Directed graphs, DFS, DAGs, TopSort

Sides based on material by Kani, Erickson, Chekuri, et. al.

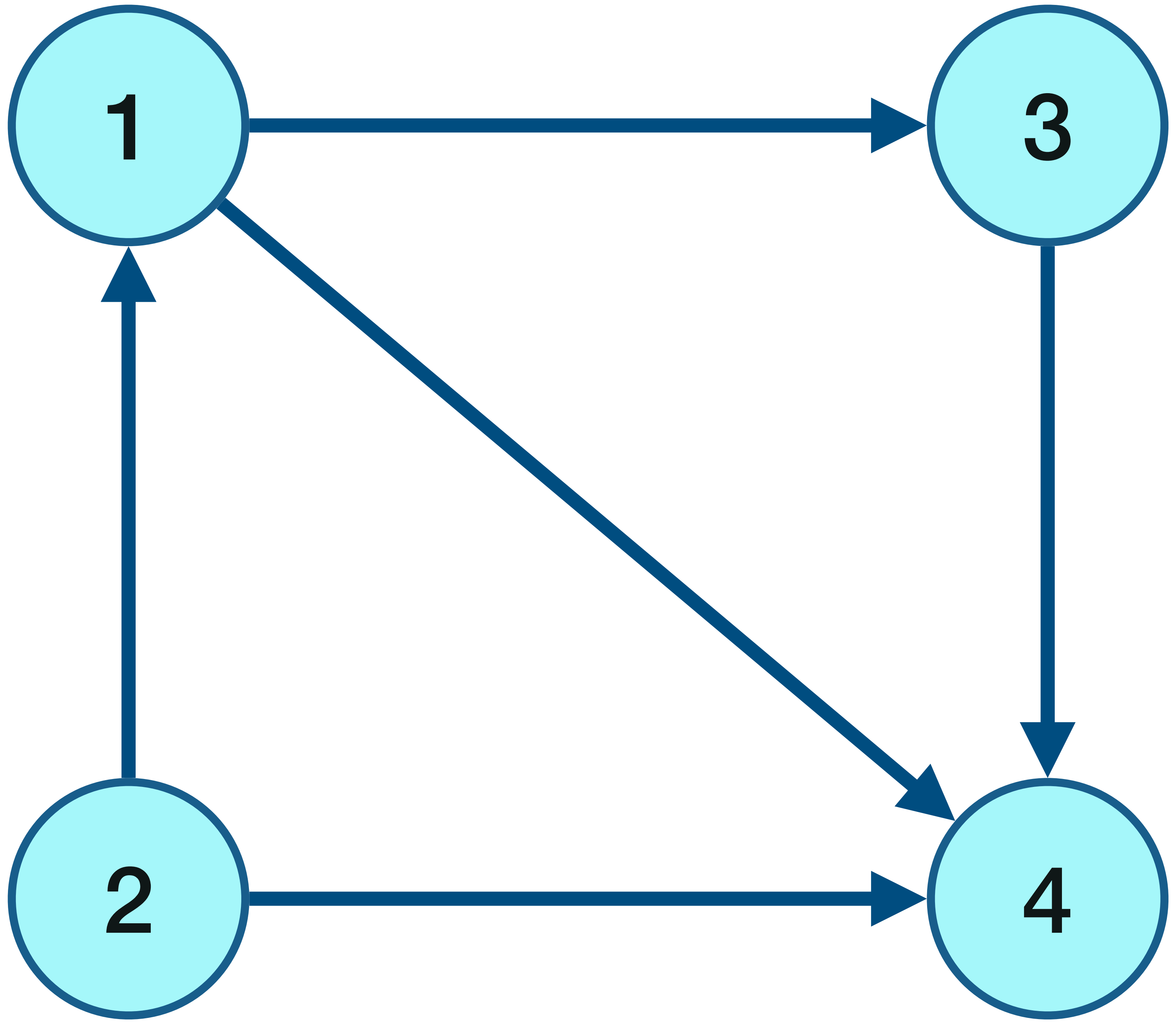
All mistakes are my own! - Ivan Abraham (Fall 2024)

Image by ChatGPT (probably collaborated with DALL-E)

Directed acyclic graphs

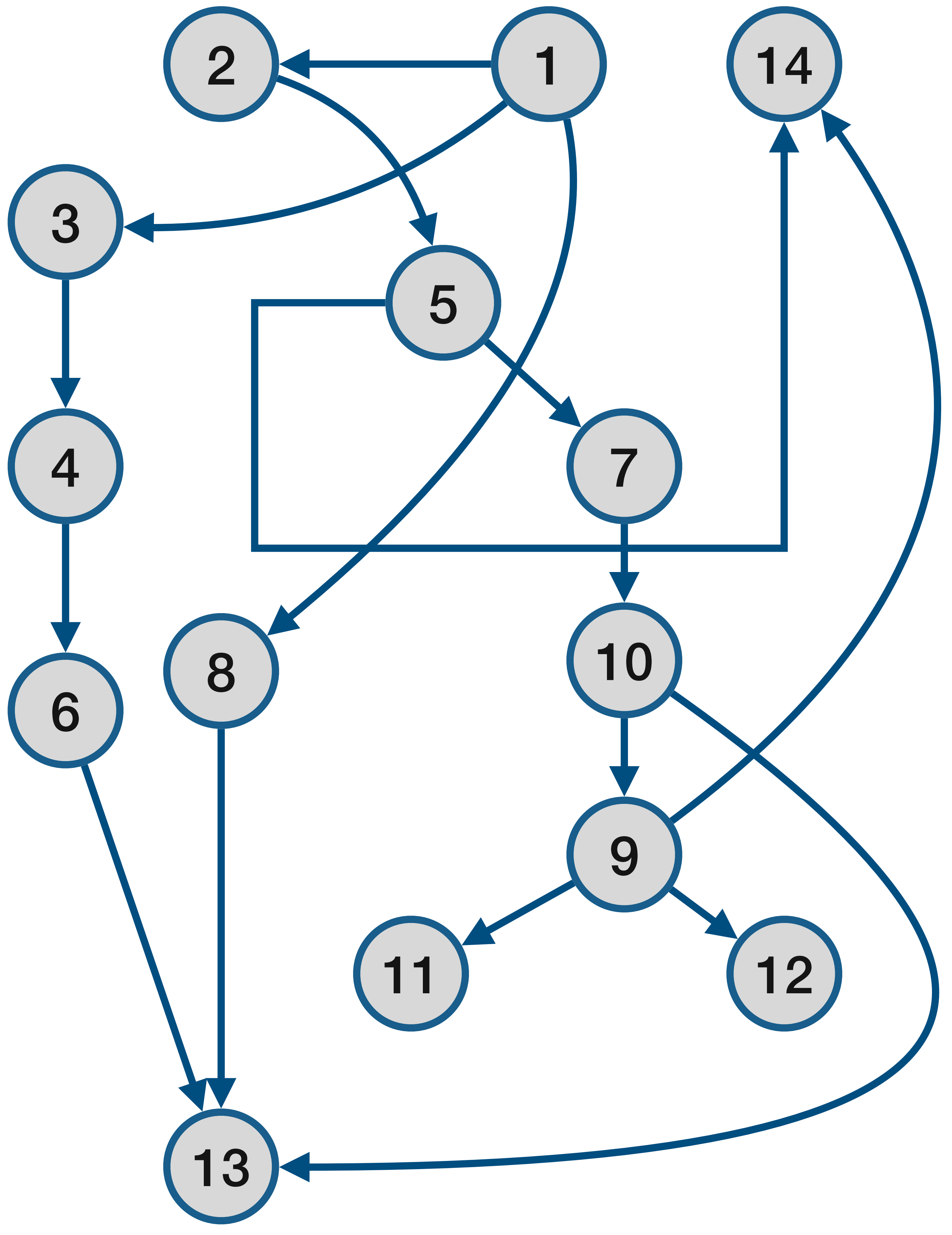
Definition

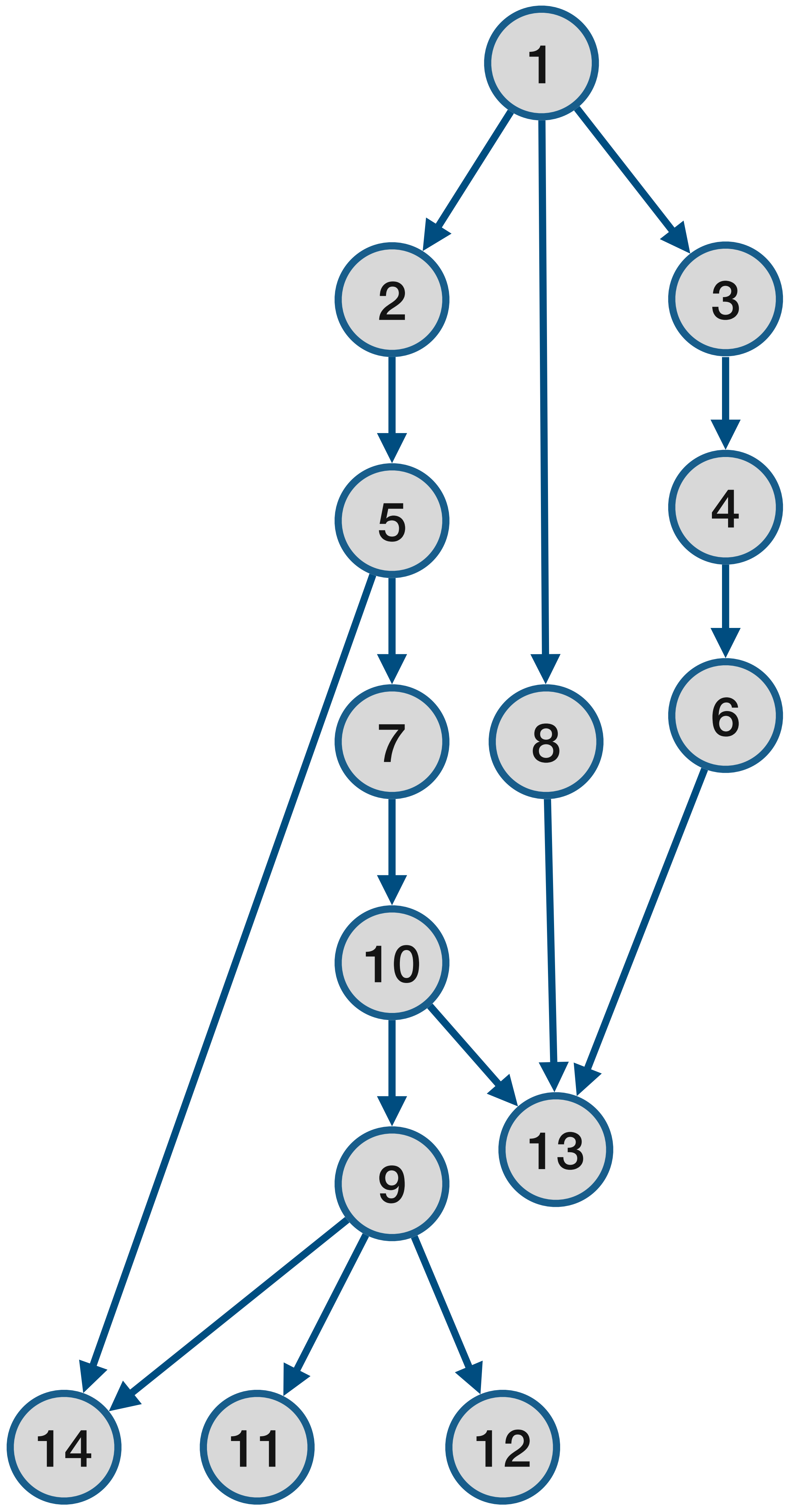
A directed graph G is called a *directed acyclic graph* (DAG) if there is no *directed* cycle in G.



Directed acyclic graphs

Is this a DAG?

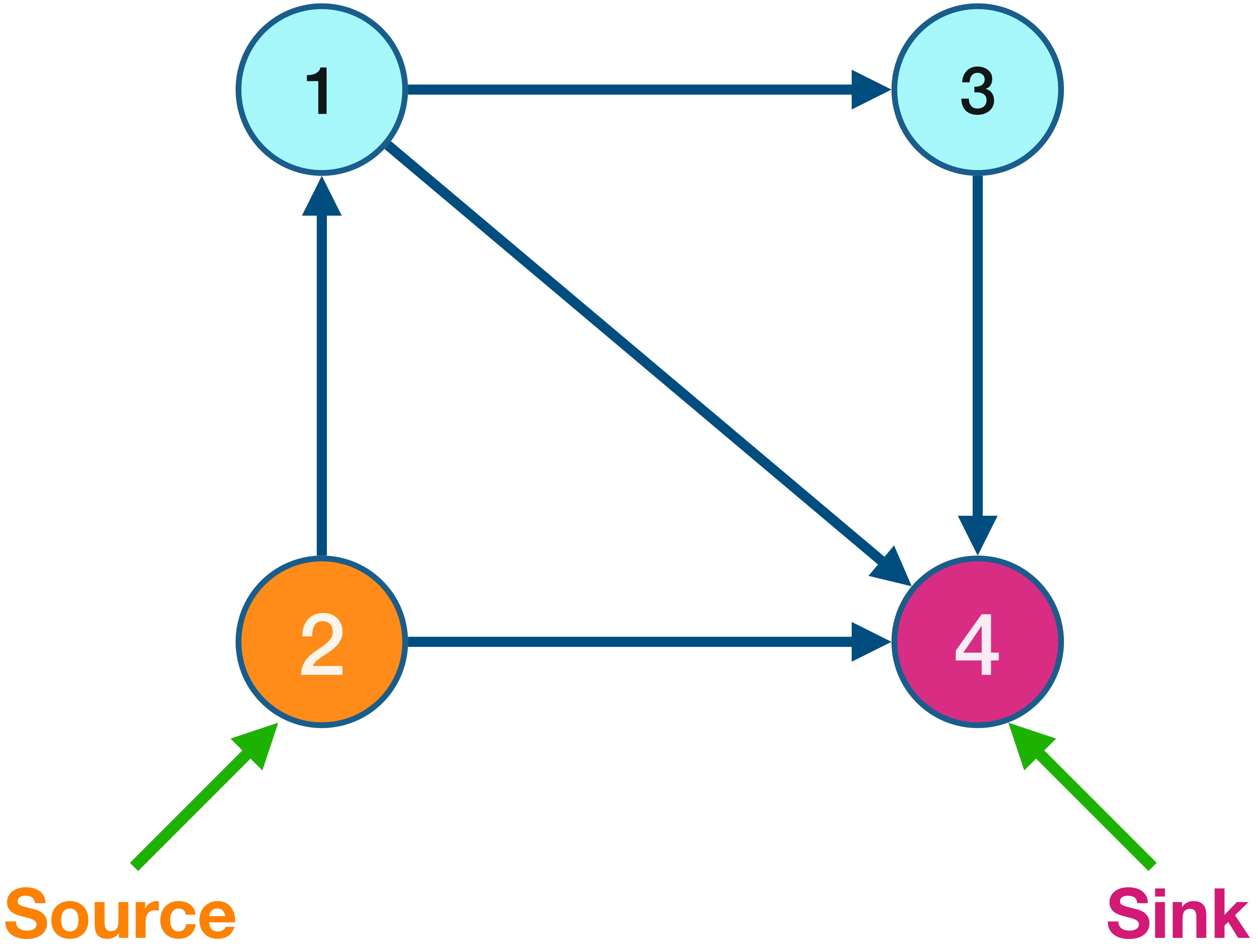




Directed acyclic graphs

Sources and sinks

- A vertex u is a **source** if it has no in-coming edges.
- A vertex u is a **sink** if it has no out-going edges



Directed acyclic graphs

Properties

Proposition: Every *finite* DAG G has at least one source and at least one sink

Proof

Let $P = v_1, v_2, \ldots, v_k$ be the longest path in G. We claim that v_1 is a source and v_k is a sink.

For contradiction, suppose it is not. Then v_1 has an incoming edge which either creates a cycle or a longer path both of which are contradictions.

Similarly so if v_k has an outgoing edge.

Directed acyclic graphs

Properties

- G is a DAG if and only if G^{rev} is a DAG.
  - Recall G^{rev} is the graph G with orientation of all edges reversed.

- G is a DAG if and only each node is its own strongly connected component.
  - In other words, a (directed) graph is acyclic, iff it has no strongly connected subgraphs with more than one vertex.

Topological ordering

Order on a set

A *strict total* order on a set X is a binary relation $\prec$ on X such that:

- $\prec$ is transitive.
- For any $x, y \in X$, exactly one of the following holds:

$$x \prec y \text{ or } y \prec x \text{ or } x = y$$

- Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec x_2, \ldots, x_{m-1} \prec x_m$ and $x_m \prec x_1$.

