Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis

Giovanna K. Conrado, Amir K. Goharshady, Kerim Kochekov, Yun Chen Tsai, and Ahmed K. Zaher

October 26th, 2023

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

Agenda

1. Context and contribution
2. Algorithms
3. Experiments and conclusion

Agenda

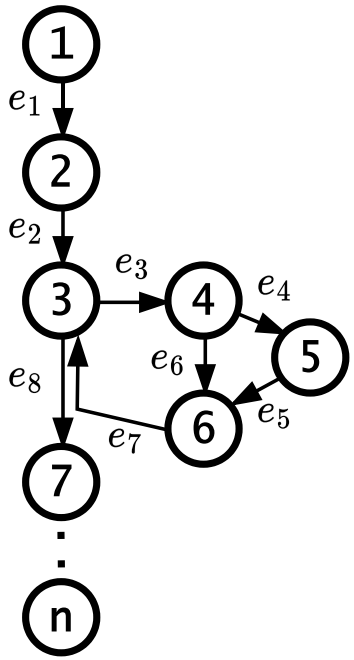
1. Context and contribution
2. Algorithms
3. Experiments and conclusion

Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```

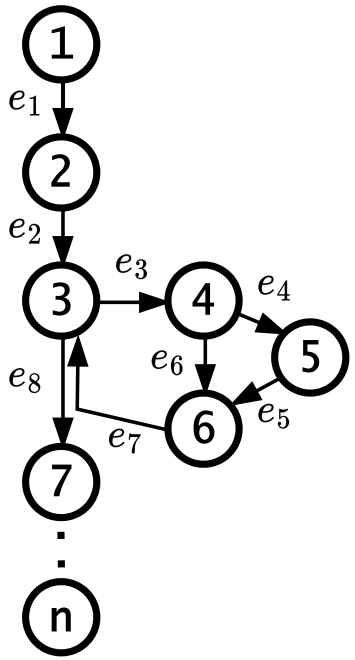
Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

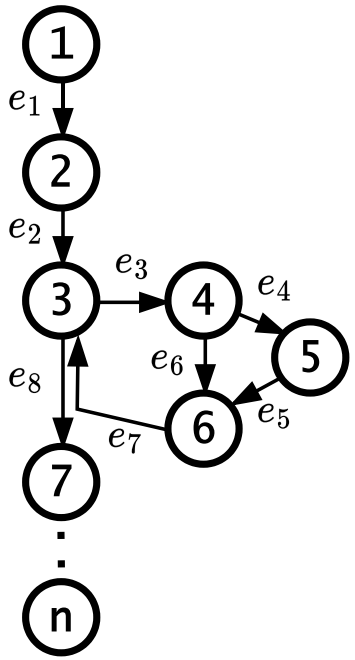
$$[\![\cdot]\!] : E \to A$$

$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

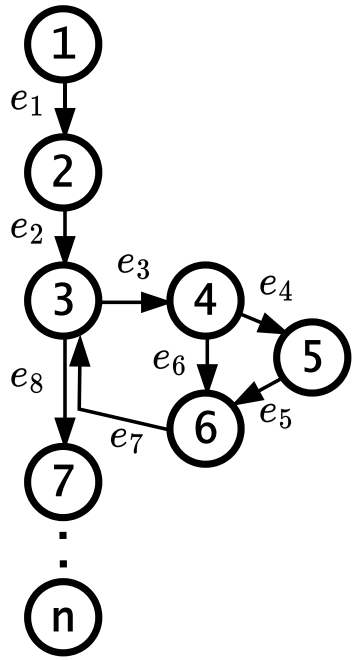
$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

Algebraic approach to find $X_{i,j}$:

Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

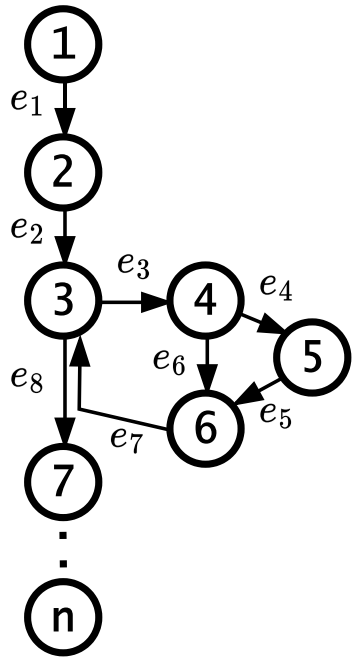
Algebraic approach to find $X_{i,j}$:

1. Find a regular expression $\rho_{i,j}$ over E recognizing exactly $Paths(i,j)$:

$$\rho_{2,7} = e_2 \cdot \Big(e_3 \cdot \big(e_6 + e_4 \cdot e_5 \big) \cdot e_7 \Big)^* \cdot e_8.$$

Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

Algebraic approach to find $X_{i,j}$:

1. Find a regular expression $\rho_{i,j}$ over E recognizing exactly $Paths(i,j)$:

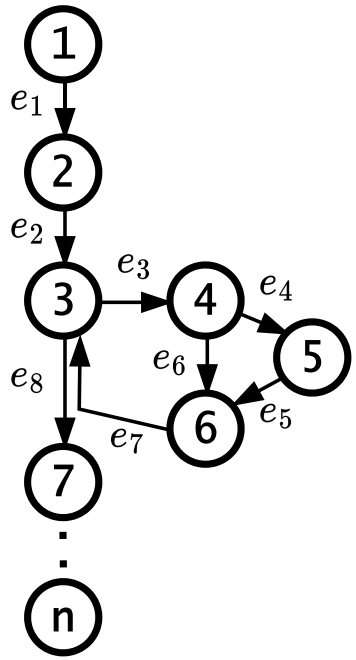
$$\rho_{2,7} = e_2 \cdot \Big(e_3 \cdot \big(e_6 + e_4 \cdot e_5 \big) \cdot e_7 \Big)^* \cdot e_8.$$

2. To obtain $X_{i,j}$, interpret $\rho_{i,j}$ using the algebra:

$$X_{2,7} = [\![\rho_{2,7}]\!] = [\![e_2]\!] \otimes \Big([\![e_3]\!] \otimes \big([\![e_6]\!] \oplus [\![e_4]\!] \otimes [\![e_5]\!] \big) \otimes [\![e_7]\!] \Big)^{\circledast} \otimes [\![e_8]\!].$$

Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

Algebraic approach to find $X_{i,j}$:

1. Find a regular expression $\rho_{i,j}$ over E recognizing exactly $Paths(i,j)$:

$$\rho_{2,7} = e_2 \cdot \Big(e_3 \cdot \big(e_6 + e_4 \cdot e_5\big) \cdot e_7\Big)^* \cdot e_8.$$

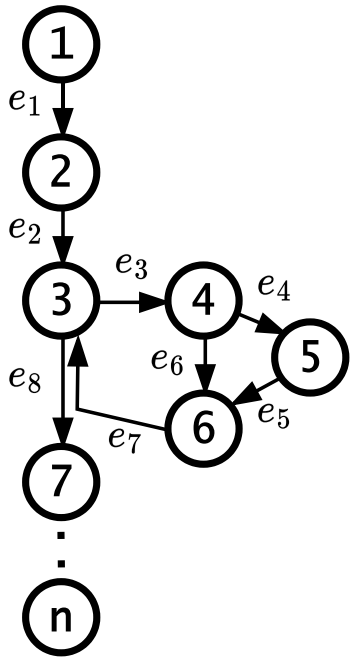
2. To obtain $X_{i,j}$, interpret $\rho_{i,j}$ using the algebra:

$$X_{2,7} = [\![\rho_{2,7}]\!] = [\![e_2]\!] \otimes \Big([\![e_3]\!] \otimes \big([\![e_6]\!] \oplus [\![e_4]\!] \otimes [\![e_5]\!]\big) \otimes [\![e_7]\!]\Big)^{\circledast} \otimes [\![e_8]\!].$$

Applications: numerical invariant generation, predicate abstraction.

On-demand Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

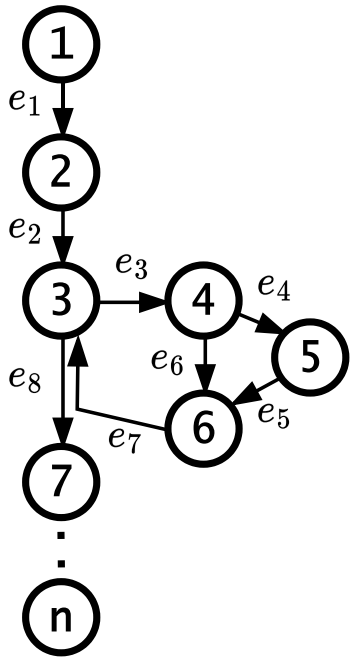
$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

On-demand: a large stream of online queries (i, j) asking for $X_{i,j}$.

On-demand Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

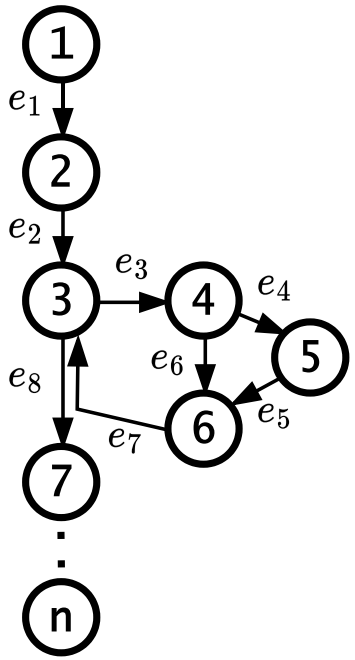
$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

On-demand: a large stream of online queries (i, j) asking for $X_{i,j}$.

Goal: answer these queries fast.

On-demand Algebraic program analysis

```
1 int main () {
2     int x = 50, y = 0;
3     while (x-- >= 0) {
4         if (x & 1)
5             y += 3;
6         y = y * 2;
7     }
..    ..
..    ..
n }
```



Universe of summaries A

$$[\![\cdot]\!] : E \to A$$

$$(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$$

$$\{X_{i,j}\}_{i,j \in \{1 \ldots n\}}$$

On-demand: a large stream of online queries (i, j) asking for $X_{i,j}$.

