STATISTICS, OPTIMIZATION AND INFORMATION COMPUTING
Stat., Optim. Inf. Comput., Vol. **16**, July 2026, pp 749–776.
Published online in International Academic Press (www.IAPress.org)

Graph Optimization and Programming through Interval Reduction: Case for Secure Control Flow Graph Analysis

Meryem HAFID [1,*], Seddik ABDELALIM[2], Driss KARIM [1], Ilias ELMOUKI [3]

[1]*Laboratory of mathematics, computer science and applications (LMCSA), Faculty of Sciences and Techniques of Mohammedia (FSTM), University Hassan II of Casablanca, Casablanca, Morocco*
[2]*Laboratory of Mathematical Analysis, Algebra and Applications (LAM2A), Faculty of Sciences Aïn Chock (FSAC), University Hassan II of Casablanca, Casablanca, Morocco*
[3]*MoNum, EHTP, Casablanca, Morocco*

Abstract Control Flow Graphs (CFGs) provide a central representation for reasoning about program structure, but their increasing size can limit both manual interpretation and automated analysis. This paper revisits interval-based reduction as a structurally grounded approach to CFG simplification. Based on classical notions of dominance, intervals, and reducibility, we reformulate the reduction process in an explicit algorithmic form and study the structural properties preserved by the resulting derived graphs. The proposed approach is implemented and evaluated on a diverse collection of CFGs, with comparison to Sequential Node Merging as a baseline reduction technique. The results indicate that interval reduction offers a principled way to reduce graph complexity more effectively, while Sequential Node Merging may better preserve the hierarchical and semantic organization of control flow, thereby both supporting its use in modern program analysis settings, including software security and reliability. Our results guarantee a formal basis for future malware-detection frameworks built upon interval-reduced control-flow graphs.

Keywords Graph theory, Optimization, Control flow graph, Interval reduction, Sequential node merging, Complexity.

AMS 2010 subject classifications 68R10, 68Q25, 90C35

DOI: 10.19139/soic-2310-5070-3253

1. Introduction

Graph-based representations have become increasingly important in modern security-oriented computing, including blockchain analysis, structural dependency analysis, and software verification, highlighting the broader applicability of graph-theoretic techniques in cybersecurity contexts [1, 2].

The concept of control flow analysis and interval reduction draws on classical compiler optimization literature, dating back to the foundational work of Frances Allen [5, 6] and the detailed exposition by Hecht [18]. By representing a program's execution flow as nodes (basic blocks) and directed edges, Control Flow Graphs (CFGs) clarify the structural relations among branching and looping constructs [12]. Such clarity is pivotal in optimization tasks, which range from code motion to loop transformations, as well as in verification, testing, and fault localization [8, 17]. However, large-scale CFGs can become excessively large [10], impeding both automated algorithms and manual comprehension. This complexity poses acute difficulties in malware detection, as understanding control-flow patterns is crucial for identifying malicious behaviors that evade signature-based detection [9, 13, 22, 23]. Modern adversaries use polymorphism, metamorphism, and dynamic code manipulation

*Correspondence to: Meryem Hafid (Email: Meryemh553@gmail.com). Laboratory of mathematics, computer science and applications (LMCSA), Faculty of Sciences and Techniques of Mohammedia (FSTM), University Hassan II of Casablanca, Casablanca, Morocco

ISSN 2310-5070 (online) ISSN 2311-004X (print)

to complicate static analysis [3, 30], often inflating the CFG to thousands of nodes and edges. The resulting complexity hinders near-real-time analysis and detection accuracy [32]. Similarly, such complexity affects software defect detection, as locating bugs—particularly those involving conditions or execution sequences—becomes increasingly difficult in deeply nested or tangled control flows [11, 20].

To address the challenges posed by large-scale CFGs, one promising remedy is graph reduction: collapsing multiple nodes into single intervals, or more generally, merging subgraphs while preserving essential path properties [24, 26]. Paige's early partitioning scheme [24] laid down the path-preserving approach, which subsequent authors refined for single-function CFGs [14] and interprocedural frameworks [12, 27]. Gold [15, 16] unified these ideas, discussing the composition of different reductions and the identification of "decision nodes" (with outdegree ≥ 2). Stepwise or iterative reductions enable engineers to simplify the CFG until it is manageable, ensuring that critical control-flow paths are preserved. Meanwhile, to enhance malware detection, advanced analysis frameworks [21] have increasingly adopted control flow graph (CFG) representations alongside machine learning and deep neural networks, such as Graph Neural Networks (GNNs) [13]. These models excel at learning from the structural properties of programs, enabling effective classification of malicious samples and achieving high detection rates. Moreover, explainable AI methods like GNNExplainer [31] help illuminate which control-flow fragments contribute to a model's prediction, improving the transparency of automated decisions. Beyond malware, CFG-based learning approaches are also proving effective for detecting software vulnerabilities, such as buffer overflows, privilege escalation, and logic flaws in conditional branches [11, 30]. These vulnerabilities often reside in complex or deeply nested control paths, making traditional static analysis insufficient.

Recent techniques using static and graph-based representations—such as program slicing and code property graphs—improve precision by extracting vulnerability-specific features and comparing against patch structures to reduce false positives [30]. Similarly, recent work demonstrates that GNNs can also localize software defects, including condition-related bugs, execution order violations, and unreachable code [20, 29, 32]. In both cases, leveraging structural insights from CFGs enables more robust detection. Furthermore, graph reduction techniques— such as interval collapsing or decision node pruning—have shown to enhance not only performance but also interpretability by simplifying CFGs without losing essential semantic information [21, 22]. As a result, control flow learning is emerging as a versatile solution that bridges security and software quality concerns.

Building on the classical framework of Allen [5], Cocke [6], and Hecht [18], the paper reformulates interval reduction in an explicit algorithmic form and studies the structural properties preserved by the resulting derived graphs. The proposed implementation is evaluated through a controlled comparison with Sequential Node Merging [16] on a benchmark composed of canonical, real-world, and synthetic CFGs.

We do not aim to introduce interval reduction as a new concept, but to make this classical construction operational, formally transparent, and experimentally situated within contemporary program analysis.

We first note that sequential node merging is about combining adjacent nodes in a way that maintains the logical order of execution, while interval reduction focuses on optimizing the representation of intervals by merging overlapping intervals.

Our contribution treats these three main aspects.

1. The classical interval-reduction construction is presented as an explicit sequence of composable procedures, making it directly implementable and testable within modern static-analysis pipelines.
2. The paper studies the main properties preserved by the derived graphs, with particular attention to reachability, quotient structure, and the hierarchical organization induced by iterative reduction.
3. The implementation is evaluated through a controlled comparison with Sequential Node Merging on a benchmark composed of canonical, real-world, and synthetic CFGs.

Malware binaries intentionally create huge CFGs. Thus, reducing CFG complexity is therefore directly useful for malware analysts and detection systems.

In this way, the paper positions interval reduction as a classical but still relevant structural technique for reducing CFG complexity while retaining the control-flow organization required by downstream analyses.

The remainder of the paper is organized as follows. Section 2 recalls the necessary background on dominance, intervals, and reducibility, and introduces the formal terminology used throughout the paper. Section 3 presents the interval-reduction construction and establishes its main structural properties, including reachability preservation

under the quotient map and the strictly decreasing character of the derived sequence. Section 4 presents the algorithmic implementation and experimental evaluation of the method, including a comparison with Sequential Node Merging. Section 5 positions the proposed approach within the broader landscape of CFG-reduction techniques and discusses its limitations. Finally, Section 6 summarizes the main contributions and outlines directions for future work.

2. Fundamentals of Control Flow Graphs

Before delving into our interval-based reduction techniques, we first establish the essential definitions and properties of CFGs formalizing their structure, dominance relations, and interval constructs to ground our subsequent algorithms.

2.1. Dominance

Definition 2.1
A CFG is defined as a directed graph $G = (V, E, s)$ where

- V is the set of nodes, each representing a basic block, which is a linear sequence of instructions with a single entry and exit point.
- $E \subseteq V \times V$ is the set of directed edges representing the flow of control between basic blocks.
- s is the initial node, ensuring connectivity across the graph.

Definition 2.2
Let x and y be two (not necessarily distinct) nodes in a CFG G.
We say that x **dominates** y if every path in G from the initial node to y includes x, and we define

$$\mathrm{DOM}(y) = \{x \mid x \text{ dominates } y\}.$$

Remark 1.
Let s_0 be the source node in a CFG. The dominators of a node s are given by the maximal solution to the following data flow equations,

$$\mathrm{DOM}(s) = \begin{cases} \{s\} & \text{if } s = s_0, \\ \{s\} \cup \bigcap_{p \in \mathrm{preds}(s)} \mathrm{DOM}(p) & \text{if } s \neq s_0. \end{cases}$$

where predecessors (preds) are the nodes from which s is directly reachable.

Practically in programming, we recall the following algorithm [18] for one who would be interested in computing the dominators.

Algorithm 1 DOM: Computing Dominators in a Flow Graph

Input: CFG $G = (N, E)$ with entry node s such that every $n \in N$ is reachable from s.
Output: Dominator sets $\mathrm{DOM}(n)$ for each $n \in N$.
$\mathrm{DOM}(s) \leftarrow \{s\}$
for all $n \in N \setminus \{s\}$ **do**
    $\mathrm{DOM}(n) \leftarrow N$
end for
`changed` $\leftarrow$ `true`
while `changed` **do**
    `changed` $\leftarrow$ `false`
    for all $n \in N \setminus \{s\}$ in reverse postorder **do**
        `new` $\leftarrow \{n\} \cup \bigcap_{p \in \mathrm{pred}(n)} \mathrm{DOM}(p)$ ▷ $\mathrm{pred}(n) \neq \emptyset$ by reachability
        if `new` $\neq \mathrm{DOM}(n)$ **then**
            $\mathrm{DOM}(n) \leftarrow$ `new`
            `changed` $\leftarrow$ `true`
        end if
    end for
end while
return DOM

Example 2.1.
Consider a CFG taking the following form as in Figure 1.
The dominator sets for the nodes are:
$\mathrm{DOM}(s) = \{s\}$
$\mathrm{DOM}(a) = \{s, a\}$
$\mathrm{DOM}(b) = \{s, b\}$
$\mathrm{DOM}(c) = \{s, c\}$
$\mathrm{DOM}(d) = \{s, c, d\}$

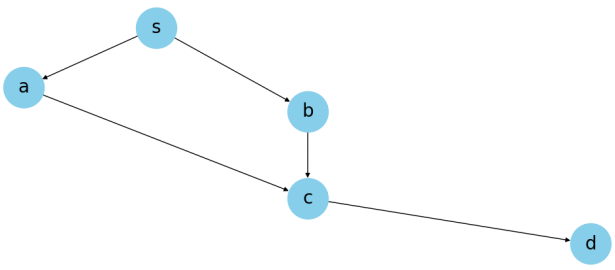


Figure 1. A CFG of 5 nodes.