Note about convention

- We will consider the following notations equivalent
  - Undirected graph edges:

$$uv = \{u, v\} = vu \in E$$

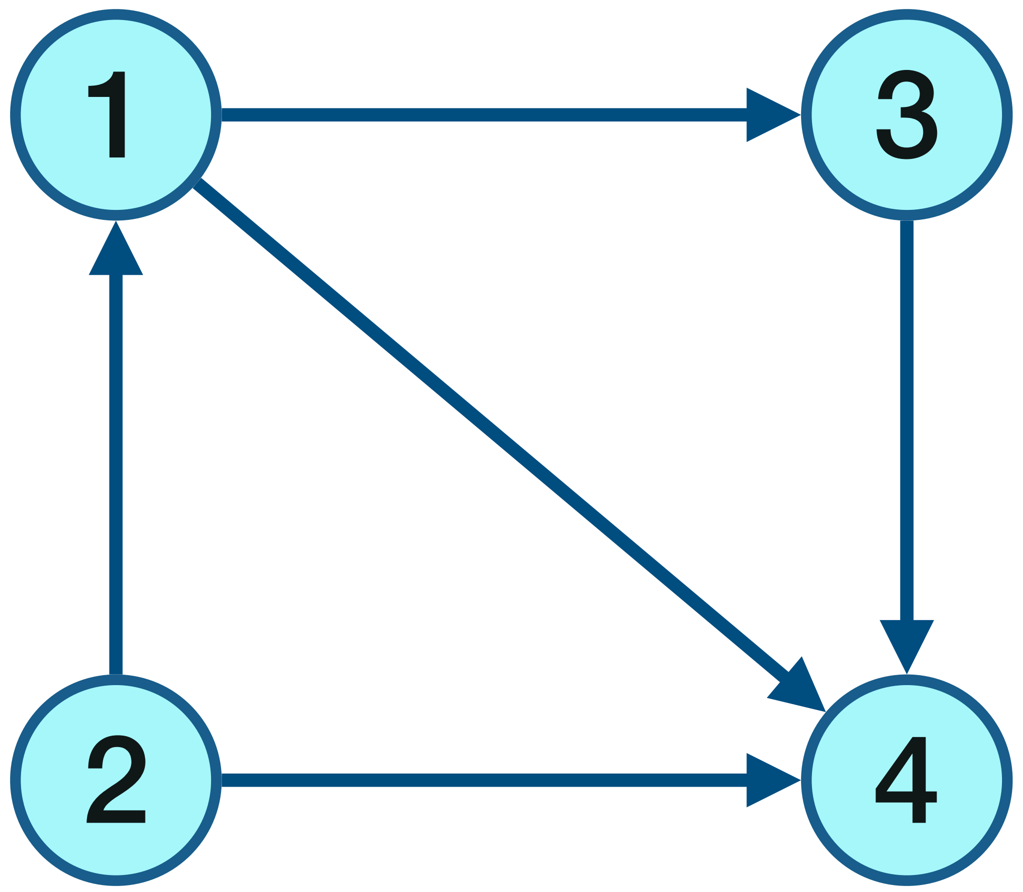
  - Directed graph edges:

$$u \to v \quad \equiv \quad (u, v) \quad \equiv \quad (u \to v)$$

Topological ordering/sorting

Definition

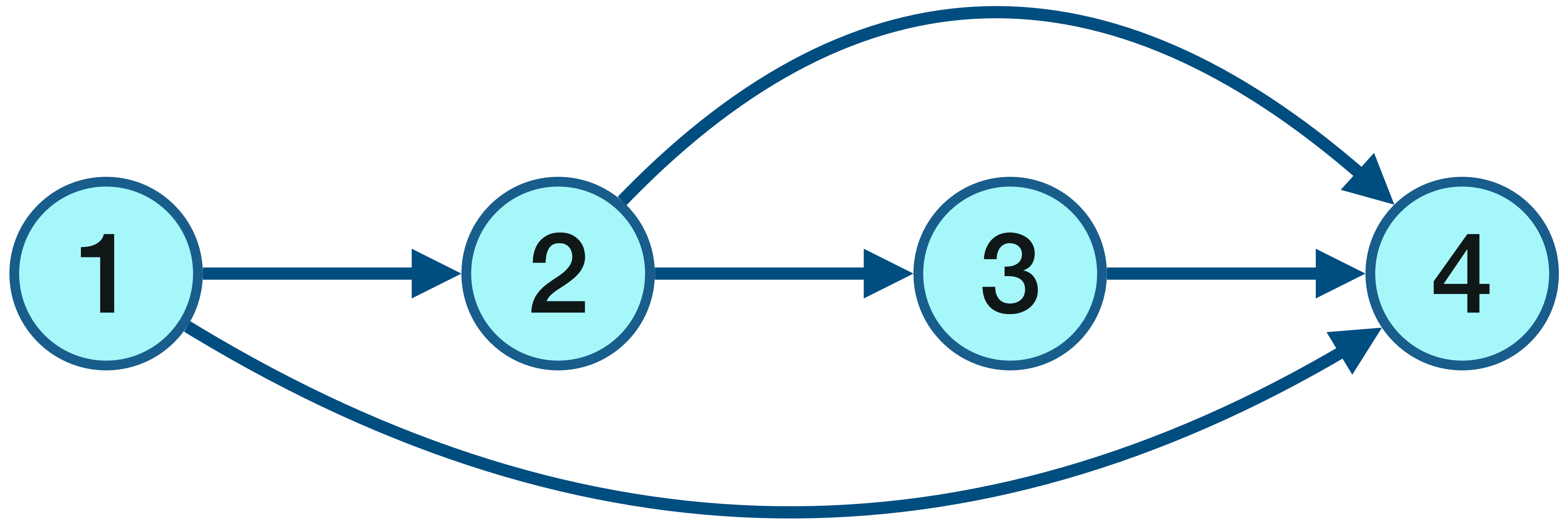
A ***topological ordering / topological sorting*** of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u \to v) \in E$ then $u \prec v$.



Graph G

Informal equivalent definition:

One can order the vertices of the graph along a line (say the x-axis) such that all edges are from left to right.



Topological Ordering of G

Topological ordering in linear time

Exercise

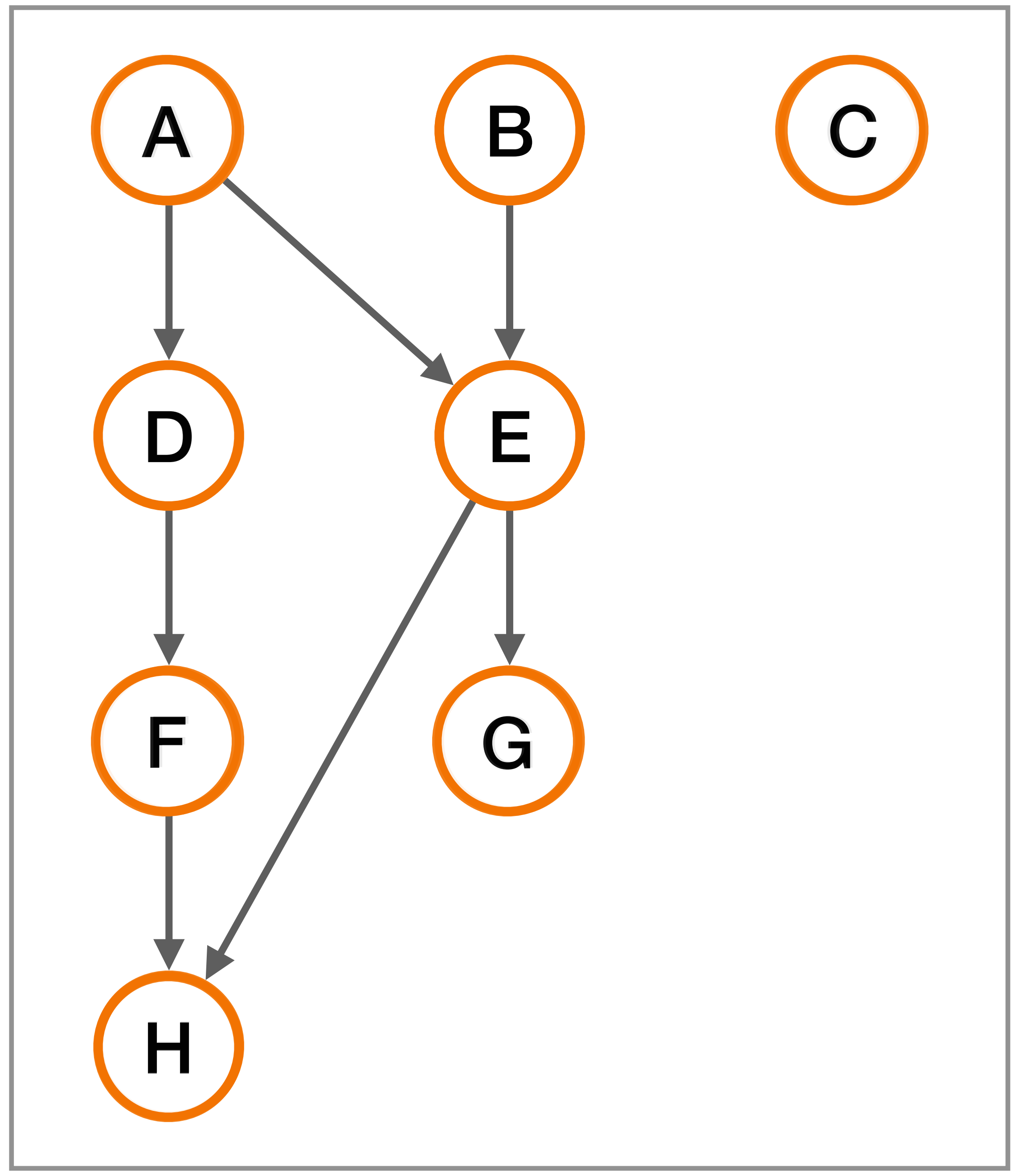
Show algorithm can be implemented in $O(m+n)$ time

Simple algorithm:

- Count the in-degree of each vertex
- For each vertex that is source, i.e., $\deg_{In}(v) = 0$:
  - Add v to the topological sort
  - Lower degree of vertices v is connected to.

Topological sort

Example



Adjacency List:

Node	Neighbors
A	D E
B	E
C	
D	F
E	H G
F	H
G	
H	

Generate $\deg_{In}(v)$:

Degree	Vertices
0	A B C
1	
2	

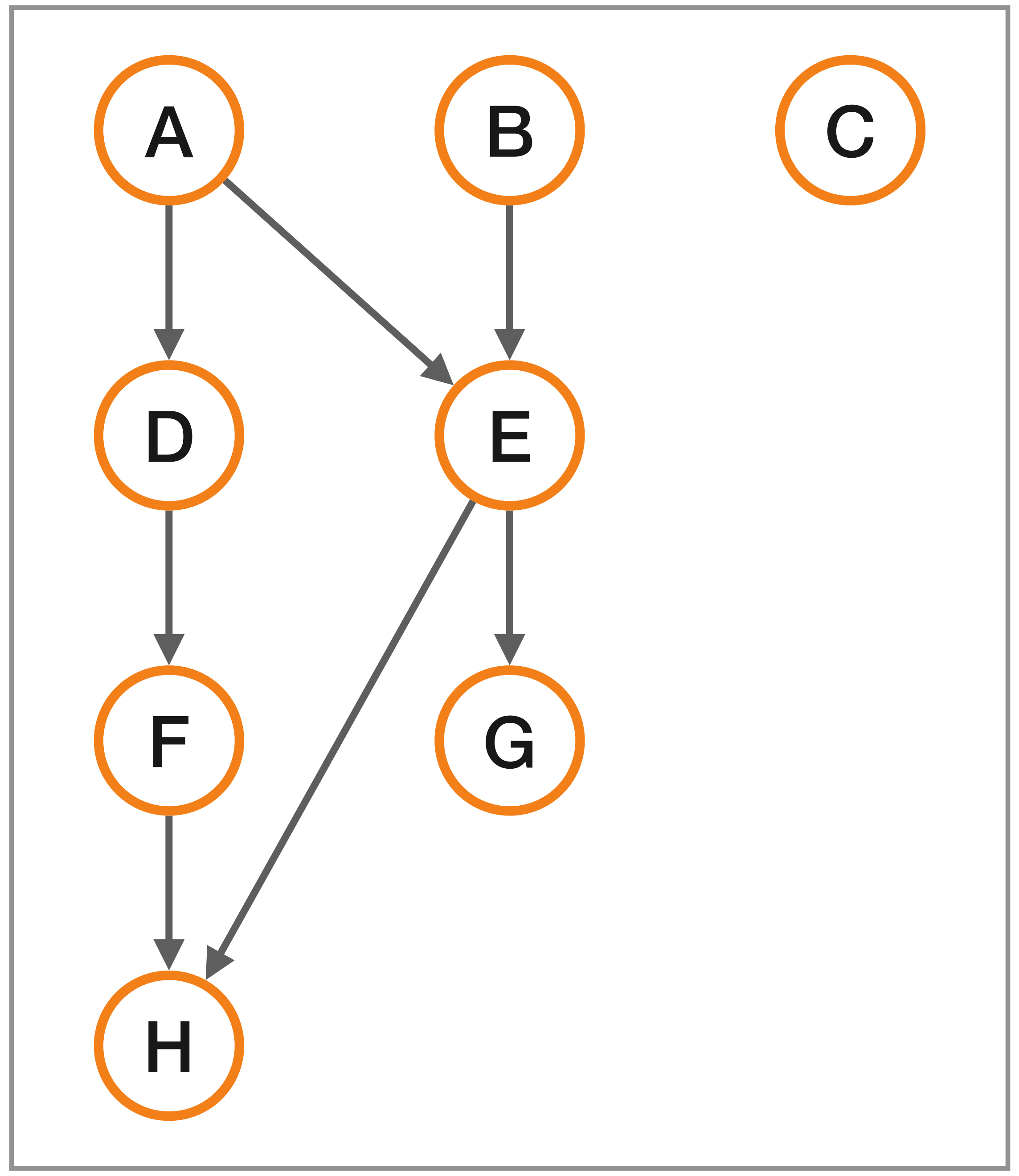
For each vertex that is source ($deg_{in}(v) = 0$):

- Add v to the topological sort
- Lower degree of vertices v is connected to.

Repeat the steps again.

Topological Ordering:

Topological Sort



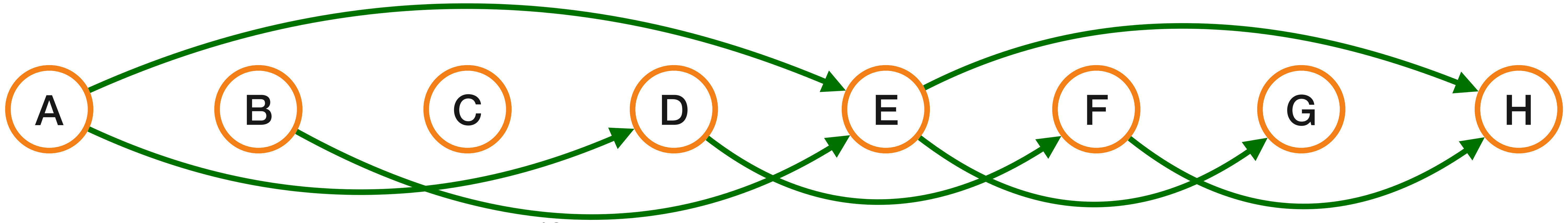
Node	Neighbors
A	D E
B	E
C	
D	F
E	H G
F	H
G	
H	

Degree	Vertices
0	A B C D E F G H
1	
2	

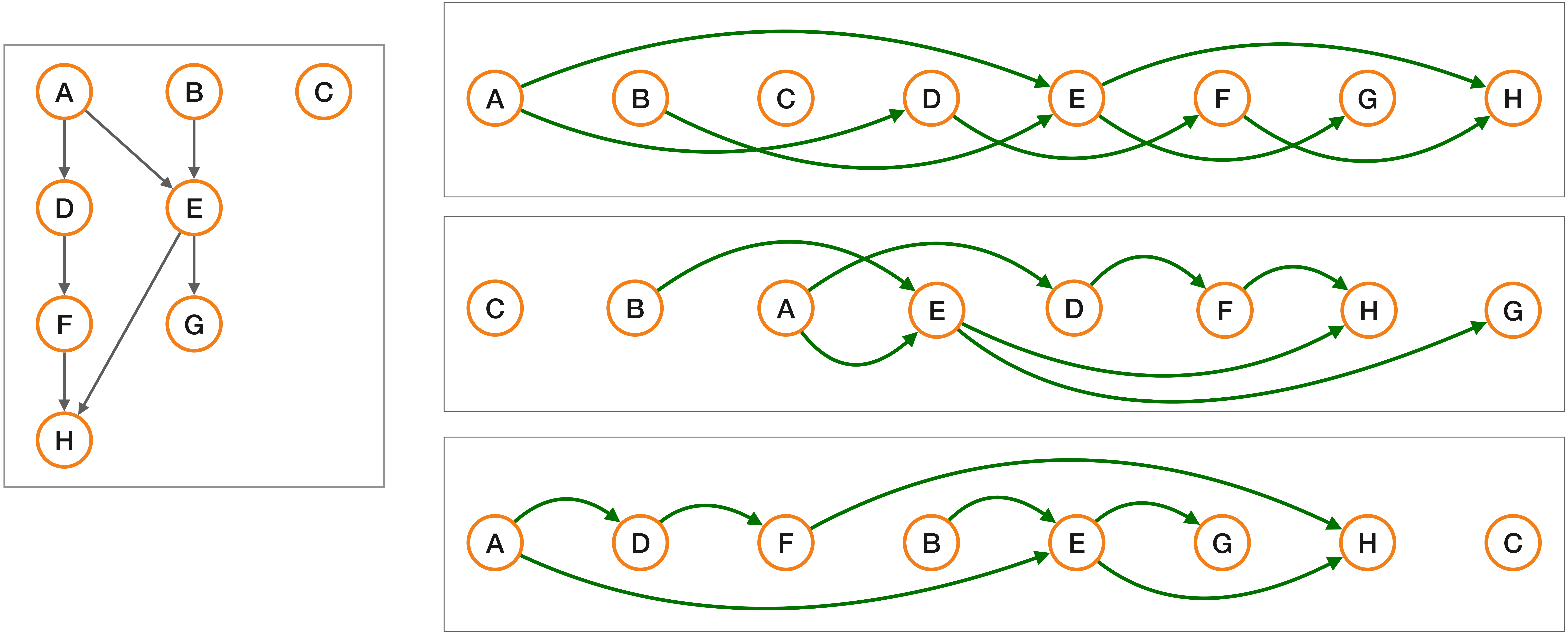
For each vertex that is source ($deg_{in}(v) = 0$):

- Add v to the topological sort
- Lower degree of vertices v is connected to.

