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)

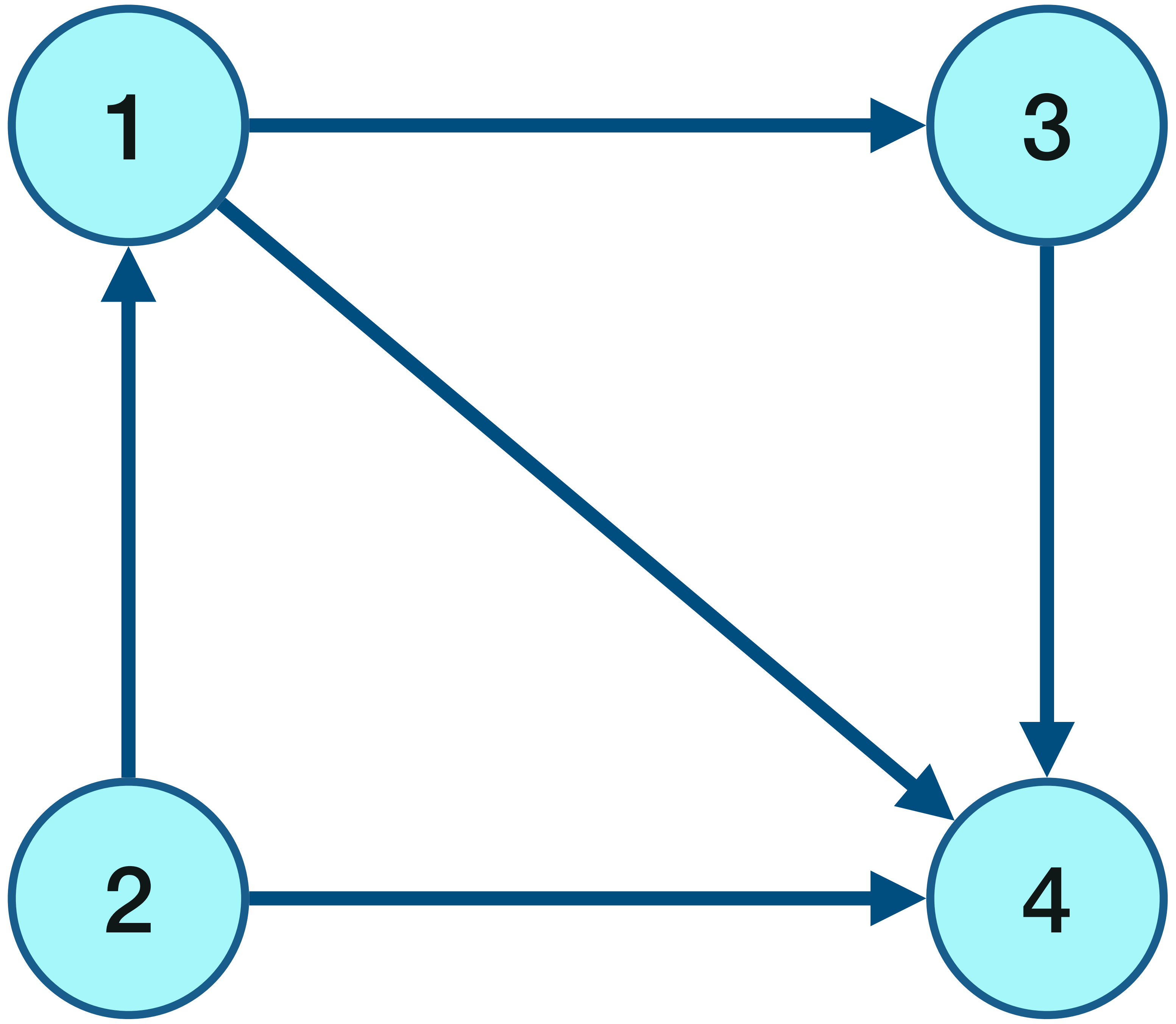
Directed acyclic graphs

Definition

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

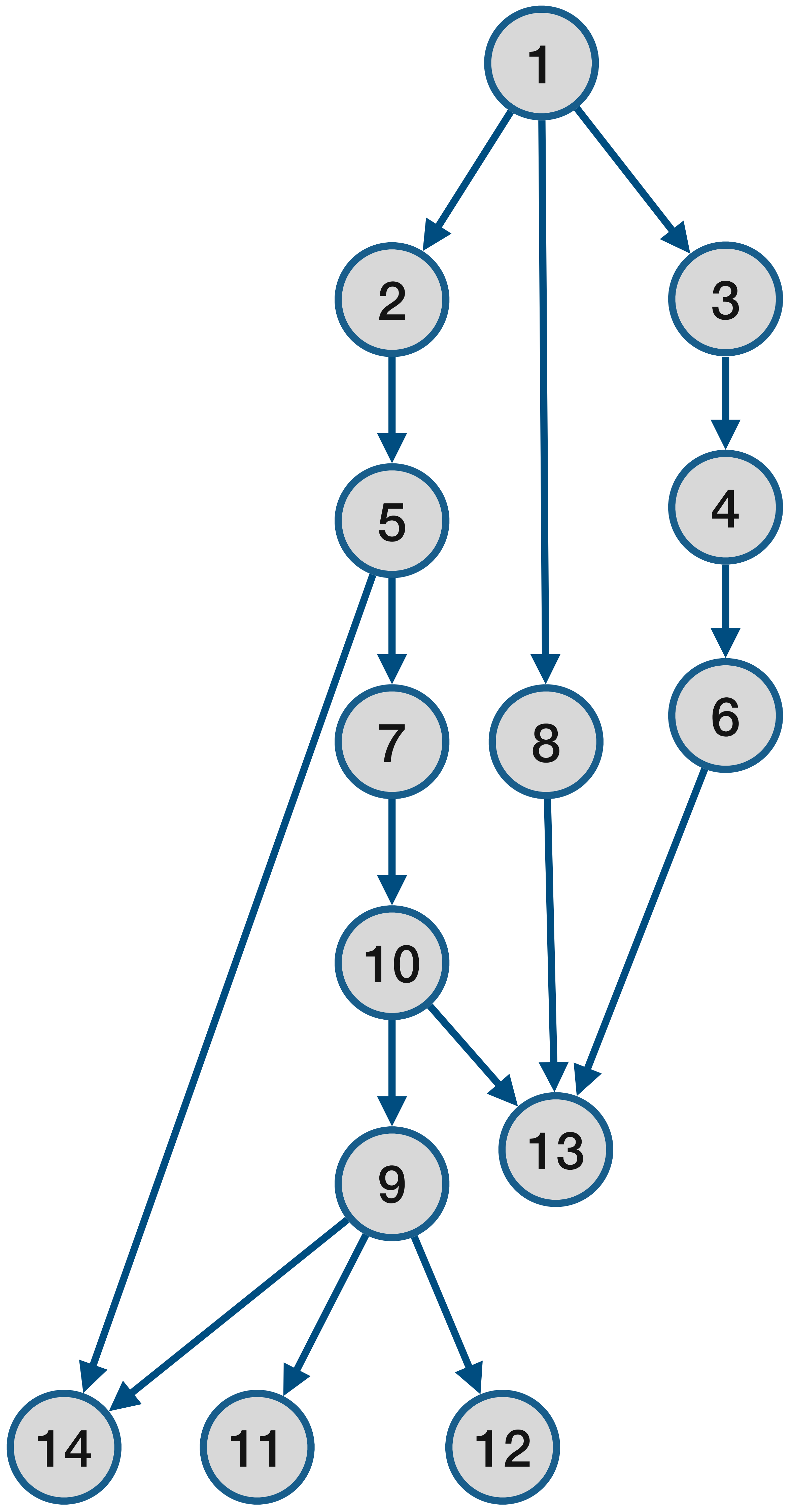
Tells us that G is directed.

No such thing as an "undirected cycle", or a directed G ... yet anyway.



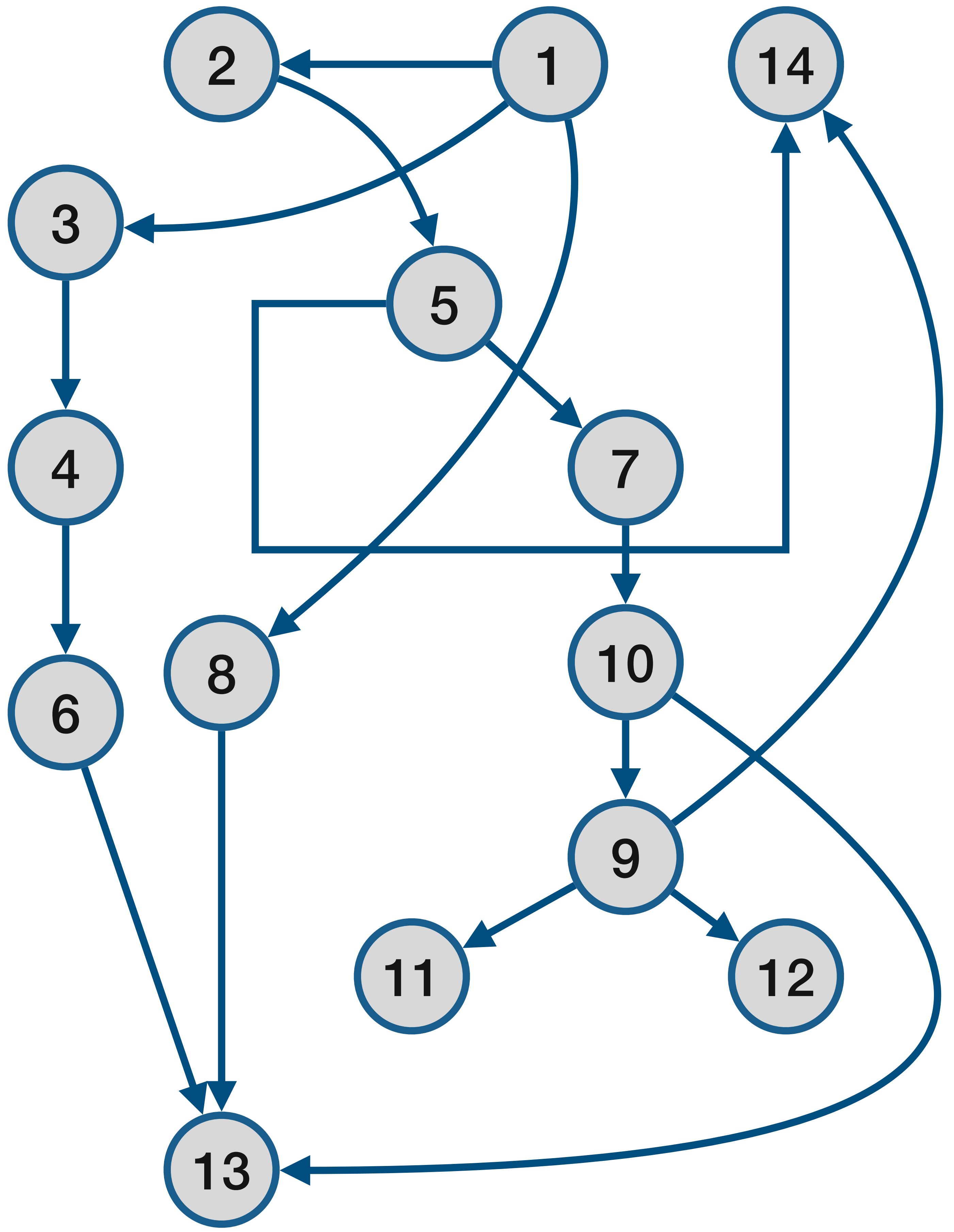
Directed acyclic graphs

Is this a DAG?

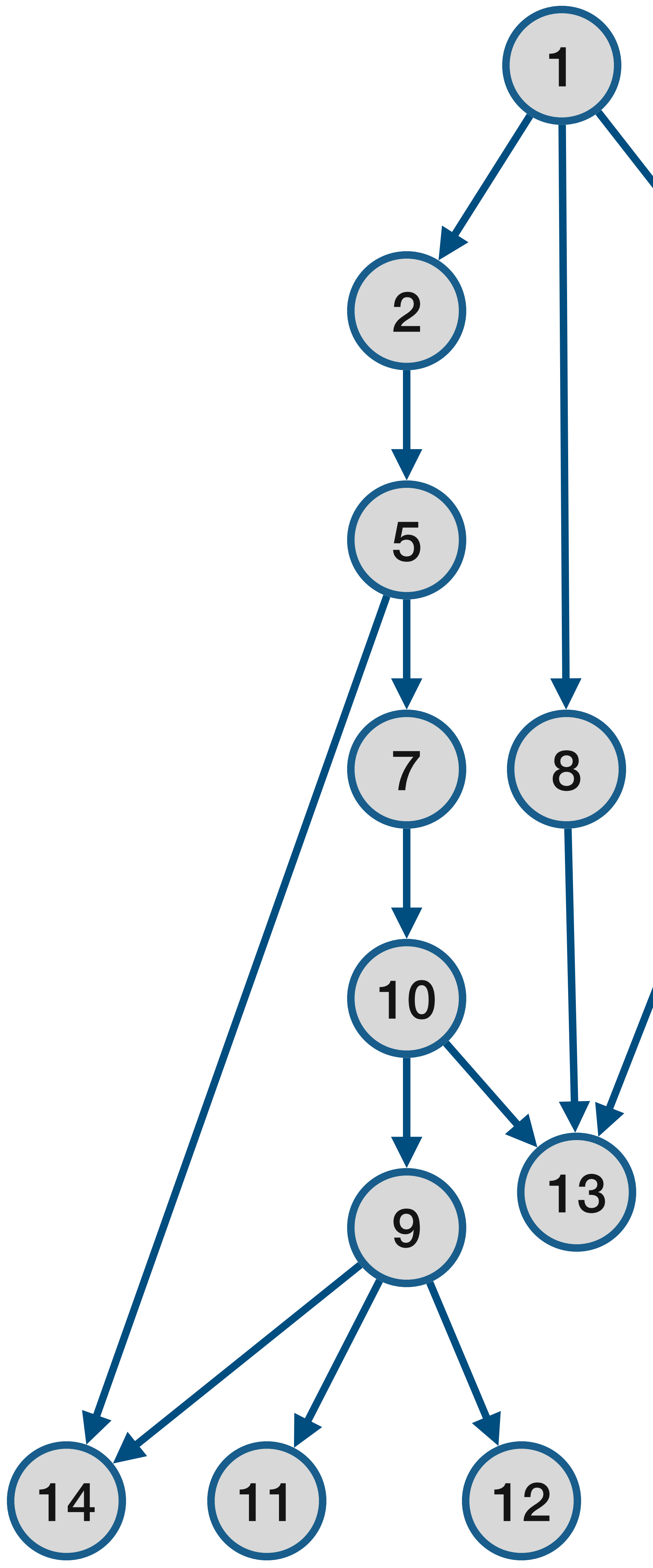


Directed acyclic graphs

Is this a DAG?



Hmm...

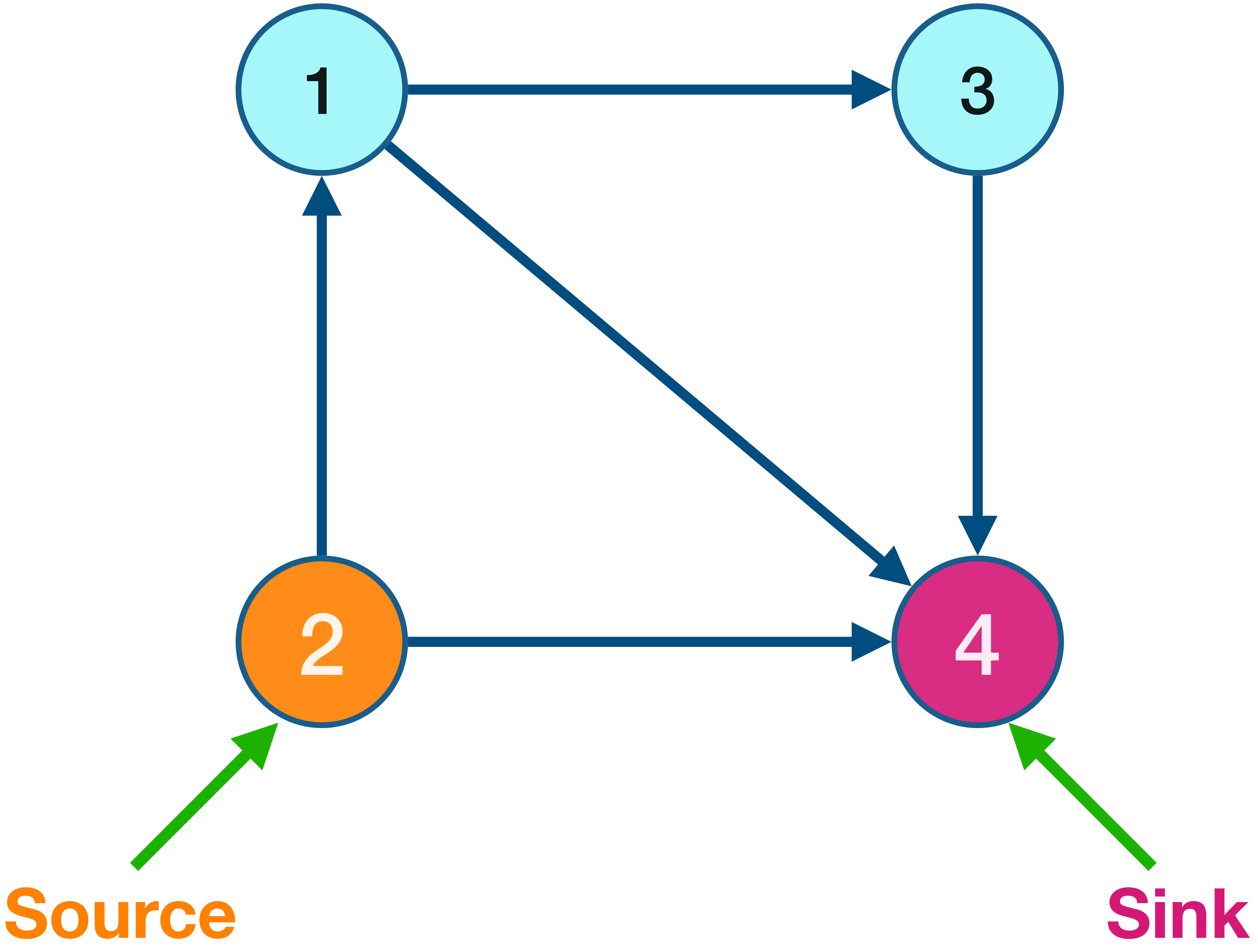


easy peasy
lemon is
squeezy

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.

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.

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.

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

\prec.

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

- $\prec$ is transitive. $a \prec b \prec c \Rightarrow a \prec c$

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:

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:

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

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:

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

↳ Different sources use them.... but
I will use them <u>all</u> freely.

Topological ordering/sorting

Definition

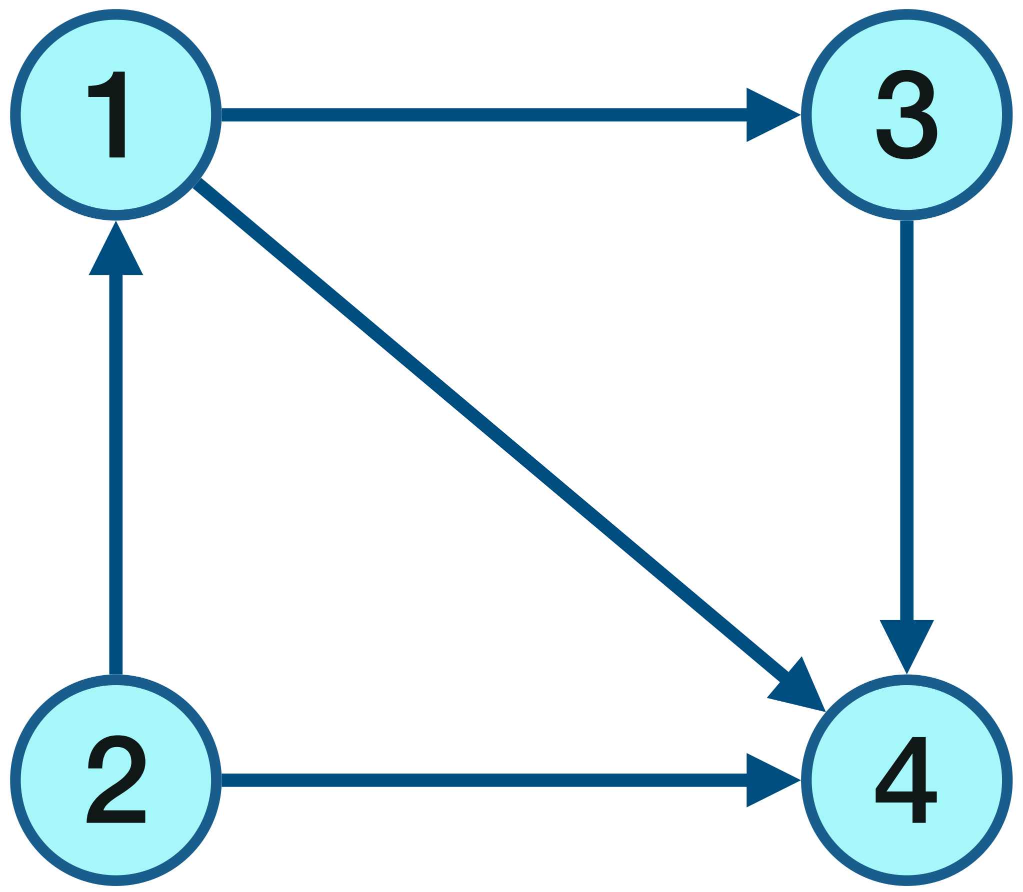
English: if edges goes from u to v then u is "smaller" than v.

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

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 $u \prec v$.

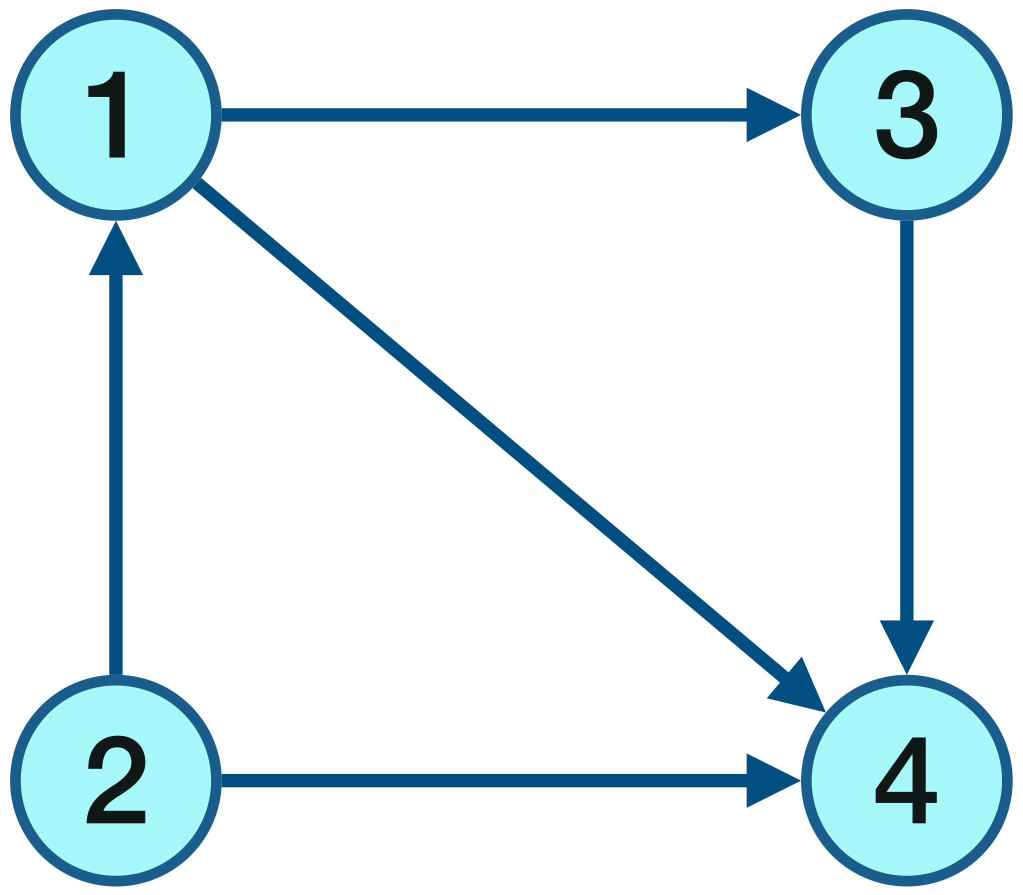


Graph G

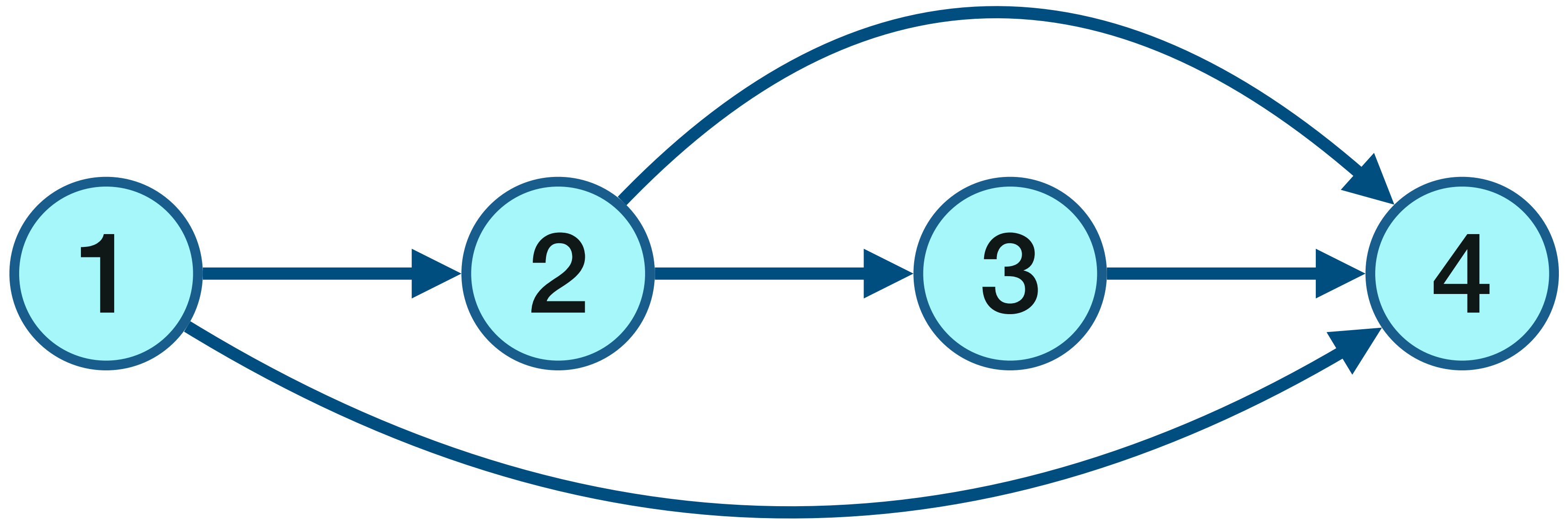
Topological ordering/sorting

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

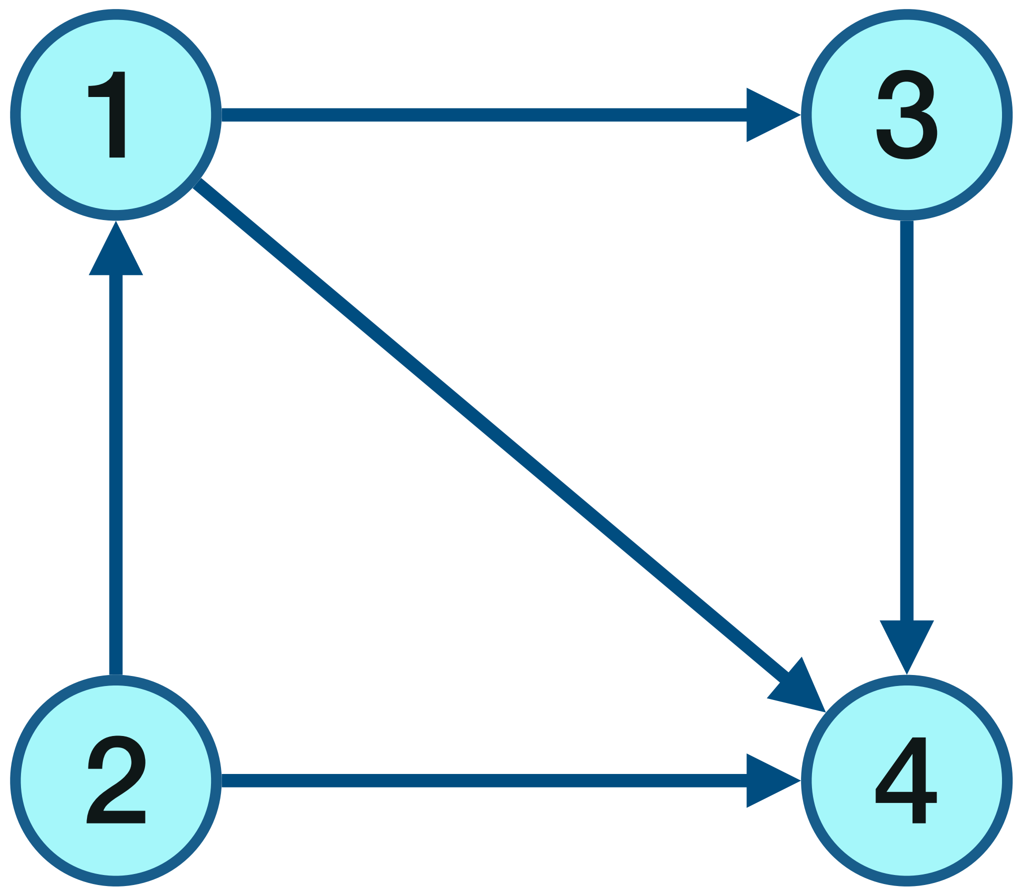


Topological Ordering of G

Topological ordering/sorting

Definition

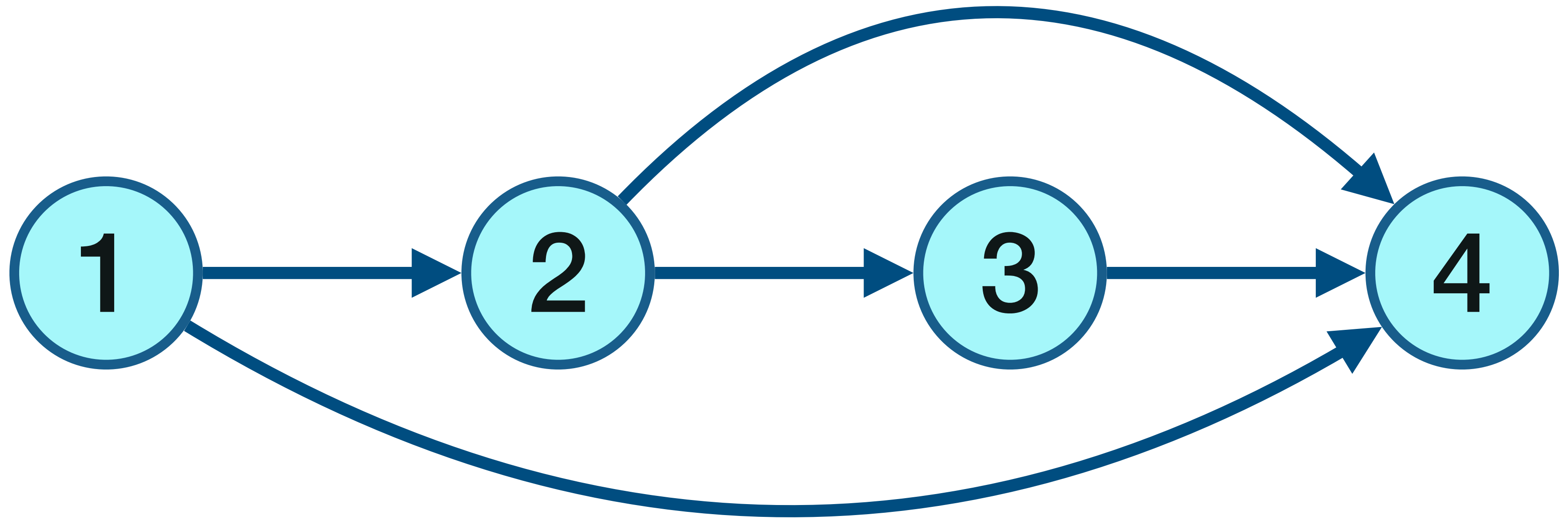
A ***topological ordering / topological sorting*** of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u \to v) \in E$ then $u \prec v$.



