

Introduction to

Algorithm Design and Analysis

[12] Directed Acyclic Graph

Jingwei Xu

<https://ics.nju.edu.cn/~xjw/>

Institute of Computer Software
Nanjing University

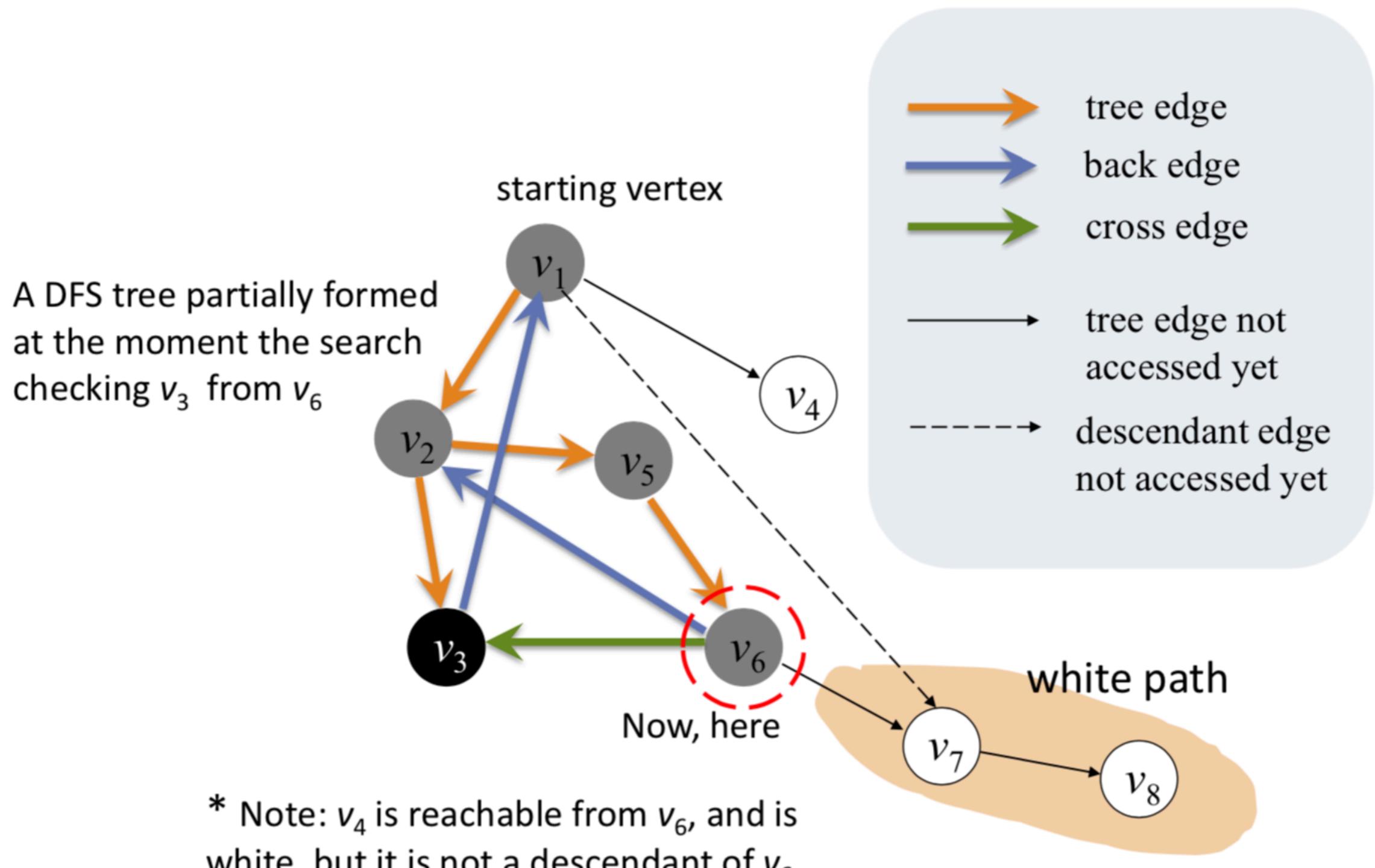
In the last class . . .

- Depth-first and breadth-first search
- Finding connected components
- General DFS/BFS **skeleton**
- Depth-first search **trace**

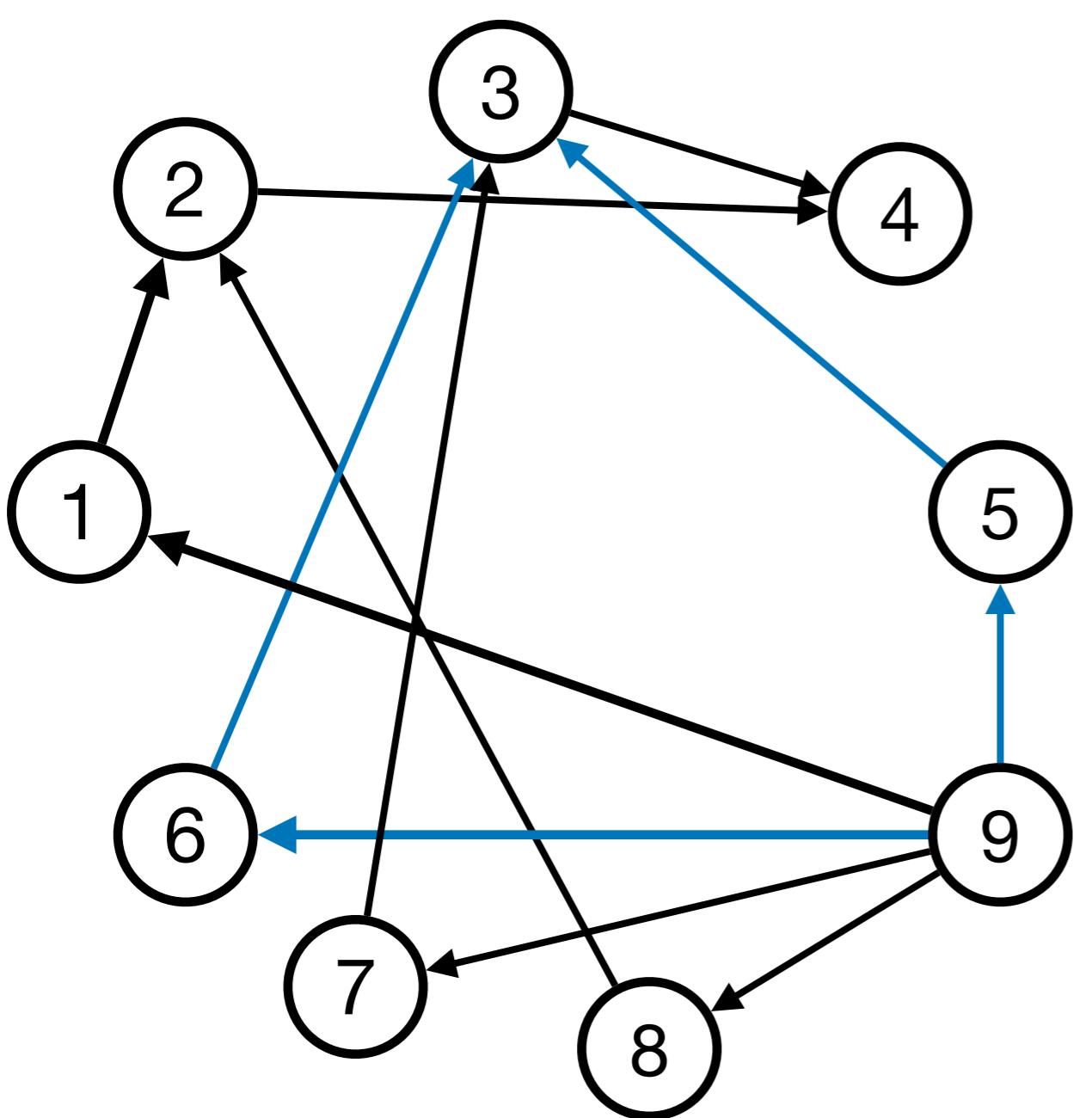
Applications of Graph Decomposition

- Directed Acyclic Graph
 - Topological order
 - Critical path analysis
- Strongly Connected Component (SCC)
 - Strong connected component and condensation
 - The algorithm
 - Leader of strong connected component

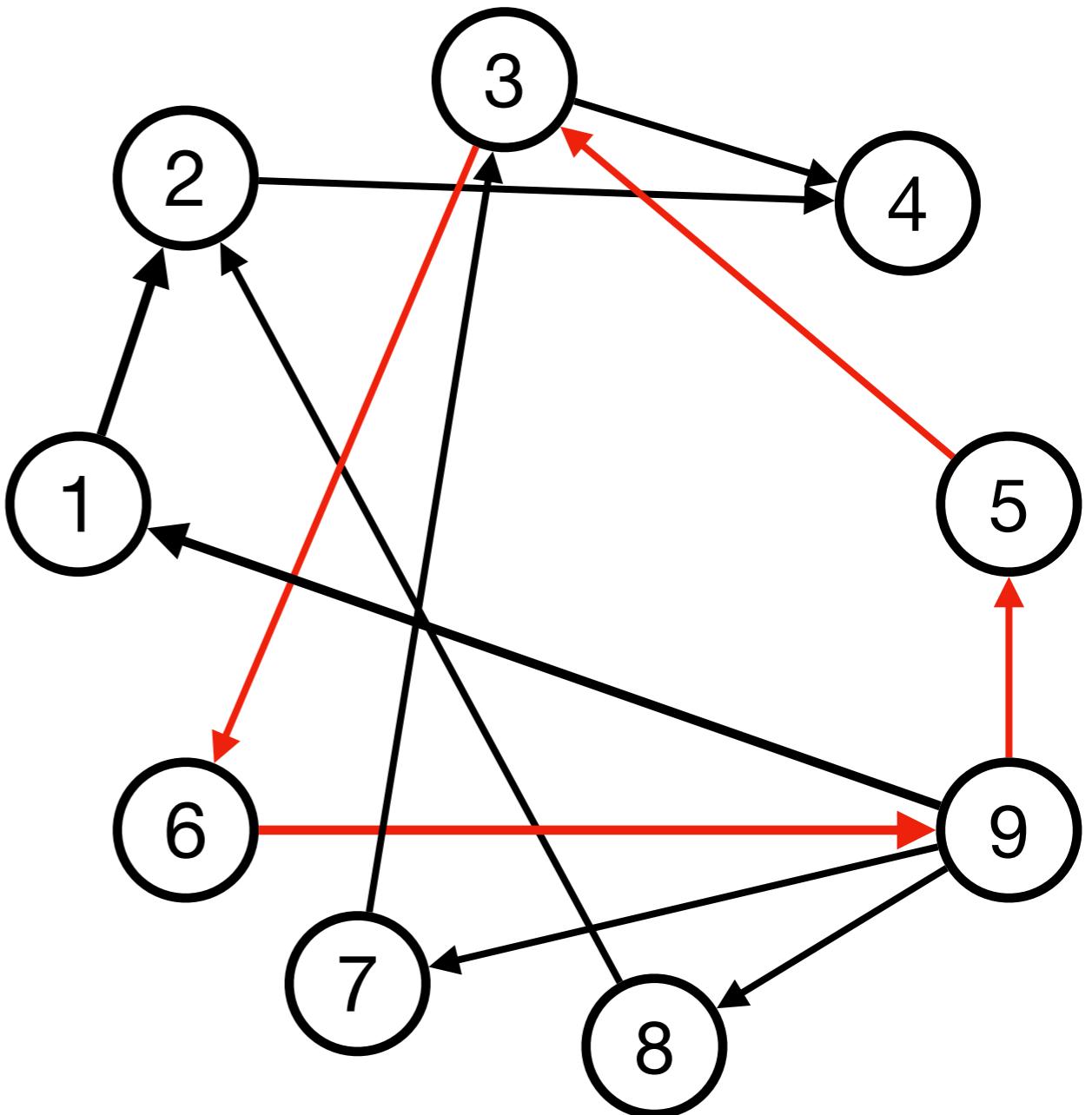
For Your Reference



Directed Acyclic Graph (DAG)