Topological Ordering:



Multiple possible topological orderings



DAGs and topological ordering

- **Note:** A DAG G may have many different topological sorts.
- **Exercise:** What is a DAG with the most number of distinct topological sorts given n vertices?
- **Exercise:** What is a DAG with the least number of distinct topological sorts for given n vertices?

Direct topological ordering

```
TopSort(G):
  Sorted ← NULL
  deg_in[1 … n] ← −1
  Tdeg_in[1 … n] ← NULL
  Generate in-degree for each vertex
  for each edge xy in G do
       deg_in[y]++
  for each vertex v in G do
      Tdeg_in[deg_in[v]].append(v)
  Next we recursively add vertices with in-degree = 0 to
  the sort list
  while (Tdeg_in[0] is non-empty) do
        Remove node x from Tdeg_in[0]
        Sorted.append(x)
        for each edge xy in Adj(x) do
            deg_in[y]--
            move y to Tdeg_in[deg_in[y]]
  Output Sorted
```

DAGs and topological ordering

Lemma: A directed graph G can be topologically ordered $\implies$ G is a DAG

Proof: Proof by contradiction. Suppose G is not a DAG and ***has*** a topological ordering $\prec$. Since G is not a DAG, WLOG, take a cycle:

$$C = u_1 \to u_2 \to \ldots u_k \to u_1 \,.$$

Then $u_1 \prec u_2 \prec \ldots \prec u_k \prec u_1 \implies u_1 \prec u_1$

A contradiction (to $\prec$ being an order). Not possible to topologically order the vertices.

DFS in undirected graphs

Deep Dive into Depth First Search (DDiDFS?)

- Recall DFS is a special case of `BasicSearch`.
- DFS is useful in understanding graph structure.
- DFS also used to obtain linear time ($O(m+n)$) algorithms for
  - Finding *cycles,* search trees, etc.
  - Finding strong connected components of directed graphs
- ...many other applications as well.

Recursive DFS

Recursive version commonly implemented, has some desirable properties.

```
DFS(G):
    for all u ∈ V(G) do
        Mark u as unvisited
        Set pred(u) to null
    T is set to ∅
    while ∃ unvisited u do
        DFS(u)
    Output T
```

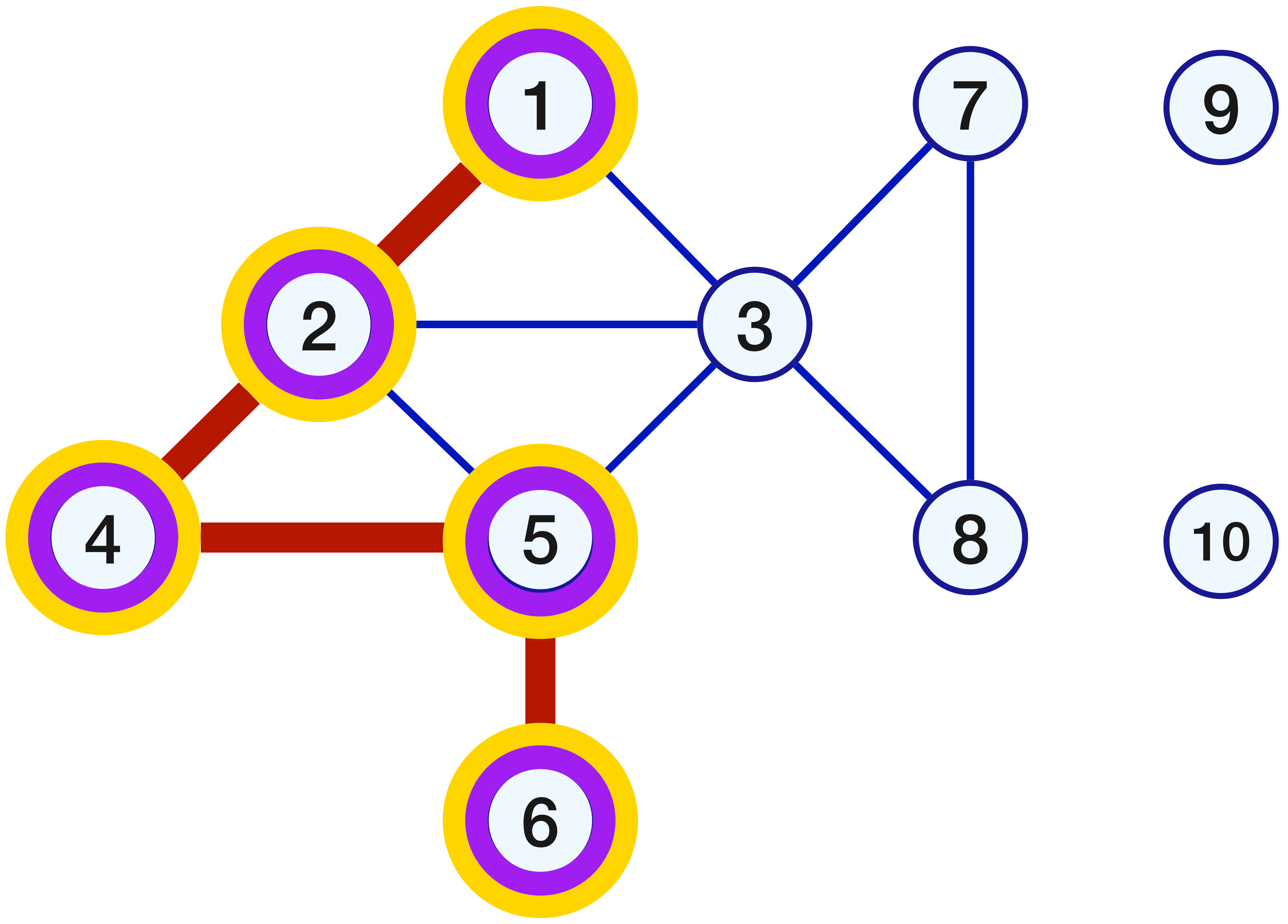
```
DFS(u):
    Mark u as unvisited
    for each v ∈ Out(u) do
        if v is not visited then
            add edge u → v to T
            set pred(v) to u
            DFS(v)
```

Implemented using a global array *Visited* for all recursive calls. T is the search tree/forest/

DFS with pre-post numbering

Time = 0

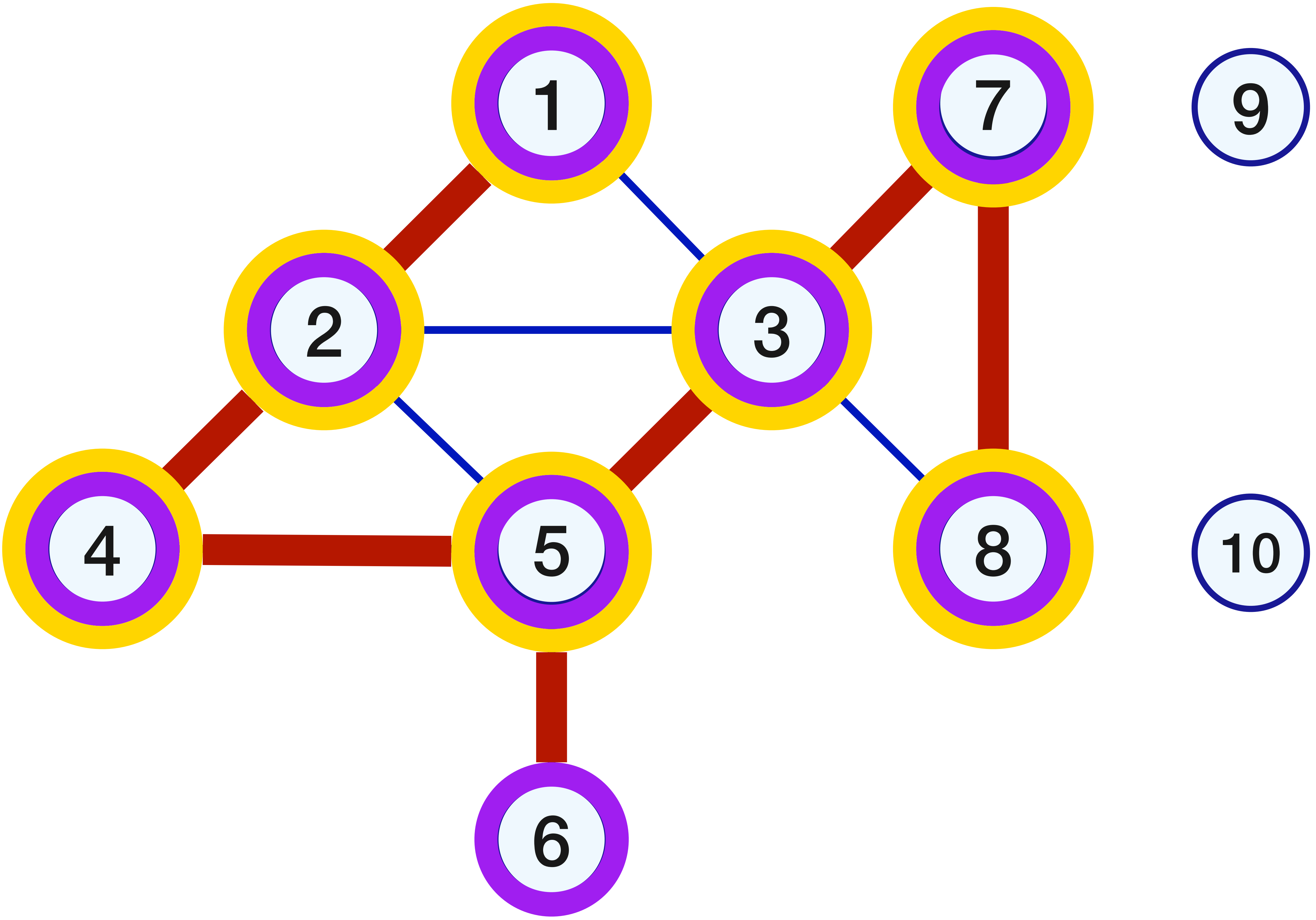
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5,]



DFS with pre-post numbering

Time = 9

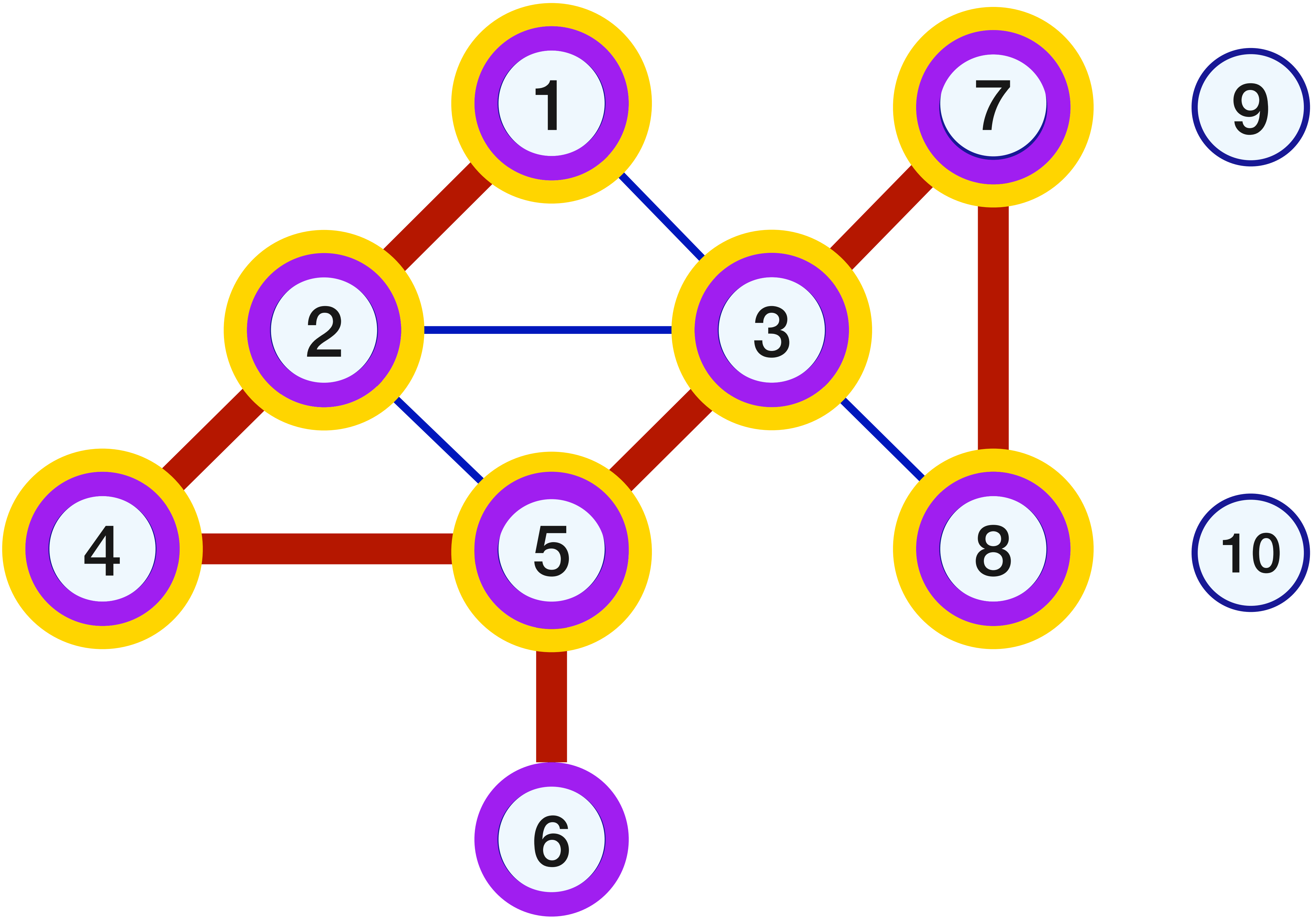
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9,]



DFS with pre-post numbering

Time = 16

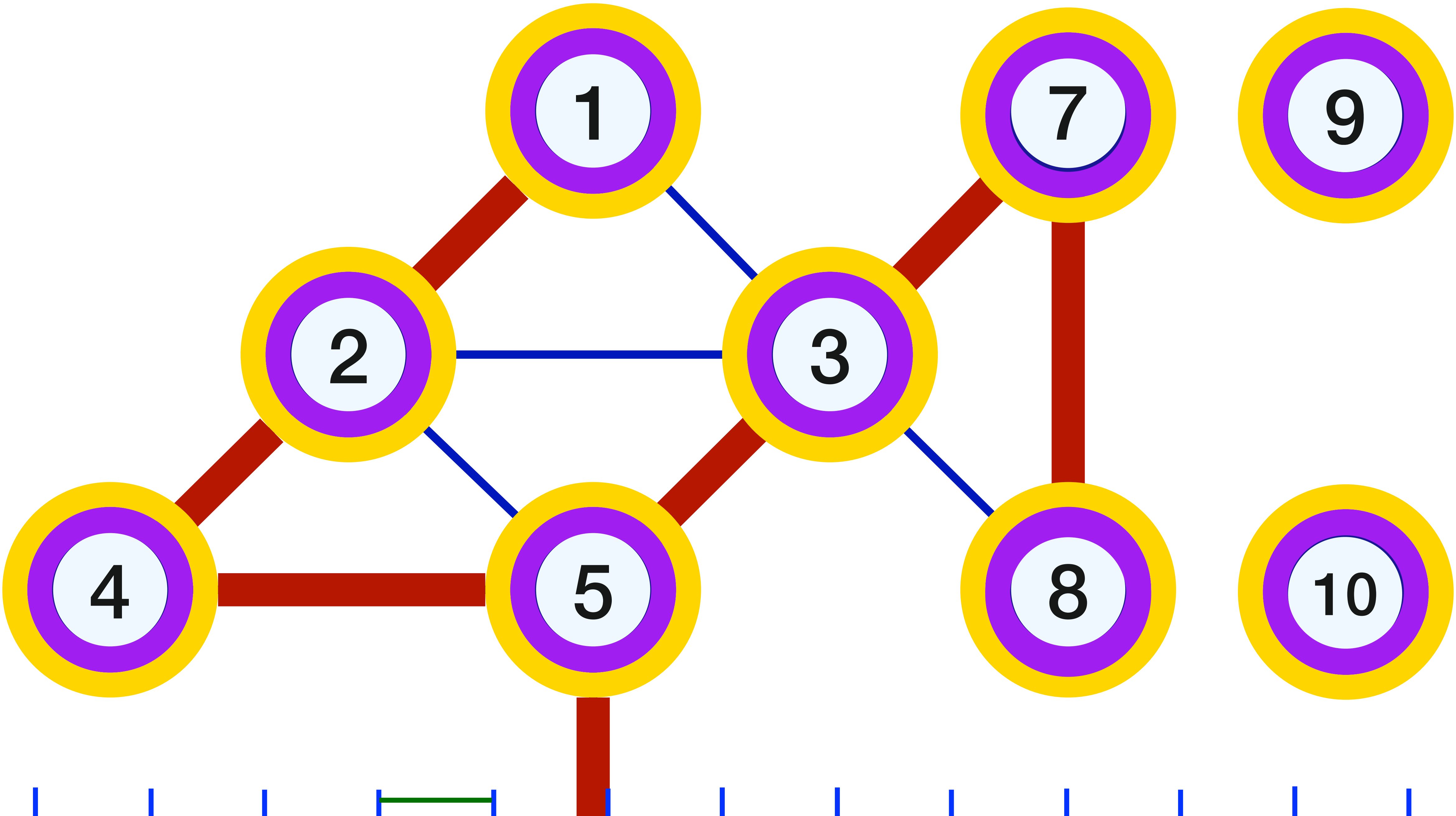
Vertex	*[Pre, Post]*
1	[1, 16]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]