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

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., $\deg_{In}(v) = 0$:

Topological ordering in linear time

Exercise

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

Simple algorithm: number of edges coming in

- 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

Topological ordering in linear time

Exercise

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

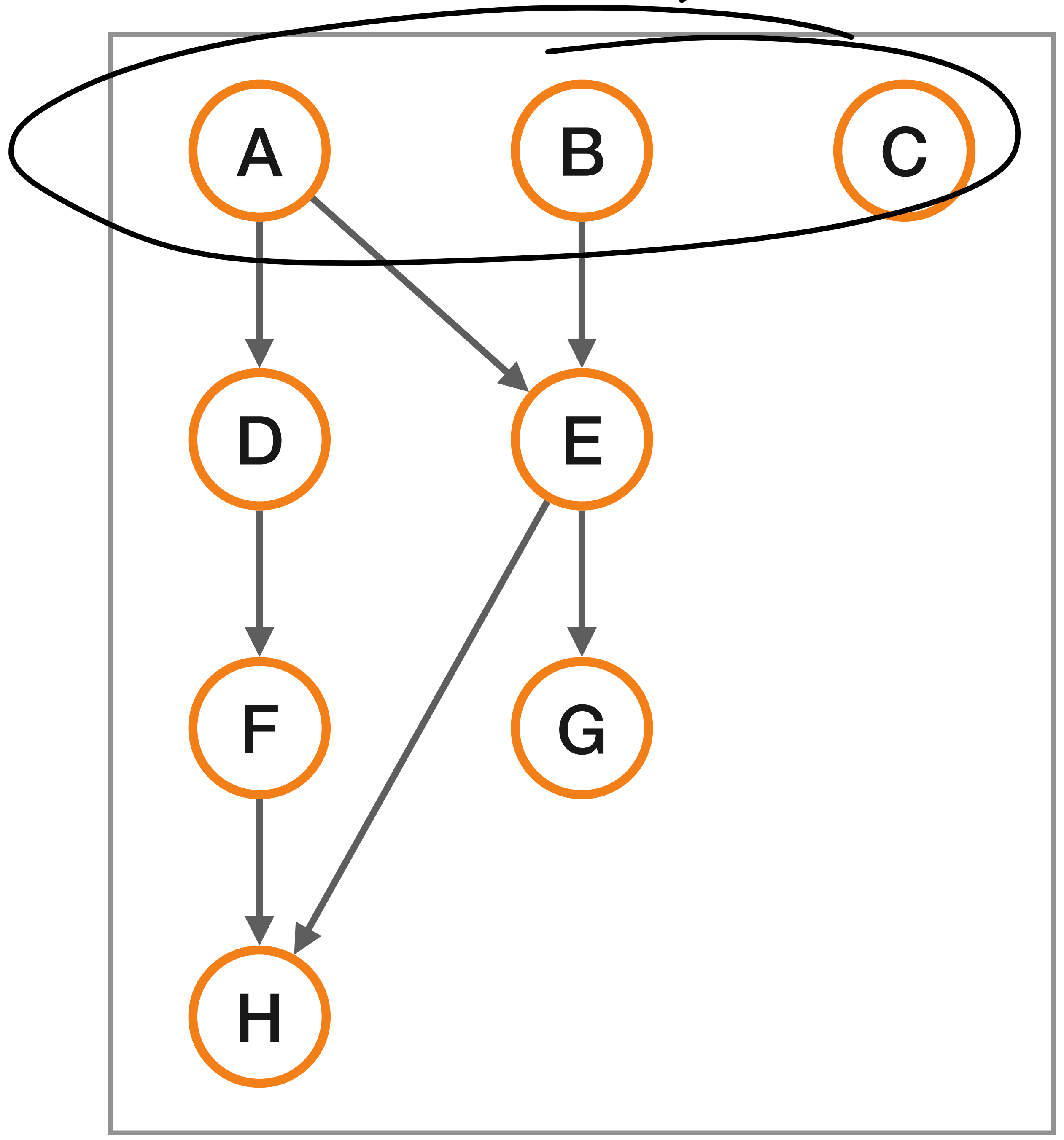
Simple algorithm:

- 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

no incoming edges



Adjacency List:

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

Initialize

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

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

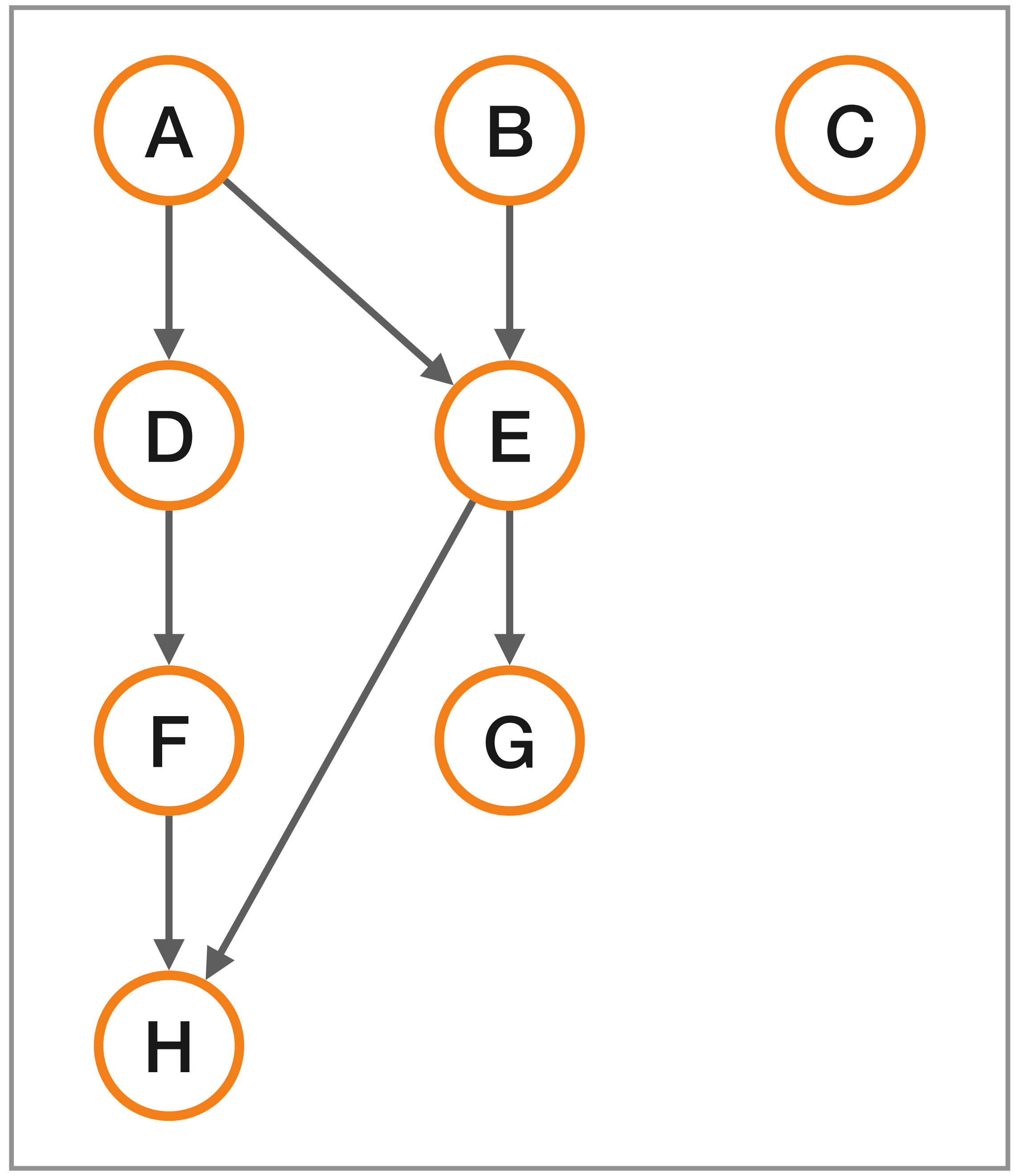
For each vertex that is source ($deg_{in}(v) = 0$):

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

Topological Ordering:

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	$\boxed{\text{A B C}}$
1	D F G
2	E H

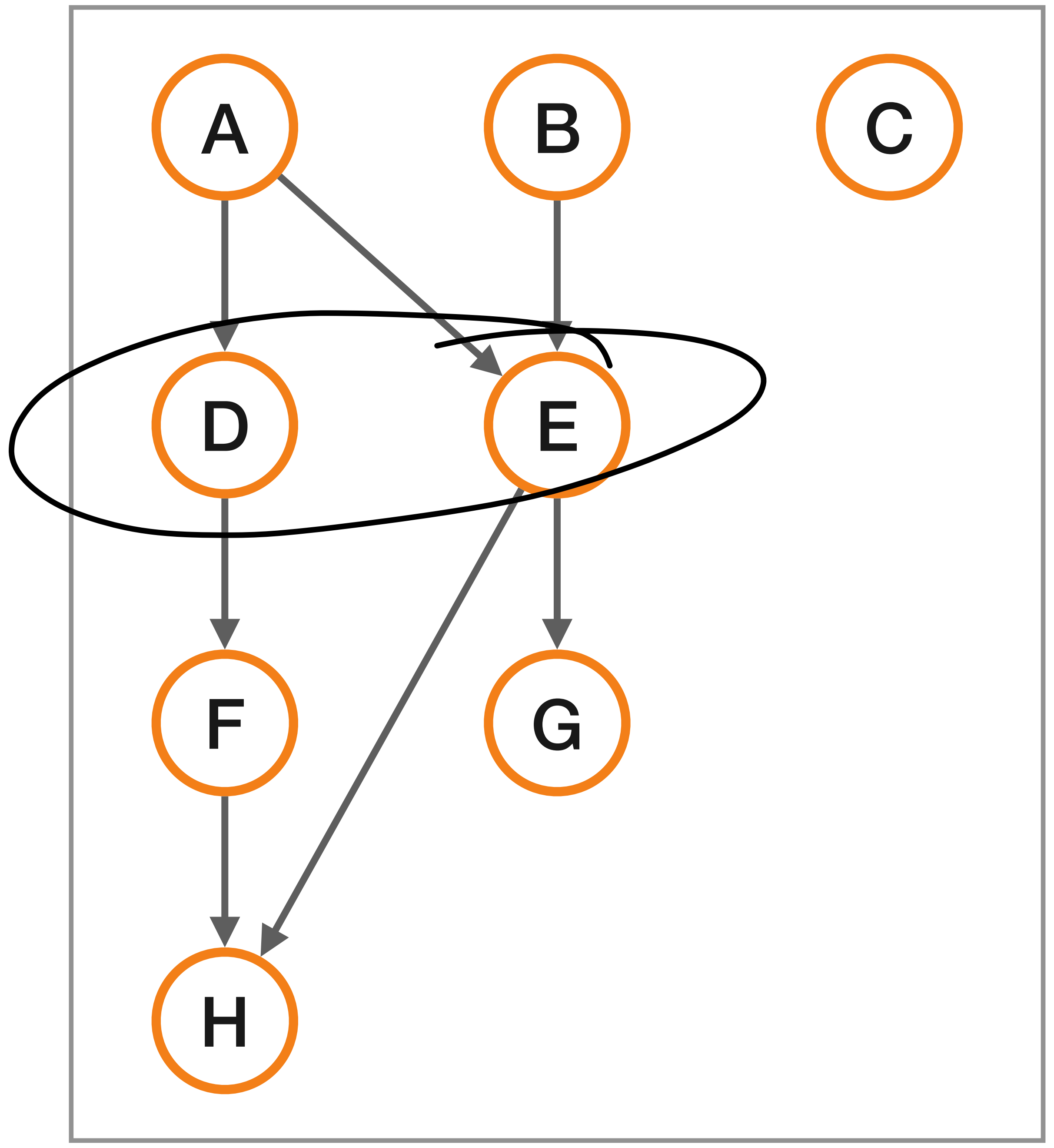
For each vertex that is source ($deg_{in}(v) = 0$):

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

Topological Ordering:

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	$\boxed{A}$ B C
1	D F G
2	E H

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

- Add v to the topological sort

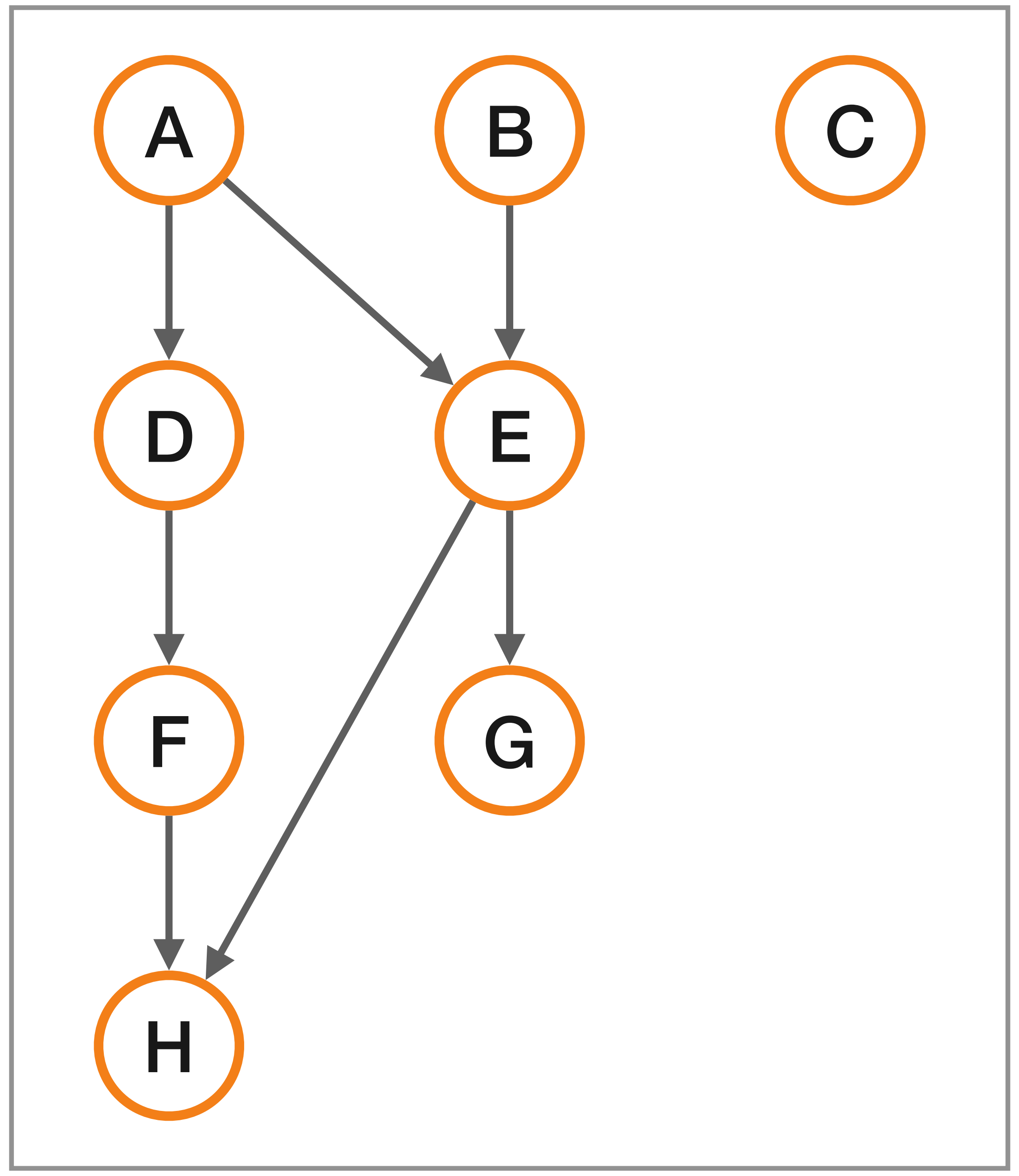
- Lower degree of vertices v is connected to.

Topological Ordering:

A

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	$\boxed{A}$ B C
1	D F G
2	E H

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

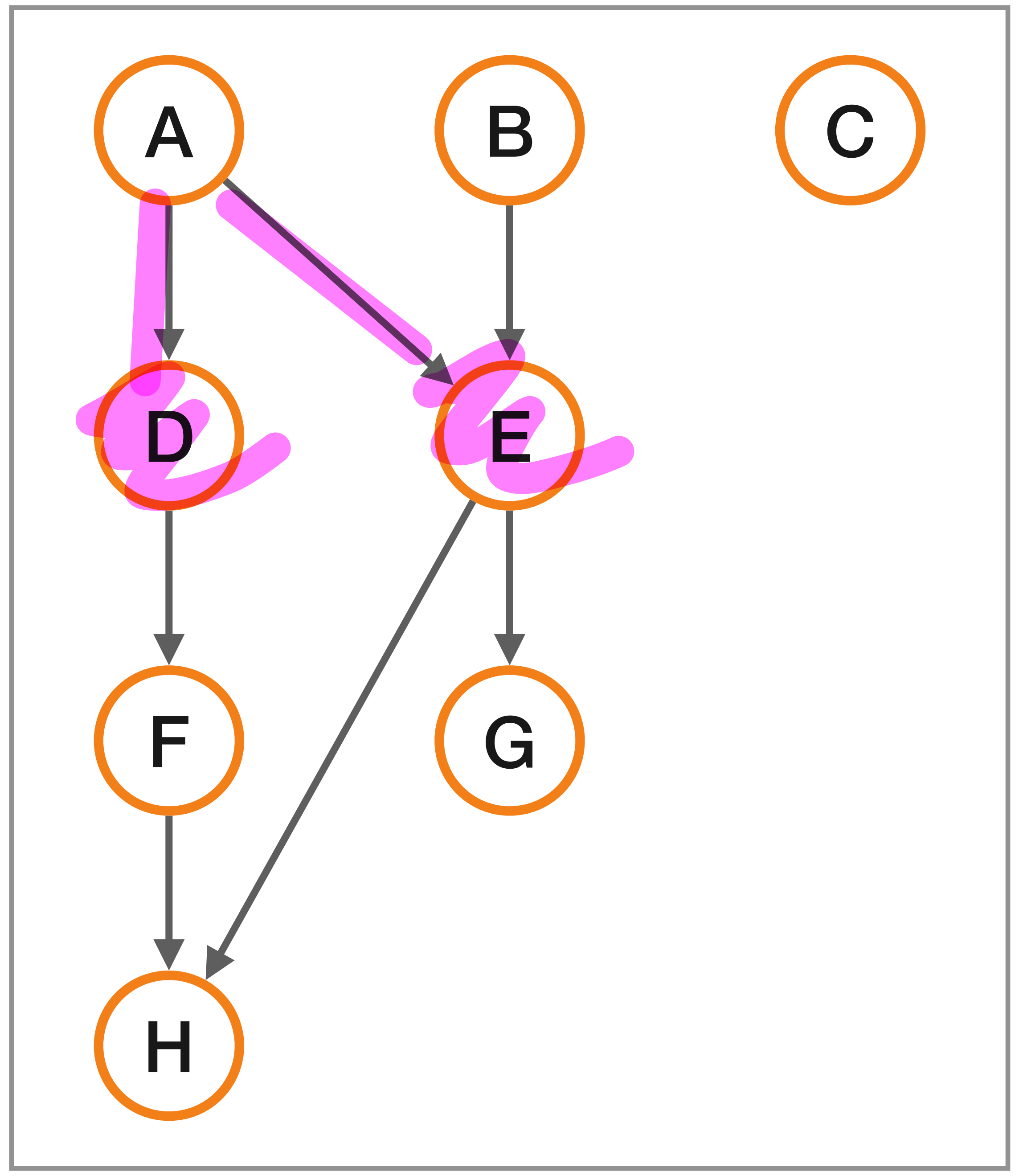
- Add v to the topological sort
- Lower degree of vertices v is connected to.

Topological Ordering:

A

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 D
1	F G E
2	H

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

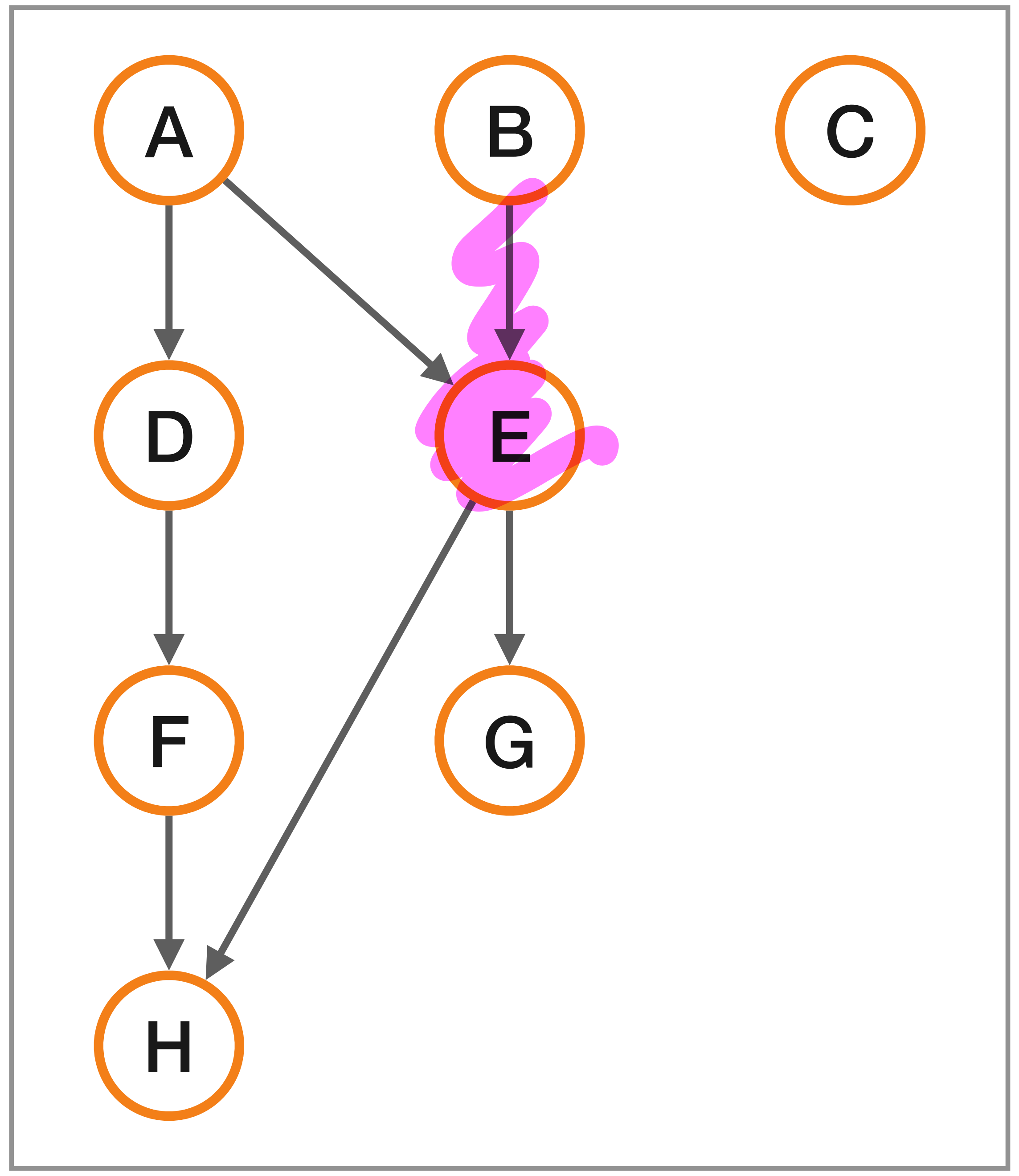
- Add v to the topological sort
- Lower degree of vertices v is connected to.

Topological Ordering:

A

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 $\boxed{\text{B}}$ C D
1	F G E
2	H

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

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

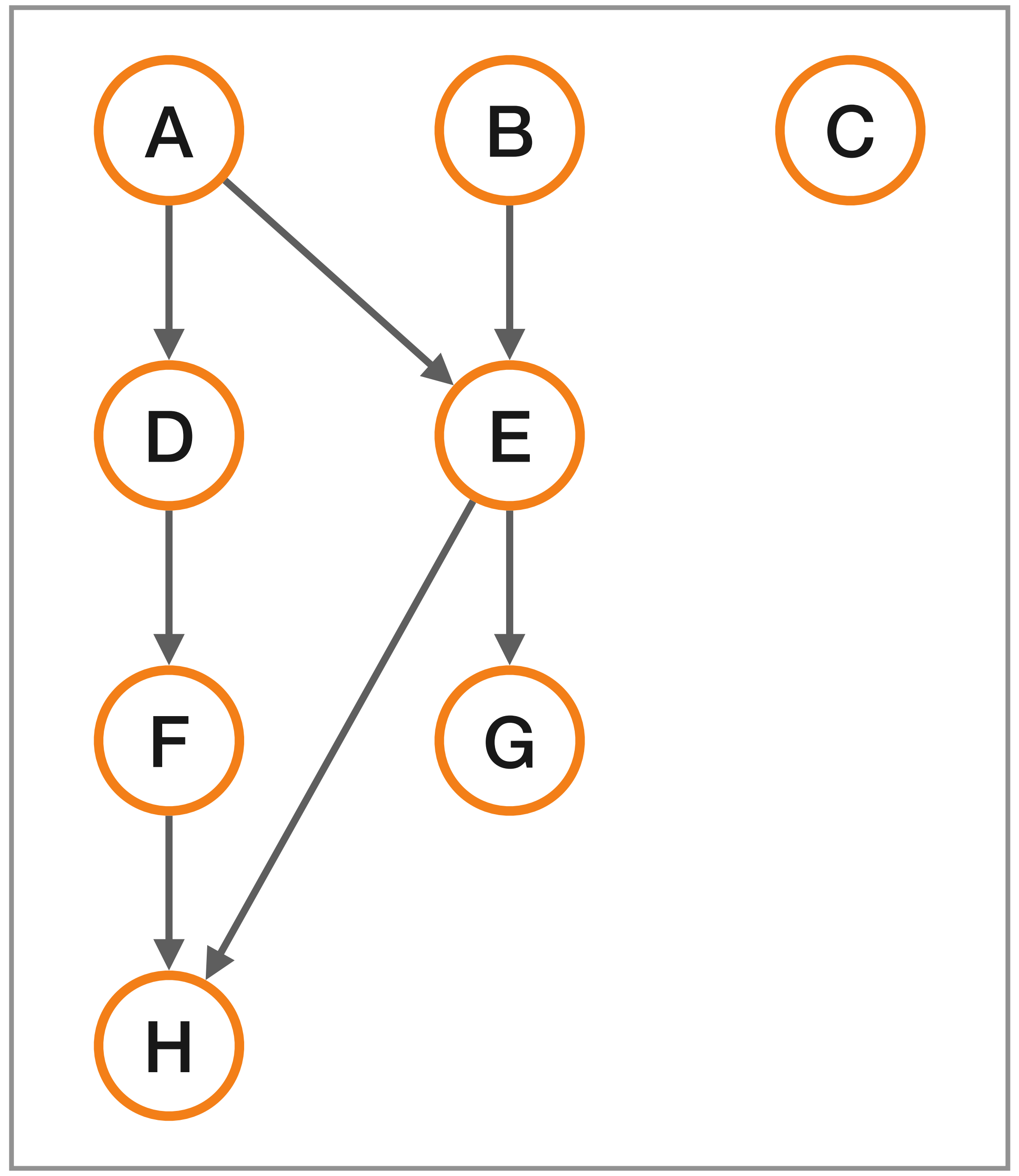
Repeat the steps again.

Topological Ordering:

A B

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 $\boxed{B}$ C D E
1	F G
2	H

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

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

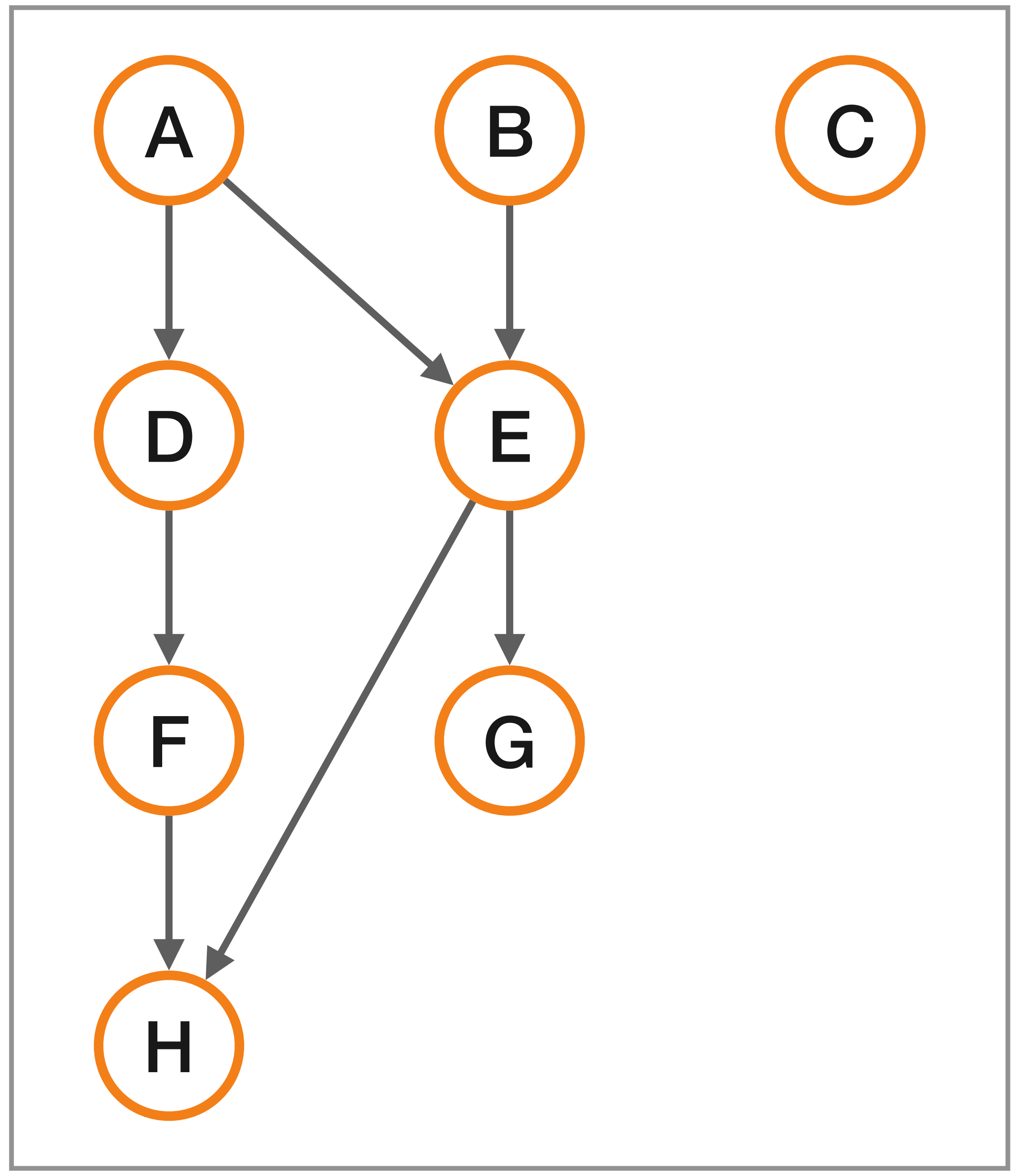
Repeat the steps again.

Topological Ordering:

A B

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 D E
1	F G
2	H

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

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

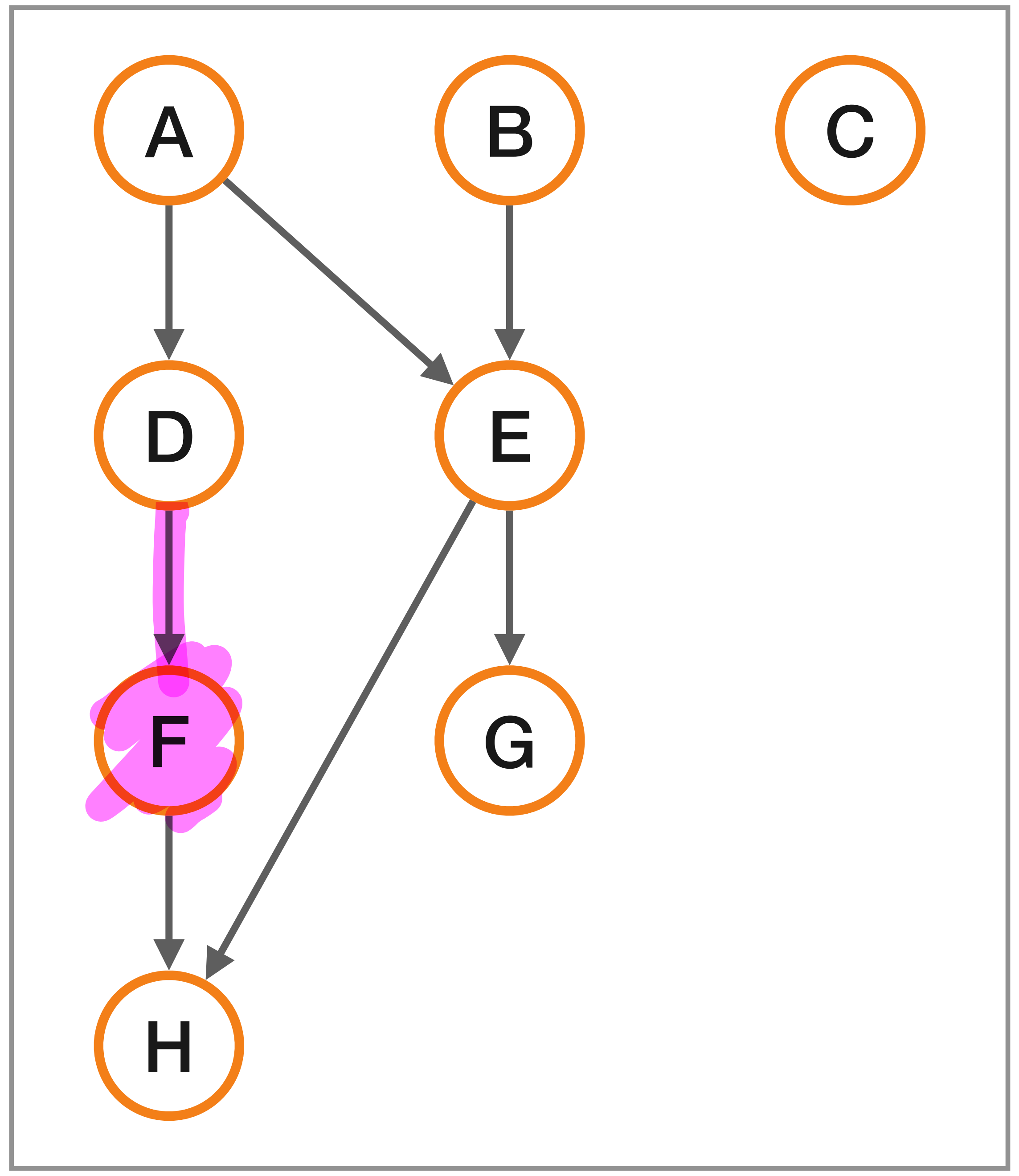
Repeat the steps again.