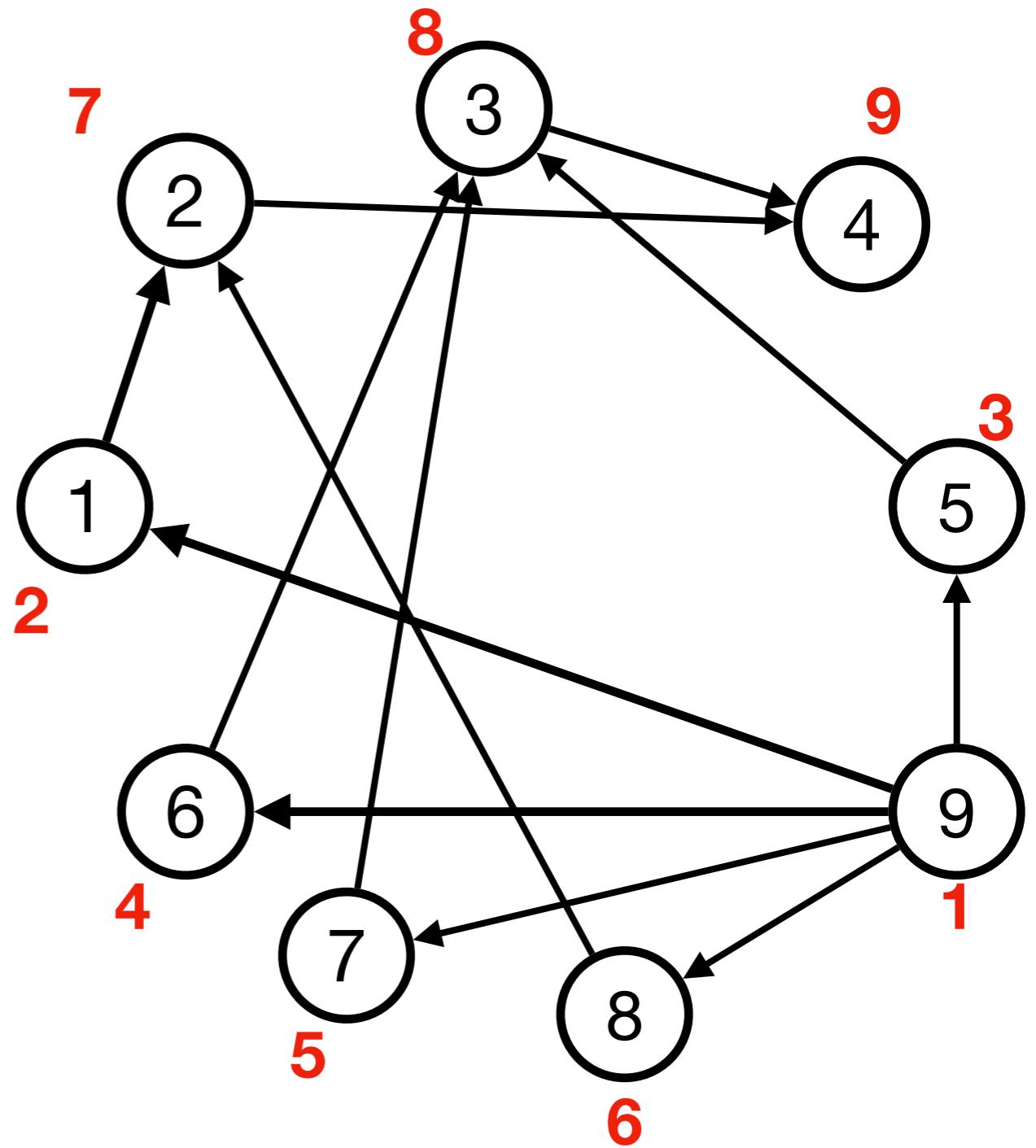
A Directed Acyclic Graph



Not a DAG

Topological Order for $G=(V, E)$

- Topological number
 - An assignment of distinct integer $1, 2, \dots, n$ to the vertices of V
 - For every $vw \in E$, the topological number of v is less than that of w .
- Reverse topological order
 - Defined similarly (“greater than”)



Existence of Topological Order - a Negative Result

- If a directed graph G has a cycle, then G has no topological order

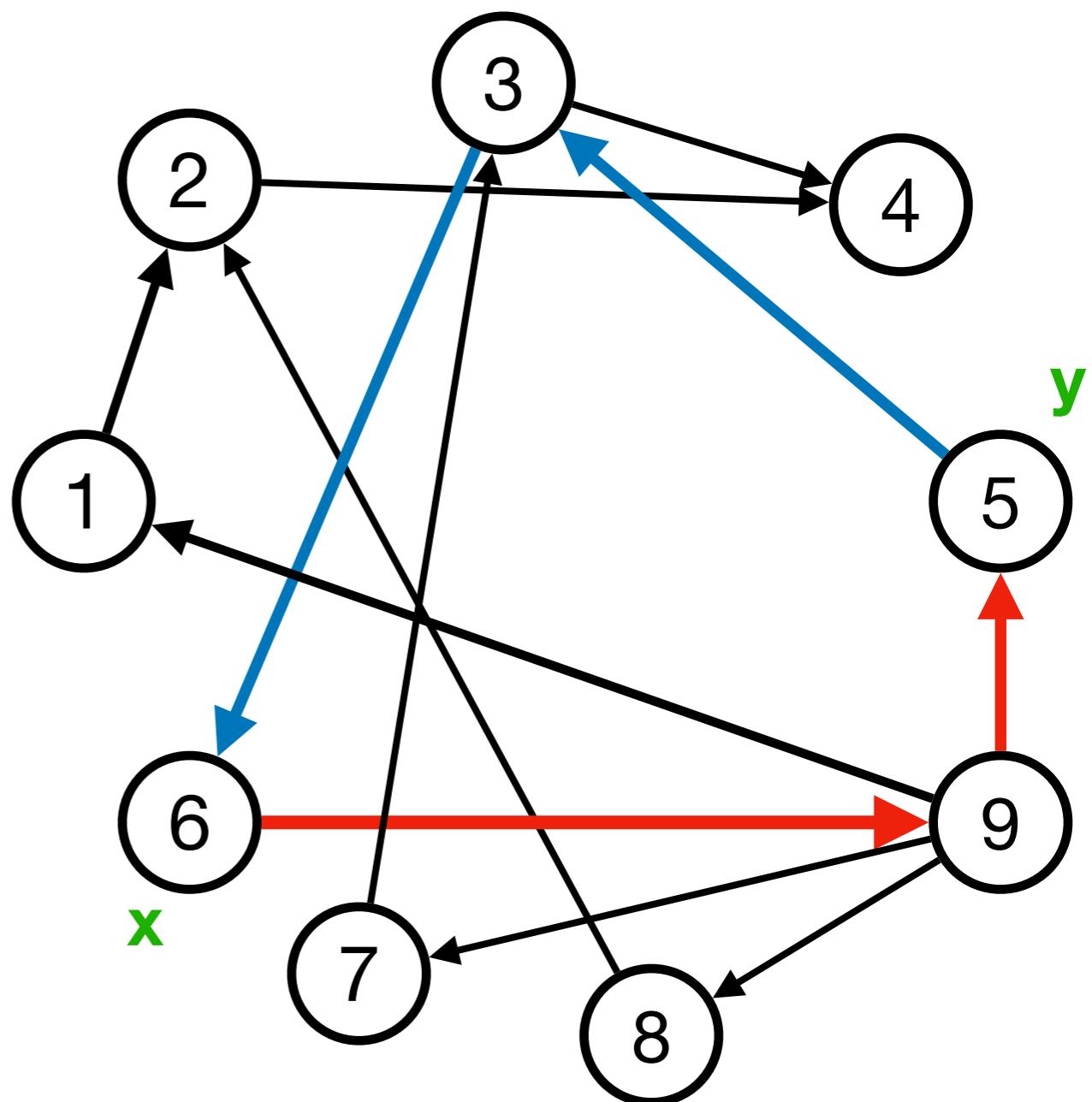
- Proof

- [By contradiction]

- **yx-path**

- **xy-path**

For any given topological order,
all the vertices on both paths
must be in increasing order.
Contradiction results for any
assignments for x and y.



Reverse Topological Ordering

- Specialized parameters
 - Array topo, keeps the topological number assigned to each vertex.
 - Counter topoNum to provide the integer to be used for topological number assignments
- Output
 - Array topo as filled.

Reverse Topological Ordering

- void dfsTopoSweep(intList[] adjVertices, int n, int[] topo)
- int topoNum=0;
- <Allocate color array and initialize to white>
- for each vertex v of G, in some order
 - if(color[v]==white)
 - dfsTopo(adjVertices, color, v, topo, topoNum);
 - //continue loop
- return;

For non-reverse topological ordering, initialized as n+1

Reverse Topological Ordering

- int dfsTopo(intList[] adjVertices, int[] color, int v, int[] topo, int topoNum)
- int w; intList remAdj; color[v]=gray;
- remAdj=adjVertices[v];
- while(remAdj != nil)
 - w=first(remAdj);
 - if(color[w]==white)
 - dfsTopo(adjVertices, color, w, topo, topoNum);
 - remAdj=rest(remAdj);
 - topoNum++; topo[v]=topoNum;
 - color[v]=black;
- return;

Obviously, in $\Theta(m+n)$

Filling topo is a post-order processing, so, the earlier discovered vertex has relatively greater top number

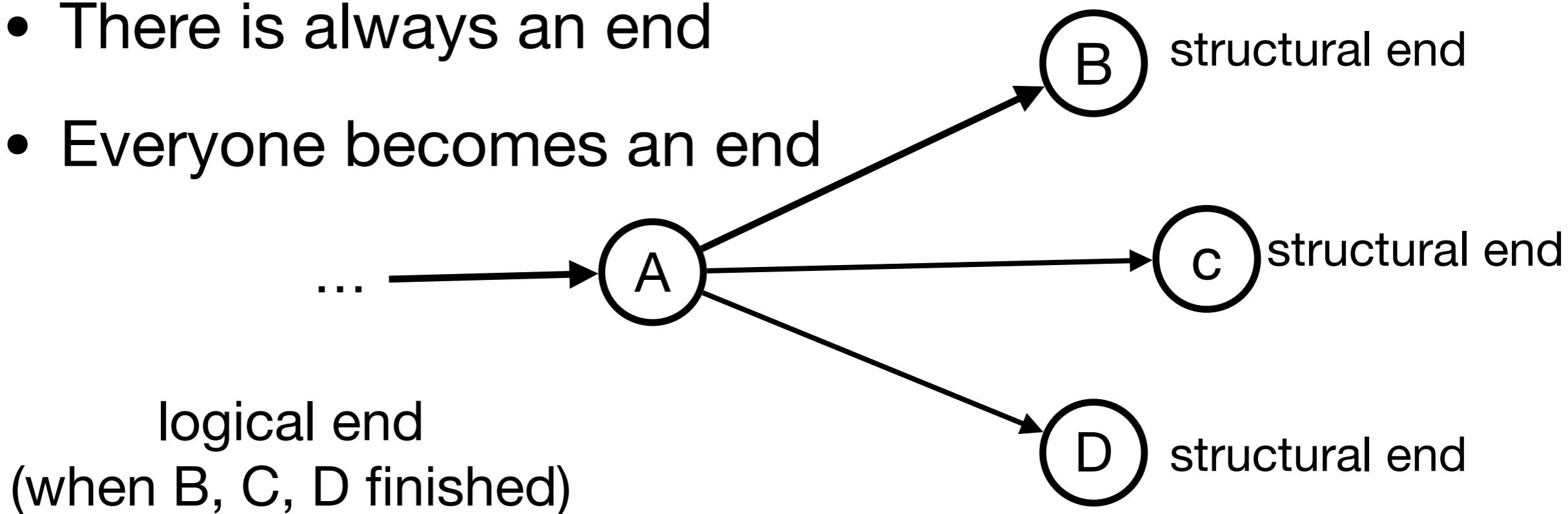
Reverse Topological Ordering

- For an “end node”

- Easy to decide

- Acyclic

- There is always an end
 - Everyone becomes an end



Correctness of the Algorithm

- If G is a DAG with n vertices, the procedure *dfsTopoSweep* computes a reverse topological order for G in the array *topo*.
- Proof
 - The procedure *dfsTopo* is called exactly once for a vertex, so, the numbers in *topo* must be distinct in the range $1,2,\dots,n$.
 - For any edge vw , vw can't be a back edge (otherwise, a cycle is formed). For any other edge types, we have $\text{finishTime}(v) > \text{finishTime}(w)$, so, $\text{topo}(w)$ is assigned earlier than $\text{topo}(v)$. Note that *topoNum* is incremented monotonically, so, $\text{topo}(v) > \text{topo}(w)$.

