Path & Reachability

COMP9312_24T2

UNSW SYDNEY

Outline

- Reachability

Transitive closure

Optimal Tree cover

Two-Hop labelling

- Shortest Path

Dijkstra's algorithm

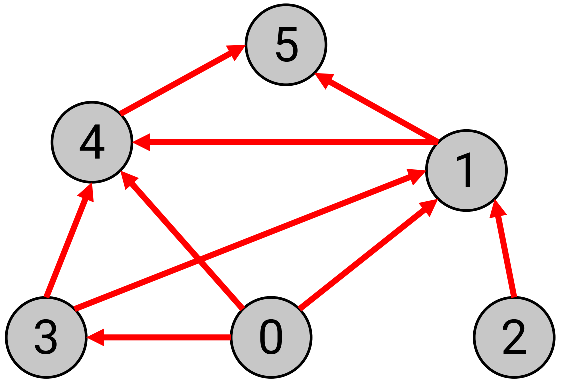
A* algorithm

Floyd-Warshall algorithm

Reachability

Problem formulation

Given an *unweighted directed graph G* and two nodes u and v, is there a path connecting u to v (denoted $u \rightsquigarrow v$)?



$0 \rightsquigarrow 5$? YES

$0 \rightsquigarrow 2$? NO

Directed Graph → DAG (directed acyclic graph) by coalescing the strongly connected components

Motivation

- Classical problem in graph theory.
- Studying the influence flow in social networks.
  - Even undirected graphs (facebook) are converted to directed w.r.t a certain attribute distribution
- Security: finding possible connections between suspects.
- Biological data: is that protein involved directly or indirectly in the expression of a gene?
- Primitive for many graph related problems (pattern matching).

An Online Approach

Whether or not $u \rightsquigarrow v$

- Conduct DFS or BFS starting from u
- if the node v is discovered:
  - then stop search, report YES
- If the stack/queue is empty:
  - then report NO

No index and thus no construction overhead and no extra space consumption

TOO GOOD

Query time: O(m+n)
the entire graph will be traversed in the worst case

TOO BAD

Index-based methods

1. Transitive closure

   Run the Floyd-Warshall algorithm and store all possible query results in a matrix.

2. Tree cover (DAG)

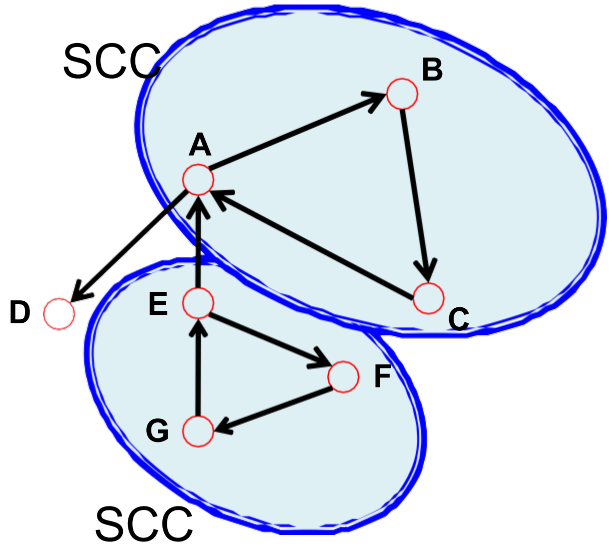
   Use spanning trees to store the reachability information that is originally stored in transitive closure in hierarchy.

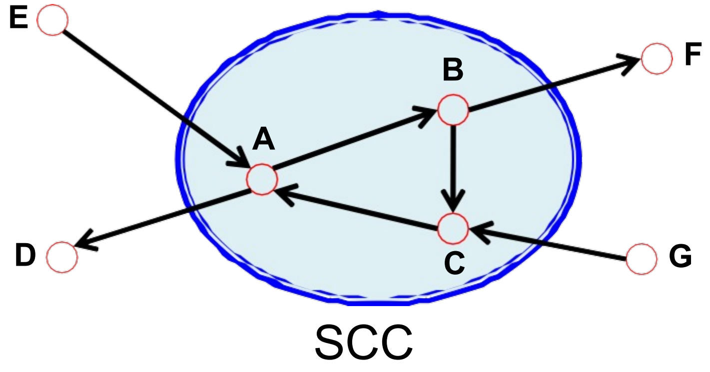
3. 2-hop labeling

   For each node in the graph, assign two label sets for it to store the reachability information that is originally stored in transitive closure.

Index-based methods for directed graph

Most index-based reachability methods assume the directed graph is a DAG (directed acyclic graph), which can be derived by contracted all SCCs (strongly connected components).

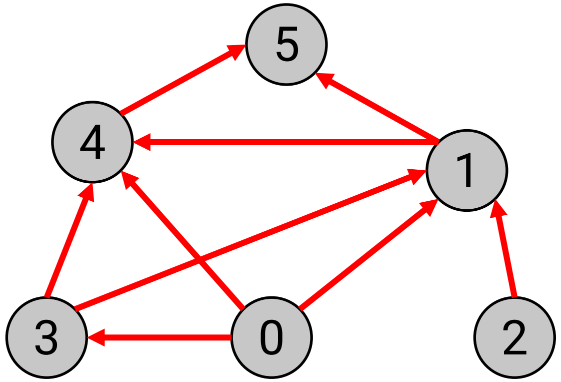




Transitive Closure

Transitive Closure (TC)

A transitive closure is a Boolean matrix storing the answers of all possible reachability queries. The size of the matrix is $O(n^2)$, where n denotes the number of vertices in the graph.



The original graph G

	0	1	2	3	4	5
0	1	1	0	1	1	1
1	0	1	0	0	1	1
2	0	1	1	0	1	1
3	0	1	0	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	0	1

The transitive closure of G

Transitive Closure (TC)

The *transitive closure* is a Boolean matrix:

```
bool tc[num_vertices][num_vertices];

// Initialize the matrix tc:  O(n^2)
tc[i][j] = 1 if there is an edge from i to j, or i == j;

// Run Floyd-Warshall
for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
        }
    }
}
```

The Floyd-Warshall algorithm will be covered in Topic 2.2 (Shortest Path)

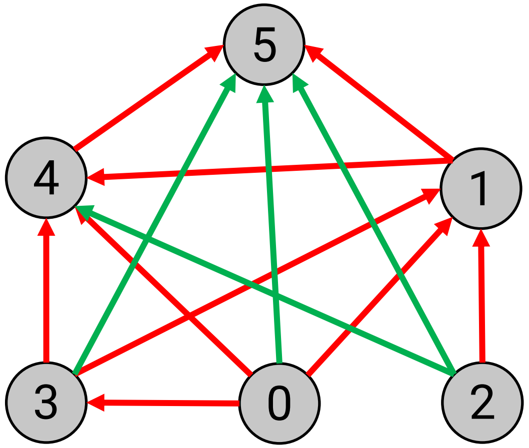
Transitive Closure (TC)

After the iteration k, we find the reachability pairs (i,j) where the reachability path is formed by {v_0,v_1,…,v_k}

```
// Run Floyd-Warshall
for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
        }
    }
}
```

Transitive Closure (TC)

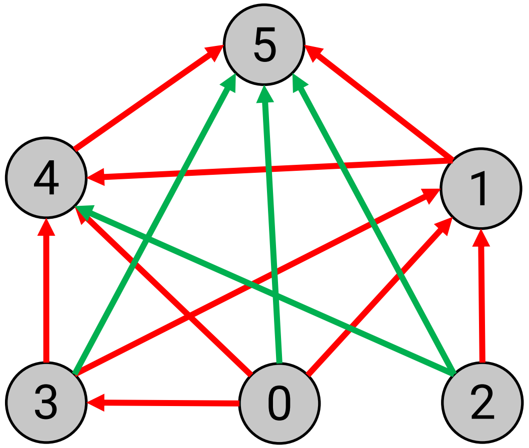
TC(G): each edge indicates the reachability information.



	0	1	2	3	4	5
0	1	1	0	1	1	1
1	0	1	0	0	1	1
2	0	1	1	0	1	1
3	0	1	0	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	0	1

Transitive Closure (TC)

TC(G): each edge indicates the reachability information.



	0	**1**	**2**	**3**	**4**	**5**
0	1	1	0	1	1	1
1	0	1	0	0	1	1
2	0	1	1	0	1	1
3	0	1	0	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	0	1

- It can be done by dynamic programming algorithm *Floyd–Warshall* in $O(n^3)$
- It takes $O(n^2)$ space

TOO BAD

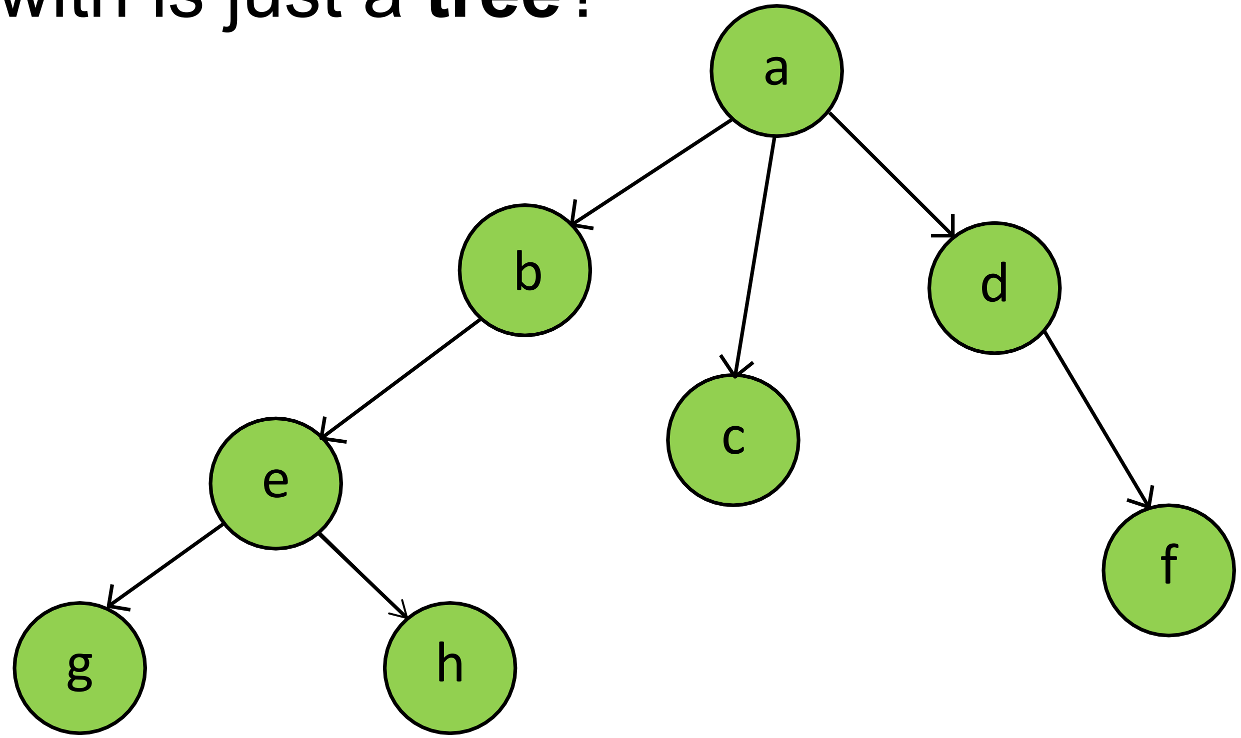
- BUT, queries can be answered in constant time $O(1)$

TOO GOOD

Tree Cover

Reachability in a Tree

What if the DAG we are dealing with is just a **tree**?



A tree T

Reachability in a Tree

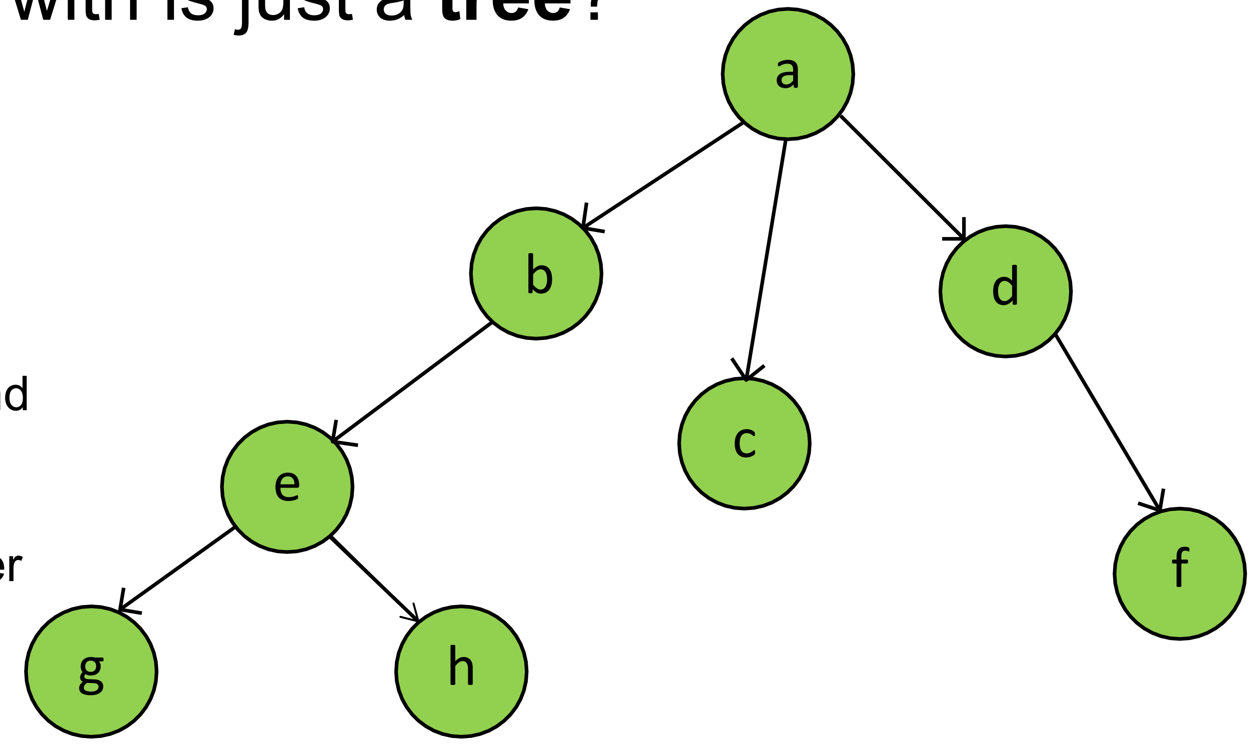
What if the DAG we are dealing with is just a **tree**?

Main idea:
For each node in the tree, we assign a label to indicate the nodes reachable.

Implementation:

1. Conduct a post-order traversal on the tree and record the post-order number.

2. For each node, record the minimum post-order number of its descendants.



A tree T

Reachability in a Tree

Pseudo code for post-order-traversal

post-order-traversal(root):

for each v of root's children from left to right:

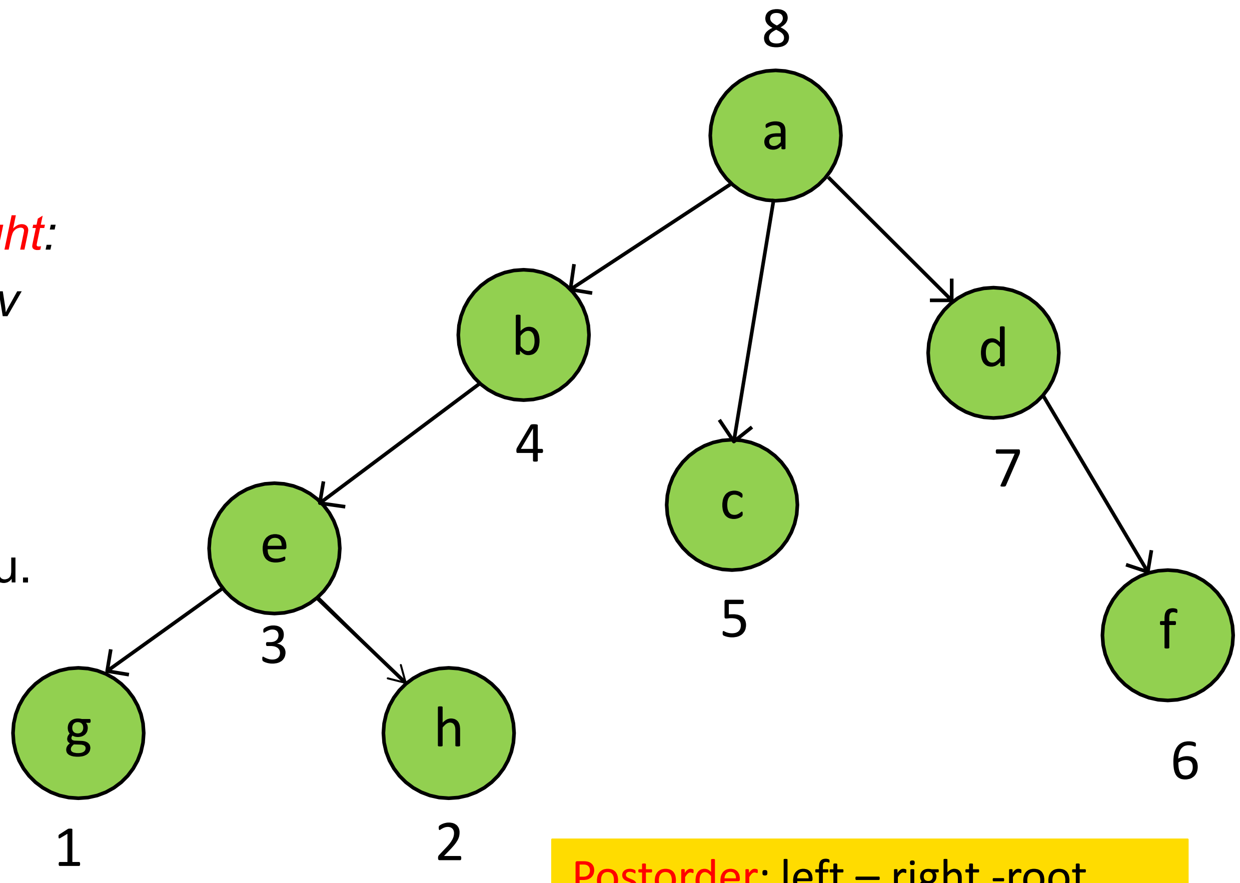
// traverse the subtree rooted at v

post-order-traversal(v)

visit root

We use p(u) to denote the **post-order number** of u.

An example is shown on the right.



Postorder: left – right -root

Reachability in a Tree

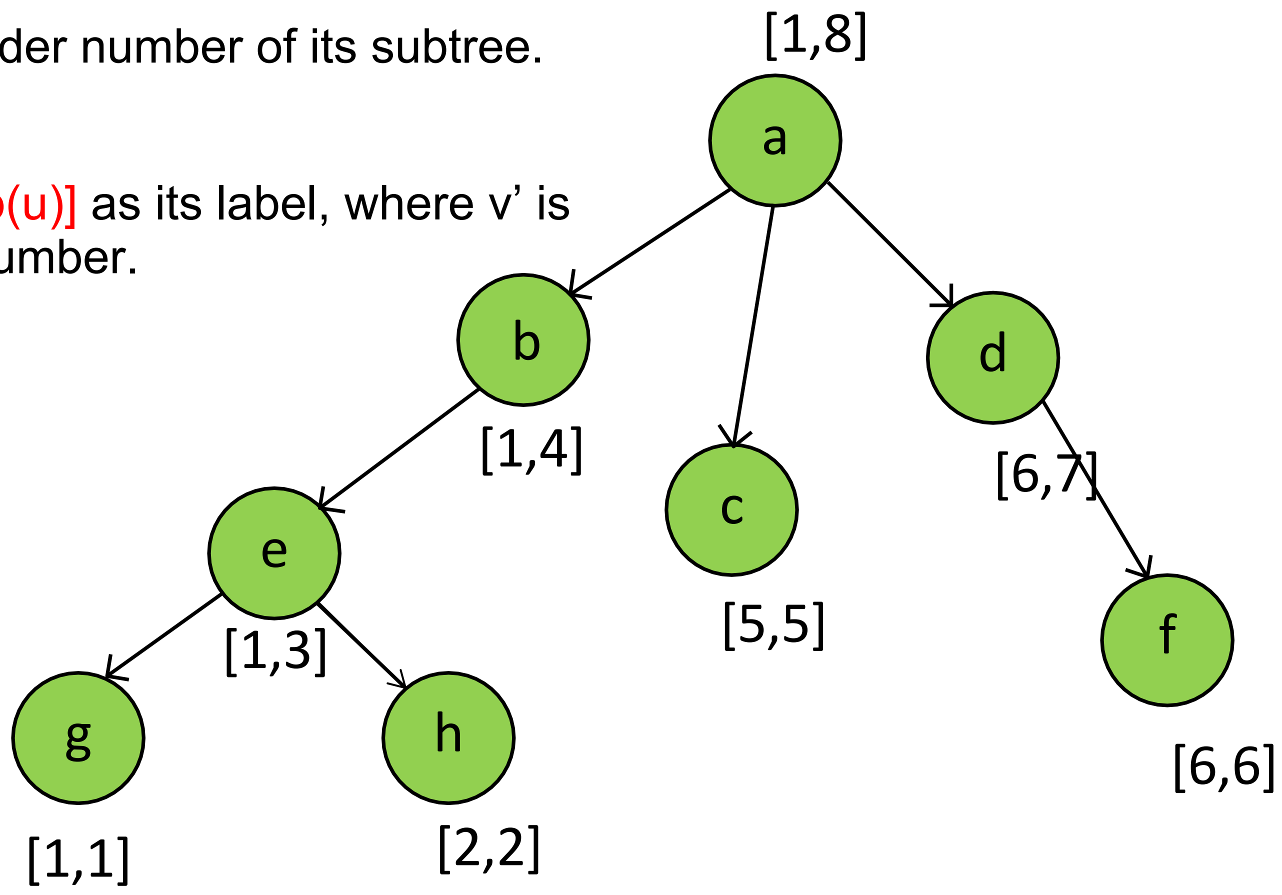
For each vertex, we compute the minimum post-order number of its subtree.

For each vertex u, we construct an interval [p(v'), p(u)] as its label, where v' is the descendant of u with the smallest post-order number.

What do you observe?

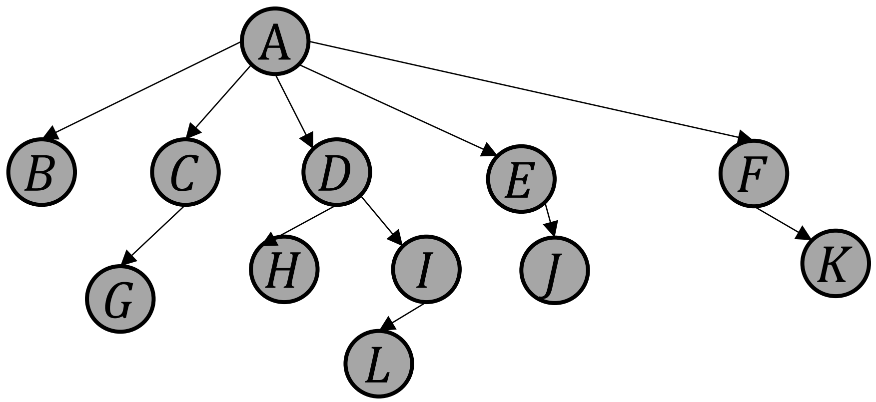
Query Processing: $?(u \rightsquigarrow v) \Longrightarrow$
$?(u_{start} \leq v_{end} < u_{end})$

(example) $?(b \rightsquigarrow h) \Longrightarrow ?(1 \leq 2 < 4) \Rightarrow YES$
$?(b \rightsquigarrow c) \Longrightarrow ?(1 \leq 5 < 4) \Rightarrow NO$



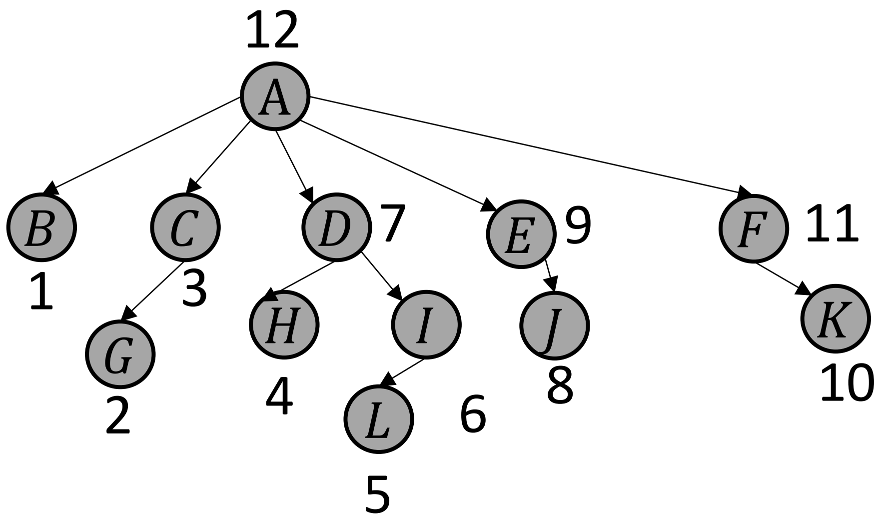
Quick Exercise

Assign the post-order numbers for this tree:



Quick Exercise

ANSWER: the post-order numbers for this tree:



Tree Cover

How to generalize the above steps to any DAG ?

Main idea:

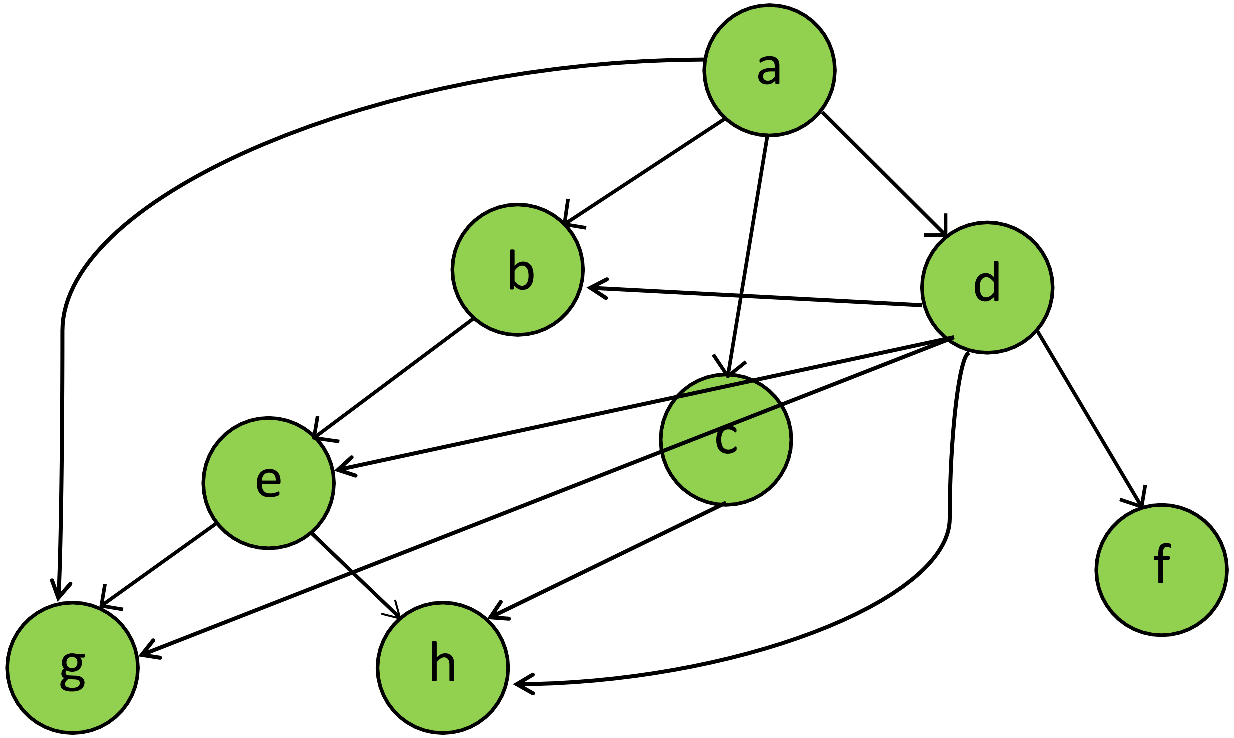
1. Consider a spanning tree (tree cover) of the DAG.

2. Go through the above steps for the tree.

3. Recover the non-tree edges and use them to pass on the reachability information.

We assume the DAG G has only one connected component.

If G contains multiple connected components, we connect them to a virtual root node.



A general DAG G

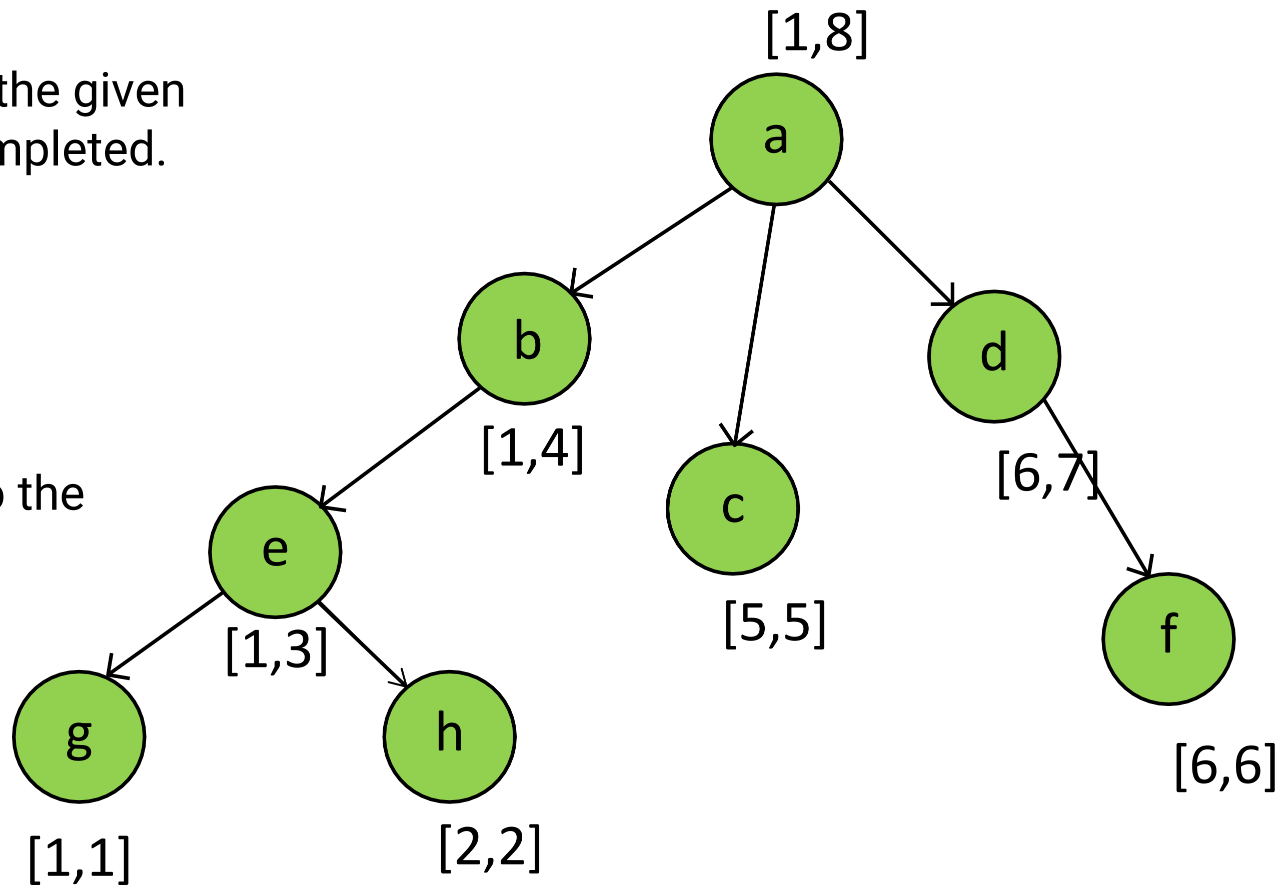
Tree Cover

The tree T we considered is also a spanning tree of the given DAG G. Thus, step one and step two are already completed.

Now we need to restore the non-tree edges:

Topological sort the vertices.
For each vertex q in reverse topological order:
    for each edge (p,q) add the intervals of q to the intervals of p.

Note:
(1) need to consider both tree- and non-tree edges
(2) remove the subsumed intervals



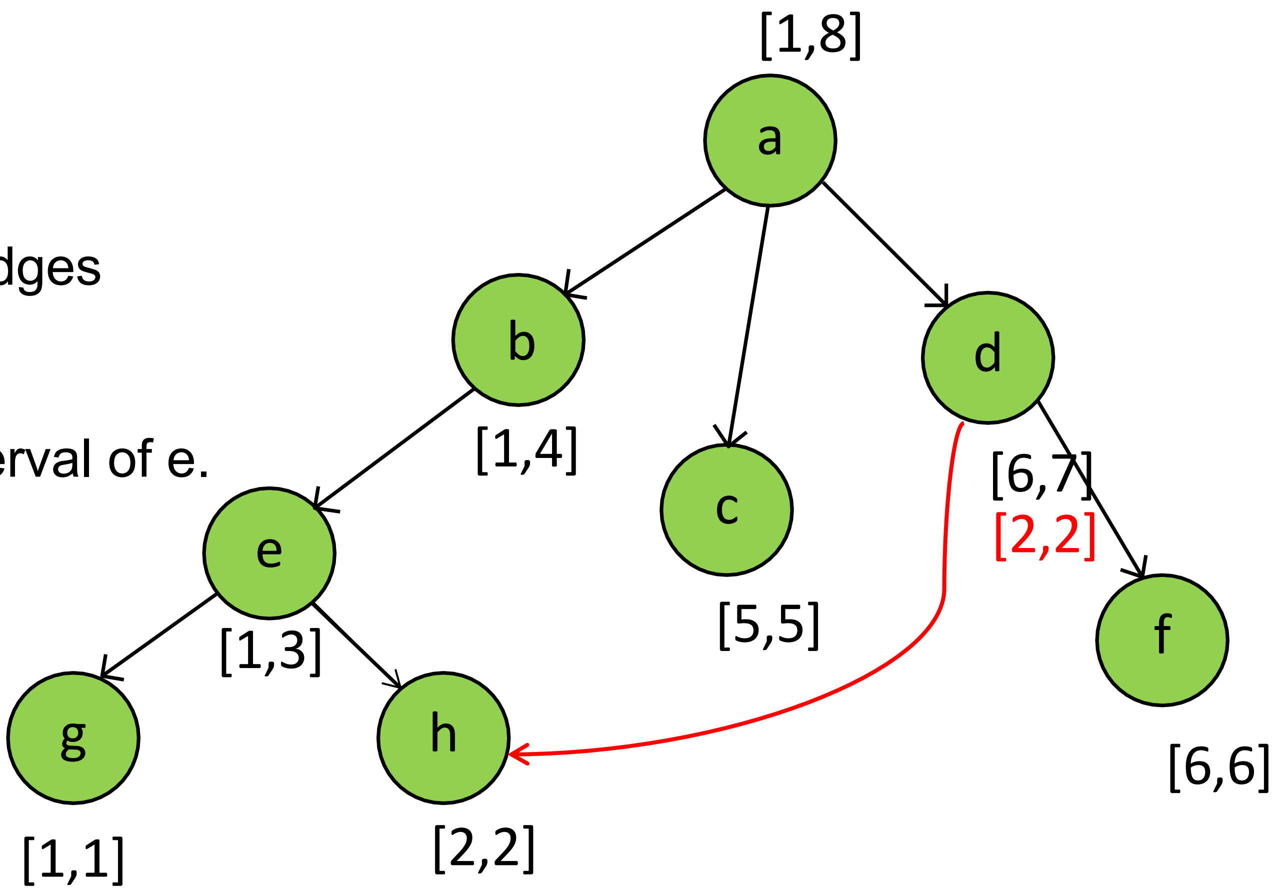
Tree Cover

A topological ordering of the vertices:
a, d, b, c, f, e, g, h

First consider vertex h, which has incoming edges (d, h), (c, h), and (e, h).

Considering (e, h), no need to change the interval of e.

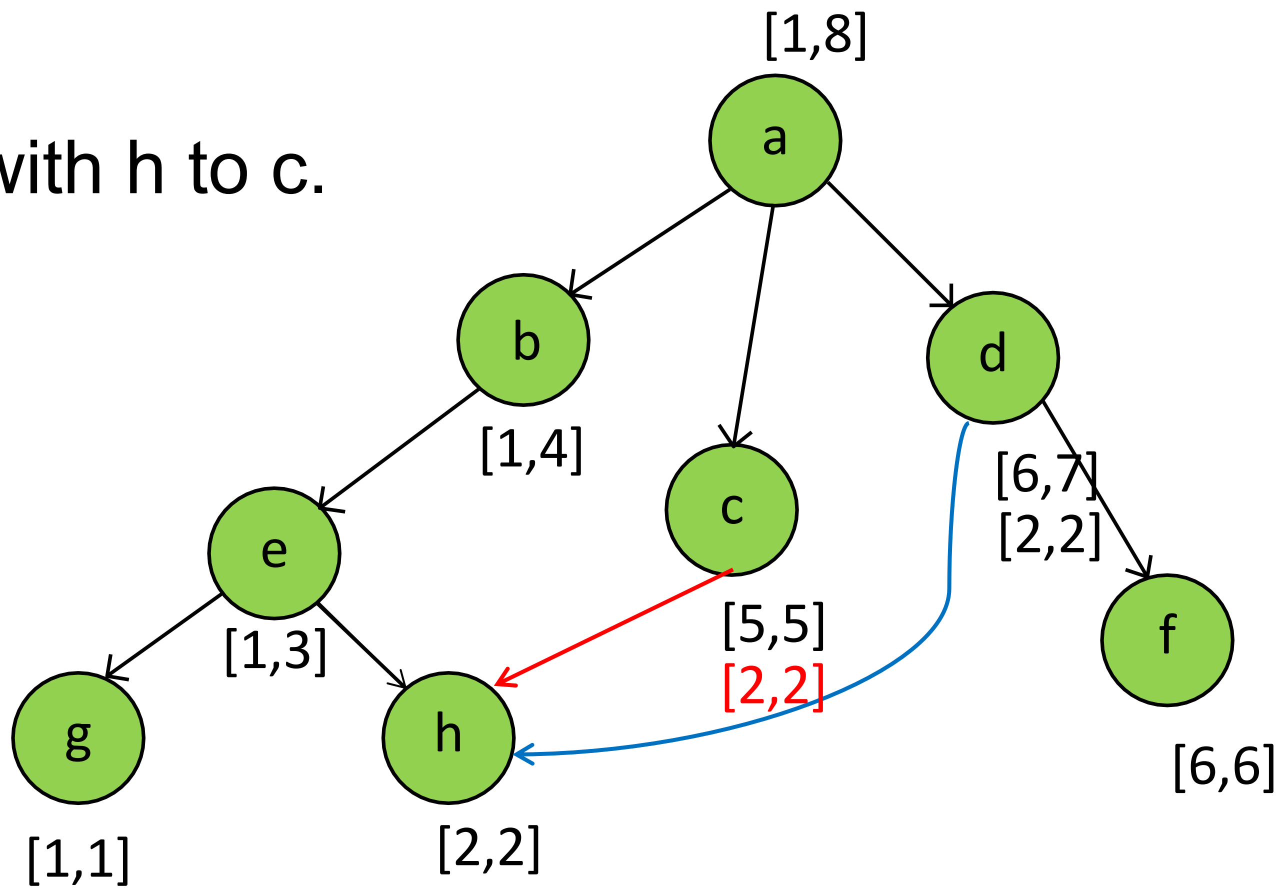
Add edge (d, h):
We add the interval associated with h to d.



Tree Cover

Add edge (c, h):

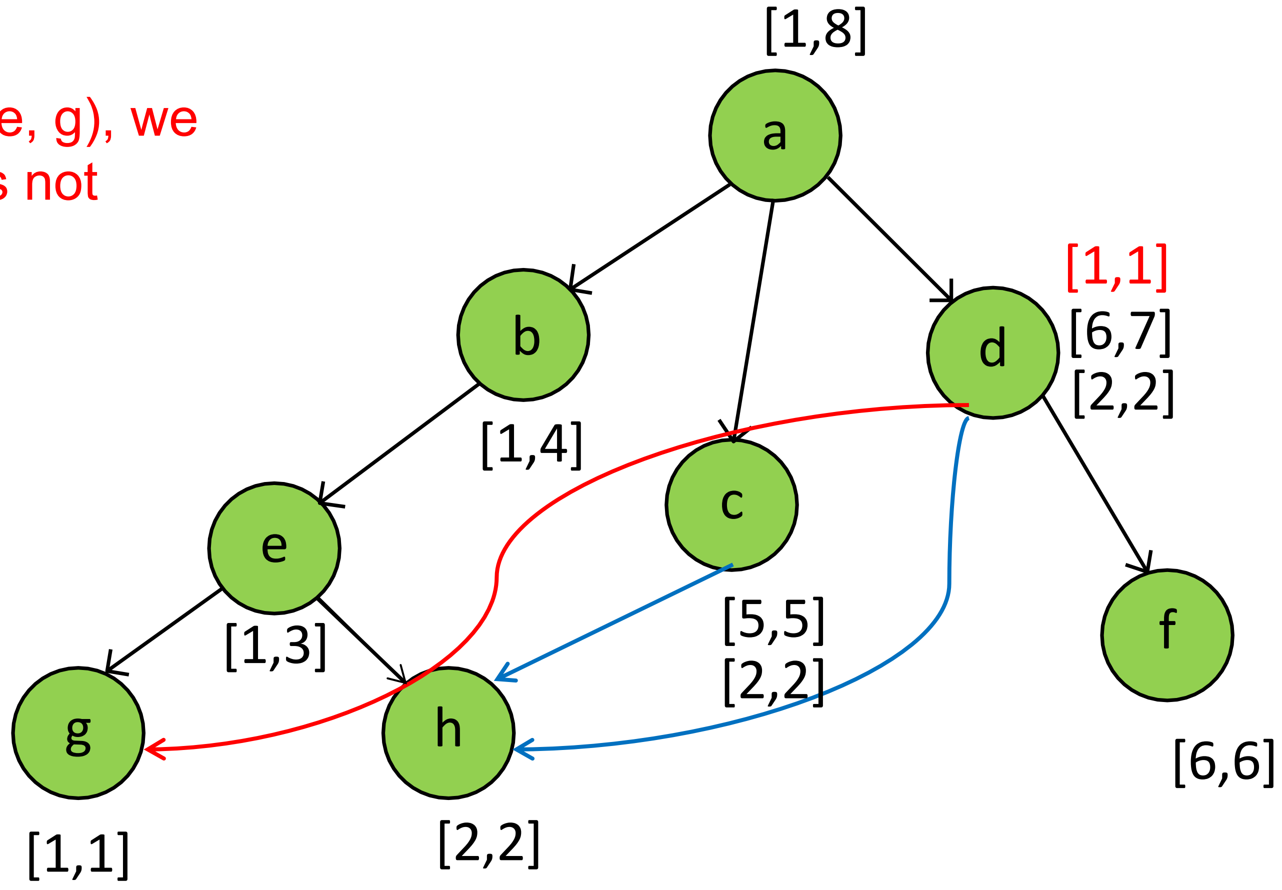
We add the interval associated with h to c.



Tree Cover

The next vertex to consider is g.
Among its incoming edges (d, g), (a, g), and (e, g), we consider (d, g), and (a, g) because (e, g) does not change the interval of e.

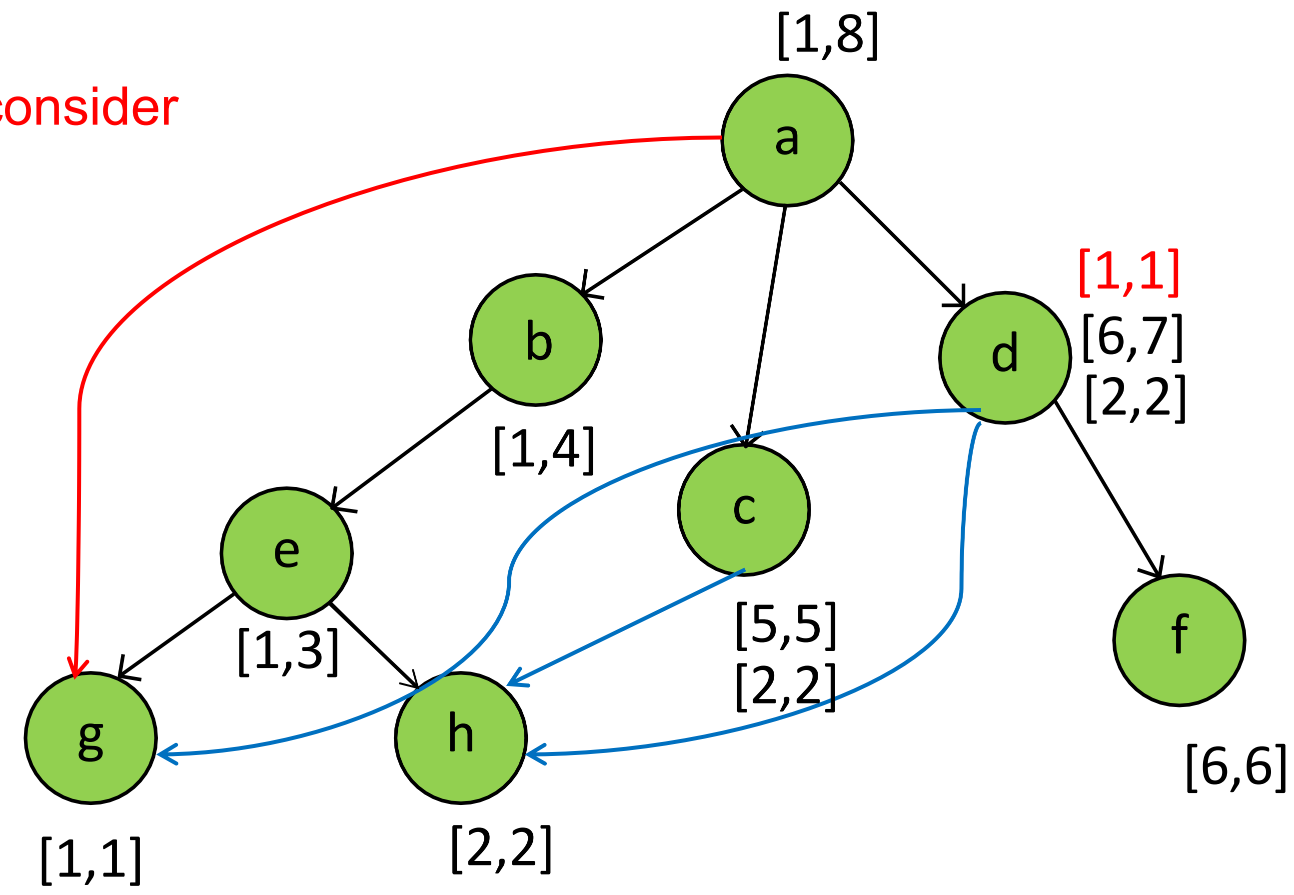
Add edge (d, g):
We add the interval [1,1] to d.



Tree Cover

The next vertex to consider is g.
(e, g) does not change the interval of e. We consider (d, g), and (a, g):

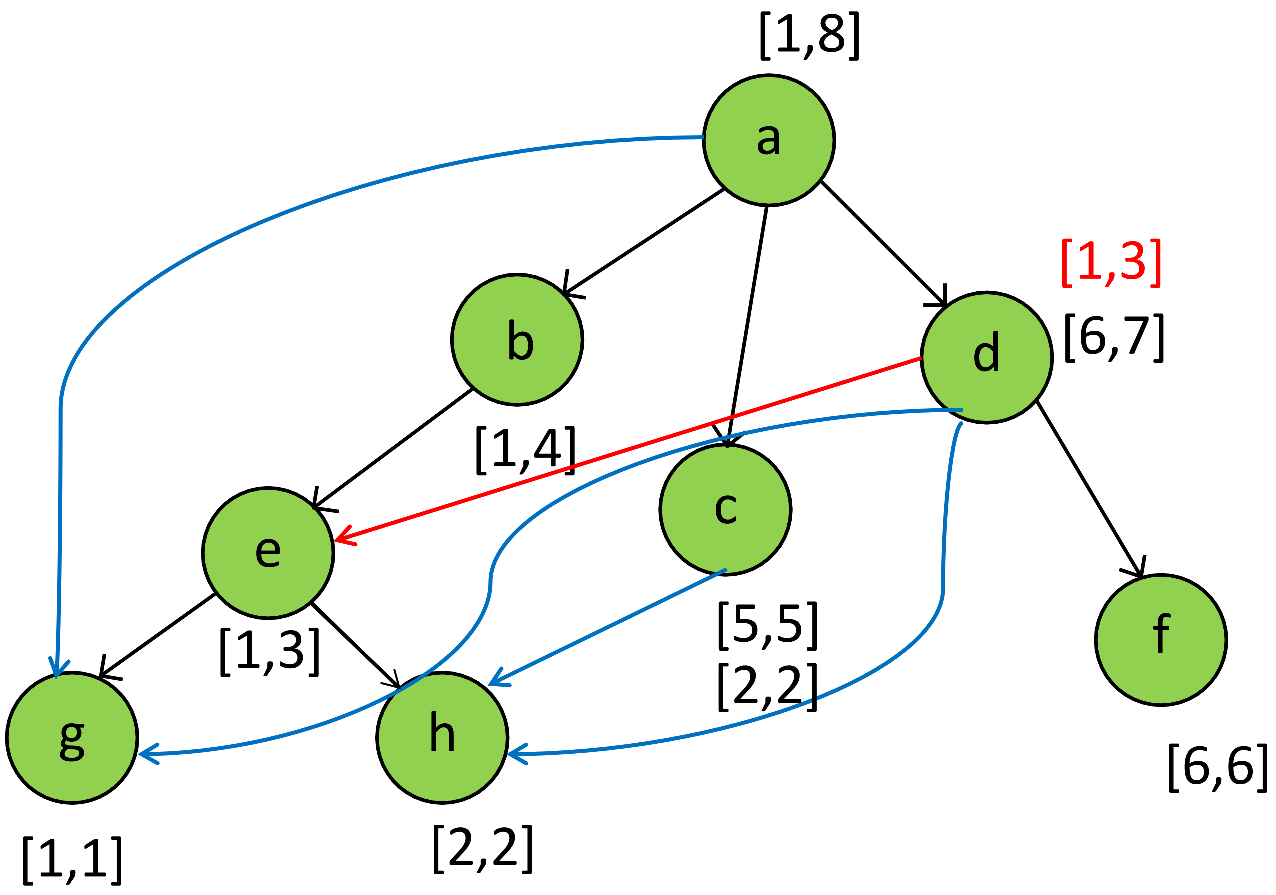
Add edge (a, g):
We DO NOT add the interval [1,1] to a.
This is because [1,1] is contained by [1,8].



Tree Cover

The next vertex to consider is e.
(b, e) does not change the interval of b. We consider the edge: (d, e)

Add edge (d, e):
We add the interval [1,3] to d.
Since [1,1] and [2,2] are contained by [1,3], we only keep [1 ,3]



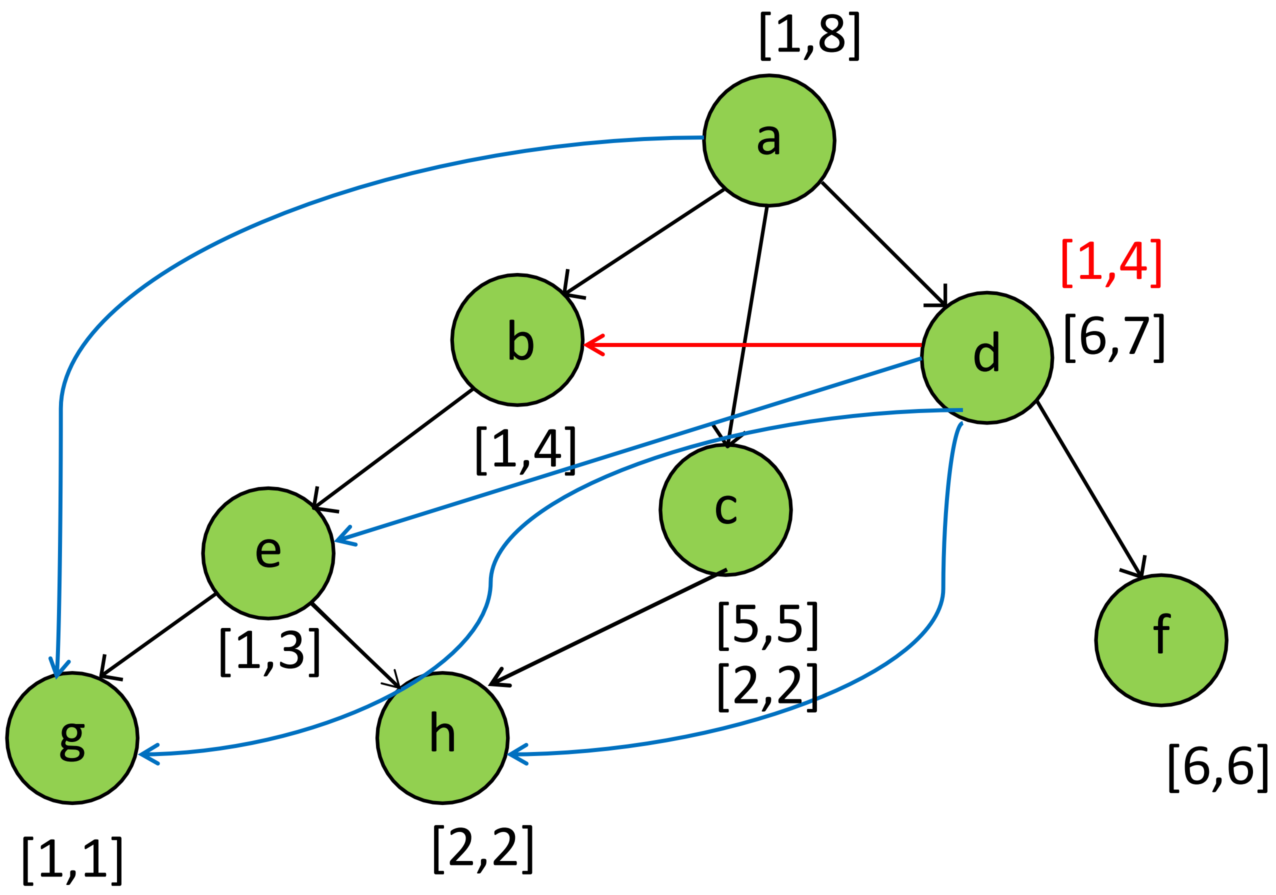
Tree Cover

The next vertex to consider is f.
(d, f) does not change the interval of d.