Goal: answer these queries fast.

Why?

Formulation and contribution

On-demand algebraic program analysis

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)

- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)

– a program P,

– an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,

– a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)

– a program P,

– an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,

– a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)
- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)
- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Our contribution: algorithms to solve this efficiently.

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)
- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Our contribution: algorithms to solve this efficiently.
- Preprocessing: precompute queries of special forms.
- Query: express input queries as combination of precomputed queries.
- Light preprocessing and fast query time.

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)
- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Our contribution: algorithms to solve this efficiently.
- Preprocessing: precompute queries of special forms.
- Query: express input queries as combination of precomputed queries.
- Light preprocessing and fast query time.

Intra-procedural case: we **exploit sparsity of control-flow graphs.**

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)
- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Our contribution: algorithms to solve this efficiently.
- Preprocessing: precompute queries of special forms.
- Query: express input queries as combination of precomputed queries.
- Light preprocessing and fast query time.

Intra-procedural case: we **exploit sparsity of control-flow graphs.**

Inter-procedural case:
- We assume function summaries are computed and are given in input.
- We additionally **exploit sparsity of call graphs.**

Formulation and contribution

On-demand algebraic program analysis

Offline input: (can be preprocessed)
- a program P,
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_P(i, j)$.

Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis

Agenda

1. Context and contribution
2. Algorithms
3. Experiments and conclusion

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_G(i, j)$.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

- $n = |V| \approx |E|$, cost per algebra operation is k:
- *Tarjan's algorithm:*
  - Works on reducible flow graphs ($\approx$ CFGs).
  - Answers all queries $(i, -)$ for a fixed i in $O(n\alpha(n) \cdot k)$.
  - Doesn't suit our on-demand setting:
    - for n queries with different i's, naive repetition $\implies \Omega(n^2)$ time.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

- *The paper presents two algorithms:*

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
– *a CFG $G = (V, E)$ of a single function,*
– an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
– a semantic function $[\![\cdot]\!] : E \to A$.
Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

– *The paper presents two algorithms:*
Algorithm #1: exploits nesting depth. See the paper.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
– *a CFG $G = (V, E)$ of a single function,*
– an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
– a semantic function $[\![\cdot]\!] : E \to A$.
Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

– *The paper presents two algorithms:*
Algorithm #1: exploits nesting depth. See the paper.

Algorithm #2: exploits treewidth.

Treewidth of CFGs

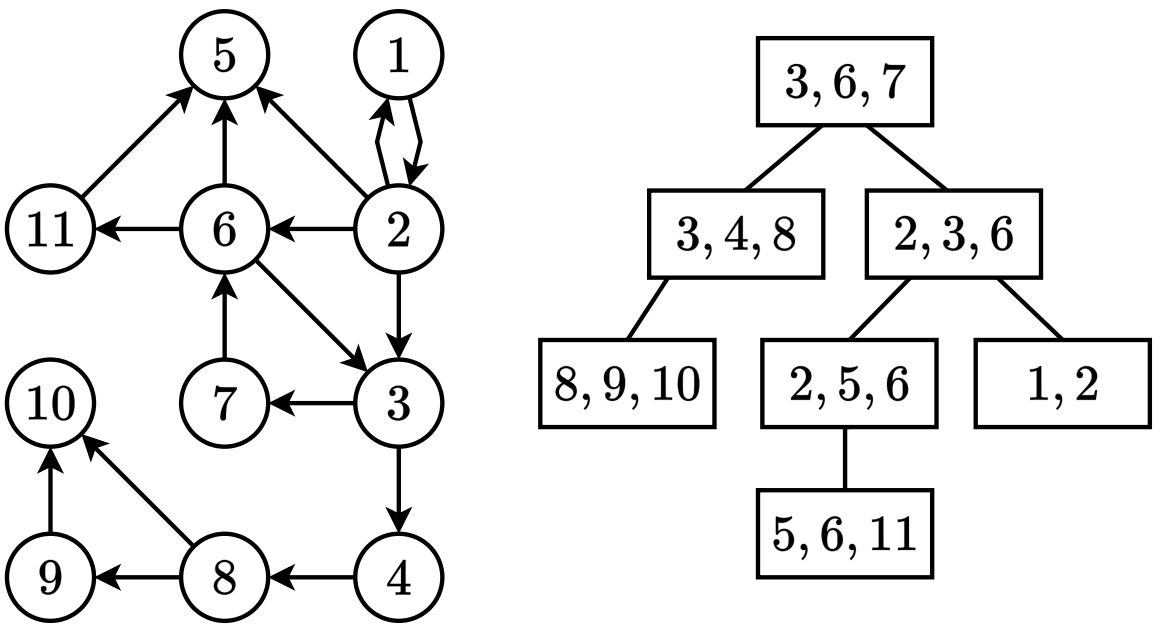
Treewidth:

- (Informally) a parameter that measures "tree-likeness" of a graph.

Treewidth of CFGs

Treewidth:

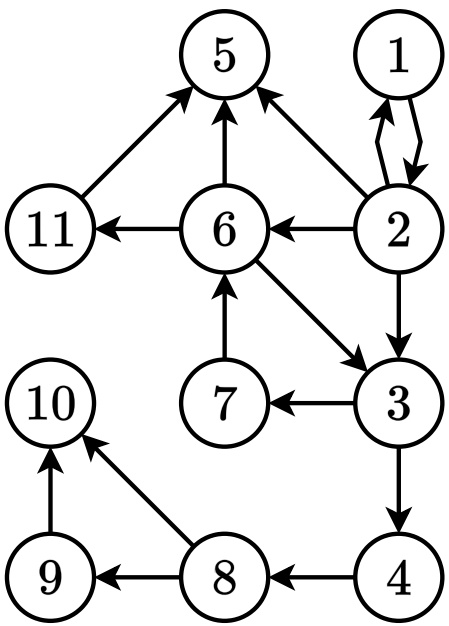
- (Informally) a parameter that measures "tree-likeness" of a graph.
- Small-treewidth graphs admit a *tree decomposition* with small *bags.*
- Such decomposition enable us to successively break a graph into smaller *disconnected* graphs separated by *small cuts.*

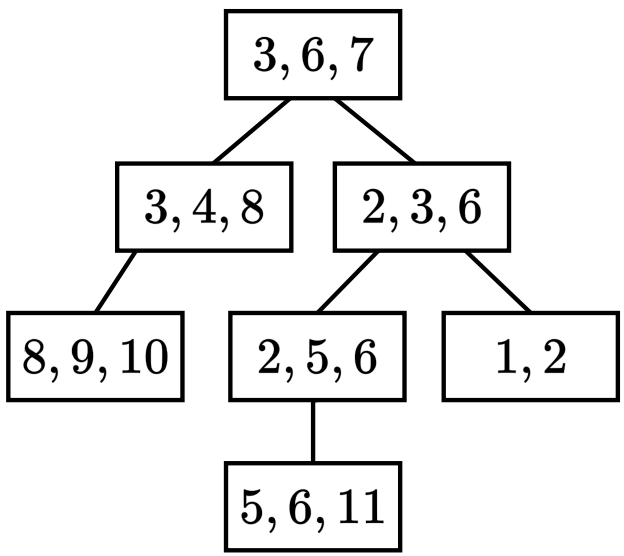


Treewidth of CFGs

Treewidth:

- (Informally) a parameter that measures "tree-likeness" of a graph.
- Small-treewidth graphs admit a *tree decomposition* with small *bags.*
- Such decomposition enable us to successively break a graph into smaller *disconnected* graphs separated by *small cuts.*
- CFGs have constant treewidth (Thorup '98).





Intra-procedural algorithm #2

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG* $G = (V, E)$ *of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$,
- *a tree decomposition* T *of* G *with small bags.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_G(i, j)$.

Intra-procedural algorithm #2

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
- *a CFG* $G = (V, E)$ *of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$,
- *a tree decomposition* T *of* G *with small bags.*

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_G(i, j)$.

Naive: n^2 possible queries, precompute all of them.

$\implies$ takes too much time and space.

Intra-procedural algorithm #2

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
- *a CFG* $G = (V, E)$ *of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$,
- *a tree decomposition* T *of* G *with small bags.*

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_G(i, j)$.

Naive: n^2 possible queries, precompute all of them.

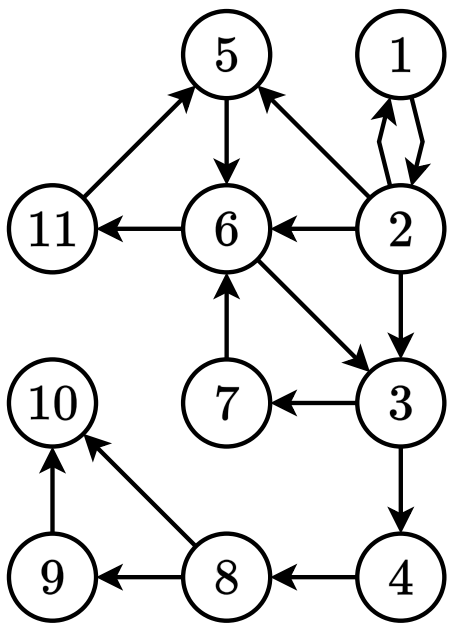
$\Longrightarrow$ takes too much time and space.

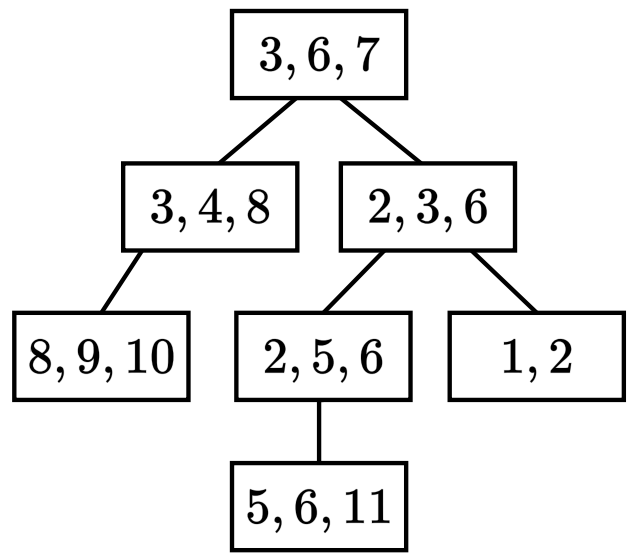
Better: precompute only *"special queries"* having a certain form s.t.,

- **Expressiveness:** a general query can be expressed with special queries.
- **Space:** the number of special queries should be $\ll n^2$.
- **Time:** total runtime should be small.