Existence of Topological Order

- In fact, the proof of correctness of topological ordering has proved that: DAG always has a topological order.
- So, G has a topological ordering, iff. G is a directed acyclic graph.

Task Scheduling

- Problem:

- Scheduling a project consisting of a set of **interdependent** tasks to be done by **one** person.

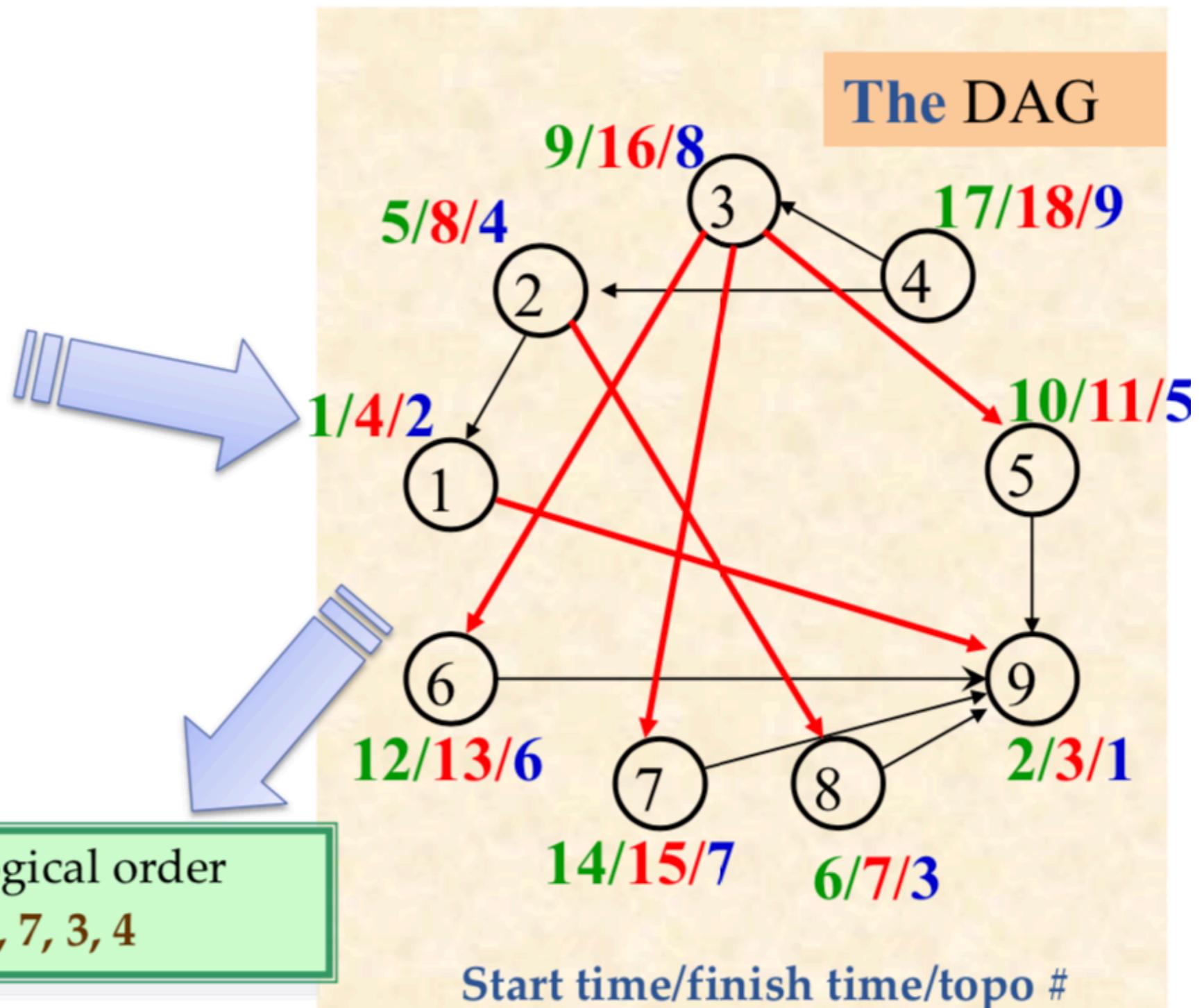
- Solution:

- Establishing a dependency graph, the vertices are tasks, and edge vw is included iff. the execution of v depends on the completion of w ,
- Making task scheduling according to the topological order of the graph (if existing).

Task Scheduling: an Example

Tasks(No.)	Depends on
choose clothes(1)	9
dress(2)	1,8
eat breakfast(3)	5,6,7
leave(4)	2,3
make coffee(5)	9
make toast(6)	9
pour juice(7)	9
shower(8)	9
wake up(9)	-

A reverse topological order
9, 1, 8, 2, 5, 6, 7, 3, 4



Project Optimization Problem

Assuming that parallel executions of tasks (v_i) are possible except for prohibited by interdependency.

- Observation

- In a **critical path**, v_{i-1} , is a critical dependency of v_i , i.e., any delay in v_{i-1} will result in delay in v_i .
- The time for entire project depends on the time for the critical path.
- Reducing the time of an off-critical-path task is of no help for reducing the total time for the project.

- The Problems

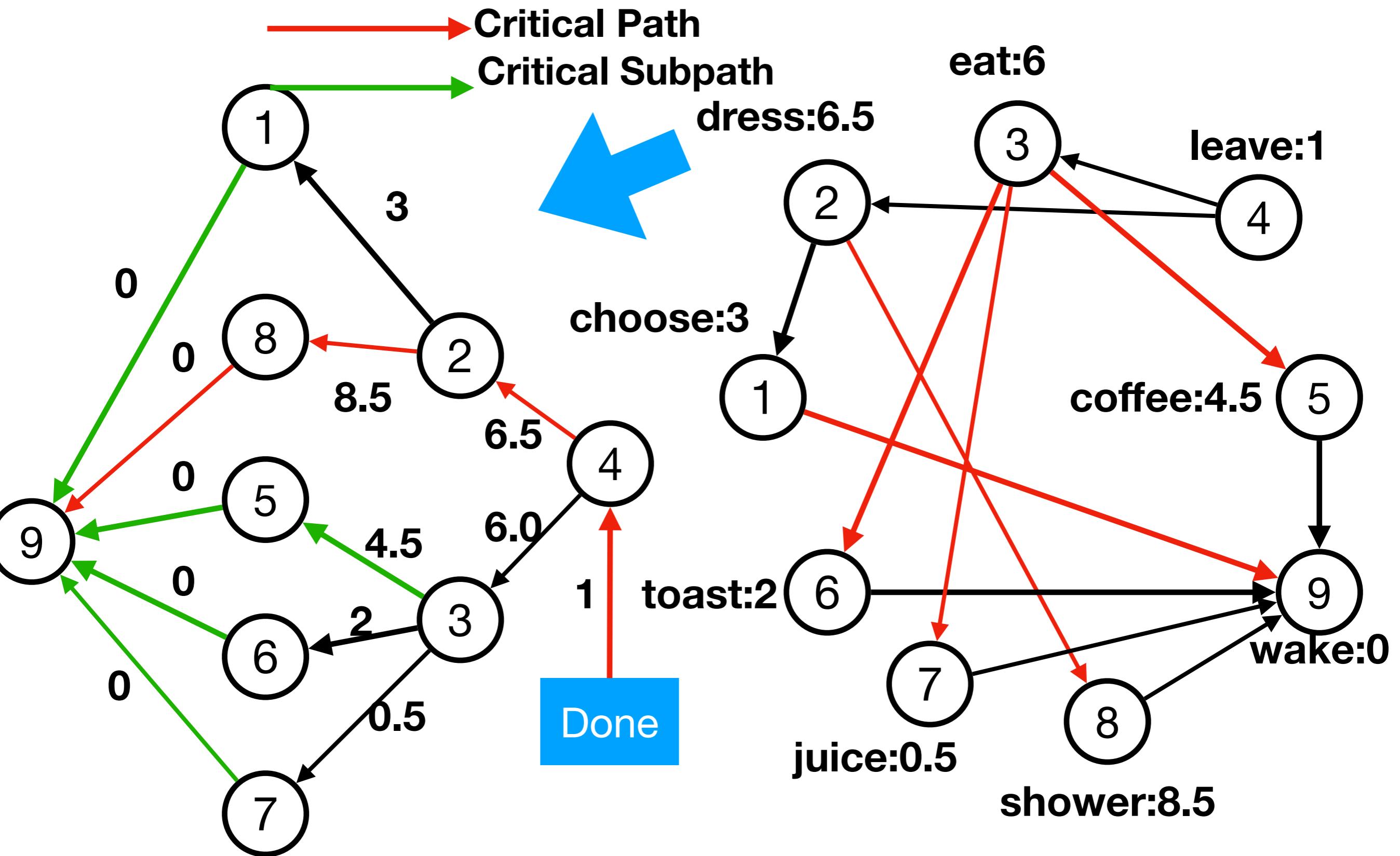
- Find the critical path in a DAG
- (Try to reduce the time for the critical path)

This is a precondition.

Critical Path in a Task Graph

- **Earliest start time (est) for a task v**
 - If v has no dependencies, the est is 0
 - If v has dependencies, the est is the maximum of the **earliest finish time** of its dependencies.
- **Earliest finish time (eft) for a task v**
 - For any task: **eft = est + duration**
- **Critical path** in a project is a sequence of tasks: v_0, v_1, \dots, v_k , satisfying:
 - v_0 has no dependencies;
 - For any $v_i = (i=1, 2, \dots, k)$, v_{i-1} is a dependency of v_i , such that est of v_i equals eft of v_{i-1} ;
 - eft of v_k , is maximum for all tasks in the project.

DAG with Weights



Critical Path Finding - DFS

- Specialized parameters

- Array duration, keeps the execution time of each vertex.
- Array critDep, keeps the critical dependency of each vertex.
- Array eft, keeps the earliest finished time of each vertex.