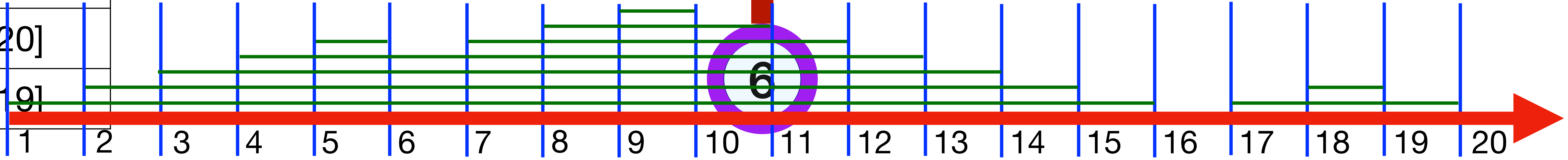


DFS with pre-post numbering

Time = 20

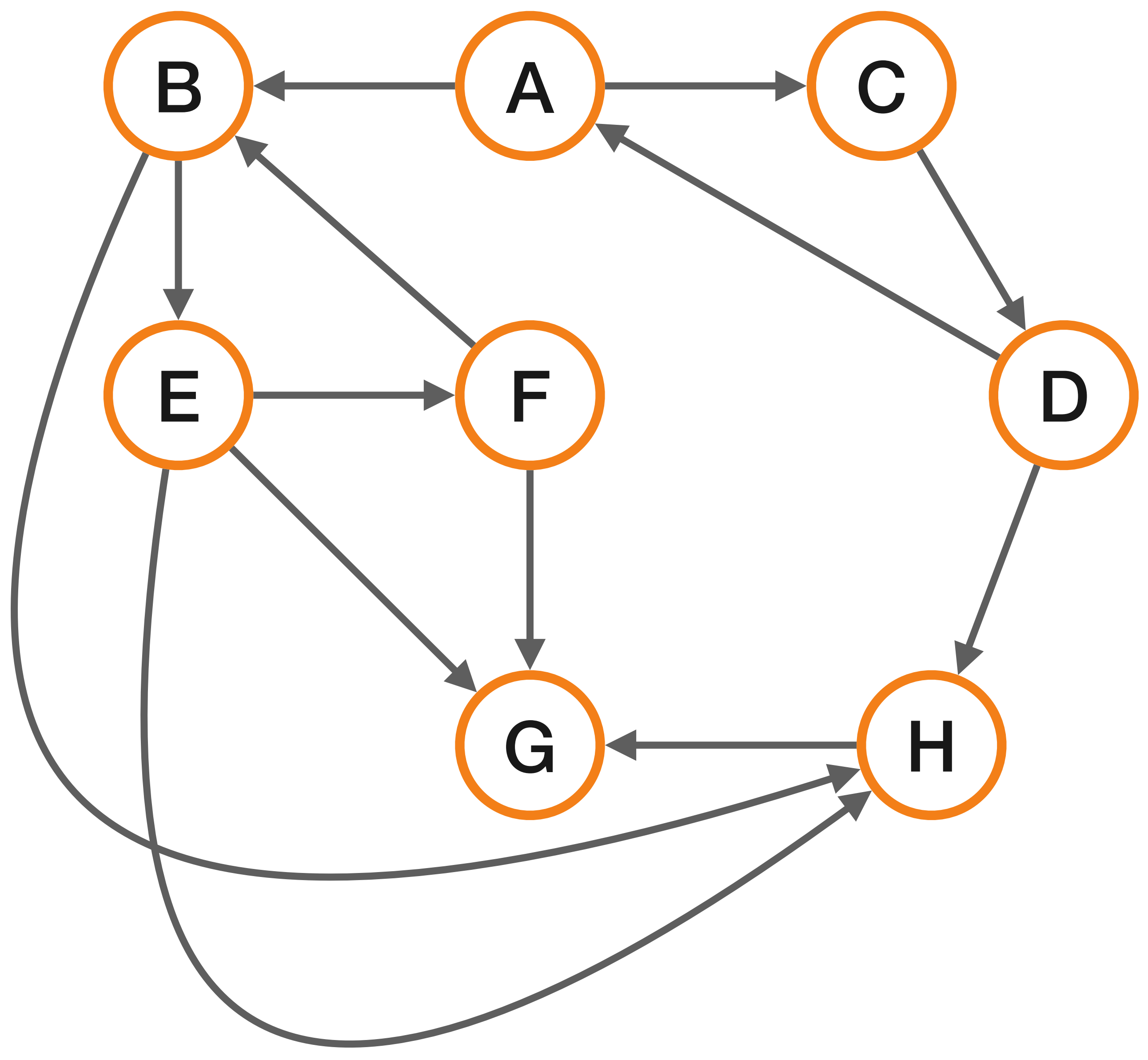
Vertex	*[Pre, Post]*
1	[1, 16]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 13]
8	[9, 10]
9	[17, 20]
10	[19, 19]

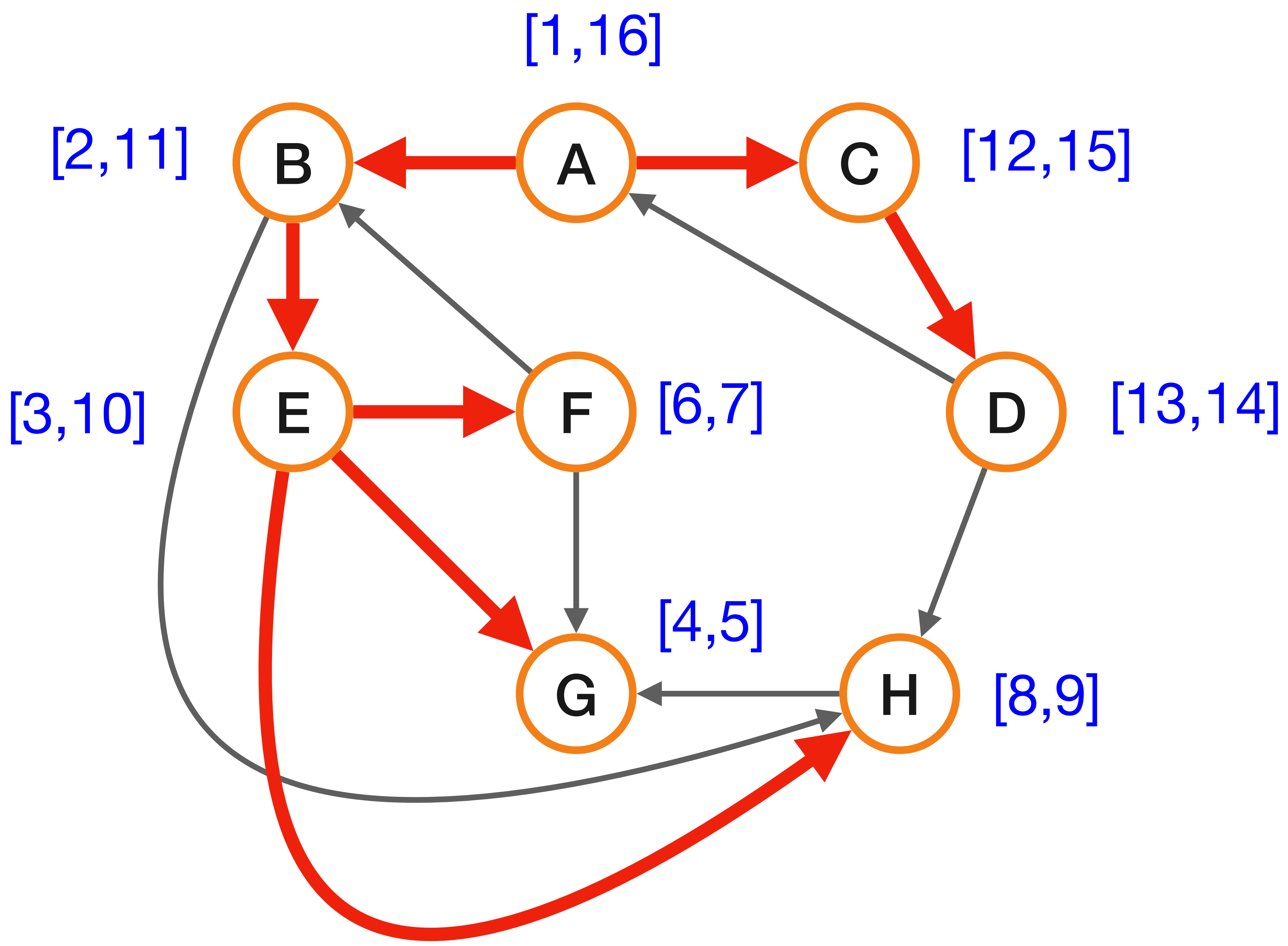




DFS in directed graphs

Exercise - do DFS on this graph and verify search tree





Directed DFS with pre/post numbering

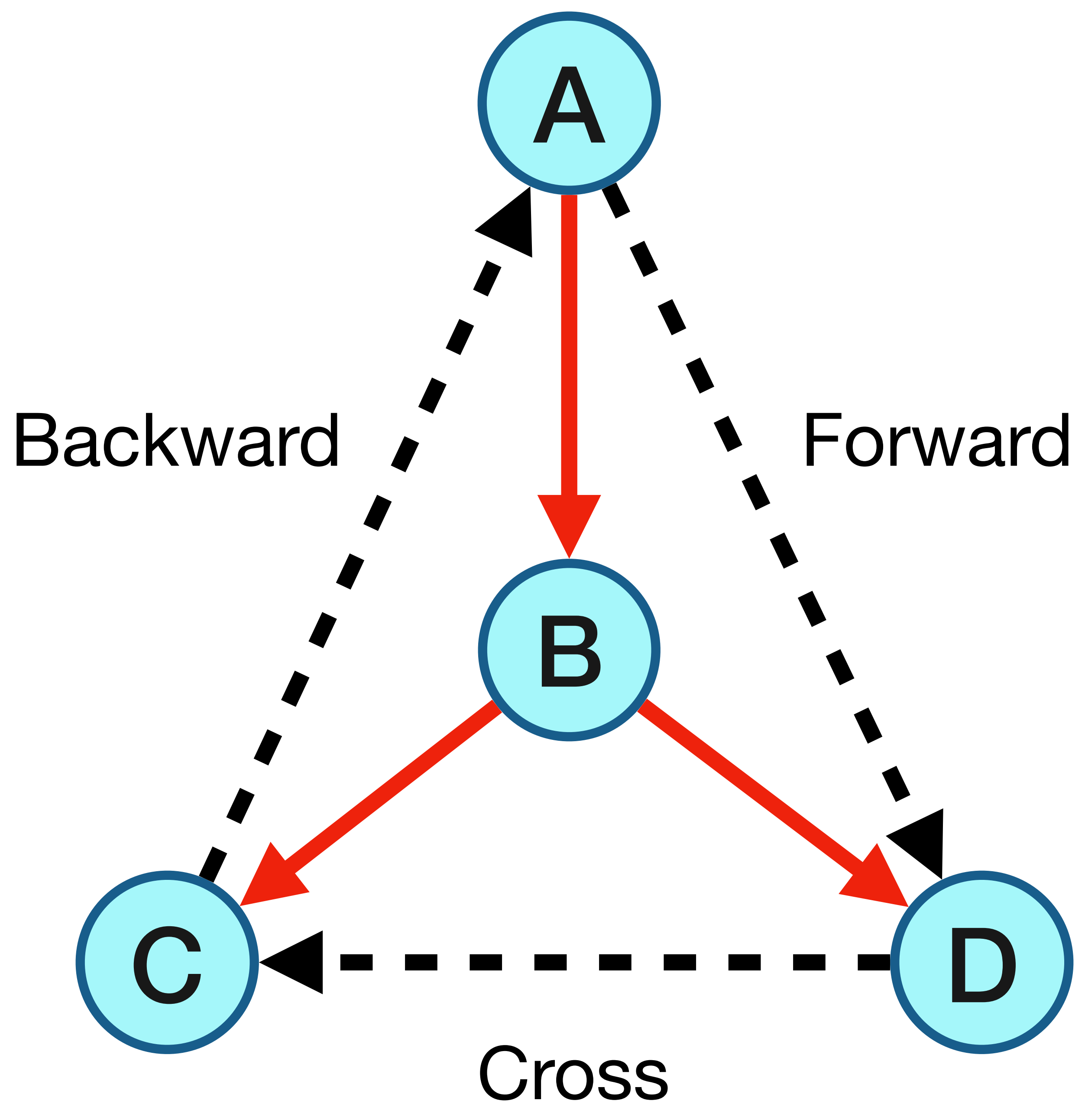
- *DFS(G)* takes $O(m+n)$ time.
- Edges added form a *branching*: a forest of **out**-trees.
  - *Output of DFS(G) depends on the order in which vertices are considered.*
- If u is the first vertex considered by *DFS(G)* then *DFS(u)* outputs a directed out-tree T rooted at u and a vertex v is in T if and only if $v \in rch(u)$
- For any two vertices x, y the intervals $[\text{pre}(x), \text{post}(x)]$and $[\text{pre}(y), \text{post}(y)]$ are either disjoint or one is contained in the other.

DFS trees and edge types

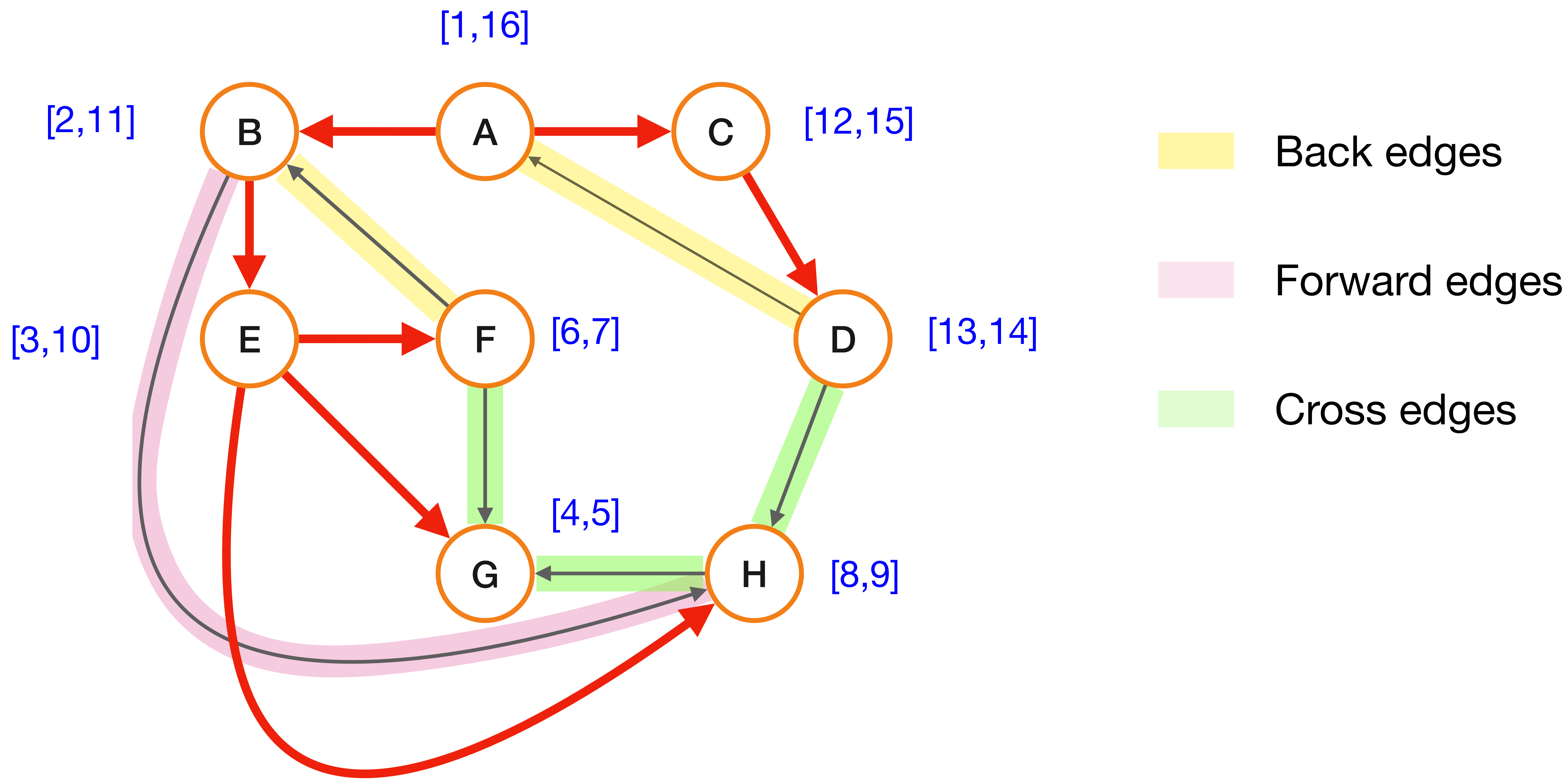
Edge classisifcations

Edges of G can be classified with respect to the DFS tree T as:

- Tree edges that belong to T
- A forward edge is a non-tree edges (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A backward edge is a non-tree edge (y, x) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A cross edge is a non-tree edges (x, y) such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.