Intra-procedural algorithm #2

We look at the tree decomposition T of the CFG G.



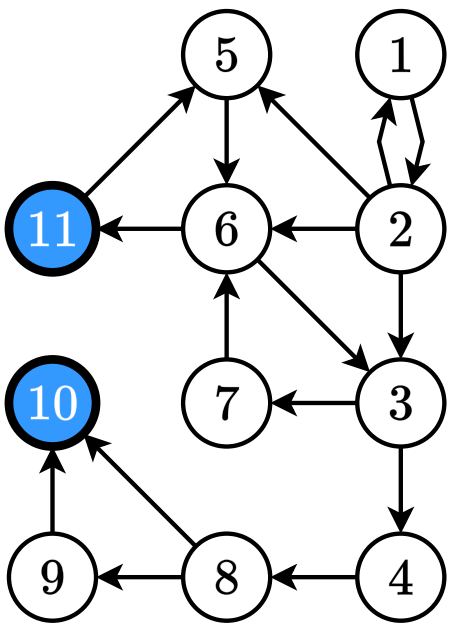


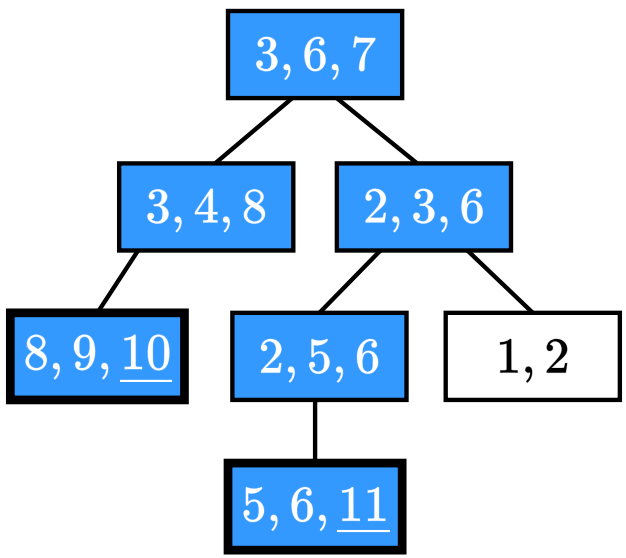
Intra-procedural algorithm #2

We look at the tree decomposition T of the CFG G.

Lemma

For any u, v in G, there there are bags b_u, b_v *in* T where every bag b on P_{b_u,b_v} separates u from v in G.



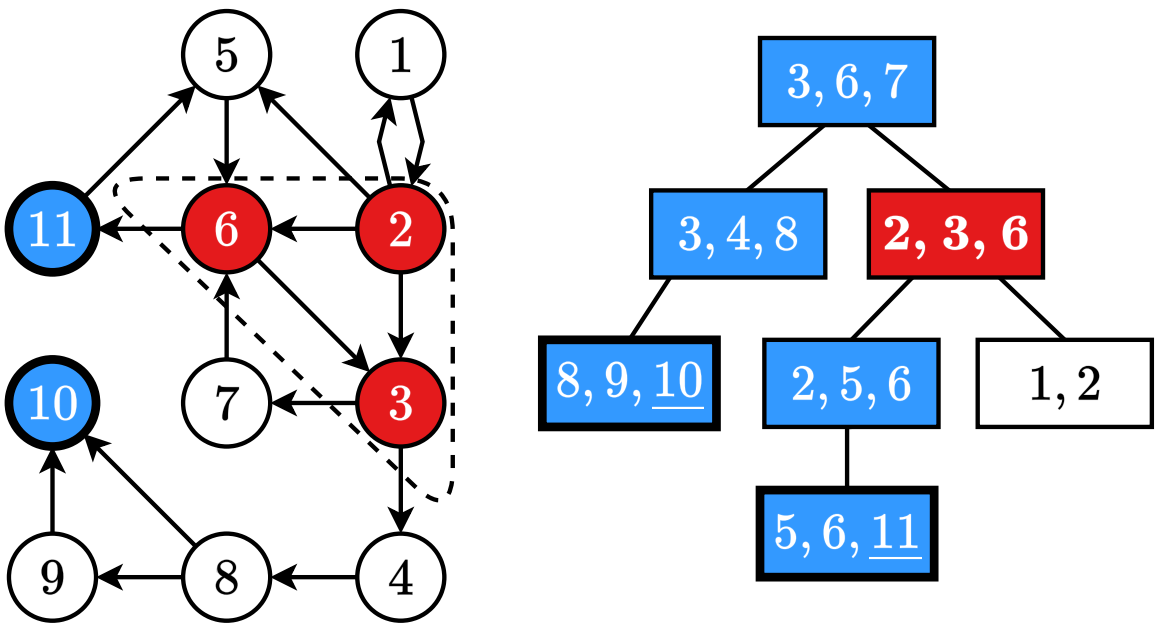


Intra-procedural algorithm #2

We look at the tree decomposition T of the CFG G.

Lemma

For any u, v in G, there there are bags b_u, b_v *in* T where every bag b on P_{b_u,b_v} separates u from v in G.

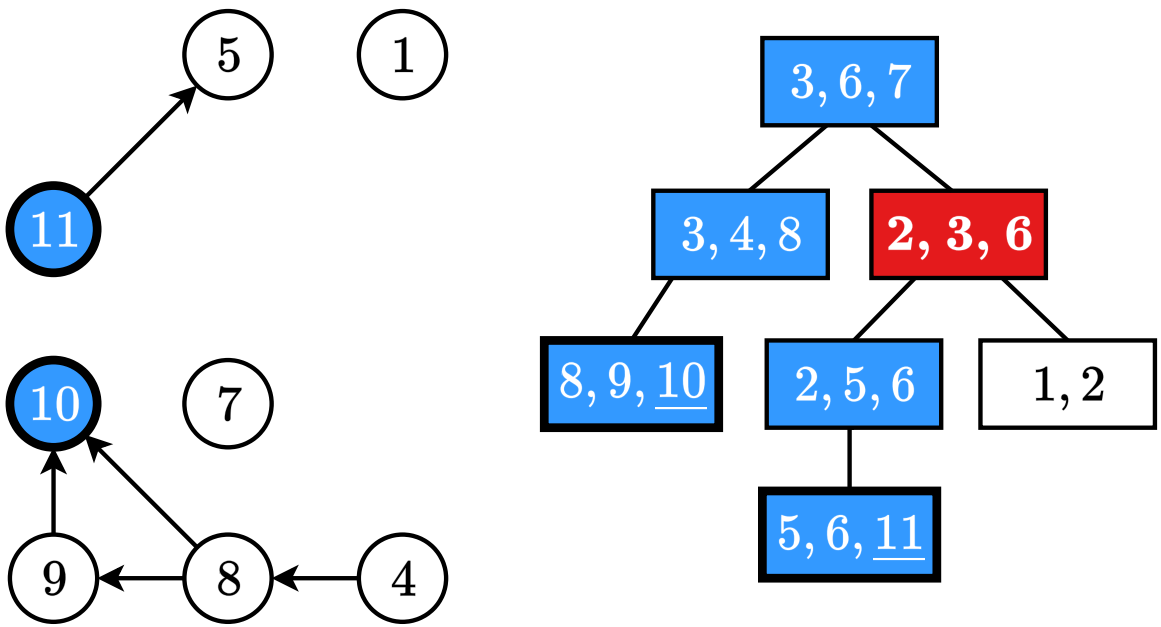


Intra-procedural algorithm #2

We look at the tree decomposition T of the CFG G.

Lemma

For any u, v in G, there there are bags b_u, b_v *in* T where every bag b on P_{b_u,b_v} separates u from v in G.

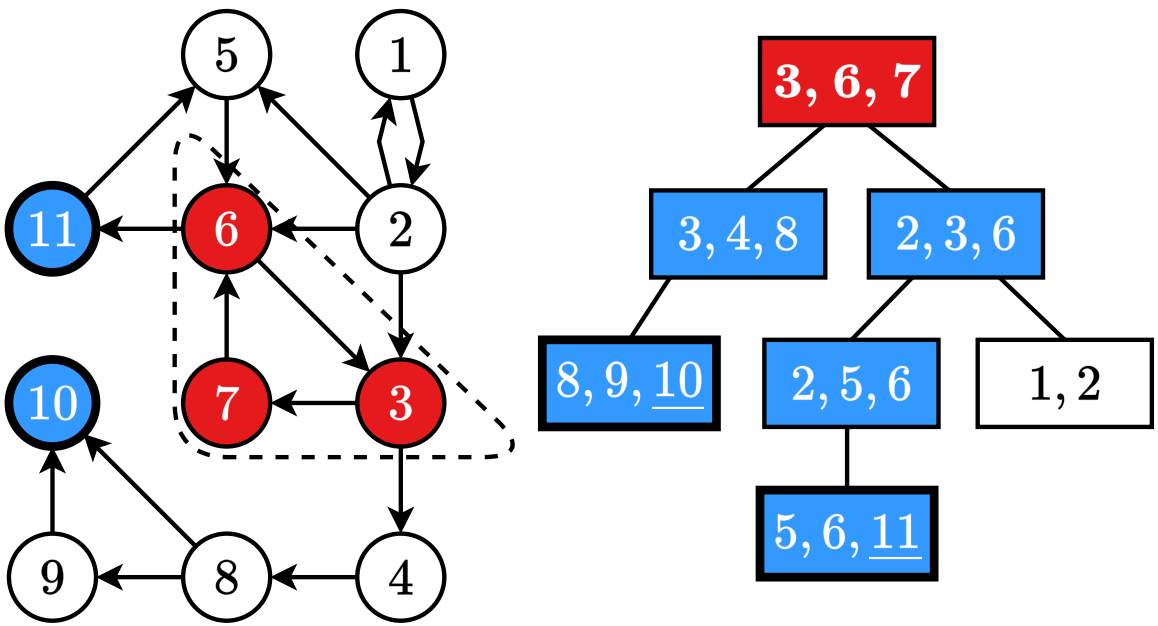


Intra-procedural algorithm #2

We look at the tree decomposition T of the CFG G.