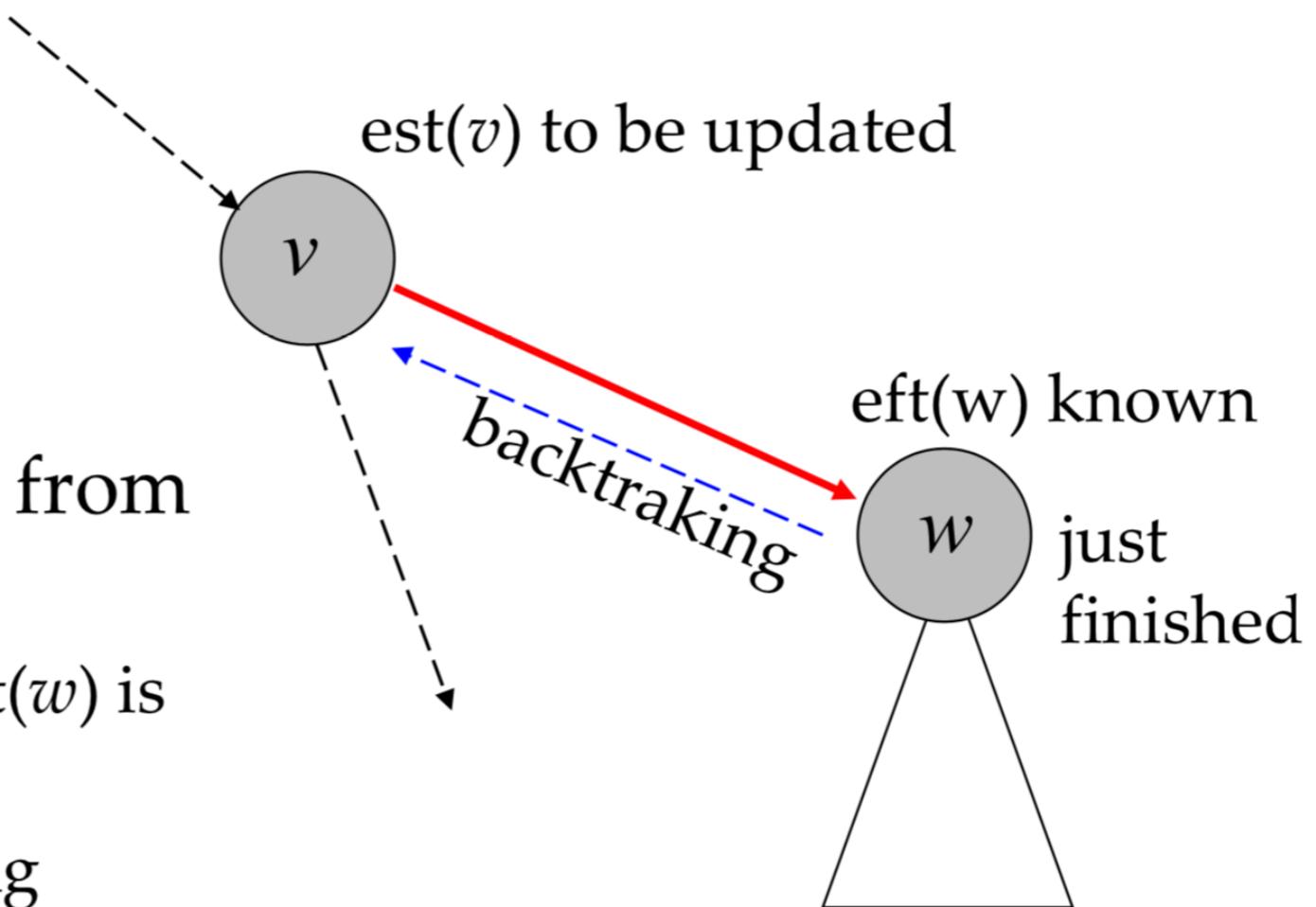
- Output

- Array topo, critDep, eft as filled.
- **Critical path is built by tracing the output.**

Critical Path - Case 1

Upon **backtracking** from
 w :

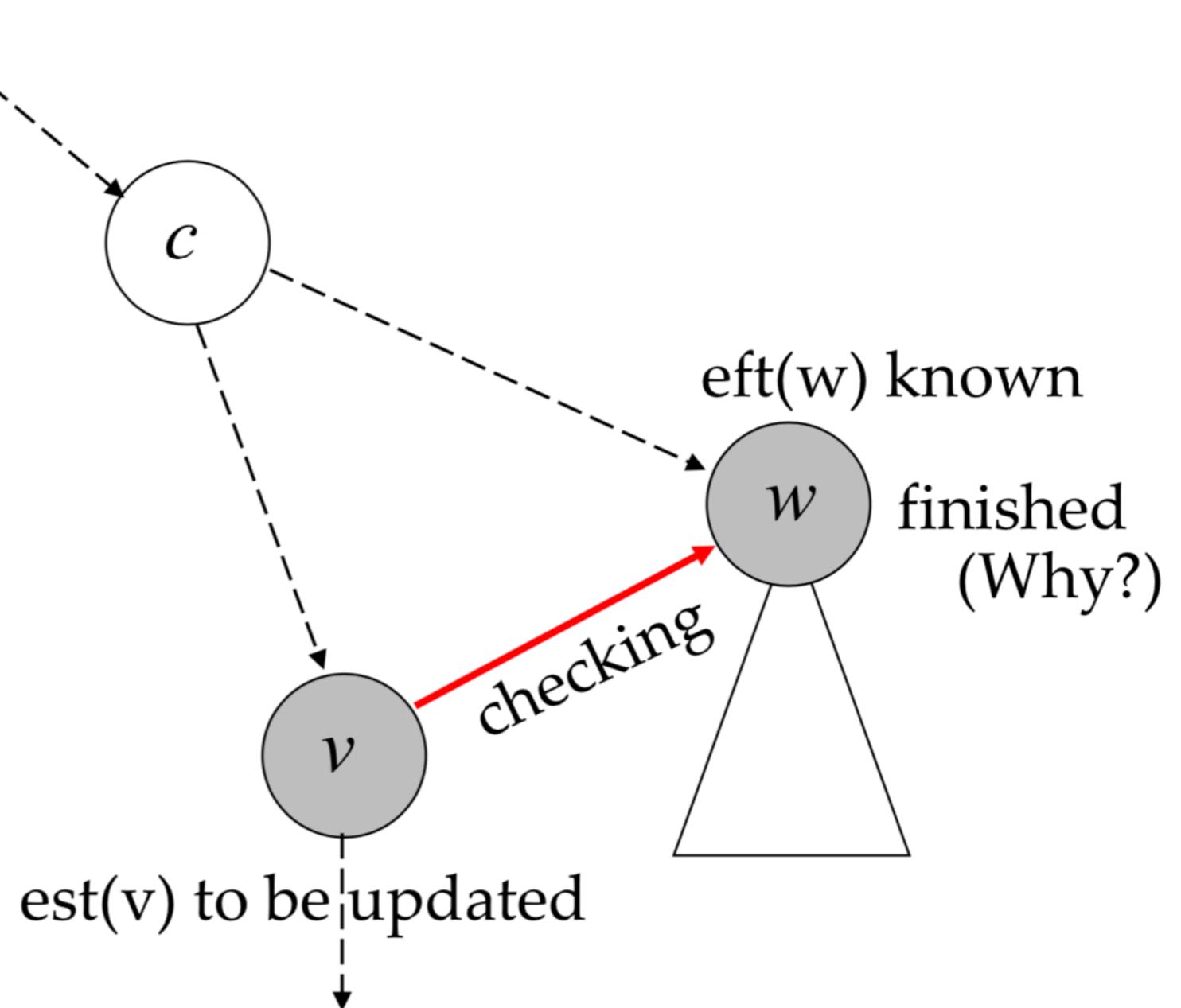
- $\text{est}(v)$ is updated if $\text{eft}(w)$ is larger than $\text{est}(v)$
- and the path including edge vw is recognized as the critical path for task v
- and the $\text{eft}(v)$ is updated accordingly



Critical Path - Case 2

Checking w :

- $\text{est}(v)$ is updated if $\text{eft}(w)$ is larger than $\text{est}(v)$
- and the path including edge vw is recognized as the critical path for task v
- and the $\text{eft}(v)$ is updated accordingly



Critical Path by DFS

- `void dfsCritSweep(intList[] adjVertices, int n, int[] duration, int[] critDep, int[] eft)`
- `<Allocate color array and initialize to white>`
- for each vertex v of G, in some order
 - `if(color[v]==white)`
 - `dfsCrit(adjVertices, color, v, duration, critDep, eft);`
- `//continue loop`
- `return;`

Critical Path by DFS

- int dfsCrit(intList[] adjVertices, int[] color, int v, int[] duration int[] critDep, int eft)
 - int w; intList remAdj; int est=0;
 - color[v]=gray; critDep[v]=-1; remAdj=adjVertices[v];
 - while(remAdj != nil) w=first(remAdj);
 - if(color[w]==white)
 - dfsCrit(adjVertices, color, w, duration, critDep, eft);
 - if (eft[w]≥est) est=eft[w]; critDep[v]=w;
 - else//checking for nontree edge
 - if (eft[w]>est) est=eft[w]; critDep[v]=w;
 - remAdj=rest(remAdj);
 - eft[v]=est+duration[v]; color[v]=black;
 - return;

When is the eft[w] initialized?
Only black vertex

Analysis of Critical Path Algorithm

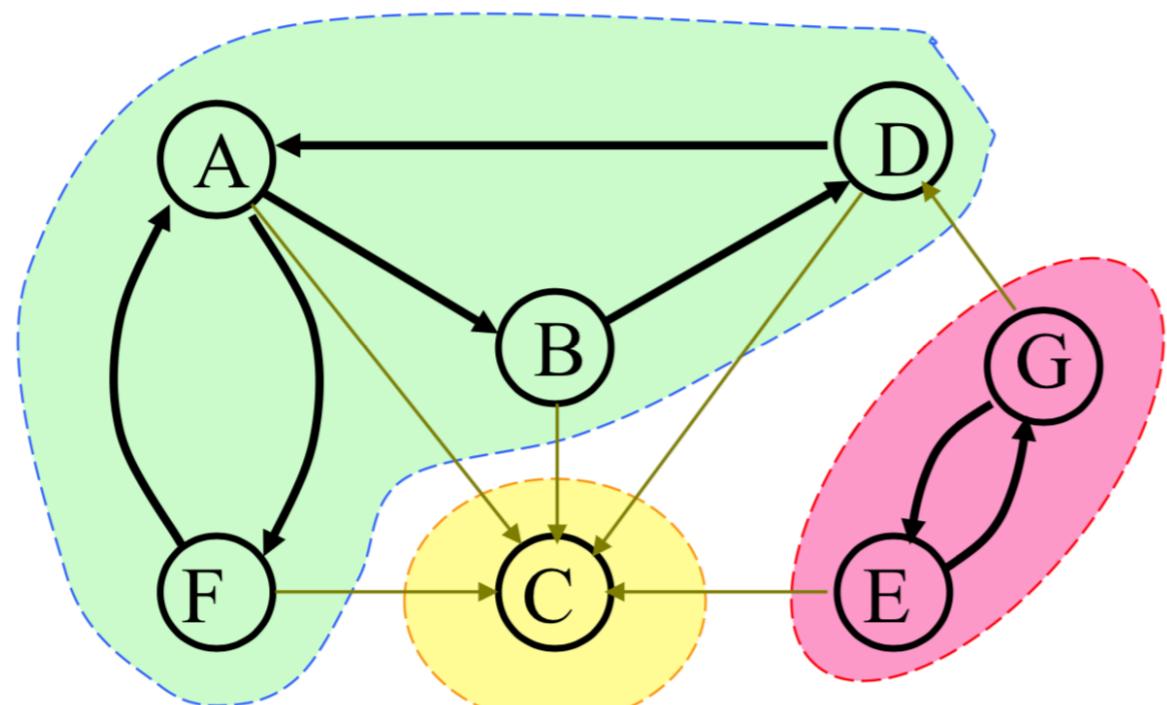
- Correctness:

- When $\text{eft}[w]$ is accessed in the while-loop, the w must not be gray(otherwise, there is a cycle), so, it must be black, with eft initialized.
- According to DFS, each entry in the eft array is assigned a value **exactly once**. The value satisfies the definition of eft .

- Complexity

- Simply same as DFS, that is **$\Theta(n+m)$** .