Types of edges



DFS and cycle detection

Cycles in graphs

- **Question:** Given an undirected graph how do we check whether it has a cycle and output one if it has one?

- **Question:** Given an directed graph how do we check whether it has a cycle and output one if it has one?

Cycle detection in directed graphs

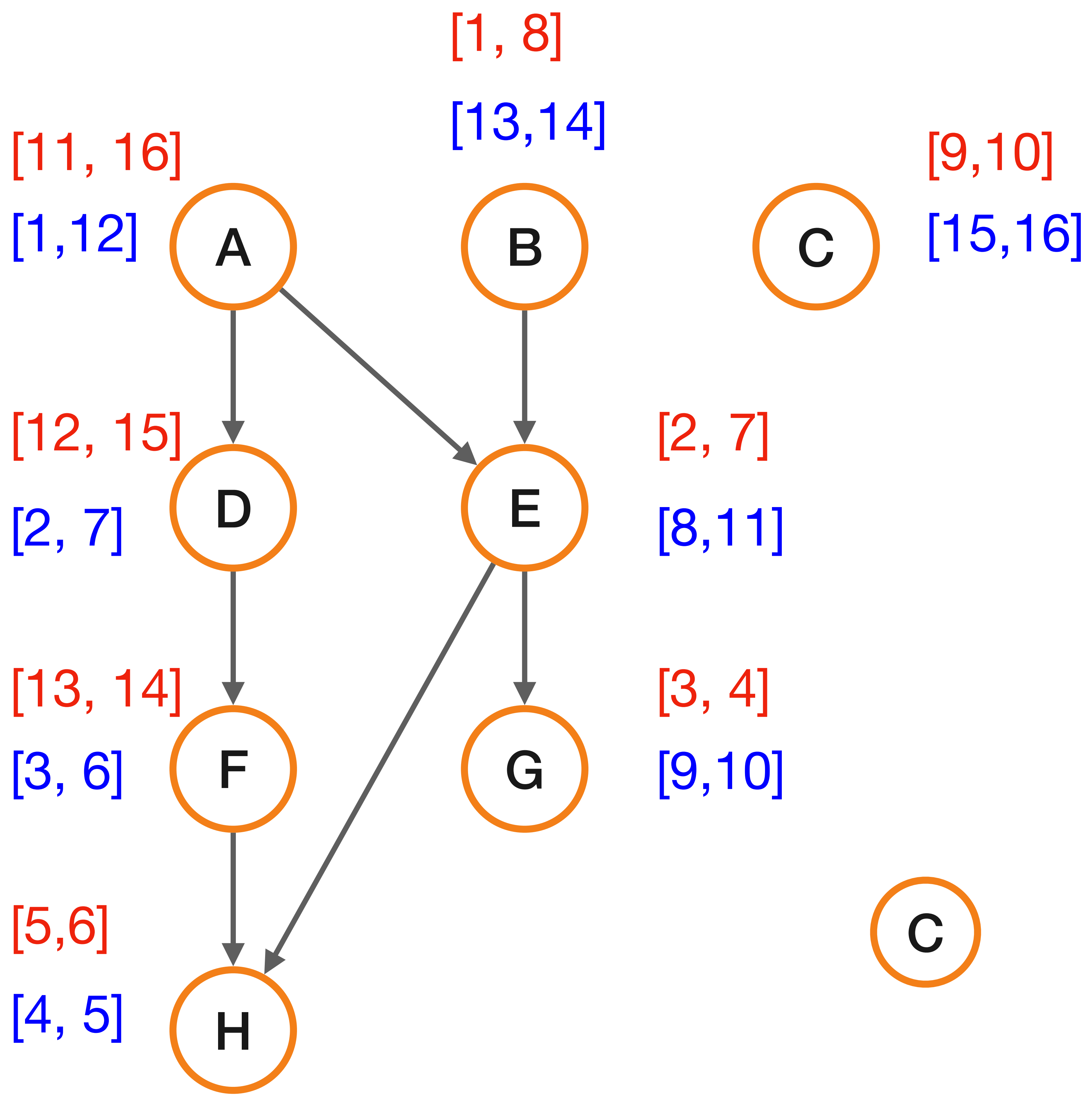
Use topological sorts

Question: Given G, is it a DAG?

- If it is, compute a topological sort. If it fails, then output the cycle C.
  - Compute $DFS(G)$.
  - If there is a back edge $e = (v, u)$ then G is not a DAG. Output cycle C formed by path from u to v in T plus edge (v, u).
  - Otherwise output nodes in decreasing post-visit order.
  - **Note**: no need to sort, $DFS(G)$ can output nodes in this order!

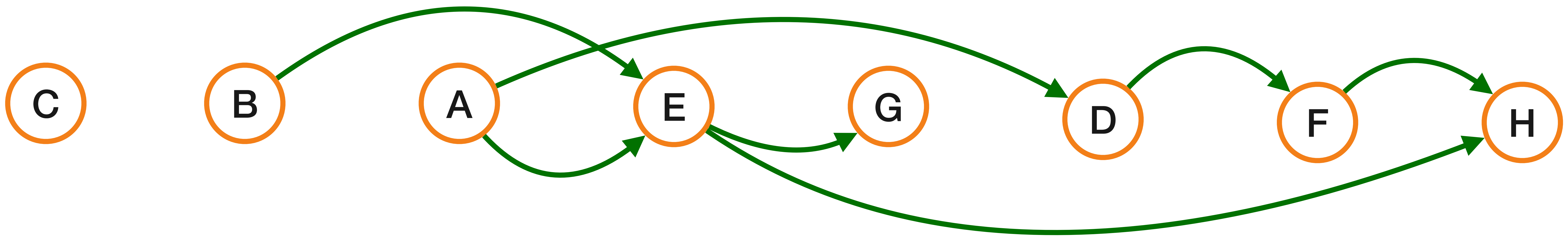
Topological sort a graph using DFS

Example



Listing out the vertices in descending order of post-visit numbers gives:

C, B, A, E, G, D, F, H



Back edge and cycles

Proposition: G has a cycle $\iff$ there is a *back-edge* in DFS(G).

Proof: That (u, v) is a back edge implies there is a cycle C consisting of the path from v to u in DFS search tree and the edge (u, v).

Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.

Let v_i be first node in C visited in DFS. All other nodes in C are descendants of v_i since they are reachable from v_i.

Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge

Decreasing post-visit order is a TS

Proposition: If G is a DAG and $\text{post}(v) > \text{post}(u)$, then $(u \to v)$ is not in G.

Proof: Assume $\text{post}(u) < \text{post}(v)$ and $(u \to v)$ is an edge in G. One of two holds:

- Case 1: $[\text{pre}(u), \text{post}(u)]$ is contained in $[\text{pre}(v), \text{post}(v)]$. Implies that u is explored during $DFS(v)$ and hence is a descendent of v. Edge (u, v) implies a cycle in G but G is assumed to be DAG.
- Case 2: $[\text{pre}(u), \text{post}(u)]$ is disjoint from $[\text{pre}(v), \text{post}(v)]$. This cannot happen since v would have been explored from u.

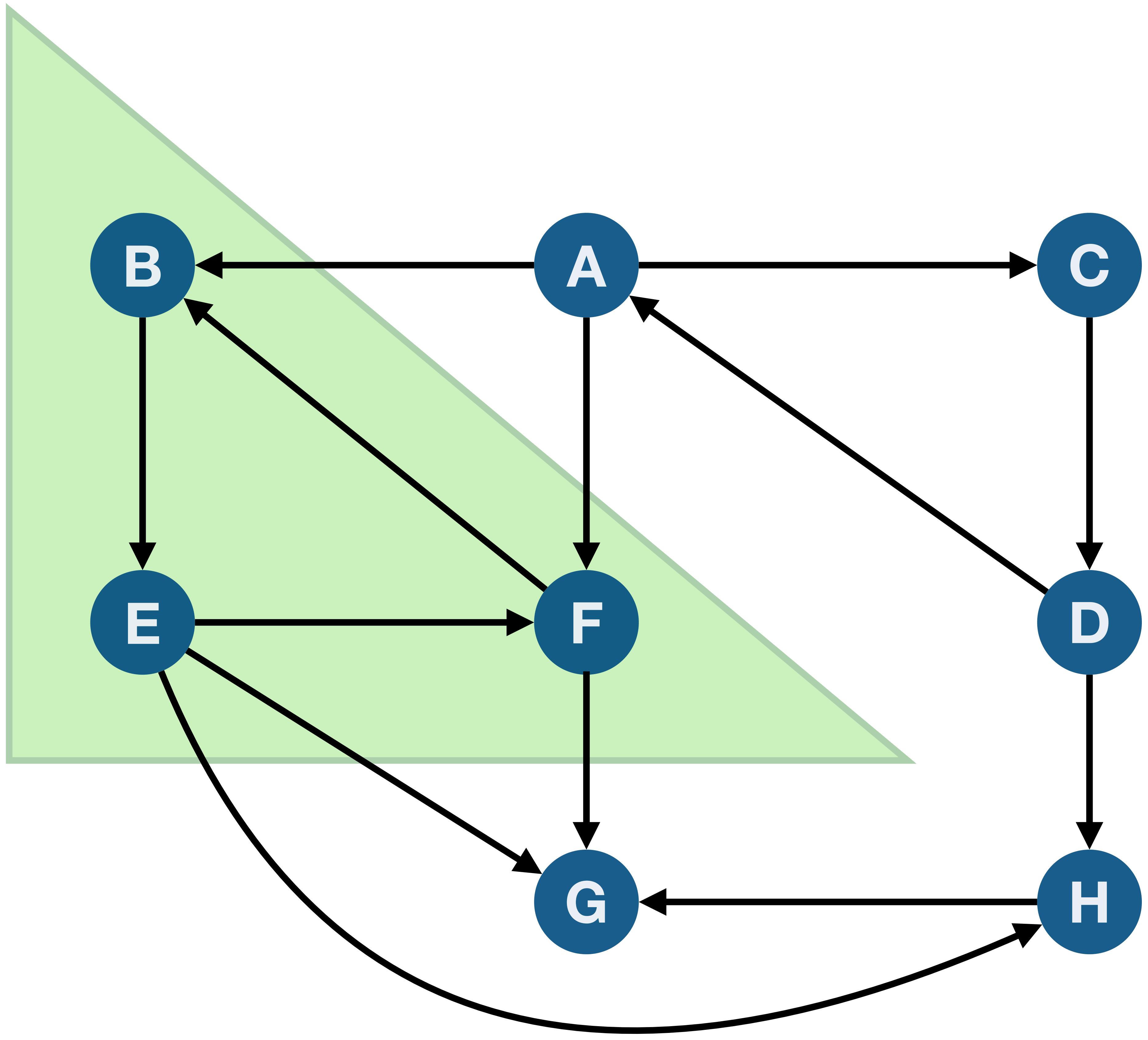
Strongly connected components (SCCs)

Algorithmic Problem

Find all SCCs of a given directed graph.

Previous lecture: Saw an $O(n \,.\, (n + m))$ time algorithm.

This lecture: Sketch of a $O(n + m)$ time algorithm.



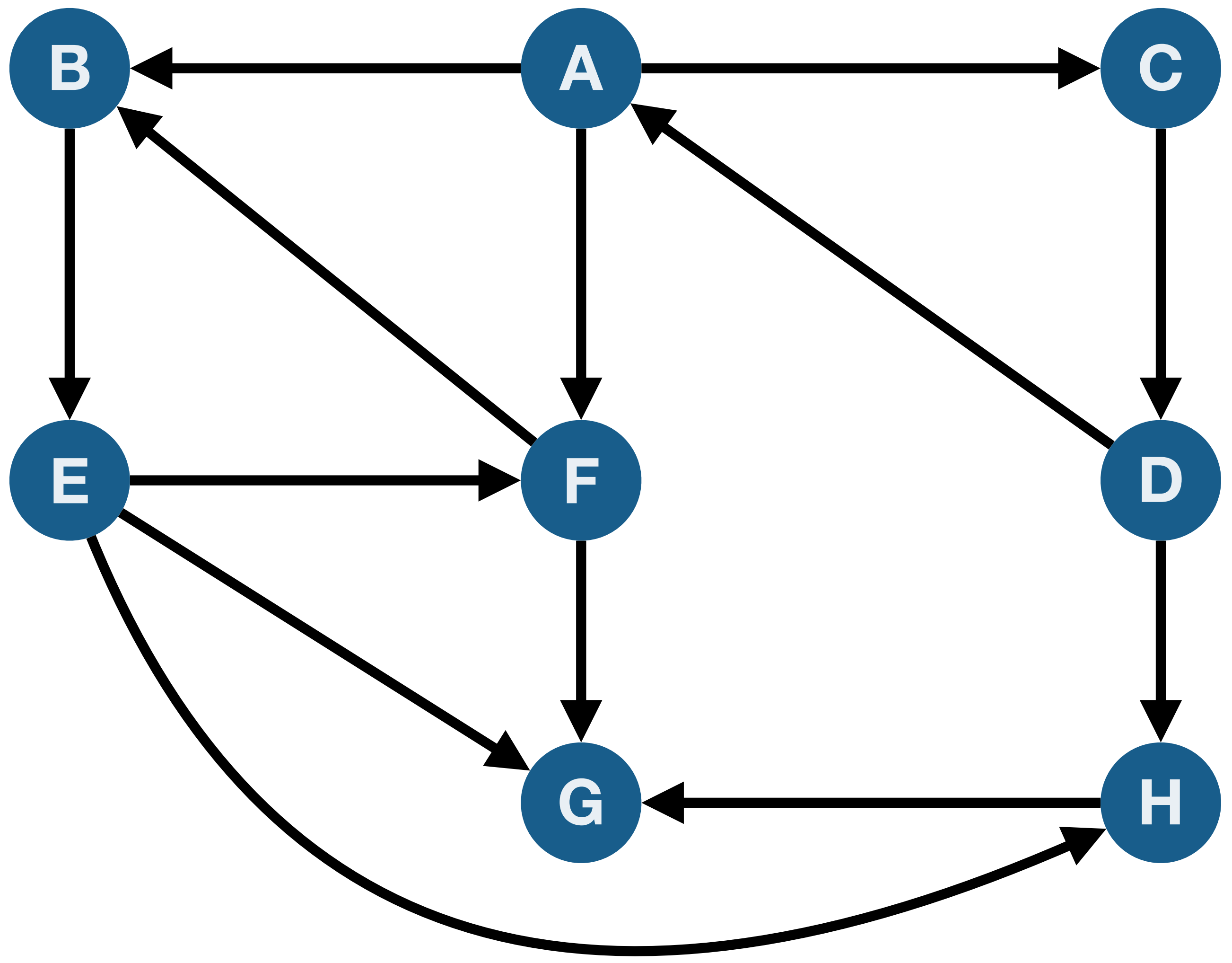
Graph of SCCs

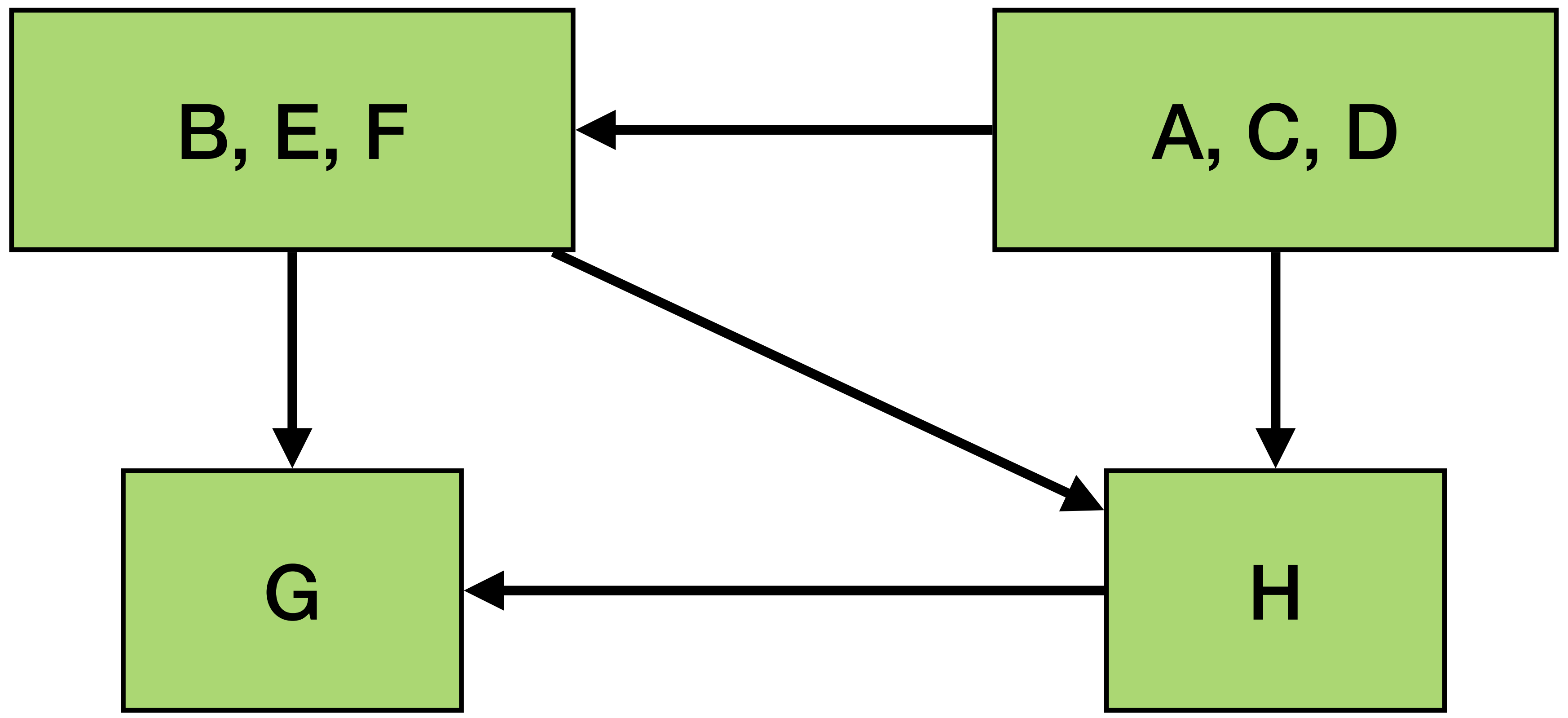
Meta-graph of SCCs

Let $S_1, S_2, \ldots S_k$ be the strongly connected components (i.e., SCCs) of G. Denote graph of SCCs is G^{SCC}:

- Vertices are $S_1, S_2, \ldots S_k$
- There is an edge (S_i, S_j) if there is some $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge in G.