The next vertex to consider is c.
(a, c) does not change the interval of a.

The next vertex to consider is b.
(a, b) does not change the interval of a.
Its incoming non-tree edge: (d, b).

Add edge (d, b):
We add the interval [1,4] to d. Since [1,3] is contained by [1,4], we keep [1, 4].



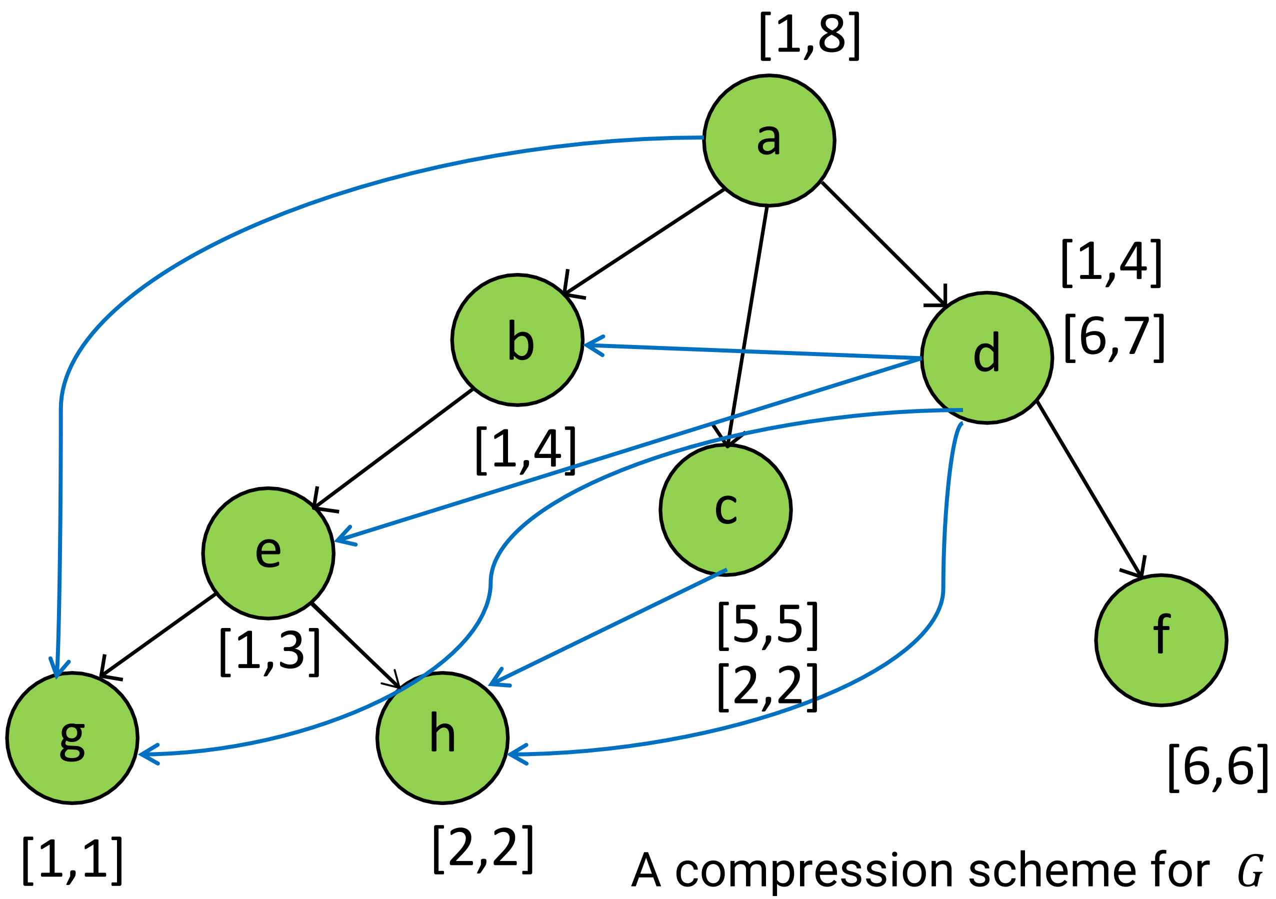
Tree Cover

The next vertex to consider is d.
(a, d) does not change the interval of a.

The next vertex to consider is a.
It does not have any incoming edges.

Done.

Question:
how many intervals are used in this compression scheme?



A compression scheme for G

Tree Cover

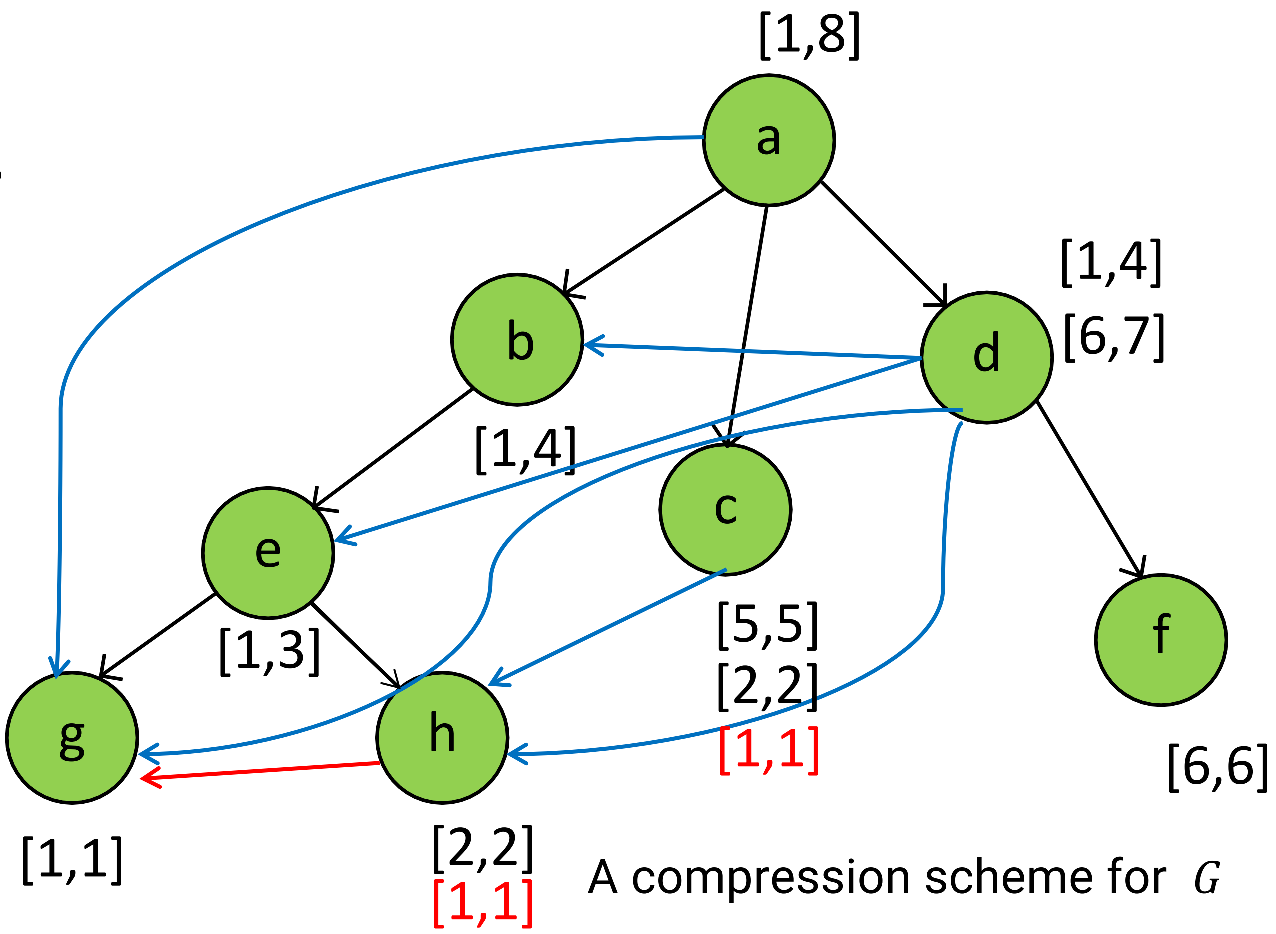
What if there is one more edge from h->g?

1. It will change the topological order (process g first then h)

2. Add the interval of g to h

3. When processing the incoming-edges of h, **remember to update the new intervals**!

How about h->f?



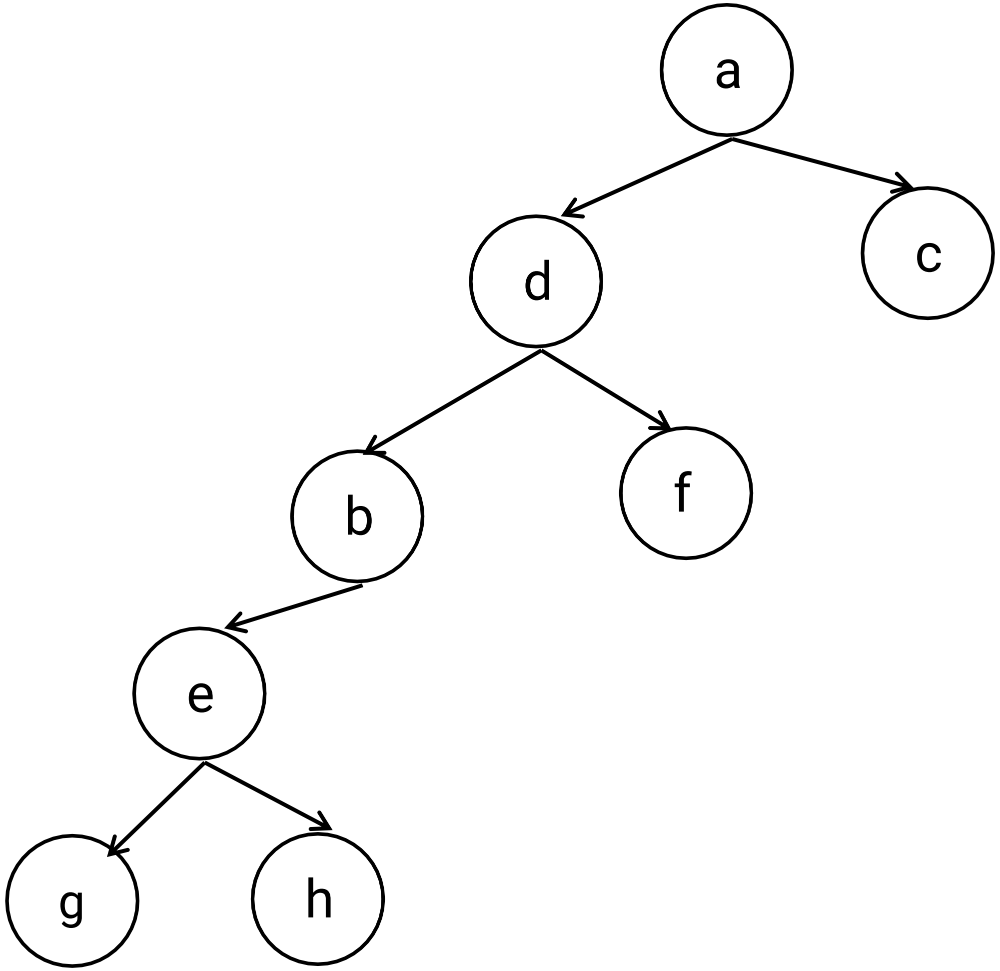
A compression scheme for G

Optimal Tree Cover

Question:
are all spanning trees (tree covers) equally good?

Optimality:
the tree cover with the minimum number of intervals in the resulting compression scheme.

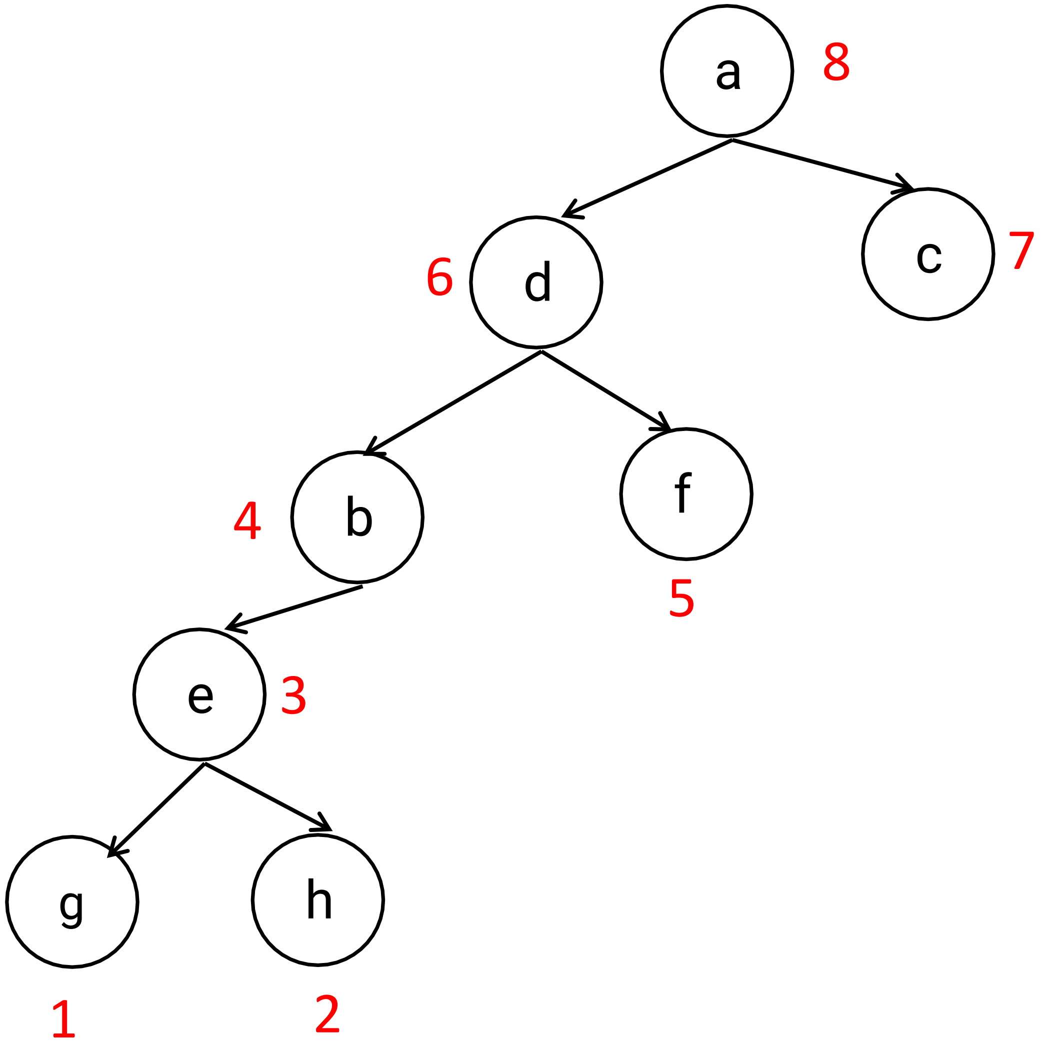
An optimal tree cover is shown here.
Construct the associated compression scheme.



