New Algorithms for Static Analysis via Dyck Reachability

Andreas Pavlogiannis

4th Inria/EPFL Workshop

February 15, 2018

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE


REMOTE HOME CONTROL
ONLINE HOME AUTOMATION
ECO MODE
22°C


MEDICAL

- Does it crash?
- Is it efficient?
- Does it leak information?
- Is it safe?
- Is it responsive?

Static Analysis

- Program analysis is hard
- All non-trivial problems undecidable
- Relax ...

Static Analysis

- Program analysis is hard
- All non-trivial problems undecidable
- Relax . . .
  - . . . your ambitions
  - . . . your model

Static Analysis

- Program analysis is hard
- All non-trivial problems undecidable
- Relax . . .
  - . . . your ambitions
  - **. . . your model**

Static analysis

- Lightweight, "clever" scans of the program
- Detect "obvious" bugs

1. Fast
2. On demand
3. Dynamic

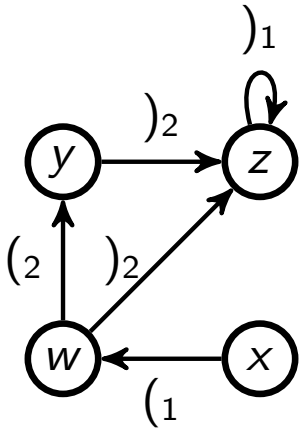
Static Analysis via Dyck Reachability

Dyck Reachability

$$\Sigma = \{(_1,)_1, \ldots (_k,)_k\} \cup \{\epsilon\}$$

$$G = (V, E, \lambda : E \to \Sigma)$$

$$\mathcal{S} \to \mathcal{S}\ \mathcal{S} \mid (_1\ \mathcal{S}\ )_1 \mid \ldots \mid (_k\ \mathcal{S}\ )_k \mid \epsilon$$



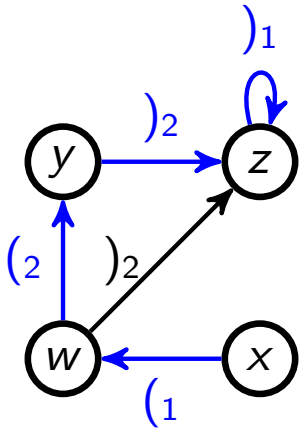
Dyck Reachability

$$\Sigma = \{(_1,)_1, \ldots (_k,)_k\} \cup \{\epsilon\}$$

$$G = (V, E, \lambda : E \to \Sigma)$$

$$\mathcal{S} \to \mathcal{S}\ \mathcal{S} \mid (_1\ \mathcal{S}\ )_1 \mid \ldots \mid (_k\ \mathcal{S}\ )_k \mid \epsilon$$

$$P : x \rightsquigarrow z \quad \text{with} \quad \lambda(P) = (_1(_2)_2)_1$$

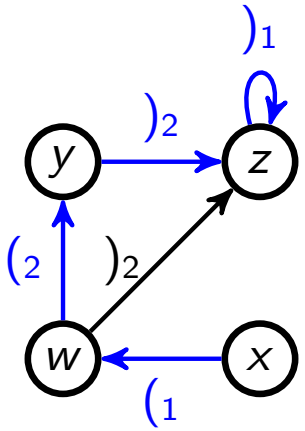


Dyck Reachability

$$\Sigma = \{(_1,)_1, \ldots (_k,)_k\} \cup \{\epsilon\}$$

$$G = (V, E, \lambda : E \to \Sigma)$$

$$\mathcal{S} \to \mathcal{S}\ \mathcal{S} \mid (_1\ \mathcal{S}\ )_1 \mid \ldots \mid (_k\ \mathcal{S}\ )_k \mid \epsilon$$



$$P : x \rightsquigarrow z \quad \text{with} \quad \lambda(P) = (_1(_2)_2)_1$$

- Alias analysis
- Data-dependence analysis
- Data-flow analysis
- Shape analysis
- Impact analysis
- Bloat analysis
- Type-based flow analysis
- Program slicing

Algorithms

Q: Is v Dyck-reachable from u?

- Dyck languages are a class of CFL
- Solution similar to CYK for CFL parsing
- $O(n^3)$ [Yannakakis '90]
- $O(n^3/\log n)$ [Chaudhuri '08]
- Often prohibitive for lightweight static analysis

Algorithms

Q: Is v Dyck-reachable from u?

- Dyck languages are a class of CFL
- Solution similar to CYK for CFL parsing
- $O(n^3)$ [Yannakakis '90]
- $O(n^3/\log n)$ [Chaudhuri '08]
- Often prohibitive for lightweight static analysis

Can do better if the graph is simple!

1. Low treewidth
2. Bidirected

Static Analysis

- Analyze source code without executing it

Typical paradigm in static analysis: reduce the problem to a graph problem P:
Input: A program of methods M_i

1. Extract control flow graphs G_i
2. Annotate G_i
3. Run best graph algorithm for P on G

Static Analysis

- Analyze source code without executing it

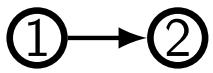
Typical paradigm in static analysis: reduce the problem to a graph problem P:
Input: A program of methods M_i

1. Extract control flow graphs G_i
2. Annotate G_i
3. Run best graph algorithm for P on G

Data-dependence analysis

- Which variable depends on which others
- Identify Def-Use chains in a program
- LHS depends on RHS

1	$x \leftarrow 2$
2	$y \leftarrow x + 1$



Static Analysis

- Analyze source code without executing it

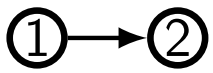
Typical paradigm in static analysis: reduce the problem to a graph problem P:
Input: A program of methods M_i

1. Extract control flow graphs G_i
2. Annotate G_i
3. Run best graph algorithm for P on G

Data-dependence analysis

- Which variable depends on which others
- Identify Def-Use chains in a program
- LHS depends on RHS
  y depends on x if $x \rightsquigarrow y$

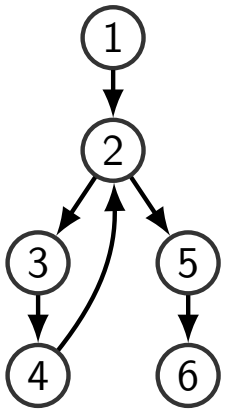
1	$x \leftarrow 2$
2	$y \leftarrow x + 1$



Intra-procedural → Finite Graphs

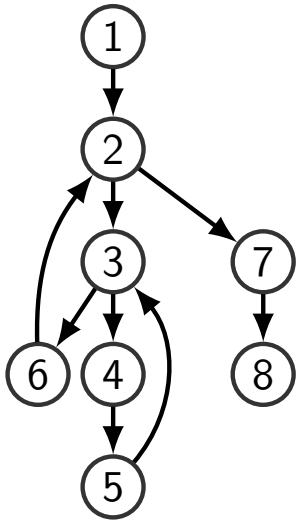
Method 1: dot_vector

```
result ← 0
for i ← 1 to n do
    z ← x[i] · y[i]
    result ← result + z
end
return result
```



Method 2: dot_matrix

```
1 C ← zero matrix of size n × m
2 for i ← 1 to n do
3     for j ← 1 to m do
4         Call dot_vector(A[i, :], B[:, j])
5         C[i, j] ← the value returned by the
            call of line 4
6     end
7 end
8 return C
```