For any graph G, the graph G^{SCC} has no directed cycle!





Graph of SCCs G^{SCC}

Structure of Graphs

- **Undirected graph:** connected components of $G = (V, E)$ and a partition of V can be computed in $O(m + n)$ time.
- **Directed graph:** the meta-graph G^{SCC} of G can be computed in $O(m + n)$ time. G^{SCC} gives information on the partition of V into strong connected components and how they form a DAG structure.

Above structural decomposition will be useful in several algorithms.

Linear time algorithm for finding all SCCs

Finding all SCCs of a Directed Graph

Problem: Given a directed graph $G = (V, E)$, output all its strong connected components.

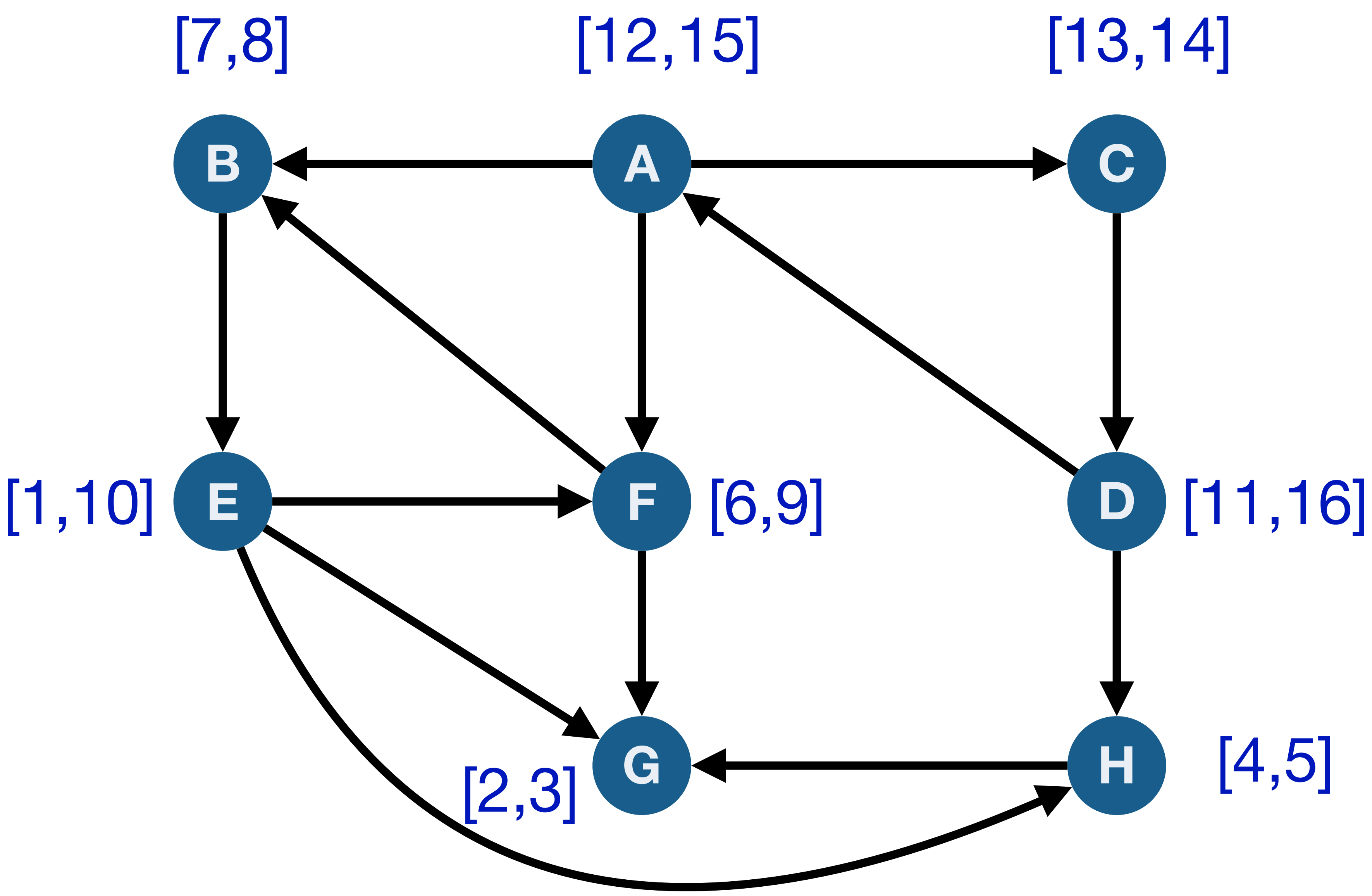
Straightforward algorithm:

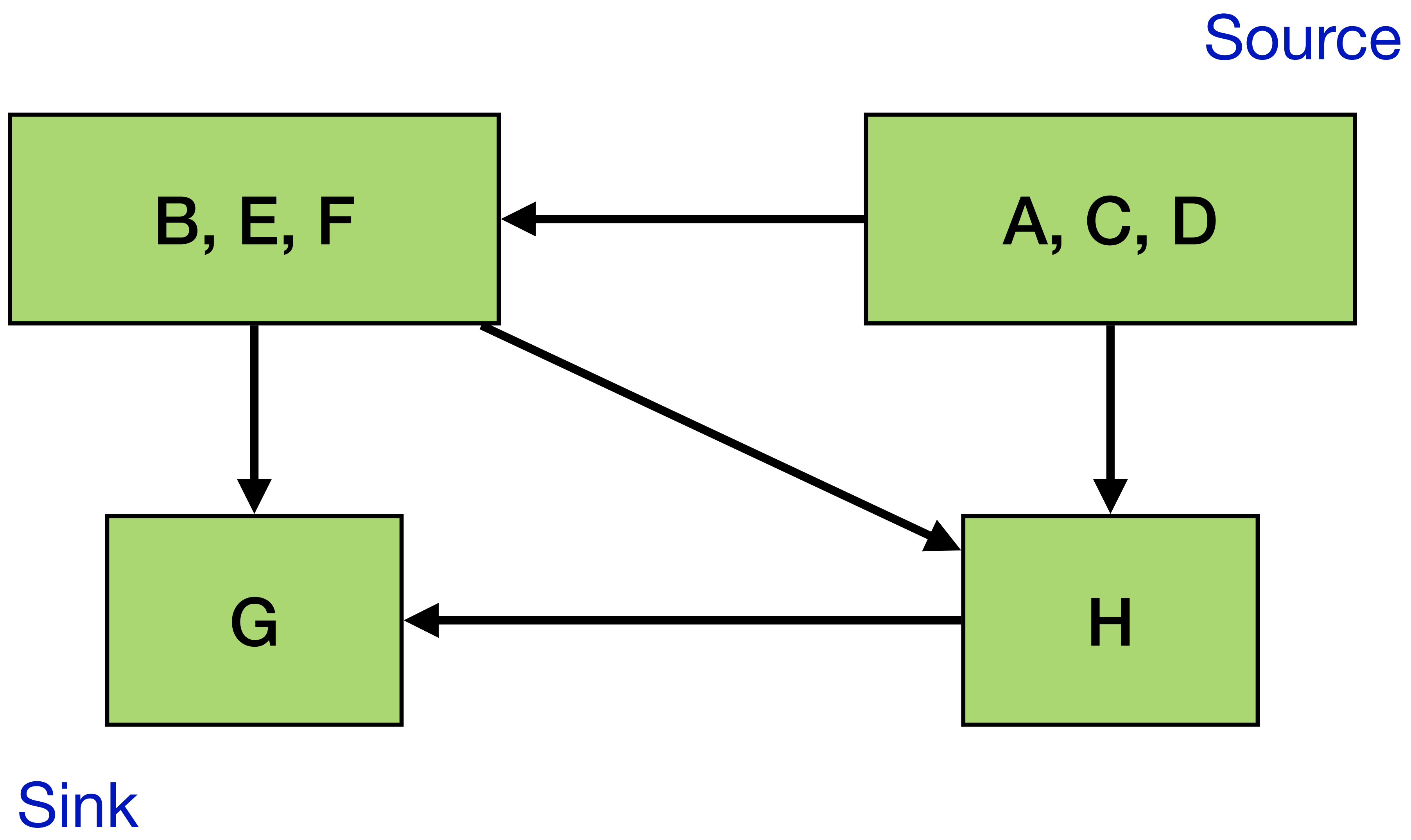
```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev , u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev , u)
        ∀u ∈ SCC(G, u): Mark u as visited.
```

Running time: $O(n(n+m))$

Is there an $O(n+m)$ time algorithm?

Structure of a Directed Graph





Reminder G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition: For a directed graph G, its meta-graph G^{SCC} is a DAG.

Linear-time Algorithm for SCCs

Ideas

Wishful thinking algorithm

- Let u be a vertex in a sink SCC of G^{SCC} .
- Do $DFS(u)$ to compute $SCC(u)$.
- Remove $SCC(u)$ and repeat.

Justification

- $DFS(u)$ only visits vertices (and edges) in $SCC(u)$ since there are no edges coming out of a sink!
- $DFS(u)$ takes time proportional to size of $SCC(u)$.
- Therefore, total time $O(n + m)$!

Questions

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an implicit topological sort of G^{SCC} without computing G^{SCC}?

Answer: $DFS(G)$ gives some information!

Pre/post-visit numbering and the meta graph

Claim: Let v be the vertex with maximum post-visit numbering in $DFS(G)$. Then v is in a SCC S, such that S is a source of G^{SCC}.

Claim: Let v be the vertex with maximum post-visit numbering in $DFS(G^{rev})$. Then v is in a SCC S, such that S is a sink of G^{SCC}.

Holds even after we delete the vertices of S (i.e., the vertex with the maximum post numbering, is in a sink of the meta graph).

