Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth

Krishnendu Chatterjee [1], Rasmus Ibsen-Jensen [1],
Andreas Pavlogiannis [1], Prateesh Goyal [2]

[1]IST Austria, [2]IIT Bombay

July 21, 2015

Interprocedural Analysis

Classic algorithmic problem on the control-flow graphs of programs

- reachability
- constant propagation
- live variable analysis
- ...

Algorithmic Improvements

1. Multiple queries
   - Preprocess phase
   - Query phase
2. Control-flow graphs have special structure

Recursive State Machines

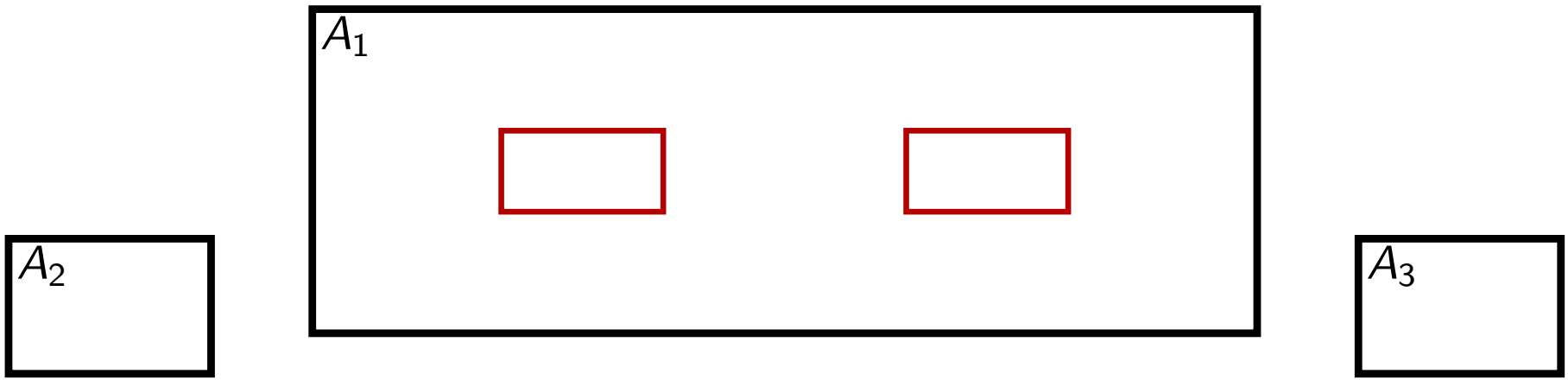
A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.

Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

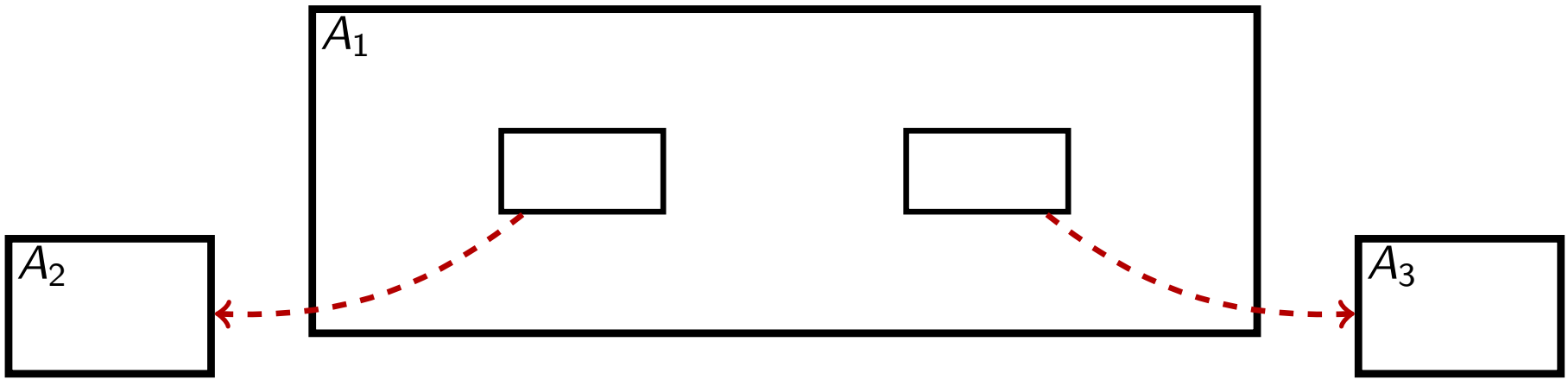
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

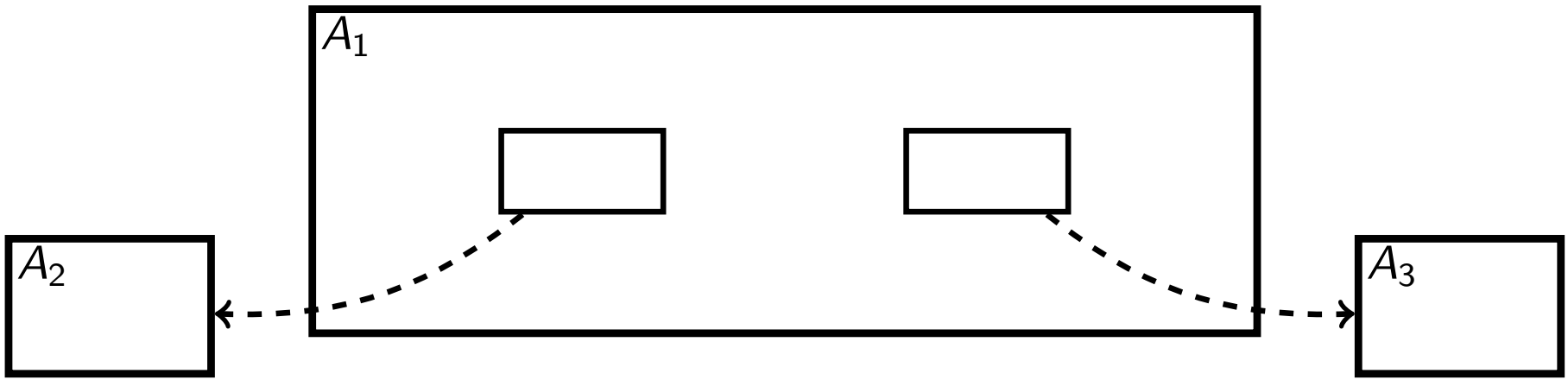
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

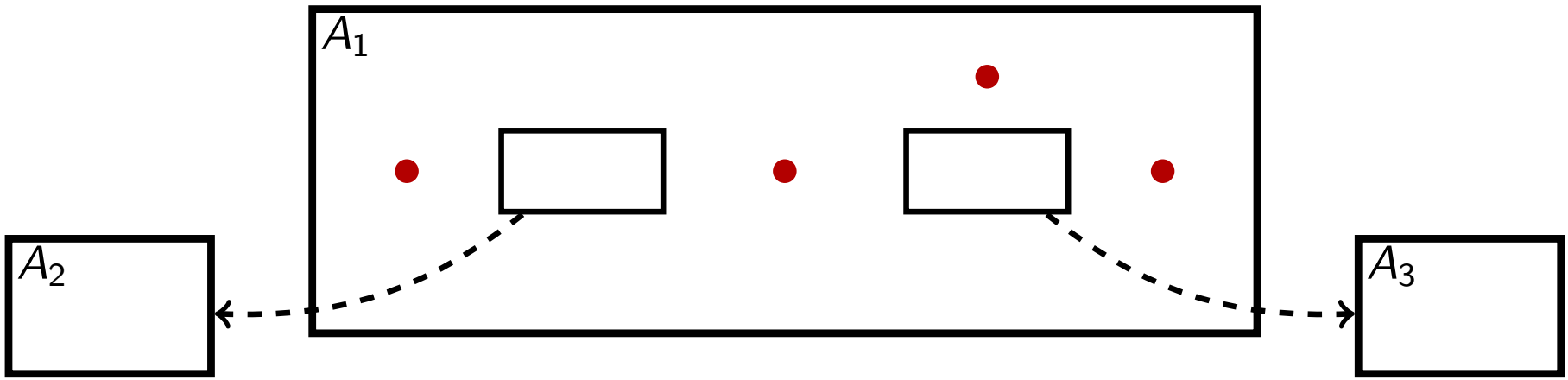
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

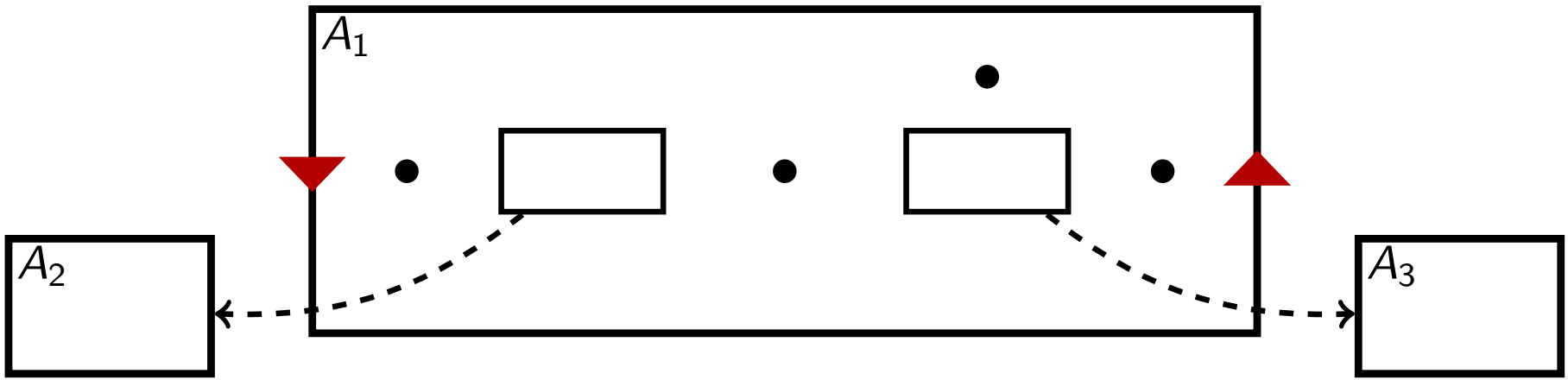
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

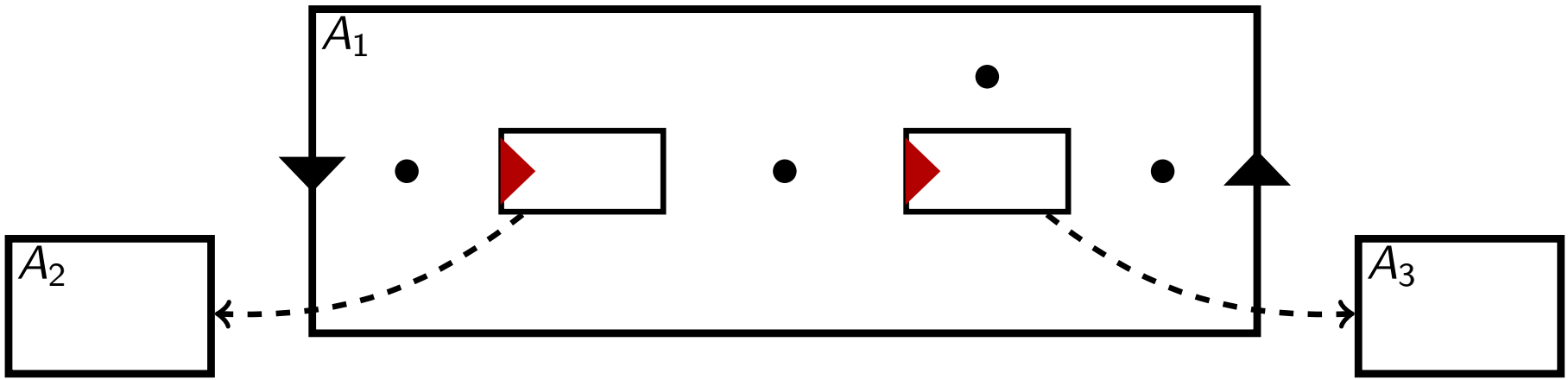
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

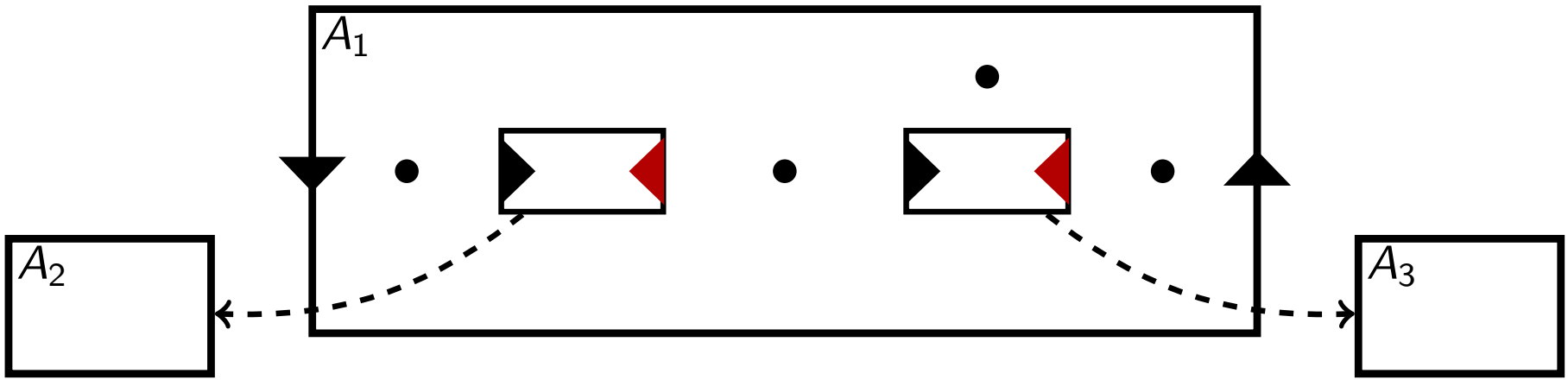
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

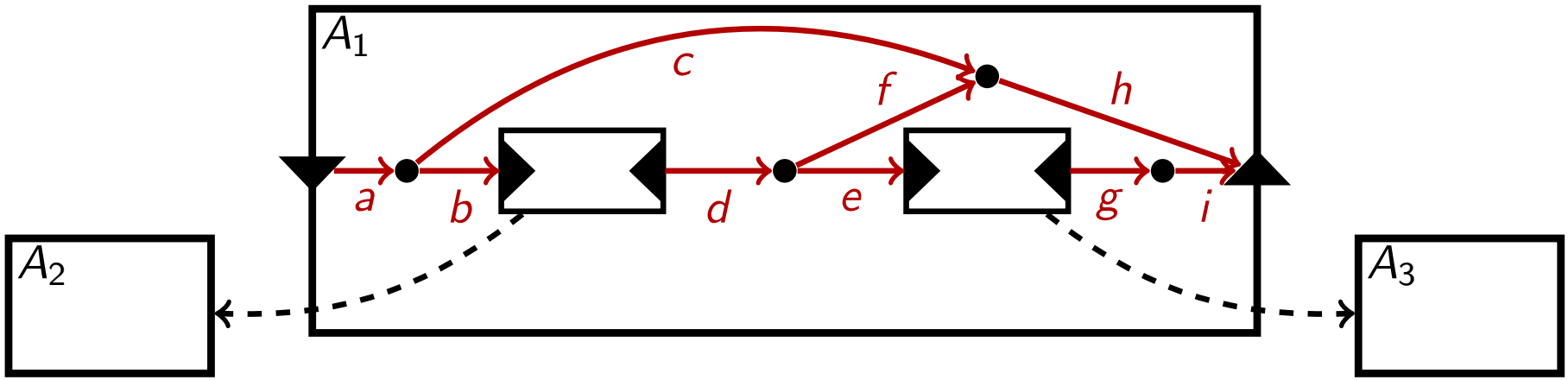
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Recursive State Machines

A **Recursive State Machine** is a collection of modules $\{A_1, A_2, \ldots, A_k\}$. Each A_i consists of

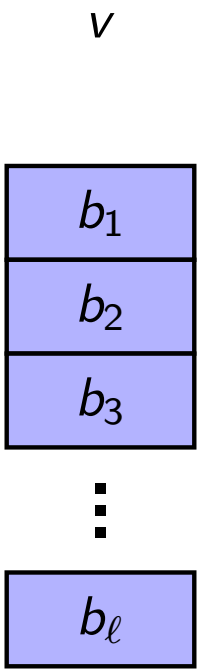
1. A set B_i of boxes
2. A map $Y_i : B_i \to \{1, \ldots k\}$ from each box to a module
3. A set V_i of nodes:
   1. A set N_i of internal nodes
   2. An entry node En_i, an exit node Ex_i
   3. A set C_i of call nodes
   4. A set R_i of return nodes
4. A set of internal edges E_i, and a weight function $\mathrm{wt}_i : E_i \to \Sigma$.



Configurations

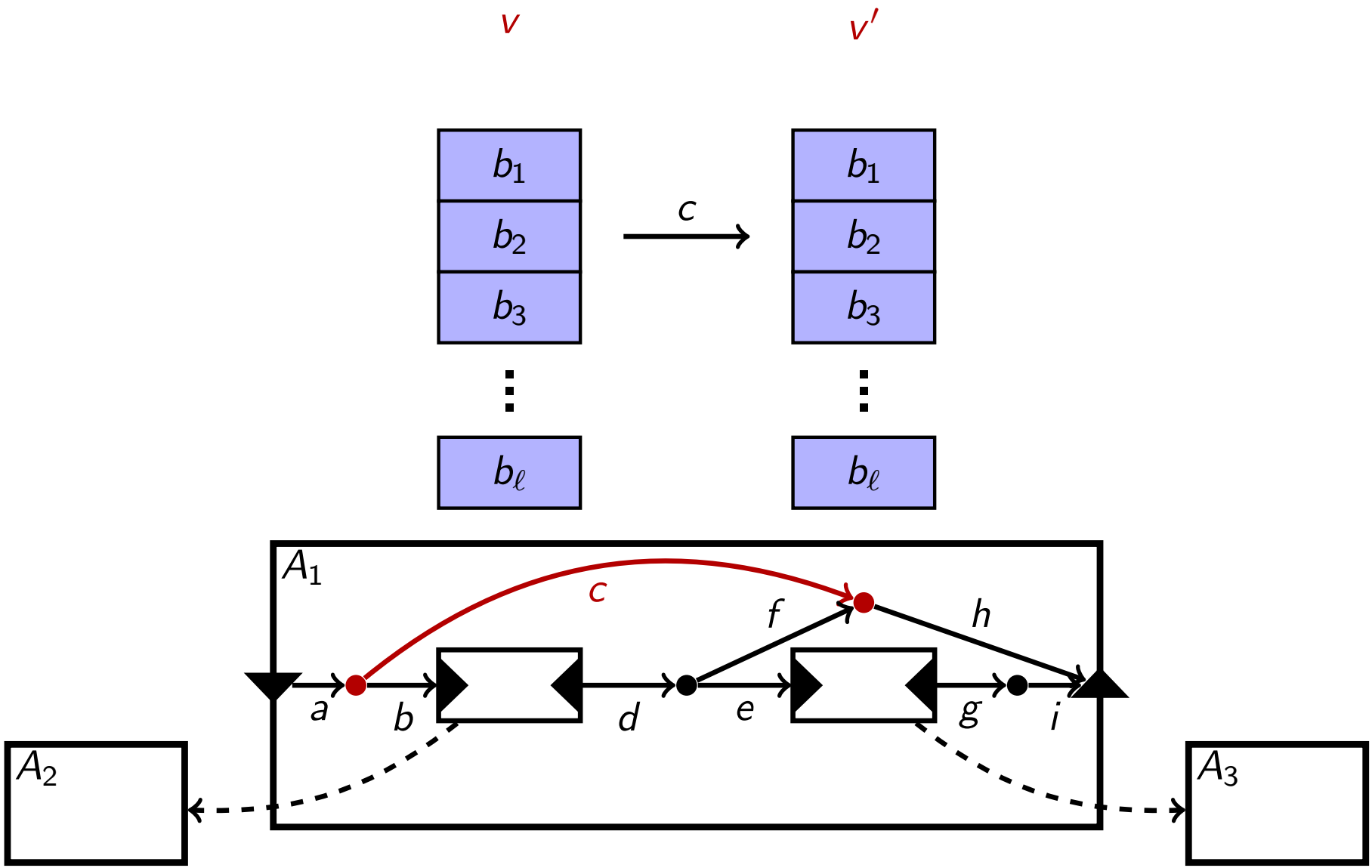
A **configuration** is a pair $c = (v, L)$