Inter-procedural → Recursive Graphs

Method 1: $f_1()$

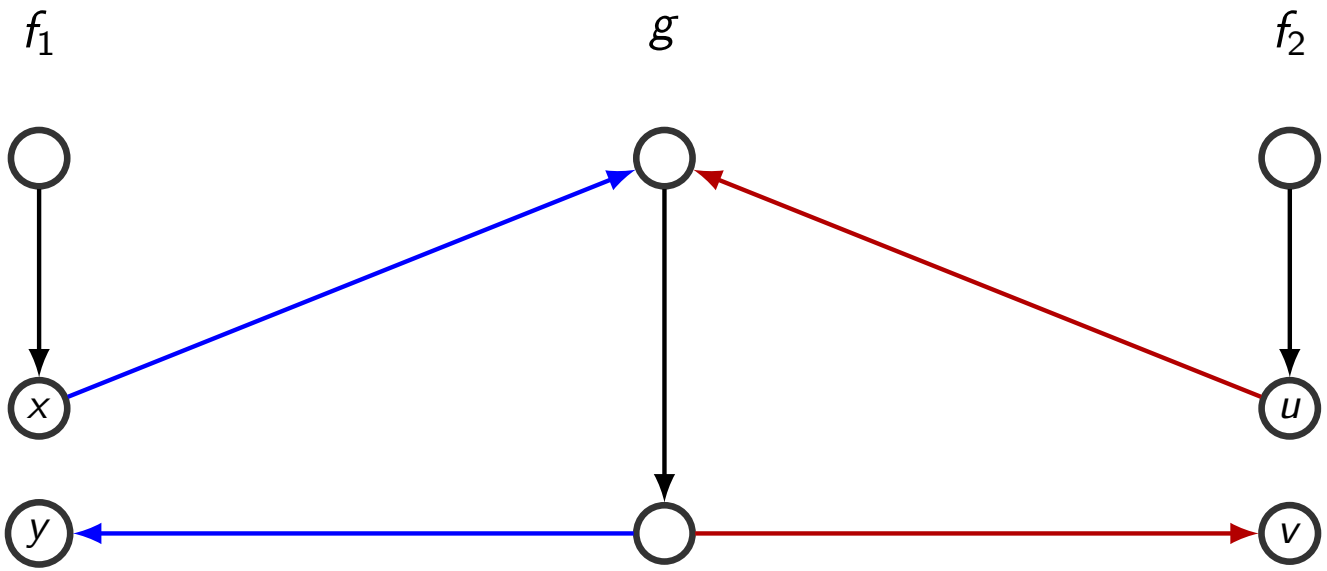
```
1 ...
2 a ← g()
3 ...
```

Method 3: $g()$

```
1 ...
```

Method 2: $f_2()$

```
1 ...
2 b ← g()
3 ...
```



Filter out interprocedurally invalid paths

Inter-procedural → Recursive Graphs

Method 1: $f_1()$

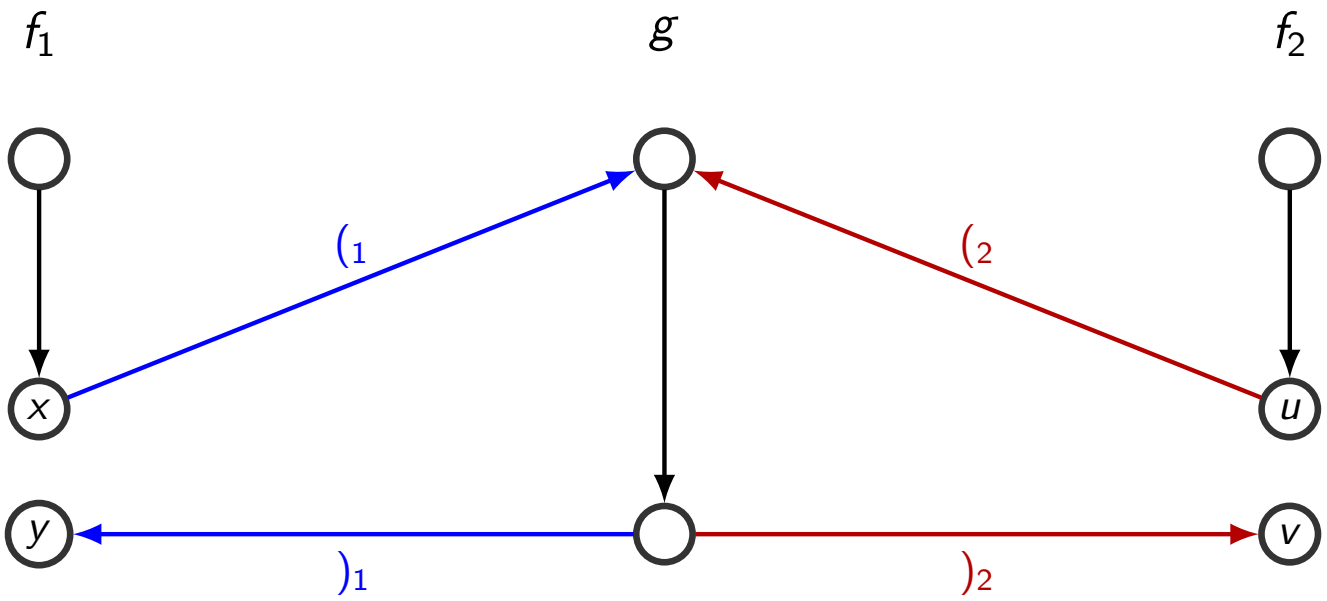
```
1 ...
2 a ← g()
3 ...
```

Method 3: $g()$

```
1 ...
```

Method 2: $f_2()$

```
1 ...
2 b ← g()
3 ...
```



Filter out interprocedurally invalid paths

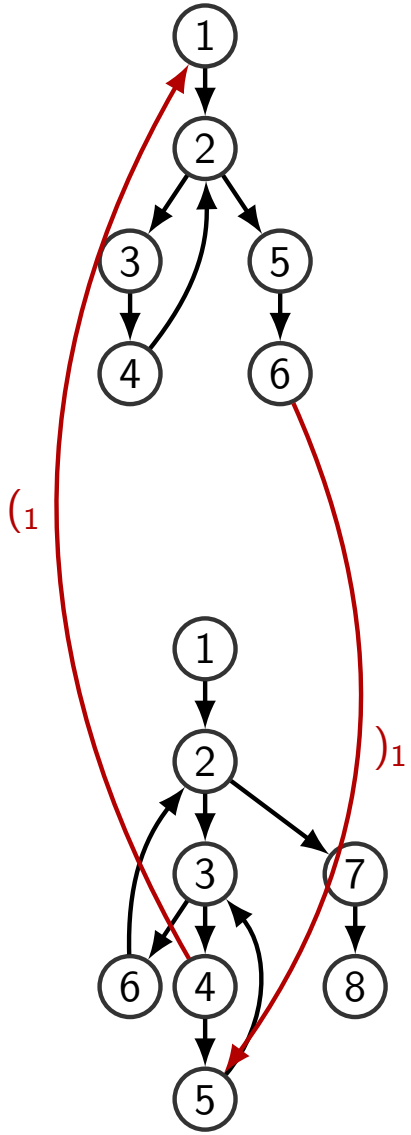
Inter-procedural → Recursive Graphs

Method 4: dot_vector

```
result ← 0
for i ← 1 to n do
    z ← x[i] · y[i]
    result ← result + z
end
return result
```

Method 5: dot_matrix

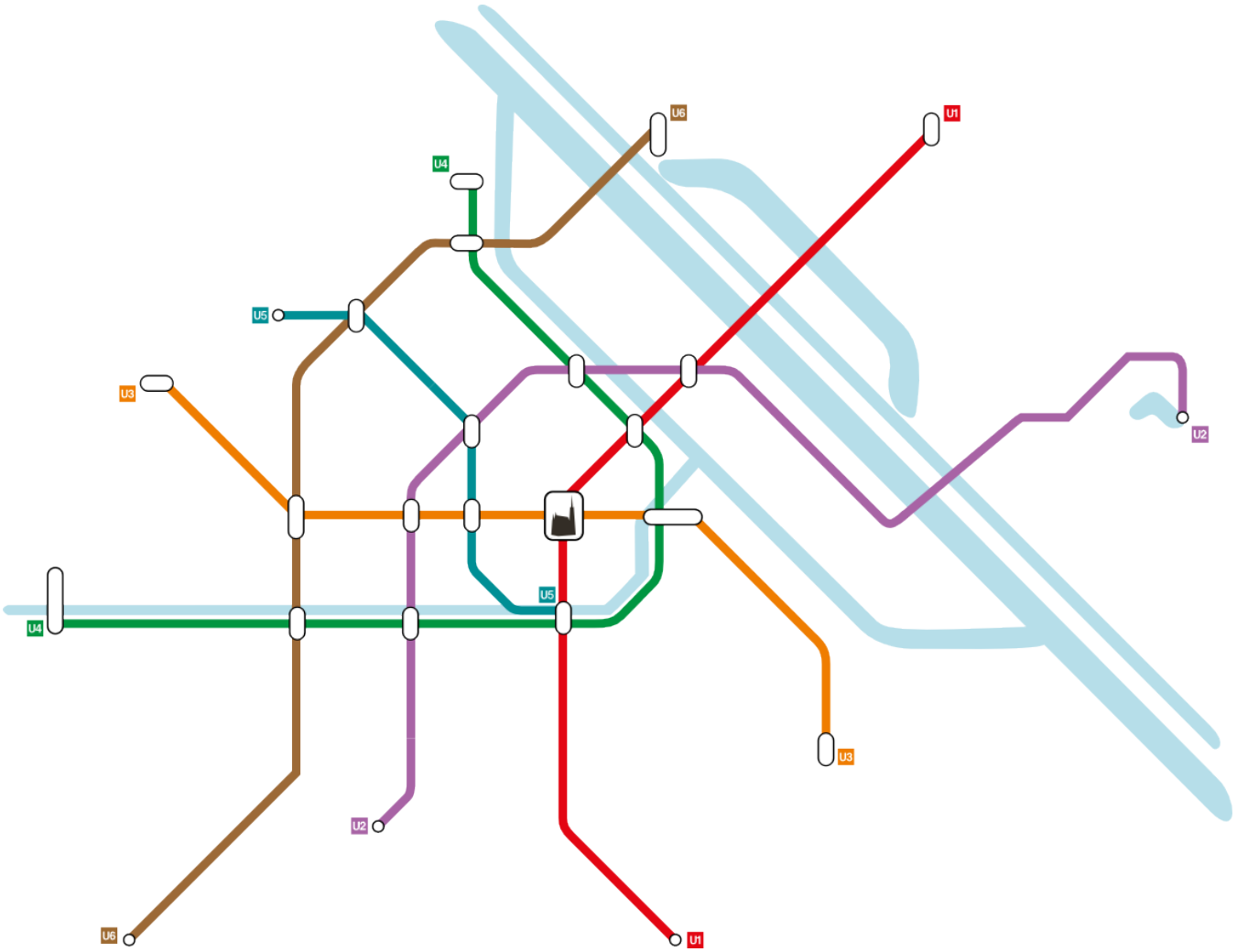
```
1 C ← zero matrix of size n × m
2 for i ← 1 to n do
3     for j ← 1 to m do
4         Call dot_vector(A[i, :], B[:, j])
5         C[i, j] ← the value returned by the
            call of line 4
6     end
7 end
8 return C
```



- Control-flow graphs are simple
- "Similar to trees"
- Precise notion: treewidth

Treewidth

Cops and Robber



Cops and Robber

- You enter the metro and "forget" to buy a ticket
- t inspectors are looking for you in the metro stations
- Everyone is aware of everyone else's position

Cops and Robber

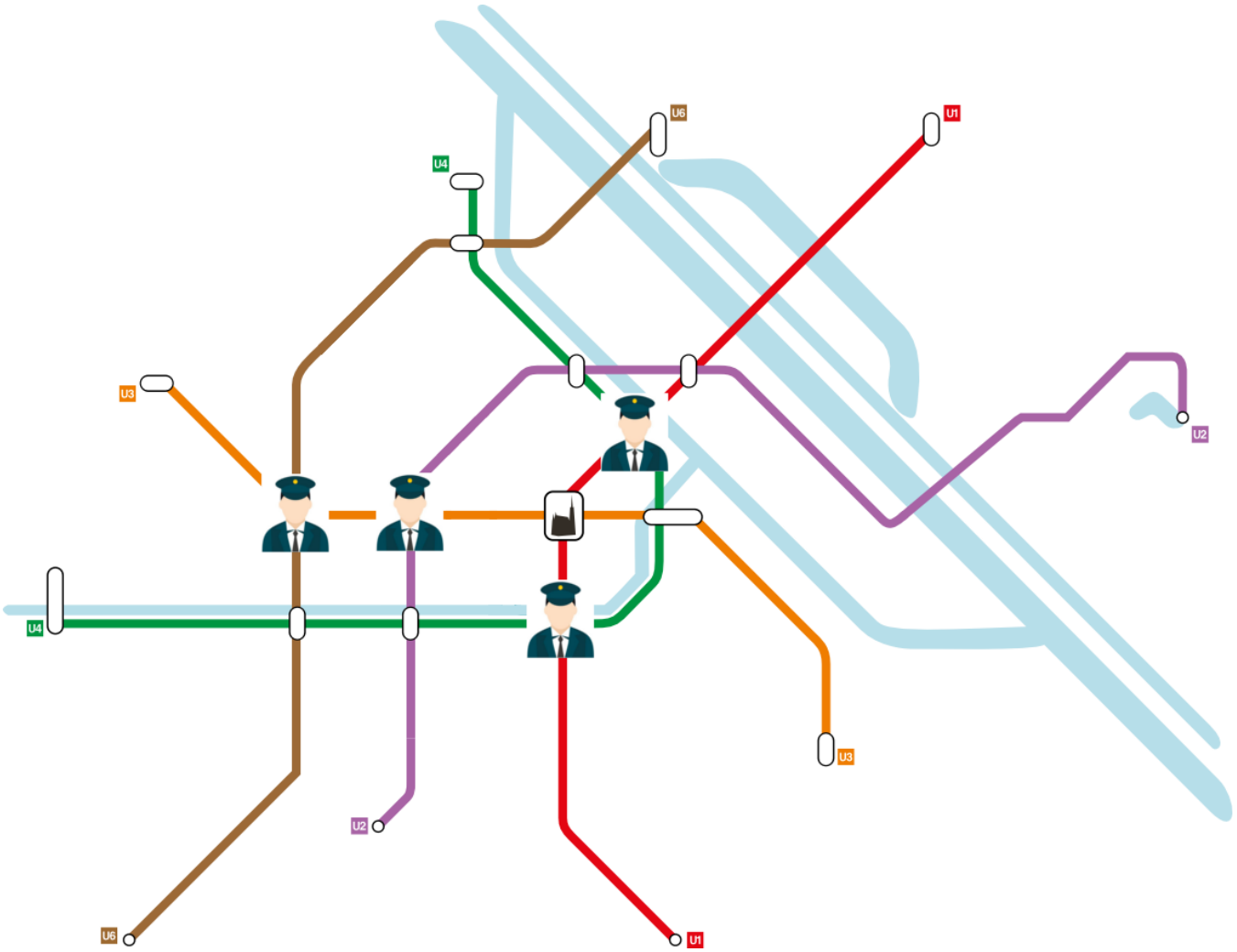
- You enter the metro and "forget" to buy a ticket
- t inspectors are looking for you in the metro stations
- Everyone is aware of everyone else's position
- In every round
  - Some inspectors move between stations to reach you
  - You use the metro system to move between stations
  - You cannot cross an occupied station