Lemma

For any u, v in G, there there are bags b_u, b_v *in* T where every bag b on P_{b_u,b_v} separates u from v in G.

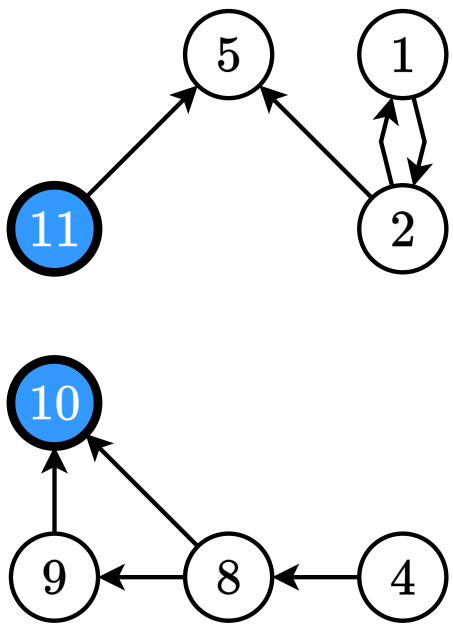


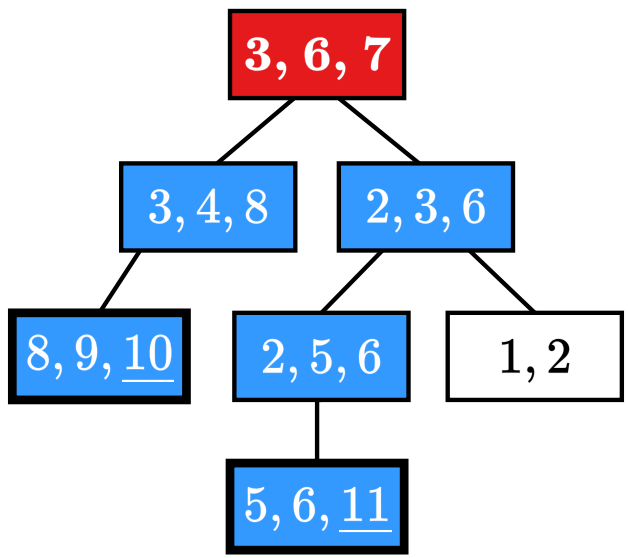
Intra-procedural algorithm #2

We look at the tree decomposition T of the CFG G.

Lemma

For any u, v in G, there there are bags b_u, b_v *in* T where every bag b on P_{b_u,b_v} separates u from v in G.





Intra-procedural algorithm #2

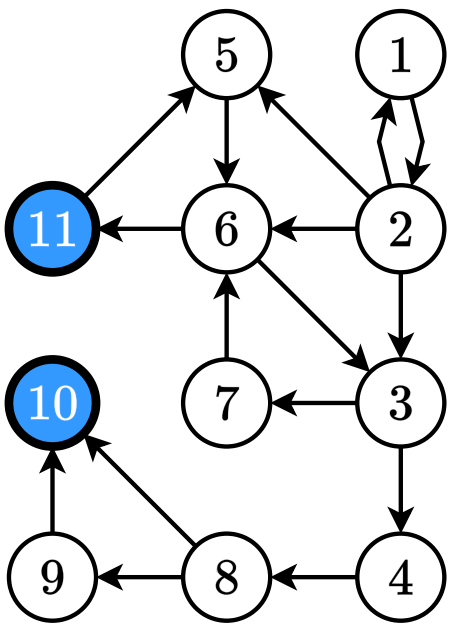
We look at the tree decomposition T of the CFG G.

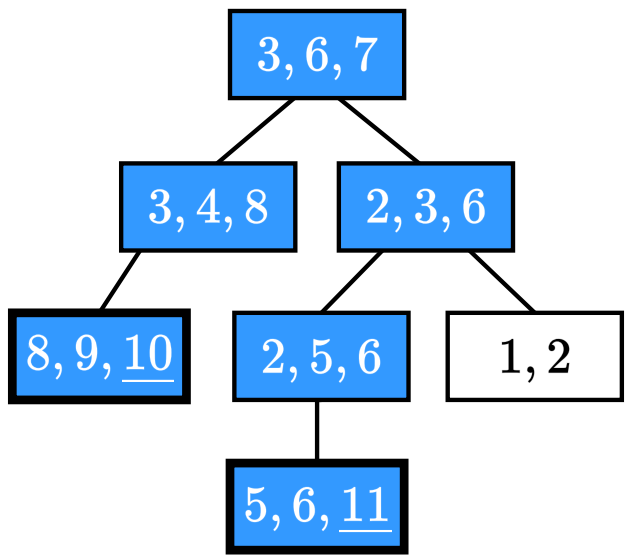
Lemma

For any u, v in G, there there are bags b_u, b_v *in* T where every bag b on P_{b_u,b_v} separates u from v in G.

A way to break the query!

For <u>any</u> bag $b \in P_{b_u,b_v}$, $[\![\rho_{u,v}]\!] = \bigoplus_{w \in V_b} [\![\rho_{u,w}]\!] \otimes [\![\rho_{w,v}]\!]$





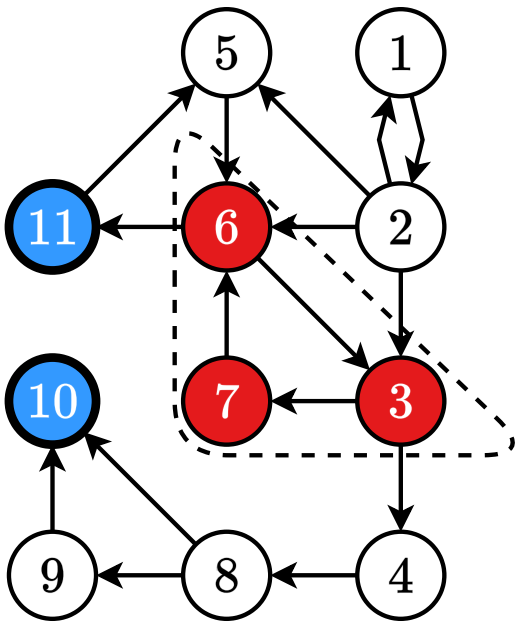
Intra-procedural algorithm #2 via tree decomposition

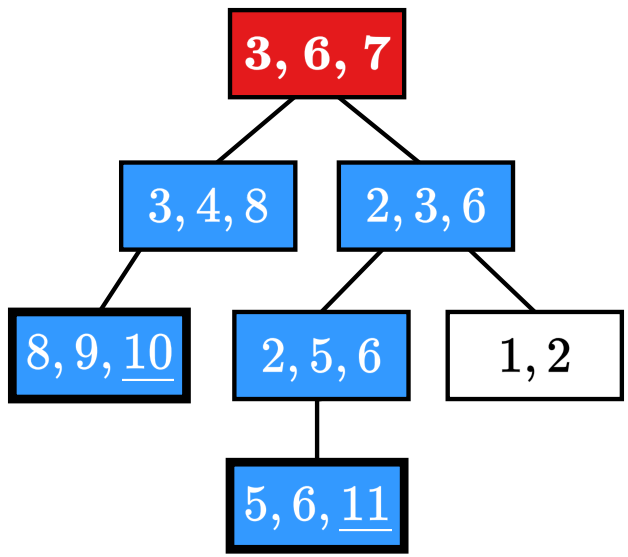
Idea:

- Always choose b to be the least common ancestor bag of b_u, b_v in T.

Picking $V_b = \{2, 3, 6\}$,

$$[\![\rho_{11,10}]\!] = ([\![\rho_{11,2}]\!] \otimes [\![\rho_{2,10}]\!]) \oplus ([\![\rho_{11,3}]\!] \otimes [\![\rho_{3,10}]\!]) \oplus ([\![\rho_{11,6}]\!] \otimes [\![\rho_{6,10}]\!])$$





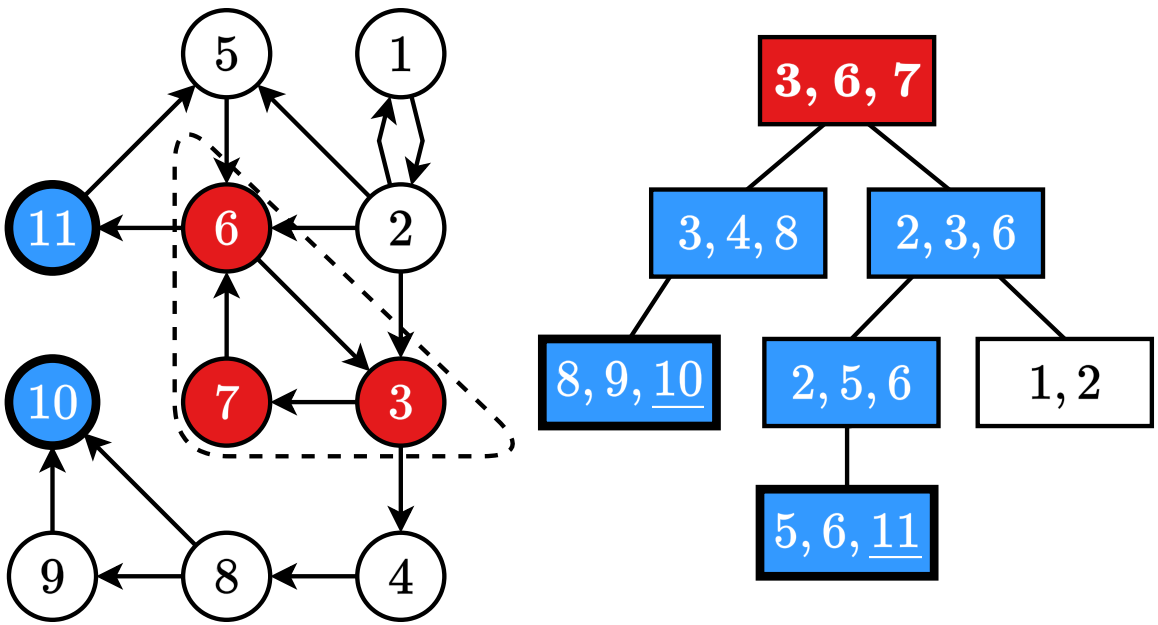
Intra-procedural algorithm #2 via tree decomposition

Idea:

- Always choose b to be the least common ancestor bag of b_u, b_v in T.
- Define "special queries" to be all pairs (u, v) where b_u is an ancestor/descendant of a b_v.