1. v is a node of a CSM
2. L is a sequence of boxes called the stack



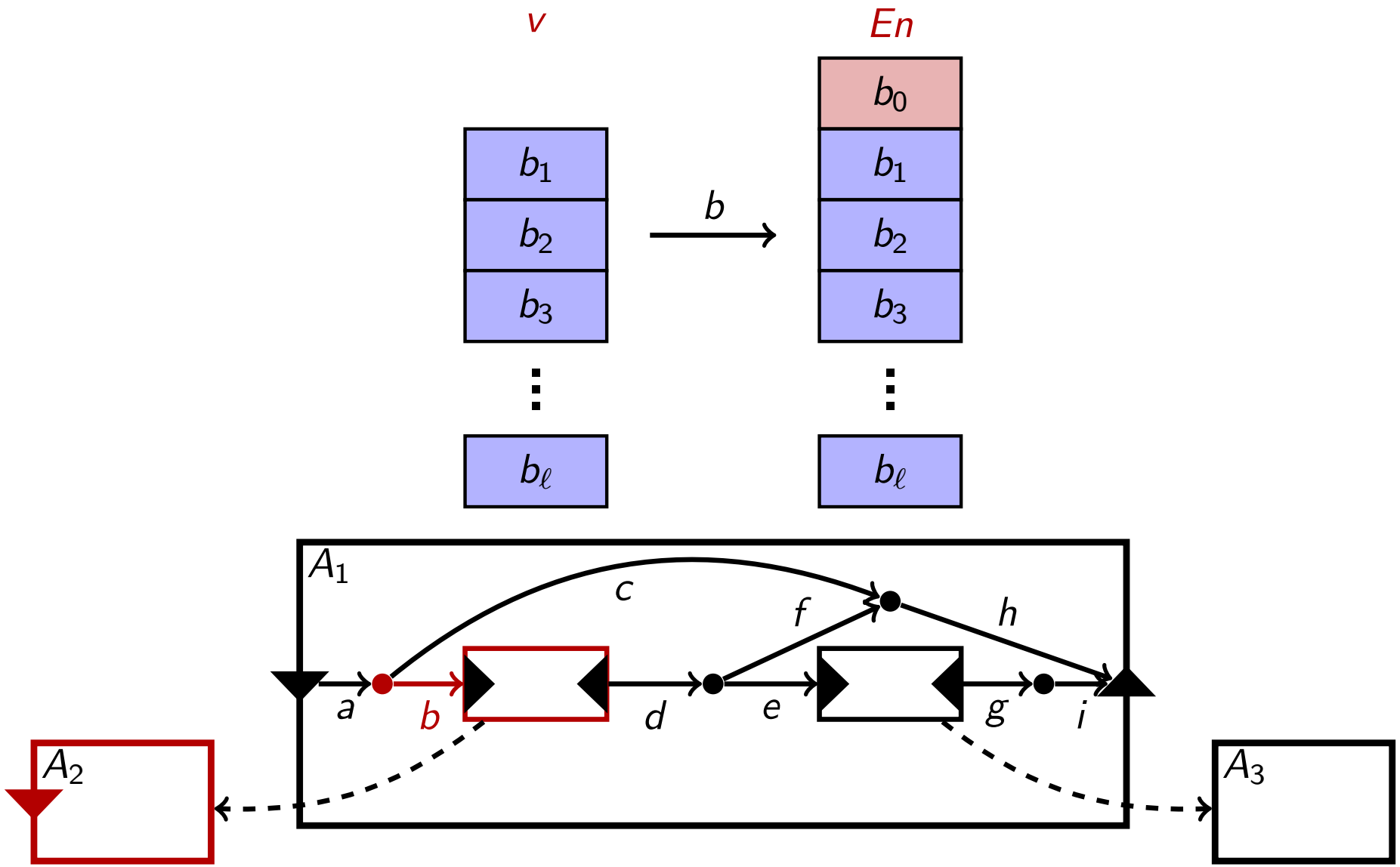
Transitions

Internal transition



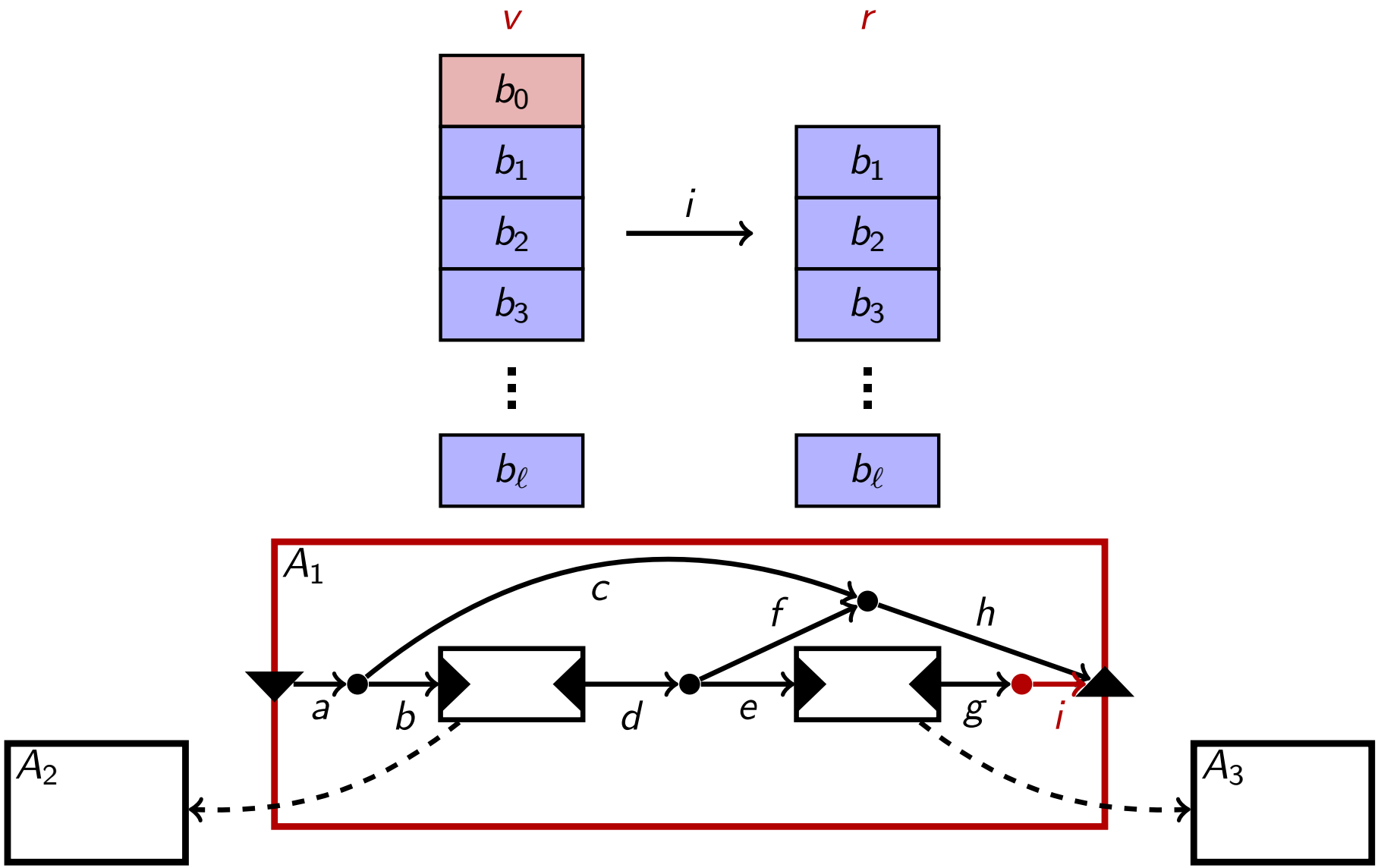
Transitions

Call transition



Transitions

Return transition



Paths & Distances

- A **path** $P = (c_1, \ldots c_\ell)$ is a sequence of labeled transitions
  - Only consider *realizable* paths
- Σ together with $\bigotimes$ and $\bigoplus$ form a semiring
- The weight $\otimes(P)$ of P is $(\sigma_1 \otimes \sigma_2 \cdots \otimes \sigma_\ell)$
- The **distance** from node u to v is

$$d(u, v) = \bigoplus_{P: u \to v} \otimes(P)$$

Paths & Distances

- A **path** $P = (c_1, \ldots c_\ell)$ is a sequence of labeled transitions
  - Only consider *realizable* paths
- Σ together with $\bigotimes$ and $\bigoplus$ form a semiring
- The weight $\otimes(P)$ of P is $(\sigma_1 \otimes \sigma_2 \cdots \otimes \sigma_\ell)$
- The **distance** from node u to v is

$$d(u, v) = \bigoplus_{P: u \to v} \otimes(P)$$

Modeled Problems

- Reachability Boolean semiring. $\Sigma = \{\text{True}, \text{False}\}$, $\bigotimes = \wedge$, $\bigoplus = \vee$
- Shortest path Tropical semiring. $\Sigma = \mathbb{R}_{\geq 0} \cup \{\infty\}$, $\bigotimes = +$, $\bigoplus = \min$
- IFDS (Interprocedural, Finite, Distributed, Subset Problems)
  - Finite set D of data facts
  - Universe F of distributive flow functions $f : 2^D \to 2^D$
  - Functions are composed along a path $f_1 \circ f_2, \ldots$
  - Meet $\sqcap$ over all paths

  Meet-composition semiring $\Sigma = F$, $\bigotimes = \circ$, $\bigoplus = \sqcap$

Reachability and Shortest Path

Existing algorithms for single instances

Reachability	
Time	Space
$O(n)$	$O(n)$

Shortest Path	
Time	Space
$O(n^4)$	$O(n)$

Consideration 1: Preprocess vs Query

Preprocess vs Query

- **Preprocess phase**: Spend some resources to preprocess the RSM
- **Query phase**:
  - Pair query: Given nodes u, v in A_i, return $d(u, v)$
  - Single-source query: Given a node u in A_i, return $d(u, v)$ for every other node v in A_i

Complete vs No Preprocessing

Preprocessing spectrum

No Preprocessing

Complete Preprocessing

Our Work

Complete vs No Preprocessing

Preprocessing spectrum

No Preprocessing

Our Work

Complete Preprocessing

Consideration 2: Treewidth of control-flow graphs

Our Results (Reachability)

Reachability				
	Preprocess time	Space	Single-source query	Pair query
Complete preprocessing	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
No preprocessing	-	$O(n)$	$O(n)$	$O(n)$
Our Results	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$
	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$

Our Results (Reachability)

Reachability				
	Preprocess time	Space	Single-source query	Pair query
Complete preprocessing	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
No preprocessing	-	$O(n)$	$O(n)$	$O(n)$
Our Results	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$
	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$

Our Results (Shortest Path)

Shortest Path				
	Preprocess time	Space	Single-source query	Pair query
Complete preprocessing	$O(n^5)$	$O(n^2)$	$O(n)$	$O(1)$
No preprocessing	-	$O(n)$	$O(n^4)$	$O(n^4)$
Our Results	$\mathbf{O(n^2 \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$
	$\mathbf{O(n^2 \cdot \log n)}$	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$

Our Results (Shortest Path)

Shortest Path				
	Preprocess time	Space	Single-source query	Pair query
Complete preprocessing	$O(n^5)$	$O(n^2)$	$O(n)$	$O(1)$
No preprocessing	-	$O(n)$	$O(n^4)$	$O(n^4)$
Our Results	$\mathbf{O(n^2 \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$
	$\mathbf{O(n^2 \cdot \log n)}$	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$

Our Results (IFDS)

IFDS				
	Preprocessing time	Space	S.s. query	Pair query
1	$O(n^2 \cdot \|D\|^3)$	$O(n^2 \cdot \|D\|)$	$O(n \cdot \|D\|)$	$O(\|D\|)$
2	-	$O(n \cdot \|D\|)$	$O(n \cdot \|D\|^3)$	$O(n \cdot \|D\|^3)$
3a	$O(n \cdot \log n \cdot \|D\|^3)$	$O(n \cdot \log n \cdot \|D\|^2)$	$O(n \cdot \|D\|^2)$	$O(\|D\|^2)$
3b	$O(n \cdot \log n \cdot \|D\|^3)$	$O(n \cdot \|D\|^2)$	$O(n \cdot \|D\|^2)$	$O(\log n \cdot \|D\|^2)$

- **1.** IFDS complete preprocessing
- **2.** IFDS no preprocessing
- **3a** Our results (time optimized)
- **3b** Our results (space optimized)

Tree-Decompositions and Treewidth

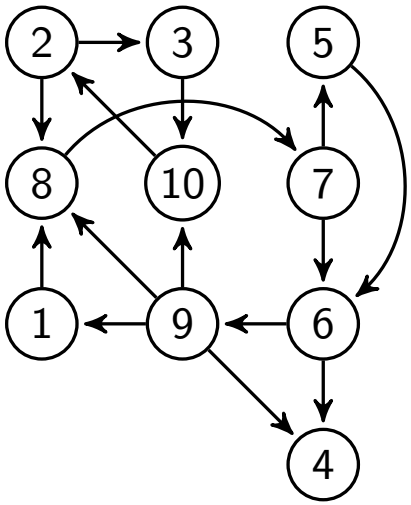
Tree Decomposition

Definition (Tree decomposition)

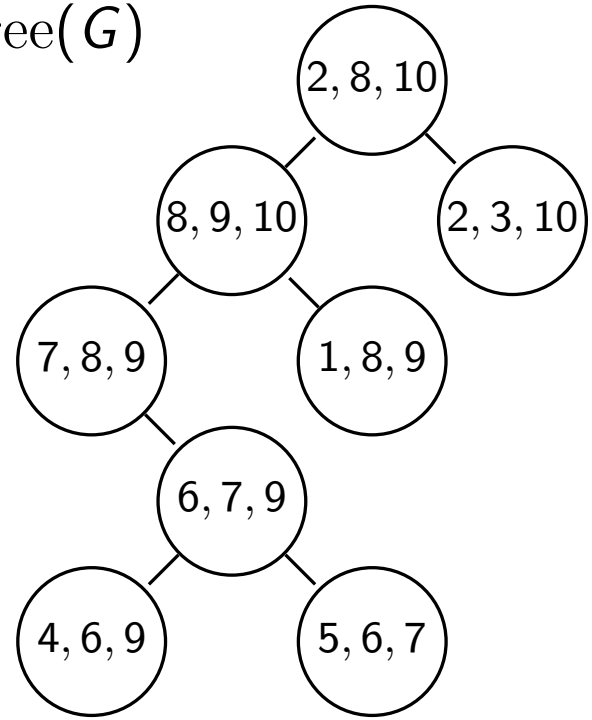
Given a graph $G = (V, E)$, a **tree-decomposition** $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree of bags $B_i \subseteq V$ such that:

1. Every node of G is contained in a bag
2. Every edge of G is contained in a bag
3. Every node of G appears in a contiguous subtree of $\mathrm{Tree}(G)$.

G



$\mathrm{Tree}(G)$



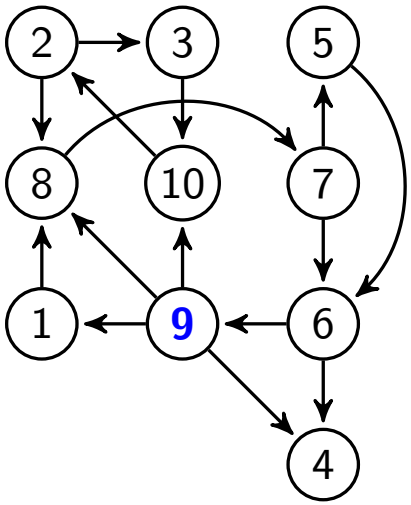
Tree Decomposition

Definition (Tree decomposition)

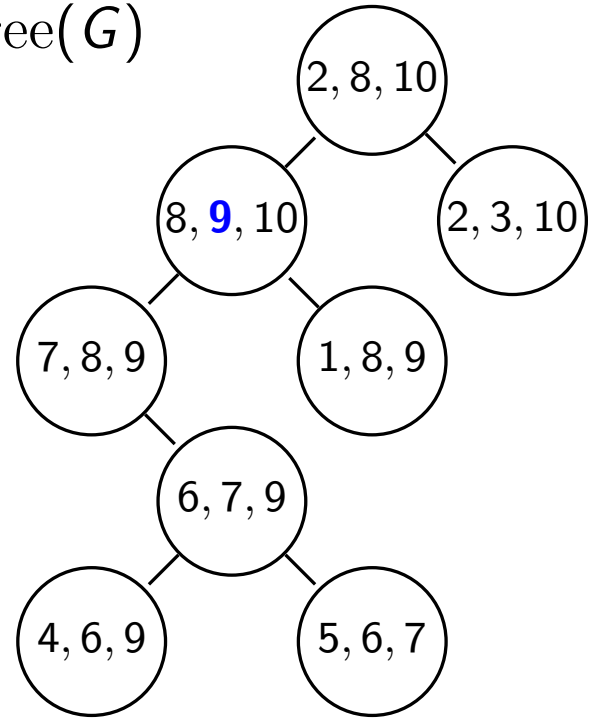
Given a graph $G = (V, E)$, a **tree-decomposition** $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree of bags $B_i \subseteq V$ such that:

1. Every node of G is contained in a bag
2. Every edge of G is contained in a bag
3. Every node of G appears in a contiguous subtree of $\mathrm{Tree}(G)$.

G



$\mathrm{Tree}(G)$



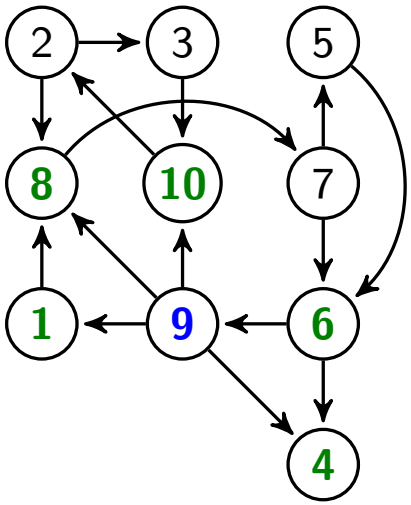
Tree Decomposition

Definition (Tree decomposition)

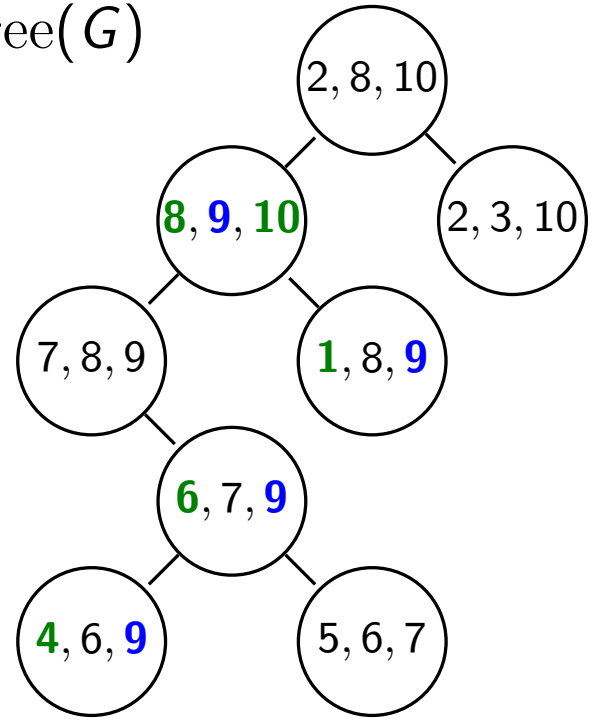
Given a graph $G = (V, E)$, a **tree-decomposition** $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree of bags $B_i \subseteq V$ such that:

1. Every node of G is contained in a bag
2. Every edge of G is contained in a bag
3. Every node of G appears in a contiguous subtree of $\mathrm{Tree}(G)$.

G



$\mathrm{Tree}(G)$



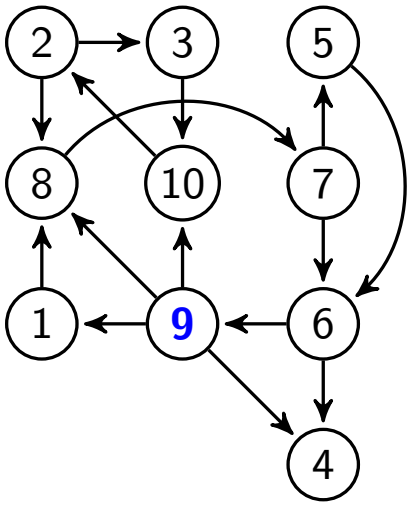
Tree Decomposition

Definition (Tree decomposition)

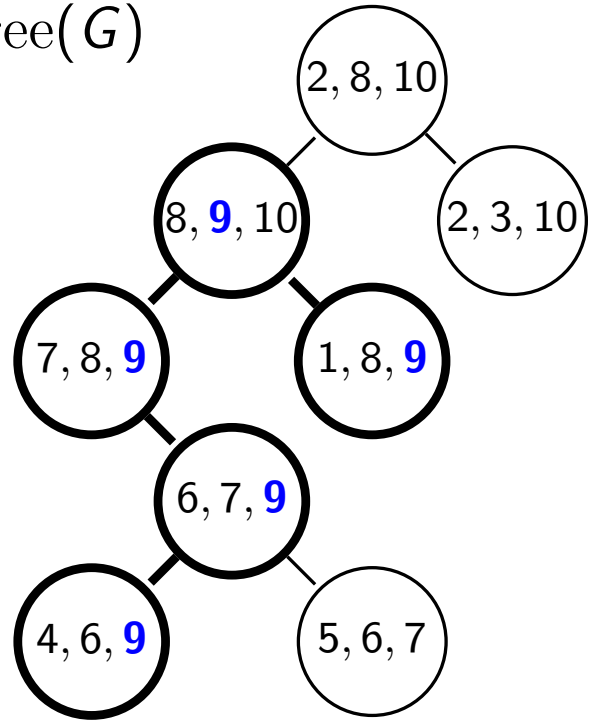
Given a graph $G = (V, E)$, a **tree-decomposition** $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree of bags $B_i \subseteq V$ such that:

1. Every node of G is contained in a bag
2. Every edge of G is contained in a bag
3. Every node of G appears in a contiguous subtree of $\mathrm{Tree}(G)$.

G



$\mathrm{Tree}(G)$

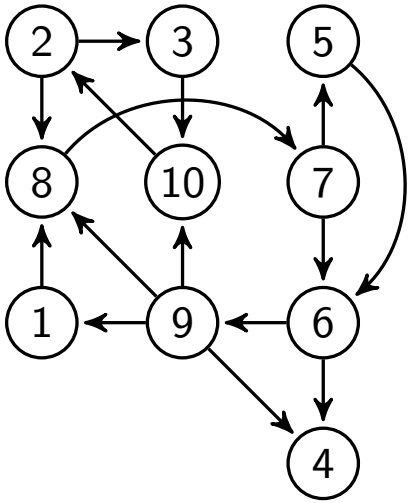


Treewidth

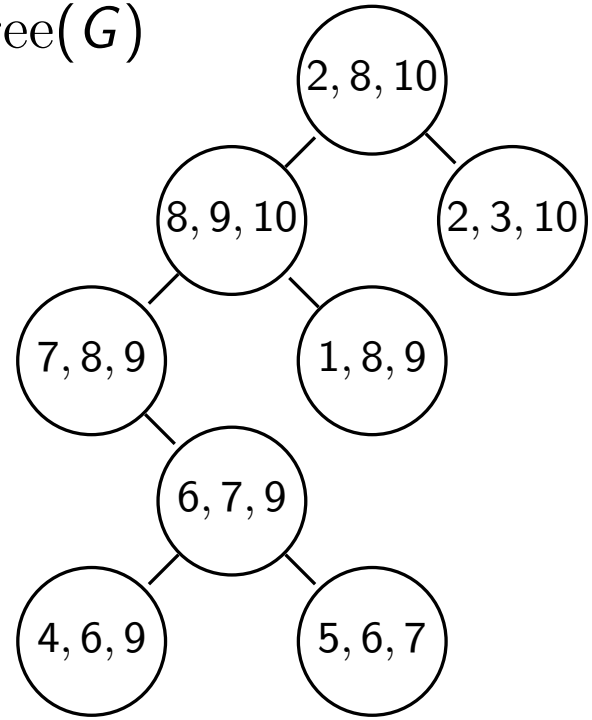
Definition

G has **constant treewidth** if every bag of $\mathrm{Tree}(G)$ has constant size.

G



$\mathrm{Tree}(G)$



Treewidth of Structured Programs

Theorem (Treewidth of structured programs)

Control-flow graphs of goto-free programs have constant treewidth.