SCC: Strongly Connected Component

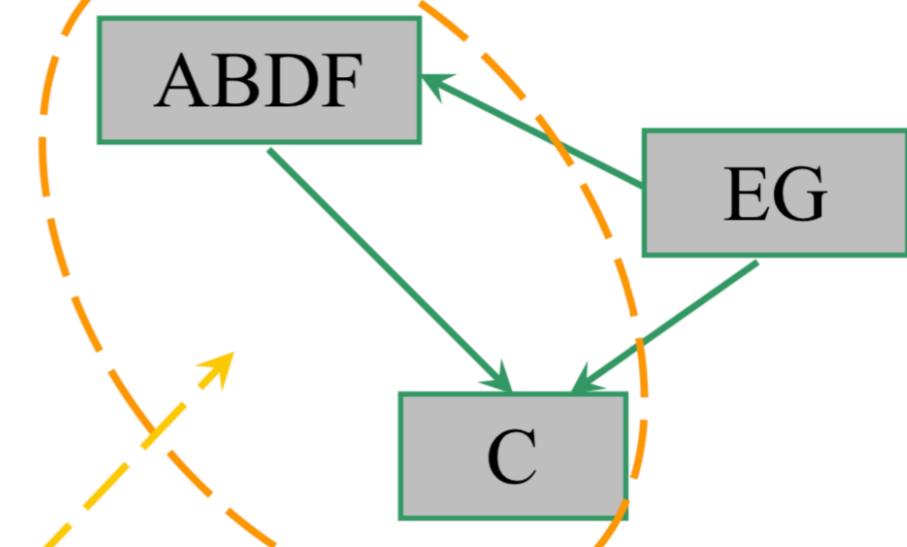


Graph G

3 Strongly Connected Components

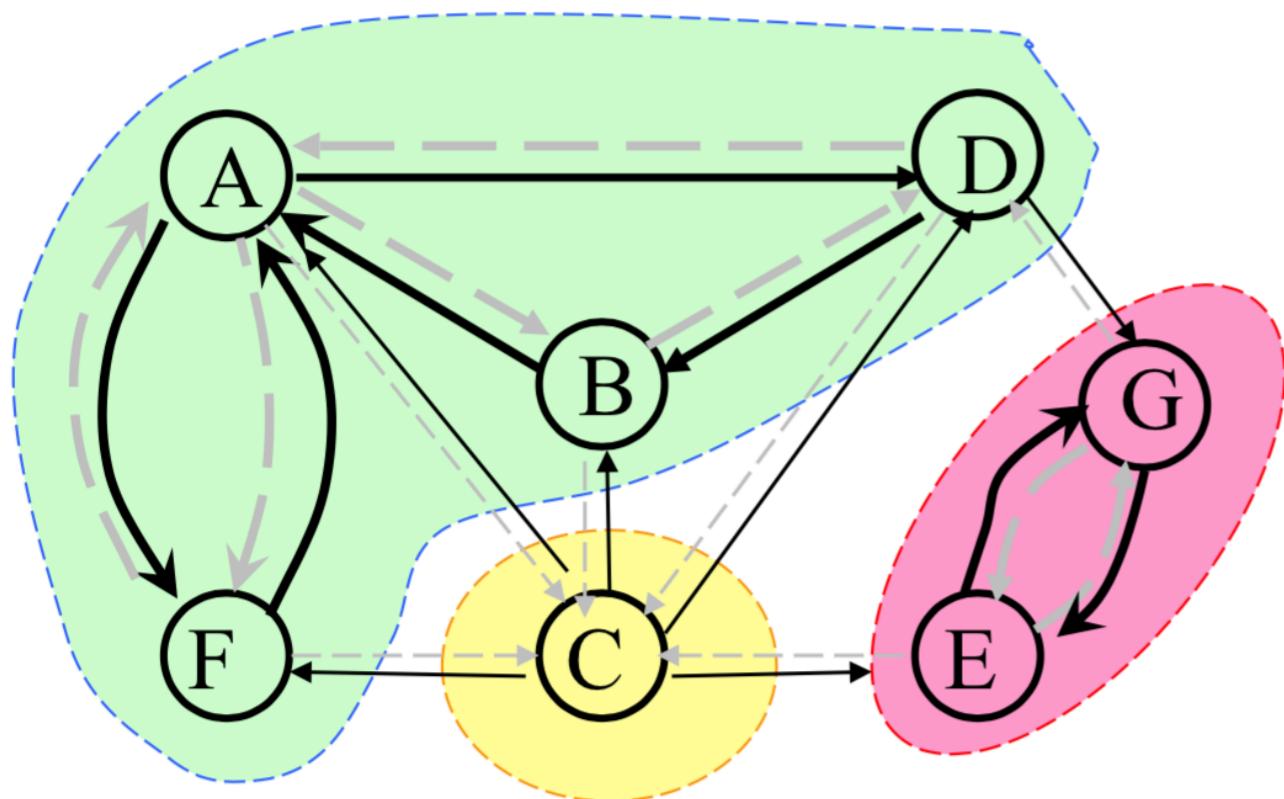
Note: two SCC in one DFS tree

Condensation Graph G^\downarrow



It's acyclic, **Why?**

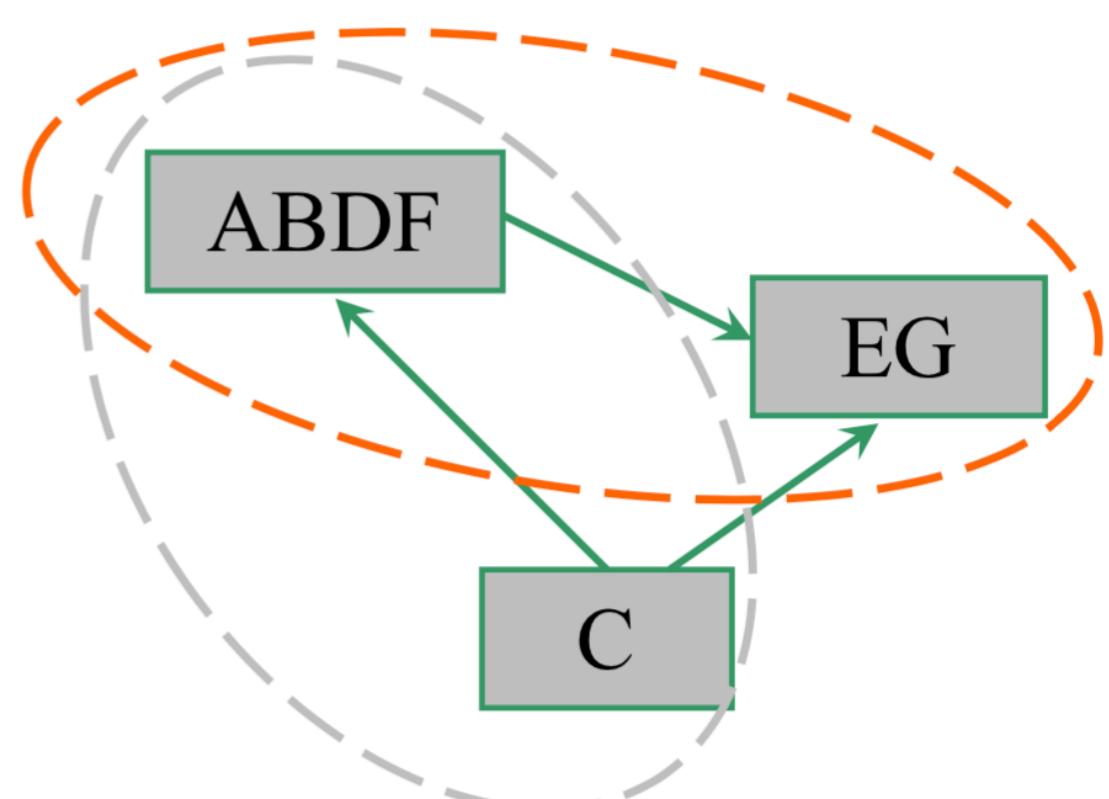
Transpose Graph



Transpose Graph G^T

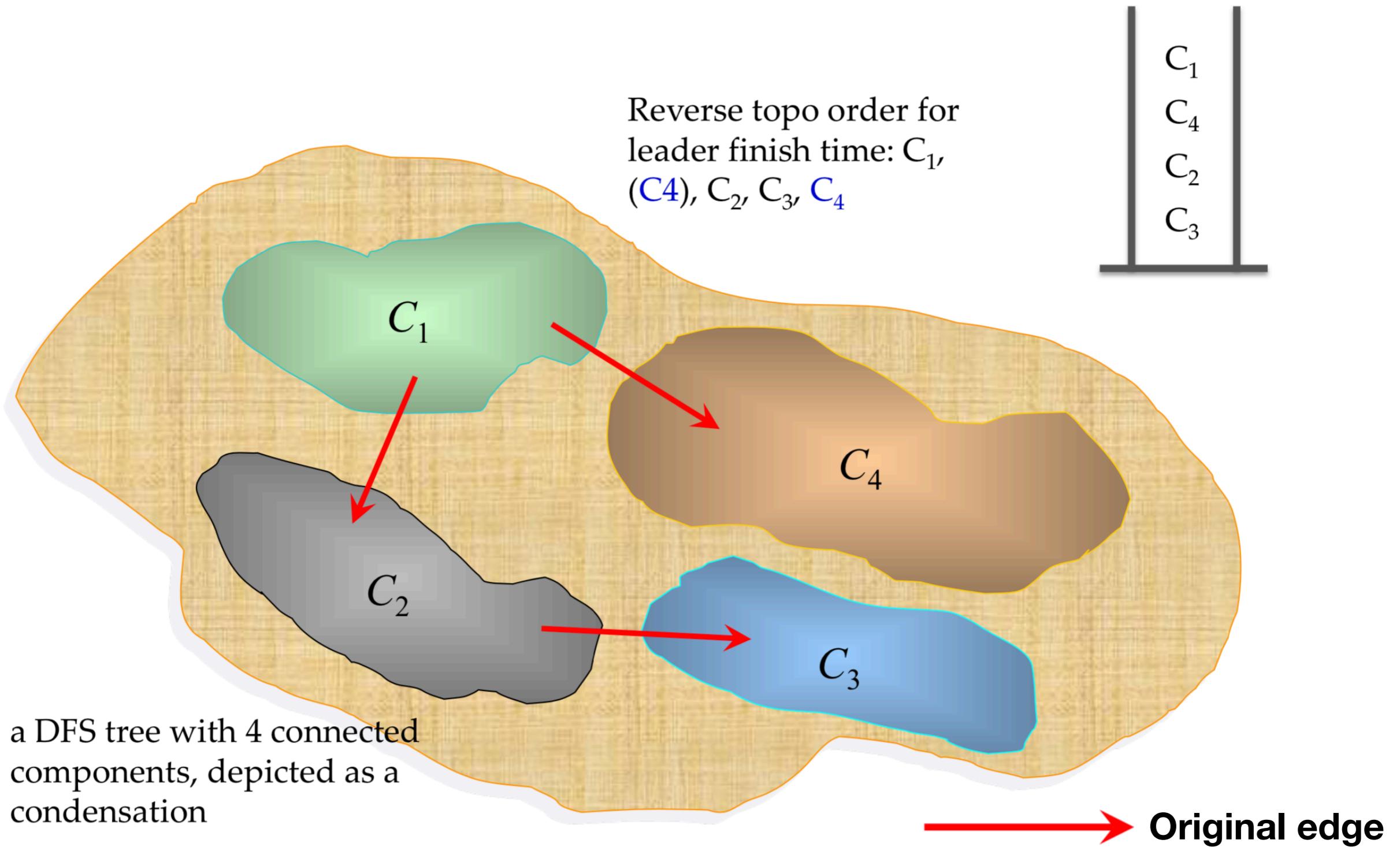
Connected Components **unchanged**
according to vertices