Cops and Robber

- You enter the metro and "forget" to buy a ticket
- t inspectors are looking for you in the metro stations
- Everyone is aware of everyone else's position
- In every round
  - Some inspectors move between stations to reach you
  - You use the metro system to move between stations
  - You cannot cross an occupied station

The **treewidth** of the metro graph is the largest t from which you can always escape

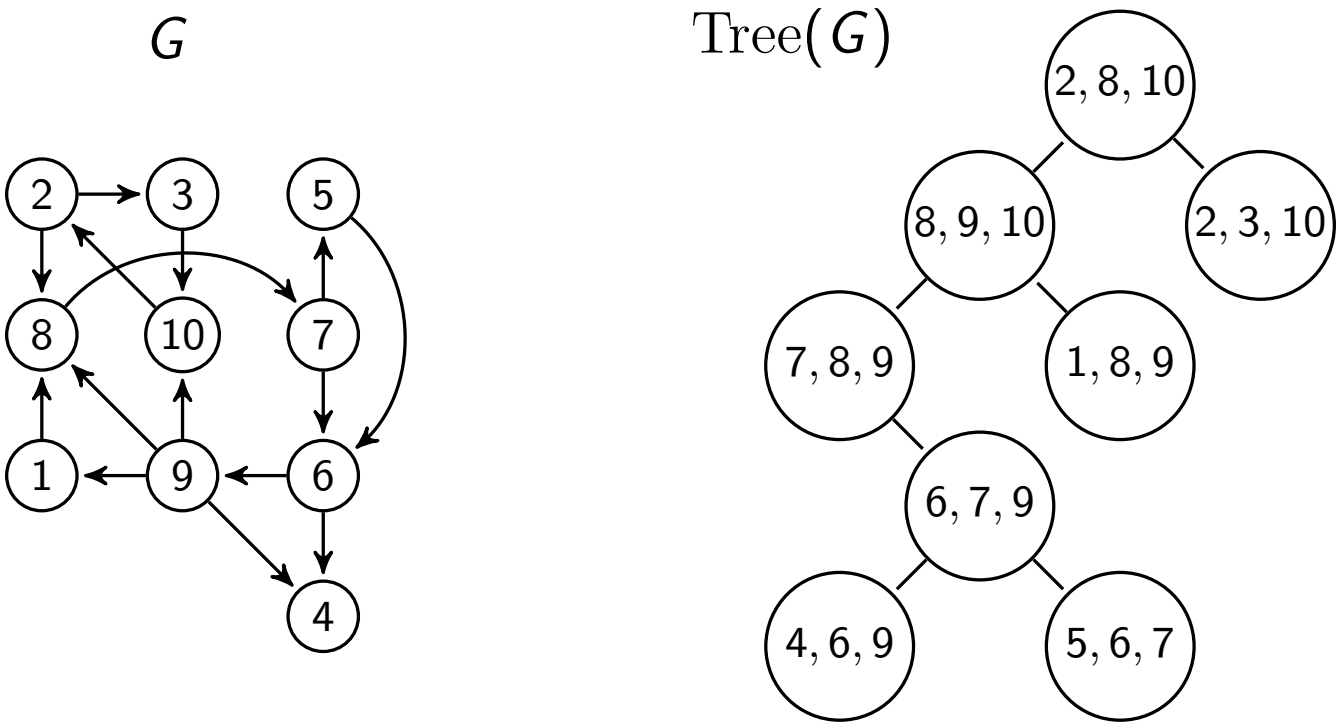
Cops and Robber



Tree Decompositions

Definition (Tree decomposition)

Given a graph $G = (V, E)$, a **tree-decomposition** $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree of bags $B_i \subseteq V$

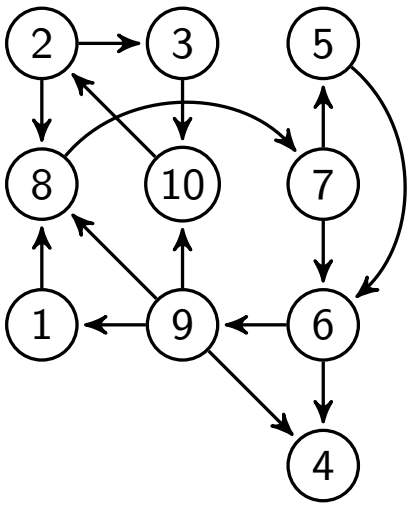


Treewidth

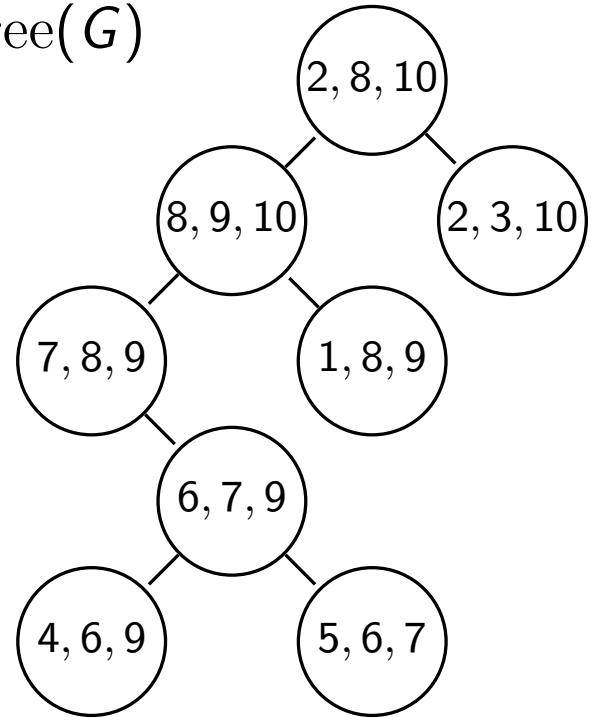
Definition (Treewidth)

The width of $\mathrm{Tree}(G)$ is $\max_i |B_i| - 1$. The **treewidth** of G is the minimum width of a tree decomposition of G.

G



$\mathrm{Tree}(G)$



Treewidth

Definition (Treewidth)

The width of $\mathrm{Tree}(G)$ is $\max_i |B_i| - 1$. The **treewidth** of G is the minimum width of a tree decomposition of G.

CFGs of typical imperative programs have small treewidth

- For goto-free programs [Thorup '98]
  - Pascal $\leqslant 3$
  - C $\leqslant 6$
- In practice small in imperative programs (e.g. Java $\leqslant 7$)

Treewidth

Definition (Treewidth)

The width of $\mathrm{Tree}(G)$ is $\max_i |B_i| - 1$. The **treewidth** of G is the minimum width of a tree decomposition of G.

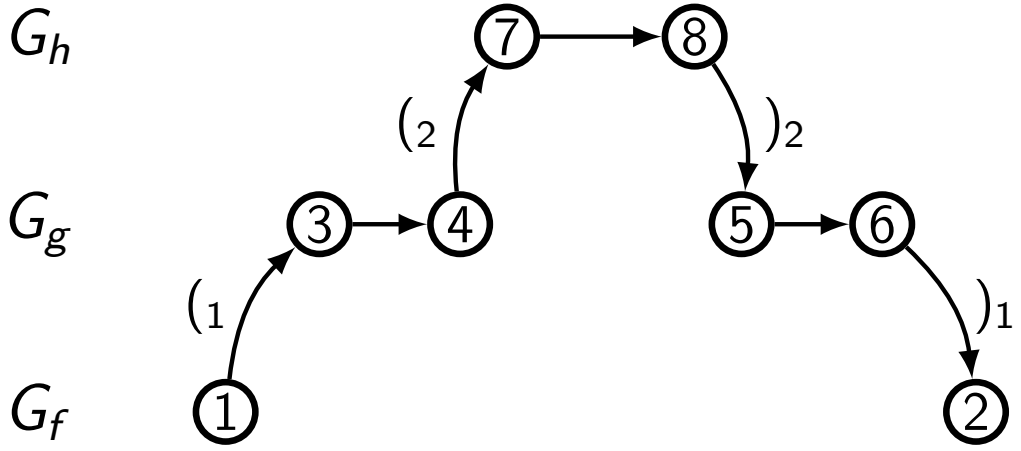
CFGs of typical imperative programs have small treewidth

- For goto-free programs [Thorup '98]
  - Pascal $\leqslant 3$
  - C $\leqslant 6$
- In practice small in imperative programs (e.g. Java $\leqslant 7$)

Theorem (Tree decomposition)

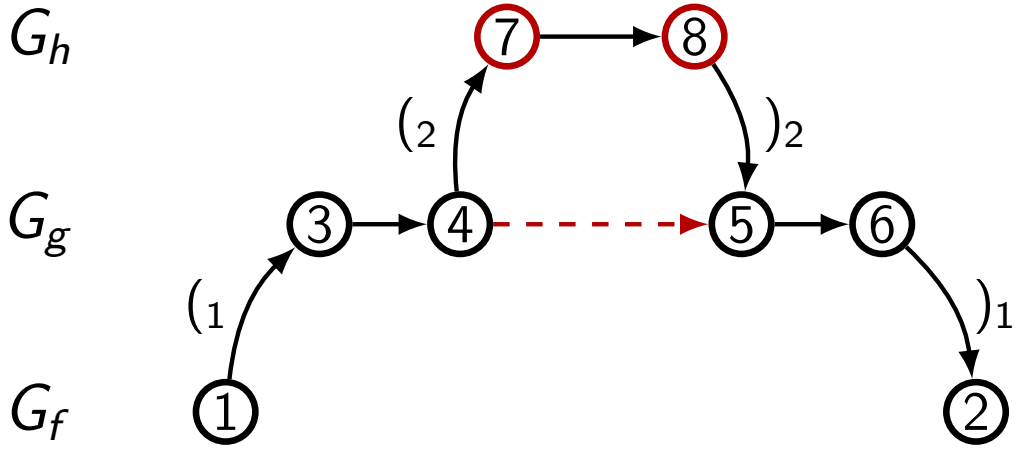
For constant treewidth graphs, $\mathrm{Tree}(G)$ *can be constructed in* $O(n)$ *time*

Algorithmic Principle



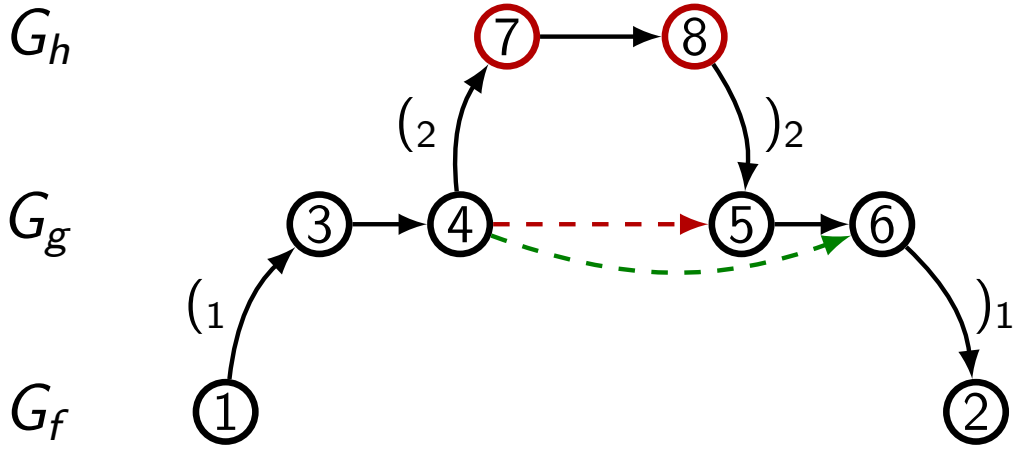
1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