Example 2.2.
Consider a CFG taking the following form as in Figure 2.
The dominator sets for the nodes are:

$\text{DOM}(s) = \{s\}$
$\text{DOM}(a) = \{s, a\}$
$\text{DOM}(b) = \{s, a, b\}$
$\text{DOM}(c) = \{s, a, c\}$
$\text{DOM}(d) = \{s, a, d\}$
$\text{DOM}(e) = \{s, a, d, e\}$
$\text{DOM}(f) = \{s, a, d, e, f\}$
$\text{DOM}(g) = \{s, a, d, e, f, g\}$
$\text{DOM}(h) = \{s, a, d, e, f, g, h\}$

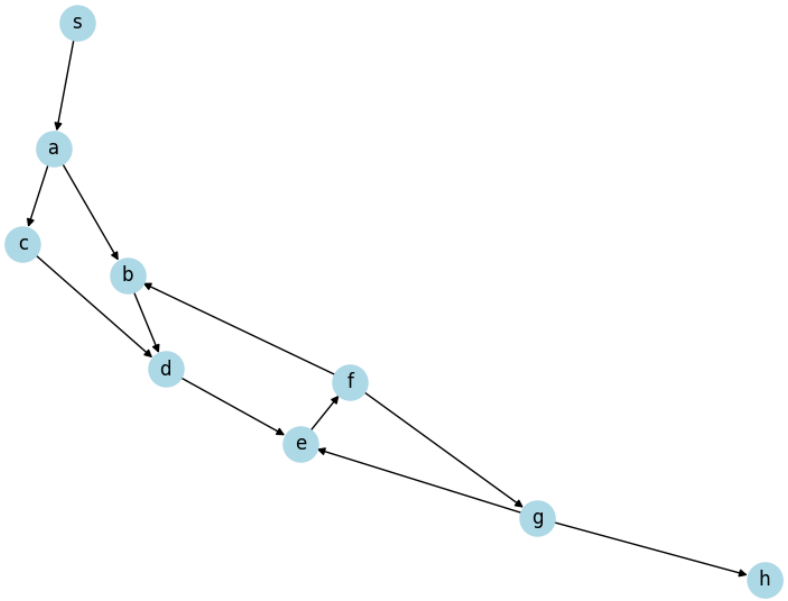


Figure 2. A CFG of 9 nodes.

2.2. Intervals

Definition 2.3
Let G be a flow graph, and let h be a node in G. The interval with header h, denoted by $I(h)$, is defined as the subset of nodes of G obtained as follows,

$$I(h) := \{h\}$$

As long as there exists a node m such that $m \notin I(h)$ and $m \neq s$ (the source node), and all arcs entering m leave nodes within $I(h)$, we extend $I(h)$ by adding m,

$$I(h) := I(h) \cup \{m\}$$

Remark 2. 1. This process continues until no such node m can be added.
It is also important to note that $I(h)$ is unique and does not depend on the order of selection of the nodes during the loop.
2. For any CFG $G = (V, E, s)$ and any interval $I(h)$, the subgraph induced by $I(h)$ forms a flow graph. [18]
3. Let $I(h)$ be the interval with header h, then following properties hold,
   - Every arc entering a node of the interval $I(h)$ from the outside must enter the header node h. That is, intervals are single-entry.
   - The header node h dominates every node in the interval $I(h)$.
   - Every cycle in $I(h)$ must include the header node h.[18]

3. Optimizing CFGs through Interval Reduction

Interval reduction focuses on minimizing the number of intervals in a CFG without losing essential control flow information. This section introduces algorithms designed to achieve this objective.

3.1. Reducibility

Definition 3.1
Let $G = (N, A, s)$ be a flow graph, where N is the set of nodes, A is the set of arcs, and s is the start node. The **derived flow graph** of G, denoted $I(G)$, is defined as follows:

1. The nodes of $I(G)$ are the intervals of G.
2. There is an arc from the node representing interval J to the node representing interval K if there exists an arc from any node in J to the header of K, and $J \neq K$.
3. The initial node of $I(G)$ is to the interval containing the start node s.

Definition 3.2
A sequence of flow graphs $G = G_0, G_1, \ldots, G_k$ is called a **derived sequence** of G if:

$$\begin{cases} G_{i+1} = \mathcal{I}(G_i), & \text{for } 0 \leq i < k, \\ G_k \neq G_{k-1}. \end{cases}$$

And $I(G_k) = G_k$ is the graph G_k called the **limit flow graph** of G and is denoted $I(G)$.

Definition 3.3
A flow graph is **reducible** (RFG) if and only if its limit flow graph satisfies the condition:

$$\mathcal{I}(G) = \{v\},$$

where v is a single node and no arcs exist.
If this condition is not met, the flow graph is called **irreducible** (IRFG) or **non-reducible**, meaning its limit flow graph contains either multiple nodes or at least one arc.

Example 3.1.
Consider the flow graph G in Figure 3 with edges $\{(1,2),(1,3),(2,3),(3,2),(2,4),(3,4)\}$ and start node 1.
The interval partition yields four singleton intervals: $I(1) = \{1\}$, $I(2) = \{2\}$, $I(3) = \{3\}$, $I(4) = \{4\}$.
Since no interval contains more than one node, the derived flow graph $I(G) = G$, and the graph is irreducible.
The mutual edges $2 \to 3$ and $3 \to 2$ create a multi-entry cycle: node 2 has a predecessor outside its interval (node 1 and node 3), and node 3 similarly has predecessors from different intervals.
Neither node can be absorbed into the other's interval because neither has all predecessors contained within a single interval.

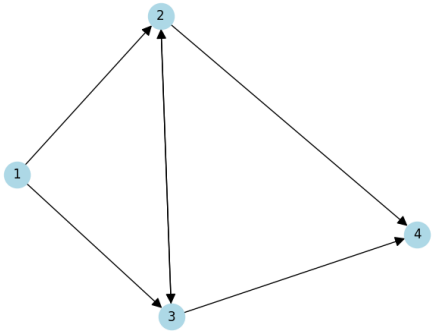


Figure 3. Irreducible graph with 4 nodes

Proposition 3.1.
If $\{G_0, G_1, \ldots, G_k\}$ is a sequence of derived graphs formed by iterative interval reduction, then this sequence forms a strictly decreasing chain under the quotient ordering,

$$G_k \preceq G_{k-1} \preceq \cdots \preceq G_0.$$

Proof

Let $G_i = (V_i, E_i)$ and $G_{i+1} = (V_{i+1}, E_{i+1})$ be two consecutive graphs in the sequence, where G_{i+1} is obtained from G_i by collapsing an interval $I(h)$ into a single node h'. Define the quotient map $\phi_i : V_i \to V_{i+1}$ as follows,

$$\phi_i(v) = \begin{cases} h', & \text{if } v \in I(h), \\ v, & \text{otherwise.} \end{cases}$$

The edge set E_{i+1} is defined such that,

$$(u, v) \in E_i \implies (\phi_i(u), \phi_i(v)) \in E_{i+1}.$$

Thus, G_{i+1} is a homomorphic image of G_i, and we write $G_{i+1} \preceq G_i$.

The sequence satisfies the following properties:
- Reflexivity: For all i, $G_i \preceq G_i$ holds trivially.
- Antisymmetry: If $G_{i+1} \preceq G_i$ and $G_i \preceq G_{i+1}$, then $G_i = G_{i+1}$.
This holds because interval reduction strictly decreases the number of nodes unless G_i is already reduced to a single node.
- Transitivity: If $G_k \preceq G_j$ and $G_j \preceq G_i$, then $G_k \preceq G_i$. This follows from the composition of the quotient maps $\phi_j \circ \phi_k$.

Since each step of interval reduction satisfies $|V_{i+1}| < |V_i|$, the process is strictly decreasing and terminates when $|V_k| = 1$. Thus, the sequence $\{G_0, G_1, \ldots, G_k\}$ forms a total order under $\preceq$:

$$G_k \preceq G_{k-1} \preceq \cdots \preceq G_0.$$

□

Theorem 3.1. (Reachability preservation under interval reduction)
Let $G = (V, E, s)$ be a flow graph and let $\mathcal{I}(G) = (V', E', s')$ be its derived flow graph, with quotient map $\varphi : V \to V'$ sending each node to the interval that contains it.
Write $u \xrightarrow{*} v$ to mean that v is reachable from u by a directed path of length ≥ 0. Then reachability is preserved by φ in the following sense:

(i) (Forward.) For all $u, v \in V$, if $u \xrightarrow{*} v$ in G, then $\varphi(u) \xrightarrow{*} \varphi(v)$ in $\mathcal{I}(G)$.
(ii) (Backward.) For all $u, v \in V$, if $\varphi(u) \xrightarrow{*} \varphi(v)$ in $\mathcal{I}(G)$, then there exist nodes $\tilde{u} \in \varphi^{-1}(\varphi(u))$ and $\tilde{v} \in \varphi^{-1}(\varphi(v))$ with $\tilde{u} \xrightarrow{*} \tilde{v}$ in G.

Proof
We prove the two directions in turn.
(i) Forward direction. Let

$$\pi : u = x_0 \to x_1 \to \cdots \to x_t = v$$

be a directed path in G of length $t \geq 0$.
We construct a corresponding walk in $\mathcal{I}(G)$ from $\varphi(u)$ to $\varphi(v)$ by applying φ pointwise to the vertices of π.
For each edge $(x_j, x_{j+1}) \in E$, exactly one of two cases holds.
Same interval. If $\varphi(x_j) = \varphi(x_{j+1})$, the two endpoints lie in the same interval and the projected step is a zero-length path at the single node $\varphi(x_j)$, which is valid under $\xrightarrow{*}$.
Distinct intervals. If $\varphi(x_j) \neq \varphi(x_{j+1})$, the endpoints lie in two distinct intervals $I(h)$ and $I(h')$.
By the single-entry property of intervals, the edge (x_j, x_{j+1}) must enter the header of the second interval, so $x_{j+1} = h'$.
By the construction of the derived flow graph, the pair $(\varphi(x_j), \varphi(x_{j+1}))$ is therefore an arc of E'.
Concatenating the projected steps yields a directed walk in $\mathcal{I}(G)$ from $\varphi(u)$ to $\varphi(v)$, which witnesses $\varphi(u) \xrightarrow{*} \varphi(v)$.

(ii) Backward direction. Let

$$\pi' : \varphi(u) = y_0 \to y_1 \to \cdots \to y_m = \varphi(v)$$

be a directed path in $\mathcal{I}(G)$ of length $m \geq 0$.
We construct a path in G between some pre-image of $\varphi(u)$ and some pre-image of $\varphi(v)$.
If $m = 0$, then $\varphi(u) = \varphi(v)$.
Choose $\tilde{u} = \tilde{v}$ to be the header of the interval $\varphi^{-1}(\varphi(u))$. The zero-length path $\tilde{u} \xrightarrow{*} \tilde{u}$ satisfies the conclusion.
If $m \geq 1$, let h_j denote the header of the interval y_j, for $j = 0, 1, \ldots, m$.
By the definition of the derived flow graph, each derived edge $(y_j, y_{j+1}) \in E'$ is induced by at least one original edge $(a_j, b_j) \in E$ with $a_j \in \varphi^{-1}(y_j)$ and $b_j = h_{j+1}$.
The interval $\varphi^{-1}(y_j)$ is itself a flow graph with header h_j as its single entry, and the inductive construction of the interval adds a node w only when all its predecessors are already in the interval; hence every node of $\varphi^{-1}(y_j)$ is reachable from h_j within the interval, and in particular there is a directed path $h_j \xrightarrow{*} a_j$ inside $\varphi^{-1}(y_j)$.
Concatenating these internal paths with the inter-interval edges yields the directed path in G

$$\tilde{u} := h_0 \xrightarrow{*} a_0 \to h_1 \xrightarrow{*} a_1 \to h_2 \xrightarrow{*} \cdots \to h_m =: \tilde{v},$$

with $\tilde{u} = h_0 \in \varphi^{-1}(\varphi(u))$ and $\tilde{v} = h_m \in \varphi^{-1}(\varphi(v))$ by construction. □

4. Implementation of the Interval Reduction

This section develops the practical procedure for iteratively merging "intervals" in a CFG.[†] Despite reducing the graph in multiple stages, the procedure preserves essential control-flow properties such as reachability and dominance. Our goal is to obtain a final, minimal form of the CFG without sacrificing correctness.