Picking $V_b = \{2, 3, 6\}$,

$$[\![\rho_{11,10}]\!] = ([\![\rho_{11,2}]\!] \otimes [\![\rho_{2,10}]\!]) \oplus ([\![\rho_{11,3}]\!] \otimes [\![\rho_{3,10}]\!]) \oplus ([\![\rho_{11,6}]\!] \otimes [\![\rho_{6,10}]\!])$$



Intra-procedural algorithm #2 via tree decomposition

Idea:

- Always choose b to be the least common ancestor bag of b_u, b_v in T.
- Define "special queries" to be all pairs (u, v) where b_u is an ancestor/descendant of a b_v.

Preprocessing: precompute all special queries.
Query (u, v):

- Find b_u and b_v.
- Let $b_{LCA} := LCA(b_u, b_v)$, the least common ancestor in T.
- return

$$[\![\rho_{u,v}]\!] = \bigoplus_{w \in b_{LCA}} [\![\rho_{u,w}]\!] \otimes [\![\rho_{w,v}]\!].$$

Intra-procedural algorithm #2 via tree decomposition

Idea:

- Always choose b to be the least common ancestor bag of b_u, b_v in T.
- Define "special queries" to be all pairs (u, v) where b_u is an ancestor/descendant of a b_v.

Preprocessing: precompute all special queries.
Query (u, v):

- Find b_u and b_v.
- Let $b_{LCA} := LCA(b_u, b_v)$, the least common ancestor in T.
- return

$$[\![\rho_{u,v}]\!] = \bigoplus_{w \in b_{LCA}} [\![\rho_{u,w}]\!] \otimes [\![\rho_{w,v}]\!].$$

Number of "special queries" $= O(n \cdot$ height of the TD$)$.

Intra-procedural algorithm #2 via tree decomposition

Idea:

- Always choose b to be the least common ancestor bag of b_u, b_v in T.
- Define "special queries" to be all pairs (u, v) where b_u is an ancestor/descendant of a b_v.

Preprocessing: precompute all special queries.
Query (u, v):

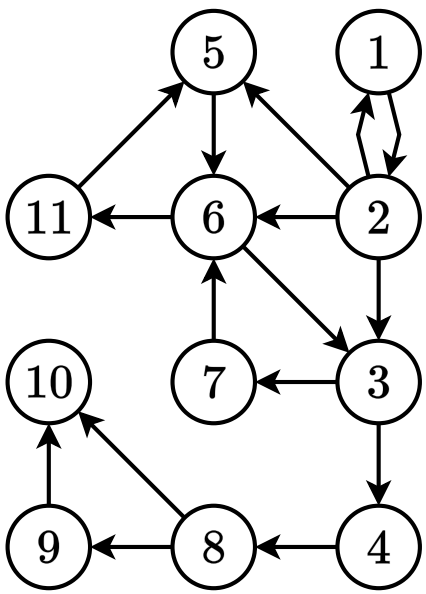
- Find b_u and b_v.
- Let $b_{LCA} := LCA(b_u, b_v)$, the least common ancestor in T.
- return

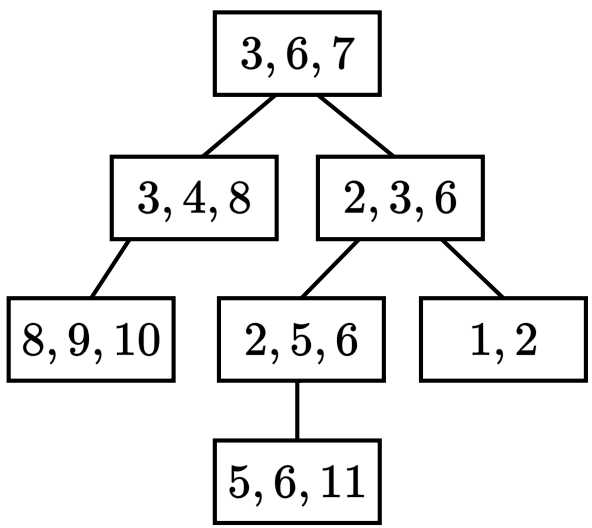
$$[\![\rho_{u,v}]\!] = \bigoplus_{w \in b_{LCA}} [\![\rho_{u,w}]\!] \otimes [\![\rho_{w,v}]\!].$$

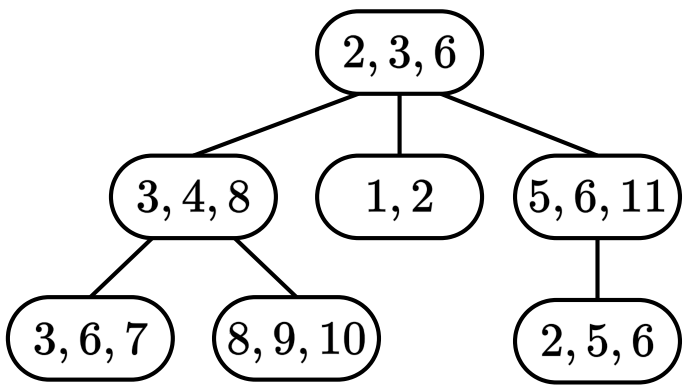
Number of "special queries" $= O(n \cdot$ height of the TD$)$.
Problem: if tree decomposition is too long $\implies O(n^2)$ special queries.

Intra-procedural algorithm #2 via tree + centroid decomp.

Solution: build a *centroid decomposition T' of the tree decomposition T, which has height $O(\log n)$*.



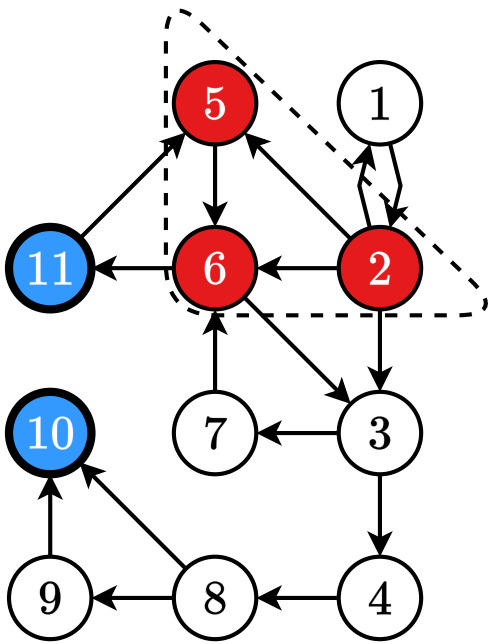


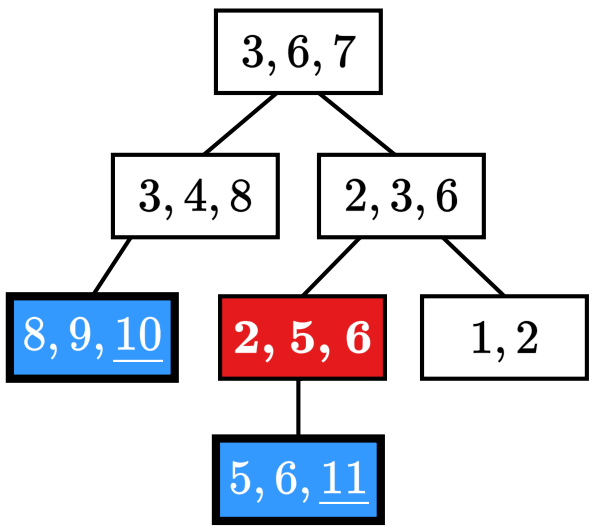


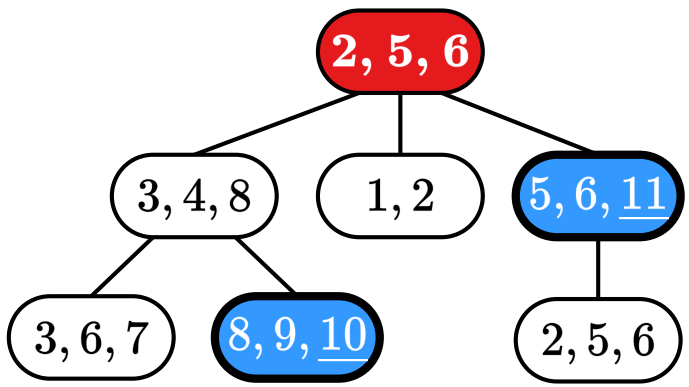
Intra-procedural algorithm #2 via tree + centroid decomp.

Solution: build a *centroid decomposition T' of the tree decomposition T, which has height $O(\log n)$.*

- Apply previous slide with b_{LCA} being LCA in T' not T.





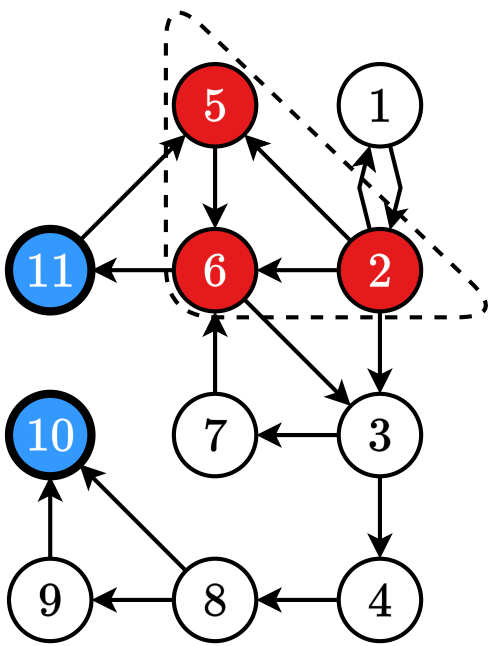


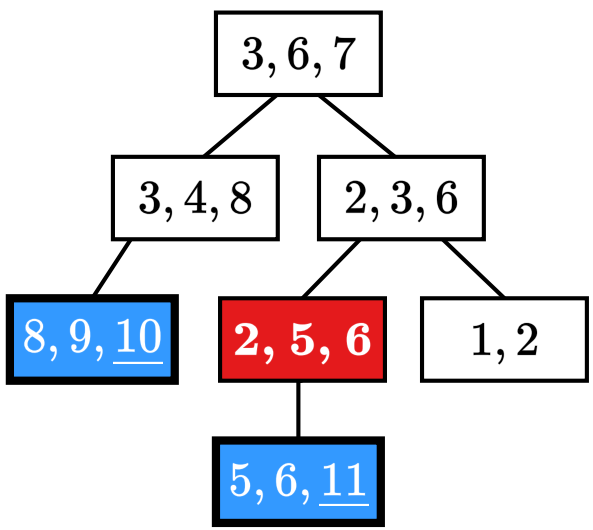
Intra-procedural algorithm #2 via tree + centroid decomp.

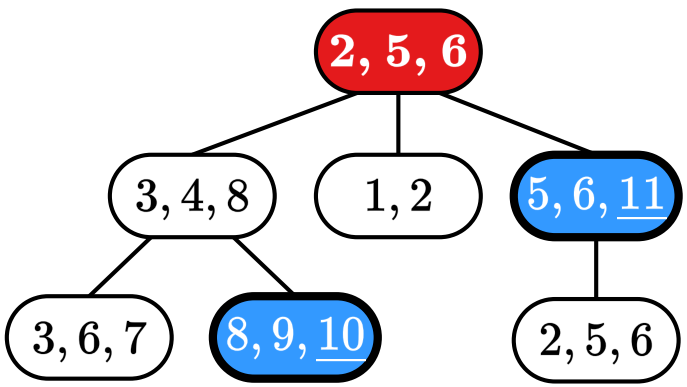
Solution: build a *centroid decomposition* T' *of the tree decomposition* T, *which has height* $O(\log n)$.

- Apply previous slide with b_{LCA} being LCA in T' not T.

Number of "special queries" $= O(n \cdot \log n)$.
We precompute all special queries in $O(n \cdot \log n \cdot k)$.







Inter-procedural algorithm

– *We extend our algorithm by exploiting sparsity of the call graph.*
– *Capture the sparsity using treedepth.*

See the paper for details.

Agenda

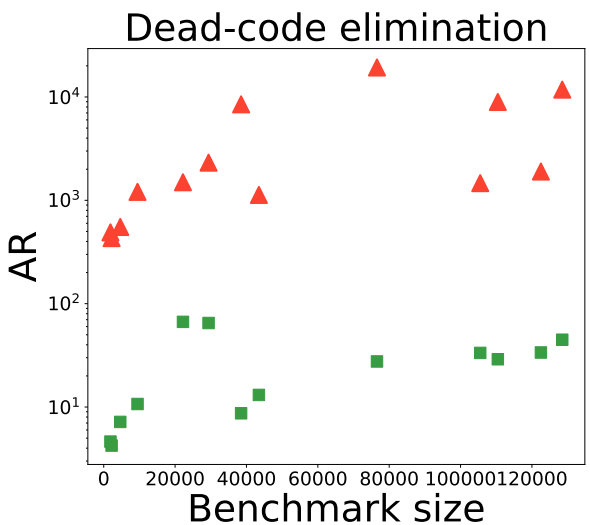
1. Context and contribution
2. Algorithms
3. Experiments and conclusion

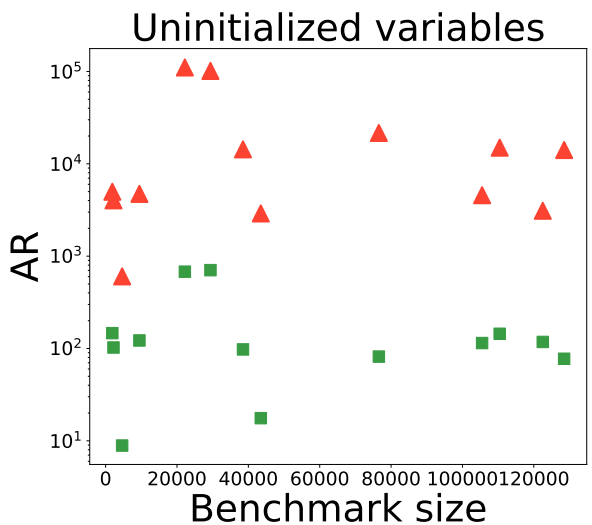
Experiments

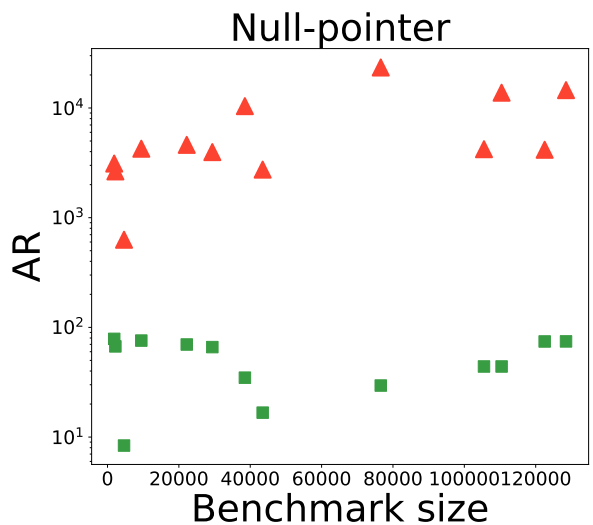
1. IFDS dataflow analyses (reachability, uninitialized variables, null-ptr):

- Each algebra element is the graph representation of IFDS.
- Used programs from DaCapo benchmarks.

Comparison of our algorithms vs. running Tarjan's algo. at every query:





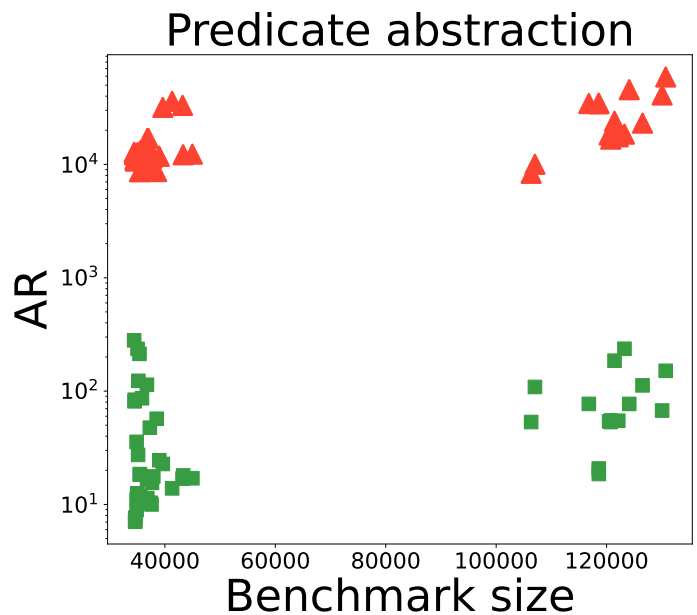


Experiments

2. Analysis of boolean programs:

- Each algebra element is a state transformer represented by a BDD.
- Used boolean programs generated from applying Predicate Abstraction on Windows drivers.

Comparison of our algorithms vs. running Tarjan's algo. at every query:



Conclusion

Fast algorithms for on-demand algebraic program analysis.

- Exploiting sparseness of CFGs (via treewidth) to handle the intra-procedural queries.
- Exploiting sparseness of CGs (via treedepth) to extend the solution to the inter-procedural case.
- Experiments showing efficiency in comparison with using Tarjan's algorithm.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_G(i, j)$.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

Existing solutions ($n = |V| \approx |E|$, cost per algebra operation is k):

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

Existing solutions ($n = |V| \approx |E|$, cost per algebra operation is k):

—*Kleene's NFA-to-regexp translation:*

- Works for arbitrary graphs.
- Precomputes all queries in $O(n^3 \cdot k)$. Too slow.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)
- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.
Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

Existing solutions ($n = |V| \approx |E|$, cost per algebra operation is k):

—*Kleene's NFA-to-regexp translation:*
- Works for arbitrary graphs.
- Precomputes all queries in $O(n^3 \cdot k)$. Too slow.

—*Tarjan's algorithm:*
- Works on reducible flow graphs ($\approx$ CFGs).
- Answers all queries $(i, -)$ for a fixed i in $O(n\alpha(n) \cdot k)$.
- Doesn't suit our on-demand setting:
  - for n queries with different i's, naive repetition $\implies \Omega(n^2)$ time.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG* $G = (V, E)$ *of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_G(i, j)$.

—*The paper presents two algorithms:*

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

—*The paper presents two algorithms:*

Algorithm #1:

- Operates directly on the structure of CFG.
- Assumes programs have constant nesting depth.
- Preprocessing: $O(n \cdot \log \log n \cdot k)$; query $O(k)$.

Intra-procedural algorithms

Intra-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a CFG $G = (V, E)$ of a single function,*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$,
- a semantic function $[\![\cdot]\!] : E \to A$.

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = Paths_G(i, j)$.

—*The paper presents two algorithms:*

Algorithm #1:

- Operates directly on the structure of CFG.
- Assumes programs have constant nesting depth.
- Preprocessing: $O(n \cdot \log \log n \cdot k)$; query $O(k)$.

Algorithm #2:

- Operates on the "tree decomposition" of the CFG.
- Assumes CFGs have constant treewidth: more robust assumption.
- Preprocessing: $O(n \cdot \log n \cdot k)$; query $O(k)$.

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.

P

$\sigma_1;$

$\sigma_2;$

$\texttt{branch}_3$

$P_1,$

P_2

$\texttt{end}_3;$

$\sigma_4;$

$\texttt{loop}_5$

P'

$\texttt{end}_5;$

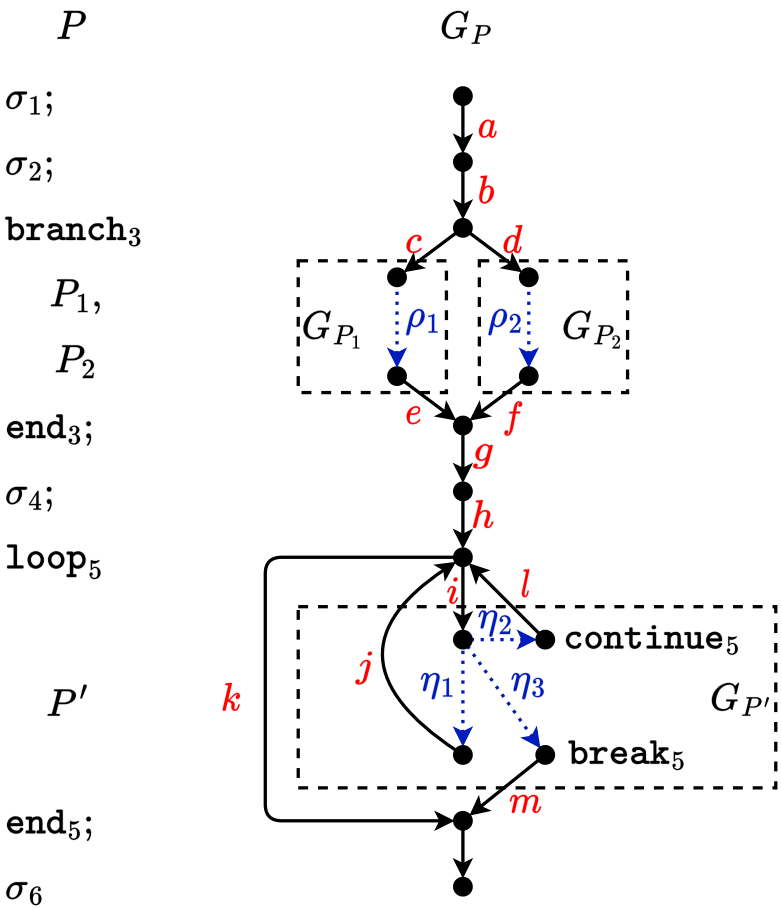
σ_6

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.

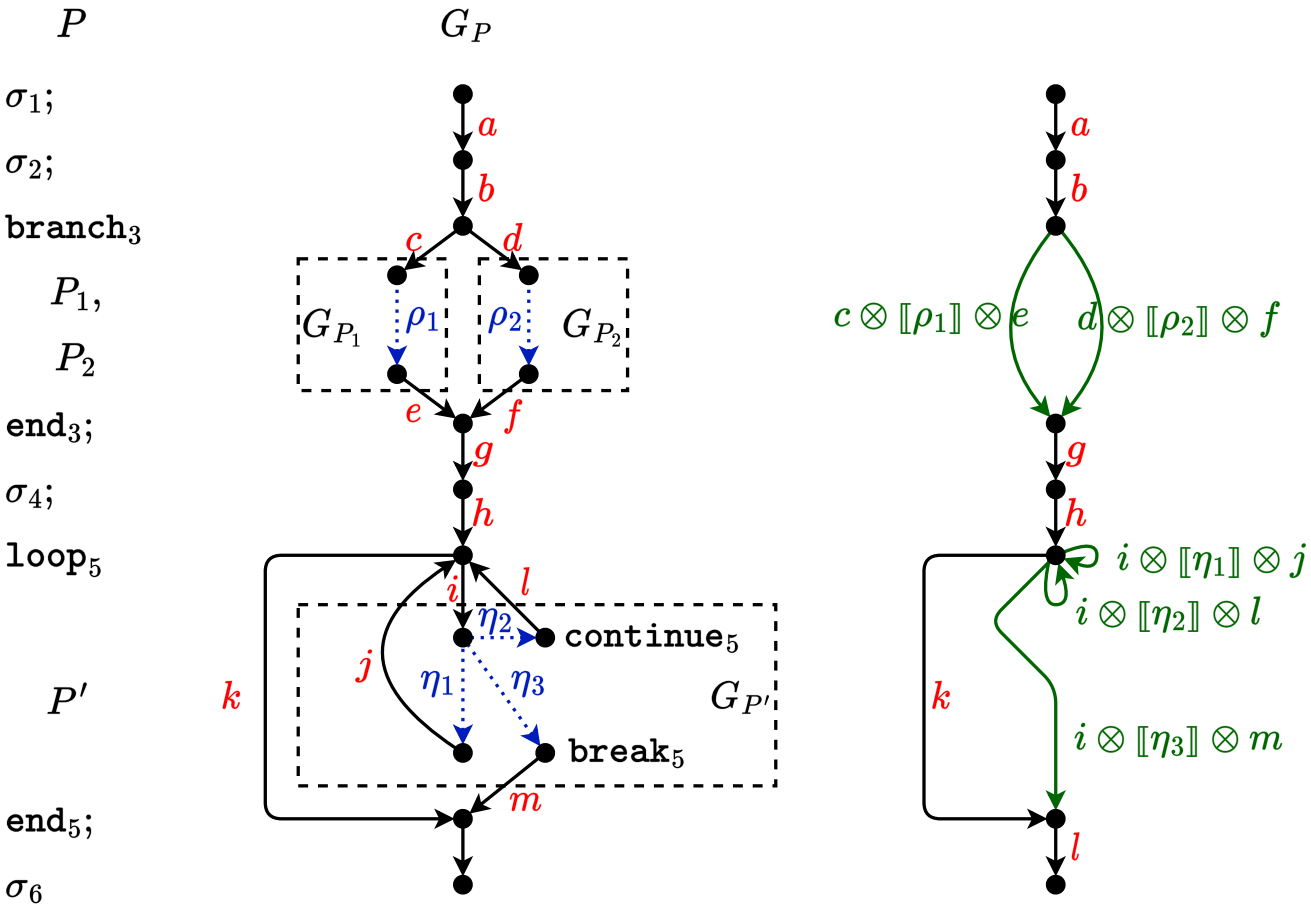


Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.

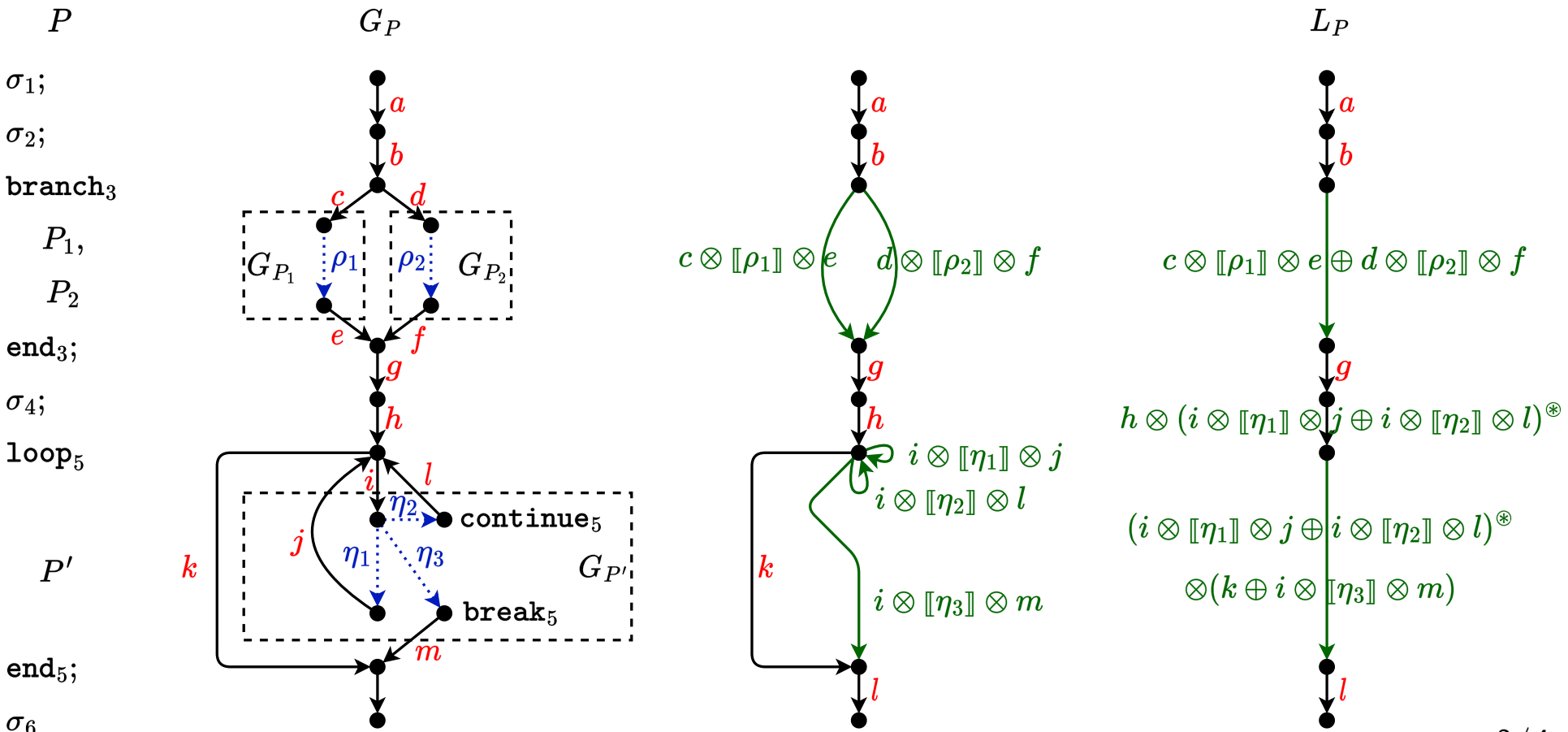


Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.



Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.
- Build a sqrt-tree data structure to efficiently answer same-level queries.

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.
- Build a sqrt-tree data structure to efficiently answer same-level queries.

To answer a query (i, j), a path $i \rightsquigarrow j$ either:

- Visits some node at the top-level,
- Or it doesn't

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.
- Build a sqrt-tree data structure to efficiently answer same-level queries.

To answer a query (i, j), a path $i \rightsquigarrow j$ either:

- Visits some node at the top-level,
  — can be answered with the sqrt-tree.
- Or it doesn't
  — reduces to a query in a subprogram with smaller depth.

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.
- Build a sqrt-tree data structure to efficiently answer same-level queries.

To answer a query (i, j), a path $i \rightsquigarrow j$ either:

- Visits some node at the top-level,
  — can be answered with the sqrt-tree.
- Or it doesn't
  — reduces to a query in a subprogram with smaller depth.
- Answers for different paths are combined with $\oplus$ and $\otimes$

Intra-procedural algorithm #1

$$P := \sigma \mid P; P \mid \texttt{branch}_l\ P, P, \ldots, P\ \texttt{end}_l \mid \texttt{loop}_l\ P\ \texttt{end}_l \mid \texttt{break}_l \mid \texttt{continue}_l$$

Preprocessing:

- Structurally recursive: before processing P, process its subprograms first.
- For each sub-program, precompute queries only at the top-level.
- Build a sqrt-tree data structure to efficiently answer same-level queries.

To answer a query (i, j), a path $i \rightsquigarrow j$ either:

- Visits some node at the top-level,
  — can be answered with the sqrt-tree.
- Or it doesn't
  — reduces to a query in a subprogram with smaller depth.
- Answers for different paths are combined with $\oplus$ and $\otimes$
- Efficiency relies on having a small nesting depth.

Call graphs (CGs) and treedepth

Call graph (CG): $C = (\{f_1, \ldots, f_m\}, E_C), (f_i, f_j) \in E_C \iff \{f_i \text{ calls } f_j\}$.

Call graphs (CGs) and treedepth

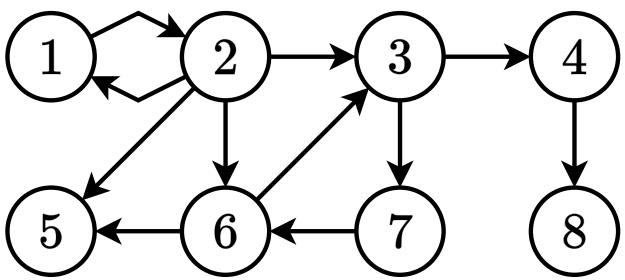
Call graph (CG): $C = (\{f_1, \ldots, f_m\}, E_C), (f_i, f_j) \in E_C \iff \{f_i \text{ calls } f_j\}$.
Treedepth:

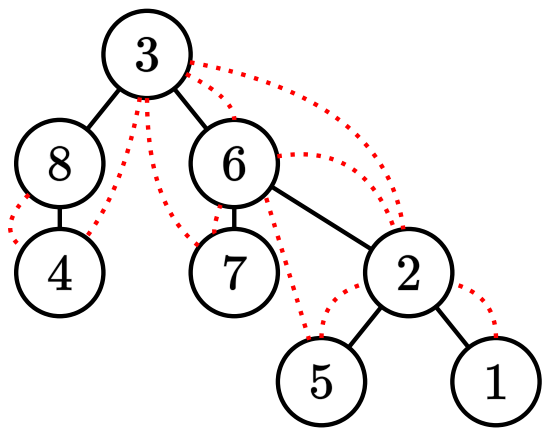
- (Informally) measures for a graph how similar it is to a *shallow tree*.

Call graphs (CGs) and treedepth

Call graph (CG): $C = (\{f_1, \ldots, f_m\}, E_C), (f_i, f_j) \in E_C \iff \{f_i \text{ calls } f_j\}$.
Treedepth:

- (Informally) measures for a graph how similar it is to a *shallow tree*.
- Small-treedepth graphs admit a *depth decomposition* with small *depth.*
- Similar to treewidth, such decomposition enable us to successively divide a graph into smaller components separated by *small cuts.*

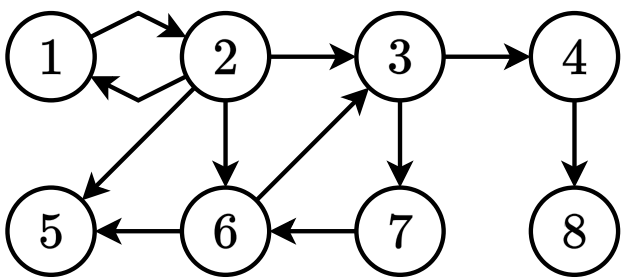


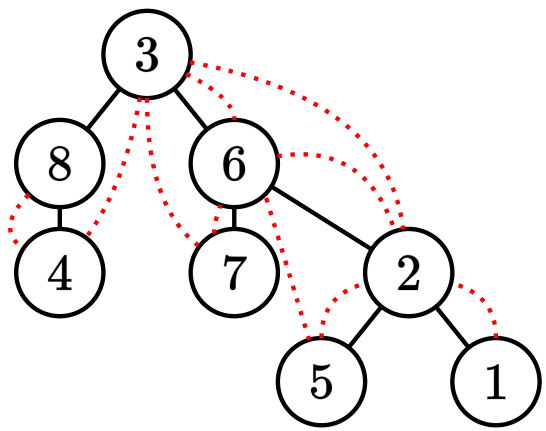


Call graphs (CGs) and treedepth

Call graph (CG): $C = (\{f_1, \ldots, f_m\}, E_C), (f_i, f_j) \in E_C \iff \{f_i \text{ calls } f_j\}$.
Treedepth:

- (Informally) measures for a graph how similar it is to a *shallow tree*.
- Small-treedepth graphs admit a *depth decomposition* with small *depth.*
- Similar to treewidth, such decomposition enable us to successively divide a graph into smaller components separated by *small cuts.*
- We assume CGs have small treedepth w.r.t. program size; justified experimentally.

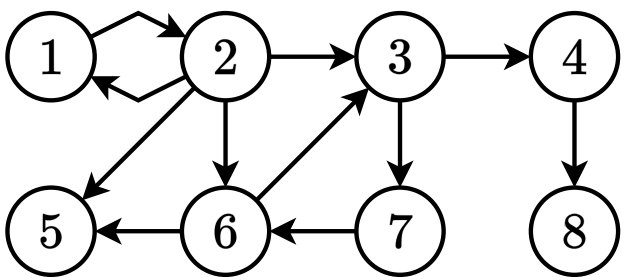


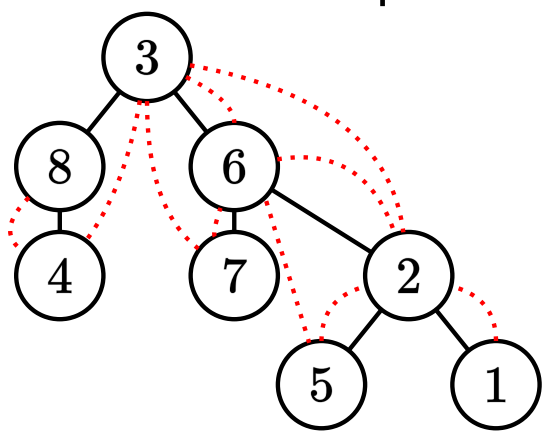


Call graphs (CGs) and treedepth

Call graph (CG): $C = (\{f_1, \ldots, f_m\}, E_C), (f_i, f_j) \in E_C \iff \{f_i \text{ calls } f_j\}$.
Treedepth:

- (Informally) measures for a graph how similar it is to a *shallow tree*.
- Small-treedepth graphs admit a *depth decomposition* with small *depth.*
- Similar to treewidth, such decomposition enable us to successively divide a graph into smaller components separated by *small cuts.*
- We assume CGs have small treedepth w.r.t. program size; justified experimentally.
- We exploit this assumption to efficiently solve the inter-procedural case.





Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Function summaries are given $\to$ no *return edges* $\to$ same as the intra-procedural case, but on a larger graph *with different structure.*

Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Function summaries are given $\to$ no *return edges* $\to$ same as the intra-procedural case, but on a larger graph *with different structure.*

Preprocessing:

Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Function summaries are given $\to$ no *return edges* $\to$ same as the intra-procedural case, but on a larger graph *with different structure.*

Preprocessing:

- Run the intra-procedural algorithm for each function.

Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Function summaries are given $\to$ no *return edges* $\to$ same as the intra-procedural case, but on a larger graph *with different structure.*

Preprocessing:

- Run the intra-procedural algorithm for each function.
- For each CG edge (f_i, f_j), compute a value $[\![(f_i, f_j)]\!]$ summarizing all all paths lying in f_i with only last vertex in f_j.

Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Function summaries are given $\to$ no *return edges* $\to$ same as the intra-procedural case, but on a larger graph *with different structure.*

Preprocessing:

- Run the intra-procedural algorithm for each function.
- For each CG edge (f_i, f_j), compute a value $[\![(f_i, f_j)]\!]$ summarizing all all paths lying in f_i with only last vertex in f_j.

Break a query into: intra-procedural queries, and *call-graph queries*.

Inter-procedural algorithm

Inter-procedural on-demand algebraic program analysis

Offline input: (can be preprocessed)

- *a augmented graph* $\widehat{G} = (V, \widehat{E})$ *(union of CFGs + call edges),*
- an algebra $(A, \oplus, \otimes, \circledast, \overline{0}, \overline{1})$, a semantic function $[\![\cdot]\!] : \widehat{E} \to A$,
- *for each function* f_i*, a value* $[\![f_i]\!]$ *summarizing* f_i*'s execution.*

Online input: a series of queries (i, j), each is a pair of program points.

Output: for each query (i, j), compute $[\![\rho_{i,j}]\!]$ where $\langle \rho_{i,j} \rangle = \mathit{Paths}_{\widehat{G}}(i, j)$.

Function summaries are given $\to$ no *return edges* $\to$ same as the intra-procedural case, but on a larger graph *with different structure.*

Preprocessing:

- Run the intra-procedural algorithm for each function.
- For each CG edge (f_i, f_j), compute a value $[\![(f_i, f_j)]\!]$ summarizing all all paths lying in f_i with only last vertex in f_j.

Break a query into: intra-procedural queries, and *call-graph queries.*

Answering call-graph queries: find depth decomp. $\to$ convert to tree decomp $\to$ apply treewidth-based algorithm. Done!