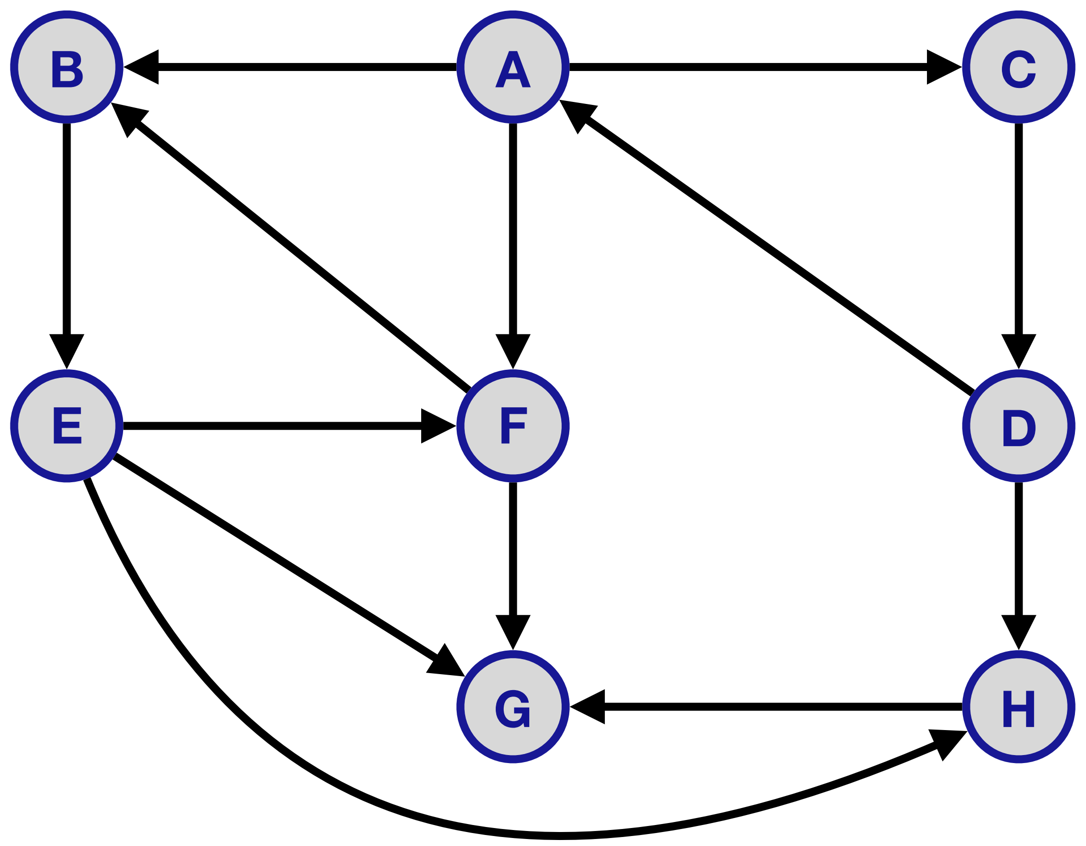
Linear Time SCC Algorithm

```
do DFS(G^rev) and output vertices in decreasing postvisit order.
Mark all nodes as unvisited.
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

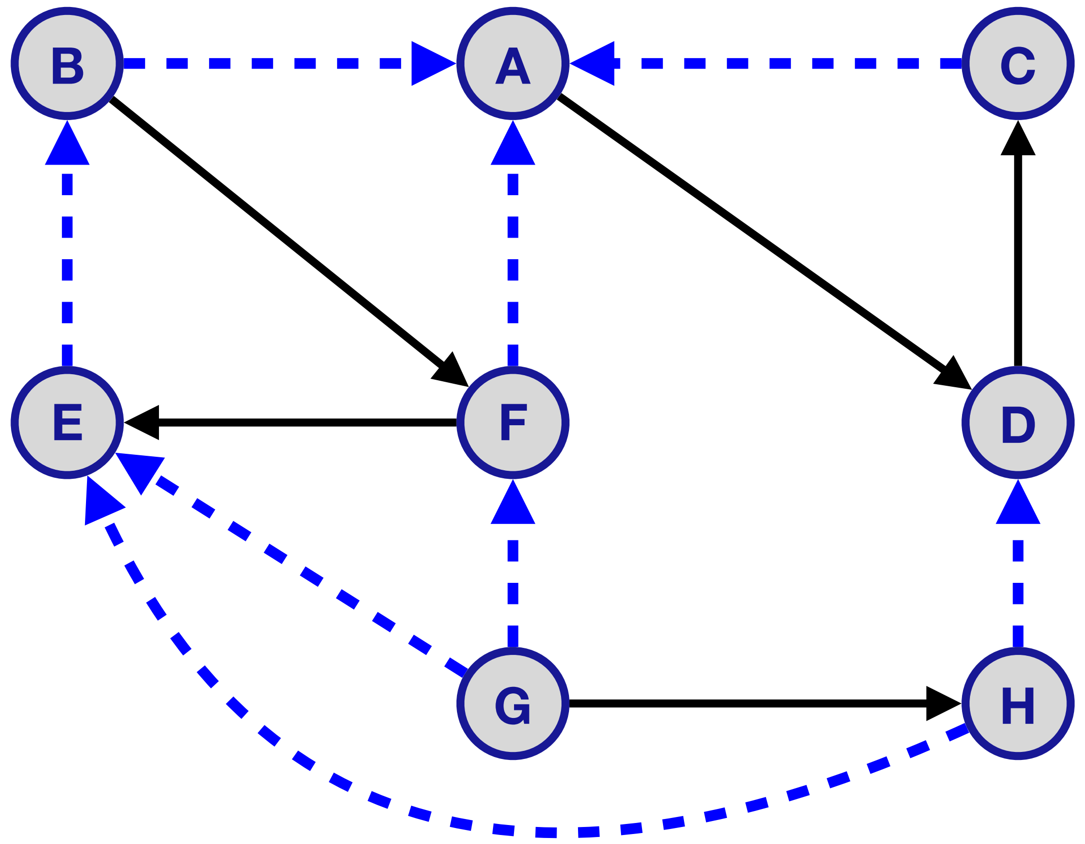
Theorem: Algorithm runs in time $O(m + n)$ and correctly outputs all the SCCs of G.

Linear Time Algorithm - An Example

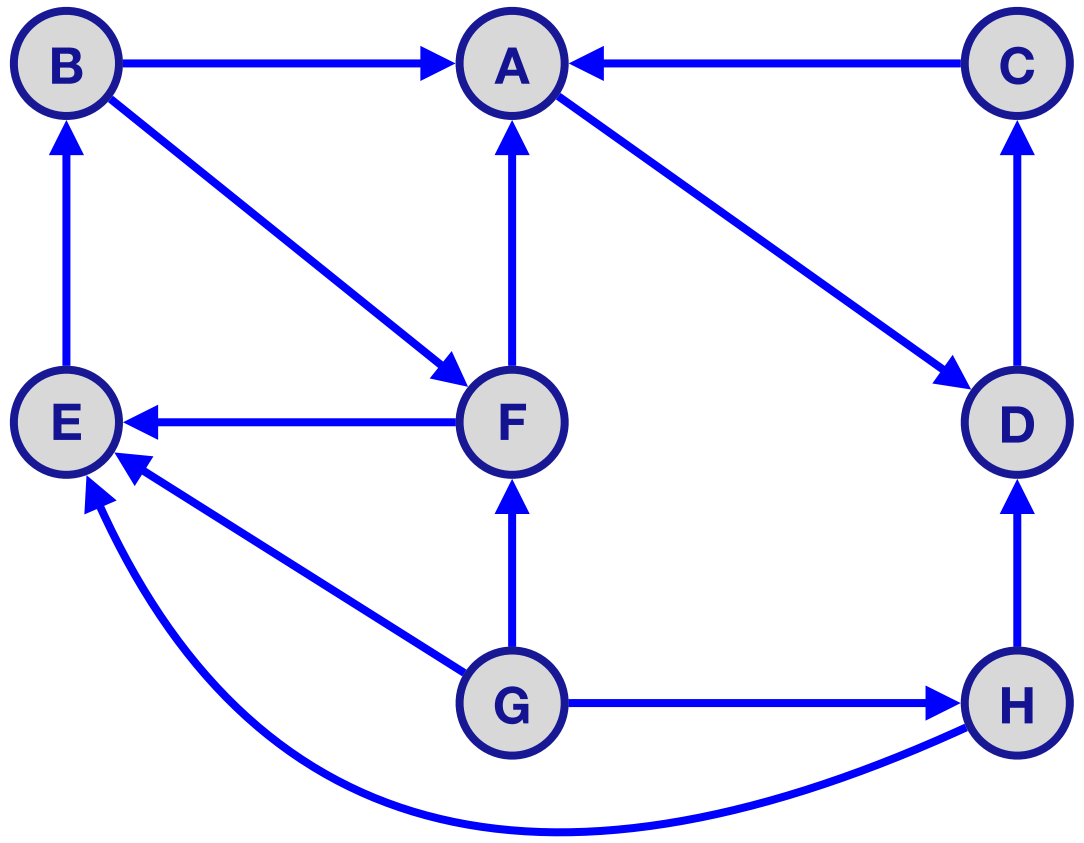
Graph G



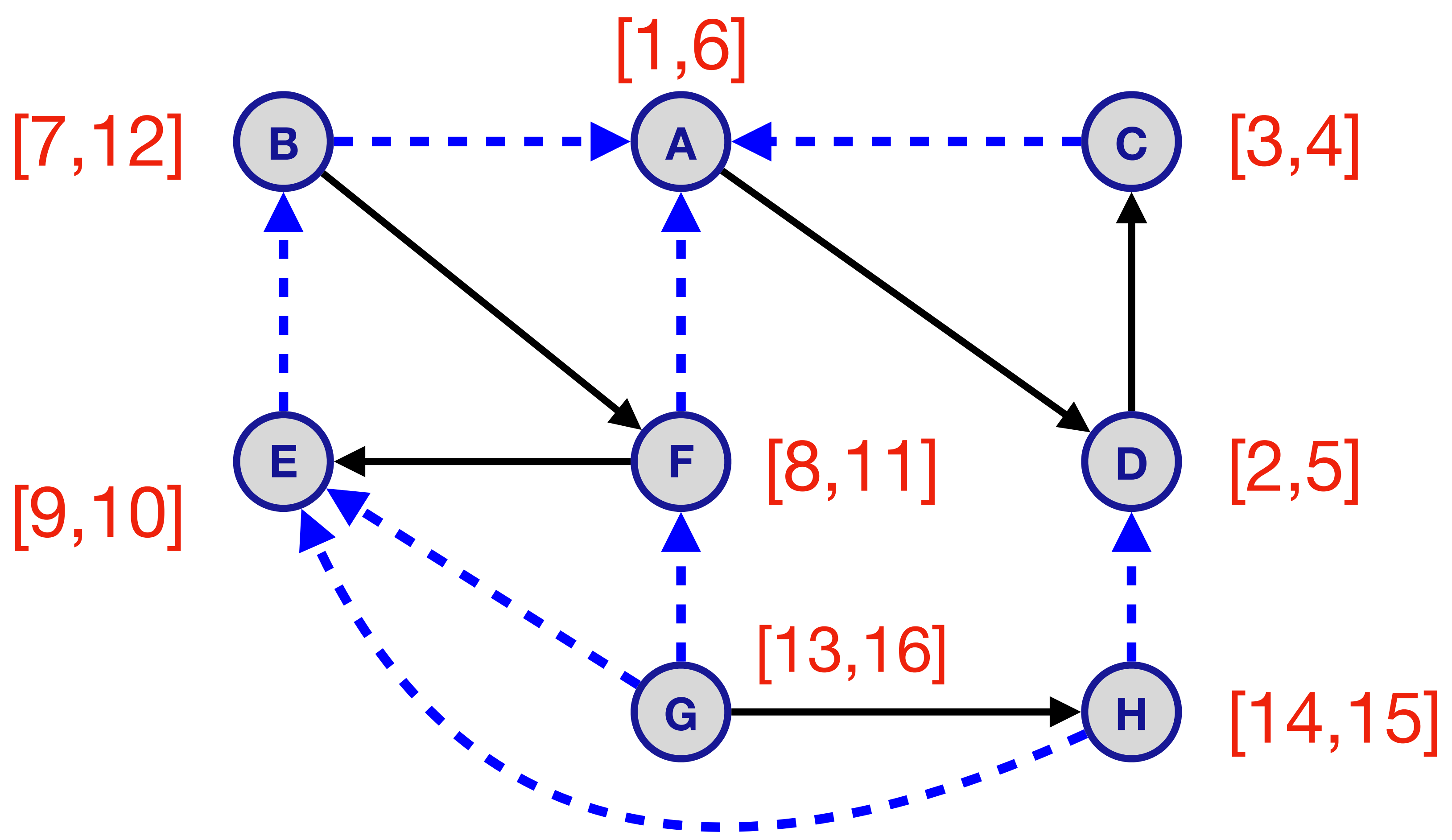
Reverse Graph G^{rev}



DFS of reverse graph

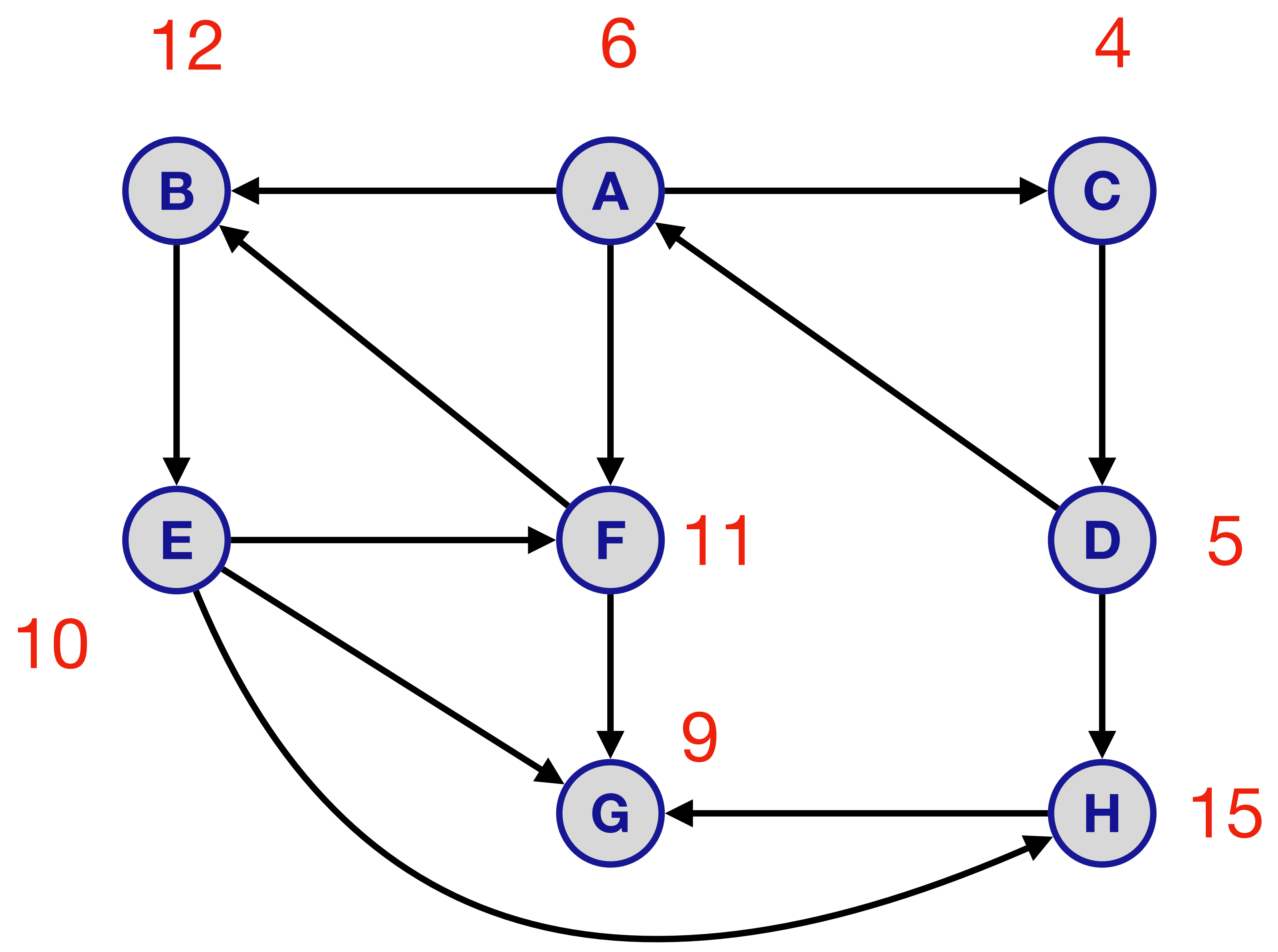


Pre/Post **DFS** numbering
of reverse graph



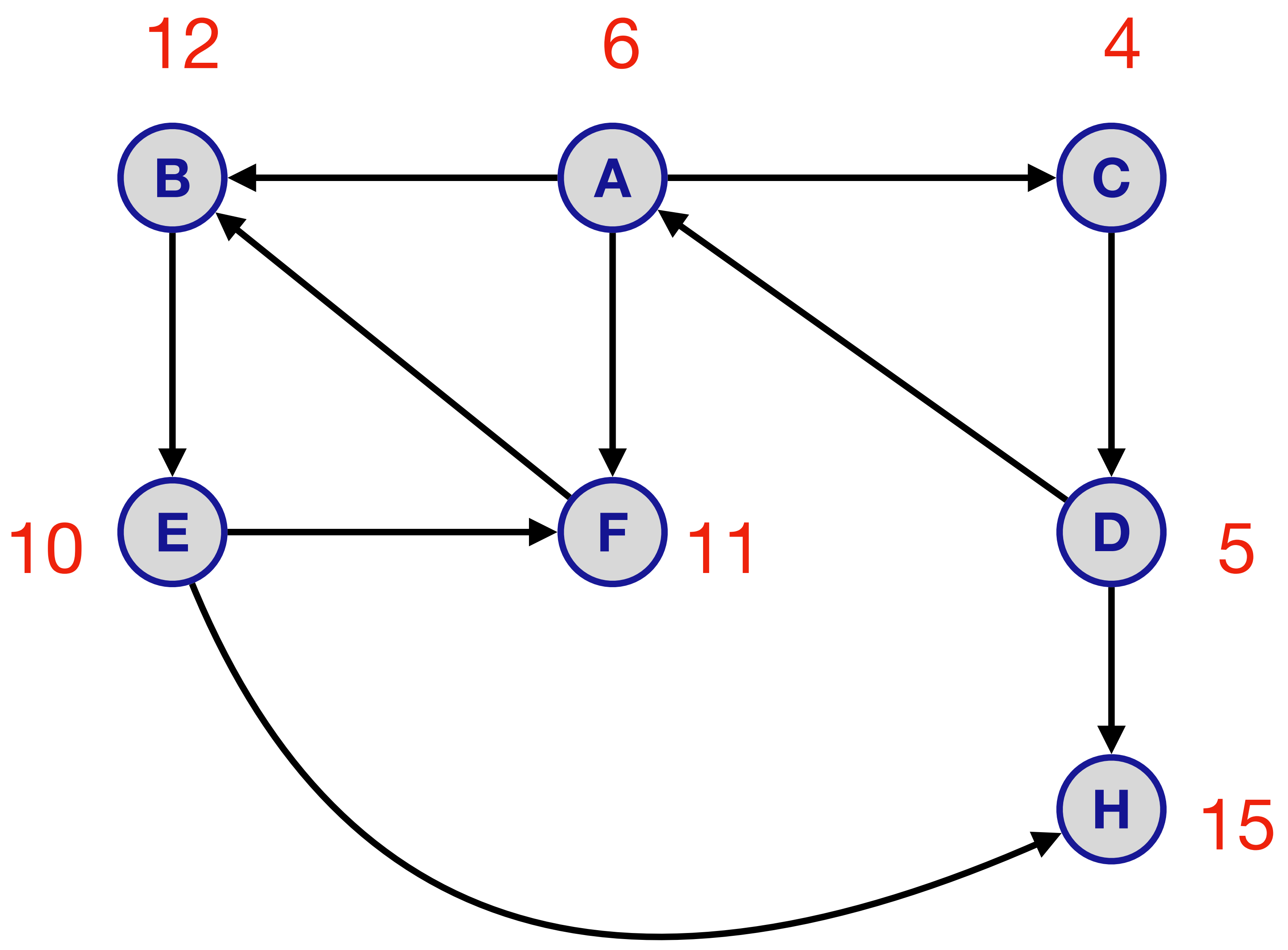
Linear Time Algorithm - An Example

Original graph G with rev post numbers



⟹

Do **DFS** from vertex G, remove it

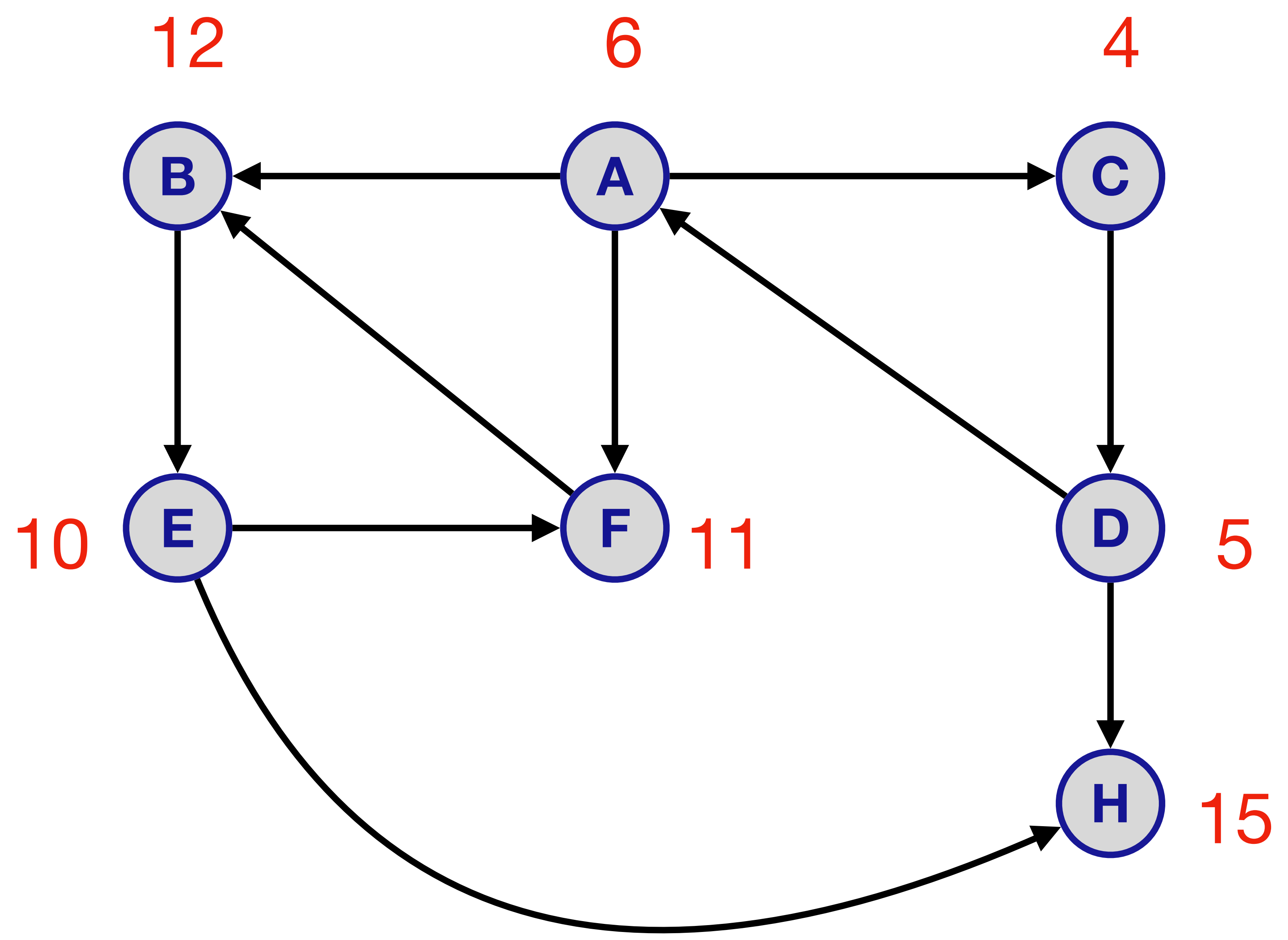


SCC computed:

{G}

Linear Time Algorithm - An Example

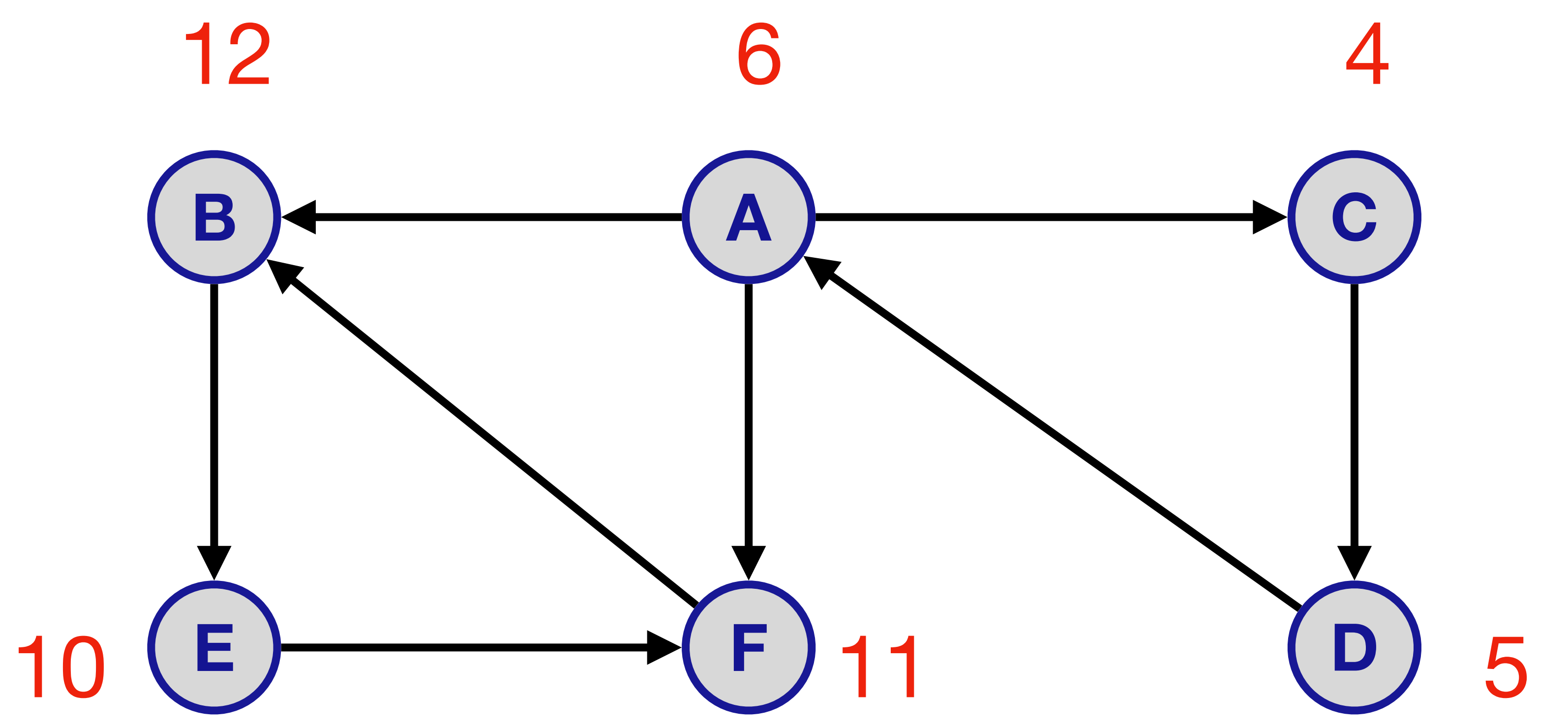
Do **DFS** from vertex G remove it



SCC computed:

{G}

Do **DFS** from vertex H, remove it