An optimal tree cover

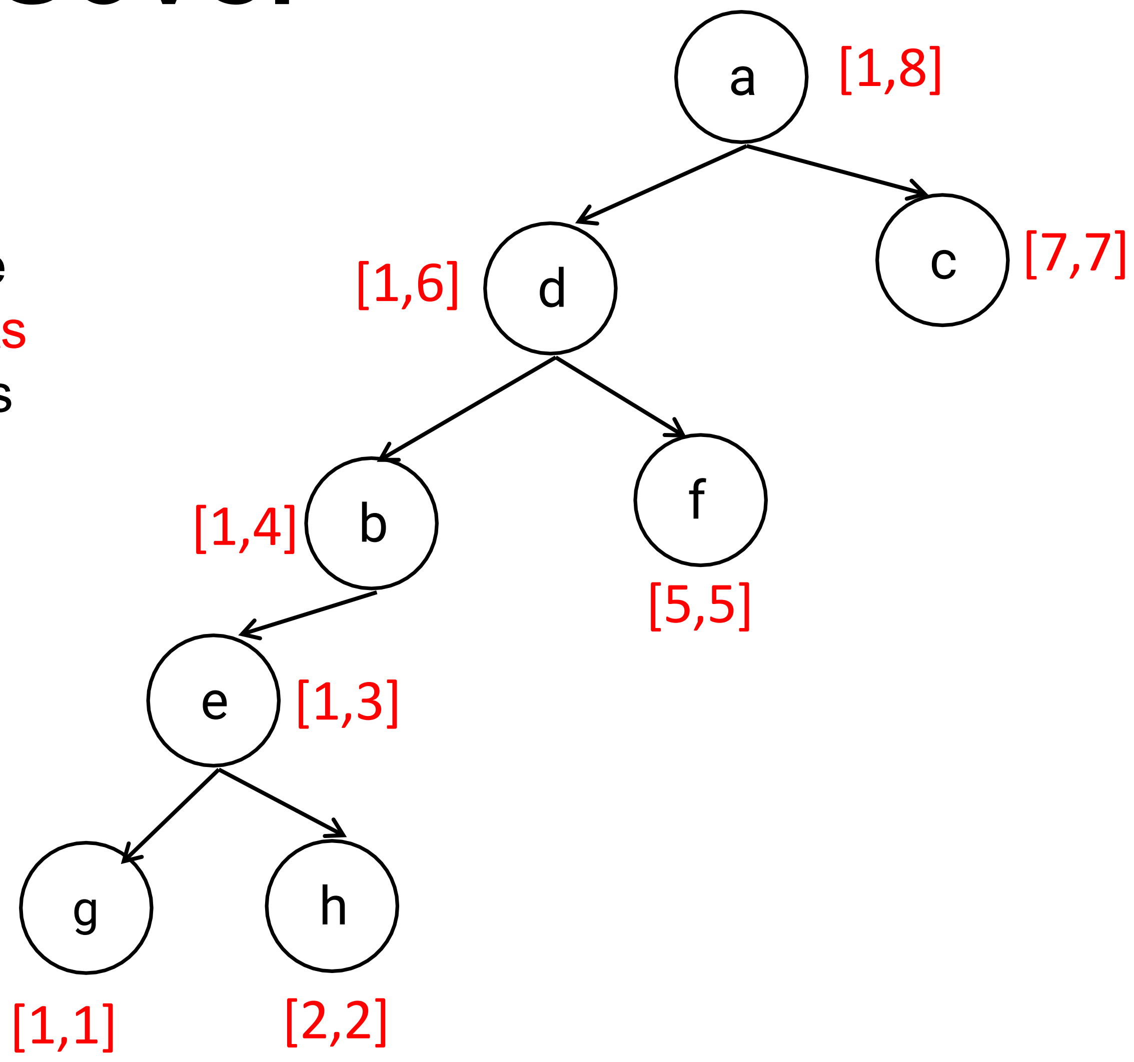
Optimal Tree Cover

Step 1:
Assign post-order number:



Optimal Tree Cover

Step 2:
For each vertex, we compute the minimum post-order number of its subtree and assign an interval as the reachability label.



Optimal Tree Cover

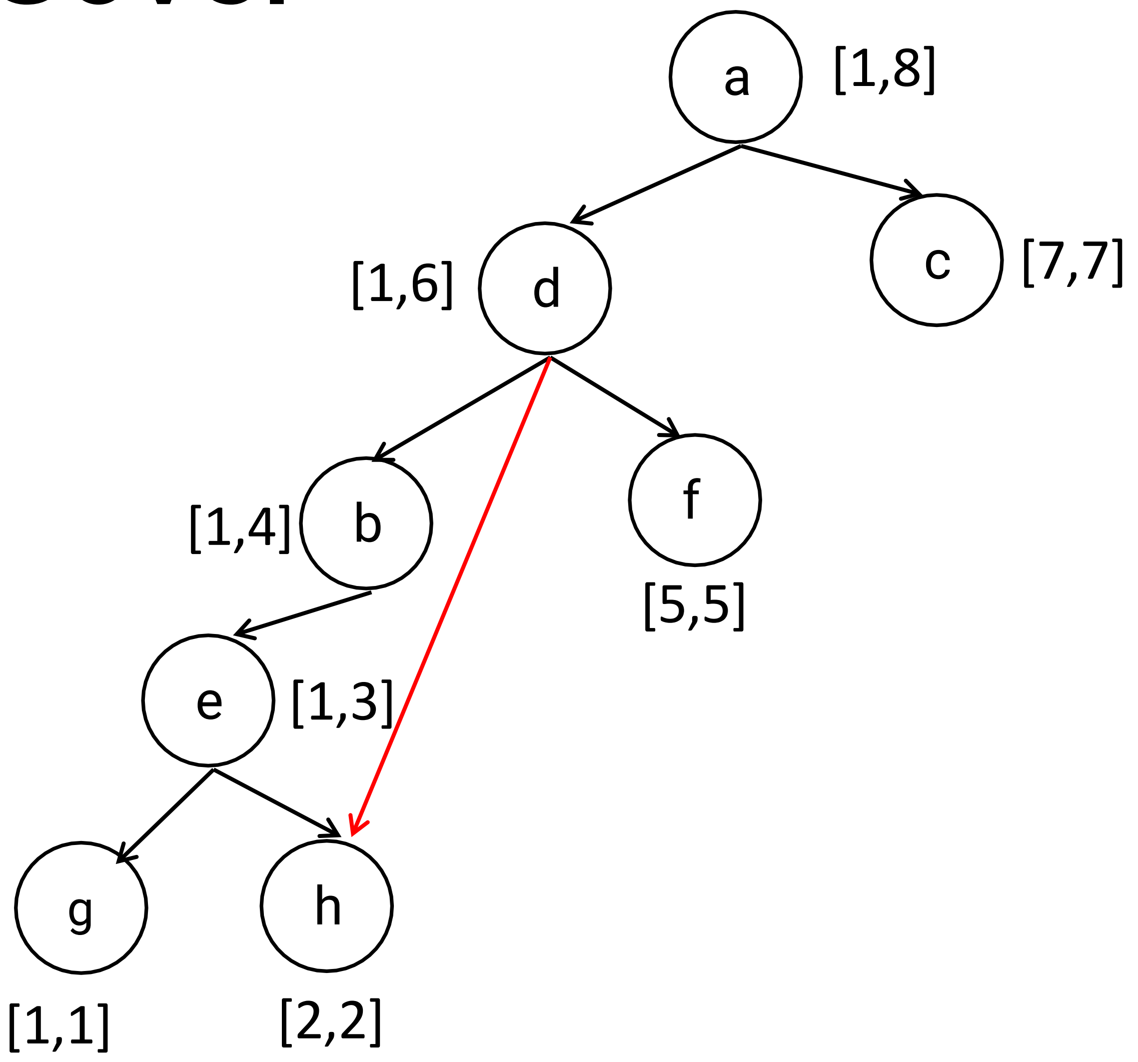
Follow **reverse topological order** and recover the **non-tree** edges.

A topological ordering of the vertices:
a, d, b, c, f, e, g, h

First consider vertex h, which has incoming non-tree edges (d, h), (c, h) and tree edge (e, h).

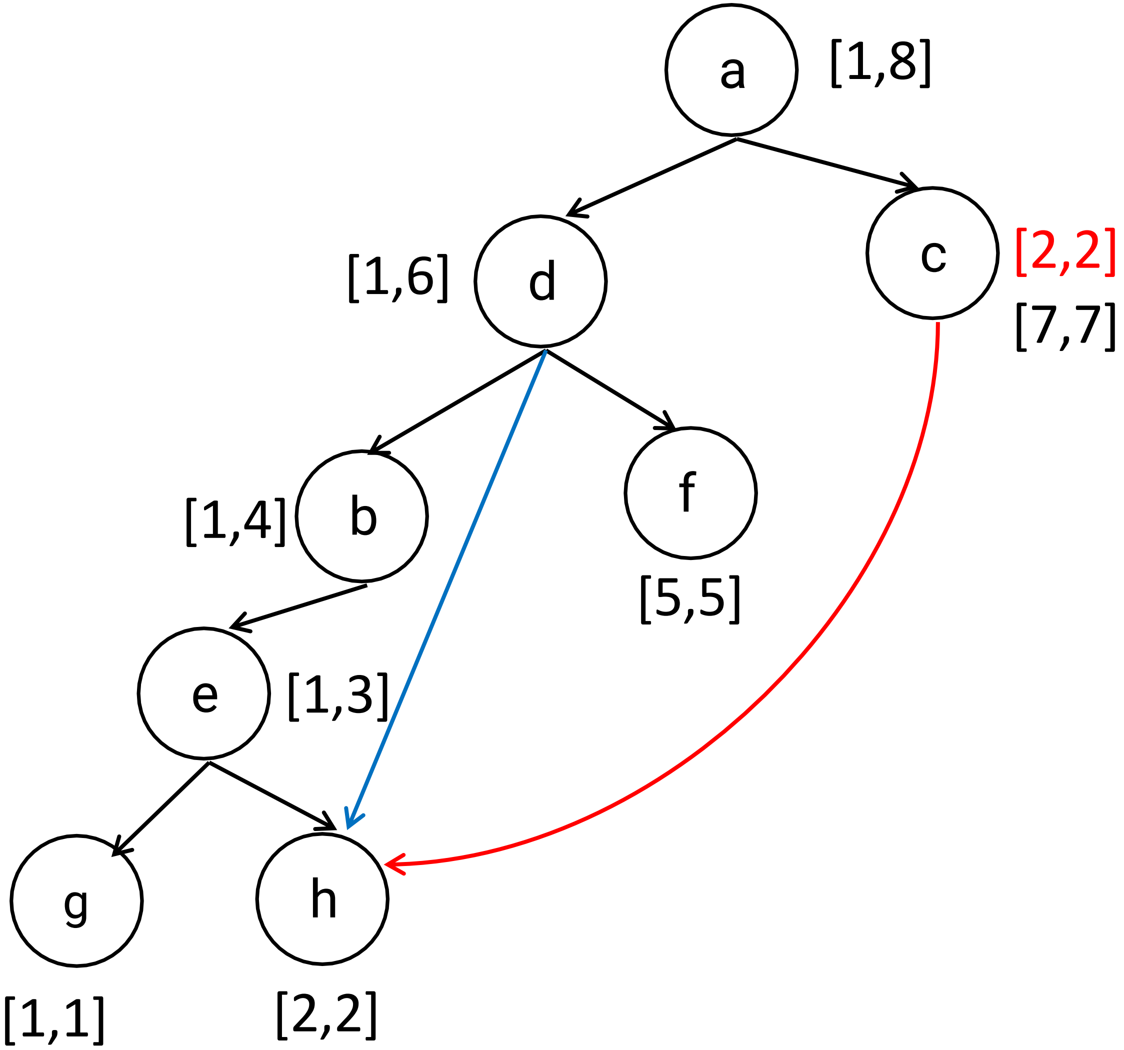
(e, h) does not change the interval of e.

Add edge (d, h):
We add the interval associated with h to d.
[2 ,2] is subsumed by [1, 6].



Optimal Tree Cover

Add edge (c, h):
We add the interval associated with h to c.

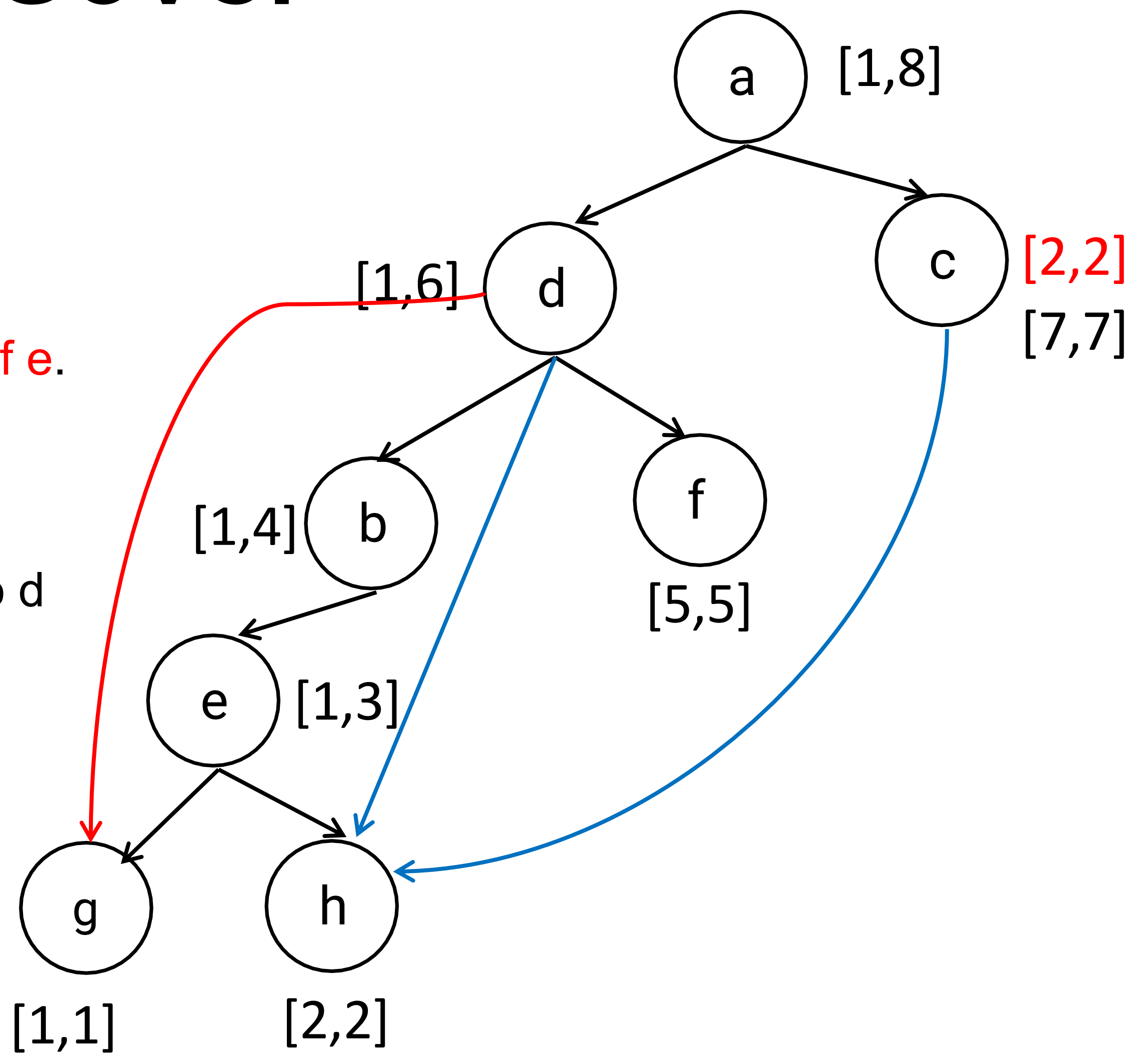


Optimal Tree Cover

The next vertex to consider is g.
We consider (d, g), and (a, g):

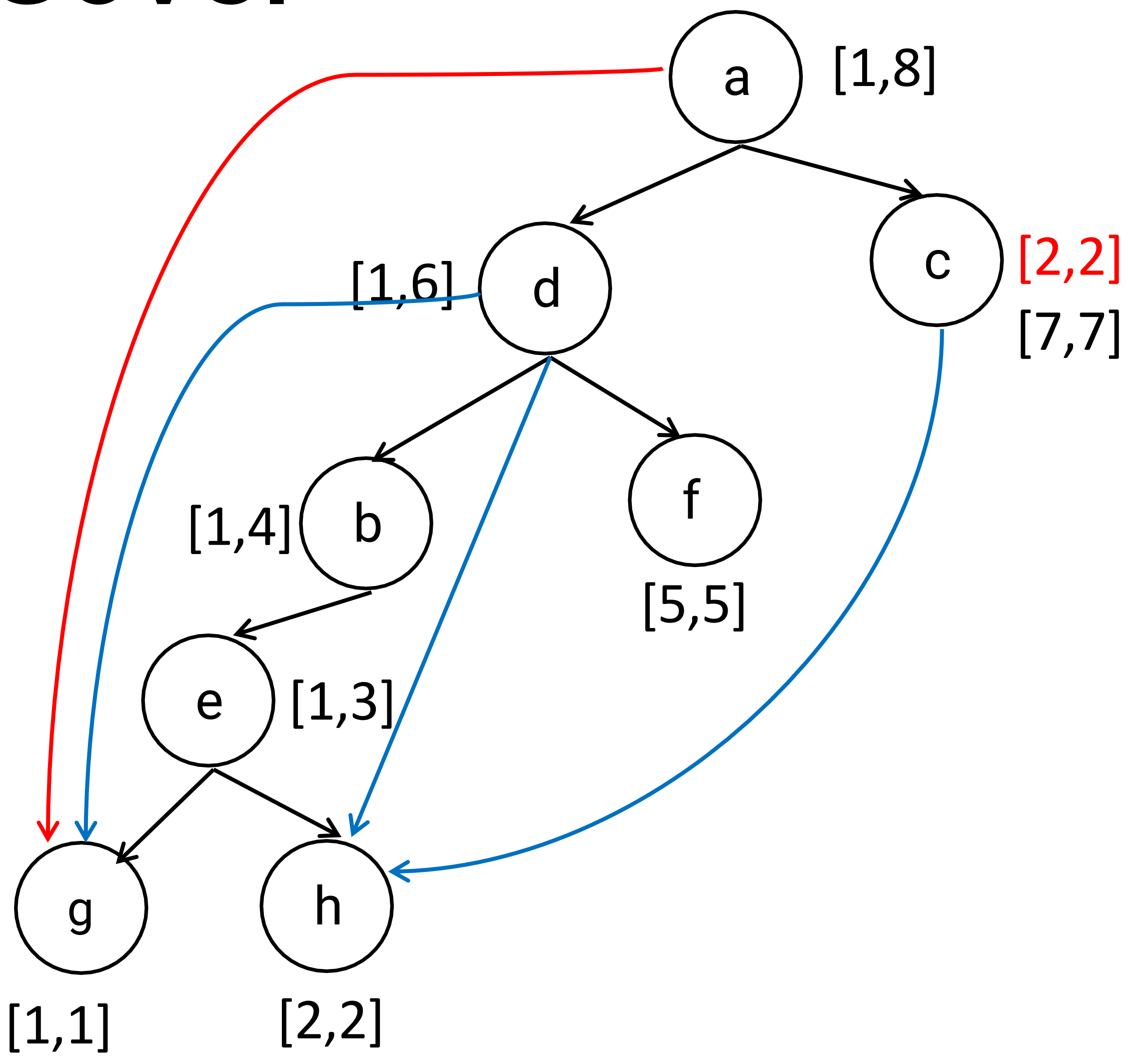
(e, g) does not change the interval of e.

Add edge (d, g):
We DO NOT add the interval [1,1] to d because it is subsumed by [1 ,6].