Algorithmic Principle



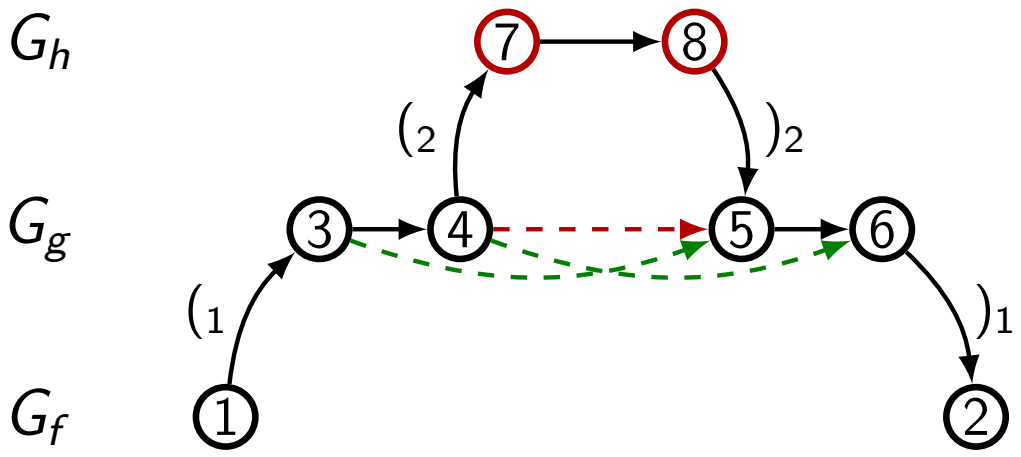
1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

Algorithmic Principle



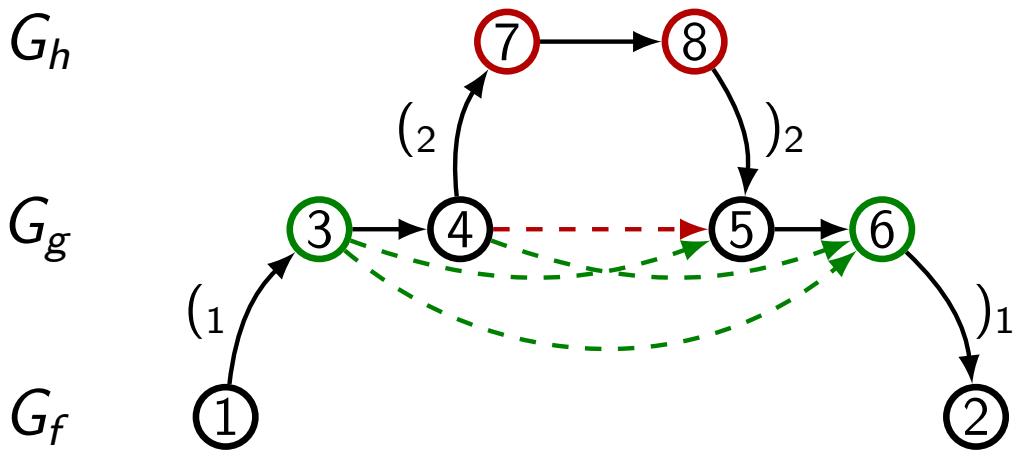
1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

Algorithmic Principle



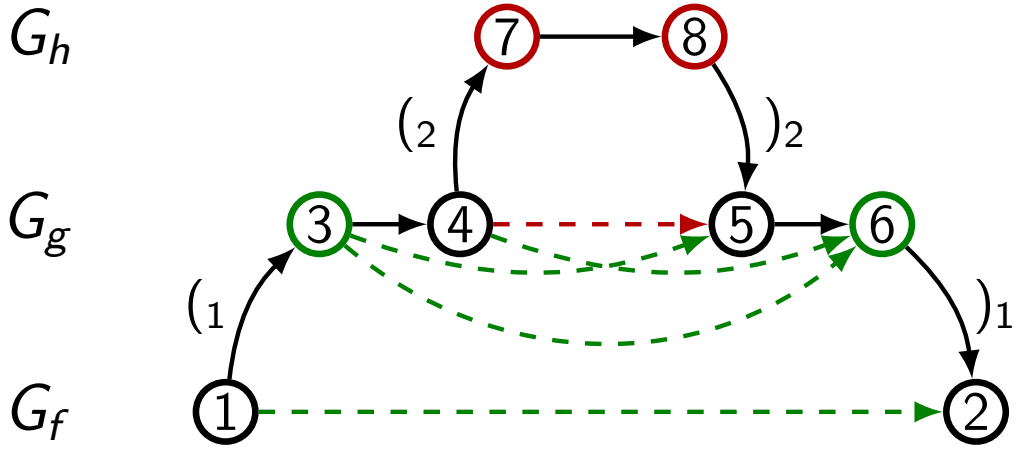
1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

Algorithmic Principle



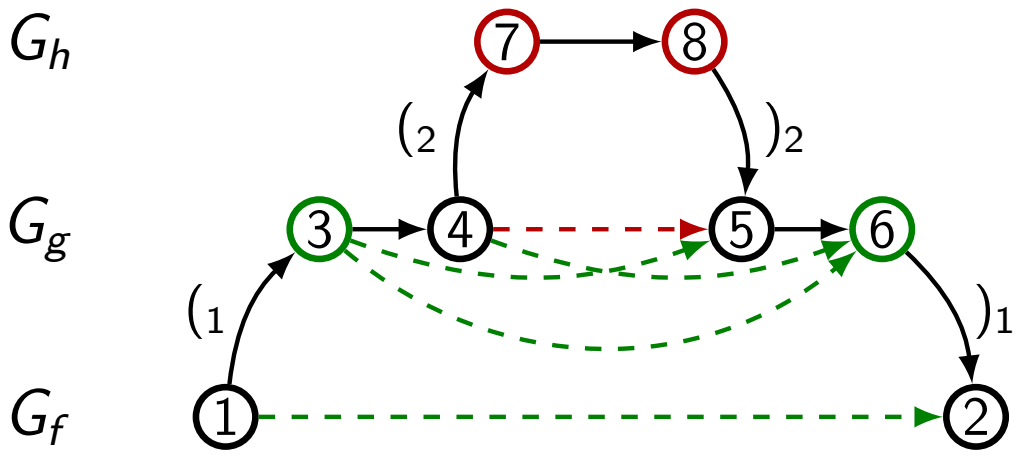
1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

Algorithmic Principle



1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

Algorithmic Principle



1. Reachability ignoring parenthesis edges
2. If Entry $\rightsquigarrow$ Exit
   - Insert summary edge

- Most time spent in (1)
- Incremental reachability
- Low treewidth

Reachability on low-treewidth graphs

Reachability [ESA'16]

Given a graph $G = (V, E)$ of n nodes and bounded treewidth

Preprocessing time	**Space**	**Pair query time**	**Single-source query time**
–	$O(n)$	$O(n)$	$O(n)$
$O\left(\frac{n^2}{\log n}\right)$	$O\left(\frac{n^2}{\log n}\right)$	$O(1)$	$O\left(\frac{n}{\log n}\right)$
$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n)$
$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O\left(\frac{n}{\log n}\right)}$

- Optimal
- Faster than DFS/BFS after a constant number of queries

Reachability [ESA'16]

Given a graph $G = (V, E)$ of n nodes and bounded treewidth

Preprocessing time	**Space**	**Pair query time**	**Single-source query time**
–	$O(n)$	$O(n)$	$O(n)$
$O\left(\frac{n^2}{\log n}\right)$	$O\left(\frac{n^2}{\log n}\right)$	$O(1)$	$O\left(\frac{n}{\log n}\right)$
$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n)$
$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O\left(\frac{n}{\log n}\right)}$

- Optimal
- Faster than DFS/BFS after a constant number of queries

Reachability [ESA'16]

Given a graph $G = (V, E)$ of n nodes and bounded treewidth

Preprocessing time	**Space**	**Pair query time**	**Single-source query time**
–	$O(n)$	$O(n)$	$O(n)$
$O\left(\frac{n^2}{\log n}\right)$	$O\left(\frac{n^2}{\log n}\right)$	$O(1)$	$O\left(\frac{n}{\log n}\right)$
$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n)$
$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O\left(\frac{n}{\log n}\right)}$

- Optimal
- Faster than DFS/BFS after a constant number of queries

Reachability [ESA'16]

Given a graph $G = (V, E)$ of n nodes and bounded treewidth

Preprocessing time	**Space**	**Pair query time**	**Single-source query time**
–	$O(n)$	$O(n)$	$O(n)$
$O\left(\frac{n^2}{\log n}\right)$	$O\left(\frac{n^2}{\log n}\right)$	$O(1)$	$O\left(\frac{n}{\log n}\right)$
$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n)$
$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O\left(\frac{n}{\log n}\right)}$

- Optimal
- Faster than DFS/BFS after a constant number of queries

Reachability [ESA'16]

Given a graph $G = (V, E)$ of n nodes and bounded treewidth

Preprocessing time	**Space**	**Pair query time**	**Single-source query time**
–	$O(n)$	$O(n)$	$O(n)$
$O\left(\frac{n^2}{\log n}\right)$	$O\left(\frac{n^2}{\log n}\right)$	$O(1)$	$O\left(\frac{n}{\log n}\right)$
$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n)$
$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$	$\mathbf{O\left(\frac{n}{\log n}\right)}$

- Optimal
- Faster than DFS/BFS after a constant number of queries

Interprocedural Dataflow Analysis on Recursive Graphs of Low Treewidth

Interprocedural Dataflow Analysis

Input:

- Recursive graph G with n nodes
- Finite graphs G_i of n_i nodes and bounded treewidth
  - Every G_i has a unique entry and exit

Interprocedural Dataflow Analysis

Input:

- Recursive graph G with n nodes
- Finite graphs G_i of n_i nodes and bounded treewidth
  - Every G_i has a unique entry and exit
- D data fact set
  - e.g. "variable x is uninitialized", "variable y is constant"
  - Weighted graph
  - Semiring distance on lattice over D

Interprocedural Dataflow Analysis

Input:

- Recursive graph G with n nodes
- Finite graphs G_i of n_i nodes and bounded treewidth
  - Every G_i has a unique entry and exit
- D data fact set
  - e.g. "variable x is uninitialized", "variable y is constant"
  - Weighted graph
  - Semiring distance on lattice over D

Theorem (POPL'15)