Topological Ordering:

A B C

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 $\boxed{D}$ E
1	F G
2	H

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

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

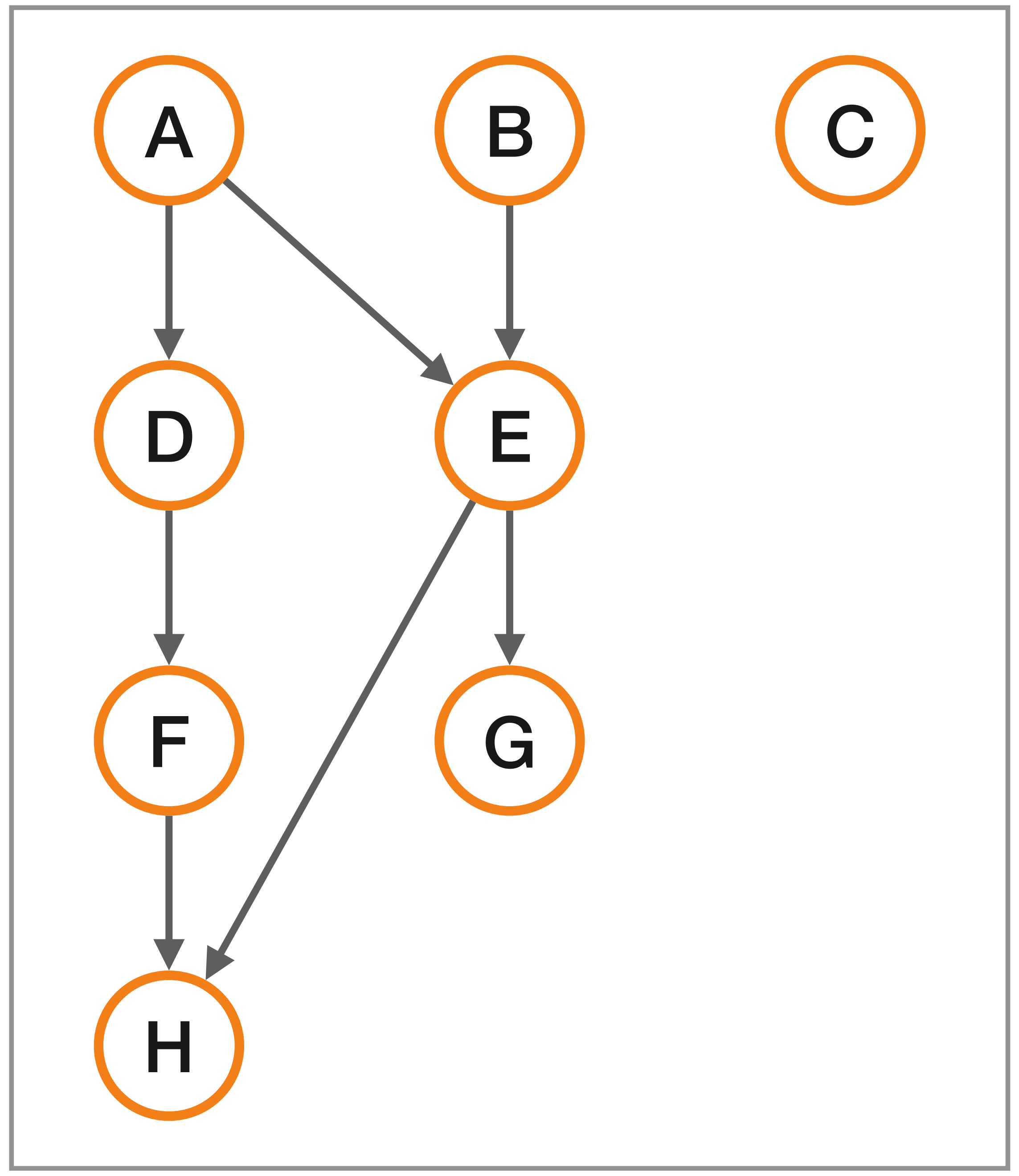
Repeat the steps again.

Topological Ordering:

A B C D

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 $\boxed{D}$ E F
1	G
2	H

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

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

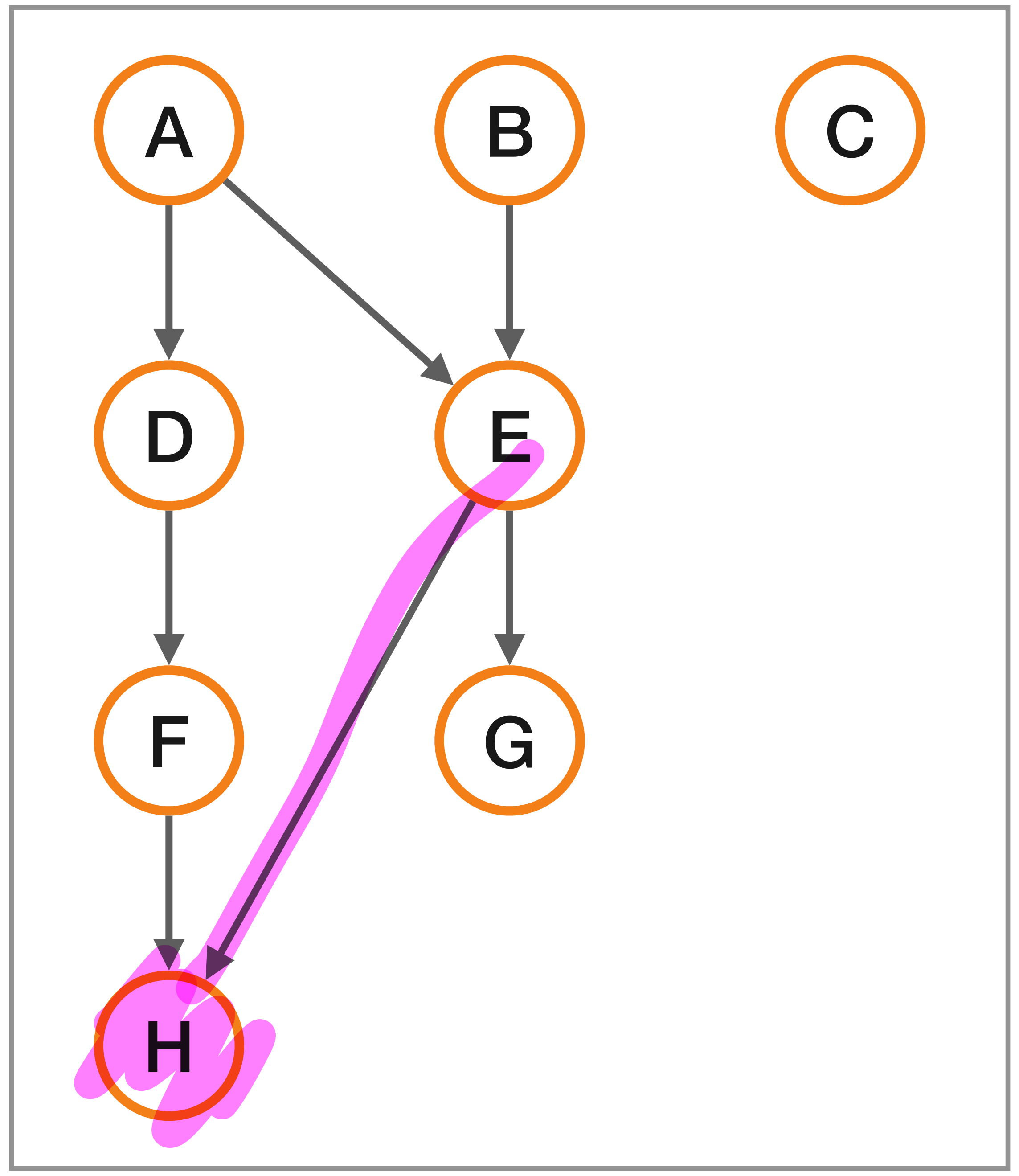
Repeat the steps again.

Topological Ordering:

A B C D

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 D $\boxed{E}$ F
1	G
2	H

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

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

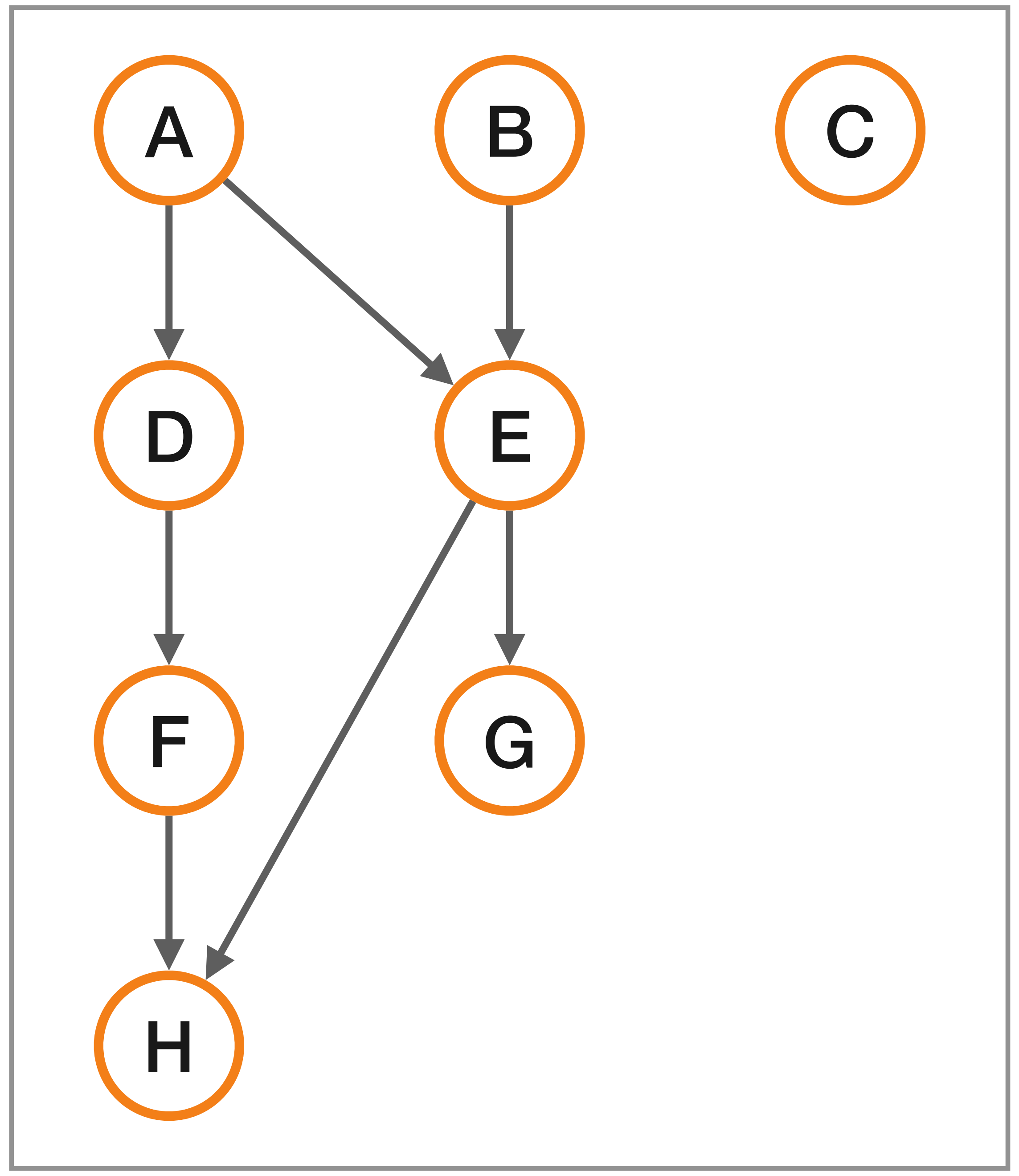
Repeat the steps again.

Topological Ordering:

A B C D E

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 D $\boxed{\text{E}}$ F G
1	H
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.

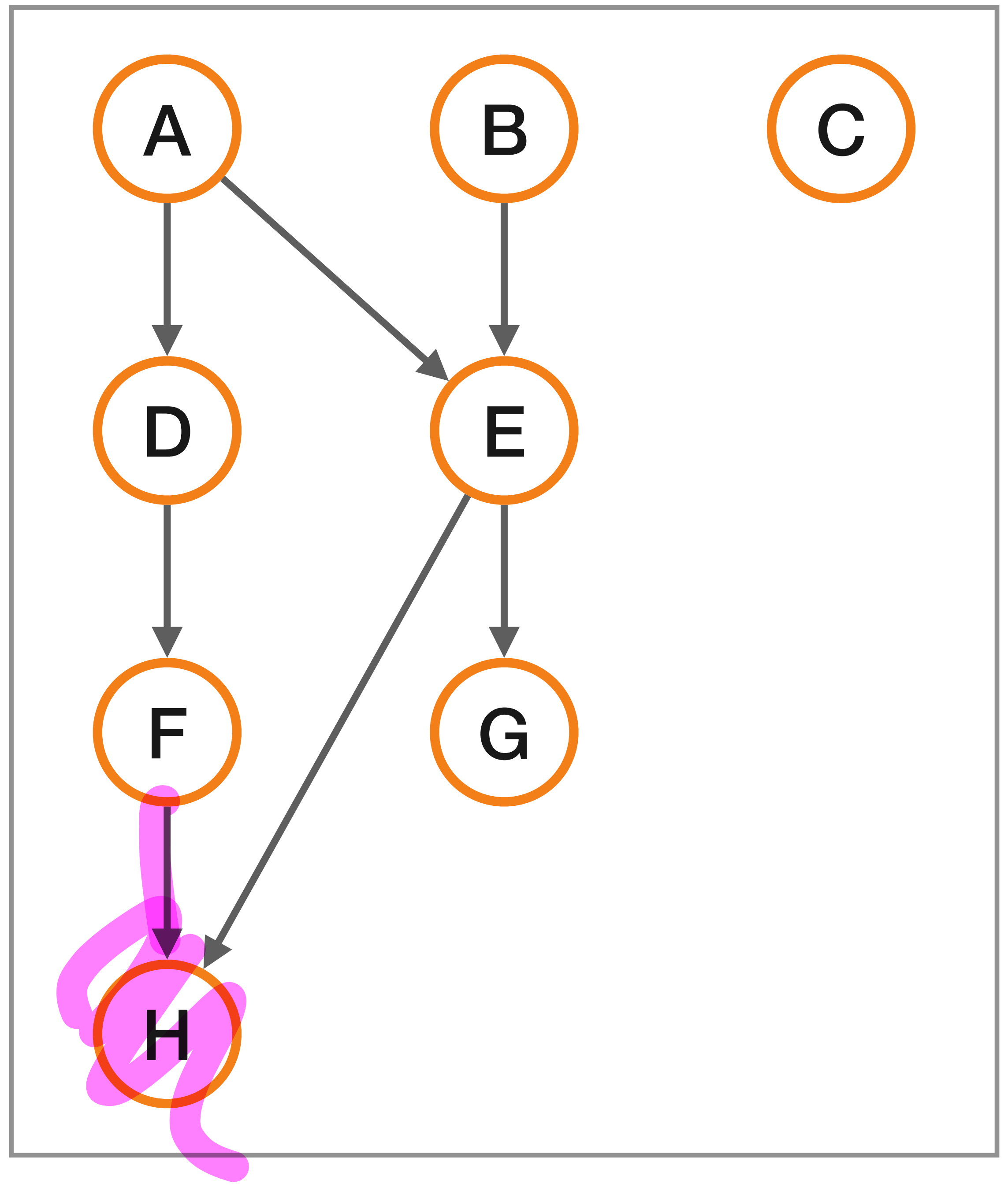
Repeat the steps again.

Topological Ordering:

A B C D E

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 D E $\boxed{F}$ G
1	H
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.

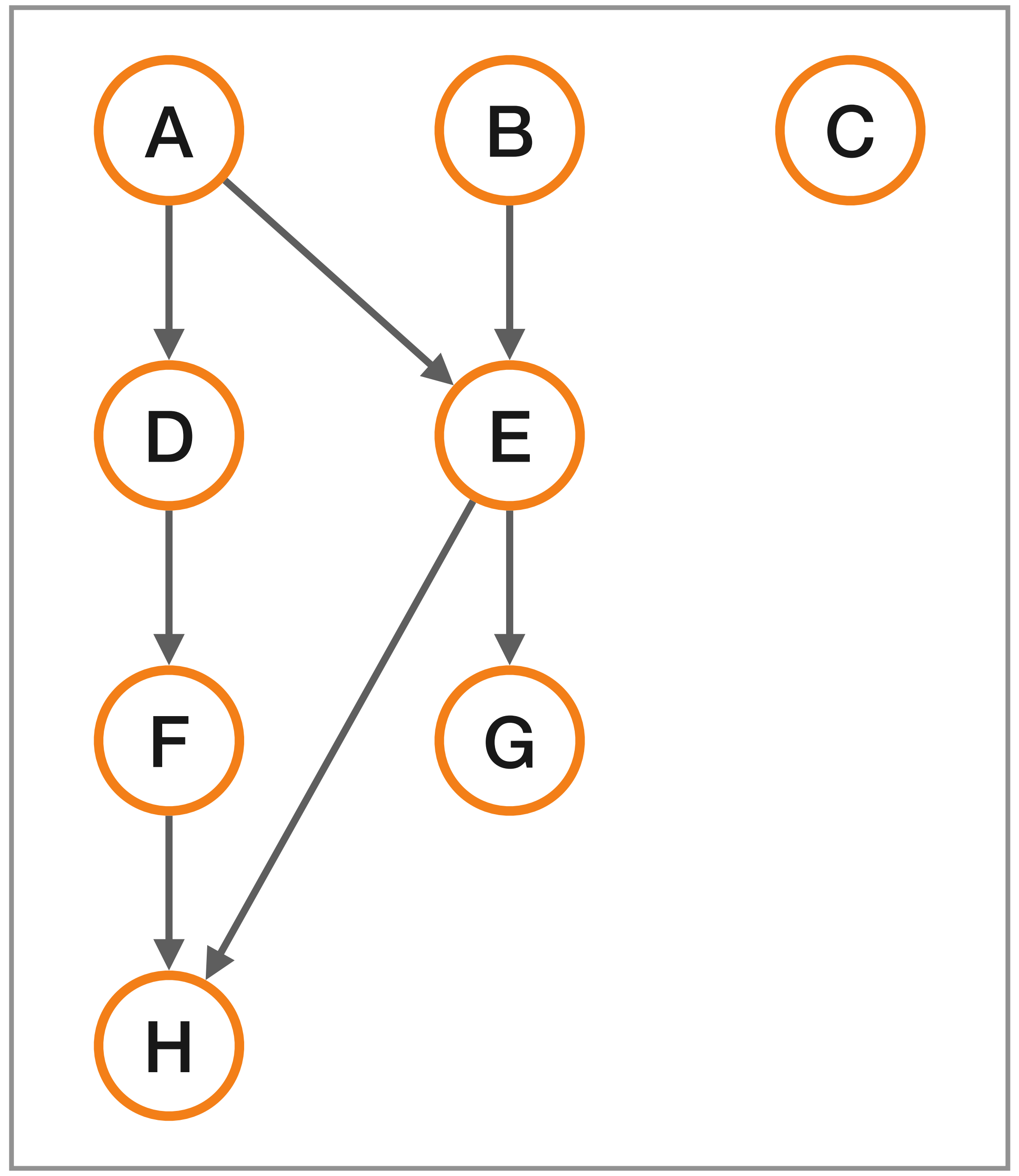
Repeat the steps again.

Topological Ordering:

A B C D E F

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 D E $\boxed{F}$ 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.

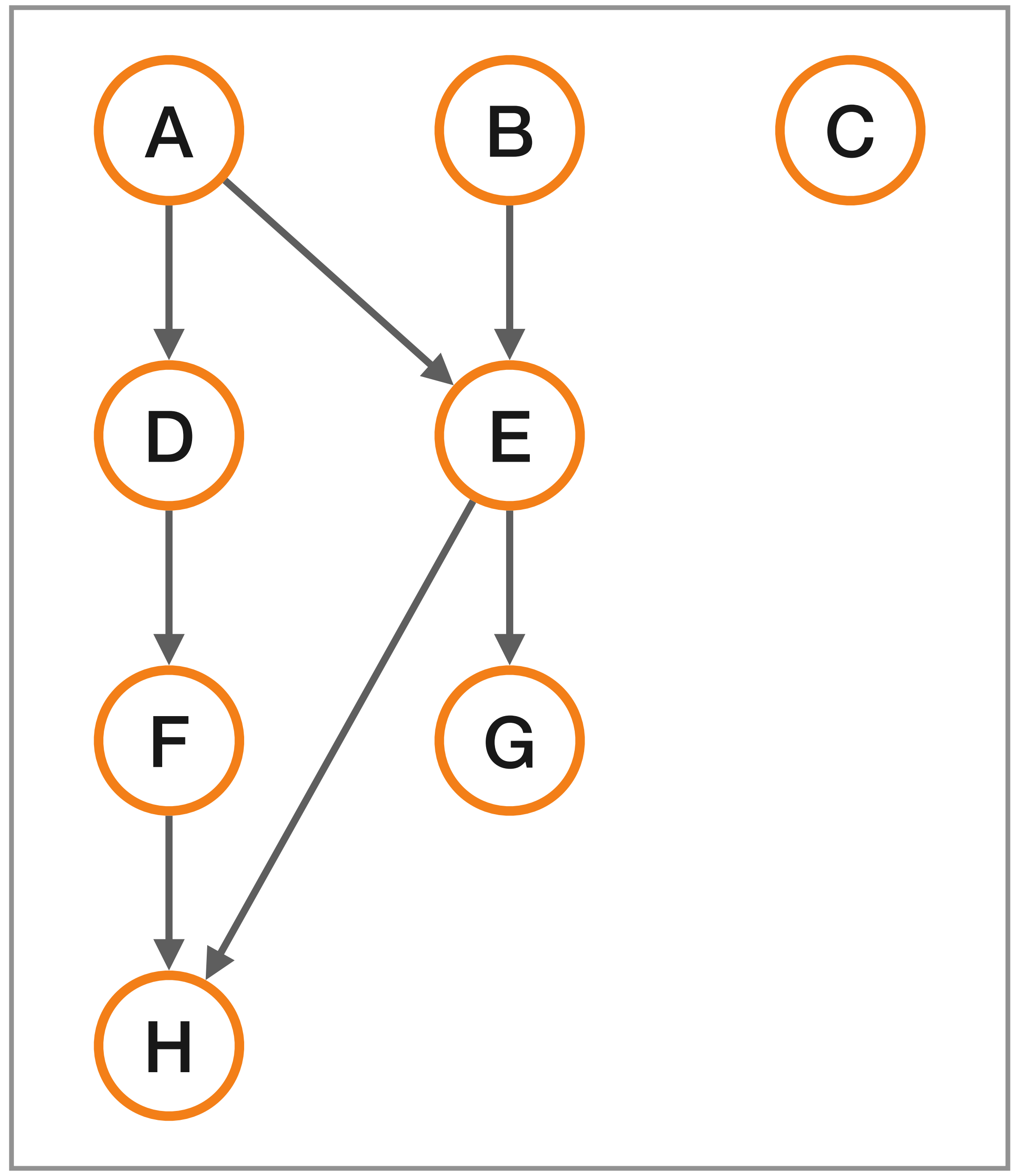
Repeat the steps again.

Topological Ordering:

A B C D E F

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 D E F $\boxed{\text{G}}$ 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.

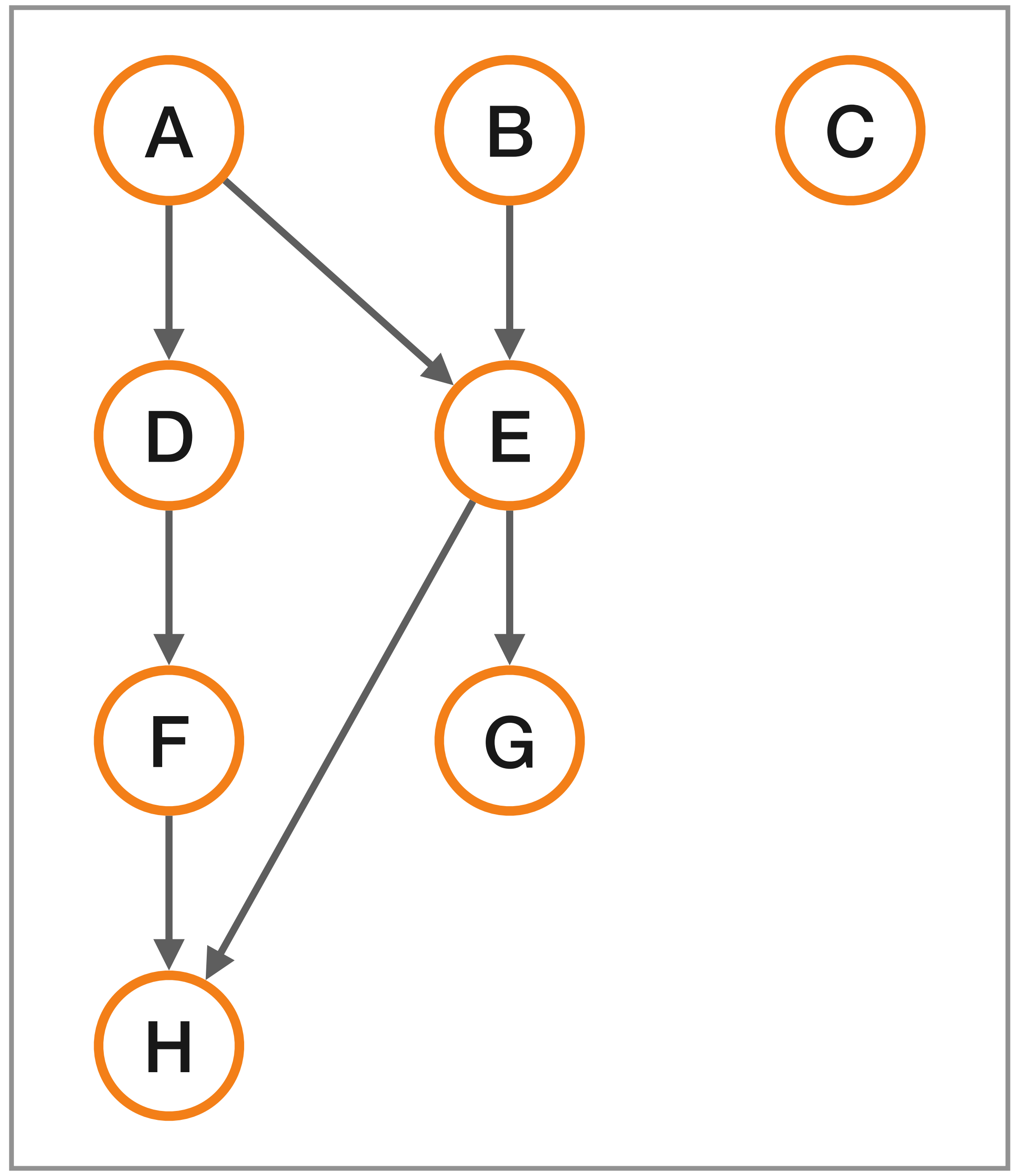
Repeat the steps again.

Topological Ordering:

A B C D E F G

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 D E F G $\boxed{\text{H}}$
1	
2	

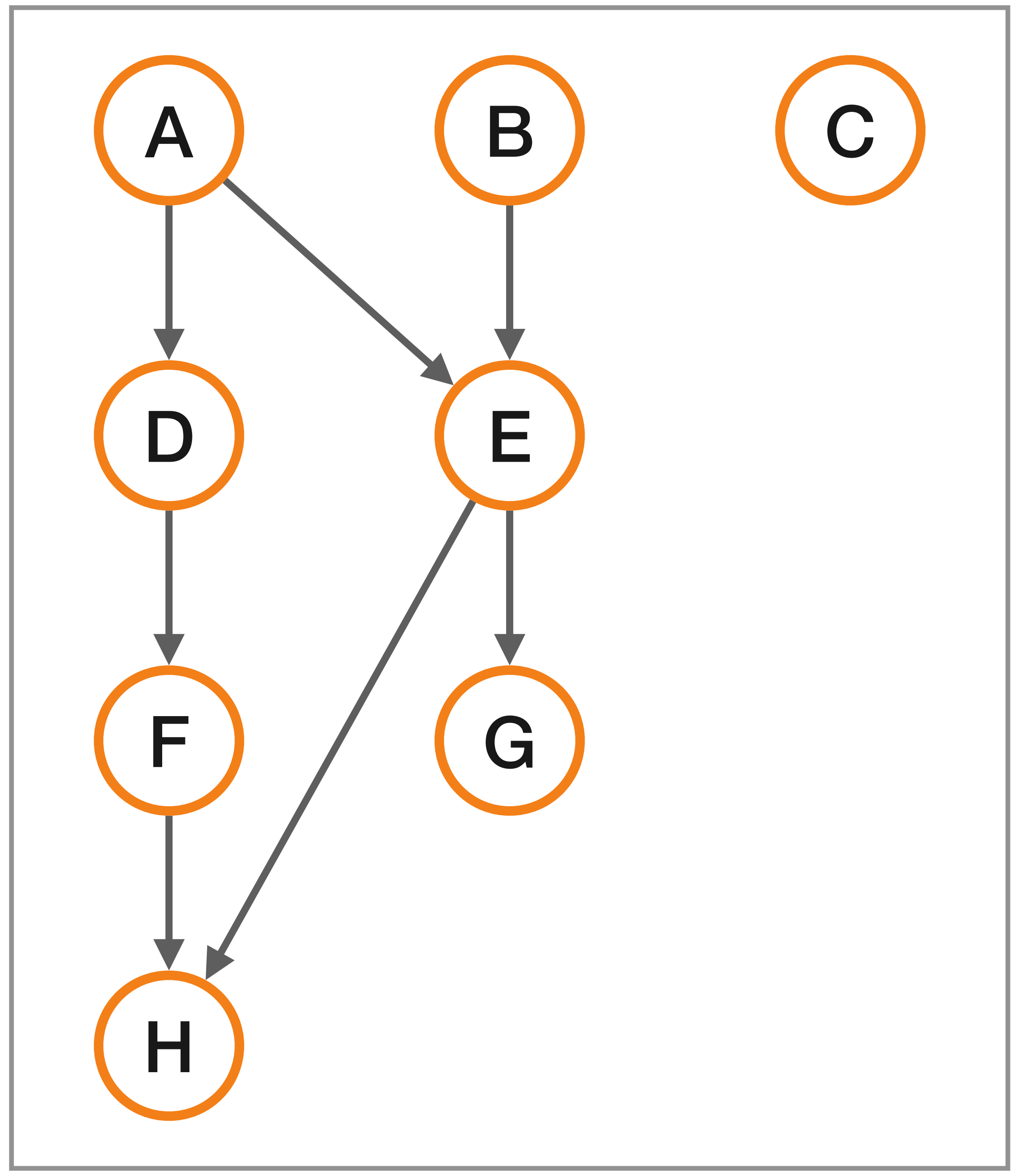
For each vertex that is source ($deg_{in}(v) = 0$):

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

Repeat the steps again.

Topological Ordering:

Topological sort



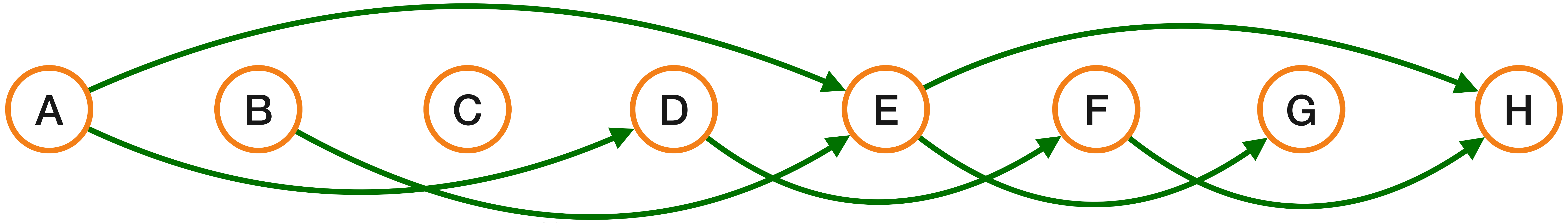
Node	Neighbors
A	D E
B	E
C	
D	F
E	H G
F	H
G	
H	

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

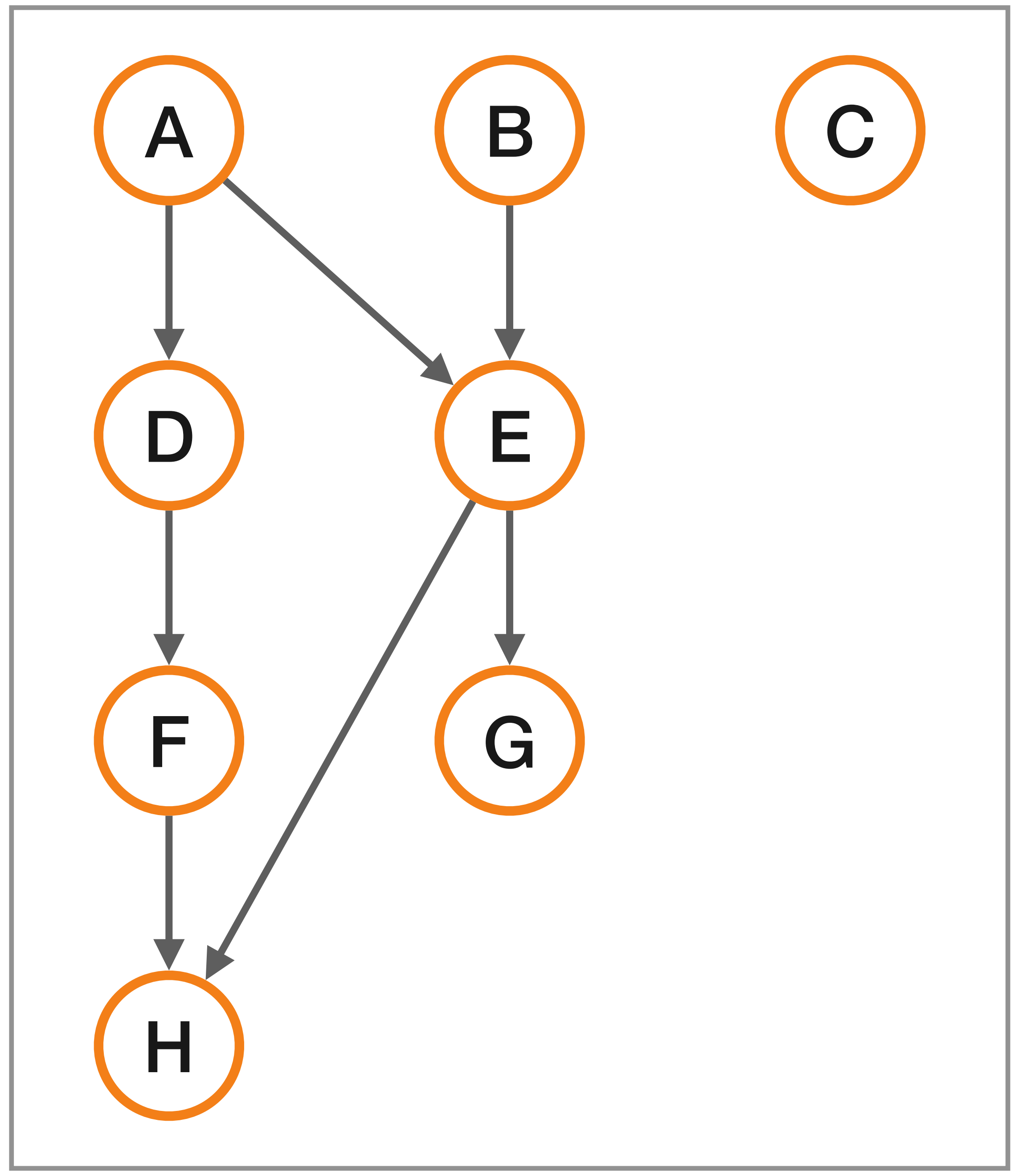
For each vertex that is source ($deg_{in}(v) = 0$):

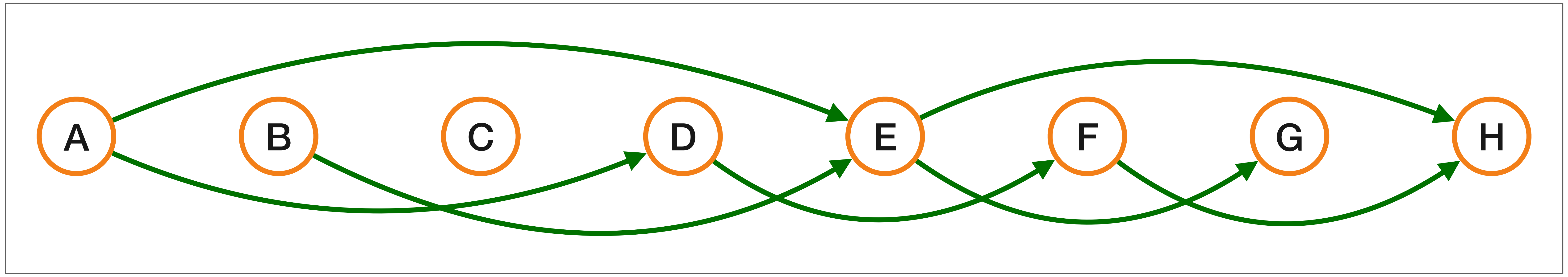
- Add v to the topological sort
- Lower degree of vertices v is connected to.

Topological Ordering:

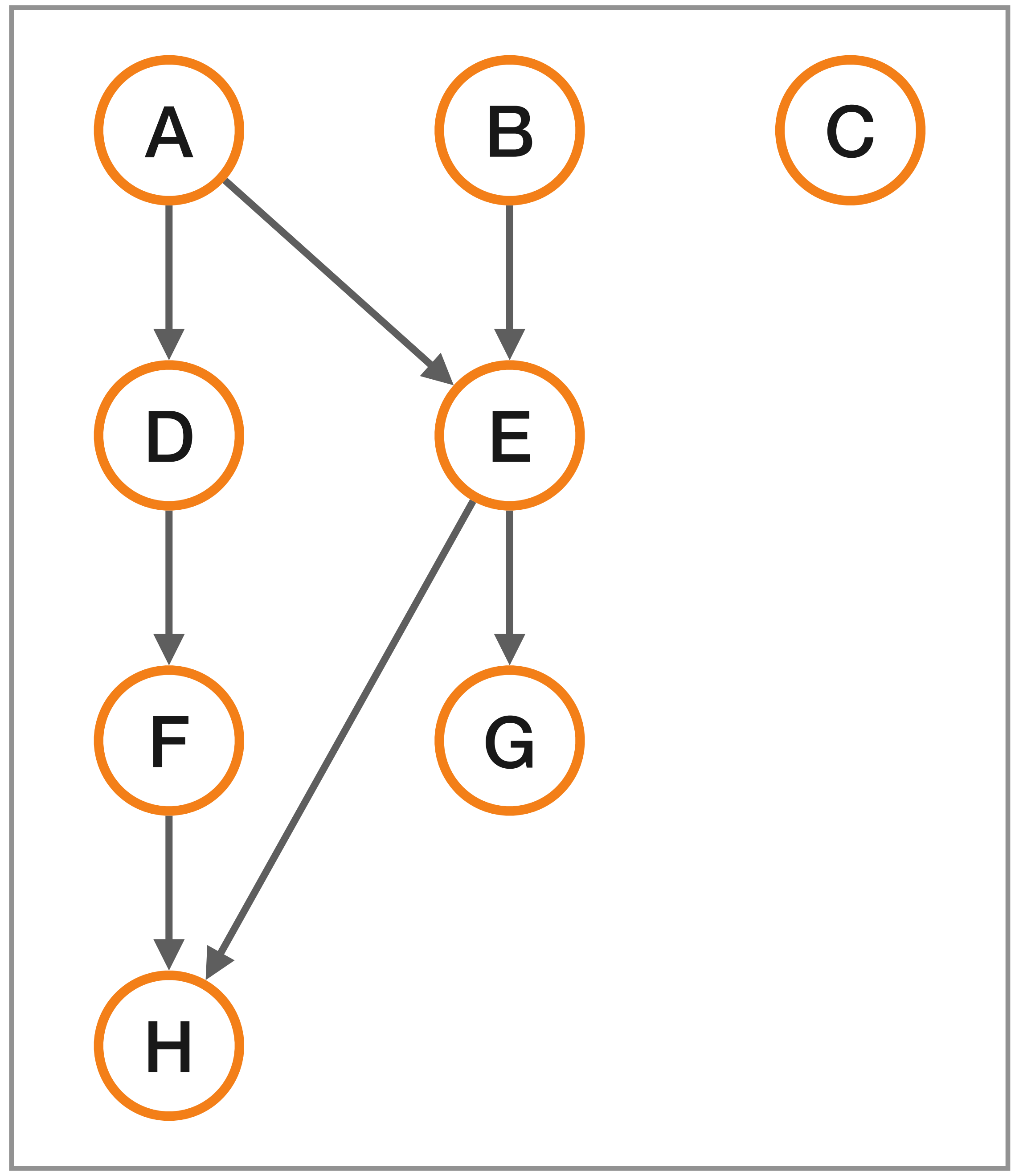


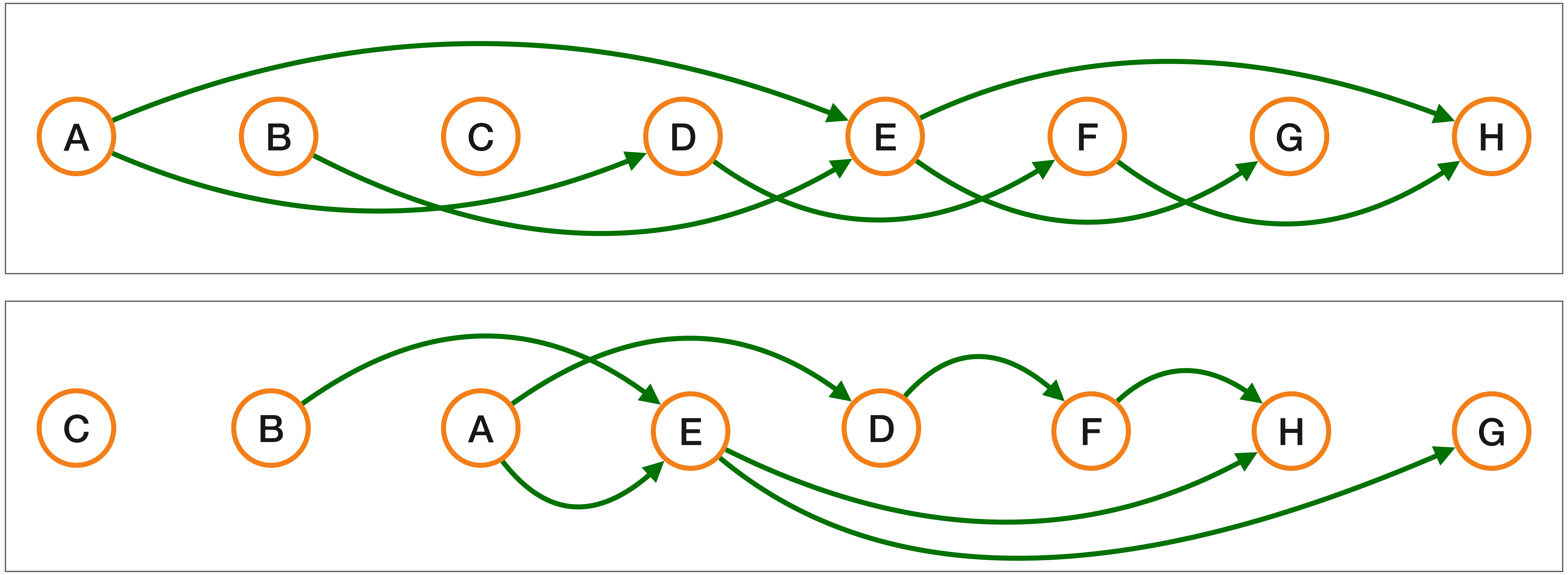
Multiple possible topological orderings



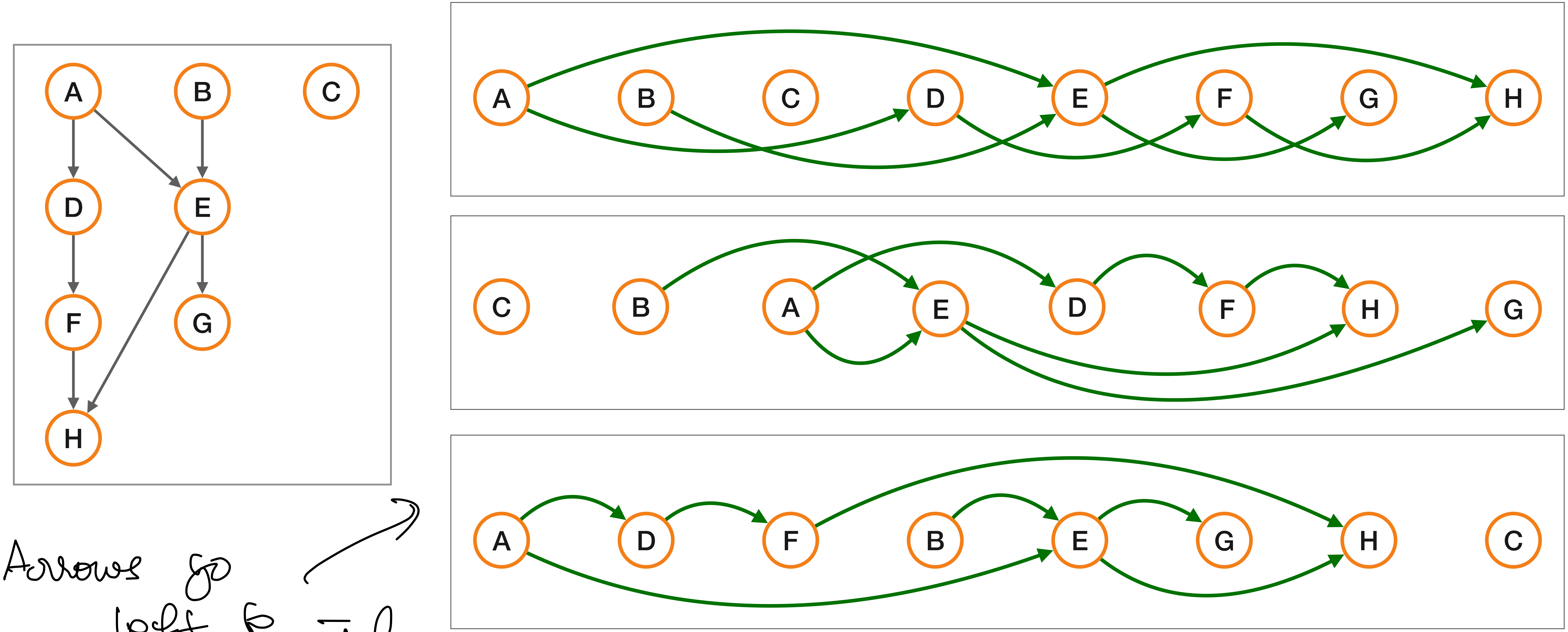


Multiple possible topological orderings





Multiple possible topological orderings



Arrows go left to right

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?

  completely disconnected (no edges whatsoever).

- **Exercise:** What is a DAG with the least number of distinct topological sorts for given n vertices?

  A graph that is a path (or "chain").

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

without loss of generality

Lemma: A directed graph G can be topologically ordered $\Longrightarrow$ 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:

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:

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

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:

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

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

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:

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

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

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 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 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

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.

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

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.