Condensation Graph $G \downarrow$



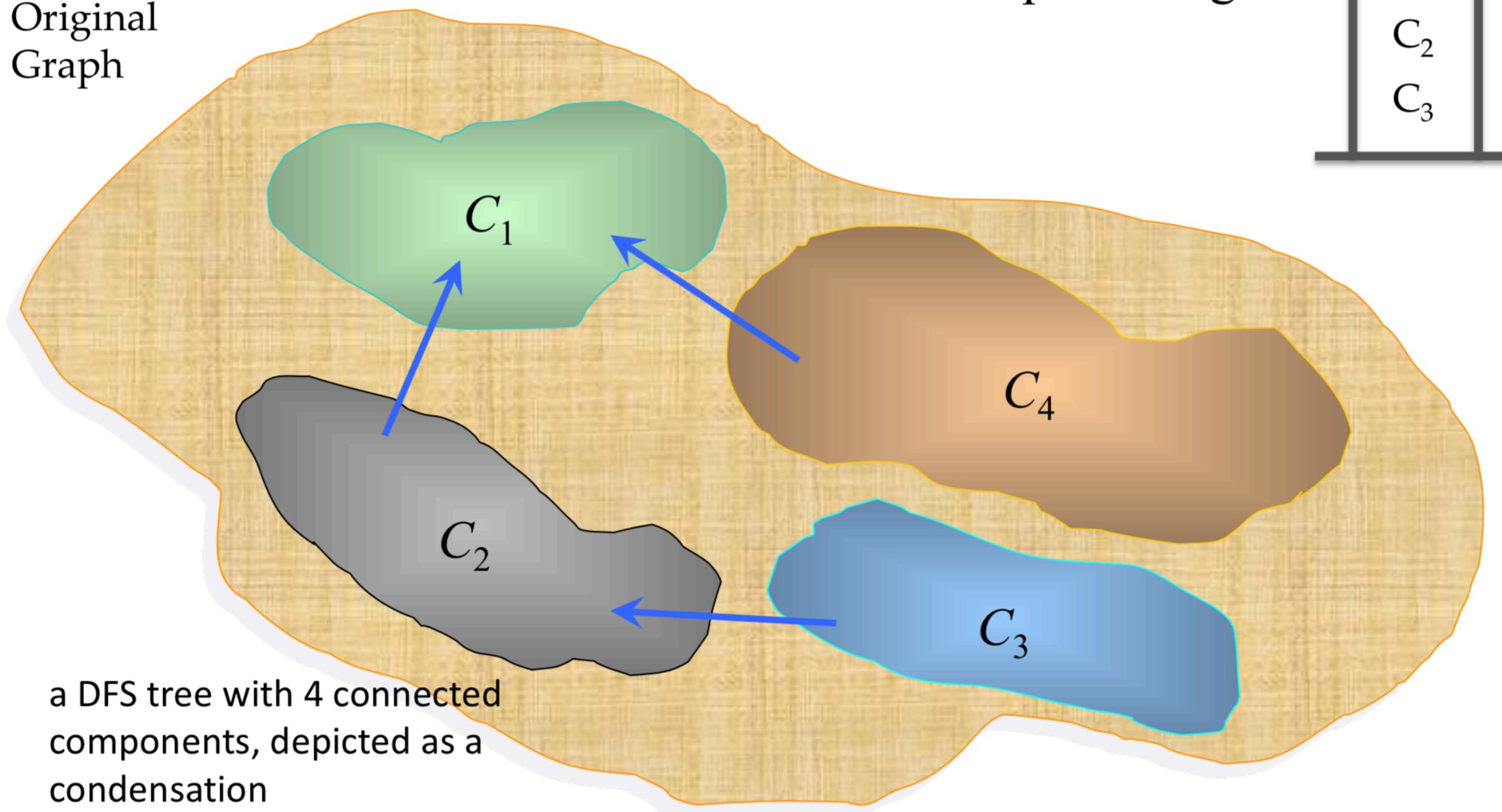
But, DFS tree **changed**

Basic Idea - G

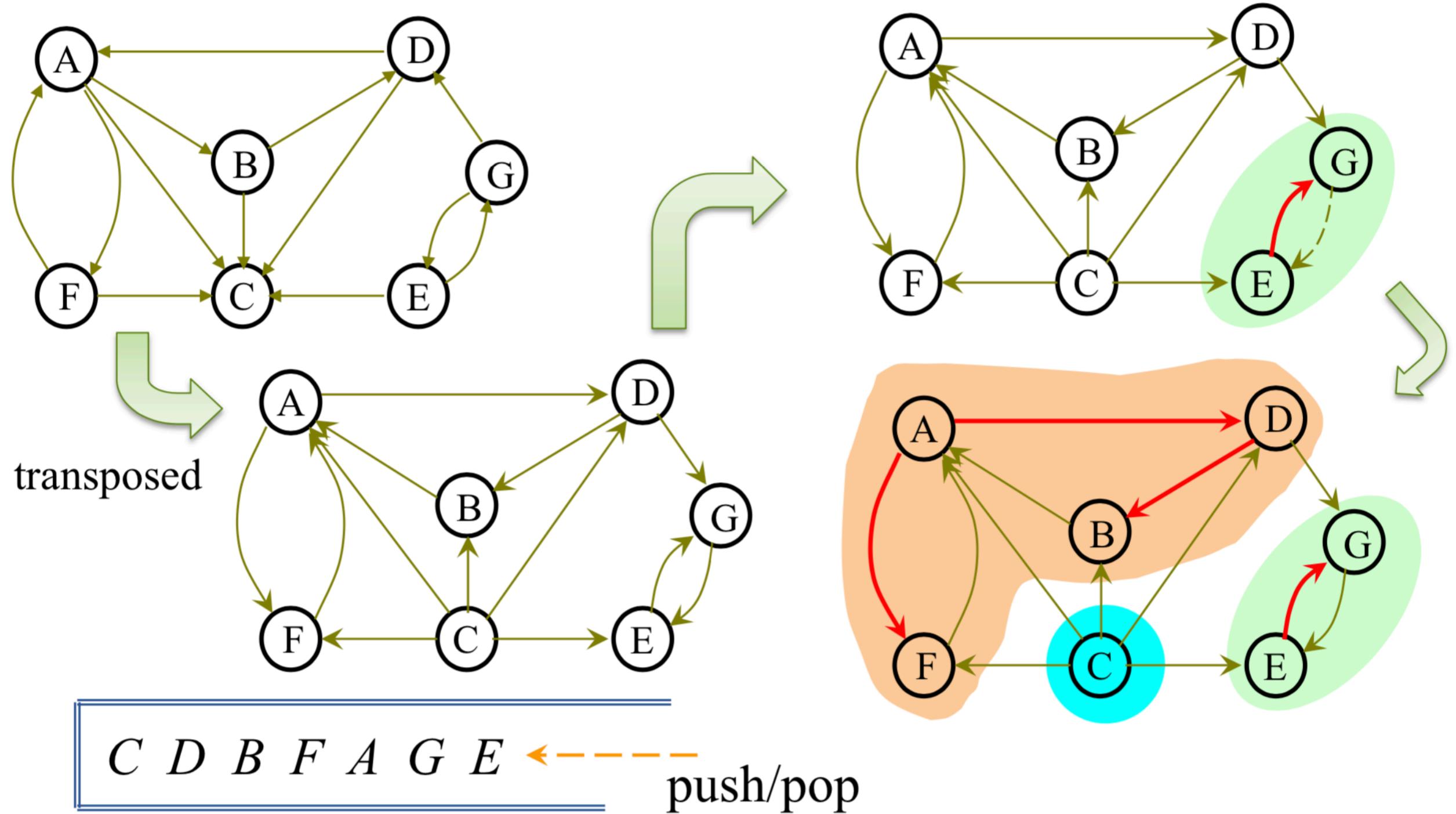


Basic Idea - G^T

Original
Graph



SCC - An Example



Strong Component Algorithm: Outline

- void strongComponents(intList[] adjvertices, int n, int[] scc)
- //Phase 1
 - 1. intStack finishStack=create(n);
 - 2. perform a depth-first search on G, using the DFS skeleton. At postorder processing for vertex v, insert the statement: **push(finishStack, v)**
- //Phase 2
 - 3. Compute G^T , the transpose graph, represented as array adjTrans of adjacency list.
 - 4. **dfsTsweep(adjTrans, n, finishStack, scc);**
 - return Note: G and G^T have the same SCC sets

Strong Component Algorithm: Core

- void dfsTSweep(intList[] adjVertices, int n, intStack finishStack, int[] scc)
- <Allocate color array and initialize to white>
- while(finishStack is not empty)
 - int v=top(finishStack);
 - pop(finishStack);
 - if(color[v]==white)
 - dfsT(adjVertices, color, v, scc);
- return;
- void dfsT(intList[] adjTrans, int[] color, int v, int leader, int[] scc)
 - Use the standard depth-first search skeleton. At postorder processing for vertex v insert the statement:
 - scc[v]=leader;
 - Pass leader and scc into recursive calls.

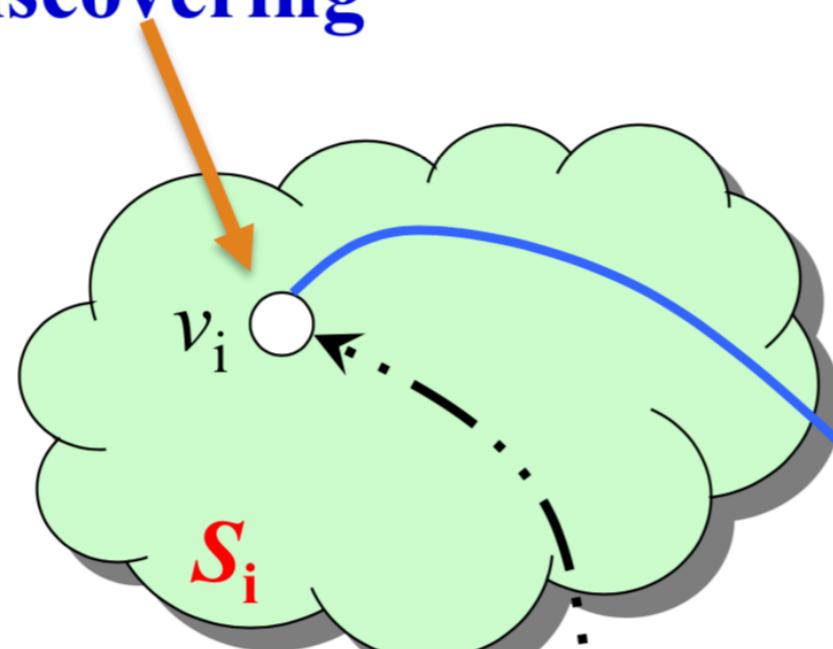
Leader of a Strong Component

- For a DFS, the first vertex discovered in a strong component S_i is called the **leader** of S_i .
- Each DFS tree of a digraph G contains **only complete** strong components of G , one or more.
 - Proof: Applying White Path Theorem whenever the leader of S_i ($i=1,2,\dots,p$) is discovered, starting with all vertices being white.
- The leader of S_i is the last vertex to finish among all vertices of S_i . **(since all of them in the same DFS tree)**

Path between SCCs

The leader of S_i

At discovering

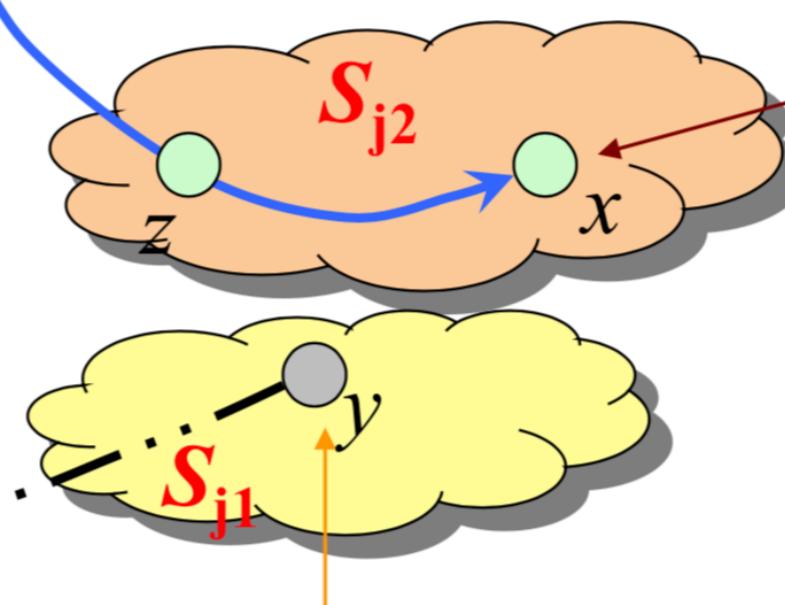


Existing a yv_i -path, so x must be in a different strong component.

No v_iy -path can exist.

x can't be gray.

- White case: $v_i x$ -path is a White Path, or
- Black case: x is black (consider the [possible] last non-white vertex z on the $v_i x$ -path)



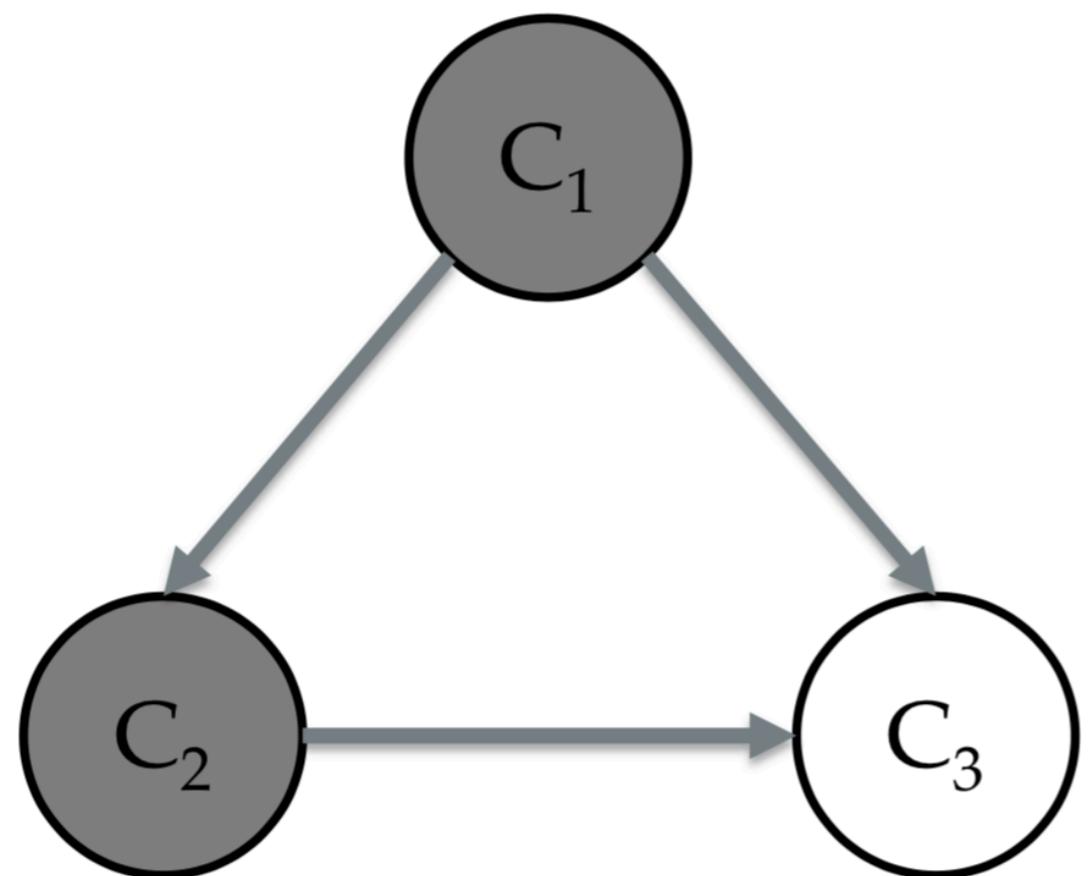
What's the color?

See Lemma 7.8 & 7.9
p. 360 of [Baase01]