[†]A reference implementation of the algorithms is publicly available at https://github.com/hafid-m/cfg-interval-reduction.

4.1. Conceptual View

We identify, in each iteration, distinct subgraphs (intervals) that share a single, unique entry point. We then collapse the nodes of each interval into a single representative node. This contraction is repeated until the CFG can no longer be simplified or collapses into a single node. Formally, we follow these steps,

1. **Identify** subgraphs $I(h)$ for each header node h.
2. **Collapse** each $I(h)$ into a single node, updating edges so that all incoming and outgoing arcs remain well-defined and consistent with the original graph's behavior.
3. **Repeat** until no additional intervals can be formed.

4.2. Computing a Single Interval

Algorithm 2 (`Compute Interval`) systematically collects the nodes of an interval $I(h)$. Beginning with the header node h, we iterate through the unassigned nodes in the CFG. A node v is added to $I(h)$ if every predecessor of v is already contained in $I(h)$. This mechanism ensures the subgraph has a single point of entry: the header h.

Algorithm 2 Compute_Interval

Input: Graph G, header node h, assigned nodes $A \subseteq V$
Output: Interval $I(h) \subseteq V \setminus A$ with header h
$I \leftarrow \{h\}$
`changed` $\leftarrow$ `true`
while `changed` **do**
    `changed` $\leftarrow$ `false`
    for all $v \in V \setminus (I \cup A)$ **do**
        if $\text{pred}(v) \subseteq I$ **then**
            $I \leftarrow I \cup \{v\}$
            `changed` $\leftarrow$ `true`
        end if
    end for
end while
return I

4.3. Partitioning the Graph

Algorithm 3 (`interval_partition`) divides the entire graph G into intervals by running `Compute Interval` repeatedly. Initially, the set of headers H is seeded with the start node s. We pick a header $h \in H$, form the interval $I(h)$, and label those nodes as assigned. Any node whose predecessor belongs to $I(h)$ but has not yet been assigned is considered a new potential header. This continues until every node lies in exactly one interval.

Algorithm 3 INTERVAL_PARTITION

Input: Graph G; entry node s.
Output: List L of pairs $(h_i, I(h_i))$ partitioning V.
$L \leftarrow$ empty list
$A \leftarrow \emptyset$
$H \leftarrow \{s\}$
while $H \neq \emptyset$ **do**
    $h \leftarrow$ remove an element from H
    $I_h \leftarrow$ COMPUTE_INTERVAL(G, h, A)
    Append (h, I_h) to L
    $A \leftarrow A \cup I_h$
    for all $n \in V(G)$ such that $n \notin A$ **do**
        if any predecessor of n is in I_h **then**
            $H \leftarrow H \cup \{n\}$
        end if
    end for
end while
return L

4.4. Iterative Interval Collapse

We then employ Algorithm 5 (`reduce_flow_graph`), which calls `interval_partition` to find intervals in the current graph, then replaces each interval with a single node. Edges are updated so that the new collapsed graph preserves all fundamental flow relationships. We store each intermediate reduced graph in a sequence $\{G_0, G_1, \ldots, G_k\}$ until convergence, which arises because every iteration reduces the node set strictly.

Algorithm 4 COLLAPSE_INTERVAL

Input: Graph $G = (V, E)$, interval $I(h) \subseteq V$, fresh node $z \notin V$, color map C
Output: Reduced graph $G' = (V', E')$, updated color map C
$V' \leftarrow (V \setminus I(h)) \cup \{z\}$
$E' \leftarrow \emptyset$
for all $(u, v) \in E$ **do**
    if $u \notin I(h)$ and $v \in I(h)$ **then**
        $E' \leftarrow E' \cup \{(u, z)\}$
    else if $u \in I(h)$ and $v \notin I(h)$ **then**
        $E' \leftarrow E' \cup \{(z, v)\}$
    else if $u \notin I(h)$ and $v \notin I(h)$ **then**
        $E' \leftarrow E' \cup \{(u, v)\}$
    end if
end for
$C(z) \leftarrow C(h)$
for all $x \in I(h)$ **do**
    $C(x) \leftarrow C(h)$
end for
return $\big((V', E'), C\big)$

Algorithm 5 REDUCE_FLOW_GRAPH

Input: Graph G, start node s
Output: Sequence of reduced graphs R, color mapping (for visualization)
$R \leftarrow \{G\}$
$current\_graph \leftarrow G$
$node\_counter \leftarrow \max(V(G)) + 1$
Initialize $color\_map$ with unique colors for each node in G
while $|V(current\_graph)| > 1$ **do**
    $L \leftarrow$ INTERVAL_PARTITION$(current_graph, s)$
    if every interval in L is a singleton **then**
        break ▷ Irreducible graph (Def. 3.3)
    end if
    $s_{\text{next}} \leftarrow s$
    for all $(h, I) \in L$ **do**
        if $s \in I$ **then**
            $s_{\text{next}} \leftarrow node\_counter$ ▷ image of entry under collapse
        end if
        $current\_graph \leftarrow$ COLLAPSE_INTERVAL$(current_graph, I, node_counter, color_map, h)$
        $node\_counter \leftarrow node\_counter + 1$
    end for
    Append $current\_graph$ to R
    $s \leftarrow s_{\text{next}}$
end while
return $(R, color\_map)$

4.5. Numerical Experiments

This section provides a numerical evaluation of the interval-reduction pipeline. We first illustrate the iterative behavior of the algorithm on four representative examples. We then describe the experimental setup and report a controlled comparison with Sequential Node Merging (SNM) [16] on a benchmark of CFGs.

Numerical experiment 1.
Let us consider a CFG of 10 nodes taking the following form as shown in Figure 4.

The intervals partitioning the graph are as follows:

$$\text{Intervals found: } \big\{I(1) = \{1\},\ I(2) = \{2,3,4,5\},\ I(6) = \{6,7\},\ I(8) = \{8,9\},\ I(10) = \{10\}\big\}.$$

$$\begin{aligned}
\text{Step 1: } \big\{&I(11) = \{11\},\\
&I(12) = \{12\},\\
&I(13) = \{13,14,15\}\big\},\\
\text{Step 2: } \big\{&I(16) = \{16\},\\
&I(17) = \{17,18\}\big\},\\
\text{Step 3: } \big\{&I(19) = \{19,20\}\big\},\\
\text{Step 4: } \big\{&I(21) = \{21\}\big\}.
\end{aligned}$$

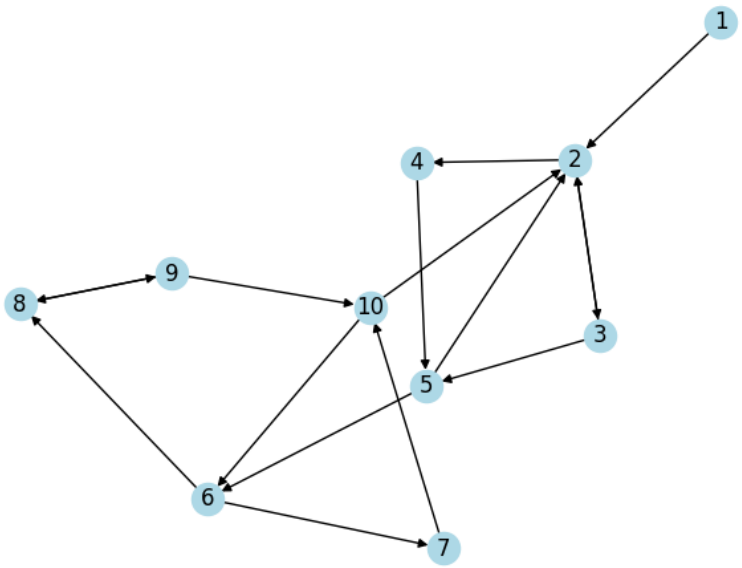


Figure 4. Original form of graph associated to numerical experiment 1.

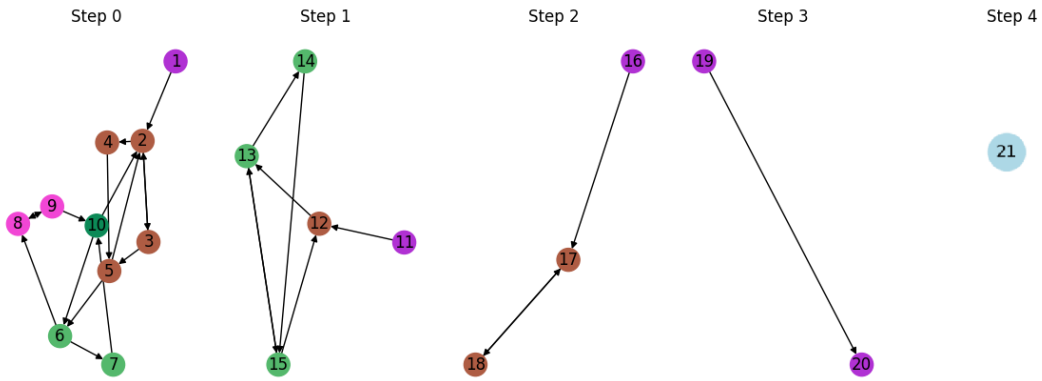


Figure 5. Reduction steps for graph 4. Node colours indicate interval membership, and the progressive coarsening illustrates the collapse of reducible regions into the limit graph.

Numerical experiment 2.
Let us consider a CFG of 8 nodes taking the following form as shown in Figure 6.