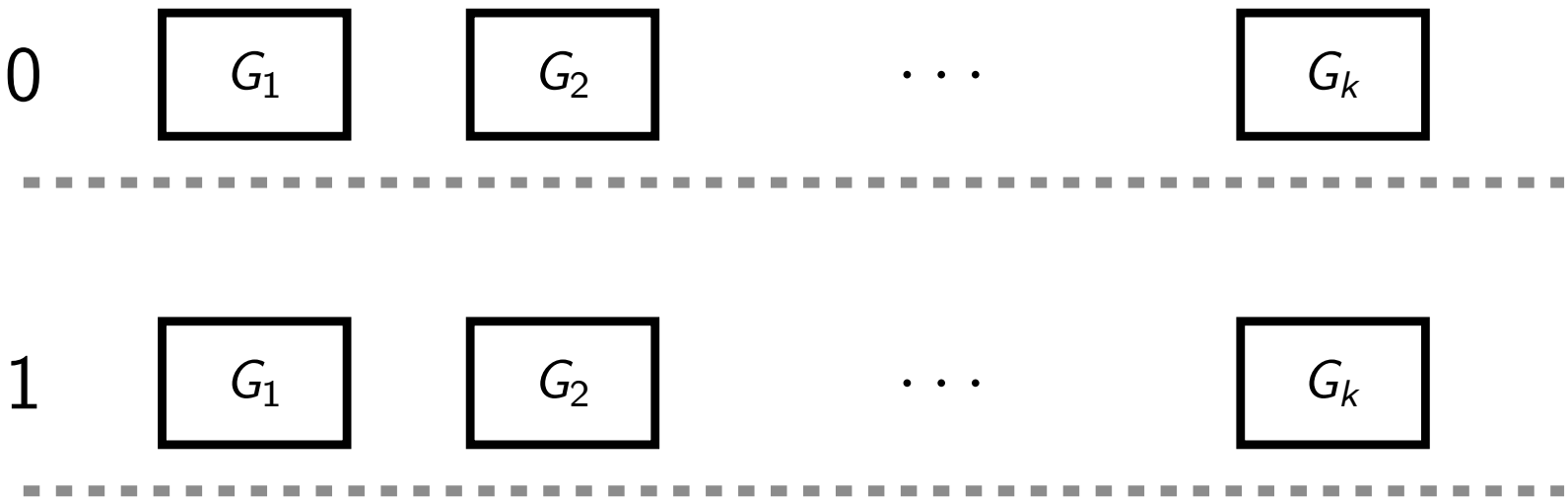
For low-treewidth graphs we construct dynamic reachability oracles:

- **Preprocess** G_i *in* $O(n_i \cdot |D|^3)$ *time*
- **Query** *for the distance* $d(u, v)$ *in* $O(|D|^3 \cdot \log n_i)$ *time*
- **Update** *the weight of an edge* (u, v) *in* $O(|D^3| \cdot \log n_i)$ *time*