Optimal Tree Cover

Add edge (a, g):
We DO NOT add the interval [1,1] to a because it is subsumed by [1 ,8].

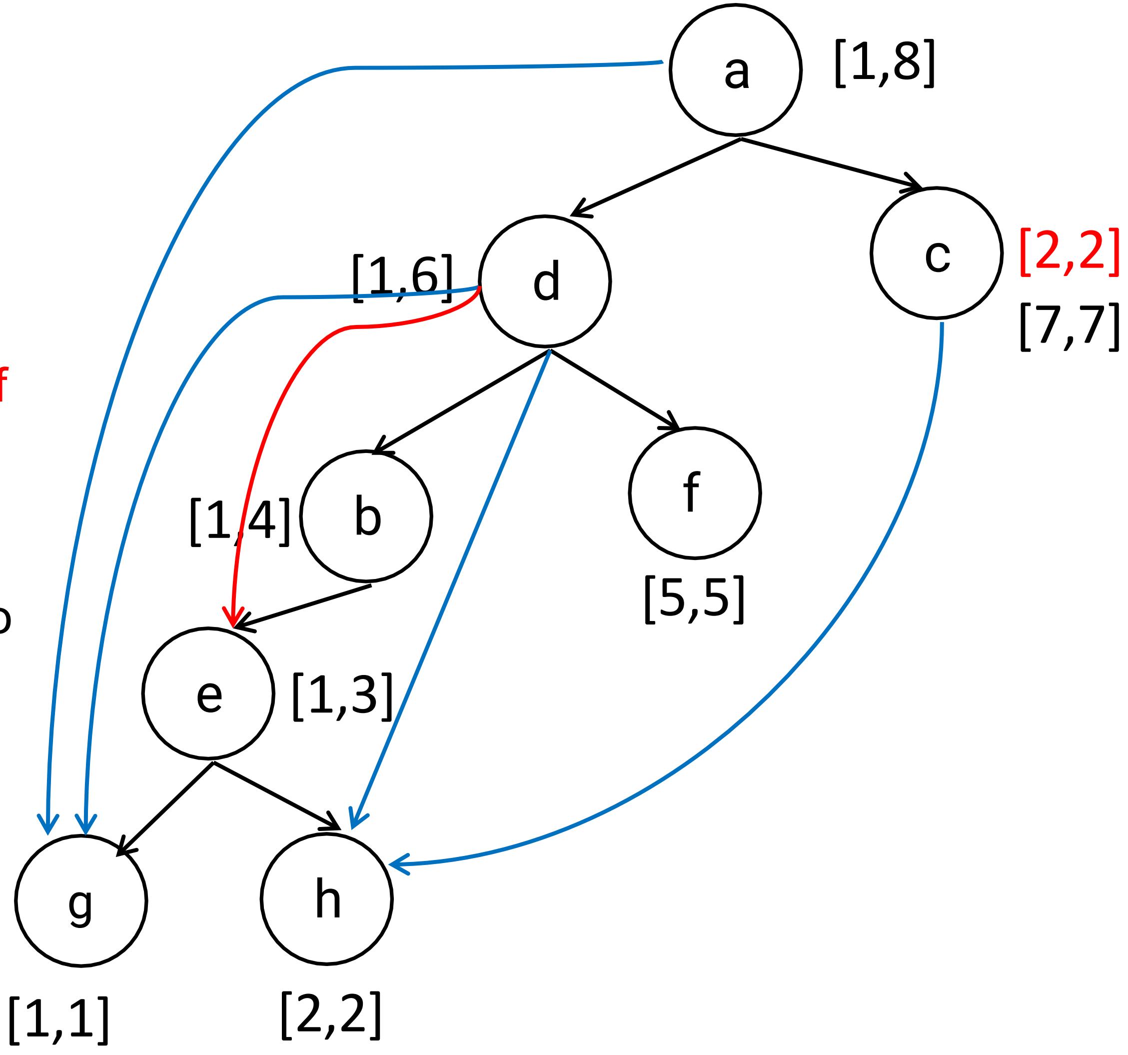


Optimal Tree Cover

The next vertex to consider is e.
We consider (b, e) and (d, e):

(b, e) does not change the interval of b.

Add edge (d, e):
We DO NOT add the interval [1, 3] to d because it is subsumed by [1, 6].



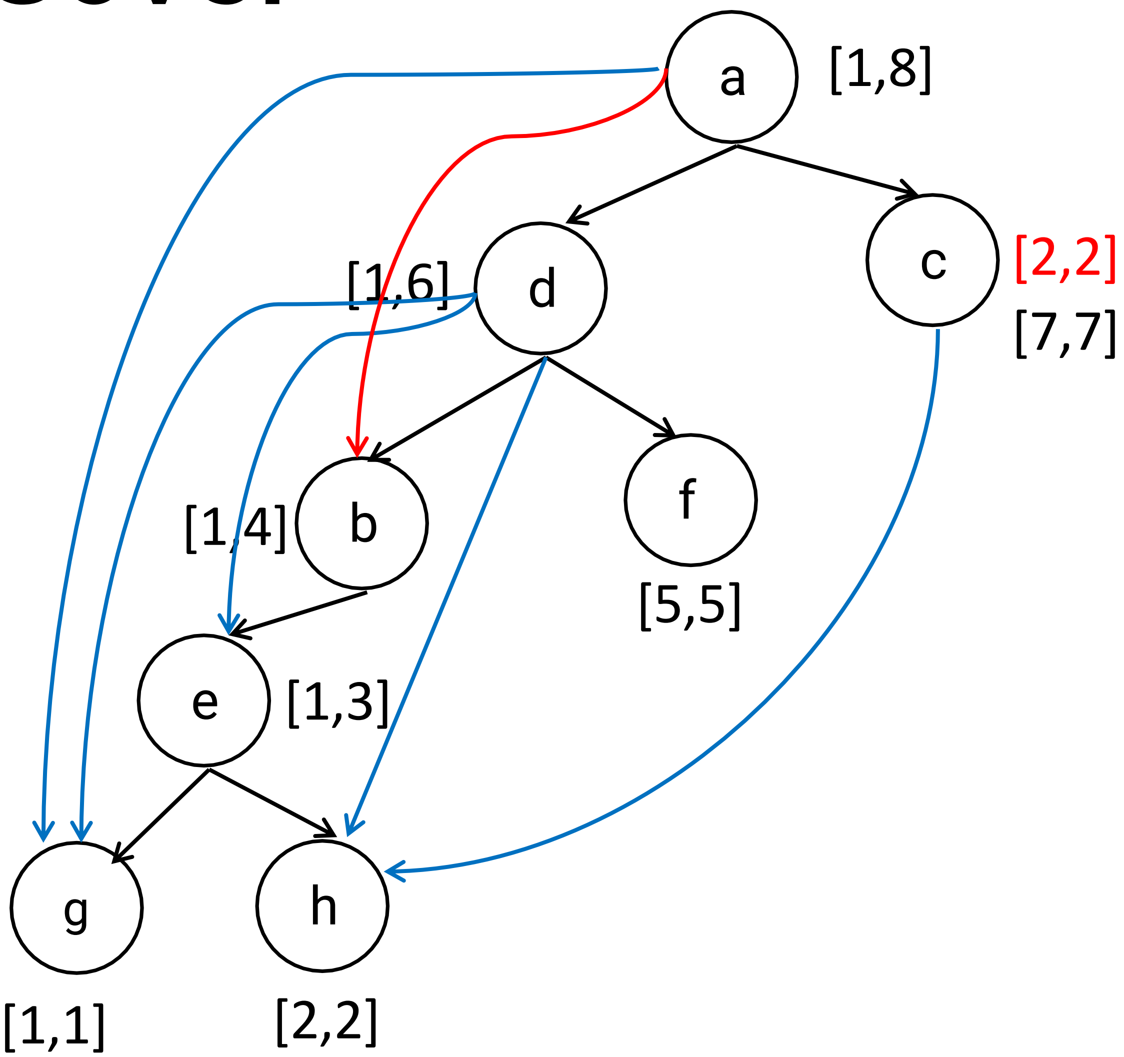
Optimal Tree Cover

The vertices f, c, and d do not have any incoming edges that can make any changes.

The next vertex to consider is b, which has one incoming non-tree edge (a, b) and tree-edge (d, b).

(d, b) does not change the interval of d.

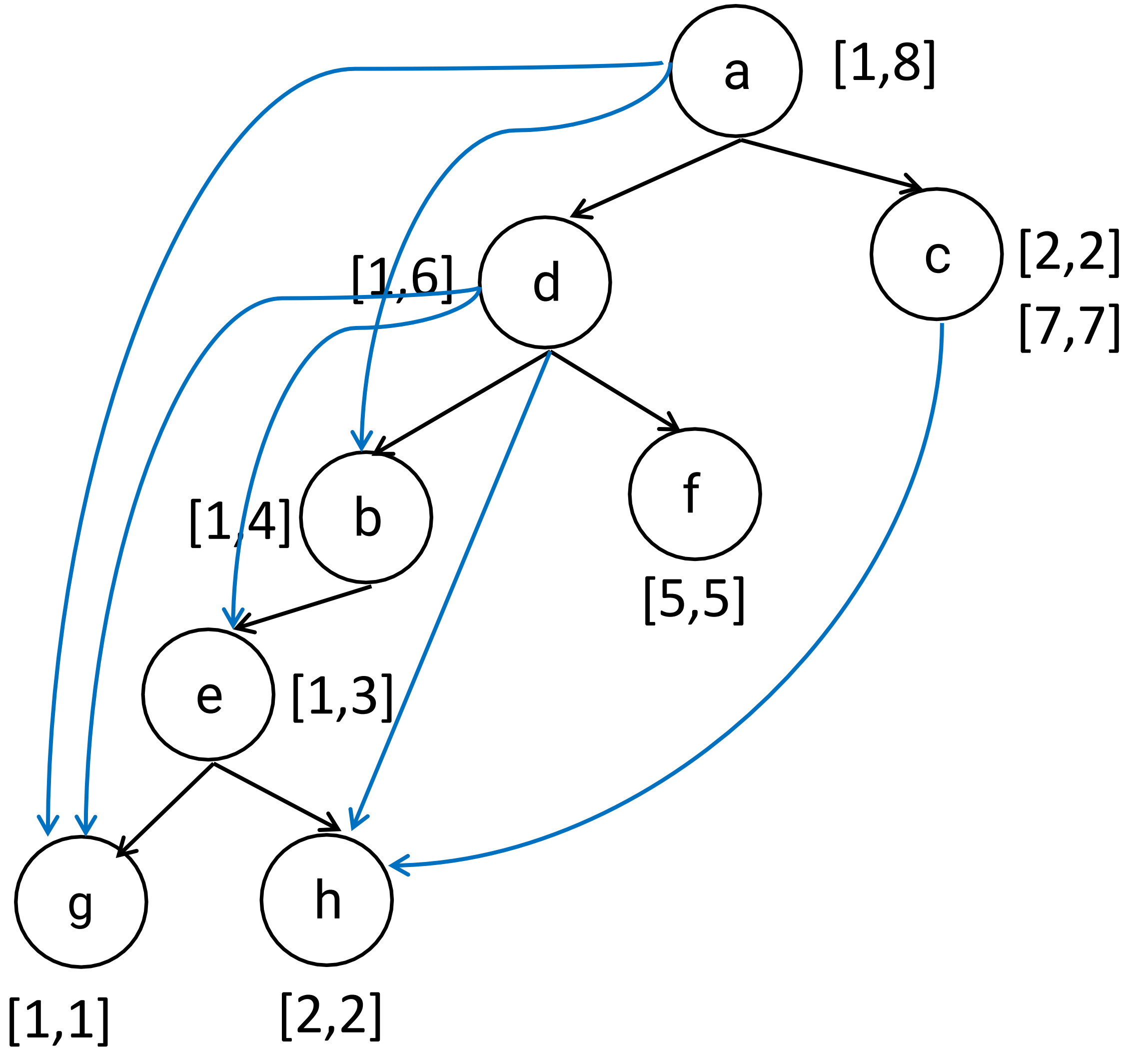
Add edge (a, b):
We DO NOT add the interval [1, 4] to a because it is subsumed by [1, 8].



Optimal Tree Cover

Question:
how many intervals are used in this compression scheme?

Question:
Compared to the previous compression scheme, what do you observe?

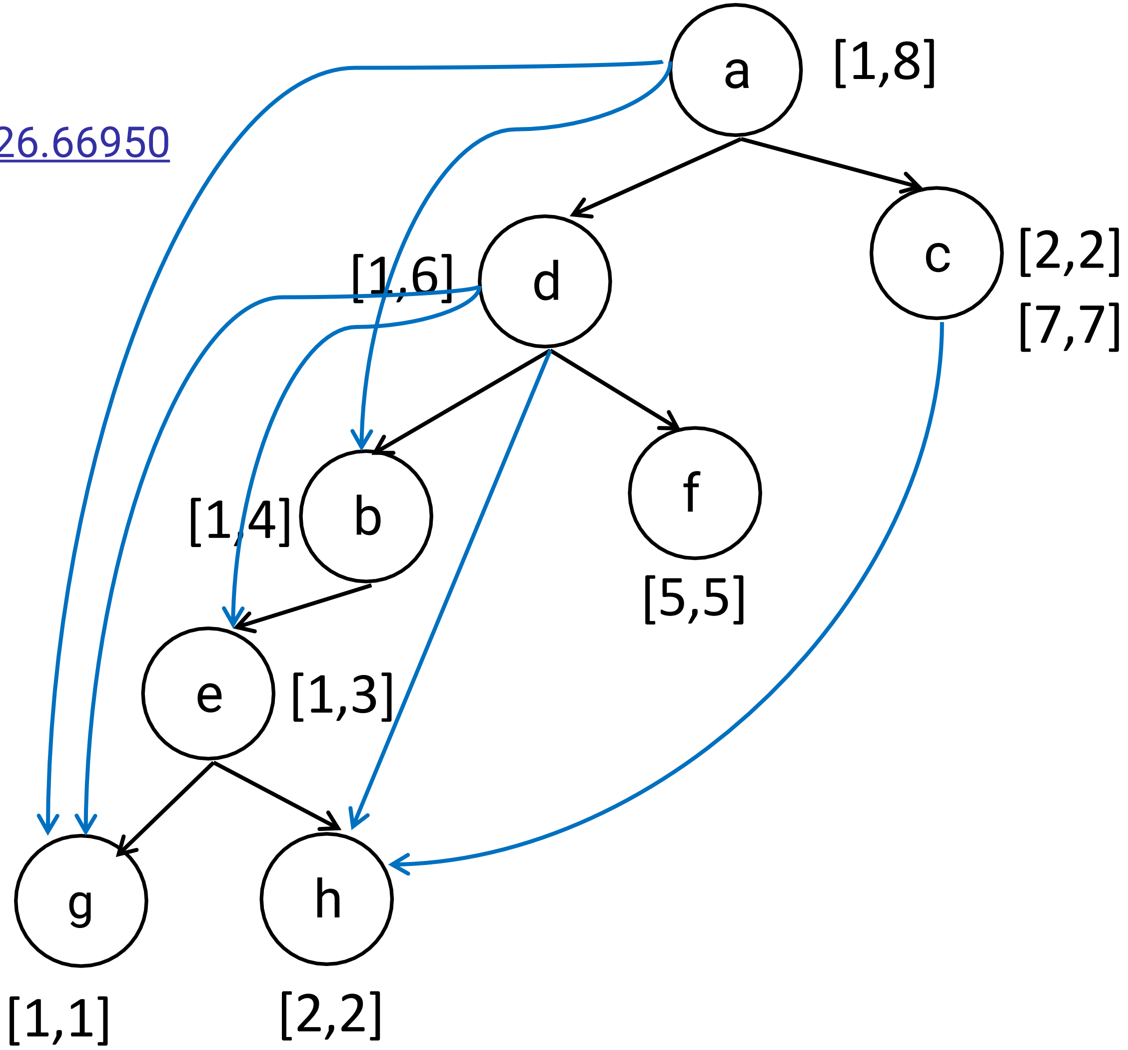


Computing Optimal Tree Cover

https://dl.acm.org/doi/pdf/10.1145/66926.66950

Intuition: Make the tree like a path (unbalanced)

How to compute optimal tree cover is optional.