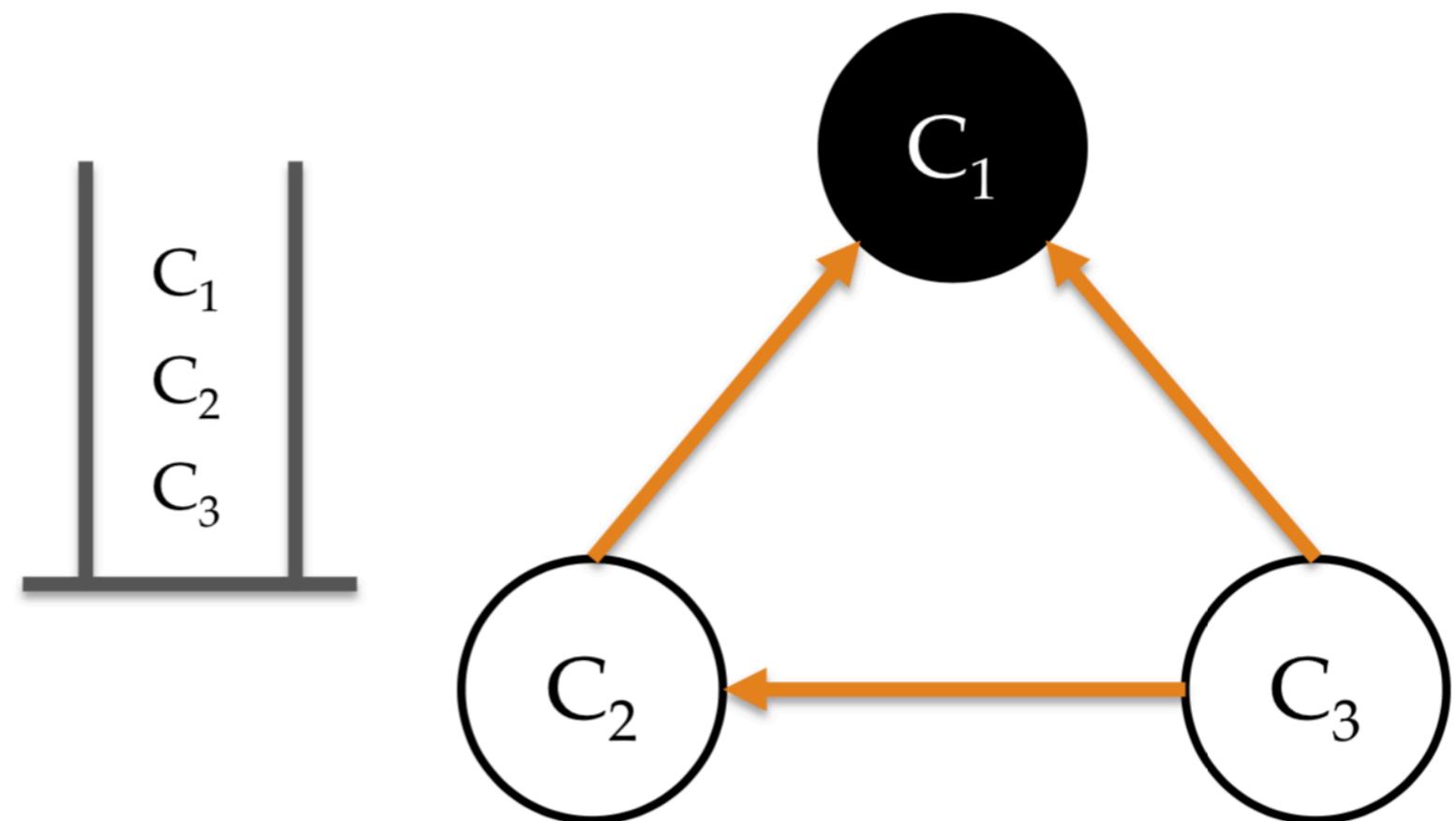
C_1 : The End Case

Looking at C_2, C_3 from C_1

G



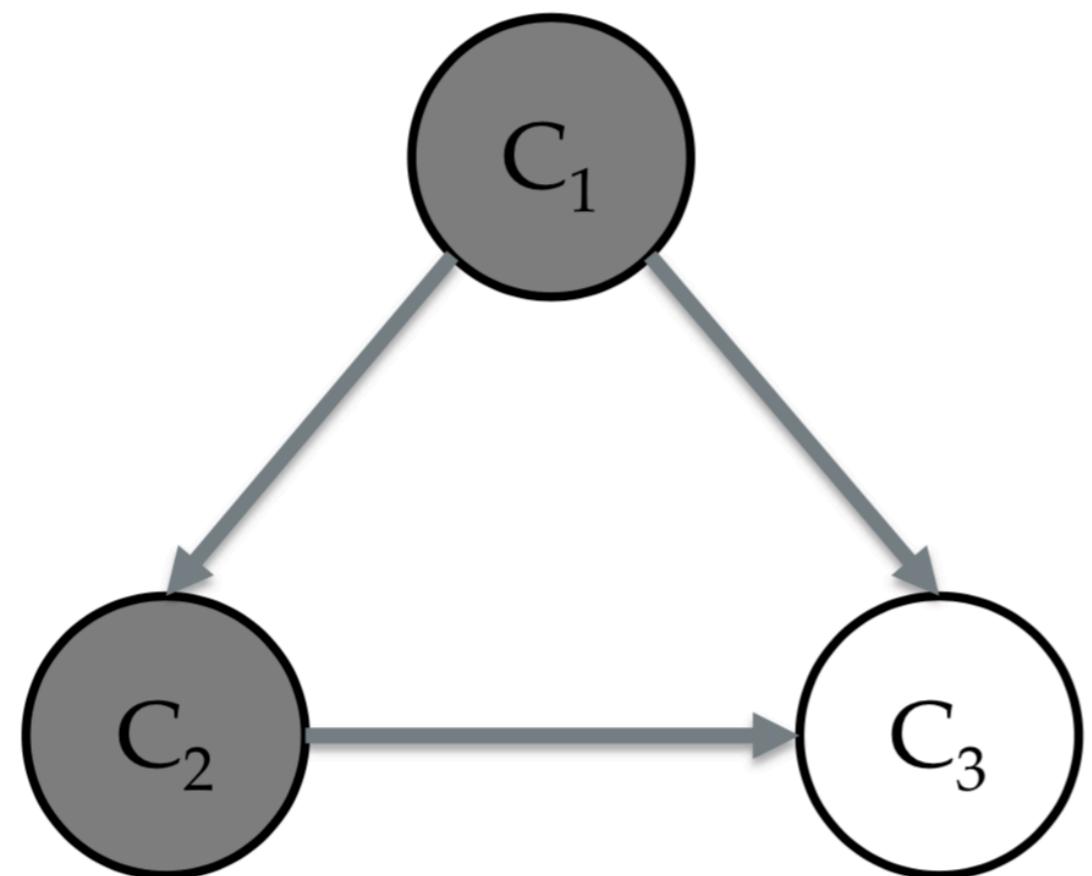
G^T



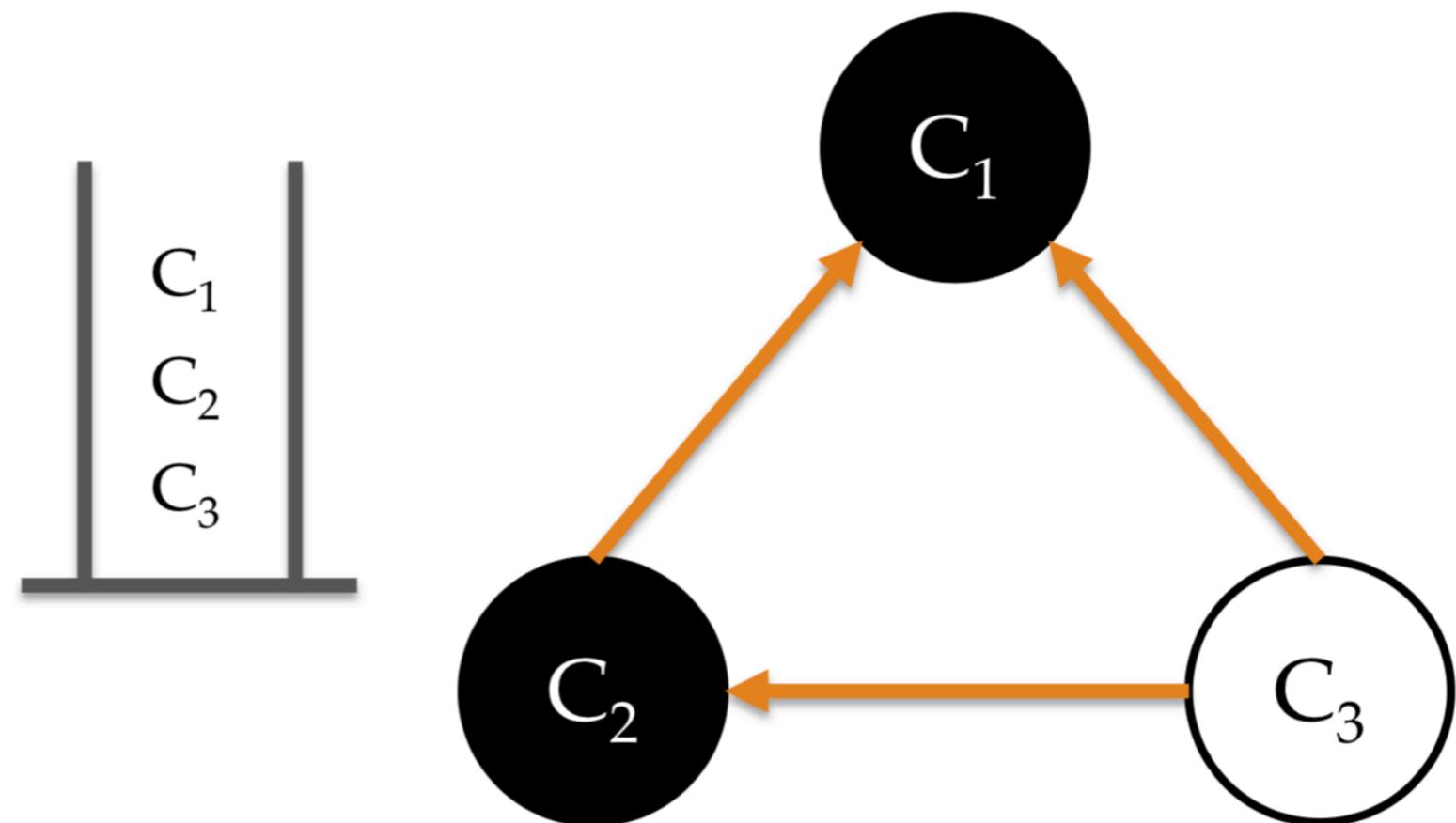
C_2 : The White Case

Looking at C_3 from C_2

G



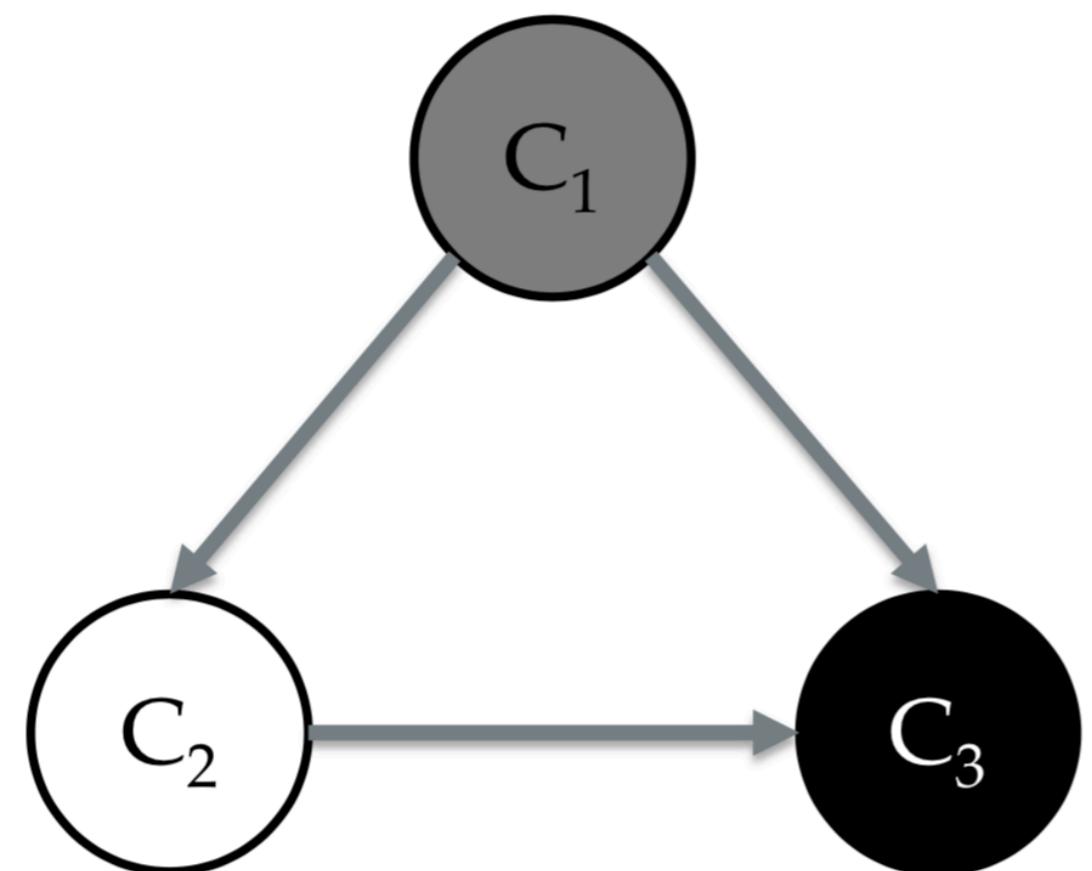
G^T



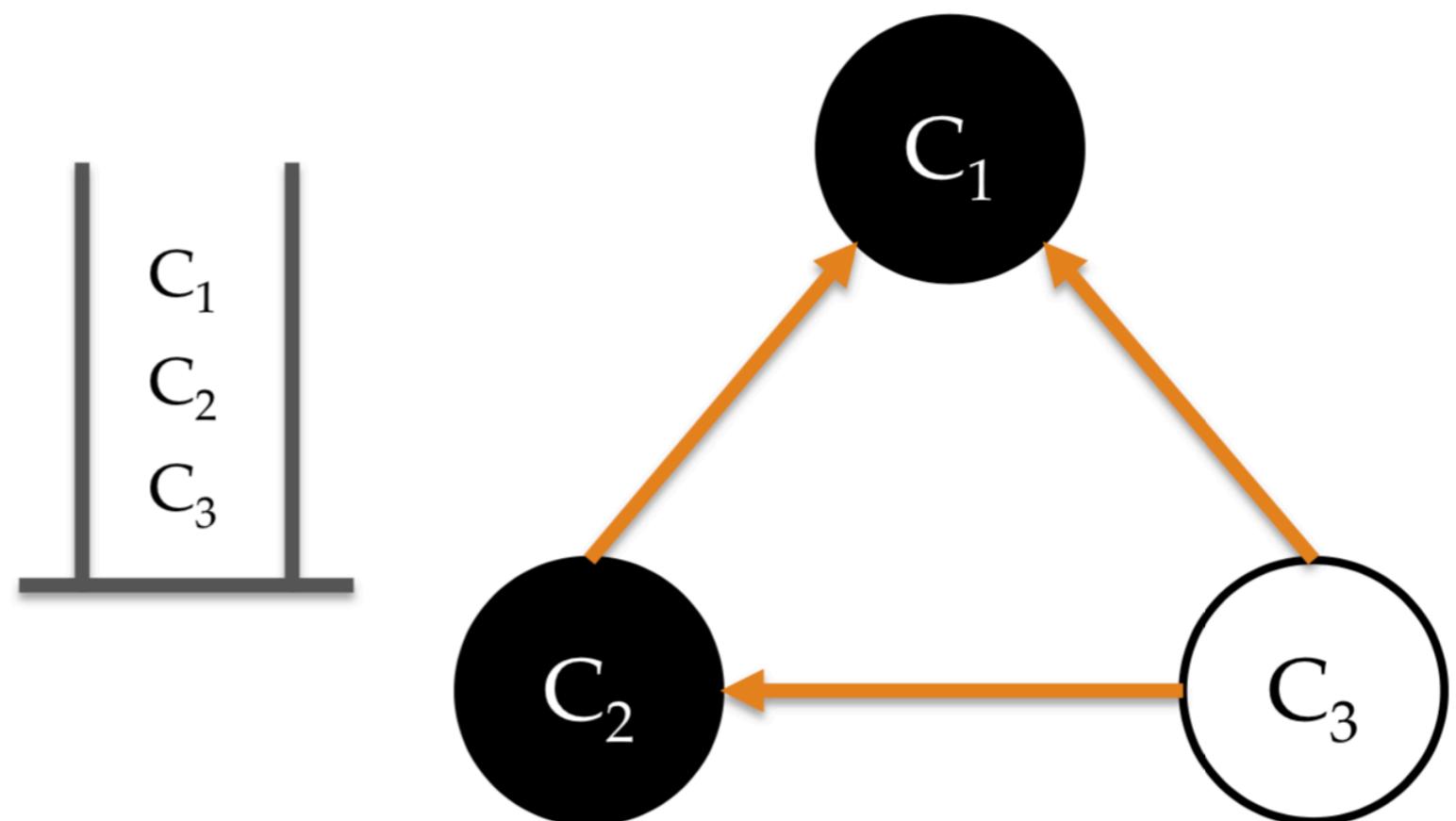
C_2 : The Black Case

Looking at C_3 from C_2

G



G^T



Active Intervals

- If there is an edge from S_i to S_j , then it is **impossible** that the active interval of v_j is **entirely after** that of v_i . (Note: for leader v_i only)
- There is no path from a leader of a strong component to any gray vertex.
- If there is a path from the leader v of a strong component to any x in a different strong component, v finishes later than x .

Correctness of Strong Component Algorithm (1)

- In phase 2, each time a white vertex is popped from *finishStack*, that vertex is the Phase 1 leader of a strong component.
 - The later finished, the earlier popped
 - The leader is the first to get popped in the strong component it belongs to
 - If x popped is not a leader, then some other vertex in the strong component has been visited previously. But not a partial strong component can be in a DFS tree, so, x must be in a completed DFS tree, and is not white.

Correctness of Strong Component Algorithm (2)

- In phase 2, each depth-first search tree contains exactly one strong component of vertices
 - Only “exactly one” need to be proved
 - Assume that v_i , a phase 1 leader is popped. If another component S_j is reachable from v_i in G^T , there is a path in G from v_j to v_i . So, in phase 1, v_j finished later than v_i , and popped earlier than v_i in phase 2. So, when v_i popped, all vertices in S_j are black. So, S_j are not contained in DFS tree containing $v_i(S_i)$.

Thank you!
Q & A