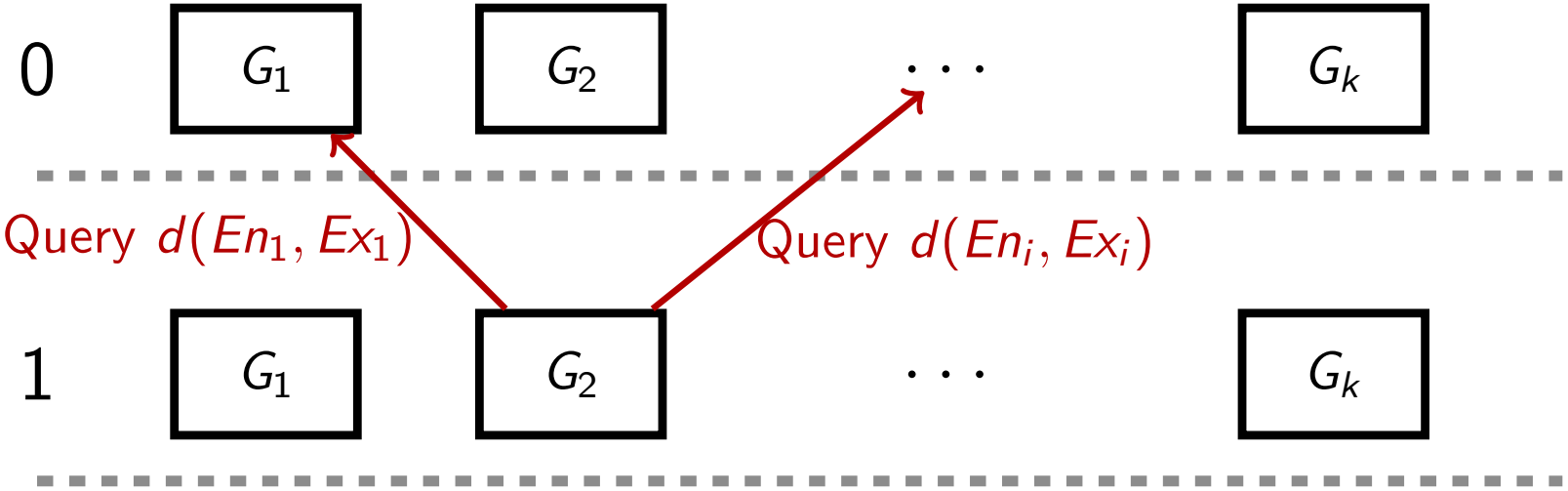
Interprocedural Dataflow Analysis

Interprocedural Dataflow Analysis

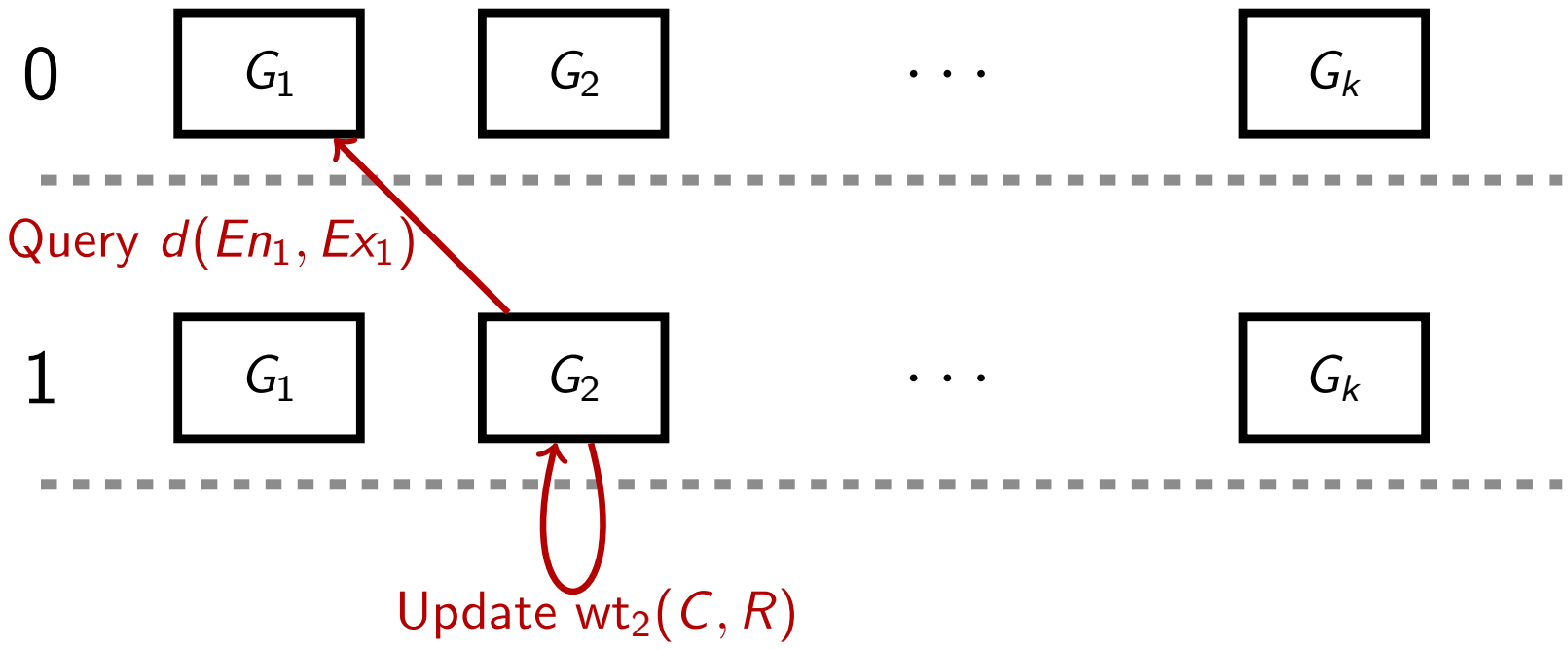
Interprocedural Dataflow Analysis



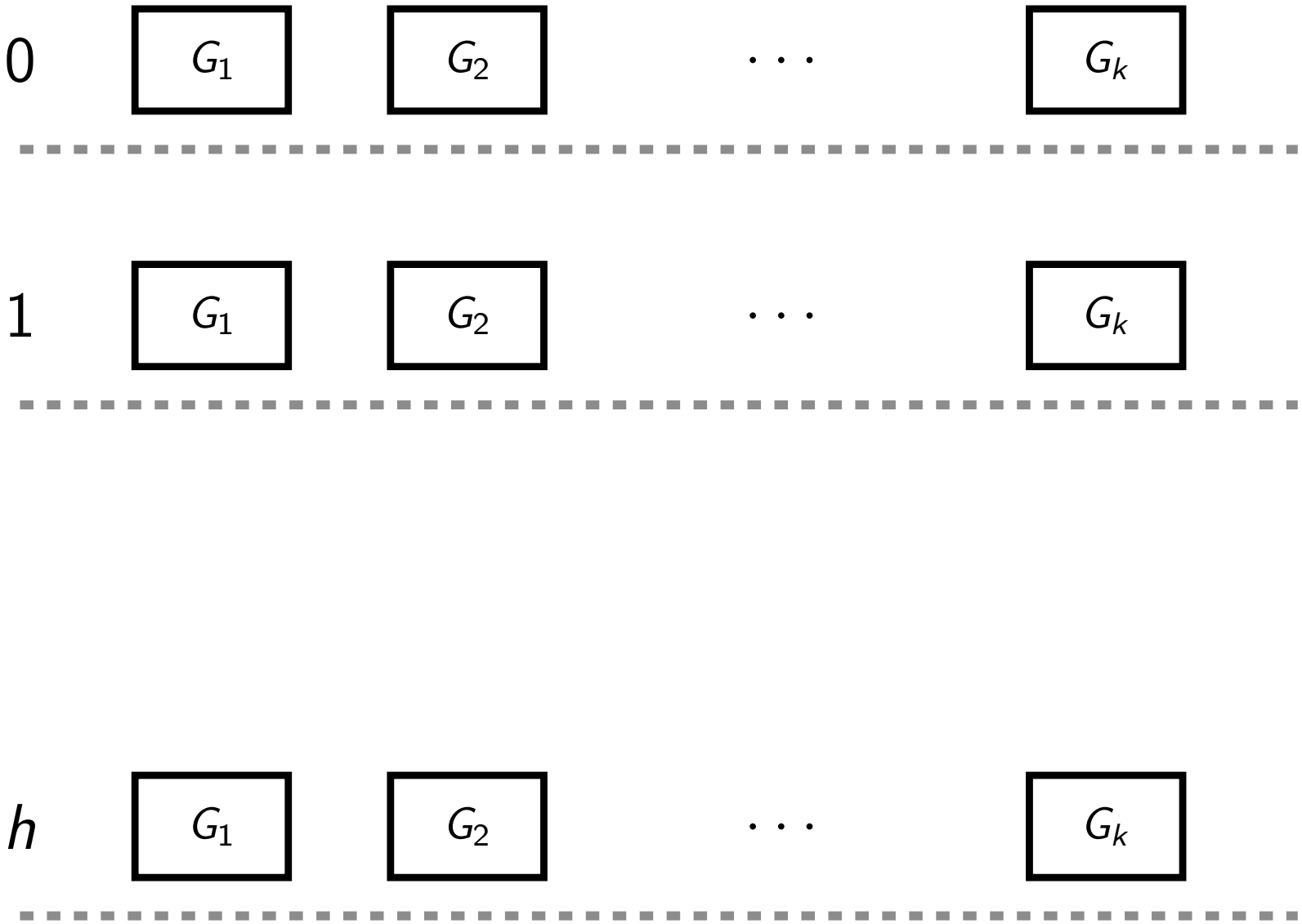
Interprocedural Dataflow Analysis



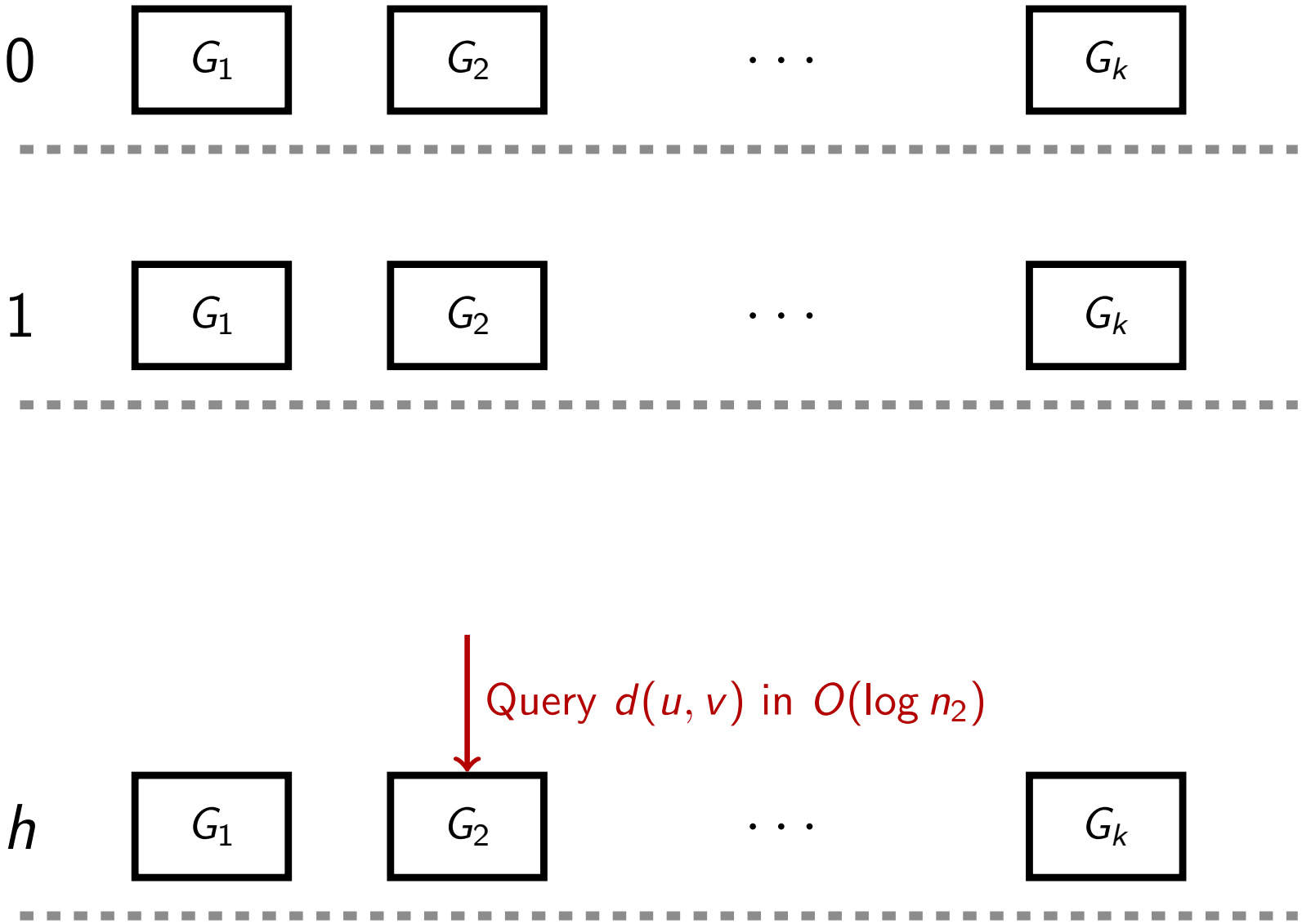
Interprocedural Dataflow Analysis



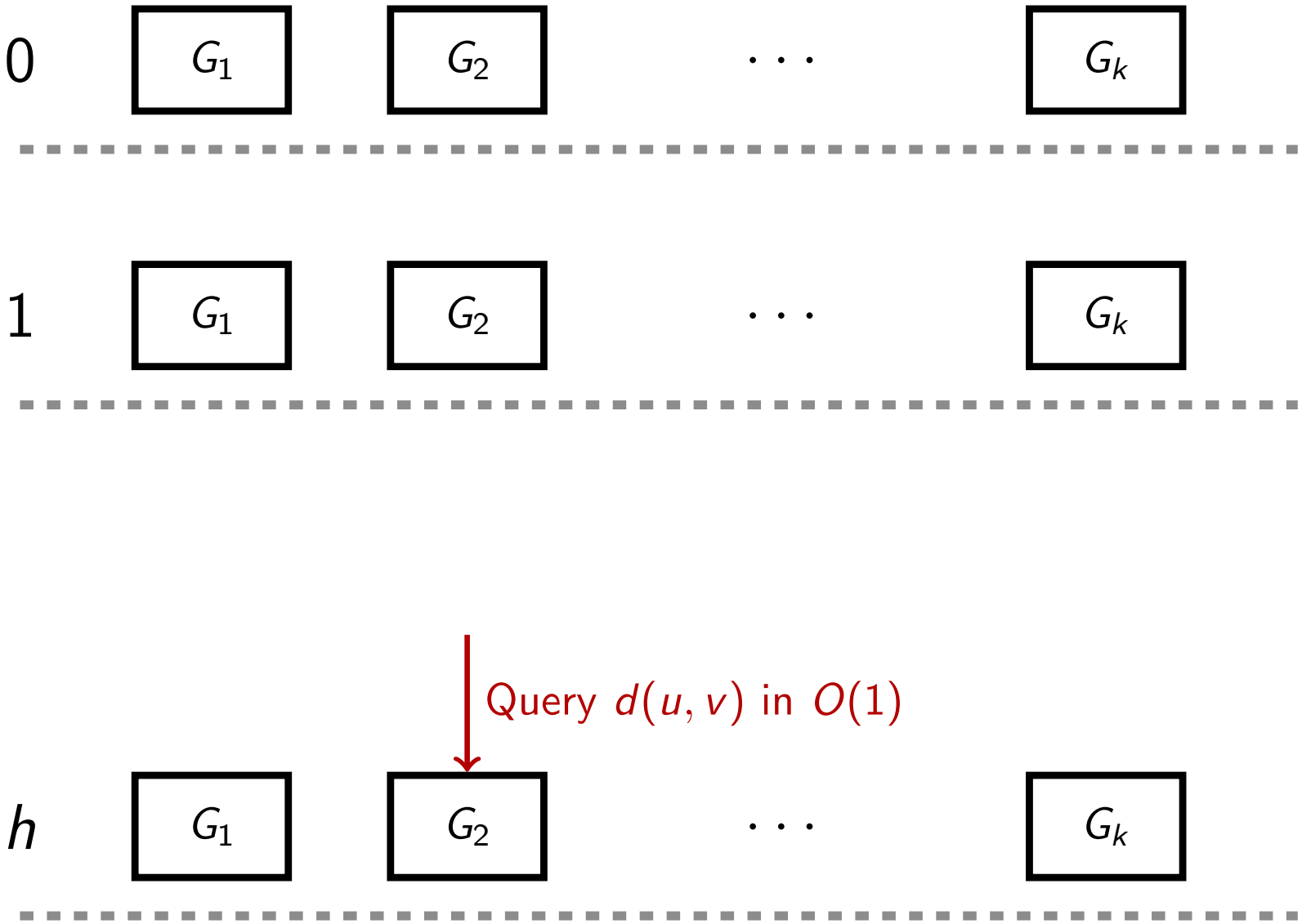
Interprocedural Dataflow Analysis



Interprocedural Dataflow Analysis



Interprocedural Dataflow Analysis



Dataflow Analysis [POPL'15]

Input graph with n nodes, D data fact set, $|D| = \Omega(\log n)$

	Preprocessing time	**Space**	**Query**	
			Single-source	**Pair**
No preprocessing	-	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^3)$
Complete preprocessing	$O(n^2 \cdot \lvert D\rvert^3)$	$O(n^2 \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\lvert D\rvert^2)$
Ours	$\mathbf{O(n \cdot \lvert D\rvert^3)}$	$\mathbf{O(n\lvert D\rvert^2)}$	$\mathbf{O(n \cdot \lvert D\rvert^2)}$	$\mathbf{O(\lvert D\rvert^2)}$

Dataflow Analysis [POPL'15]

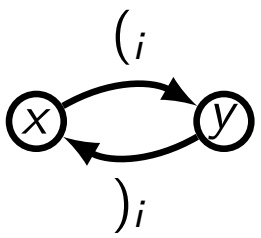
Input graph with n nodes, D data fact set, $|D| = \Omega(\log n)$

	Preprocessing time	**Space**	**Query**	
			Single-source	**Pair**
No preprocessing	-	$O(n \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^3)$	$O(n \cdot \lvert D\rvert^3)$
Complete preprocessing	$O(n^2 \cdot \lvert D\rvert^3)$	$O(n^2 \cdot \lvert D\rvert^2)$	$O(n \cdot \lvert D\rvert^2)$	$O(\lvert D\rvert^2)$
Ours	$\mathbf{O(n \cdot \lvert D\rvert^3)}$	$\mathbf{O(n\lvert D\rvert^2)}$	$\mathbf{O(n \cdot \lvert D\rvert^2)}$	$\mathbf{O(\lvert D\rvert^2)}$

Dyck Reachability on Bidirected Graphs

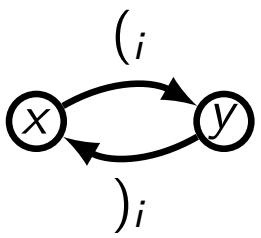
Bidirected graphs

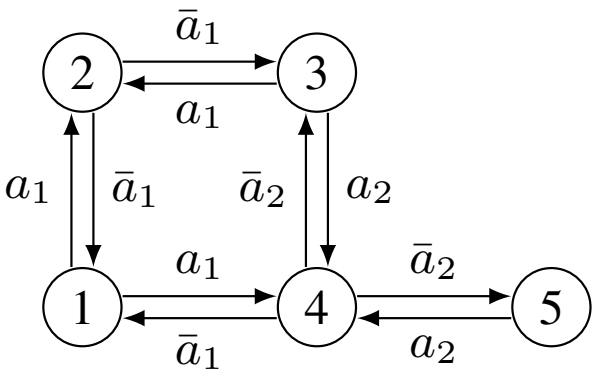
$$\forall u, v \in V : \ \lambda(u, v) = (_i \quad \Longleftrightarrow \quad \lambda(v, u) =)_i$$



Bidirected graphs

$$\forall u, v \in V : \ \lambda(u, v) = (_i \quad \Longleftrightarrow \quad \lambda(v, u) =)_i$$





- Demand-driven field-sensitive alias analysis
- [YXR, ISSTA '11] [ZLYS, PLDI '13]

Reachability is Equivalence

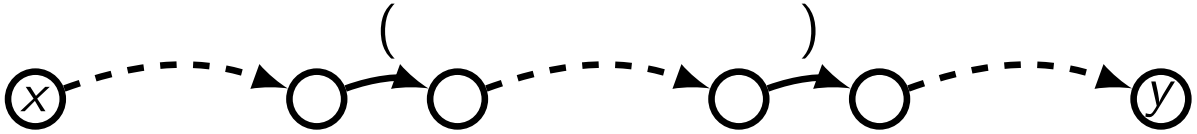
Theorem (ZLYS '13)

Dyck reachability on bidirected graphs is an equivalence relation.

Reachability is Equivalence

Theorem (ZLYS '13)

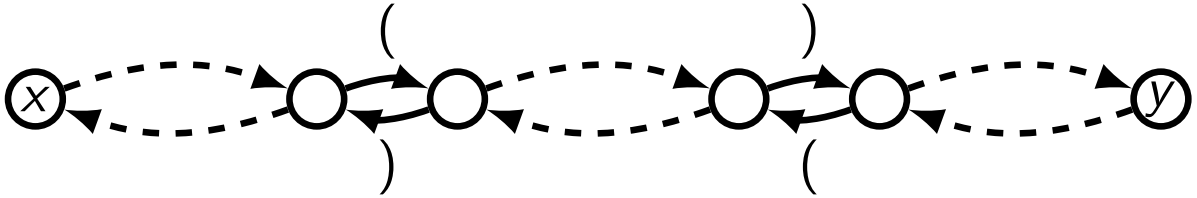
Dyck reachability on bidirected graphs is an equivalence relation.



Reachability is Equivalence

Theorem (ZLYS '13)

Dyck reachability on bidirected graphs is an equivalence relation.



Dyck Reachability on Bidirected Graphs [POPL'18]

Graphs of n nodes, $\alpha(n)$ the inverse Ackermann

Source	**Worst-case Time**	**Average-case Time**
[ZLYS '13]	$O\left(n^2\right)$	$O\left(n \cdot \log n\right)$
Our Result	$\mathbf{O(n \cdot \alpha(n))}$	$\mathbf{O(n)}$

Dyck Reachability on Bidirected Graphs [POPL'18]

Graphs of n nodes, $\alpha(n)$ the inverse Ackermann

Source	**Worst-case Time**	**Average-case Time**
[ZLYS '13]	$O\left(n^2\right)$	$O\left(n \cdot \log n\right)$
Our Result	$\mathbf{O(n \cdot \alpha(n))}$	$\mathbf{O(n)}$
	optimal!	optimal!

Hardness?

Complexity of Dyck Reachability [POPL'18]

- $O(n^3)$
- 2NPDA-hard
  - Conditional cubic lower bound

Complexity of Dyck Reachability [POPL'18]

- $O(n^3)$
- 2NPDA-hard
  - Conditional cubic lower bound

Theorem

Dyck reachability is Boolean Matrix Multiplication - hard.

Complexity of Dyck Reachability [POPL'18]

- $O(n^3)$
- 2NPDA-hard
  - Conditional cubic lower bound

Theorem

Dyck reachability is Boolean Matrix Multiplication - hard.

Complexity of Dyck Reachability [POPL'18]

- $O(n^3)$
- 2NPDA-hard
  - Conditional cubic lower bound

Theorem

Dyck reachability on low-treewidth graphs is Boolean Matrix Multiplication - hard.

Can we do $O(n^3/\log^2 n)$?

Complexity on Bidirected Graphs [POPL'18]

Theorem

Dyck reachability on constant-treewidth, bidirected graphs requires $\Omega(n \cdot \alpha(n))$ *time.*

Complexity on Bidirected Graphs [POPL'18]

Theorem

Dyck reachability on constant-treewidth, bidirected graphs requires $\Omega(n \cdot \alpha(n))$ time.

Compare with

Theorem (ZLYZ '13)

Dyck reachability on bidirected trees solvable in $O(n)$ time.

Typically complexity on low-treewidth graphs = complexity on trees

Implementation & Experiments

Implementation & Experiments

Alias Analysis

- Implementation in C++
- Compared with [ZLYS, PLDI '13]
- DaCapo-2006 benchmarks
- SPGs from [YXR, ISSTA '11]

Implementation & Experiments

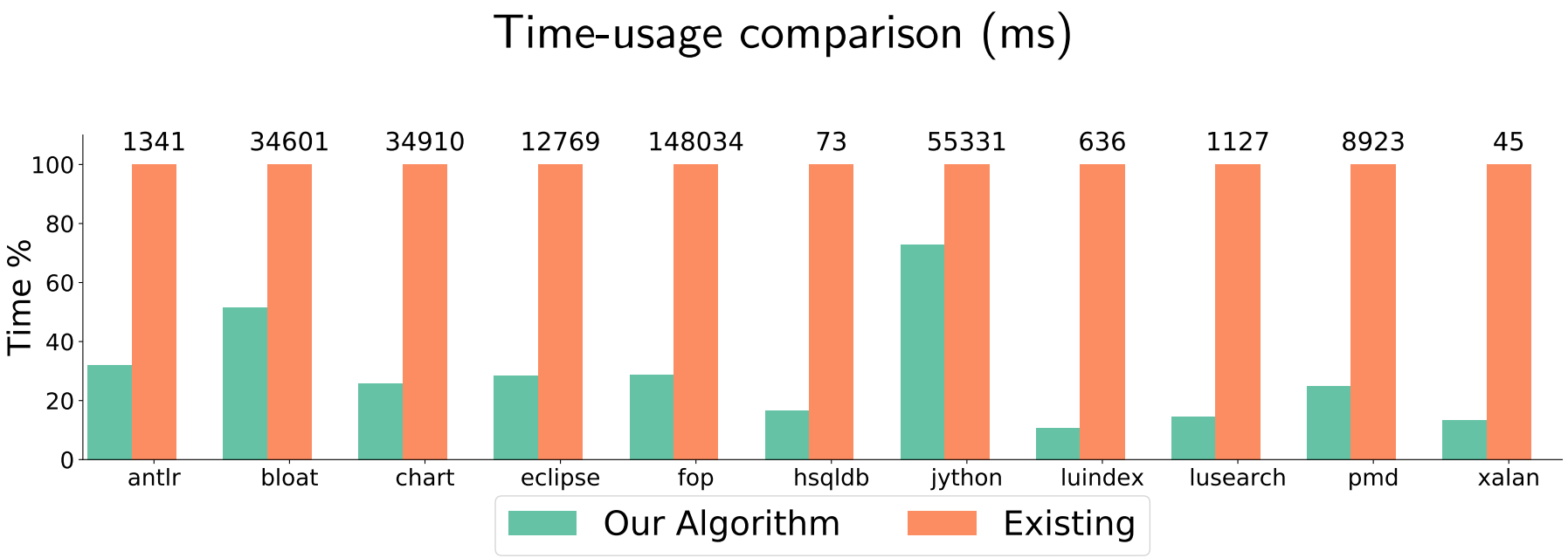
Alias Analysis

- Implementation in C++
- Compared with [ZLYS, PLDI '13]
- DaCapo-2006 benchmarks
- SPGs from [YXR, ISSTA '11]

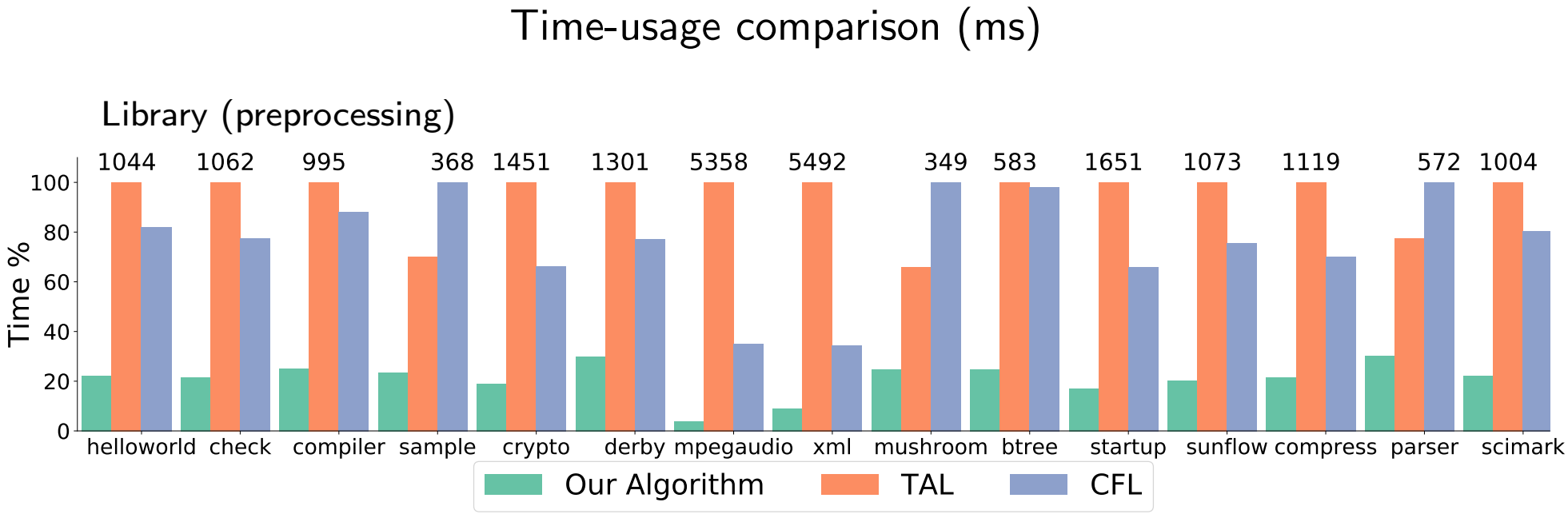
Data-dependence Analysis

- Implementation in Java
- Compared with TAL reachability [TWZXZM, POPL '15]
- Benchmarks:
  - SPECjvm2008
  - 4 randomly chosen GitHub projects

Alias Analysis

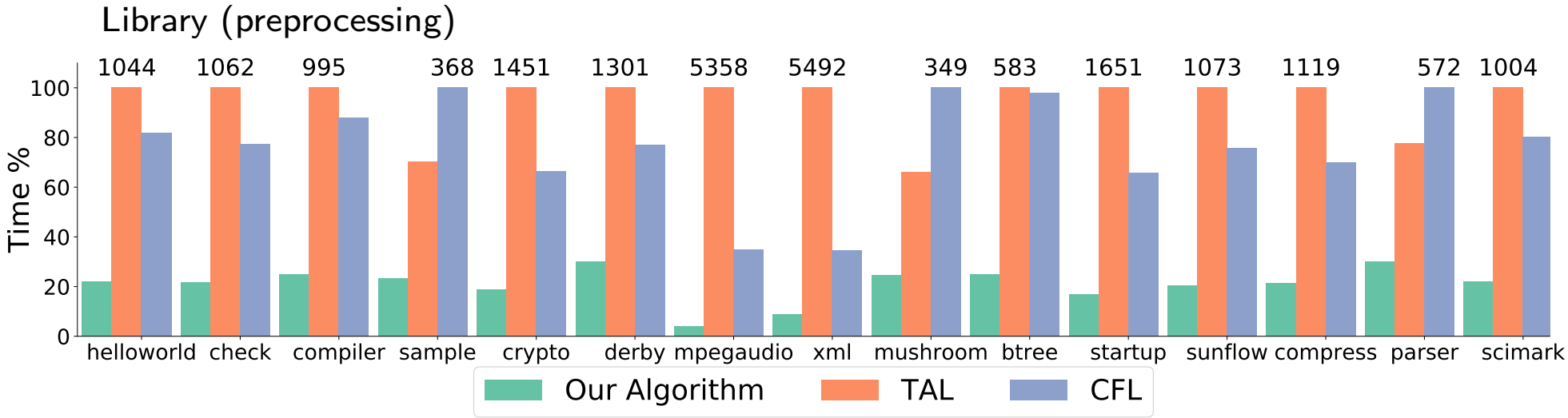


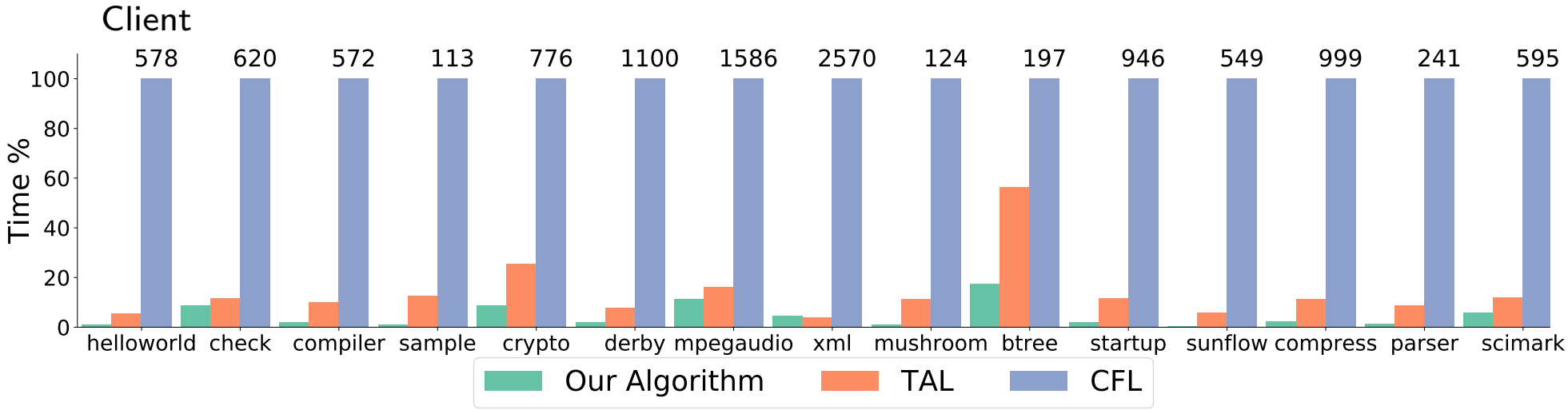
Data-dependence Analysis



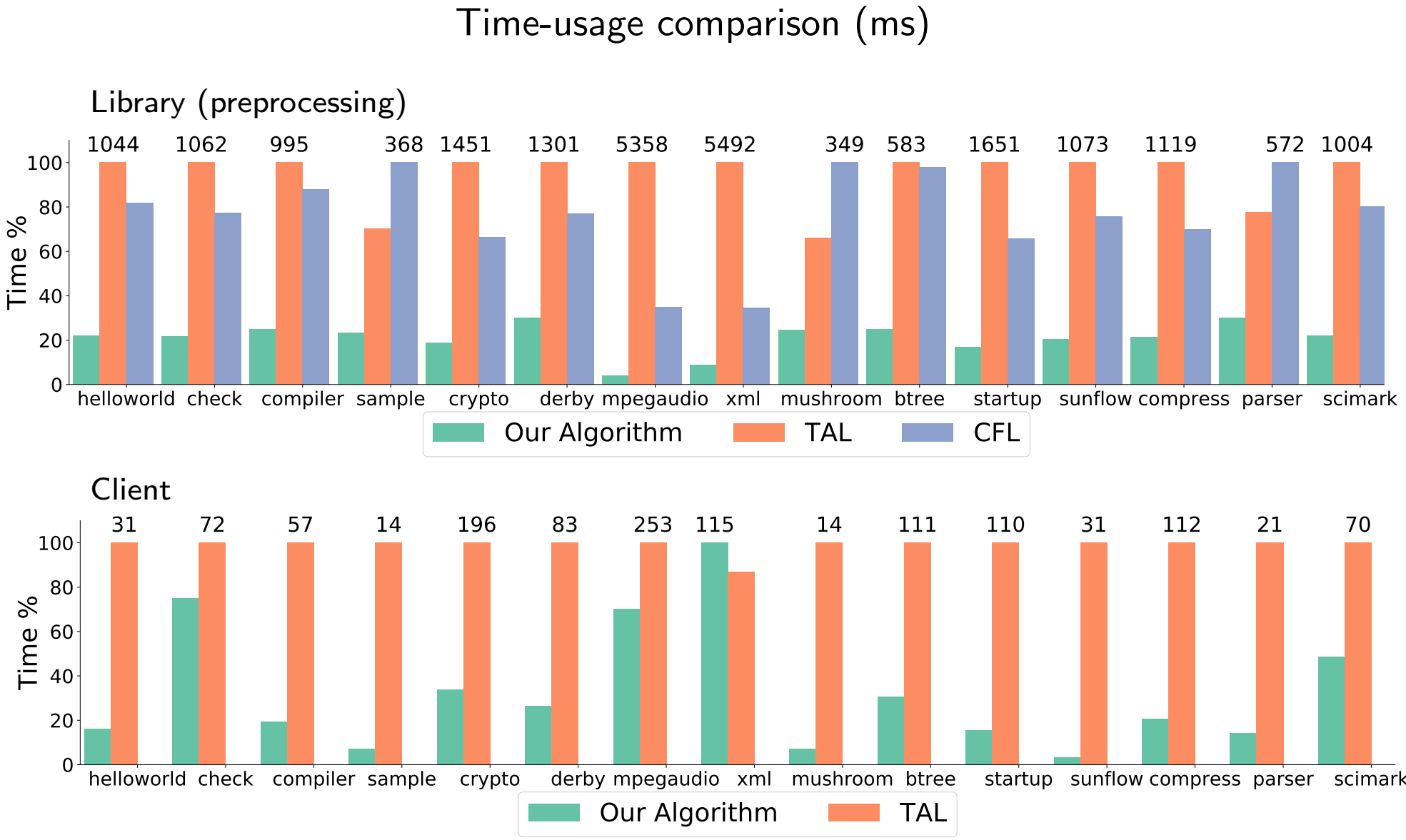
Data-dependence Analysis

Time-usage comparison (ms)

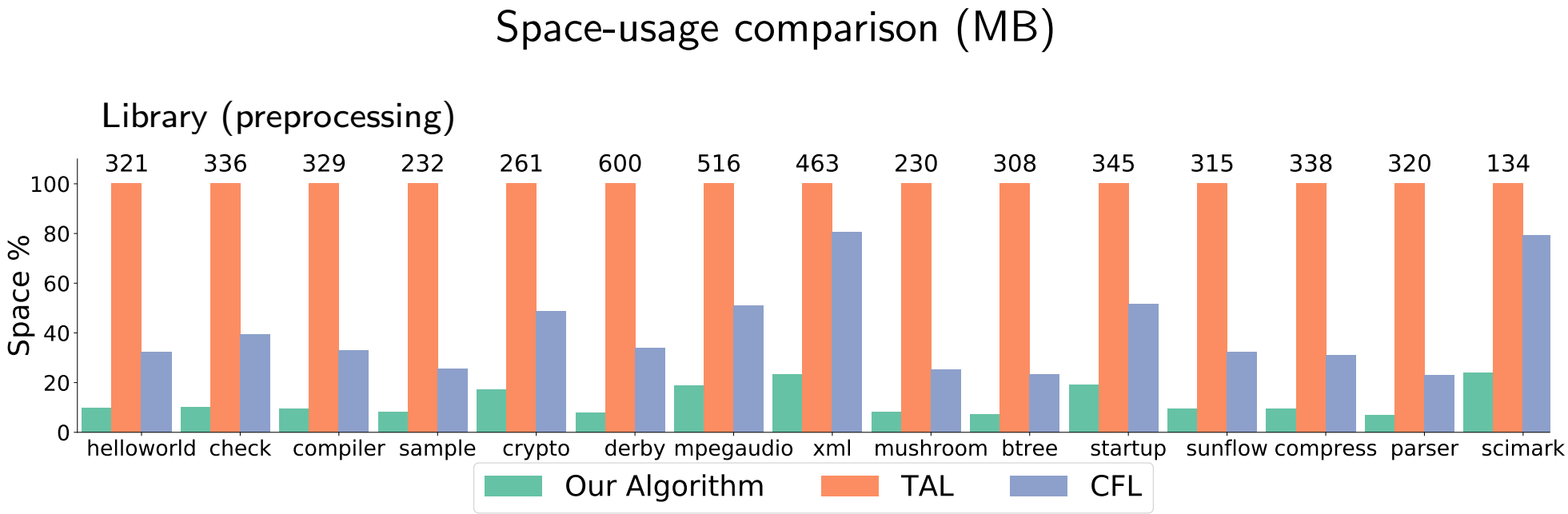




Data-dependence Analysis

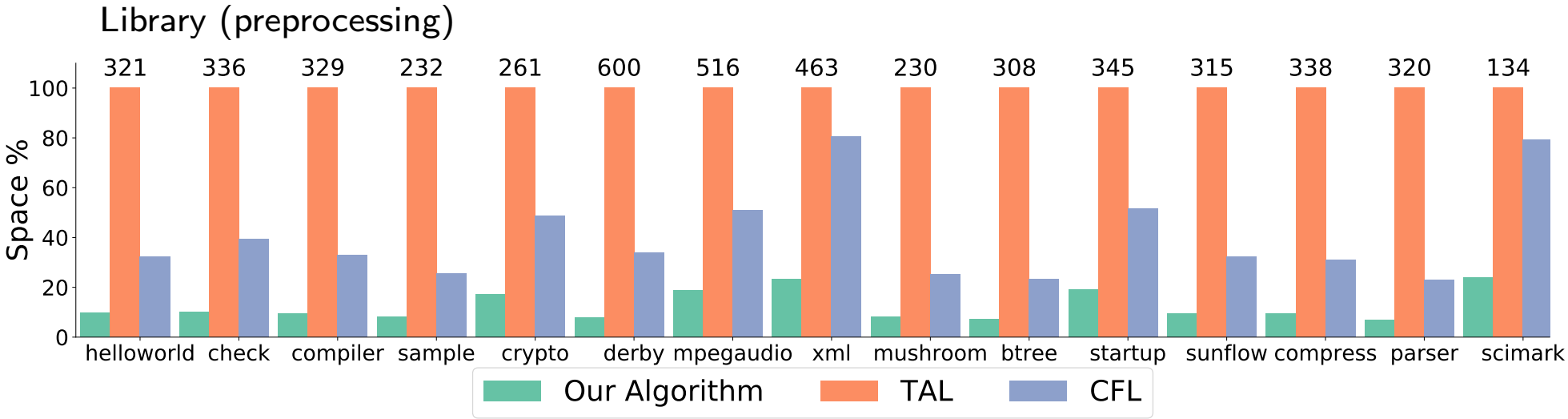


Data-dependence Analysis



Data-dependence Analysis

	Nodes		Our Algo		TAL		CFL	
Benchmark	Lib.	Cl.	Lib.	Cl.	Lib.	Cl.	Li.	Cl.
serial	69999	468	130	130	>12000	>12000	3964	4314



Thank you!

Questions?