Theorem (Tree decomposition)

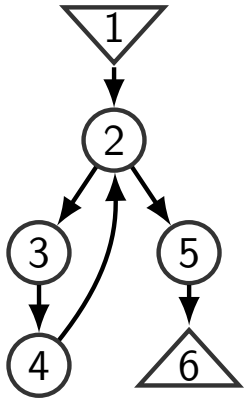
For constant treewidth graphs, $\mathrm{Tree}(G)$ *can be constructed in* $O(n)$ *time.*

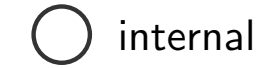
Example: RSM & Tree Decompositions

Method 1: dot_vector

Input: $x, y \in \mathbb{R}^n$
Output: The dot product $x^\top y$

```
1 result ← 0
2 for i ← 1 to n do
3     z ← x[i] · y[i]
4     result ← result + z
5 end
6 return result
```

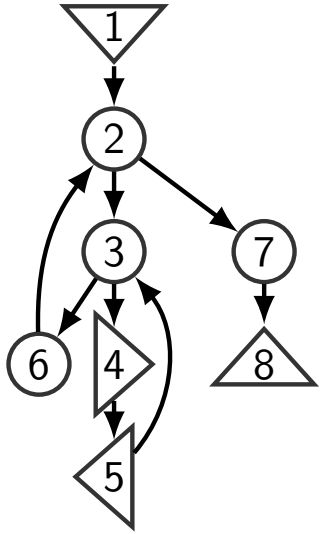




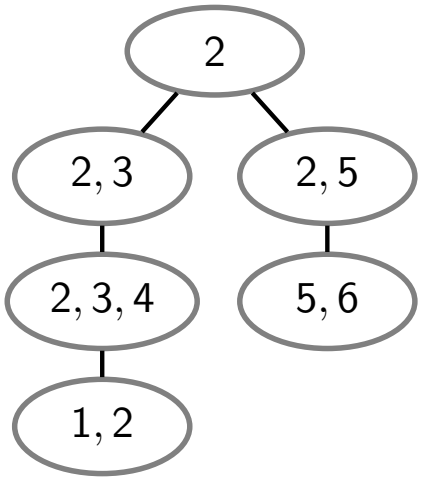
Method 2: dot_matrix

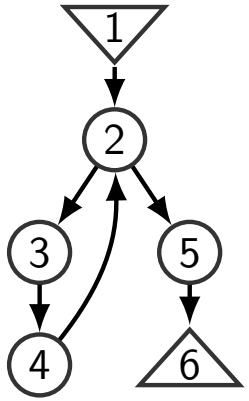
Input: $A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{k \times m}$
Output: The dot product $A \times B$

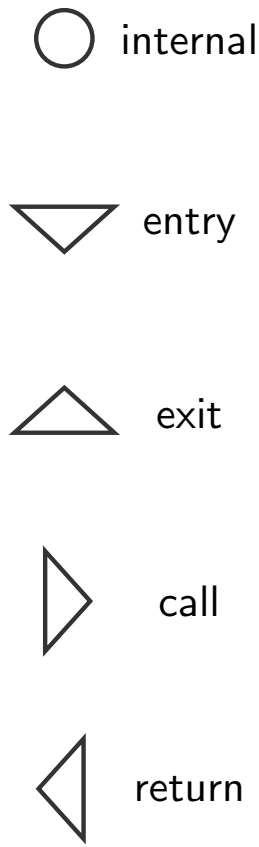
```
1 C ← zero matrix of size n × m
2 for i ← 1 to n do
3     for j ← 1 to m do
4         Call dot_vector(A[i, :], B[:, j])
5         C[i, j] ← the value returned by the
          call of line 4
6     end
7 end
8 return C
```

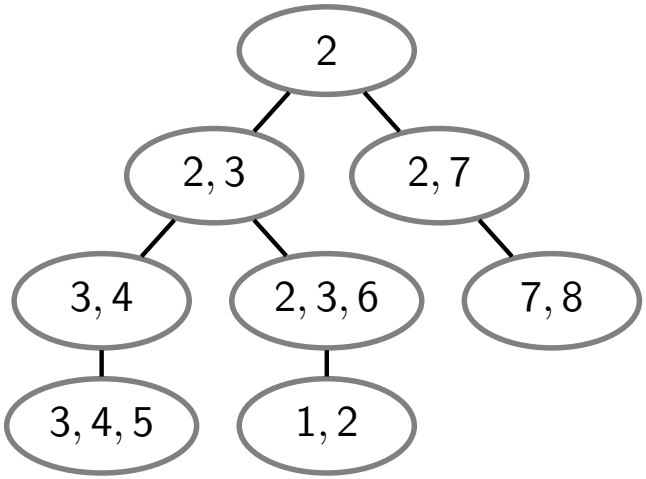


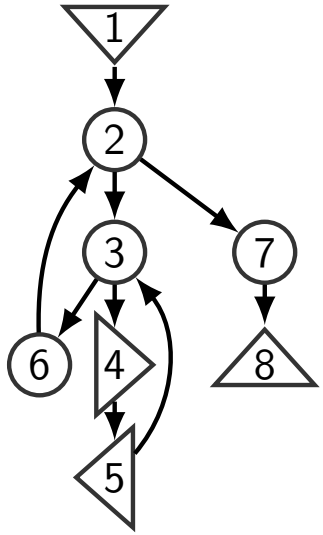
Example: RSM & Tree Decompositions











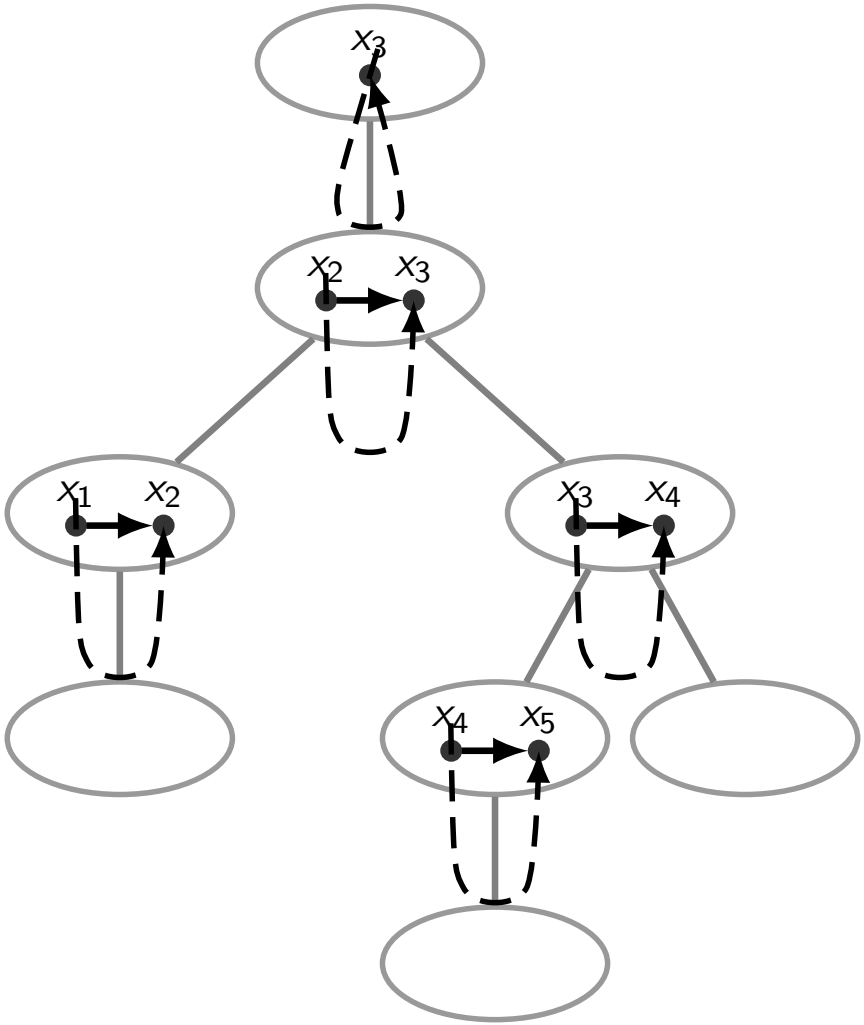
Algorithms for Preprocess, Update & Query on $\mathrm{Tree}(G)$

Focus on graph $G = (V, E)$ and a tree-decomposition $\mathrm{Tree}(G)$

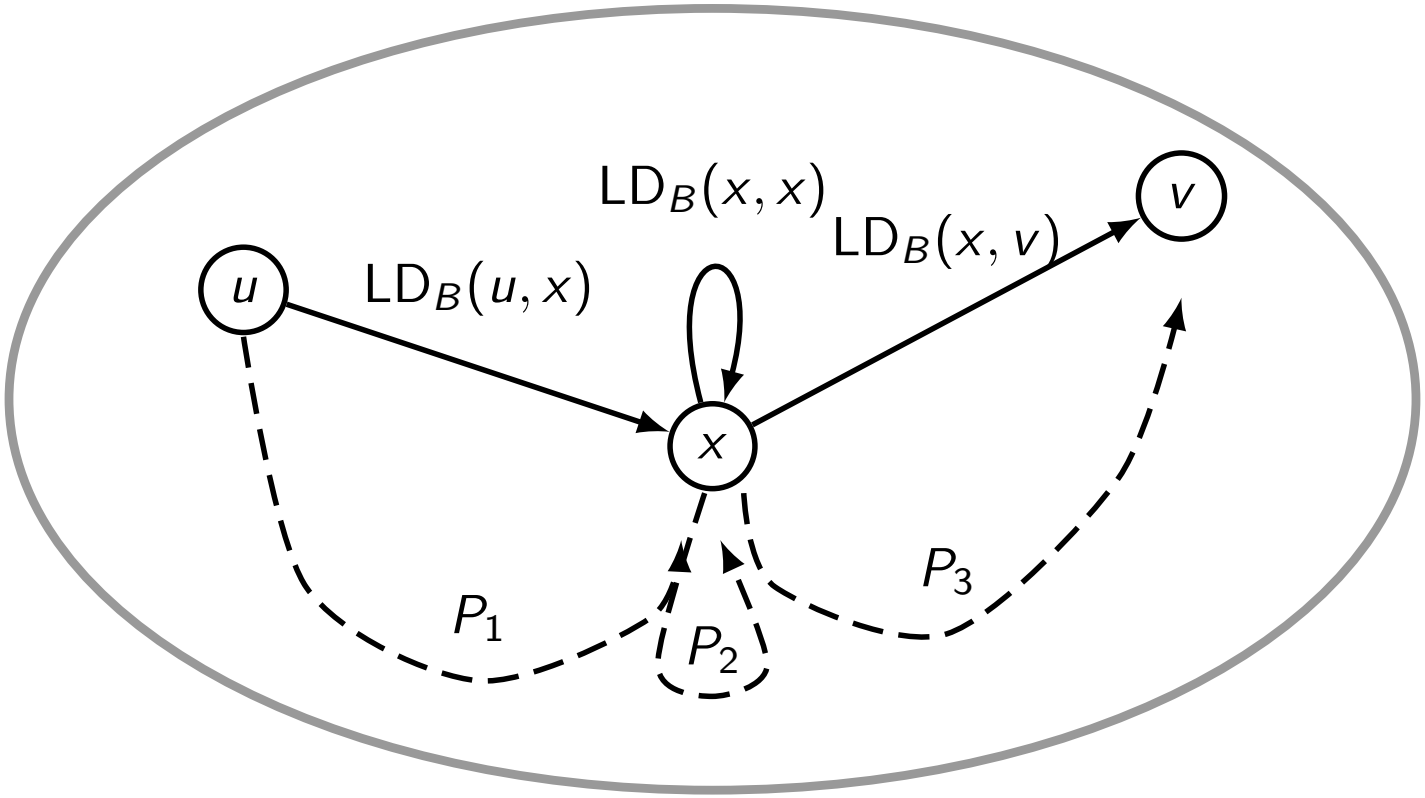
- Preprocess $\mathrm{Tree}(G)$ in $O(n)$ time
- Query for the distance $d(u, v)$ in $O(\log n)$ time
- Update the weight of an edge of G in $O(\log n)$ time

Local Distances

- **Local distances** in a bag B: $\mathrm{LD}_B(x_i, x_j) = \bigoplus_{P: x_i \to x_j} \otimes(P)$
  - where P is a U-shaped path



Path Shortening

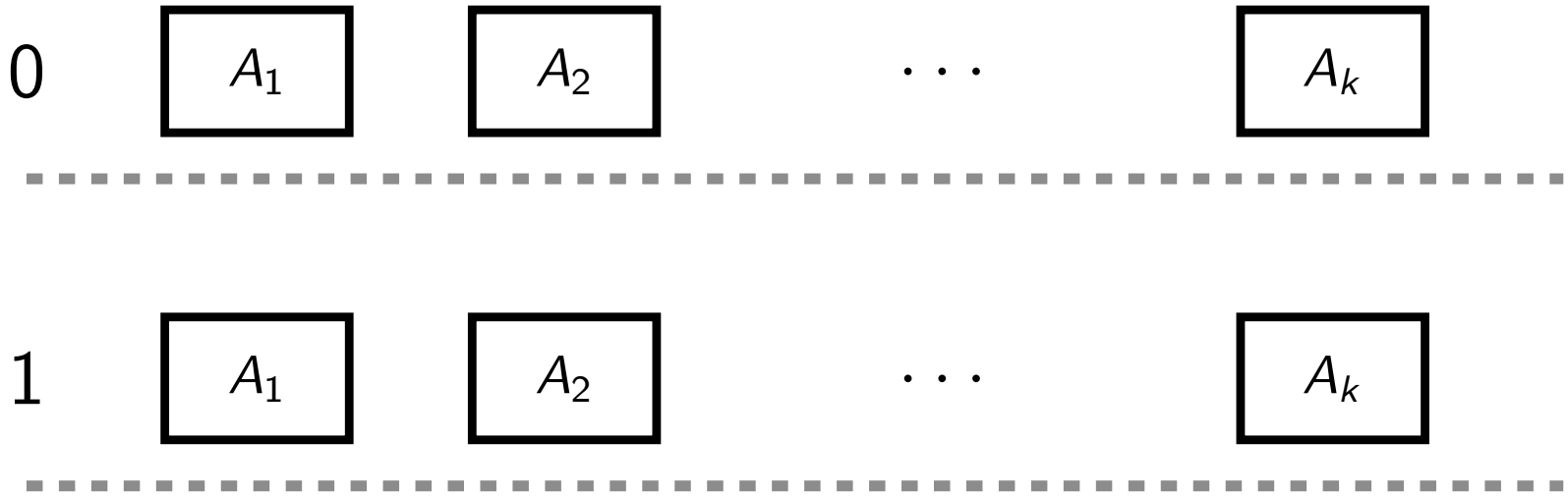


$$LD_B(u,v) = LD_B(u,v) \bigoplus LD_B(u,x) \bigotimes (LD_B(x,x))^* \bigotimes LD_B(x,v)\}$$

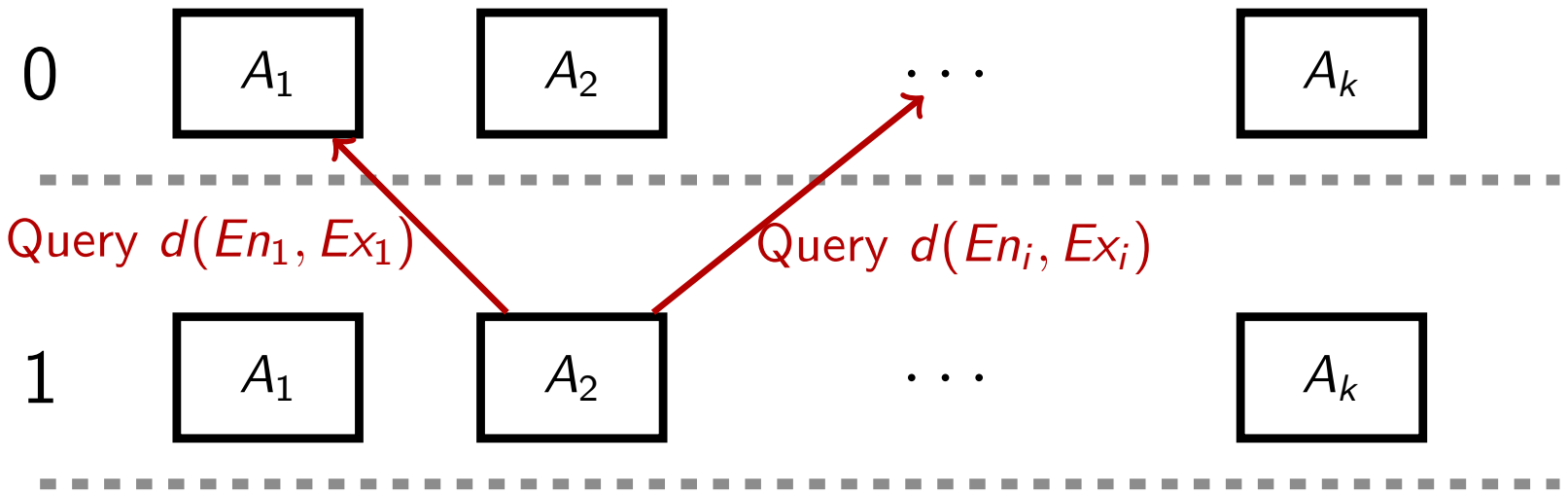
Preprocessing of an RSM

Preprocessing of an RSM

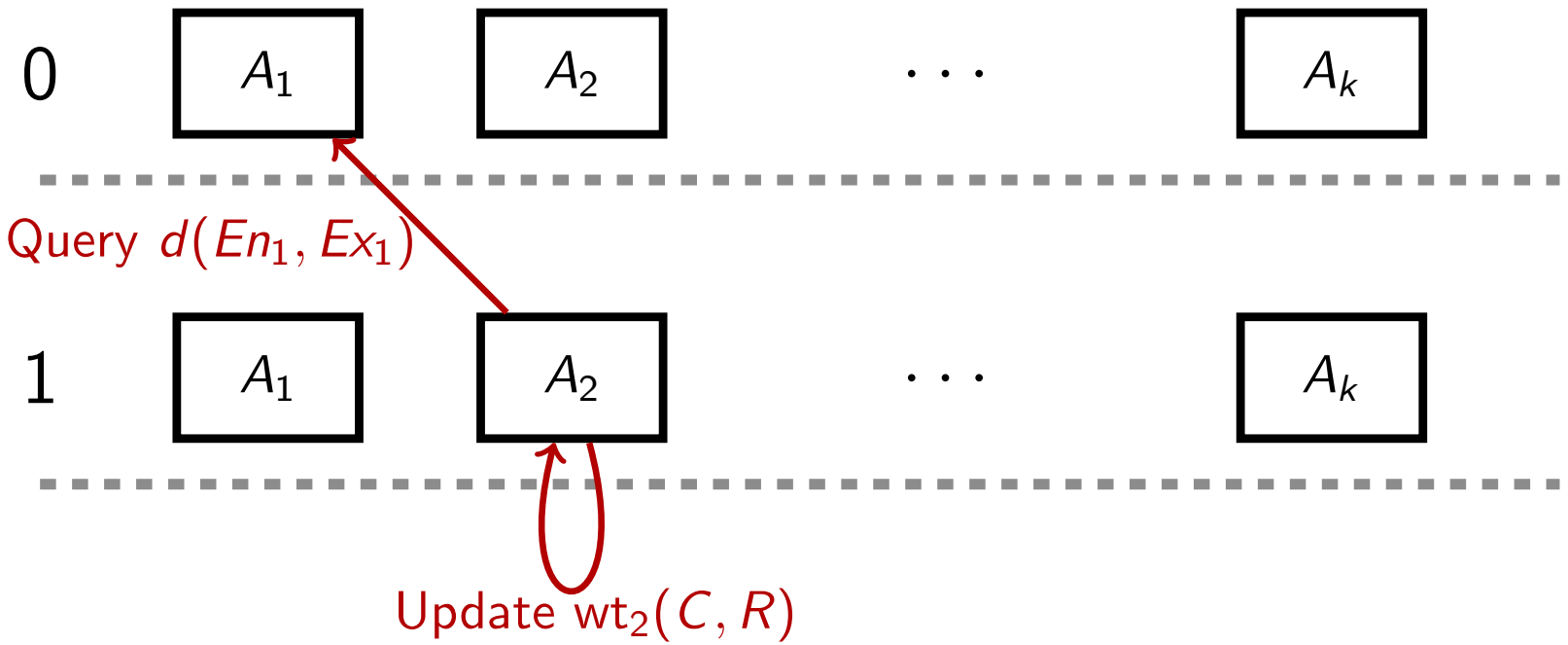
Preprocessing of an RSM



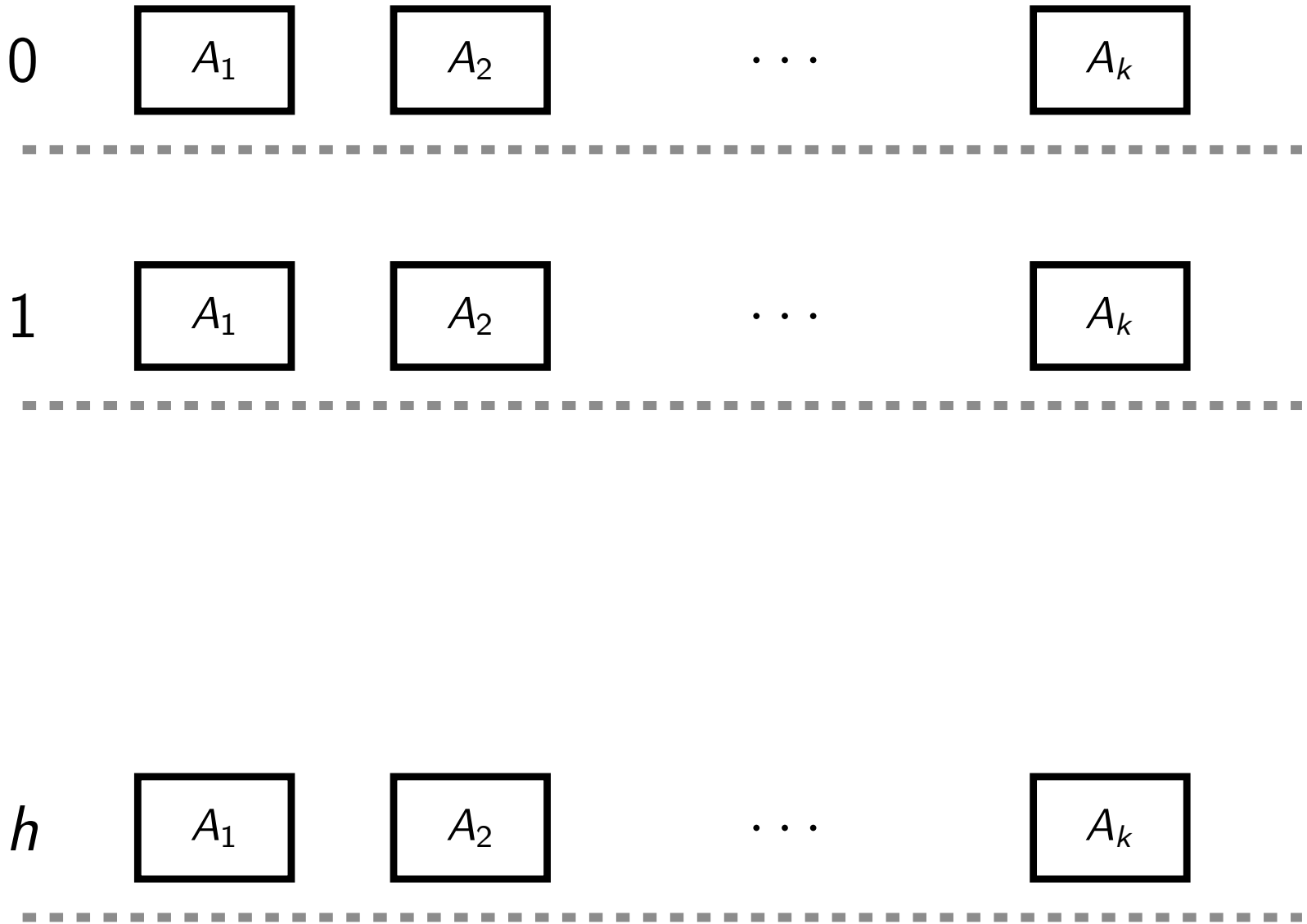
Preprocessing of an RSM



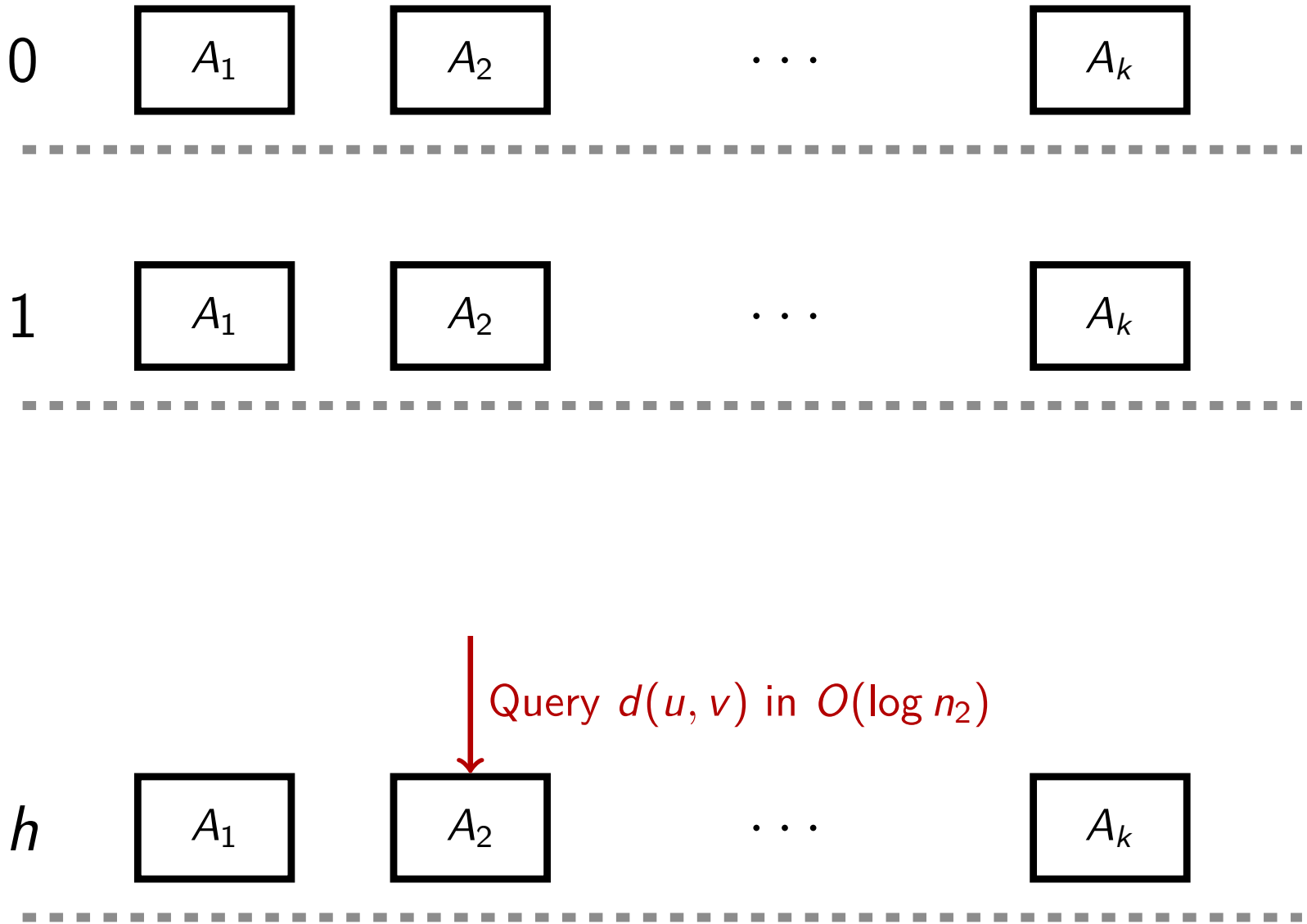
Preprocessing of an RSM



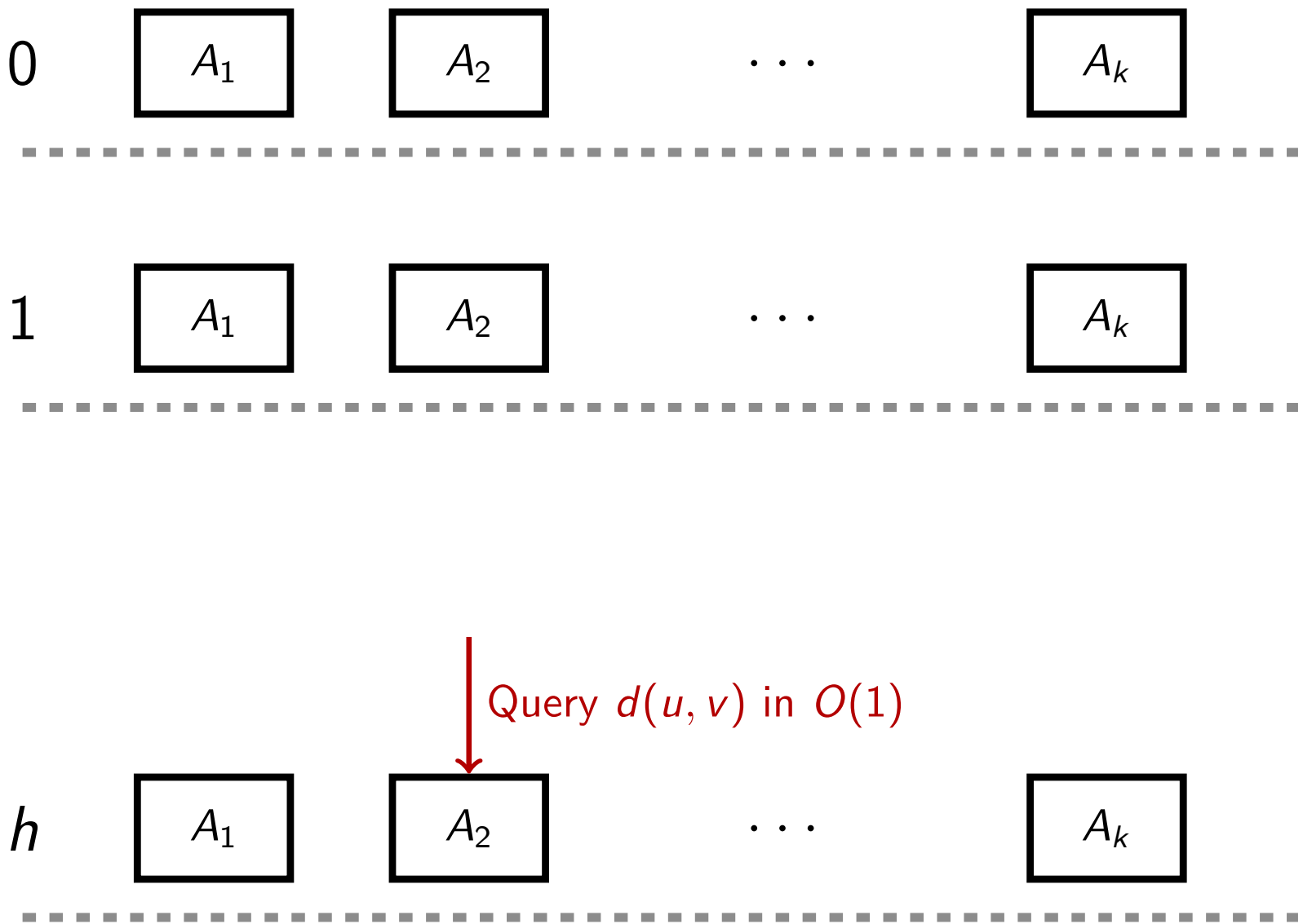
Preprocessing of an RSM



Preprocessing of an RSM



Preprocessing of an RSM



Results

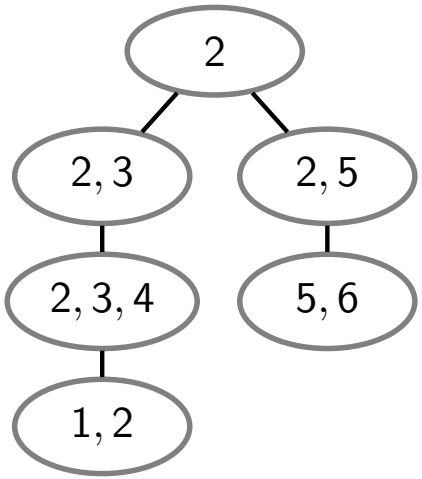
Reachability				
	Preprocess time	Space	Single-source query	Pair query
Complete preprocessing	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
No preprocessing	-	$O(n)$	$O(n)$	$O(n)$
Our Results	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$
	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$

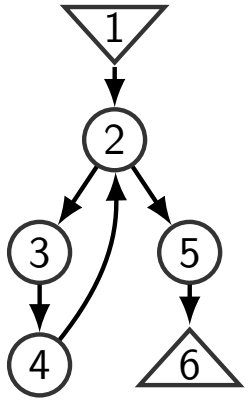
Shortest Path				
	Preprocess time	Space	Single-source query	Pair query
Complete preprocessing	$O(n^5)$	$O(n^2)$	$O(n)$	$O(1)$
No preprocessing	-	$O(n)$	$O(n^4)$	$O(n^4)$
Our Results	$\mathbf{O(n^2 \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(n)}$	$\mathbf{O(\log n)}$
	$\mathbf{O(n^2 \cdot \log n)}$	$\mathbf{O(n \cdot \log n)}$	$\mathbf{O(n)}$	$\mathbf{O(1)}$

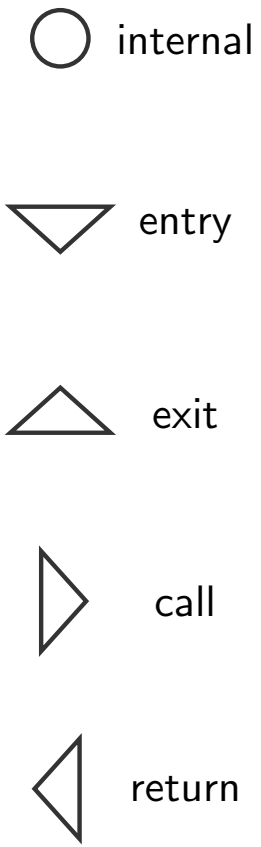
Results

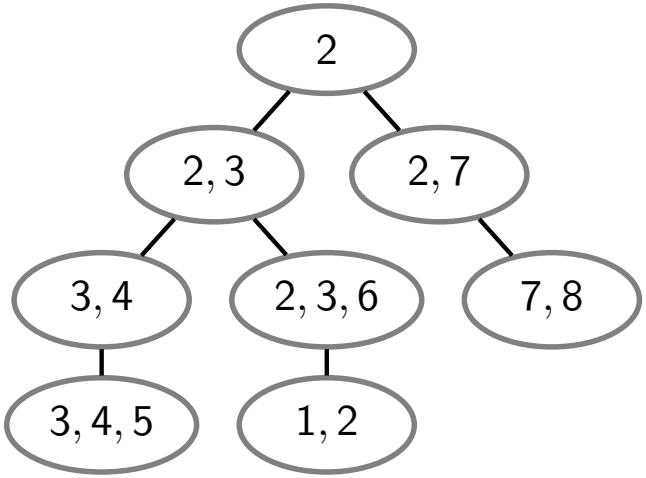
- Algorithms apply to complete semirings
- Improved bounds for IFDS
- Byproduct: Dynamic data-structure

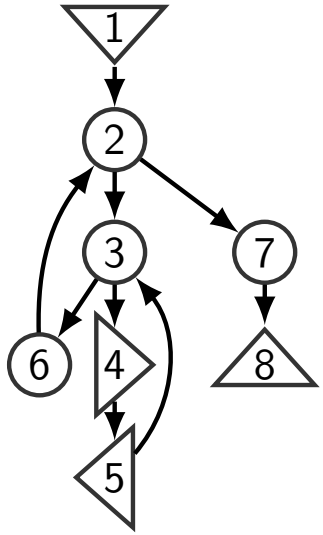
Illustration of Bounded Stack Height Preprocessing



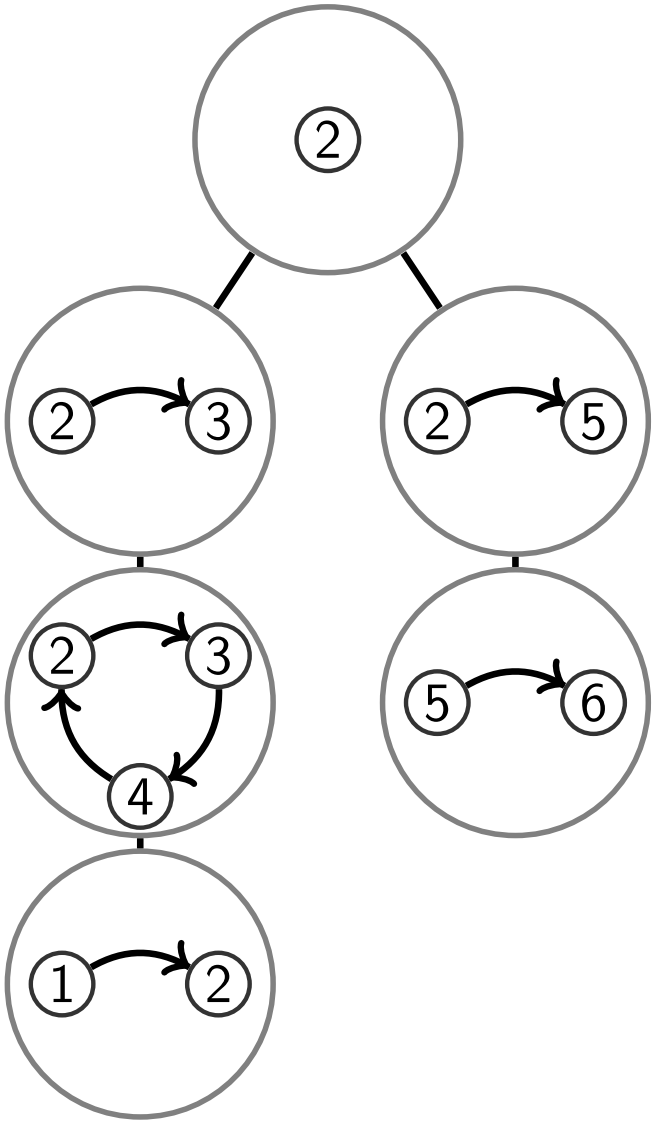


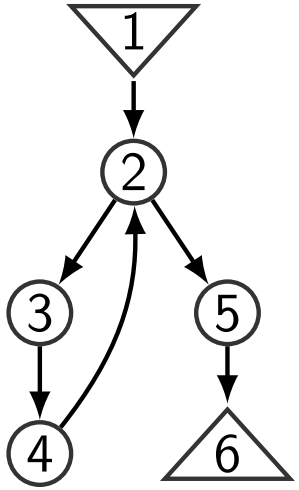




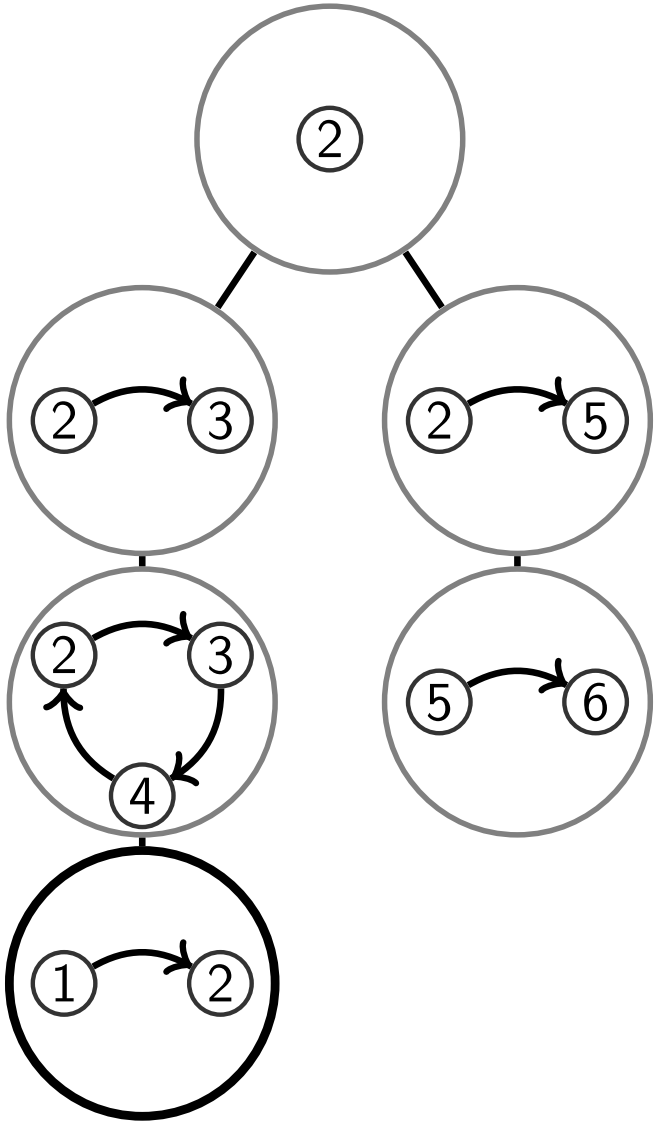


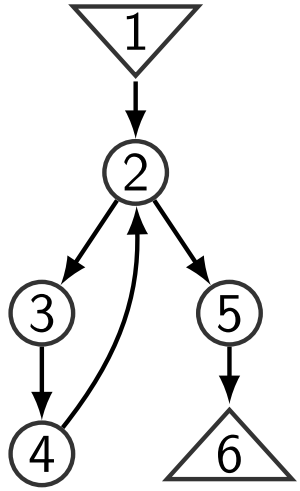
Example: Preprocessing $\mathrm{Tree}(G_1)$



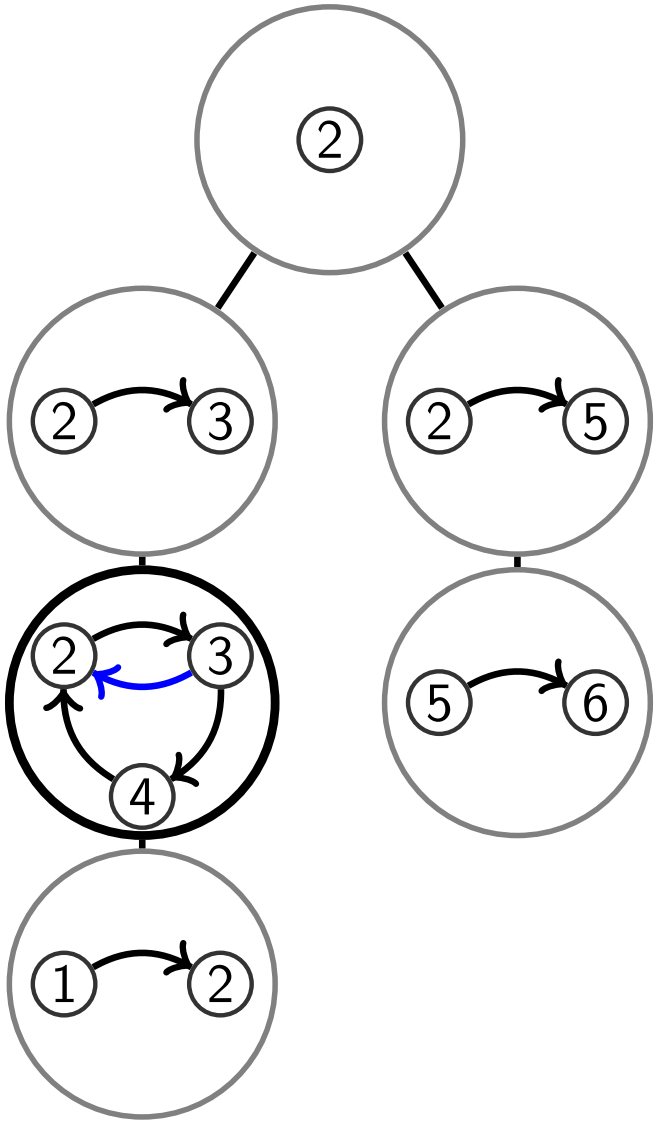


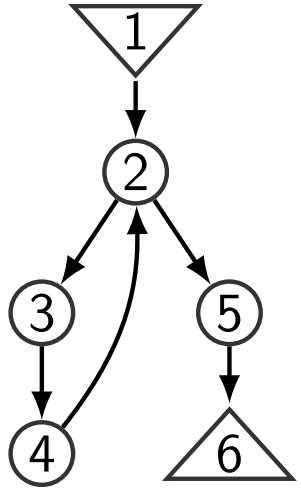
Example: Preprocessing $\mathrm{Tree}(G_1)$



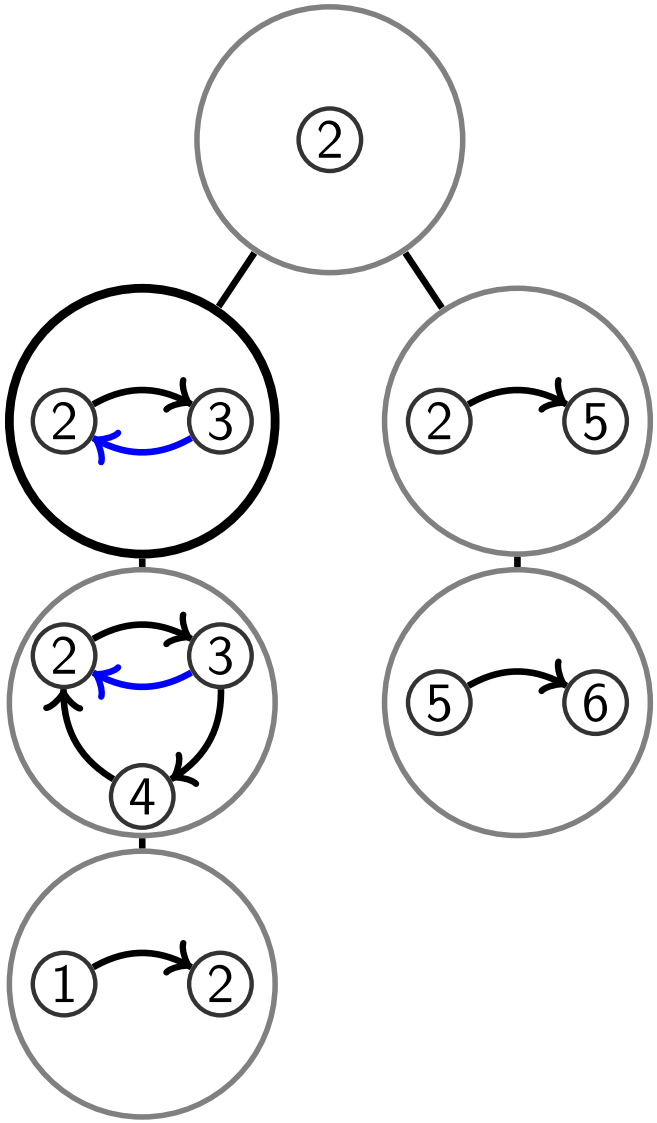


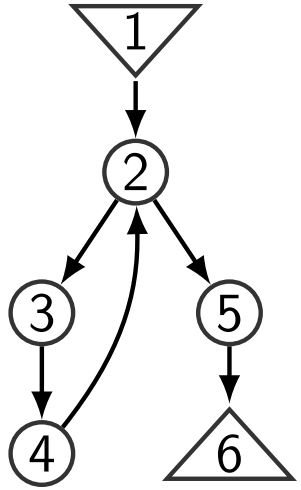
Example: Preprocessing $\mathrm{Tree}(G_1)$