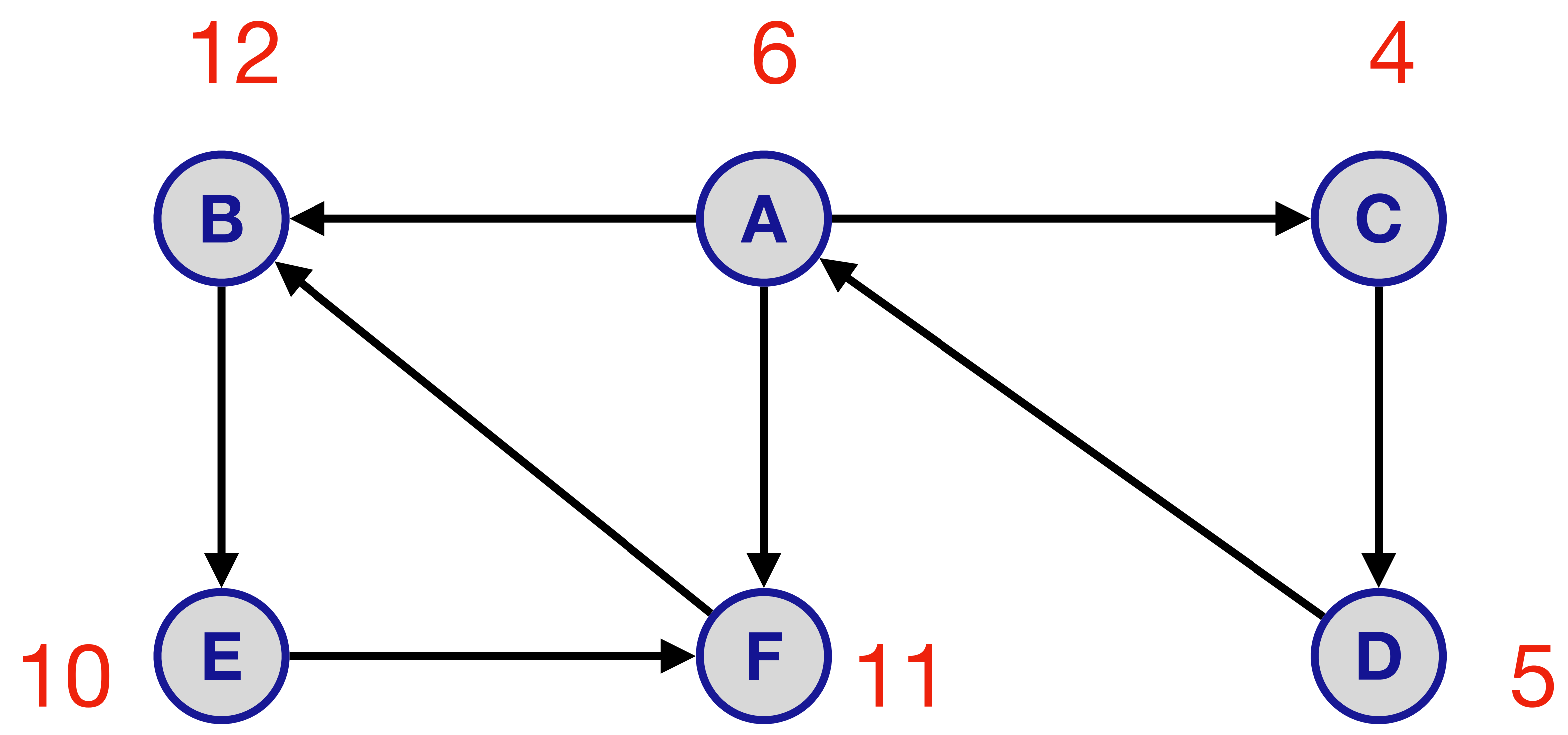


SCC computed:

{G}, {H}

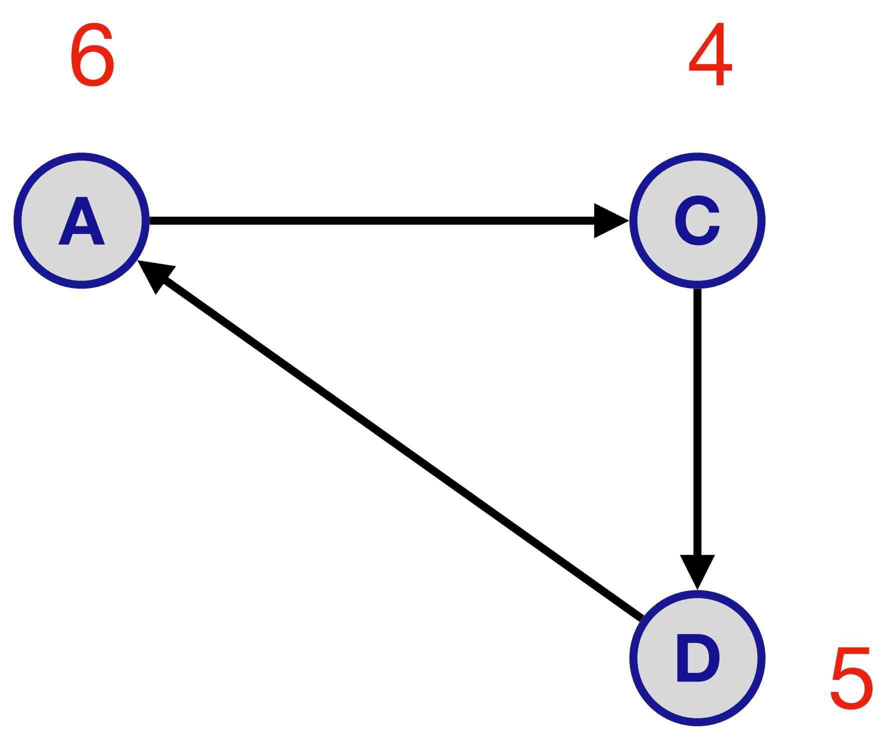
Linear Time Algorithm - An Example

Do **DFS** from vertex H, remove it



⟹

Do **DFS** from vertex B,
remove visited vertices: {F, B, E}.



SCC computed:

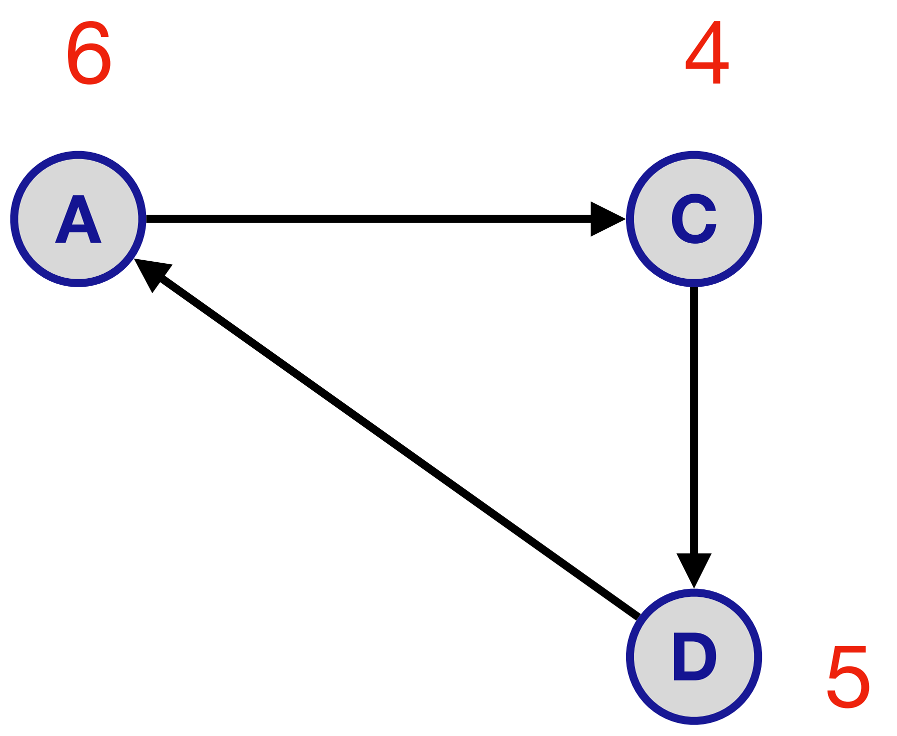
{G}, {H}

SCC computed:

{G}, {H}, {F, B, E}

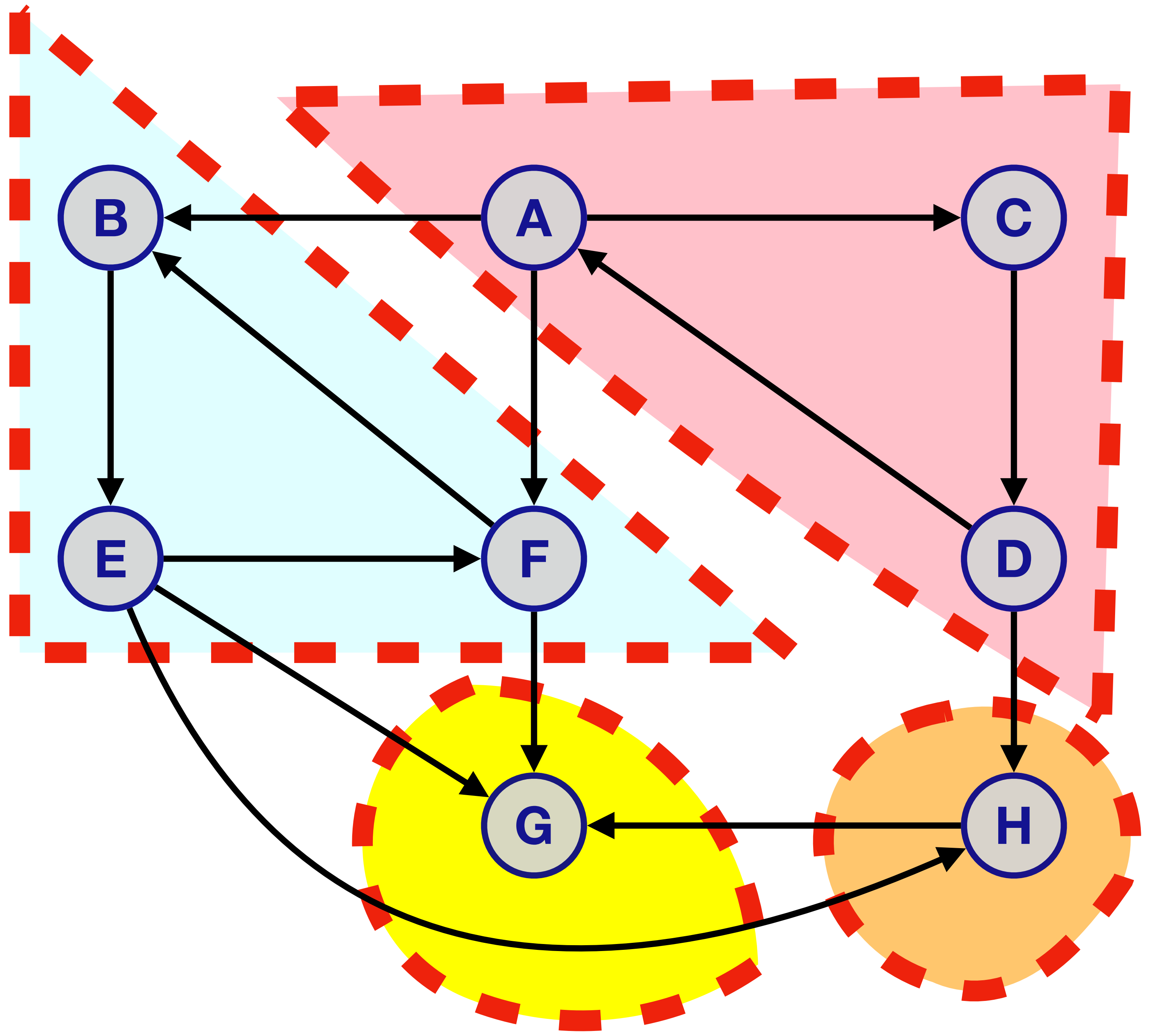
Linear Time Algorithm - An Example

Do **DFS** from vertex B,
remove visited vertices: {F, B, E}.



$\Longrightarrow$

Do **DFS** from vertex A,
remove visited vertices: {A, C, D}.



SCC computed:

{G}, {H}, {F, B, E}

SCC computed:

{G}, {H}, {F, B, E}, {A,C,D}

Summary

Take away points

- DAGs and topological orderings.
- **DFS** with pre/post numbering.
- Given a directed graph G, its SCCs and the associated acyclic meta-graph G^{SCC} give a structural decomposition of G.
- There is a DFS based linear time algorithm to compute all the SCCs and the meta-graph.
- DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms!