Parameterized Algorithms for Scalable Interprocedural Data-flow Analysis

Ahmed K. Zaher

July 31st, 2023

香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

Agenda

1. Motivation
2. The IFDS framework
3. Sparsity parameters
4. Solving IFDS problems
5. Experimental results

Table of Contents

Maiden flight of Ariane 5

Back in June of 1996, the Ariane 5 rocket had its first launch.

Maiden flight of Ariane 5

Back in June of 1996, the Ariane 5 rocket had its first launch.

Maiden flight of Ariane 5

40 seconds later...

Maiden flight of Ariane 5

The rocket self destruct due to a software error: an unsafe conversion from 64-bit float to a 16-bit integer was not caught and led to uncontrollable behavior.

This error cost US$370 million.

Null pointers

A null pointer is a pointer that does not point to anything.

Null pointers

A null pointer is a pointer that does not point to anything.

"You either have to check every reference, or you risk disaster."

Null pointers

A null pointer is a pointer that does not point to anything.

"You either have to check every reference, or you risk disaster."

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

- Tony Hoare

Static program analysis

- Software bugs can incur great costs.
- Programs can be too complicated for humans to catch all bugs.
- We need more formal, automated, methods to do this for us.

Static program analysis

- Software bugs can incur great costs.
- Programs can be too complicated for humans to catch all bugs.
- We need more formal, automated, methods to do this for us.

Static program analysis: the science of automatically finding bugs in programs without running them.

Uses of static program analysis

Static program analysis attempts to answer question like:

- Does the program use a variable `x` before it is initialized?
- Can the program have a null-pointer dereferencing?
- If expression `e` is inside a loop, does `e`'s value depend on the loop iteration?

Uses of static program analysis

Static program analysis attempts to answer question like:

- Does the program use a variable `x` before it is initialized?
- Can the program have a null-pointer dereferencing?
- If expression `e` is inside a loop, does `e`'s value depend on the loop iteration?

Applications:

- Optimizing compilers.
- IDEs.
- Verification of safety-critical systems.
  - In 2003, Astrée was used to verify the flight control software of Airbus A340.

Overview

We will consider the IFDS framework, which captures a large class of useful static analyses such as:

- possibly-uninitialized variables,
- null-pointer,
- reaching definitions,
- available expressions,
- live variables, and
- dead-code elimination.

Overview

We will consider the IFDS framework, which captures a large class of useful static analyses such as:

- possibly-uninitialized variables,
- null-pointer,
- reaching definitions,
- available expressions,
- live variables, and
- dead-code elimination.

Setting:

- *Large scale.* We have a large codebase (e.g., in Google/Meta) on which we want to perform some IFDS analysis.
- *On-demand.* We receive a large stream of queries (e.g., from developers) inquiring about the analysis result between two particular statements in the codebase.

Overview

We will consider the IFDS framework, which captures a large class of useful static analyses such as:

- possibly-uninitialized variables,
- null-pointer,
- reaching definitions,
- available expressions,
- live variables, and
- dead-code elimination.

Setting:

- *Large scale.* We have a large codebase (e.g., in Google/Meta) on which we want to perform some IFDS analysis.
- *On-demand.* We receive a large stream of queries (e.g., from developers) inquiring about the analysis result between two particular statements in the codebase.

Standard IFDS algorithms. Authors of IFDS (POPL'95 [1], FSE'95 [2]) gave algorithms to achieve this, but they do not scale to large codebases with over 10^5 LoC.

Overview

Idea: exploit sparsity of graphs appearing in the problem.

Graphs that arise in the problem often have nice structures that can enable faster algorithms.

Overview

Idea: exploit sparsity of graphs appearing in the problem.

Graphs that arise in the problem often have nice structures that can enable faster algorithms.

Chatterjee (ESOP'20) [3] took this approach.

- They exploited low treewidth of control-flow graphs.
- Pro: fast preprocessing and query time.
- Con: they solve a restricted case of the problem.

Overview

Idea: exploit sparsity of graphs appearing in the problem.

Graphs that arise in the problem often have nice structures that can enable faster algorithms.

Chatterjee (ESOP'20) [3] took this approach.

- They exploited low treewidth of control-flow graphs.
- Pro: fast preprocessing and query time.
- Con: they solve a restricted case of the problem.

This work: exploit low treedepth of call graphs to solve the general case.

Contribution

- *Identify a new sparsity parameter: treedepth of the program's call graph.*

Contribution

- *Identify a new sparsity parameter: treedepth of the program's call graph.*
- *Solve the general case of IFDS problems.* We exploit this new parameter to develop fast algorithm that extends that of Chatterjee's and solves the general case of IFDS.

Contribution

- *Identify a new sparsity parameter: treedepth of the program's call graph.*
- *Solve the general case of IFDS problems.* We exploit this new parameter to develop fast algorithm that extends that of Chatterjee's and solves the general case of IFDS.
- *Experimental results.* We experimentally showed on real-world programs that:
  - Call graphs do have low treedepth.
  - Our algorithm outperforms the standard algorithms of [1, 2].

Contribution

- *Identify a new sparsity parameter: treedepth of the program's call graph.*
- *Solve the general case of IFDS problems.* We exploit this new parameter to develop fast algorithm that extends that of Chatterjee's and solves the general case of IFDS.
- *Experimental results.* We experimentally showed on real-world programs that:
  - Call graphs do have low treedepth.
  - Our algorithm outperforms the standard algorithms of [1, 2].

For a program of n lines:

Approach	General?	Preprocessing	Query
Reps et. al. (POPL'95)	✓	$O(n)$	
Horwitz et. al. (FSE'95)	✓	$O(n)$	
Chatterjee et. al. (ESOP'20)	✗	$O(n)$	$O(1)$
Our result	✓	$O(n)$	$O(1)$

Table of Contents

Abstractions for programs

We'll need 3 abstractions to formalize the structure of a program:

1. Control-flow graphs.
2. Supergraphs.
3. Call graphs.

Control flow graphs (CFGs)

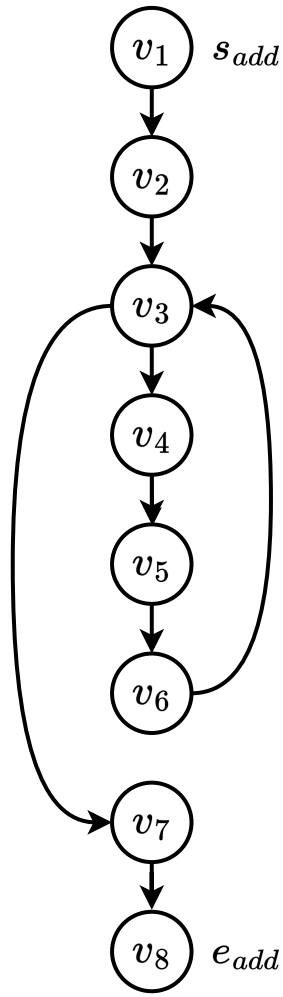
A program P with a single function f is formalized by a control-flow graph $G_f = (V_f, E_f)$:

Control flow graphs (CFGs)

A program P with a single function f is formalized by a control-flow graph $G_f = (V_f, E_f)$:

- V_f corresponds to statements of P.
- $(u_1, u_2) \in E_f$ represents flow of control from u_1 to u_2.
- G_f has a *start* vertex s_f and *exit* vertex e_f.

```
1 int add(int a, int b) {
2     int sum = a;
3     while (b > 0) {
4         sum = sum + 1;
5         b = b - 1;
6     }
7     return sum;
8 }
```

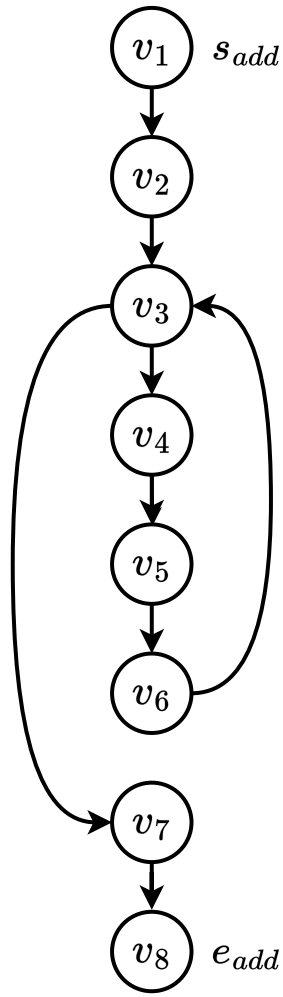


Control flow graphs (CFGs)

A program P with a single function f is formalized by a control-flow graph $G_f = (V_f, E_f)$:

- V_f corresponds to statements of P.
- $(u_1, u_2) \in E_f$ represents flow of control from u_1 to u_2.
- G_f has a *start* vertex s_f and *exit* vertex e_f.

```
1 int add(int a, int b) {
2     int sum = a;
3     while (b > 0) {
4         sum = sum + 1;
5         b = b - 1;
6     }
7     return sum;
8 }
```



Observe:

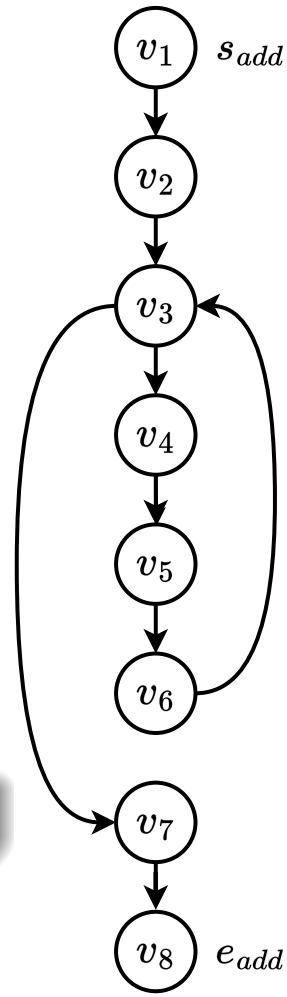
- A path in $G_f \equiv$ an execution of P.
- The paths in G_f completely characterize f's behavior at runtime.

Control flow graphs (CFGs)

A program P with a single function f is formalized by a control-flow graph $G_f = (V_f, E_f)$:

- V_f corresponds to statements of P.
- $(u_1, u_2) \in E_f$ represents flow of control from u_1 to u_2.
- G_f has a *start* vertex s_f and *exit* vertex e_f.

```
1 int add(int a, int b) {
2     int sum = a;
3     while (b > 0) {
4         sum = sum + 1;
5         b = b - 1;
6     }
7     return sum;
8 }
```



Analyzing $P \equiv$ compute the meet-over-all-paths (MOP).

Supergraphs

Program consisting of functions $f_1, \ldots, f_k$ is formalized by a supergraph G

$$G \equiv \text{ CFGs } G_{f_1}, \ldots, G_{f_k} + \textit{interprocedural edges}$$

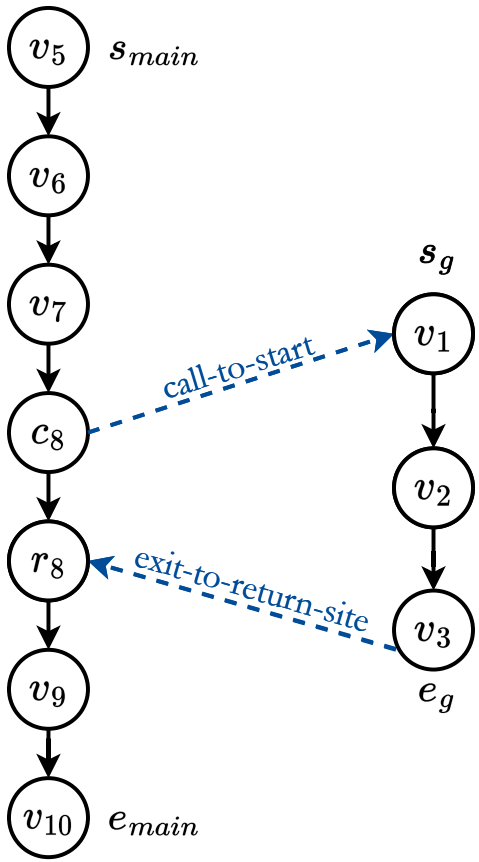
Supergraphs

Program consisting of functions $f_1, \ldots, f_k$ is formalized by a supergraph G

$$G \equiv \text{ CFGs } G_{f_1}, \ldots, G_{f_k} + \textit{interprocedural edges}$$

- A function call from f to $f' \equiv$ two vertices c and r in f.
- Intrerprocedural edges: $(c, s_{f'})$ and $(e_{f'}, r)$.

```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```



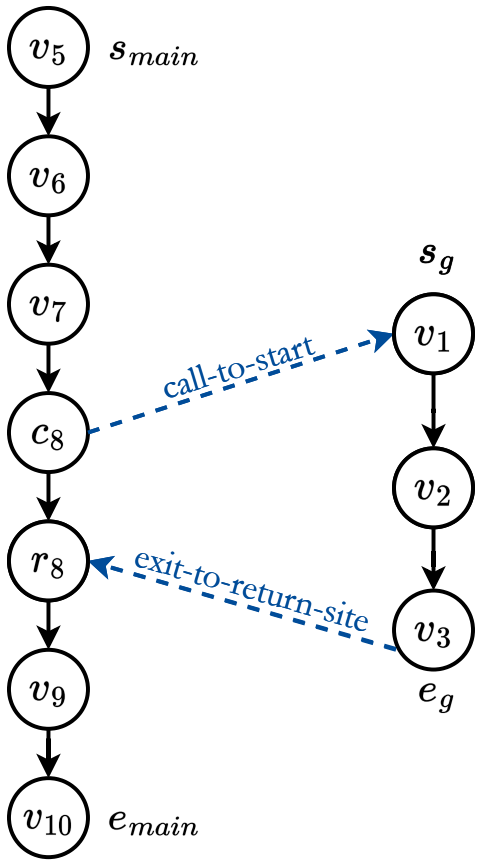
Supergraphs

Program consisting of functions $f_1, \ldots, f_k$ is formalized by a supergraph G

$$G \equiv \text{ CFGs } G_{f_1}, \ldots, G_{f_k} + \textit{interprocedural edges}$$

- A function call from f to $f' \equiv$ two vertices c and r in f.
- Intrerprocedural edges: $(c, s_{f'})$ and $(e_{f'}, r)$.

```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```



Question: a path in $G \equiv$ an execution of P?

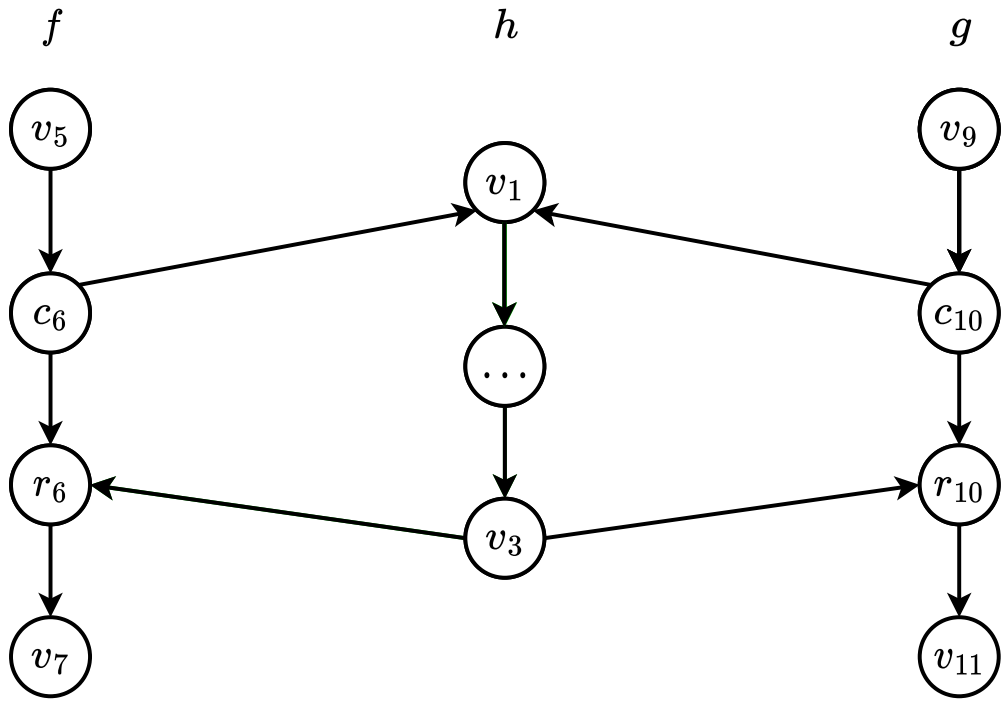
Invalid paths

- In a CFG, any path can be realized.
- In a supergraph, need to be more careful..

Invalid paths

- In a CFG, any path can be realized.
- In a supergraph, need to be more careful..

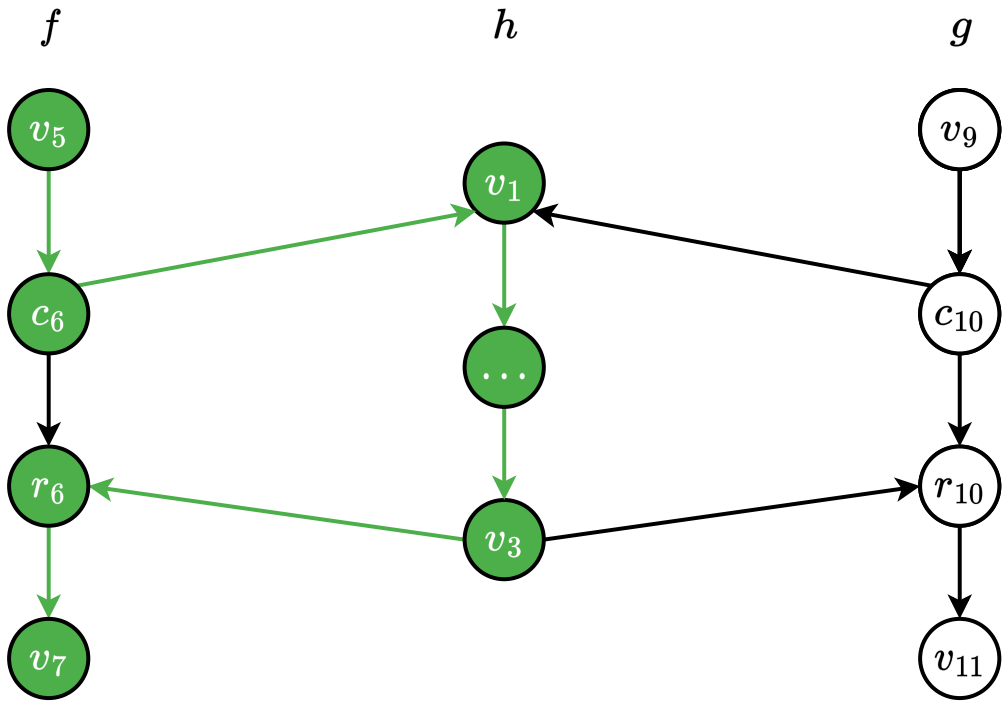
```
1  void h() {
2          ...
3  }
4
5  int f() {
6          h();
7  }
8
9  int g() {
10         h();
11 }
```



Invalid paths

- In a CFG, any path can be realized.
- In a supergraph, need to be more careful..

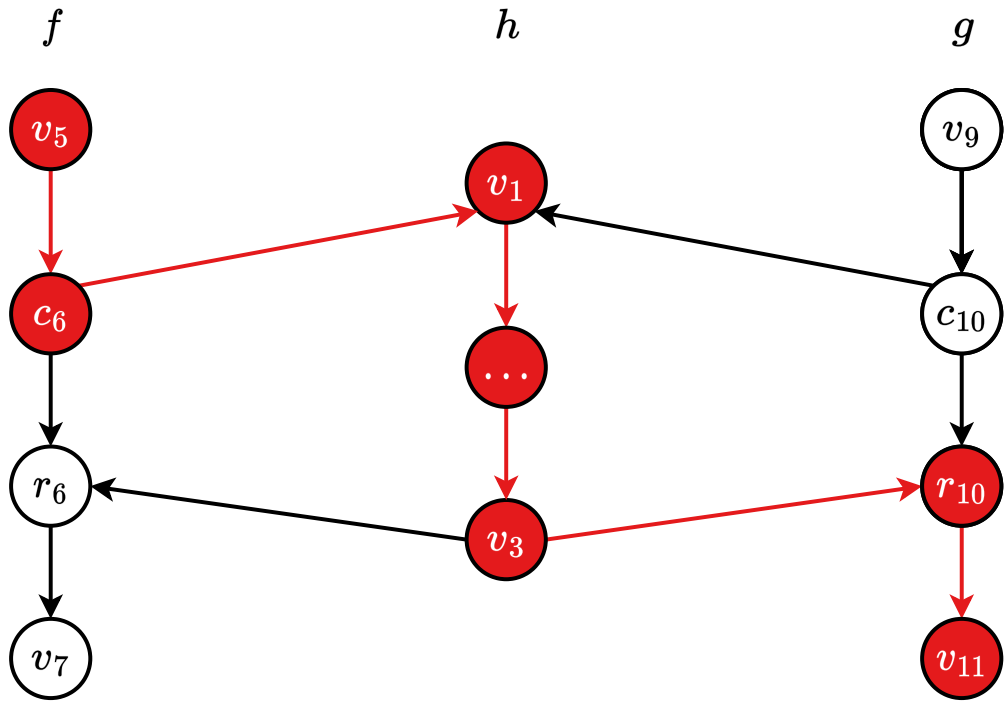
```
1  void h() {
2          ...
3  }
4
5  int f() {
6          h();
7  }
8
9  int g() {
10         h();
11 }
```



Invalid paths

- In a CFG, any path can be realized.
- In a supergraph, need to be more careful..

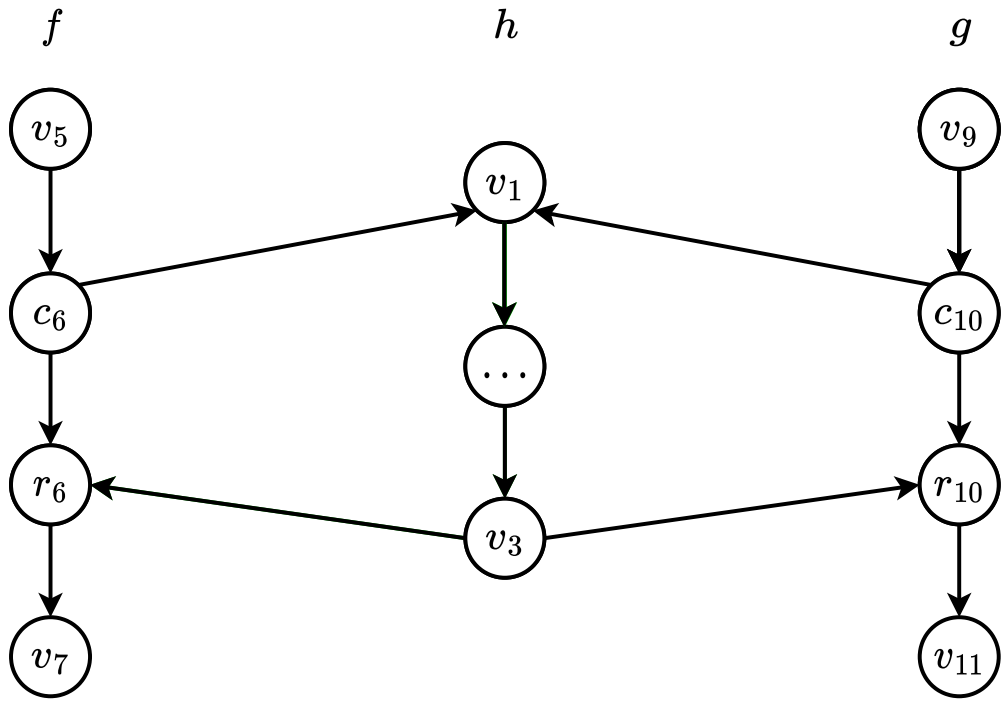
```
1  void h() {
2          ...
3  }
4
5  int f() {
6          h();
7  }
8
9  int g() {
10         h();
11 }
```



Invalid paths

- In a CFG, any path can be realized.
- In a supergraph, need to be more careful..

```
1  void h() {
2          ...
3  }
4
5  int f() {
6          h();
7  }
8
9  int g() {
10         h();
11 }
```

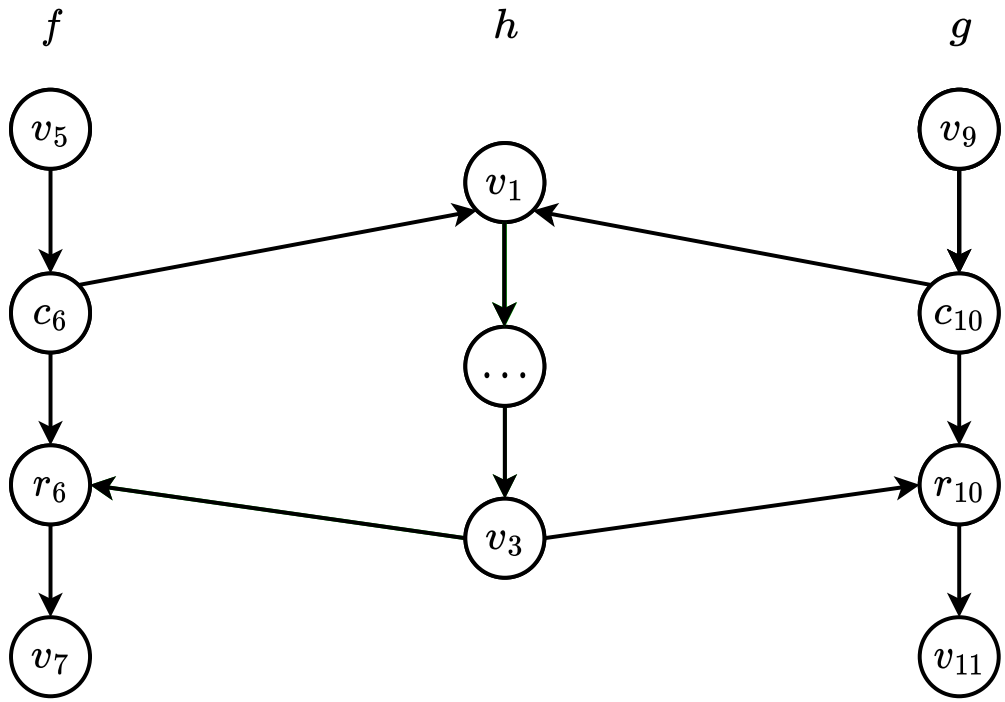


- An interprocedurally valid path (IVP) is a path where returns are to the correct matching calls.
- An IVP in $G \equiv$ an execution of P.

Invalid paths

- In a CFG, any path can be realized.
- In a supergraph, need to be more careful..

```
1  void h() {
2          ...
3  }
4
5  int f() {
6          h();
7  }
8
9  int g() {
10         h();
11 }
```



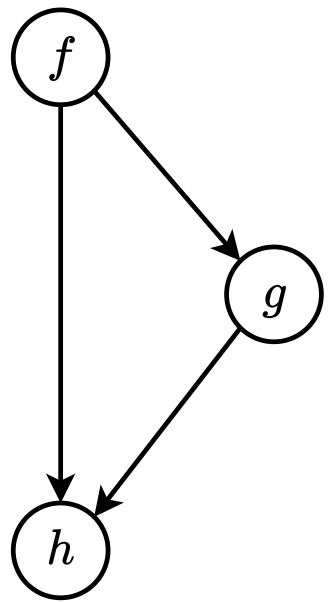
Analyzing $P \equiv$ compute the meet-over-all-valid-paths (MIVP).

Call graphs

A *Call graph* $C = (F, E_C)$ has:

- Vertices are functions of the program.
- $(f, f') \in E_C \equiv$ there is a call from some line in f to f'.

```
1  void h() {}
2
3  void f() {
4          g();
5          h();
6  }
7
8  void g() {
9          h();
10 }
```

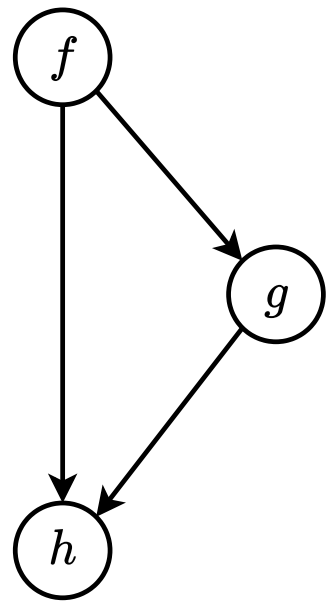


Call graphs

A *Call graph* $C = (F, E_C)$ has:

- Vertices are functions of the program.
- $(f, f') \in E_C \equiv$ there is a call from some line in f to f'.

```
1  void h() {}
2
3  void f() {
4          g();
5          h();
6  }
7
8  void g() {
9          h();
10 }
```



- The call graph can be inferred from the supergraph.
- Describes program's behavior at the function level.

IFDS problems

In an IFDS problem, the input is:

- Supergraph $G = (V, E)$.
- Finite set of *data-facts* D.
- Each edge $e = (I, I') \in E$ has a flow function $M_e : 2^D \to 2^D$.
- *Meet* operator $\sqcap \in \{\cup, \cap\}$.
- M_e distributes over $\sqcap$, i.e., $M_e(D_1 \sqcap D_2) = M_e(D_1) \sqcap M_e(D_2)$.

To solve the IFDS problem: compute at each program point (vertex) which data-facts hold.

IFDS problems: example

Example: null-pointer analysis.

- $D =$ set of variables in the program.
- A solution: for every $l \in V$, compute $S_l \subseteq D$, the set of variable that may be null after l if we start execution from `main`.

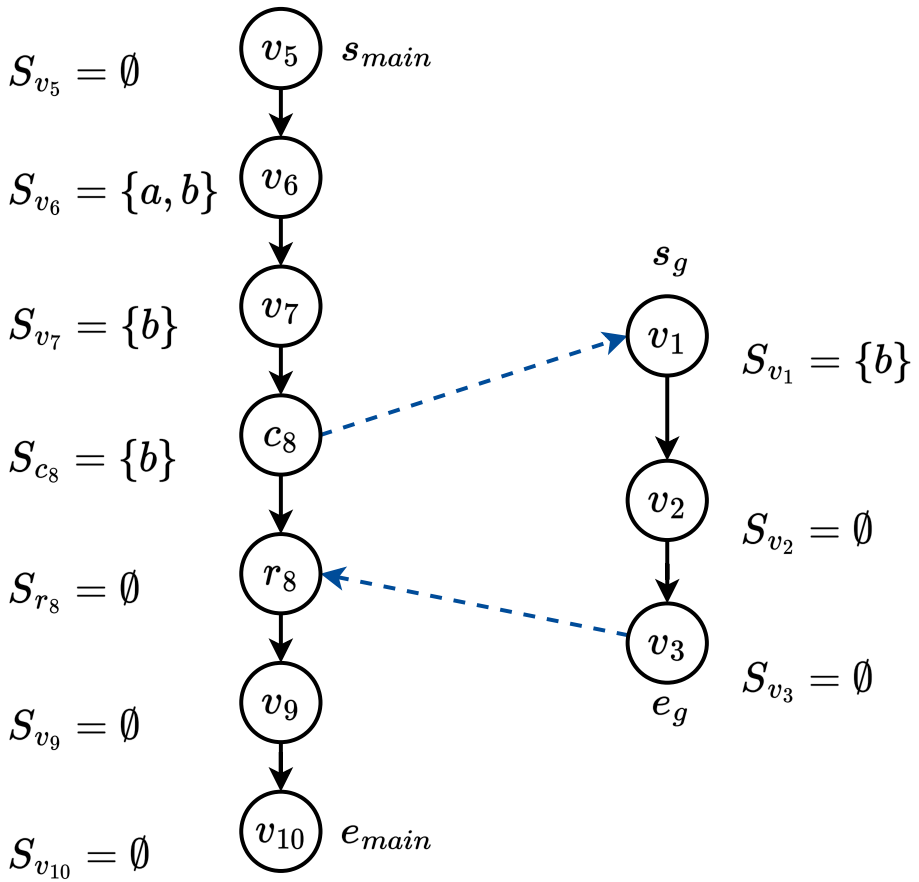
```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```

IFDS problems: example

Example: null-pointer analysis.

- $D =$ set of variables in the program.
- A solution: for every $I \in V$, compute $S_I \subseteq D$, the set of variable that may be null after I if we start execution from `main`.

```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```



IFDS problems: formalizing a solution

More formally:

- For a path $\pi = e_1 \cdot e_2 \cdots e_k$ in G, define:

$$M_\pi = M_{e_k} \circ M_{e_k} \circ \cdots \circ M_{e_1}.$$

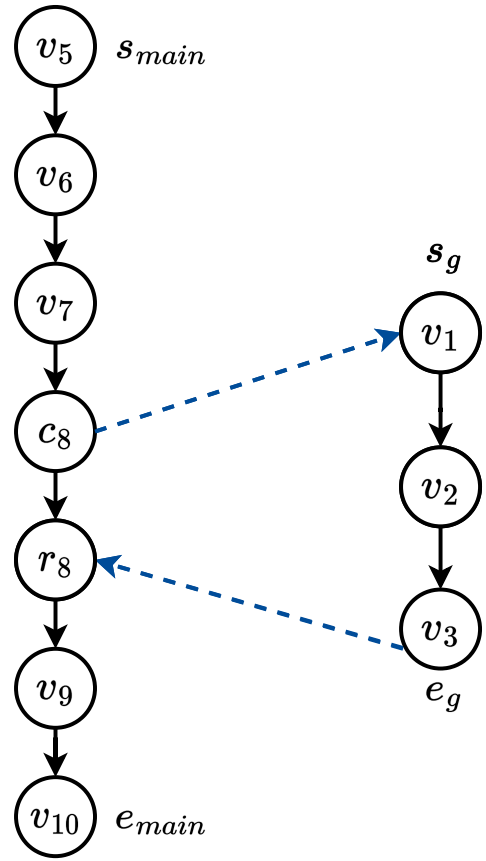
- For $u_1, u_2 \in V$, define:

$$\texttt{IVP}(u_1,u_2) = \{P \mid P \text{ is an IVP from } u_1 \text{ to } u_2 \text{ in } G\}.$$

- For $u_1, u_2 \in V$ and $D_1 \subseteq D$, we want to compute:

$$\texttt{MIVP}(u_1, D_1, u_2) := \bigsqcap_{\pi \in \texttt{IVP}(u_1,u_2)} M_\pi(D_1)$$

- We'll assume wlog that $\sqcap = \cup$.



IFDS problems

IFDS problem

Input: $\langle G = (V, E), D, \{M_e\}_{e \in E} \rangle$ queries of the form $\langle u_1, D_1, u_2 \rangle$.

Output: for each query $\langle u_1, D_1, u_2 \rangle$, return:

$$\texttt{MIVP}(u_1, D_1, u_2).$$

IFDS problems

IFDS problem

Input: $\langle G = (V, E), D, \{M_e\}_{e \in E} \rangle$ queries of the form $\langle u_1, D_1, u_2 \rangle$.

Output: for each query $\langle u_1, D_1, u_2 \rangle$, return:

$$\texttt{MIVP}(u_1, D_1, u_2).$$

Objective: develop an algorithm that has:

- Lightweight preprocessing phase after which,
- Queries can be answered as fast as possible.

IFDS problems

IFDS problem

Input: $\langle G = (V, E), D, \{M_e\}_{e \in E} \rangle$ queries of the form $\langle u_1, D_1, u_2 \rangle$.

Output: for each query $\langle u_1, D_1, u_2 \rangle$, return:

$$\texttt{MIVP}(u_1, D_1, u_2).$$

Objective: develop an algorithm that has:

- Lightweight preprocessing phase after which,
- Queries can be answered as fast as possible.

In the remainder of this talk, we will give a series of observations, each of which give us a simpler problem to solve.

Where we are

Computing
meet-over-all-valid-paths
in the supergraph

Where we are

But first, let's introduce sparsity parameters..

Table of Contents

Treewidth

- *Treewidth* measures "tree-likeness" of a graph.
- Graphs of small treewidth are tree-like.
- Graph problems are typically easier on trees.
- Similarly, graph problems are also typically easier on graphs of low treewidth.

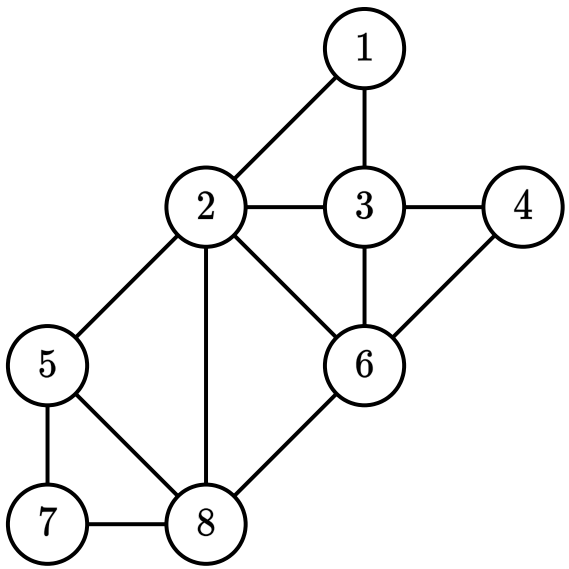
Treewidth

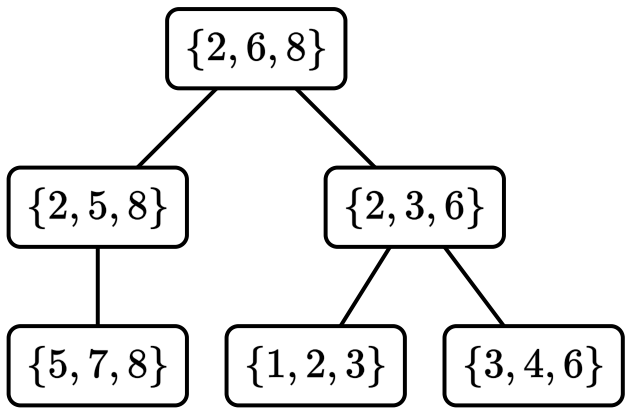
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





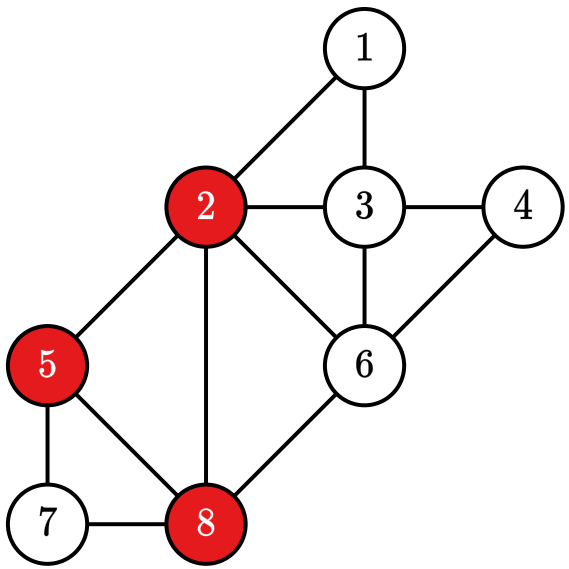
Treewidth

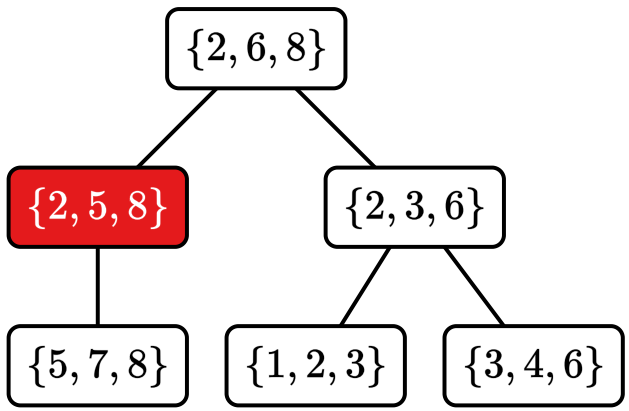
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





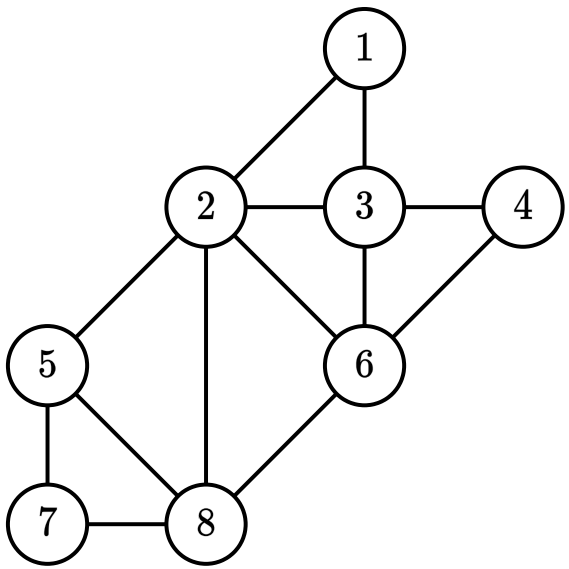
Treewidth

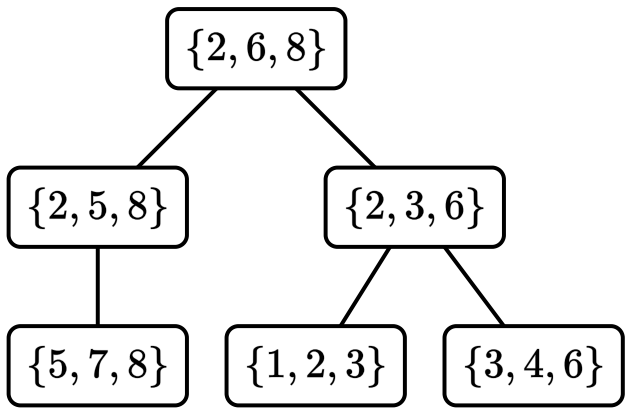
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





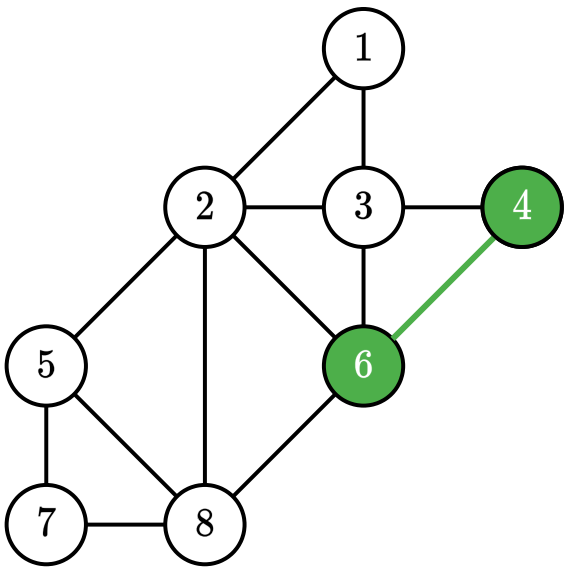
Treewidth

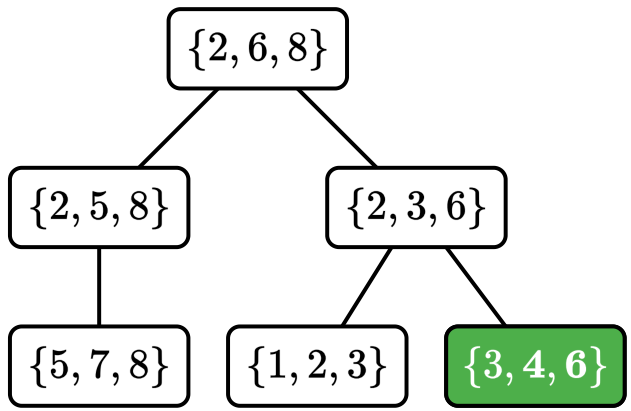
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





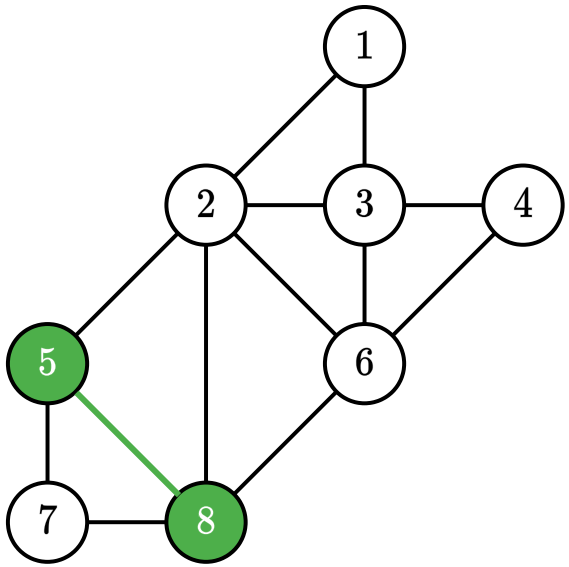
Treewidth

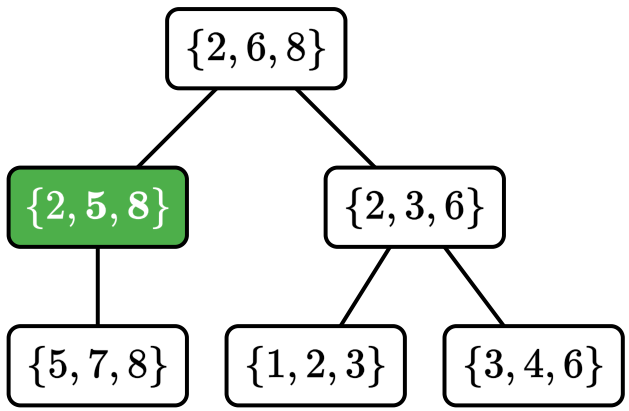
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





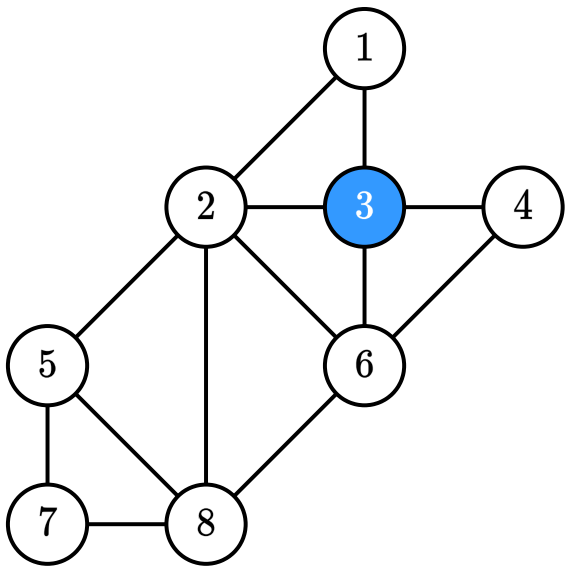
Treewidth

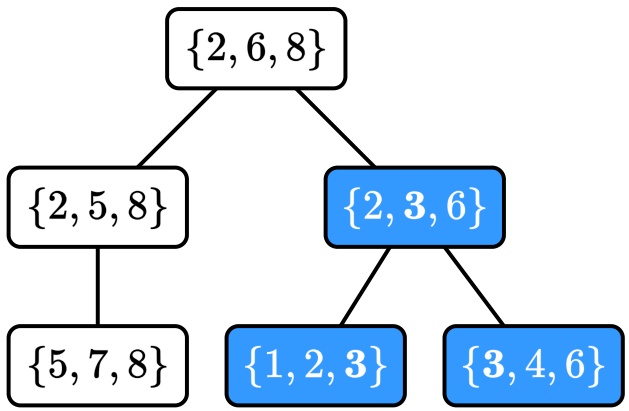
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





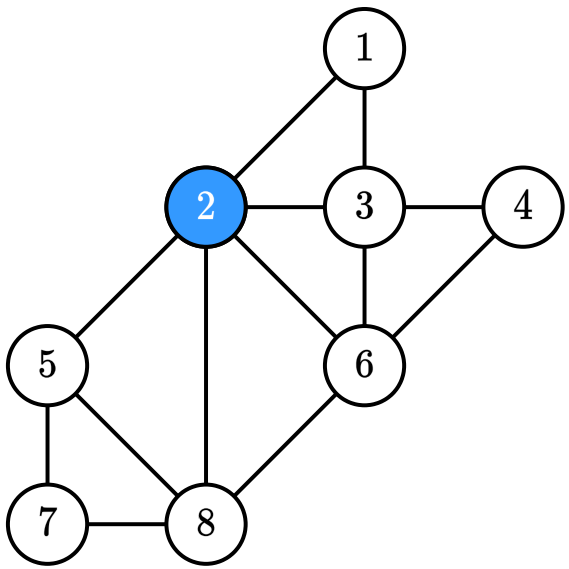
Treewidth

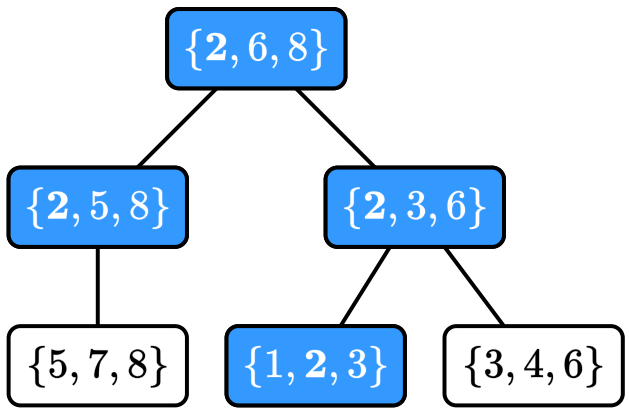
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





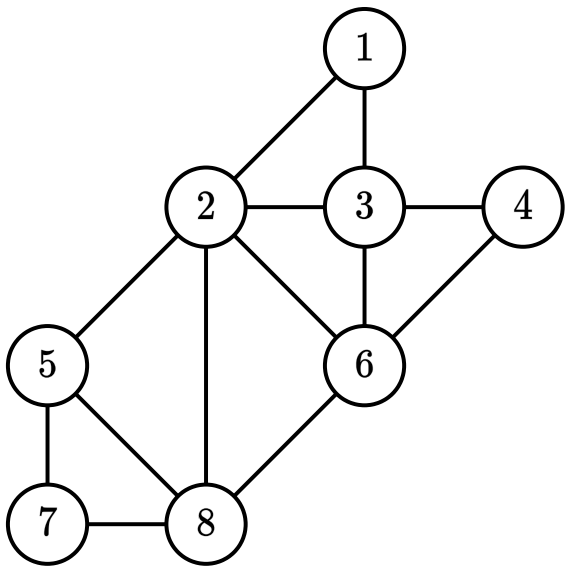
Treewidth

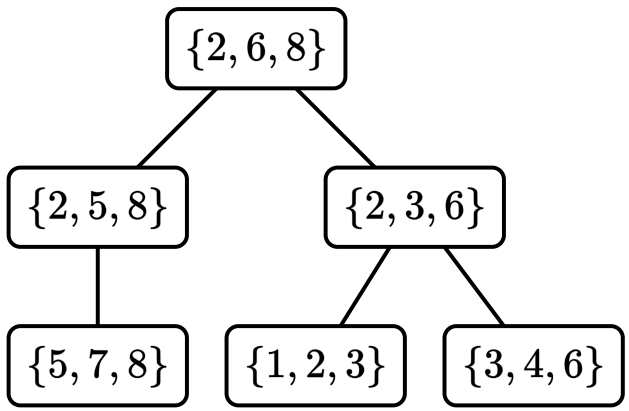
Tree Decompositions (TDs)

Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.





Treewidth

Tree Decompositions (TDs)

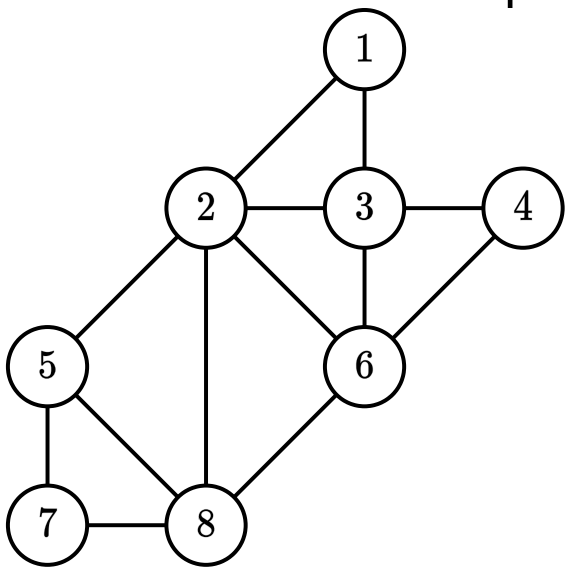
Given $G = (V, E)$, a *tree decomposition* of G is a tree $T = (\mathfrak{B}, E_T)$:

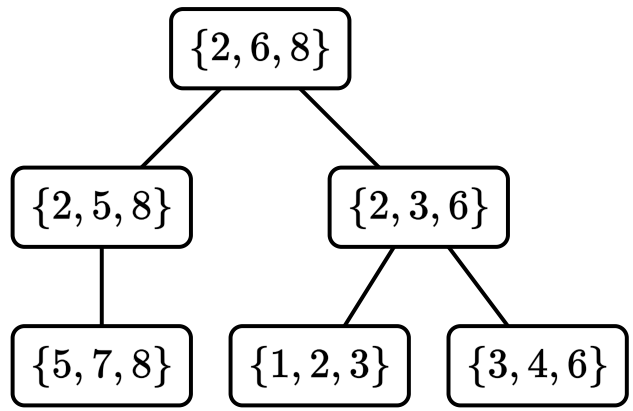
1. Every node $b \in \mathfrak{B}$ of the tree T has a corresponding *bag* $V_b \subseteq V$.
2. $\bigcup_{b \in \mathfrak{B}} V_b = V$.
3. $\forall u, v \in V, \{u, v\} \in E \implies \exists b \in \mathfrak{B} \ \ \{u, v\} \subseteq V_b$.
4. $\forall v \in V, \{b \in \mathfrak{B} | v \in V_b\}$ forms a connected subtree of T.

The width of T is $\max_{b \in \mathfrak{B}} |V_b| - 1$.

The *treewidth* of G is the smallest width among all TDs over G.

This TD has width 2 and is optimal $\implies$ the graph has treewidth 2.



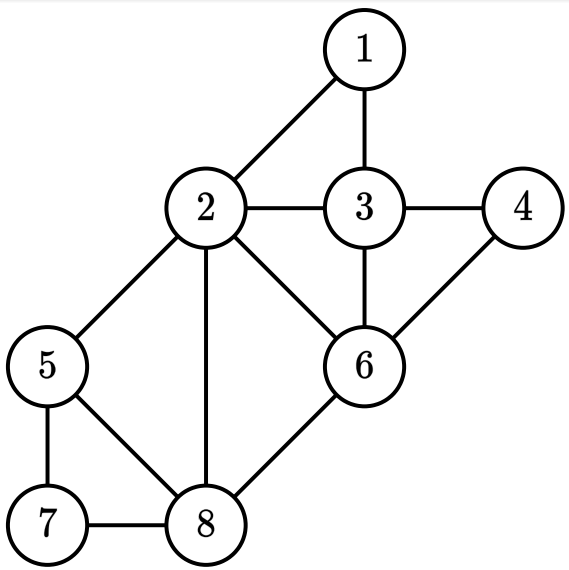


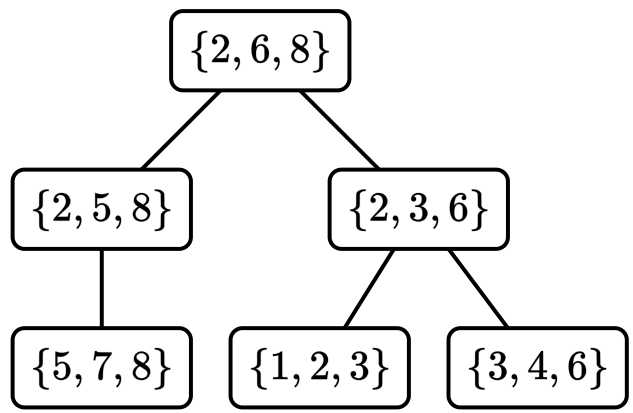
Cut property of TDs

Cut property of TDs

Consider $G = (V, E)$ and a TD $T = (\mathfrak{B}, E_T)$ of it. Pick any edge $e = \{b, b'\} \in E_T$. Removing e will break T into two connected subtrees T^b and $T^{b'}$, containing b and b', respectively. We have that:

$$V_b \cap V_{b'} \text{ separates } \bigcup_{c \in T^b} V_c \text{ from } \bigcup_{c \in T^{b'}} V_c.$$



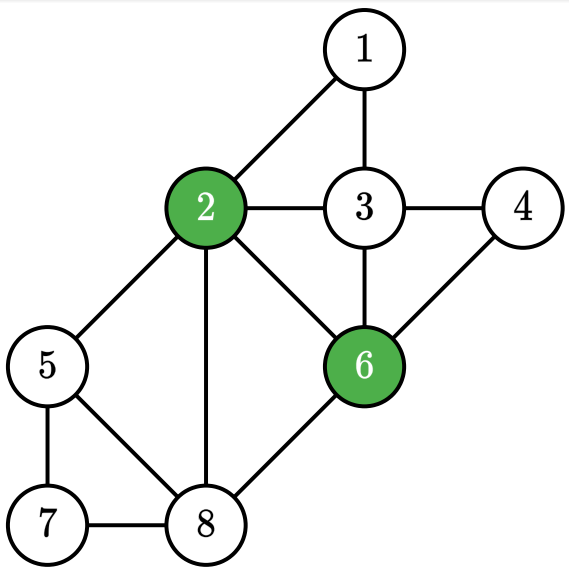


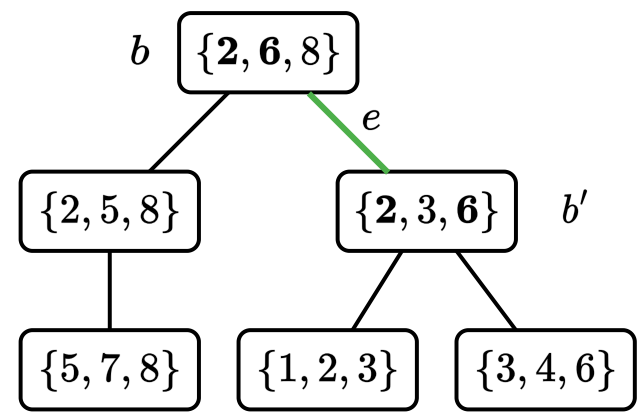
Cut property of TDs

Cut property of TDs

Consider $G = (V, E)$ and a TD $T = (\mathfrak{B}, E_T)$ of it. Pick any edge $e = \{b, b'\} \in E_T$. Removing e will break T into two connected subtrees T^b and $T^{b'}$, containing b and b', respectively. We have that:

$$V_b \cap V_{b'} \text{ separates } \bigcup_{c \in T^b} V_c \text{ from } \bigcup_{c \in T^{b'}} V_c.$$



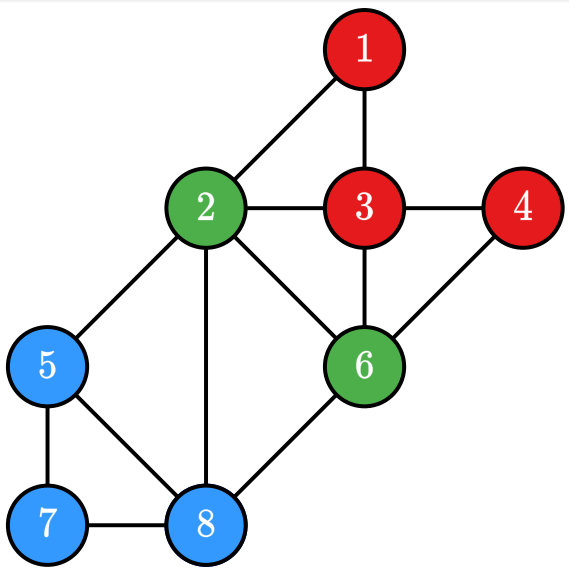


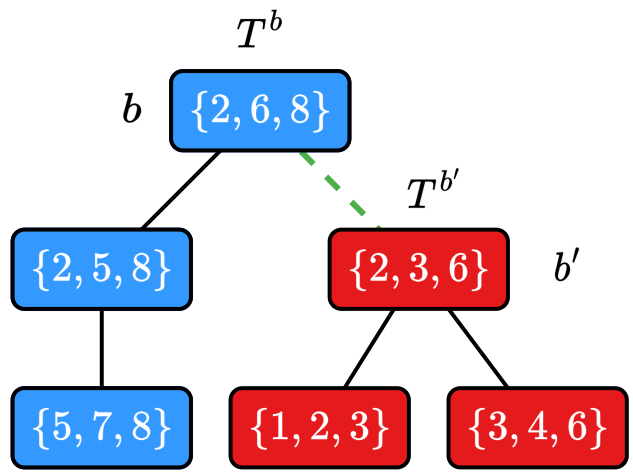
Cut property of TDs

Cut property of TDs

Consider $G = (V, E)$ and a TD $T = (\mathfrak{B}, E_T)$ of it. Pick any edge $e = \{b, b'\} \in E_T$. Removing e will break T into two connected subtrees T^b and $T^{b'}$, containing b and b', respectively. We have that:

$$V_b \cap V_{b'} \text{ separates } \bigcup_{c \in T^b} V_c \text{ from } \bigcup_{c \in T^{b'}} V_c.$$



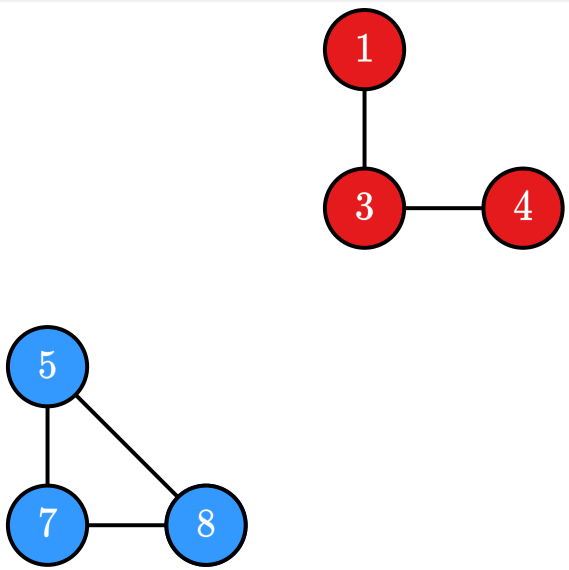


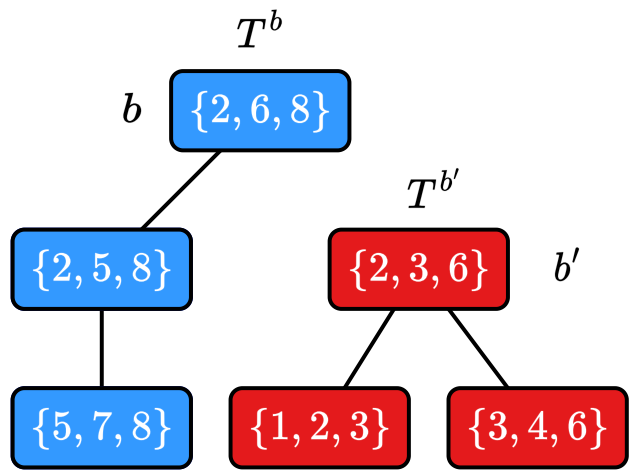
Cut property of TDs

Cut property of TDs

Consider $G = (V, E)$ and a TD $T = (\mathfrak{B}, E_T)$ of it. Pick any edge $e = \{b, b'\} \in E_T$. Removing e will break T into two connected subtrees T^b and $T^{b'}$, containing b and b', respectively. We have that:

$$V_b \cap V_{b'} \text{ separates } \bigcup_{c \in T^b} V_c \text{ from } \bigcup_{c \in T^{b'}} V_c.$$





Treedepth

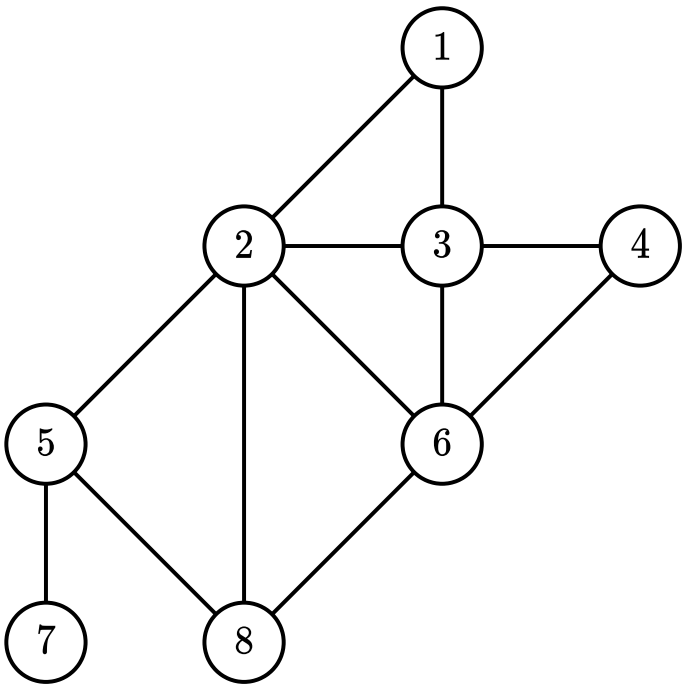
- *Treedepth* measures how much a graph resembles a *shallow tree*.
- Graphs of small treedepth have simpler structure.

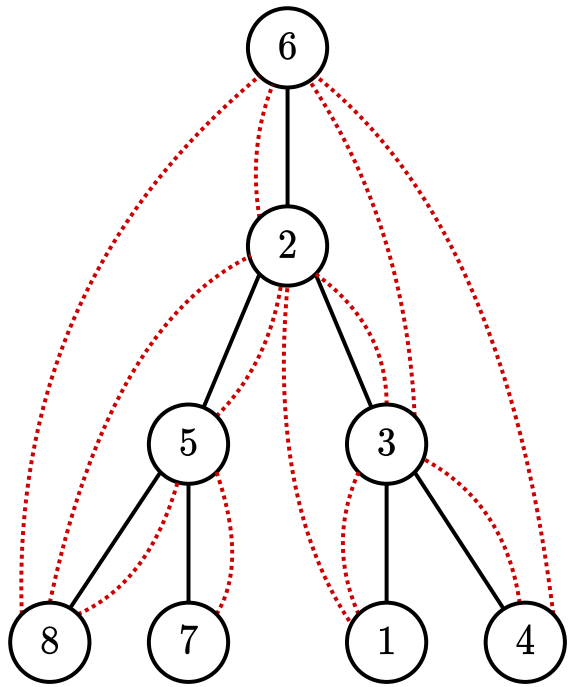
Treedepth

Partial Order Trees (POTs)

Given a graph $G = (V, E)$, a *partial order tree* over G is a *rooted* tree $T = (V, E_T)$ where

$(u, v) \in E \implies u$ and v are in an ancestor-descendant relationship in T.



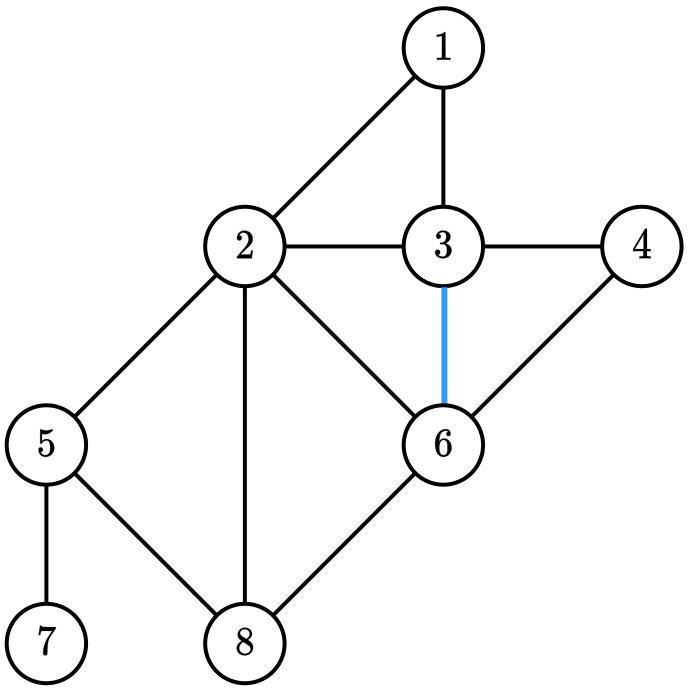


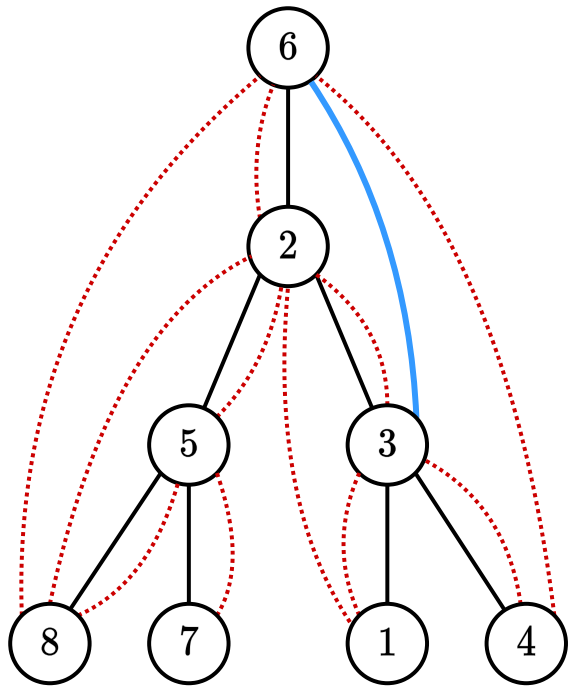
Treedepth

Partial Order Trees (POTs)

Given a graph $G = (V, E)$, a *partial order tree* over G is a *rooted* tree $T = (V, E_T)$ where

$(u, v) \in E \implies u$ and v are in an ancestor-descendant relationship in T.



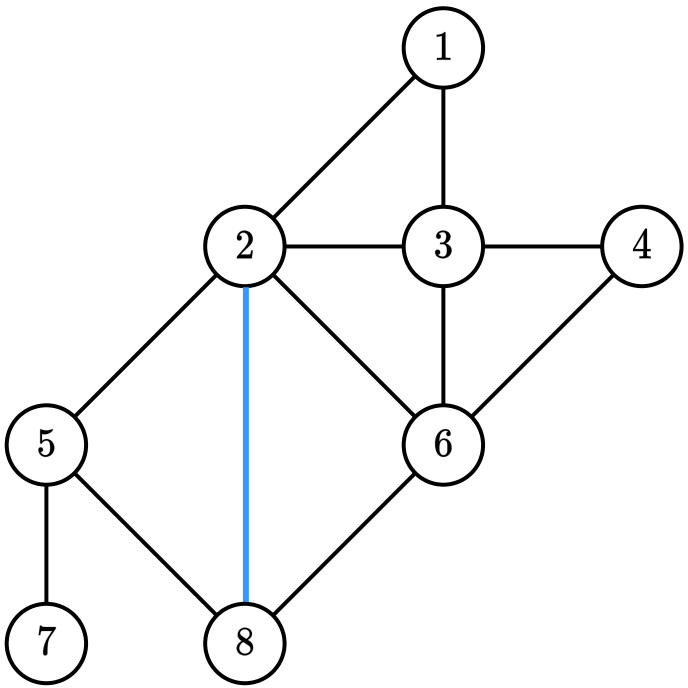


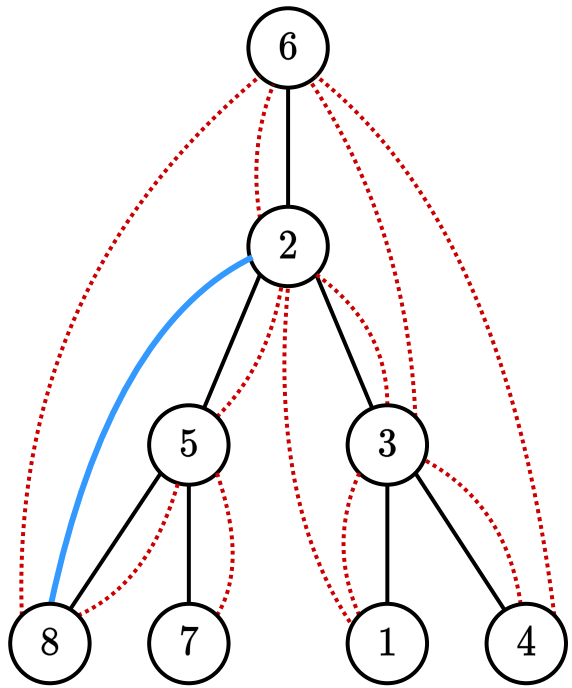
Treedepth

Partial Order Trees (POTs)

Given a graph $G = (V, E)$, a *partial order tree* over G is a *rooted* tree $T = (V, E_T)$ where

$(u, v) \in E \implies u$ and v are in an ancestor-descendant relationship in T.





Treedepth

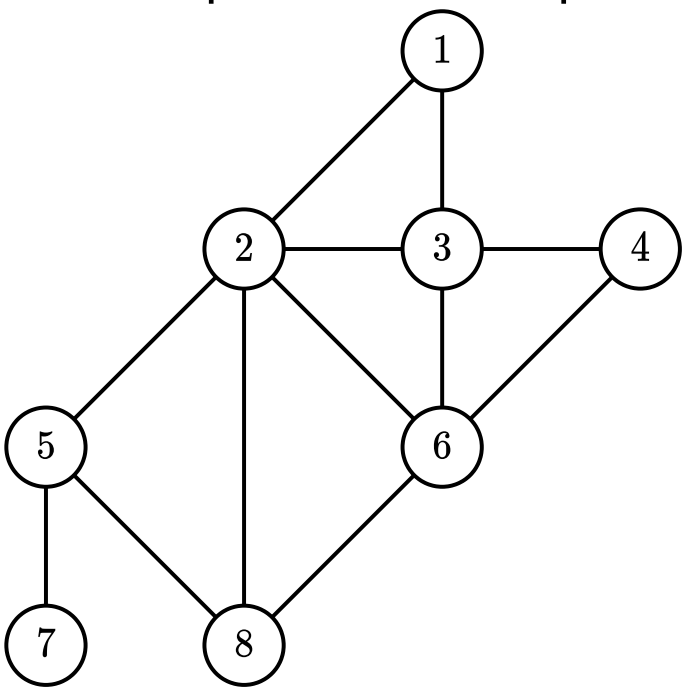
Partial Order Trees (POTs)

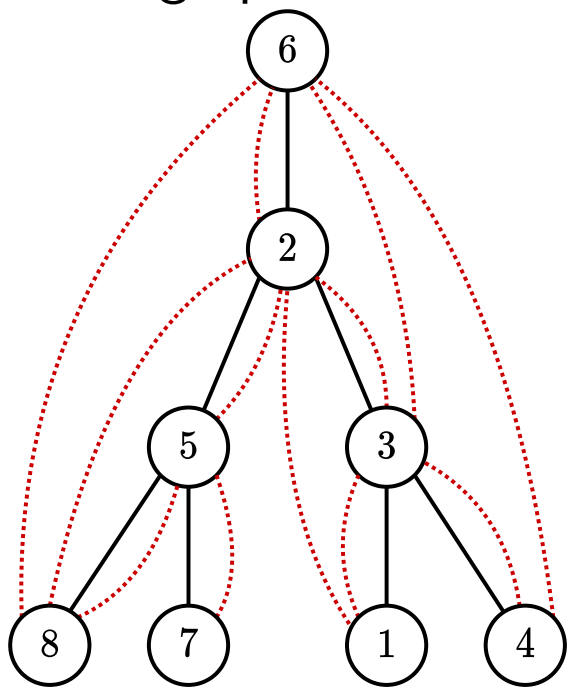
Given a graph $G = (V, E)$, a *partial order tree* over G is a *rooted* tree $T = (V, E_T)$ where

$(u, v) \in E \implies u$ and v are in an ancestor-descendant relationship in T.

The *treedepth* of G is the smallest depth among all POTs over G.

This POT has depth 3 and is optimal $\implies$ the graph has treedepth 3.



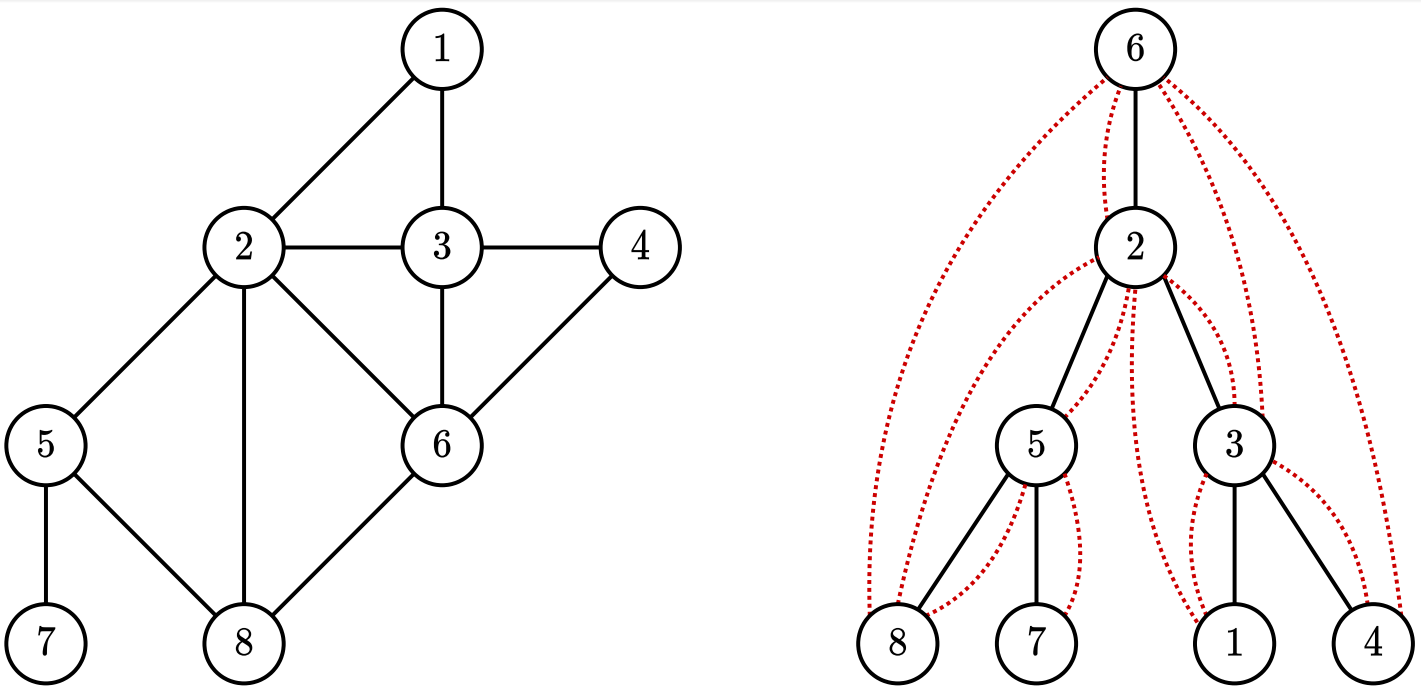


Cut property of POTs

Cut property of POTs

Consider $G = (V, E)$ and a POT $T = (V, E_T)$ over it. For any $u, v \in V$, let A be the set of common ancestors of u and v in T. For *any* path ρ from u to v in G, we have:

$$A \cap \rho \neq \emptyset$$

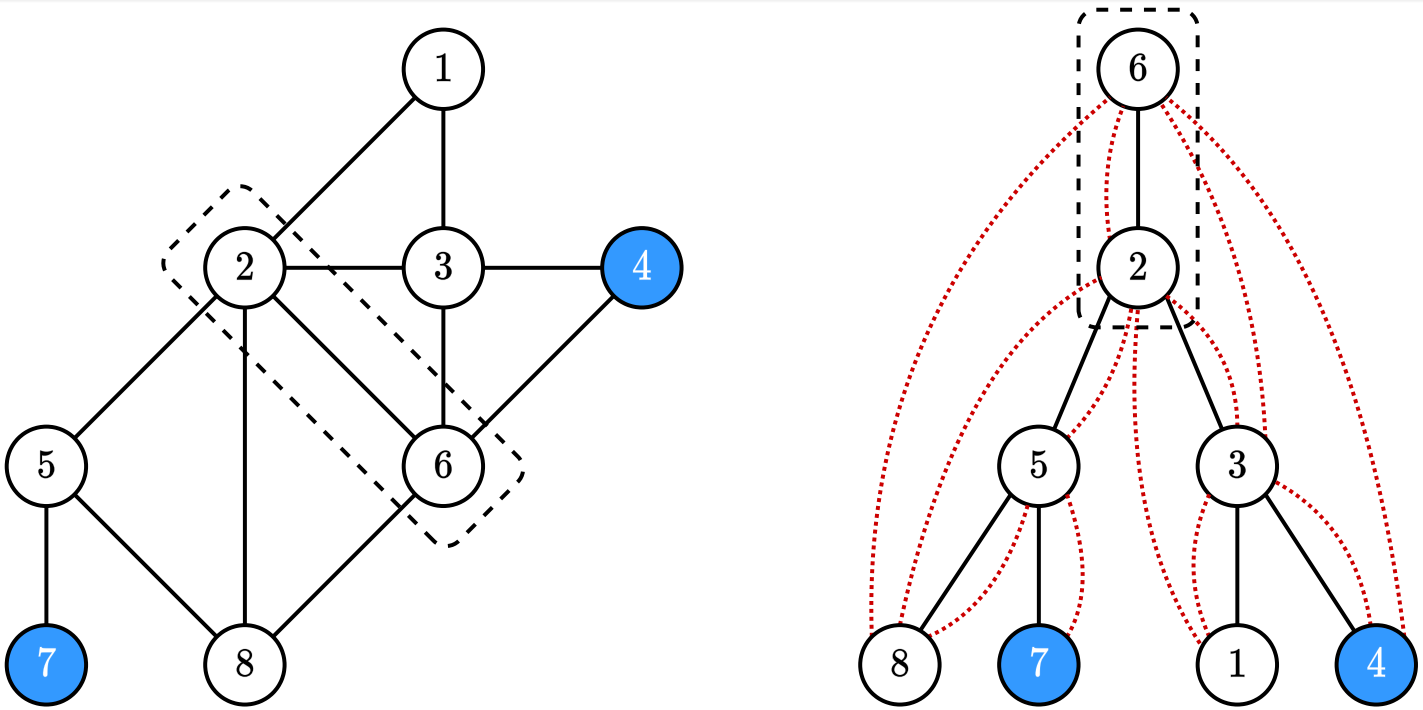


Cut property of POTs

Cut property of POTs

Consider $G = (V, E)$ and a POT $T = (V, E_T)$ over it. For any $u, v \in V$, let A be the set of common ancestors of u and v in T. For *any* path ρ from u to v in G, we have:

$$A \cap \rho \neq \emptyset$$

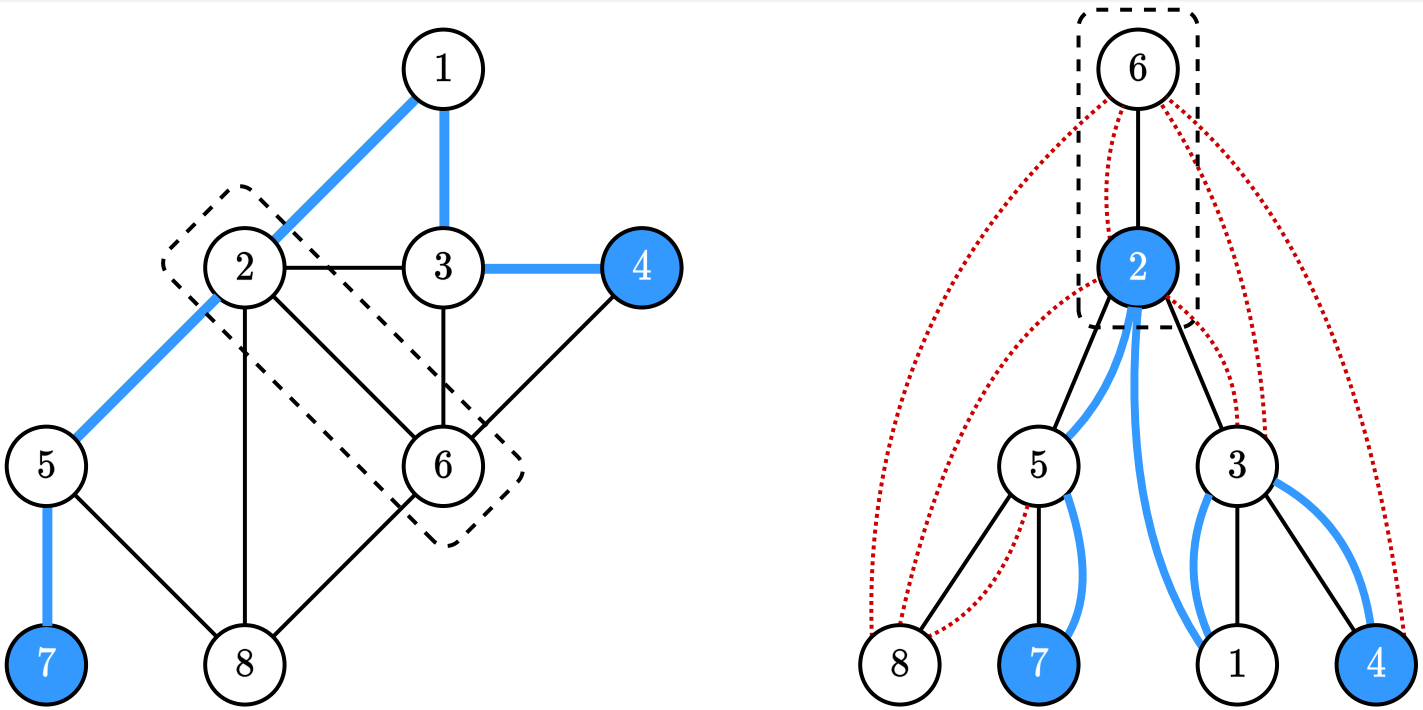


Cut property of POTs

Cut property of POTs

Consider $G = (V, E)$ and a POT $T = (V, E_T)$ over it. For any $u, v \in V$, let A be the set of common ancestors of u and v in T. For *any* path ρ from u to v in G, we have:

$$A \cap \rho \neq \emptyset$$

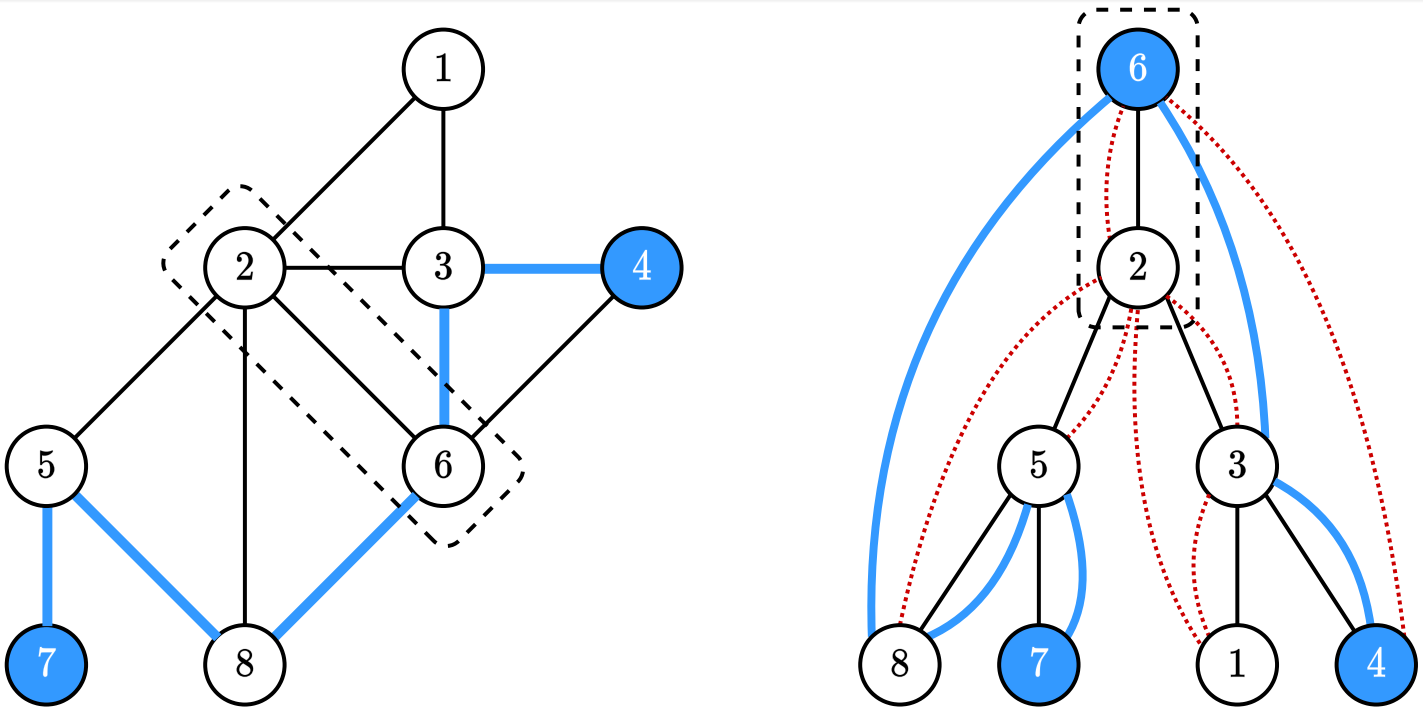


Cut property of POTs

Cut property of POTs

Consider $G = (V, E)$ and a POT $T = (V, E_T)$ over it. For any $u, v \in V$, let A be the set of common ancestors of u and v in T. For *any* path ρ from u to v in G, we have:

$$A \cap \rho \neq \emptyset$$

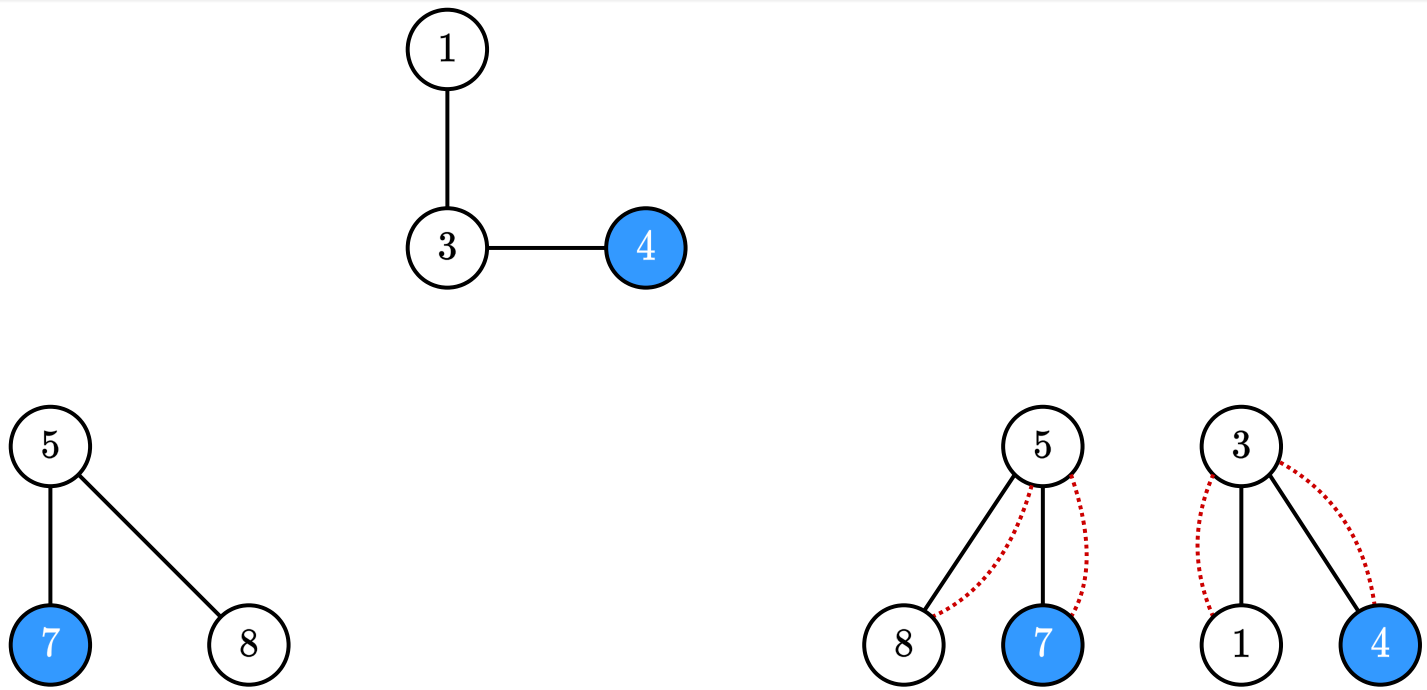


Cut property of POTs

Cut property of POTs

Consider $G = (V, E)$ and a POT $T = (V, E_T)$ over it. For any $u, v \in V$, let A be the set of common ancestors of u and v in T. For *any* path ρ from u to v in G, we have:

$$A \cap \rho \neq \emptyset$$



Exploiting sparsity

Idea: exploit *sparsity* of graphs that arise the program.

In CFGs: each if/while-node has 2 outgoing edges, others have only 1.

In call graphs: we don't expect a function to call a lot of other functions.

Exploiting sparsity

Idea: exploit *sparsity* of graphs that arise the program.

In CFGs: each if/while-node has 2 outgoing edges, others have only 1.

In call graphs: we don't expect a function to call a lot of other functions.

Magic formula

For any problem involving graphs:

$$\text{input graphs are sparse} \wedge \text{problem is simpler on sparse graphs} \implies \text{faster algorithms}$$

Applying the formula

Applying the magic formula on CFGs:

- CFGs have small treewidth, shown by Thorup (Inf. Comput.'98) [4].
- Chattarjee et al. (ESOP'20) used this to develop an algorithm with:
  - Preprocessing in $O(n \cdot D^3)$ time and $O(\lceil \frac{D}{\lg n} \rceil)$ time per query.
  - Can only answer same-context queries.
  - Used by our algorithm as a black box.

Applying the formula

Applying the magic formula on CFGs:

- CFGs have small treewidth, shown by Thorup (Inf. Comput.'98) [4].
- Chattarjee et al. (ESOP'20) used this to develop an algorithm with:
  - Preprocessing in $O(n \cdot D^3)$ time and $O(\lceil \frac{D}{\lg n} \rceil)$ time per query.
  - Can only answer same-context queries.
  - Used by our algorithm as a black box.

Applying the magic formula on call graphs:

- Call graphs have small treedepth.
- *Experimentally:* We analyzed program from DaCapo,
  - Avg. # of functions = 803.1.
  - Avg. treedepth = 43.8.
  - Max. treedepth = 135.
- *Intuition:* functions are developed in chronological order, each function uses a small subset of previously-developed functions as a library.
- In general, we expect treedepth to scale very slowly with program size.

Extra input

Our algorithm uses both parameters, so we'll assume:

- For every function $f \in F$, we are given a TD T_f of f's CFG G_f, and T_f has small width $\texttt{tw}$.
- We are given a POT T over the call graph C, and T has small depth $\texttt{td}$.

Extra input

Our algorithm uses both parameters, so we'll assume:

- For every function $f \in F$, we are given a TD T_f of f's CFG G_f, and T_f has small width $\mathtt{tw}$.
- We are given a POT T over the call graph C, and T has small depth $\mathtt{td}$.

We know that such T_f's and T exist, but how to compute them?

- This is NP-hard in general.
- There are efficient algorithms that compute a TD/POT if its width/depth is small [5, 6].
- In our experiments, we use heuristic solvers.

Where we are

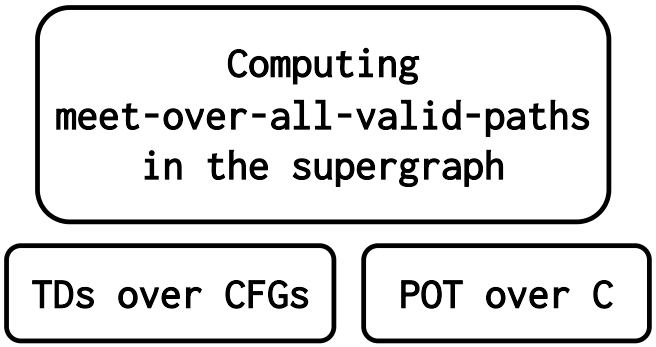

Computing
meet-over-all-valid-paths
in the supergraph
TDs over CFGs
POT over C

Table of Contents

IFDS problems

IFDS problem

Input: $\langle G = (V, E), D, \{M_e\}_{e \in E} \rangle$ queries of the form $\langle u_1, D_1, u_2 \rangle$.

Output: for each query $\langle u_1, D_1, u_2 \rangle$, return:

$$\texttt{MIVP}(u_1, D_1, u_2).$$

Simplification via distributivity

- Flow functions are distributive:
  $M_e(\{d_1, \ldots, d_k\}) = M_e(\emptyset) \cup M_e(\{d_1\}) \cup \cdots \cup M_e(\{d_k\})$
  $\Longrightarrow$ It suffices to know $M_e(\emptyset)$ and $M_e(\{d\})$ for every $d \in D$.

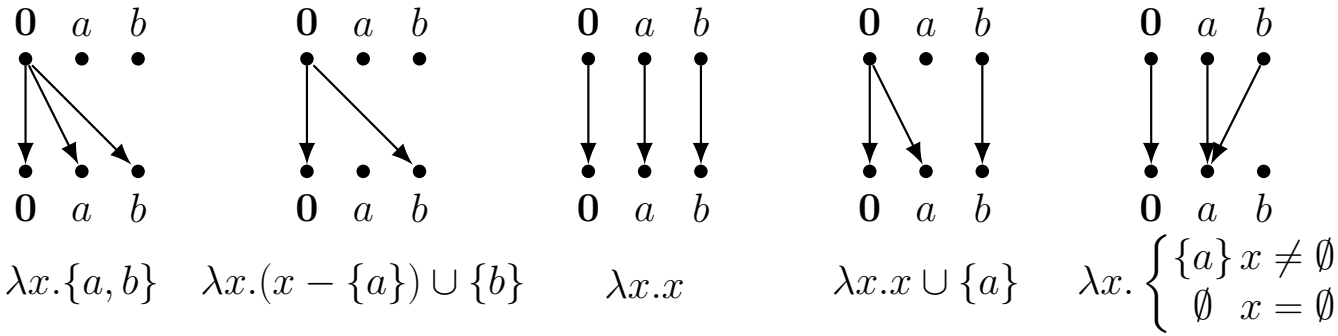
Simplification via distributivity

- Flow functions are distributive:

  $M_e(\{d_1, \ldots, d_k\}) = M_e(\emptyset) \cup M_e(\{d_1\}) \cup \cdots \cup M_e(\{d_k\})$

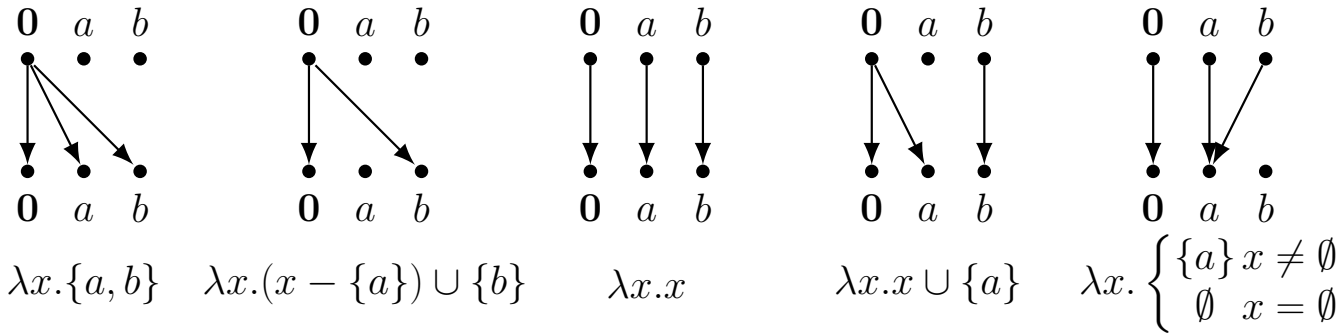
  $\Longrightarrow$ It suffices to know $M_e(\emptyset)$ and $M_e(\{d\})$ for every $d \in D$.

- Can represent a function with a bipartite graph of sides $D^* := D \cup \{\mathbf{0}\}$:

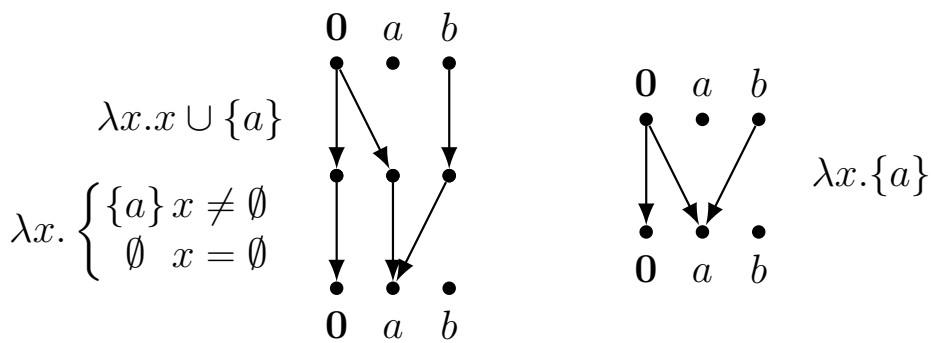


Simplification via distributivity

- Flow functions are distributive:
  $$M_e(\{d_1, \ldots, d_k\}) = M_e(\emptyset) \cup M_e(\{d_1\}) \cup \cdots \cup M_e(\{d_k\})$$
  $\implies$ It suffices to know $M_e(\emptyset)$ and $M_e(\{d\})$ for every $d \in D$.
- Can represent a function with a bipartite graph of sides $D^* := D \cup \{\mathbf{0}\}$:

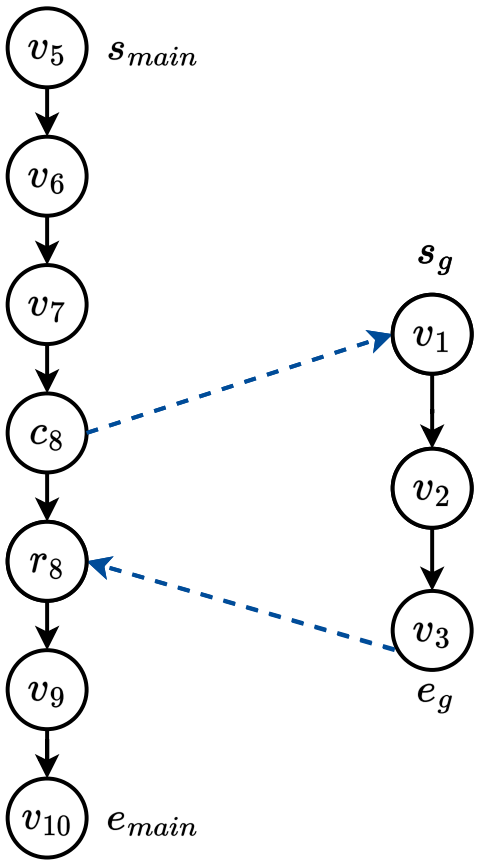


- Compositions of functions $\equiv$ reachability of their representation:



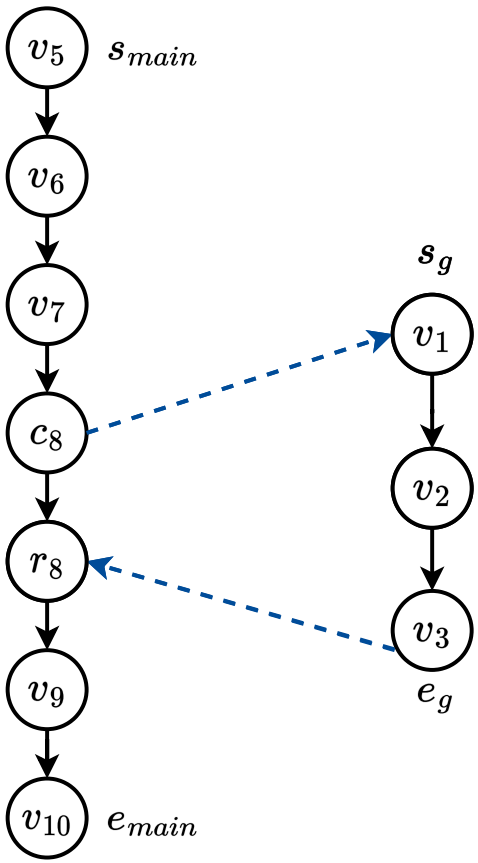
Exploded supergraphs

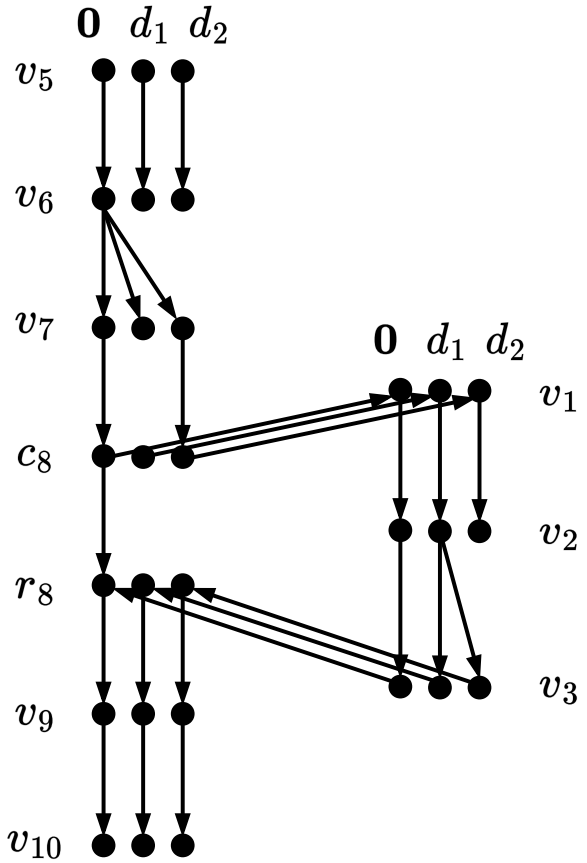
Replace each edge in a supergraph $G = (V, E)$ with their graph representation, which gives an exploded supergraph $G = \left(V \times D^*, \overline{E}\right)$:



Exploded supergraphs

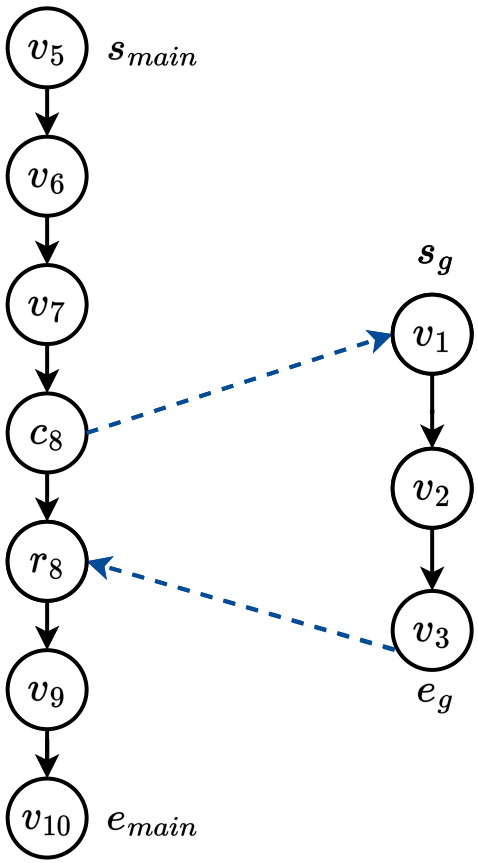
Replace each edge in a supergraph $G = (V, E)$ with their graph representation, which gives an exploded supergraph $G = \left(V \times D^*, \overline{E}\right)$:

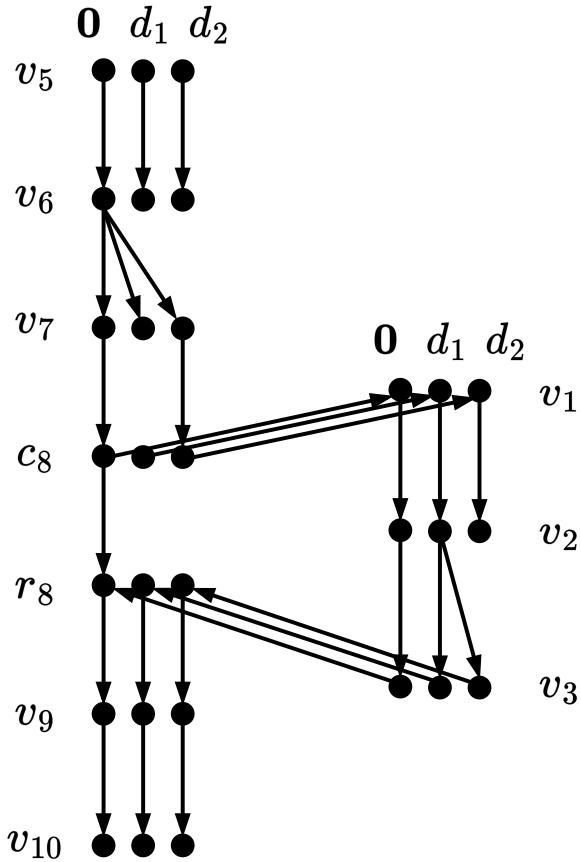




Exploded supergraphs

Replace each edge in a supergraph $G = (V, E)$ with their graph representation, which gives an exploded supergraph $G = \left(V \times D^*, \overline{E}\right)$:



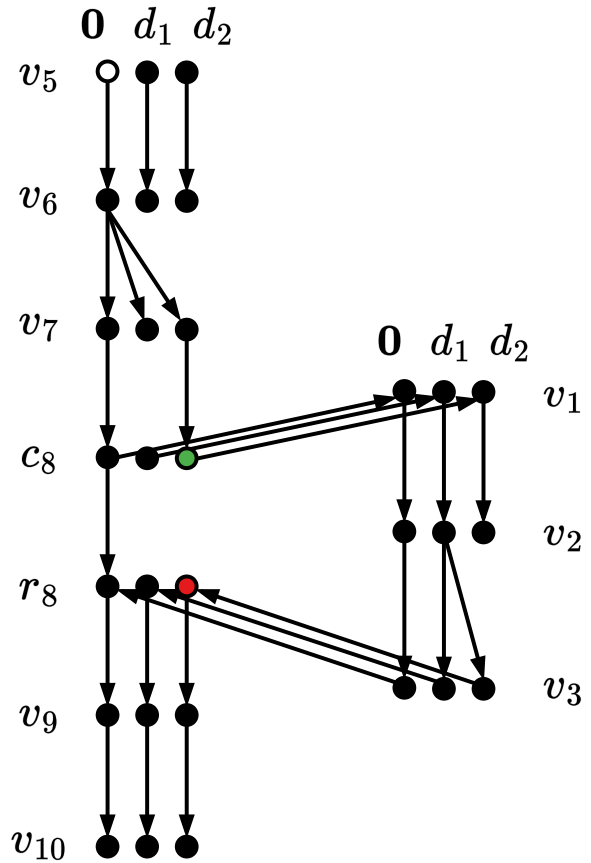


$\mathtt{Q}((u_1, d_1), (u_2, d_2)) := 1$ iff there is an IVP from (u_1, d_1) to (u_2, d_2) in $\overline{G}$.

Exploded supergraphs: example

- $d_2 =$ "b may be null"
- $\mathtt{Q}((v_5, \mathbf{0}), (c_8, d_2)) = 1 \implies$ b may be null after line 7.
- $\mathtt{Q}((v_5, \mathbf{0}), (r_8, d_2)) = 0 \implies$ b is not null after returning from call to g.

```
1  void g(int *&a, int *&b) {
2      b = a;
3  }
4
5  int main() {
6      int *a, *b;
7      a = new int(42);
8      g(a, b);
9      *b = 0;
10 }
```



IFDS problems

Simpler problem: checking existence of an IVP in $\overline{G}$.

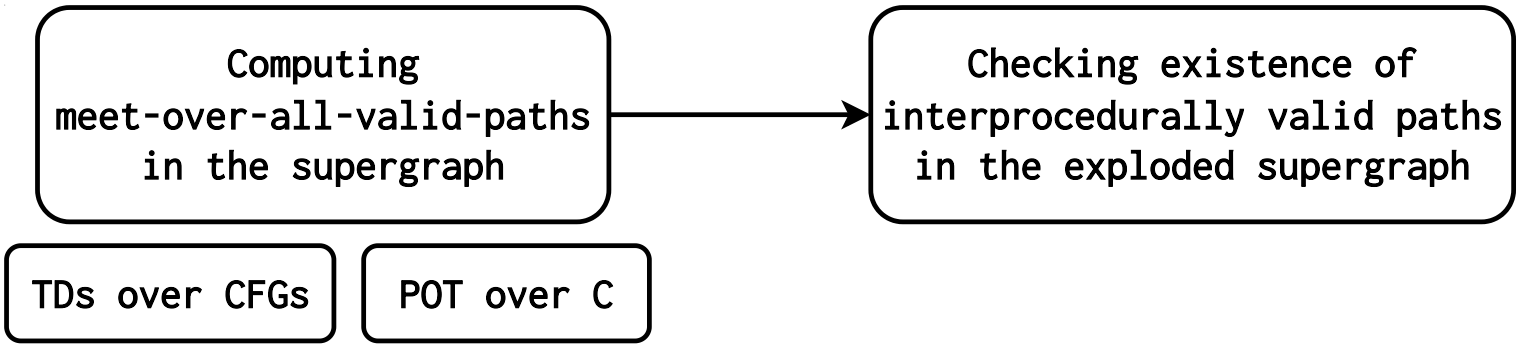
IFDS problem #2

Input: $\langle \overline{G} \rangle$ and queries of the form $\langle (u_1, d_1), (u_2, d_2) \rangle$.

Output: for each query $\langle (u_1, d_1), (u_2, d_2) \rangle$, return:

$$\mathtt{Q}((u_1, d_1), (u_2, d_2)).$$

Where we are

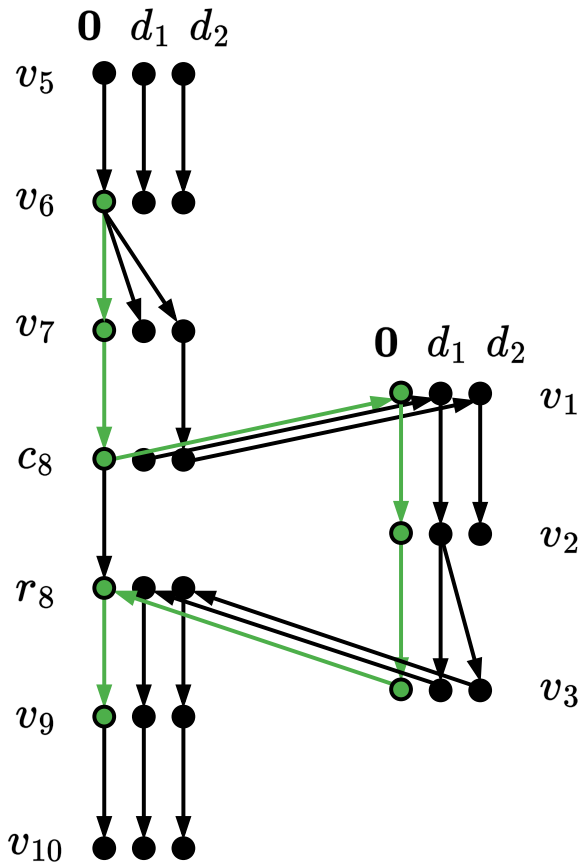

Computing
meet-over-all-valid-paths
in the supergraph
Checking existence of
interprocedurally valid paths
in the exploded supergraph
TDs over CFGs
POT over C

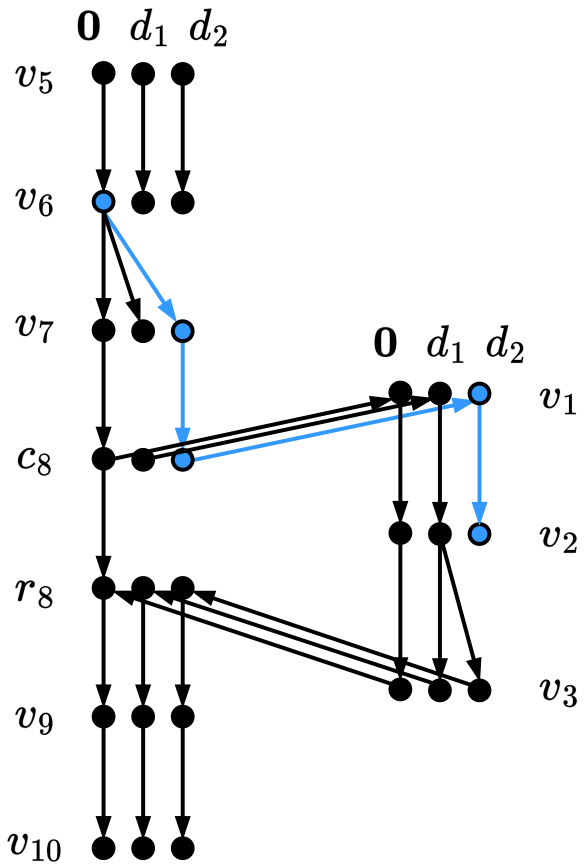
Same-context paths

A same-context path (SCP) in $G/\overline{G}$ is a special IVP that keeps the call-stack intact.

Same-context paths

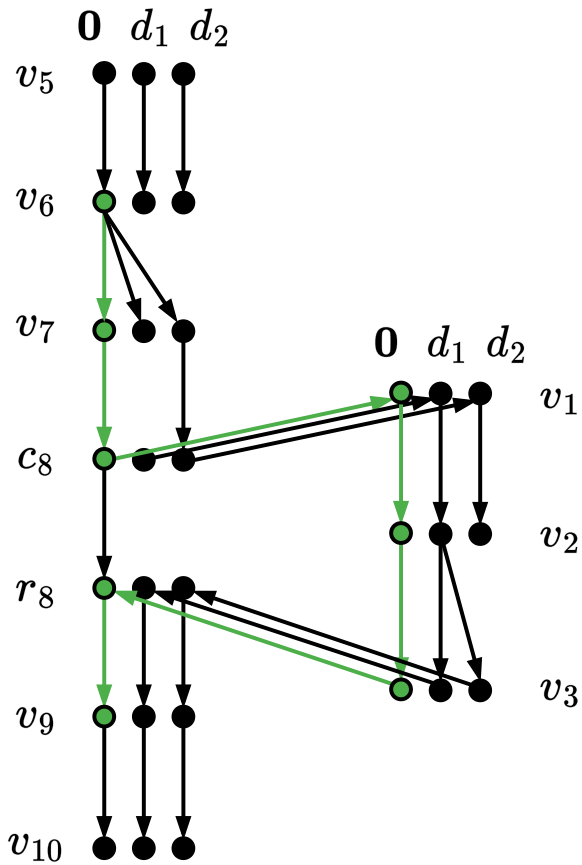
A same-context path (SCP) in $G/\overline{G}$ is a special IVP that keeps the call-stack intact.

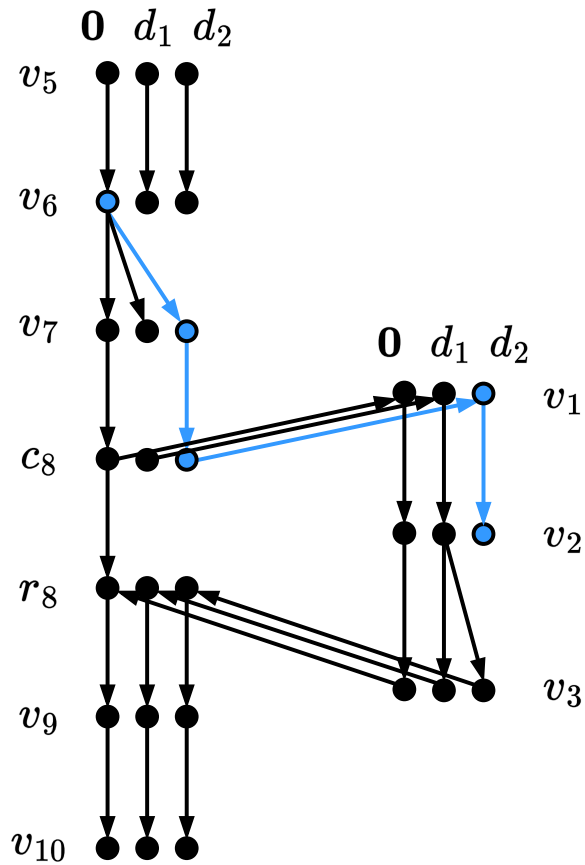




Same-context paths

A same-context path (SCP) in $G/\overline{G}$ is a special IVP that keeps the call-stack intact.





$\texttt{SCQ}((u_1, d_1), (u_2, d_2)) := 1$ iff there is an SCP from (u_1, d_1) to (u_2, d_2) in $\overline{G}$.

Canonical partition

Idea: consider an IVP π in $\overline{G}$, there are two types of call nodes in π:

- Temporary calls: calls c with a corresponding return node r later in π.
- Persistent calls: no corresponding return.

Canonical partition

Idea: consider an IVP π in $\overline{G}$, there are two types of call nodes in π:

- Temporary calls: calls c with a corresponding return node r later in π.
- Persistent calls: no corresponding return.

Canonical partition

π can always be written as:

$$\pi = \Sigma_1 \cdot c_1 \cdot \Sigma_2 \cdot c_2 \cdots \Sigma_k \cdot c_k \cdot \Sigma_{k+1}$$

Where $c_1, \ldots, c_k$ are the *persistent* calls in π.

Canonical partition

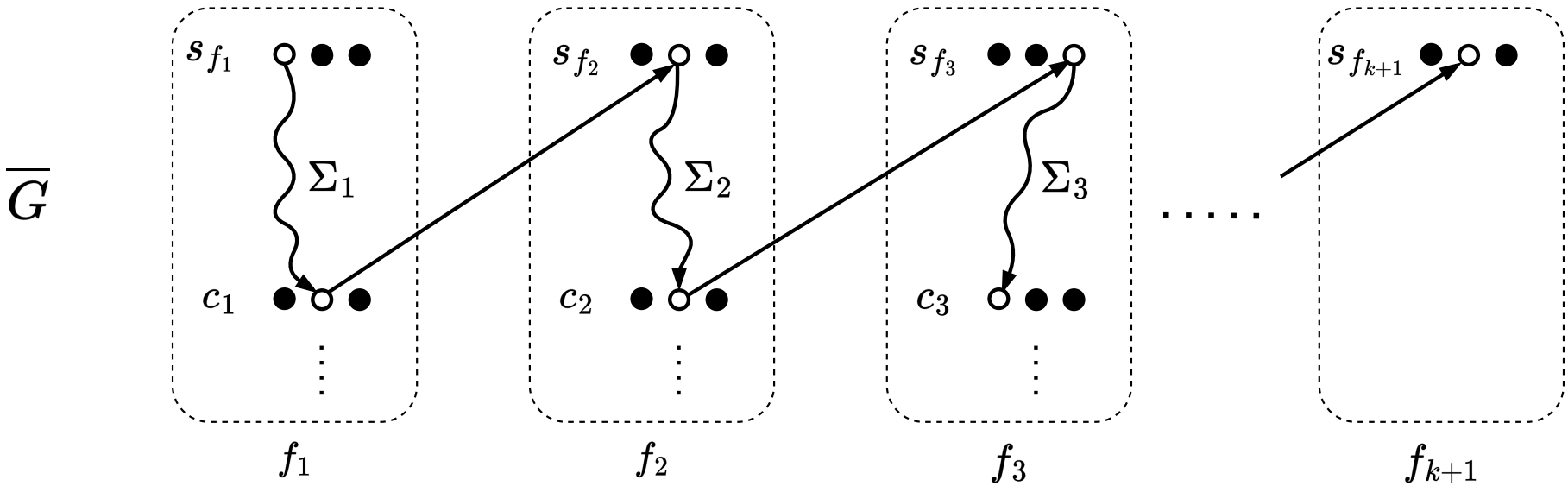
$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

Assumption: suppose π begins and ends at some start-node.

Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

Assumption: suppose π begins and ends at some start-node.

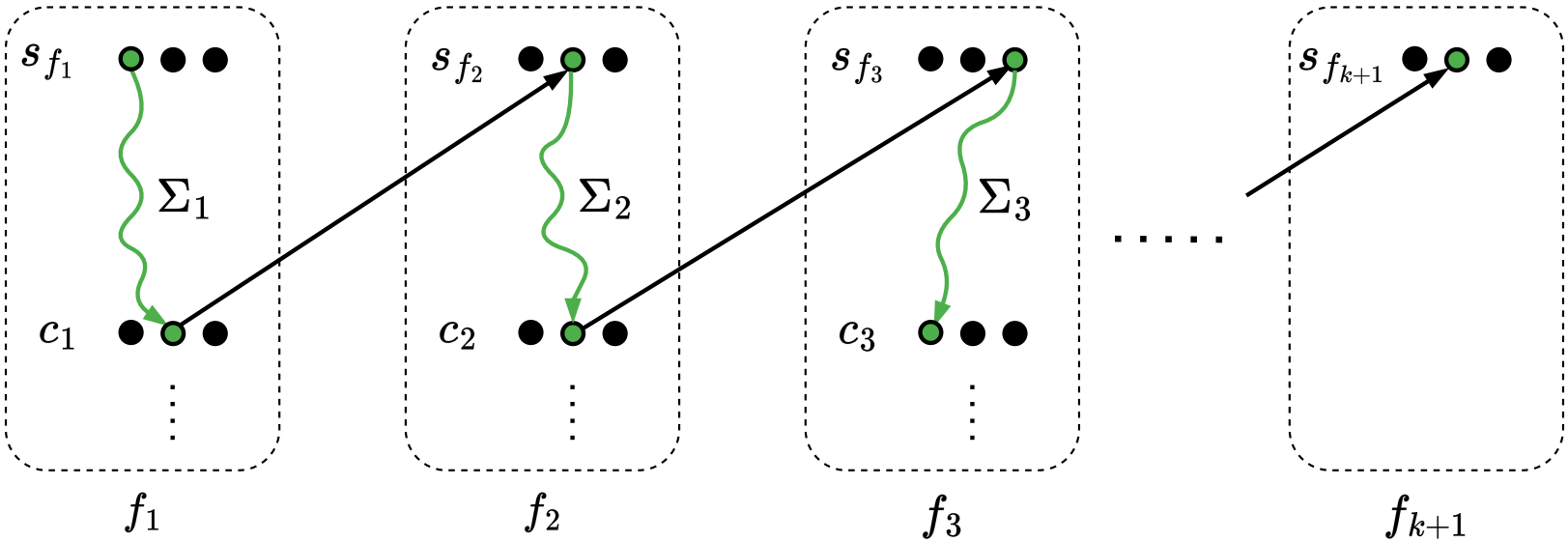


Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

Observation #1: each $\Sigma_i \cdot c_i$ is a same-context path.

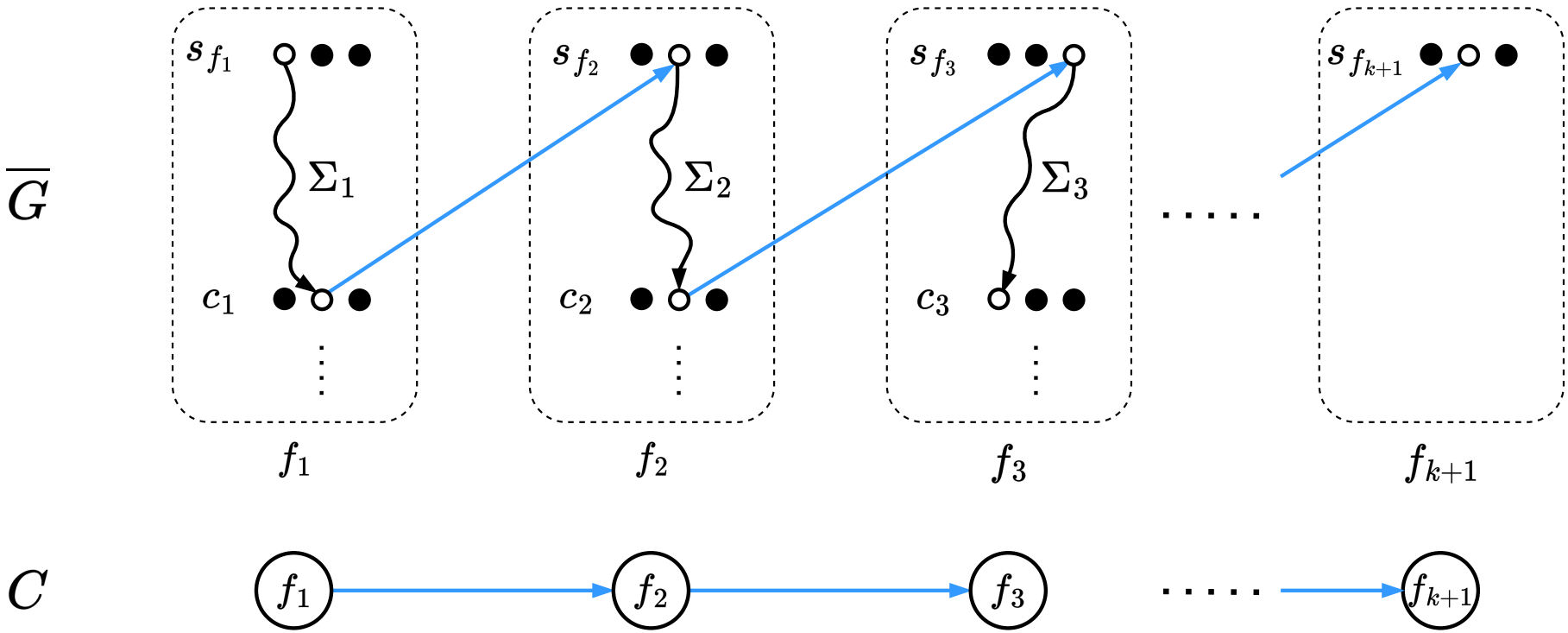
$\overline{G}$



Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

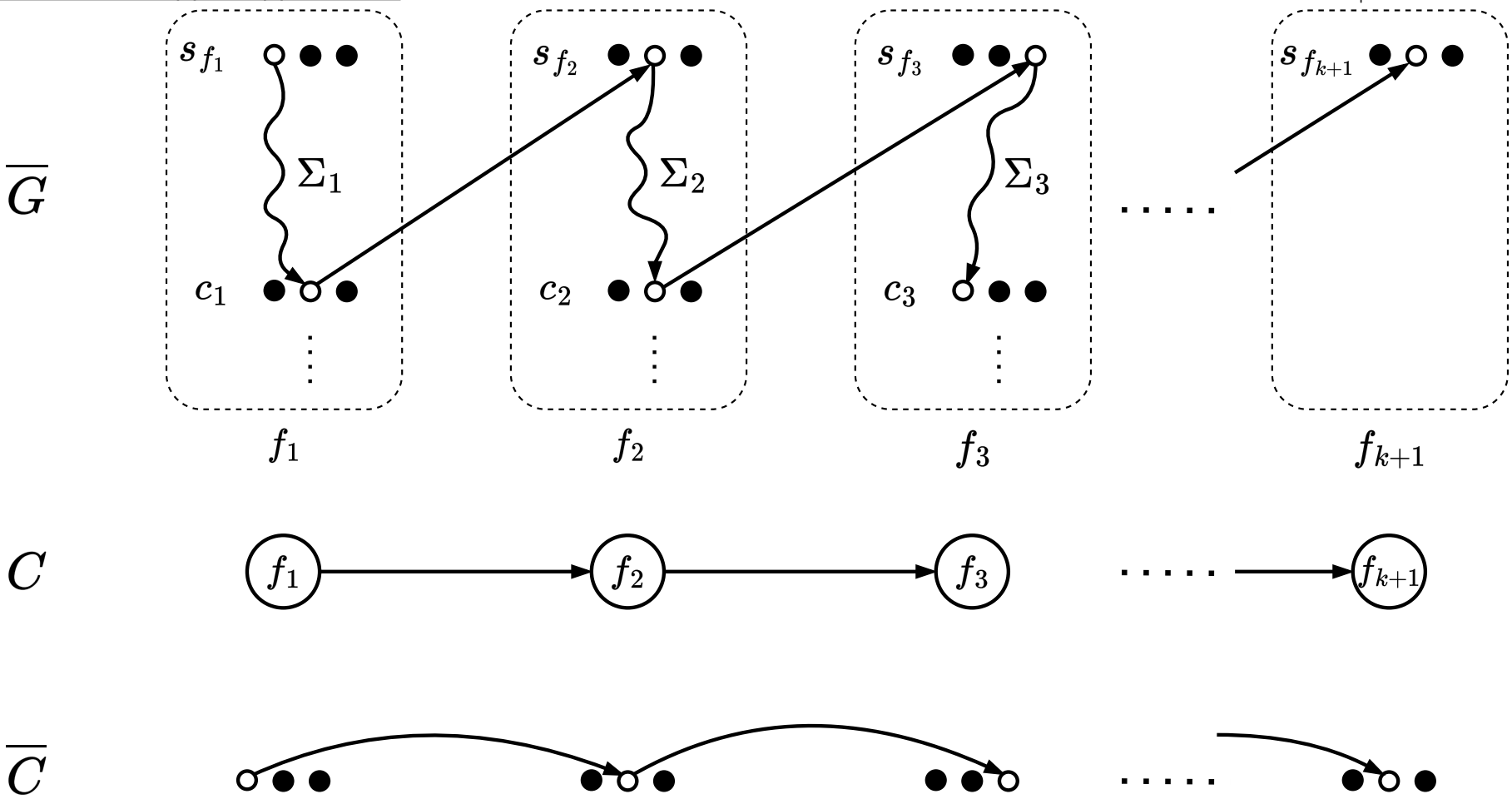
Observation #2: c_i calls $f_{i+1} \implies (f_i, f_{i+1})$ is an edge of the call graph.



Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

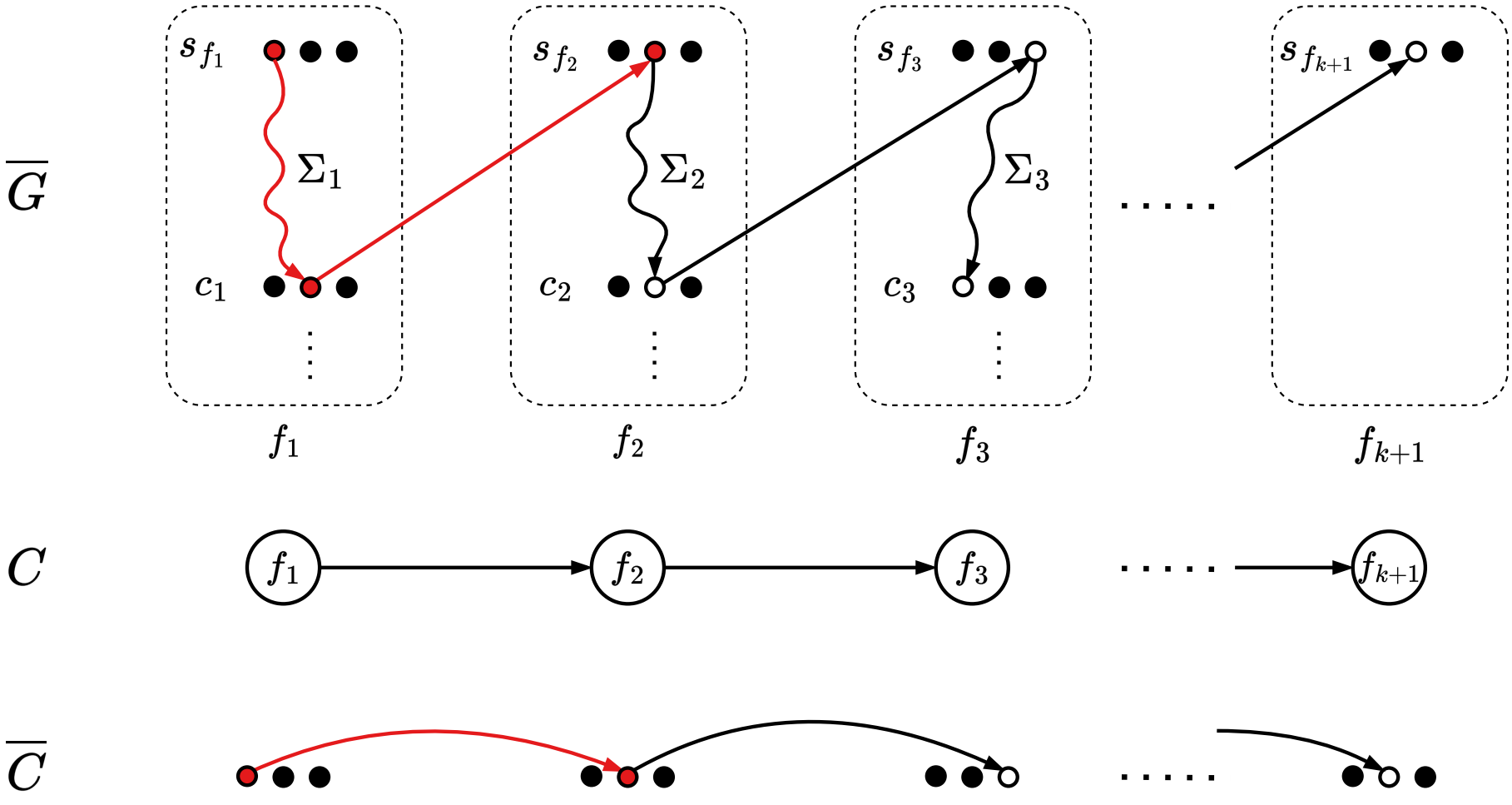
Exploded call graph $\overline{C}$: each edge abstracts a segment $\Sigma_i \cdot c_i \cdot s_{f_{i+1}}$.



Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

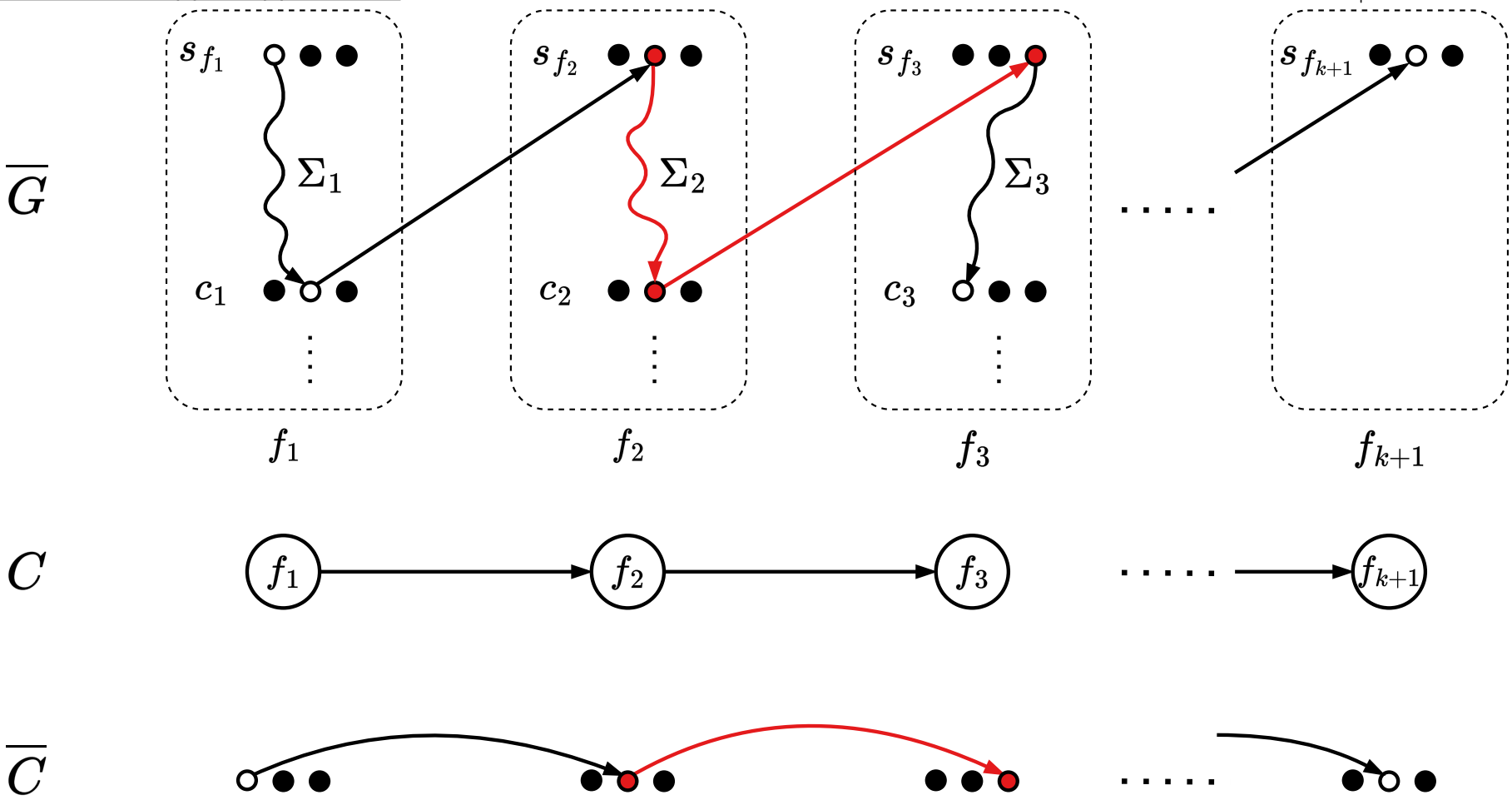
Exploded call graph $\overline{C}$: each edge abstracts a segment $\Sigma_i \cdot c_i \cdot s_{f_{i+1}}$.



Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

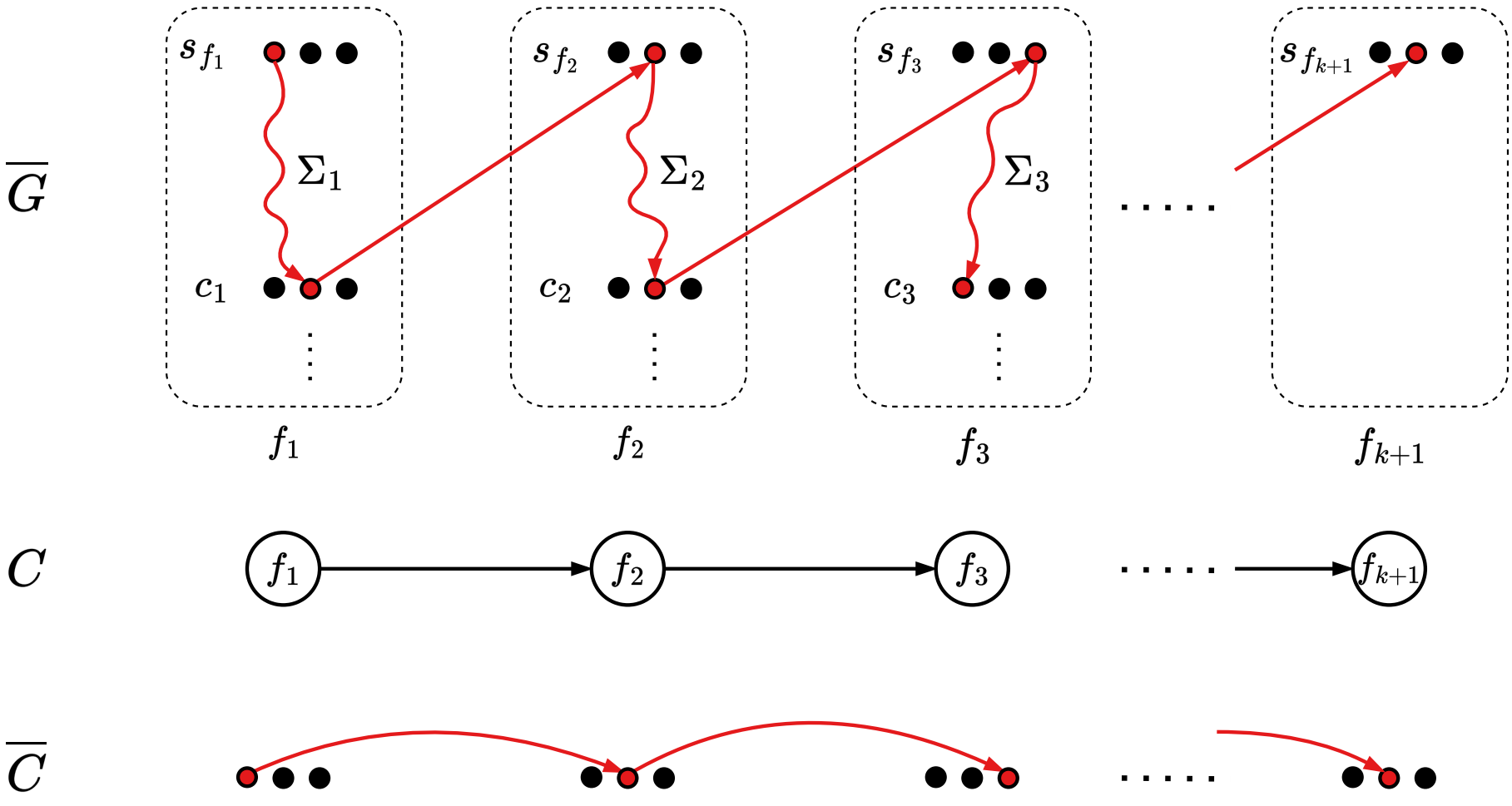
Exploded call graph $\overline{C}$: each edge abstracts a segment $\Sigma_i \cdot c_i \cdot s_{f_{i+1}}$.



Canonical partition

$$\pi = (\Sigma_1 \cdot c_1) \cdot (\Sigma_2 \cdot c_2) \cdots (\Sigma_k \cdot c_k) \cdot \Sigma_{k+1}$$

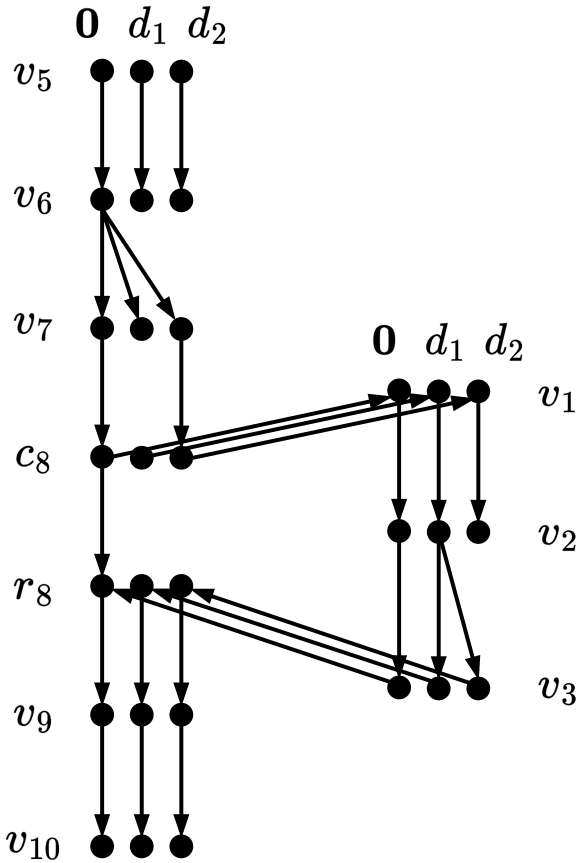
Existence of $\pi \implies$ existence of a path from $(f_1, \cdot)$ to $(f_{k+1}, \cdot)$ in $\overline{C}$



Exploded call graphs

For a call graph $C = (F, E_C)$, an exploded call graph $\overline{C} = (F \times D^*, \overline{E_C})$ has $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t.

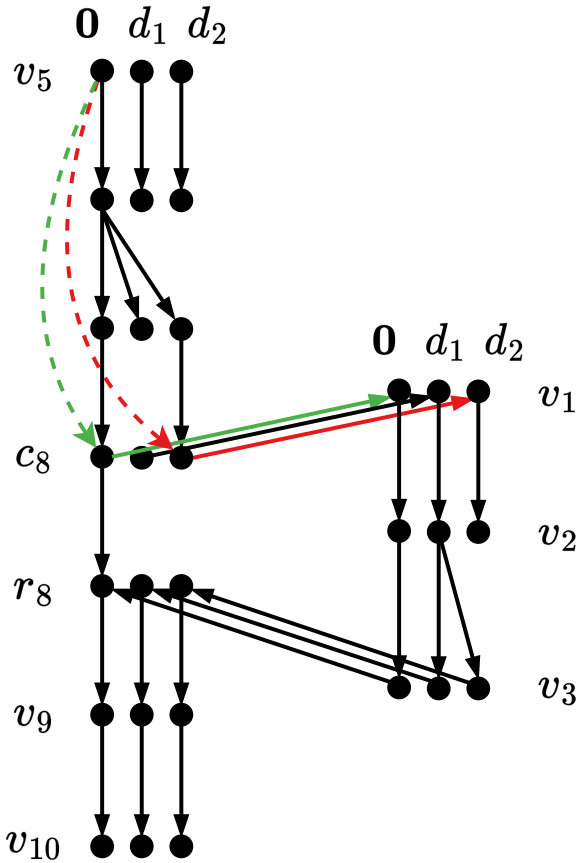
- $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).

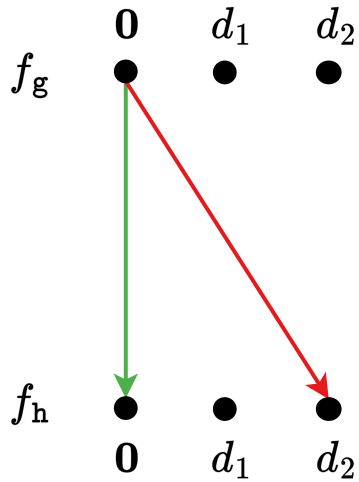


Exploded call graphs

For a call graph $C = (F, E_C)$, an exploded call graph $\overline{C} = (F \times D^*, \overline{E_C})$ has $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t.

- $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).

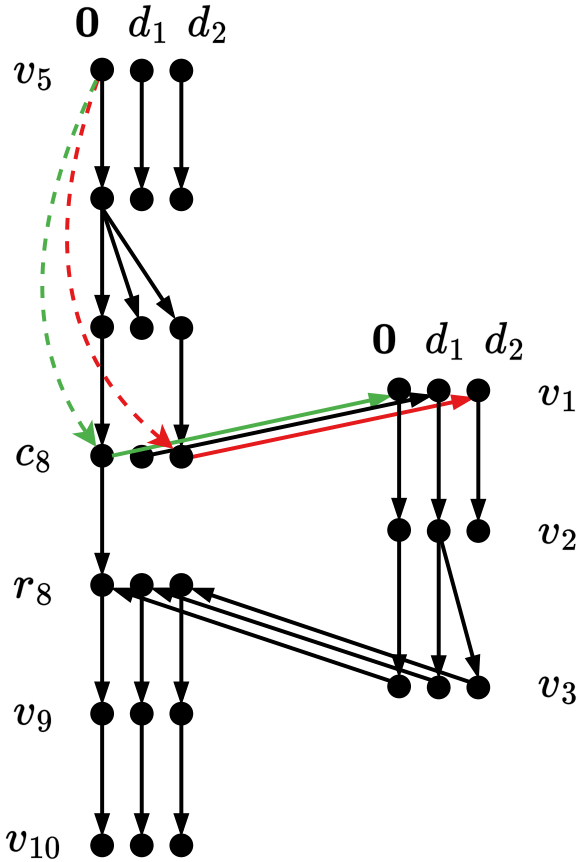


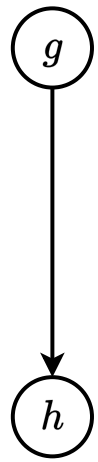


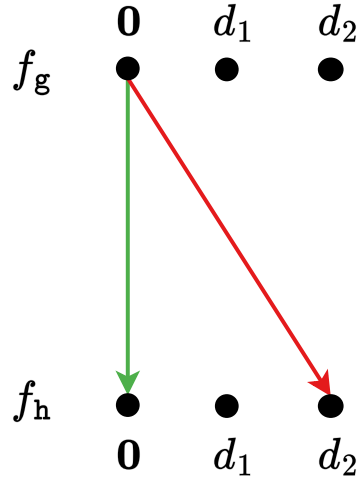
Exploded call graphs

For a call graph $C = (F, E_C)$, an exploded call graph $\overline{C} = (F \times D^*, \overline{E_C})$ has $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t.

- $\mathtt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).







$$\forall \langle (s_{f_u}, d_1), (s_{f_v}, d_2) \rangle, \quad \mathtt{Q}((s_{f_u}, d_1), (s_{f_v}, d_2)) = (f_u, d_1) \rightsquigarrow_{\overline{C}} (f_v, d_2).$$

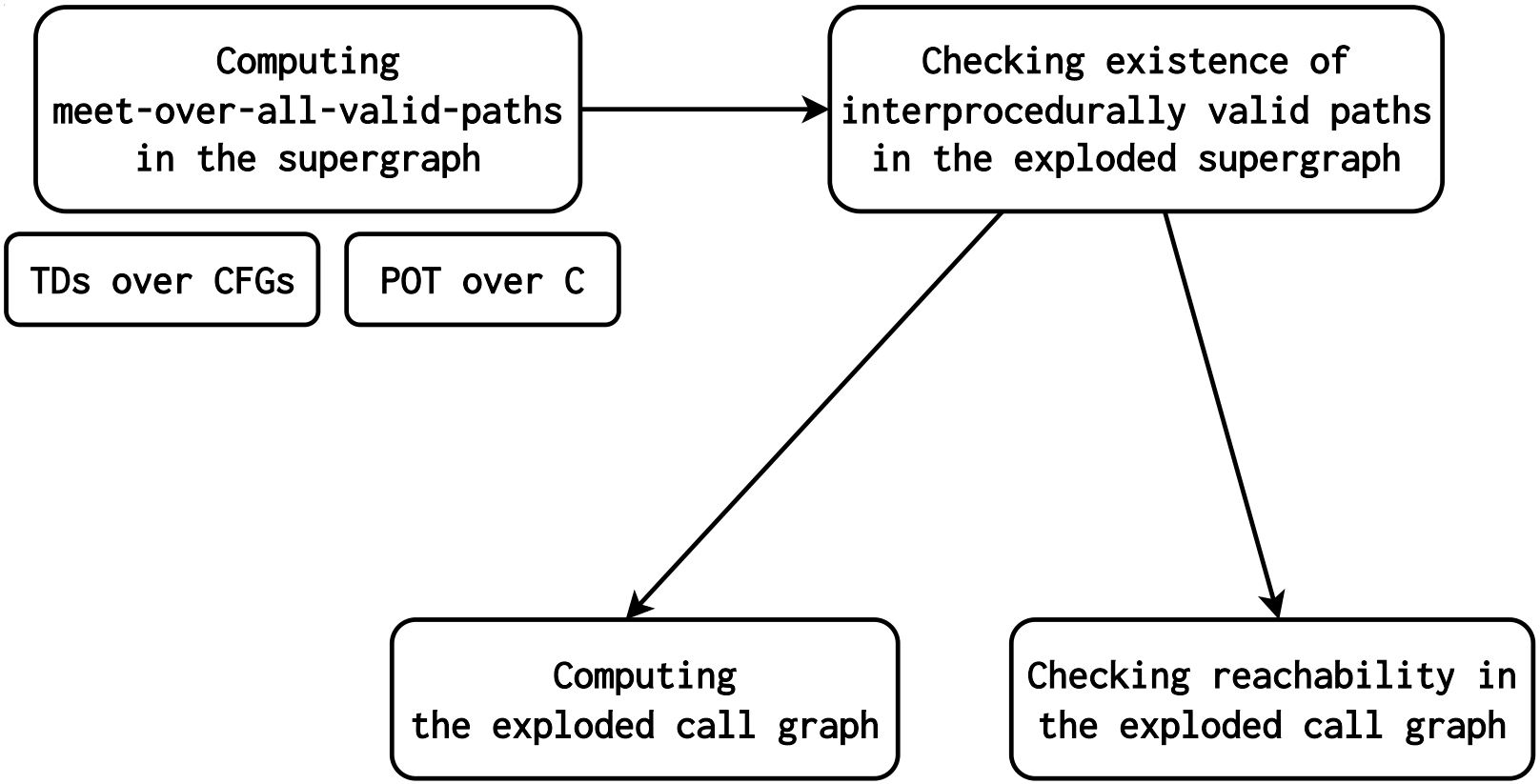
Exploded call graph

We now have two subproblems to answer queries of the form

$$\langle (s_{f_u}, d_1), (s_{f_v}, d_2) \rangle,$$

1. Computing the exploded call graph $\overline{C}$.
2. Answering reachability queries on $\overline{C}$.

Where we are


Computing
meet-over-all-valid-paths
in the supergraph
Checking existence of
interprocedurally valid paths
in the exploded supergraph
TDs over CFGs
POT over C
Computing
the exploded call graph
Checking reachability in
the exploded call graph

Computing $\overline{C}$

Recall, $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t

- $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).

Computing $\overline{C}$

Recall, $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t

- $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).

We already have an algorithm of Chatterjee to answer $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$.

Computing $\overline{C}$

Recall, $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t

- $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).

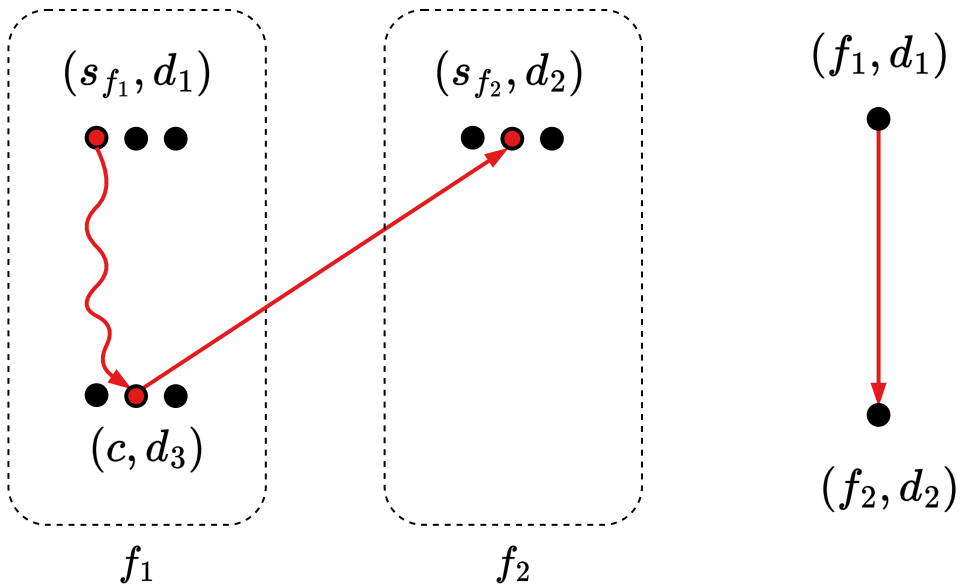
We already have an algorithm of Chatterjee to answer $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$. We'll use it as a black box.

Computing $\overline{C}$

Recall, $((f_1, d_1), (f_2, d_2)) \in \overline{E_C}$ iff there is a $(c, d_3) \in V_f \times D^*$ s.t

- $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$
- (c, d_3) calls (s_{f_2}, d_2).

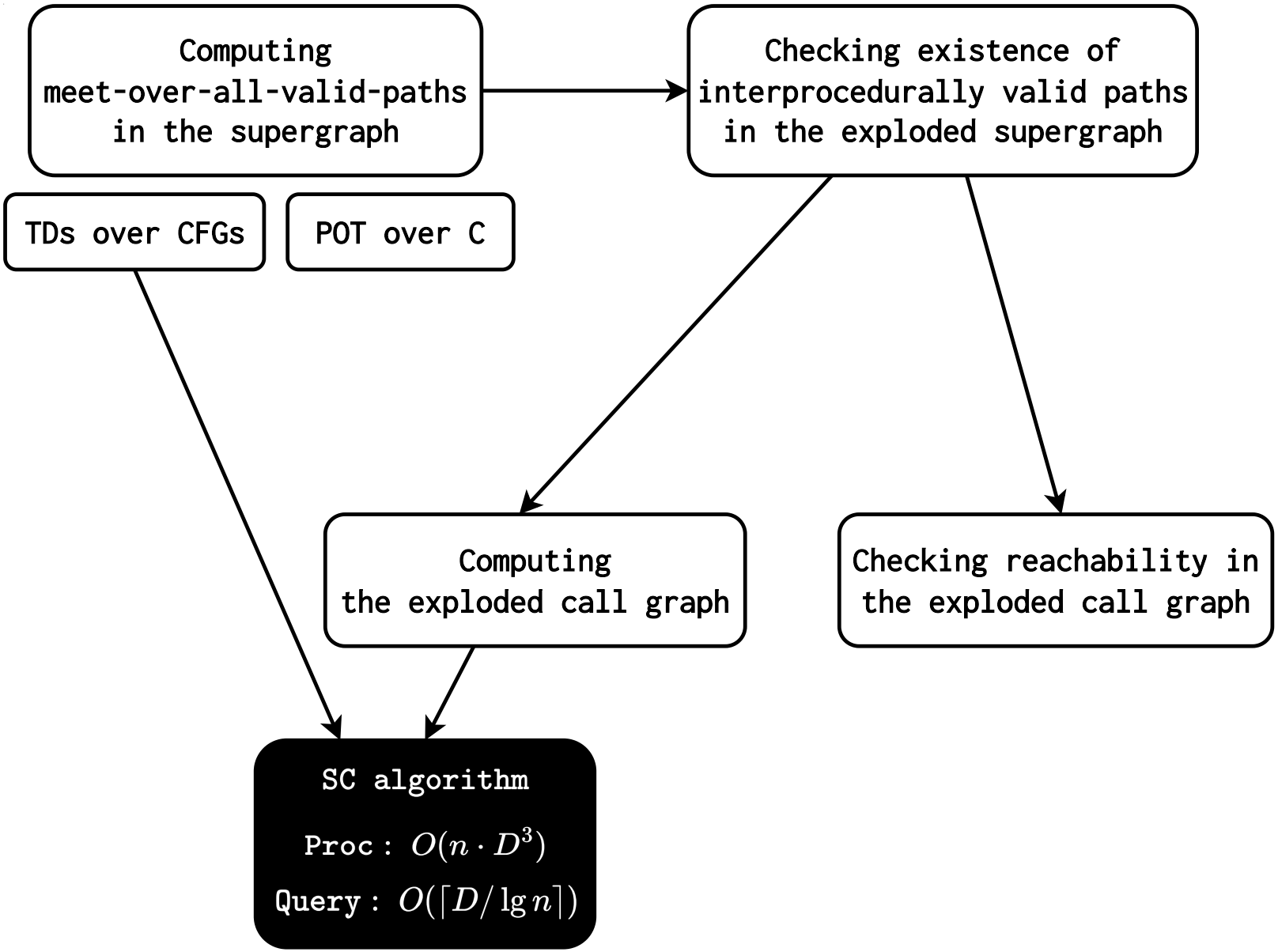
We already have an algorithm of Chatterjee to answer $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$. We'll use it as a black box.



Algorithm:

1. Iterate over all possible $((f_1, d_1), (c, d_3))$.
2. Invoke Chatterjee's algorithm to compute $\texttt{SCQ}((s_{f_1}, d_1), (c, d_3))$.
3. If it returns 1, add the corresponding $((f_1, d_1), (f_2, d_2))$ to $\overline{C}$.

Where we are


Computing
meet-over-all-valid-paths
in the supergraph
Checking existence of
interprocedurally valid paths
in the exploded supergraph
TDs over CFGs
POT over C
Computing
the exploded call graph
Checking reachability in
the exploded call graph
SC algorithm
Proc : $O(n \cdot D^3)$
Query : $O(\lceil D/\lg n \rceil)$

Checking reachability in $\overline{C}$

Reachability on $\overline{C}$

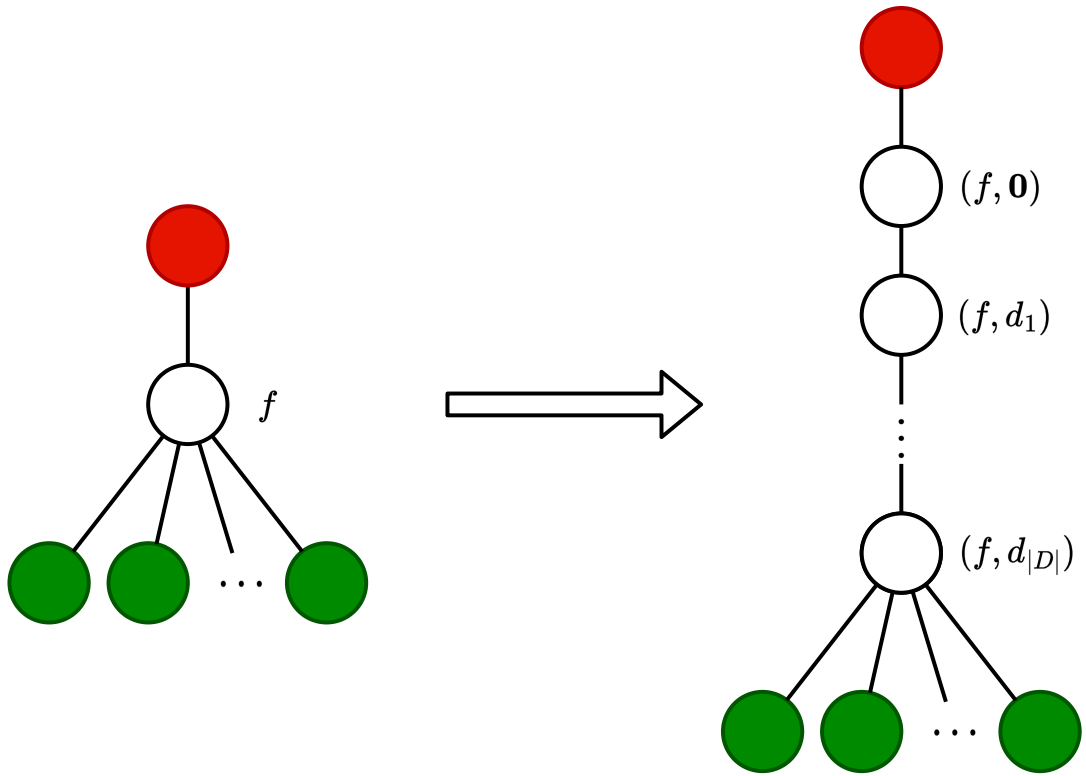
Input: $\langle \overline{C} \rangle$ and queries of the form $\langle (f_u, d_1), (f_v, d_2) \rangle$.

Output: for each query $\langle (f_u, d_1), (f_v, d_2) \rangle$, return:

$$(f_u, d_1) \rightsquigarrow_{\overline{C}} (f_v, d_2).$$

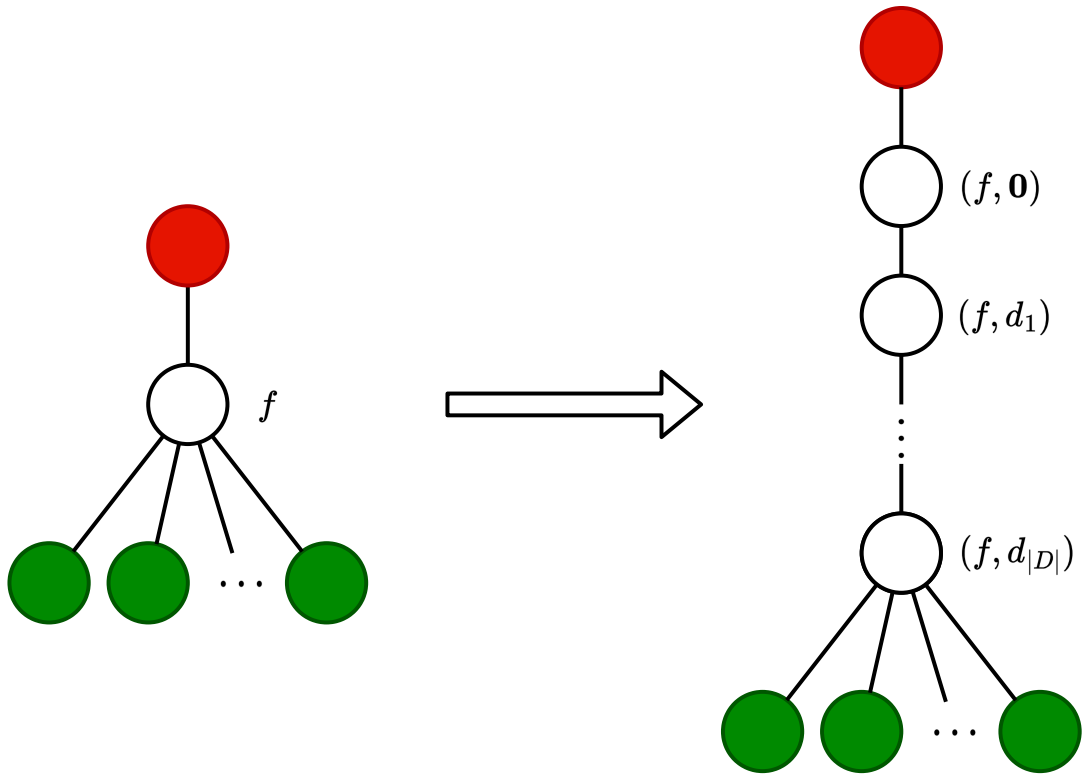
Checking reachability in $\overline{C}$

We have a POT T over C : explode it into a POT $\overline{T}$ over $\overline{C}$.



Checking reachability in $\overline{C}$

We have a POT T over C : explode it into a POT $\overline{T}$ over $\overline{C}$.



T has depth $\mathtt{td} \implies \overline{T}$ has depth $\mathtt{td} \cdot D$, which is still small.

Exploiting treedepth

Reachability on $\overline{C}$ using POT $\overline{T}$

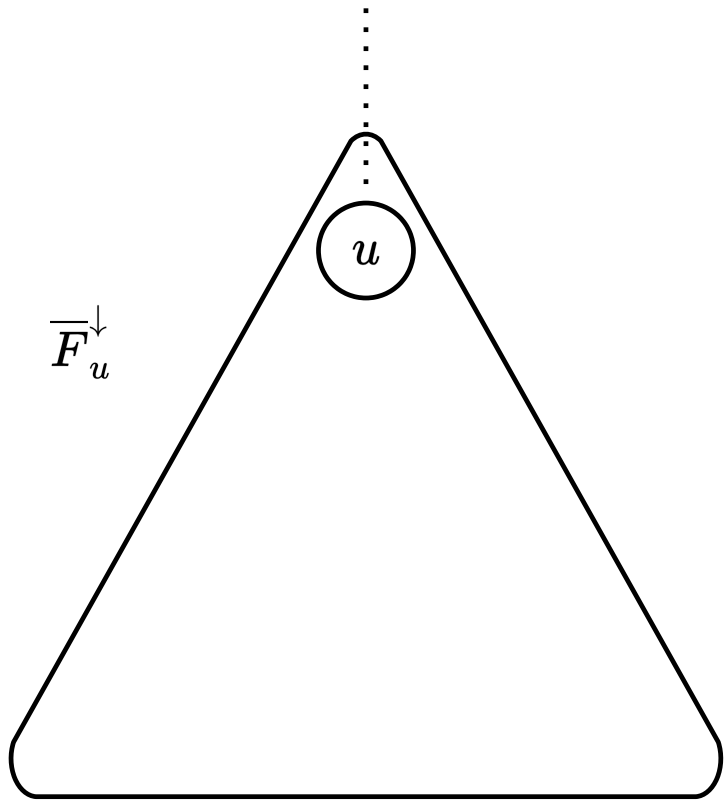
Input: $\langle \overline{C}, \overline{T} \rangle$ and queries of the form $\langle (f_u, d_1), (f_v, d_2) \rangle$.

Output: for each query $\langle (f_u, d_1), (f_v, d_2) \rangle$, return:

$$(f_u, d_1) \rightsquigarrow_{\overline{C}} (f_v, d_2).$$

Reachability on $\overline{C}$ using POT $\overline{T}$

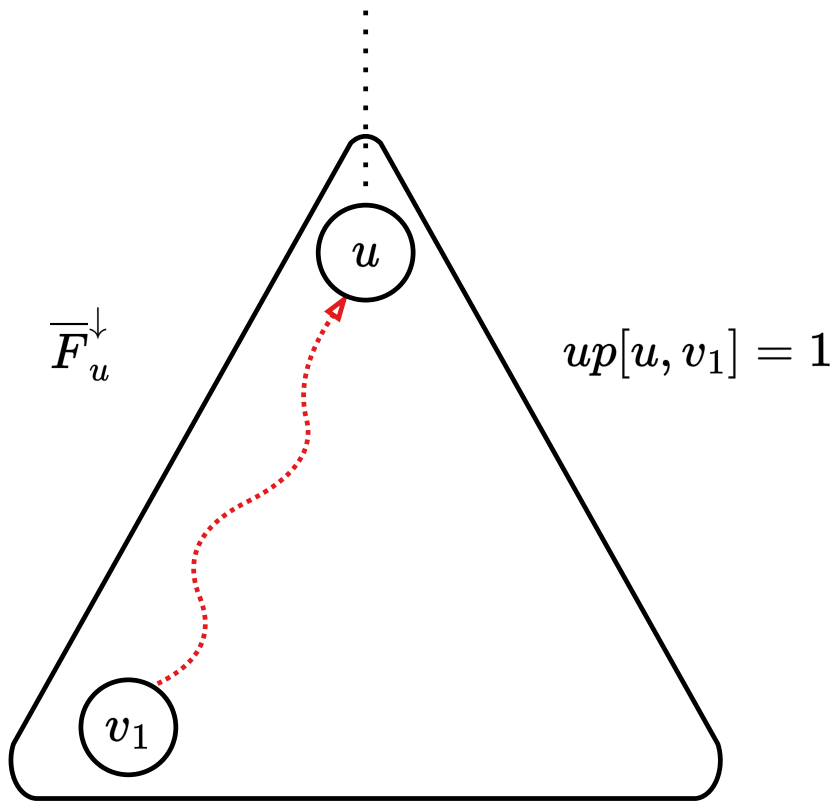
Let $\overline{F}_u^{\downarrow}$ be the set of descendants of u in $\overline{T}$.



Reachability on $\overline{C}$ using POT $\overline{T}$

For every u and every descendant v of it, define:

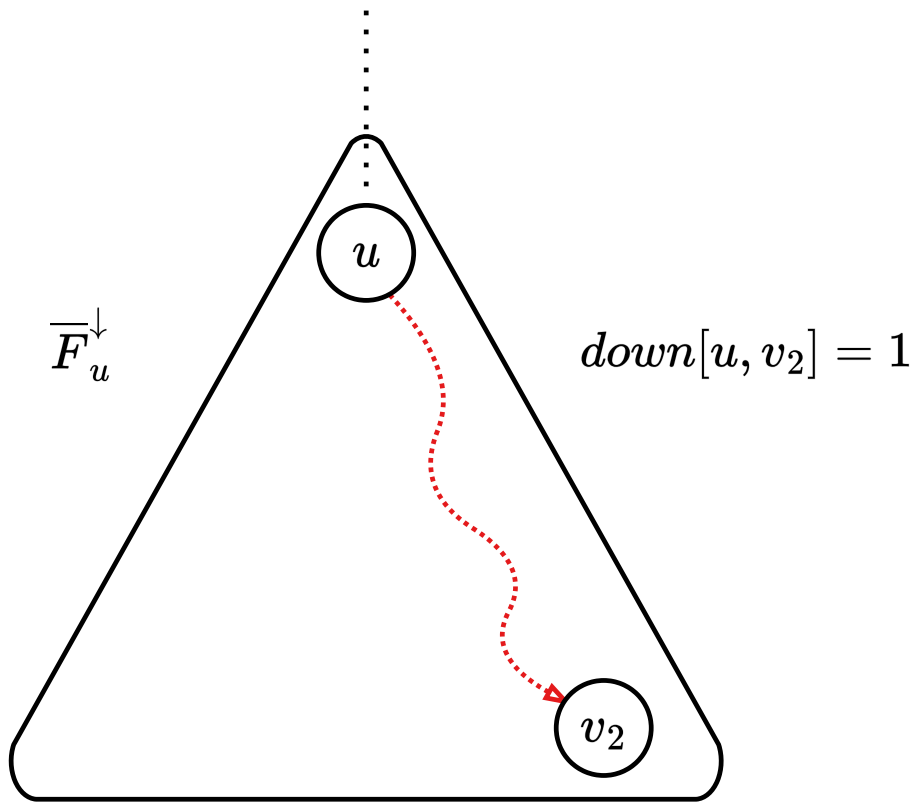
$$up[u, v] := \begin{cases} 1 & \text{there is a path from } v \text{ to } u \text{ in } \overline{C}[\overline{F}_u^{\downarrow}] \\ 0 & \text{otherwise} \end{cases}.$$



Reachability on $\overline{C}$ using POT $\overline{T}$

For every u and every descendant v of it, define:

$$down[u, v] := \begin{cases} 1 & \text{there is a path from } u \text{ to } v \text{ in } \overline{C}[\overline{F}_u^{\downarrow}] \\ 0 & \text{otherwise} \end{cases}.$$



Reachability on $\overline{C}$ using POT $\overline{T}$: preprocessing

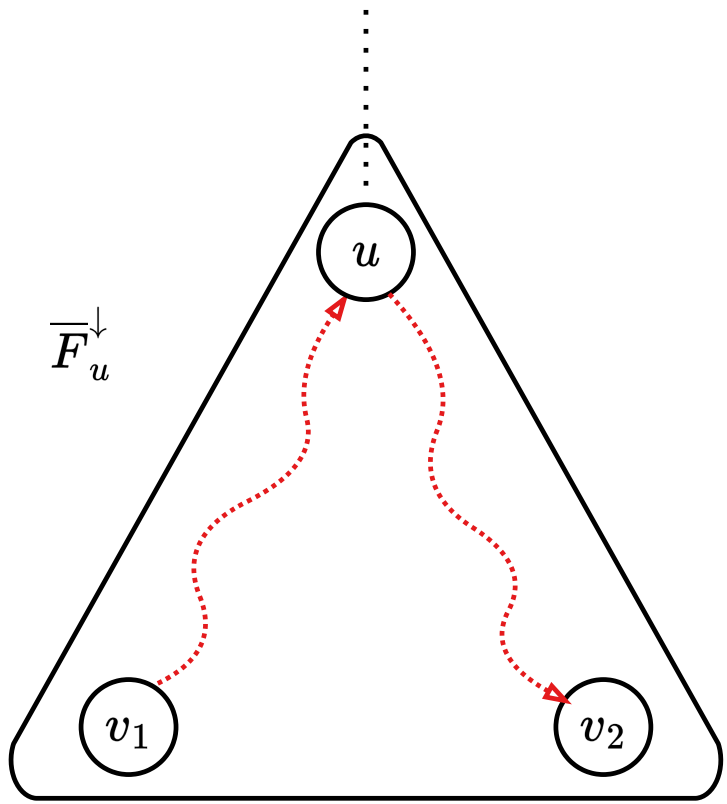
Preprocessing: compute *up* and *down*.

Reachability on $\overline{C}$ using POT $\overline{T}$: preprocessing

Preprocessing: compute *up* and *down*.

— $down[u, \cdot]$ is computed by a DFS from u, ignoring edges leaving $\overline{C}[\overline{F}_u^{\downarrow}]$.

— $up[u, \cdot]$ is similarly computed by reversing edges of $\overline{C}$.

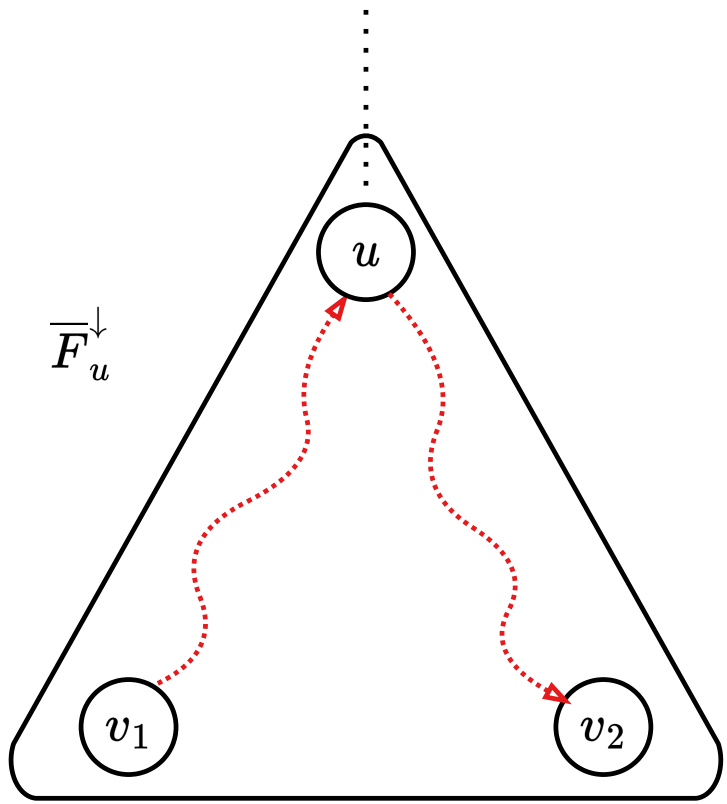


Reachability on $\overline{C}$ using POT $\overline{T}$: preprocessing

Preprocessing: compute *up* and *down*.

— $down[u, \cdot]$ is computed by a DFS from u, ignoring edges leaving $\overline{C}[\overline{F}_u^{\downarrow}]$.

— $up[u, \cdot]$ is similarly computed by reversing edges of $\overline{C}$.

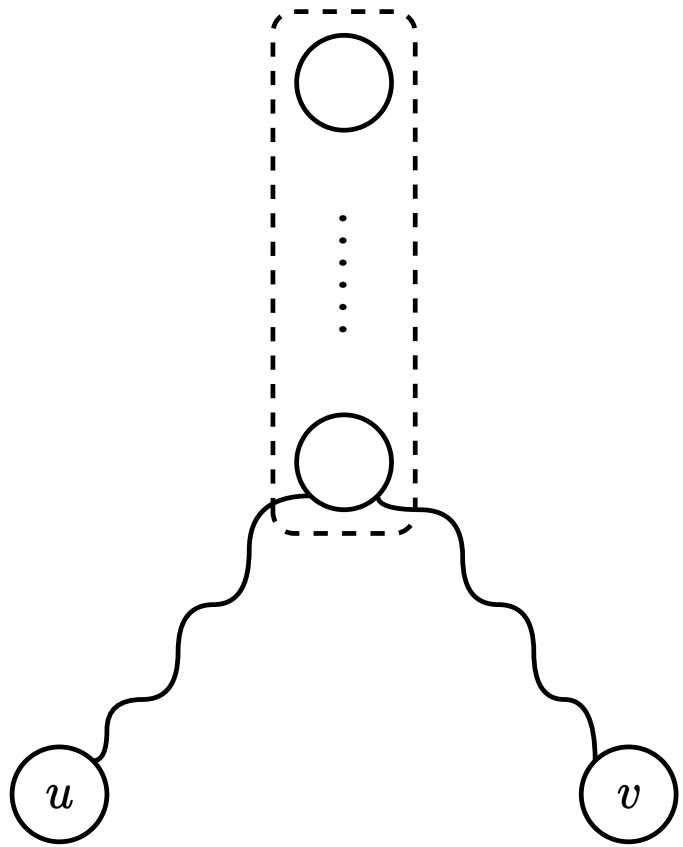


Each edge is traversed $O(\text{depth of } \overline{T}) = O(\mathtt{td} \cdot D)$ times.

$\Longrightarrow$ *up* and *down* can be computed in $O(n \cdot D^3 \cdot \mathtt{td})$ time.

Reachability on $\overline{C}$ using POT $\overline{T}$: queries

For any u, v in $\overline{C}$, let A be the set of their common ancestors in $\overline{T}$.

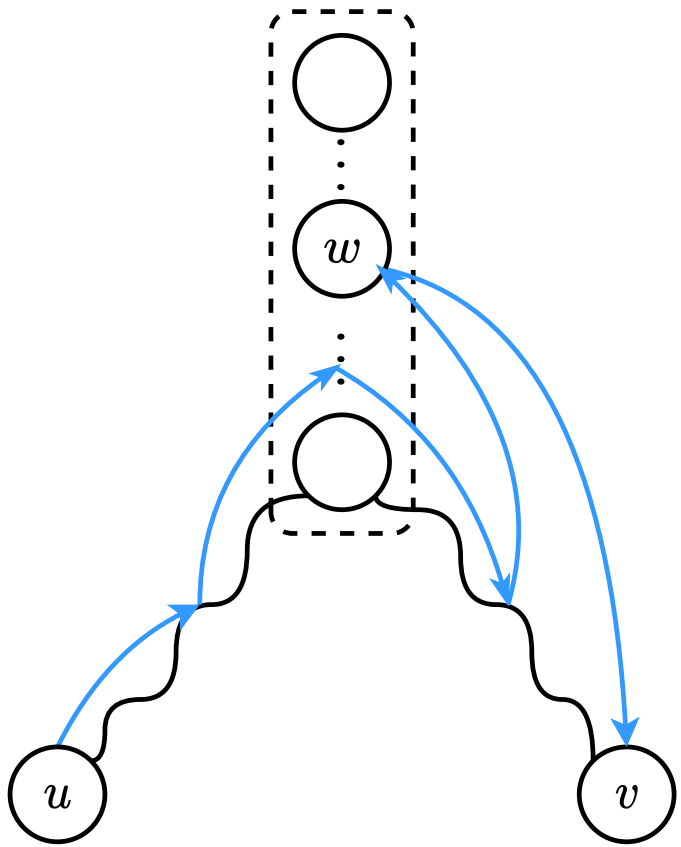


Reachability on $\overline{C}$ using POT $\overline{T}$: queries

By the cut property of POTs, any path ρ from u to v in $\overline{C}$ has:

$$\rho \cap A \neq \emptyset$$

.

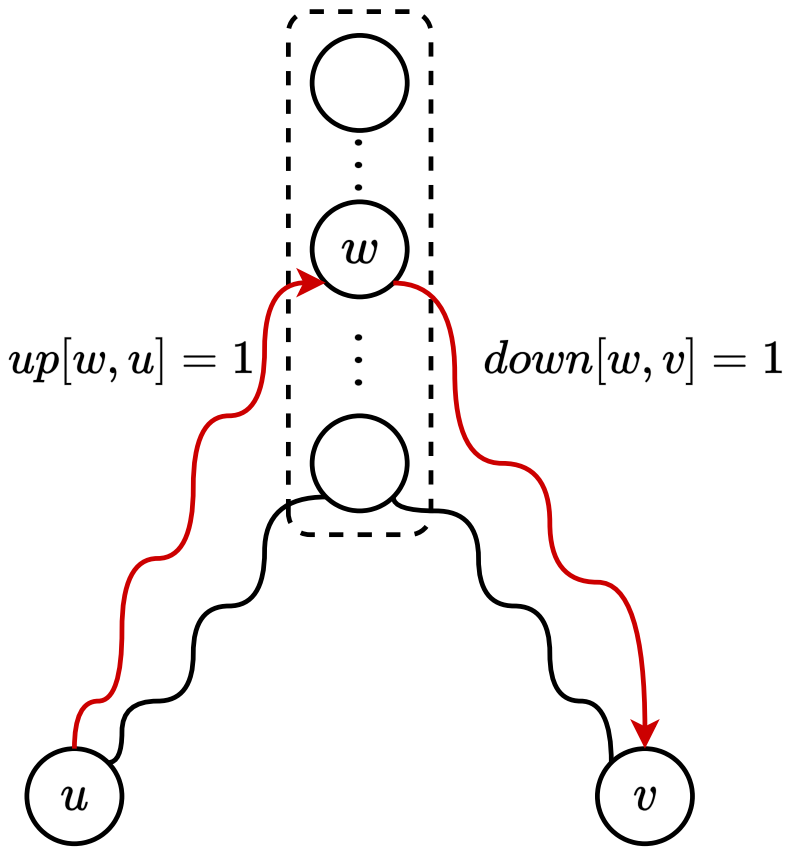


Reachability on $\overline{C}$ using POT $\overline{T}$: queries

Let $w \in A$ be the highest node in $\rho \cap A$.

We must have:

$$up[w, u] = 1 \wedge down[w, v] = 1.$$



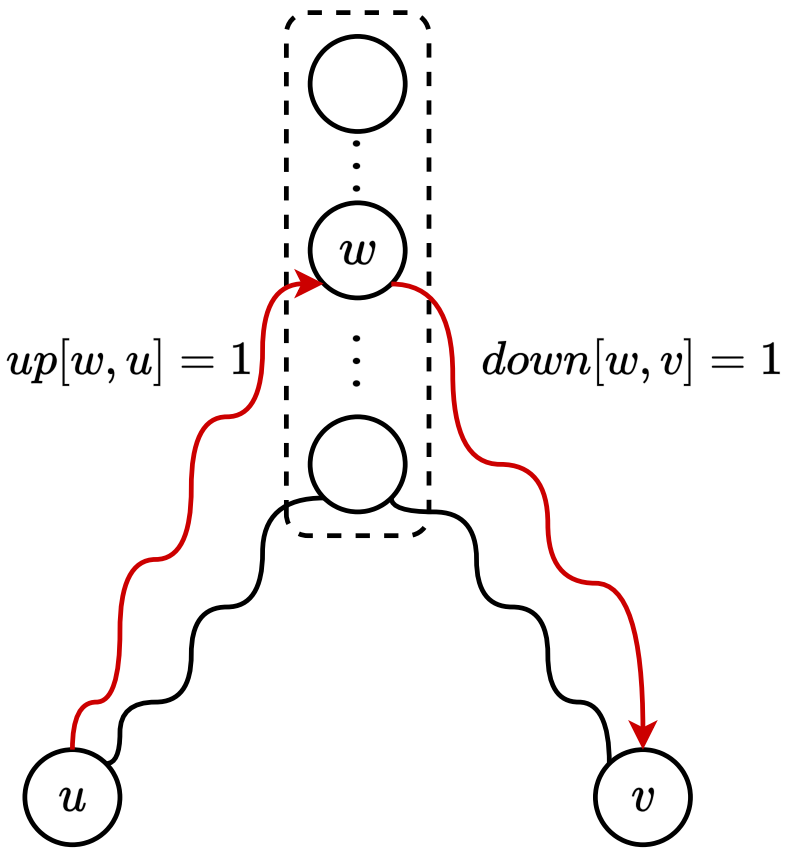
Reachability on $\overline{C}$ using POT $\overline{T}$: queries

$$u \rightsquigarrow_{\overline{C}} v \text{ iff } \exists w \in A \text{ s.t. } up[w,u] = 1 \wedge down[w,v] = 1.$$

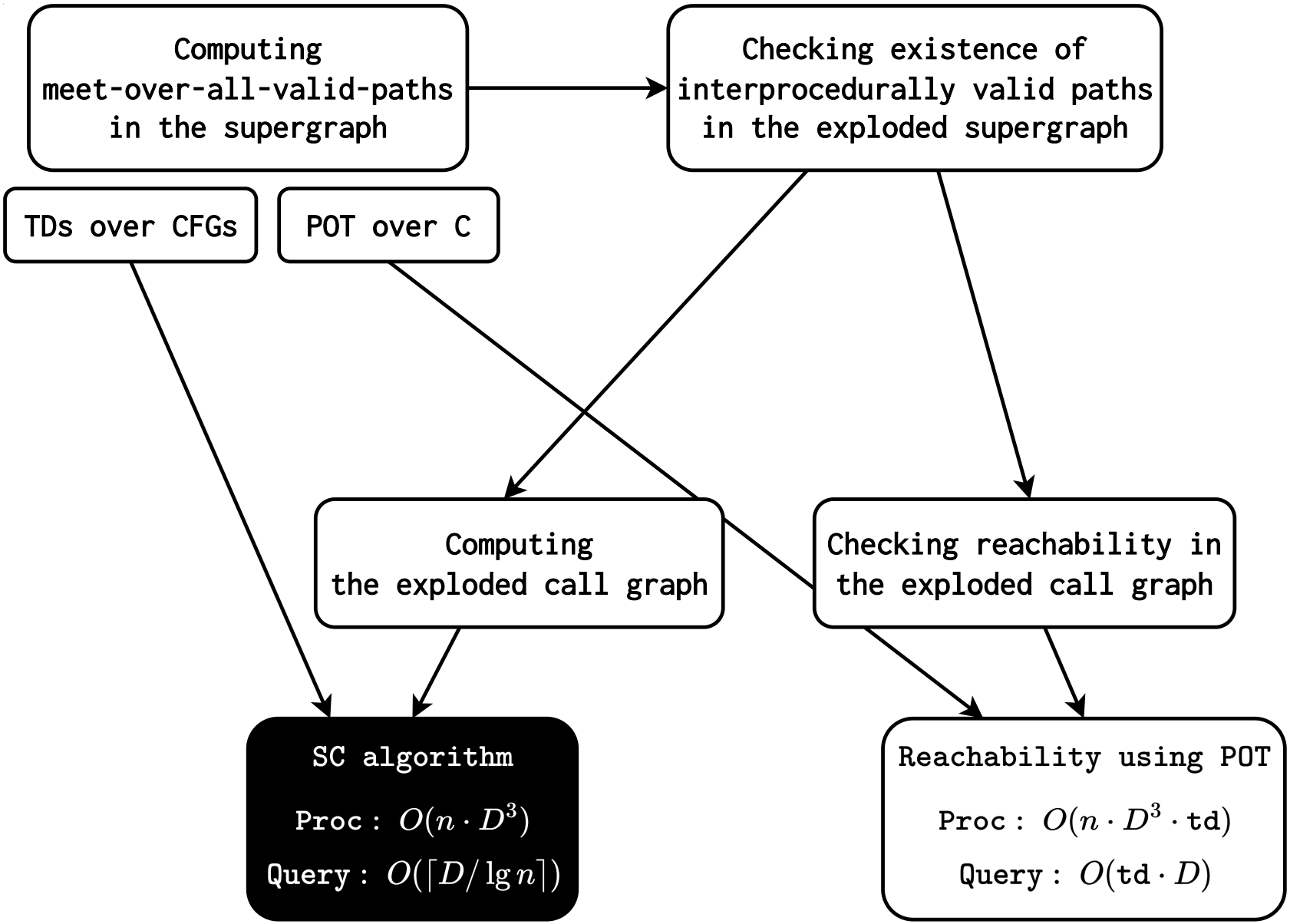
To answer a query $\langle u, v \rangle$: we iterate over w and check if

$$up[w,u] = 1 \wedge down[w,v] = 1$$

$\implies$ query time $O(\text{depth of } \overline{T}) = O(\texttt{td} \cdot D)$.



Where we are

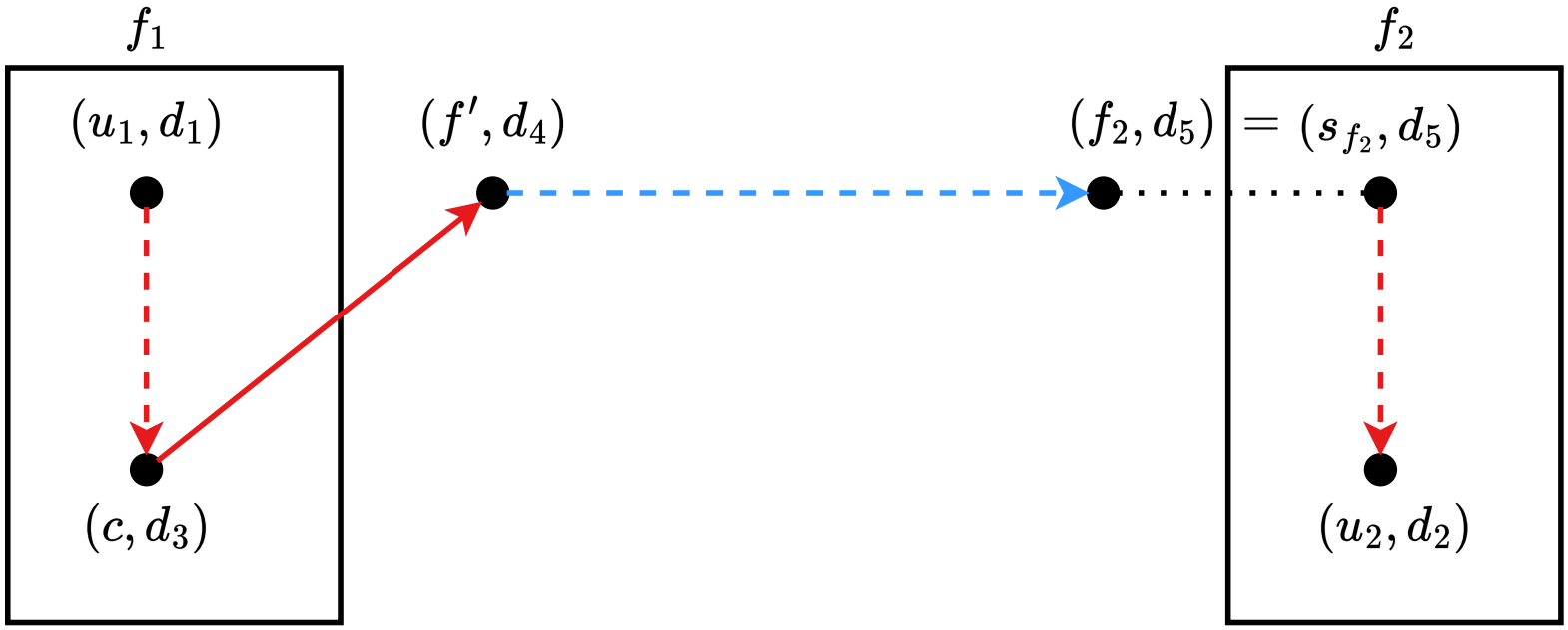

Computing
meet-over-all-valid-paths
in the supergraph
Checking existence of
interprocedurally valid paths
in the exploded supergraph
TDs over CFGs
POT over C
Computing
the exploded call graph
Checking reachability in
the exploded call graph
SC algorithm
Proc : $O(n \cdot D^3)$
Query : $O(\lceil D / \lg n \rceil)$
Reachability using POT
Proc : $O(n \cdot D^3 \cdot \mathtt{td})$
Query : $O(\mathtt{td} \cdot D)$

Answering a general query on $\overline{G}$

$\overline{C}$ helps us compute $\mathtt{Q}((s_{f_u}, d_1), (s_{f_v}, d_2))$, which is a restricted form.

Answering a general query on $\overline{G}$

$\overline{C}$ helps us compute $\mathtt{Q}((s_{f_u}, d_1), (s_{f_v}, d_2))$, which is a restricted form.



To compute $\mathtt{Q}((u_1, d_1), (u_2, d_2))$:

- Iterate over calls (c, d_3) in the same function as u_1.
- If (c, d_3) calls $(s_{f'}, d_4)$, perform:
  - *Same-context query:* check $\mathtt{SCQ}((u_1, d_1), (c, d_3))$.
  - *Reachability query on $\overline{C}$:* check $(f', d_4) \rightsquigarrow_{\overline{C}} (f_2, d_5)$, and
  - *Same-context query:* check $\mathtt{SCQ}((s_{f_2}, d_5), (u_2, d_2))$.

Answering a general query on $\overline{G}$

Done!

Runtime

- Preprocessing: $O(n \cdot D^3 \cdot \texttt{td})$.
- Query: $O(D^3 \cdot \texttt{td})$.

where:

- $n =\#$ lines in the program
- D number of possible data facts.
- $\texttt{td} =$ treedepth of the call graph.

Runtime

- Preprocessing: $O(n \cdot D^3 \cdot \texttt{td})$. $\leftarrow O(n)$ in practice.
- Query: $O(D^3 \cdot \texttt{td})$. $\leftarrow O(1)$ in practice.

where:

- $n = \#$ lines in the program
- D number of possible data facts.
- $\texttt{td} =$ treedepth of the call graph.

Table of Contents

Experiments: setup

- Ran the algorithm on real-world programs from DaCapo benchmarks.
- Extracted the CFGs and call graph using Soot.
- Used PACE solvers [7, 8] to compute:
  - TDs of the CFGs of small width.
  - POT over the call graph of small depth.
- On each benchmark we ran reachability, null-pointer, and possibly-uninitialized variables analyses.
- For a program of n lines, we generate n random queries.
- Ran each analysis on:
  - (`PARAM`) our algorithm,
  - (`IFDS`) standard IFDS algorithm [1], and
  - (`DEM`) its demand version [2].

  timing out at 10 minutes.

Experiments: results

Average/maximum are over 13 programs from DaCapo benchmarks.

- $|V|$ ($\approx$ lines of code):
  - Average: 22.7K.
  - Maximum: 58.5K.
- Number of functions:
  - Average: 803.1.
  - Maximum: 2028.
- Treewidth of CFGs:
  - Average: 9.1.
  - Maximum: 10.
- Treedepth of call graphs:
  - Average: 43.8.
  - Maximum: 135.

Experiments: reachability

Preprocessing:

- Average: 0.93s.
- Maximum: 1.53s.

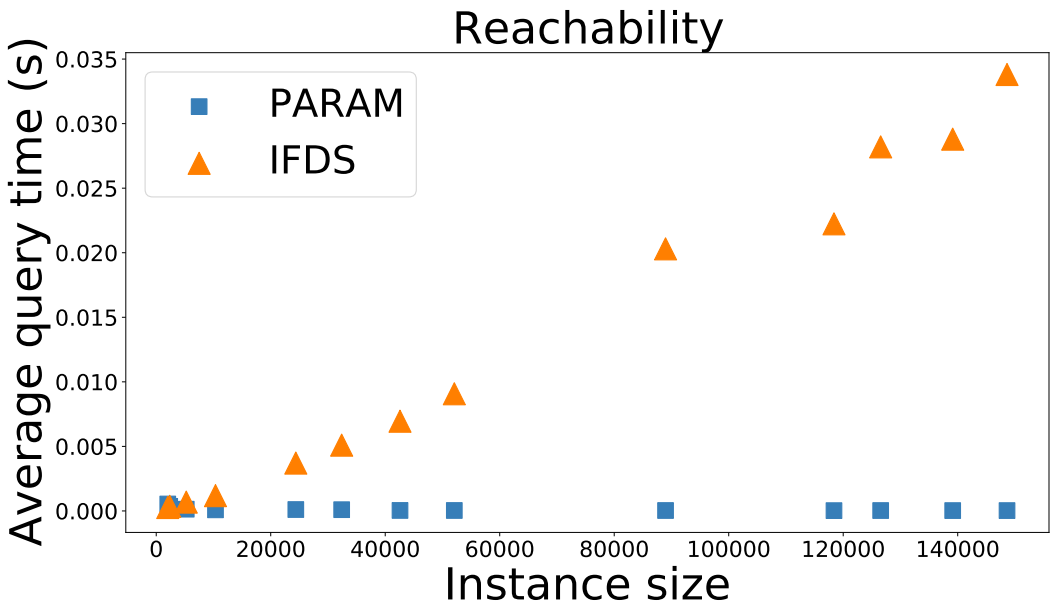
Query:

- Average: 0.11ms.
- Maximum: 0.53ms.

`IFDS`'s query:

- Average: 12.3ms.
- Maximum: 33.80ms.

`IFDS/PARAM`: 390.55.



Experiments: reachability analysis

Preprocessing:

- Average: 0.93s.
- Maximum: 1.53s.

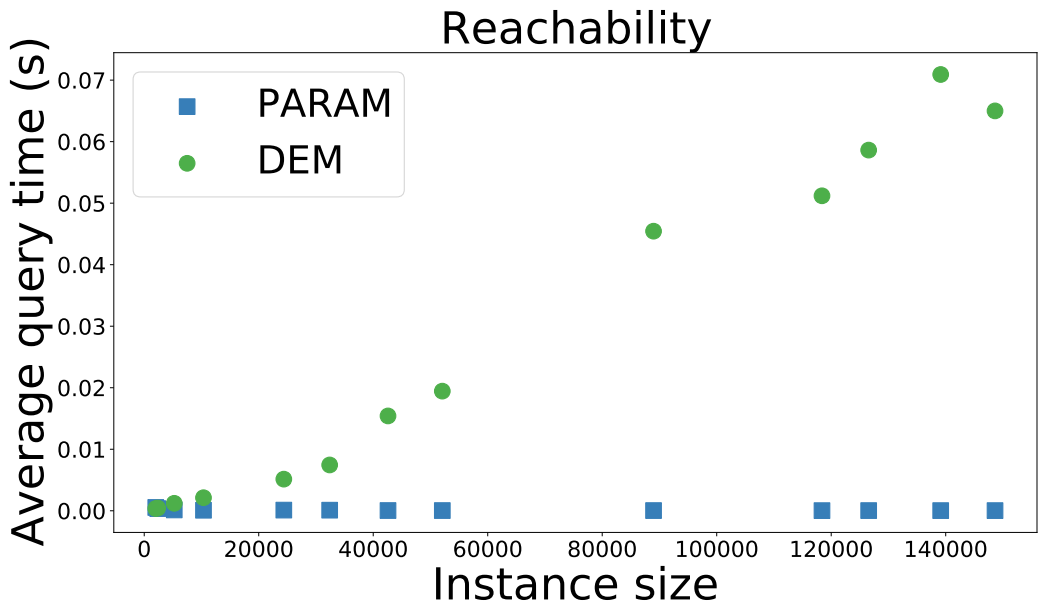
Query:

- Average: 0.11ms.
- Maximum: 0.53ms.

`DEM`'s query:

- Average: 26.36ms.
- Maximum: 70.91ms.

`DEM/PARAM`: 848.13.



Experiments: null-pointer analysis

Preprocessing:

- Average: 41.80s.
- Maximum: 140.85s.

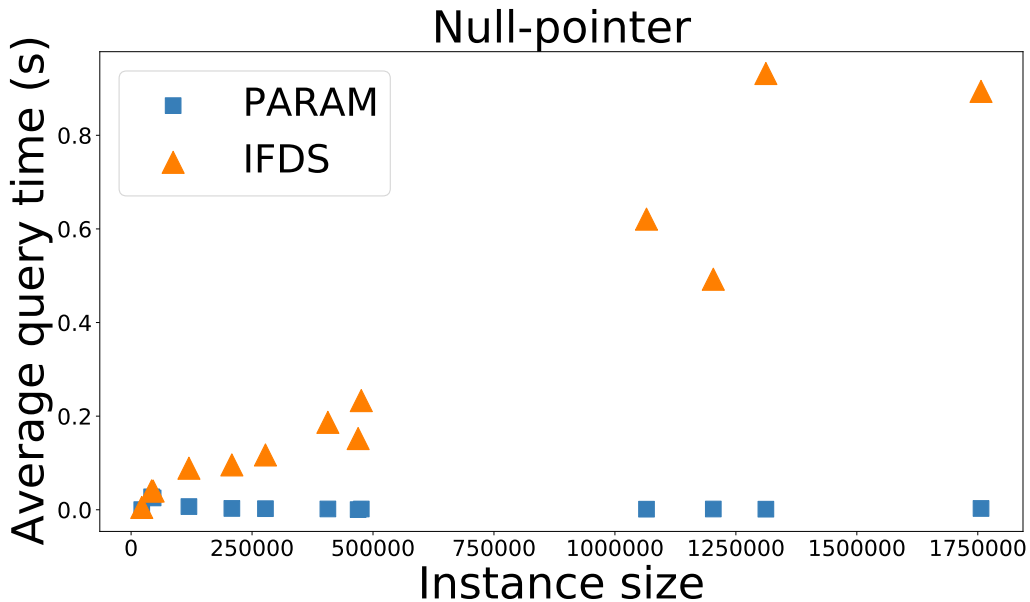
Query:

- Average: 5.84ms.
- Maximum: 27.63ms.

`IFDS`'s query:

- Average: 299.91ms.
- Maximum: 932.04ms.

`IFDS/PARAM`: 202.92.



Experiments: null-pointer analysis

Preprocessing:

- Average: 41.80s.
- Maximum: 140.85s.

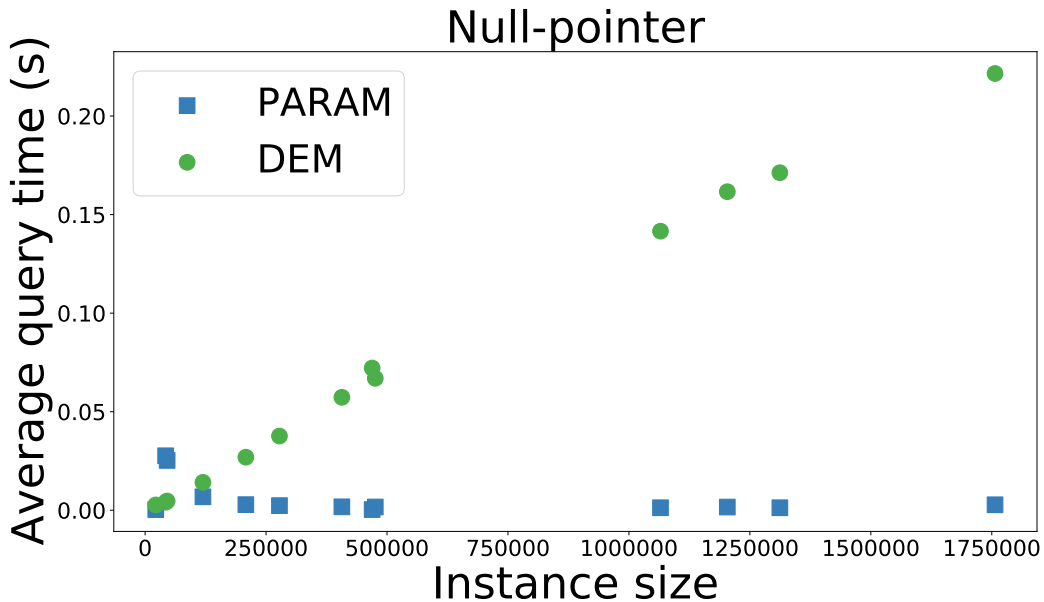
Query:

- Average: 5.84ms.
- Maximum: 27.63ms.

`DEM`'s query:

- Average: 75.58ms.
- Maximum: 221.58ms.

`DEM/PARAM`: 56.86.



Experiments: possibly-uninitialized variables analysis

Preprocessing:

- Average: 89.44s.
- Maximum: 265.31.

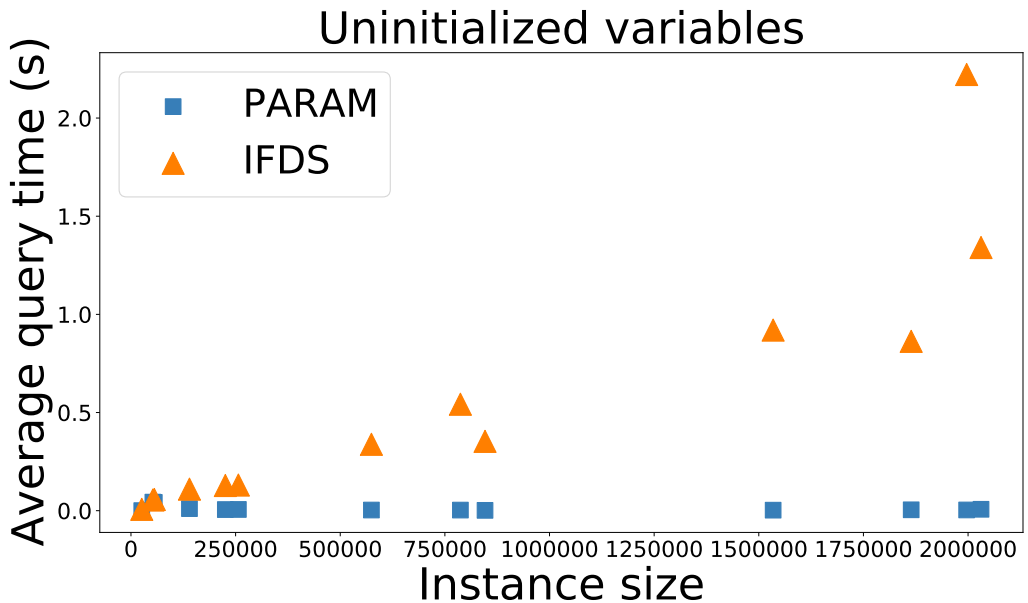
Query:

- Average: 10.39ms.
- Maximum: 43.70ms.

`IFDS`'s query:

- Average: 543.53ms.
- Maximum: 2221.90ms.

`IFDS/PARAM`: 143.96



Experiments: possibly-uninitialized variables analysis

Preprocessing:

- Average: 89.44s.
- Maximum: 265.31.

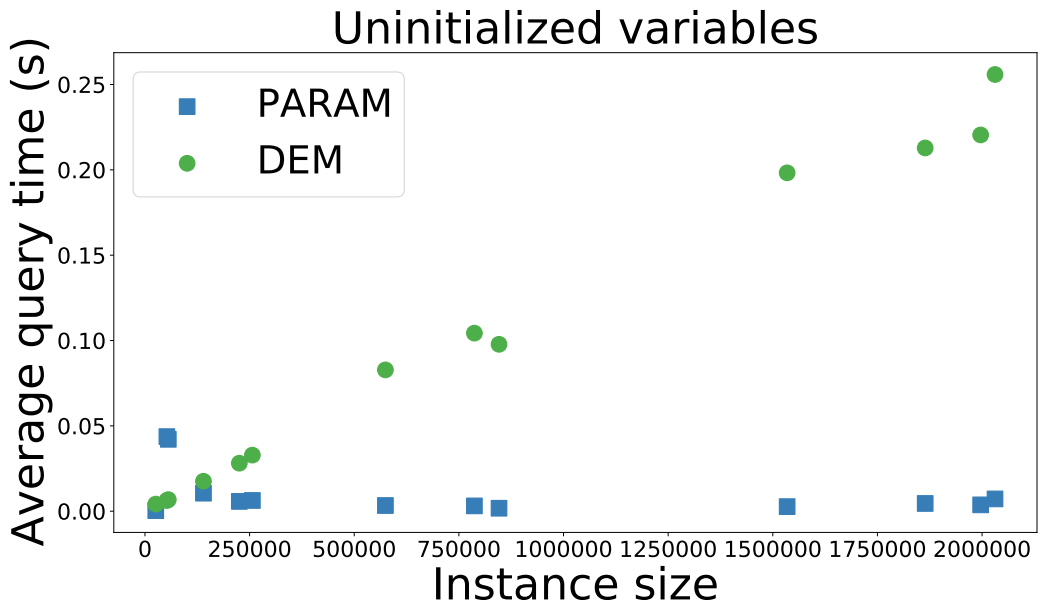
Query:

- Average: 10.39ms.
- Maximum: 43.70ms.

`DEM`'s query:

- Average: 97.54ms.
- Maximum: 255.86ms

`DEM/PARAM`: 27.25.



Conclusion

- Identify and exploit a new sparsity parameter: treedepth of call graphs.
- Fast parameterized algorithm for general on-demand IFDS.
- Theoretical improvement over previous works.
- Experimentally outperforming the standard IFDS algorithms by two orders of magnitude.

Approach	General?	Preprocessing	Query
Reps et. al. (POPL'95)	✓	$O\left(n \cdot D^3\right)$	
Horwitz et. al. (FSE'95)	✓	$O\left(n \cdot D^3\right)$	
Chatterjee et. al. (ESOP'20)	✗	$O\left(n \cdot D^3\right)$	$O\left(\lceil D / \lg n \rceil\right)$
Our result	✓	$O\left(n \cdot D^3 \cdot \mathtt{td}\right)$	$O\left(D^3 \cdot \mathtt{td}\right)$

Publications

— A.K. Goharshady, A.K. Zaher, *"Efficient Interprocedural Data-Flow Analysis using Treedepth and Treewidth,"* in VMCAI'23.

— G.K. Conrado, A.K. Goharshady, K. Kochekov, Y.C. Tsai, A.K. Zaher, *"Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis,"* in OOPSLA'23.

References

[1] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in POPL, 1995, pp. 49–61.

[2] S. Horwitz, T. W. Reps, and S. Sagiv, "Demand interprocedural dataflow analysis," in FSE, 1995, pp. 104–115.

[3] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and A. Pavlogiannis, "Optimal and perfectly parallel algorithms for on-demand data-flow analysis," in ESOP, 2020, pp. 112–140.

[4] M. Thorup, "All structured programs have small tree-width and good register allocation," Inf. Comput., vol. 142, no. 2, pp. 159–181, 1998.

[5] H. L. Bodlaender, "A linear time algorithm for finding tree-decompositions of small treewidth," in STOC, 1993, pp. 226–234.

[6] W. Nadara, M. Pilipczuk, and M. Smulewicz, "Computing treedepth in polynomial space and linear FPT time," CoRR, vol. abs/2205.02656, 2022.

[7] H. Dell, C. Komusiewicz, N. Talmon, and M. Weller, "The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration," in IPEC, 2018, pp. 30:1–30:12.

[8] Łukasz Kowalik, M. Mucha, W. Nadara, M. Pilipczuk, M. Sorge, and P. Wygocki, "The PACE 2020 Parameterized Algorithms and Computational Experiments Challenge: Treedepth," in IPEC, 2020, pp. 37:1–37:18.