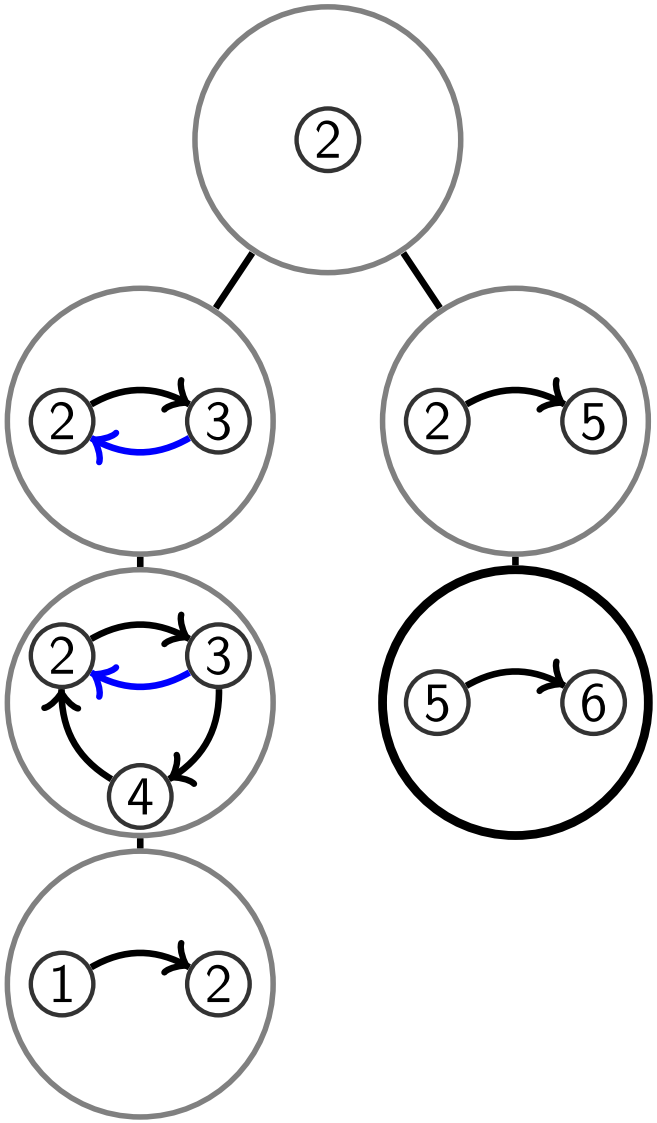


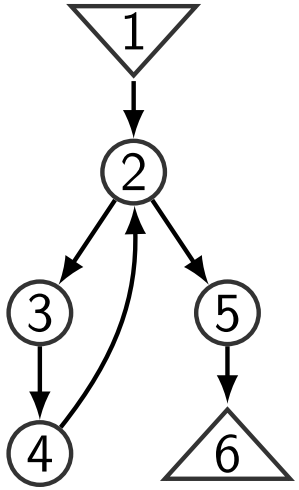
Example: Preprocessing $\mathrm{Tree}(G_1)$



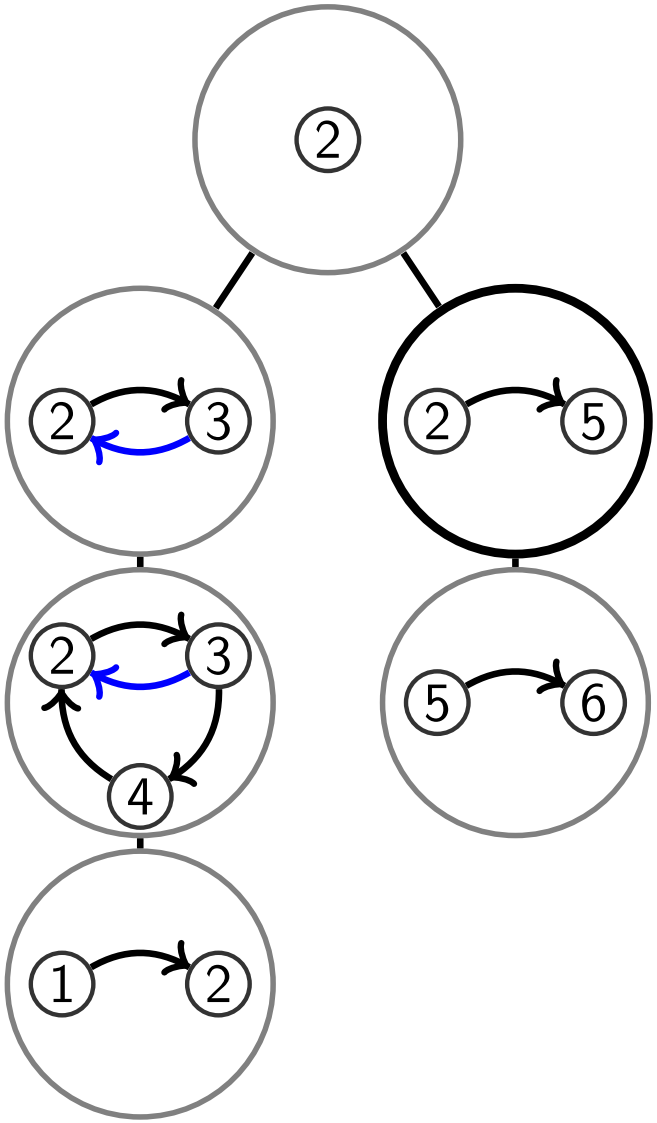


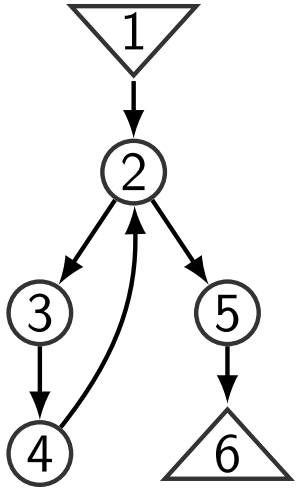
Example: Preprocessing $\mathrm{Tree}(G_1)$



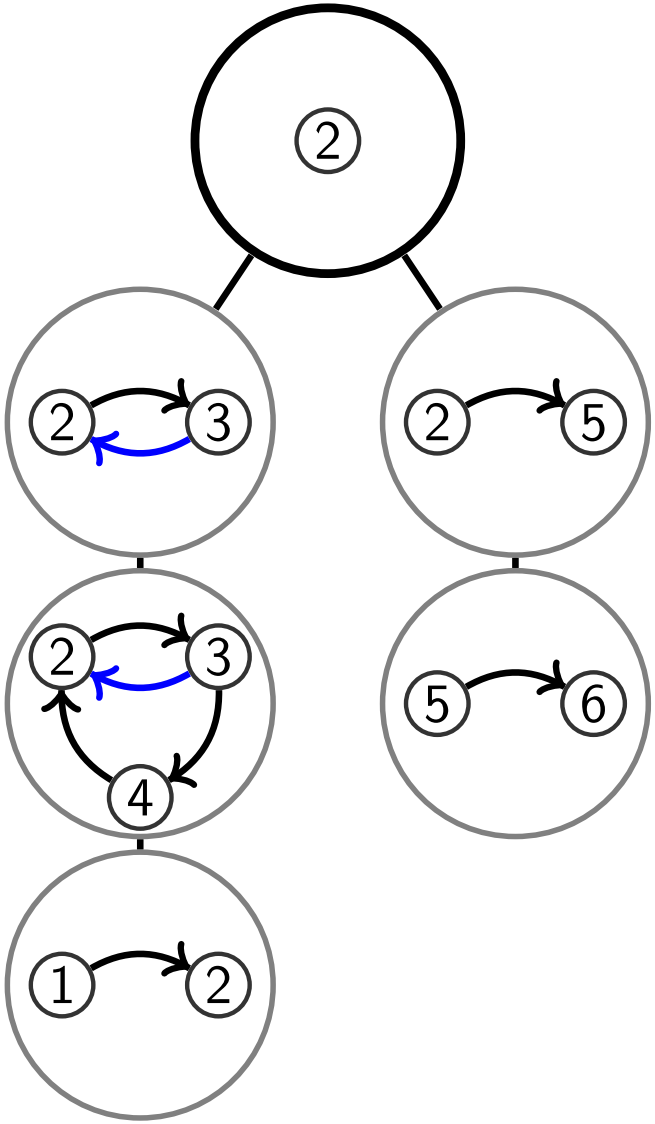


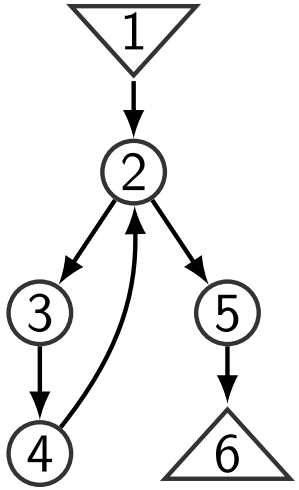
Example: Preprocessing $\mathrm{Tree}(G_1)$



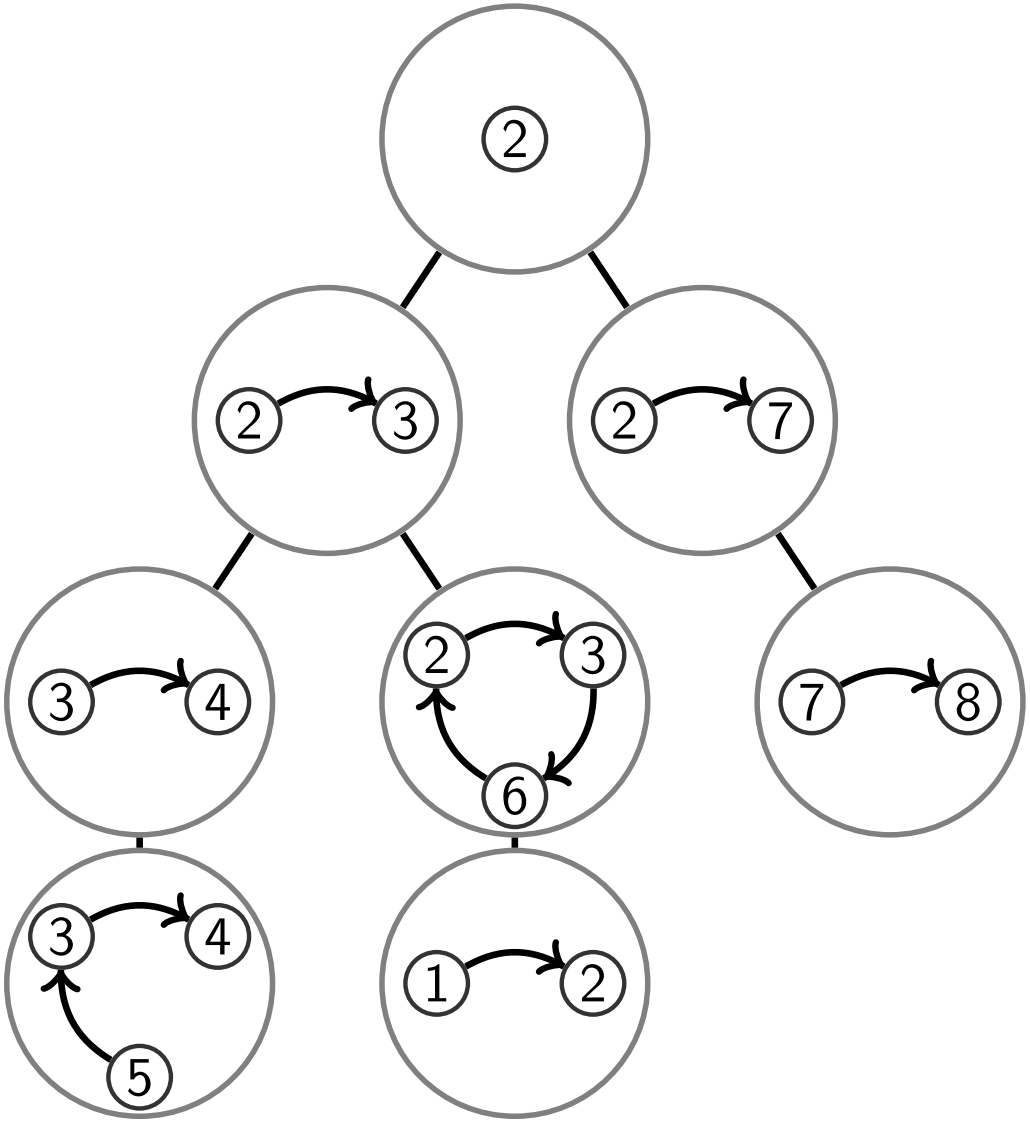


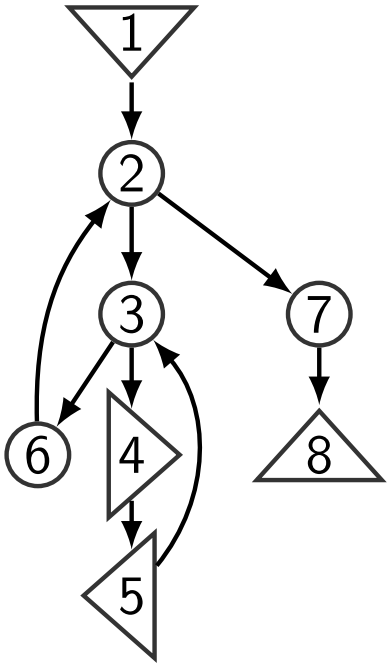
Example: Preprocessing $\mathrm{Tree}(G_1)$



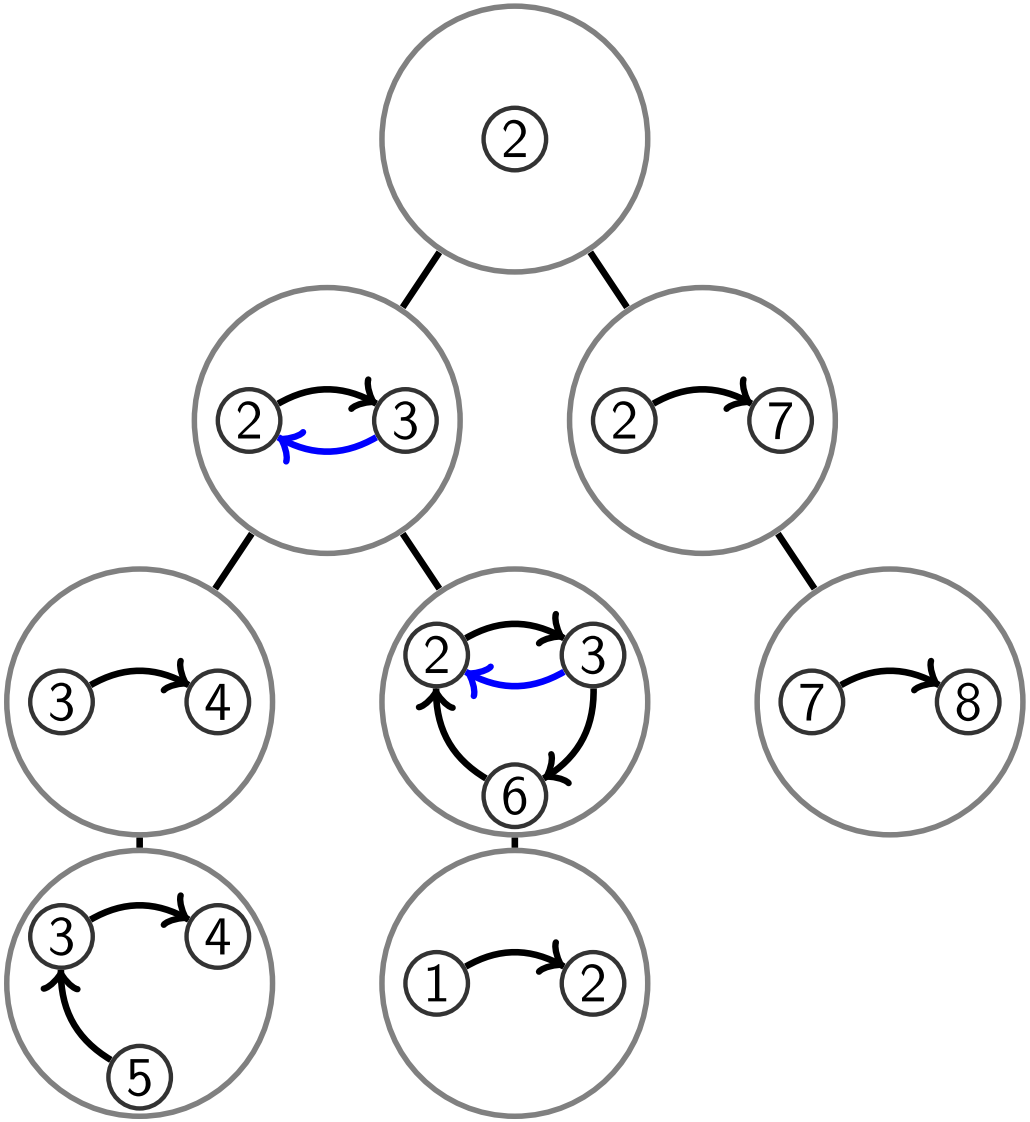


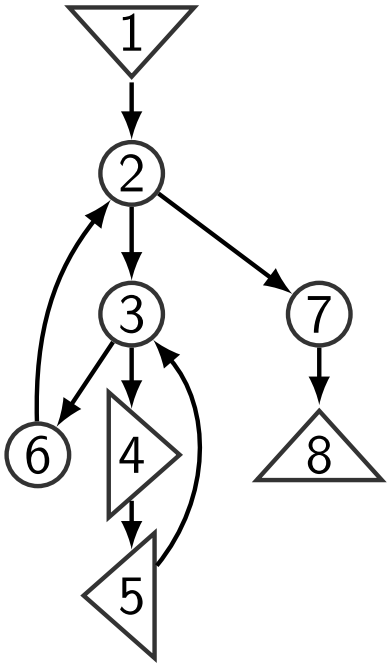
Example: Preprocessing $\mathrm{Tree}(G_2)$



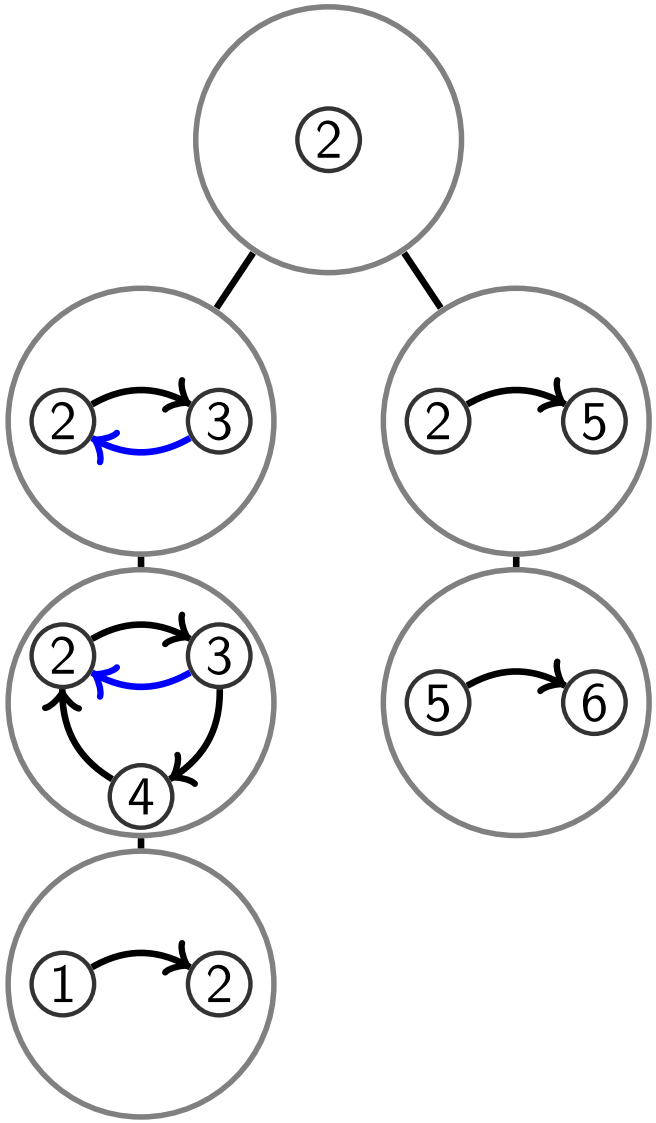


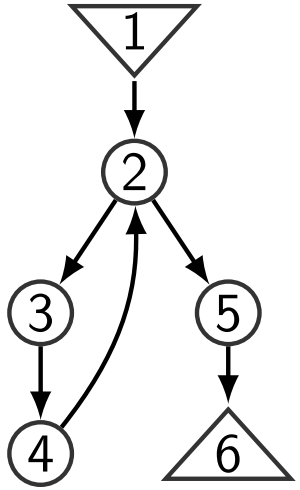
Example: Preprocessing $\mathrm{Tree}(G_2)$



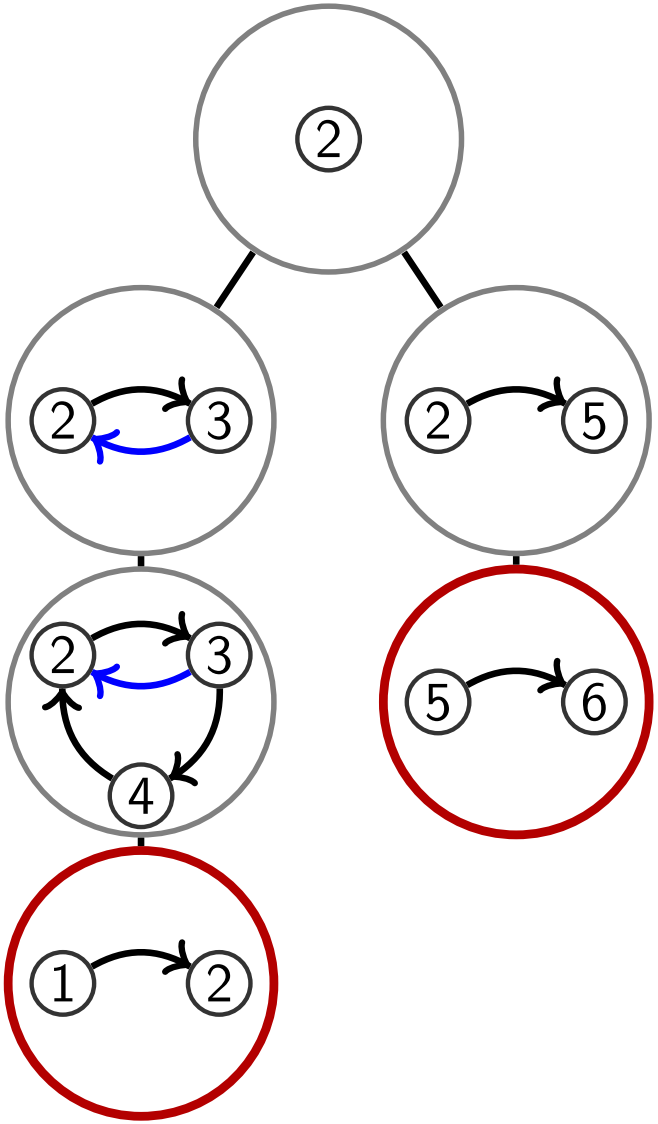


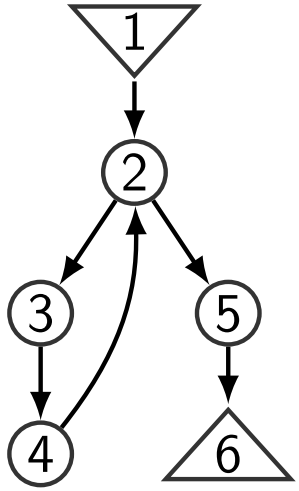
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$



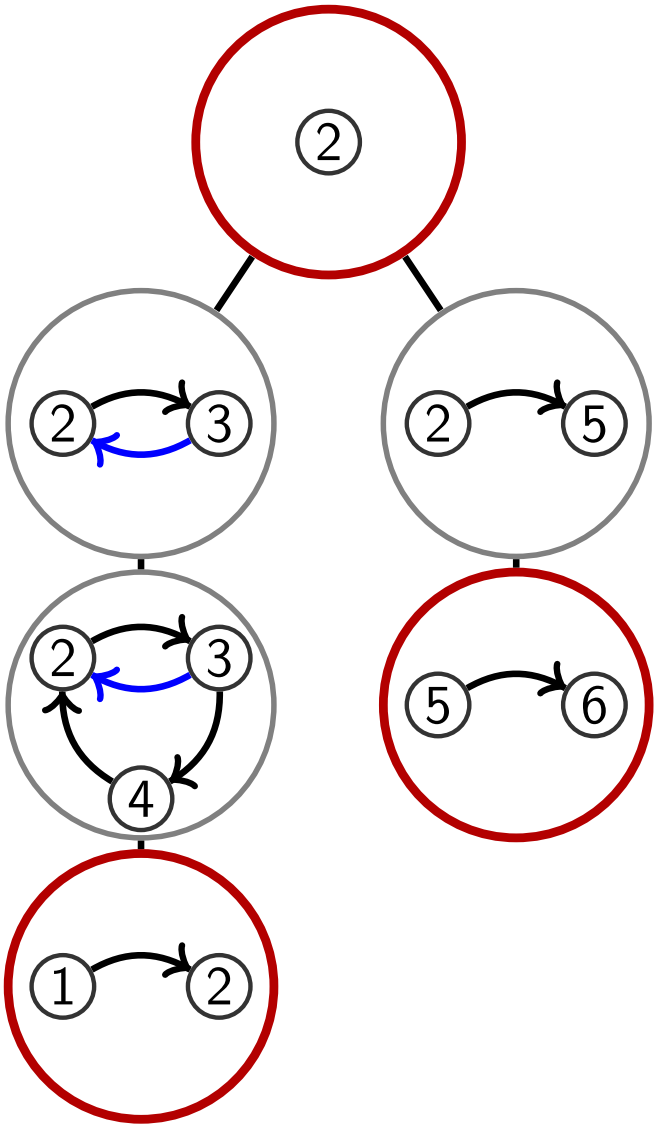


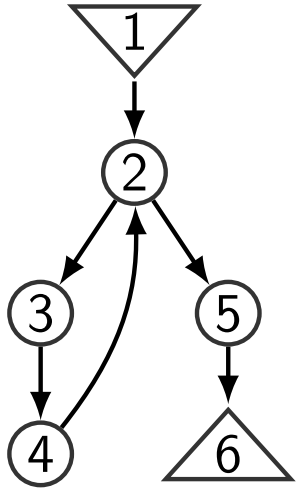
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$



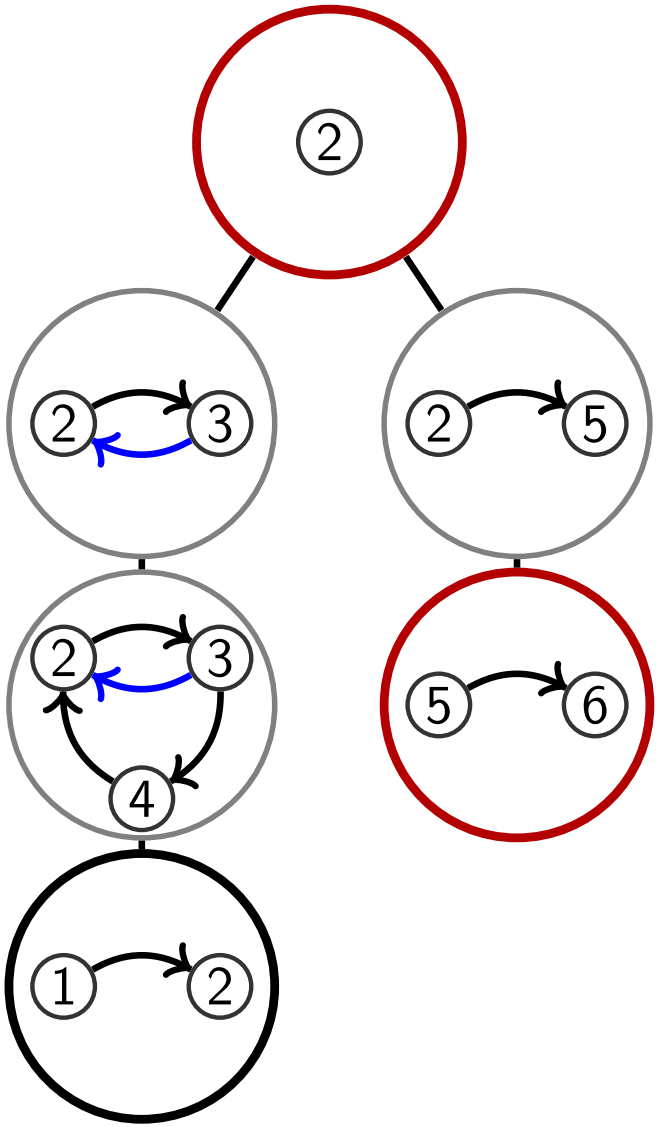


Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$

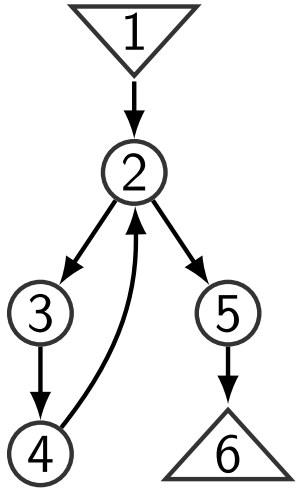




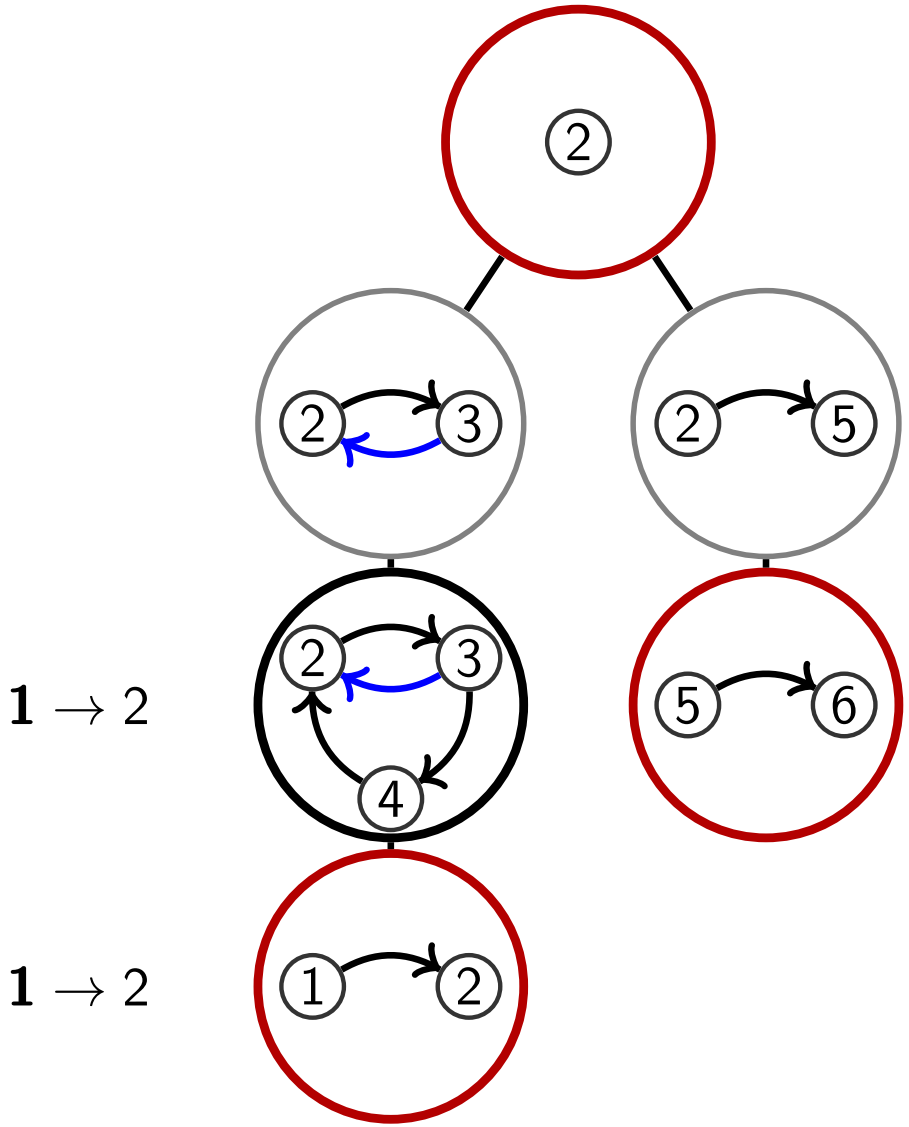
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$

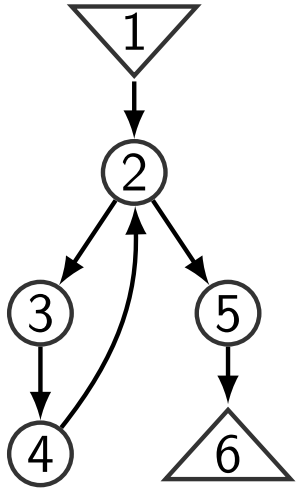


$\mathbf{1} \to 2$

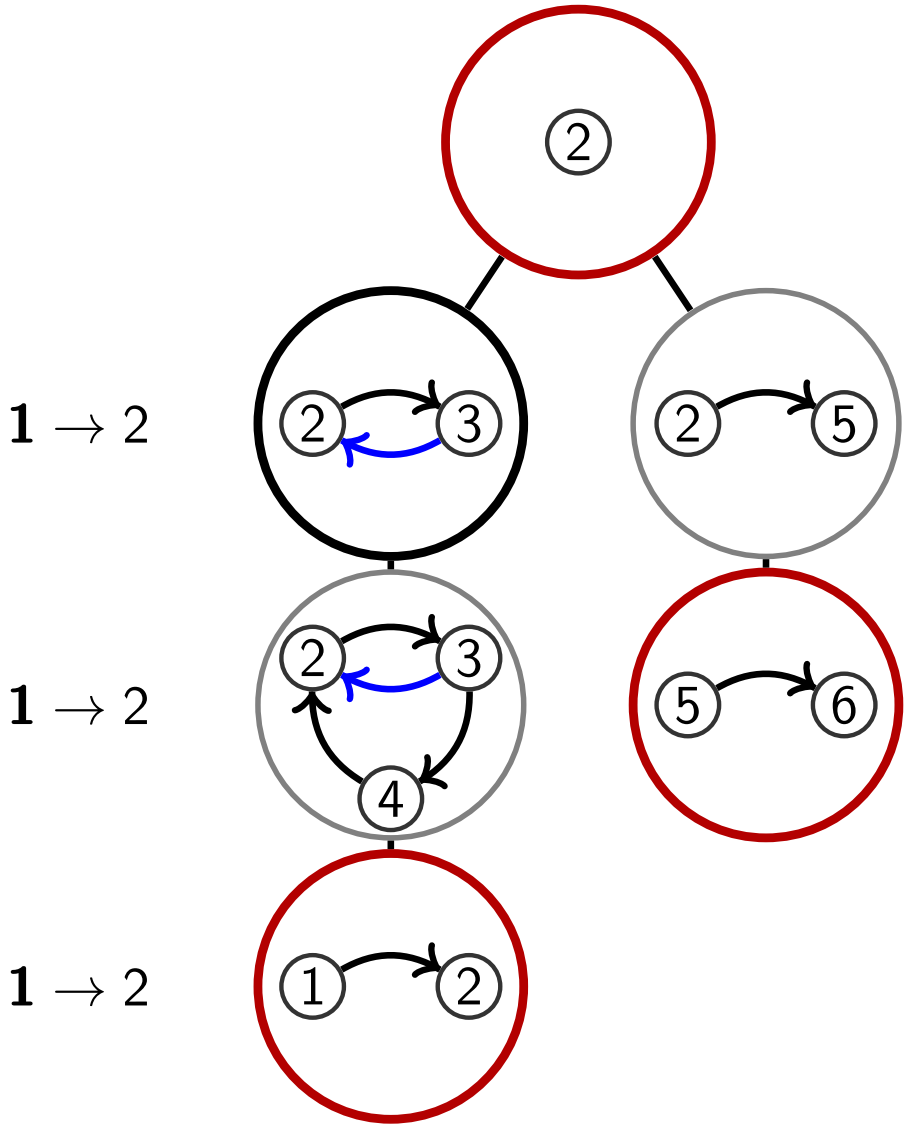


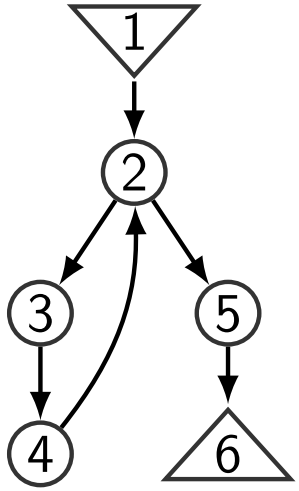
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$



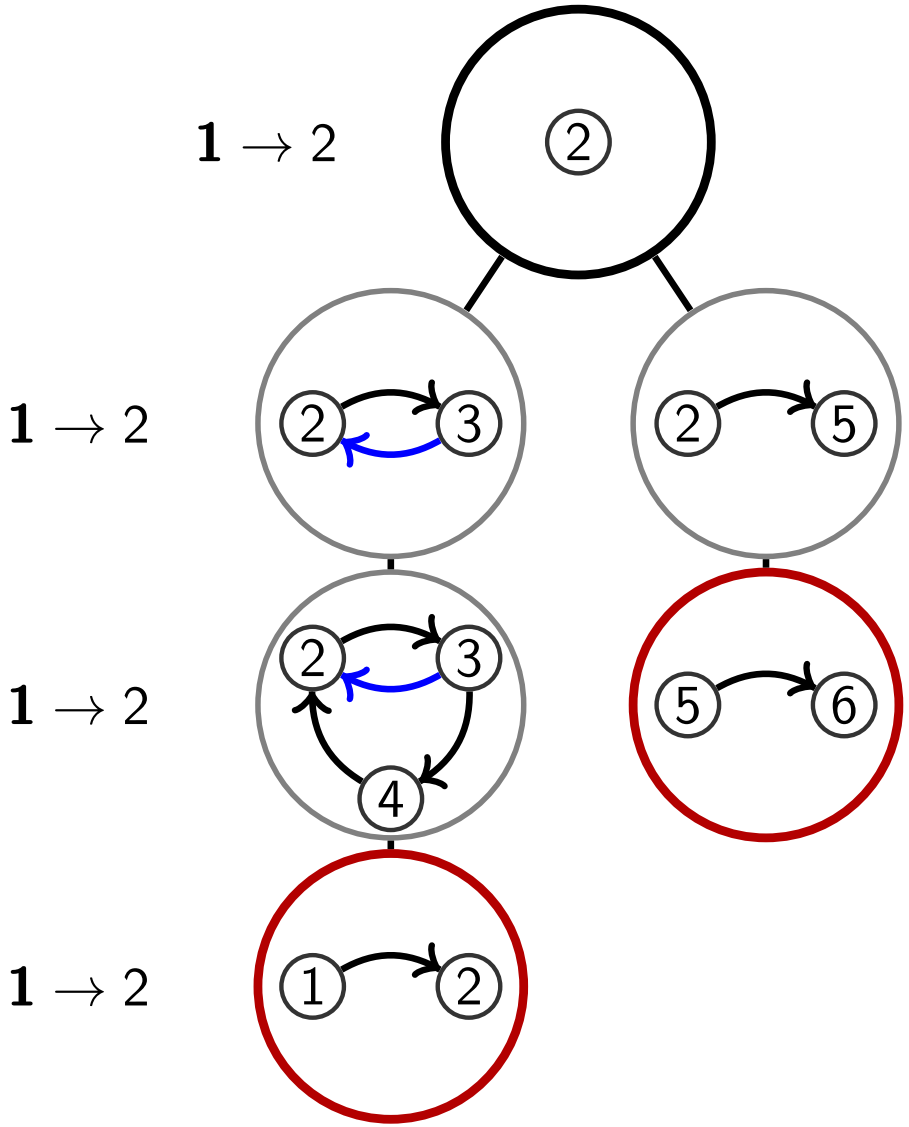


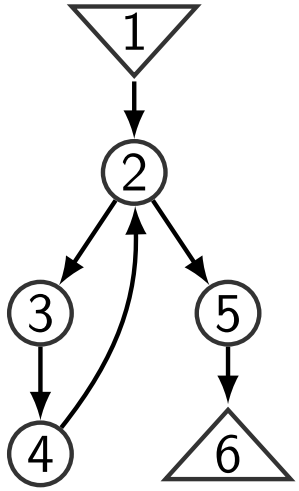
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$



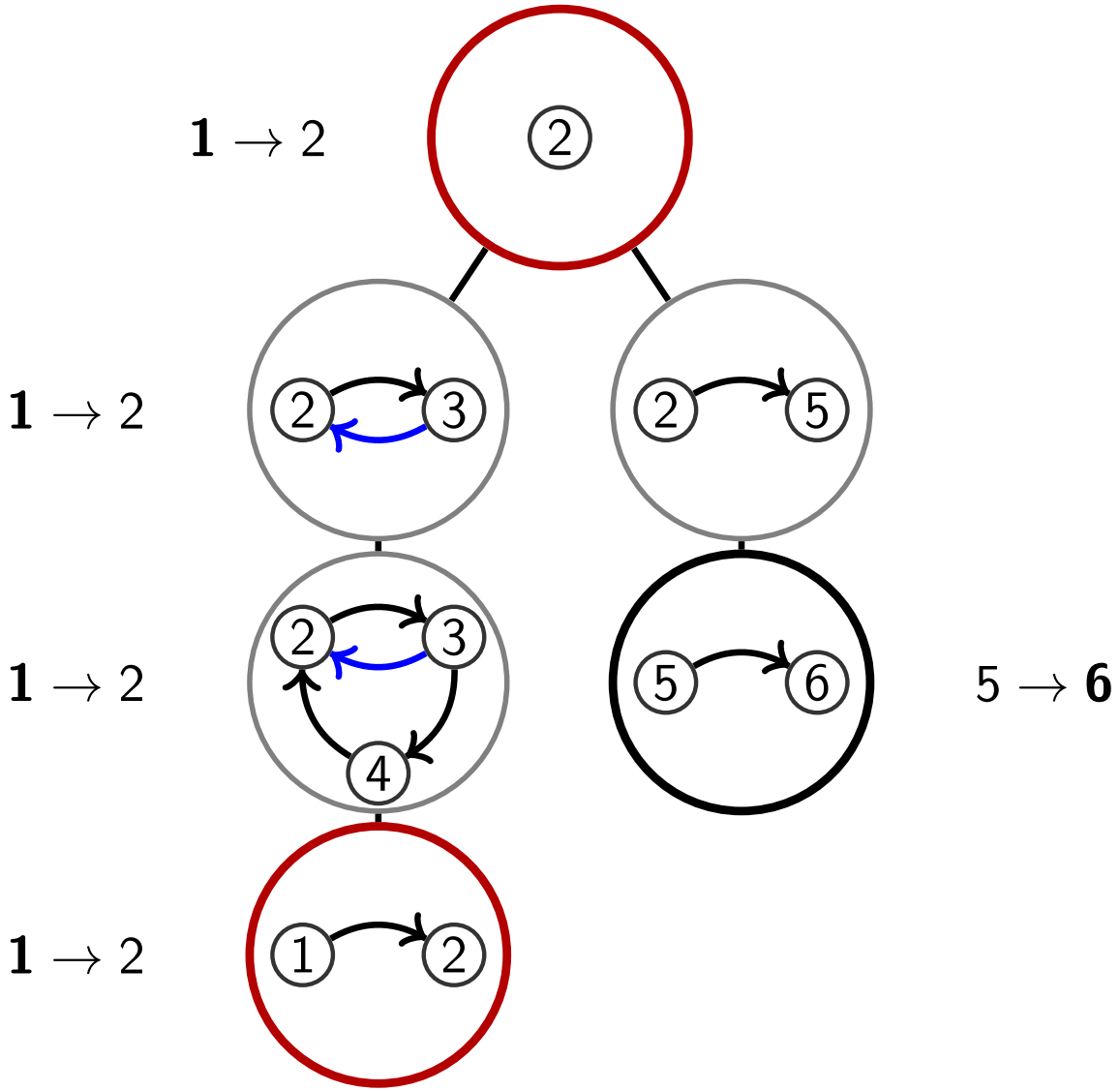


Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$

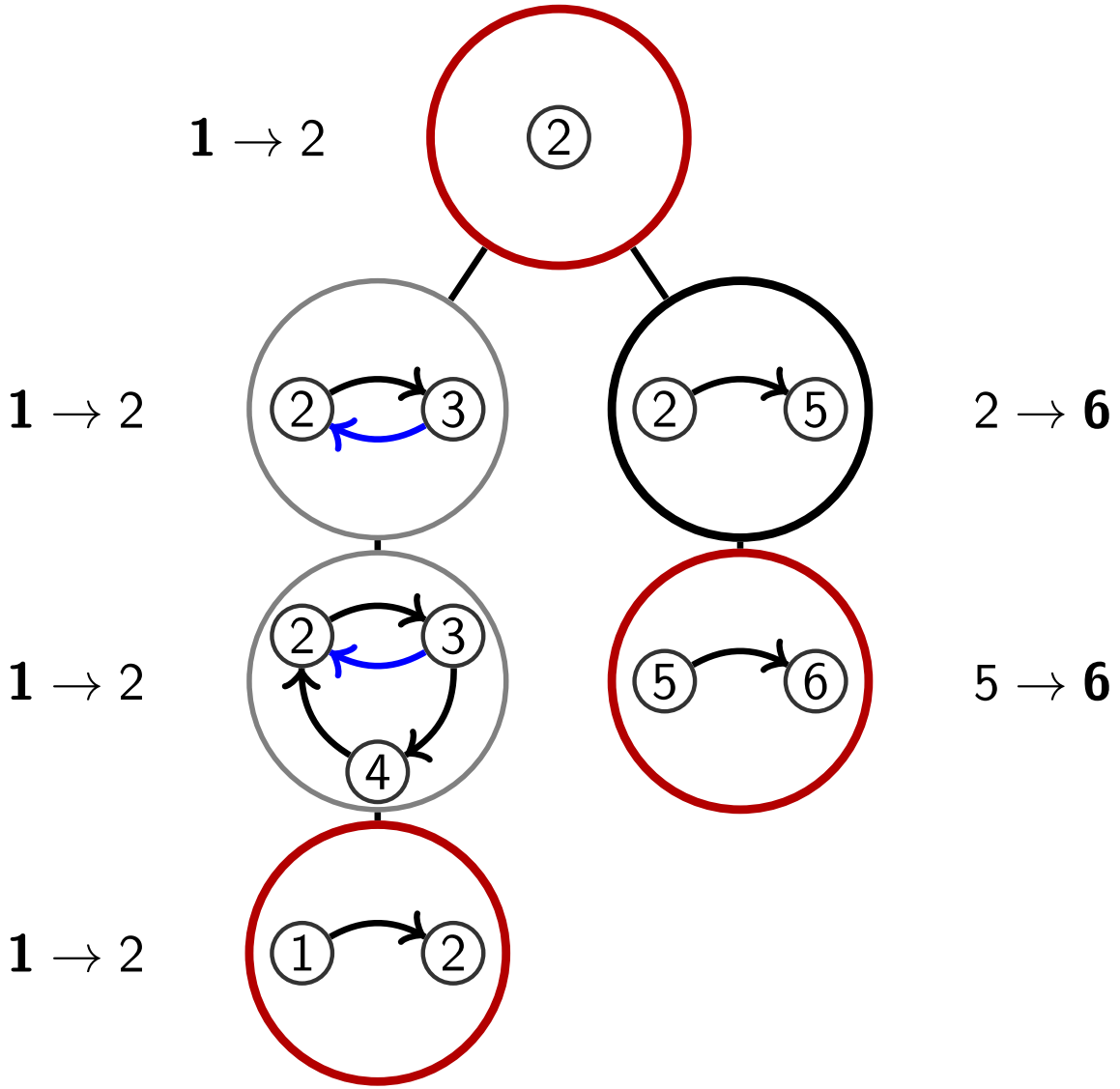


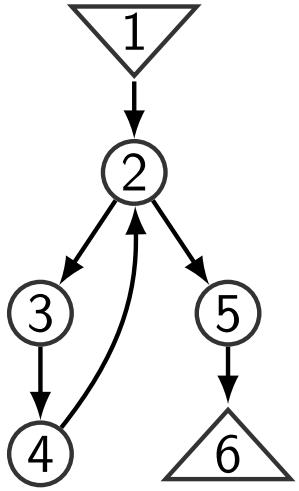


Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$

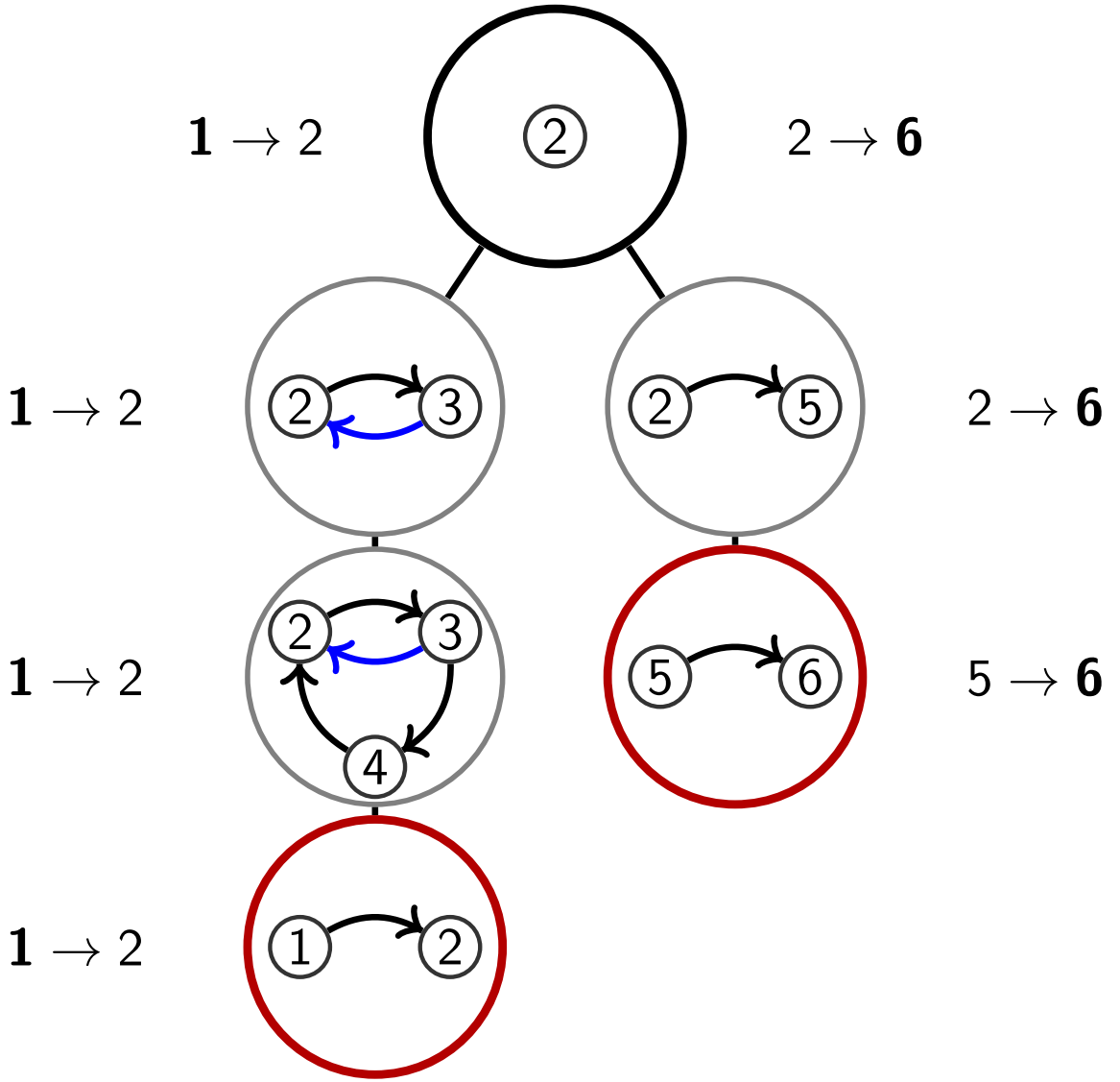


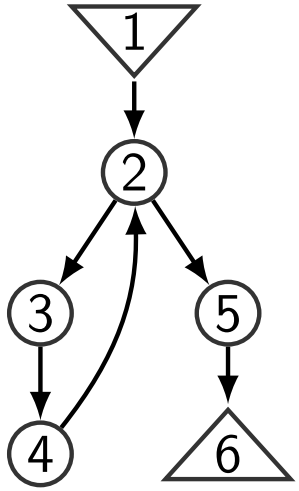
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$



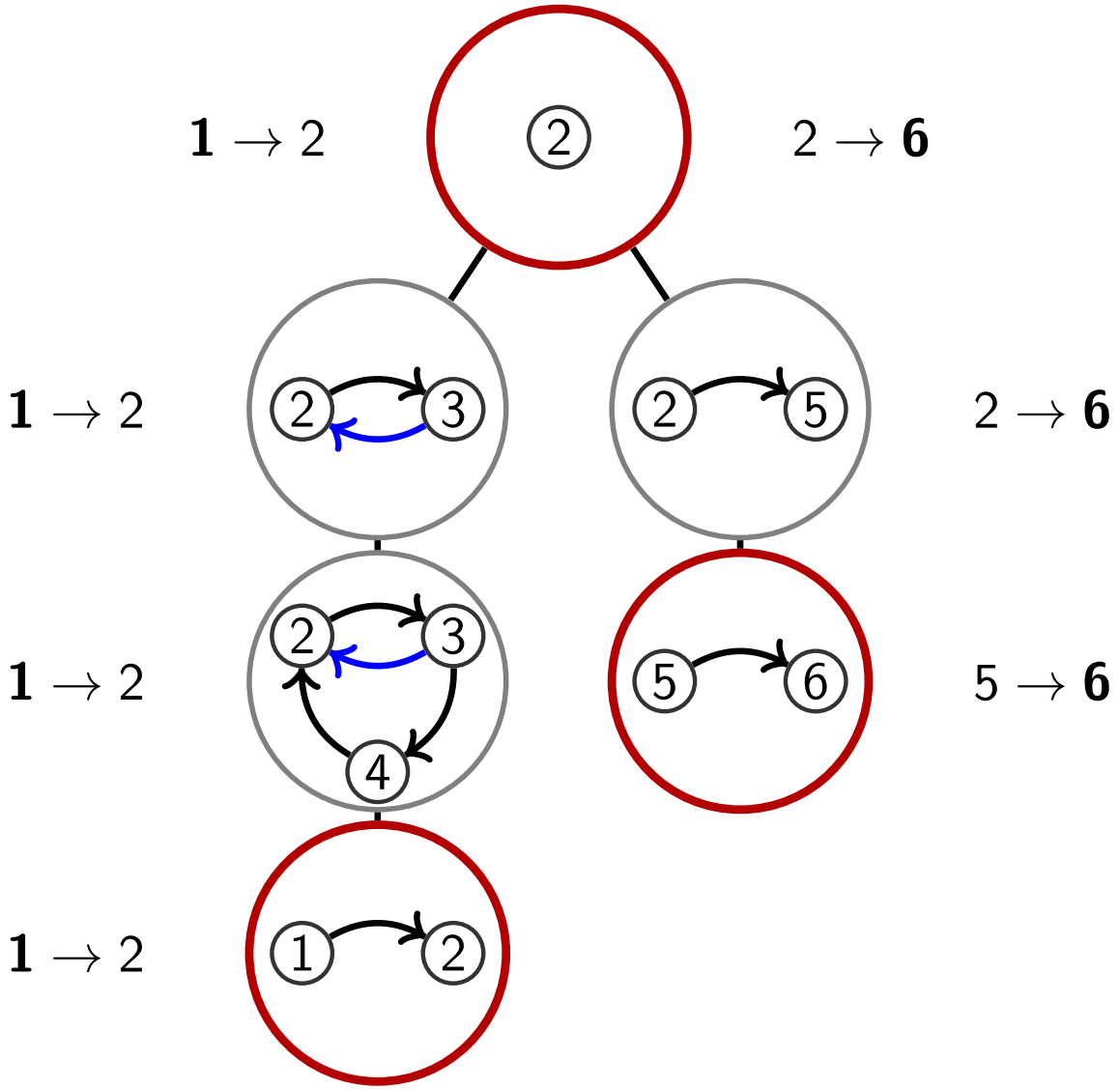


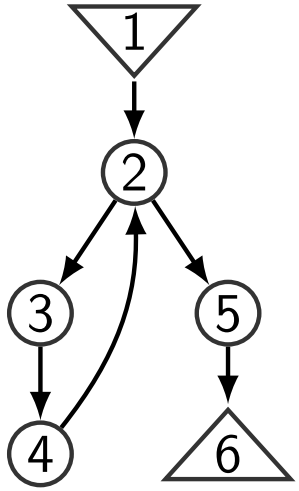
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$





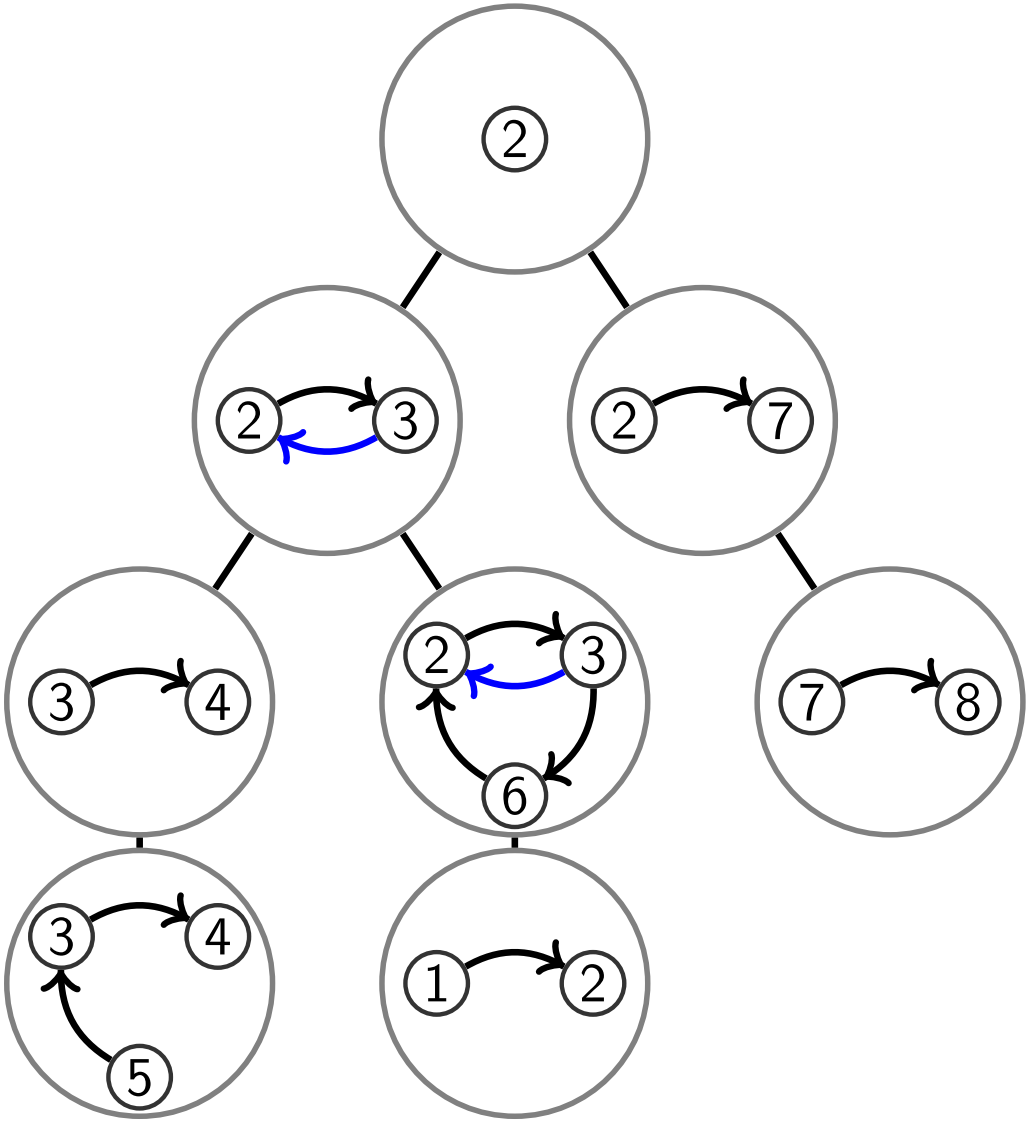
Example: Query $\mathrm{Tree}(G_1)$ for $d(1,6)$

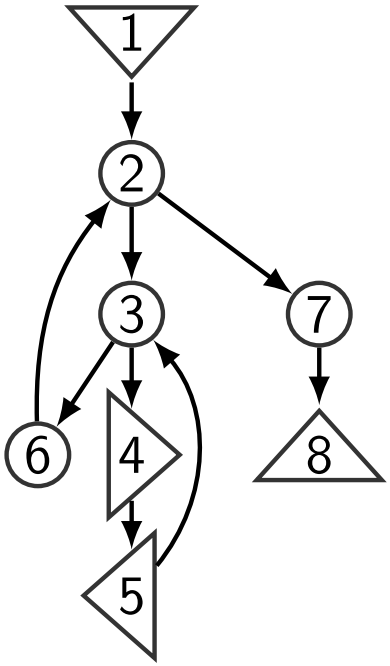




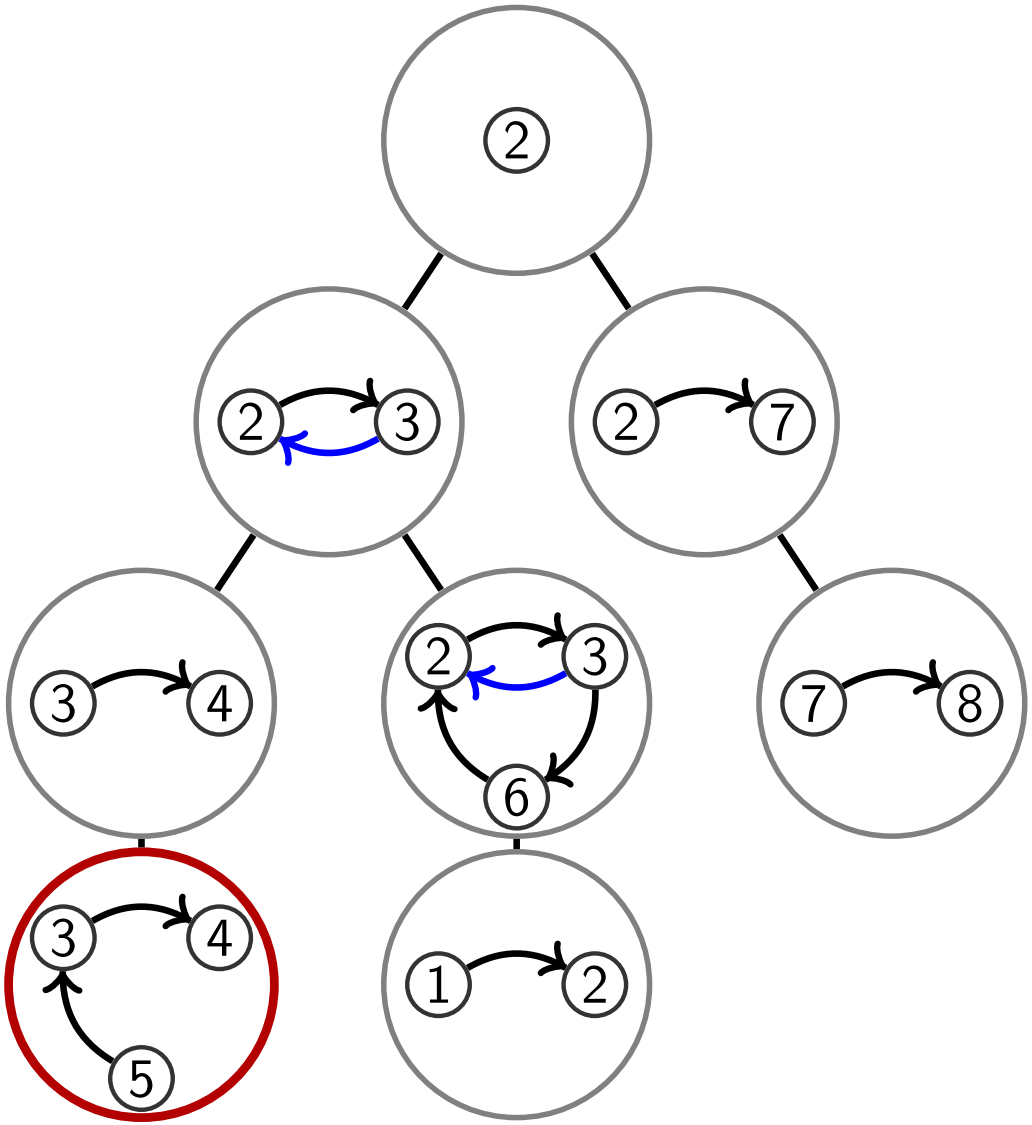
1 → 6

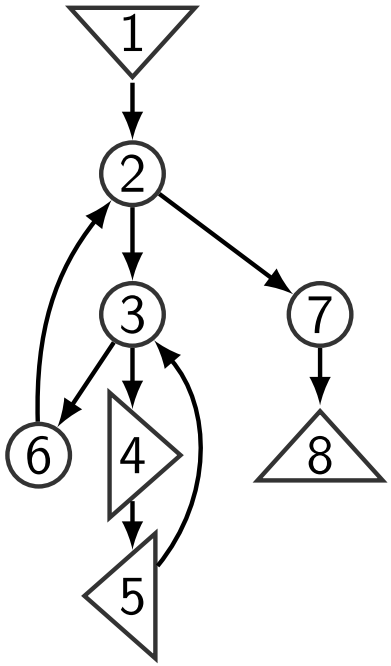
Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$



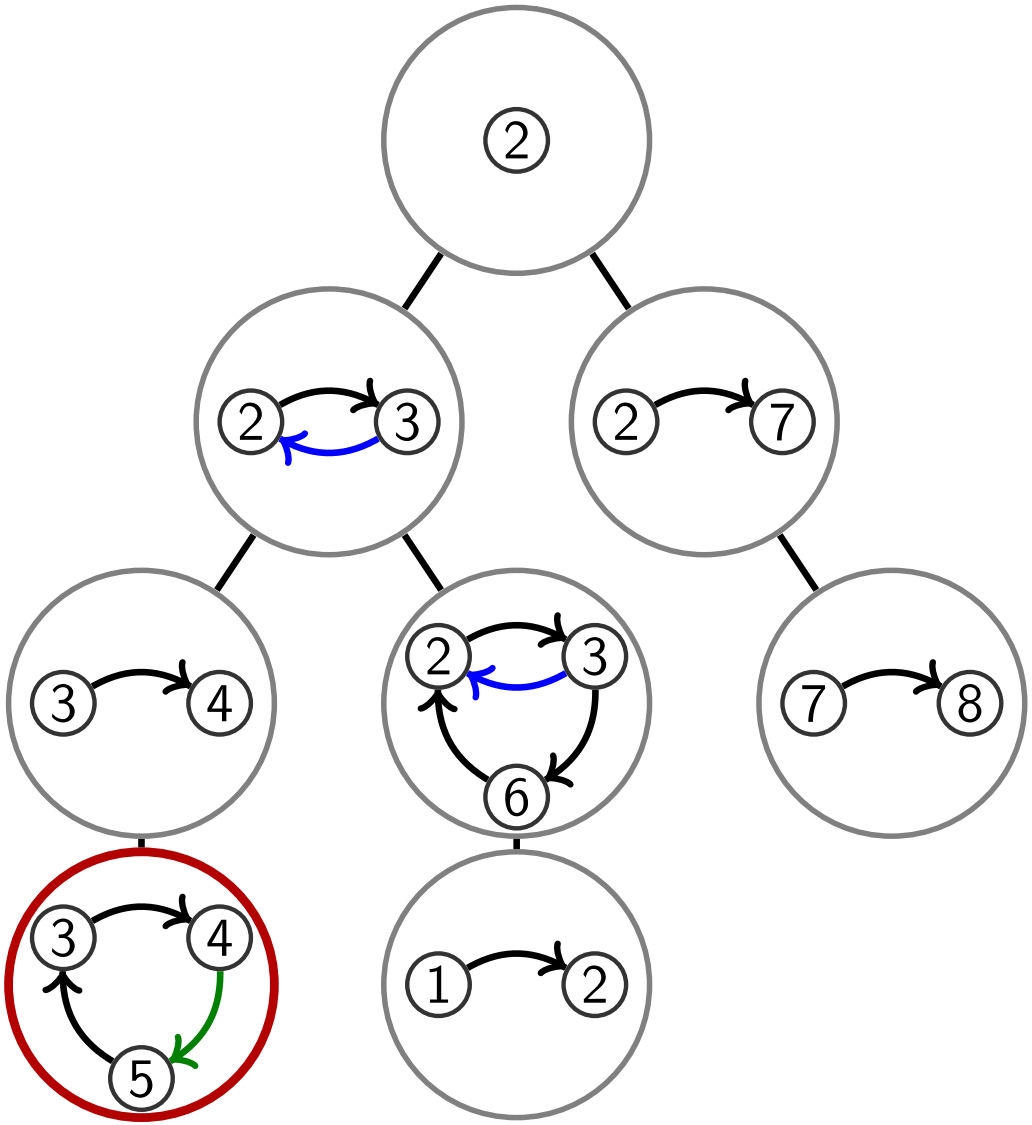


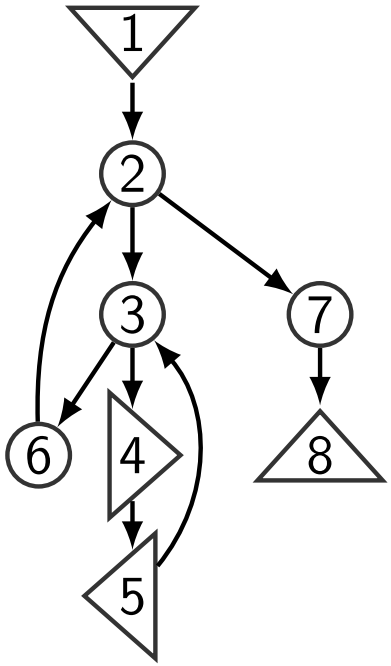
Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$



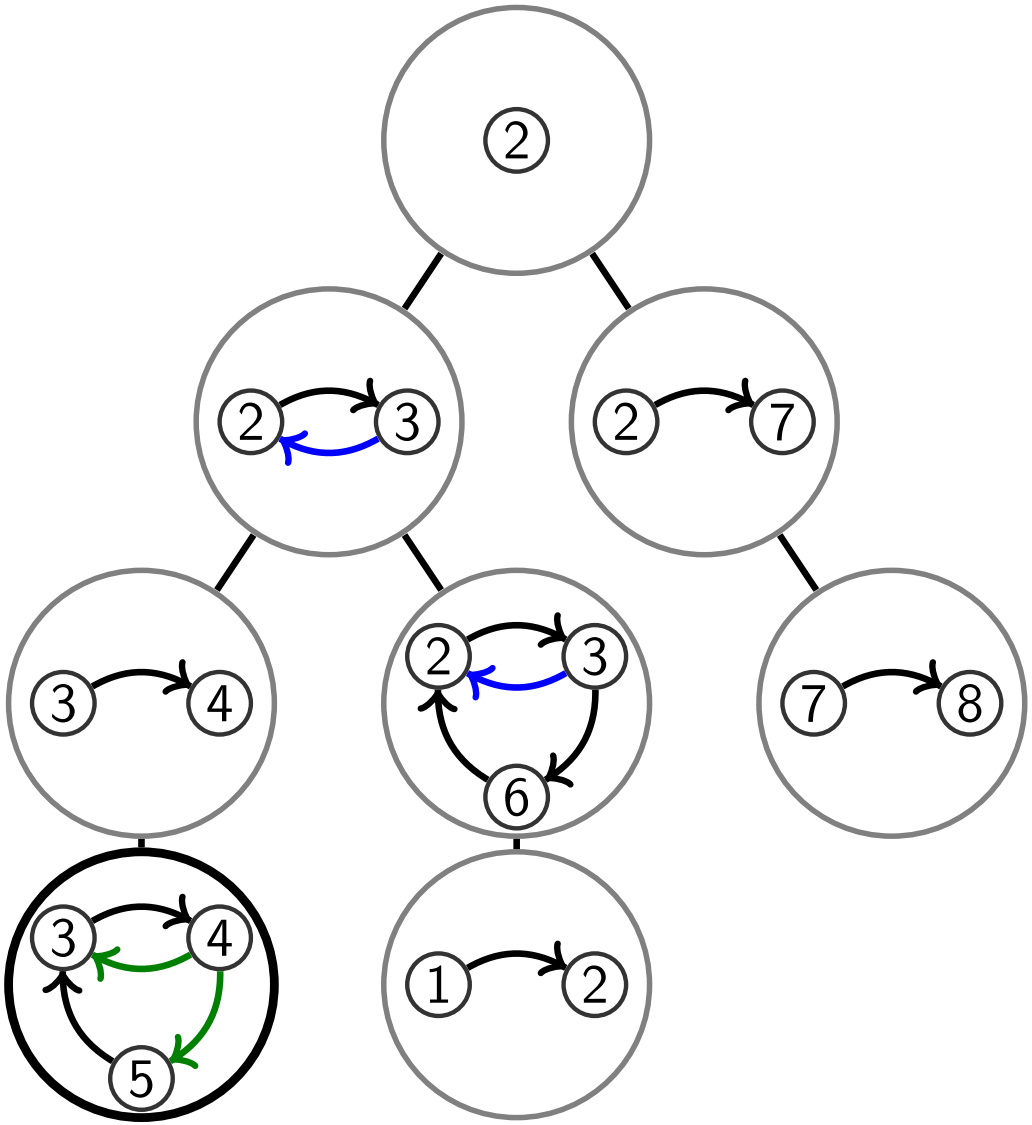


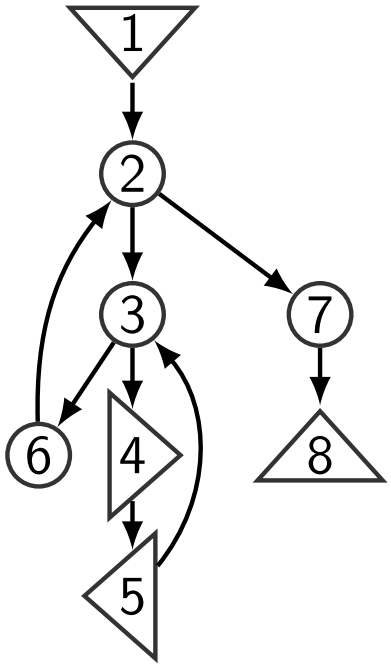
Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$



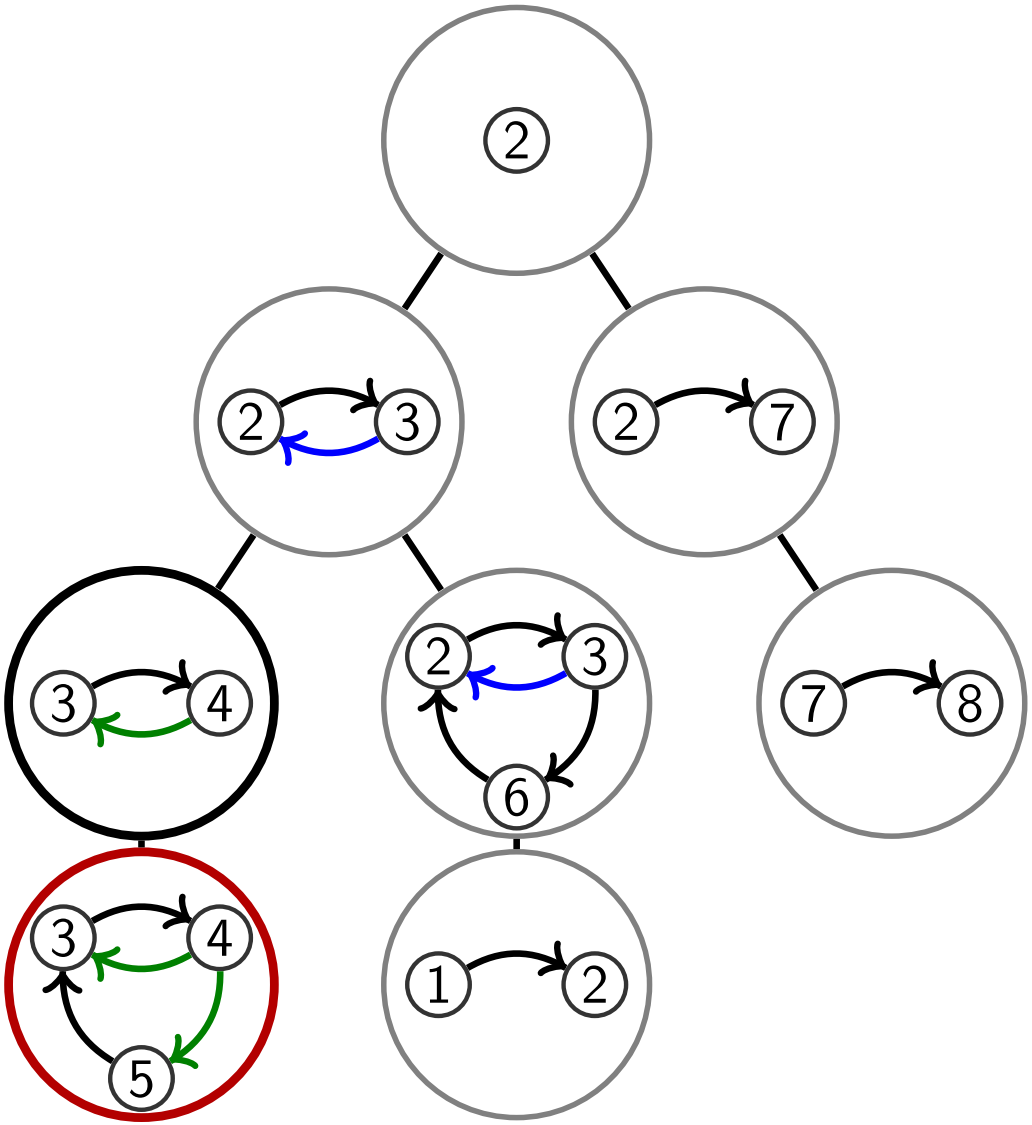


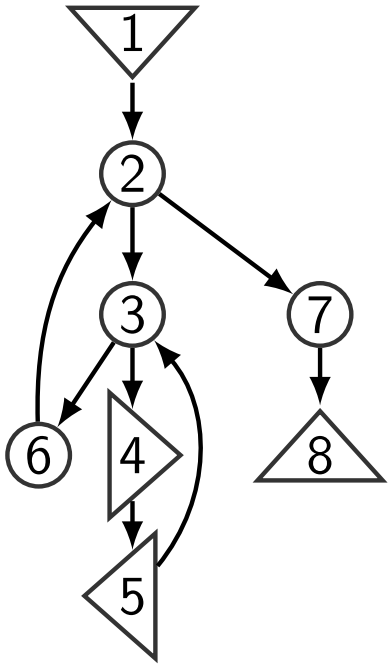
Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$



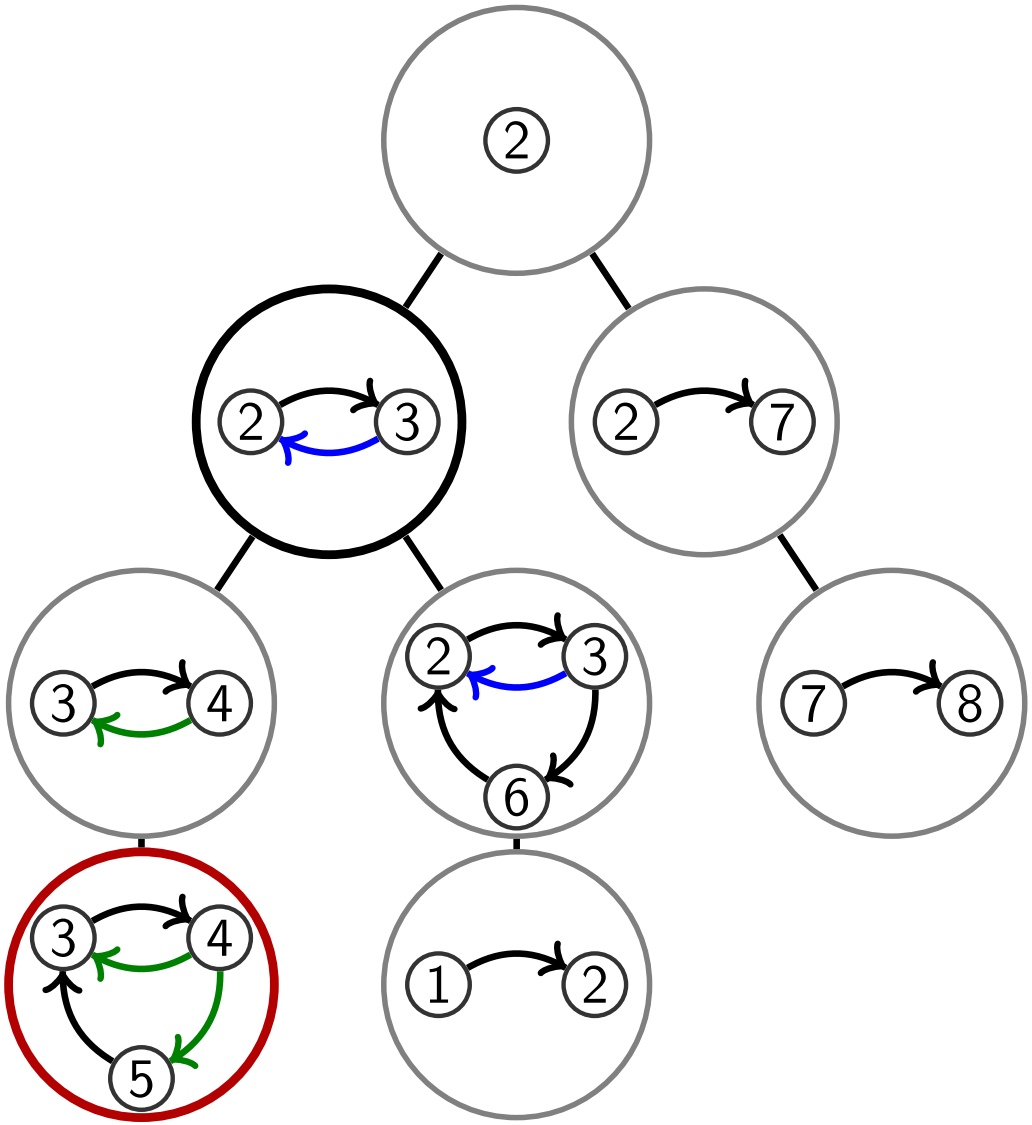


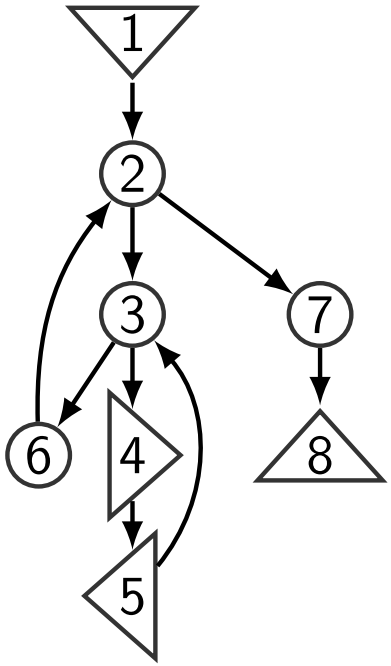
Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$



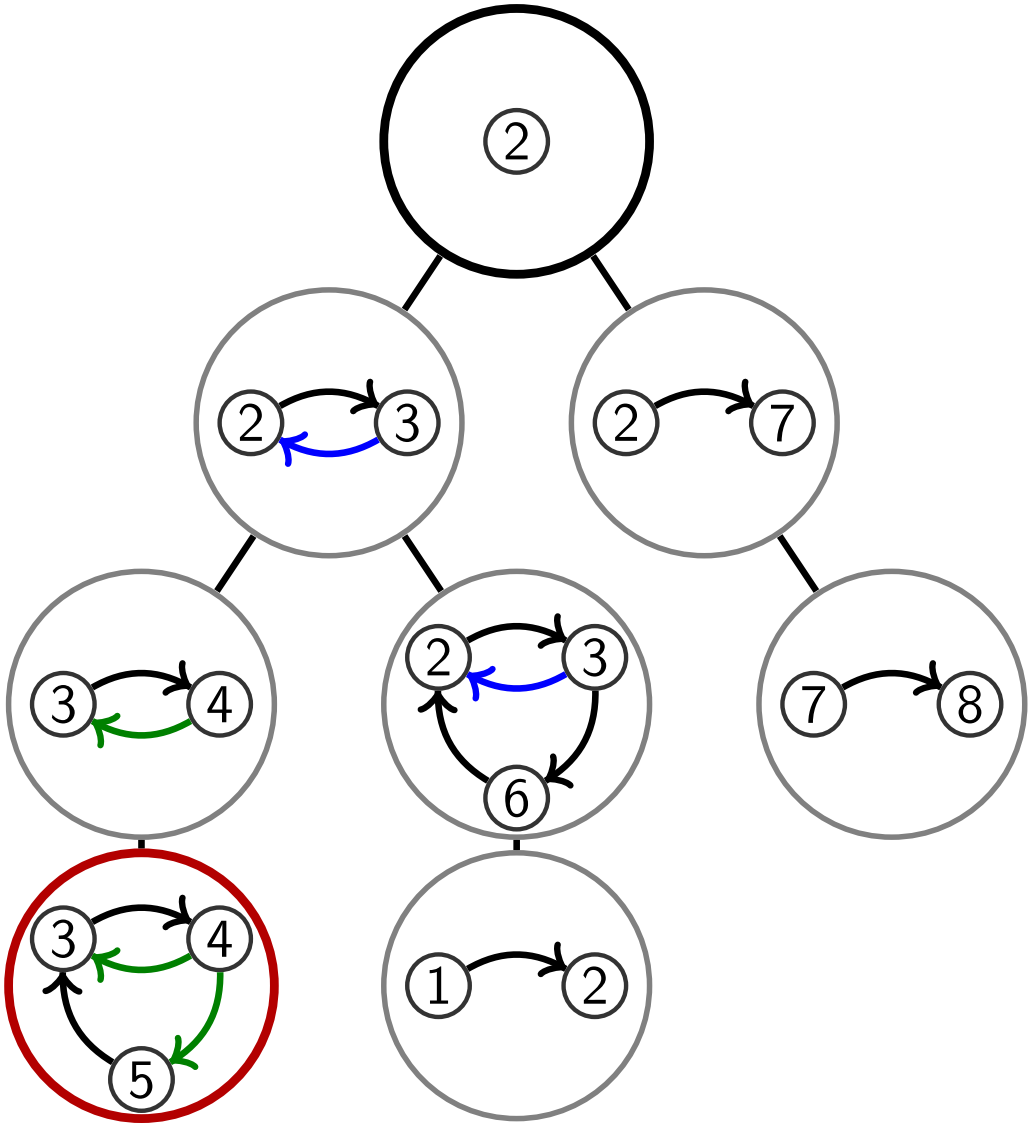


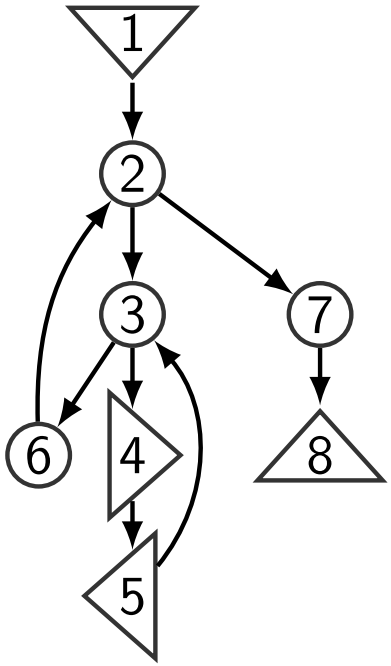
Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$





Example: Update $\mathrm{Tree}(G_2)$ with $d(4,5) = \mathrm{True}$





Experimental Results

- Prototype implementation (Java & Soot)
- Intraprocedural reachability and shortest path
- Compared with standard algorithms
- Reachability:
  - Preprocessing: All pairs reachability DFS
  - Query: DFS
- Shortest path:
  - Preprocessing: Floyd-Warshall
  - Query: Bellman-Ford

Experimental Results

- Prototype implementation (Java & Soot)
- Intraprocedural reachability and shortest path
- Compared with standard algorithms
- Reachability:
  - Preprocessing: All pairs reachability DFS
  - Query: DFS
- Shortest path:
  - Preprocessing: Floyd-Warshall
  - Query: Bellman-Ford

Experimental Results

- Prototype implementation (Java & Soot)
- Intraprocedural reachability and shortest path
- Compared with standard algorithms
- Reachability:
  - Preprocessing: All pairs reachability DFS
  - Query: DFS
- Shortest path:
  - Preprocessing: Floyd-Warshall
  - Query: Bellman-Ford

Experimental Results

Speedup table

Benchmarks			Reachability			Shortest path		
			Preproc.	Query		Preproc.	Query	
	n	t		*Single*	*Pair*		*Single*	*Pair*
antlr	698	1.0	1.78	10	95	51	97	$6 \cdot 10^4$
bloat	696	2.3	1.97	18	143	130	98	$6 \cdot 10^4$
chart	1159	1.5	4.09	35	171	988	223	$2 \cdot 10^5$
eclipse	656	1.6	3.75	35	82	70	44	$5 \cdot 10^4$
fop	1209	1.7	3.04	20	358	2562	59	$4 \cdot 10^4$
hsqldb	698	1.0	3.24	16	99	57	103	$6 \cdot 10^4$
jython	748	1.5	1.58	11	116	89	77	$9 \cdot 10^4$
luindex	885	1.3	3.94	36	165	182	198	$1 \cdot 10^5$
lusearch	885	1.3	3.68	16	200	132	199	$1 \cdot 10^5$
pmd	644	1.4	1.67	33	96	64	69	$6 \cdot 10^4$
xalan	698	1.0	2.40	29	75	58	63	$5 \cdot 10^4$
jflex	1091	1.6	1.78	16	186	800	359	$2 \cdot 10^5$
muffin	1022	1.7	2.23	20	185	809	305	$2 \cdot 10^5$
javac	711	1.8	1.81	15	107	111	70	$7 \cdot 10^4$
polyglot	698	1.0	2.20	15	101	50	100	$6 \cdot 10^4$

Experimental Results

Speedup table

Benchmarks			Reachability			Shortest path		
			Preproc.	Query		Preproc.	Query	
	n	t		*Single*	*Pair*		*Single*	*Pair*
antlr	698	1.0	1.78	10	95	51	97	$6 \cdot 10^4$
bloat	696	2.3	1.97	18	143	130	98	$6 \cdot 10^4$
chart	1159	1.5	4.09	35	171	988	223	$2 \cdot 10^5$
eclipse	656	1.6	3.75	35	82	70	44	$5 \cdot 10^4$
fop	1209	1.7	3.04	20	358	2562	59	$4 \cdot 10^4$
hsqldb	698	1.0	3.24	16	99	57	103	$6 \cdot 10^4$
jython	748	1.5	1.58	11	116	89	77	$9 \cdot 10^4$
luindex	885	1.3	3.94	36	165	182	198	$1 \cdot 10^5$
lusearch	885	1.3	3.68	16	200	132	199	$1 \cdot 10^5$
pmd	644	1.4	1.67	33	96	64	69	$6 \cdot 10^4$
xalan	698	1.0	2.40	29	75	58	63	$5 \cdot 10^4$
jflex	1091	1.6	1.78	16	186	800	359	$2 \cdot 10^5$
muffin	1022	1.7	2.23	20	185	809	305	$2 \cdot 10^5$
javac	711	1.8	1.81	15	107	111	70	$7 \cdot 10^4$
polyglot	698	1.0	2.20	15	101	50	100	$6 \cdot 10^4$

Experimental Results

Speedup table

Benchmarks			Reachability			Shortest path		
			Preproc.	Query		Preproc.	Query	
	n	t		*Single*	*Pair*		*Single*	*Pair*
antlr	698	1.0	1.78	10	95	51	97	$6 \cdot 10^4$
bloat	696	2.3	1.97	18	143	130	98	$6 \cdot 10^4$
chart	1159	1.5	4.09	35	171	988	223	$2 \cdot 10^5$
eclipse	656	1.6	3.75	35	82	70	44	$5 \cdot 10^4$
fop	1209	1.7	3.04	20	358	2562	59	$4 \cdot 10^4$
hsqldb	698	1.0	3.24	16	99	57	103	$6 \cdot 10^4$
jython	748	1.5	1.58	11	116	89	77	$9 \cdot 10^4$
luindex	885	1.3	3.94	36	165	182	198	$1 \cdot 10^5$
lusearch	885	1.3	3.68	16	200	132	199	$1 \cdot 10^5$
pmd	644	1.4	1.67	33	96	64	69	$6 \cdot 10^4$
xalan	698	1.0	2.40	29	75	58	63	$5 \cdot 10^4$
jflex	1091	1.6	1.78	16	186	800	359	$2 \cdot 10^5$
muffin	1022	1.7	2.23	20	185	809	305	$2 \cdot 10^5$
javac	711	1.8	1.81	15	107	111	70	$7 \cdot 10^4$
polyglot	698	1.0	2.20	15	101	50	100	$6 \cdot 10^4$

Thank you!
Questions?