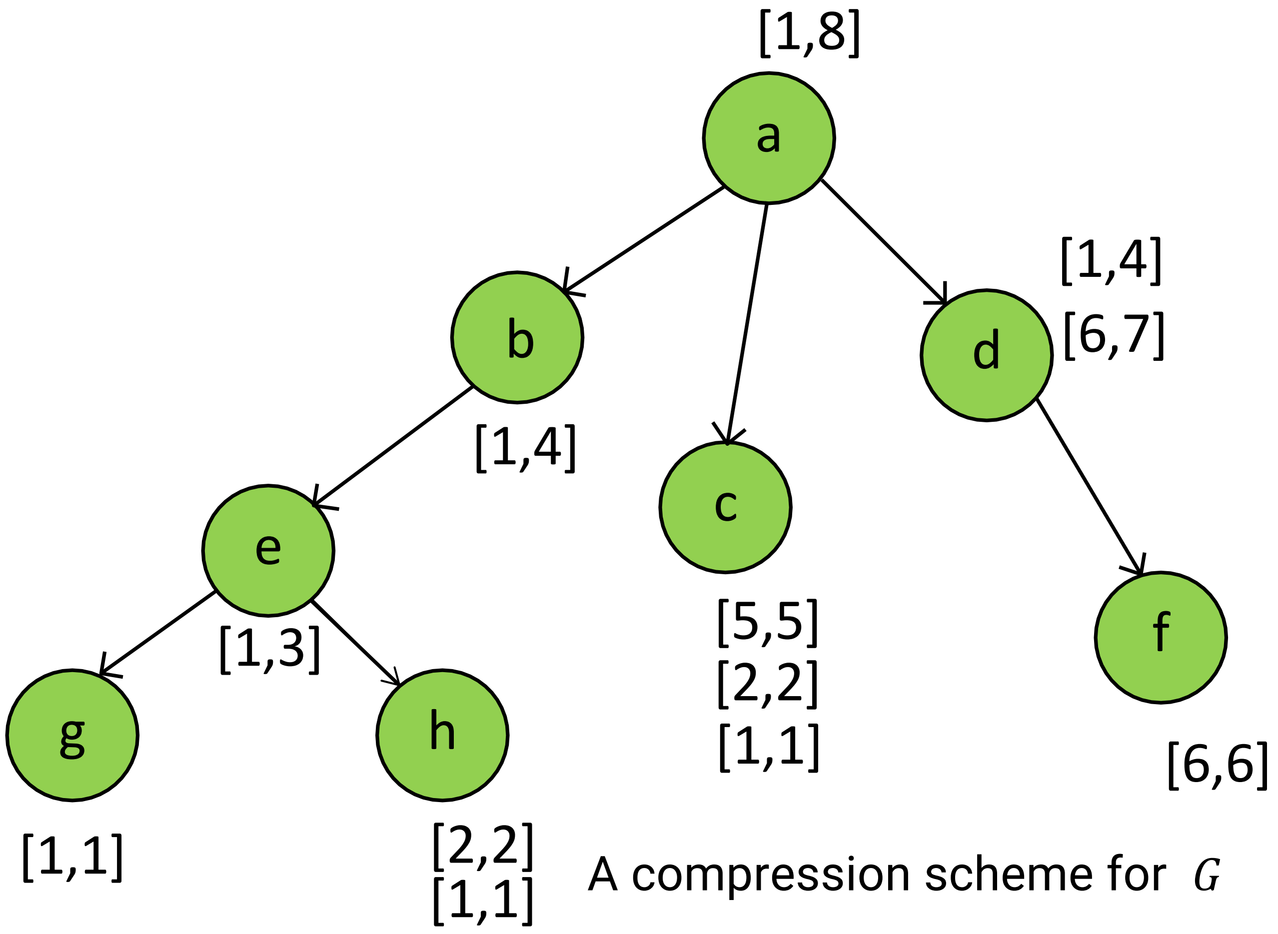
Optimal Tree is not optimal?

Practical Optimization:

[2,2]
[1,1] ➡ [1,2]

The number of intervals in the optimal tree cover may not be the smallest when merging intervals are allowed.

Do not need to merge intervals in assignment/exam.



A compression scheme for G

Complexity analysis

Query time: $O(n)$
Each vertex u has at most n intervals. Iterate through them and check if v is contained by one of them.

Index construction time: $O(n \times n \times m)$
The dominating cost: for each non-tree edge (u, v), attach the intervals of v to u, which takes $O(n)$ time. We need to update labels for all ancestors of u if necessary. The number of non-tree edges is bounded by $O(m)$.

Space complexity: $O(n^2)$
In the worst case, the space complexity of a tree cover is the same as the transitive closure, but in practice its storage cost is much smaller.

Tree Cover results

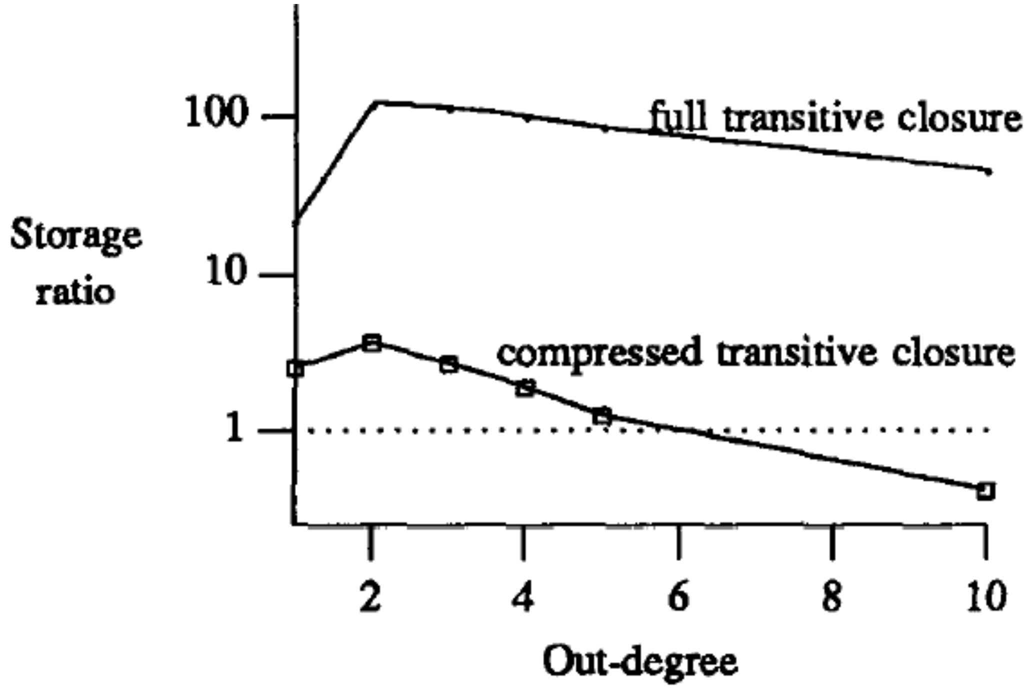


Figure 3.9. Storage required for a 1000 node graph as a function of average degree

Two-Hop Cover

2-Hop Cover SODA'02

An index which compresses transitive closure…

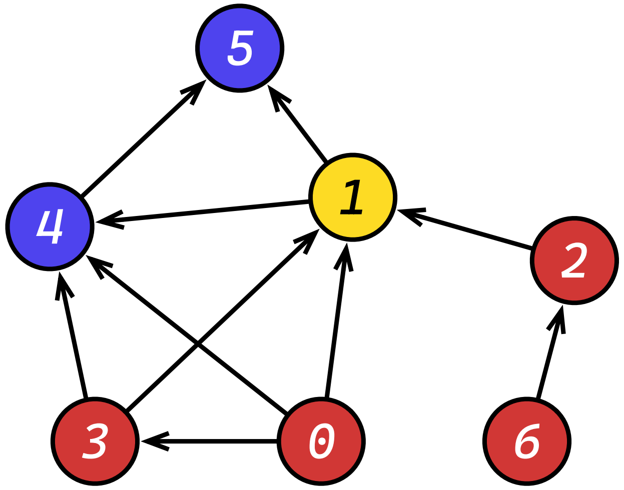
Intuition: if we choose a node u as a center node, then all u's ancestors can reach u's descendants.

2-Hop Cover

An index which compresses transitive closure…

Intuition: if we choose a node u as a center node, then all u's ancestors can reach u's descendants.

Example: So if we choose node 1 as a center node, each of its ancestors of {0, 2, 3, 6} can reach any node in its descendants of {4,5}



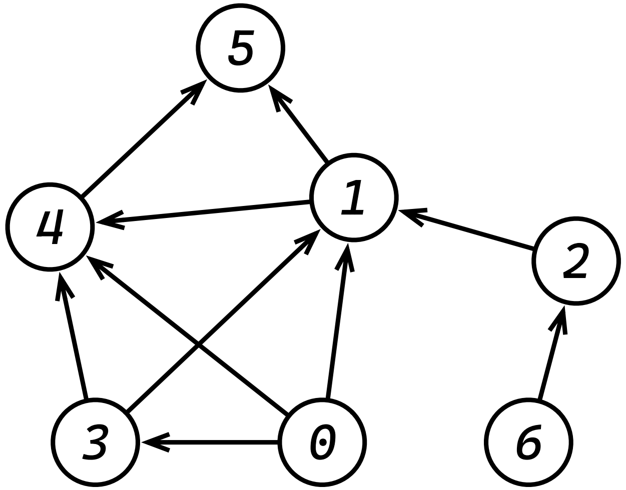
2-Hop Cover

Based on that, we can label nodes as follows:

- each node u is assigned two label sets $L_{in}(u) \subseteq V$ and $L_{out}(u) \subseteq V$
- for each $v \in L_{out}(u)$, it indicates that node u reaches node v.
- for each $v' \in L_{in}(u)$, it indicates that node v' reaches node u.

A 2-hop cover includes two label sets L_{out} and L_{in} that can cover all the edges in TC(G)….

A possible 2-hop cover



	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1,4}	{1}	{1,2}	{1,3,4}	{4}	{5}	{1,2,6}

2-Hop Cover: Query Processing

Now reachability queries can be answered using the labels:

- ? $u \leadsto v$

  if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ then return true

  if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then return false

2-Hop Cover: Query Processing

Now reachability queries can be answered using the labels:

– ? $u \rightsquigarrow v$

if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ then return true

if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then return false

– Time complexity is $O(|L_{out}(u)| + |L_{in}(v)|)$

More about time complexity:
$O(|L_{out}(u)| + |L_{in}(v)|)$: Hash table
$O(\log(|L_{out}(u)|)|L_{out}(u)| + \log(|L_{in}(v)|)|L_{in}(v)|)$: Sort-merge join
$O(\min(|L_{out}(u)|, |L_{in}(v)|))$: Precomputed hash table
$O(|L_{out}(u)| + |L_{in}(v)|)$: Precomputed order + merge join

2-Hop Cover: Query Processing

Now reachability queries can be answered using the labels:

– ? $u \leadsto v$

if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ then return true

if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then return false

	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1,4}	{1}	{1,2}	{1,3,4}	{4}	{5}	{1,2,6}

For example,

? $0 \leadsto 5$

2-Hop Cover: Query Processing

Now reachability queries can be answered using the labels:

– ? $u \leadsto v$

if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ then return true

if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then return false

	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1,4}	{1}	{1,2}	{1,3,4}	{4}	{5}	{1,2,6}

For example,

? $0 \leadsto 5$ $\quad$ $L_{out}(0) \cap L_{in}(5) = \{1,4,0\} \cap \{1,4,5\} \neq \emptyset$ YES

2-Hop Cover: Query Processing

Now reachability queries can be answered using the labels:

– $? \, u \rightsquigarrow v$

if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ then return true

if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then return false

	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1,4}	{1}	{1,2}	{1,3,4}	{4}	{5}	{1,2,6}

For example,

$? \, 0 \rightsquigarrow 5$ $L_{out}(0) \cap L_{in}(5) = \{1,4,0\} \cap \{1,4,5\} \neq \emptyset$ YES

$? \, 0 \rightsquigarrow 2$

2-Hop Cover: Query Processing

Now reachability queries can be answered using the labels:

– $? \; u \rightsquigarrow v$

if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ then return true

if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then return false

	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1,4}	{1}	{1,2}	{1,3,4}	{4}	{5}	{1,2,6}

For example,

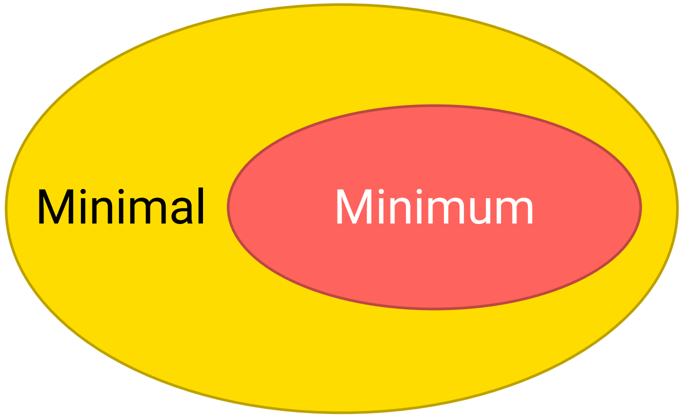
$? \; 0 \rightsquigarrow 5$ $\quad L_{out}(0) \cap L_{in}(5) = \{1,4,0\} \cap \{1,4,5\} \neq \emptyset$ YES

$? \; 0 \rightsquigarrow 2$ $\quad L_{out}(0) \cap L_{in}(2) = \{1,4,0\} \cap \{2\} = \emptyset$ NO

2-Hop Cover Index: Minimum VS Minimal

When we say something is **minimum**, that means it is the globally smallest.

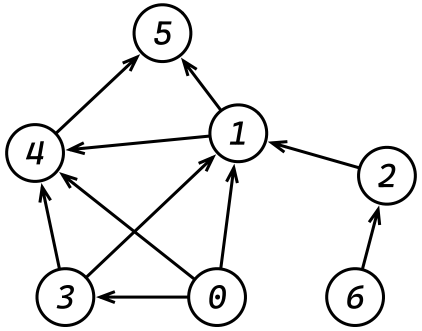
When we say some thing is **minimal**, that means it has no redundancy.



Conceptually, using all reachable vertices as the label is also a 2-hop cover index, but it is not minimal.

Compute the minimum 2-hop cover index is NP-hard.

2-Hop Cover: the minimal index



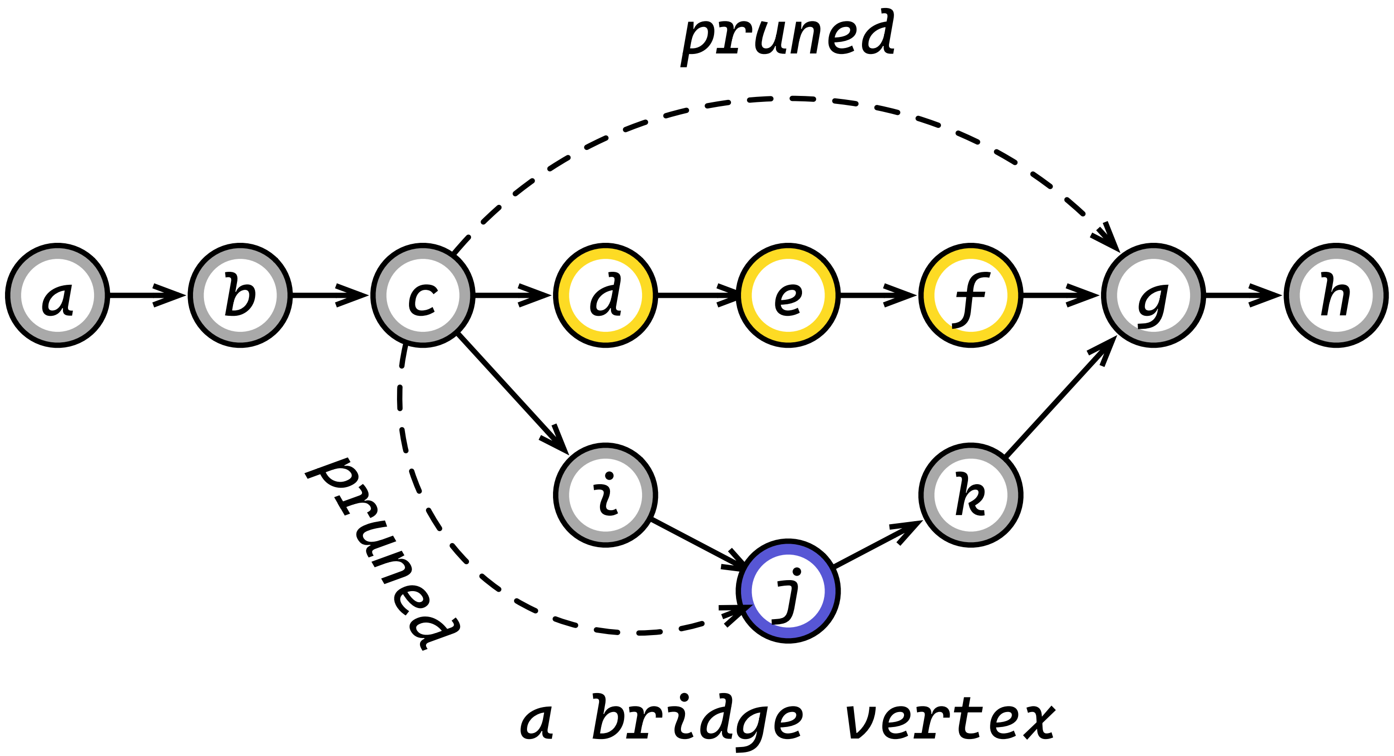
	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1,**4**}	{1}	{1,2}	{1,3,**4**}	{4}	{5}	{1,2,6}

Naive Index

	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{0,1}	{1}	{1,2}	{1,3}	{4}	{5}	{1,2,6}

Minimal Index

Motivation



Total-order-based 2-Hop Cover

An algorithm to compute a minimal 2-hop cover

For each node u in the graph from high-degree to low-degree:

- *add u into both $L_{in}(u)$ and $L_{out}(u)$;*
- *mark u as processed;*
- *conduct BFS from u and for each reached node w:*
  - *if (u,w) has been covered: stop exploring out-neighbors of w;*
  - *else: add u into $L_{in}(w)$;*
- *conduct reverse BFS from u and for each reached node w':*
  - *if (w',u) has been covered: stop exploring in-neighbors of w';*
  - *else: add u into $L_{out}(w')$;*

