flow graphs translated from programs adhering to certain programming standards even linear time algorithms exist.

Deutsche Zusammenfassung

Control Dependence ist ein wichtiges Konzept in der Softwareanalyse und bildet die Grundlagen for viele Techniken wie zum Beispiel Program Slicing. Control Dependence beschreibt den Einfluss den die Ausführung einer Anweisung auf die Ausführung einer anderen Ausführung hat. In der Literatur kann man viele verschiedene Definitionen von Control Dependence finden. Daher wurden die unterschiedlichen Charakterisierungen 2011 vereinheitlicht und die nicht-Terminierungs-insensitive Weak Control Closure und die nicht-Terminierungs-sensitive Strong Control Closure definiert. In dieser Arbeit stellen wir zwei neue Algorithmen zur Berechnung des Weak Control Closure und Strong Control Closure vor, die einen neuen Ansatz basierend auf starken Zusammenhangskomponenten verwenden. Für einen Control Flow Graph $G = (V, E)$, bei dem alle Knoten nur Ausgangsgrad zwei haben, erreichen beide Algorithmen eine erwartete Gesamtlaufzeit von $\mathcal{O}(|V|^2)$, und falls G azyklisch ist sogar $\mathcal{O}(|V|)$. Für das Berechnen des Weak Control Closure entspricht das der Laufzeit bestehender Algorithmen und für das Berechnen des Strong Control Closure ist das bereits eine Verbesserung gegenüber existierender Algorithmen. Durch das Einsetzen einer komplexeren Datenstruktur zum Verwalten von starken Zusammenhangskomponenten unter Kantenlöschungen erreichen beide Algorithm eine erwartet Gesamtlaufzeit von $\mathcal{O}(|V| \log(|V|)^4)$, was für beide Algorithmen eine Verbesserung gegenüber bestehender Algorithmen darstellt. Beide Algorithmen können auch angewendet werden um die Weak und Strong Control Closure auf allgemeinen Graphen, bei denen der Ausgangsgrad eines Vertex nicht beschränkt ist, zu berechnen. Die erwartete Gesamtlaufzeit beträgt dann $\mathcal{O}(|E| \log(|V|)^4)$. Außerdem haben wir die Algorithmen in Java implementiert und mit Implementierungen

der existierenden Algorithmen verglichen. Des Weiteren zeigen wir dass für eine bestimmte Klasse an Control Flow Graphen, nämlich die deren zugrundeliegendes Programm gewisse Programmierstandards erfüllt, sogar Algorithmen mit linearer Laufzeitkomplexität existieren.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Weak Control Closure . . . 5
2.2 Computing the Weak Control Closure . . . 9
2.3 Strong Control Closure . . . 10
2.4 Computing the Strong Control Closure . . . 15
2.5 Related Work . . . 17
2.6 Generalized ES-Tree . . . 18
2.6.1 Initializing the GES-tree . . . 19
2.6.2 Updating the GES-tree after Edge Deletion . . . 20
2.6.3 Augmenting the GES-tree . . . 21

3 Linear-Time Algorithms on Acyclic CFGs **23**
3.1 Computing the Weak Control Closure on Acyclic CFGs . . . 24
3.2 Pseudocode for Computing the Weak Control Closure . . . 27
3.3 Computing the Strong Control Closure on Acyclic CFGs . . . 27
3.4 Pseudocode for Computing the Strong Control Closure . . . 30

4 Quadratic-Time Algorithms **33**
4.1 Computing the Weak Control Closure . . . 34
4.1.1 Optimizing the Search for a Sink . . . 39
4.1.2 Optimizing the Reachability-Computation . . . 41
4.2 Pseudocode for Computing the Weak Control Closure . . . 46
4.3 Computing the Strong Control Closure . . . 47
4.4 Pseudocode for Computing the Strong Control Closure . . . 55

5 Decremental Strongly Connected Components **57**
5.1 Overview of the Data Structure . . . 57
5.2 Computing Good Separators . . . 61
5.2.1 Out-Separator . . . 62
5.2.2 In-Separator . . . 63
5.2.3 Satisfying the Size Requirement . . . 65
5.2.4 Simultaneous SCC Computation . . . 67
5.3 Initialization . . . 68
5.4 Edge Deletion . . . 69
5.5 Maintaining a Topologically Ordering of the SCCs . . . 74

6 Evaluation **77**
6.1 Random CFGs . . . 77
6.1.1 Computing the Weak Control Closure . . . 78
6.1.2 Computing the Strong Control Closure . . . 81
6.2 Comparing the DSCC Data Structures . . . 84

6.3 Efficient Implementation 85
6.4 Comparison to Masud's Algorithms 87
6.5 Comparison to the Algorithm of Léchenet, Kosmatov, and Le Gall 90

7 Weak and Strong Control Closure on General Graphs 93

8 Graphs with Bounded Tree-width 95
8.1 MSO_2 Formula for the Weak Control Closure 96
8.2 MSO_2 Formula for the Strong Control Closure 98
8.3 CFGs from Programs 101
8.4 Graphs with Bounded Clique-width 102

9 Conclusion 103

Bibliography 105

Appendix 107
A Evaluation 107

1. Introduction

Software is becoming more complex and the codebases larger, so much so that automated software analysis is becoming an increasingly important tool. A crucial concept in software analysis is *control dependence*, which models the influence one program statement has on the execution of another. Control dependence is for example the foundation of *program slicing* [Wei81], which, given a set of statements called the *slicing criterion*, computes the parts of the program that may affect the behavior at the criterion. In other words, this allows to identify the parts of a program that can be disregarded in an analysis that only affects the slicing criterion. Naturally, by only considering a subset of the instructions of the program the running time of algorithms operating on the program are reduced.

Control dependence is defined on *control flow graphs* [All70], which represent programs as graphs. The exact definitions greatly vary. Often times control flow graphs were assumed to have a unique start and end vertex [FOW87], which does not accurately represent software as there can be multiple different exit points in a program, and exceptions and signals may cause unexpected program termination. As such over the years there have been many different characterizations of control dependence. Danicic et al. [Dan+11] then unified the different notions by describing two different *control closures*. These are the *weak control closure*, which is non-termination insensitive, and the *strong control closure* that is non-termination sensitive. Danicic et al. also gave simple algorithms to compute the weak and strong control closure that run in time $\mathcal{O}(n^3)$ and $\mathcal{O}(n^4)$ respectively with n being the number of vertices in the control flow graph, and every vertex in the control flow graph only has at most two outgoing edges. Noteworthy is that the running time of the algorithm to compute the strong control closure can be improved to $\mathcal{O}(n^3)$ with one slight modification. Masud [Mas22] then described an algorithm that computes the weak control closure in time $\mathcal{O}(n^2)$.

Specifically for acyclic control flow graph we will introduce linear time algorithms, and then, through a new approach using strongly connected components, we will first give deterministic algorithms that both run in time $\mathcal{O}(n^2)$, which, in the case of the strong control closure, is an improvement over the existing algorithms, and then describe algorithms that compute the weak and strong control closure in near linear time in expectation. We will also empirically compare the different algorithms using runtime results of their implementations in the programming language Java. After that we also generalize the results without restricting the number of outgoing edges a vertex is allowed to have.

2. Preliminaries

We only consider finite directed graphs $G = (V, E)$ with vertex set V and edge set E. If we do not explicitly define the vertex set or edge set we write $V(G)$ for the vertex set of G and $E(G)$ for the edge set. Additionally, for a vertex set $V' \subseteq V$ we denote with $E_G^{\text{out}}(V') = \{(u, v) \in E \mid u \in V'\}$ the set of all the outgoing edges of vertices in V', with $E_G^{\text{in}}(V') = \{(u, v) \in E \mid v \in V'\}$ the set of all the incoming edges of vertices in V', and we also introduce the shorthand notation $E_G(V') = E_G^{\text{out}}(V') \cup E_G^{\text{in}}(V')$. The out-degree of a vertex $u \in V$ is defined as $\deg^+(u) = |E_G^{\text{out}}(\{v\})|$, and analogously the in-degree is $\deg^-(u) = |E_G^{\text{in}}(\{v\})|$. We call the graph $G[V'] = (V', \{(u, v) \in E \mid u, v \in V'\})$ the *subgraph* of G induced by V'. As a shorthand for removing a set $E' \subseteq E$ from G we define $G - E' = (V, E \backslash E')$ and for adding edges $E'' \subseteq V^2$ we define $G + E'' = (V, E \cup E'')$. A *path* in G is a sequence of vertices $v_1, ..., v_n$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \{1, ..., n\}$ and $n \in \mathbb{N}_+$. The vertex v_1 is called the *start vertex*, v_n is called the *end vertex*, and the vertices $v_2, ..., v_{n-1}$ are called *intermediate vertices*. Note that for paths containing a single vertex, which we call *trivial paths*, the start vertex is also the end vertex. If there is a path from vertex $u \in V$ to a vertex $v \in V$ we say u can *reach* v. If a path starts and ends at the same vertex, and contains at least two vertices, then it is called a *cycle*. Finally, a path does not necessarily have to be finite, but because G contains only finitely many vertices, an infinite path in G must contain a cycle.

Definition 2.1. *A graph $G = (V, E)$ is called* strongly connected *if and only if for all vertices $u, v \in V$ there exists a path from u to v.*

In other words, a graph is strongly connected if and only if every pair of vertices lies on a cycle. A vertex set is strongly connected if the subgraph induced by it is strongly connected. We say that two vertices $u, v \in V$ are strongly connected if and only if there exists a path from u to v and vice versa. Note that a vertex u is always strongly connected to itself, because there always exists the trivial path u, which only consists of a single vertex.

Definition 2.2. *A* strongly connected component *(SCC) of a graph $G = (V, E)$ is a maximal vertex set that is strongly connected.*

The strongly connected components of a graph partition the vertices uniquely. To show this we define the equivalence relation $\sim$ between two vertices $u, v \in V$ as follows:

$$u \sim v \text{ if and only if } u \text{ and } v \text{ are strongly connected.}$$

This relation is clearly reflexive and symmetric, and because paths are transitive it is also transitive. For a vertex u the equivalence class $[u] = \{v \in V \mid u \sim v\}$ is exactly the SCC that contains u. Because equivalence classes partition the vertex set uniquely the SCCs do as well. To compute the SCCs we can for example use Tarjan's algorithm [Tar72]. Also note that in acyclic graphs the SCCs are single-vertex sets.

The algorithms that we will describe need a topological ordering of vertices, which we define as follows:

Definition 2.3 (Topological Ordering)**.** *For a graph $G = (V, E)$, a* topological ordering *of V is a total order $\prec$ of V such that:*

$$\forall (u, v) \in E : u \prec v$$

A topological ordering can be represented by the bijection $\sigma : \{1, ..., |V|\} \to V$, which maps an index to the vertex at that index in the topological ordering.

Control flow graphs [All70] were originally defined such that every vertex represents a *basic block* of the program. A basic block is a sequence of instructions that are executed one after another, and have one entry and one exit point, which means there is a unique first and last instruction. The edges of the control flow graph then represented the control flow, which means there is an edge (u, v) in the control flow graph if the program can execute the basic block represented by v immediately after it has executed u. A different definition of a control flow graph [KLS20] is more fine-grained and states that every instruction of the program is a vertex in the graph. Danicic et al. [Dan+11, Definition 1] gave a more abstract definition and defined a control flow graph to be a triple $G = (V, E, \beta)$ where (V, E) is a graph, the vertex set V is partitioned into *predicate vertices* $P \subseteq V$ and *non-predicate vertices* $N \subseteq V$, and $\beta : E \to \mathcal{P}(T, F)$ an edge labelling function such that the following hold:

- $N \cup P = V$ and $N \cap P = \emptyset$.
- For all $p \in P$ we have $\deg^+(p) \leq 2$ and for all $v \in N$ it holds that $\deg^+(v) \leq 1$.
- If $x \in P$ and $(x, y) \in E$ then $\beta(x, y) \neq \emptyset$.
- If $x \in N$ and $(x, y) \in E$ then $\beta(x, y) = \emptyset$.
- For $p \in P$ and $(p, y), (p, z) \in E$ with $y \neq z$ it holds that $\beta(p, y) \cap \beta(p, z) = \emptyset$.

To connect this definition to programs, one can think of the predicate vertices as the branching instructions in the program, which is why non-predicate vertices can have at most one outgoing edge. The edge labelling function then describes whether the condition at the instruction that is represented by the predicate vertex is satisfied. Because non-predicate vertices do not represent branching instructions their labels must always be empty. Finally, the last condition just states that the edge labelling must be deterministic, meaning a predicate vertex cannot have two outgoing edges that both represent the true-branch or false-branch and end at different vertices. Note that it is allowed for edges to have the label $\{T, F\}$. Observe that this definition of a control flow graph does also not capture real programs completely because in a program there can be branching instructions with more than two possible branches, such as switch-statements.

2.1 Weak Control Closure

In the following, we will introduce the *weak control closure* as defined by Danicic et al. and illustrate a simple algorithm for computing the weak control closure. For our purposes a control flow graph neither requires the edge labelling function β nor the partition into predicate and non-predicate vertices. As such, we give a much more abstract and for our purposes sufficient definition. For now, we will keep the restriction that every vertex has at most two outgoing edges because this aligns nicely with if-else branches in programs. In Chapter 7 we will generalize our results to general graphs.

Definition 2.4 (CFG)**.** *A graph $G = (V, E)$ is a* control flow graph *if for every vertex $u \in V$ it holds that $\deg^+(u) \leq 2$.*

The weak control closure is always computed for some vertex set $V' \subseteq V$. A vertex being contained in the weak control closure highly depends on which vertices it can reach in V' without encountering another vertex of V' on that path. To simplify this Danicic et al. [Dan+11, Definition 9] defined so called *V'-paths*.

Definition 2.5 (V'-path)**.** *For a graph $G = (V, E)$ and $V' \subseteq V$ a* V'-path *in G is a path $p_1, ..., p_m$ with $m \geq 1$ such that $p_m \in V'$ and for all $i < m$ it holds that $p_i \notin V'$.*

In other words, a V'-path is a path that ends in V' but none of the intermediate vertices, and the start vertex for non-trivial paths, are contained in V'. Figure 2.1 shows a CFG with vertex set $V = \{v_1,\ v_2,\ v_3,\ v_4,\ v_5,\ v_6\}$ and highlighted vertices $V' = \{v_1,\ v_4,\ v_5,\ v_6\}$. Originating at v_2 there are two V'-paths, namely v_2, v_3, v_4 and v_2, v_6. The path v_2, v_3, v_4, v_5 is not a V'-path because $v_4 \in V'$. Neither is v_2, v_3 a V'-path because $v_3 \notin V'$. Vertex v_1 is only contained in the trivial V'-path v_1.

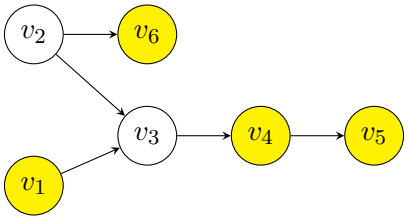

Figure 2.1: A CFG with highlighted vertices $V' = \{v_1,\ v_4,\ v_5,\ v_6\}$.

Note that this definition of a V'-path differs slightly from the one given by Danicic et al. [Dan+11, Definition 9] because they also allow that the start vertex of any V'-path is contained in V', while we only allow the start vertex to be contained in V' if the path is trivial, and as such the start vertex is also the end vertex. This change allows us to skip certain case distinctions regarding the start vertex of a V'-path while still leading to an equivalent notion of a set being weakly control closed, as we will see shortly. First we introduce another classification for vertices, as defined by Danicic et al. [Dan+11, Definition 34].

Definition 2.6 (V'-weakly Committing Vertex)**.** *Let $G = (V, E)$ be a graph and $V' \subseteq V$. A vertex $v \in V$ is* V'-weakly committing *in G if and only if all V'-paths originating at v have the same end vertex.*

Note that with our modified definition of a V'-path every vertex contained in V' must be V'-weakly committing, as the only V'-path originating at a $v \in V'$ is the trivial V'-path

v, while with the definition given by Danicic et al. [Dan+11, Definition 9] vertices from V' can also not be V'-weakly committing. Figure 2.2 depicts a CFG G with highlighted vertex set $V' = \{v_1,\ v_5,\ v_6\}$. In this example the vertices v_2 and v_3 are not V'-weakly committing because both of them can reach vertices v_5 and v_6 via V'-paths. On the other hand, the vertex v_7 is V'-weakly committing because the only V'-path starting at v_7 is the path v_7, v_6. Vertex v_4 is also V'-weakly committing because there are no V'-paths starting at v_4.

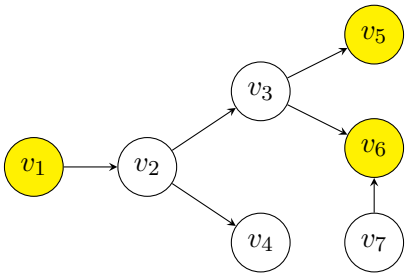

Figure 2.2: A CFG with highlighted vertices $V' = \{v_1,\ v_5,\ v_6\}$.

We will often need vertices of a graph $G = (V, E)$ to be reachable by vertices contained in a set $V' \subseteq V$. We say that a vertex $v \in V$ is *V'-reachable* if there exists a path from some $u \in V'$ to v in G. Finally, we introduce the concept of a *weakly control closed* set, as defined by Danicic et al. [Dan+11, Definition 35]

Definition 2.7 (Weakly Control Closed)**.** *Let $G = (V, E)$ be a graph. A set $V' \subseteq V$ is* weakly control closed *in G if and only if every V'-reachable vertex $v \in V \backslash V'$ is V'-weakly committing in G. A minimal weakly control closed set $X \supseteq V'$ is called a* weak control closure *of V' in G.*

More intuitively, the set V' is weakly control closed if no V'-reachable vertex that is not already contained in V' can reach two different vertices of V' via V'-paths. Note that Definition 2.7 is independent of whether we allow the start vertex of a V'-paths to be included in V' or not because for V' being weakly control closed it does not matter if a vertex that is contained in V' is V'-weakly committing. Also note that vertices that cannot be reached from vertices in V' are allowed to reach multiple vertices of V' via V'-paths, which means not-V'-reachable vertices are never contained in a weak control closure. We also want to briefly mention that if the context is clear we omit the graph in which a vertex is weakly committing or a vertex set is weakly control closed. If we again take the CFG in Figure 2.2 as an example, the set $V' = \{v_1,\ v_5,\ v_6\}$ is not weakly control closed because v_2 and v_3 are V'-reachable but are not V'-weakly committing. The set $X = \{v_1,\ v_3,\ v_5,\ v_6\}$ on the other hand is weakly control closed.

Definition 2.7 indicates that there always exists a weak control closure of V', and it does indeed always exist because V is always weakly control closed in G, as there are no vertices in $V \backslash V = \emptyset$ that may not be V'-weakly committing. Further, the weak control closure is also unique. We will show the uniqueness with Lemma 2.8. For this, we define that two paths are *disjoint* if they do not share any vertices besides the start vertex. Formally, two paths $p_1, ..., p_n$ and $q_1, ..., q_m$ with $n, m \in \mathbb{N}_+$ are disjoint if for all $i \in \{2, ..., n\}$ and $j \in \{2, ..., m\}$ we have $p_i \neq q_j$.

Lemma 2.8. *Let $G = (V, E)$ be a graph, $V' \subseteq V$ a vertex set and $A, B \supseteq V'$ two weakly control closed sets in G. The set $A \cap B$ is again weakly control closed in G.*

Proof. Let $C = A \cap B$ and $v \in V \backslash C$ be a C-reachable vertex. We need to show that v is C-weakly committing in G. First we assume that v is not C-weakly committing. Then there are at least two C-paths originating at v with distinct end points $c_1, c_2 \in C$. Let x be the last vertex that is shared by the two C-paths to c_1 and c_2 originating at v. Now we know that there are two disjoint C-paths p_1 and p_2 originating at x to c_1 and c_2 respectively. Since $x \notin C$, there are the two possible cases $x \notin B$ and $x \notin A$. In the first case, because $x \notin B$ and the C-paths p_1 and p_2 are disjoint, there must also be two disjoint B-paths originating at x because vertices contained in C are also contained in B. Note that those paths end on vertices $b_1, b_2 \in B$, which are the first vertices that are contained in B on the paths p_1 and p_2 respectively. But because p_1 and p_2 are disjoint, we have $b_1 \neq b_2$. This means that B is not weakly control closed, which is a contradiction. The second case where $x \notin A$ is symmetric, which means we get the contradiction that A is not weakly control closed. Because both cases resulted in a contradiction, v must be C-weakly committing, and because our choice of v was arbitrary, C must be weakly control closed. □

Lemma 2.8 also implies that, given the set $\mathcal{X}$ of all weakly control closed supersets of V' in G, the set $\bigcap_{X \in \mathcal{X}} X$ is the weak control closure of V' in G. From this we immediately get the uniqueness of the weak control closure.

Corollary 2.9. *Let $G = (V, E)$ be a graph and $V' \subseteq V$. The weak control closure $X \supseteq V'$ of V' in G does exist and is unique.*

To find the weak control closure, one might think to just compute all the vertices that are not V'-weakly committing and add them to V'. This approach does result in a weakly control closed superset of V', but it is in general not minimal, as we can see in the following example. In the CFG depicted in Figure 2.2, we already determined that the vertices v_2 and v_3 are not V'-weakly committing, so V' is not weakly control closed. The weak control closure of V' does not need to contain v_2 because the set $X = \{v_1,\ v_3,\ v_5,\ v_6\}$ is already weakly control closed. Consequently, we must carefully choose which vertices, besides V', are contained in it and which are not. First we show a sufficient condition for when a vertex is contained in the weak control closure of V'.

Lemma 2.10. *Let $G = (V, E)$ be a graph, $V' \subseteq V$ a vertex set and $p \in V$ a V'-reachable vertex. If there exist two disjoint V'-paths originating at p then p is contained in every weakly control closed set $X \supseteq V'$.*

Proof. If $p \in V'$ then p can only lie on the trivial V'-path p. So assume $p \notin V'$, which means every V'-path that originates at p contains at least two vertices. Assume $p \notin X$. We know that vertex p is X-reachable because it was V'-reachable and X contains V'. Consequently, p must be X-weakly committing, otherwise X would not be weakly control closed. Because there are two disjoint V'-paths p_1 and p_2 originating at p there are also two X-paths originating at p, which are the subpaths of p_1 and p_2 up to the first vertex that is contained in X. Because $p \notin X$, each of these two paths must contain at least two vertices. Let $x_1, x_2 \in X$ be the end vertices of those two X-paths. Because p_1 and p_2 are disjoint, the two paths must also be disjoint, and therefore $x_1 \neq x_2$. But this means that p is not X-weakly committing, which is a contradiction. As such, it must hold that $p \in X$. □

Now we can determine vertices that are certainly not contained in the weak control closure. We mentioned that every vertex that is not V'-reachable is not contained in the weak control closure. To show this we prove that the weak control closure can be computed on only the induced subgraph that contains V'-reachable vertices.

Theorem 2.11. *Let $G = (V, E)$ be a graph, $V' \subseteq V$ a vertex set and $R \subseteq V$ all the V'-reachable vertices in G. Further, let X be the weak control closure of V' in G and X_R be the weak control closure of V' in $G[R]$. It holds that $X = X_R$.*

Proof. Because X is weakly control closed in G, it must also be weakly control closed in $G[R]$, as $G[R]$ cannot have more X-paths than G. Thus, $X_R \subseteq X$ because otherwise, by Lemma 2.8, there would exist a smaller weakly control closed set $X \cap X_R$, and X_R would not be the weak control closure.

Now we show that X_R is weakly control closed in G. Assume that X_R is not weakly control closed in G, then there exists an X_R-reachable vertex $v \in V \backslash X_R$ that is not X_R-weakly committing. Then there originate two X_R-paths p_1 and p_2 at v in G. Because v is X_R-reachable, and every vertex contained in X_R is V'-reachable, we have $v \in R$. Consequently, every vertex on p_1 and p_2 is also contained in R, and thus p_1 and p_2 are present in $G[R]$, which means v is not X_R-weakly committing in $G[R]$, which is a contradiction to X_R being weakly control closed in $G[R]$. By the same argument as before we have $X \subseteq X_R$ and thus $X = X_R$. □

Consider a vertex $v \in V \backslash V'$ that is V'-weakly committing. By Definition 2.7 this vertex does not cause V' to not be weakly control closed in G. Intuitively it does not make sense that v would be contained in X because X should be minimal. This is indeed the case, and a consequence of Lemma 2.12.

Lemma 2.12. *Let $G = (V, E)$ be a graph, $V' \subseteq V$ a vertex set and $X \subseteq V$ the weak control closure of V' in G. For $p \in X$ it holds that $p \in V'$ or that p is a V'-reachable vertex at which at least two disjoint V'-paths originate.*

Proof. Let D be the set of all the vertices at which at least two disjoint V'-paths originate. We first show that $V' \cup D$ is weakly control closed in G. Consider a $(V' \cup D)$-reachable vertex $v \in V \backslash (V' \cup D)$. Clearly $v \notin V'$ and there cannot originate two or more disjoint V'-paths at v. If v can reach none or at most one vertex from V' via a V'-path then it also cannot reach a vertex in D because every vertex in D can reach at least two vertices from V'. Consequently, v is $(V' \cup D)$-weakly committing in those cases. If v can reach two or more vertices from V' via V'-paths then there must exist a vertex z that is the last vertex shared on all those paths. At z there originate now at least two disjoint V'-paths, which means $z \in D$. This again means that v is $(V' \cup D)$-weakly committing. Consequently, $V' \cup D$ is weakly control closed in G, and because X is the minimal weakly control closed superset of V' in G, we have $X \subseteq V' \cup D$.

By Theorem 2.11 p is V'-reachable and if $p \notin V'$ then, because of the previous observation, it holds that $p \in D$. □

Because every V'-weakly committing vertex $v \in V \backslash V'$ can reach at most one vertex in V' there do not exist at least two disjoint V'-paths that originate at v. Consequently, by Lemma 2.12, v is not contained in the weak control closure of V'.

Corollary 2.13. *Let $G = (V, E)$ be a graph, $V' \subseteq V$ a vertex set and $v \in V \backslash V'$ a V'-weakly committing vertex. Let X be the weak control closure of V' in G, then $v \notin X$.*

2.2 Computing the Weak Control Closure

Given a CFG $G = (V, E)$ and $V' \subseteq V$, Danicic et al. [Dan+11, Algorithm 61] described an algorithm to compute the weak control closure $X \supseteq V'$. First observe that it is not necessary to explicitly compute all the V'-paths originating at a vertex $v \in V$ to determine whether it is V'-weakly committing or not. It suffices to compute how many distinct vertices from V' are reachable from v without passing another vertex in V'. Danicic et al. [Dan+11, Definition 56] defined a function to compute those vertices.

Definition 2.14 (Definition 56 [Dan+11])**.** *The set $\Theta(G, V', v)$ for a graph $G = (V, E)$, $V' \subseteq V$ and $v \in V$ is the set of all the vertices from V' reachable from v in the graph G, after removing all the outgoing edges from vertices in V'.*

Using this function, a vertex v is V'-weakly committing if and only if $|\Theta(G, V', v)| \leq 1$. Computing $\Theta(G, V', v)$ is straight forward, we just remove the edges $E_G^{\text{out}}(V')$ from G and run a DFS starting at v to compute all the vertices of V' that are reachable from v. This can be implemented to run in time $\mathcal{O}(|V|)$ if G is a CFG because then $|E| = \mathcal{O}(|V|)$.

Danicic et al. [Dan+11, Lemma 60] have then shown a systematic way of finding vertices that must be contained in the weak control closure, which allows us to construct a simple algorithm to compute the weak control closure iteratively.

Lemma 2.15 (Lemma 60 [Dan+11])**.** *Given a graph $G = (V, E)$ and vertex set $V' \subseteq V$, then V' is not weakly control closed in G if and only if there is an edge $(p, r) \in E$ for which:*

- $|\Theta(G, V', r)| = 1$
- $|\Theta(G, V', p)| \geq 2$
- *p is V'-reachable in G*

Further, vertex p must be contained in every weakly control closed set $X \supseteq V'$.

Proof. The proof is slightly adapted from the proof of Danicic et al. [Dan+11, Lemma 60].

$\Rightarrow$: If V' is not weakly control closed in G then there exists a minimal V'-path $p, p_1, ..., p_m$ with $m \in \mathbb{N}_+$ originating at a V'-reachable vertex $p \in V \backslash V'$ that is not V'-weakly committing. Note that $p \notin V'$ implies that this path contains at least two vertices. Then $r = p_1$ must be V'-weakly committing, otherwise the path would not be minimal and as such it holds that $|\Theta(G, V', r)| = 1$. Further, because p is not V'-weakly committing, we also have $|\Theta(G, V', p)| \geq 2$.

$\Leftarrow$: Now assume that there is an edge $(p, r) \in E$ that satisfies the conditions. Because $|\Theta(G, V', p)| \geq 2$ holds p is not V'-weakly committing and therefore V' is not weakly control closed. Let $X \supseteq V'$ be weakly control closed. Because of $|\Theta(G, V', r)| = 1$ there must be a V'-path $p, r, ..., p_n$ with $n \in \mathbb{N}_+$ and $p_n \in V'$. Further, because $|\Theta(G, V', p)| \geq 2$ there must be a second V'-path $p, ..., q_m \neq p_n$ with $m \in \mathbb{N}_+$, $q_m \in V'$ and $p_n \neq q_m$. The vertex q_m cannot be reached from r via a V'-path because $|\Theta(G, V', r)| = 1$, as such the two paths do not share any vertices besides p. Therefore, the two V'-paths are disjoint and by Lemma 2.10 p is contained in every weakly control closed set $X \supseteq V'$. □

Theorem 2.16. *Given a CFG $G = (V, E)$ and a vertex set $V' \subseteq V$, there exists an algorithm that computes the weak control closure of V' in G in time $\mathcal{O}(|V|^3)$.*

Proof. The algorithm described by Danicic et al. [Dan+11, Algorithm 61] starts by adding all the vertices of V' to X, then it iterates over all the edges $(p, r) \in E$ and checks for each edge if the three conditions of Lemma 2.15 hold. If the edge satisfies the conditions p is added to X, and the algorithm starts another iteration of the edges, otherwise X is weakly control closed by Lemma 2.15 and we stop.

Checking the three conditions for a single edge (p, r) takes time $\mathcal{O}(|V|)$, as computing the Θ-functions takes time $\mathcal{O}(|V|)$ and computing if p is reachable by V' also takes time $\mathcal{O}(|V|)$. Executing the check for every edge in E takes time $\mathcal{O}(|V|^2)$, as there are $\mathcal{O}(|V|)$ edges. In the worst case, we need to iterate $\mathcal{O}(|V|)$ times over all the edges, as it may be necessary to add every vertex to X. This results in a total runtime of $\mathcal{O}(|V|^3)$. □

2.3 Strong Control Closure

Danicic et al. also described the *strong control closure* and gave an algorithm to compute it. Contrary to the weak control closure, the strong control closure depends on which vertices are predicate and non-predicate vertices. Therefore, our simplified definition of a CFG is not sufficient for describing the strong control closure. Instead of extending the definition to include the partition into predicate and non-predicate vertices, our algorithms will additionally get the predicate vertices P as input. We call a predicate vertex $p \in P$ *incomplete* if $\deg^+(p) < 2$, and we call a vertex *final* [Dan+11, Definition 4] if it is either an incomplete predicate vertex or a non-predicate vertex n with $\deg^+(n) = 0$. Finally, a path in a CFG is *complete* [Dan+11, Definition 10] if either its last vertex is final, or the path is infinite. Now we can start to describe what the strong control closure of a vertex set $V' \subseteq V$ is. Similar to V'-weakly committing vertices, Danicic et al. [Dan+11, Definition 36] define V'-strongly committing vertices.

Definition 2.17 (V'-strongly Committing Vertex)**.** *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G and $V' \subseteq V$. A vertex $v \in V$ is* V'-strongly committing *in G for P if and only if it is V'-weakly committing in G and all complete paths starting at v in G contain at least one vertex from V'.*

Definition 2.17 can be interpreted as every complete path originating at a vertex $v \in V$ having to enter V' at the same vertex of V'. Note that this is stronger than V'-weakly committing because all the complete paths must enter V', whereas Definition 2.6 only states that if a complete path enters V' it must be at the same vertex for every path. For a vertex set $X \subseteq V$ we say for simplicity that a path is *X-free* if and only if it does not contain a vertex from X.

Figure 2.3 shows a CFG G with the vertices $V' = \{v_4,\ v_6\}$ highlighted in yellow and the predicate vertices $P = \{v_1,\ v_2,\ v_7\}$ highlighted in green. The vertex v_2 is not V'-weakly committing in G and therefore also not V'-strongly committing in G for P. The vertex v_1 on the other hand is V'-weakly committing, but it is not V'-strongly committing in G for P because it lies on the V'-free complete path v_1, v_5. The reason why this path is complete is that v_5 is a non-predicate vertex with no outgoing edges. The predicate vertex v_7 is also not V'-strongly committing in G for P because it is itself an incomplete predicate vertex and does therefore lie on the trivial V'-free complete path v_7.

Definition 2.18 (V'-avoiding)**.** *Let $G = (V, E)$ be a CFG and $V' \subseteq V$. A vertex $v \in V$ is* V'-avoiding *in G if and only if no vertex in V' can be reached by v.*

The vertex v_5 in the graph shown in Figure 2.3 is for example V'-avoiding for $V' = \{v_4,\ v_6\}$ because it cannot reach v_4 nor v_6.

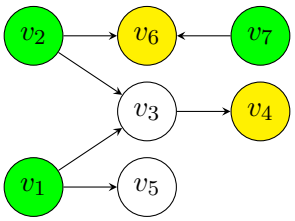

Figure 2.3: A CFG with highlighted vertices $V' = \{v_4,\ v_6\}$ and $P = \{v_1,\ v_2,\ v_7\}$.

Analogously to a set being weakly control closed, we introduce the concept of a set being *strongly control closed* as defined by Danicic et al. [Dan+11, Definition 38]

Definition 2.19 (Strongly Control Closed)**.** *Let $G = (V, E)$ be a CFG and $P \subseteq V$ the predicate vertices of G. A set $V' \subseteq V$ is* strongly control closed *in G for P if and only if every V'-reachable vertex $v \in V \backslash V'$ is V'-strongly committing in G for P or V'-avoiding in G. A minimal strongly control closed set $X \supseteq V'$ is called a* strong control closure *of V' in G for P.*

The strong control closure X is an extension of the weak control closure in the sense that both closures forbid a vertex that is not contained in the closure to reach two different vertices in the closure, but the strong control closure also forbids that a vertex lies both on an X-path and an X-free complete path. Because the strong control closure is stronger than the weak control closure, it holds that if a set V' is not weakly control closed then it also cannot be strongly control closed. Note that when the context is clear we will often omit the graph G and predicate set P for which a set is strongly control closed. Also note that Definition 2.19, just like Definition 2.7, is independent of whether the start vertex of a V'-path is allowed to be contained in V' or not.

Corollary 2.20. *Let $G = (V, E)$ be a CFG and $P \subseteq V$ the predicate vertices of G. Every strongly control closed set $V' \subseteq V$ in G for P is also weakly control closed.*

As an example for strongly control closed sets consider the CFG G in Figure 2.4 with the vertex set $V' = \{v_1,\ v_5\}$ highlighted in yellow and the predicate set $P = \{v_2,\ v_3,\ v_4,\ v_6,\ v_8\}$ highlighted in green. A strongly control closed superset of V' is $X = \{v_1,\ v_2,\ v_3,\ v_4,\ v_5,\ v_6\}$. Vertex v_8 is an incomplete predicate but because it is V'-avoiding it does not violate X being strongly control closed. Vertex v_7 does not need to be contained in X because it can only reach a single vertex from X, which is v_6, so it is X-weakly committing, and it also does not lie on an X-free complete path. Note that X is a minimal strongly control closed superset of V'. Vertices v_3 and v_4 are both incomplete predicate vertices, and therefore lie on a trivial V'-free complete path, and vertex v_6 lies on the infinite V'-free complete path $v_6, v_7, v_6, ...$, and because all three can reach v_5, they are all not V'-strongly committing, nor are they V'-avoiding. If vertex v_2 was not contained in X then it is not X-weakly committing because it can reach $v_5 \in X$ and $v_6 \in X$. This implies that v_2 is also not X-strongly committing. Note that because X is minimal it is a strong control closure of V' for P in G.

Similar to the weak control closure, the strong control closure of V' does always exist and it is also unique. The existence is simply a consequence of V being always strongly control closed, as $V \backslash V = \emptyset$ does not contain any vertex that is not V'-strongly committing. For the uniqueness we will again show that the intersection of two strongly control closed sets is itself strongly control closed.

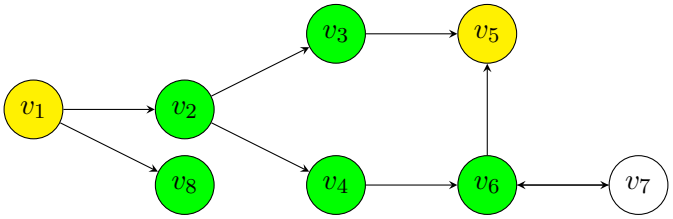

Figure 2.4: A CFG with highlighted vertices $V' = \{v_1,\ v_5\}$ and $P = \{v_2,\ v_3,\ v_4,\ v_6,\ v_8\}$.

Lemma 2.21. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G, $V' \subseteq V$, and $A, B \supseteq V'$ two strongly control closed sets in G for P. The set $A \cap B$ is again strongly control closed in G for P.*

Proof. Let $C = A \cap B$ and $v \in V \backslash C$ be a C-reachable vertex. We need to show that v is C-strongly committing or C-avoiding in G. Assume that v is neither C-strongly committing nor C-avoiding. Then there are two cases. Either v is not C-weakly committing, or there exist both a C-free complete path and a C-path originating at v.

In the first case, using the same reasoning as in the proof of Lemma 2.8, we would get a contradiction that either A or B are not weakly control closed, and as such are also not strongly control closed.

In the second case, we know that there must both be a C-path and a C-free complete path originating at v. Consequently, there must be a vertex x which is the last vertex shared by those paths. We again distinguish between the two cases $x \notin B$ and $x \notin A$ like in the proof of Lemma 2.8. Let $x, ..., c$ be the C-path originating at v and $x, ...$ the C-free complete path originating at v. Note that $x, ...$ may be an infinite path. In the first case, where $x \notin B$, if the complete path $x, ...$ does not contain a vertex contained in B, then x is not B-strongly committing, which is a contradiction because then B is not strongly control closed. Otherwise, meaning there exists a vertex $b \in B$ on the path $x, ...$, then it must hold that $b \neq c$ because the path does not contain a vertex from C. This means that x is not B-weakly committing, and as such not B-strongly committing, which is again a contradiction to B being strongly control closed. Analogously the second case where $x \notin A$ results in the contradiction that A is not strongly control closed.

Because we always get a contradiction that either A or B is not strongly control closed, v must be C-strongly committing or C avoiding, and because our choice of v was arbitrary, C must be strongly control closed. □

Note that in the proof of Lemma 2.21 we did not explicitly use the predicate vertex set P or the fact that G is a CFG, but we need both such that the complete paths are well-defined. Lemma 2.21 implies that, given the set $\mathcal{X}$ of all strongly control closed supersets of V' in G, the set $\bigcap_{X \in \mathcal{X}} X$ is the strong control closure of V' in G for predicate set P. From this we immediately get the uniqueness of the strong control closure.

Corollary 2.22. *Let $G = (V, E)$ be a CFG, P the predicate vertices of G and $V' \subseteq V$. The strong control closure $X \supseteq V'$ of V' in G for P does exist and is unique.*

The reason why the strong control closure is considered non-termination sensitive and the weak control closure is non-termination insensitive is the following. Consider how non-termination of a program can be represented in a CFG. The program entering an infinite loop is equivalent to the CFG containing a cycle. Another possibility is that the program appears to do nothing but also does not terminate, which Danicic et al.

[Dan+11, Section 1.2] call silent non-termination. This can be represented in the CFG as incomplete predicate vertices, or non-predicate vertices with zero outgoing edges. Note that these are exactly the final vertices, which means complete paths are used to represent non-termination. The weak control closure does not depend on complete paths, which is why it is considered non-termination insensitive, but the strong control closure does depend on the complete paths, and is as such considered non-termination sensitive.

We already know from Lemma 2.10 that if there are two disjoint V'-paths that originate at a V'-reachable vertex p then p must be contained in X because X is also weakly control closed. We can also prove a similar argument with a V'-path and a disjoint V'-free complete path.

Lemma 2.23. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G, $V' \subseteq V$ a vertex set and $p \in V$ a V'-reachable vertex. If both a V'-path and a V'-free complete path originate at p, and those paths are disjoint, then p is contained in every strongly control closed set $X \supseteq V'$.*

Proof. If $p \in V'$ then p can only lie on the trivial V'-path p, and every complete path originating at p contains a vertex from V'. Therefore, assume $p \notin V'$, which means every V'-path that originates at p contains at least two vertices. Assume $p \notin X$. We know that vertex p is X-reachable because it was V'-reachable and X contains V'. Consequently, p must be X-strongly committing, otherwise X would not be strongly control closed.

We know that there originate a V'-path p_1 and also a V'-free complete path p_2 at p. If for all $q \in p_2$ it holds that $q \notin X$ then p would not be X-strongly committing, which means X is not strongly control closed. This is a contradiction, so there must be a vertex $q \in p_2$ such that $q \in X$ with minimal distance from p. Because the two paths p_1 and p_2 are disjoint, we also have that p_1 and the path $p, ..., q$ are disjoint. Combined, we have $|\Theta(G, X, p)| \geq 2$, so p is not X-weakly committing, and as such X is not strongly control closed, which is another contradiction. Therefore, it has to hold that $p \in X$. □

Just like for the weak control closure, we can show that certain vertices are never contained in the strong control closure. Naturally, vertices that are not V'-reachable are never contained in the strong control closure. We will use this fact later to design an algorithm that never even processes those vertices.

Theorem 2.24. *Let $G = (V, E)$ be a CFG, P the predicate vertices of G, let $V' \subseteq V$ be a vertex set and $R \subseteq V$ all the V'-reachable vertices in G. Further, let X be the strong control closure of V' in G for P and X_R be the strong control closure of V' in $G[R]$ for P. It holds that $X = X_R$.*

Proof. It is clear that every X-path in $G[R]$ must also be an X-path in G. Consider an X-free complete path in $G[R]$. Then this path must also be X-free in G. More importantly, it must also be complete in G. If it was not then a final vertex f in $G[R]$ is suddenly not a final vertex in G, which means it would have more outgoing edges in G than in $G[R]$. This can only be the case if a neighbor w of f is not contained in $G[R]$, which would mean that w is not V'-reachable, but this is a contradiction because f is V'-reachable. Thus, because X is strongly control closed in G it is also strongly control closed in $G[R]$. Lemma 2.21 then implies that $X_R \subseteq X$.

Now we show that X_R is strongly control closed in G. Assume that X_R is not strongly control closed in G, then there exists an X_R-reachable vertex $v \in V \backslash X_R$ that is neither X_R-strongly committing nor X_R-avoiding. Therefore, there originate either two X_R-paths

p_1 and p_2 or one X_R-path p and an X_R-free complete path c at v in G. In the first case we already saw in the proof of Theorem 2.11 that v is then not X_R-weakly committing in $G[R]$, and consequently not X_R-strongly committing in $G[R]$, which is a contradiction. Thus, we only consider the second case. Because v is X_R-reachable, and every vertex contained in X_R is V'-reachable, we have $v \in R$. Consequently, every vertex on p and c is also contained in R, and thus the two paths p and c are present in $G[R]$, which means v is not X_R-strongly committing nor X_R-avoiding in $G[R]$, which is a contradiction to X_R being strongly control closed in $G[R]$. Using again Lemma 2.21 we get $X \subseteq X_R$ and consequently $X = X_R$. □

Further, just as a V'-weakly committing vertex from $V \backslash V'$ is not contained in the weak control closure, a V'-strongly committing vertex from $V \backslash V'$ is not contained in the strong control closure. Additionally, vertices that are V'-avoiding are also not contained in the strong control closure. Again, we do not show this directly, but derive it from the following lemma.

Lemma 2.25. *Let $G = (V, E)$ be a CFG, P the predicate vertices of G, let $V' \subseteq V$ be a vertex set and $X \subseteq V$ the strong control closure of V' in G for P. For $p \in X$ it holds that $p \in V'$ or that p is a V'-reachable vertex at which at least two disjoint V'-paths or a V'-path and a disjoint V'-free complete path originate.*

Proof. Let D be the set of all the vertices at which at least two disjoint V'-paths or a V'-path and a disjoint V'-free complete path originate. Note that every vertex in D must be able to reach a vertex from V', and if it does not lie on a V'-free complete path, then it must even reach two different vertices from V'. We first show that $V' \cup D$ is strongly control closed in G for P. Consider a $(V' \cup D)$-reachable vertex $v \in V \backslash (V' \cup D)$. Clearly $v \notin V'$ and there cannot originate two or more disjoint V'-paths at v nor can there originate both a V'-path and a disjoint V'-free complete path at v.

If v cannot reach a vertex from V' then it can also not reach a vertex from D because every vertex in D can reach at least one vertex from V'. Thus, v is $(V' \cup D)$-avoiding in this case.

Secondly, if v can reach a single vertex from V' via a V'-path and also lies on a V'-free complete path, then those paths cannot be disjoint, which means there exists a last vertex z that is shared by all those paths. Clearly for this vertex it holds that $z \in D$. But then v is $(V' \cup D)$-strongly committing.

Thirdly, if there are at least two vertices from V' that v can reach via V'-paths, but it does not lie on a V'-free complete path, then those paths cannot be disjoint which means there exists a last vertex z that is shared by all those paths. At z at least two disjoint V'-paths originate, which means $z \in D$. Again, v is $(V' \cup D)$-strongly committing.

In all three cases v is $(V' \cup D)$-strongly committing or $(V' \cup D)$-avoiding, which means $V' \cup D$ is strongly control closed in G for P and as X is the minimal strongly control closed superset of V' in G for P we have $X \subseteq V' \cup D$. By Theorem 2.24 p is V'-reachable and if $p \notin V'$ then we have because of the previous observation that $p \in D$. □

A consequence of Lemma 2.25 is that no V'-strongly committing or V'-avoiding vertex $v \in V \backslash V'$ is contained in the strong control closure of V' because at those vertices there do not originate two or more disjoint V'-paths or a V'-path and a disjoint V'-free complete path.

Corollary 2.26. *Let $G = (V, E)$ be a CFG, P the predicate vertices of G, $V' \subseteq V$ a vertex set and $v \in V \backslash V'$ a V'-strongly committing or a V'-avoiding vertex. Let X be the strong control closure of V' in G for P, then $v \notin X$.*

2.4 Computing the Strong Control Closure

Danicic et al. [Dan+11, Algorithm 65] described an algorithm to compute the strong control closure. The algorithm is similar to the one for computing the weak control closure, with the addition that we also need to compute all the vertices that lie on a complete path that does not pass through a vertex of the strong control closure. For this Danicic et al. [Dan+11, Definition 55] defined another function. Because we use a simplified CFG we need to slightly modify this function by adding the predicate vertex set P as an input.

Definition 2.27. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G and $V' \subseteq V$. The set $\Gamma(G, P, V')$ is the set of all vertices that lie on a V'-free complete path in G.*

Lemma 2.28. *For a given CFG $G = (V, E)$, the predicate vertices $P \subseteq V$ of G and $V' \subseteq V$, it is possible to compute the set $\Gamma(G, P, V')$ in time $\mathcal{O}(|V|^2)$.*

Proof. The algorithm to compute $\Gamma(G, P, V')$ and the correctness, as well as the time complexity were given by Danicic et al. [Dan+11, Algorithm 63, Theorem 64]. The idea is to compute the set X that contains every vertex that only lies on complete paths that contain a vertex from V' and then return $V \backslash X$. Initially, $X = V'$, because clearly the vertices in V' cannot lie on a V'-free complete path. Then we choose an edge $(u, v) \in E$ with $v \in X$ and $u \notin X$ as long as such an edge exists. If there is no such edge we stop and return $V \backslash X$, otherwise the edge (u, v) is removed from E, and we check if there is another edge $(u, w) \in E$ that starts at u and for which $v \neq w$. If this is the case then u must be a complete predicate vertex. We then convert u to a non-predicate vertex. If there is no such edge we check if u is a non-predicate vertex, and if so, we add it to X. Note that u may be an incomplete predicate vertex. In that case it is not added to X.

Clearly, the only vertices added to X are the ones that have at least one edge to a vertex already in X. If the added vertex was originally just a non-predicate vertex then we know that it can exclusively reach a vertex that only lies on complete paths that contain a vertex from V'. If the added vertex u was originally a predicate vertex then we must check both outgoing edges because u only lies on complete paths that contain a vertex from V' if both endpoints of its outgoing edges only lie on complete paths that contain a vertex from V'. This is also the reason why we convert complete predicate vertices to non-predicate vertices after determining for the first outgoing edge (u, v) that $v \in X$, as we cannot be sure that for the second outgoing edge (u, w) we already determined if $w \in X$.

To find an edge $(u, v) \in E$ with $v \in X$ and $u \notin X$ we need time $\mathcal{O}(|E|) = \mathcal{O}(|V|)$, and in the worst-case we need to repeat this $\mathcal{O}(|V|)$ times. □

Using this function, Danicic et al. [Dan+11, Lemma 57] have then shown a systematic way of finding vertices that must be contained in the strong control closure.

Lemma 2.29. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G and $V' \subseteq V$. The vertex set V' is not strongly control closed in G for P if and only if there is an edge $(p, r) \in E$ for which:*

- $|\Theta(G, V', r)| = 1$

- $r \notin \Gamma(G, P, V')$
- $|\Theta(G, V', p)| \geq 2$ *or* $p \in \Gamma(G, P, V')$
- p *is* V'*-reachable in* G

Further, vertex p *must be contained in every strongly control closed set* $X \supseteq V'$*.*

Proof. The proof is slightly adapted from the proof of Danicic et al. [Dan+11, Lemma 57].

$\Rightarrow$: If V' is not strongly control closed in G then there exists a vertex $p \in V \backslash V'$ that is not V'-strongly committing and not V'-avoiding. We choose p such that the V'-path $p, p_1, ..., p_n$ with $n \in \mathbb{N}_+$ is minimal. Note that $p \notin V'$ implies that this path contains at least two vertices. Now there are two cases for why p is not V'-strongly committing. Either it is not V'-weakly committing, or there originate both a V'-path and a V'-free complete path at p.

In the first case it holds that $|\Theta(G, V', p)| \geq 2$. Additionally, $r = p_1$ must be V'-strongly committing, otherwise the path would not be minimal. This means r is both V'-weakly committing and also does not lie on a V'-free complete path. As such $|\Theta(G, V', r)| = 1$ and $r \notin \Gamma(G, P, V')$.

In the second case, we have $p \in \Gamma(G, P, V')$. Again, $r = p_1$ must be V'-strongly committing, otherwise the path would not be minimal. This means we again have $|\Theta(G, V', r)| = 1$ and $r \notin \Gamma(G, P, V')$.

$\Leftarrow$: Now assume that there is an edge $(p, r) \in E$ that satisfies the conditions. Let $|\Theta(G, V', p)| \geq 2$. Then p is not V'-weakly committing, which means V' can neither be weakly control closed nor strongly control closed. Because this case is identical to the one in the proof of Lemma 2.15 vertex p must be contained in every weakly control closed set superset of V'. But because every strongly control closed set $X \supseteq V'$ is also weakly control closed we have $p \in X$.

Now let $p \in \Gamma(G, P, V')$. Because $|\Theta(G, V', r)| = 1$ holds, p is not V'-strongly committing as there originate both a V'-path and a V'-free complete path at p. Therefore, V' is not strongly control closed. Further, because $r \notin \Gamma(G, P, V')$ the V'-path and the V'-free complete path must be disjoint. Then by Lemma 2.23 p must be contained in every strongly control closed set $X \supseteq V'$ □

Note that Lemma 2.29 has one less condition that the corresponding lemma of Danicic et al. [Dan+11, Lemma 57]. We do not need the condition that $p \in V \backslash V'$. If $p \in V'$ then it cannot hold that $|\Theta(G, V', p)| \geq 2$, because the only V'-path originating at p is the trivial path p. Further, $p \in \Gamma(G, P, V')$ can also not hold because p is a vertex in V', so every complete path originating at p contains a vertex from V'.

Theorem 2.30. *Let* $G = (V, E)$ *be a CFG,* $P \subseteq V$ *the predicate vertices of* G *and* $V' \subseteq V$*. There exists an algorithm that computes the strong control closure of* V' *in* G *for* P *in time* $\mathcal{O}(|V|^3)$*.*

Proof. The algorithm described by Danicic et al. [Dan+11, Algorithm 65] starts by adding all the vertices in V' to X, then it iterates over all the edges $(p, r) \in E$ and checks for each edge if the four conditions of Lemma 2.29 hold. If the edge satisfies the conditions p is added to X, and the algorithm starts another iteration of the edges, otherwise X is strongly control closed by Lemma 2.29 and we stop.

Danicic et al. [Dan+11, Theorem 66] have shown that the algorithm has a time complexity of $\mathcal{O}(|V|^4)$. They argued that finding an edge satisfying the four conditions takes time

$\mathcal{O}(|V|^3)$, as we need to iterate over all the edges and for each edge need to check if the four conditions hold. This check takes time $\mathcal{O}(|V|^2)$ because computing $\Gamma(G, P, X)$ takes time $\mathcal{O}(|V|^2)$ by Lemma 2.28. In the worst case we need to add $\mathcal{O}(|V|)$ vertices to X, and because each addition takes time $\mathcal{O}(|V|^3)$ we get the time complexity of $\mathcal{O}(|V|^4)$.

We argue that the Γ-function does not need to be computed for every edge, as it only depends on G and X, and X only changes after adding a vertex to it. If we compute $\Gamma(G, P, X)$ first and only then iterate over all the edges $(p, r) \in E$, the check of the conditions only takes time $\mathcal{O}(|V|)$ for computing the Θ-functions. This reduces the time complexity for adding a single vertex to X down to $\mathcal{O}(|V|^2)$. The total asymptotic runtime is then $\mathcal{O}(|V|^3)$. □

2.5 Related Work

Masud [Mas22] described two algorithms that compute the weak and strong control closure in quadratic and cubic time respectively. While both algorithms are an improvement to the original algorithms given by Danicic et al., we have seen in Theorem 2.30 that the algorithm to compute the strong control closure can already be modified to have a cubic running time. As such we will only briefly illustrate the algorithm to compute the weak control closure. Given a CFG $G = (V, E)$ and a vertex set $V' \subseteq V$ the algorithm described by Masud [Mas22, Algorithm 5] computes the weak control closure X of V' in G by first computing the set of all vertices at which at least two disjoint V'-paths originate. Masud [Mas22, Definition 8] called vertices at which two or more disjoint V'-paths originate *V'-weakly deciding*.

We have shown in Lemma 2.10 that every V'-reachable V'-weakly deciding vertex is contained in every weakly control closed superset of V'. Note that a V'-weakly deciding vertex may also be not-V'-reachable, and thus is not always contained in X. Given the set $W(V')$ of all V'-weakly deciding vertices and the set R of all V'-reachable vertices we have by Lemma 2.10 and Lemma 2.12:

$$X = [W(V') \cap R] \cup V' \tag{2.1}$$

Note that in Equation (2.1) we must include V' separately as a vertex contained in V' is never V'-weakly deciding. Thus, instead of computing X directly it also suffices to compute $W(V')$. Masud's algorithm computes $W(V')$ in two phases. In the first phase, a set W that is a superset of $W(V')$ is computed and then in the second phase the V'-weakly deciding vertices in W are identified. In other words, W is a set of potentially V'-weakly deciding vertices. Finally, the algorithm computes X as in Equation (2.1) by removing every vertex that is not V'-reachable from $W(V')$ and adding the vertices in V'.

Masud [Mas22, Algorithm 2] computed W as follows. Each vertex v stores a vertex $A(v)$, which is a vertex that v can reach via a $(V' \cup W)$-path. We traverse G backwards, meaning along incoming edges, starting at the vertices in V'. For each vertex m we encounter this way we compute the vertices $S_m := \{A(m') \mid (m, m') \in E \wedge A(m') \neq \bot\}$, which are the vertices m can reach via $(V' \cup W)$-paths through a neighboring vertex m'. If there are at least two different vertices in S_m we add m to W. It is easy to see that every vertex added to W must be able to reach two different vertices from V' because initially W is empty, but not necessarily via distinct V'-paths. If we added m to W we set $A(m) \leftarrow m$, but if we do not add it, we know that S_m only contains at most one vertex x, so we set $A(m) \leftarrow x$. Note that because of cycles in G this procedure may not terminate, which means we only continue along the incoming edges of m if $A(m)$ actually changed. Masud [Mas22, Theorem 1] showed that W is indeed a superset of $W(V')$, and that the procedure has a worst case time complexity of $\mathcal{O}(|V|^2)$ [Mas22, Lemma 2].

In the second phase we must identify the vertices in W that are actually V'-weakly deciding. For this, Masud [Mas22, Definition 12] constructed a graph $\mathcal{G}$ with vertex set $\mathcal{V} = V' \cup W$ and edge set $\mathcal{E}$, which contains for every $x \in W$ and every $y \in V$ with $(x, y) \in E$ an edge $(x, A(y))$ if $A(y) \neq \perp$. In other words, if there exists an edge $(u, v) \in \mathcal{E}$ then there exists a $(V' \cup W)$-path from u to v in G. Specifically, if $v \in V'$ then there exists a V'-path from u to v in G.

Using this graph we can then compute the V'-weakly deciding vertices by adding them to a set W_D that is initially empty. The second phase [Mas22, Algorithm 3] is similar to the first phase. For every vertex $v \in \mathcal{V}$ we store in $\bar{A}(v)$ a vertex that v can reach in $\mathcal{G}$ via a $(V' \cup W_D)$-path. Initially, we have $\bar{A}(v) = \perp$ for every vertex $v \in \mathcal{V} \backslash V'$ and $\bar{A}(x) = x$ for every vertex $x \in V'$. Note that if $\bar{A}(v) = w$ holds for $w \in \mathcal{V}$ then there is not only a $(V' \cup W_D)$-path in $\mathcal{G}$, but also in G because of how we constructed $\mathcal{G}$. We again process $\mathcal{G}$ along incoming edges but this time starting at all the vertices in $\mathcal{V}$ that can reach a vertex from V' in $\mathcal{G}$. Note that $V' \subseteq \mathcal{V}$. For every vertex m that we process we compute $\bar{S}_m := \{\bar{A}(m') \mid (m, m') \in \mathcal{E} \wedge \bar{A}(m') \neq \perp\}$, and if $|\bar{S}_m| \geq 2$ then m is V'-weakly deciding, and we add it to W_D. The reason why m is V'-weakly deciding in this case is the following. If there are two different vertices $x_1, x_2 \in V' \cup W_D$ contained in $\bar{S}_m$ then m must be able to reach both of them in G without passing a vertex in W because of the construction of $\mathcal{G}$, which implies that the two paths are disjoint. If the $(V' \cup W_D)$-paths to x_1 and x_2 were not disjoint then we would have added the vertex that lies on both paths to W in the first phase, which is a contradiction. Additionally, if $|\bar{S}_m| = 1$, Masud [Mas22, Algorithm 3, Line 12] explicitly computes whether there is a different vertex $x \in V' \cup W_D$ with $x \notin \bar{S}_m$ that m can reach via a disjoint $(V' \cup W_D)$-path because $\mathcal{G}$ may not capture all disjoint $(V' \cup W_D)$-paths in G. Masud [Mas22, Theorem 3] proved that the computed set W_D is indeed equal to $W(V')$, and that the algorithm has a worst case time complexity of $\mathcal{O}(|V|^2)$.

Thus, both phases can be implemented to run in time $\mathcal{O}(|V|^2)$, and the final computation of Equation (2.1) clearly only needs time $\mathcal{O}(|V|)$.

Theorem 2.31. *Given a graph $G = (V, E)$ and a vertex set $V' \subseteq V$, there exists an algorithm that computes the weak control closure of V' in G in time $\mathcal{O}(|V|^2)$.*

2.6 Generalized ES-Tree

In Chapter 5 we will describe a data structure that efficiently maintains SCCs of a graph under edge deletions. This data structure makes use of *generalized Even-Shiloach-trees* (GES-trees). Before we start describing the GES-tree, we need to introduce the distance measure used by the GES-trees. Let $G = (V, E)$ be a graph. For $u, v \in V$ we denote with $\text{dist}_G(u, v)$ the length of the shortest path from u to v in G. If there is no path between u and v we set $\text{dist}_G(u, v) = \infty$. The *diameter* of a graph is $\max_{u,v \in V} \text{dist}_G(u, v)$. We also define the diameter for a vertex set $U \subseteq V$ as $\max_{u,v \in U} \text{dist}_{G[U]}(u, v)$.

Shiloach and Even [SE81] introduced a BFS-structure that maintains single-source shortest paths in an unweighted graph under edge deletions. This data structure is now known as an *Even-Shiloach-tree* (ES-tree). It allows querying the distance from and to a root vertex $r \in V$ in a directed graph $G = (V, E)$ under edge deletions. This can be used to detect if an SCC in G is no longer strongly connected after an edge deletion because then there must be a vertex that is either no longer reachable from r or can no longer reach r. An ES-tree is usually given a threshold δ that limits the distance a vertex is allowed to have from and to the root vertex. Vertices further away than δ from r are classified as *unreachable* even if they could reach r or could be reached from r via a path longer than δ.

The ES-tree is especially useful for maintaining SCCs with a small diameter because it can detect if an edge deletion increases the diameter of the SCC above the threshold. Let $n = |V|$ and $m = |E|$. The ES-tree can be implemented efficiently with total update time $\mathcal{O}(m \cdot \delta)$, assuming $m = \Omega(n)$ because then iterating over the vertices does not dominate the runtime.

Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 5] then generalized the ES-tree. The current notion of distance is equivalent to each edge having weight one. We can now generalize this such that only outgoing edges of a certain vertex set $S \subseteq V$ have weight one and the remaining edges have weight zero. The GES-tree then uses *S-distances* to measure the distance from and to the root vertex.

Definition 2.32 (S-distance)**.** *For a graph $G = (V, E)$, vertices $u, v \in V$ and a vertex set $S \subseteq V$, we define the S-distance between u and v as the minimum number of vertices from $S \backslash \{v\}$ contained in a path from u to v in G or ∞ if no path between u and v exists and denote it by* $\text{dist}_G(u, v, S)$.

Note that the reason why we exclude v in Definition 2.32 when counting the number of vertices contained in S on the path from u to v is because then the definition of S-distances aligns with measuring distances on a weighted graph where edges in $E^{\text{out}}_G(S)$ have weight one and the other edges weight zero. For every vertex contained in $S \backslash \{v\}$ on the path from u to v we have to cross an outgoing edge from a vertex in S, but for v this is not the case because it is the end vertex of the path, and as such no more edges are traversed after reaching it. Using S-distances we can also define the S-diameter of graph G as $\max_{u,v \in V} \text{dist}_G(u, v, S)$, and additionally for a vertex set $U \subseteq V$ as $\max_{u,v \in U} \text{dist}_{G[U]}(u, v, S)$. Consider the path shown in Figure 2.5. The S-distance from v_1 to v_2, and also from v_2 to v_3 is one because it does not matter whether the target vertex is contained in S. The S-distance can also be zero, which is the case for example from v_3 to v_4.

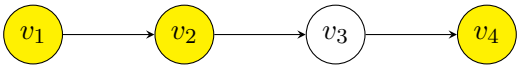

Figure 2.5: A graph with highlighted vertices $S = \{v_1,\ v_2,\ v_4\}$ that are used to measure S-distances.

To measure S-distances a GES-tree, unlike the normal ES-tree, gets an additional set $S \subseteq V$. The underlying logic of the GES-tree is the same as that of an ES-tree, it maintains the shortest path in-tree T_{in} and the shortest path out-tree T_{out} rooted at a vertex r, and if the S-distance of a vertex v to r or from r exceeds the threshold δ it gets labeled as unreachable. Note that Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Definition 5.1] required S to be a feedback vertex set because then the S-distance of two vertices on a cycle is at least one, and the data structure can be implemented efficiently. Why this is important will be explained later. But with this restriction on S the total update time of the GES-tree is the same as that of the ES-tree. The data structure can be implemented to have a total update time of $\mathcal{O}(m \cdot \delta)$. In the following, we will briefly give an overview of how to initialize and update a GES-tree, for a more detailed analysis we refer to Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Appendix B].

2.6.1 Initializing the GES-tree

In the initialization of a GES-tree for a graph $G = (V, E)$, a threshold $\delta \in \mathbb{N}_+$, a root vertex $r \in V$ and vertex set $S \subseteq V$ we need to construct the shortest path in- and out-tree. We denote with $w(u, v)$ the weight of an edge $(u, v) \in E$, which is 1 if $u \in S$ and 0 otherwise. This weighting function is then used to compute S-distances, as S-distances align with

measuring distances in a 0/1-weighted graph where the edges $E_G^{\text{out}}(S)$ have weight 1. The shortest path in-tree T_{in} and shortest path out-tree T^{out} are then simply computed by a weighted 0/1-BFS using w as the weight function. While computing the shortest-path trees the S-distances of $v \in V$ to and from r are stored in distance labels $l_{\text{in}}(v)$ and $l_{\text{out}}(v)$ respectively. Note that for unreachable vertices in T_{in} or T_{out} the respective labels are set to ∞. With this, we can answer distance queries in constant time.

If an SCC Y has S-diameter less than δ then there are no unreachable vertices in $G[Y]$, but if an edge deletion causes the S-diameter to increase above δ there must exist an unreachable vertex v in $G[Y]$. If this happens, we want to be able to query which vertices became unreachable. By maintaining two lists U_{in}, which contains every vertex v that cannot reach r with S-distance at most δ, and U_{out}, which contain every vertex v that is too far away from r, we can answer the query for an unreachable vertex in constant time.

Initializing the distance labels can be done in constant time per vertex. Initializing U_{in} and U_{out} takes time at most $\mathcal{O}(n)$, and the 0/1-BFS to compute T_{in} and T_{out} takes time $\mathcal{O}(n+m)$.

2.6.2 Updating the GES-tree after Edge Deletion

When deleting an edge $(u, v) \in E$, the data structure deletes the edge from G, and then it must update both T_{in} and T_{out}, as well as the distance labels. Here we will look at how to update T_{out}, changes to T_{in} work analogously.

First, the GES-tree needs to determine if the edge deletion even affected T_{out} because if it did not then the distance labels remain unchanged. Clearly, if (u, v) is not an edge in T_{out} then deleting (u, v) does not affect it. If the edge is present in T_{out} we need to delete the edge from the shortest path out-tree. This results in the subtree T_v rooted at v being disconnected. Note that we can test if an edge (u, v) is contained in T_{out} by storing for each vertex which vertex is its parent, and then checking if u is the parent of v.

To reconnect T_v we give a fixing procedure that reconnects a vertex if it is possible to do so. This procedure uses a queue Q that stores the vertex that needs to be reconnected, as well as the distance at which we want to reconnect it. Initially, we insert $(v, l_{\text{out}}(v))$ to Q because we may be able to reconnect T_v at the same S-distance. Then, as long as Q is not empty we execute the following for $(z, d) = Q.\textsc{pop}()$. To check if z can be reconnected at the S-distance d we iterate over its incoming edges, excluding self loops. Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 5] named the set of incoming edges without a self loop *level edges* and let the GES-tree maintain those for every vertex, and analogously the outgoing edges without self loops. To make the implementation simpler we just let G maintain those sets in addition to outgoing and incoming edges. If there is a level edge (y, z) we test if $d = l_{\text{out}}(y) + w(y, z)$, because then z can be reconnected at S-distance d. Observe that if we add v to Q, and it can immediately be reconnected at S-distance $l_{\text{out}}(v)$, then the distance labels in T_v remain unaffected and we can stop. Otherwise, if z can not be reconnected at S-distance d then we insert z with S-distance $d+1$ at the end of Q. Importantly, in this case the distance labels of the children of z may increase, but only if they cannot be reconnected to a different vertex in T_{out} at the same S-distance. Therefore, for every child c of z in T_{out} we insert it at the start of Q with their current distance label $l_{\text{out}}(c)$, because then the S-distance entries in Q are monotonically increasing, and disconnect it from z. Eventually, either every vertex in Q is reconnected or we encounter an entry in Q with distance label $d = \delta + 1$. Then the vertex is unreachable, and because Q is sorted by distance labels, every remaining vertex in Q is also unreachable. Therefore, we empty Q by adding all the vertices still contained in it to U_{out}. Note that when reconnecting a vertex we also update its distance label.

This approach only works because we require S to be a feedback vertex set, and as such every cycle in G increases the S-distance by at least one. If this is not guaranteed, meaning if there are 0-weight cycles, it may happen that we reconnect a vertex z at a descended of itself in T_z, which would break the algorithm.

For each vertex z in Q, we iterate over all the incoming edges of z at most δ times because S-distances never decrease, and if z is unreachable it will never not be unreachable. Therefore, the total number of edges processed by the GES-tree can at most be $m \cdot \delta$.

2.6.3 Augmenting the GES-tree

One further addition that the GES-tree needs, and a normal ES-tree does not have, is that we may want to add vertices $S' \subseteq V$ to S. We call this addition to S *augmenting* the GES-tree. We again only consider how to update T_{out} because updating T_{in} can be done analogously. For every vertex $s \in E_G^{\text{out}}(S' \setminus S)$ the weights of all the outgoing edges $(s, v) \in E$ increase to 1. As such, for every child c of s in T_{out} the S-distances may have increased. Again, if a child can be reconnected to a different vertex in the shortest path out-tree at the same S-distance no distance labels are affected. Therefore, we just insert c into the fixing queue Q, and let the fixing procedure run. Bernstein, Gutenberg, and Wulff-Nilsen argued that the total update time does not increase when adding this augmenting procedure, because the fixing procedure can at most spend time $\mathcal{O}(m \cdot \delta)$ over the lifetime of the GES-tree. Finally, we can summarize this section by defining an interface for the GES-tree data structure in Theorem 2.33

Theorem 2.33 (Lemma 5.2 and 5.3 [BGW23])**.** *Given a graph $G = (V, E)$, a threshold $\delta \in \mathbb{N}_+$, a root vertex r and a feedback vertex set $S \subseteq V$, there exists a data structure that supports the following operations and has total update time $\mathcal{O}(m \cdot \delta)$.*

- *InitGES(r, G, S, δ): Initializes the GES-tree rooted at r up to depth δ measured using S-distances on G.*
- *Distance(r, v): Returns in constant time $\text{dist}_G(r, v, S)$ for $v \in V(G)$ if $\text{dist}_G(r, v, S) \leq \delta$ and otherwise returns ∞.*
- *Distance(v, r): Returns in constant time $\text{dist}_G(v, r, S)$ for $v \in V(G)$ if $\text{dist}_G(v, r, S) \leq \delta$ and otherwise returns ∞.*
- *Delete(u, v): Deletes the edge $(u, v) \in E$ from G and updates the GES-tree.*
- *Delete(V'): Deletes $E_G(V')$ from G and updates the GES-tree.*
- *GetUnreachableVertex(): Returns in constant time an unreachable vertex.*
- *Augment(S'): Adds S' to S and augments the GES-tree.*

3. Linear-Time Algorithms on Acyclic CFGs

For an acyclic CFG $G = (V, E)$, predicate vertices $P \subseteq V$ of G and $V' \subseteq V$ the weak and strong control closure X of V' can be computed in time $\mathcal{O}(|V|)$. The first improvement of the algorithms given by Danicic et al. [Dan+11, Algorithms 61 and 65] is the removal of all the vertices of V that are not V'-reachable, and therefore will never satisfy the third condition of Lemma 2.15 and the fourth condition of Lemma 2.29. This optimization alone does not change the asymptotic running time, but it allows us to skip the test of these conditions for every edge we iterate over. Clearly this removal can be implemented in $\mathcal{O}(|V|)$ time. Because of this, we assume from now on that G only contains V'-reachable vertices. Note that the correctness of this approach is a direct result of Theorem 2.11 and Theorem 2.24.

The second improvement comes from the order in which we iterate over the edge set. Both the algorithm described in Theorem 2.16 and the algorithm described in Theorem 2.30 iterate over all the edges in no specific order and restart the iteration after adding a vertex to X. The reason why both algorithms restart the iteration is that the sets $\Theta(G, X, r)$ and $\Theta(G, X, p)$ for an edge $(p, r) \in E$, and $\Gamma(G, X)$ in the case of the algorithm computing the strong control closure, may change due to the addition of a new vertex to X because it changes the X-paths that are present in G.

We will briefly illustrate why we need to iterate over the same edge multiple times with an example on the CFG G shown in Figure 3.1. To compute the weak control closure of $V' = \{v_1,\ v_5,\ v_6,\ v_7,\ v_8\}$ the algorithm described in Theorem 2.16 initializes $X = \{v_1,\ v_5,\ v_6,\ v_7,\ v_8\}$. In the first iteration of the edges the vertex v_2 will not be added to X, as for both edges (v_2, v_3) and (v_2, v_4) the first condition of Lemma 2.15 is not satisfied because we have $|\Theta(G, X, v_3)| = 2 > 1$ and $|\Theta(G, X, v_4)| = 2 > 1$. But then for edge (v_3, v_4), the three conditions of Lemma 2.15 are satisfied and v_3 is added to X. In the next iteration, edge (v_2, v_3) satisfies all three conditions because now $|\Theta(G, X, v_3)| = 1$ holds. We see that it may be necessary to consider a specific edge multiple times when iterating the edges in arbitrary order.

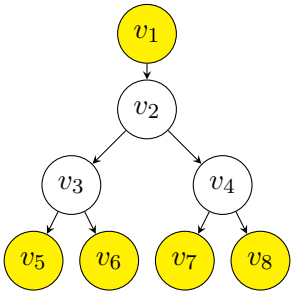

Figure 3.1: A CFG with highlighted vertices $V' = \{v_1, v_5,\ v_6,\ v_7,\ v_8\}$.

3.1 Computing the Weak Control Closure on Acyclic CFGs

Consider the vertices in topological order given by $\sigma : \{1, ..., |V|\} \to V$. For any given $i \in \{1, ..., |V|\}$ and $X \subseteq V$ the X-paths originating at $p = \sigma(i)$ only contain vertices of $V^\sigma_{\geq i} = \{\sigma(j) \mid j \in \{i, ..., |V|\}\}$ because σ represents a topological ordering and edges originating at p can only end in $V^\sigma_{\geq i}$. Note that analogously to $V^\sigma_{\geq i}$ we define $V^\sigma_{>i}$, $V^\sigma_{\leq i}$ and $V^\sigma_{<i}$. To avoid cluttering we also define for a set $X \subseteq V$ the set $X^\sigma_{\geq i} = X \cap V^\sigma_{\geq i}$.

Using this observation, we can determine whether $p \in X$ without processing any vertices in $V^\sigma_{<i}$. This allows us to compute the weak control closure X of V' by iteratively computing $X^\sigma_{\geq i}$ for $i = n, ..., 1$. As an example consider Figure 3.2. It shows the topological ordering of n vertices. Vertex $\sigma(i)$ can only reach vertices in $V^\sigma_{>i}$ that are not itself because neither $\sigma(i)$ nor vertices in $V^\sigma_{>i}$ have edges to any vertex in $V^\sigma_{<i}$. Clearly, the X-paths originating at $\sigma(i)$ are not affected by which vertices from $V^\sigma_{<i}$ are contained in X. In this example $\sigma(i)$ can reach two vertices contained in $X^\sigma_{>i} = \{\sigma(k),\ \sigma(n)\}$. So $\sigma(i)$ is not X-weakly committing and it must be contained in X. Therefore, we can set $X^\sigma_{\geq i} = X^\sigma_{>i} \cup \{\sigma(i)\}$.

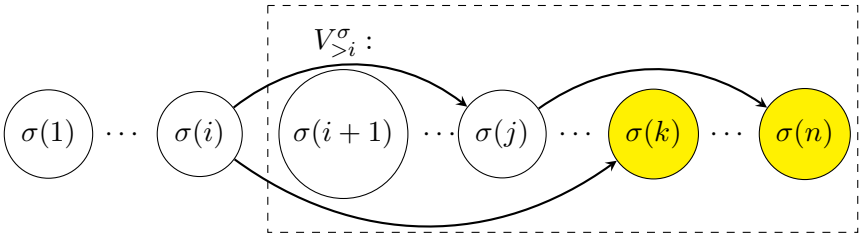

Figure 3.2: A topological ordering of n vertices with highlighted vertices $X^\sigma_{>i} = \{\sigma(k),\ \sigma(n)\}$.

It is easy to see that every vertex $v \in V^\sigma_{\geq i}$ must be $X^\sigma_{\geq i}$-weakly committing, as X is weakly control closed. This also means that there is at most one $X^\sigma_{\geq i}$-path with distinct end vertex originating at v. In Lemma 3.1 we briefly prove this because we will rely on this fact often.

Lemma 3.1. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $n = |V|$, let $X \supseteq V'$ be weakly control closed in G and let σ be a topological ordering of V.*

For any $i \in \{1, ..., n\}$ it holds for every vertex $v \in V^\sigma_{\geq i}$ that $|\Theta(G, X^\sigma_{\geq i}, v)| \leq 1$.

Proof. Assume there exists a vertex $v \in V^\sigma_{\geq i}$ with $|\Theta(G, X^\sigma_{\geq i}, v)| \geq 2$. Then v is not $X^\sigma_{\geq i}$-weakly committing, and because the only vertices from X that v can reach are contained in $X^\sigma_{\geq i}$, vertex v is also not X-weakly committing. But then X is not weakly control closed, which is a contradiction. □

We described that it is always possible to determine whether a vertex $\sigma(i)$ is contained in X if we only know $X^\sigma_{>i}$. Of course, we also need to prove this.

Lemma 3.2. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $n = |V|$, denote with X the weak control closure of V' in G and let σ be a topological ordering of V.*

For every $i \in \{1, ..., n\}$ it holds that $\sigma(i) \in X$ if and only if $\sigma(i) \in V'$ or $|\Theta(G, X^\sigma_{>i}, \sigma(i))| \geq 2$.

Proof. Let $p = \sigma(i)$. Assume that $p \in V'$. Then p must clearly be contained in X because X is a superset of V'.

Now assume that $|\Theta(G, X^\sigma_{>i}, p)| \geq 2$. For every edge $(p, r) \in E$, by Lemma 3.1, it must hold that $|\Theta(G, X^\sigma_{>i}, r)| \leq 1$. Consequently, by Lemma 2.15, p must be contained in every weakly control closed superset of $X^\sigma_{>i}$, and therefore $p \in X$.

Finally, assume that neither $p \in V'$ nor $|\Theta(G, X^\sigma_{>i}, \sigma(i))| \geq 2$ hold. Then p is $X^\sigma_{>i}$-weakly committing. Because every vertex from V' that p can reach is contained in $V_{>i}$, and $V' \subseteq X$, every vertex p can reach from V' is contained in $X^\sigma_{>i}$. Consequently, there do not exist at least two disjoint V' paths that originate at p, and Lemma 2.12 implies that $p \notin X$. □

It remains to show that it is possible to compute X by iteratively computing $X^\sigma_{>i}$ for $i = n, ..., 1$, which we do in the proof of Theorem 3.3.

Theorem 3.3. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices and let $n = |V|$. There exists an algorithm that computes the weak control closure X of V' in G in time $\mathcal{O}(n^2)$.*

Proof. First we compute the topological ordering σ of V. This takes time $\mathcal{O}(n)$ because G is a CFG and therefore $|E| = \mathcal{O}(n)$.

We iteratively compute X by computing $X^\sigma_{\geq i}$ for $i = n, ..., 1$. At iteration i, for $p = \sigma(i)$, we need to determine if $p \in X$. By Lemma 3.2 we have $p \in X$ if and only if $p \in V'$ or $|\Theta(G, X^\sigma_{>i}, \sigma(i))| \geq 2$. Testing if $p \in V'$ takes time $\mathcal{O}(1)$, but for testing the second condition we need to compute $\Theta(G, X^\sigma_{>i}, \sigma(i))$. With a BFS this can be done in time $\mathcal{O}(n)$. If either of the two conditions hold we have $p \in X$ and as such set $X^\sigma_{\geq i} = X^\sigma_{>i} \cup \{p\}$.

At the end, after iteration $i = 1$, we have $X^\sigma_{\geq 1} = X \cap V^\sigma_{\geq 1} = X \cap V = X$. The algorithm executes n iterations, each takes time $\mathcal{O}(n)$, so the total running time is $\mathcal{O}(n^2)$. □

The algorithm described in Theorem 3.3 only executes the check of Lemma 2.15 once per edge and as such reduces the running time from $\mathcal{O}(|V|^3)$ to $\mathcal{O}(|V|^2)$. It still has quadratic running time because we still compute $\Theta(G, X^\sigma_{>i}, \sigma(i))$ for each vertex $i \in \{1, ..., n\}$ from scratch. But because we work with a topological ordering we can also compute these sets more efficiently.

Lemma 3.4. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $n = |V|$, let $X \supseteq V'$ be weakly control closed in G and let σ be a topological ordering of V.*

For every $i \in \{1, ..., n\}$ if $\sigma(i) \in X$ then $\Theta(G, X^\sigma_{\geq i}, \sigma(i)) = \{\sigma(i)\}$, otherwise:

$$\Theta(G, X^\sigma_{\geq i}, \sigma(i)) = \bigcup_{(\sigma(i), r) \in E} \Theta(G, X^\sigma_{>i}, r)$$

Proof. Let $p = \sigma(i)$. If $p \in X$, then $p \in X^\sigma_{\geq i}$. Consequently, there is only one $X^\sigma_{\geq i}$-path that originates at p, which is the trivial path p, and as such $\Theta(G, X^\sigma_{\geq i}, p) = \{p\}$.

If $p \notin X$ then multiple $X^\sigma_{\geq i}$-paths can originate at p. But they all need to cross an edge $(p, r) \in E$. This means the vertices that p can reach through $X^\sigma_{\geq i}$-paths are exactly the vertices that r can reach through $X^\sigma_{\geq i}$-paths. And because σ is a topological ordering, and the index of r is greater than i, these are exactly the vertices reachable by r through $X^\sigma_{>i}$-paths. Therefore, we have:

$$\Theta(G, X^\sigma_{\geq i}, p) = \bigcup_{(p,r)\in E} \Theta(G, X^\sigma_{>i}, r)$$

□

Using Lemma 3.4 we can now design an algorithm that computes the weak control closure in linear time on acyclic CFGs.

Theorem 3.5. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices and let $n = |V|$. There exists an algorithm that computes the weak control closure X of V' in G in time $\mathcal{O}(n)$.*

Proof. Again, like in Theorem 3.3, we initially compute the topological ordering σ of V. This takes time $\mathcal{O}(n)$ because G is a CFG and therefore $|E| = \mathcal{O}(n)$.

We iteratively compute X by computing $X^\sigma_{\geq i}$ for $i = n, ..., 1$. Before each iteration i we ensure that we know the sets $X^\sigma_{>i}$ and $\Theta(G, X^\sigma_{>i}, \sigma(j))$ for every $i < j \leq n$. Note that initially, before iteration $i = n$, we have $X^\sigma_{>n} = \emptyset$.

Then, at iteration i, for $p = \sigma(i)$, we need to determine if $p \in X$. For this, we compute $\Theta(G, X^\sigma_{>i}, p)$, which is simply done by computing the following:

$$\Theta(G, X^\sigma_{>i}, p) = \bigcup_{(p,r)\in E} \Theta(G, X^\sigma_{>i}, r) \tag{3.1}$$

As p has at most two outgoing edges, and for every outgoing edge (p, r) we already know the contents of the set $\Theta(G, X^\sigma_{>i}, r)$, which by Lemma 3.1 contains at most one element, the computation only takes constant time.

By Lemma 3.2, we have $p \in X$ if $|\Theta(G, X^\sigma_{>i}, p)| \geq 2$ or $p \in V'$. Testing these two conditions clearly only takes constant time, as we already computed $\Theta(G, X^\sigma_{>i}, p)$. If $p \in X$ we have:

- $X^\sigma_{\geq i} = X^\sigma_{>i} \cup \{p\}$
- $\Theta(G, X^\sigma_{\geq i}, p) = \{p\}$, by Lemma 3.4.

Otherwise, meaning if $p \notin X$, then we have the following instead:

- $X^\sigma_{\geq i} = X^\sigma_{>i}$
- $\Theta(G, X^\sigma_{\geq i}, p) = \Theta(G, X^\sigma_{>i}, p)$, by Lemma 3.4 and Equation (3.1).

Note that in both cases computing $X^\sigma_{\geq i}$ and $\Theta(G, X^\sigma_{\geq i}, p)$ only takes constant time.

Before continuing with iteration $i - 1$, we still need to compute the sets $\Theta(G, X^\sigma_{>i-1}, \sigma(j))$ for all $i - 1 < j \leq n$. Note that it holds that $\Theta(G, X^\sigma_{>i-1}, \sigma(j)) = \Theta(G, X^\sigma_{\geq i}, \sigma(j))$, so for $j = i$ we already know this set because we already computed $\Theta(G, X^\sigma_{\geq i}, p)$. For every other $i < j \leq n$ observe that all the X-paths originating at the vertex $\sigma(j)$ can only end in vertices contained in $X^\sigma_{\geq j}$. Therefore, it holds for all $1 < k < j$ that

$\Theta(G, X^\sigma_{\geq k}, \sigma(j)) = \Theta(G, X^\sigma_{\geq j}, \sigma(j))$. And because we already know $\Theta(G, X^\sigma_{>i}, \sigma(j))$ we also know $\Theta(G, X^\sigma_{>i-1}, \sigma(j))$ without further computation, and we can continue with the next iteration.

After iteration $i = 1$, we again have $X^\sigma_{\geq 1} = X$. The algorithm executes n iterations, and because each iteration can be implemented to run in constant time, the total running time is $\mathcal{O}(n)$. □

3.2 Pseudocode for Computing the Weak Control Closure

To be able to easily implement the described algorithm, we also give the pseudocode for it in Algorithm 3.1. This algorithm also works for graphs that contain vertices that are not V'-reachable because it initially removes them with the method RemoveUnreachable. We compute the topological ordering of the vertices of a graph with the TopoOrdering method. Note that after each iteration i the set X is equivalent to $X^\sigma_{\geq i}$, and for every $v \in \{\sigma(i), ...\sigma(n)\}$ the set $\Theta(v)$ stores $\Theta(G, X^\sigma_{\geq i}, v)$.

Algorithm 3.1: WeakControlClosure(G, V')

Input : An acyclic CFG $G = (V, E)$ and a vertex set $V' \subseteq V$.
Output : The weak control closure $X \subseteq V$ of V' in G.

```
1  H ← RemoveUnreachable(G, V');
2  σ ← TopoOrdering(H);
3  n ← |V(H)|;
4  X ← ∅;
5  for i = n, ..., 1 do
6      p ← σ(i);
7      Θ(p) ← ∅;
8      for (p, r) ∈ E(H) do
9          Θ(p) = Θ(p) ∪ Θ(r);
10     if p ∈ V' or |Θ(p)| ≥ 2 then
11         X ← X ∪ {p};
12         Θ(p) = {p};
13 return X;
```

3.3 Computing the Strong Control Closure on Acyclic CFGs

Computing the strong control closure X on an acyclic CFG $G = (V, E)$ for predicate vertices $P \subseteq V$ is similar to computing the weak control closure, only that now we also need to keep track of complete paths. The idea is that, using the topological ordering σ of V, we determine whether $\sigma(i) \in X$ using partial knowledge of the set $\Gamma(G, P, X)$. So before processing index $i = n, ..., 1$ we ensure that we know $\Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$ which we then use to compute $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i})$. Lemma 3.6 is the base for our iterative computation of $\Gamma(G, P, X)$.

Lemma 3.6. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $P \subseteq V$ be the predicate vertices of G, $n = |V|$, let $X \supseteq V'$ be strongly control closed for P in G and let σ be a topological ordering of V.*

For $i \in \{1, ..., n\}$ it holds that $\sigma(i) \in \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$ if and only if $\sigma(i)$ is a final vertex or there exists an edge $(\sigma(i), v) \in E$ such that $v \in \Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$.

Proof. Let $p = \sigma(i)$. Note that because we are only interested in $X^\sigma_{>i}$-free complete paths it does not matter whether $p \in V'$. Clearly, if p is a final vertex then it lies on an $X^\sigma_{>i}$-free complete path in $G[V^\sigma_{\geq i}]$. Now assume that p is not a final vertex and that there exists an edge $(p, v) \in E$ such that $v \in \Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$. This means that there exists an $X^\sigma_{>i}$-free complete path in $G[V^\sigma_{>i}]$ that originates at v. Because $p \in V^\sigma_{\geq i}$, and there exists an edge from p to v there is also an $X^\sigma_{>i}$-free complete path originating at p. Therefore, $p \in \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$. On the other hand, if there is no such edge then every complete path originating at p contains at least one vertex from $X^\sigma_{>i}$. □

Using Lemma 3.6 our algorithm computes $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$, which has two important use cases. Firstly, we need $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$ to determine if $\sigma(i) \in X$, as described in the proof of Lemma 3.8. Secondly, if $\sigma(i) \in X$, we can compute the set $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i})$ because then every complete path that contains $\sigma(i)$ contains a vertex from X, which means that the following equation holds:

$$\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i}) = \Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i}) \tag{3.2}$$

Before we continue, we first prove a lemma that is analog to Lemma 3.1, which states that every vertex $v \in V^\sigma_{\geq i}$ is $X^\sigma_{\geq i}$-strongly committing.

Lemma 3.7. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $P \subseteq V$ be the predicate vertices of G, $n = |V|$, let $X \supseteq V'$ be strongly control closed for P in G and let σ be a topological ordering of V.*

For any $i \in \{1, ..., n\}$ it holds for every vertex $v \in V^\sigma_{\geq i}$ that $|\Theta(G, X^\sigma_{\geq i}, v)| \leq 1$, and if $|\Theta(G, X^\sigma_{\geq i}, v)| = 1$ it also holds that $v \notin \Gamma(G, P, X^\sigma_{\geq i})$.

Proof. Let $v \in V^\sigma_{\geq i}$. Because X is strongly control closed and therefore also weakly control closed, by Lemma 3.1, we have that $|\Theta(G, X^\sigma_{\geq i}, v)| \leq 1$.

Now assume that $|\Theta(G, X^\sigma_{\geq i}, v)| = 1$, which also means that $|\Theta(G, X, v)| = 1$ because v can only reach vertices in $V^\sigma_{\geq i}$. If we also have that $v \in \Gamma(G, P, X^\sigma_{\geq i})$ then $v \notin X^\sigma_{\geq i} \subseteq X$ must hold because every complete path that originates at a vertex contained in $X^\sigma_{\geq i}$ trivially contains a vertex from $X^\sigma_{\geq i}$ and would therefore not be contained in $\Gamma(G, P, X^\sigma_{\geq i})$. But then v is neither X-avoiding nor X-strongly committing, which means that X is not strongly control closed, which is a contradiction. Therefore, it must hold that $v \notin \Gamma(G, P, X^\sigma_{\geq i})$. □

Analogously to Lemma 3.2, we can also find some conditions that allow us to efficiently determine whether $\sigma(i) \in X$ only using the partial knowledge of X and $\Gamma(G, P, X)$. These conditions are stated in Lemma 3.8.

Lemma 3.8. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $P \subseteq V$ be the predicate vertices of G, $n = |V|$, denote with X the strong control closure of V' for P in G and let σ be a topological ordering of V.*

For $i \in \{1, ..., n\}$, we have that $\sigma(i) \in X$ if and only if one of the following hold:

- $\sigma(i) \in V'$
- $|\Theta(G, X^\sigma_{>i}, \sigma(i))| \geq 2$
- $|\Theta(G, X^\sigma_{>i}, \sigma(i))| = 1$ *and* $\sigma(i) \in \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$

Proof. Let $p = \sigma(i)$. Clearly, if $p \in V'$ then $p \in X$. Otherwise, we need to test the conditions of Lemma 2.29.

First, if $|\Theta(G, X^\sigma_{>i}, p)| \geq 2$ holds then by applying Lemma 3.7 for $i+1$, we get that p must have two outgoing edges (p, r) and (p, r') such that $r, r' \in V_{>i}$ and the following hold:

- $|\Theta(G, X^\sigma_{>i}, r)| = 1$
- $r \notin \Gamma(G, P, X^\sigma_{>i})$
- $|\Theta(G, X^\sigma_{>i}, r')| = 1$
- $r' \notin \Gamma(G, P, X^\sigma_{>i})$

Because p is a V'-reachable vertex, by Lemma 2.29, we know that p must be contained in every strongly control closed superset of $X^\sigma_{>i}$. Therefore, $p \in X$.

Now let $|\Theta(G, X^\sigma_{>i}, p)| = 1$ and $p \in \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$. Then it also holds that $p \in \Gamma(G, P, X^\sigma_{>i})$. Further, there must be an edge $(p, r) \in E$ such that $|\Theta(G, X^\sigma_{>i}, r)| = 1$. Then, by Lemma 3.7, we also have that $r \notin \Gamma(G, P, X^\sigma_{>i})$, which allows us to apply Lemma 2.29 again, and we get that $p \in X$.

On the other hand, if p does not satisfy any of the three conditions then it is not contained in V' and clearly either $X^\sigma_{>i}$-strongly committing or $X^\sigma_{>i}$-avoiding. Lemma 2.25 then implies $p \notin X$. □

Now we can formally describe our algorithm, which iteratively applies Lemma 3.8.

Theorem 3.9. *Let $G = (V, E)$ be an acyclic CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $P \subseteq V$ be the predicate vertices of G and $n = |V|$. There exists an algorithm that computes the strong control closure X of V' in G for P in time $\mathcal{O}(n)$.*

Proof. We initially compute the topological ordering σ of V. This takes time $\mathcal{O}(n)$, because G is a CFG and therefore $|E| = \mathcal{O}(n)$.

We iteratively compute X by computing $X^\sigma_{\geq i}$ for $i = n, ..., 1$. Before each iteration i we ensure that we know the sets $X^\sigma_{>i}$, $\Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$ and for every $i < j \leq n$ the set $\Theta(G, X^\sigma_{>i}, \sigma(j))$. Note that initially, before iteration $i = n$, these sets are all empty sets.

Then, at iteration i, for $p = \sigma(i)$, we need to determine if $p \in X$. First we compute $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$. We already know $\Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$, so we only need to determine whether $p \in \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$. By Lemma 3.6, if p is a final vertex, or there exists an edge $(p, r) \in E$ such that $r \in \Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$, then:

$$\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i}) = \Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i}) \cup \{p\} \tag{3.3}$$

Testing the conditions of Lemma 3.6 can be done in constant time. To check if p is a final vertex we only need to consider $\deg^+(p)$, and because p only has at most two outgoing edges for which we need to determine if their endpoints are contained in the given set $\Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$ we can also test the second condition in constant time.

Secondly, we need to compute $\Theta(G, X^\sigma_{>i}, p)$, which is again done by computing the following:

$$\Theta(G, X^\sigma_{>i}, p) = \bigcup_{(p,r)\in E} \Theta(G, X^\sigma_{>i}, r) \tag{3.4}$$

This set can be computed in constant time as p has at most two outgoing edges, and for every outgoing edge (p, r) we already know the contents of the set $\Theta(G, X^\sigma_{>i}, r)$, which, by

Lemma 3.7, contains at most one element. Now we determine if $p \in X$ using Lemma 3.8. Testing the three conditions clearly only takes constant time, as we already computed $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$ and $\Theta(G, X^\sigma_{>i}, p)$.

If $p \in X$, we know that the following hold:

- $X^\sigma_{\geq i} = X^\sigma_{>i} \cup \{p\}$
- $\Theta(G, X^\sigma_{\geq i}, p) = \{p\}$, by Lemma 3.4.
- $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i}) = \Gamma(G[V^\sigma_{>i}], P, X^\sigma_{>i})$, by Equation (3.2).

Note that $p \notin \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i})$ because every path that starts at p trivially contains a vertex from $X^\sigma_{\geq i}$.

Otherwise, if $p \notin X$, then we have the following instead:

- $X^\sigma_{\geq i} = X^\sigma_{>i}$
- $\Theta(G, X^\sigma_{\geq i}, p) = \Theta(G, X^\sigma_{>i}, p)$, by Lemma 3.4 and Equation (3.4).
- $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i}) = \Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{>i})$, by Equation (3.3).

Note that for every $i < j \leq n$, we have that $\Theta(G, X^\sigma_{>i}, \sigma(j)) = \Theta(G, X^\sigma_{\geq i}, \sigma(j))$, because vertices at those indices can only reach vertices from $X^\sigma_{\geq j} \subseteq X^\sigma_{\geq i}$. Therefore, we can now proceed with the next iteration without further computation.

After iteration $i = 1$ we know $X^\sigma_{\geq 1} = X$. The algorithm executes n iterations, and each iteration can be implemented to run in time $\mathcal{O}(1)$, which means the total running time is $\mathcal{O}(n)$. □

3.4 Pseudocode for Computing the Strong Control Closure

To be able to also easily implement the algorithm that computes the strong control closure on acyclic CFGs we give the pseudocode for it in Algorithm 3.2. Like Algorithm 3.1, this algorithm also works for graphs that contain vertices that are not V'-reachable. It again holds that after each iteration i the set X is equivalent to $X^\sigma_{\geq i}$, and for every $v \in \{\sigma(i), ...\sigma(n)\}$ the set $\Theta(v)$ is equivalent to $\Theta(G, X^\sigma_{\geq i}, v)$. Additionally, the set Γ is equivalent to $\Gamma(G[V^\sigma_{\geq i}], P, X^\sigma_{\geq i})$.

Algorithm 3.2: StrongControlClosure(G, P, V')

Input : An acyclic CFG $G = (V, E)$, the predicate vertices $P \subseteq V$ of G and a vertex set $V' \subseteq V$.

Output : The strong control closure $X \subseteq V$ of V' in G for P.

```
1  H ← RemoveUnreachable(G, V');
2  σ ← TopoOrdering(H);
3  n ← |V(H)|;
4  X ← ∅;
5  Γ ← ∅;
6  for i = n, ..., 1 do
7      p ← σ(i);
8      γ ← IsFinalVertex(p, P, G);
9      for (p, r) ∈ E(H) do
10         if r ∈ Γ then
11             γ ← ⊤;
12     Θ(p) ← ∅;
13     for (p, r) ∈ E(H) do
14         Θ(p) = Θ(p) ∪ Θ(r);
15     if p ∈ V' or |Θ(p)| ≥ 2 or (|Θ(p)| = 1 and γ = ⊤) then
16         X ← X ∪ {p};
17         Θ(p) = {p};
18     else
19         if γ = ⊤ then
20             Γ = Γ ∪ {p};
21 return X;
```

4. Quadratic-Time Algorithms

By lifting the restriction that the given CFG must be acyclic we loose the crucial ability to order the vertices of the graph topologically. Fortunately, the topological ordering of the vertices was just a tool to solve a different problem, as we will see now. Consider the algorithm to compute the weak control closure X of $V' \subseteq V$ for an acyclic graph $G = (V, E)$, as described in Theorem 3.5. Let σ be the topological ordering of V and $i \in \{1, ..., |V|\}$. The algorithm needs to decide whether $\sigma(i) \in X$, and does this by computing $\Theta(G, X^\sigma_{>i}, \sigma(i))$ and testing if $|\Theta(G, X^\sigma_{>i}, \sigma(i))| \geq 2$ because then, by Lemma 3.2, it holds that $\sigma(i) \in X$. In the proof of why this is the case we used Lemma 3.1, where we argued that because a vertex $v \in V^\sigma_{>i}$ can only reach vertices contained in $V^\sigma_{>i}$ it must be $X^\sigma_{>i}$-weakly committing. This reasoning does neither require that $V^\sigma_{<i}$ nor $V^\sigma_{>i}$ are topologically ordered. We only need that no vertex in $V^\sigma_{>i}$ has an outgoing edge to a vertex contained in $V^\sigma_{\leq i}$. Further, to efficiently compute $\Theta(G, X^\sigma_{>i}, \sigma(i))$ the algorithm used the previously computed sets $\Theta(G, X^\sigma_{>i}, v)$ for $v \in V^\sigma_{>i}$. For this to work we also do not need that $V^\sigma_{<i}$ and $V^\sigma_{>i}$ are topologically ordered, the only requirement is that $\sigma(i)$ does not have any outgoing edges to vertices contained in $V^\sigma_{<i}$. We used the topological ordering because in a topological ordering these requirements always hold, but we could also change the algorithm to work on a vertex partition $\{L, R\}$ with the constraints $L \cup R = V$, $L \cap R = \emptyset$ and for every vertex $u \in R$ and for all $(u, v) \in E$ it holds that $v \in R$. The algorithms on acyclic CFGs use the partition $L = V^\sigma_{\leq i}$ and $R = V^\sigma_{>i}$. Then we pick a vertex $p \in L$ such that every outgoing edge from p has its endpoint in R. As long as the graph is acyclic such a vertex must always exist in L. Using the topological ordering, this choice of p is trivial because we know that $\sigma(i)$ is such a vertex that only has outgoing edges into $R = V^\sigma_{>i}$. Before the next iteration, instead of decreasing the index i by one, we would just modify the partition by removing p from L and adding it to R.

By approaching the algorithms this way we do not need to sort the vertices topologically, but there is still an issue. While in acyclic graphs it is always possible to find at least one vertex in L where the endpoint of every outgoing edge is contained in R, in a CFG that contains a cycle this may not be possible. Consider the CFG G depicted in Figure 4.1 and the partition $\{L = \{v_1, v_2\}, R = \{v_3\}\}$. Then both $v_1, v_2 \in L$ have an outgoing edge that again ends in L.

Instead of working with vertices we can also consider the SCCs of G. We construct a graph $\hat{G}$ as follows. For every SCC the constructed graph $\hat{G}$ contains a vertex, and for every edge $(u, v) \in E$ the graph $\hat{G}$ contains an edge between the two vertices that represent the SCCs containing u and v. This graph may still contain self loops if both u and v are in the same

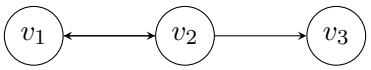

Figure 4.1: A CFG where the partition $\{L = \{v_1, v_2\}, R = \{v_3\}\}$ does not contain a vertex where the endpoint of every outgoing edge ends in R.

SCC, but after removing the self loops $\hat{G}$ is acyclic. This is because if it still contained a cycle the vertices on that cycle would not represent maximal strongly connected vertex sets of G, which is a contradiction because in Definition 2.2 we stated that SCCs must be maximal. Because $\hat{G}$ is acyclic we can find in every partition $\{L, R\}$ of the vertices $V(\hat{G})$ a vertex that only has outgoing edges that end in R. Instead of constructing such a graph $\hat{G}$ and then using its vertices as placeholders for SCCs we will just use a partition $\{\mathcal{L}, \mathcal{R}\}$ of SCCs of G. We call a partition $\{\mathcal{L}, \mathcal{R}\}$ of the SCCs of a graph G *restricted* if for all $R \in \mathcal{R}$ and every edge $(u, v) \in E_G^{\text{out}}(R)$ it holds that $v \in R'$ for an SCC $R' \in \mathcal{R}$. An example for a restricted partition is given in Figure 4.2. Note that neither $\mathcal{L}$ nor $\mathcal{R}$ are topologically ordered if we consider the ordering that results from numbering the SCCs from left to right. It only matters that there is no edge from a vertex in $\mathcal{R}$ to $\mathcal{L}$. The SCC L that will be processed by our algorithms will be chosen from $\mathcal{L}$ such that no vertex in L has an outgoing edge to a vertex contained in a different SCC $L' \in \mathcal{L}$. In other words, L only has edges to itself and SCCs in $\mathcal{R}$. We call such an SCC a *sink*. Figure 4.2 also shows an example for a sink $L = \{v_6,\ v_7,\ v_8\} \in \mathcal{L}$ that has no edges to a different SCC in $\mathcal{L}$. Note that the SCC $\{v_1,\ v_2\}$ would also be a sink.

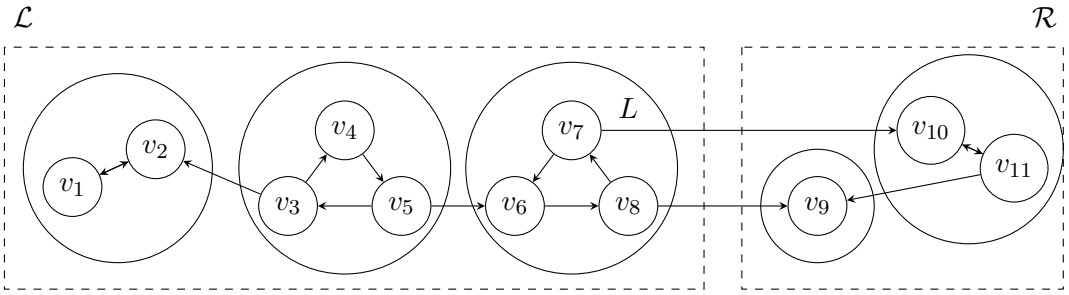

Figure 4.2: An example for a restricted partition $\{\mathcal{L}, \mathcal{R}\}$ of SCCs and a sink L that has no outgoing edges to a different SCC also contained in $\mathcal{L}$.

4.1 Computing the Weak Control Closure

Let $V' \subseteq V$ be the vertex set for which we want to compute the weak control closure X in G. We assume that G only contains V'-reachable vertices because by Theorem 2.11 it suffices to compute the weak control closure on the graph that only contains V'-reachable vertices. Our goal is to design an algorithm that works with a restricted partition $\{\mathcal{L}, \mathcal{R}\}$ of SCCs, and to get an efficient algorithm we want to avoid processing the vertices and edges that are contained in an SCC as much as possible. This means we need to decide if the vertices in an SCC are V'-weakly committing by only considering the reachability between the SCCs. For the SCCs of G this may cause some problems as the next example illustrates. Figure 4.3 shows graphs G_1 and G_2 with vertex set $V' = \{v_2,\ v_4\}$ highlighted and the graphs $\hat{G}_1$ and $\hat{G}_2$ where SCCs were merged and self loops removed. In both $\hat{G}_1$ and $\hat{G}_2$ there exists a path from SCC $\{v_1\}$ to SCC $\{v_2,\ v_4\}$, which implies that vertex v_1 can reach both vertices $v_2 \in V'$ and $v_4 \in V'$. But in G_1 vertex v_1 can only reach v_2 via V'-paths, while in G_2 it can reach both v_2 and v_4 via V'-paths. In G_1 vertex v_1 is V'-weakly committing, but in G_2 it is not. We cannot distinguish this by only considering reachability of the SCCs of G.

Instead, the algorithm will maintain a set X' that will be equal to the weak control closure X when the algorithm finishes, and work on a restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$ because then every vertex that is eventually added to X', and as such contained in X, forms its own single-vertex SCC in $\mathcal{L}$ and $\mathcal{R}$. This solves the above described problem because if there is a path from Y to another SCC that contains a vertex $x \in X'$ then every vertex $y \in Y$ can reach x via an X'-path because no intermediary vertex of that path can be contained in X' as those vertices do not have outgoing edges. Initially, we can set $X' = V'$ because it holds that $X \supseteq V'$. Also note that by removing the outgoing edges of vertices in X' we do not lose any important reachability information because only X'-paths decide whether a vertex is contained in X, and X'-paths already disregard any outgoing edges from vertices in X'.

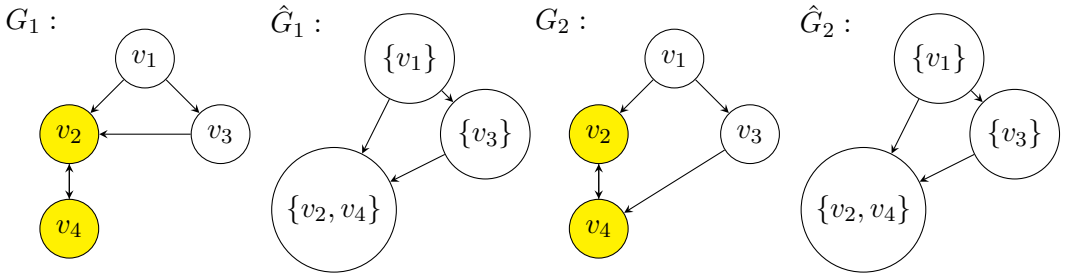

Figure 4.3: Two CFGs G_1 and G_2 and their respective graphs where SCCs are merged. The vertices v_2 and v_4 are highlighted.

Using a restricted partition of SCCs of $G - E_G^{\text{out}}(X')$ has another advantage. When processing a sink $L \in \mathcal{L}$ we do not have to determine all the vertices from L that are contained in X, it suffices to only determine a subset of the vertices from L that are contained in X. If we have such vertices we must remove their outgoing edges and update $\mathcal{L}$ accordingly because $\mathcal{L}$ only contains SCCs of $G - E_G^{\text{out}}(X')$. These edge deletions cause L to split into multiple new SCCs of $G - E_G^{\text{out}}(X')$, which are processed by the algorithm at a later step. Therefore, we can determine the rest of the vertices contained in $X \cap L$ at a later step. Because of this we can rather efficiently process an SCC L as follows.

We define $\hat{\Theta}(G, X', L) = \bigcup_{v \in L} \Theta(G, X', v)$, which is the set of all the vertices contained in X' that are reachable via X'-paths originating at a vertex $v \in L$ in G. If $|\hat{\Theta}(G, X', L)| \leq 1$, we know that every vertex $v \in L$ is X'-weakly committing, and none of them are contained in the weak control closure of V' by Corollary 2.13. On the other hand, if $|\hat{\Theta}(G, X', L)| \geq 2$ then every vertex $v \in L$ is not X'-weakly committing because they all lie on X'-paths to the vertices in $\hat{\Theta}(G, X', L)$. In this case, some vertices from L must be contained in X. Because of the previous observation we do not have to determine all the vertices from L that are contained in L, but we do determine those that lie on the boundary of L, which are the vertices that have an outgoing edge with endpoint is not contained in L. Formally, we call a path $p_1, p_2, ..., p_m$ in G with $m \geq 2$ a *direct path* if none of the vertices $p_2, ..., p_m$ are contained in the SCC that contains p_1. In other words, a direct path immediately leaves the SCC in which it started and never returns to it. Further, let $B(G, X', L)$ be all the vertices of the SCC L where a direct X'-path originates. We will show later that these vertices are certainly contained in X if $|\hat{\Theta}(G, X', L)| \geq 2$. Here we describe the idea for why this is the case. A vertex $v \in B(G, X', L)$ can reach a vertex $x \in X'$ via the direct X'-path originating at v. The second vertex on that X'-path is a vertex $r \in R$ with $R \in \mathcal{R}$. Because $\mathcal{R}$ are the already processed SCCs it must hold for R that $|\hat{\Theta}(G, X', R)| \leq 1$. Therefore, we have $|\Theta(G, X', r)| \leq 1$ and $|\Theta(G, X', v)| \geq 2$, which means by Lemma 2.15 vertex v is contained in X.

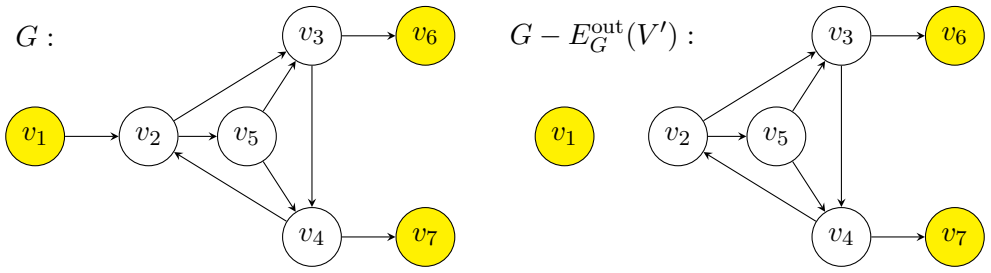

Figure 4.4: A CFG G with SCC $\{v_2,\ v_3,\ v_4,\ v_5\}$ and highlighted vertices $V' = \{v_1,\ v_6,\ v_7\}$ and the graph $G - E_G^{\text{out}}(V')$.

To better illustrate this reasoning consider Figure 4.4 that shows a CFG G with highlighted vertices $V' = \{v_1,\ v_6,\ v_7\}$. Initially we have $X' = V'$. The vertices v_2, v_3, v_4 and v_5 make up an SCC of the CFG. The SCCs of the graph $G - E_G^{\text{out}}(X')$, which is also shown in Figure 4.4, are $\{v_1\}$, $\{v_2,\ v_3,\ v_4,\ v_5\}$, $\{v_6\}$, $\{v_7\}$. When processing SCC $L = \{v_2,\ v_3,\ v_4,\ v_5\}$ we have $\hat{\Theta}(G, X', L) = \{v_6,\ v_7\}$ and $B(G, X', Y) = \{v_3,\ v_4\}$. Both v_3 and v_4 are not X'-weakly committing because both of them can reach the vertices v_6 and v_7. Assume we first add v_3 to X', then v_4 no longer lies on an X'-path to v_6 because to reach v_6 from v_4 the path must traverse v_3. But this does not mean that v_4 is X'-weakly committing. Because $v_3 \in X'$ there is now an X'-path from v_4 to v_3, meaning we still need to add v_4 to X'. On the other hand, if we first add v_4 to X' then v_3 is no longer on the X'-path to v_7, but it lies on an X'-path to v_4, which means it is still not X'-weakly committing, and we must add it to X'.

After adding $B(G, X', L)$ to X', the remaining vertices in L may not all be X'-weakly committing, and they need to be processed again. If we delete $E_G^{\text{out}}(B(G, X', L))$ then L is no longer strongly connected, and it splits into new SCCs. These SCCs then replace L in $\mathcal{L}$. Eventually, vertices that are not X'-weakly committing must have direct X'-paths originating at them, and the algorithm will add them to X'.

We continue with the previous example on the graph shown in Figure 4.4. After adding v_3 and v_4 to X' we remove their outgoing edges from G. The resulting graph is shown in Figure 4.5. The SCC $\{v_2,\ v_3,\ v_4,\ v_5\}$ was split into SCCs $\{v_2\}$, $\{v_5\}$, $\{v_3\}$ and $\{v_4\}$. When processing SCC $\{v_5\}$, the algorithm will add v_5 to X' because it can reach both v_3 and v_4.

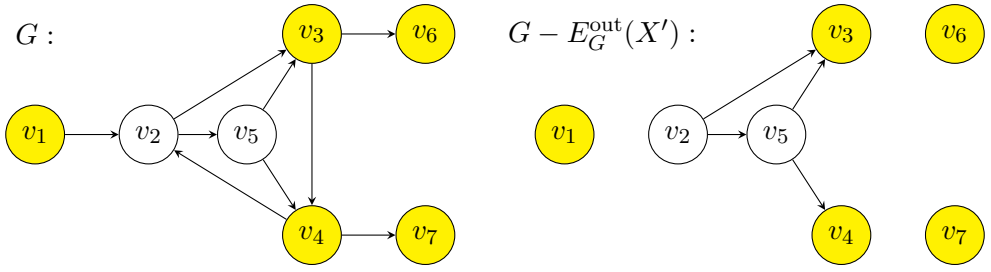

Figure 4.5: A CFG G with SCC $\{v_2,\ v_3,\ v_4,\ v_5\}$ and highlighted vertices $X' = \{v_1,\ v_3,\ v_4,\ v_6,\ v_7\}$ and the graph $G - E_G^{\text{out}}(X')$.

From now on, we drop the argument G in the $\hat{\Theta}$- and B-functions to avoid cluttering. Further, for two SCCs Y and Z we say that Y can *reach* Z if and only if there exists a vertex $y \in Y$ that can reach a vertex $z \in Z$. Additionally, we extend our notion of restricted SCC partitions.

Definition 4.1. *Let $G = (V, E)$ be a CFG, $V' \subseteq V$ a vertex set and let $X' \supseteq V'$. A partition $\{\mathcal{L}, \mathcal{R}\}$ of the SCCs of $G - E_G^{\text{out}}(X')$ is called* X'-weakly restricted *if the following hold.*

- *For all $R \in \mathcal{R}$ and every edge $(u, v) \in E_G^{\text{out}}(R)$ it holds that $v \in R'$ for an SCC $R' \in \mathcal{R}$.*
- *For all $R \in \mathcal{R}$ and every vertex $v \in R$ it holds that v is X'-weakly committing in G.*

A simple consequence of Definition 4.1 is that each SCC $R \in \mathcal{R}$ can only reach at most one distinct SCC that contains a vertex from X' in $G - E_G^{\text{out}}(X')$ because otherwise the vertices in R can reach two distinct vertices contained in X' through X'-paths, which would mean that they are not X'-weakly committing in G.

Corollary 4.2. *Let $G = (V, E)$ be a CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$ and let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.*

For an SCC $R \in \mathcal{R}$ it holds that $|\hat{\Theta}(X', R)| \leq 1$.

For an X'-weakly restricted partition $\{\mathcal{L}, \mathcal{R}\}$ of the SCCs of $G - E_G^{\text{out}}(X')$ and a sink $L \in \mathcal{L}$ we previously argued that we can add $B(X', L)$ to X' if $|\hat{\Theta}(X', L)| \geq 2$ because then $B(X', L) \subseteq X$. In Lemma 4.3 we formally prove this.

Lemma 4.3. *Let $G = (V, E)$ be a CFG, $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices and let X be the weak control closure of V' in G. Further, let $V' \subseteq X' \subseteq X$, let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$ and let $L \in \mathcal{L}$ be a sink.*

If $|\hat{\Theta}(X', L)| \geq 2$ then for every vertex $v \in B(X', L)$ it holds that $v \in X$.

Proof. Let $|\hat{\Theta}(X', L)| \geq 2$. Then for all $u \in L$ we have $|\Theta(G, X', u)| \geq 2$. Further, because $v \in B(X', L)$ there exists a direct X'-path originating at v. Let r be the second vertex on this path, which exists because direct paths always have at least two vertices. Vertex r is contained in an SCC $R \in \mathcal{R}$ because the only SCCs that L can reach and that are different from itself must be contained in $\mathcal{R}$ as L is a sink. By Corollary 4.2 it holds that $|\hat{\Theta}(X', R)| \leq 1$. Consequently, it must also hold that $|\Theta(G, X', r)| \leq 1$. Finally, by Lemma 2.15, vertex v must be contained in every weakly control closed superset of X'. The weak control closure X is such a superset, and therefore $v \in X$. □

After adding $B(X', L)$ to X', we need to update the SCCs in $\mathcal{L}$ because $G - E_G^{\text{out}}(X')$ now has fewer edges than before and L is no longer strongly connected, as we removed $E_G^{\text{out}}(B(X', L))$. Afterward, the partition $\{\mathcal{L}, \mathcal{R}\}$ must still be X'-weakly restricted.

Lemma 4.4. *Let $G = (V, E)$ be a CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$, let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$ and let $L \in \mathcal{L}$ be a sink.*

Let $X'' = X' \cup B(X', L)$ and $\mathcal{L}'$ be the SCCs of $G[\bigcup \mathcal{L}] - E_G^{\text{out}}(X'')$. Then $\{\mathcal{L}', \mathcal{R}\}$ is an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X'')$.

Proof. Clearly is every SCC in $\mathcal{L}'$ an SCC of $G - E_G^{\text{out}}(X'')$, and because $B(X', L) \not\subseteq \bigcup \mathcal{R}$, the SCCs in $\mathcal{R}$ are unaffected by the removal of the edges $E_G^{\text{out}}(B(X', L))$, which means $\mathcal{R}$ also only consists of SCCs of $G - E_G^{\text{out}}(X'')$.

The first condition of Definition 4.1 clearly still holds because $G - E_G^{\text{out}}(X'')$ contains fewer edges than $G - E_G^{\text{out}}(X')$.

Additionally, also because $B(X', L) \not\subseteq \bigcup \mathcal{R}$, the X''-paths in $G[\bigcup \mathcal{R}]$ are exactly the X'-paths in $G[\bigcup \mathcal{R}]$, which means that every vertex in $\bigcup \mathcal{R}$ is also X''-weakly committing, and the second condition of Definition 4.1 is also satisfied. □

If $|\hat{\Theta}(X', L)| \leq 1$ then the algorithm does not add any vertices to X', but it should still make some progress in this case. We know that every vertex contained in L is X'-weakly committing, which means we can remove it from $\mathcal{L}$ and add it to $\mathcal{R}$ to again get an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.

Lemma 4.5. *Let $G = (V, E)$ be a CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$, let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$ and let $L \in \mathcal{L}$ be a sink.*

If $|\hat{\Theta}(X', L)| \leq 1$ then $\{\mathcal{L}\backslash\{L\}, \mathcal{R} \cup \{L\}\}$ is an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.

Proof. Let $\mathcal{L}' = \mathcal{L}\backslash\{L\}$ and $\mathcal{R}' = \mathcal{R} \cup \{L\}$. Clearly, $\mathcal{L}'$ and $\mathcal{R}'$ only contain SCCs of $G - E_G^{\text{out}}(X')$. Then, because L can only reach SCCs in $\mathcal{R}$ and itself, every SCC in $\mathcal{R}'$ can only reach vertices in SCCs in $\mathcal{R}'$, which means the first condition of Definition 4.1 is satisfied. Further, because every vertex in L is X'-weakly committing, and every vertex contained in an SCC in $\mathcal{R}$ is X'-weakly committing, every vertex contained in an SCC in $\mathcal{R}'$ must be X'-weakly committing, and the second condition of Definition 4.1 is also satisfied. □

Finally, we can describe an algorithm that computes the weak control closure by maintaining an X'-weakly restricted partition of SCCs.

Theorem 4.6. *Let $G = (V, E)$ be a CFG, $n = |V|$ and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. There exists an algorithm that computes the weak control closure X of V' in G in time $\mathcal{O}(n^2)$.*

Proof. **Initialization:** The algorithm initializes $X' = V'$ because V' is always contained in X. Further, it initializes the partition $\{\mathcal{L}, \mathcal{R}\}$ with $\mathcal{L}$ being the SCCs of $G - E_G^{\text{out}}(X')$ and $\mathcal{R} = \emptyset$. Note that computing $\mathcal{L}$ can be done in $\mathcal{O}(n)$ time using Tarjan's algorithm [Tar72]. Over the course of the algorithm we maintain that $X' \subseteq X$, and that $\{\mathcal{L}, \mathcal{R}\}$ is an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$. Clearly, this holds initially because $\mathcal{R}$ is empty.

Iteration: The algorithm executes as long as $\mathcal{L} \neq \emptyset$ the following. It picks a sink $L \in \mathcal{L}$, which can be implemented to run in time $\mathcal{O}(n)$ by maintaining a mapping of vertices to the SCCs they are contained in as follows. For every SCC $L \in \mathcal{L}$ we can iterate over the outgoing edges of vertices contained in it in time $\mathcal{O}(|E_G^{\text{out}}(L)|)$, and for every edge we can test if the endpoint is contained in a different SCC $L' \in \mathcal{L}$ using the mapping from vertex to SCC in constant time. In the worst case we do this for every SCC in $\mathcal{L}$, which sums up to a total running time of $\mathcal{O}(n)$. Note that we also could just sort the SCCs $\mathcal{L}$ topologically and take the last SCC in the ordering because we know that this SCC cannot have any edges to a different SCC in $\mathcal{L}$.

Then the algorithm computes $\hat{\Theta}(X', L)$. If $|\hat{\Theta}(X', L)| \leq 1$ then by Lemma 4.5 the partition $\{\mathcal{L}\backslash\{L\}, \mathcal{R} \cup \{L\}\}$ is also a valid partition, so we remove L from $\mathcal{L}$ and add it to $\mathcal{R}$. Note that

X' stays unchanged, and it still holds that $X' \subseteq X$. Otherwise, meaning $|\hat{\Theta}(X', L)| \geq 2$, we compute $B(X', L)$ and add those vertices to X'. By Lemma 4.3 it still holds that $X' \subseteq X$. Subsequently, we delete the edges $E_G^{\text{out}}(B(X', L))$, which causes L to no longer be strongly connected, and therefore we also compute the new SCCs $\mathcal{L}'$ of $G[\bigcup \mathcal{L}] - E_G^{\text{out}}(X')$. This can again be done in linear time using for example Tarjan's algorithm. By Lemma 4.4 setting $\mathcal{L} \leftarrow \mathcal{L}'$ results in the partition $\{\mathcal{L}, \mathcal{R}\}$ being an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.

Running Time: Every vertex x added to X' is only added once because as soon as $x \in X'$ it is X'-weakly committing, which means when processing the SCC L that contains x we have $|\hat{\Theta}(X', L)| \leq 1$. The algorithm does terminate after at most $2n$ iterations. The weak control closure X of V' in G can contain at most n elements, which means we can have at most n iterations that add vertices to X' because we always have that $X' \subseteq X$ and never add a vertex twice to X'. Additionally, because $\mathcal{R}$ only contains SCCs of $G - E_G^{\text{out}}(X')$, and there can be at most n SCCs in $G - E_G^{\text{out}}(X')$, the algorithm can move at most n SCCs from $\mathcal{L}$ to $\mathcal{R}$. We already argued that finding a sufficient $L \in \mathcal{L}$ and recomputing the SCCs takes only linear time. Additionally, computing $\hat{\Theta}(X', L)$ and $B(X', L)$ can also be implemented to run in time $\mathcal{O}(n)$ with a BFS. As such, we have $\mathcal{O}(n)$ iterations that all take time $\mathcal{O}(n)$, which means the total running time of the algorithm is $\mathcal{O}(n^2)$.

Correctness: At the end of the algorithm it holds that $X' \subseteq X$. It remains to show that $X \subseteq X'$. Assume that $X \not\subseteq X'$, which then implies that $X' \subset X$. Then, because X is the minimal weakly control closed superset of V', the set X' must be not weakly control closed in G. Consequently, there must exist a vertex $v \in V \backslash V'$ that is not X'-weakly committing. Because we have $\mathcal{L} = \emptyset$, there must be an SCC $R \in \mathcal{R}$ that contains v. But this is a contradiction because the second condition of Definition 4.1 states that every vertex that is contained in an SCC contained in $\mathcal{R}$ must be X'-weakly committing. As such, $X \subseteq X'$ and $X = X'$. □

Note that the algorithm described in Theorem 4.6 can be easily extended to work on CFGs that also contain vertices that are not V'-reachable by just removing those vertices in the initialization. While we now have an algorithm that is faster than the one described in Theorem 2.16, we can still further optimize it. There are two things in the algorithm that are currently wasting time. The first is the search for a sink $L \in \mathcal{L}$, and the second is that we compute the sets $\hat{\Theta}(X', L)$ and $B(X', L)$ at each step from scratch. First, we will illustrate how to pick a sink more efficiently.

4.1.1 Optimizing the Search for a Sink

Currently, our algorithm potentially iterates over all the outgoing edges of vertices contained in an SCC in $\mathcal{L}$. This results in every iteration the algorithm executes taking linear time. In the algorithm for acyclic CFGs, we use the topological ordering to determine the next vertex the algorithm should process. Here we can do something similar. Because the SCCs can be sorted topologically we can maintain the topological ordering σ of $\mathcal{L}$, and then, instead of searching for a sink L, we just pick the last SCC in σ because we know that this SCC cannot have any edges to an SCC $L' \in \mathcal{L}$ with an index in σ that is lower than the index of L.

When the algorithm adds vertices to X' we compute new SCCs that replace L in $\mathcal{L}$ because L is no longer strongly connected in $G - E_G^{\text{out}}(X')$. We also need to update our topological ordering σ to represent this change. For this, we use a *decremental strongly connected components* (DSCC) data structure. Such a data structure maintains the SCCs of a graph under edge deletions in topological order. This is very useful for our algorithm, as adding vertices to X' is equivalent to deleting further edges from $G - E_G^{\text{out}}(X')$. We define the following interface for such a data structure.

Definition 4.7. *Given a graph $G = (V, E)$, $n = |V|$ and $m = |E|$, a* decremental strongly connected components *data structure is a data structure D with total update time $T(n, m)$ that offers the following operations:*

- *σ: Returns the topological ordering of the SCCs of G in time $\mathcal{O}(1)$.*
- *DELETE(u, v): Deletes the edge $(u, v) \in E$ from G and updates the data structure, such that the relative positioning of unaffected SCCs in σ does not change.*
- *SCC(u): Returns the SCC that contains vertex $u \in V$ in time $\mathcal{O}(1)$.*

Our algorithm can then use such a DSCC data structure D in the following way. We initialize D on $G - E_G^{\text{out}}(V')$ because then $D.\sigma$ contains every SCC of $G - E_G^{\text{out}}(V')$, and as such $D.\sigma$ can be used as a viable set $\mathcal{L}$ in a restricted partition. The algorithm will use $D.\sigma$ to implicitly maintain both $\mathcal{L}$ and $\mathcal{R}$. We do not explicitly remove a processed SCC L from $D.\sigma$ when adding it to $\mathcal{R}$, instead we keep track of the SCC R_L most recently moved to $\mathcal{R}$. Using R_L, we determine L by querying for the predecessor of R_L in $D.\sigma$. Note that this can be implemented, using a doubly linked-list for example, to run in constant time. Given the index $i \in \{1, ..., |D.\sigma|\}$ of R_L in $D.\sigma$ the implicitly maintained restricted partition is $\mathcal{L} = \{D.\sigma(1), ..., D.\sigma(i-1)\}$ and $\mathcal{R} = \{D.\sigma(i), ..., D.\sigma(|D.\sigma|)\}$. Moving L to $\mathcal{R}$ is then as easy as setting $R_L \leftarrow L$. If the algorithm instead adds vertices to X' we delete their outgoing edges using the data structure. This causes L to split into k new SCCs in $D.\sigma$. But because we required the data structure to not change the relative ordering of unaffected SCCs, all the newly created SCCs are inserted into $D.\sigma$ before R_L, which means that $\mathcal{R}$ is unaffected, and the implicitly maintained partition remains restricted.

With this approach, our algorithm can find a sink L in constant time. Another advantage is that now the algorithm is not responsible for recomputing the SCCs after adding vertices to X', instead it outsources the work to D. To show that such a data structure exists, in Theorem 4.8, we describe how to implement D with total update time $\mathcal{O}(m(n+m))$ using Tarjan's algorithm [Tar72].

Theorem 4.8. *Given a graph $G = (V, E)$ with $n = |V|$ and $m = |V|$ there exists a DSCC data structure D with total update time $\mathcal{O}(m(n+m))$.*

Proof. Initially, we compute all the SCCs of G with Tarjan's algorithm, which returns them in topological order. Additionally, for every vertex we store a pointer to the SCC that contains the vertex. This allows us to implement the SCC-operation to run in constant time. Because Tarjan's algorithm has running time $\mathcal{O}(n+m)$, the initialization takes time $\mathcal{O}(n+m)$.

When deleting an edge $(u, v) \in E$, we determine if both u and v are in the same SCC. If not, then the deletion does not affect the SCCs, and we just delete the edge in G. If they are contained in the same SCC Y we must run Tarjan's algorithm on the subgraph $G[Y]$ after deleting edge (u, v) in G to determine the newly created SCCs. We then replace Y with the new SCCs in σ and update for every vertex contained in Y the pointer to the SCC it is contained in. This needs time $\mathcal{O}(n+m)$, because the induced subgraph may be large. With at most m edge deletions the total update time is $\mathcal{O}(m(n+m))$.

Finally, we argue why σ stays topologically ordered after replacing an SCC Y with the SCCs of $G[Y]$ after an edge e was deleted. Assume that σ is not topologically ordered. Then there must be an edge $(z, w) \in E$ with $z \in Z$ and $w \in W$ for two SCCs Z and W where Z has a higher index than W in σ. At least one of the two SCCs must be a newly added SCC of $G[Y]$ that replaced Y because otherwise σ would not have been topologically ordered before the edge deletion. W.l.o.g let W be a new SCC that was added to σ when

replacing Y. Because the SCCs of $G[Y]$ were returned in topological order by Tarjan's algorithm the SCC Z cannot also be a new SCC. But because W is an SCC in $G[Y]$ it must also hold that $w \in Y$, which means that there was an edge from Z to Y before e was deleted. Because W replaced Y the index of Z in σ must be higher than that of Z. This means that σ was not topologically ordered before deleting e, which is a contradiction. □

Note that the running time $\mathcal{O}(m(n+m))$ of a DSCC data structure D as described in Theorem 4.8 simplifies to $\mathcal{O}(n^2)$ for CFGs because for those graphs $m = \mathcal{O}(n)$. We also want to give an example on why D has a worst-case running time of $\mathcal{O}(n^2)$. The CFG shown in Figure 4.6 contains vertices $V = \{v_1, ..., v_n\}$ and edges (v_i, v_{i+1}) for all $i \in \{1, ..., n-1\}$ and (v_j, v_1) for $j \in \{2, ..., n\}$. If we now iteratively delete the edge (v_i, v_{i+1}) at step $i = n-1, ..., 1$, we always need to run Tarjan's algorithm on the vertex set $V_i = \{v_1, ..., v_i\}$ because each V_i is an SCC after deleting the edge (v_i, v_{i+1}).

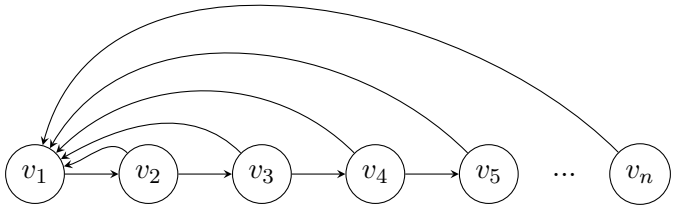

Figure 4.6: A graph where maintaining SCCs under edge deletions using Tarjan's algorithm has a quadratic running time.

4.1.2 Optimizing the Reachability-Computation

The second optimization to our current algorithm is to compute the sets $\hat{\Theta}(X', L)$ and $B(X', L)$ when processing an SCC L more efficiently instead of computing them from scratch in each iteration. Similar to the algorithm for acyclic graphs, we can store for each SCC $R \in \mathcal{R}$ the set $\hat{\Theta}(X', R)$. Then, given $\mathcal{R}_L = \{R \in \mathcal{R} \mid \exists (u, v) \in E_G^{\text{out}}(L) : v \in R\}$, we can compute:

$$\hat{\Theta}(X', L) = \bigcup_{R \in \mathcal{R}_L} \hat{\Theta}(X', R)$$

This computation takes time $\mathcal{O}(|E_G^{\text{out}}(L)|)$ because we just need to iterate over all the outgoing edges of the vertices in L, and by Corollary 4.2 every set $\hat{\Theta}(X', R)$ contains at most one element. Unfortunately we cannot guarantee that every edge is only processed a constant number of times, as a vertex can be contained in many different SCCs in $\mathcal{L}$ over the course of the algorithm, if the edge deletions cause the SCCs to decompose slowly. If this is the case, the outgoing edges of such a vertex may be processed $\mathcal{O}(n)$ times, which would result again in a quadratic running time.

The CFG G shown in Figure 4.7 is an example, where this happens. Initially, we have $X' = \{v_{n-1},\ v_n\}$ and $\mathcal{L}$ contains the SCCs $\{v_1,\ ...,\ v_{n-2}\}$, $\{v_{n-1}\}$ and $\{v_n\}$. The vertices in $L = \{v_1,\ ...,\ v_{n-2}\}$ all have two outgoing edges, which means to compute $\hat{\Theta}(X', L)$ we need to process $2 \cdot |L|$ edges. Then, because $|\hat{\Theta}(X', L)| \geq 2$, the algorithm adds $B(X', L) = \{v_{n-3},\ v_{n-2}\}$ to X', and L splits into SCCs $\{v_1,\ ...,\ v_{n-4}\}$, $\{v_{n-3}\}$ and $\{v_{n-2}\}$. To compute for $L' = \{v_1,\ ...,\ v_{n-4}\}$ the set $\hat{\Theta}(X', L')$ we need to again process $2 \cdot |L'|$ edges. For an even number $k \in \{2, ..., n-2\}$ every vertex set $\{v_1,\ ...,\ v_k\}$ is strongly connected, which means the algorithm will continuously add the two rightmost vertices to X' until every vertex is contained in its own single-vertex SCC. The total number of edges that were processed is $2 \cdot \sum_{i=1}^{\frac{n}{2}-1} 2i = \mathcal{O}(n^2)$.

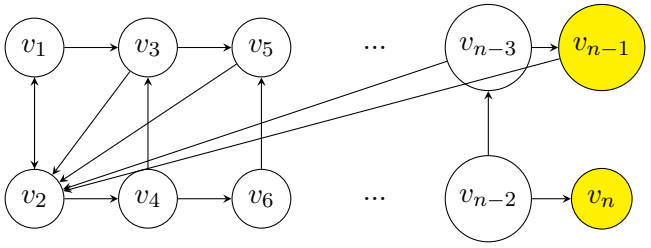

Figure 4.7: A CFG with n vertices and highlighted vertex set $X' = \{v_{n-1},\ v_n\}$.

But we can approach this differently. Instead of processing the outgoing edges we can pass the information along incoming edges after we finished processing an SCC and having moved it to $\mathcal{R}$. This ensures that a single edge is only processed a single time, as every SCC can only be moved to $\mathcal{R}$ once. Now, instead of computing $\hat{\Theta}(X', L)$ fully when processing SCC L, we partially compute the set in the iterations leading up to the processing of L. To more easily illustrate this approach, we need a notion of *restricted* reachability. We define for an SCC $L' \in \mathcal{L}$:

$$\hat{\Theta}_R(G, \mathcal{R}, X', L') := \hat{\Theta}(G[\bigcup \mathcal{R} \cup L'], X', L')$$

The set $\hat{\Theta}_R(G, \mathcal{R}, X', L')$ contains all the vertices from X' that vertices in L' can reach via X'-paths that only pass through L' and SCCs in $\mathcal{R}$. If the only X'-path to a vertex $x \in X$ from a vertex in L' passes through another SCC contained in $\mathcal{L}$ then $x \notin \hat{\Theta}_R(G, \mathcal{R}, X', L')$, even though $x \in \hat{\Theta}(X', L')$. Similarly, we want to compute $B(X', L')$ partially in the iterations leading up to the processing of L'. As such, we also define:

$$B_R(G, \mathcal{R}, X', L') := B(G[\bigcup \mathcal{R} \cup L'], X', L')$$

These sets have two important properties. When processing a sink $L \in \mathcal{L}$ the following equalities hold:

$$\hat{\Theta}_R(G, \mathcal{R}, X', L) = \hat{\Theta}(X', L) \text{ and } B_R(G, \mathcal{R}, X', L) = B(X', L) \tag{4.1}$$

This is simply a consequence of the fact that all the vertices that are reachable from a vertex contained in L are contained in $G[\bigcup \mathcal{R} \cup L]$. These equalities are very useful because if the algorithm compute $\hat{\Theta}_R(G, \mathcal{R}, X', L)$ and $B_R(G, \mathcal{R}, X', L)$ by propagating information along incoming edges of previously processed SCCs, we already know $\hat{\Theta}(X', L)$ and can immediately decide if $B(X', L) \subseteq X$ without further computation.

The second property is that these sets can be easily updated when moving an SCC L to $\mathcal{R}$, which we formalize in Lemma 4.9 and Lemma 4.10. Note that in the following, we also drop the argument G for those functions, just like for the $\hat{\Theta}$- and B-functions.

Lemma 4.9. *Let $G = (V, E)$ be a CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$ and let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$. Let $L \in \mathcal{L}$ be a sink and let $\mathcal{R}' = \mathcal{R} \cup \{L\}$.*

For an SCC $L' \in \mathcal{L}$ with $L' \neq L$, it holds that if there is an edge $(u, v) \in E_G^{\text{out}}(L')$ such that $v \in L$ then:

$$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L') \cup \hat{\Theta}_R(\mathcal{R}, X', L)$$

Otherwise it holds that:

$$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L')$$

Proof. Assume there is no edge to L. Then the only X'-paths that originate at vertices in L' in $G[\bigcup \mathcal{R}' \cup L']$ are the ones that are also present in $G[\bigcup \mathcal{R} \cup L']$. As such

$$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L')$$

Assume there is an edge $(u, v) \in E_G^{\text{out}}(L')$ such that $v \in L$. Clearly L' can still reach the vertices from X' in $G[\bigcup \mathcal{R}' \cup L']$ that it could also reach in $G[\bigcup \mathcal{R} \cup L']$ because $\mathcal{R} \subseteq \mathcal{R}'$. The only new X'-paths in $G[\bigcup \mathcal{R}' \cup L']$ that were not already present in $G[\bigcup \mathcal{R} \cup L']$ are the ones that pass through L. Therefore,

$$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L') \cup \hat{\Theta}_R(\mathcal{R}', X', L)$$

Then, because $G[\bigcup \mathcal{R}' \cup L] = G[\bigcup \mathcal{R} \cup L]$, it holds that:

$$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L') \cup \hat{\Theta}_R(\mathcal{R}, X', L)$$

□

Lemma 4.10. *Let $G = (V, E)$ be a CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$ and let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$. Let $L \in \mathcal{L}$ be a sink and let $\mathcal{R}' = \mathcal{R} \cup \{L\}$.*

For an SCC $L' \in \mathcal{L}$ with $L' \neq L$ and $U = \{u \in L' \mid \exists (u, v) \in E : v \in L\}$ it holds that if $\hat{\Theta}_R(\mathcal{R}, X', L) \neq \emptyset$ then:

$$B_R(\mathcal{R}', X', L') = B_R(\mathcal{R}, X', L') \cup U$$

Otherwise:

$$B_R(\mathcal{R}', X', L') = B_R(\mathcal{R}, X', L')$$

Proof. Clearly, every direct X'-path in $G[\bigcup \mathcal{R} \cup L']$ is still a direct X'-path in $G[\bigcup \mathcal{R}' \cup L']$, so in both cases it holds that $B_R(\mathcal{R}, X', L') \subseteq B_R(\mathcal{R}', X', L')$.

Now assume that $\hat{\Theta}_R(\mathcal{R}, X', L) \neq \emptyset$. Then every vertex in L has an X'-path in $G[\bigcup \mathcal{R} \cup L]$ originating at it. This X'-path is still present in $G[\bigcup \mathcal{R}' \cup L]$, so at every vertex $u \in U$, because there exists the edge $(u, v) \in E$ with $v \in L$, there originates the direct X'-path that traverses $v \in L$. Therefore, $U \subseteq B_R(\mathcal{R}', X', L')$. On the other hand consider a vertex $y \in L'$ such that neither $y \in U$ nor $y \in B_R(\mathcal{R}, X', L')$. Then clearly there does not originate a direct X'-path in $G[\bigcup \mathcal{R} \cup L']$ at y, and because there is also no direct X'-path trough L there is no direct X'-path in $G[\bigcup \mathcal{R}' \cup L']$ that originates at y. Therefore, $y \notin B_R(\mathcal{R}', X', L')$. We have shown that if $\hat{\Theta}_R(\mathcal{R}, X', L) \neq \emptyset$ then we have the equality:

$$B_R(\mathcal{R}', X', L') = B_R(\mathcal{R}, X', L') \cup U$$

Now assume that $\hat{\Theta}_R(\mathcal{R}, X', L) = \emptyset$. For every vertex $y \in L'$ with $y \notin B_R(\mathcal{R}, X', L')$ it holds that there does not originate a direct X'-path in $G[\bigcup \mathcal{R} \cup L']$ at y. Further, because L cannot reach a vertex contained in X' there is also no direct X'-path in $G[\bigcup \mathcal{R}' \cup L']$ that originates at y, even if y can reach L. Therefore, $y \notin B_R(\mathcal{R}', X', L')$. It follows that $B_R(\mathcal{R}', X', L') = B_R(\mathcal{R}, X', L')$ holds. □

Lastly, we need to determine what happens to the SCCs that are newly added to $\mathcal{L}$ after the algorithm adds vertices to X' and L is split because it is no longer strongly connected in $G - E_G^{\text{out}}(X')$.

Lemma 4.11. *Let $G = (V, E)$ be a CFG and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$ and let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$. Let $L \in \mathcal{L}$ be a sink, let $X'' = X' \cup B(X', L)$ and $\mathcal{L}'$ be the SCCs of $G[\bigcup \mathcal{L}] - E_G^{\text{out}}(X'')$.*

For every $L' \in \mathcal{L}'$ with $L' \subseteq L$ it holds that $B_R(\mathcal{R}, X'', L') = \emptyset$. Further, if L' contains a vertex $z \in B(X', L)$ it holds that $\hat{\Theta}_R(\mathcal{R}, X'', L') = \{z\}$, and otherwise $\hat{\Theta}_R(\mathcal{R}, X'', L') = \emptyset$.

Proof. Let $L' \in \mathcal{L}'$ such that $L' \subseteq L$. We consider two cases, in the first case L' contains a vertex from $B(X', L)$, and in the second it does not.

Assume L' contains a vertex $z \in B(X', Y)$. Then L' must be a single-vertex SCC only containing z because z has no outgoing edges in $G - E_G^{\text{out}}(X'')$. This implies that $B_R(\mathcal{R}, X'', L') = \emptyset$. Further, we have $z \in X''$, and therefore $\hat{\Theta}_R(\mathcal{R}, X'', L') = \{z\}$.

Now assume L' does not contain a vertex from $B(X', L)$. Assume that $\hat{\Theta}_R(\mathcal{R}, X'', L') \neq \emptyset$. Then there must be a vertex $z \in L'$ with a direct X''-path in $G[\bigcup \mathcal{R} \cup L']$. Consequently, every vertex on that path besides the start vertex must be contained in an SCC in $\mathcal{R}$. Then, because the only vertices contained in X'' that are also contained in SCCs of $\mathcal{R}$ are the vertices in X', the X''-path is an X'-path. But then, because $z \in L$, we would have that $z \in B(X', L)$. This is a contradiction. Therefore, it must hold that $\hat{\Theta}_R(\mathcal{R}, X'', L') = \emptyset$, and as such also $B_R(\mathcal{R}, X'', L') = \emptyset$. □

Finally, using these results and a DSCC data structure, we can describe an algorithm that computes the weak control closure in linear time, when neglecting the total update time of D.

Theorem 4.12. *Let $G = (V, E)$ be a CFG, $n = |V|$ and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Given a DSCC data structure D with total update time $T(n)$, there exists an algorithm that computes the weak control closure X of V' in G in time $\mathcal{O}(n + T(n))$.*

Proof. **Initialization:** The algorithm again initializes $X' = V'$ because V' is always contained in X, and now it additionally initializes D on $G - E_G^{\text{out}}(X')$. Initially, we have $\{\mathcal{L}, \mathcal{R}\} = \{D.\sigma, \emptyset\}$, which is an X'-weakly restricted partition. We additionally keep track of the SCC R_L most recently moved to $\mathcal{R}$, which, given its index i in $D.\sigma$, implicitly defines the X'-weakly restricted partition $\{\mathcal{L}, \mathcal{R}\}$ with $\mathcal{L} = \{D.\sigma(1), ..., D.\sigma(i-1)\}$ and $\mathcal{R} = \{D.\sigma(i), ..., D.\sigma(|D.\sigma|)\}$. In the first iteration, because we did not yet move an SCC to $\mathcal{R}$, we can just pick $L = D.\sigma(|D.\sigma|)$, meaning the last element in the topological ordering, as this SCC cannot have any outgoing edges. Over the course of the algorithms' execution, we ensure that before each iteration $X' \subseteq X$ holds and that for each $L' \in \mathcal{L}$ we know the sets $\hat{\Theta}_R(\mathcal{R}, X', L')$ and $B_R(\mathcal{R}, X', L')$. Initially, because $\mathcal{R}$ is empty, the set $B_R(\mathcal{R}, X', L')$ is also empty. If L' does not contain a vertex from V' then $\hat{\Theta}_R(\mathcal{R}, X', L')$ is also empty, and otherwise $\hat{\Theta}_R(\mathcal{R}, X', L') = \{x\}$ for the single vertex $x \in L'$. Thus, the initialization can be implemented to run in linear time plus the time needed for initializing D.

Iteration: At each iteration besides the first, we query $L = D.\sigma.\textsc{Pre}(R_L)$, which is the predecessor of R_L in $D.\sigma$ because then L does not reach any other SCC in $\mathcal{L}$. Note that in the first iteration L will always be moved to $\mathcal{R}$ because it has no outgoing edges and thus can only contain X-weakly committing vertices, so querying the predecessor of R_L in the subsequent iterations is well-defined. For L we know the set $\hat{\Theta}_R(\mathcal{R}, X', L)$ already and by Equation (4.1) also $\hat{\Theta}(X', L)$.

This means we do not have to spend time to compute $\hat{\Theta}(X', L)$, because we already know the contents of this set and can immediately execute one of the two following branches.

- $|\hat{\Theta}(X', L)| \leq 1$: In this case, by Lemma 4.5, the partition $\{\mathcal{L}', \mathcal{R}'\}$ with $\mathcal{L}' = \mathcal{L} \backslash \{L\}$ and $\mathcal{R}' = \mathcal{R} \cup \{L\}$ is an X'-weakly restricted partition for the SCCs of $G - E_G^{\text{out}}(X')$. Note that after setting R_L to L, we implicitly have that $\mathcal{R} = \mathcal{R}'$, so we only need to compute the sets $\hat{\Theta}_R(\mathcal{R}', X', L')$ and $B_R(\mathcal{R}', X', L')$ for all $L' \in \mathcal{L}$ before continuing with the next iteration.

  We iterate over all the edges $(u, v) \in E_G^{\text{in}}(L)$. Because no SCC in R can reach L, we know that all those edges must come from an SCC $L' \in \mathcal{L}$. If $L' \neq L$, then by Lemma 4.9 we have:

  $$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L') \cup \hat{\Theta}_R(\mathcal{R}, X', L)$$

  Note that we already know the contents of both $\hat{\Theta}_R(\mathcal{R}, X', L')$ and $\hat{\Theta}_R(\mathcal{R}, X', L)$. Then, because $\hat{\Theta}_R(\mathcal{R}, X', L) = \hat{\Theta}(X', L)$ and $|\hat{\Theta}(X', L)| \leq 1$ we can compute the union of the two sets in constant time by adding at most one element to $\hat{\Theta}_R(\mathcal{R}, X', L')$.

  Additionally, if $|\hat{\Theta}(X', L)| = 1$ we add u to $B_R(\mathcal{R}, X', L')$, because by Lemma 4.10 the set $B_R(\mathcal{R}', X', L')$ contains $B_R(\mathcal{R}, X', L')$, and every vertex in L' that has an outgoing edge to L.

  Afterward, we set $R_L \leftarrow L$, which results in $\mathcal{L} = \mathcal{L}'$ and $\mathcal{R} = \mathcal{R}'$. Therefore, our invariant that for each $L' \in \mathcal{L}$ we know the sets $\hat{\Theta}_R(\mathcal{R}, X', L')$ and $B_R(\mathcal{R}, X', L')$ again holds, and because X' was unchanged, it also still holds that $X' \subseteq X$.

- $|\hat{\Theta}(X', L)| \geq 2$: Because L cannot reach any other SCCs contained in $\mathcal{L}$, and we already know $B_R(\mathcal{R}, X', L)$, by Equation (4.1) we also get $B(X', L)$ without further computation. We add $B(X', L)$ to X' and delete $E_G^{\text{out}}(B(X', L))$ from D. By Lemma 4.3, it still holds that $X' \subseteq X$. This deletion causes L to split into new SCCs. But because the relative ordering of SCCs in $\mathcal{R}$ is unaffected by the deletion, not modifying R_L ensures that the partition $\{\mathcal{L}, \mathcal{R}\}$ is again an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$, by Lemma 4.4.

  For a new SCC L' we still need to compute $\hat{\Theta}_R(\mathcal{R}, X', L')$ and $B_R(\mathcal{R}, X', L')$. By Lemma 4.11 the set $B_R(\mathcal{R}, X', L')$ is empty regardless of L'. Then we decide if L' is a single-vertex SCC that contains a vertex $x \in X'$, and if so, by the same lemma, we set $\hat{\Theta}_R(\mathcal{R}, X', L') = \{x\}$, otherwise we have $\hat{\Theta}_R(\mathcal{R}, X', L') = \emptyset$. This branch can be implemented to run in time $\mathcal{O}(|B(X', L)|)$, disregarding the update time of D.

Running Time: Like the algorithm in Theorem 4.6, this algorithm executes both branches at most n times, so it terminates after at most $2n$ iterations. The first branch can be implemented to run in time $\mathcal{O}(|E_G^{\text{in}}(L)|)$ and because every SCC can only be moved once from $\mathcal{L}$ to $\mathcal{R}$, which implies that every incoming edge is processed once, the total time spent in the first branch is $\mathcal{O}(n)$. The second branch, without considering the update time of D, only takes time $\mathcal{O}(|B(X', L)|)$. Because every vertex in $B(X', L)$ is only added once to X', and X' can contain at most n vertices, the total time spent in this branch is also $\mathcal{O}(n)$. As such, the total running time of the algorithm is $\mathcal{O}(n)$ plus the total update time $T(n)$ of D.

Correctness: The argument for why the computed set X' is the weak control closure is the same as in Theorem 4.6. □

When using Tarjan's algorithm to implement D as shown in Theorem 4.8 we get a total running time of $\mathcal{O}(n^2)$ for the algorithm in Theorem 4.12. We also want to briefly mention that on acyclic graphs the running time is even linear, as there are only single-vertex SCCs, which means we do not recompute SCCs with Tarjan's algorithm, and the initialization of D needs only linear time.

4.2 Pseudocode for Computing the Weak Control Closure

In Algorithm 4.1 we also provide the pseudocode for the algorithm described in Theorem 4.12. The following methods not previously defined were used:

- RemoveOutgoingEdges(H, V'): Removes the edges $E_H^{\text{out}}(V')$ from H.
- InitializeDSCC(G): Initializes a DSCC data structure D on H.
- σ.Pre(R_L): Returns the predecessor of an SCC R_L in σ, and if $R_L = \text{null}$ then it returns the last SCC in σ.

For an SCC L the set $\Theta(L)$ stores $\hat{\Theta}_R(\mathcal{R}, X, L)$ for the by D implicitly maintained X-weakly restricted partition $\{\mathcal{L}, \mathcal{R}\}$, and $B(L)$ stores $B_R(\mathcal{R}, X, L)$. Observe that the pseudocode does not exactly represent the described algorithm. In Theorem 4.12 we stated that in the initialization for an SCC $L' = \{v'\}$ with $v' \in V'$ we must set $\Theta(L') = \{v'\}$, and after deleting edges from D for a newly created SCC $L'' = \{x\}$ with $x \in X$ we must set $\Theta(L'') = \{x\}$ by Lemma 4.11. In the implementation this is not immediately necessary, neither L' nor L'' will ever be split because they are single-vertex SCCs without outgoing edges. The contents of $\Theta(L')$ and $\Theta(L'')$ only become necessary for propagating X-path information, which is why we postponed the computation to line 12. Additionally, as a shorthand for iterating over all the edges $(u, v) \in E_G^{\text{out}}(B(L))$ and individually calling D.Delete(u, v) to remove all the outgoing edges of the vertices in $B(L)$ from D we just write D.Delete$(B(L))$ in line 9.

Algorithm 4.1: WeakControlClosure$(G,\ V')$

```
   Input  : A CFG G = (V, E) and a vertex set V' ⊆ V.
   Output : The weak control closure X ⊆ V of V' in G.
 1 H ← RemoveUnreachable(G, V');
 2 H ← RemoveOutgoingEdges(H, V');
 3 D ← InitializeDSCC(H);
 4 X ← V';
 5 R_L ← null;
 6 while (L ← D.σ.Pre(R_L)) ≠ null do
 7     if |Θ(L)| ≥ 2 then
 8         X ← X ∪ B(L);
 9         D.Delete(B(L));
10     else
11         if L = {x} for x ∈ X then
12             Θ(L) ← {x};
13         if |Θ(L)| ≥ 1 then
14             for v ∈ L do
15                 for (u, v) ∈ E(H) do
16                     X_u ← D.SCC(u);
17                     X_v ← D.SCC(v);
18                     if X_u ≠ X_v then
19                         B(X_u) ← B(X_u) ∪ {u};
20                         Θ(X_u) ← Θ(X_u) ∪ Θ(X_v);
21         R_L ← L;
22 return X;
```

4.3 Computing the Strong Control Closure

Given a CFG $G = (V, E)$, a vertex set $V' \subseteq V$ and the predicate vertices $P \subseteq V$ of G, we want to compute the strong control closure X of V' in G for P. Again we assume that G only contains V'-reachable vertices because we can just remove all the vertices that are not V'-reachable by Theorem 2.24. The algorithm is similar to the one computing the weak control closure, which means it also keeps track of a set X' that initially contains all the vertices from V', and adds vertices to X' such that at the end of the algorithm we have $X' = X$. The algorithm again works on a restricted partition $\{\mathcal{L}, \mathcal{R}\}$ of the SCCs of $G - E_G^{\text{out}}(X')$, but now every vertex in an SCC in $\mathcal{R}$ must be X'-strongly committing instead of just X'-weakly committing.

Definition 4.13. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G, let $V' \subseteq V$ be a vertex set and $X' \supseteq V'$. A partition $\{\mathcal{L}, \mathcal{R}\}$ of the SCCs of $G - E_G^{\text{out}}(X')$ is called* X'-strongly restricted *if the following hold.*

- *For all $R \in \mathcal{R}$ and every edge $(u, v) \in E_G^{\text{out}}(R)$ it holds that $v \in R'$ for an SCC $R' \in \mathcal{R}$.*
- *For all $R \in \mathcal{R}$ and every vertex $v \in R$ it holds that v is X'-strongly committing in G for P or X'-avoiding in G.*

A result of Definition 4.13 is that every SCC $R \in \mathcal{R}$ can at most reach one distinct SCC that contains a vertex from X', which was also the case for X'-weakly restricted partitions. In addition to that every vertex $v \in R$ can only lie on an X'-free complete path if R cannot reach any SCC that contains a vertex from X'. This is because if v would lie on such a complete path, and v could also reach a vertex contained in X', it would not be X'-strongly committing nor X'-avoiding.

Corollary 4.14. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$ and let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-strongly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.*

For an SCC $R \in \mathcal{R}$ it holds that $|\hat{\Theta}(X', R)| \leq 1$ and if $|\hat{\Theta}(X', R)| = 1$, then for all $v \in R$ it holds that $v \notin \Gamma(G, P, X')$.

Like before, the algorithm chooses an SCC $L \in \mathcal{L}$ that can only reach itself and SCCs in $\mathcal{L}$, and then determines if L contains vertices that are also contained in X. If so, we again do not need to find all the vertices from L that are contained in X, it is enough to determine a subset of those vertices, add them to X', split L because it is no longer strongly connected in $G - E_G^{\text{out}}(X')$, and let the algorithm proceed.

To decide whether the algorithm should add vertices to X' we now also need to consider complete paths. For this, we define the set $C(G, P, X', L)$, which contains all the vertices from L that lie on a direct X'-free complete path in G. Note that P is implicitly used to identify the final vertices needed to determine if a path is complete or not. Because the arguments G and P will not change, we drop them and only write $C(X', L)$. Note that $C(X', L) \subseteq \Gamma(G, P, X')$.

In this algorithm, we distinguish between multiple cases for when to add vertices to X'. Independent of which of the following cases holds, we add both the vertices contained in $B(X', L)$ and $C(X', L)$ to X'. We will give the formal proof for why these vertices are contained in X after describing the different cases, but the intuition is that the vertices in those sets always lie on either two disjoint X'-paths or on an X'-path and a disjoint X'-free complete path.

The first case where we add vertices to X' is when $|\hat{\Theta}(X', L)| \geq 2$ holds. In this case, just like in the context of computing the weak control closure, every vertex in L is not X'-weakly committing as it can reach two different vertices from X'. Every vertex $v \in B(X', L)$ can reach a vertex r that can reach a vertex in X' and that is contained in an SCC in $\mathcal{R}$, which means $|\Theta(G, X', r)| = 1$, and therefore, $v \in X$ by Lemma 2.15. For a vertex $v \in C(X', L)$, we will show that there exists an X'-path originating at v that is disjoint from the complete path originating at v. An example for this case is illustrated in Figure 4.8. For $X' = \{v_4,\ v_7\}$ and $L = \{v_1,\ v_2\ , v_3\}$ we have $\hat{\Theta}(X', L) = \{v_4,\ v_7\}$, $B(X', L) = \{v_2,\ v_3\}$ and $C(X', L) = \{v_1\}$. The edges (v_2, v_7) and (v_3, v_4) then satisfy the first two conditions of Lemma 2.15. Therefore, if they also are V'-reachable in G, they are contained in X. The vertex v_1 lies on the complete path $v_1, v_5, v_6, v_5, ...$, but also on the X'-path v_1, v_2, v_7. By Lemma 2.23, if we again assume that v_1 is V'-reachable, it then also holds that $v_1 \in X$.

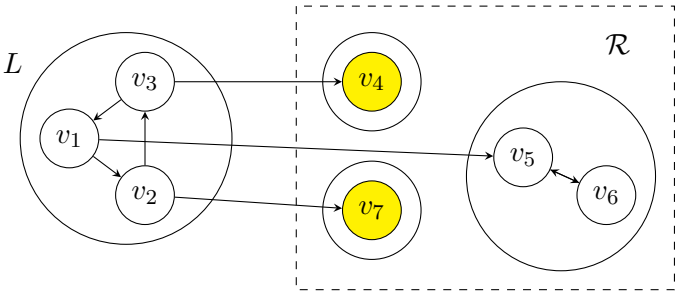

Figure 4.8: An SCC $L = \{v_1,\ v_2\ , v_3\}$ and SCCs $\mathcal{R}$ with vertices $X' = \{v_4,\ v_7\}$ highlighted.

In the remaining cases we always have that $|\hat{\Theta}(X', L)| = 1$, but L also always contains a vertex from $\Gamma(G, P, X')$. We will later show that if those two conditions hold, every vertex from $B(X', L)$ as well as $C(X', L)$ lies on an X'-path and a disjoint X'-free complete path. If the vertices are additionally V'-reachable then by Lemma 2.23 they are all contained in X. In the second case, in addition to $|\hat{\Theta}(X', L)| = 1$, it also holds that $|C(X', L)| \geq 1$. Clearly, L contains a vertex from $\Gamma(G, P, X')$, because $C(X', L) \subseteq \Gamma(G, P, X')$. Consider the SCC L depicted in Figure 4.9 and $X' = \{v_4\}$. The vertex $v_3 \in B(X', L)$ lies on the X'-path v_3, v_4 and the disjoint X'-free complete path $v_3, v_1, v_2, v_5, v_6, v_5, ...$ Vertex $v_2 \in C(X', L)$ lies on an X'-free complete path and the disjoint X'-path v_2, v_3, v_4.

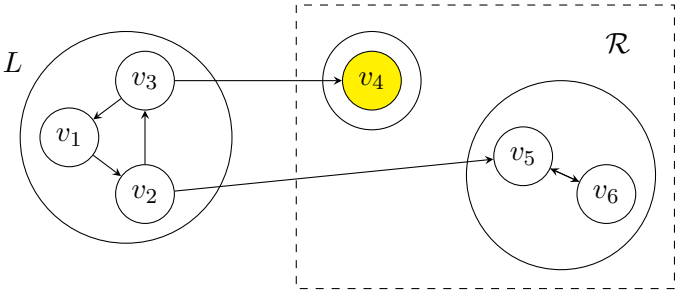

Figure 4.9: An SCC $L = \{v_1,\ v_2\ , v_3\}$ and SCCs $\mathcal{R}$ with vertex set $X' = \{v_4\}$ highlighted.

Thirdly, we have the case where $|L| \geq 2$, because then every vertex in L lies on an infinite path, which means $L \subseteq \Gamma(G, P, X')$. Another example is illustrated in Figure 4.10. Here, $v_1 \in B(X', L)$ lies on the X'-path v_1, v_3 and the disjoint complete path $v_1, v_2, v_1, ...$, which does not contain a vertex from X'.

In the fourth case L only contains a single vertex v, but v is an incomplete predicate, meaning $v \in P$ with $\deg^+(v) < 2$, and thus $v \in \Gamma(G, P, X')$. In Figure 4.11 we have an example for this case. If $P = \{v_1\}$ then v_1 is an incomplete predicate vertex, and as such

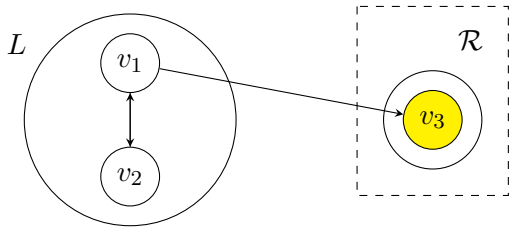

Figure 4.10: An SCC $L = \{v_1,\ v_2\}$ and SCC $\{v_3\} \in \mathcal{R}$ with vertex set $X' = \{v_3\}$ highlighted.

lies on the trivial complete path v_1. Additionally, v_1 lies on the disjoint X'-path v_1, v_2 for $X' = \{v_2\}$.

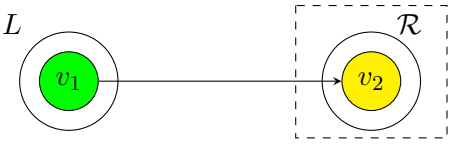

Figure 4.11: An SCC $L = \{v_1\}$ and SCC $\{v_2\} \in \mathcal{R}$ with vertex set $X' = \{v_2\}$ highlighted in yellow and $P = \{v_1\}$ highlighted in green.

Lastly, L also only contains a single vertex v, but the self loop $(v, v) \in E$ exists. Then v lies on the infinite path $v, v, ...$, and as such $v \in \Gamma(G, P, X')$. An example for this is given in Figure 4.12 where v_1 lies on the complete path $v_1, ...$, and the disjoint X-path v_1, v_2.

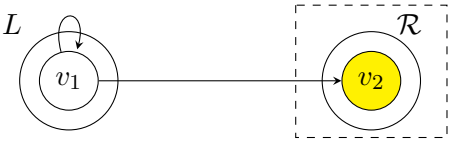

Figure 4.12: An SCC $L = \{v_1\}$ and SCC $\{v_2\} \in \mathcal{R}$ with vertex set $X' = \{v_2\}$ highlighted.

Lemma 4.15. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G, let $V' \subseteq V$ be a vertex set such that G only contains V'-reachable vertices and let X be the strong control closure of V' in G for P. Further, let $V' \subseteq X' \subseteq X$, let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-strongly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$ and $L \in \mathcal{L}$ a sink.*

It holds that $B(X', L) \cup C(X', L) \subseteq X$ if one of the following conditions is satisfied:

- $|\hat{\Theta}(X', L)| \geq 2$
- $|\hat{\Theta}(X', L)| = 1$ *and* $|C(X', L)| \geq 1$
- $|\hat{\Theta}(X', L)| = 1$ *and* $|L| \geq 2$
- $|\hat{\Theta}(X', L)| = 1$, $L = \{v\}$, $v \in P$ *and* $\deg^+(v) < 2$.
- $|\hat{\Theta}(X', L)| = 1$, $L = \{v\}$ *and* $(v, v) \in E$

Proof. Note that if L contains a vertex from X' it is a single-vertex SCC without outgoing edges in $G - E_G^{\text{out}}(X')$, and as such $B(X', L) = C(X', L) = \emptyset$. Therefore, we assume that L does not contain a vertex from X'.

We first proof another result. Assume that $|\hat{\Theta}(X', L)| \geq 1$ holds and there exists a vertex $v \in L$ for which $v \in \Gamma(G, P, X')$. We show that then $B(X', L) \cup C(X', L) \subseteq X$ holds. Let $p \in B(X', L)$, which exists because $|\hat{\Theta}(X', L)| \geq 1$. Then there exists the direct X'-path

$p, r, ..., x$ with $r \in R$ for an SCC $R \in \mathcal{R}$ and $x \in X'$. Further, because $v \in \Gamma(G, P, X')$ and L is strongly connected, it also holds that $p \in \Gamma(G, P, X')$. Consequently, there exists an X'-free complete path $p, ...$ Because it holds that $|\hat{\Theta}(X', R)| = 1$, this complete path must be disjoint from the X'-path $p, r, ..., x$ because if it were not, we would have $r \in \Gamma(G, P, X')$ which is a contradiction to Corollary 4.14. Then, because p is also V'-reachable, by Lemma 2.23 we have that $p \in X$. Now let $p' \in C(X', L)$, if such a vertex exists. Then there exists the direct complete path $p', r', ...$, and because $|\hat{\Theta}(X', L)| \geq 1$, there must also be an X'-path $p', ..., x'$ to a vertex $x' \in X'$. Because $r' \in R$, by Corollary 4.14, it must hold that $|\Theta(G, X', r')| = 0$, which means the two paths are disjoint and because p' is also V'-reachable, again by Lemma 2.23 we have that $p' \in X$.

For every of the five cases above we have that $|\hat{\Theta}(X', L)| \geq 1$ holds. In the first case, if $C(X', L) \neq \emptyset$ then L contains a vertex from $\Gamma(G, P, X')$ because $C(X', L) \subseteq \Gamma(G, P, X')$. In the second case, using the same reasoning, we also have that L contains a vertex from $\Gamma(G, P, X')$. Then, in the third and fifth case, every vertex contained in L lies on an infinite path and is therefore contained in $\Gamma(G, P, X')$. For the fourth case we have that $v \in L$ is a final vertex, which means it lies on the trivial complete path v, and as such is also contained in $\Gamma(G, P, X')$. By the previous observation we have that $B(X', L) \cup C(X', L) \subseteq X$ holds.

Finally, if $|\hat{\Theta}(X', L)| \geq 2$ holds, and $C(X', L) = \emptyset$. Then for every vertex $p \in B(X', L)$ we have $|\Theta(G, X, p)| \geq 2$, and p lies on a direct X'-path $p, r, ..., x$ with $r \in R$ for an SCC $R \in \mathcal{R}$ and $x \in X'$. This means that r is X'-strongly committing, and we have $|\Theta(G, X', r)| = 1$. Because p is also V'-reachable, by Lemma 2.15, vertex p is contained in every weakly control closed superset of V', and because X is such a set we have that $p \in X$. As $C(X', L) = \emptyset$ we have shown the claim also in this case. □

Lemma 4.15 will be used later to determine whether the algorithm should add vertices to X' or not. If we do add vertices to X' we need to delete edges from L and update $\mathcal{L}$ to correctly represent SCCs of $G - E_G^{\text{out}}(X')$ again. Afterward, the partition $\{\mathcal{L}, \mathcal{R}\}$ should again be X'-strongly restricted.

Lemma 4.16. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$ and let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-strongly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.*

Let $L \in \mathcal{L}$ be a sink, let $X'' = X' \cup B(X', L) \cup C(X', L)$ and $\mathcal{L}'$ be the SCCs of $G[\bigcup \mathcal{L}] - E_G^{\text{out}}(X'')$. Then $\{\mathcal{L}', \mathcal{R}\}$ is an X'-weakly restricted partition of the SCCs of $G - E_G^{\text{out}}(X'')$.

Proof. Every SCC in $\mathcal{L}'$ is an SCC of $G - E_G^{\text{out}}(X'')$, and because $B(X', L) \cup C(X', L) \not\subseteq \bigcup \mathcal{R}$, the SCCs in $\mathcal{R}$ are unaffected by the removal of the edges $E_G^{\text{out}}(B(X', L))$, which means $\mathcal{R}$ are also SCCs of $G - E_G^{\text{out}}(X'')$. The first condition of Definition 4.13 clearly still holds because $G - E_G^{\text{out}}(X'')$ contains fewer edges than $G - E_G^{\text{out}}(X')$. Additionally, also because $B(X', L) \cup C(X', L) \not\subseteq \bigcup \mathcal{R}$, the X''-paths and complete paths in $G[\bigcup \mathcal{R}]$ are exactly the X'-paths and complete paths in $G[\bigcup \mathcal{R}]$, which means that every vertex in $\mathcal{R}$ is also X''-strongly committing or X''-avoiding, and the second condition of Definition 4.13 is also satisfied. □

If the algorithm does not add vertices to X' it moves L from $\mathcal{L}$ to $\mathcal{R}$ to still make some kind of progress towards termination. After moving L the partition $\{\mathcal{L}, \mathcal{R}\}$ must still be X'-strongly restricted, which we prove in Lemma 4.17.

Lemma 4.17. *Let $G = (V, E)$ be a CFG, $P \subseteq V$ the predicate vertices of G and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Further, let $X' \supseteq V'$, let $\{\mathcal{L}, \mathcal{R}\}$ be an X'-strongly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$ and $L \in \mathcal{L}$ a sink.*

The partition $\{\mathcal{L}\backslash\{L\}, \mathcal{R} \cup \{L\}\}$ *is an* X'*-strongly restricted partition of the SCCs of* $G - E_G^{\text{out}}(X')$ *if none of the following conditions hold:*

- $|\hat{\Theta}(X', L)| \geq 2$.
- $|\hat{\Theta}(X', L)| = 1$ *and* $|C(X', L)| \geq 1$.
- $|\hat{\Theta}(X', L)| = 1$ *and* $|L| \geq 2$.
- $|\hat{\Theta}(X', L)| = 1$, $L = \{v\}$, $v \in P$ *and* $\deg^+(v) \leq 1$.
- $|\hat{\Theta}(X', L)| = 1$, $L = \{v\}$ *and* $(v, v) \in E$.

Proof. Assume none of the conditions above hold. We show that every vertex $v \in L$ is X'-strongly committing or X'-avoiding. We know that v is X'-weakly committing because $|\hat{\Theta}(X', L)| \leq 1$. As such, v can only be not-X'-strongly committing and not X'-avoiding if $|\Theta(G, X', v)| = 1$ holds, and it lies on an X'-free complete path. Assume this is the case. Then $|\hat{\Theta}(X', L)| = 1$ must hold. Because the second condition is not satisfied, we must have $|C(X', L)| = 0$. Therefore, v can not lie on a direct X'-free complete path. Further, because the third and fifth conditions are also not satisfied, we must have $|L| = 1$ and $(v, v) \notin E$, so v does not lie on an infinite path. The only remaining possibility for a complete path that originates at v and does not contain a vertex from X' is the trivial path v itself with v being a final vertex. If v was a non-predicate vertex with no outgoing edges then it would clearly be X'-strongly committing or X'-avoiding, so it must be an incomplete predicate vertex. But this can also not be the case because then the fourth condition would be satisfied. Therefore, v cannot lie on an X'-free complete path, which is a contradiction.

Let $\mathcal{L}' = \mathcal{L}\backslash\{L\}$ and $\mathcal{R}' = \mathcal{R} \cup \{L\}$. Clearly, $\mathcal{L}'$ and $\mathcal{R}'$ only contain SCCs of $G - E_G^{\text{out}}(X')$. Then, because L can only reach SCCs in $\mathcal{R}$ and itself, every SCC in $\mathcal{R}'$ can only reach vertices in $\mathcal{R}'$, which means the first condition of Definition 4.13 is satisfied. Further, we have shown that every vertex in L is X'-strongly committing or X'-avoiding, and because every vertex contained in an SCC in $\mathcal{R}$ is already X'-strongly committing or X'-avoiding, every vertex contained in an SCC in $\mathcal{R}'$ must be X'-strongly committing or X'-avoiding, and the second condition of Definition 4.13 is also satisfied. □

Recall that the algorithm described in Theorem 4.12 did not recompute the sets $\hat{\Theta}(X', L)$ and $B(X', L)$ from scratch in each iteration and instead maintained the sets $\hat{\Theta}_R(\mathcal{R}, X', L')$ and $B_R(\mathcal{R}, X', L')$ for every $L' \in \mathcal{L}$ over the course of the algorithm. The algorithm to compute the strong control closure does the same, but it also needs to compute the set $C(X', L)$. As such we define:

$$C_R(G, P, \mathcal{R}, X', L') := C(G[\bigcup \mathcal{R} \cup L'], P, X', L')$$

For an SCC L that can only reach itself and SCCs in $\mathcal{R}$ it holds that

$$C_R(G, P, \mathcal{R}, X', L) = C(X', L) \tag{4.2}$$

Note that we again drop the arguments G and P and only write $C_R(\mathcal{R}, X', L)$. The algorithm can efficiently compute those sets for the SCCs in $\mathcal{L}$ after moving L from $\mathcal{L}$ to $\mathcal{R}$ by iterating over the incoming edges of L.

Lemma 4.18. *Let* $G = (V, E)$ *be a CFG,* $P \subseteq V$ *the predicate vertices of* G *and* $V' \subseteq V$ *a vertex set such that* G *only contains* V'*-reachable vertices. Further, let* $X' \supseteq V'$ *and let* $\{\mathcal{L}, \mathcal{R}\}$ *be an* X'*-strongly restricted partition of the SCCs of* $G - E_G^{\text{out}}(X')$*. Let* $L \in \mathcal{L}$ *be a sink and let* $\mathcal{R}' = \mathcal{R} \cup \{L\}$.

For an SCC $L' \in \mathcal{L}$ *with* $L' \neq L$ *and* $U = \{u \in L' \mid \exists (u, v) \in E : v \in L\}$ *it holds that* $C_R(\mathcal{R}', X', L') = C_R(\mathcal{R}, X', L') \cup U$ *if one of the following holds:*

- $C_R(\mathcal{R}, X', L) \neq \emptyset$.
- $|L| \geq 2$.
- $|L| = \{v\}$, *vertex* v *is a final vertex and* $v \notin X'$.
- $|L| = \{v\}$, $(v, v) \in E$ *and* $v \notin X'$.

If none of the conditions are satisfied then: $C_R(\mathcal{R}', X', L') = C_R(\mathcal{R}, X', L')$

Proof. Every direct complete path in $G[\bigcup \mathcal{R} \cup L']$ is still a direct complete path in $G[\bigcup \mathcal{R}' \cup L']$ because $\mathcal{R} \subseteq \mathcal{R}'$. Consequently, every vertex contained in $C_R(\mathcal{R}, X', L')$ still lies on a direct X'-free complete path in $G[\bigcup \mathcal{R}' \cup L']$ and is as such contained in $C_R(\mathcal{R}', X', L')$ independent of whether one of the conditions is satisfied. It holds that

$$C_R(\mathcal{R}, X', L') \subseteq C_R(\mathcal{R}', X', L')$$

Let $u \in U$. If $C_R(\mathcal{R}, X', L) \neq \emptyset$ then a vertex in L lies on a direct X'-free complete path. We also know that L does not contain a vertex contained in X' because those vertices do not have outgoing edges. Therefore, because u can reach a vertex in L, it also lies on a direct X'-free complete path. In the second case where $|L| > 1$, it again holds that L cannot contain a vertex from X', and because L contains at least two vertices u lies on an infinite path, as such $u \in C_R(\mathcal{R}', X', L')$. For the other two cases it holds that L only contains a single vertex that is not contained in X'. In the third case the vertex itself is a final vertex, so the path u, v is a complete path, and in the fourth case u lies on the infinite path $u, v, v, ...$ Again it holds that $u \in C_R(\mathcal{R}', X', L')$. We have shown that if one of the conditions is satisfied then:

$$C_R(\mathcal{R}, X', L') \cup U \subseteq C_R(\mathcal{R}', X', L')$$

Let $u \in C_R(\mathcal{R}', X', L')$, which means u lies on a direct complete path $u, v, ...$ If $v \in R$ for an SCC $R \in \mathcal{R}$ then $u \in C_R(\mathcal{R}, X', L')$ holds. If $v \in L$ then $u \in U$. We have $C_R(\mathcal{R}', X', L') \subseteq C_R(\mathcal{R}, X', L') \cup U$, and the claimed equality holds.

Finally, we show that if none of the conditions are satisfied, then there does not exist a direct X'-free complete path $u, v, ...$ with $v \in L$. Then $C_R(\mathcal{R}', X', L') \subseteq C_R(\mathcal{R}, X', L')$, and the second equality of the claim holds. Assume that none of the conditions hold and let $u \in U$. Assume that there exists a direct X'-free complete path $u, v, ...$ with $v \in L$. Then there must also exist an X'-free complete path $v, ...$ Note that this implies that $v \notin X'$. Because $C_R(\mathcal{R}, X', L) = \emptyset$, this complete path cannot leave L, and as such only contains vertices from L. Then, as $|L| = 1$, it must either be the trivial complete path v or the infinite path $v, v, ...$ But because the third condition is not satisfied, v is not a final vertex, and the trivial path v is not complete. Lastly, because the fourth condition is also not satisfied the edge (v, v), and therefore the path $v, v, ...,$ does not exist. This is a contradiction. □

After adding vertices to X' we get new SCCs in $\mathcal{L}$. For a new SCC L' we need to compute the sets $\hat{\Theta}_R(\mathcal{R}, X', L')$, $B_R(X', L')$ and $C_R(X', L')$. Fortunately these sets are mostly empty and therefore do not require a lot of time to compute.

Lemma 4.19. *Let* $G = (V, E)$ *be a CFG,* $P \subseteq V$ *the predicate vertices of* G *and* $V' \subseteq V$ *a vertex set such that* G *only contains* V'*-reachable vertices. Further, let* $X' \supseteq V'$ *and let* $\{\mathcal{L}, \mathcal{R}\}$ *be an* X'*-strongly restricted partition of the SCCs of* $G - E_G^{\text{out}}(X')$*. Let* $L \in \mathcal{L}$ *be a sink, let* $X'' = X' \cup B(X', L) \cup C(X', L)$ *and* $\mathcal{L}'$ *be the SCCs of* $G[\bigcup \mathcal{L}] - E_G^{\text{out}}(X'')$*.*

For every $L' \in \mathcal{L}'$ with $L' \subseteq L$ it holds that $B_R(\mathcal{R}, X'', L') = C_R(\mathcal{R}, X'', L') = \emptyset$. Further, if L' contains a vertex $z \in B(X', L) \cup C(X', L)$ it holds that $\hat{\Theta}_R(\mathcal{R}, X'', L') = \{z\}$, and otherwise $\hat{\Theta}_R(\mathcal{R}, X'', L') = \emptyset$.

Proof. Let $L' \in \mathcal{L}'$ such that $L' \subseteq L$. We consider two cases, in the first case L' contains a vertex from $B(X', L) \cup C(X', L)$, and in the second it does not.

Assume L' contains a vertex $z \in B(X', L) \cup C(X', L)$. Then L' must be a single-vertex SCC only containing z because z has no outgoing edges in $G - E_G^{\text{out}}(X'')$. A direct result of z having no outgoing edges is that $B_R(\mathcal{R}, X'', L') = C_R(\mathcal{R}, X'', L') = \emptyset$. Further, we have $z \in X''$, and therefore $\hat{\Theta}_R(\mathcal{R}, X'', L') = \{z\}$.

Now assume L' does not contain a vertex from $B(X', L) \cup C(X', L)$. If $C_R(\mathcal{R}, X'', L') \neq \emptyset$ then there exists a vertex $z \in L'$ that lies on a direct X''-free complete path. This path is also X'-free because $X' \subseteq X''$. As such $z \in C_R(\mathcal{R}, X', L')$, which is a contradiction, and it needs to hold that $C_R(\mathcal{R}, X'', L') = \emptyset$. If $\hat{\Theta}_R(\mathcal{R}, X'', L') \neq \emptyset$, then, just as in the proof of Lemma 4.11, there must be a vertex $z \in L'$ that lies on a direct X''-path, which means it also lies on a direct X'-path, and as such $z \in B(X', L)$, which is a contradiction. Therefore, it must hold that $\hat{\Theta}_R(\mathcal{R}, X'', L') = \emptyset$, which implies that $B_R(\mathcal{R}, X'', L') = \emptyset$. □

In Theorem 4.20 we describe the algorithm to compute the strong control closure in detail. Note that just like the algorithm in Theorem 4.12 the running time without the total update time of D is linear, and when using Tarjan's algorithm to implement D as shown in Theorem 4.8 we get a total running time of $\mathcal{O}(n^2)$.

Theorem 4.20. *Let $G = (V, E)$ be a CFG, $n = |V|$, let $V' \subseteq V$ be a vertex set such that G only contains V'-reachable vertices and let $P \subseteq V$ be the predicate vertices of G. Given a DSCC data structure D with total update time $T(n)$, there exists an algorithm that computes the strong control closure X of V' in G for P in time $\mathcal{O}(n + T(n))$.*

Proof. **Initialization:** The algorithm again initializes $X' = V'$ because V' is always contained in X, and initializes D on $G - E_G^{\text{out}}(X')$. We keep track of the SCC R_L most recently moved to $\mathcal{R}$, which, given its index i in $D.\sigma$, implicitly defines the X'-strongly restricted partition $\{\mathcal{L}, \mathcal{R}\}$ with $\mathcal{L} = \{D.\sigma(1), ..., D.\sigma(i-1)\}$ and $\mathcal{R} = \{D.\sigma(i), ..., D.\sigma(|D.\sigma|)\}$.

Over the course of the algorithm execution, we ensure that before each iteration $X' \subseteq X$ holds and that for each $L' \in \mathcal{L}$ we know the sets $\hat{\Theta}_R(\mathcal{R}, X', L')$, $B_R(\mathcal{R}, X', L')$ and $C_R(\mathcal{R}, X', L')$. Initially, because $\mathcal{R} = \emptyset$, the sets $B_R(\mathcal{R}, X', L')$ and $C_R(\mathcal{R}, X', L')$ are empty. If L' does not contain a vertex from V' then $\hat{\Theta}_R(\mathcal{R}, X', L')$ is also empty, and otherwise $\hat{\Theta}_R(\mathcal{R}, X', L') = \{x\}$ for $x \in L'$. The initialization of this algorithm can thus also be implemented to run in linear time plus the time needed for initializing D.

Iteration: In the first iteration we choose L as the last SCC in $D.\sigma$, and in subsequent iterations we query $L = D.\sigma.\textsc{Pre}(R_L)$ because then L does not reach any other SCC in $\mathcal{L}$. Again note that in the first iteration L will always be moved to $\mathcal{R}$, so querying the predecessor of R_L in the subsequent iterations is well-defined. For this SCC L, because it can only reach SCCs in $\mathcal{R}$ we know the sets $\hat{\Theta}_R(\mathcal{R}, X', L)$ and $B_R(\mathcal{R}, X', L)$, and by Equation (4.1) also the contents of the sets $\hat{\Theta}(X', L)$ and $B(X', L)$. Further, by Equation (4.2), we also know $C(X', L)$ because we already have $C_R(\mathcal{R}, X', L)$.

The algorithm then decides whether to add the vertices $B(X', L) \cup C(X', L)$ to X'. It adds those vertices to X' in the following cases:

- $|\hat{\Theta}(X', L)| \geq 2$.

- $|\hat{\Theta}(X', L)| = 1$ and $|C(X', L)| \geq 1$.
- $|\hat{\Theta}(X', L)| = 1$ and $|L| \geq 2$.
- $|\hat{\Theta}(X', L)| = 1$, $L = \{v\}$, $v \in P$, $\deg^+(v) < 2$ and $v \notin X'$.
- $|\hat{\Theta}(X', L)| = 1$, $L = \{v\}$, $(v, v) \in E$ and $v \notin X'$.

By Lemma 4.15 if one of the above conditions holds then we have $B(X', L) \cup C(X', L) \subseteq X$, which means adding those vertices to X' does not violate the condition that $X' \subseteq X$. Note that in the fourth and fifth case we also check if the vertex contained in L is already contained in X' to avoid adding the same vertex to X' multiple times. In the other cases we do not need to make this distinction, because there L cannot contain a vertex from X'. It is possible to test for every condition whether it is satisfied in constant time, because we already know the contents of $\hat{\Theta}(X', L)$ and $C(X', L)$, as well as the size of L.

Subsequently, depending on whether the algorithm added $B(X', L) \cup C(X', L)$ to X' it executes one of the following two branches:

- No vertex added to X':

  In this case, by Lemma 4.17, the partition $\{\mathcal{L}', \mathcal{R}'\}$ with $\mathcal{L}' = \mathcal{L} \backslash \{L\}$ and $\mathcal{R}' = \mathcal{R} \cup \{L\}$ is an X'-strongly restricted partition for the SCCs of $G - E_G^{\text{out}}(X')$. Note that after setting R_L to L, we have that $\mathcal{R} = \mathcal{R}'$. It remains to compute sets $\hat{\Theta}_R(\mathcal{R}', X', L')$, $B_R(\mathcal{R}', X', L')$ and $C_R(\mathcal{R}', X', L')$ for all $L' \in \mathcal{L}$ before continuing with the next iteration. We do this as follows.

  We iterate over all the edges $(u, v) \in E_G^{\text{in}}(L)$. Because no SCC in R can reach L, we know that all those edges must come from an SCC $L' \in \mathcal{L}$. If $L' \neq L$, then by Lemma 4.9, we have:

  $$\hat{\Theta}_R(\mathcal{R}', X', L') = \hat{\Theta}_R(\mathcal{R}, X', L') \cup \hat{\Theta}_R(\mathcal{R}, X', L)$$

  Note that we already know the contents of both $\hat{\Theta}_R(\mathcal{R}, X', L')$ and $\hat{\Theta}_R(\mathcal{R}, X', L)$ from the invariant. We have $\hat{\Theta}_R(\mathcal{R}, X', L) = \hat{\Theta}(X', L)$, and by Corollary 4.14 it holds that $|\hat{\Theta}(X', L)| \leq 1$, which means we can compute the union of the two sets in constant time by adding at most one element to $\hat{\Theta}_R(\mathcal{R}, X', L')$.

  Additionally, if $|\hat{\Theta}(X', L)| = 1$ we add u to $B_R(\mathcal{R}, X', L')$ because by Lemma 4.10 the set $B_R(\mathcal{R}', X', L')$ contains $B_R(\mathcal{R}, X', L')$ and all vertices from L' that have an outgoing edge to L.

  Finally, we test if one of the following conditions hold:

  - $C_R(\mathcal{R}, X', L) \neq \emptyset$.
  - $|L| \geq 2$.
  - $L = \{v\}$, v is a final vertex and $v \notin X'$.
  - $L = \{v\}$, $(v, v) \in E$ and $v \notin X'$.

  If one of them holds, we add u also to $C_R(\mathcal{R}, X', L')$, because in this case, by Lemma 4.18, the set $C_R(\mathcal{R}', X', L')$ consists of $C_R(\mathcal{R}, X', L')$ and every vertex from L' with an outgoing edge to L. We can test if one of the four conditions hold in constant time because we already know, and as such do not have to compute, $C_R(\mathcal{R}, X', L)$.

  Afterward, we can implicitly move L to $\mathcal{R}$ by setting $R_L \leftarrow L$, which results in $\mathcal{L} = \mathcal{L}'$ and $\mathcal{R} = \mathcal{R}'$. Therefore, our invariant that for each $L' \in \mathcal{L}$ we know the sets

$\hat{\Theta}_R(\mathcal{R}, X', L')$, $B_R(\mathcal{R}, X', L')$ and $C_R(\mathcal{R}, X', L')$ again holds, and because X' was unchanged it also still holds that $X' \subseteq X$.

Processing a single incoming edge (u, v) can be implemented to run in constant time, so the total time the algorithm spends in this branch is $\mathcal{O}(|E_G^{\text{in}}(L)|)$.

- $B(X', L) \cup C(X', L)$ were added to X':

  After adding $B(X', L) \cup C(X', L)$ we delete $E_G^{\text{out}}(B(X', L) \cup C(X', L))$ from D. This deletion causes L to split into new SCCs. But, the SCCs in $\mathcal{R}$ are unaffected by this edge deletion, so by Lemma 4.16 the implicitly maintained partition $\{\mathcal{L}, \mathcal{R}\}$ is again an X'-strongly restricted partition of the SCCs of $G - E_G^{\text{out}}(X')$.

  For a new SCC L' we still need to compute $\hat{\Theta}_R(\mathcal{R}, X', L')$, $B_R(\mathcal{R}, X', L')$ and $C_R(\mathcal{R}, X', L')$. By Lemma 4.19 the sets $B_R(\mathcal{R}, X', L')$ and $C_R(\mathcal{R}, X', L')$ are empty regardless of L'. Then, by the same lemma, we decide if L' is a single-vertex SCC that contains a vertex $x \in X'$, and if so we set $\hat{\Theta}_R(\mathcal{R}, X', L') = \{x\}$, otherwise we have $\hat{\Theta}_R(\mathcal{R}, X', L') = \emptyset$. This branch can be implemented to run in time $\mathcal{O}(|B(X', L) \cup C(X', L)|)$, disregarding the update time of D.

Running Time: The first branch can only be executed at most n times because every SCC can only be moved once from $\mathcal{L}$ to $\mathcal{R}$. The second branch can also only be executed at most n times, because in each iteration a vertex is added to X', and we do not add vertices that are already contained in X' twice. Therefore, the algorithm terminates after at most $2n$ iterations. In the first branch, the algorithm spends time $\mathcal{O}(|E_G^{\text{in}}(L)|)$, and because every SCC is only moved once from $\mathcal{L}$ to $\mathcal{R}$, the total time spent in this branch is $\mathcal{O}(n)$. Without considering the update time of D, the second branch takes time $\mathcal{O}(|B(X', L) \cup C(X', L)|)$. Consequently, because every vertex in $B(X', L) \cup C(X', L)$ is only added once to X', and X' can contain at most n vertices, the total time spent in this branch is also $\mathcal{O}(n)$. The total running time of the algorithm is then $\mathcal{O}(n)$ plus the total update time $T(n)$ of D.

Correctness: At the end of the algorithm it holds that $X' \subseteq X$ and $\mathcal{L} = \emptyset$. Therefore, $V = \bigcup \mathcal{R}$, which means every vertex is X'-strongly committing or X'-avoiding. Thus, and because of $V' \subseteq X'$, it holds that X' is a strongly control closed superset of V'. Finally, because X is the minimal strongly control closed superset of V' and $X' \subseteq X$, we have $X' = X$. □

Finally note that this algorithm, just like the algorithm described in Theorem 4.12 has a linear running time on acyclic graphs when using D as described in Theorem 4.8. In general, as long as the implementation of D only spends linear time in the initialization, both algorithms to compute the weak and strong control closure have linear running time on acyclic CFGs.

4.4 Pseudocode for Computing the Strong Control Closure

We also provide pseudocode for the algorithm described in Theorem 4.20 in Algorithm 4.2. For an SCC L we additionally maintain $C(L)$ which stores $C_R(\mathcal{R}, X, L)$ for the by D implicitly maintained X-strongly restricted partition $\{\mathcal{L}, \mathcal{R}\}$. We stated in Theorem 4.20 that in the initialization for an SCC $L' = \{v'\}$ with $v' \in V'$ we must set $\Theta(L') = \{v'\}$ and after deleting edges from D for a newly created SCC $L'' = \{x\}$ with $x \in X$ we need to set $\Theta(L'') = \{x\}$. Just as in Algorithm 4.1 we postponed this computation to just before propagating X-path information to make the implementation simpler. Also note that in lines 11 and 29 we only test if there exists a self loop in H because then we do not need to test if z is contained in X as vertices from X do not have outgoing edges in H.

Algorithm 4.2: StrongControlClosure($G,\ P,\ V'$)

Input : A CFG $G = (V, E)$, the predicate vertices $P \subseteq V$ of G and a vertex set $V' \subseteq V$.

Output : The strong control closure $X \subseteq V$ of V' in G for P.

```
1  H ← RemoveUnreachable(G, V');
2  H ← RemoveOutgoingEdges(H, V');
3  D ← InitializeDSCC(H);
4  X ← V';
5  R_L ← null;
6  while (L ← D.σ.Pre(R_L)) ≠ null do
7      if |Θ(L)| ≥ 2 or
8           [|Θ(L)| = 1 and |C(L)| ≥ 1] or
9           [|Θ(L)| = 1 and |L| ≥ 2] or
10          [|Θ(L)| = 1 and L = {z} for z ∉ X and z is a final Vertex ] or
11          [|Θ(L)| = 1 and L = {z} and (z, z) ∈ E(H)]
12      then
13          X ← X ∪ B(L) ∪ C(L);
14          D.Delete(B(L) ∪ C(L));
15      else
16          if L = {x} for x ∈ X then
17              Θ(L) ← {x};
18          if |Θ(L)| ≥ 1 then
19              for v ∈ L do
20                  for (u, v) ∈ E(H) do
21                      X_u ← D.SCC(u);
22                      X_v ← D.SCC(v);
23                      if X_u ≠ X_v then
24                          B(X_u) ← B(X_u) ∪ {u};
25                          Θ(X_u) ← Θ(X_u) ∪ Θ(X_v);
26          if  |C(L)| ≥ 1 or
27               |L| ≥ 2 or
28               [L = {z} for z ∉ X and z is a final Vertex ] or
29               [L = {z} and (z, z) ∈ E(H)]
30          then
31              for v ∈ L do
32                  for (u, v) ∈ E(H) do
33                      X_u ← D.SCC(u);
34                      X_v ← D.SCC(v);
35                      if X_u ≠ X_v then
36                          C(X_u) ← C(X_u) ∪ {u};
37          R_L ← L;
38 return X;
```

5. Decremental Strongly Connected Components

Up until now we used Tarjan's algorithm to implement the DSCC data structure D with worst case total update time $\mathcal{O}(n^2)$. This resulted in our algorithms that compute the weak and strong control closure to also have a worst case running time of $\mathcal{O}(n^2)$, even though they have a linear running time if we neglect the time spent on maintaining D. As such, we can improve the running time of the two algorithms by using a different DSCC data structure. We will take a closer look at the data structure described by Bernstein, Gutenberg, and Wulff-Nilsen [BGW23], which allows querying whether two vertices are in the same SCC and has an expected total update time of $\mathcal{O}(m \log(n)^4)$. For a CFG, where $m = \mathcal{O}(n)$, the expected total update time is $\mathcal{O}(n \log(n)^4)$. In the following, we will modify this data structure to implement the interface that we defined in Definition 4.7, which specifically means that we maintain a topological ordering of the SCCs. Additionally, we show that our modifications do not worsen the expected total update time.

5.1 Overview of the Data Structure

First, we will give an overview of how the data structure works in general. In Chapter 4 we briefly described how we can construct an acyclic graph $\hat{G}$ by merging the SCCs of a graph G and removing self loops to illustrate how to order SCCs topologically. Here those graphs will play a more prominent role, but we also allow them to have self loops.

Definition 5.1. *For a graph $G = (V, E)$ the* condensation graph $\hat{G}$ *is a graph that contains a vertex for every SCC of G and has edge set $\hat{E}$, which maps the endpoints of every edge in E to the vertices in $\hat{G}$ that represent the SCC containing the endpoint.*

Further, we introduce the operations Condensation(G) and Condensation(V, E) to compute the condensation graph of $G = (V, E)$. In general, we denote the condensation graph of a graph G with $\hat{G}$. Additionally, we call the vertices of a condensation graph $\hat{G}$ *nodes* and also denote them with upper case letters to distinguish them from the vertices in G. For example, $v \in V$ is a vertex and $Y \in V(\hat{G})$ is a node. By storing the vertices of the SCC represented by a certain node we can use nodes and SCCs interchangeably. If v is contained in the SCC represented by node Y, it holds that $v \in Y$. For a node set $\mathcal{X} = \{X_1, ..., X_k\}$ we use the shorthand notation $\bigcup \mathcal{X}$ instead of writing $\bigcup_{i=1}^{k} X_i$. Given an

edge $e = (u, v) \in E$ we define the *edge-equivalent* of e as $\hat{e} = (X_u, X_v)$ where X_u and X_v are the nodes in $\hat{G}$ that contain u and v respectively.

The main idea behind the data structure is to maintain a graph hierarchy with $\alpha + 1$ levels of condensation graphs $\hat{G}_0$ to $\hat{G}_\alpha$. Each $\hat{G}_i$ condenses graph G with only a subset of its original edges. This subset is not arbitrary, it is monotonically increasing when moving up the graph hierarchy. On the lowest level we want to have that $\hat{G}_0$ condenses $G - E$, which means $\hat{G}_0$ contains a node for every vertex in G, and on the highest level the condensation graph $\hat{G}_\alpha$ should condense G, and therefore contains a node for every SCC of G. We call a graph hierarchy *based on* G if every condensation graph in the hierarchy contains an edge-equivalent for every edge in E and no additional edges. By maintaining this hierarchy efficiently we then also maintain the SCCs of G efficiently. Formally, we have a partition $E_0, ..., E_\alpha$ of E, and each graph $\hat{G}_i$ condenses graph $G_i = (V, \bigcup_{j<i} E_j)$. Note that $E_\alpha = \emptyset$ must hold because otherwise $\hat{G}_\alpha$ does not condense G with all its edges.

An important part for the efficiency of the data structure is that for each $i \in \{1, ..., \alpha\}$, the graph $\hat{G}_i$ is computed using $\hat{G}_{i-1}$. Because $\bigcup_{j<i} E_j$ is monotonically increasing, if G_{i-1} contains an SCC X then G_i must contain an SCC Y such that $X \subseteq Y$. Consequently, if $\hat{G}_{i-1}$ contains the node X then $\hat{G}_i$ must contain a node $Y \supseteq X$, and we do not need to spend time processing the subgraph $G[X]$ because we already know that X is strongly connected. We can also completely compute $\hat{G}_i$ using $\hat{G}_{i-1}$ if we add the edge set $\hat{E}_{i-1}$, which contains an edge-equivalent (X_u, X_v) for every edge $(u, v) \in E_{i-1}$, to $\hat{G}_{i-1}$. Then we can compute the node Y of $\hat{G}_i$ by determining the SCC of $\hat{G}_{i-1}$ that contains node X. Mathematically, a condensation graph on level $i \in \{0, ..., \alpha\}$ in the hierarchy is now defined as:

$$\hat{G}_i = \textsc{Condensation}(V, \bigcup_{j<i} E_j) + \hat{E}_i$$

We want to highlight that $\hat{G}_i$ is not the condensation of $\hat{G}_{i-1}$. The SCCs of $\hat{G}_{i-1}$ are sets of nodes, if we would condense $\hat{G}_{i-1}$ we would get a meta-node that contains multiple nodes. This would result in some kind of node nesting, but the nodes of every condensation graph are always just vertex sets. Formally, for an SCC $\mathcal{X}$ that contains node X in $\hat{G}_{i-1}$ we construct node Y by taking the union of all the nodes contained in $\mathcal{X}$, meaning $Y = \bigcup \mathcal{X}$.

This concept might be better understood with an example. Figure 5.1 shows a graph hierarchy for a strongly connected graph G with three vertices a, b and c. Because the graph is strongly connected it has a single SCC containing all three vertices. As described above, the first condensation graph $\hat{G}_0$ contains only single-vertex nodes and the last graph in the hierarchy $\hat{G}_2$ contains a node for every SCC of G. Graph G restricted to edges $E_0 = \{(a, b),\ (b, a)\}$ contains the SCC $\{a,\ b\}$, which is represented by the node $\{a, b\}$ in $\hat{G}_1$. Note that $\hat{G}_0$ also contains an SCC, because the condensation graph contains the edge set $\hat{E}_0$, but the SCC $\{\{a\}, \{b\}\}$ is now a set of nodes. The data structure would compute this SCC, and then construct node $\{a, b\}$ in $\hat{G}_1$. The graph $\hat{G}_1$ itself contains another SCC which is made up of nodes $\{a, b\}$ and $\{c\}$, and by computing this SCC the node $\{a, b, c\}$ in $\hat{G}_2$ is constructed.

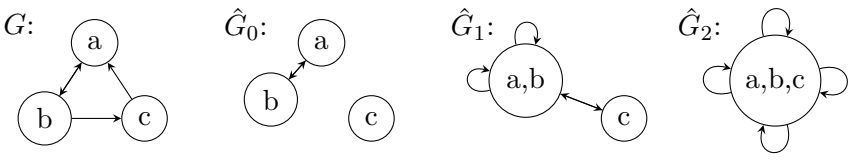

Figure 5.1: A graph hierarchy for the partition $E_0 = \{(a, b),\ (b, a)\}$, $E_1 = \{(b, c),\ (c, a)\}$ and $E_2 = \emptyset$ on a graph G with edges $E = \{(a, b),\ (b, a),\ (b, c),\ (c, a)\}$.

Using the Graph Hierarchy to maintain SCCs: While it is nice to be able to compute the condensation graph hierarchy iteratively, it clearly introduces additional computation overhead compared to just computing the condensation graph of G. But when deleting an edge from G the advantage of this approach becomes clear. With Tarjan's algorithm, when deleting an edge in an SCC Y, we need to recompute the SCCs of $G[Y]$, which in some cases, for example in the graph shown in Figure 4.6, means processing a large path of the graph again and again. In comparison, this algorithm only needs to process a few nodes to maintain the hierarchy after deleting an edge. The first observation to make is that if we delete an edge $e \in E_i$ then the graphs $\hat{G}_0$ up to $\hat{G}_{i-1}$ are unaffected by the deletion because those graphs condense the graphs $G_0, ..., G_{i-1}$, which do not contain E_i. Additionally, in $\hat{G}_i$ the nodes are unaffected because $\hat{G}_i$ condenses $G_i = (V, \bigcup_{j<i} E_j)$, which also does not contain E_i. But the edges of $\hat{G}_i$ are affected because $\hat{G}_i$ has edges $\hat{E}_i$, which contains the edge-equivalent $\hat{e}$ of e, and by deleting e from G we must also delete $\hat{e}$ from $\hat{G}_i$. Further, if the deletion of edge $\hat{e}$ in $\hat{G}_i$ causes an SCC $\mathcal{Y}$ of $\hat{G}_i$ to no longer be strongly connected then this also means that a node $Z \in V(\hat{G}_{i+1})$ with $Z = \bigcup \mathcal{Y}$ is no longer correctly representing an SCC of G_{i+1}. Consequently, we need to split the node such that the new nodes represent the new SCCs of G_{i+1}.

Fortunately, Z cannot split into arbitrary many new SCCs. We know that the nodes in $\hat{G}_i$ still represent SCCs of G_i, meaning Z can at most split into the nodes contained in $\mathcal{Y}$. So instead of recomputing the SCCs on $G_{i+1}[Z]$, we only need to determine the SCCs of $\hat{G}_i[\mathcal{Y}]$ and replace Z with corresponding nodes, which is often faster because we do not need to process large parts of the graphs for which we already know that they are strongly connected. After fixing $\hat{G}_{i+1}$ we continue with $\hat{G}_{i+2}, ..., \hat{G}_\alpha$ because they may also contain nodes affected by the edge deletion. How to implement this follows in detail in Section 5.4.

Choosing the Edge Partition: Up until now we only have one restriction for the edge partition, which is that $E_\alpha = \emptyset$ must hold. We can trivially satisfy this by setting $E_0 = E$ and $E_i = \emptyset$ for every level $i \in \{1, ..., \alpha\}$ in the hierarchy as this would be a valid partition of the edge set. But then $\hat{G}_1$ would immediately be the condensation of G, and we would not get any advantage from using the hierarchy because we do not have any intermediary condensation graphs. This would mean that after deleting an edge contained in a large SCC we need to recompute the new SCCs of a large part of the graph. What we want is that large SCCs are contracted not immediately but over multiple levels such that some strongly connected subgraphs of the large SCC are represented by nodes.

To achieve this we can limit the diameter of the SCCs a condensation graph $\hat{G}_i$ is allowed to have. This, especially if the number of outgoing edges of a vertex is low, which is the case in CFGs, means that the number of nodes in an SCC is not too large, and if an SCC splits after an edge deletion we can find the new SCCs quickly. Unfortunately, it is not always possible to partition the edges of a graph such that all the condensation graphs in the hierarchy have SCCs with diameter of at most $\delta \in \mathbb{N}_+$. One simple example is the directed cycle-graph C_5, where all the edges run in a clockwise-direction with threshold $\delta = 3$. This graph has diameter 4 and no subset of edges induces an SCC with smaller diameter. This means regardless of how we choose the edge partition there must be a condensation graph containing an SCC with diameter $4 > \delta$. Because of this, Bernstein, Gutenberg, and Wulff-Nilsen use S-distances, which we defined in Definition 2.32. Using $S = \emptyset$, every pair of vertices in C_5 has S-distance 0 and the S-diameter of C_5 would also be 0. Therefore, we can always choose a set S such that all condensation graph in the hierarchy have S-diameter smaller than the threshold δ.

But because setting $S = \emptyset$ results in any SCC having S-diameter 0 we now have the additional problem of finding appropriate sets $S_i \subseteq V(\hat{G}_i)$ for every level $i \in \{0, ...\alpha\}$ in the hierarchy. At first, introducing another complexity might be counterintuitive, but

we can use these sets to define the edge partition such that the SCCs of $\hat{G}_i$ have small S_i-diameter. A set S_i is called a *separator*, and in Section 5.2 we will see that this naming is appropriate, because those sets are chosen to separate parts of the graph. Further, we impose the restriction that the separators are monotonically decreasing and also form a hierarchy. Formally this means that $S_0 \supseteq S_1 \supseteq ... \supseteq S_\alpha$. We call this a *separator hierarchy*. Note that this is well-defined even though the sets $S_0, ..., S_\alpha$ are node-sets of different condensation graphs because every node in the hierarchy is just a vertex set, and as such, every separator is just a set of vertex sets.

Because of the monotonicity of the separator hierarchy it makes sense to choose S_0 as large as possible, meaning it contains all the nodes of $\hat{G}_0$. Recall that these are all single-vertex nodes, which consequently means that every node in the separator hierarchy is a single-vertex set. Because the separators are monotonically decreasing they may lose nodes from one level to the next. Based on this we define for an $i \in \{0, ..., \alpha\}$ the set E_i as all the edges incident to vertices in S_i, but not incident to vertices in S_{i+1}, i.e. $E_i = E_G\left(\bigcup S_i\right) \setminus E_G\left(\bigcup S_{i+1}\right)$. Note that for E_α to be well-defined we introduce a set $S_{\alpha+1} = \emptyset$. Because it must hold that $E_\alpha = \emptyset$ we must also have that $S_\alpha = \emptyset$. Inserting this new definition of E_i into the formula for $\hat{G}_i$, we get:

$$\hat{G}_i = \textsc{Condensation}\left(G - E_G\left(\bigcup S_i\right)\right) + \hat{E}_G\left(\bigcup S_i\right) \setminus \hat{E}_G\left(\bigcup S_{i+1}\right)$$

To better illustrate how the edge partition is defined by the separator hierarchy consider again the example graph hierarchy in Figure 5.1. By choosing $S_0 = \{\{a\}, \{b\}, \{c\}\}$, $S_1 = \{\{c\}\}$ and $S_2 = S_3 = \emptyset$ we get exactly the hierarchy of condensation graphs shown in Figure 5.1.

Implicit Edge Storage: We want to mention that despite having described how the separators partition the edges of G, the data structure does not explicitly maintain this edge partition, it is only implicitly maintained by using the separators and S-distances. What this means is that in our graph hierarchy, every condensation graph $\hat{G}_i$ will contain for every edge $(u, v) \in E$ an edge between the nodes that contain u and v respectively. This solves the problem that currently we cannot fully construct $\hat{G}_i$ without knowing S_{i+1}, because $\hat{G}_i$ contains the edges $\hat{E}_G\left(\bigcup S_i\right) \setminus \hat{E}_G\left(\bigcup S_{i+1}\right)$.

The main task of the data structure is to maintain a separator hierarchy such that for every level $i \in \{0, ..., \alpha\}$ the S_i-diamater of the SCCs of $\hat{G}_i$ are small. We also do not want too many levels in the hierarchy because then just iterating over the hierarchy would dominate the total update time. Bernstein, Gutenberg, and Wulff-Nilsen used a hierarchy with only logarithmically many levels by ensuring that a separator S_{i+1} only contains at most half of the nodes that S_i contains. We define so-called *good separators*.

Definition 5.2. *A separator S_{split} is called* good *for a graph $G = (V, E)$, a set $S \subseteq V$ and threshold $d \in \mathbb{N}_+$, if all the following hold:*

- $S_{\text{split}} \subset S$.
- *Every SCC of $G - E_G(S_{\text{split}})$ has S-diameter at most d.*
- $|S_{\text{split}}| \leq \frac{1}{2}|S|$.

Given a good separator S_{i+1} for a graph $\hat{G}_i$ and separator S_i, the second condition of Definition 5.2 implies that the SCCs in $\hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ have small S_i-diameter. Observe that $\hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ only contains edges equivalent to edges in G that are not incident to vertices in S_{i+1}, which aligns with our above definition of $\hat{G}_i$. Therefore, by computing good

separators for every level in the hierarchy, we can implicitly maintain the edge partition such that the SCCs in the condensation graphs have small diameter.

We also mentioned that every node $X \in V(\hat{G}_{i+1})$ should represent an SCC of the condensation graph in the level below. With the implicit edge partition we can formalize this as there being an SCC $\mathcal{X} \in \hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ such that $X = \bigcup \mathcal{X}$. To conclude this section, we have collected all the conditions that need to hold for the separator and graph hierarchy in Definition 5.3.

Definition 5.3. *Let $G = (V, E)$ be a graph. A separator hierarchy $S_0 \supseteq S_1 \supseteq ... \supseteq S_\alpha$ is called* good *for a graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G and a threshold $\delta \in \mathbb{N}_+$ if all the following hold:*

- $S_0 = \{\{v\} \mid v \in V\}$, $S_\alpha = \emptyset$
- *For all $i \in \{1, ..., \alpha\}$, the separator S_i is a good separator for S_{i-1}, $\hat{G}_{i-1}$ and 2δ.*
- *For all $i \in \{1, ..., \alpha\}$ and for every node $X \in V(\hat{G}_i)$ there must exist an SCC $\mathcal{X} \in \hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_i)$ such that $X = \bigcup \mathcal{X}$.*

Observe that the second condition only requires that the SCCs of a condensation graph $\hat{G}_i$ in the hierarchy all have S_i-diameter at most 2δ. We will see in Section 5.3 that we compute the separator hierarchy such that all SCCs in $\hat{G}_i$ have S_i-diameter at most δ, and to avoid immediately splitting an SCC because its S_i-diamater increases by one after a single edge deletion we allow a certain buffer. Also note that for a good separator hierarchy we have for every $i \in \{0, ..., \alpha - 1\}$ that $S_{i+1} \subseteq S_i$, by the first condition of Definition 5.2. Consequently, for $\hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ to be well-defined, we must have that $S_i \subseteq V(\hat{G}_i)$.

5.2 Computing Good Separators

In this section we will cover how to compute a good separator S_{split} for a graph $G = (V, E)$, a set $S \subseteq V$ and a threshold $d \in \mathbb{N}_+$. The idea is to compute S_{split} iteratively by ensuring that for $v \in V$ the separator S_{split} is a *good separator for v and d*, which is defined as follows.

Definition 5.4. *For a graph $G = (V, E)$, set $S \subseteq V$, a threshold $d \in \mathbb{N}_+$ and a vertex $v \in V$, a separator S_{split} is called* good for v *if all the following hold:*

- $S_{\text{split}} \subset S$.
- *The diameter of the SCC that contains v in $G - E_G(S_{\text{split}})$ has S-diameter at most d.*

Clearly, if we ensure that S_{split} is a good separator for every vertex in V, and additionally ensure that $|S_{\text{split}}| \leq \frac{1}{2}|S|$, it holds that S_{split} is a good separator.

Computing a set S_{split} that is a good separator for every vertex in V is simple. Initially, S_{split} is empty, then we pick an arbitrary root vertex $r \in V$. Now we need to determine if S_{split} is a good separator for r and d. To do this we compute all the vertices $v \in V$ with $\text{dist}_G(r, v, S) > \frac{d}{2}$ or $\text{dist}_G(v, r, S) > \frac{d}{2}$. If there is no such vertex, then the S-diameter of the SCC that contains r in $G - E_G(S_{\text{split}})$ is at most d because for each pair of vertices x and y, the path from x to r to y has length at most d. On the other hand, if there is such a vertex then the diameter may exceed d, and we precautiously separate r from v as follows.

If $\text{dist}_G(r, v, S) > \frac{d}{2}$ holds, which means that the path from r to v is too long, we compute vertices S_{sep} such that removing their incident edges ensures that there is no more path

from r to v. These vertices are added to S_{Split}, which splits $G - E_G(S_{\text{split}})$ into two parts, one that contains r and one that contains v. The part of the graph that contains v must be processed separately because it may still contain SCCs that have a large diameter, but for ensuring that S_{split} is a good separator for r it can be neglected and is therefore pruned away. Analogously, if $\text{dist}_G(v, r, S) > \frac{d}{2}$ holds, which means that the path from v to r is too long, the procedure computes vertices S_{sep} for which removing their incident edges ensures that there is no more path from v to r.

If we iteratively execute this for every vertex v for which the S-distance from or to r exceeds $\frac{d}{2}$ then the only part of the graph that r can reach in $G - E_G(S_{\text{split}})$ has an S-diameter of at most d. Afterward, S_{split} is a good separator for every vertex contained in the SCC of $G - E_G(S_{\text{split}})$ that contains r, which means we do not further process those vertices. What remains is to process the parts of the graph that were pruned away and contain a vertex v for which the S-distance to or from r was too large. For such a subgraph G' we do this by recursively computing a good separator S'_{split} for every vertex in $V(G')$, and then adding S'_{split} to S_{split}.

The result of the described procedure is a set S_{split} that is a good separator for every vertex $v \in V$, but it does not necessarily satisfy the size constraint of Definition 5.2. Finding a set S_{split} that it is also small is much more involved. But first we will illustrate how to compute vertices to separate r and v.

5.2.1 Out-Separator

For two vertices u and v with $\text{dist}_G(u, v, S) > \frac{d}{2}$ we need to find vertices $S_{\text{out}} \subseteq S$ such that in $G - E_G(S_{\text{out}})$ there is no more path from u to v. Additionally, because there could be multiple vertices that are too far away from u, we want that for every vertex z that is still reachable from u in $G - E_G(S_{\text{out}})$ it holds that $\text{dist}_G(u, z, S) \leq \frac{d}{2}$, as otherwise we would just need to unnecessarily repeat computing such sets S_{out}. Further, for S_{split} to be a good separator in the end the set S_{out} must not be too large. To achieve this, Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 7] described a procedure OutSep. In the following, we will illustrate this procedure, but less procedurally, and more declaratively to better apply it later.

To find such a set S_{out} we can compute layers of vertices similar to a 0/1-BFS. Each layer L_i contains the vertices with S-distance i from u. Observe that the vertices in one layer L_i separate u from the next layer L_{i+1} and every layer after that. Further, not all vertices contained in L_i are necessary to separate u from L_{i+1}, the vertices $L_i \cap S$ are enough because only vertices contained in S can increase the S-distance. Therefore, setting $S_{\text{out}} = L_i \cap S$ for $0 \leq i \leq \frac{d}{2}$ ensures that u cannot reach any vertex z with $\text{dist}_G(u, z, S) > \frac{d}{2}$. We could just choose $S_{\text{out}} = L_0 \cap S$, but Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Lemma 7.1] impose an additional restriction on the size of S_{out}, which is necessary if we want S_{split} to be a good separator. Namely, S_{out} should contain fewer vertices from S than the smaller side of the cut multiplied by a certain factor. We can formalize these conditions in Definition 5.5.

Definition 5.5 (Out-Separator)**.** *Let $G = (V, E)$ be a graph, $u \in V$, let $S \subseteq V$ be a vertex set and $\tilde{d} \in \mathbb{N}_+$ a threshold. Further, let $S_{\text{out}} \subseteq S$, and let $V_{\text{out}} \subseteq V$ be all the vertices that are reachable from u in $G - E_G(S_{\text{out}})$. The set S_{out} is called* out-separator *for u and $\tilde{d}$ if the following hold:*

- *For all $v \in S_{\text{out}} \cup V_{\text{out}}$ it holds that $\text{dist}_G(u, v, S) \leq \tilde{d}$.*
- $|S_{\text{out}}| \leq \min\{|V_{\text{out}} \cap S|, |\,[V \backslash (S_{\text{out}} \cup V_{\text{out}})] \cap S|\} \cdot \frac{2 \ln |V|}{\tilde{d}}$

- *For any $x \in V_{\text{out}}$ and $y \in V \setminus (S_{\text{out}} \cup V_{\text{out}})$ there exists no path from x to y in $G - E_G(S_{\text{out}})$.*

Lemma 5.6. *[Lemma 7.1 [BGW23]] Given a graph $G = (V, E)$, a vertex $u \in V$, a vertex set $S \subseteq V$ and a threshold $\tilde{d} \geq 4\ln(|V|)$, it is possible to compute an out-separator $S_{\text{out}} \subseteq S$ for u and $\tilde{d}$ and a set $V_{\text{out}} \subseteq V$ that contains all the vertices that are reachable by u in $G - E_G(S_{\text{out}})$ in time $\mathcal{O}(|E_G(V_{\text{out}})|)$.*

Proof. The layers can be computed similar to a 0/1-BFS by using a queue Q and marking vertices that were already visited. We start by marking u and adding it to L_0, and if $u \notin S$ we also add it to Q, otherwise we let Q be empty.

To compute layer L_i we take as long as the queue is not empty a vertex x from Q and then for each edge $(x, y) \in E$, if y is not yet marked, we add y to L_i and mark it. If $y \notin S$ we also add it to Q because then all the vertices to which y has an outgoing edge are also contained in L_i. When Q is empty exactly the vertices reachable from u with S-distance i are contained in L_i, and we test the inequality:

$$|L_i \cap S| \leq \min\{\sum_{j<i} |L_j \cap S|, |(V \setminus \bigcup_{j<i} L_j) \cap S|\} \cdot \frac{2\ln(|V|)}{\tilde{d}}$$

If the inequality holds, we return the tuple:

$$\left(S_{\text{out}} = L_i \cap S,\ V_{\text{out}} = \bigcup_{j \leq i} L_j \setminus S_{\text{out}} \right)$$

Before entering the iteration to compute L_{i+1} we add all the vertices $L_i \cap S$ to Q as starting vertices for the next layer.

Clearly, V_{out} is the set of vertices that u can reach in $G - E_G(S_{\text{out}})$, excluding S_{out}. The size constraint of Definition 5.5 also clearly holds for S_{out}. It remains to show that the S-distance from u to any vertex in $S_{\text{out}} \cup V_{\text{out}}$ does not exceed $\tilde{d}$. For this proof we refer to Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Lemma 7.1] where they have shown that there always exists a layer L_i with $i \leq \tilde{d}$ that satisfies the equality above. It is easy to see that the procedure only takes time $\mathcal{O}(|E_G(V_{\text{out}})|)$ because after finding a layer that satisfies the inequality no more edges are processed and the edges that were processed are at most the outgoing edges of V_{out}. □

Note that in Lemma 5.6 we have the additional requirement that $\tilde{d} \geq 4\ln(|V|)$, which is not present in the lemma given by Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Lemma 7.1]. If $\tilde{d}$ is a constant independent of $|V|$ it is possible to construct a graph where the lemma does not hold. We pointed this, and the part of the proof that was incorrect, out to Bernstein, Gutenberg, and Wulff-Nilsen, and they responded that with this restricted on $\tilde{d}$ the lemma should be correct again.

5.2.2 In-Separator

Analogously for two vertices u and v with $\text{dist}_G(v, u, S) > \frac{d}{2}$ we need to find vertices $S_{\text{in}} \subseteq S$ such that in $G - E_G(S_{\text{in}})$ there is no more path from v to u. Bernstein, Gutenberg, and Wulff-Nilsen did not explicitly describe an InSep procedure to compute S_{in}, they only state that analogously to OutSep there exists an InSep procedure.

But if we use the same approach of computing layers of vertices at certain S-distances we run into a problem. The flaw arises from how S-distances are measured. The S-distance

between two vertices u and v is by Definition 2.32 the minimum number of vertices from $S \backslash \{v\}$ on any path from u to v. This is equivalent to measuring distances on a weighted graph where outgoing edges from vertices in S have weight 1. If we have two layers L_i^{out} and L_{i+1}^{out}, which are the layers containing the vertices with S-distance i and $i+1$ from u respectively, then the boundary vertices of L_i^{out}, which are the vertices that have edges to the next layer, are vertices of S because as soon as we pass such a vertex, we increase the S-distance. Now we take a look at L_i^{in} and L_{i+1}^{in}. Layer L_i^{in} contains exactly the vertices that have S-distance i to u. Here, the boundary vertices of L_i^{in} to L_{i+1}^{in} can also be vertices that are not contained in S. By computing V_{in} analogously to V_{out} as in Lemma 5.6, meaning we compute the intersection of a layer and S, some vertices in contained in V_{in} may not be able to reach u in $G - E_G(S_{\text{in}})$.

To better illustrate this problem, consider the graph G shown in Figure 5.2 where the vertices $S = \{v_1,\ v_2,\ v_4\}$ are highlighted. For root vertex $u = v_1$ we get $L_0^{\text{in}} = \{v_1\},\ L_1^{\text{in}} = \{v_2, v_3\}$ and $L_2^{\text{in}} = \{v_4\}$. The procedure as described by Bernstein, Gutenberg, and Wulff-Nilsen would stop at layer L_1^{in} because it satisfies the inequality from above. Then we would have:

$$S_{\text{in}} = \{v_2\} \text{ and } V_{\text{in}} = (L_0^{\text{in}} \cup L_1^{\text{in}}) \backslash S_{\text{in}} = \{v_1,\ v_3\}$$

But this is not correct because now $v_3 \in V_{\text{in}}$ cannot reach v_1 in $G - E_G(S_{\text{in}})$, which means the S-distance from v_3 to v_1 is ∞ and exceeds threshold $\tilde{d}$. Additionally, V_{in} is not separated from $V \backslash (S_{\text{in}} \cup V_{\text{in}})$ in $G - E_G(S_{\text{in}})$.

Figure 5.2: A graph with highlighted vertices $S = \{v_1,\ v_2,\ v_4\}$ that are used to measure S-distances.

To solve this problem we must ensure that the vertices of S are always located at the boundary of a layer. To do this, we assign the incoming edges of vertices in S weight 1, instead of the outgoing edges. Applying this to the graph shown in Figure 5.2, we get the layers $L_0 = \{v_0\}$, $L_1 = \{v_1\}$ and $L_3 = \{v_2, v_3\}$. Now v_2 is no longer contained in V_{in} if we choose $S_{\text{in}} = L_1 \cap S$. Additionally, because in this case we have $v_0 \in S$, every layer i contains exactly the vertices that have S-distance i to v_0. This holds in general for any root $u \in S$. Unfortunately, if u is not contained in S the layers do not align with S-distances anymore. It may happen that the S-distance of a vertex $v \in L_i$ to u is actually $i+1$ because S-distances are measured along outgoing, not incoming edges. As such, we can only formulate a slightly weaker variant of Definition 5.5.

Definition 5.7 (In-Separator)**.** *Let $G = (V, E)$ be a graph, $u \in V$, let $S \subseteq V$ be a vertex set and $\tilde{d} \in \mathbb{N}_+$ a threshold. Further, let $S_{\text{in}} \subseteq S$ and let $V_{\text{in}} \subseteq V$ be all the vertices that are reachable from u in $G - E_G(S_{\text{in}})$ along incoming edges. The set S_{in} is called* in-separator *for u and $\tilde{d}$ if the following hold:*

- *For all $v \in S_{\text{in}} \cup V_{\text{in}}$ it holds that $\text{dist}_G(v, u, S) \leq \tilde{d} + 1$.*
- $|S_{\text{in}}| \leq \min\{|V_{\text{in}} \cap S|, |\,[V \backslash (S_{\text{in}} \cup V_{\text{in}})] \cap S|\} \cdot \frac{2 \ln |V|}{\tilde{d}}$
- *For any $x \in V_{\text{in}}$ and $y \in V \backslash (S_{\text{in}} \cup V_{\text{in}})$ there exists no path from y to x in $G - E_G(S_{\text{in}})$.*

Then, by analogously implementing the procedure described in the proof of Lemma 5.6, we get that Lemma 5.8 holds.

Lemma 5.8. *Given a graph $G = (V, E)$, a vertex $u \in V$, a vertex set $S \subseteq V$ and a threshold $\tilde{d} \geq 4\ln(|V|)$, it is possible to compute an in-separator $S_{\text{in}} \subseteq S$ for u and $\tilde{d}$ and a set $V_{\text{in}} \subseteq V$ that contains all the vertices that can reach u in $G - E_G(S_{\text{in}})$ in time $\mathcal{O}(|E_G(V_{\text{in}})|)$.*

Note that because of how the algorithm determines the threshold $\tilde{d}$ the off-by-one error in Definition 5.7 does not affect the correctness of the data structure. More details follow in the next section where we apply the lemmas.

5.2.3 Satisfying the Size Requirement

Now we have all the tools needed to compute a good separator for G, S and threshold d. We again pick an arbitrary root vertex $r \in V$. But now, we do not immediately compute the vertices for which the S-distance to or from r exceeds $\frac{d}{2}$. It may be the case that there are many such vertices because r is only strongly connected to a few vertices without exceeding the distance threshold. In this case we would need to prune away a large subgraph, which then needs to be recursively processed. Instead, if we know that r is only strongly connected to few vertices it is more efficient to prune away a small subgraph that contains r, and process this subgraph recursively. To process the rest of the graph we then pick a new arbitrary root vertex r.

To determine if there exists a small subgraph that can be separated with few vertices from the rest of the graph we use Lemma 5.6 and Lemma 5.8 for $u = r$ and distance threshold $\tilde{d} = \frac{d}{16}$ to compute the out-separator S_{out} and in-separator S_{in} as well as V_{out} and V_{in}, which are the vertices still reachable by r in $G - E_G(S_{\text{out}})$ along outgoing edges and in $G - E_G(S_{\text{in}})$ along incoming edges respectively. If either V_{out} or V_{in} contains at most $\frac{2}{3}|V|$ vertices then it is advantageous to prune away $G[V_{\text{out}}]$ or $G[V_{\text{in}}]$ respectively, recursively process that subgraph and then pick a new root vertex r from the remaining graph. On the other hand, if this is not the case then Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Claim 7.1] have shown that r is strongly connected to at least $\frac{1}{3}|V|$ vertices with small S-distance to and from each of those vertices.

Lemma 5.9 (Claim 7.1 [BGW23])**.** *Let $G = (V, E)$ be a graph, $r \in V$, $d \in \mathbb{R}_+$, $S \subseteq V$ a vertex set, $S_{\text{out}} \subseteq S$ an out-separator for r and $\frac{d}{16}$, and $S_{\text{in}} \subseteq S$ an in-separator for r and $\frac{d}{16}$. Let V_{out} be the vertices still reachable by r in $G - E_G(S_{\text{out}})$ and V_{in} be the vertices that can still reach r in $G - E_G(S_{\text{in}})$.*

If $|V_{\text{out}}| > \frac{2}{3}|V|$ and $|V_{\text{in}}| > \frac{2}{3}|V|$ then r is contained in a set $V_S \subseteq V$ that has S-diameter at most $\frac{d}{8}$ and $|V_S| > \frac{1}{3}|V|$.

Proof. The proof is an adapted version of the one given by Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Claim 7.1]. For $B_{\text{out}}(r) = \{v \in V \mid \text{dist}_G(r, v, S) \leq \frac{d}{16}\}$, it holds that $|B_{\text{out}}(r)| > \frac{2}{3}|V|$ because $V_{\text{out}} \subseteq B_{\text{out}}(r)$. Analogously we have that $|B_{\text{in}}(r)| > \frac{2}{3}|V|$.

As such the two sets $B_{\text{out}}(r)$ and $B_{\text{in}}(r)$ share more than $\frac{1}{3}|V|$ vertices, and we have for $V_S = B_{\text{out}}(r) \cap B_{\text{in}}(r)$ that $|V_S| > \frac{1}{3}|V|$.

Let $v \in V_S$. Then $v \in B_{\text{out}}(r)$ and $v \in B_{\text{out}}(r)$, which means that v has S-distance at most $\frac{d}{16}$ from and to r. This implies that any two vertices $u, v \in V_S$ have S-distance from and to each other of at most $\frac{d}{8}$ because they can always reach each other by a path to and from r. □

Lemma 5.10. *Given a graph $G = (V, E)$, $S \subseteq V$ and a threshold $d \geq 64\ln(|V|)$, it is possible to compute a separator $S_{\text{split}} \subseteq S$ in time $\mathcal{O}(d \cdot |E| \cdot \log(|V|))$ such that:*

- *Every SCC of* $G - E_G(S_{\text{split}})$ *has* S*-diameter at most* d*.*
- $|S_{\text{split}}| \leq \frac{32 \ln(|V|)^2}{d}|S|$.

Proof. Initially, we copy G into G' and pick an arbitrary vertex r. Now we test whether to prune away parts of G' until the rest that contains r is strongly connected and has small S-diameter, or if we instead prune away a small subgraph that contains r. For this we compute the out-separator S_{out} and in-separator S_{in} as well as V_{out} and V_{in} for r and $\tilde{d} = \frac{d}{16}$ using Lemma 5.6 and Lemma 5.8. Note that applying those lemmas with $\tilde{d} = \frac{d}{16}$ requires $d \geq 64 \ln(|V|)$.

Assume either $|V_{\text{out}}| \leq \frac{2}{3}|V|$ or $|V_{\text{in}}| \leq \frac{2}{3}|V|$ holds and let V_{sep} be the first set that satisfies the inequality and S_{sep} the respective separator. We add S_{sep} to S_{split} to separate V_{sep} from the remaining graph. Then we recursively compute a good separator S_{small} for every vertex in V_{sep} and add S_{small} to S_{split}. Now we can remove both V_{sep} and S_{sep} from G', as S_{split} is already a good separator for those vertices. Finally, we pick a new arbitrary vertex r that is still contained in G' and repeat the process until G' is empty. Computing V_{sep} can be done in time $\mathcal{O}(|E_G(V_{\text{sep}})|)$ as follows. We run the computation of V_{out} and V_{in} in parallel, as soon as the first process finishes we test the inequality, and if it is satisfied we terminate the second process. Note that the size of the graph on which we recurse on is at most $\frac{2}{3}|V|$.

On the other hand, assume that neither $|V_{\text{out}}| \leq \frac{2}{3}|V|$ nor $|V_{\text{in}}| \leq \frac{2}{3}|V|$ holds. By Lemma 5.9 vertex r is contained in a strongly connected set V_S with $|V_S| > \frac{1}{3}$ and S-diamater at most $\frac{d}{8}$. We initialize a GES-tree $\mathcal{E}$ on G' rooted at r to efficiently determine all the vertices with S-distance greater than $\frac{d}{2}$ from or to r. The initialization of $\mathcal{E}$ takes time $\mathcal{O}(d \cdot |E|)$ and allows us to query a vertex v with $\text{dist}_G(r, v, S) > \frac{d}{2}$ or $\text{dist}_G(v, r, S) > \frac{d}{2}$ in constant time.

Consider the case where $\text{dist}_G(r, v, S) > \frac{d}{2}$ holds, the other case works analogously. Using Lemma 5.8 with arguments $u = v$ and $\tilde{d} = \frac{d}{4}$ we compute an in-separator S'_{sep} and V'_{sep}, which are all the vertices that are reachable from v along incoming edges in $G - E_G(S'_{\text{sep}})$. This takes time $\mathcal{O}(|E_G(V'_{\text{sep}})|)$. By adding S'_{sep} to S_{split} we separate r from V'_{sep}, and we can recursively compute a good separator S'_{small} for every vertex in V'_{sep}. As such, if we add S'_{small} to S_{split}, we can remove $V'_{\text{sep}} \cup S'_{\text{small}}$ from $\mathcal{E}$ because S_{split} is a good separator for all those vertices. Note that because $v \in V'_{\text{sep}}$ vertex v was removed from $\mathcal{E}$, and as such also from G', and we can continue with a new vertex $v = \mathcal{E}.\text{GetUnreachableVertex}()$.

If there is no more unreachable vertex then every remaining vertex maintained by $\mathcal{E}$ is strongly connected to r, and those vertices have an S-diameter of at most d, so S_{split} is now a good separator for every vertex in V. By Lemma 5.8 the vertices in V'_{sep} and S'_{small} have S-distance at most $\frac{d}{4} + 1$ from v. Because the S-distance from r to v was at least $\frac{d}{2}$ every vertex in $V'_{\text{sep}} \cup S'_{\text{small}}$ has S-distance at least $\frac{d}{4} - 1$ to r. For $d > 8$ this is greater than $\frac{d}{8}$, which means $(V'_{\text{sep}} \cup S'_{\text{small}}) \cap V_S = \emptyset$ because vertices in V_S have S-distance at most $\frac{d}{8}$ to r. This means no vertex from V_S was pruned away, and they must all still be contained in $\mathcal{E}$. Therefore, there must be at least $\frac{1}{3}|V|$ remaining vertices in $\mathcal{E}$, which means the subgraphs we recursed on had a total size of at most $\frac{2}{3}|V|$.

Running Time: Running the computation for S_{in}, vertex set V_{in}, separator S_{out} and V_{out} can be done in time $\mathcal{O}(V_{\text{sep}})$ by running the computation in parallel, which means we switch from one execution to the other after each instruction. If the first process terminates and already satisfies the size constraint we can stop the other process, which then also only spent time $\mathcal{O}(V_{\text{sep}})$. As such, the running time of the described procedure without recursive calls is dominated by initializing the GES-tree $\mathcal{E}$, which takes time $\mathcal{O}(d \cdot |E|)$. Because we only ever recurse on a graph with size at most $\frac{2}{3}|V|$ every vertex can at most take part in $\mathcal{O}(\log(|V|))$ recursions, which gives a total running time of $\mathcal{O}(d \cdot |E| \cdot \log(|V|))$.

Size of S_{split}: To determine the size of S_{split} Bernstein, Gutenberg, and Wulff-Nilsen, Claim 7.2 used a charging argument. If the set V_{sep} is small we add S_{sep} and S_{small} to S_{split}. Because of Lemma 5.6 or Lemma 5.8, depending on which lemma produced the separator, we have that $|S_{\text{sep}}| \leq \min\{|V_{\text{sep}} \cap S|, |\,[V \backslash (S_{\text{sep}} \cup V_{\text{sep}})] \cap S|\} \cdot \frac{32 \ln |V|}{d}$ because we used $\tilde{d} = \frac{d}{16}$. Note that this just means that S_{sep} contains at most $\frac{32 \ln |V|}{d}$ times as many vertices as the smaller side of the cut counting only S-vertices. Otherwise, we add S'_{sep} and S'_{small} to S_{split}. In this case we even get a lower bound of $|S'_{\text{sep}}| \leq \frac{8 \ln |V|}{d}$ because we used $\tilde{d} = \frac{d}{4}$.

Now we charge every vertex in S contained in the smaller side of the cut, in regard to the number of S-vertices, with $\frac{32 \ln |V|}{d}$ to cover the costs of adding S_{sep} or S'_{sep}. Note that the costs for adding S_{small} and S'_{small} are covered in the recursive calls. The smaller side of the cut can only contain at most half of the S-vertices, as such every vertex in S can only be contained in the smaller side of the cut $\log_2(|V|)$ times and as such can only be charged at most $\log_2(|V|)$ times. The total cost and size of S_{split} is then at most $\frac{32 \ln(|V|)^2}{d}|S|$. □

5.2.4 Simultaneous SCC Computation

The third requirement in Definition 5.3 states that for every $i \in \{1, ..., \alpha\}$ and every node $X \in V(\hat{G}_i)$ there must be an SCC $\mathcal{X}$ of $\hat{G}_{i-1} - E_{\hat{G}_{i-1}(S_i)}$ such that $X = \bigcup \mathcal{X}$. The separator S_i will be computed using the procedure described in Lemma 5.10 applied to $\hat{G}_{i-1}$ and S_{i-1}. As such it makes sense to extend Lemma 5.10 to simultaneously compute the SCCs of $G - E_G(S_{\text{split}})$ when computing S_{split}.

Lemma 5.11. *Given a graph $G = (V, E)$, $S \subseteq V$ and a threshold $d \geq 64 \ln(|V|)$, it is possible to compute a separator $S_{\text{split}} \subseteq S$ and the SCCs P of $G - E_G(S_{\text{split}})$ in time $\mathcal{O}(d \cdot |E| \cdot \log(|V|))$ such that:*

- *Every SCC of $G - E_G(S_{\text{split}})$ has S-diameter at most d.*
- $|S_{\text{split}}| \leq \frac{32 \ln(|V|)^2}{d}|S|$.

Proof. Recall that the procedure described in Lemma 5.10 consists of two phases. In the first phase we run the computation of Lemma 5.6 and Lemma 5.8 in parallel to decide whether the arbitrarily chosen root vertex r is contained in a small neighborhood V_{sep}. If this is the case the procedure adds S_{sep} to S_{split} and recurses on V_{sep}. This recursion yields a separator S_{small}, which is also added to S_{split}. Our extended procedure now additionally returns the SCCs P_{small} of $G[V_{\text{sep}}] - E_G(S_{\text{small}})$ in the recursive call, which we add to P. Then, we also add every vertex in S_{sep} as a single-vertex SCC to P because those vertices are do not have any incident edges in $G - E_G(S_{\text{split}})$.

In the second phase, where r is strongly connected to a sufficiently large subgraph, it suffices to iteratively prune away small neighborhoods V'_{sep} of not strongly connected vertices. Recall that this is also done by adding a separator S'_{sep} to S_{split}, and then recursing on $G[V'_{\text{sep}}]$ to compute S'_{small}. Now the recursive call additionally returns the SCCs P'_{small} of $G[V'_{\text{sep}}] - E_G(S'_{\text{small}})$, which we add to P. And again, because the vertices in S'_{sep} do not have any incident edges in $G - E_G(S_{\text{split}})$, we add them as single-vertex SCCs to P.

After both phases the only part that remains is a strongly connected subgraph of G with small S-distance. Therefore, we add the remaining vertices to P as a single SCC. Finally, we return P in addition to S_{split}. □

5.3 Initialization

In the following we will have a detailed look at the initialization of the data structure. Let $G = (V, E)$ be a graph, $n = |V|$ and $m = |E|$. Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Algorithm 1] additionally assume that the input graph G is strongly connected. They do not further mention why this is important, but for our purposes to extend the data structure to also maintain a topological ordering of SCCs this is important, as we will see in Section 5.5. If the graph is not strongly connected, we can run Tarjan's algorithm [Tar72] to find the SCCs of G and run the data structure on each SCC separately. This is correct, as an edge deletion only affects the connectivity of an SCC if the edge runs between two vertices contained in the same SCC. As a second option, we can make G strongly connected by taking an arbitrary ordering of the vertices $v_1, ..., v_n$ and adding the edges (v_i, v_{i+1}) for all $i \in \{1, ..., n-1\}$ and edge (v_n, v_1) if those edges do not exist yet and then deleting the edges after the initialization. This adds at most n edges and if $m = \Omega(n)$ the expected total update time stays unchanged.

In the initialization of the data structure, we need to compute a graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G and a good separator hierarchy $S_0, ..., S_\alpha$ for that graph hierarchy and a threshold $\delta \in \mathbb{N}_+$. By Definition 5.3 we know that $S_0 = \{\{v\} \mid v \in V\}$ and, because $S_0 \subseteq V(\hat{G}_0)$ needs to hold, $\hat{G}_0 = (\{\{v\} \mid v \in V\}, \hat{E})$, where $\hat{E}$ contains for every edge $(u, v) \in E$ an edge $(\{u\}, \{v\})$. This gives us a starting point for computing the graph hierarchy and the good separator hierarchy. Note that because in a good separator hierarchy the size of the separator halves with each level, at level $\alpha = \lfloor \log_2(n) \rfloor + 1$ we have $S_\alpha = \emptyset$.

The hierarchies are constructed iteratively for $i = 1, ..., \alpha$. Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Algorithm 1] use Lemma 5.11 to compute the good separator S_i for S_{i-1}, $\hat{G}_{i-1}$ and threshold $d = \frac{\delta}{2}$. By the second condition of Lemma 5.11 it then holds that $|S_i| \leq \frac{64 \ln(n)^2}{\delta} |S_{i-1}|$. Therefore, we must choose $\delta = 128 \ln(n)^2$ because then $|S_i| \leq \frac{1}{2} |S_{i-1}|$. The result of this is that every SCC of a condensation graph $\hat{G}_i$ of the graph hierarchy has initially an S_i-diamater of at most $\frac{\delta}{2}$ even though by Definition 5.3, diameters of up to 2δ are allowed. We want to briefly mention that Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 6] chose the value $64 \ln(n)^2$ for δ, which would be correct if they used the threshold $d = \delta$ when applying Lemma 5.11.

Lemma 5.12. *Let $G = (V, E)$ be a strongly connected graph, $n = |V|$, $m = |E|$ and $\delta = 128 \ln(n)^2$. It is possible to construct a graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G with $\alpha = \lfloor \log_2(n) \rfloor + 1$ and a good separator hierarchy $S_0, ..., S_\alpha$ for δ and that graph hierarchy in time $\mathcal{O}(m \log(n)^4)$.*

Proof. First, we set $S_0 = V(\hat{G}_0) = \{\{v\} \mid v \in V\}$ and $E(\hat{G}_0) = \{(\{u\}, \{v\}) \mid (u, v) \in E\}$. Given $\hat{G}_{i-1}$ and S_{i-1} for $i \in \{1, ..., \alpha\}$, we can compute S_i and $\hat{G}_i$ as follows. By applying Lemma 5.11 with threshold $d = \frac{\delta}{2} = 64 \ln(n)^2$ we can compute a good separator S_{split} for $\hat{G}_{i-1}$, S_{i-1} and δ, and the SCCs P of $\hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_{\text{split}})$. We set $S_i = S_{\text{split}}$, and for every SCC $\mathcal{X} \in P$ we construct a new node $X = \bigcup \mathcal{X}$ in $\hat{G}_i$. Then, for every edge $(u, v) \in E$, we construct the edge (X_u, X_v) with X_u and X_v being the nodes in $\hat{G}_i$ that contain u and v respectively.

By construction and our choice of δ the second condition of Definition 5.3 is satisfied, and because the size of every separator halves with each level we also have at level $\alpha = \lfloor \log_2(n) \rfloor + 1$ that $S_\alpha = \emptyset$, which means the first condition is also satisfied. Further, by construction for every $i \in \{1, ..., \alpha\}$ and node $X \in \hat{G}_i$ there exists an SCC $\mathcal{X}$ in $\hat{G}_{i-1}$ such that $X = \bigcup \mathcal{X}$, and the third condition of Definition 5.3 is also satisfied.

Each application of Lemma 5.11 takes time $\mathcal{O}(\delta m \log(n)) = \mathcal{O}(m \log(n)^3)$, and because we have $\alpha = \mathcal{O}(\log(n))$ we only apply Lemma 5.11 logarithmically many times, so the total time spent is $\mathcal{O}(m \log(n)^4)$. □

Before we continue describing how to delete an edge from the data structure we need to augment the graph hierarchy with GES-trees. Currently, the graph hierarchy satisfies that for every $i \in \{1, ..., \alpha\}$ and every node $X \in V(\hat{G}_i)$ there exists an SCC $\mathcal{X}$ in $\hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_i)$ such that $X = \bigcup \mathcal{X}$. After deleting an edge we need to determine if $\mathcal{X}$ is still strongly connected. For this task we use the GES-tree data structure as described in Section 2.6. As such, we initialize for X a GES-tree $\mathcal{E}$ on $\hat{G}_{i-1}[\mathcal{X}]$. We call a graph hierarchy *augmented* if it additionally stores for every $i \in \{1, ..., \alpha\}$ and node $X \in V(\hat{G}_i)$ a GES-tree on $\hat{G}_{i-1}[\mathcal{X}]$ where $\mathcal{X}$ is an SCC in $\hat{G}_{i-1}$ such that $X = \bigcup \mathcal{X}$. In Theorem 5.13 we show that initializing GES-trees for every node does not negatively affect the running time of initializing the graph hierarchy and good separator hierarchy.

Theorem 5.13. *Let $G = (V, E)$ be a strongly connected graph, $n = |V|$, $m = |E|$ and $\delta = 128 \ln(n)^2$. It is possible to construct an augmented graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G and a good separator hierarchy $S_0, ..., S_\alpha$ for δ and that graph hierarchy in time $\mathcal{O}(m \log(n)^4)$.*

Proof. The construction is the same as in Lemma 5.12, with one addition. After applying Lemma 5.11 on $\hat{G}_{i-1}$, S_{i-1} and $\frac{\delta}{2}$ for an $i \in \{1, ..., \alpha\}$ to compute a good separator S_{split} for $\hat{G}_{i-1}$ and S_{i-1}, and the SCCs P of $\hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_{\text{split}})$ we additionally initialize for every SCC $\mathcal{X} \in P$ a GES-tree $\mathcal{E}$ on $\hat{G}_{i-1}[\mathcal{X}]$ up to depth δ. We then assign the node $X \in V(\hat{G}_i)$ for which $X = \bigcup \mathcal{X}$ holds the GES-tree $\mathcal{E}$. By Lemma 5.12, we have constructed a graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G and a good separator hierarchy $S_0, ..., S_\alpha$ for G, δ and that graph hierarchy. By construction this graph hierarchy is also augmented.

By Theorem 2.33 constructing the GES-tree for an SCC $\mathcal{X}$ in $\hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_{\text{split}})$ takes time $\mathcal{O}(|E_{\hat{G}_{i-1}}(\mathcal{X})| \cdot \delta)$. Because the SCCs partition the graph, and the graph contains an edge-equivalent for every edge in E, the total time spent to initialize the GES-trees for all nodes on a level i is $\mathcal{O}(m \cdot \delta)$. Because we have $\mathcal{O}(\log(n))$ levels and $\delta = \mathcal{O}(\log(n)^2)$, the total time spent on initializing GES-trees is $\mathcal{O}(m \cdot \delta \cdot \log(n)) = \mathcal{O}(m \log(n)^3)$. □

Note that we also maintain on every level a mapping of vertex to the node that contains the vertex. Clearly such a mapping can be initialized in time $\mathcal{O}(n \log(n))$ for $\mathcal{O}(\log(n))$ hierarchy levels.

5.4 Edge Deletion

Before deleting an edge $e = (u, v) \in E$ from G we have an augmented graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G and a good separator hierarchy $S_0, ..., S_\alpha$ for that graph hierarchy and a $\delta = 128 \ln(n)^2$. What this means is that for every $i \in \{0, ..., \alpha - 1\}$ we have that the S_i-diameter of an SCC in $\hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ is at most 2δ, and that for every node $X \in V(\hat{G}_{i+1})$ there exists an SCC $\mathcal{X}$ in $\hat{G}_i$ such that $X = \bigcup \mathcal{X}$. After deleting e from G we clearly also need to remove the edge-equivalents in the condensation graphs such that $\hat{G}_0, ..., \hat{G}_\alpha$ stays based on G. This removal of edges could cause SCCs in the condensation graphs to no longer be strongly connected. Then the nodes in the level above do no longer represent SCCs of the condensation graph below them. Therefore, we may also need to update the nodes of the condensation graphs. It may also happen that the S_i diameter of an SCC in $\hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ exceeds 2δ, which means we need to add nodes to S_{i+1}. In the

following we use the superscript of a node to indicate the level of the condensation graph that contains that node. For example, we denote with X^i a node contained in graph $\hat{G}_i$.

The deletion happens from the bottom of the hierarchy to the top, so for each $i = 0, ..., \alpha$ we delete (X_u^i, X_v^i) from $\hat{G}_i$, where X_u^i is the node containing u and X_v^i the node containing v. For $i < \alpha$ we also determine X_u^{i+1} and X_v^{i+1}, which are the nodes that contain u and v in $\hat{G}_{i+1}$. If these two nodes are different then we know that the edge deletion did not cause an SCC in $\hat{G}_i - E_{\hat{G}_i}(S_{i+1})$ to exceed S_i-diameter 2δ nor to no longer be strongly connected because the edge was not contained in an SCC to begin with. On the other hand, if $X_u^{i+1} = X_v^{i+1}$, then the deleted edge runs between two nodes of the same SCC $\mathcal{X}^i$ with $X_u^i, X_v^i \in \mathcal{X}^i$. In this case, and if $\mathcal{X}^i$ is no longer strongly connected or the data structure cannot guarantee that $\mathcal{X}^i$ has S_i-diamater at most 2δ then we compute a set $S_{\text{aug}} \subseteq S_i$ and the SCCs P_{new} of $\hat{G}_i[\mathcal{X}^i] - E_{\hat{G}_i[\mathcal{X}^i]}(S_{i+1} \cup S_{\text{aug}})$ such that every SCC in P_{new} has S_i-diamater at most 2δ. Afterward, we add S_{aug} to S_{i+1}, and we need to create a new node Z^{i+1} for every SCC $\mathcal{Z}^i \in P_{\text{new}}$, and update the edges in $\hat{G}_{i+1}$.

To satisfy the third condition of Definition 5.3 we simply create a node $Z^{i+1} = \bigcup \mathcal{Z}^i$ and add it to $\hat{G}_{i+1}$. Then we need to update the edges in $\hat{G}_{i+1}$ such that Z^{i+1} has an incident edge-equivalent for every edge in $E_G(Z^{i+1})$. Currently, those edges are still incident to the node $X^{i+1} \in V(\hat{G}_{i+1})$ for which $\mathcal{X}^i = X^{i+1}$ holds. Note that X^{i+1} also still contains all the vertices from Z^{i+1}. As such for every $z \in Z^{i+1}$ we remove z from X^{i+1}, update the vertex to node mapping for z to point to Z^{i+1}, and for every outgoing edge $(z, w) \in E$, and analogously for every incoming edge $(w, z) \in E$ we redirect (X^{i+1}, W^{i+1}) to now be the edge (Z^{i+1}, W^{i+1}) with W^{i+1} being the node that contains w. Note that in total this takes time $\mathcal{O}(|E_G(Z^{i+1})|)$. Consider Figure 5.3, it shows a graph G after deleting edge (v_2, v_3), which caused $X = \{v_1, v_2, v_3\}$ to no longer be strongly connected. As such, the node $Z = \{v_2\}$ was inserted into $\hat{G}$, which is also depicted. Before redirecting edges, node Z does not have any incident edges, and node X still contains v_2. Further, the condensation graph $\hat{G}$ still contains the edge (X, X), which represents edge (v_1, v_2), and edge (X, Y), which represents (v_2, v_4). These edges need to be redirected. Figure 5.3 also shows $\hat{G}$ after redirection of (X, X) to (X, Z), and (X, Y) to (Z, Y).

Computing S_{aug} and P_{new}: After this high-level explanation, we will also give a more detailed description on how to actually implement the deletion efficiently. To determine whether $\mathcal{X}^i$ is still strongly connected, or it potentially exceeds S_i-diameter 2δ, we use the GES-tree $\mathcal{E}_{X^{i+1}}$ associated to X^{i+1} because this GES-tree maintains $\hat{G}_i[\mathcal{X}^i]$. If $\mathcal{E}_{X^{i+1}}$ has an unreachable node Y^i then we know that the S_i-distance of the root R^i of $\mathcal{E}_{X^{i+1}}$ to or from Y^i is greater than δ. This could be the case because there is no more path from R^i to Y^i or vice versa after deleting (X_u^i, X_v^i), or that the path became too long. In the first case, $\mathcal{X}^i$ is no longer strongly connected, and in the second $\mathcal{X}^i$ may exceed S_i-diameter 2δ. Independent of the case, and whether the S_i-diameter actually exceeds 2δ in the second case, the data structure then computes S_{aug} and P_{new}.

We could compute those sets from scratch using Lemma 5.11, but this is slow. Alternatively we can prune away a small neighborhood of the unreachable node Y^i until $\mathcal{E}_{X^{i+1}}$ has no more unreachable nodes because then the rest of the nodes contained in $\mathcal{E}_{X^{i+1}}$ are strongly connected and the S_i-diameter must be small.

Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Algorithm 4] combined both options. If $\mathcal{E}_{X^{i+1}}$ has an unreachable node Y^i they decide if root R^i of $\mathcal{E}_{X^{i+1}}$ cannot reach Y^i or vice versa. We only consider the case where $\mathcal{E}_{X^{i+1}}.\textsc{Distance}(R^i, Y^i) > \delta$ because the other case where $\mathcal{E}_{X^{i+1}}.\textsc{Distance}(Y^i, R^i) > \delta$ works analogously. We know that the S_i-distance from R^i to Y^i exceeds δ, which could mean that there is no more path from R^i to Y^i, but it could also mean that the path is just too long. In any case, Y^i must be separated from R^i. First we apply Lemma 5.8 on $\hat{G}_i[\mathcal{X}^i]$, Y^i, $S_i \cap \mathcal{X}^i$ and threshold $\tilde{d} = \frac{d}{2}$ to compute

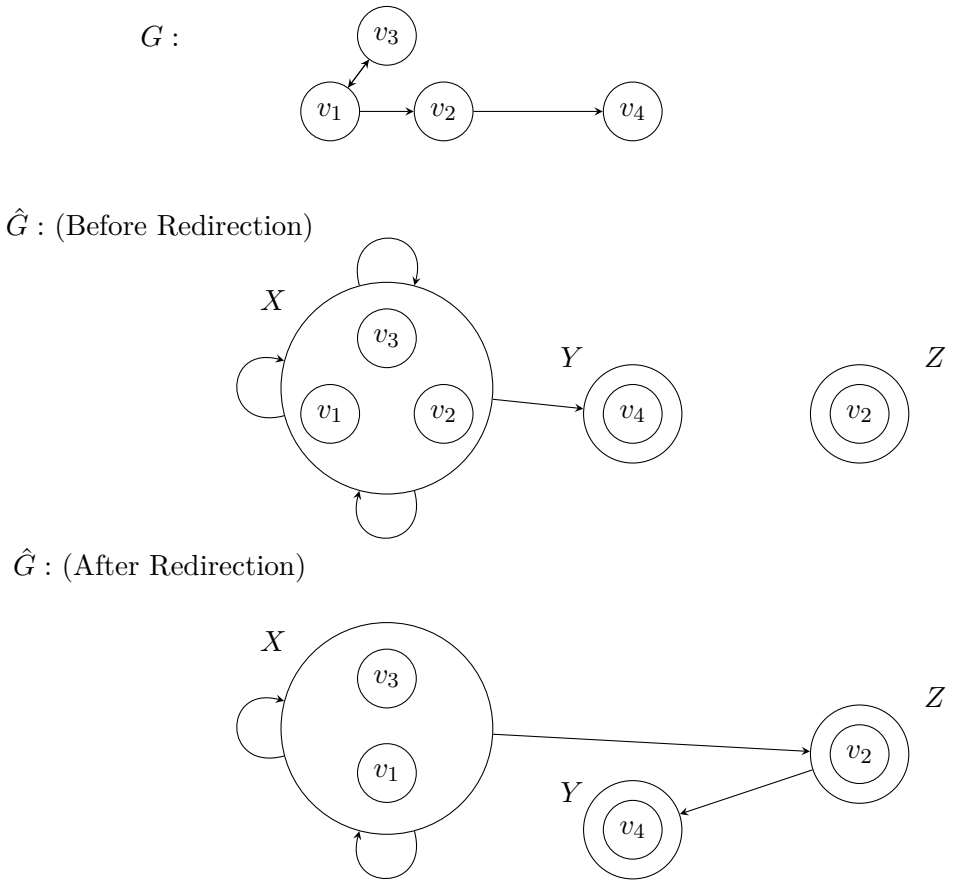

Figure 5.3: A graph G after deleting the edge (v_2, v_3) and the condensation graph $\hat{G}$ after adding a new node Z, but before redirecting the edges, as well as $\hat{G}$ after redirecting the edges.

an in-separator S_{in} and node set V_{in} that contains all the nodes reachable by Y^i along incoming edges in $\hat{G}_i[\mathcal{X}^i] - E_{\hat{G}_i[\mathcal{X}^i]}(S_{\text{in}})$. Now Bernstein, Gutenberg, and Wulff-Nilsen compare the number of vertices in nodes contained in V_{in} with the number of nodes in X^{i+1}. The reason for this is that if we were to separate V_{in} then the vertices in those nodes would also need to be removed from X^{i+1}, and for this to not become inefficient we need to limit how often a vertex can change the node it is contained in. If the size of the node that contains a vertex v always reduces by a constant factor every time v switches nodes then v can only be contained in $\mathcal{O}(\log n)$ nodes in total. As such, if $|\bigcup V_{\text{in}}| \leq \frac{2}{3}|\bigcup X^{i+1}|$ it suffices to just separate V_{in}. Afterward we still need to process V_{in} because the subgraph $\hat{G}_i[V_{\text{in}}]$ may contain SCCs with a large S_i-diameter. For this we apply Lemma 5.11 on $\hat{G}_i[V_{\text{in}}]$, $S_i \cap V_{\text{in}}$ and threshold $d = \frac{\delta}{2}$ to compute both a good separator S_{small} for every node in V_{in}, and the SCCs P_{small} of $\hat{G}_i[V_{\text{in}}] - E_{\hat{G}_i[V_{\text{in}}]}(S_{\text{small}})$ that all have S_i-diameter at most $\frac{\delta}{2}$. Then, to actually separate V_{in}, we remove $V_{\text{in}} \cup S_{\text{in}}$ from $\mathcal{E}_{X^{i+1}}$, and add $S_{\text{aug}} = S_{\text{in}} \cup S_{\text{small}}$ to S_{i+1}. In this case, the SCCs for which to create new nodes in $\hat{G}_{i+1}$ are $P_{\text{new}} = P_{\text{small}} \cup \{\{s\} \mid s \in S_{\text{in}}\}$, which means in addition to the SCCs P_{small} every node in S_{in} is included as a single-node SCC.

On the other hand, if $|\bigcup V_{\text{in}}| > \frac{2}{3}|\bigcup X^{i+1}|$ then Bernstein, Gutenberg, and Wulff-Nilsen use the fallback and compute S_{aug} and P_{new} directly by applying Lemma 5.11 on $\hat{G}_i[\mathcal{X}^i]$, $S_i \cap \mathcal{X}^i$ and threshold $d = \frac{\delta}{2}$. This initially seems inefficient, but Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Lemma 8.1] showed that in this case, with probability $\frac{2}{3}$, all

the SCCs in P_{new} contain at most $\frac{2}{3}|X^{\text{i+1}}|$ vertices. This then implies that every vertex is in expectation contained in only $\mathcal{O}(\log n)$ different nodes in a single level of the hierarchy.

Creating a new Node: We also want to go into more detail on how to actually implement the node split. As mentioned before, after computing S_{aug} and P_{new} we create for every SCC $\mathcal{Z}^i \in P_{\text{new}}$ a new node $Z^{i+1} = \bigcup \mathcal{Z}^i$ in $\hat{G}_{i+1}$. Let Y^{i+1} be the node such that $Z^{i+1} \subseteq Y^{i+1}$. Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Lemma 5.3] handled this node split in the GES-tree of Y^{i+1}. We on the other hand think it is more fitting to let the data structure handle the node split because after splitting Z^{i+1} off of Y^{i+1} we not only need to redirect edges in $\hat{G}_{i+1}$, but we may also need to fix the GES-tree assigned to a node $X^{i+2} \supseteq Y^{i+1}$ in $\hat{G}_{i+2}$.

First, we test if Z^{i+1} contains more than half of the vertices of Y^{i+1}, if this is the case we instead split the node $Y^{i+1} \backslash Z^{i+1}$ off of Y^{i+1}. The reason why Bernstein, Gutenberg, and Wulff-Nilsen do it this way is that then every vertex can only be part of $\mathcal{O}(\log n)$ node splits, and thus can only be contained in $\mathcal{O}(\log n)$ many different GES-trees, as the size of the node that contains the vertex at least halves with each node split. Now we need to redirect edges in $\hat{G}_{i+1}$. For each vertex $z \in Z^{i+1}$, and every outgoing edge $(z, w) \in E$, we find the edge-equivalent (Y^{i+1}, W^{i+1}), and convert it to the edge (Z^{i+1}, W^{i+1}). We do the same for an incoming edge $(w, z) \in E$. The node W^{i+1} that contains vertex w can be found in constant time with the mappings from vertex to node that we maintain at every level in the hierarchy. As such this can be implemented to run in time $\mathcal{O}(|E_G(Z^{i+1})|)$. Note that the edge (X_u^{i+1}, X_v^{i+1}), which is the edge-equivalent of the edge (u, v), is also redirected and not yet deleted in $\hat{G}_{i+1}$. Afterward we update the vertex-to-node mappings on level $i + 1$ such that for every $z \in Z^{i+1}$ this mapping points to Z^{i+1} instead of Y^{i+1}, and we remove z from Y^{i+1}.

If $i \leq \alpha - 2$ consider the node $X^{i+2} \supseteq Y^{i+1}$ in $\hat{G}_{i+2}$. This node has a GES-tree $\mathcal{E}_{X^{i+2}}$ associated to it that maintains an SCC $\mathcal{X}^{i+1}$ of $\hat{G}_{i+1} - E_{\hat{G}_{i+1}}(S_{i+2})$ before the node split. Observe that because we did not yet delete (X_u^{i+1}, X_v^{i+1}) from $\hat{G}_{i+1}$ this SCC is still strongly connected, even after splitting Z^{i+1} from Y^{i+1}, but now $\mathcal{X}^{i+1}$ also contains Z^{i+1}, and some of the edges incident to Y^{i+1} before the split may now be incident to Z^{i+1}. Consequently, after redirecting the edges, there is now an inconsistency in $\mathcal{E}_{X^{i+2}}$ between the edges in $\mathcal{X}^{i+1}$ and the edges in the shortest path in- and out-tree. Therefore, when redirecting an edge we additionally check if that edge is a tree edge in either the in- or out-tree maintained by $\mathcal{E}_{X^{i+2}}$, and if so we also redirect that tree edge. Observe that because Y^{i+1} only has one parent in both the in- and out-tree, either Y^{i+1} or Z^{i+1} is disconnected in the shortest path trees. To fix this we just reuse the fixing procedure of the GES-tree, which, if possible, reconnects the disconnected node. Recall that the total time spent in the fixing procedure for a single GES-tree is at most $\mathcal{O}(|E(\hat{G}_{i+1}[\mathcal{X}^{i+1}])| \cdot \delta)$. Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Appendix C] argued that the time spent on the fixing procedure at this point can be included in that total.

After creating a new node Z^{i+1} we also need to augment it with a GES-tree. As such we initialize a new GES-tree $\mathcal{E}$ on $\hat{G}_i[\mathcal{Z}^i]$ rooted at a node $R^i \in \mathcal{Z}^i$. The root node R^i is chosen by choosing a vertex in $v \in \bigcup \mathcal{Z}^i$ uniformly at random, and then finding the node R^i such that $v \in R^i$. Now we need to be careful. We mentioned that if Z^{i+1} contained more than half of the vertices of Y^{i+1} we instead split off $Y^{i+1} \backslash Z^{i+1}$ to ensure that the node that is split off always contains at most half of the vertices of Y^{i+1}. If Z^{i+1} was split off Y^{i+1} we assign Z^{i+1} the GES-tree $\mathcal{E}$. Otherwise, meaning we split off and remove all the vertices in $Y^{i+1} \backslash Z^{i+1}$ from Y^{i+1}, we have $Y^{i+1} = \bigcup \mathcal{Z}^i$ because Y^{i+1} now only contains the vertices from Z^{i+1}. The GES-tree $\mathcal{E}_{Y^{i+1}}$, which is the GES-tree still assigned to Y^{i+1} only contains vertices $Y^{i+1} \backslash Z^{i+1}$ in the nodes it maintains because we removed all the

nodes in $\mathcal{Z}^i$ from it. Thus, we need to assign $\mathcal{E}_{Y^{i+1}}$ to the newly created node $Y^{i+1} \setminus Z^{i+1}$. Then, because node Y^{i+1} now has no GES-tree associated to it, we assign $\mathcal{E}$ to Y^{i+1}.

Augmenting a GES-tree: After adding S_{aug} to S_{i+1}, for $i \leq \alpha - 2$, we also need to augment the GES-trees associated to nodes in $\hat{G}_{i+2}$ because otherwise the S_{i+1}-distance labels maintained by the GES-trees do not align with the actual S_{i+1}-distances. Fortunately, we only need to augment the GES-tree $\mathcal{E}_{X^{i+2}}$ with $X^{i+2} \in V(\hat{G}_{i+2})$ being the node, before the update, such that $X^{i+1} \subseteq X^{i+2}$ holds. This suffices because all the newly created nodes, which are split off, are added to the GES-tree that maintained X^{i+1} before the update, which is $\mathcal{E}_{X^{i+2}}$, so every node in S_{aug} is maintained by $\mathcal{E}_{X^{i+2}}$. In Theorem 2.33 we defined an appropriate AUGMENT method for which the running time is already considered in the total update time of the GES-tree. We want to mention that Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Algorithm 4] augment the GES-tree $\mathcal{E}_{X^{i+1}}$, which is not correct as this GES-tree maintains a subgraph of $\hat{G}_i$ using S_i-distances, not S_{i+1}-distances.

After having deleted the edge-equivalent of (u, v) from the graph hierarchy and G the graph hierarchy is clearly again based on G. After executing the just described update procedure the separator hierarchy should again be a good separator hierarchy for the graph hierarchy and δ. Clearly, for all $i \in \{1, ..., \alpha\}$ the SCCs in $\hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_i)$ have S_{i-1}-diameter at most 2δ. Further, Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Corollary 8.3] have shown that the following equality holds after the update procedure was executed:

$$|S_i| \leq \frac{32 \ln(n)^2}{\delta} |S_{i-1}|$$

Note that the constant should actually be 64 because they always apply Lemma 5.11 with threshold $d = \frac{\delta}{2}$ instead of just δ. Either way, when inserting $\delta = 128 \ln(n)^2$ we get that $|S_i| \leq \frac{1}{2}|S_{i-1}|$, and as such S_i is by Definition 5.2 a good separator for S_{i-1}, $\hat{G}_{i-1}$ and diameter threshold 2δ. Finally, we require that for every SCC $\mathcal{X}^{i-1}$ in $\hat{G}_{i-1} - E_{\hat{G}_{i-1}}(S_i)$ there exists a node $X^i \in V(\hat{G}_i)$. By construction of the new nodes this is also still the case. Thus, the updated separator hierarchy is good for the updated graph hierarchy and threshold δ. For more details we refer to Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 8].

Theorem 5.14. *Given a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, there exists a data structure that maintains the SCCs of G under edge deletions with total expected update time $\mathcal{O}(m \log(n)^5)$ offering the following operations:*

- *DELETE(u, v): Deletes the edge $(u, v) \in E$ from G and updates the data structure.*
- *SCC(u): Returns the SCC that contains vertex $u \in V$ in time $\mathcal{O}(1)$.*

Proof. By Theorem 5.13 we can construct an augmented graph hierarchy $\hat{G}_0, ..., \hat{G}_\alpha$ based on G and a good separator hierarchy $S_0, ..., S_\alpha$ for threshold δ that graph hierarchy in time $\mathcal{O}(m \log(n)^4)$. Note that $\alpha = \lfloor \log_2(n) \rfloor + 1$.

By keeping a mapping from vertex to node on every level of the hierarchy, we can implement the SCC(u) operation to run in constant time by evaluating the mapping for u on level α, and we implement the DELETE as described in this section.

Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 9] have argued that the total time spent on a single level by the GES-trees is $\mathcal{O}(m \log(n)^3)$ because every vertex $v \in V$ is only contained in $\mathcal{O}(\log(n))$ different GES-trees in expectation, and contributes $\mathcal{O}(\delta \cdot (\deg^+(v) + \deg^-(v)))$ time to the fixing procedure. And because $\delta = \mathcal{O}(\log(n)^2)$ the total time spent with updating the GES-trees on a single level is $\mathcal{O}(m \log(n)^3)$, and on

all levels, because there are $\mathcal{O}(\log(n))$ levels, $\mathcal{O}(m \log(n)^4)$. By Lemma 5.11 computing S_{split} and P for $X \subseteq V$ on an induced graph $G[X]$ takes time $\mathcal{O}(d \cdot |E(X)| \cdot \log(n))$. Because we always have $d = \frac{\delta}{2}$, and $\delta = \mathcal{O}(\log(n)^2)$ the time needed for the computation is $\mathcal{O}(|E(X)| \cdot \log(n)^3)$. Then, because every vertex is contained in expectation in $\mathcal{O}(\log(n))$ many nodes in a single hierarchy level, the total time spent computing S_{split} and P in a single level is $\mathcal{O}(m \cdot \log(n)^4)$. For $\mathcal{O}(\log(n))$ levels the total time spent is therefore $\mathcal{O}(m \cdot \log(n)^5)$. □

Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Chapter 9] also claimed that the expected total update time can be reduced to $\mathcal{O}(|E| \log(|V|)^4)$ with a more complex analysis of the time needed to compute S_{split} and P in Lemma 5.11.

Theorem 5.15. *Given a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, there exists a data structure that maintains the SCCs of G under edge deletions with total expected update time $\mathcal{O}(m \log(n)^4)$ offering the following operations:*

- *Delete(u, v): Deletes the edge $(u, v) \in E$ from G and updates the data structure.*
- *SCC(u): Returns the SCC that contains vertex $u \in V$ in time $\mathcal{O}(1)$.*

5.5 Maintaining a Topologically Ordering of the SCCs

The data structure given by Bernstein, Gutenberg, and Wulff-Nilsen already supports the operation SCC as defined in Definition 4.7. It remains to extend the data structure to also maintain a topologically ordering σ of the SCCs of $G = (V, E)$. Thus, we need to modify the initialization to also initialize σ, and the Deletion operation to update σ.

Recall that we assumed G to be strongly connected to initialize the hierarchy. This makes it easy to initialize σ because we know that G only has a single SCC containing all the vertices of V. We also know that the nodes at level $\alpha = \lfloor \log_2(|V|) \rfloor + 1$ of the graph hierarchy maintained by the data structure are precisely the SCCs of G, as such the single node of $\hat{G}_\alpha$ will be our only node in the topological ordering σ. It is clear that this does not impact the performance of the initialization of the data structure.

Now we modify the edge deletion, more precisely, we need to handle node splits in the highest level of the graph hierarchy. Let Z be a node that is split from Y in $\hat{G}_\alpha$. Bernstein, Gutenberg, and Wulff-Nilsen [BGW23, Lemma 5.3] required that Z and $Y \backslash Z$ are not strongly connected in $\hat{G}_\alpha - E_{\hat{G}_\alpha}(S_{\alpha+1})$. Recall that we required $S_{\alpha+1} = \emptyset$ when we gave an overview of the graph hierarchy. As such, Z and $Y \backslash Z$ must not be strongly connected in $\hat{G}_\alpha$ when splitting Z off of Y. Here, we need to be careful because if we are precise, when deleting edge $(u, v) \in E$, we only delete the edge (X_u, X_v) with X_u and X_v being the nodes in $\hat{G}_\alpha$ that contain u and v respectively after we split Z from Y. So formally Z and $Y \backslash Z$ are still strongly connected when redirecting edges. Up until now this did not matter, because even though the edge (X_u, X_v) is redirected in the node split, it will be deleted in the next iteration of the deletion procedure. But for our purposes of maintaining a topological ordering of the nodes, we need to keep the existence of this edge in mind. Now assume this edge is already deleted, then the nodes Z and $Y \backslash Z$ are not strongly connected. Because of this, there are either no edges between them or only edges from Z to $Y \backslash Z$ or only edges from $Y \backslash Z$ to Z. To determine the ordering of Z and $Y \backslash Z$, it suffices to determine if there is an edge from Z to $Y \backslash Z$, and we do this when redirecting the edges.

Recall that we can assume that $|Z| \leq |Y \backslash Z|$, because otherwise we would have switched Z with $Y \backslash Z$. The vertices in Z are then removed from Y and node Y now only contains the vertices $Y \backslash Z$. We then redirect certain edges from node Y to Z. While we do this

redirection we can keep track of whether an outgoing edge of Z has its endpoint in Y. If this is the case, and the redirected edge is not (X_u, X_v), then we know there is an edge from Z to $Y \backslash Z$, which means that there cannot be an edge from $Y \backslash Z$ to Z. This only adds constant overhead to each redirection, and therefore does not affect the total update time of the data structure.

If there is such an edge, we can insert Z in σ directly before Y. Note that Y, which is contained in σ, now only contains the vertices $Y \backslash Z$. The ordering σ stays globally topologically ordered because if there was a node W with higher index than Z in σ with an edge to Z then W would also have had an edge to Y before splitting node Y and the topological ordering would have been violated.

If there is no edge from Z to $Y \backslash Z$, we insert Z directly after node Y in σ. The two nodes are still topologically sorted, even if there are edges from $Y \backslash Z$ to Z and the global ordering is also not violated, using the same argument as before.

Finally, we argue that all the elements in σ are indeed SCCs of G. When splitting Z from Y, we know that Z is an SCC of G, as it is a new node in $\hat{G}_\alpha$. After removing vertices from Y, the node Y, that now contains only the vertices $Y \backslash Z$ may not be an SCC anymore. So $\hat{G}_\alpha$ and σ both may contain a node that temporarily is not an SCC of G. But in this case, because at the end of deleting an edge all nodes in $\hat{G}_\alpha$ represent SCCs of G, a node Z' will be split off of Y eventually, which further removes vertices from Y. Finally, because we never delete nodes from $\hat{G}_\alpha$, every node present in σ is still contained in $\hat{G}_\alpha$ and therefore represents an SCC of G.

In total, we only insert $\mathcal{O}(n)$ nodes into σ, which can be done in constant time when using a linked list for example, so the total time spent with modifying σ is $\mathcal{O}(n)$.

Now we want to give an example on how adapt σ when splitting nodes. Consider the condensation graph $\hat{G}_\alpha$ depicted in Figure 5.4. This graph was the result of node splits when processing the deletion of edge (c, a) in the underlying graph G. Before the edge deletion $\hat{G}_\alpha$ was strongly connected, so σ initially contains the node $\{a, b, c, d, e, f, g\}$. Then the following node splits were executed:

- Split off $\{a\}$:

  Vertex a is removed from node $\{a, b, c, d, e, f, g\}$, which means σ contains node $\{b, c, d, e, f, g\}$.

  The new node $\{a\}$ is added before $\{b, c, d, e, f, g\}$, as there is an edge from $\{a\}$ to $\{b, c, d, e, f, g\}$.

  $\sigma = (\{a\},\ \{b, c, d, e, f, g\})$.

- Split off $\{b\}$:

  Similarly, vertex b is removed from $\{b, c, d, e, f, g\}$ and the new node $\{b\}$ is added before $\{c, d, e, f, g\}$.

  Now $\sigma = (\{a\},\ \{b\},\ \{c, d, e, f, g\})$.

- Split off $\{c\}$:

  This time the new node $\{c\}$ is added after the node from which it was split off, as there is no edge from $\{c\}$ to $\{d, e, f, g\}$.

  $\sigma = (\{a\},\ \{b\},\ \{d, e, f, g\},\ \{c\})$.

- Split off $\{g\}$: Node $\{g\}$ similarly does not have an edge to $\{d, e, f\}$, so it is added after it.

  $\sigma = (\{a\},\ \{b\},\ \{d, e, f\},\ \{g\},\ \{c\})$.

- Split off $\{e, f\}$:

  As $|\{e, f\}| > |\{d\}|$, we actually split off node $\{d\}$, and because $\{d\}$ does have an edge to $\{e, f\}$, we place it before $\{e, f\}$.

  Finally: $\sigma = (\{a\},\ \{b\},\ \{d\},\ \{e, f\},\ \{g\},\ \{c\})$

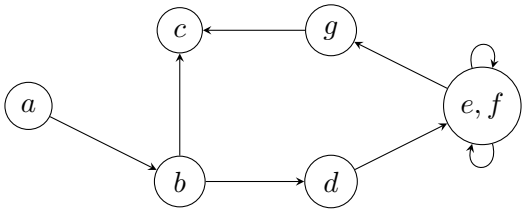

Figure 5.4: A condensation graph $\hat{G}_\alpha$.

As such we can extend the data structure described by Bernstein, Gutenberg, and Wulff-Nilsen to also maintain the SCCs as topologically ordered list without affecting the time complexity of the Initialization or the deletion operation.

Theorem 5.16. *Given a graph $G = (V, E)$ with $n = |V|$ and $m = |V|$ there exists a DSCC data structure D with total expected update time $\mathcal{O}(m \log(n)^4)$.*

6. Evaluation

Naturally, we also implemented the algorithms to compute the weak and strong control closure [Hin26]. First we will compare our algorithms with the ones given by Danicic et al., which we also implemented to get a more meaningful comparison. Then we compare the DSCC data structure based on Tarjan's algorithm as described in Theorem 4.8 with the DSCC data structure given by Bernstein, Gutenberg, and Wulff-Nilsen, as in theory the latter has a better asymptotic running time but in practice, because CFGs are not very dense and SCCs should decompose quickly, the simple implementation, which has much less overhead, should be very fast. Finally, we will also compare our algorithms to the ones given by Masud [Mas22] and our algorithm to compute the weak control closure with the one given by Léchenet, Kosmatov, and Le Gall [LKL23].

The algorithms were implemented in Java and all measurements were run on an Intel Xeon E5-2690v2 CPU at 3.00 GHz with 64 GB RAM memory and JVM arguments "-Xss512m -Xmx60g". Regarding the correctness of the implementations, we designed test graphs that cover all the different cases of when a vertex is not weakly or strongly committing, and additionally tested if our implementations, including those of the algorithms described by Danicic et al., computed the same results for randomly generated instances. Further, we compared the output of our algorithms with the output of Masud's algorithms and his implementation of the algorithms given by Danicic et al.

6.1 Random CFGs

First, we will compare the running times on randomly generated CFGs. To generate a CFG we are given the number of vertices $n \geq 2$ as well as a probability parameter $p \in [0, 1]$. The probability parameter p is the probability that an outgoing edge of a vertex exists, which means with probability p the first outgoing edge exists, and also with probability p the second outgoing edge exists. The target of an outgoing edge is chosen uniformly at random out of all the n vertices, excluding duplicates in the case of vertices that have two outgoing edges. Then, the set V' of vertices is chosen based on another probability parameter $p' \in [0, 1]$ such that each of the n vertices is contained in V' with probability p'. In the case of computing the strong control closure the algorithms also determine whether a vertex is a final vertex. For this a set P of predicate vertices is needed. Naturally, every vertex with two outgoing edges is contained in P but because those are not final vertices we also include some incomplete predicate vertices. Using another probability parameter $p_F \in [0, 1]$ we add a vertex that has at most one outgoing edge to P with probability p_F.

6.1.1 Computing the Weak Control Closure

For simplicity, we call the algorithm given by Danicic et al. and that we described in Theorem 2.16 the *cubic* algorithm because of its asymptotic running time of $\mathcal{O}(n^3)$. We call the algorithm described in Theorem 4.12 the *quadratic* algorithm if it uses the DSCC data structure implemented with Tarjan's algorithm, and the *polylog* algorithm if it uses the DSCC data structure described by Bernstein, Gutenberg, and Wulff-Nilsen.

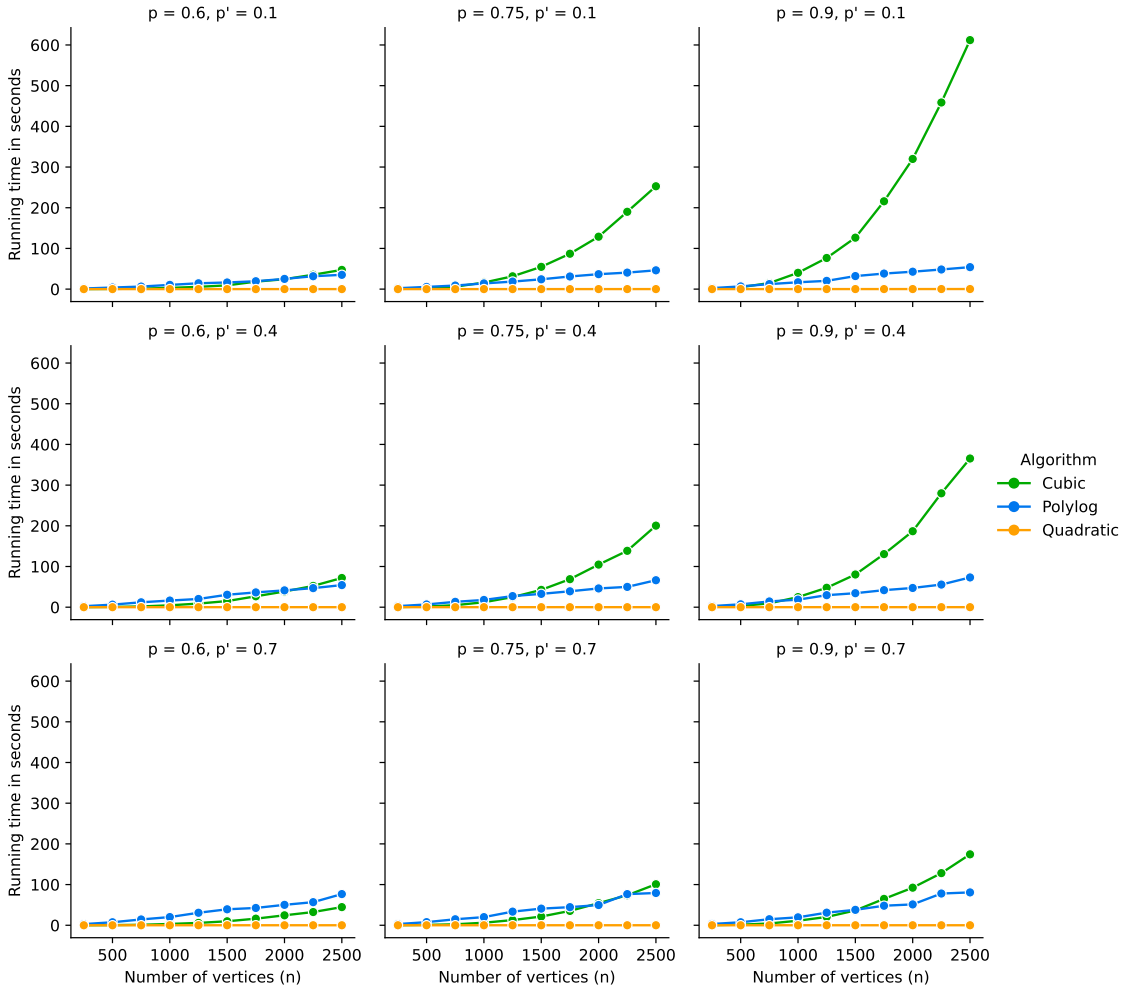

Figure 6.1: The running times of different algorithms that compute the weak control closure on randomly generated CFGs.

Figure 6.1 shows a matrix of plots where each plot shows the running time of the three algorithms on randomly generated CFGs with an increasing number of vertices for different values of the parameters p and p'. Note that every data point represents the average running time on 20 CFGs randomly generated with the corresponding parameters. The first observation that can be made is that if p is small, which is the case in the first column of the matrix, then the running time is lower than for larger values of p for all three algorithms, independent of p'. For the cubic algorithm this is immediately apparent when comparing the plots in a single row from left to right. For the polylog algorithm the increase in the running time is not as significant, but nevertheless noticeable. We isolated the running time of the quadratic algorithm in Figure 6.2 to better see the differences as it is the fastest of the three algorithms. Especially in the first row we can see that increasing p results in a higher running time. This result is not surprising because a higher parameter p means more edges to process. The increasing number of edges is especially noticeable in the cubic

algorithm, as this algorithm iterates potentially once over all the edges for every vertex contained in the weak control closure. For the quadratic and polylog algorithm a higher number of edges means a denser graph and as such potentially larger SCCs. Noteworthy is that for $p = 0.9$ and $p' = 0.1$ the running time of the quadratic algorithm appears to grow slightly faster than linear growth.

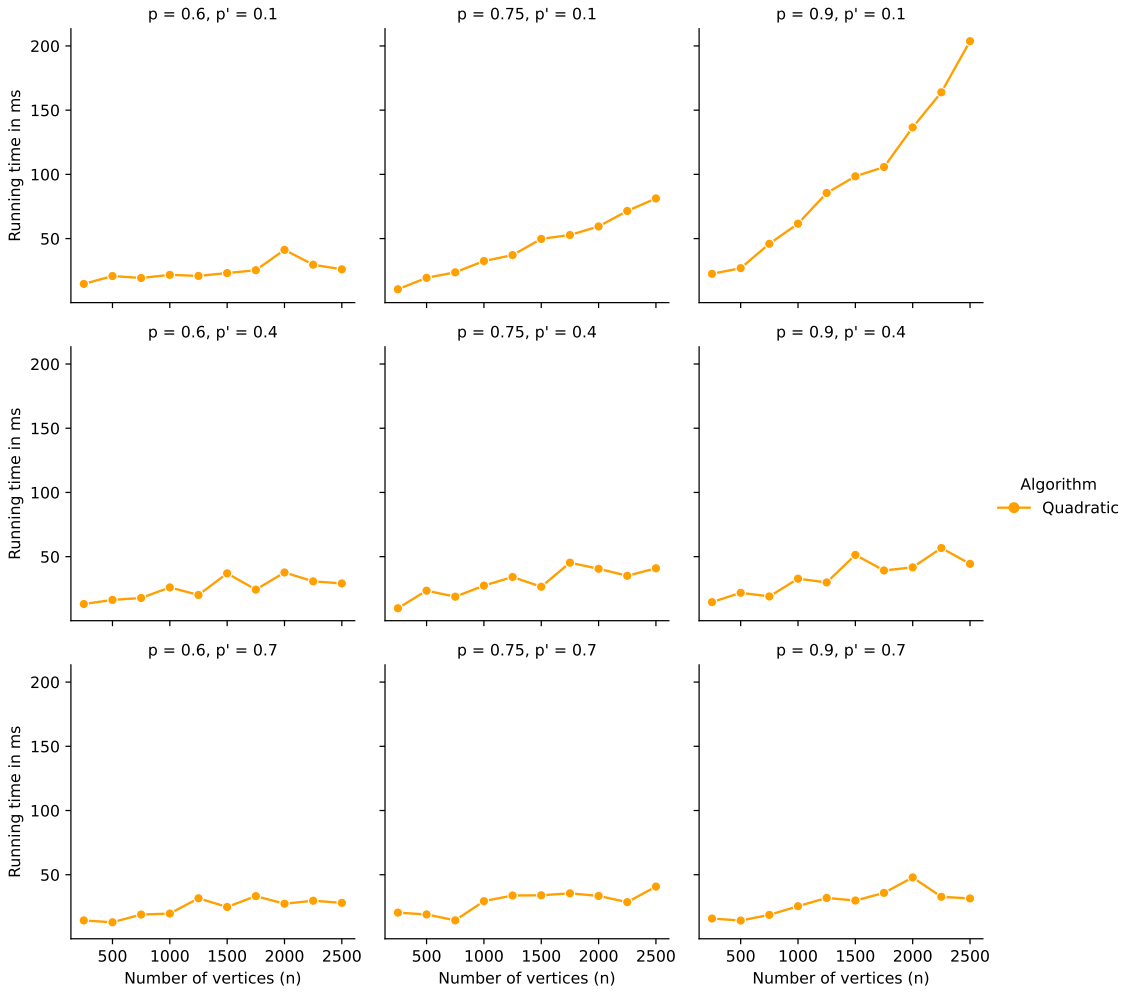

Figure 6.2: The running time of the quadratic algorithm that computes the weak control closure on randomly generated CFGs.

Analogously we can consider the columns of the plot matrix independently of p and observe, especially in the last column, that the lower parameter p' is the higher the running time is. This correlation is also not surprising, as a higher p' value corresponds to more vertices contained in V' in expectation. For the cubic algorithm this means that it needs to iterate fewer times over all the edges, as there are fewer vertices in $V \backslash V'$, and for the quadratic and polylog algorithms, which both initially remove the edges $E_G^{\text{out}}(V')$, it means that G is much sparser and thus has smaller SCCs.

What we can also observe is that the quadratic and polylog algorithms running times behave more like linear algorithms. The reason for this is that their running time depends on the decomposition of SCCs instead of just the number of edges, and because every vertex only has at most two outgoing edges CFGs are not very dense so SCCs decompose quickly. We also measured this decomposition. Figure 6.3 shows different statistics of the execution of our algorithm. The results are averaged over 20 different randomly generated graphs. Note that the statistics are the same for both the quadratic and polylog algorithm, as the

algorithms must have processed exactly the same SCCs when terminating. First, we can see that independent of n the four metrics are approximately the same. One metric shown is the average size of the processed SCC before it splits. Observe that for large values of p' the average size of the processed SCC is close to one, which means the algorithm mostly processes single-vertex SCCs and no SCC decomposition happens. Additionally, the lower p is the closer the average size of the processed SCC is to one. This is because for small p-values the input graph is sparse, which means there are many single-vertex SCCs, and for large p'-values V' contains more vertices for all of which we initially remove the outgoing edges so the graph $G - E_G^{\text{out}}(V')$ on which the DSCC data structure is initialized is also sparse. But for $p' = 0.1$ we see that our algorithm does indeed process SCCs that contain more than one vertex. For $p = 0.6$ we see that the average number of new SCCs is nearly equivalent to the SCC size. This means that processed SCCs immediately decompose into single-vertex SCCs, which explains why the running time appears to be linear in Figure 6.2 for smaller values of p. This is further strengthened by the ratio of the size of the largest new SCC to the size of the processed SCC being close to zero, which indicates that the largest new SCC is very small. By increasing p we can see that the gap between the size of the processed SCC and the number of new SCCs grows. Such a gap indicates that SCCS do not immediately decompose into single vertex SCCs, which explains why we can observe a slightly superlinear running time in Figure 6.2 for $p = 0.9$.

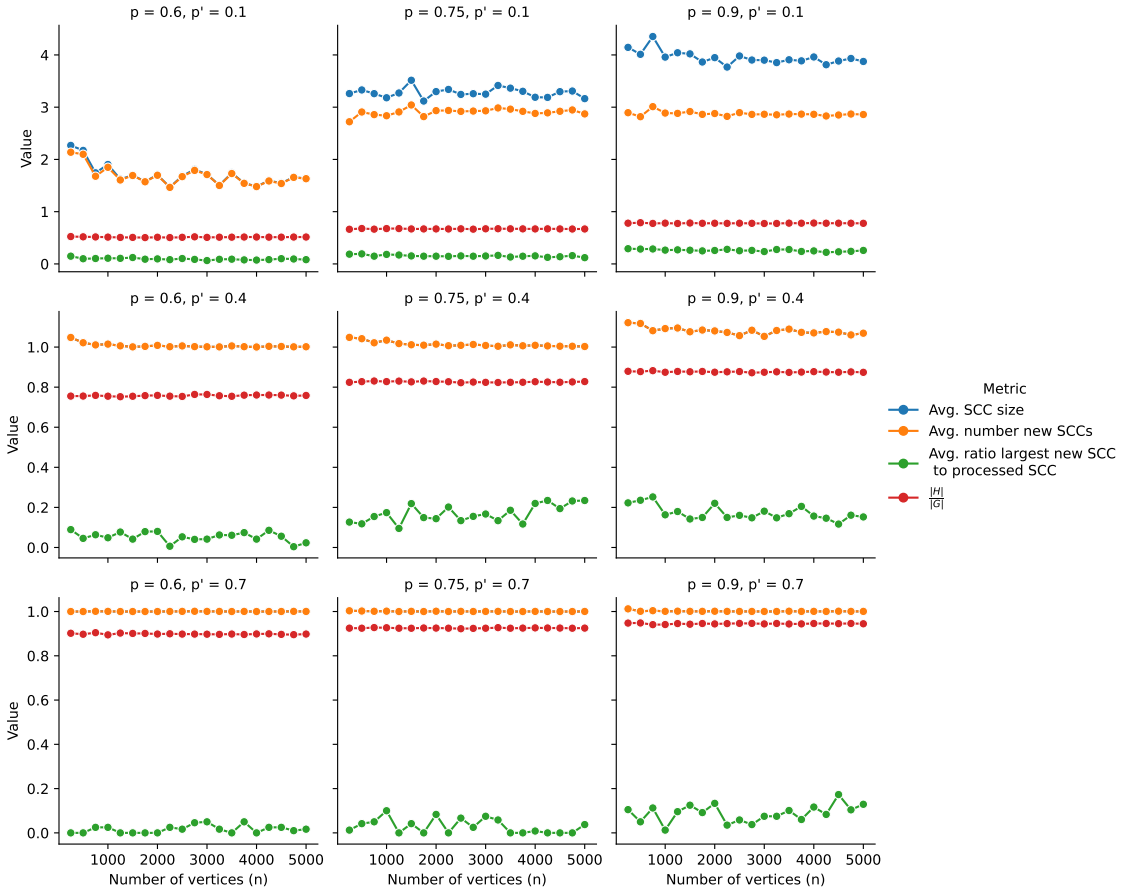

Figure 6.3: Execution statistics of our algorithm that computes the weak control closure on different sized random graphs generated with different values for p and p' independent of DSCC data structure used.

Another interesting metric is the ratio of the size of graph H, which is the graph constructed in the initialization that only contains V'-reachable vertices, to the size of the input graph G. This ratio depicts the percentage of vertices of G that are actually V'-reachable and

thus processed. The higher this ratio, the more work our algorithm needs to do. We can clearly observe that when increasing only p the percentage increases. This is because with more edges the graph is denser and more vertices become V'-reachable. Additionally, when only increasing p' then the ratio also grows because then V' contains more vertices and naturally more vertices are V'-reachable.

6.1.2 Computing the Strong Control Closure

For completeness, we will also compare the algorithms that compute the strong control closure, even though the results are nearly identical. In the following we call the algorithm described in Theorem 3.9 the *cubic* algorithm, we call our algorithm the *quadratic* algorithm if it uses Tarjan's algorithm to implement the DSCC data structure, and lastly we call our algorithm the *polylog* algorithm if it utilizes the DSCC data structure given by Bernstein, Gutenberg, and Wulff-Nilsen.

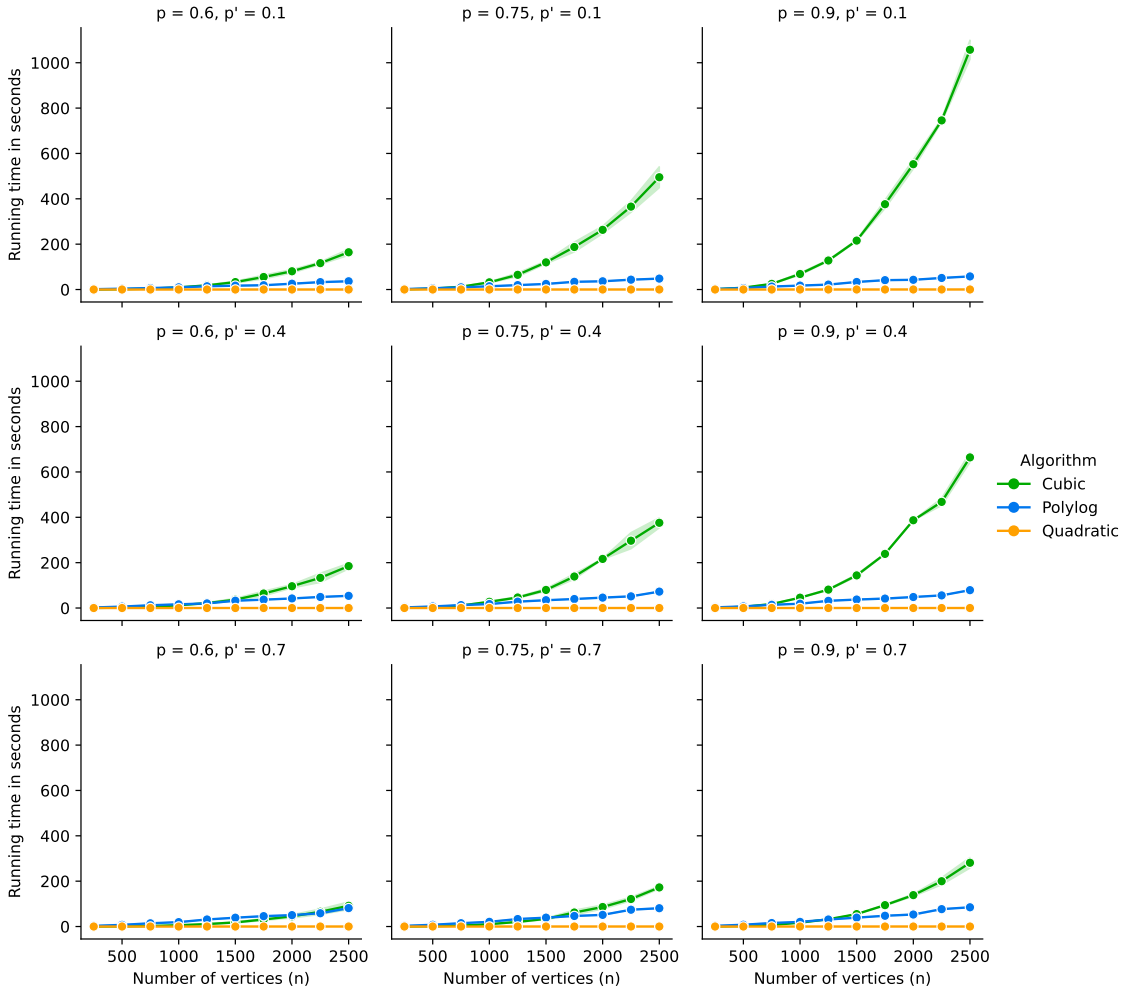

Figure 6.4: The running times of different algorithms that compute the strong control closure on randomly generated CFGs.

In Figure 6.4 the running times of all three algorithms are given, again, in a plot matrix for different combinations of p and p'. For the parameter p_F we used two values, 0.1 and 0.4. Because of this the plots show a range of running times, but observe that this range is only slightly noticeable in the cubic algorithm. Note that the running times were again averaged over 20 randomly generated CFGs for each parameter combination. This brings us to the first observation, which is that the choice of p_F does not influence the running

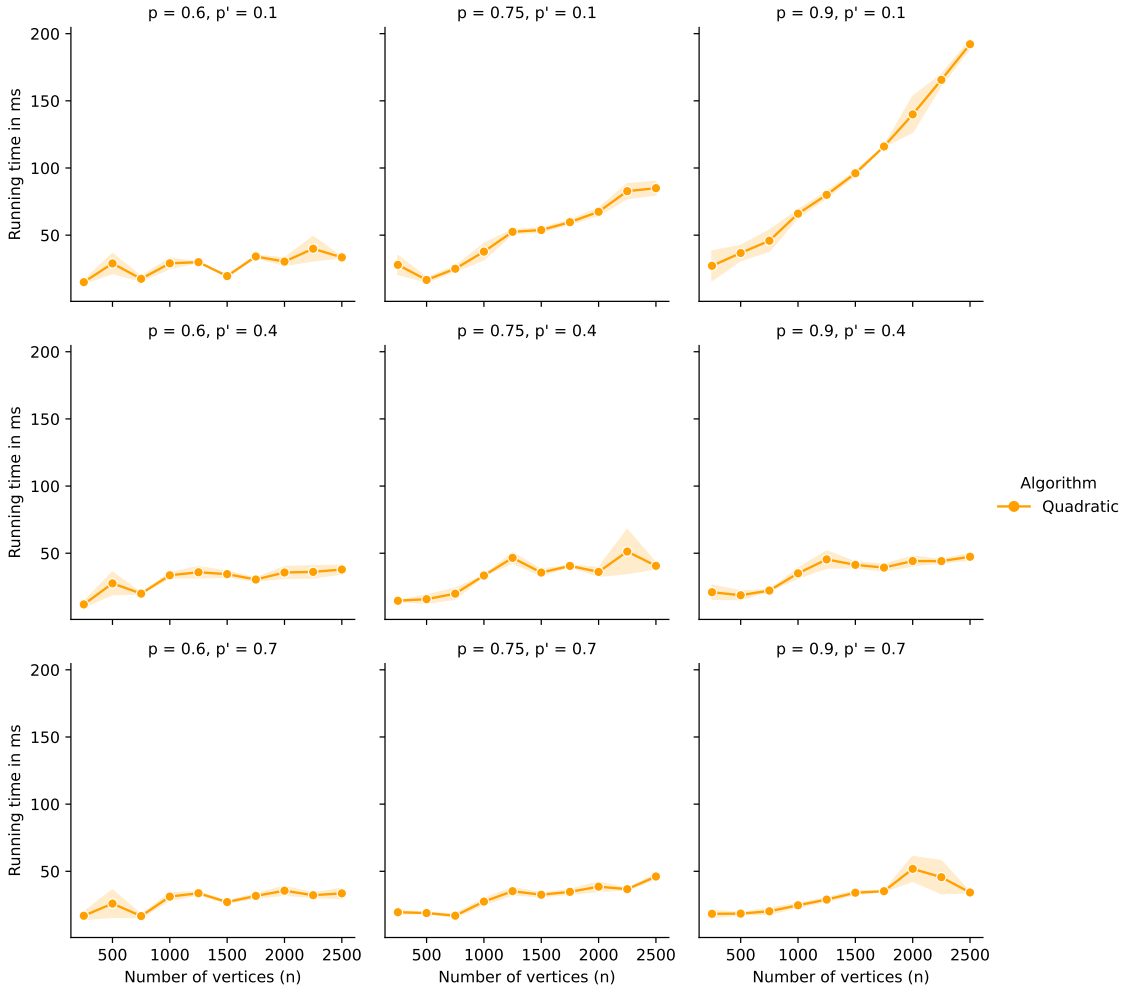

Figure 6.5: The running time of the quadratic algorithm that computes the strong control closure on randomly generated CFGs.

time of the algorithms significantly. For our algorithms this can be explained by the fact that final vertices do not greatly impact the SCC decomposition. Consider the SCC L and the set X' to which the algorithms add vertices contained in X. If L has at least two vertices we do not need to test for final vertices as we then know that every vertex lies on an X'-free complete path. It only matters if a vertex is a final vertex if $|L| = 1$, but then the SCC is already a single-vertex SCC and does not decompose any further. Note that similar to the algorithms that compute the weak control closure the running time increases with increasing p and decreasing p', which can be observed by comparing the plots in a row from left to right and in a column from bottom to top. Also, in general, the cubic algorithm is the slowest, and the quadratic algorithm is the fastest. In Figure 6.5 we isolated the running time of the quadratic algorithm again. Here we can better see the effect that the different p and p' values have on the quadratic algorithm, and we can also better observe the slight variation caused by the different p_F values. We also see that the running time of the quadratic algorithm again appears to be linear, or slightly superlinear for parameters $p = 0.9$ and $p' = 0.1$. Further, the running time does not greatly differ from the quadratic algorithm that computes the weak control closure. The reason why the quadratic algorithm is fast can again be explained with the fact that SCCs decompose quickly.

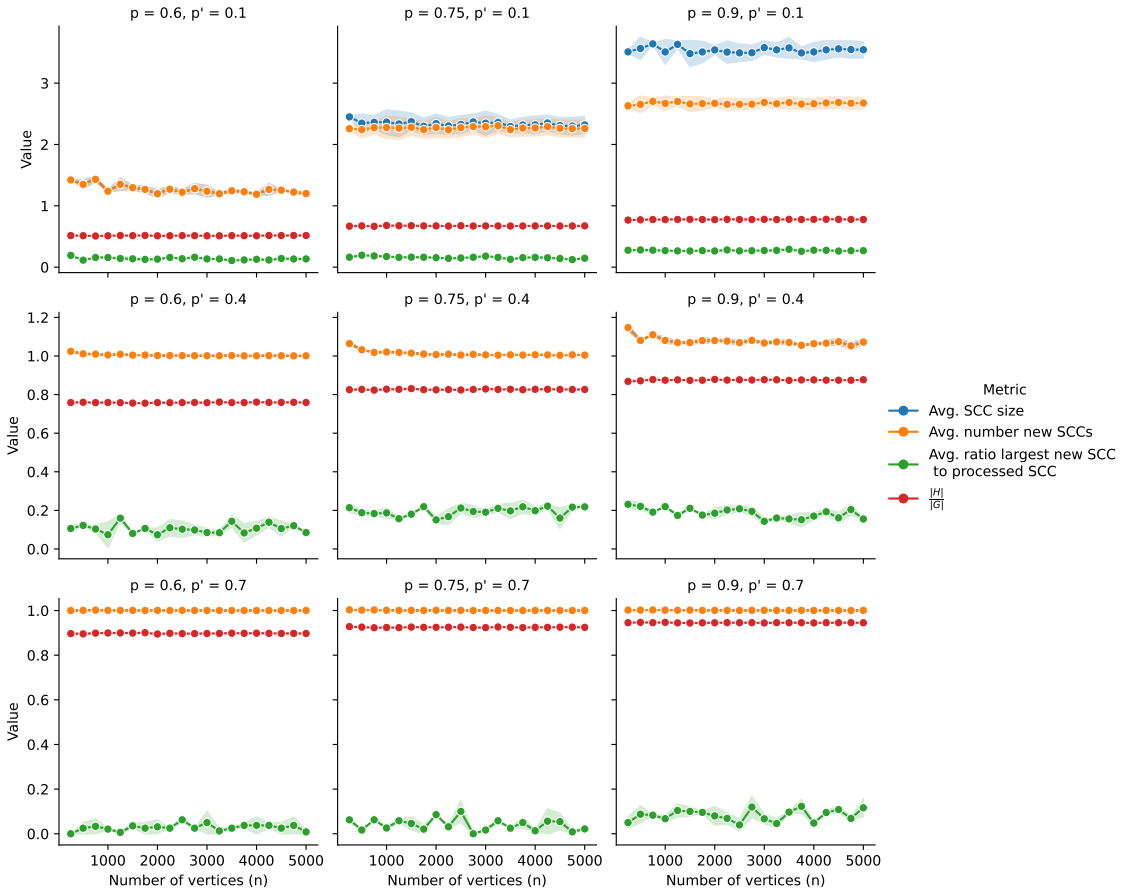

Figure 6.6: Execution statistics of our algorithm that computes the strong control closure on different sized random graphs generated with different values for p and p', as well as $p_F \in \{0.1, 0.4\}$, independent of DSCC data structure used.

In Figure 6.6 we collected different metrics of the execution of our algorithm for different values of p and p', and parameter $p_F \in \{0.1, 0.4\}$. Again each data point is the average taken from 20 different randomly generated CFGs. We can observe mostly the same results as for the algorithm that computes the weak control closure. For small values of p and large values of p' the algorithms mostly process single-vertex SCCs, for $p = 0.75$ and $p' = 0.1$ the SCCs appear to decompose immediately, and for $p = 0.9$ they do not immediately decompose, but also decompose quickly as the average number of new SCCs is close to the size of the processed SCC. We can also observe that the metrics are independent of the choice of p_F, which further strengthens our claim of different p_F-values not significantly affecting the running time. One observation we can make is that for $p' = 0.1$ the average SCC size is smaller than in Figure 6.3 for the same parameter value. This might be explained by the fact that the strong control closure X contains in general more vertices than the weak control closure, and as such SCCs in $G - E_G^{\text{out}}(X)$ are smaller. We can also see that the gap between the average size of the processed SCC and the average number of new SCCs is smaller than in Figure 6.3, especially for the parameter combination $p = 0.75$ and $p' = 0.1$. Naturally the ratio between the number of vertices in H and the number of vertices in G is the same as for the weak control closure, as the initialization is identical in both algorithms.

6.2 Comparing the DSCC Data Structures

Now that we have seen that the cubic algorithm performs far worse than our algorithm with either DSCC data structure we want to take a closer look at which DSCC data structure is better for which inputs. First we can again compare the algorithms when computing the strong control closure on randomly generated graphs, but now for even larger n. Note that we restricted ourselves to $p \in \{0.7, 0.9\}$, $p' = 0.1$ and $p_F = 0.1$ because, as we have seen before, for low p and high p' all three algorithms terminate quickly, and p_F does not sufficiently affect the running time. The results are plotted in Figure 6.7, where each data point depicts the average running time of 20 different randomly generated CFGs and as we can clearly see the algorithm that uses Tarjan's DSCC data structure is much faster. Note that the running time of both algorithms seems to increase linearly with n.

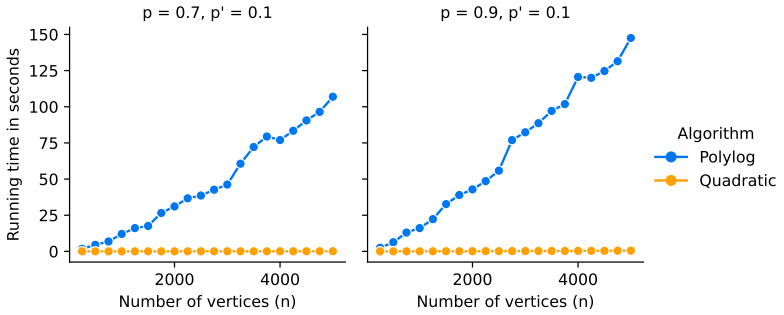

Figure 6.7: Running time of our algorithms to compute the strong control closure using different DSCC data structures on randomly generated graphs with parameters $p \in \{0.7, 0.9\}$, $p' = 0.1$ and $p_F = 0.1$.

But by taking a closer look at only the running time of the quadratic algorithm, which is depicted in Figure 6.8, we see that for $p = 0.7$ the running time appears to only grow linearly, but for $p = 0.9$ it appears to increase slightly superlinear with n, which aligns with our previous observations.

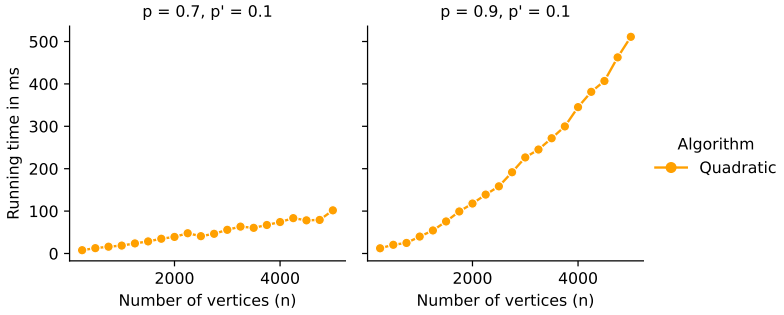

Figure 6.8: Running time of the quadratic algorithm for computing the strong control closure on randomly generated graphs with parameters $p \in \{0.7, 0.9\}$, $p' = 0.1$ and $p_F = 0.1$.

Recall that in Figure 4.7 we have depicted a graph in which Tarjan's DSCC data structure has a total update time of $\mathcal{O}(n^2)$. In theory the DSCC data structure we described in Chapter 5 should eventually be faster than Tarjan's DSCC data structure on such graphs. Because we have much larger graphs now, and as such the running time is much higher, we only ran each algorithm once for every graph. In Figure 6.9 we plotted the running time of the quadratic algorithm and the polylog algorithm to compute the weak control closure on such worst case graphs. Observe that the running time of the polylog algorithm

does no longer appear to increase linearly with n. We also isolated the running time of the quadratic algorithm to be able to better observe that the running time appears to increase quadratically in n. Still, the quadratic algorithm is much faster than the polylog algorithm, even for larger graphs.

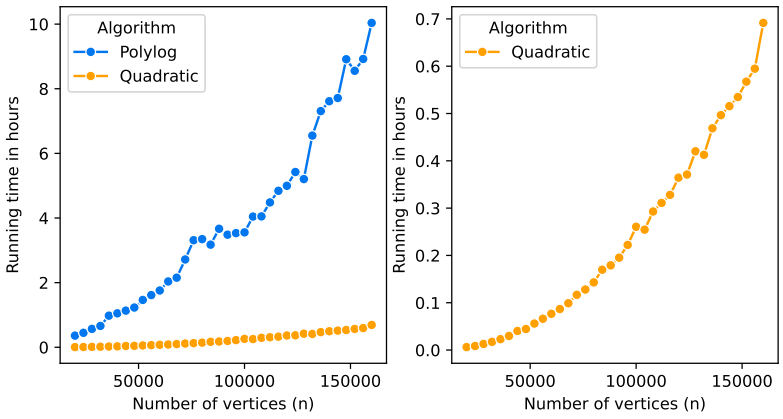

Figure 6.9: Running time of our algorithm that computes the weak control closure using different DSCC data structures on graphs where Tarjan's DSCC data structure needs quadratic time.

To conclude, while the polylog algorithm has a better theoretical expected asymptotic running time, in practice the data structure has an enormous overhead that causes the algorithm to be much slower, not only on randomly generated CFGs, but also on graphs were the quadratic algorithm performs poorly and needs time $\mathcal{O}(n^2)$. Additionally, there is the disadvantage that the polylog algorithm needs time $\mathcal{O}(n \log(n)^4)$ for initialization by Lemma 5.12, even on acyclic graphs where the quadratic algorithm has linear running time. Thus, in most practical cases it suffices to use a simple and easy to implement DSCC data structure to get a fast algorithm.

6.3 Efficient Implementation

Now that we have identified that Tarjan's DSCC data structure is faster than the one described by Bernstein, Gutenberg, and Wulff-Nilsen, even for larger graphs where the running time is quadratic, we can optimize the implementation. The first optimization we can make concerns how we delete edges from the DSCC data structure D used by the algorithm. Currently, both DSCC data structures update themselves after each edge deletion. In practice, we not only delete one edge, but multiple outgoing edges from vertices of an SCC. While the data structure described by Bernstein, Gutenberg, and Wulff-Nilsen must process each deletion individually, Tarjan's DSCC data structure can be optimized. Consider a set E' of edges and an SCC Y such that for all $(u, v) \in E'$ we have $u \in Y$. Then we can delete all the edges in E' and only then recompute the new SCCs of $G[Y]$. This way we do not need to execute Tarjan's algorithm $|E'|$ times, but only once. Further, we implemented all algorithms with an object-oriented approach, which means each vertex is represented by an object. The main reason for this decision was that the DSCC data structure described by Bernstein, Gutenberg, and Wulff-Nilsen is very complex, so an object-oriented implementation of the underlying data structures not only made the data structure easier to implement, but also made the code more readable. A consequence of this approach is that we need to allocate many objects, and for storing auxiliary data, such as the LOWLINK number in Tarjan's algorithm, we need to use hash-maps. Clearly this is not very cache-friendly. In our optimized implementation we implicitly represent each

vertex only by a number starting at zero. This allows us to store auxiliary data in arrays instead of hash-maps.

But the main improvement we made is to not maintain the complete topologically ordered list of SCCs. Consider a restricted partition $\{\mathcal{L}, \mathcal{R}\}$. This partition is maintained by the DSCC data structure as a topologically ordering σ of SCCs. When our algorithms delete edges and split an SCC L we need to replace L with the new SCCs into which L decomposed. For this insertion to be efficient we need to use a linked-list in the implementation because we do not want to move every SCC in $\mathcal{R}$ as they are contained in σ behind L. Of course a linked-list is not as cache-efficient as an array-list. Observe that our algorithms never again process an SCC in $\mathcal{R}$, which means our algorithms do not need to maintain the complete topological ordering σ, it suffices to only maintain $\sigma_{\mathcal{L}}$, which only contains SCCs from $\mathcal{L}$. Recall that to find L we always just query the predecessor in σ of the last SCC moved to $\mathcal{R}$. Therefore, in $\sigma_{\mathcal{L}}$ the SCC L is always the last SCC. This observation allows us to implement $\sigma_{\mathcal{L}}$ as an array-list because now we only need to append the new SCCs at the end of the list instead of inserting them between other SCCs.

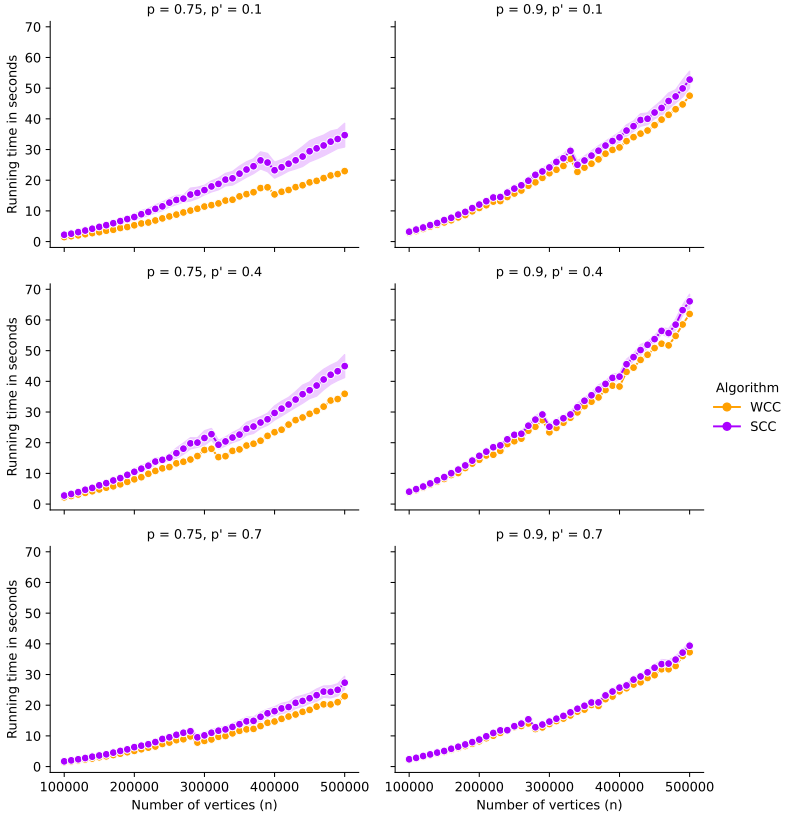

Figure 6.10: The running time of our optimized implementations on randomly generated CFGs for different values of p and p', as well as $p_F \in \{0.1, 0.4\}$ for computing the strong control closure.

Using an array-list to maintain only $\sigma_{\mathcal{L}}$ instead of a linked-list to maintain σ is much faster. From now on we will refer to the optimized implementations for computing the weak and strong control closure as WCC and SCC respectively. In Figure 6.10 we plotted the running time of both the WCC and SCC algorithm on randomly generated CFGs with different values for p and p'. Like before, each data point depicts the average running time over 20 different CFGs generated with the corresponding parameters. For p_F in the case of algorithm SCC we used both $p_F = 0.1$ and $p_F = 0.4$. Our optimized algorithms can compute both the weak and strong control closure on CFGs with up to 500 thousand

vertices in a minute for most parameter combinations depicted. Importantly the SCC algorithm is nearly as fast as the WCC algorithm. We can again observe that a higher value for p causes an increase in the running time, but now we can also observe that the running time of our algorithms on CFGs generated with parameter $p' = 0.4$ is noticeably higher than the running time for both smaller and larger values for p'. A possible explanation in the case of large p'-values is that the graph $G - E_G^{\text{out}}(V')$, which is the graph on which the DSCC data structure is initialized, contains fewer edges and as such the SCCs decompose into single-vertex SCCs faster. The lower running time on graphs with lower p'-value can be explained by the graphs containing fewer vertices that are V'-reachable, and as such more vertices are removed in the initialization. We already observed this in both Figure 6.3 and Figure 6.6 as the ratio between the size of H, which is the graph that only contains V'-reachable vertices, and the size of the input graph G is lower for small p'-values. Another noteworthy observation is that in each plot there is a drop-off of the running time at a certain n. This may be an effect of Java's garbage collector, as for larger CFGs it may reserve more memory and the garbage collector does not run as often, but this is only speculation and might be investigated in future work.

6.4 Comparison to Masud's Algorithms

Masud [Mas26] also implemented his algorithms to evaluate their running time in the Clang/LLVM compiler framework. Because the algorithm that computes the weak control closure described by Masud also has asymptotic running time $\mathcal{O}(n^2)$ just like our algorithm when using Tarjan's DSCC data structure, it makes sense to also compare the running times of the algorithms. The algorithms implemented by Masud, in contrary to our algorithms, take a C-file as input and then translate that file to a CFG. The set V' is chosen randomly, and for computing the strong control closure the set P is computed to be all the vertices with out-degree two. We will run our optimized algorithms on the graph generated by Masud's translation. Note that the generated graph may not be a CFG as it may contain a vertex v with $\deg^+(v) > 2$. Our algorithms also work on general graphs, so this is not a problem, more details follow in Chapter 7. Masud [Mas22, Chapter 6] evaluated his algorithms using the SPEC CPU 2017 [BLK18] benchmarks, as such it lends itself to compare the algorithms using the C-files in that benchmark, and afterward, because most of the CFGs generated by those C-files are relatively small, we will also compare the algorithms on custom generated C-files that produce large CFGs.

Before we compare the algorithms, we want to point out that the implementation of Masud's algorithm that computes the strong control closure does not always produce the correct results. For example consider the CFG G shown in Figure 6.11 with

$$V' = \{v_0,\ v_4,\ v_8,\ v_{17},\ v_{20},\ v_{21}\},\ P = \{v_2,\ v_4,\ v_5,\ v_7,\ v_9,\ v_{10},\ v_{12},\ v_{14},\ v_{15},\ v_{17},\ v_{18},\ v_{20}\}.$$

Both, our implementation of the algorithm described by Danicic et al. [Dan+11, Algorithm 65] and the SCC algorithm computed the strong control closure

$$X = \{v_0,\ v_4,\ v_5,\ v_8,\ v_9,\ v_{10},\ v_{17},\ v_{18},\ v_{20},\ v_{21}\}$$

for V' and P in G. Masud's implementation instead computed $X' = X \cup \{v_{15}\}$. But v_{15} is X-strongly committing in G for P, and it is also not contained in V', so it is not contained in the strong control closure. The reason why v_{15} is X-strongly control closed is that the only vertex from X it can reach is v_{10}, and it also does not lie on an X-free complete path, as the only final vertex is v_0, which is contained in X, and there exist no cycles. We colored the vertex set X in Figure 6.11 to make it easier to see that v_{15} can indeed only reach v_{10} via X-paths.

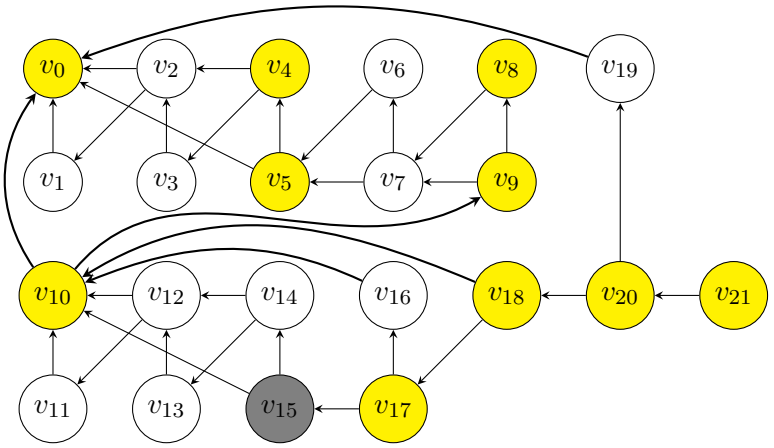

Figure 6.11: The CFG generated from the method PerlIOStdio_open in the C-file perlio.c of the perlbench benchmark with vertex v_{15} and vertex set $X = \{v_0, v_4, v_5, v_8, v_9, v_{10}, v_{17}, v_{18}, v_{20}, v_{21}\}$ highlighted.

Despite this we still want to compare our algorithms to Masud's algorithms on the instances where his implementation is correct. We ran our and Masud's algorithms on the same benchmarks that Masud [Mas22, Table 4 and 5] used to compare his algorithms to those described by Danicic et al. [Dan+11, Algorithms 61 and 65]. The excluded instances are given in Table A.1. Because V' is chosen randomly in every run for every instance every run may produce different instances on which Masud's algorithm is incorrect. As such we only ran Mausd's algorithms once, but ours again 20 times to average the running time. In Table 6.1 the results of running our optimized algorithms and those of Masud on the benchmarks are collected. In the first row the total number of instances, and the number of excluded instances, if applicable, can be found. We also computed the total number of vertices (row "Total n"), the average size of the graphs (row "Avg. n") and the median size of the graphs (row "Median n") per benchmark. The average and median graph sizes imply that the graphs in the benchmark are in general very small.

	GCC	Imagick	Mcf	Nab	Perlbench	X264	Xz
Instances	17603 (3)	359	40	328	2365 (3)	1417	466
Total n	331366	2500	773	5398	66485	15416	3434
Avg. n	18.83	6.96	19.32	16.46	28.15	10.88	7.37
Median n	7.0	3.0	14.0	9.0	9.0	6.0	5.0
WCC	366.16	5.37	3.07	7.83	70.18	21.32	6.51
WCC_M	16788.72	25.03	6.57	46.41	922.68	128.28	26.41
$\frac{\text{WCC}_M}{\text{WCC}}$	45.85	4.66	2.14	5.93	13.15	6.02	4.06
SCC	478.85	4.37	2.26	7.93	77.38	23.66	7.63
SCC_M	23369.12	38.54	10.85	81.09	1403.08	222.89	41.13
$\frac{\text{SCC}_M}{\text{SCC}}$	48.8	8.82	4.8	10.23	18.13	9.42	5.39
$\lvert V(G_{\max})\rvert$	15911	225	75	320	2270	244	87
$\frac{\text{WCC}_M(G_{\max})}{\text{WCC}(G_{\max})}$	65.19	7.11	8.82	7.99	48.4	3.93	14.05
$\frac{\text{SCC}_M(G_{\max})}{\text{SCC}(G_{\max})}$	41.51	10.05	12.53	11.84	89.17	6.78	22.45
General Graphs	1172	12	1	13	150	16	9
Largest Out-Degree	1686	54	5	13	255	28	13

Table 6.1: Statistics regarding the execution of the WCC and SCC algorithm as well as Masud's algorithms on selected benchmarks from SPEC CPU 2017 [BLK18].

The total running time in milliseconds of our WCC and SCC algorithms are given in the rows with the same name, and of the algorithms implemented by Masud are given in the rows "WCC_M" and "SCC_M". Observe that our algorithms are always faster than those of Masud, which can also be seen in the relative speedup which is given in the rows "$\frac{\text{WCC}_M}{\text{WCC}}$" and "$\frac{\text{SCC}_M}{\text{SCC}}$". Interestingly the speedup in the GCC benchmark is much higher than in other benchmarks. This may be explained by the fact that the GCC benchmark contains some instances that are relatively large, and on those specific instances our algorithms are much faster. This is further supported by the row "$|V(G_{\max})|$", which depicts the size of the largest graph in the benchmark, and the rows in which the relative speed up on this graph specifically is given. We can see that the GCC benchmark contains a very large graph and on that graph our algorithms are much faster.

Because we mentioned previously that the graphs generated by Masud's translation may not always be CFGs we also counted the number of such graphs that do not either to the out-degree restriction per benchmark (row "General Graphs"). Especially in the GCC benchmark there are many general graphs, even with high out-degrees as we can see in the row "Largest Out-Degree".

Because the graphs are in general small we want to compare the algorithms on larger CFGs where the running time may be less affected by, in our case, JVM overhead. These large CFGs are again randomly generated with parameters $n \geq 2$ and $p = 0.9$ because then CFGs are relatively dense. Because Masud's implementation needs a C-file as input we convert the CFG to a C-file as follows. The generated file contains a single method with boolean arguments cond_i for $i \in \{1, ..., n\}$, in other words for every vertex one argument. Inside this method we create for vertex i a label B_i, which will be used as goto-targets. After such a label, depending on the out-degree of vertex i, we either have a return-statement, if vertex i has no outgoing edges, or a goto-statement to label B_j if vertex i has a single outgoing edge to vertex j, or an if-else-condition with goto-statement to label B_{j_1} in the if-branch and goto-statement to label B_{j_2} in the else-branch if vertex i has two outgoing edges, one to vertex j_1 and the other to vertex j_2. Inside the generated if-statement we test the for the argument cond_i because if the if-statement were to have "True" or "False" as argument it would be reduced to a node with one outgoing edge instead of two. One example for a C-file generated this way is depicted in Figure A.1.

Afterward we run our algorithm not on the original CFG, but on the CFG generated by Masud's program because the translation does not produce exactly the same CFG. In fact, the translated CFG contains approximately 2.6 times as many vertices as the original CFG, and that number still fluctuates depending on the edges of the original CFG. Because we average the running times over 20 randomly generated CFGs with vertex count n, we plot the results for n rather than the number of vertices in the translated CFG, since the second varies between instances. In Figure 6.12 the running times of our and Masud's algorithms are depicted. We can clearly see that also for larger graphs our algorithms perform better than those described and implemented by Masud. Another observation we can make is that both computing the weak control closure and the strong control closure takes nearly the same time. Note that for each of the 20 instances per plotted n value we ran the algorithms only once.

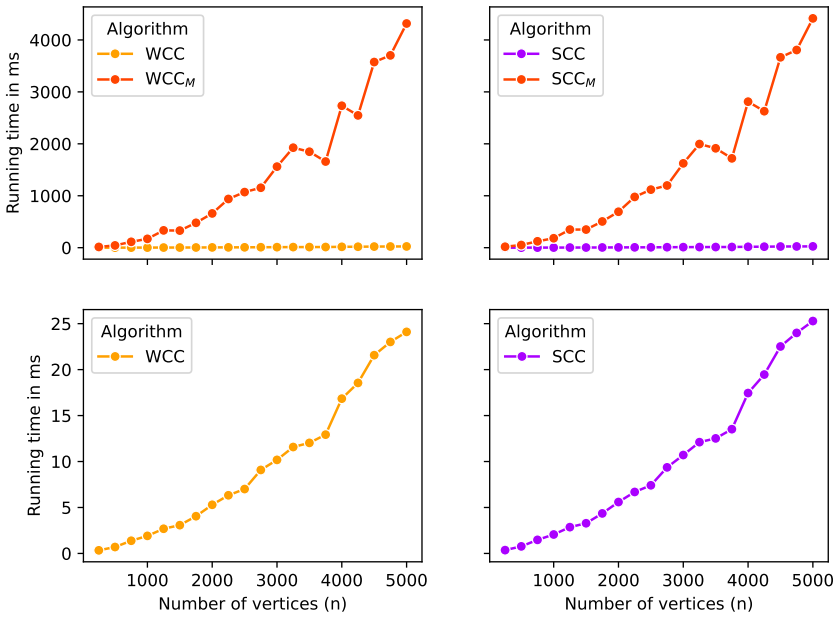

Figure 6.12: The running time of our algorithms and the algorithms described by Masud on CFGs translated from C-files generated from randomly generated CFGs with n vertices and parameter $p = 0.9$.

6.5 Comparison to the Algorithm of Léchenet, Kosmatov, and Le Gall

Léchenet, Kosmatov, and Le Gall [LKL23, Chapter 6] described the so-called lDFS algorithm that computes the weak control closure X on a graph $G = (V, E)$ for a set $V' \subseteq V$. Because the algorithm described by Danicic et al. did not reuse previously computed reachability information, Léchenet, Kosmatov, and Le Gall described an algorithm that stores some information of paths computed in previous iterations to use in the following iterations. Further, the algorithm works similar to Masud's algorithm in the sense that it computes all the V'-weakly deciding vertices and then removes all the vertices that are not V'-reachable [LKL23, Section 6.2]. They did not explicitly state the time complexity of the lDFS algorithm, but they did evaluate their algorithm by implementing it in OCaml [LKL26]. Unfortunately their implementation does not offer to input custom instances nor do they print the graphs on which their algorithm was executed. This makes a meaningful comparison to our algorithm quite difficult. Léchenet, Kosmatov, and Le Gall [LKL23, Fig. 14] plotted the running time of their algorithm on randomly generated graphs with n vertices and $m = 1.5 \cdot n$ edges and a set V' that contains three randomly chosen vertices. Note that the graphs were generated by the function Rand.rand of OCamlGraph and each data point depicts the average running time of 10 randomly generated graphs, excluding instances for which $X = V'$ [LKL23, Section 7.2]. Also note that the graphs are not necessarily CFGs because no restriction on the maximum out-degree of a vertex was made. We ran our optimized algorithm on 20 randomly generated CFGs with n vertices and $p = 0.75$, because then in expectation every graph contains exactly $1.5 \cdot n$ edges, and plotted the results in Figure 6.13. We decided to plot each result individually because averaging the running times in this case would conceal the actual results. Observe that on some instances our algorithm terminates in a few milliseconds. This is caused by the fact that V' only contains three elements, which means in some cases the graphs contain many vertices that are not V'-reachable, and thus not processed in the main loop of our algorithm. Considering these outliers would significantly reduce the average running time.

But we can also observe that for the other instances our algorithm is quite fast, in most cases terminating in under 15 seconds, even for graphs with 500 thousand vertices. This is comparable to the algorithm of Léchenet, Kosmatov, and Le Gall that computes the weak control closure on a graph with 500 thousand vertices in approximately 17 seconds on average. The main observation we wanted to make with this comparison is that our implementation is also comparable with the implementation of the lDFS algorithm.

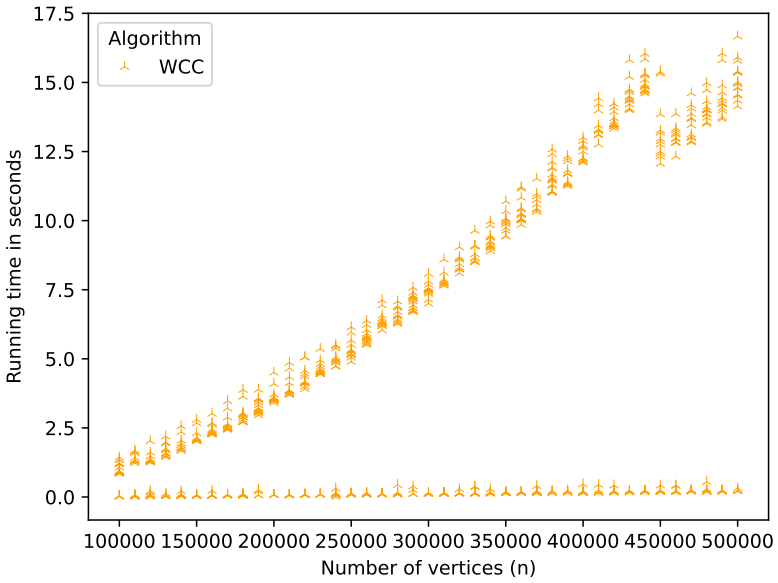

Figure 6.13: The running time of the WCC algorithm on randomly generated CFGs with $p = 0.75$ and $|V'| = 3$.

7. Weak and Strong Control Closure on General Graphs

Up until now we only considered CFGs where every vertex has only at most two outgoing edges. At first glance, this makes sense as branching instructions normally have only two branches. In reality, a program can also have branching instructions with more than two branches, for example consider switch-statements. Therefore, it makes sense to lift our algorithms to general graphs.

Definition 2.7 does not require the graph $G = (V, E)$ to be a CFG, because for $V' \subseteq V$ whether a vertex $v \in V$ is V'-weakly committing only depends on how many distinct vertices v can reach via V'-paths. Observe that in the algorithm described in Theorem 4.12 we did not explicitly use the fact that every vertex only has at most two outgoing edges, we only used the fact that G is a CFG to argue that because every edge is only processed once, the running time, excluding the time spent in the DSCC data structure D, is linear in $|V|$ as $|E| = \mathcal{O}(|V|)$ for a CFG. Let $n = |V|$ and $m = |E|$. If we lift the restriction that G must be a CFG the algorithm still computes the weak control closure of V', but with total running time $\mathcal{O}(n + m + T(n, m))$.

Theorem 7.1. *Let $G = (V, E)$ be a graph, $n = |V|$, $m = |E|$, and $V' \subseteq V$ a vertex set such that G only contains V'-reachable vertices. Given a DSCC data structure D with total update time $T(n, m)$, there exists an algorithm that computes the weak control closure X of V' in G in time $\mathcal{O}(n + m + T(n, m))$.*

By using Tarjan's DSCC data structure the total running time is then $\mathcal{O}(m(n + m))$, by Theorem 4.8. For acyclic graphs the total running time is even linear in $n + m$ because the initialization of Tarjan's DSCC data structure only takes time $\mathcal{O}(n + m)$. When using the DSCC data structure described by Bernstein, Gutenberg, and Wulff-Nilsen, the expected total running time is $\mathcal{O}(m \log(n)^4)$ by Theorem 5.16 if $m = \Omega(n)$.

Unlike the weak control closure, the strong control closure is only defined on CFGs in Definition 2.19. This is because final vertices are defined by their out-degree relative to the maximum out-degree of two in a CFG. Recall that, given the set of predicate vertices $P \subseteq V$ of G, a vertex $p \in P$ is final if it is incomplete, meaning $\deg^+(p) < 2$. If we lift the restriction that G must be a CFG, it is not necessarily correct to say that only predicate vertices with out-degree at most one are final vertices. The goal of incomplete predicates was to capture the difference between a branching instruction of a program where both

branches are depicted in the CFG, and one where only one branch is represented. When allowing arbitrarily many outgoing edges not every predicate vertex with two or more outgoing edges is necessarily complete. Consider for example a switch-statement with four branches, but in the CFG it is represented by a vertex with only three outgoing edges. Then this vertex should clearly be handled like an incomplete predicate, meaning a final vertex. Fortunately this problem is easily solved by partitioning the vertices of G into final vertices F and non-final vertices, instead of predicate vertices P and non-predicate vertices. We can modify Definition 2.17 and Definition 2.19 to use the final vertices F instead of predicate vertices P as follows.

Definition 7.2 (V'-strongly Committing Vertex)**.** *Let $G = (V, E)$ be a graph, $F \subseteq V$ the final vertices of G and $V' \subseteq V$. A vertex $v \in V$ is* V'-strongly committing *in G for F if and only if it is V'-weakly committing in G and all complete paths starting at v in G contain at least one vertex from V'.*

Definition 7.3 (Strongly Control Closed)**.** *Let $G = (V, E)$ be a graph and $F \subseteq V$ the final vertices of G. A set $V' \subseteq V$ is* strongly control closed *in G for F if and only if every V'-reachable vertex $v \in V \backslash V'$ is V'-strongly committing in G for F or V'-avoiding in G. A minimal strongly control closed set $X \supseteq V'$ is called a* strong control closure *of V' in G for F.*

Observe that the definitions are identical except for the set F replacing P because the distinction between predicate and non-predicate vertices only mattered when deciding which vertex is a final vertex, and by immediately passing F we get rid of this step. Just like our algorithm to compute the weak control closure, our algorithm to compute the strong control closure described in Theorem 4.20, excluding determining final vertices, never explicitly uses the fact that a vertex in a CFG only has two outgoing edges, we again only used it for determining the time complexity. Because we do not need to determine which vertex is a final vertex anymore, our algorithm also computes the strong control closure on general graphs.

Theorem 7.4. *Let $G = (V, E)$ be a graph, $n = |V|$, $m = |E|$, let $V' \subseteq V$ be a vertex set such that G only contains V'-reachable vertices and let $F \subseteq V$ be the final vertices of G. Given a DSCC data structure D with total update time $T(n, m)$, there exists an algorithm that computes the strong control closure X of V' in G for F in time $\mathcal{O}(n + m + T(n, m))$.*

Again, by using Tarjan's DSCC data structure the total running time is then $\mathcal{O}(m(n+m))$, and $\mathcal{O}(n + m)$ for acyclic graphs, and when using the DSCC data structure described by Bernstein, Gutenberg, and Wulff-Nilsen, the expected total running time is $\mathcal{O}(m \log(n)^4)$ if $m = \Omega(n)$.

We already gave the pseudocode for our algorithms in Algorithm 4.1 and Algorithm 4.2. The algorithm to compute the weak control closure does not need to be modified, and in Algorithm 4.2 we only need to replace the input set P with the set F. In the actual implementation one would need to modify the check in lines 10 and 28 to work with F instead of P.

8. Graphs with Bounded Tree-width

In this chapter we will briefly argue that for certain graphs even linear time algorithms to compute the weak and strong control closure are possible. For acyclic CFGs we have already described linear time algorithms in Chapter 3. Now we consider graphs with a bounded *tree-width* [RS84], which measures how similar a graph is to a tree.

Definition 8.1 (Tree Decomposition, Chapter 1 [RS84])**.** *Let $G = (V, E)$ be a graph. We call a tree T on nodes $X_1, ..., X_r$ a* tree decomposition *of G if the following hold:*

- $X_1 \cup ... \cup X_r = V$
- $(u, v) \in E \Rightarrow u, v \in X_i$ *for at least one* $i \in \{1, ..., r\}$.
- *For X_i, X_j and X_k with $i, j, k \in \{1, ..., r\}$ if X_j is on the path of T between X_i and X_k then $X_i \cap X_k \subseteq X_j$.*

The width of a tree decomposition is $\max\{|X_i|\} - 1$.

Definition 8.2 (Tree-width, Chapter 1 [RS84])**.** *The* tree-width *of a graph $G = (V, E)$ is the minimum width of a tree decomposition of G.*

In general, the tree-width of a graph can be arbitrarily large, even for a CFG where every vertex has at most two outgoing edges. But for graphs where the tree-width is bounded from above many problems can be solved efficiently. More precisely we consider problems that can be expressed using an extension of monadic second-order logic that also allows quantifications over sets of vertices and edges (MSO_2-logic). Such problems can be solved on graphs with bounded tree-width in linear time.

Theorem 8.3 (Courcelle's Theorem, Theorem 7.12 [Cyg+15])**.** *Let φ be an* MSO_2 *formula with p free monadic variables $X_1, \ldots, X_p$, and let $\alpha(x_1, \ldots, x_p)$ be an affine function. Let $G = (V, E)$ be a graph together with a tree decomposition of width t, and suppose values are given for all free variables of φ except $X_1, \ldots, X_p$.*

Let $n = |V|$, there exists an algorithm with running time $f(|\varphi|, t) \cdot n$ that computes values for $X_1, \ldots, X_p$ such that $\varphi(X_1, \ldots, X_p) = \top$ and $\alpha(|X_1|, \ldots, |X_p|)$ is maximized or minimized.

Courcelle's Theorem states for a graph $G = (V, E)$ with tree decomposition of width t and a MSO_2 formula φ with a single free monadic variable X that we can find the smallest

set satisfying φ in time $f(|\varphi|, t) \cdot n$ with $n = |V|$ by choosing the affine function $\alpha(x) = x$, which maps $|X|$ to itself. Still, we require a tree decomposition of G, but if the tree-width of G is bounded by k then we can compute a tree decomposition of width $t \leq k$ in time $f'(k) \cdot n$ [Cyg+15, Theorem 7.17].

Corollary 8.4. *Let φ be an MSO_2 formula with free monadic variable X, let $G = (V, E)$ be a graph with tree-width bounded from above by $k \in \mathbb{N}_+$, and suppose values are given for all free variables of φ except X.*

Let $n = |V|$, there exists an algorithm with running time $f(|\varphi|, k) \cdot n$ that computes the set X with minimum cardinality such that $\varphi(X) = \top$.

8.1 MSO_2 Formula for the Weak Control Closure

If we can give a fixed length MSO_2 formula that is satisfied if and only if a set $X \supseteq V'$ is weakly control closed in a graph G with bounded tree-width then by Corollary 8.4 there exists an algorithm that computes the minimal weakly control closed superset of V', which is the weak control closure, in time $\mathcal{O}(n)$. First, we need to be able to express whether a path between two vertices exists as a MSO_2 formula because then we can formulate reachability.

$$\textsc{Path}(s, t) = \forall P \subseteq V : \Big[s \in P \wedge \forall x \in P : \forall y \in V : (x, y) \in E \Rightarrow y \in P \Big] \Rightarrow t \in P.$$

Lemma 8.5. *Given a graph $G = (V, E)$ and vertices $s, t \in V$, then there exists a path from s to r in G if and only if $\textsc{Path}(s, t) = \top$.*

Proof. Assume there exists a path $s = p_1, ..., p_n = t$ with $n \in \mathbb{N}_+$ in G. Let $P \subseteq V$. If either $s \notin P$ or the formula $T(P) := \forall x \in P : \forall y \in V : (x, y) \in E \Rightarrow y \in P$ does not hold then the implication in the formula trivially holds. As such assume that both $s \in P$ and $T(P)$ hold. If $T(P) = \top$ then for every vertex $x \in P$ every neighbor along an outgoing edge in G is also contained in P. Then, because $s \in P$ it holds that $p_2 \in P$ because there exists the edge $(s = p_1, p_2)$ in E. Inductively, for every $i \in \{1, ..., n-1\}$ because $p_i \in P$ we also have that $p_{i+1} \in P$. Finally, because $t = p_n \in P$ the implication $(s \in P \wedge T(P) \Rightarrow t) \in P$ holds, and thus $\textsc{Path}(s, t) = \top$.

Assume there does not exist a path from s to t. Let P be the set of all the vertices that are reachable from s in G along outgoing edges. Clearly $t \notin P$ because otherwise there would be a path from s to t. We have that $s \in P$ because s can reach itself via the trivial path s. Now assume that $T(P) = \bot$ then there exists an edge $(x, y) \in E$ with $x \in P$ but $y \notin P$. Because $x \in P$ the vertex x can be reached by s, and because the edge $(x, y) \in E$ exists s can also reach y, but then $y \in P$, which is a contradiction. As such $T(P) = \top$. This means that the implication $(s \in P \wedge T(P)) \Rightarrow t \in P$ does not hold, and $\textsc{Path}(s, t) = \bot$. □

Now that we can express whether a path between two vertices exists, it is easy to express a vertex v being V'-reachable. We know that v is V'-reachable if and only if there exists a vertex u in V' that can reach v, meaning there exists a path from u to v. The following formula expresses exactly this, but for an arbitrary set $X \subseteq V$.

$$\textsc{Reachable}(v, X) = \exists u \in X : \textsc{Path}(u, v)$$

We also need to be able to express whether a vertex $v \in V$ lies on an X-path. Recall that v lying on an X-path is equivalent to a vertex $x \in X$ being reachable from v in $G - E_G^{\text{out}}(X)$.

As such, we extend the formula that expresses if a path between two vertices exists to exclude certain edges.

$$\begin{aligned}\text{RPath}(s,t,X) = \forall P \subseteq V : \Big[& s \in P \wedge \\ & \forall x \in P : \forall y \in V : \Big((x,y) \in E \wedge x \notin X \Big) \Rightarrow y \in P \Big] \Rightarrow t \in P.\end{aligned}$$

Lemma 8.6. *Given a graph $G = (V, E)$, vertices $s, t \in V$ and a vertex set $X \subseteq V$, then there exists a path from s to t in $G - E_G^{\text{out}}(X)$ if and only if $RPath(s, t, X) = \top$.*

Proof. For an edge $(x, y) \in E$, the formula $(x, y) \in E \wedge x \notin X$ is equivalent to $(x, y) \in (E \backslash E_G^{\text{out}}(X))$. Therefore, the formula $RPath(s, t, X)$ is equivalent to $Path(s, t)$ for the graph $G - E_G^{\text{out}}(X)$. This means that $RPath(s, t, X) = \top$ if and only if $Path(s, t) = \top$ for $G - E_G^{\text{out}}(X)$, which is the case if and only if there is a path from s to t in $G - E_G^{\text{out}}(X)$ by Lemma 8.5. □

By Definition 2.6 a vertex $v \in X$ is X-weakly committing if and only if v can reach at most one distinct vertex contained in X via an X-path, which is equivalent to at most one vertex being reachable by v in $G - E_G^{\text{out}}(X)$.

$$\begin{aligned}\text{WC}(v,X) = & \forall x \in X : \forall y \in X : \\ & \Big[\text{RPath}(v,x,V) \wedge \text{RPath}(v,y,V) \Big] \Rightarrow x = y\end{aligned}$$

Lemma 8.7. *Given a graph $G = (V, E)$, a vertex $v \in V$ and a vertex set $X \subseteq V$, then v is X-weakly committing in G if and only if $WC(v, X) = \top$.*

Proof. Assume v is X-weakly committing. Then there is at most one vertex $x \in X$ such that $\text{RPath}(v, x, X) = \top$ by Lemma 8.6. Clearly it then holds that $\text{WC}(v, X) = \top$.

Now assume that v is not X-weakly committing. Then there are two vertices $x, y \in X$ such that $x \neq y$ and v can reach both x and y via an X-path. Then, by Lemma 8.6 we have that $\text{RPath}(v, x, X) = \top$ and $\text{RPath}(v, y, X) = \top$. Therefore, the implication $\Big[\text{RPath}(v, x, V) \wedge \text{RPath}(v, y, V)\Big] \Rightarrow x = y$ does not hold and thus $\text{WC}(v, X) = \bot$. □

Finally, we can express if a set $X \subseteq V$ is a weakly control closed superset of V'.

$$\begin{aligned}\text{WCCSupSet}(X,V') = & \forall v \in V' : v \in X \wedge \\ & \forall v \in V : [v \in X \vee \neg \text{Reachable}(v,X) \vee \text{WC}(v,X)]\end{aligned}$$

Lemma 8.8. *Given a graph $G = (V, E)$ and two vertex sets $V', X \subseteq V$ then X is a weakly control closed superset of V' in G if and only if $WCCSupSet(X, V') = \top$.*

Proof. Assume X is a weakly control closed superset of V'. Then clearly the formula $\forall v \in V' : v \in X$ holds. If $v \in V$ is not X-reachable then the formula $\neg\text{Reachable}(v, X)$ is satisfied. Further, because X is weakly control closed, every X-reachable vertex $v \in V \backslash X$ is X-weakly committing. By Lemma 8.7 the formula $\text{WC}(v, X)$ holds. Combined, the formula $\forall v \in V : [v \in X \vee \neg\text{Reachable}(v, X) \vee \text{WC}(v, X)]$ also holds. Therefore, $\text{WCCSupSet}(X, V') = \top$.

Now assume X is not a weakly control closed superset of V'. If $X \not\supseteq V'$ then $\forall v \in V' : v \in X$ does not hold and $\text{WCCSupSet}(X, V') = \bot$. On the other hand, if $X \supseteq V'$ then there must exist an X-reachable vertex $v \in V \backslash X$ that is not X-weakly committing. By Lemma 8.7 we then have $\text{WC}(v, X) = \bot$. Because v is X-reachable we additionally have $\text{Reachable}(v, X) = \top$, and because $v \notin X$ the formula $\forall v \in V : [v \in X \vee \neg\text{Reachable}(v, X) \vee \text{WC}(v, X)]$ does not hold. Thus, $\text{WCCSupSet}(X, V') = \bot$. □

By applying Corollary 8.4 we can compute a minimal set $X \subseteq V$ such that WCCSupSet(X, V') is satisfied. By Lemma 8.8 the set X is a superset of V' and weakly control closed, which means, because X is minimal, that X is the weak control closure of V' in G. Because the MSO_2 formula WCCSupSet has a constant length independent of G or V' the time needed to compute X is $\mathcal{O}(f(t) \cdot n)$ where $n = |V|$ and t is the tree-width of G. If the tree-width is bounded by a constant then it is possible to compute X in time $\mathcal{O}(n)$.

Corollary 8.9. *Given a graph $G = (V, E)$ with tree-width bounded from above by a constant and a vertex set $V' \subseteq V$, there exists an algorithm that computes the weak control closure of V' in G in time $\mathcal{O}(|V|)$.*

Note that while Corollary 8.9 is a nice theoretical result, the actual running time of the algorithm may be very bad, even worse than the simple algorithm described by Danicic et al. for most practical instances, because the constant hidden by the $\mathcal{O}$-notation may be very large.

8.2 MSO_2 Formula for the Strong Control Closure

Naturally we also want to give a fixed length MSO_2 formula for a set $X \subseteq V$ being a strongly control closed superset of V' in G for predicate vertices $P \subseteq V$. First, we need to express a vertex $v \in V$ being a final vertex. Recall that v is final if and only if v is a predicate vertex with $\deg^+(v) < 2$ or v is a non-predicate vertex with no outgoing edges.

$$\begin{aligned}\text{Final}(v, P) = &\Big[v \in P \wedge \forall x \in V : \forall y \in V : \Big((v, x) \in E \wedge (v, y) \in E\Big) \Rightarrow x = y\Big] \vee \\ &\Big[v \notin P \wedge \forall w \in V : (v, w) \notin E\Big]\end{aligned}$$

Lemma 8.10. *Given a CFG $G = (V, E)$, the predicate vertices $P \subseteq V$ of G and a vertex $v \in V$, then v is a final vertex if and only if $\text{Final}(v, P) = \top$.*

Proof. Assume $v \in P$ with $\deg^+(v) < 2$, which means v is a final vertex. Then v only has at most one outgoing edge. Then the formula

$$\forall x \in V : \forall y \in V : \Big((v, x) \in E \wedge (v, y) \in E\Big) \Rightarrow x = y$$

must hold, and $\text{Final}(v, P)$ is satisfied.

On the other hand assume that $v \in P$ but $\deg^+(v) = 2$, then v is not a final vertex because it has two outgoing edges $(v, x), (v, y) \in E$. As such, the formula

$$\forall x \in V : \forall y \in V : \Big((v, x) \in E \wedge (v, y) \in E\Big) \Rightarrow x = y$$

does not hold, and neither does $\text{Final}(v, P)$.

Assume v is a non-predicate vertex with $\deg^+(v) = 0$, which also means that v is a final vertex. Then $v \notin P$ and for all $w \in V$ we have $(v, w) \notin E$ because v does not have an outgoing edge. As such $v \notin P \wedge \forall w \in V : (v, w) \notin E = \top$ and $\textsc{Final}(v, P)$ is satisfied.

Finally, assume that v is a non-predicate vertex with a single outgoing edge. Then v is not a final vertex, and the formula $\forall w \in V : (v, w) \notin E$ is false because v has an outgoing edge. Thus, $\textsc{Final}(v, P) = \bot$. □

$$\begin{aligned}\mathrm{SC}(v, X, P) =& \mathrm{WC}(v, X) \wedge \\ & \forall f \in V : \Big(\textsc{Final}(f, P) \wedge f \notin X\Big) \Rightarrow \neg \mathrm{RPath}(v, f, X) \wedge \\ & \forall x \in V : \forall y \in V : \\ & \quad \Big(\mathrm{RPath}(x, y, X) \wedge \mathrm{RPath}(y, x, X)\Big) \Rightarrow \neg \mathrm{RPath}(v, x, X)\end{aligned}$$

Lemma 8.11. *Given a CFG $G = (V, E)$, the predicate vertices $P \subseteq V$ of G, a vertex $v \in V$ and a vertex set $X \subseteq V$, then v is X-strongly committing in G for P if and only if $SC(v, X, P) = \top$.*

Proof. By Definition 2.17 vertex v is X-strongly committing if and only if it is X-weakly committing and every complete path that starts at v contains at least one vertex from X. First assume that v is X-strongly committing. Then, v is also X-weakly committing and by Lemma 8.7 it holds that $\mathrm{WC}(v, X) = \top$.

Let $T(f) = \Big(\textsc{Final}(f, P) \wedge f \notin X\Big) \Rightarrow \neg \mathrm{RPath}(v, f, X)$. We first show that $T(f) = \top$ for every $f \in V$ if v is X-strongly committing. If f is not a final vertex then by Lemma 8.10 it holds that $\textsc{Final}(f, P) = \bot$. Therefore, if f is not a final vertex or $f \in X$ then $T(f) = \top$ trivially holds. Otherwise, meaning f is a final vertex and $f \notin X$, consider a path $v, ..., f$. This path is a complete path and, because v is X-strongly committing, contains a vertex from X. Because $f \notin X$, this vertex is an intermediary vertex on the path. Consequently, $v, ..., f$ is not a path in $G - E_G^{\text{out}}(X)$. Because $v, ..., f$ is arbitrary by Lemma 8.6 we have that $\mathrm{RPath}(v, f, X) = \bot$, and thus $T(f) = \top$.

Let $x, y \in V$ such that $\mathrm{RPath}(x, y, X) \wedge \mathrm{RPath}(y, x, X)$ holds. Then by Lemma 8.6 there exists a path from x to y and vice versa in $G - E_G^{\text{out}}(X)$, which means there exists an X-free complete path between x and y. Because v is X-strongly committing it cannot lie on an X-free complete path, and therefore the path from v to x, if it exists, must contain a vertex from X. Thus, by Lemma 8.6 it holds that $\mathrm{RPath}(v, x, X) = \bot$. Combining this with $\mathrm{WC}(v, X) = \top$ and $T(f) = \top$ we get that $\mathrm{SC}(v, X, P) = \top$.

Now assume that v is not X-strongly committing. If v is also not X-weakly committing then by Lemma 8.7 we have $\mathrm{WC}(v, X) = \bot$, and as such $\mathrm{SC}(v, X, P) = \bot$. If v lies on an X-free complete path that ends in a final vertex f then by Lemma 8.6 we have $\mathrm{RPath}(v, f, X) = \top$. Because $f \notin X$ and, by Lemma 8.10, we have $\textsc{Final}\ (f, P) = \top$ it holds that $T(f) = \bot$. Thus, $\mathrm{SC}(v, X, P) = \bot$. Finally, if v lies on an X-free infinite path then there is a cycle in that path between two vertices x and y. For those vertices, by Lemma 8.6, we have that $\mathrm{RPath}(x, y, X) = \mathrm{RPath}(y, x, X) = \top$. And because v lies on that same X-free infinite path it can reach x in $G - E_G^{\text{out}}(X)$, which means, again by Lemma 8.6, that $\mathrm{RPath}(v, x, X) = \top$. Consequently, also in this case $\mathrm{SC}(v, X, P) = \bot$. □

Recall that by Definition 2.19 not every vertex $v \in V \backslash X$ must be X-strongly committing for X to be strongly control closed. If v is X-avoiding then it does not matter whether it is

X-strongly committing. Expressing whether v is X-avoiding is very simple because it just means that there does not exist a vertex $x \in X$ such that there exists a path from v to x.

$$\textsc{XAvoid}(v, X) = \forall x \in X : \neg\textsc{Path}(v, x)$$

$$\begin{aligned}\textsc{SCCSupSet}(X, V', P) =& \forall v \in V' : v \in X \wedge \\ & \forall v \in V : \Big[\\ & \quad v \in X \vee \\ & \quad \neg\textsc{Reachable}(v, X) \vee \\ & \quad \text{SC}(v, X, P) \vee \\ & \quad \textsc{XAvoid}(v, X)\Big]\end{aligned}$$

Lemma 8.12. *Given a CFG $G = (V, E)$, the predicate vertices $P \subseteq V$ of G and two vertex sets $V', X \subseteq V$, then X is a strongly control closed superset of V' in G for P if and only if $\textsc{SCCSupSet}(X, V', P) = \top$.*

Proof. Assume X is a strongly control closed superset of V'. Then the formula $\forall v \in V' : v \in X$ holds. We need to show that for every vertex $v \in V$ one of the following holds:

- $v \in X$
- $\neg\textsc{Reachable}(v, X)$
- $\text{SC}(v, X, P)$
- $\textsc{XAvoid}(v, X)$

If $v \in X$ then the first condition holds so consider a vertex $v \in V \backslash X$. If v is not X-reachable, then $\textsc{Reachable}(v, X) = \bot$, and as such the second condition is satisfied. Otherwise, meaning v is X-reachable, then, because X is strongly control closed, v must either be X-strongly committing or X-avoiding. This means that either $\text{SC}(v, X, P)$, by Lemma 8.11, or $\textsc{XAvoid}(v, X)$ hold. Therefore, $\textsc{SCCSupSet}(X, V', P) = \top$.

Now assume X is not a strongly control closed superset of V'. If $X \not\supseteq V'$ then $\forall v \in V' : v \in X$ does not hold and $\textsc{SCCSupSet}(X, V', P) = \bot$. On the other hand, if $X \supseteq V'$ then there must exist an X-reachable vertex $v \in V \backslash X$ that is not X-strongly committing and not X-avoiding. By Lemma 8.11 we then have $\text{SC}(v, X, P) = \bot$. Because v is X-reachable, we additionally have $\textsc{Reachable}(v, X) = \top$, and because v is also not X-avoiding it also holds that $\textsc{XAvoid}(v, X) = \bot$. Finally, because $v \notin X$, all four of the previously mentioned conditions are not satisfied and as such $\textsc{SCCSupSet}(X, V', P) = \bot$. □

Just like with the weak control closure we can apply Corollary 8.4 to compute a minimal set X such that $\textsc{SCCSupSet}(X, V', P)$ is satisfied. By Lemma 8.12 the vertex set X is a superset of V' and strongly control closed in G for P. Therefore, because X is minimal, it is the strong control closure of V' in G for P. And because the length of the formula SCCSupSet is constant, independent of G, V' or P, the time needed to compute X is $\mathcal{O}(f(t) \cdot n)$ where $n = |V|$ and t is the tree-width of G.

Corollary 8.13. *Given a CFG $G = (V, E)$ with tree-width bounded from above by a constant, the predicate vertices $P \subseteq V$ and a vertex set $V' \subseteq V$, there exists an algorithm that computes the strong control closure of V' in G for P in time $\mathcal{O}(|V|)$.*

Note that currently we require G to be a CFG, but we can generalize the results by replacing the predicate vertices P with the final vertices $F \subseteq V$ as defined in Definition 7.2. The generalized formula SC becomes even simpler because we do not need the formula FINAL anymore. The proof of Lemma 8.14 is nearly identical to the proof of Lemma 8.11, which is why we do not repeat it.

$$\begin{aligned}\mathrm{SC}(v, X, F) =& \mathrm{WC}(v, X) \wedge \\ & \forall f \in F : f \notin X \Rightarrow \neg \mathrm{RPATH}(v, f, X) \wedge \\ & \forall x \in V : \forall y \in V : \\ & \quad \Big(\mathrm{RPATH}(x, y, X) \wedge \mathrm{RPATH}(y, x, X)\Big) \Rightarrow \neg \mathrm{RPATH}(v, x, X)\end{aligned}$$

Lemma 8.14. *Given a graph $G = (V, E)$, the final vertices $F \subseteq V$ of G, a vertex $v \in V$ and a vertex set $X \subseteq V$, then v is X-strongly committing in G for F if and only if $SC(v, X, F) = \top$.*

The only change to the formula SCCSUPSET needed is that it now takes F instead of P as an argument because it just delegates the third input to the formula SC.

Lemma 8.15. *Given a graph $G = (V, E)$, the final vertices $F \subseteq V$ of G and two vertex sets $V', X \subseteq V$, then X is a strongly control closed superset of V' in G for F if and only if $SCCSUPSET(X, V', F) = \top$.*

Corollary 8.16. *Given a graph $G = (V, E)$ with tree-width bounded from above by a constant, the final vertices $F \subseteq V$ and a vertex set $V' \subseteq V$, there exists an algorithm that computes the strong control closure of V' in G for F in time $\mathcal{O}(|V|)$.*

8.3 CFGs from Programs

One class of graphs that have bounded tree-width are acyclic graphs, which are trees or forests, because they have tree-width of one [RS84]. So for acyclic CFGs there exists by Corollary 8.9 and Corollary 8.13 linear time algorithms to compute the weak and strong control closure. But this is nothing new as we have already given linear time algorithms in Chapter 3. More interesting are CFGs that were constructed by actual programs and may contain cycles. To translate a program to a graph we can create for each instruction a vertex, and add an edge (u, v) between two vertices u and v if the instructions represented by u and v are executed consecutively [KLS20, Chapter 1]. In other words, if instruction v can be executed immediately after instruction u we add a directed edge from u to v. Note that this graph is not necessarily a CFG by Definition 2.4 because by this construction a vertex can have arbitrarily many outgoing edges due to, for example, switch-statements. But if the switch-statements contain an unconditional break-statement in each non-empty clause, meaning there is no fall-through from one clause that executes code to another, then it can be converted to an if-else-tree, which then can be translated to a CFG. But this restriction is not even necessary because both Corollary 8.9 and Corollary 8.16 are applicable to general graphs.

For $l \in \mathbb{N}_+$ Krause, Larisch, and Salfelder [KLS20, Definition 5] called a program *l-structured* if the graph created from the program has tree-width l. Krause, Larisch, and Salfelder [KLS20, Chapter 4] then showed that C-programs, when adhering to the MISRA-C:2004 [MIS08] coding standards are 5-structured, which means every graph translated from such a program has tree-width at most 5. Important restrictions that the MISRA-C:2004

coding standards impose are that no goto-statements [MIS08, Rule 14.4] and no continue-statements [MIS08, Rule 14.5] are allowed. This severely restricts the complexity of the control flow, which results in a low tree-width for the graph translated from the program. Consequently, by Corollary 8.9 and Corollary 8.16, there exist linear time algorithms to compute the weak and strong control closure on graphs created from C-programs adhering to the MISRA-C:2004 coding standards.

Corollary 8.17. *Given a graph $G = (V, E)$ that was created from a C-program that adheres to the MISRA-C:2004 coding standards, the final vertices $F \subseteq V$ and a vertex set $V' \subseteq V$, there exist both an algorithm that computes the weak control closure of V' in G, and an algorithm that computes the strong control closure of V' in G for F with time complexity $\mathcal{O}(|V|)$.*

We can also construct graphs from Java programs. Unlike C, Java does not support goto-statements, so for Java programs we can drop the restriction that no goto-statements must be used. Unfortunately, Java supports labelling loops which can then be used as break- and continue-targets. By using labelled break- and continue-statements we can construct l-structured programs for arbitrary $l \in \mathbb{N}_+$. But if we forbid the use of labels then Java programs are 6-structured [GMT02, Chapter 3]. Thus, every graph translated from such a Java program has tree-width at most 6.

Corollary 8.18. *Given a graph $G = (V, E)$ that was created from a Java program that does not contain any labels, the final vertices $F \subseteq V$ and a vertex set $V' \subseteq V$, there exist both an algorithm that computes the weak control closure of V' in G, and an algorithm that computes the strong control closure of V' in G for F with time complexity $\mathcal{O}(|V|)$.*

8.4 Graphs with Bounded Clique-width

Observe that none of the MSO_2 formulas given in Section 8.1 or Section 8.2 use a quantification over edge set E. For problems expressible in monadic second-order logic that allows quantification over vertex sets (MSO_1-logic), but not edge sets, even more general results than for MSO_2 formulas exist.

The *clique-width* [CMR70, Definition 28] is another parameter that describes the complexity of a graph, similar to tree-width. But it is an even more general property, as every graph with bounded tree-width also has bounded clique-width [CR05, Theorem 1.2], but not every graph with bounded clique-width has also a bounded tree-width, as the tree-width of a clique with n vertices is $n - 1$, while the clique-width is at most 2 [CR05]. Thus, the class of graphs that have bounded clique-width is larger than the class of graphs that have bounded tree-width.

A computationally more useful parameter that is similar to clique-width is *rank-width* [Cyg+15, Section 7.9]. It is similar to clique-width in the sense that the rank-width is bounded both from above and below by a function of the clique-width. For rank-width there exists an analogous theorem to Theorem 8.3 for MSO_1 formulas on undirected graphs [Cyg+15, Section 7.9]. This does not directly give us linear time algorithms because we would first need to investigate if the result also holds for directed graphs, and we still need to compute a rank-decomposition, which is the analog to a tree decomposition. However, it suggests that it may be worth to investigate in future work.

9. Conclusion

In this thesis we presented new algorithms to compute both the weak and strong control closure. For acyclic CFGs we described simple linear time algorithms that utilized a topological ordering of vertices. For CFGs in general we presented algorithms that work with any DSCC data structure. Using a naive DSCC data structure that recomputes the SCCs after each update, our algorithm to compute the weak control closure matches the worst case quadratic time complexity with the state-of-the-art algorithm. Our algorithm to compute the strong control closure also achieves a quadratic time complexity, which is a one-degree improvement over the time complexity of the to our knowledge state-of-the-art algorithm. We then illustrated how to extend an existing data structure that maintains SCCs under edge deletions to fit the needed requirements of our algorithms, which improved our algorithms to have a near-linear expected asymptotic running time. Additionally, our algorithms can easily be generalized to work on any graph, not just CFGs, for appropriate generalizations of the weak and strong control closure.

We also implemented the presented algorithms and compared them to existing algorithms, and could observe that our algorithms computed both the weak and strong control closure faster than existing algorithms on both randomly generated CFGs and graphs generated from existing C-programs. Additionally, we observed that our algorithms performed best when using a naive implementation of the DSCC data structure as the running time appears to only grow linearly in many practical cases.

For future research it presents itself to implement more DSCC data structures and compare them. Currently, our algorithms are implemented in Java, which is an interpreted language, so it also may be beneficial to translate our implementation to compiled programming language such as C++ because compiled programs are significantly faster in general. Another implementation specific research topic would be to optimize our implementation of the DSCC data structure described by Bernstein, Gutenberg, and Wulff-Nilsen and compare it to Tarjan's DSCC data structure on even larger graphs because eventually it should be faster.

Another arguably more interesting research topic would be to employ our algorithms in existing programs where currently the algorithms by Danicic et al. or Masud or Léchenet, Kosmatov, and Le Gall are used, and then investigate if the use of our algorithms benefits the performance. This would require to implement our algorithms in different programming languages and therefore would allow making a more meaningful comparison to the algorithms given by Masud or Léchenet, Kosmatov, and Le Gall.

We also briefly argued using Courcelle's Theorem that both the weak and strong control closure can be computed in linear time on graphs with bounded tree-width. In theory this is a nice result, but algorithms obtained this way are often impractical as the constant factor hidden by the $\mathcal{O}$-notation can be quite large. This naturally raises the question of how large that constant would be for the algorithms induced by our MSO_2 formulas because it may be the case that it is sufficiently small, and the algorithm may even be practically relevant.

Bibliography

[All70] Frances E. Allen. "Control flow analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[BGW23] Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. "Decremental Strongly Connected Components and Single-Source Reachability in Near-Linear Time". In: *SIAM Journal on Computing* 52.2 (2023), STOC19-128-STOC19–155. DOI: 10.1137/20M1312149.

[BLK18] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. "SPEC CPU2017: Next-Generation Compute Benchmark". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 41–42. ISBN: 9781450356299. DOI: 10.1145/3185768.3185771.

[CMR70] Bruno Courcelle, Johann Makowsky, and Udi Rotics. "Linear Time Solvable Optimization Problems on Graphs of Bounded Clique Width". In: vol. 33. Jan. 1970, pp. 287–291. ISBN: 978-3-540-65195-6. DOI: 10.1007/10692760_1.

[CR05] Derek G. Corneil and Udi Rotics. "On the Relationship Between Clique-Width and Treewidth". In: *SIAM Journal on Computing* 34.4 (2005), pp. 825–847. DOI: 10.1137/S0097539701385351.

[Cyg+15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. 1st. Springer Publishing Company, Incorporated, 2015. DOI: 10.1007/978-3-319-21275-3.

[Dan+11] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John D. Howroyd, Ákos Kiss, and Michael R. Laurence. "A unifying theory of control dependence and its application to arbitrary program structures". In: *Theoretical Computer Science* 412.49 (2011), pp. 6809–6842. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2011.08.033.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041.

[GMT02] Jens Gustedt, Ole Mæhle, and Jan Telle. "The Treewidth of Java Programs". In: vol. 2409. Jan. 2002, pp. 86–97. ISBN: 978-3-540-43977-6. DOI: 10.1007/3-540-45643-0_7.

[Hin26] Manuel Hinkel. *Weak-and-Strong-Control-Closure*. https://github.com/ManuelHinkel/Weak-and-Strong-Control-Closure. GitHub repository. Accessed: 6 July 2026. 2026.

[KLS20] Philipp Klaus Krause, Lukas Larisch, and Felix Salfelder. "The tree-width of C". In: *Discrete Applied Mathematics* 278 (2020). Eighth Workshop on Graph Classes, Optimization, and Width Parameters, pp. 136–152. DOI: 10.1016/j.dam.2019.01.027.

[LKL23] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. "Efficient computation of arbitrary control dependencies". In: *Theoretical Computer Science* 969 (2023), p. 114029. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2023.114029.

[LKL26] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. *ldfs*. https://gitlab.inria.fr/jlechene/ldfs/-/tree/TCS/. Gitlab repository. Accessed: 12 July 2026. 2026.

[Mas22] Abu Naser Masud. "Efficient computation of minimal weak and strong control closure". In: *Journal of Systems and Software* 184 (2022), p. 111140. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111140.

[Mas26] Abu Naser Masud. *CDA*. https://github.com/anm-spa/CDA. GitHub repository. Accessed: 6 July 2026. 2026.

[MIS08] MISRA. *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004, 2nd Edition)*. Technical report. MISRA, 2008.

[RS84] Neil Robertson and P.D Seymour. "Graph minors. III. Planar tree-width". In: *Journal of Combinatorial Theory, Series B* 36.1 (1984), pp. 49–64. ISSN: 0095-8956. DOI: 10.1016/0095-8956(84)90013-3.

[SE81] Yossi Shiloach and Shimon Even. "An On-Line Edge-Deletion Problem". In: *J. ACM* 28.1 (Jan. 1981), pp. 1–4. DOI: 10.1145/322234.322235.

[Tar72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[Wei81] Mark Weiser. "Program slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. DOI: 10.5555/800078.802557.

Appendix

A Evaluation

Benchmark	File	Method	V'
GCC	c-typeck.c	build_binary_op	1, 276, 465
GCC	sel-sched-dump.c	dump_insn_rtx_1	3, 4, 9, 14, 23, 40
GCC	reload1.c	choose_reload_regs	1, 128, 132
Perlbench	pp_pack.c	S_unpack_rec	6, 8, 10, 19, 20, 21, 29, 38, 39, 40, 42, 44, 58, 62, 63, 70, 77, 83, 84, 110, 111, 116, 127, 132, 136, 151, 152, 157, 160, 171, 184, 194, 208, 227, 228, 262, 263, 265, 266, 289, 291, 298, 299, 305, 318, 325, 332, 342, 349, 356, 361, 362, 368, 379, 381, 382, 386, 390, 396, 423, 424, 439, 442, 457, 460, 470, 476, 483, 484, 491, 499, 503, 507, 508, 515, 518, 526, 527, 529, 532, 539, 549, 550, 555, 557, 569, 572, 573, 575, 585, 588, 591, 592, 610, 611, 622, 625, 626, 633, 634, 637, 642, 650, 651, 667, 671, 691, 692, 698, 704, 716, 720, 731, 738, 739, 762, 780, 788, 794, 798, 824, 837, 845, 849, 850, 854, 885, 891, 896, 897, 916, 924, 927, 933, 953, 954, 956, 961

Perlbench	regcomp.c	S_regclass	5, 9, 12, 16, 30, 42, 48, 69, 74, 77, 102, 103, 111, 114, 121, 123, 126, 133, 136, 137, 139, 141, 154, 160, 167, 171, 179, 184, 202, 204, 210, 219, 225, 226, 236, 237, 253, 254, 256, 260, 262, 286, 294, 297, 299, 306, 328, 330, 334, 351, 360, 361, 362, 377, 382, 386, 398, 403, 404, 406, 407, 458, 497, 498, 504, 505, 511, 513, 525, 531, 532, 536, 545, 548, 549, 552, 557, 571, 573, 575, 580, 586, 600, 601, 603, 606, 623, 627, 628, 634, 635, 641, 644, 653, 654, 667, 694, 695, 700, 714, 718, 720, 721, 748
Perlbench	sv.c	S_sv_display	1, 3, 5, 6, 8, 12, 16, 23

Table A.1: Instances excluded from benchmarks.

```
1  void generated_cfg(
2      int cond_1,    int cond_2,    int cond_3,    int cond_4)
3  {
4  B1:
5      if(cond_1)
6          goto B3;
7      else
8          goto B2;
9  B2:
10     goto B1;
11 B3:
12     if(cond_3)
13         goto B2;
14     else
15         goto B4;
16 B4:
17     return;
18 }
```

Figure A.1: A C-file generated from a randomly generated CFG with parameters $n = 4$ and $p = 0.5$.