The intervals partitioning the graph are as follows:

$$\text{Intervals found: } \big\{I(1)=\{1\},\ I(8)=\{8,2,7\},\ I(3)=\{3,4\},\ I(5)=\{5,6\}\big\}.$$

$$\text{Step 1: } \big\{I(9)=\{9\},\ I(10)=\{10,11,12\}\big\},$$

$$\text{Step 2: } \big\{I(13)=\{13,14\}\big\},$$

$$\text{Step 3: } \big\{I(15)=\{15\}\big\}.$$

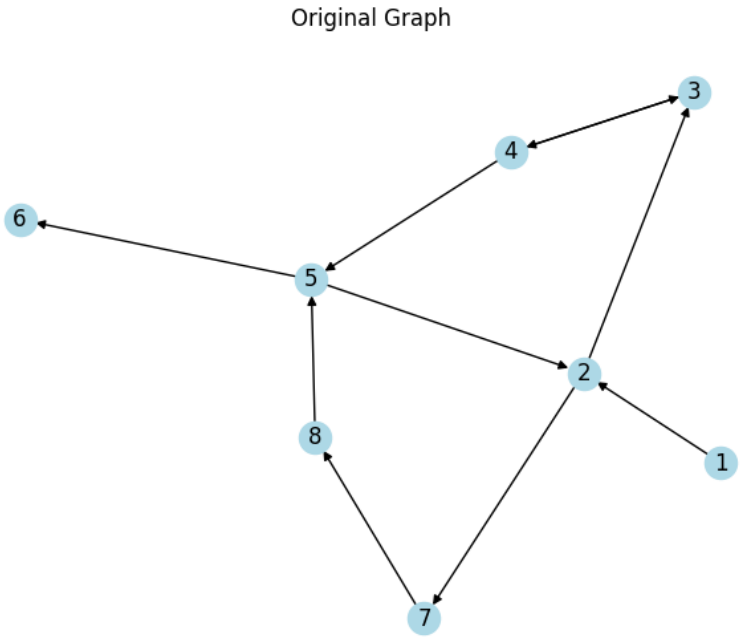


Figure 6. Original form of graph associated to numerical experiment 2.

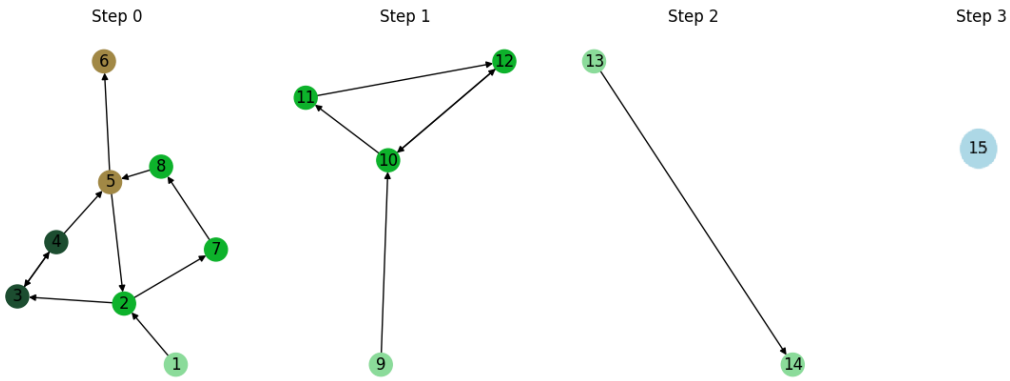


Figure 7. Steps of graph reduction applied to CFG presented in Figure 6.

Numerical experiment 3.
Let us consider a CFG of 38 nodes taking the following form as shown in Figure 8.

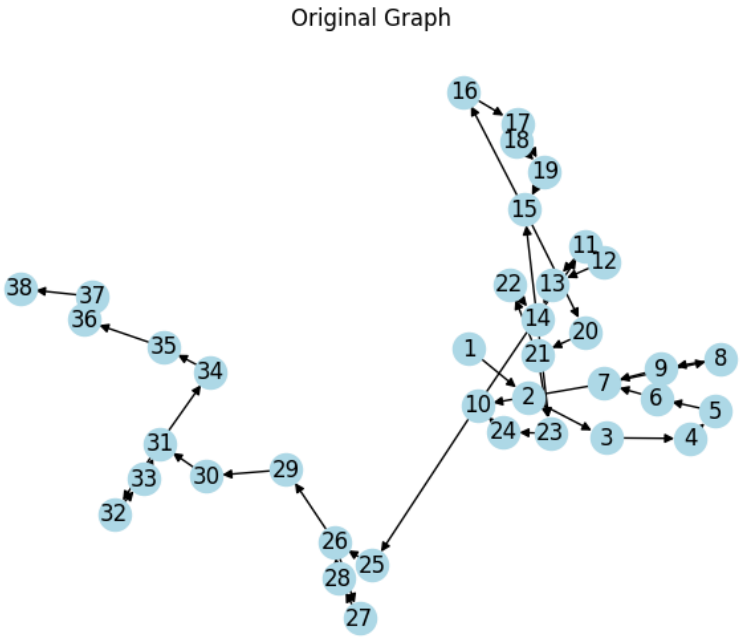


Figure 8. Original form of graph associated to numerical experiment 3.

The intervals partitioning the graph are as follows:

$$\begin{aligned}\text{Intervals found: } \{&I(1)=\{1,2,3,4,5,6\},\\ &I(8)=\{8,9,7\},\\ &I(10)=\{10,11,12,13,25\},\\ &I(14)=\{14\},\\ &I(24)=\{24,23\},\\ &I(26)=\{26,27,28,29,30\},\\ &I(32)=\{32,33,34,35,36,37,38,31\},\\ &I(15)=\{15,16,17,18,19,20,21,22\}\}.\end{aligned}$$

$$\begin{aligned}\text{Step 1: } \{&I(40)=\{40,39\},\\ &I(41)=\{41,44,45\},\\ &I(42)=\{42,43,46\}\},\end{aligned}$$

$$\begin{aligned}\text{Step 2: } \{&I(47)=\{47\},\\ &I(48)=\{48,49\}\},\end{aligned}$$

$$\text{Step 3: } \{I(50)=\{50,51\}\},$$

$$\text{Step 4: } \{I(52)=\{52\}\}.$$

Numerical experiment 4.
Let us consider a CFG of 27 nodes taking the following form as shown in Figure 10.

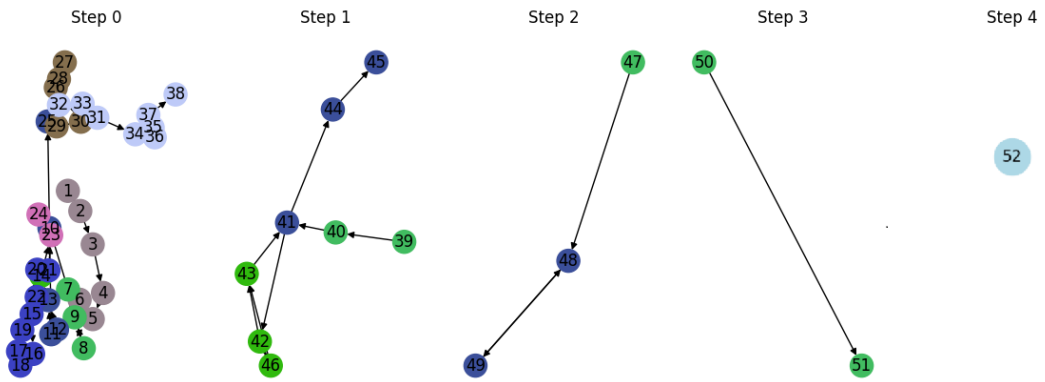


Figure 9. Steps of graph reduction applied to CFG presented in Figure 8.

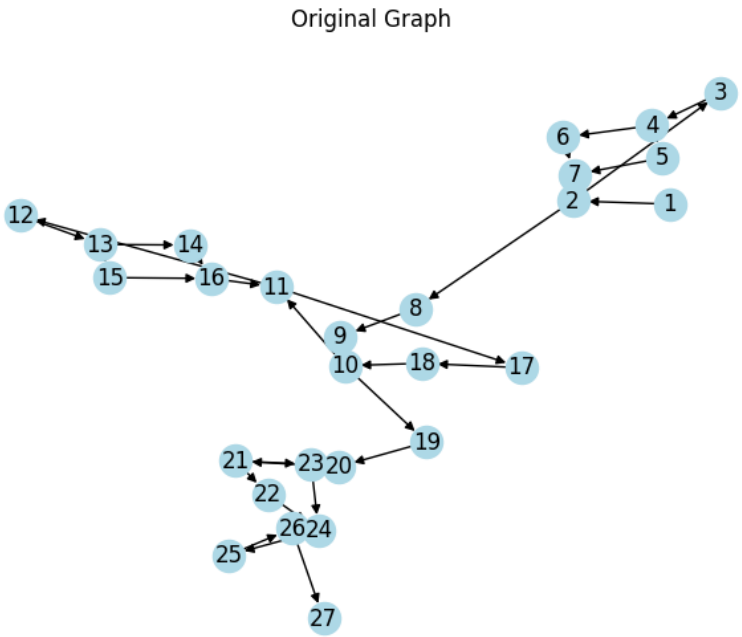


Figure 10. Original form of graph associated to numerical experiment 4.

The intervals partitioning the graph are as follows:

$$\begin{aligned}\text{Intervals found: } \big\{ & I(1) = \{1\},\\ & I(2) = \{2,3,4,5,6,7,8,9\},\\ & I(10) = \{10,19,20,21,22,23,24,25,26,27\},\\ & I(11) = \{11,12,13,14,15,16,17,18\}\big\}.\end{aligned}$$

$$\text{Step 1: } \{I(28) = \{28, 29\},$$
$$I(30) = \{30, 31\}\},$$
$$\text{Step 2: } \{I(32) = \{32, 33\}\},$$
$$\text{Step 3: } \{I(34) = \{34\}\}.$$

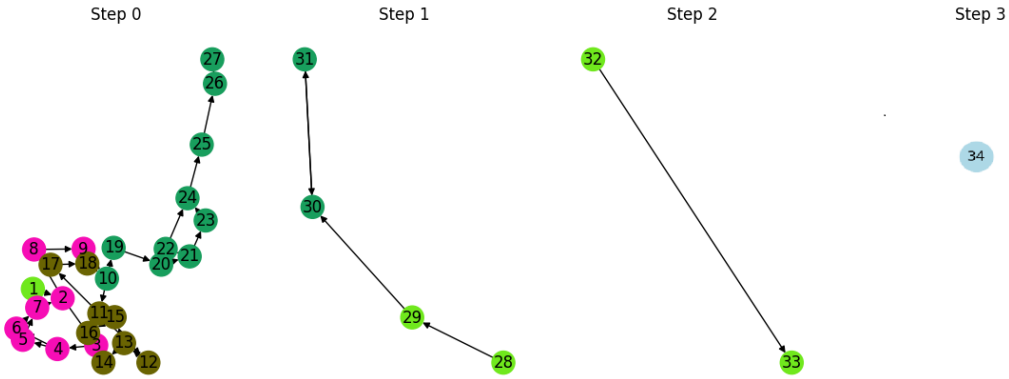


Figure 11. Steps of graph reduction applied to CFG presented in Figure 10.

Numerical experiment 5.
Let us consider a CFG of 100 nodes taking the following form as shown in Figure 12.

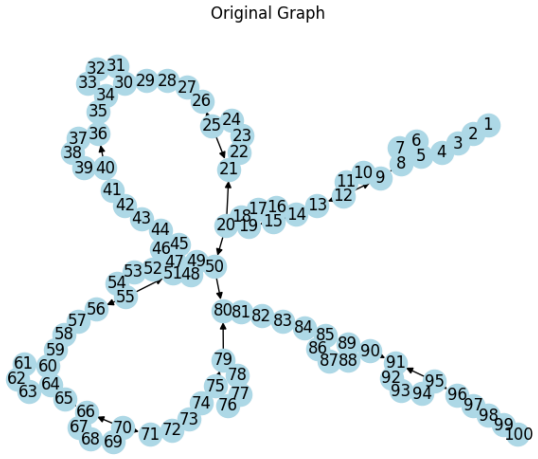


Figure 12. Original form of graph associated to numerical experiment 5

The intervals partitioning the graph are as follows:

Intervals found: $\big\{I(1) = \{1, 2, 3, 4, 5, 6, 7, 8\},$
$I(9) = \{9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\},$
$I(50) = \{50\},\ I(51) = \{51\},$
$I(52) = \{52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65\},$
$I(21) = \{21\},$
$I(22) = \{22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35\},$
$I(80) = \{80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90\},$
$I(66) = \{66\},$
$I(67) = \{67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79\},$
$I(36) = \{36\},\ I(91) = \{91\},$
$I(92) = \{92, 93, 94, 95, 96, 97, 98, 99, 100\},$
$I(37) = \{37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49\}\big\}.$

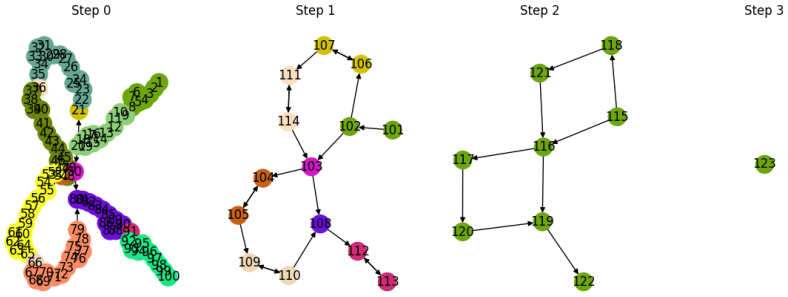
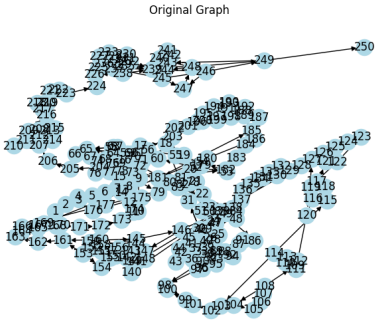


Figure 13. Steps of graph reduction applied to CFG presented in Figure 12.

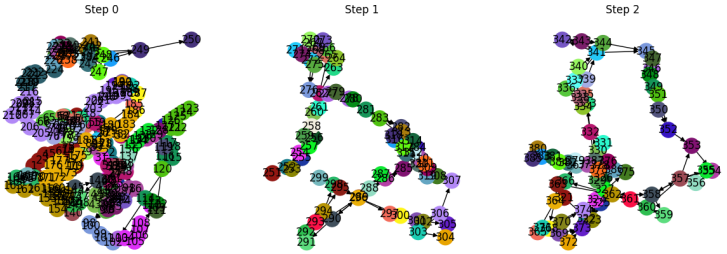
Step 1: $\big\{I(101) = \{101, 102\},\ I(103) = \{103\},$
$I(104) = \{104, 105\},\ I(106) = \{106, 107\},$
$I(108) = \{108\},\ I(109) = \{109, 110\},$
$I(111) = \{111, 114\},\ I(112) = \{112, 113\}\big\}.$

Step 2: $\big\{I(115) = \{115, 116, 117, 118, 119, 120, 121, 122\}\big\}.$

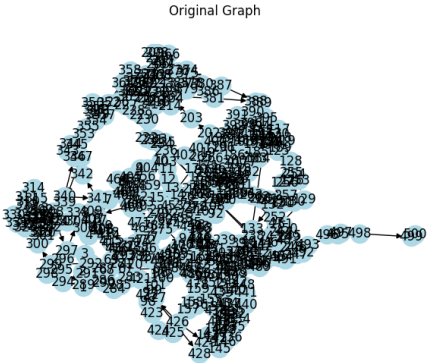
Step 3: $\big\{I(123) = \{123\}\big\}.$



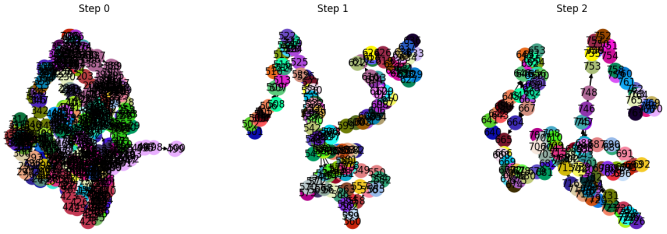
(a) 250 nodes - Original graph



(b) 250 nodes - reduction (Step $0 \rightarrow 2$)



(c) 500 nodes - Original graph

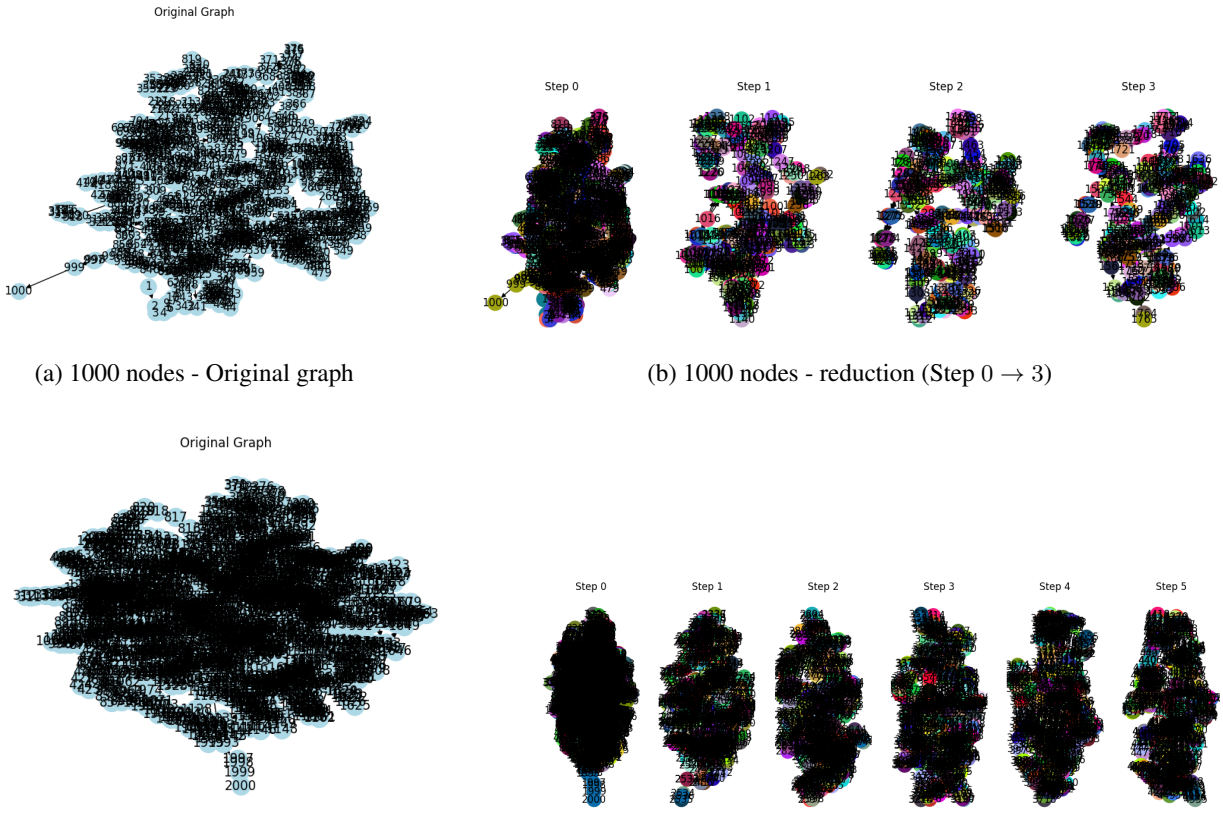


(d) 500 nodes - reduction (Step $0 \rightarrow 2$)

Figure 14. Interval reduction applied to two synthetic CFGs from Tier 3. (a) The original 250-node graph and (b) its reduction over two iterations ($G_0 \rightarrow G_2$), and (c) the original 500-node graph with (d) its reduction over two iterations. Node colours indicate interval membership at each step; the progressive coarsening illustrates how the derived sequence collapses reducible regions while exposing the residual irreducible structure.

In Figure 14, we can observe that when we increase the number of nodes, for example either by starting with 250 or 500 nodes, we are in front of a new situation where there may be just two steps of interval reduction. We could also be in a situation of delayed steps of reduction and this can be observed from the last case in Figure 15 where we have used now a very large number of nodes, namely 2000. Another very interesting situation that is applied for these cases, is that the last step could give an irreducible graph as we can observe for example when our input is 1000 nodes, deducing that one single node as a final step of reduction was never necessary but this could be observed only in the first basic examples when the number of nodes is small.

The results indicate that interval reduction offers a principled way to reduce graph complexity while preserving the hierarchical and semantic organization of control flow, supporting its application to malware detection, binary analysis, and other security-oriented program analysis tasks.



(a) 1000 nodes - Original graph

(b) 1000 nodes - reduction (Step $0 \to 3$)

(c) 2000 nodes - Original graph

(d) 2000 nodes - reduction (Step $0 \to 5$)

Figure 15. Interval reduction at larger scales. (a) The original 1000-node graph and (b) its reduction over three iterations ($G_0 \to G_3$), and (c) the original 2000-node graph with (d) its reduction over five iterations ($G_0 \to G_5$).

Observations. On the 12 real-world CFGs of Tier 2, IR achieves a mean vertex reduction ratio of about 95% versus 25% for SNM a 70 percentage point gap. All 12 Tier 2 graphs are reducible (they reduce to a single node) because real Python code naturally produces properly nested loop and branch structures. On the same graphs SNM can only collapse linear chains, leaving every branch and join point intact.

On the 5 synthetic Tier 3 graphs of 100–2000 nodes, IR reduces roughly 75% of nodes versus SNM's 43%, a 30 percentage point gap that is consistent across all sizes. The synthetic graphs contain randomly placed multi entry cycles that produce irreducible regions; the algorithm correctly halts on the residual structure rather than producing incorrect output. SNM continues to collapse the linear portions but, as in Tier 2, cannot reach the loop and branch structure that drives the gap.

Across all 28 graphs the iteration count is at most 5, including at $|V| = 2000$ nodes. This is consistent with the $O(\log n)$ practical iteration count established in Section 4.7. The behavior reflects the empirical observation of Allen and Cocke [6] that real program CFGs have shallow loop nesting depth; the same shallowness appears in our synthetic generator.

Irreducibility. Two graphs in Tier 1 (`irreducible_4` and `irreducible_21`) are deliberately irreducible. On the 4 node example, the algorithm correctly identifies that every interval at the first level is a singleton and halts without modifying the graph. On the 21 node example, the algorithm partially reduces the graph (from 21 to 7 nodes) by collapsing the reducible sub-regions, leaving a 7 node residual that exposes the multi-entry cycles in

Graph	$\|V\|$	$\|E\|$	SNM $\|V'\|$	SNM RV	SNM RE	SNM time	IR $\|V'\|$	IR RV	IR RE	IR iter	IR time
Tier 1: canonical and synthetic CFGs											
irreducible_4	4	6	4	0%	0%	<1 ms	4	0%	0%	0	<1 ms
while_loop	5	5	4	20%	20%	<1 ms	1	80%	100%	2	<1 ms
if_else	6	6	4	33%	33%	<1 ms	1	83%	100%	1	<1 ms
nested_loop	6	7	6	0%	0%	<1 ms	1	83%	100%	3	<1 ms
cfg_8	8	10	6	25%	20%	<1 ms	1	88%	100%	3	<1 ms
deep_nest	12	14	9	25%	21%	<1 ms	1	92%	100%	3	<1 ms
complex	20	23	12	40%	35%	<1 ms	1	95%	100%	2	4 ms
irreducible_21	21	26	12	43%	35%	<1 ms	7	67%	54%	1	<1 ms
cfg_38	38	46	22	42%	35%	<1 ms	1	97%	100%	4	1 ms
floyd_warshall	51	61	34	33%	28%	1 ms	1	98%	100%	4	2 ms
cfg_100	100	119	39	61%	51%	2 ms	1	99%	100%	3	5 ms
Tier 2: real-world CFGs from the Python standard library											
difflib.get_close_matches	12	13	9	25%	23%	<1 ms	1	92%	100%	2	<1 ms
bisect.bisect_right	17	22	14	18%	14%	<1 ms	1	94%	100%	2	<1 ms
difflib.unified_diff	17	25	15	12%	8%	<1 ms	1	94%	100%	4	2 ms
difflib.context_diff	18	27	13	28%	19%	<1 ms	1	94%	100%	4	2 ms
json.decoder.JSONArray	22	27	16	27%	22%	<1 ms	1	95%	100%	2	<1 ms
heapq.nlargest	27	30	19	30%	27%	<1 ms	1	96%	100%	2	<1 ms
difflib._mdiff	27	32	18	33%	28%	<1 ms	1	96%	100%	3	2 ms
json.decoder.py_scanstring	28	34	21	25%	21%	<1 ms	1	96%	100%	2	2 ms
heapq.merge	35	43	26	26%	21%	<1 ms	1	97%	100%	3	9 ms
json.decoder.JSONObject	50	60	34	32%	27%	<1 ms	1	98%	100%	2	2 ms
Tier 3: synthetic CFGs at scale											
synthetic_100	100	139	57	43%	31%	2 ms	21	79%	—	3	29 ms
synthetic_250	250	348	140	44%	32%	8 ms	67	73%	—	2	170 ms
synthetic_500	500	698	281	44%	31%	110 ms	131	74%	—	2	730 ms
synthetic_1000	1000	1398	582	42%	30%	100 ms	249	75%	—	3	3.5 s
synthetic_2000	2000	2799	1148	43%	30%	410 ms	501	75%	—	5	22 s
Per-tier averages											
Tier 1	—	—	—	**29%**	**25%**	—	—	**80%**	**87%**	—	—
Tier 2	—	—	—	**25%**	**21%**	—	—	**95%**	**100%**	—	—
Tier 3	—	—	—	**43%**	**31%**	—	—	**75%**	**63%**	—	—
Overall	—	—	—	**30%**	**24%**	—	—	**86%**	**88%**	—	—

Table 1. Comparison between Sequential Node Merging (SNM) and Interval Reduction (IR) on benchmark CFGs across three tiers. Runtimes report the mean over 30 runs (10 runs for $|V| \geq 500$).

the original. In both cases the algorithm produces a clear signal that the input is not reducible, which is the correct behavior for a path preserving reduction.

Individual cases. Two individual results are worth highlighting. On `nested_loop_6`, SNM achieves zero reduction while IR collapses the graph to a single node (about 83% RV); the entire graph is one nested loop structure with no linear chains for SNM to find. On `difflib.unified_diff`, a real world CFG, SNM reduces about 12% while IR reduces about 94%—an 82 percentage point gap on a single graph. These illustrate the fundamental difference between the two methods: SNM can only operate on linear chains, while IR addresses the full nested-region structure of the CFG.

Runtime. Both algorithms run in milliseconds on graphs up to 100 nodes. At $|V| = 2000$, IR takes about 22 seconds per run while SNM takes a fraction of a second. The gap is consistent with their respective complexities: SNM performs a single linear-time scan per iteration over the surviving graph, while IR (4.7) examines each interval's predecessor structure repeatedly within a partition step. The IR runtime can be substantially improved by a predecessor counter implementation [18] that reduces each partition step to $O(|E|)$; we identify this as future work in Section 6.

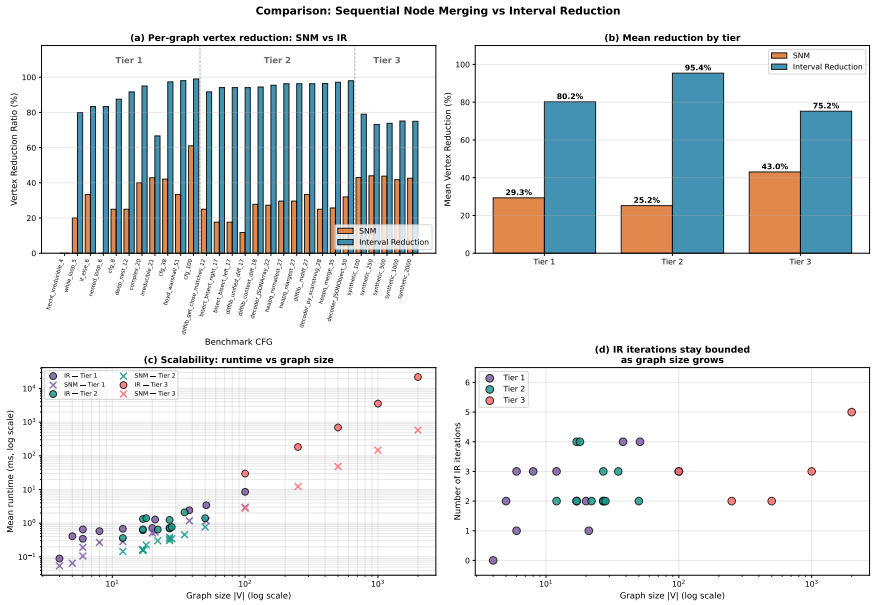


Figure 16. Comparison of Sequential Node Merging (SNM) and Interval Reduction (IR) across 28 benchmark CFGs. (a) Per-graph vertex reduction; dashed lines separate the three tiers. (b) Mean vertex reduction by tier with sample sizes. (c) Scalability on a log–log scale showing runtime versus graph size for both methods. (d) IR iteration count remains bounded (≤ 5) across all sizes, empirically supporting the $O(\log n)$ practical bound discussed in 4.7.

4.6. Information Preservation under Interval Reduction

A natural concern with any graph reduction technique is whether it discards structural information that downstream analyses depend on. We show in this subsection that interval reduction does not lose structural information; rather, it organizes the control flow graph into a hierarchy of nested single entry regions. The derived sequence $G_0, G_1, \ldots, G_k$ provides multiple levels of abstraction, from the full basic-block view (G_0) down to the minimal single-node representation (G_k). At every level, the essential structural relations of the original graph (reachability, dominance, and single entry regions) are preserved by the quotient maps. This is the fundamental difference with Sequential Node Merging, which produces a single flattened view of the graph.

To illustrate the hierarchy concretely, consider the 8-node CFG of Numerical Experiment 2 (Figure 6). At the basic block level (G_0), the graph has 8 nodes and 10 edges, and its structure is not immediately apparent from inspection. The interval partition at the first level immediately reveals the program's region structure: the entry block, the loop header region $\{2, 7, 8\}$, the inner loop body $\{3, 4\}$, and the loop exit $\{5, 6\}$. Each subsequent level abstracts further, until G_k presents the program as a single node. The analyst may choose the level of abstraction appropriate to the task at hand—basic-block precision for instruction scheduling, mid-level region structure for loop optimization, the single-node view for whole-program reasoning. SNM cannot offer this choice because it produces only one reduced graph.

We make these claims precise in the following proposition.

Proposition 4.1 (*Structural preservation under interval reduction*).
Let $G = (V, E, s)$ be a flow graph with derived sequence $G_0, G_1, \ldots, G_k$ produced by iterative interval reduction, and let $\varphi_i : V_i \to V_{i+1}$ denote the quotient map at each step. Then the following structural properties hold:

(i) Dominance preservation. If a node d dominates a node n in G_i and $\varphi_i(d) \neq \varphi_i(n)$, then $\varphi_i(d)$ dominates $\varphi_i(n)$ in G_{i+1}.

(ii) Loop-nesting hierarchy. For every node $v \in V$, the pre-images $\Phi_j^{-1}(\Phi_j(v))$ under the composed maps $\Phi_j = \varphi_{j-1} \circ \cdots \circ \varphi_0$ form a strictly nested chain of single-entry regions:

$$\{v\} \subseteq \Phi_1^{-1}(\Phi_1(v)) \subseteq \Phi_2^{-1}(\Phi_2(v)) \subseteq \cdots \subseteq \Phi_k^{-1}(\Phi_k(v)) \subseteq V.$$

This chain captures the loop-nesting structure of the program represented by G.

(iii) Forward-propagable information. For any forward data-flow property f computed at the level of G_{i+1}, the values of f can be propagated back to obtain valid information at every node of G_i by following the interval-internal successor relation, as in the procedure of Allen and Cocke [6] for global data-flow analysis.

Proof

We prove each part.

(i). Suppose d dominates n in G_i. Let $s_{i+1} = \varphi_i(s_i)$.

Suppose for contradiction that $\varphi_i(d)$ does not dominate $\varphi_i(n)$ in G_{i+1}. Then there exists a path

$$\pi' : s_{i+1} = y_0 \to y_1 \to \cdots \to y_m = \varphi_i(n) \quad \text{in } G_{i+1}$$

that avoids $\varphi_i(d)$.

By Theorem 3.1, there exist nodes $\tilde{s} \in \varphi_i^{-1}(s_{i+1})$ and $\tilde{n} \in \varphi_i^{-1}(\varphi_i(n))$ with $\tilde{s} \xrightarrow{*} \tilde{n}$ in G_i, and the lifted path passes only through nodes of $\varphi_i^{-1}(y_0), \varphi_i^{-1}(y_1), \ldots, \varphi_i^{-1}(y_m)$.

Since π' avoids $\varphi_i(d)$, the lifted path avoids the interval $\varphi_i^{-1}(\varphi_i(d))$.

In particular, the lifted path avoids the node d itself. By the definition of intervals, $\tilde{s}$ and s_i both belong to the interval $\varphi_i^{-1}(s_{i+1})$; since the header of this interval dominates every node within it, there is a path $s_i \xrightarrow{*} \tilde{s}$ entirely inside that interval, which also avoids d (since $\varphi_i(d) \neq s_{i+1}$ would place d in a different interval).

Concatenating, we obtain a path from s_i to $\tilde{n}$ in G_i that avoids d. Since $\tilde{n}$ lies in the same interval as n and the interval header dominates n, this path can be extended within the interval to reach n.

This contradicts the assumption that d dominates n in G_i.

(ii). We use induction on j. For $j = 0$, Φ_0 is the identity and $\Phi_0^{-1}(\Phi_0(v)) = \{v\}$. For the inductive step, assume the chain holds up to level j.

The map φ_j collapses each interval $I(h)$ at level j into a single node. Hence

$$\Phi_{j+1}^{-1}(\Phi_{j+1}(v)) = \Phi_j^{-1}\big(\varphi_j^{-1}(\Phi_{j+1}(v))\big) \supseteq \Phi_j^{-1}(\Phi_j(v)),$$

since φ_j maps $\Phi_j(v)$ to an interval-node that contains it. Each $\Phi_j^{-1}(\Phi_j(v))$ is the pre-image of an interval at level j, and by induction it is a union of intervals at every preceding level.

By the single-entry property at each level, each $\Phi_j^{-1}(\Phi_j(v))$ is itself a single-entry region of G.

The cycle-containment property ensures that every cycle inside such a region passes through the header at some level, which is the structural signature of a loop.

The nested sequence therefore mirrors the loop-nesting hierarchy.

(iii). Let f be a forward data-flow property computed at the level of G_{i+1}.

For each interval $I(h)$ in G_i that was collapsed to a node z in G_{i+1}, the value $f(z)$ represents information valid at the boundary of the interval. The only entry into $I(h)$ from outside is through h.

Therefore $f(z)$ provides valid entry information for h.

The information can then be propagated through the nodes of $I(h)$ in topological order (within the interval), using the interval-internal successor relation. This meets exactly Procedure F of Allen and Cocke [6], which propagates information from the reduced graph G_k back to G_0 level by level, recovering node-level precision at every block of the original CFG. □

Remark 3. Proposition 4.1 establishes that interval reduction is not lossy compression. The derived sequence $G_0, G_1, \ldots, G_k$ provides a hierarchy of abstractions, each preserving the structural relations of the original graph:

- G_0: full detail (every basic block),
- G_1: first-order region structure (innermost loops collapsed),

- $G_2, \ldots, G_{k-1}$: progressively coarser loop nesting,
- G_k: a single node (the program as a whole).

This is structurally distinct from Sequential Node Merging, which collapses linear chains in a single pass and produces only one reduced graph. The hierarchical property is precisely what enables Allen and Cocke's classical bottom-up/top-down data-flow framework [6] and, more recently, multi-level explainability for graph-learning models [21, 31], where explanations produced at a coarse level can be lifted back to the basic-block level through the chain of quotient maps.

4.7. Computational Complexity

We analyze the worst-case time complexity of the interval reduction pipeline, then compare the theoretical bound against the iteration counts and runtimes.

Per-step complexity. Each iteration of the outer loop of Algorithm 5 performs one call to INTERVAL_PARTITION followed by one COLLAPSE_INTERVAL per interval found. We analyze these in turn.

COMPUTE_INTERVAL (Algorithm 2) inspects each unassigned node v and checks whether $\mathrm{pred}(v) \subseteq I$. A naive implementation costs $O(|V| \cdot |E|)$ per call. With predecessor counters maintained incrementally—a counter per node initialized to $|\mathrm{pred}(v)|$ and decremented whenever a predecessor is added to the current interval—the check becomes constant time, and each edge is examined at most once during the entire partition step. This yields $O(|V| + |E|)$ per call to INTERVAL_PARTITION.

COLLAPSE_INTERVAL (Algorithm 4) processes each edge once and each node of the collapsed interval once, costing $O(|V| + |E|)$ per call. Since the intervals partition V_i, the total cost of all collapses at iteration i is $O(|V_i| + |E_i|)$.

Therefore, one outer iteration runs in $O(|V_i| + |E_i|)$ time using $O(|V_i| + |E_i|)$ space.

Total complexity. The outer loop terminates when $|V_k| = 1$ or when every interval at the current level is a singleton (the irreducible case). Each non-terminal iteration strictly decreases the node count, so at most $|V_0| - 1$ iterations occur in the worst case. Combined with the per-iteration bound, this gives a worst-case total complexity of $O(|V| \cdot |E|)$.

This worst-case bound is rarely tight in practice. On reducible flow graphs the iteration count is governed by the loop-nesting depth of the program rather than by $|V|$. Allen and Cocke [6] observed that real program CFGs typically exhibit nesting depth $O(\log |V|)$ or less, which suggests an effective complexity of $O((|V| + |E|) \log |V|)$.

Iteration count. Figure 16(d) reports the observed iteration count across the 28 benchmark CFGs. Across all sizes from $|V| = 4$ to $|V| = 2000$, the iteration count never exceeds 5 on reducible graphs, with most reducible graphs converging in 2–4 iterations regardless of size. This is well below the worst-case bound of $|V| - 1$ and is consistent with the $O(\log |V|)$ practical iteration count expected from shallow loop nesting. On the irreducible graphs the iteration count is either 0 (`irreducible_4`) or 1 (`irreducible_21`), in both cases corresponding to immediate detection of irreducibility.

Runtime scaling. Figure 16(c) plots runtime versus graph size on a log-log scale. The observed scaling for IR is consistent with super-linear but sub-quadratic growth, in line with the $O((|V| + |E|) \log |V|)$ practical bound. At $|V| = 2000$ our Python implementation requires about 22 seconds per run; this overhead is dominated by per-node iteration over predecessor sets in pure Python rather than by the algorithmic bound. A C-language or Cython implementation of the predecessor-counter optimization described above would reduce this constant factor by one to two orders of magnitude.

5. Discussion

Reducing CFGs is a key task in program analysis, where preserving the essential semantic structure while omitting unnecessary details is critical. Researchers have long sought effective methods to simplify these graphs without compromising the integrity of the original program's execution paths, and the literature reveals a variety of techniques, each grounded in theoretical insights and driven by practical needs. In this discussion we examine the technique of merging sequential nodes [16], address heuristic graph pruning methods [21], and position our experimental findings against both before discussing limitations.

One of the earliest and most influential methods for CFG reduction is the merging of sequential nodes—a procedure that collapses linear sequences of instructions into single nodes, producing a "decision graph". In this approach, nodes that represent non-critical, sequential statements are merged, while those corresponding to control decisions (such as branch or loop conditions) are preserved in their original form. The rationale is that many parts of a program execute in a deterministic, linear fashion; by collapsing these sequences, one can substantially reduce the size of the graph while retaining the fundamental decision points that dictate the program's behavior. Gold's work [16] on CFG reductions formalizes this procedure and proves that every execution path in the original graph projects to a corresponding path in the reduced version (path-preservation property).

Our experimental comparison (Table 1) quantifies the practical limit of this approach. On the 12 real-world standard-library CFGs of Tier 2, SNM achieves a mean vertex-reduction ratio of 25%, leaving most of the graph intact. The reason is structural: SNM collapses only those nodes whose predecessor and successor are unique, which excludes all branch points, all join points, and all loop headers. On graphs with even modest branching and looping structure—which characterizes essentially every non-trivial real program—SNM's reductive power is fundamentally bounded. This merging streamlines the graph structure for linear regions but cannot expose the loop and branch structure that dominates the graph's overall shape.

Building on the concept of simplification, subsequent approaches have introduced heuristic graph pruning techniques. Graph pruning methods aim to remove nodes and edges that contribute marginally to the overall control structure, thereby eliminating noise and reducing computational overhead. For instance, leaf pruning targets nodes that reside at the periphery of the CFG and do not influence major control decisions. Such nodes are often the result of redundant or incidental code patterns that, while present in the original graph, do not affect the core execution semantics. In many modern applications, particularly in malware detection, these heuristic techniques are integrated with machine learning frameworks. By pruning less significant nodes prior to feeding the CFG into a GNN, researchers have demonstrated notable improvements in both the efficiency and accuracy of the detection process. Moreover, when combined with explainability modules like GNNExplainer [31], pruning contributes to a more interpretable model by isolating the subgraphs that are most indicative of malicious behavior. This interplay between reduction and interpretability is crucial, as it provides not only faster processing times but also the ability to trace the origins of a classification decision, a property valued in high-stakes security applications.

Heuristic pruning is fundamentally different from interval reduction in one critical respect: it does not preserve reachability or dominance. A pruned graph may have removed nodes that are reachable from the entry, and the resulting graph is no longer a homomorphic image of the original. This is an acceptable trade-off for tasks where structural fidelity is secondary to model throughput, but it makes pruning unsuitable as a preprocessing stage for analyses that depend on path-preservation (formal verification, dataflow analysis, dominance-based optimization). Interval reduction and heuristic pruning are therefore complementary, not competing: the former preserves structure at multiple zoom levels, while the latter aggressively shrinks graphs for downstream learning tasks.

It is important to distinguish between two fundamentally different pruning methods. The first, discussed above, removes structurally peripheral yet reachable nodes in order to reduce graph size for learning pipelines. The second strategy is more conservative: it removes only *unreachable* code, namely nodes that cannot be reached from the entry node through any directed path. This second form of pruning is therefore reachability-preserving by construction.

Unreachable-code pruning is a standard and safe preprocessing step. It can be performed by a single graph traversal starting from the entry node. Either of the two classical traversal strategies, Breadth-First Search (BFS) or Depth-First Search (DFS), may be used, as shown in Algorithm 6. In both cases, the algorithm computes the

set of nodes reachable from the entry and discards all remaining nodes. This conservative cleanup is compatible with interval reduction, since it preserves the structural relations on which interval construction relies, in particular reachability and the dominance relations induced by reachable paths.

Algorithm 6 Reachability-preserving pruning by graph traversal. Both variants discard only nodes unreachable from the entry s; (a) uses depth-first and (b) breadth-first traversal. The two produce identical results and differ only in traversal order.

Require: Flow graph $G = (V, E)$; entry s
Ensure: Subgraph reachable from s
$R \leftarrow$ DFS-PREORDER(G, s)
for all $v \in V$ **do**
  if $v \notin R$ **then**
    remove v and incident edges
  end if
end for
return G

(a) DFS-based pruning

Require: Flow graph $G = (V, E)$; entry s
Ensure: Subgraph reachable from s
$R \leftarrow$ BFS-TREE(G, s)
for all $v \in V$ **do**
  if $v \notin R$ **then**
    remove v and incident edges
  end if
end for
return G

(b) BFS-based pruning

Despite the utility of sequential merging and heuristic pruning, both methods may fall short when dealing with extremely large CFGs where the sheer volume of nodes and edges obscures the underlying structure. Interval reduction addresses this by partitioning the CFG into subgraphs ("intervals"), each characterized by a unique entry point known as the header. The process starts with a header node—typically the entry node of a function or a newly identified control block—and iteratively incorporates any node whose predecessors all reside within the current interval. The iteration continues until no further nodes satisfy the condition, at which point the entire interval is collapsed into a single representative node. The construction's correctness rests on two formal properties: reachability is preserved by the quotient map, and the derived sequence forms a strictly decreasing chain of homomorphic images.

The empirical comparison confirms the practical implication of these properties: on reducible CFGs, IR collapses the graph to a single node (or to its irreducibility witness on non-reducible inputs), whereas SNM leaves intact every branch, join, and loop header. On the 12 real-world CFGs of Tier 2, IR achieves an average vertex reduction of 95% versus 25% for SNM. On the synthetic CFGs of Tier 3 (100–2000 nodes), the gap persists at $\approx 75\%$ versus $\approx 43\%$ even when the input contains randomly placed irreducible regions—because IR collapses every reducible portion and correctly halts on the residual. The structural property that distinguishes IR is the hierarchical derived sequence $G_0, G_1, \ldots, G_k$ (Proposition 4.1), which provides multiple levels of abstraction that single-pass methods cannot.

This property is particularly relevant for downstream learning applications. An explainability tool such as GNNExplainer run on G_i outputs an important subgraph at the interval level; the inverse Φ_i^{-1} lifts this back to the basic-block level of G_0. The lifted explanation is therefore aligned with a named program region (a loop, a branch). Such alignment is not possible with single-pass reduction methods.

5.1. Limitations

Interval reduction is a path-preserving method, which means it cannot collapse multi-entry cycles. On irreducible inputs the algorithm partially reduces the graph (collapsing every reducible sub-region) and halts on the residual structure, an outcome we documented on the two irreducible benchmarks of Tier 1, see Table 1 . The residual is itself useful: it localizes the multi-entry cycles that prevent further reduction, which an analyst can then handle through node-splitting or by reformulating the source program. We did not implement automatic node-splitting because it weakens the path-preservation guarantee on split nodes; integrating it as an optional post-pass is a natural extension.

The reduction ratio achieved by interval reduction is strongly dependent on graph structure. On properly nested CFGs (typical of real-world code, as confirmed by our Tier 2 stdlib results), reduction is essentially complete (95%

on average, often 100%). On CFGs with multi-entry cycles—rare in source code, but common in obfuscated binary inputs—the ratio drops sharply (Tier 3, around 75% on average, with residual subgraphs reflecting the irreducible regions). The algorithm's effectiveness is therefore a function of the structural regularity of the input, not a universal constant.

The interval construction requires that every interval be entered through its header. This assumption holds for intra-procedural CFGs by construction. It does not extend directly to interprocedural CFGs, where a function may be entered from multiple call sites; nor to CFGs derived from highly obfuscated binaries where indirect jumps create effective multi-entry regions. Extending the method to these settings is non-trivial and is left for future work.

Our implementation is in pure Python (with `networkx` for graph storage) and prioritizes correctness and readability over speed. As a consequence, the constant factor at large graph sizes is high: $|V| = 2000$ requires about 22 seconds per run (Figure 16(c)). This is a constant-factor issue, not an asymptotic one: a Cython or C implementation of the predecessor-counter optimization described in 4.7 would reduce the runtime by one to two orders of magnitude without changing the algorithm itself. We have not implemented this optimization in the present work.

6. Conclusion and Future Work

The simplification of control flow graphs sits at the intersection of two pressures that program analysis has always had to balance: the practical need to manage the scale at which modern programs are written, and the theoretical need to preserve the structural information on which any downstream reasoning depends. The history of CFG reduction reflects this tension. The classical interval-based methods of the 1970s were conceived to preserve path information rigorously, while the heuristic techniques that have come to dominate the recent literature are willing to sacrifice some of that fidelity in exchange for speed and aggressive compression. Both responses are reasonable, and each suits a different class of downstream task. The present work has argued that the classical approach retains a distinctive place between these poles, particularly where the analyses or learning models that consume the reduced graph depend on the very relations; dominance, reachability and hierarchical nesting, that the heuristic alternatives are not designed to maintain.

The case for that place is made along three dimensions. A practical implementation establishes that interval reduction can be expressed as a small set of composable steps amenable to inspection and re-use; a careful formal treatment establishes that the construction preserves the structural properties on which downstream uses rely; and a controlled study establishes that the resulting reductions are substantial in practice, scaling consistently across graphs of widely varying size and origin. Each of these dimensions has been pursued in earlier literature, but rarely in combination; bringing them together is the principal contribution we offer.

Looking outward, the natural continuation of this work lies in its integration with the modern machinery of graph-based program analysis, particularly the graph neural network architectures now central to malware detection and software defect localization. A reduction whose levels preserve the program's loop and branch structure offers, in principle, a route to explanations whose interpretation can be lifted back to the original code. Whether that route delivers in practice, and how it interacts with the obfuscation patterns observed in real binaries, is a question we leave for follow-up work, along with the corresponding engineering questions about implementation efficiency and the treatment of irreducible inputs.

In fact, from one side, the Proposition 4.1 established that interval reduction preserves dominance relations, loop hierarchy, and forward data-flow properties, which are fundamental in malware-oriented control-flow analysis because they retain semantic structures commonly exploited in malicious binaries, including obfuscation loops, dispatcher patterns, and propagation behaviors.

Malware detection requires simplification methods that are semantics-preserving as malware-analysis tools cannot tolerate reductions that destroy execution semantics. A comparison to Sequential Node Merging is useful and a good candidate for preserving hierarchy as sometimes reducing graph size arbitrarily may destroy signatures, but introducing other procedures could preserve hierarchy better than naive merging since malware detection systems

rely on structural patterns. Preserving hierarchy means the reduction keeps the behavioral organization needed for graph matching, anomaly detection, behavioral classification. From another side, even with interval reduction, the results especially when we increased the number of nodes to larger values such as 250, 500, 1000, 2000, the results indicated that our algorithms offered a principled way to reduce graph complexity while preserving the hierarchical and semantic organization of control flow, supporting its application to malware detection, binary analysis, and other security-oriented program analysis tasks.

After all, this work still supports scalability, robustness, and generality. The reduced graphs remain suitable for downstream malware-analysis tasks. A complete empirical validation on datasets for detecting software defection or malware is left for future work.

Acknowledgment

A special thanks to the editors for their effort and time. We would also like to thank the anonymous referees for their valuable recommendations, which greatly helped us improve the content of our paper.

REFERENCES

1. S. Abdelalim, A. Cherkaoui, A. Lkoiaza, I. Elmouki, and N. Abghour, *Advancing Blockchain Security Using Graph Theory: A Python Programming Perspective*, Finite Abelian Groups, Elliptic Curves, Blockchain With Hashing and Graphs, pp. 279–293, 2025.
2. S. Abdelalim, I. Elmouki, and N. Abghour, *Introductory Essentials to Blockchain, Hashing and Graphs*, Finite Abelian Groups, Elliptic Curves, Blockchain with Hashing and Graphs, pp. 170–206, 2025.
3. A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyang, and D. Mohaisen, *Soteria: Detecting adversarial examples in control flow graph–based malware classifiers*, Proceedings of the 2020 IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 888–898, 2020.
4. A. V. Aho and J. D. Ullman, *Node listings for reducible flow graphs*, Proceedings of the 7th Annual ACM Symposium on Theory of Computing, Albuquerque, NM, pp. 177–185, 1975.
5. F. E. Allen, *Control flow analysis*, SIGPLAN Notices, vol. 5, no. 7, pp. 1–19, 1970.
6. F. E. Allen and J. Cocke, *A program data flow analysis procedure*, Communications of the ACM, vol. 19, no. 3, pp. 137–147, 1976.
7. A. S. S. Anju, P. Harmya, N. Jagadeesh, and R. Darsana, "Malware Detection using Assembly Code and Control Flow Graph Optimization," in *Proceedings of A2CWiC 2010*, ACM, India, 2010.
8. A. Bertolino and M. Marré, *Automatic generation of path covers based on the control flow analysis of computer programs*, IEEE Transactions on Software Engineering, vol. 20, no. 12, pp. 885–899, 1994.
9. G. Bonfante, M. Kaczmarek, and J.-Y. Marion, *Control flow to detect malware*, Inter-Regional Workshop on Rigorous System Development and Analysis, 2007.
10. D. Bruschi, E. Carnieri, E. Ferrari, and L. Martignoni, *Detecting self-mutating malware using control-flow graph matching*, Proceedings of DIMVA 2006, pp. 129–143.
11. X. Cheng, Y. Wang, W. Zhou, X. Wang, and J. Wang, *Software Fault Detection for Sequencing Constraint Defects*, IJPE Online, vol. 16, no. 11, pp. 1814–1825, 2020.
12. J. Ferrante, K. J. Ottenstein, and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319–349, 1987.
13. Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, *Malware detection by control-flow graph level representation learning with graph isomorphism network*, IEEE Access, vol. 10, pp. 111830–111841, 2022.
14. R. Gold, *Control flow graphs and code coverage*, International Journal of Applied Mathematics & Computer Science, vol. 20, no. 4, pp. 739–749, 2010.
15. R. Gold, *Decision graphs and their application to software testing*, ISRN Software Engineering, 2013, Article ID 604279.
16. R. Gold, *Reductions of control flow graphs*, International Journal of Computer, Electrical, Automation, Control and Information Engineering, vol. 8, no. 3, pp. 427–434, 2014.
17. M. J. Harrold and G. Rothermel, *Performing data flow testing on classes*, Proceedings of FSE 1994, pp. 154–163.
18. M. S. Hecht, *Flow analysis of computer programs*, North-Holland, 1977.
19. J. D. Herath, P. P. Wakodikar, P. Yang, and G. Yan, *CFGExplainer: Explaining Graph Neural Network-Based Malware Classification from Control Flow Graphs*, Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 172–184, 2022.
20. Y.-F. Ma and M. Li, *The flowing nature matters: Feature learning from the control flow graph of source code for bug localization*, Machine Learning, vol. 111, pp. 853–870, 2022.
21. H. Mohammadian, G. Higgins, S. Ansong, R. Razavi-Far, and A. A. Ghorbani, *Explainable malware detection through integrated graph reduction and learning techniques*, arXiv:2412.03634, 2024.
22. A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, *Anomaly detection using program control flow graph mining from execution logs*, Proceedings of SIGKDD 2016, pp. 1995–2004.
23. M. H. Nguyen, Q.-T. Phung, and D.-H. Nguyen, *Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning*, Computers & Security, vol. 79, pp. 330–345, 2018.
24. M. R. Paige, *On partitioning program graphs*, IEEE Transactions on Software Engineering, SE-3(6), pp. 386–393, 1977.

25. H.-D. Phan, Q.-T. Tho, and D.-H. Nguyen, *Software homology detection with software motifs based on function-call graphs*, KSII Transactions on Internet and Information Systems, vol. 12, no. 3, pp. 1346–1365, 2018.
26. S. Rapps and E. J. Weyuker, *Data flow analysis techniques for test data selection*, Proceedings of ICSE 1982, pp. 272–278.
27. T. Reps, S. Horwitz, and M. Sagiv, *Precise interprocedural dataflow analysis via graph reachability*, Proceedings of POPL 1995, pp. 49–61.
28. J. Studer, J. Taylor, and K. Lai, *py2cfg: A Python 3 control flow graph generator, version 0.5.1*, https://py2cfg.readthedocs.io/, 2020.
29. P. Wu, J. Wang, and B. Tian, *Software Homology Detection With Software Motifs Based on Function-Call Graph*, IEEE Access, vol. 6, pp. 19007–19017, 2018.
30. P. Wu, L. Yin, X. Du, L. Jia, and W. Dong, *Graph-based vulnerability detection via extracting features from sliced code*, Proceedings of IEEE QRS-C 2020, pp. 38–45.
31. Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, *GNNExplainer: Generating explanations for graph neural networks*, NeurIPS 2019, pp. 9240–9251.
32. J. Zhang, H. Sun, H. Zhang, X. Wang, X. Liu, and Y. Liu, *Detecting Condition-Related Bugs with Control Flow Graph Neural Network*, Proceedings of ISSTA '23, Seattle, WA, 2023.