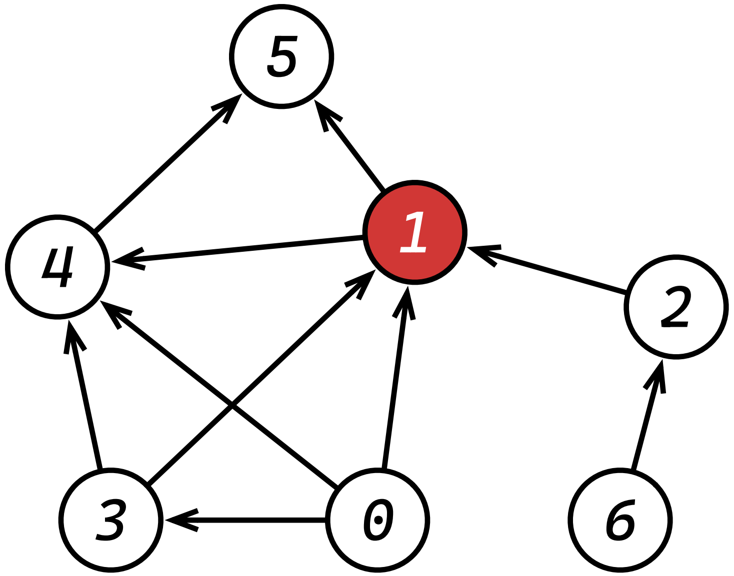
2-Hop Cover - Example

After choosing node 1, we add it at

$L_{in}(1),\ L_{out}(1)$

$L_{in}(4),\ L_{in}(5)$

$L_{out}(0),\ L_{out}(2),\ L_{out}(3), L_{out}(6)$



	0	1	2	3	4	5	6
L_{in}		{1}			{1}	{1}	
L_{out}	{1}	{1}	{1}	{1}			{1}

2-Hop Cover - Example

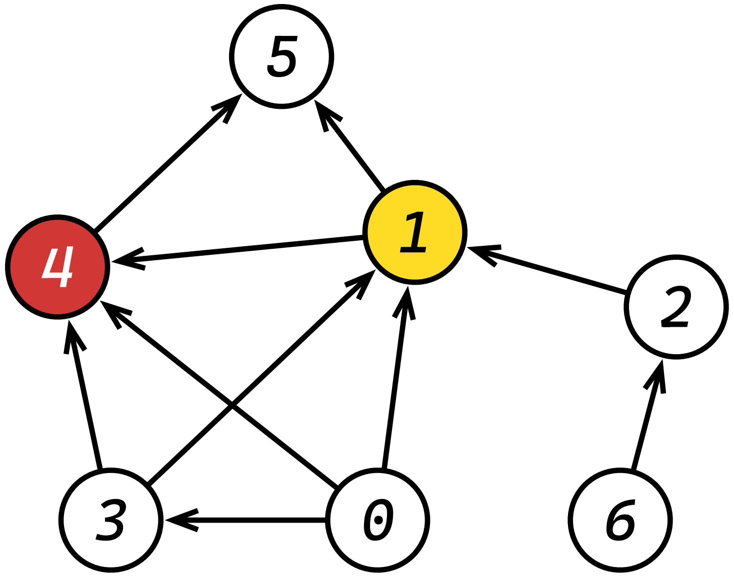
Then we choose node 4, we add it at

$L_{in}(4), L_{out}(4)$

$L_{in}(5)$

$L_{out}(0), L_{out}(3)$

$(0,4)$ *is covered by* 1
$(3,4)$ *is covered by* 1



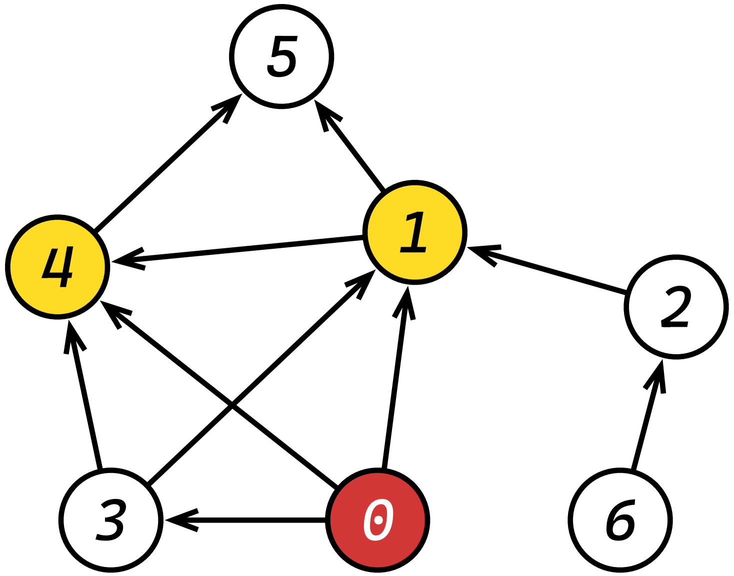
	0	1	2	3	4	5	6
L_{in}		{1}			{1,4}	{1,4}	
L_{out}	{1}	{1}	{1}	{1}	{4}		{1}

2-Hop Cover - Example

Then we choose node 0, we add it at

$L_{in}(0), L_{out}(0)$

$L_{in}(3)$

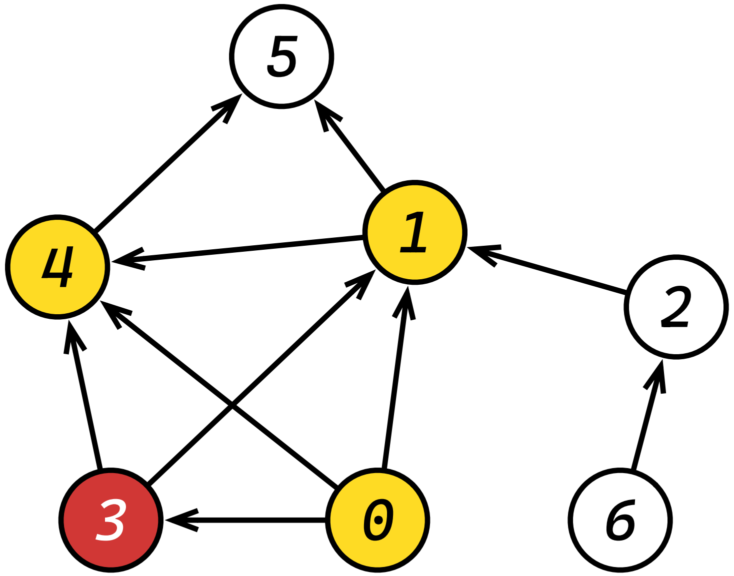


	0	1	2	3	4	5	6
L_{in}	{0}	{1}		{0}	{1,4}	{1,4}	
L_{out}	{1,0}	{1}	{1}	{1}	{4}		{1}

2-Hop Cover - Example

Then we choose node 3, we add it at

$$L_{in}(3), L_{out}(3)$$



	0	1	2	3	4	5	6
L_{in}	{0}	{1}		{0,3}	{1,4}	{1,4}	
L_{out}	{1,0}	{1}	{1}	{1,3}	{4}		{1}

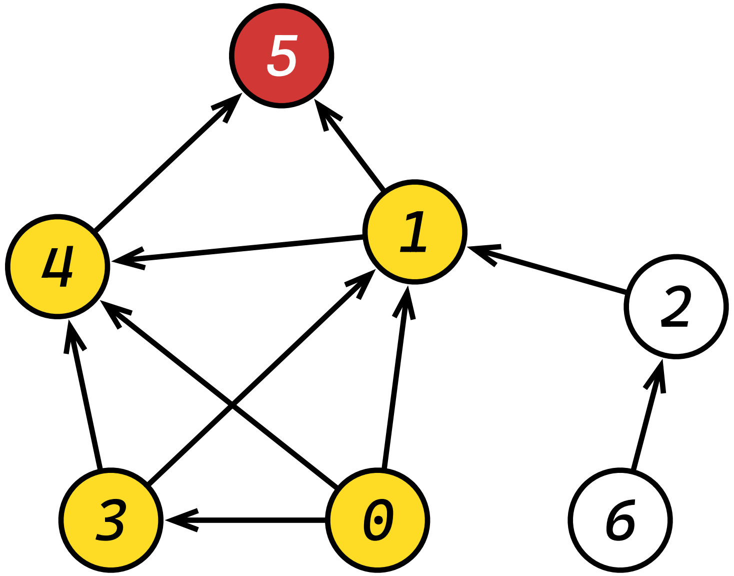
2-Hop Cover - Example

Then we choose node 3, we add it at

$$L_{in}(3), L_{out}(3)$$

Then we choose node 5, we add it at

$$L_{in}(5), L_{out}(5)$$

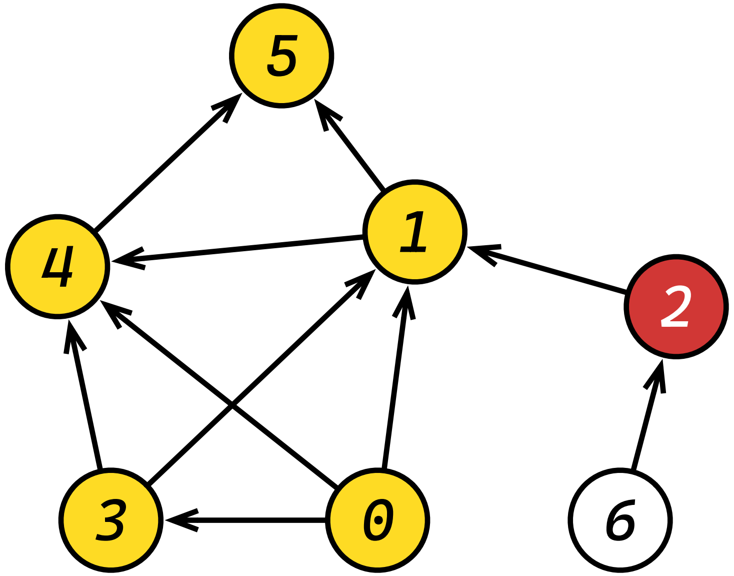


	0	1	2	3	4	5	6
L_{in}	{0}	{1}		{0,3}	{1,4}	{1,4,5}	
L_{out}	{1,0}	{1}	{1}	{1,3}	{4}	{5}	{1}

2-Hop Cover - Example

Then we choose node 2, we add it at

$$L_{in}(2), L_{out}(2), L_{out}(6)$$



	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	
L_{out}	{1,0}	{1}	{1,2}	{1,3}	{4}	{5}	{1,2}

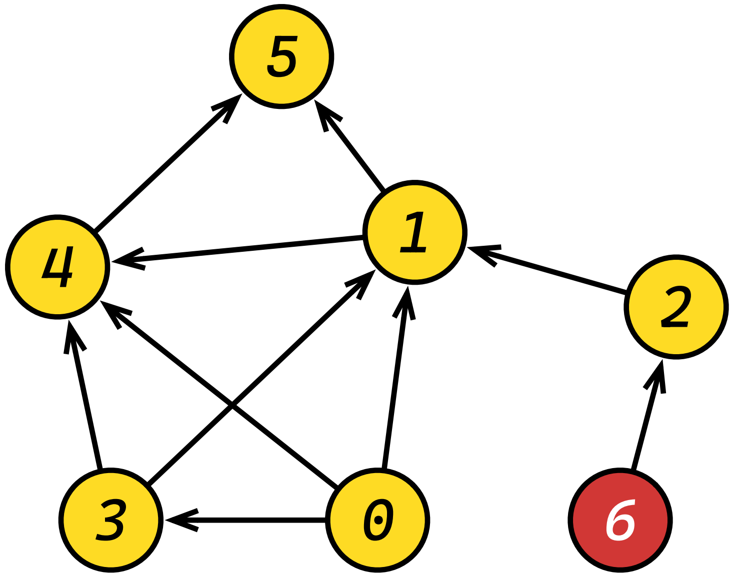
2-Hop Cover - Example

Then we choose node 2, we add it at

$$L_{in}(2), L_{out}(2), L_{out}(6)$$

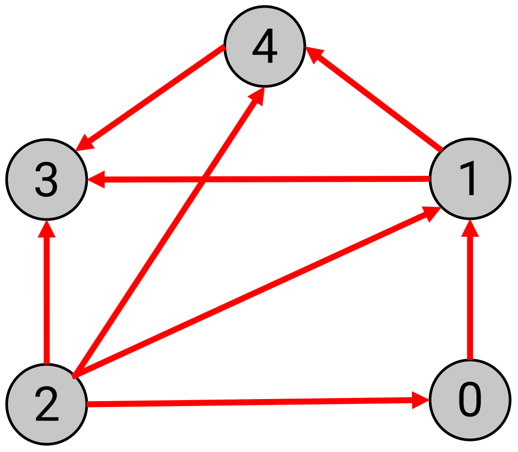
Finally we choose node 6, we add it at

$$L_{in}(6), L_{out}(6)$$



	0	1	2	3	4	5	6
L_{in}	{0}	{1}	{2}	{0,3}	{1,4}	{1,4,5}	{6}
L_{out}	{1,0}	{1}	{1,2}	{1,3}	{4}	{5}	{1,2,6}

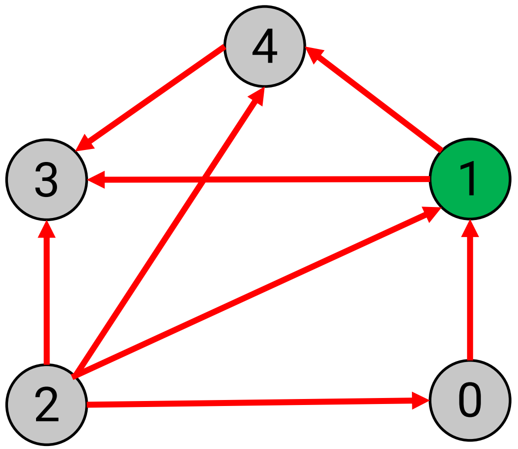
Quick Exercise



1. Can you compute the 2-hop cover of this graph?
   - Note that you need to process the nodes in the order of 1, 2, 4, 3, 0

2. Based on the computed 2-hop cover, please compute $?\ 0 \rightsquigarrow 2$ and $?\ 1 \rightsquigarrow 3$

Quick Exercise



We start with 1 and add it in

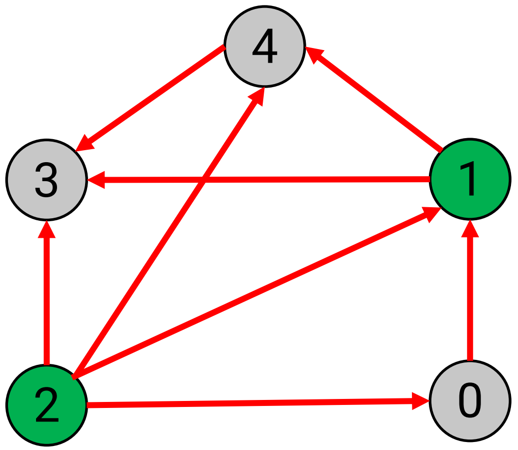
$$L_{in}(1),\ L_{out}(1)$$

$$L_{in}(4),\ L_{in}(3)$$

$$L_{out}(0),\ L_{out}(2)$$

	0	1	2	3	4
L_{in}		{1}		{1}	{1}
L_{out}	{1}	{1}	{1}		

Quick Exercise



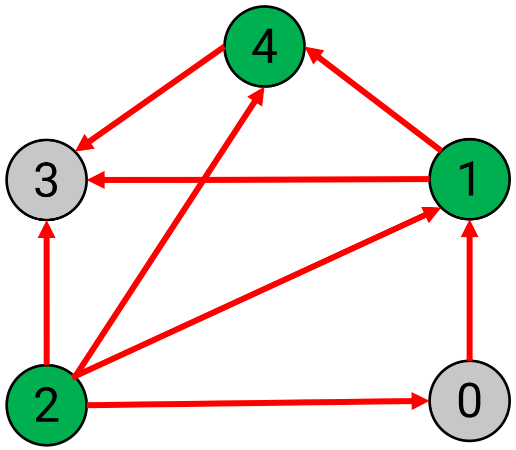
Then, we process 2 and add it in

$$L_{in}(2),\ L_{out}(2)$$

$$L_{in}(0)$$

	0	1	2	3	4
L_{in}	{2}	{1}	{2}	{1}	{1}
L_{out}	{1}	{1}	{1, 2}		

Quick Exercise



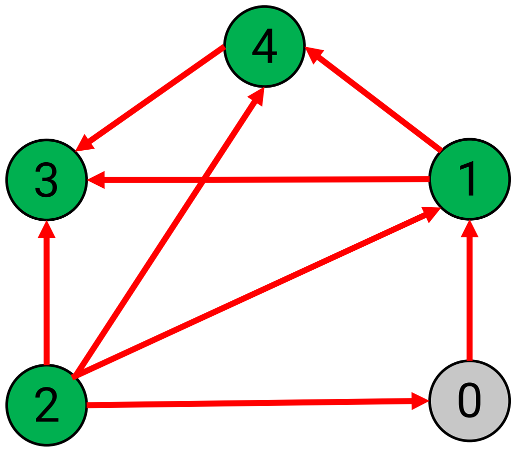
Then, we process 4 and add it in

$L_{in}(4),\ L_{out}(4)$

$L_{in}(3)$

	0	1	2	3	4
L_{in}	{2}	{1}	{2}	{1,4}	{1, 4}
L_{out}	{1}	{1}	{1, 2}		{4}

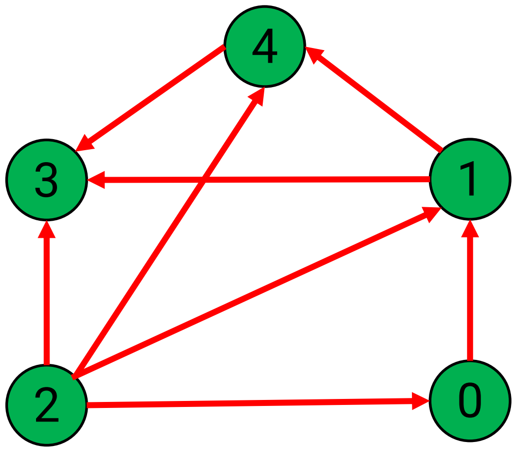
Quick Exercise



Then, we process 3 and add it in
$L_{in}(3),\ L_{out}(3)$

	0	1	2	3	4
L_{in}	{2}	{1}	{2}	{1, 4, 3}	{1, 4}
L_{out}	{1}	{1}	{1, 2}	{3}	{4}

Quick Exercise



Then, we process 0 and add it in
$L_{in}(0),\ L_{out}(0)$

	0	1	2	3	4
L_{in}	{2, 0}	{1}	{2}	{1, 4, 3}	{1, 4}
L_{out}	{1, 0}	{1}	{1, 2}	{3}	{4}

Quick Exercise

	0	1	2	3	4
L_{in}	{2, 0}	{1}	{2}	{1, 4, 3}	{1, 4}
L_{out}	{1, 0}	{1}	{1, 2}	{3}	{4}

? $0 \leadsto 2$ $\quad L_{out}(0) \cap L_{in}(2) = \{1, 0\} \cap \{2\} = \emptyset$ $\quad$ NO

? $1 \leadsto 3$ $\quad L_{out}(1) \cap L_{in}(3) = \{1\} \cap \{1, 4, 3\} \neq \emptyset$ $\quad$ YES

Learning Outcome

- Know the difference between transitive closure, tree cover, and two-Hop labelling.

- Know how to construct transitive closure, tree cover, and two-Hop labelling. In addition, how to compute the reachability queries using these structures.