Graph

```
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
```

vertex

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 visited
    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)
```

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 visited
    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.

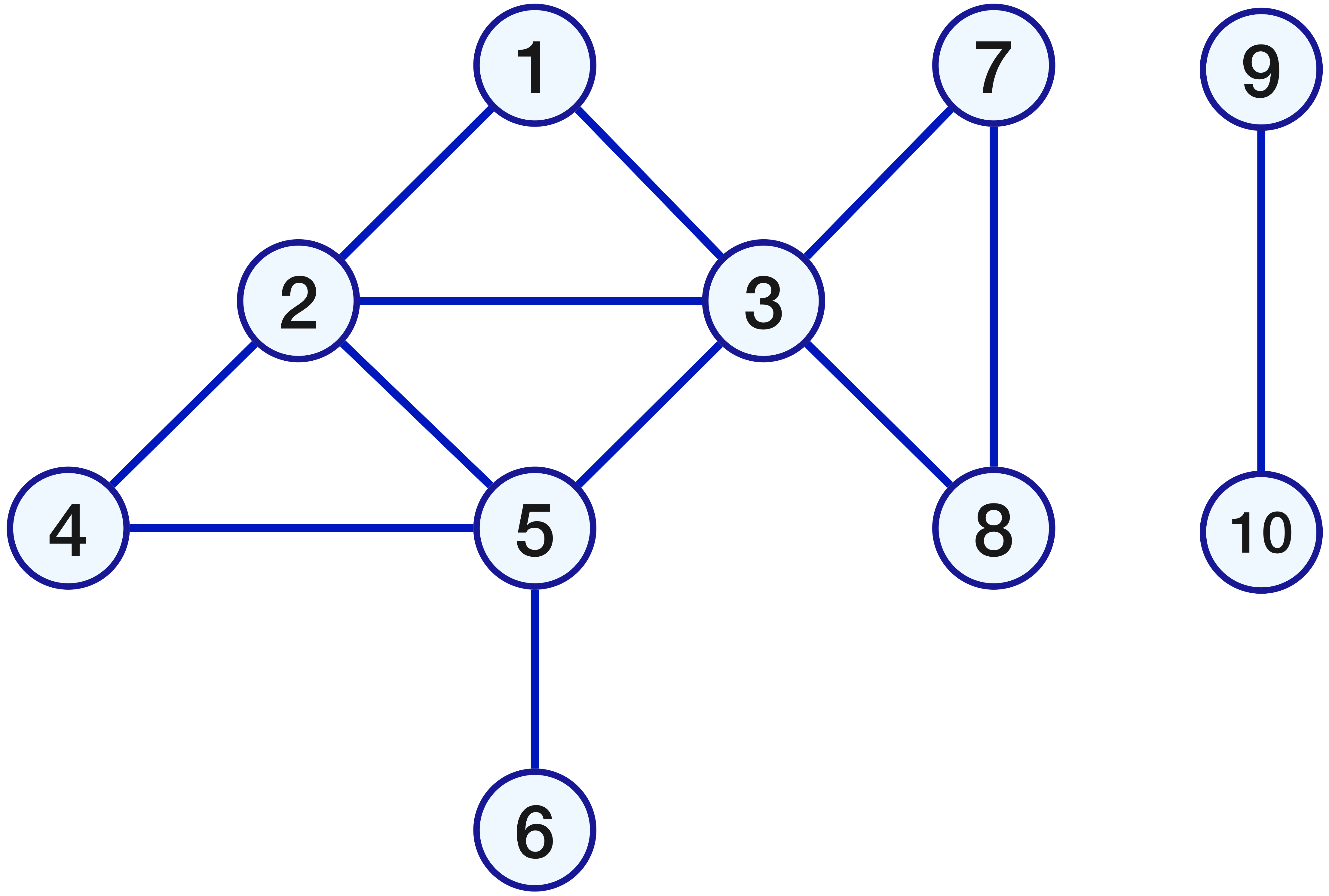
pre, post → timestamps

DFS with pre-post numbering

first visit time

when we are done w/ that vertex

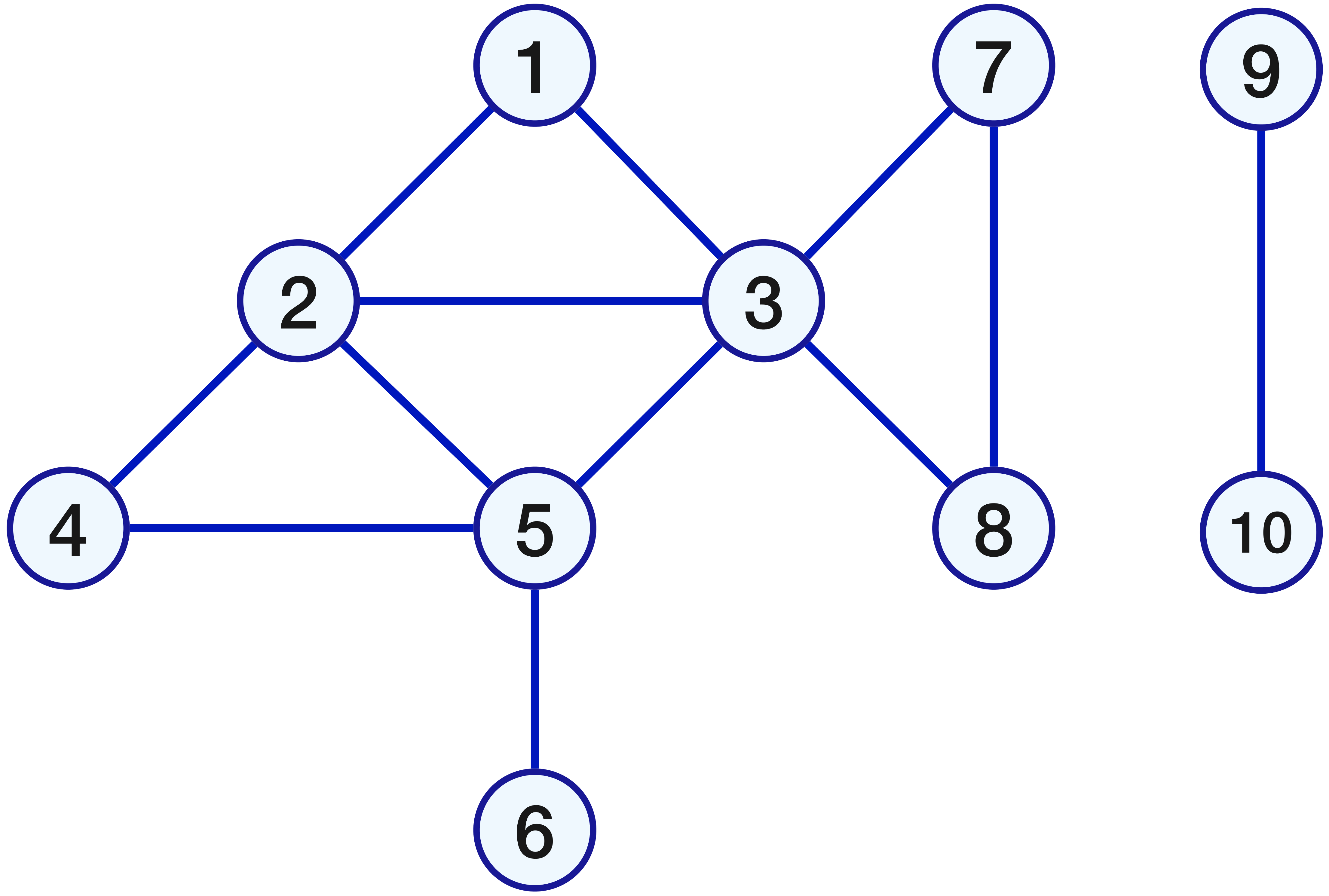
Vertex	[Pre, Post]



DFS with pre-post numbering

Time = 0

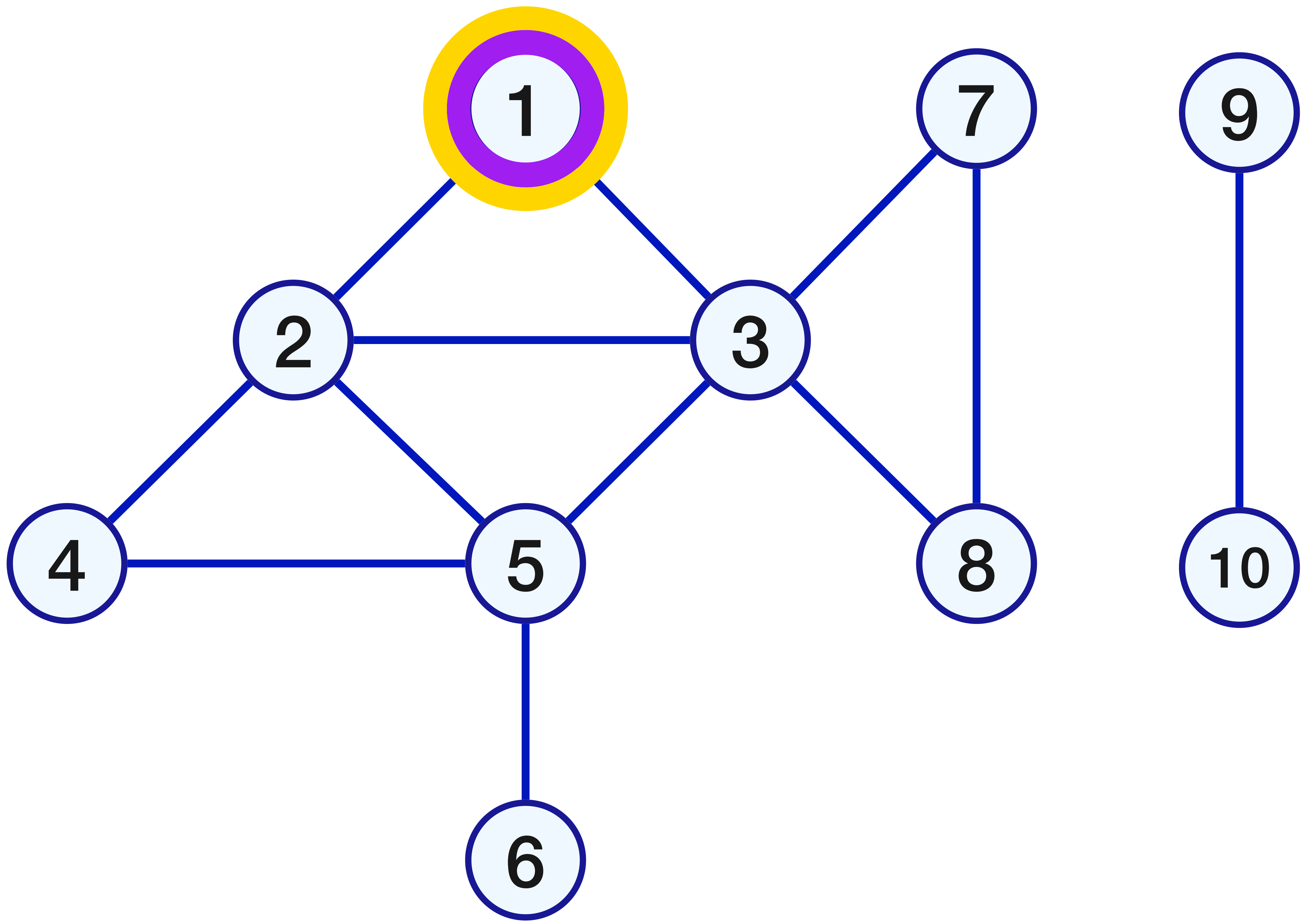
Vertex	*[Pre, Post]*



DFS with pre-post numbering

Time = 1

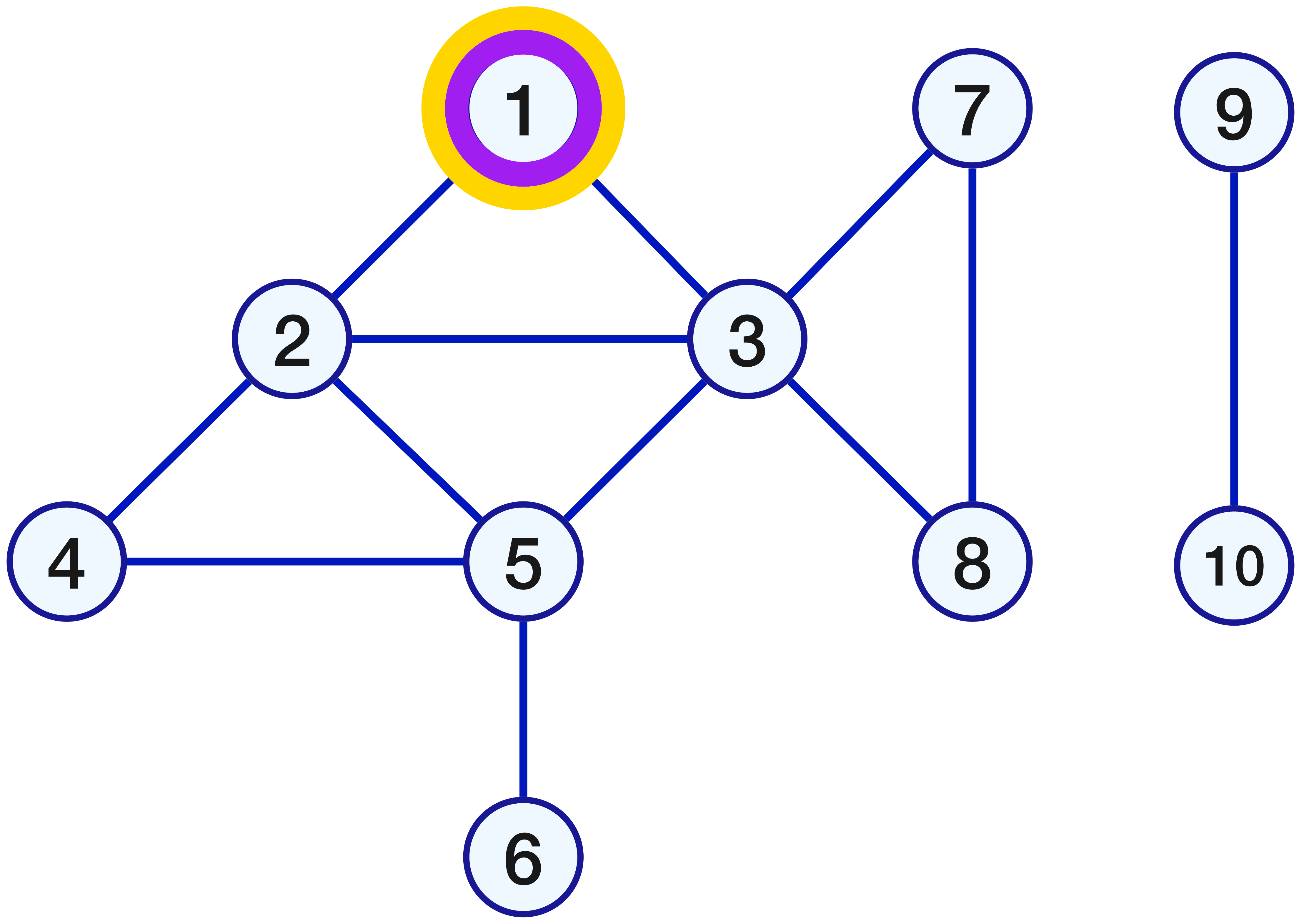
Vertex	*[Pre, Post]*



DFS with pre-post numbering

Time = 1

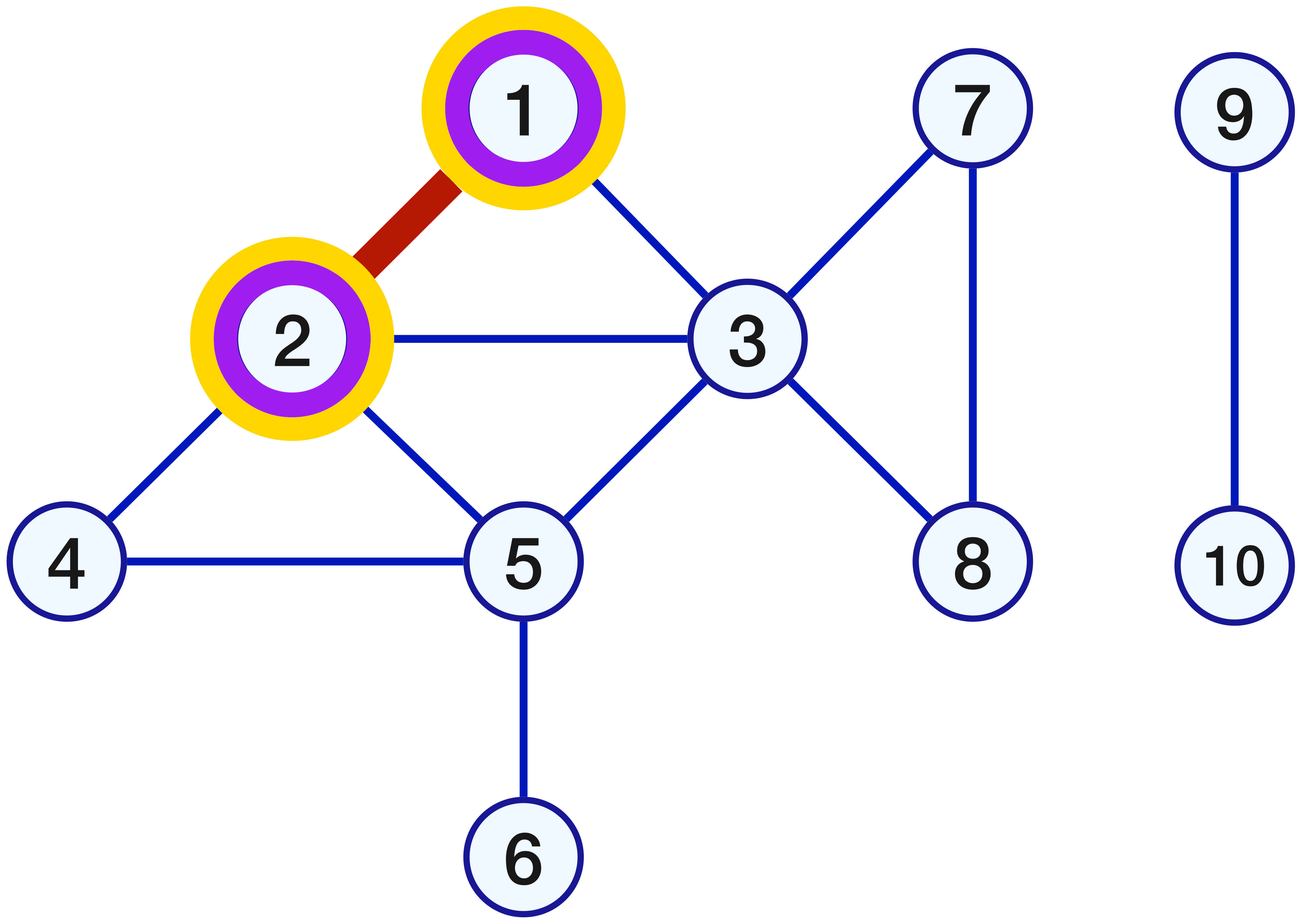
Vertex	[Pre, Post]
1	[1,]



DFS with pre-post numbering

Time = 2

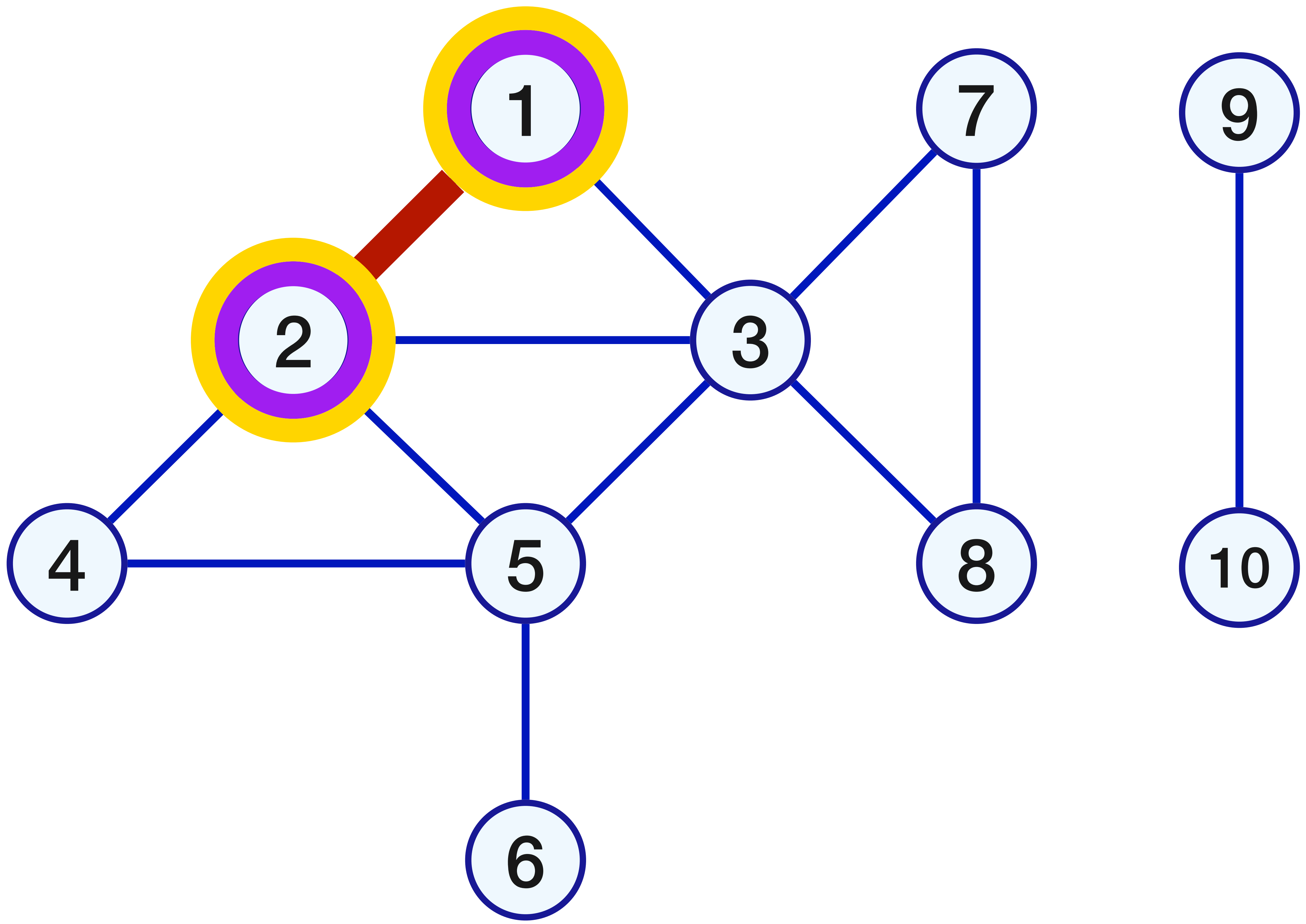
Vertex	*[Pre, Post]*
1	[1,]



DFS with pre-post numbering

Time = 2

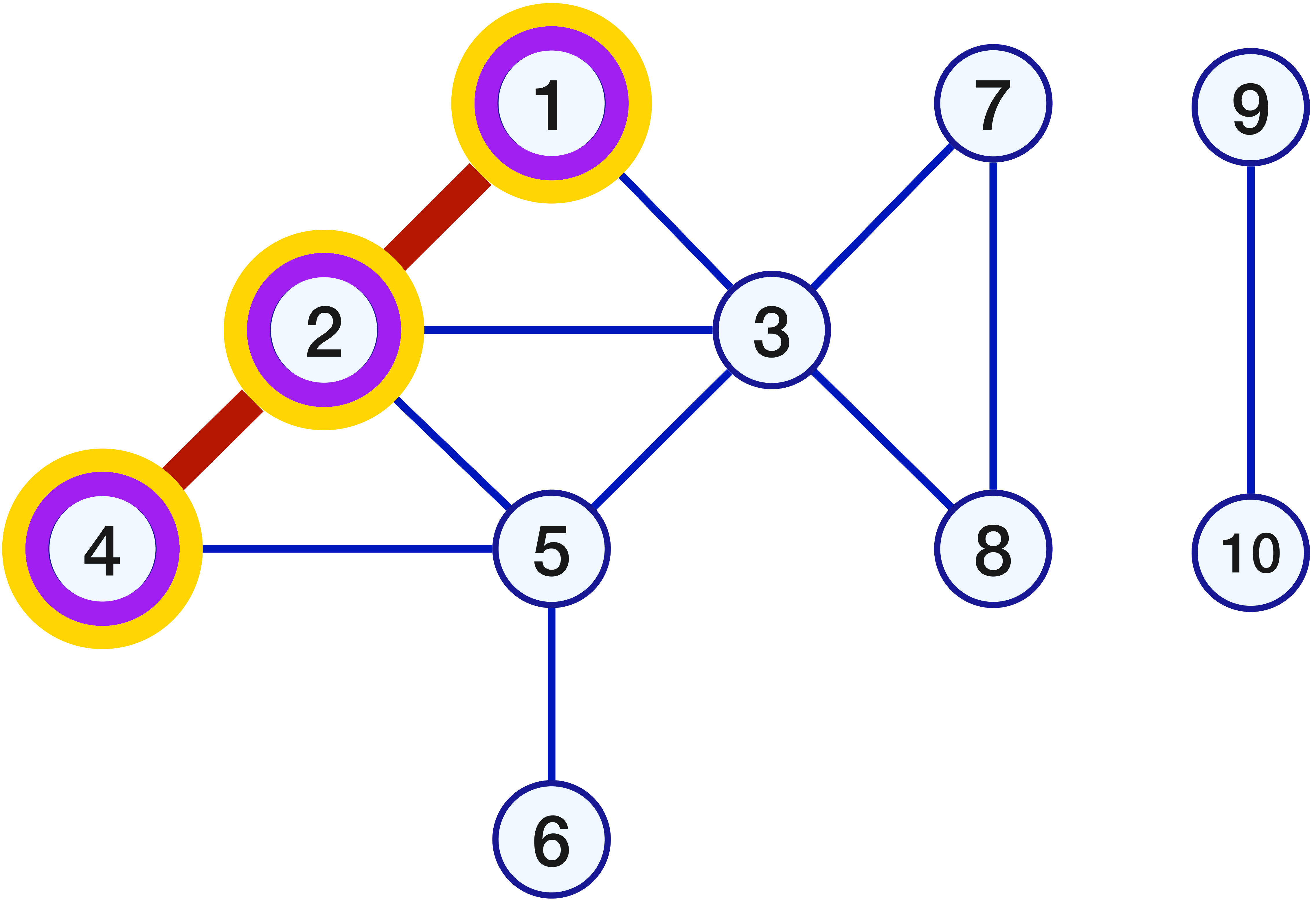
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]



DFS with pre-post numbering

Time = 3

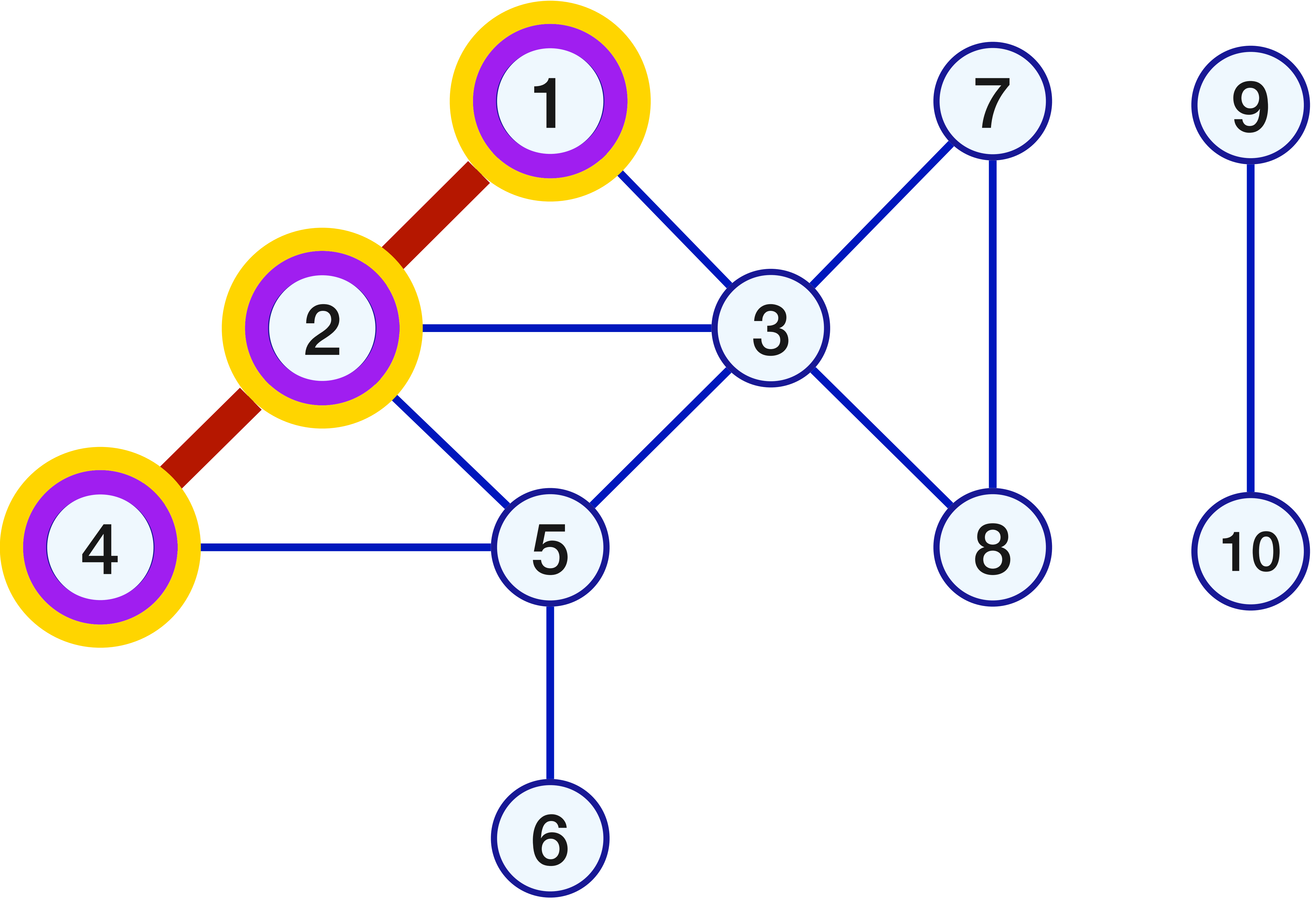
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]



DFS with pre-post numbering

Time = 3

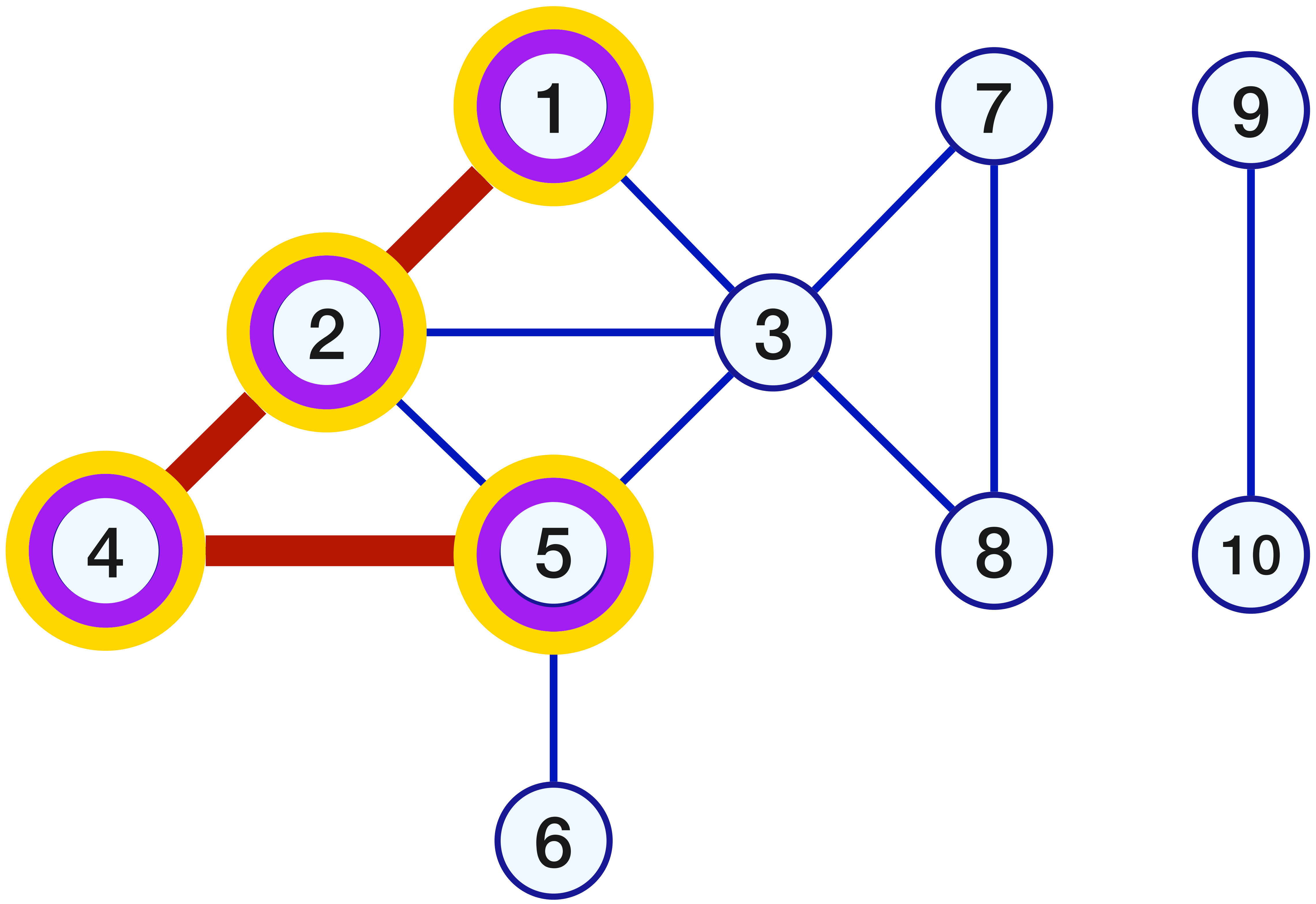
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]



DFS with pre-post numbering

Time = 4

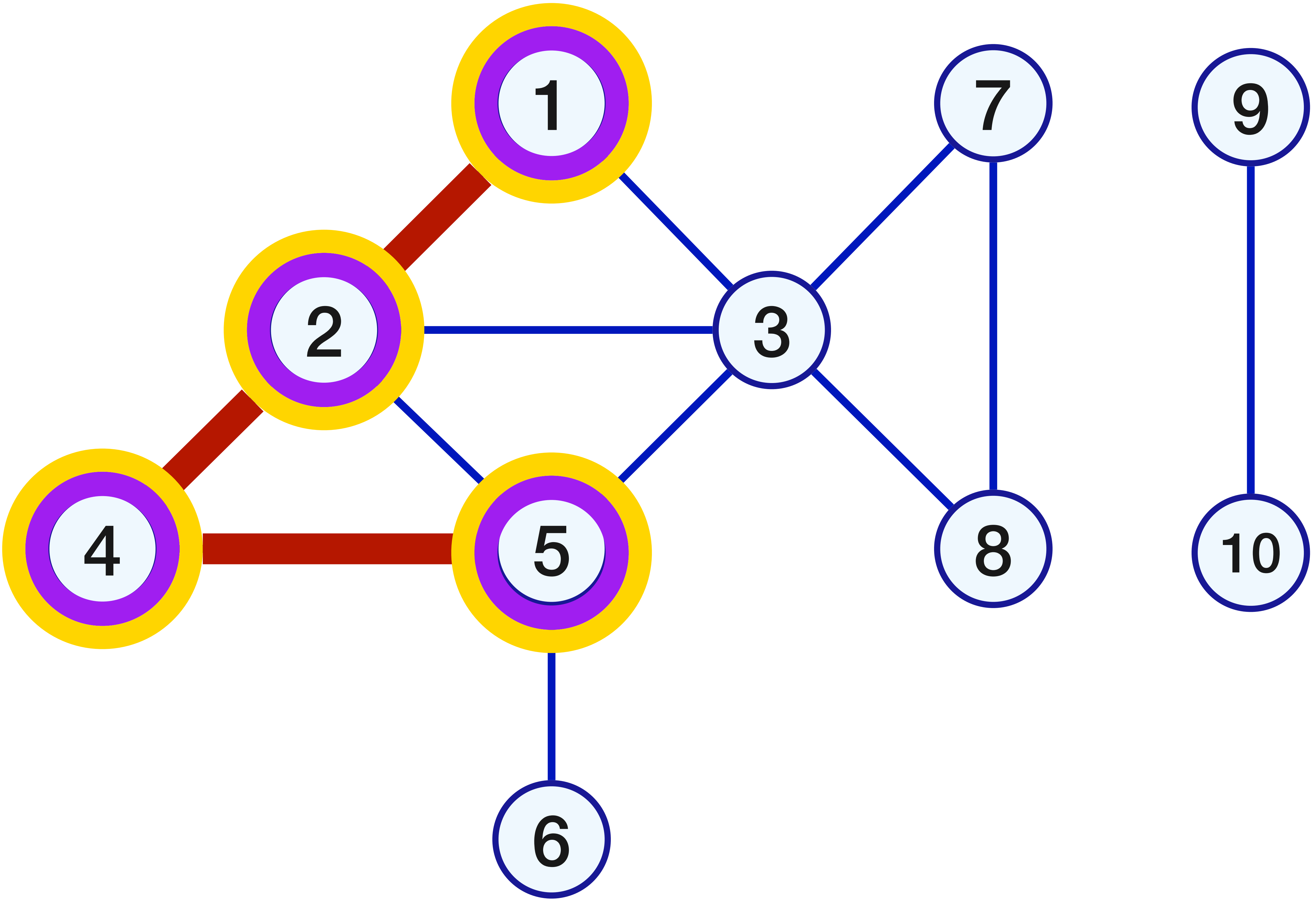
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]



DFS with pre-post numbering

Time = 4

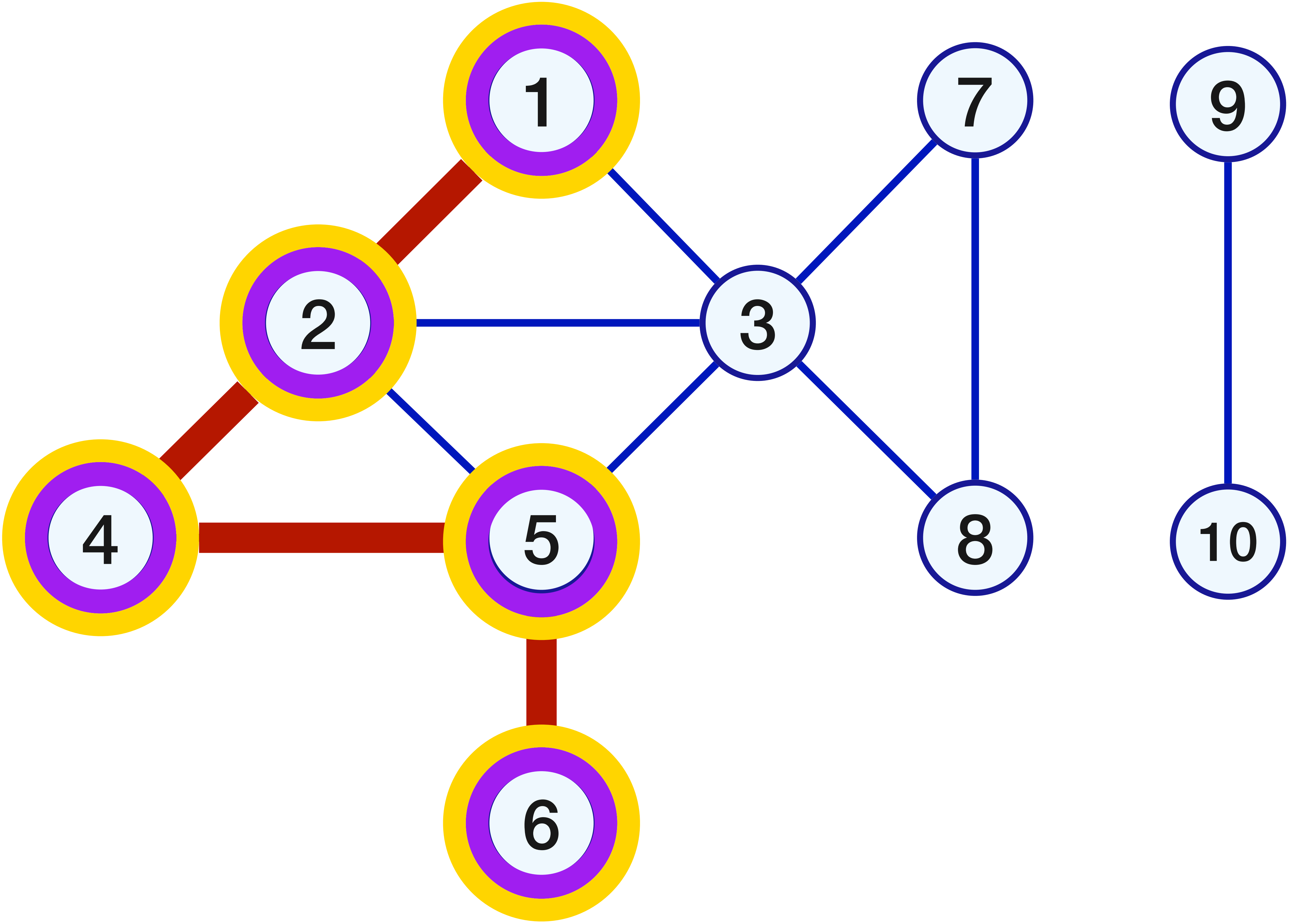
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]



DFS with pre-post numbering

Time = 5

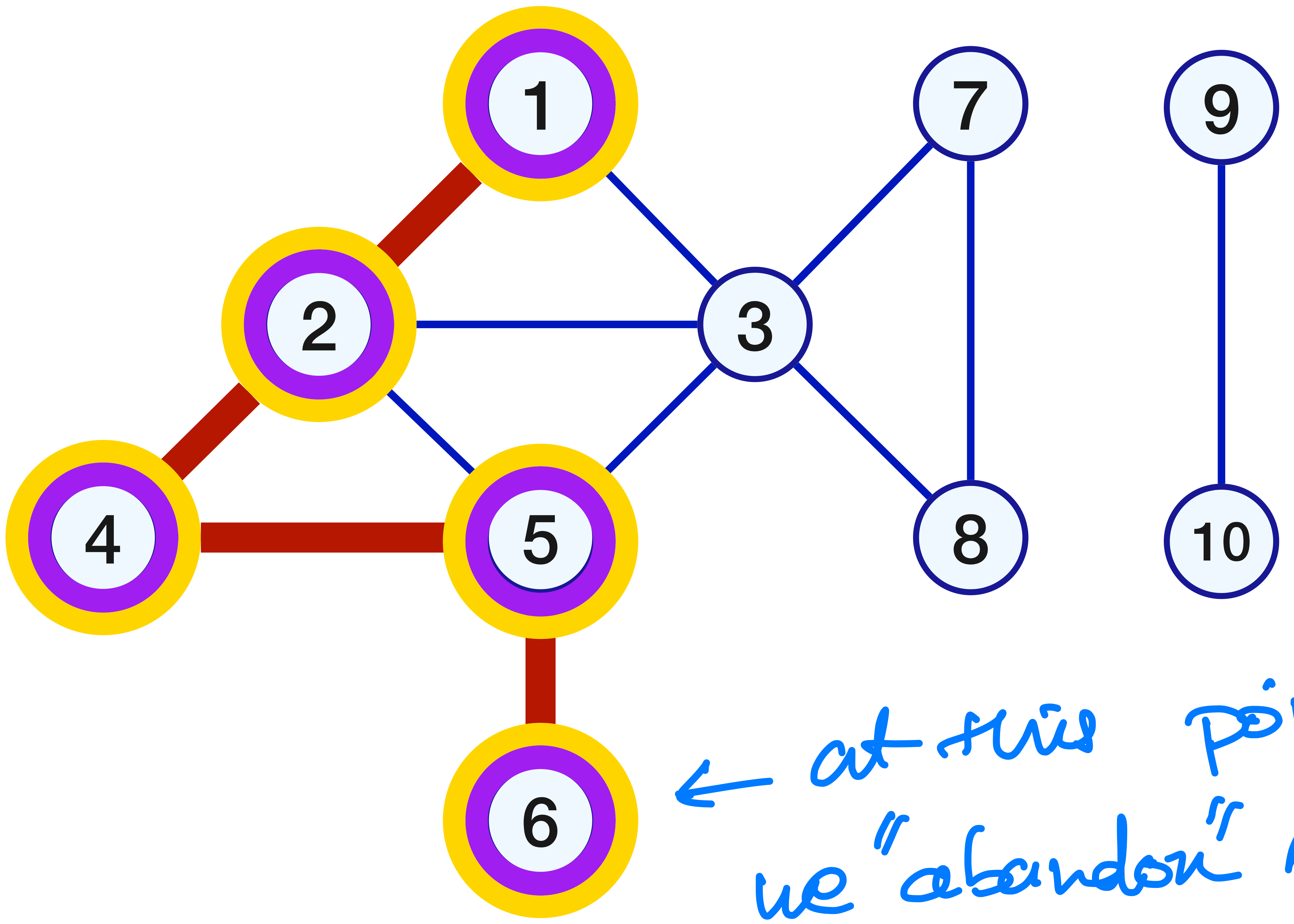
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]



DFS with pre-post numbering

Time = 5

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

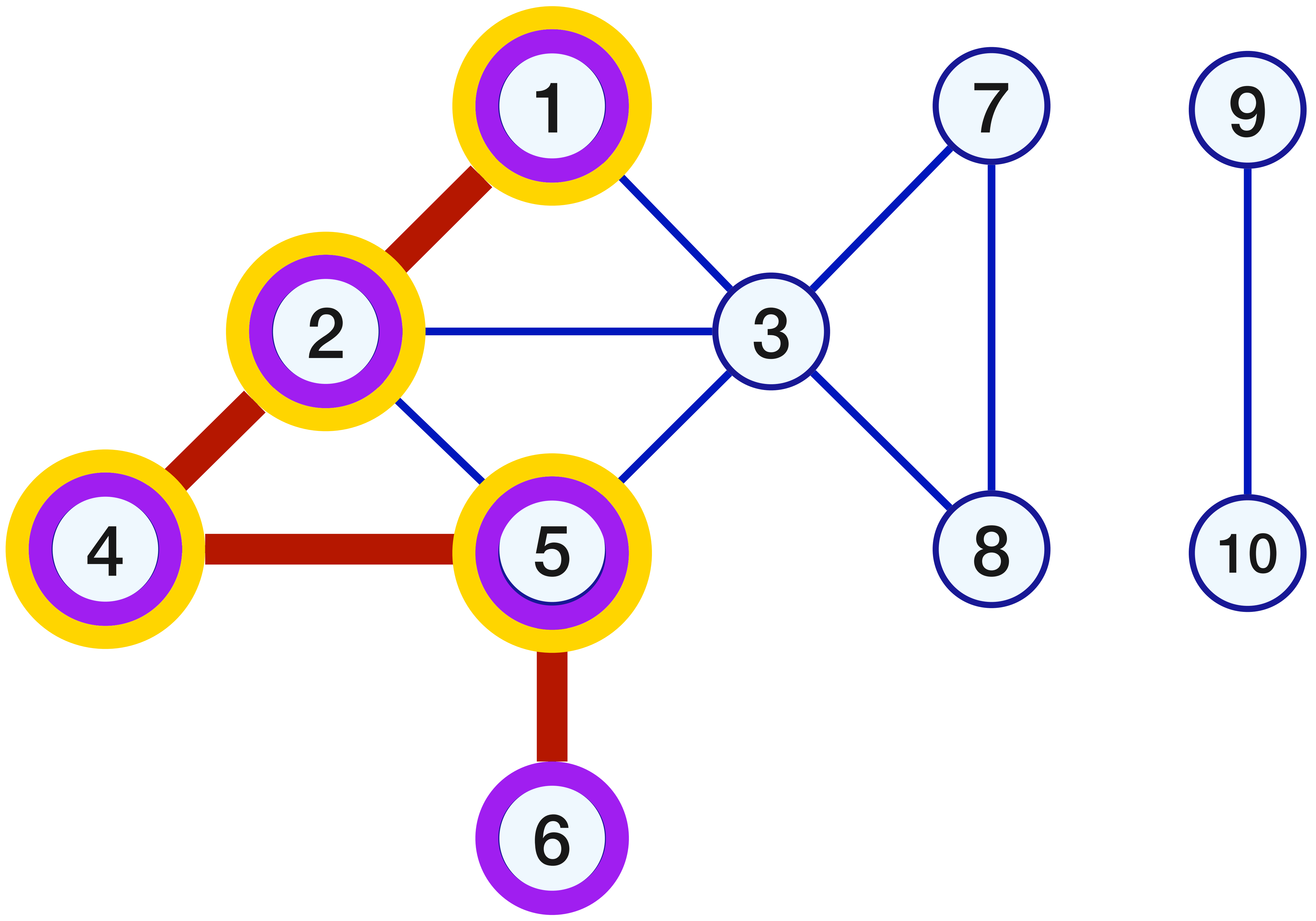


← at this point
we "abandon" node 6

DFS with pre-post numbering

Time = 6

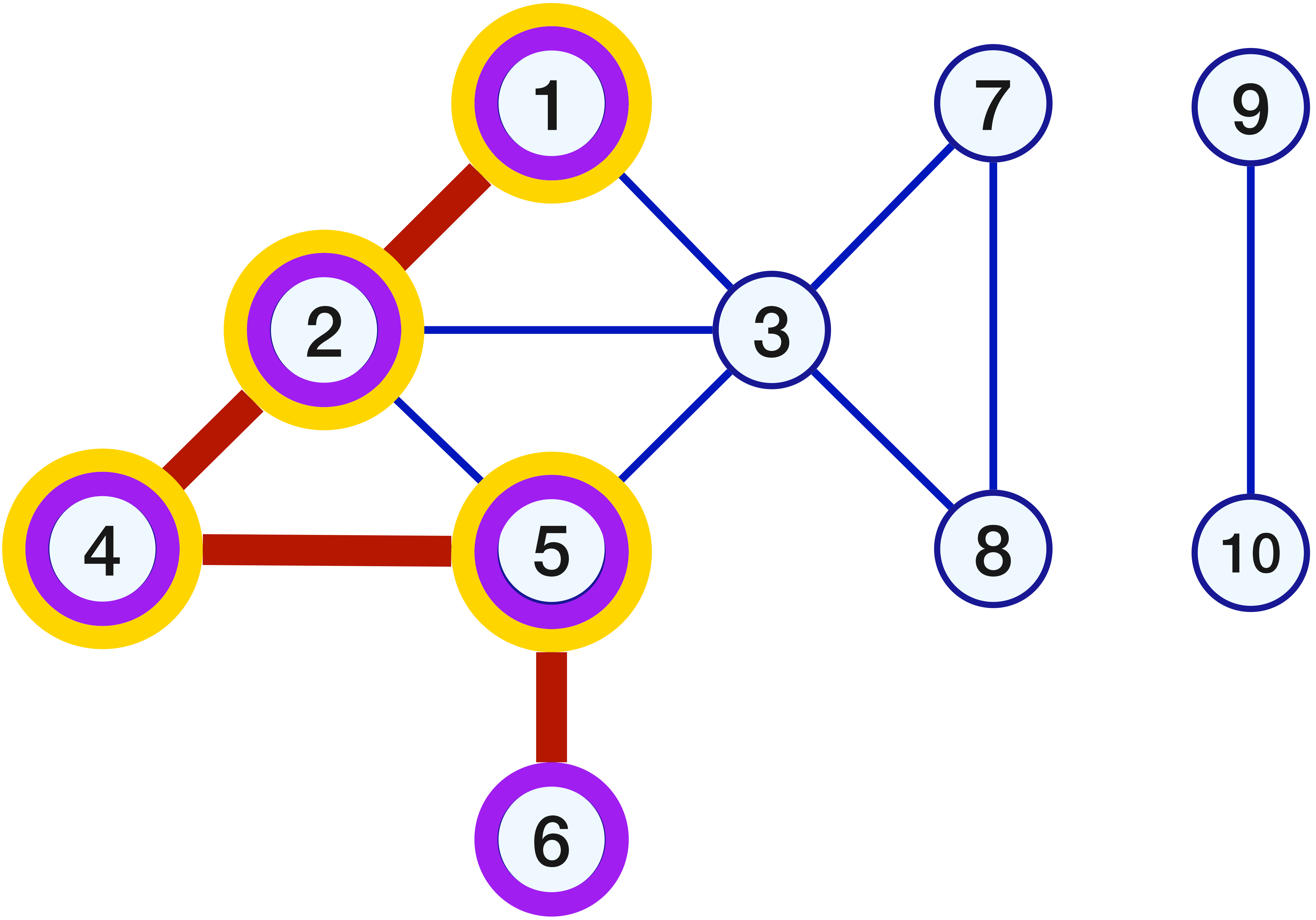
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5,]



DFS with pre-post numbering

Time = 6

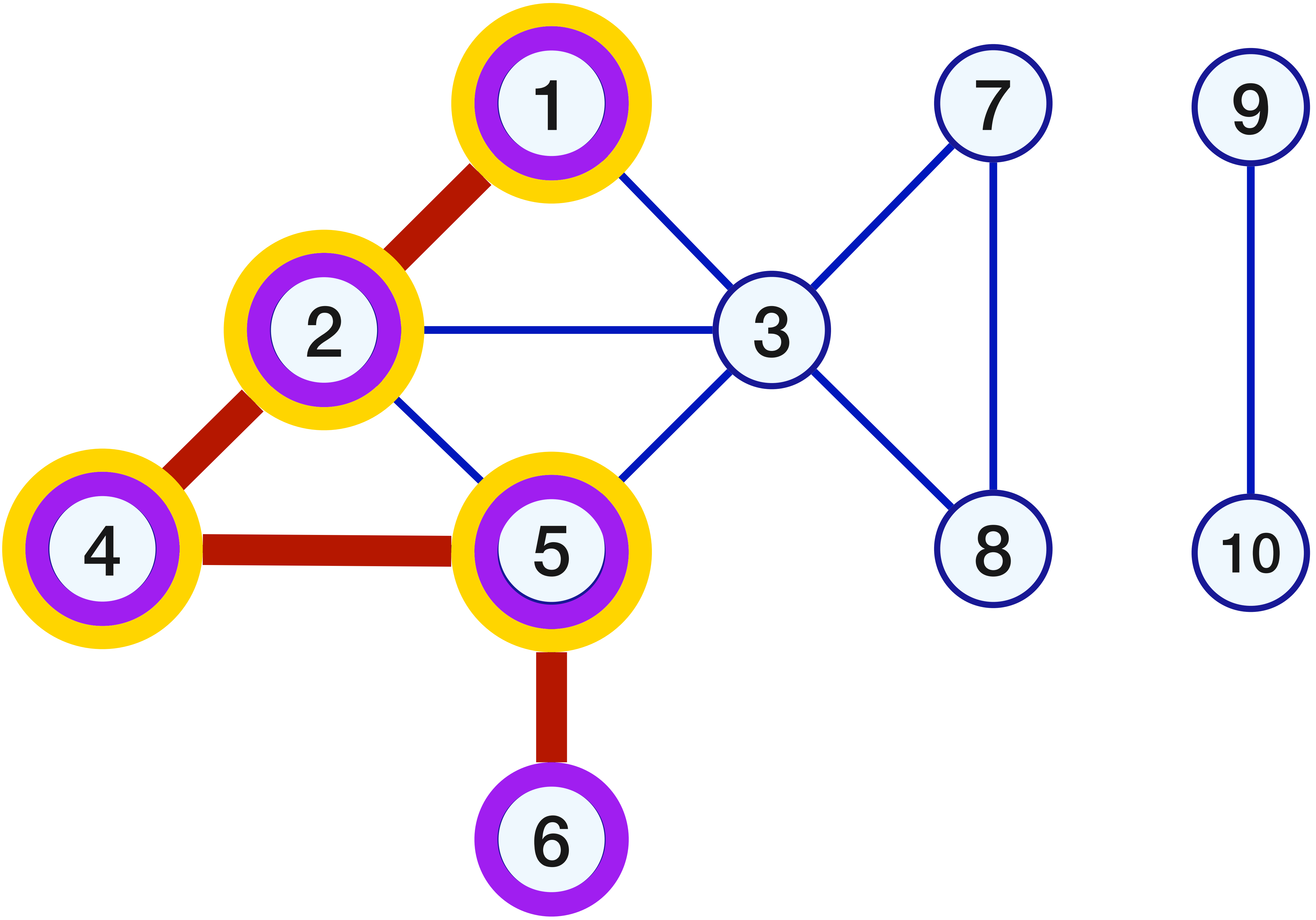
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5,]



DFS with pre-post numbering

Time = 6

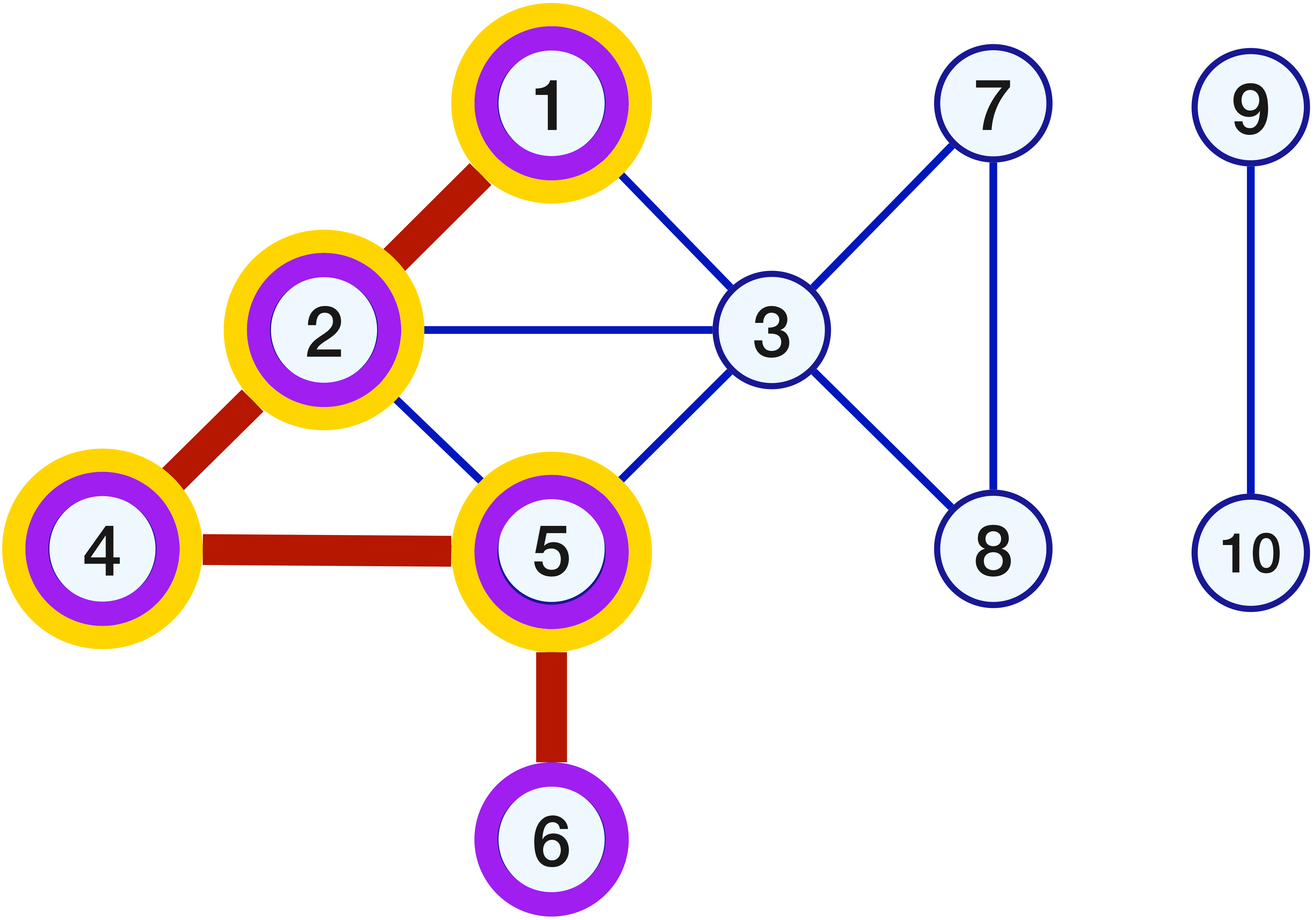
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]



DFS with pre-post numbering

Time = 7

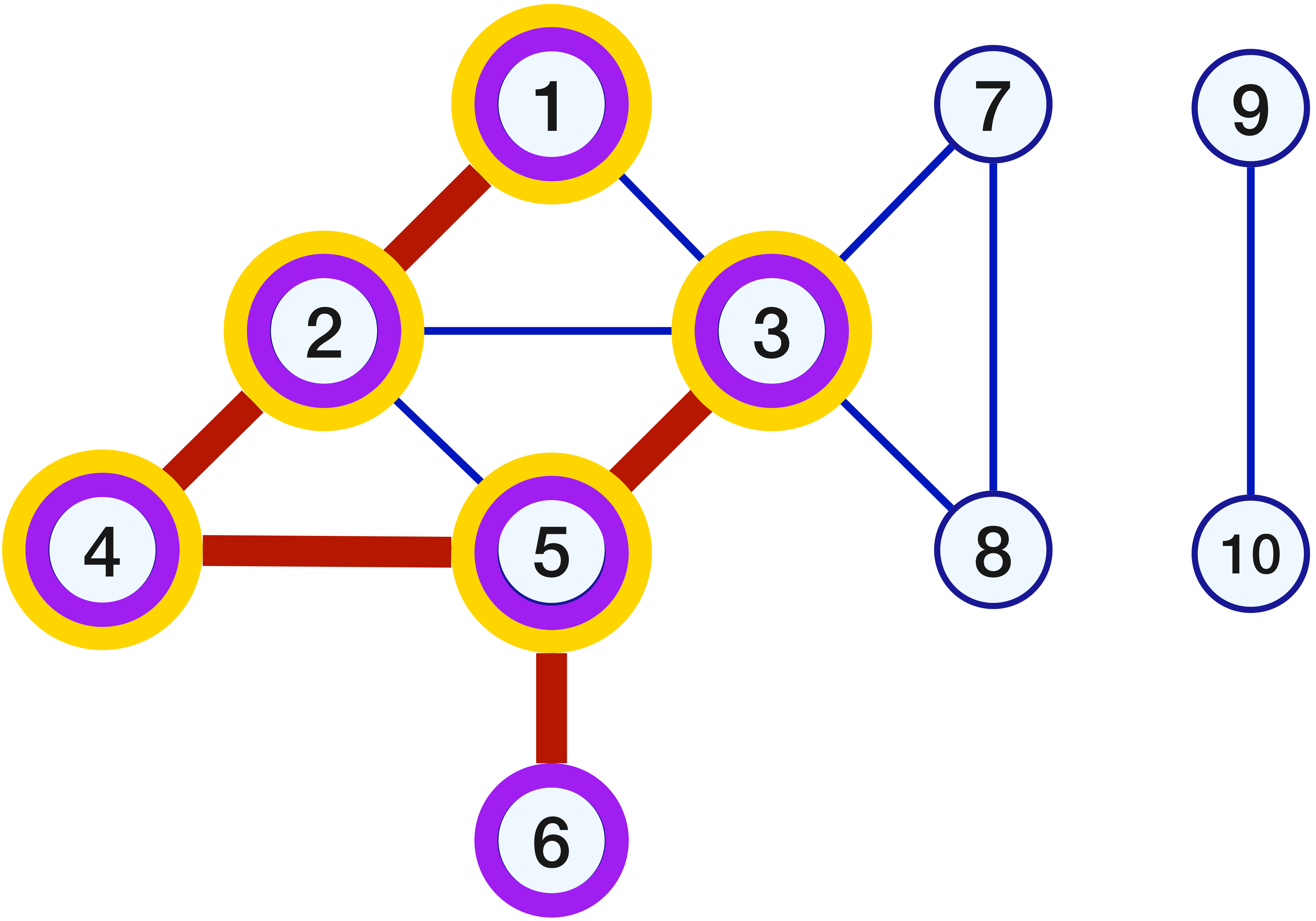
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]



DFS with pre-post numbering

Time = 7

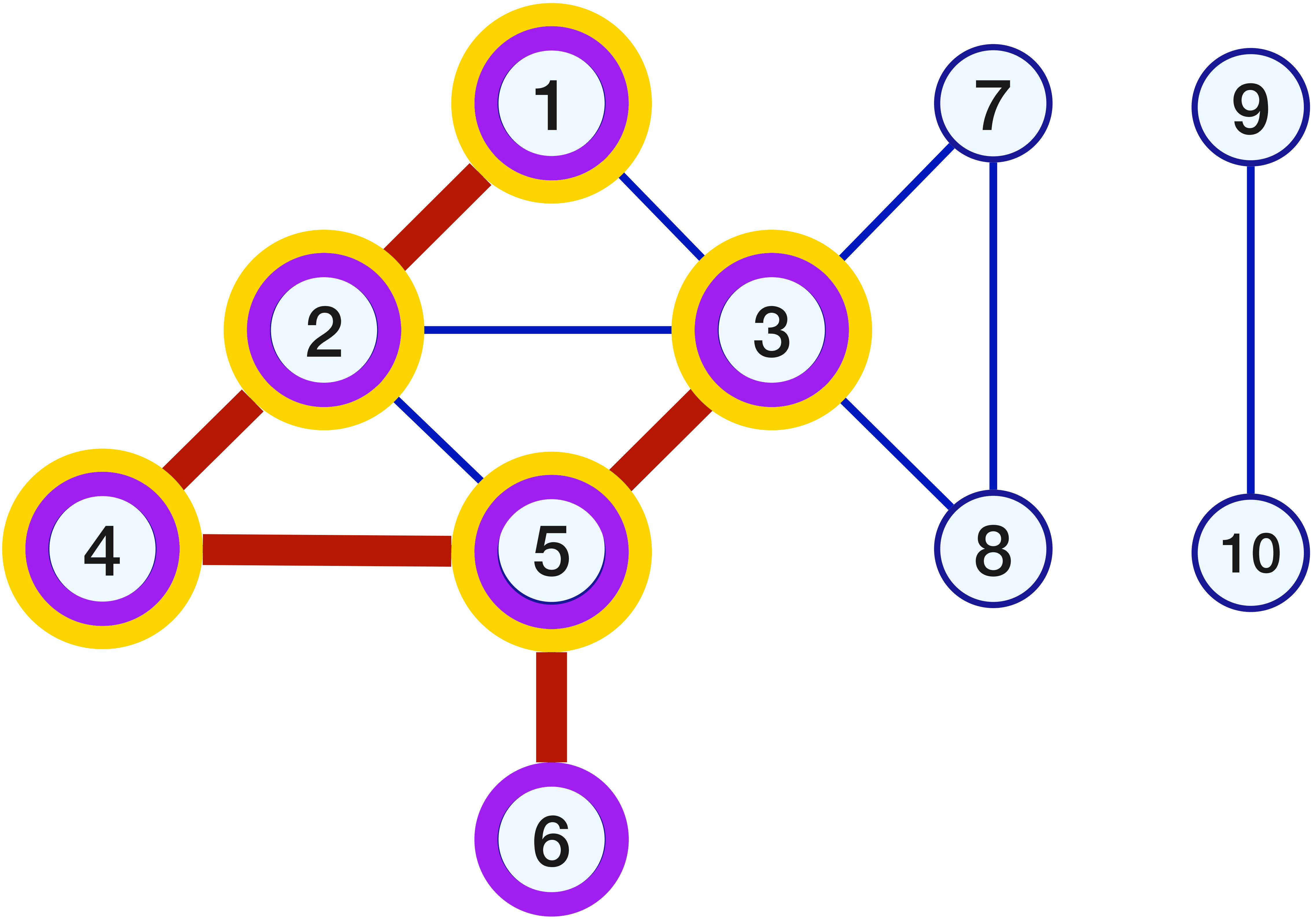
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]



DFS with pre-post numbering

Time = 7

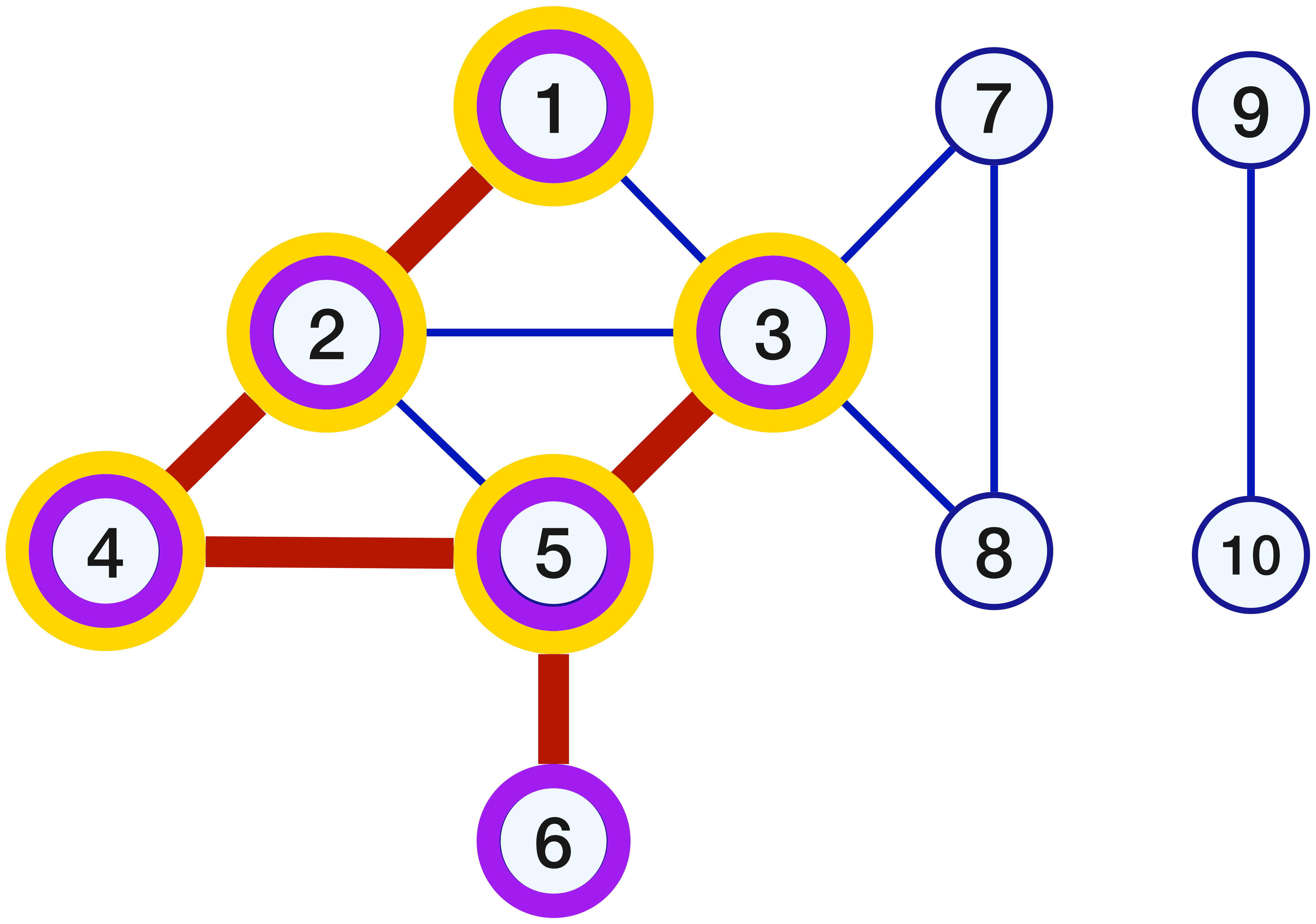
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]



DFS with pre-post numbering

Time = 8

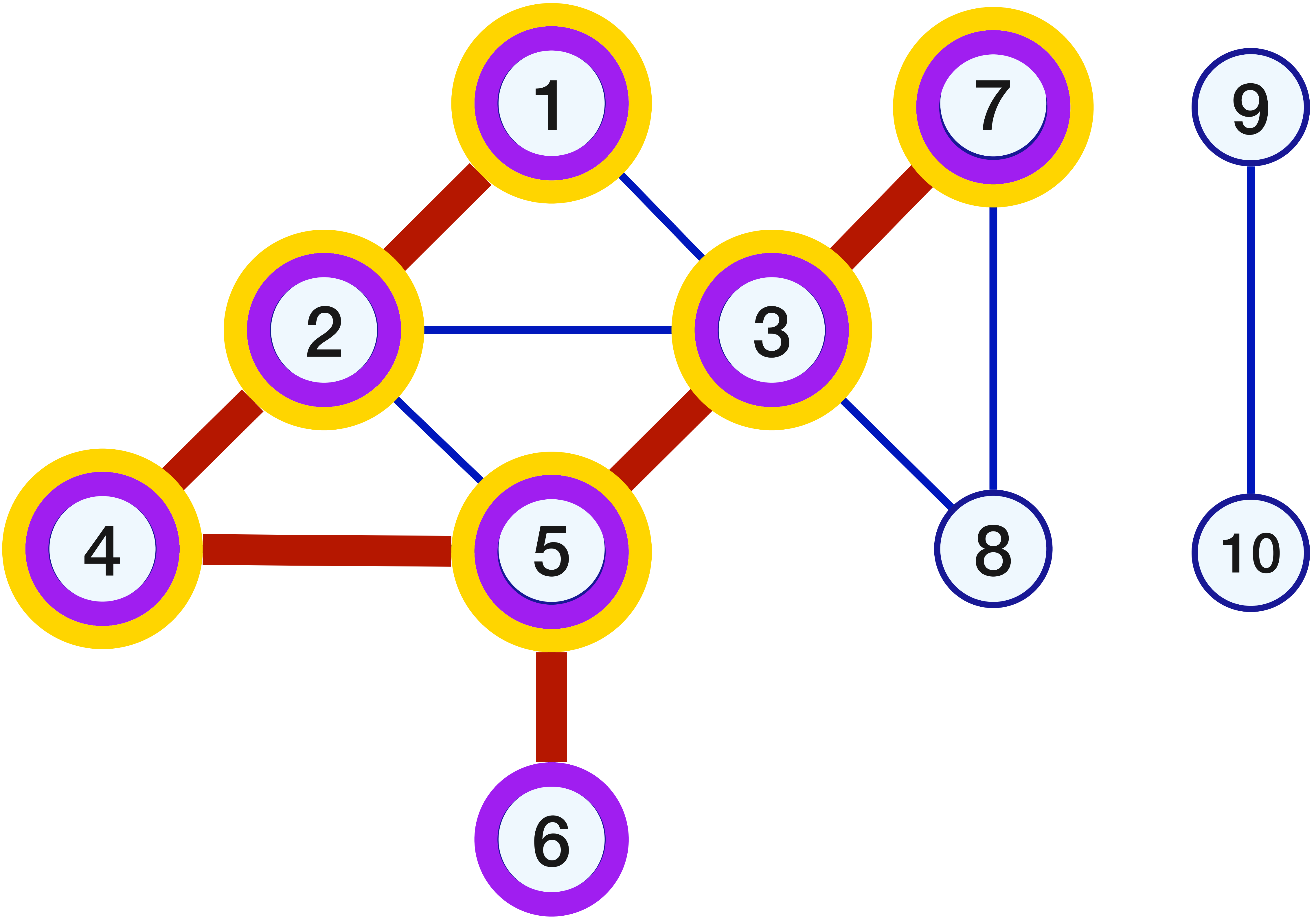
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]



DFS with pre-post numbering

Time = 8

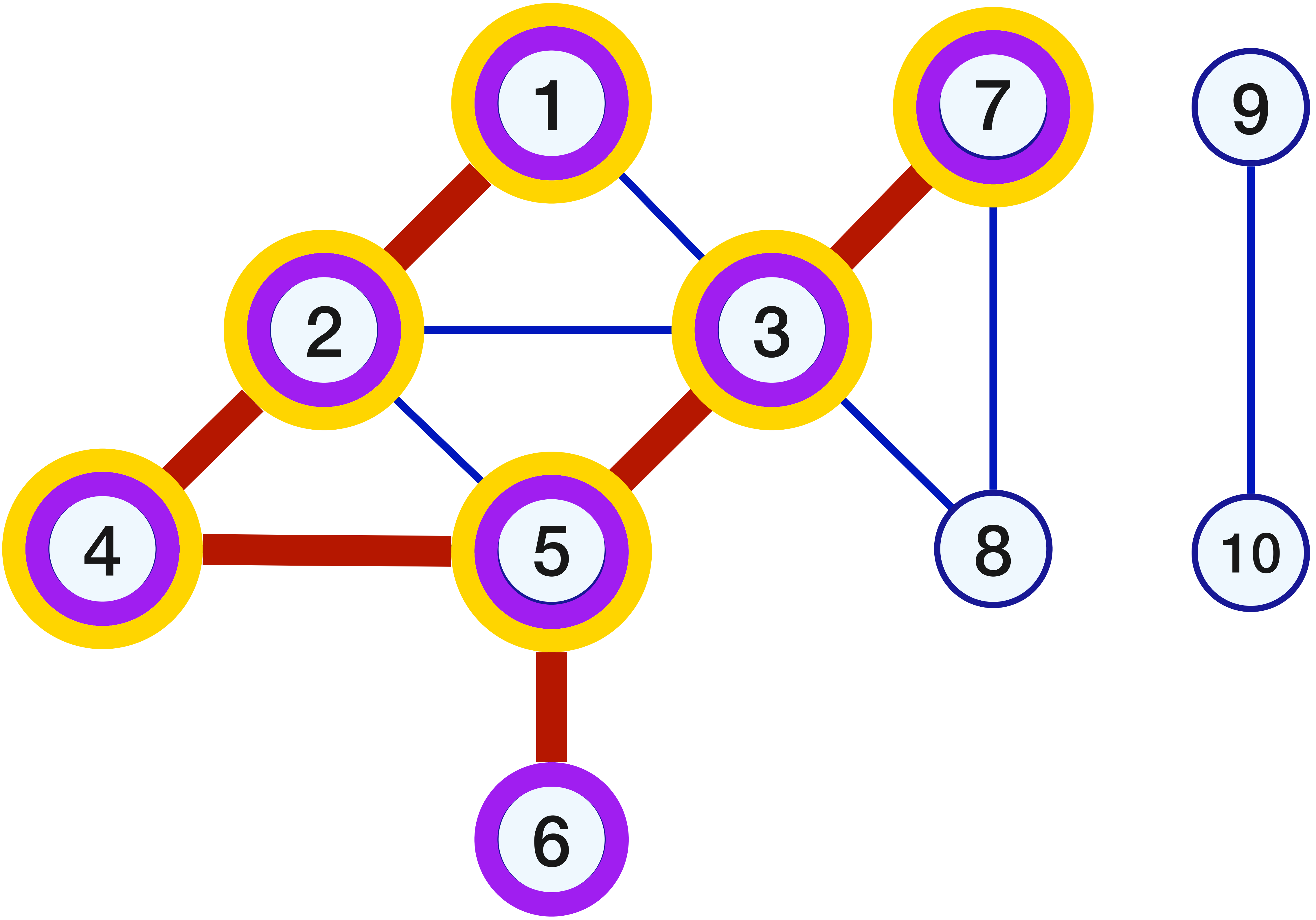
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]



DFS with pre-post numbering

Time = 8

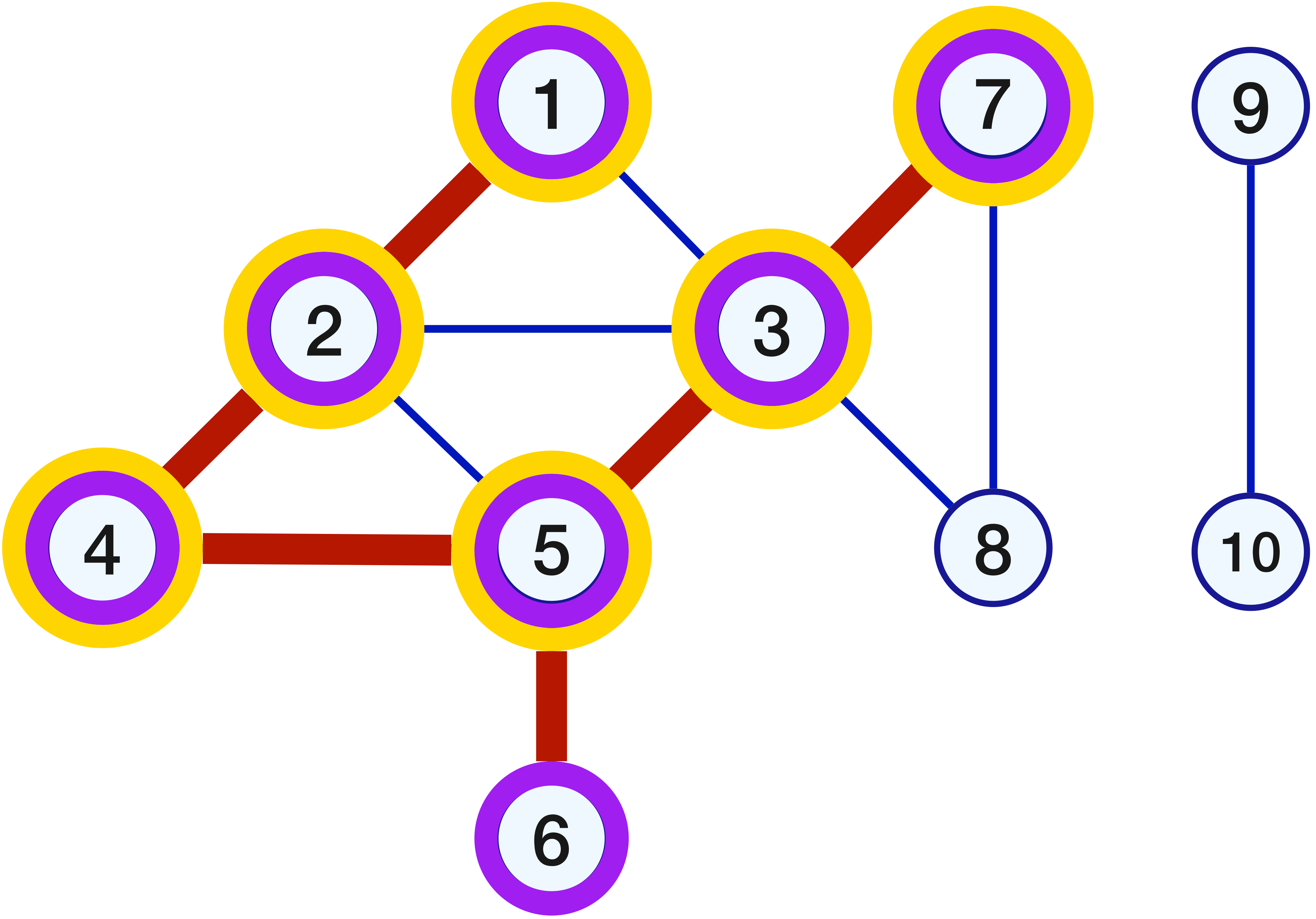
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]



DFS with pre-post numbering

Time = 9

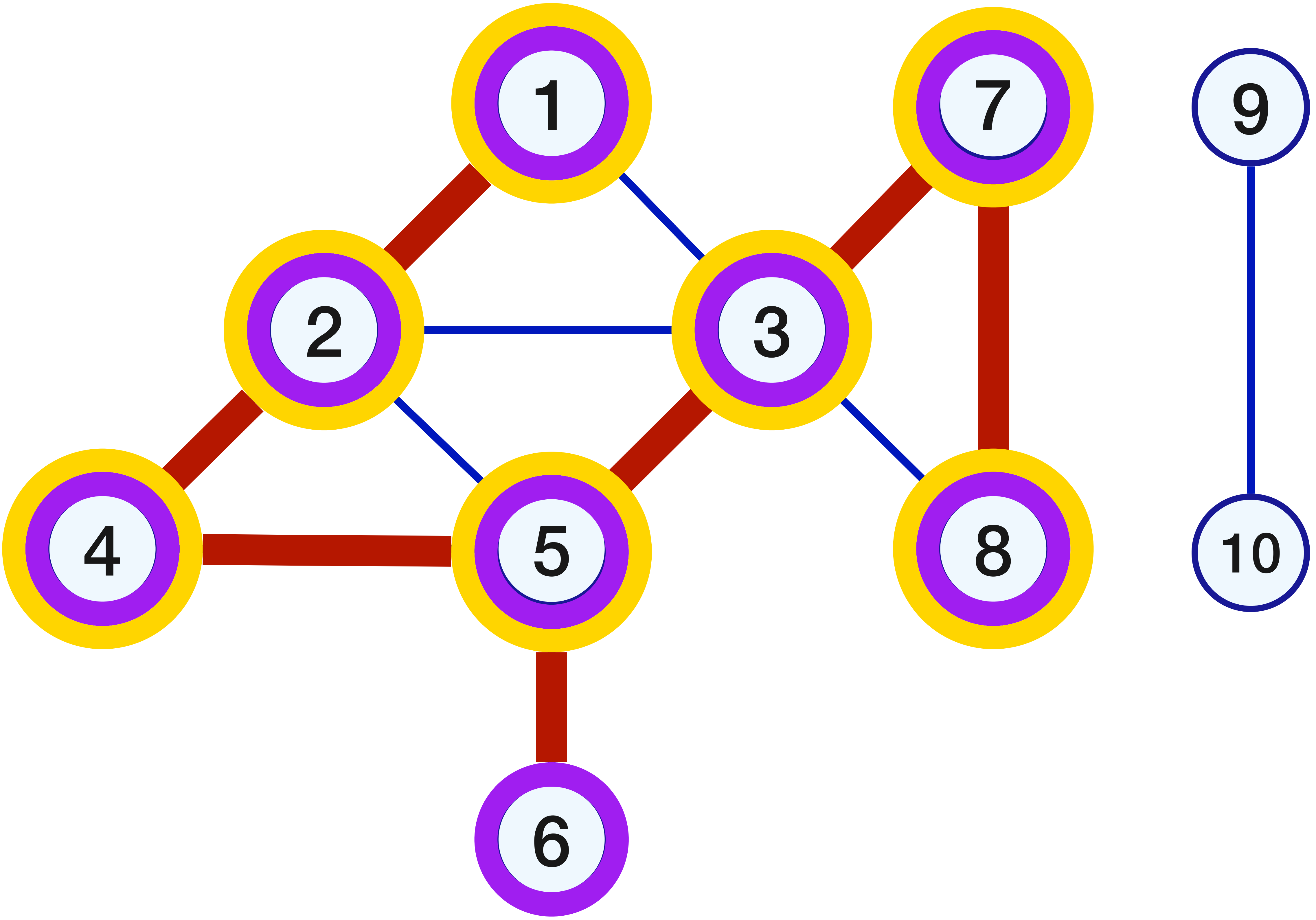
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]



DFS with pre-post numbering

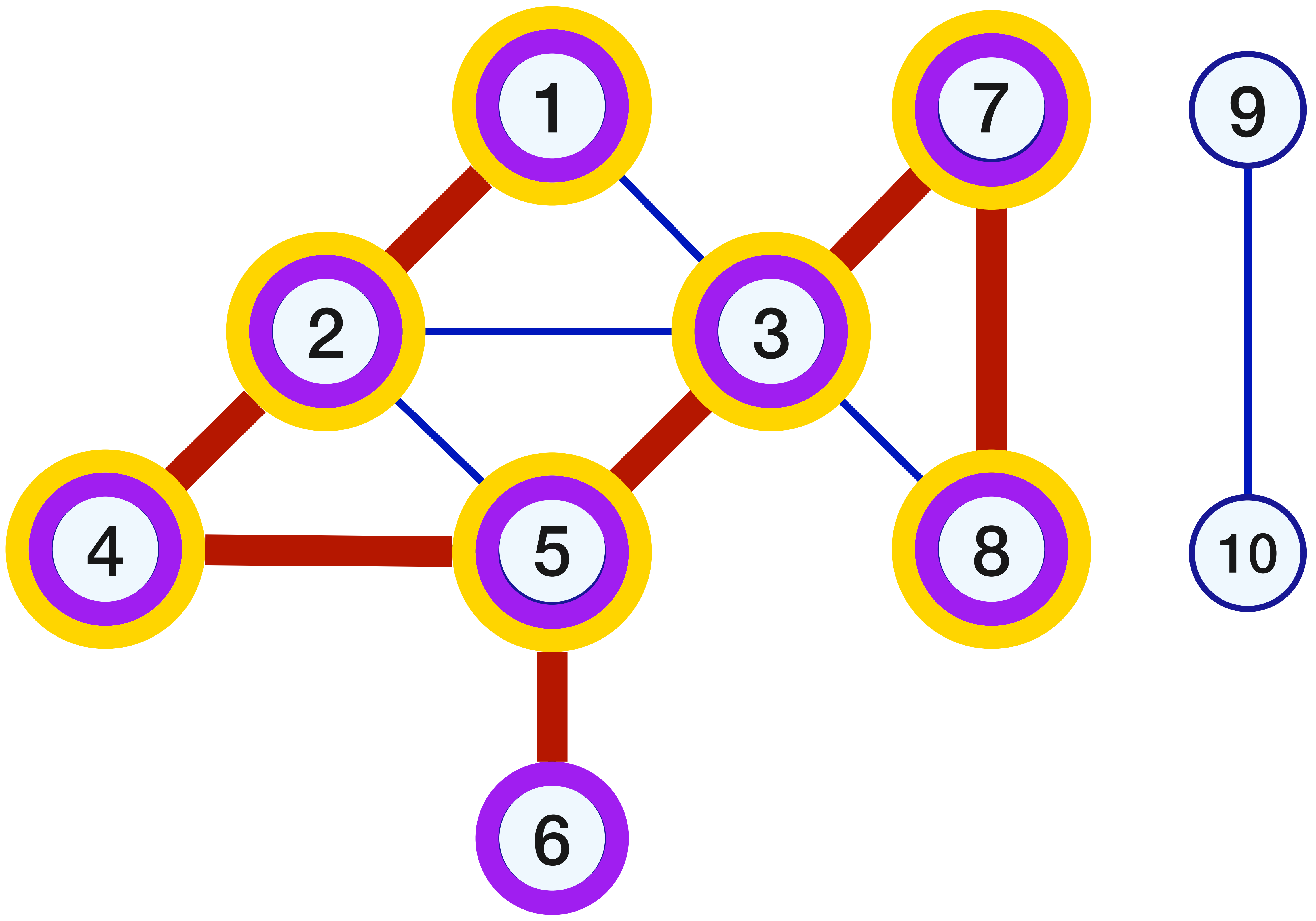
Time = 9

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



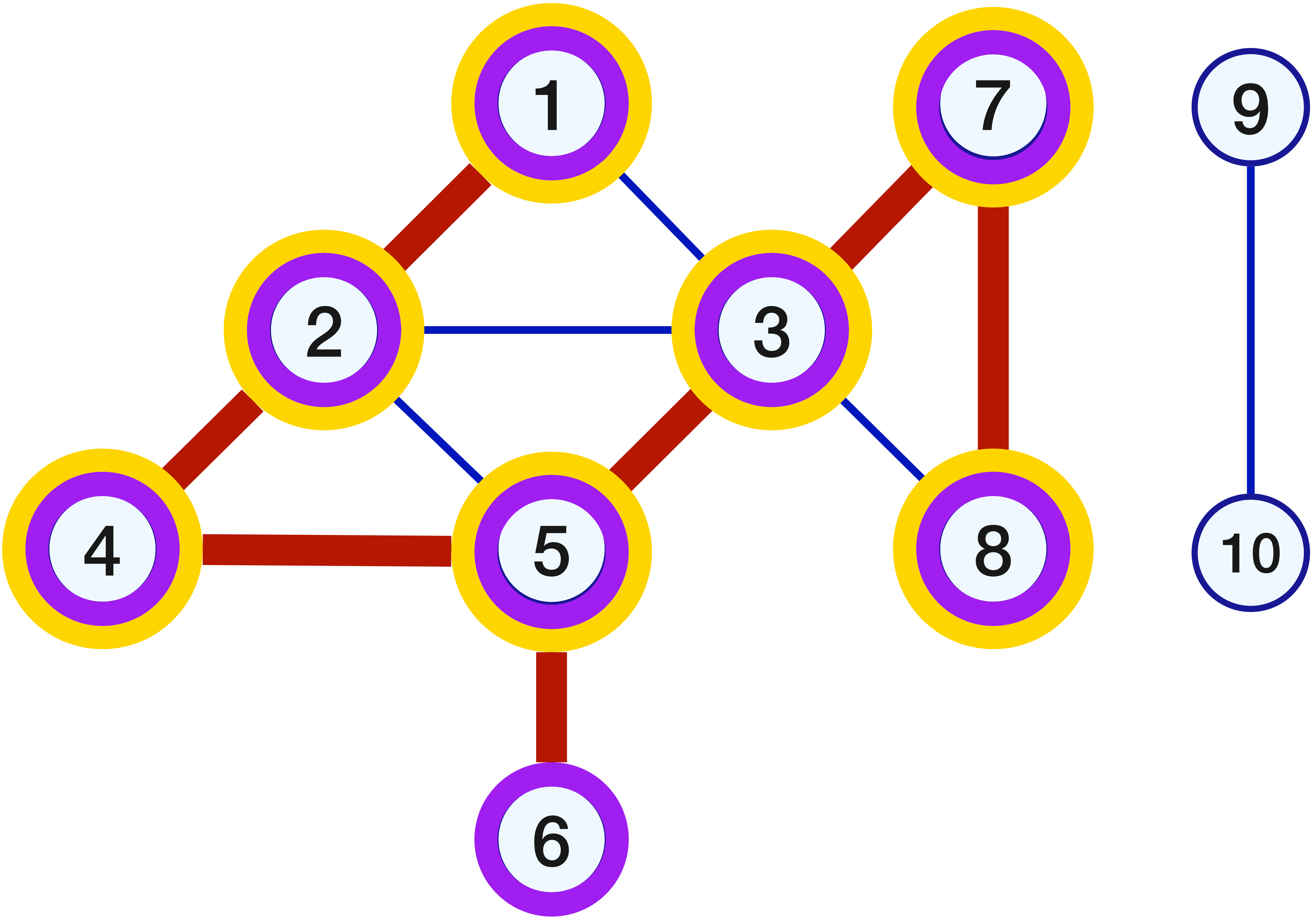
DFS with pre-post numbering

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



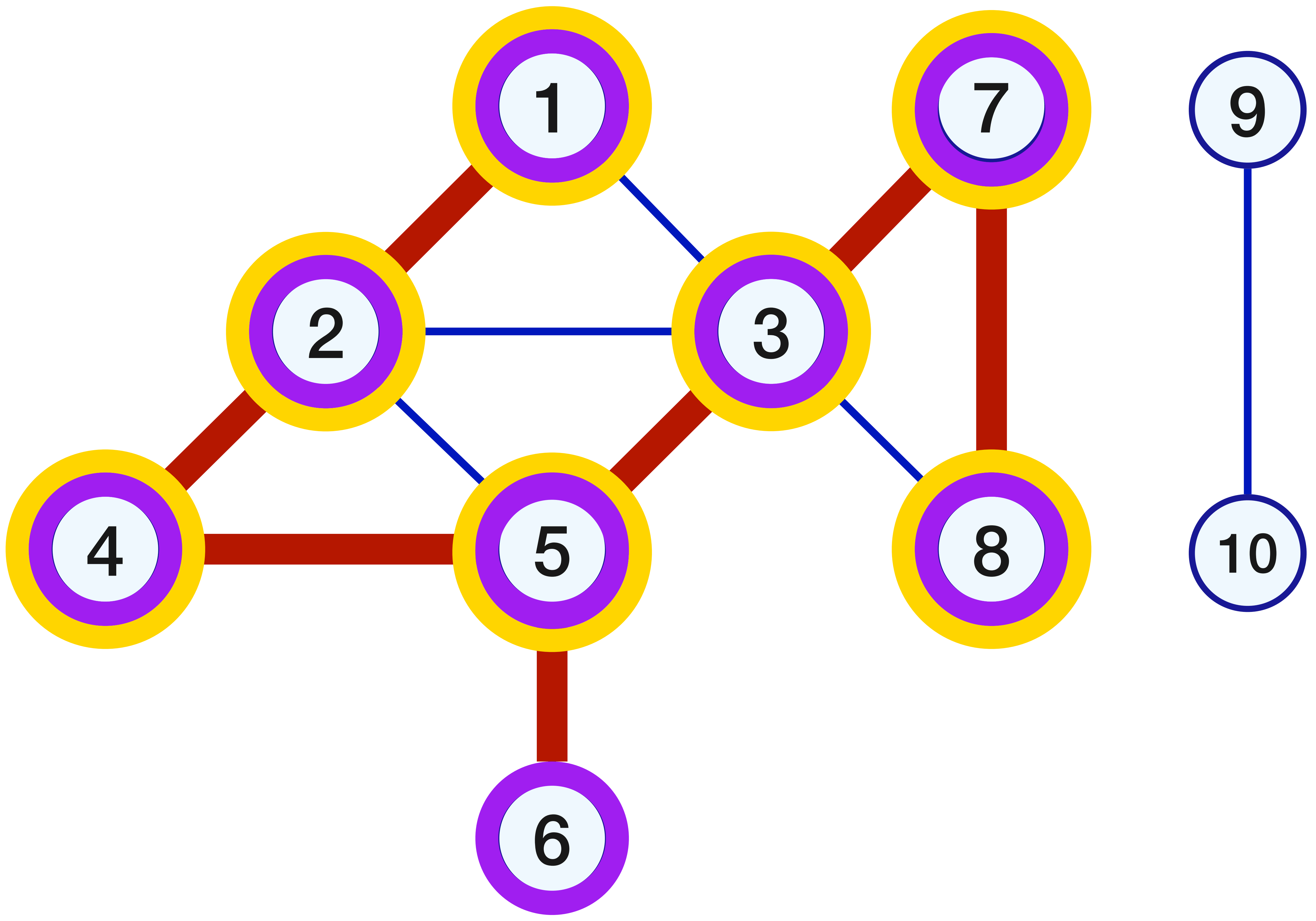
DFS with pre-post numbering

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

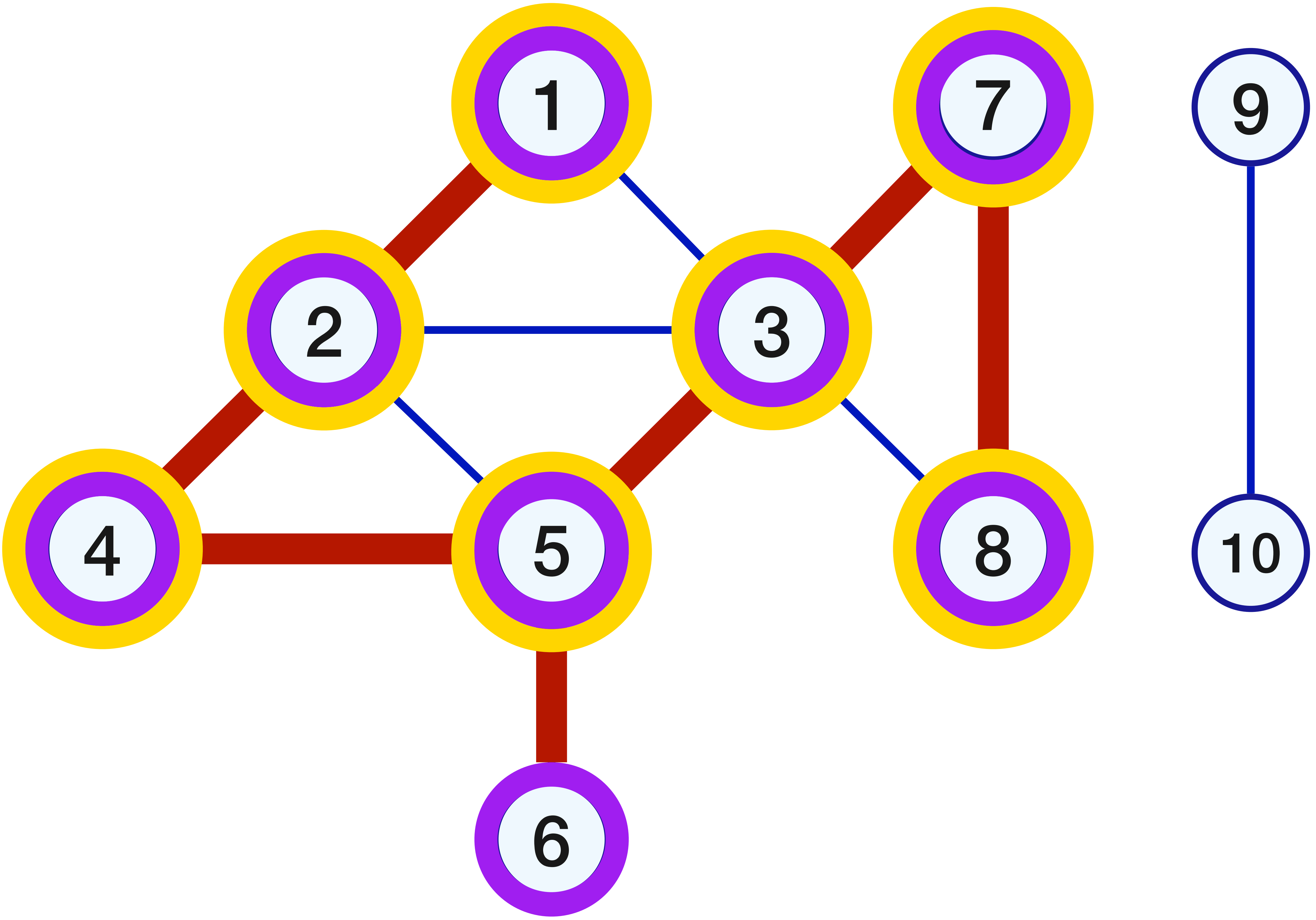
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9,]



DFS with pre-post numbering

Time = 10

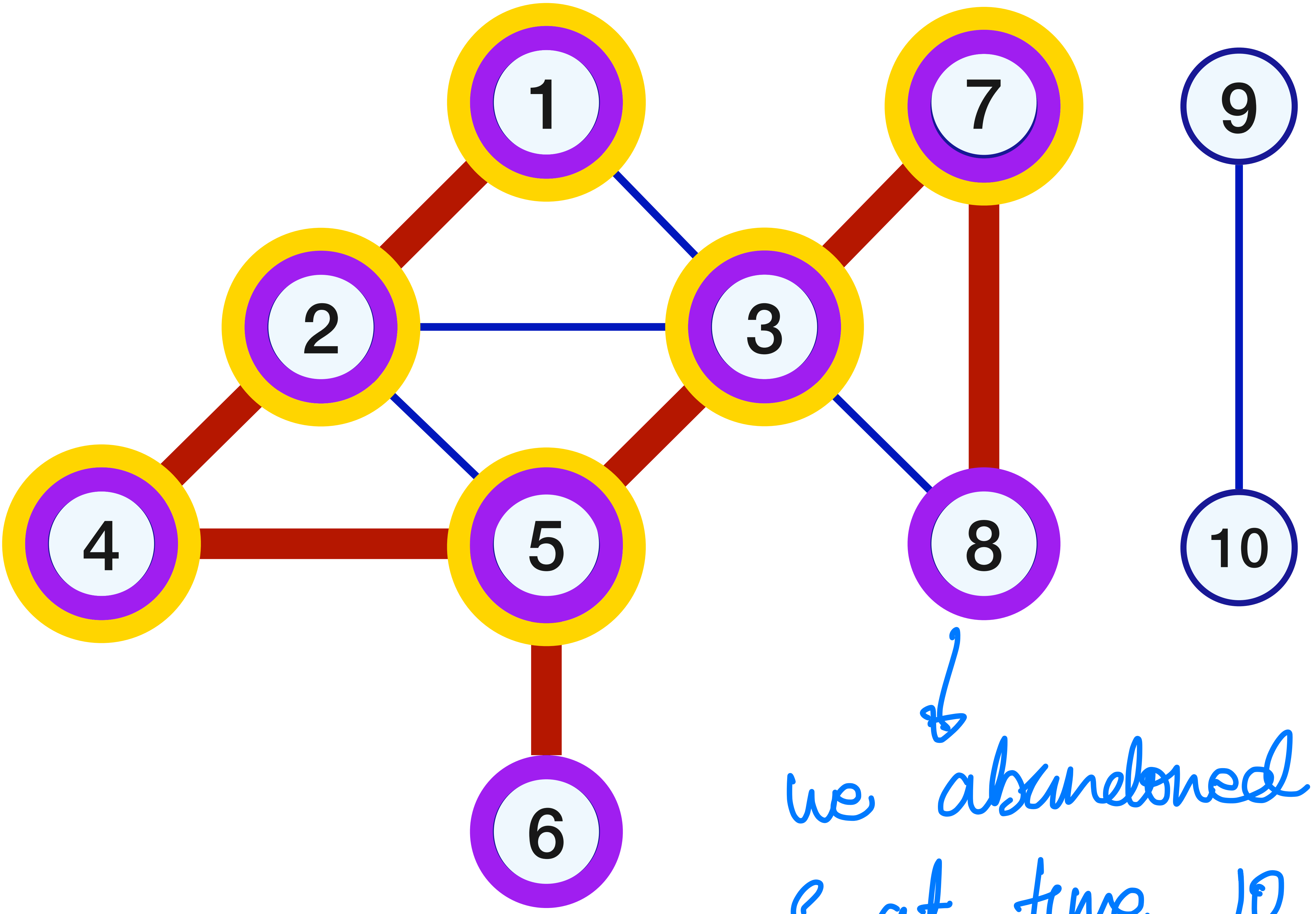
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9,]



DFS with pre-post numbering

Time = 10

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

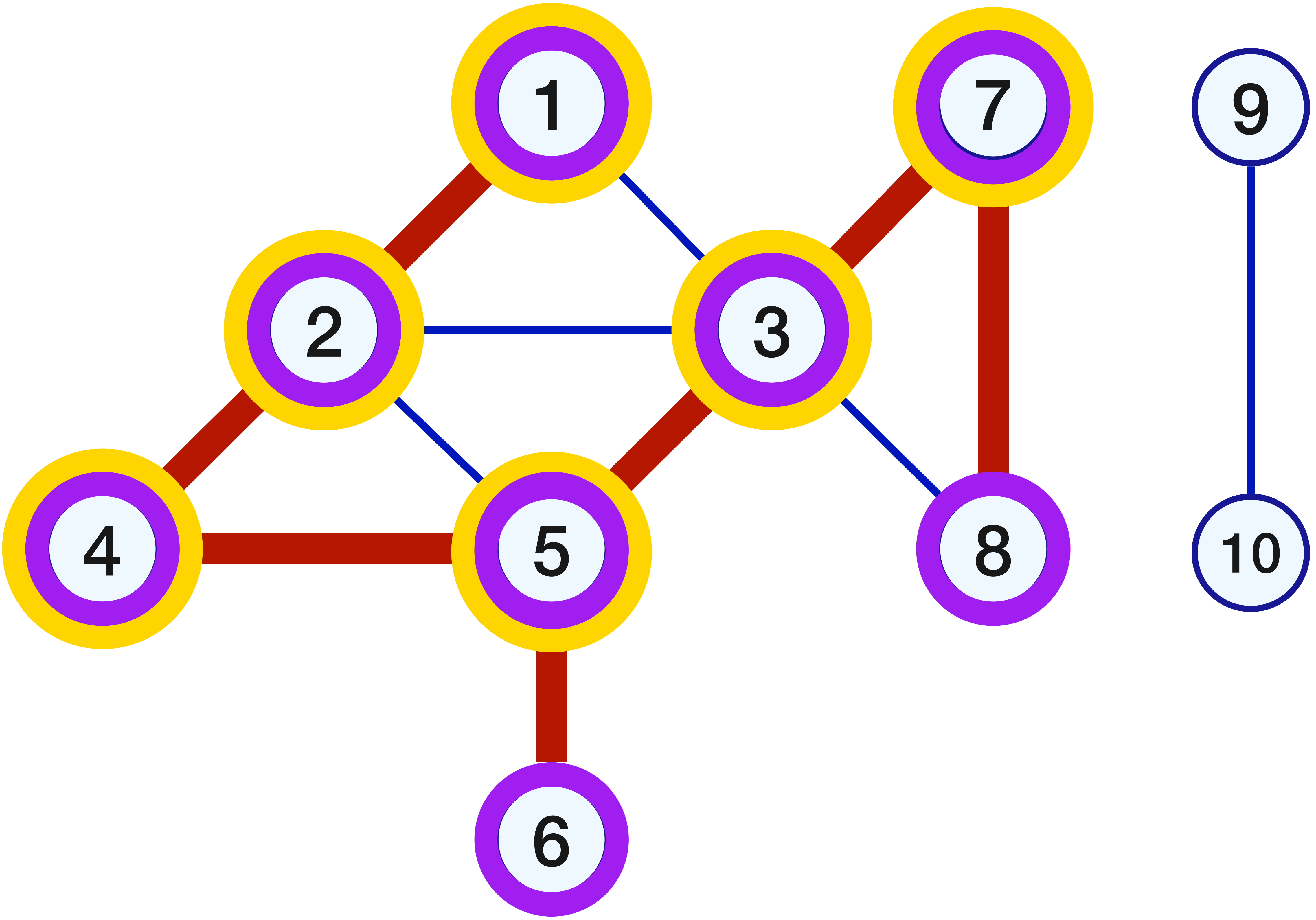


we abandoned 8 at time 10.

DFS with pre-post numbering

Time = 10

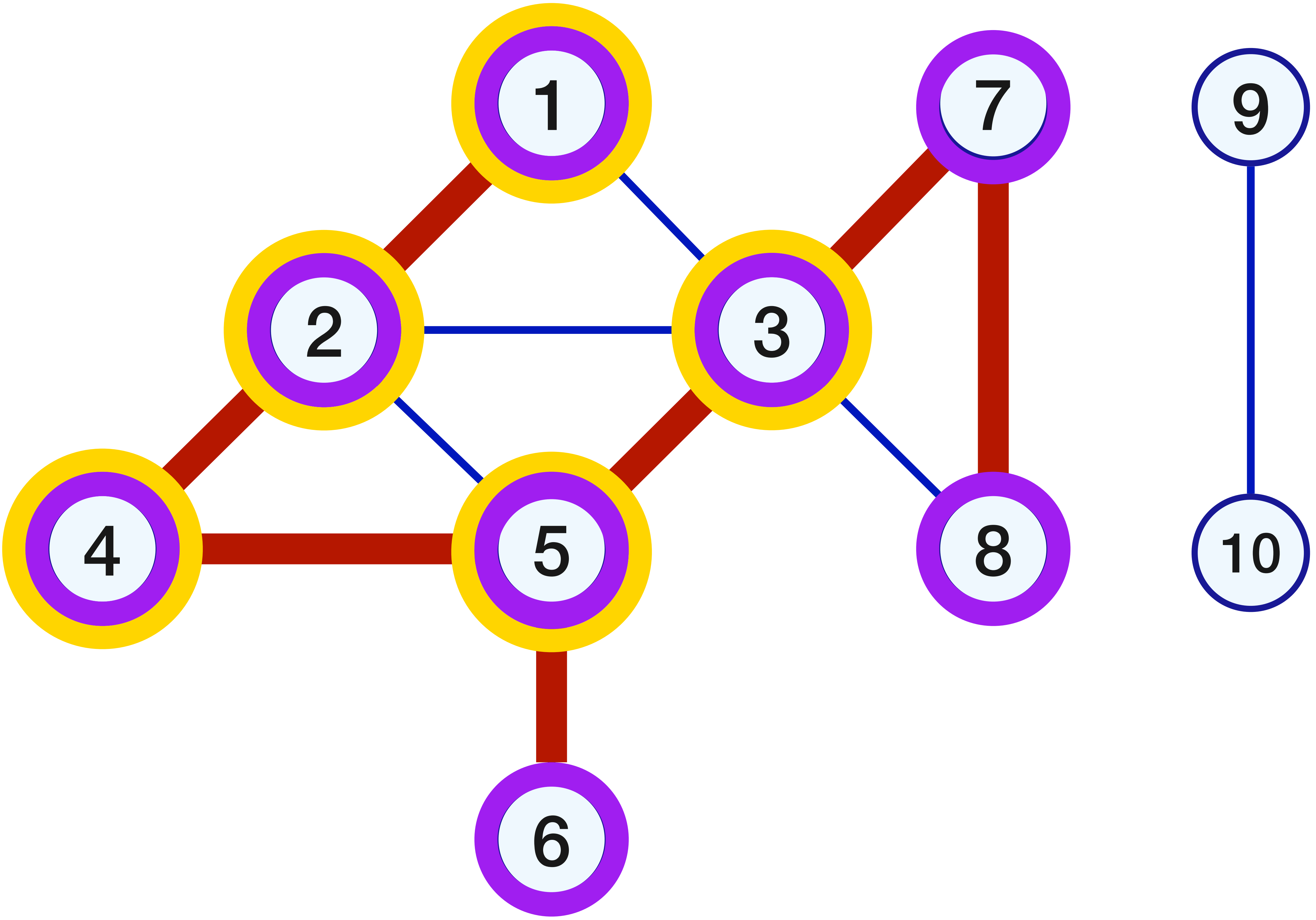
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9, 10]



DFS with pre-post numbering

Time = 11

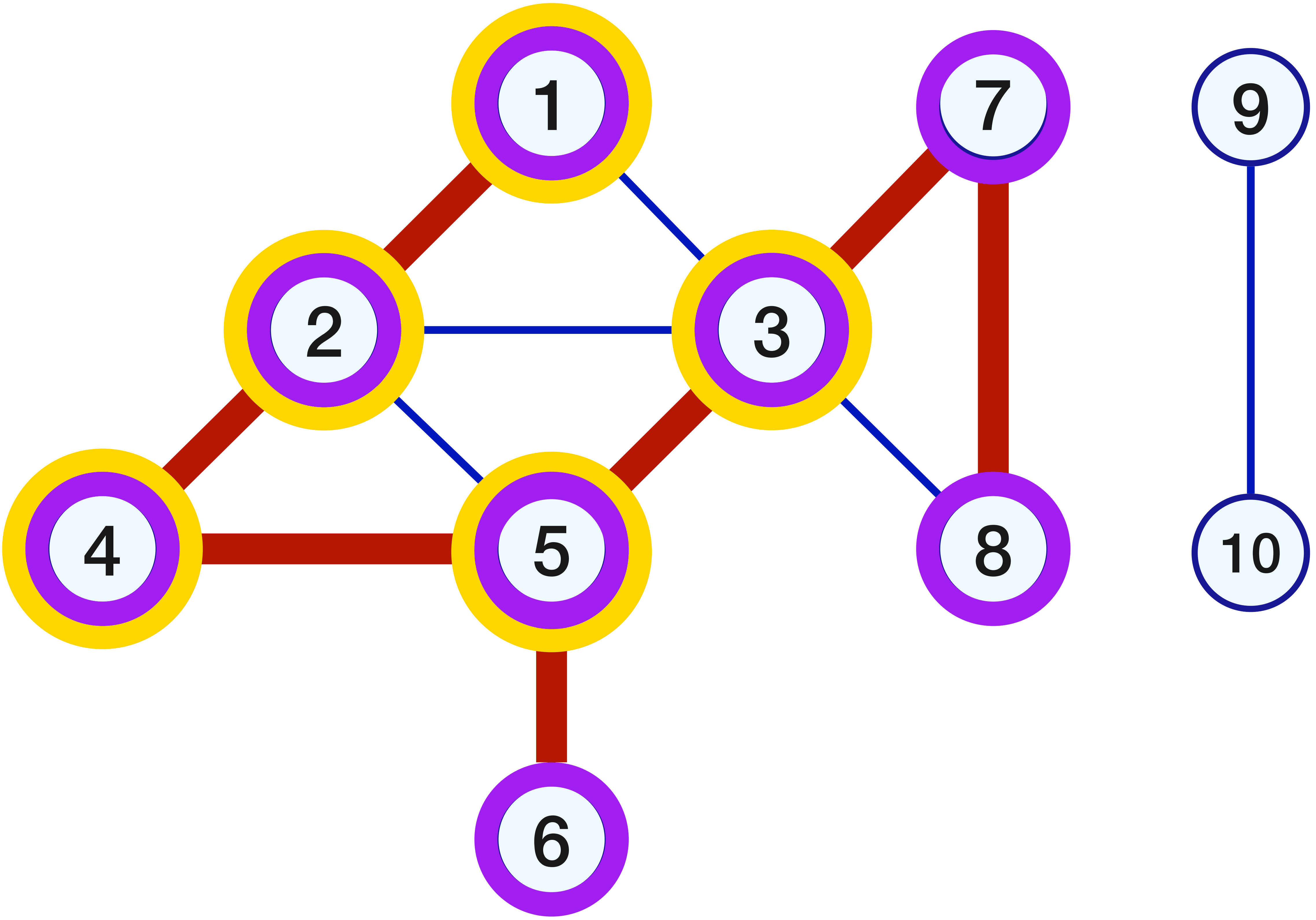
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8,]
8	[9, 10]



DFS with pre-post numbering

Time = 11

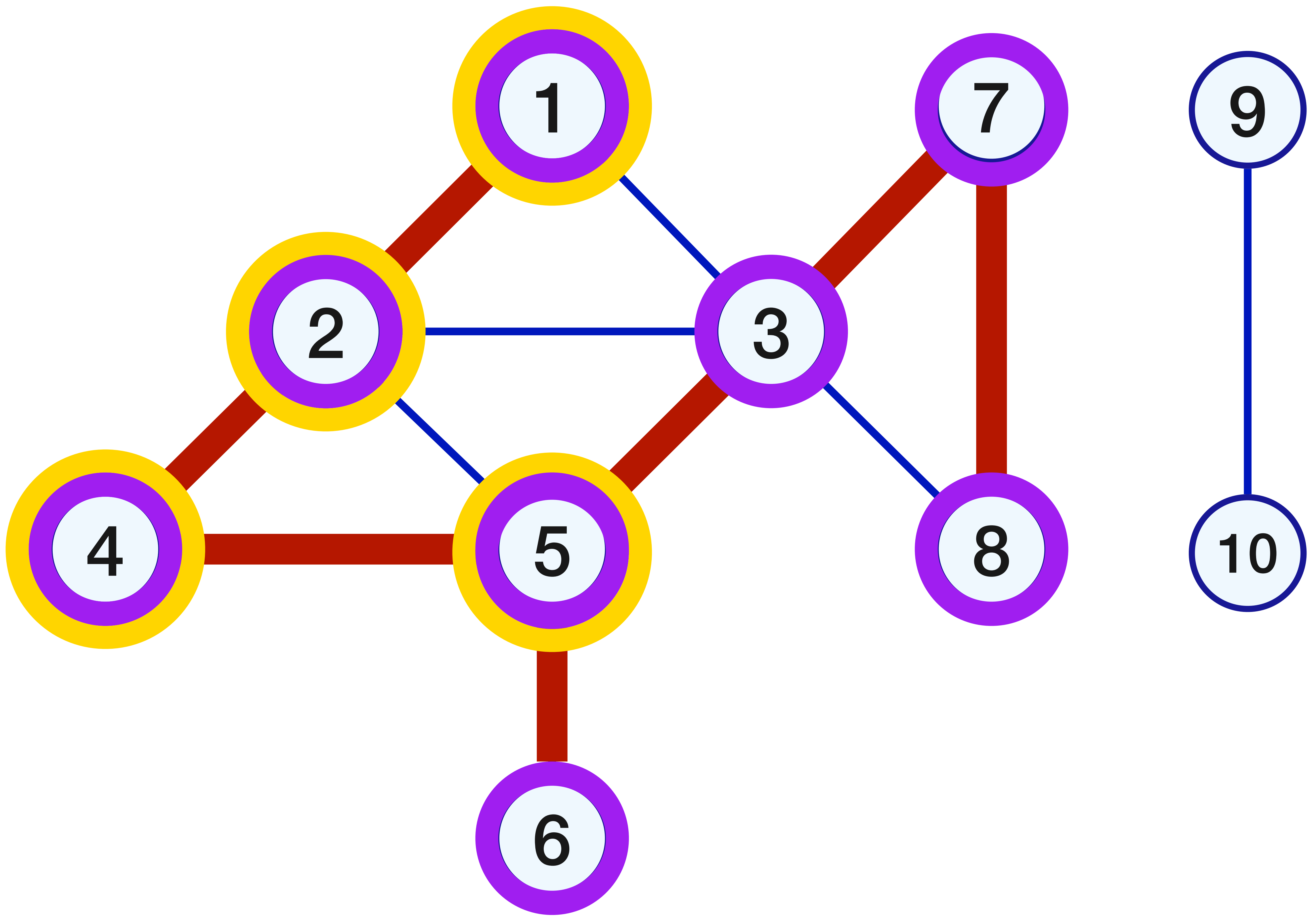
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 12

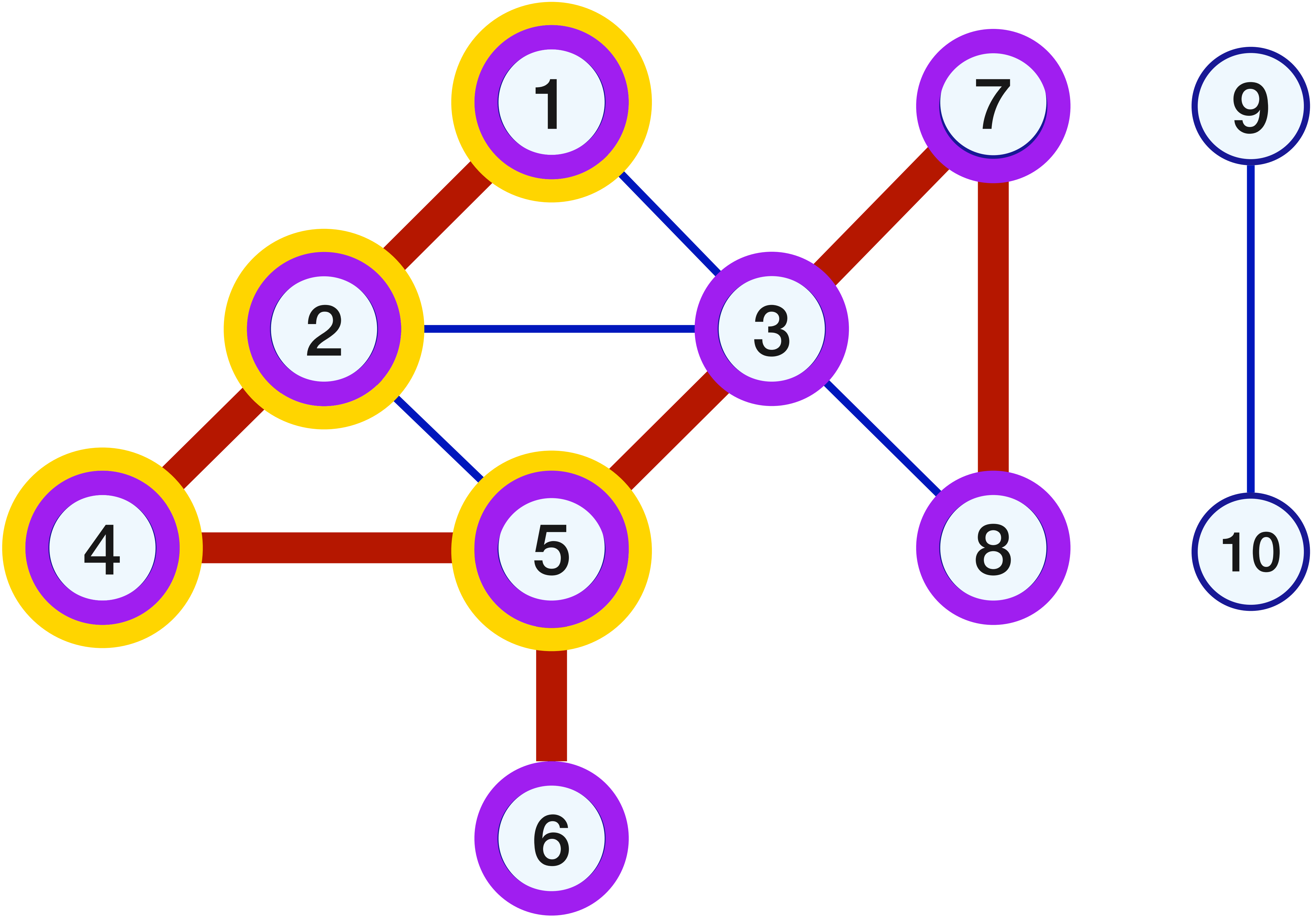
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7,]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 12

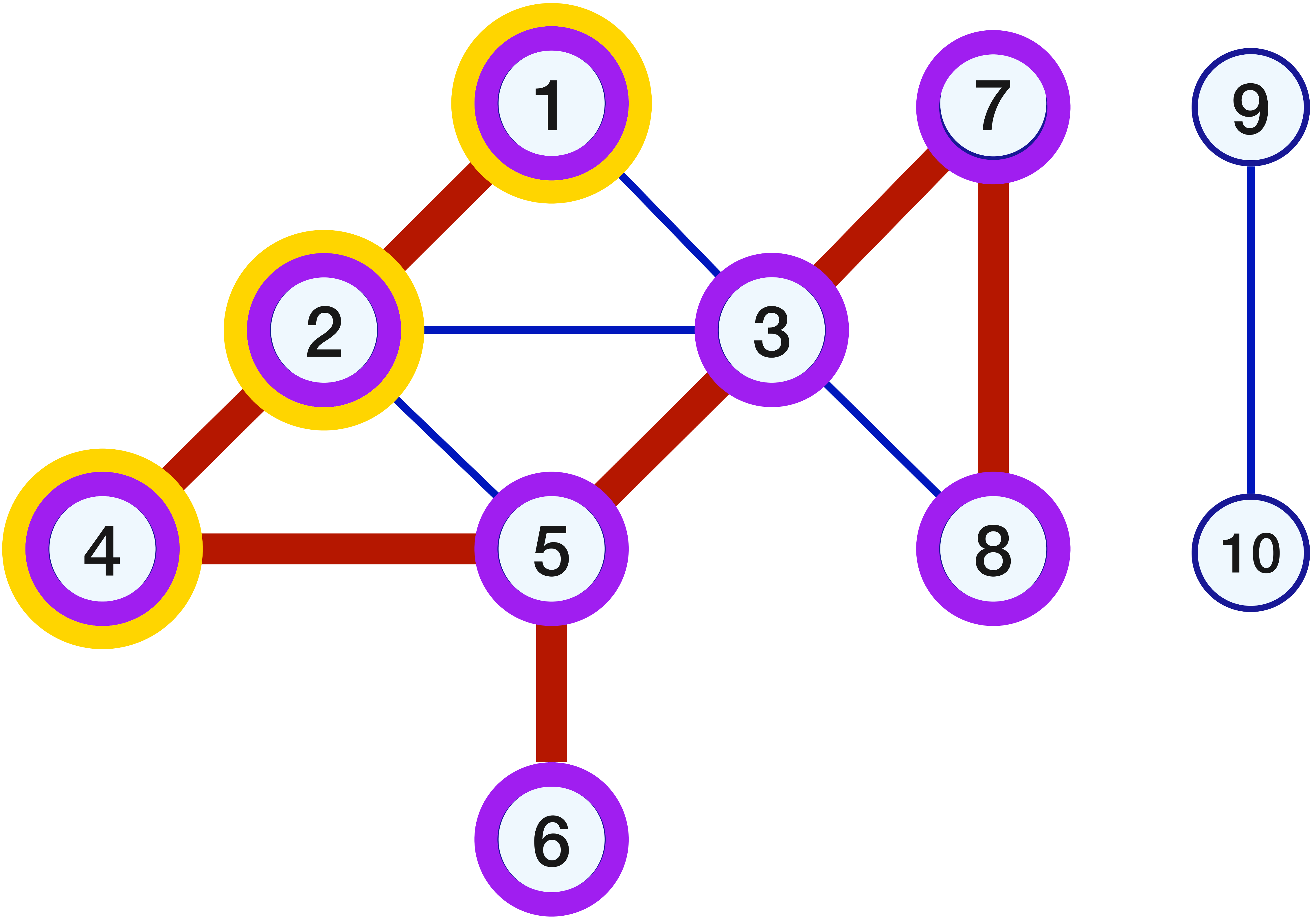
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 13

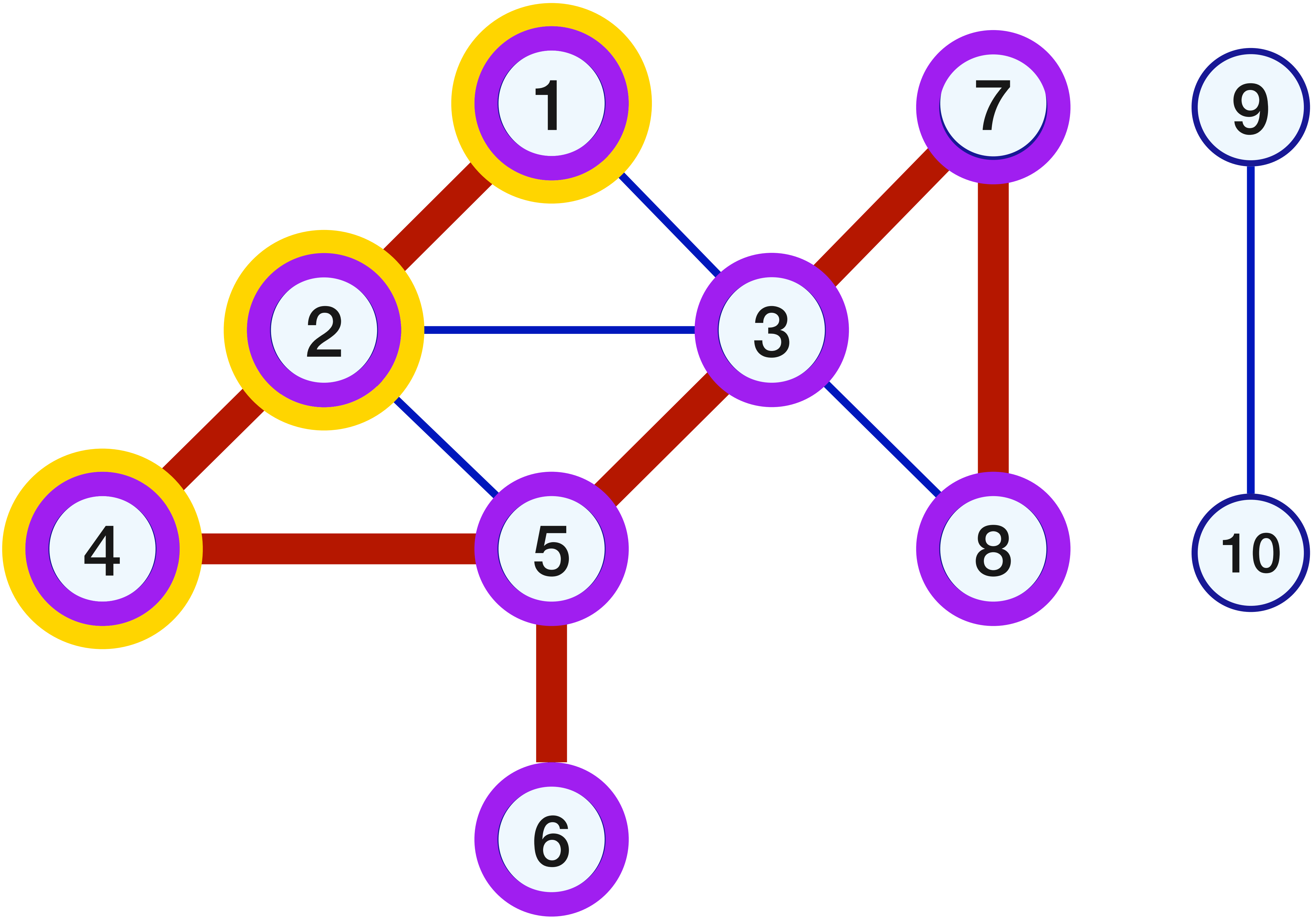
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4,]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 13

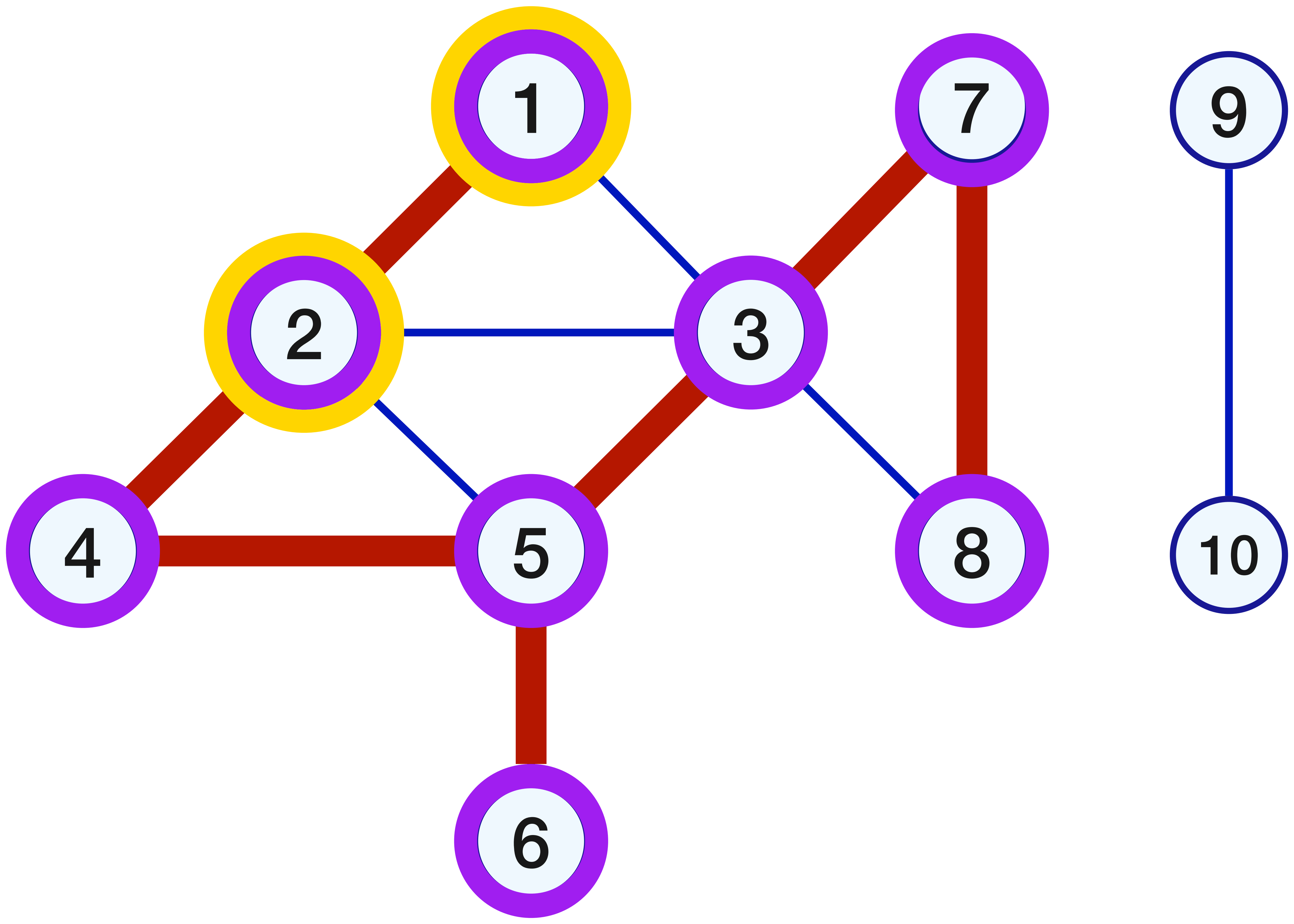
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 14

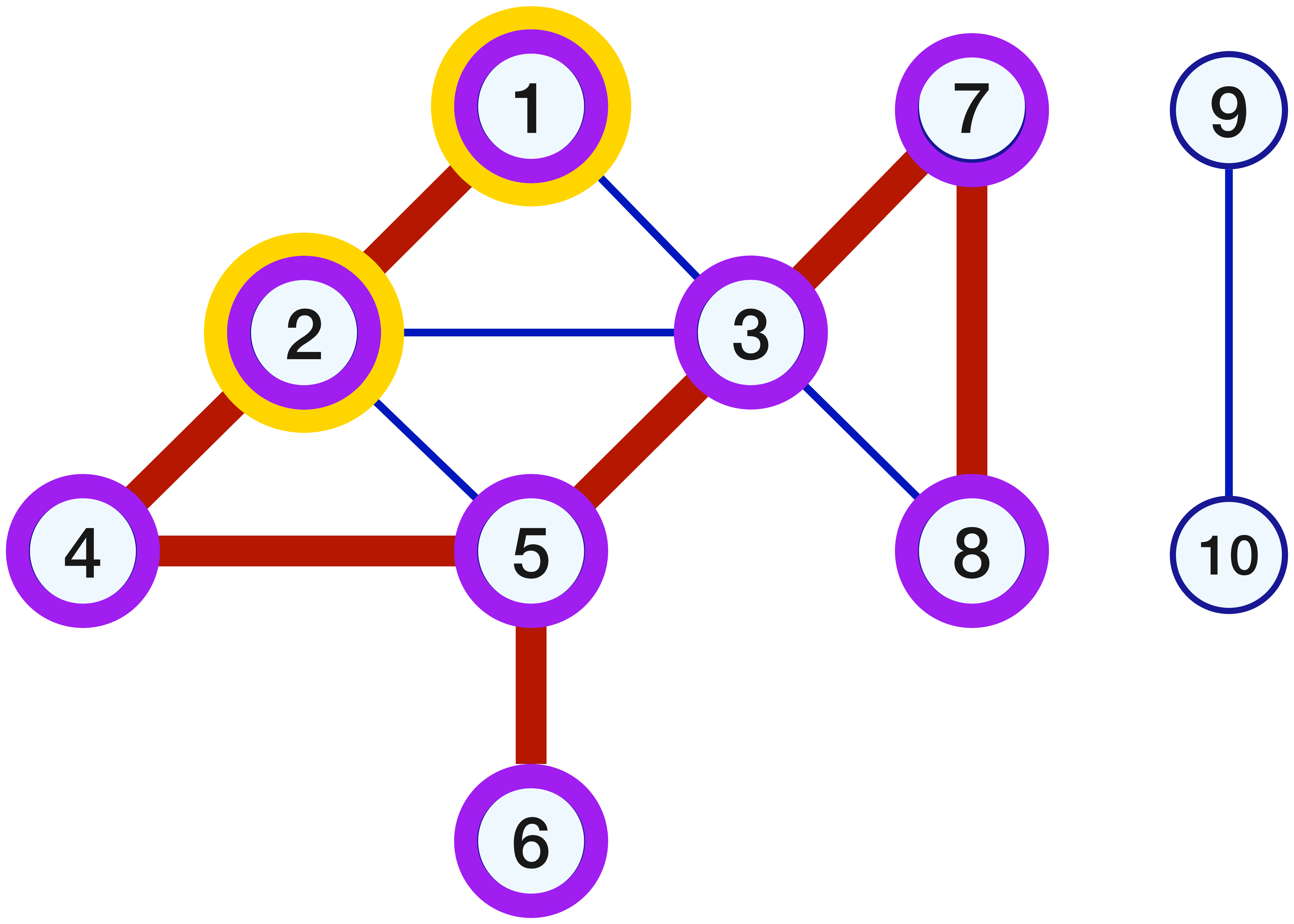
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3,]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 14

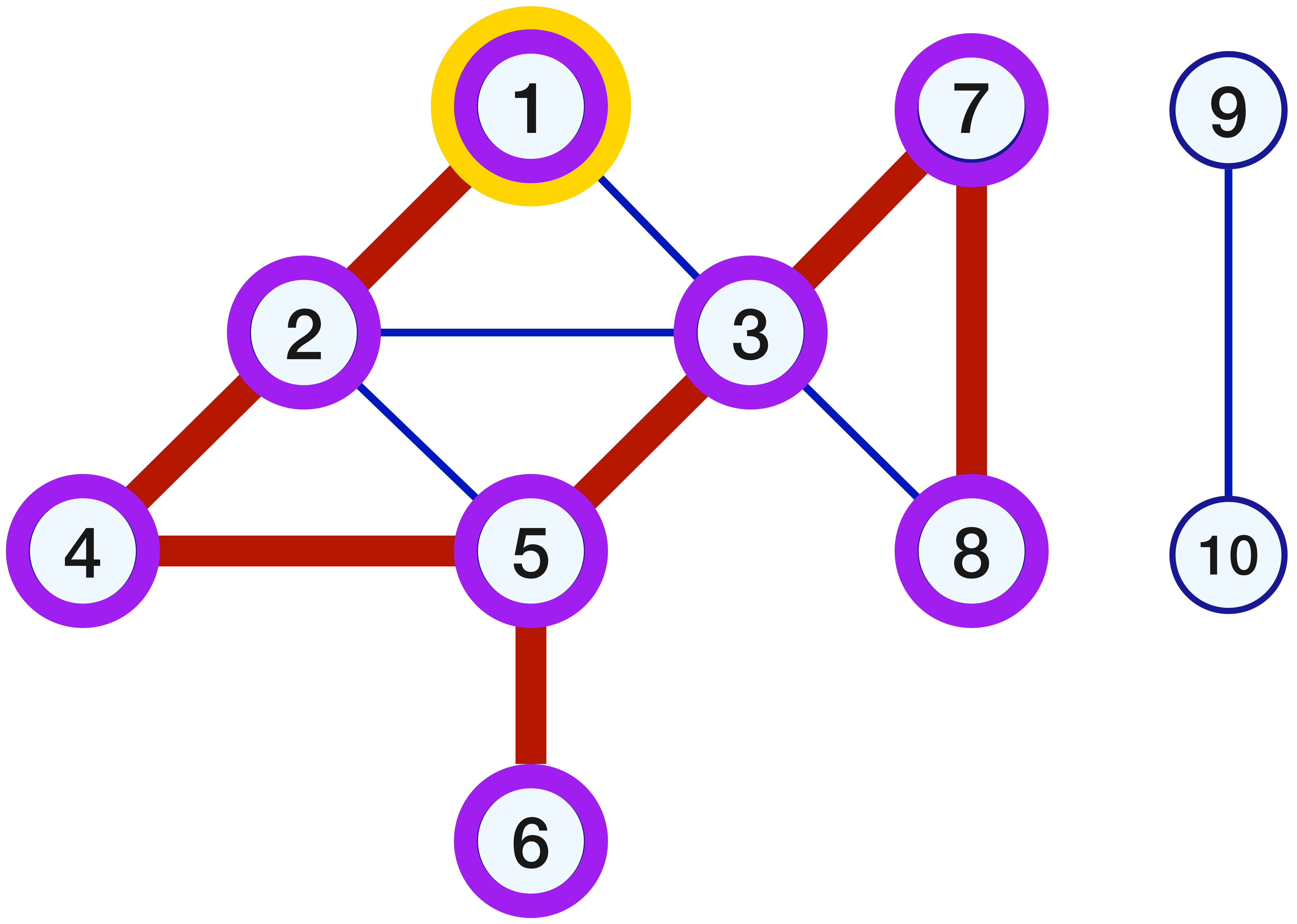
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 15

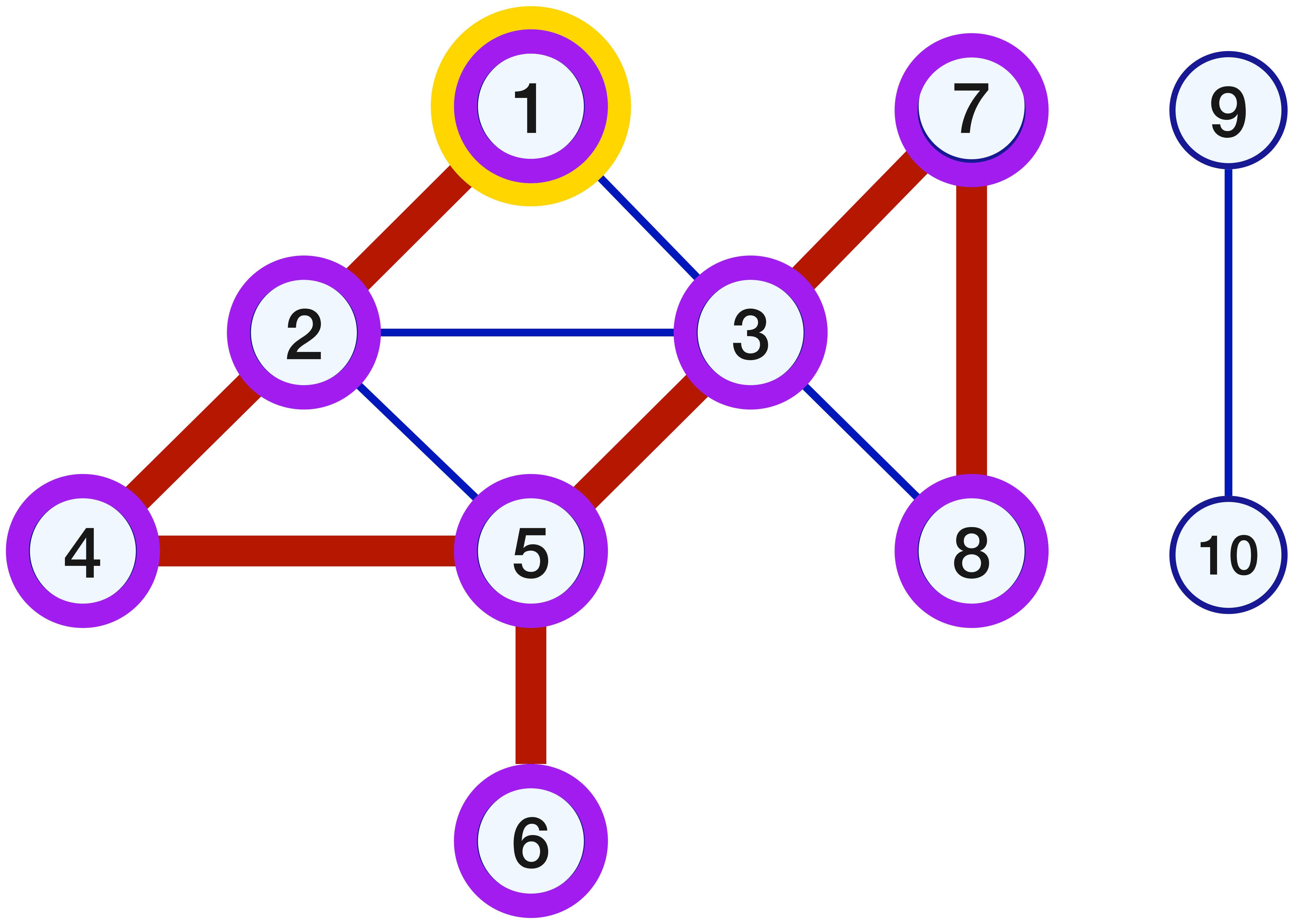
Vertex	*[Pre, Post]*
1	[1,]
2	[2,]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

Time = 15

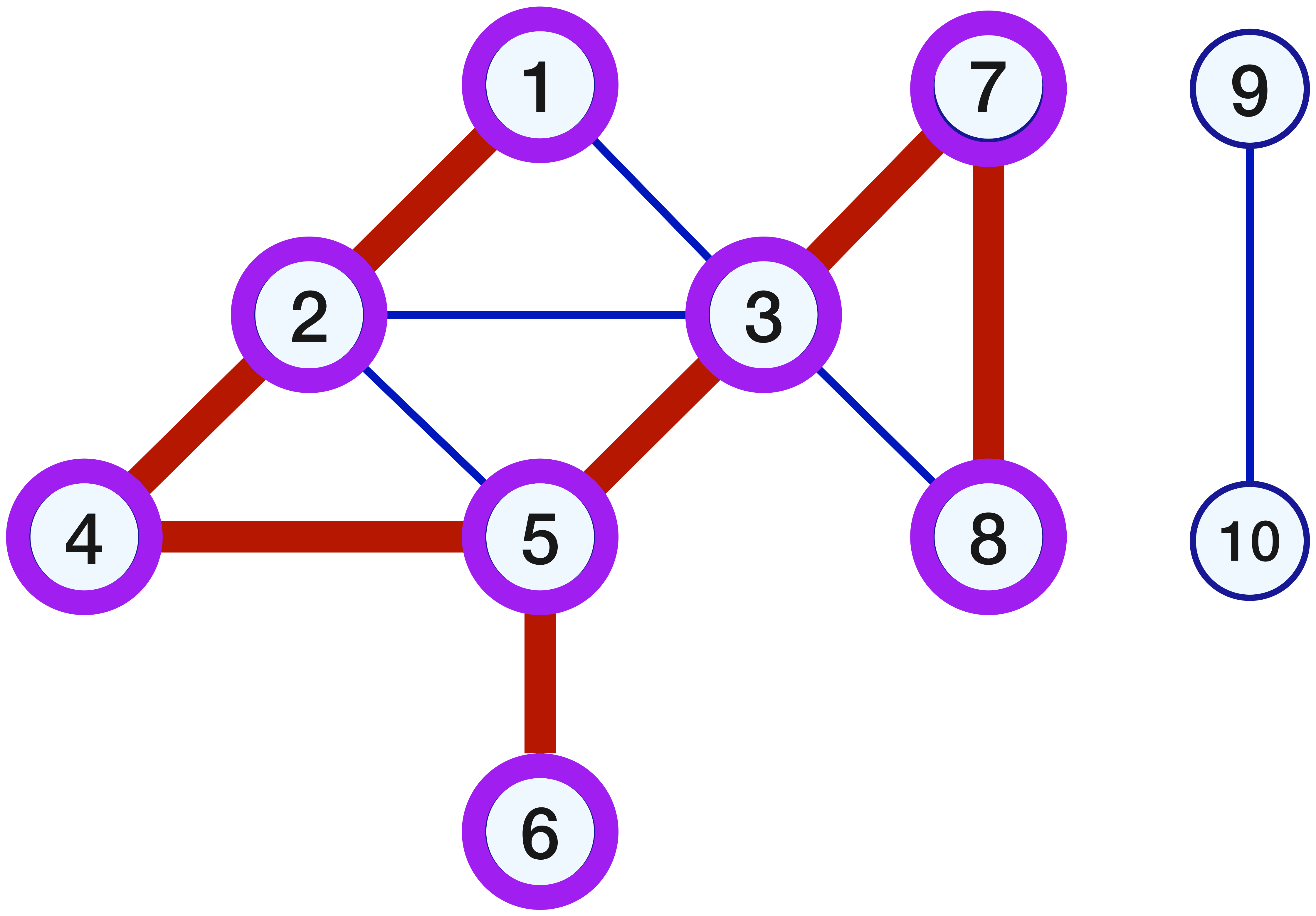
Vertex	*[Pre, Post]*
1	[1,]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



DFS with pre-post numbering

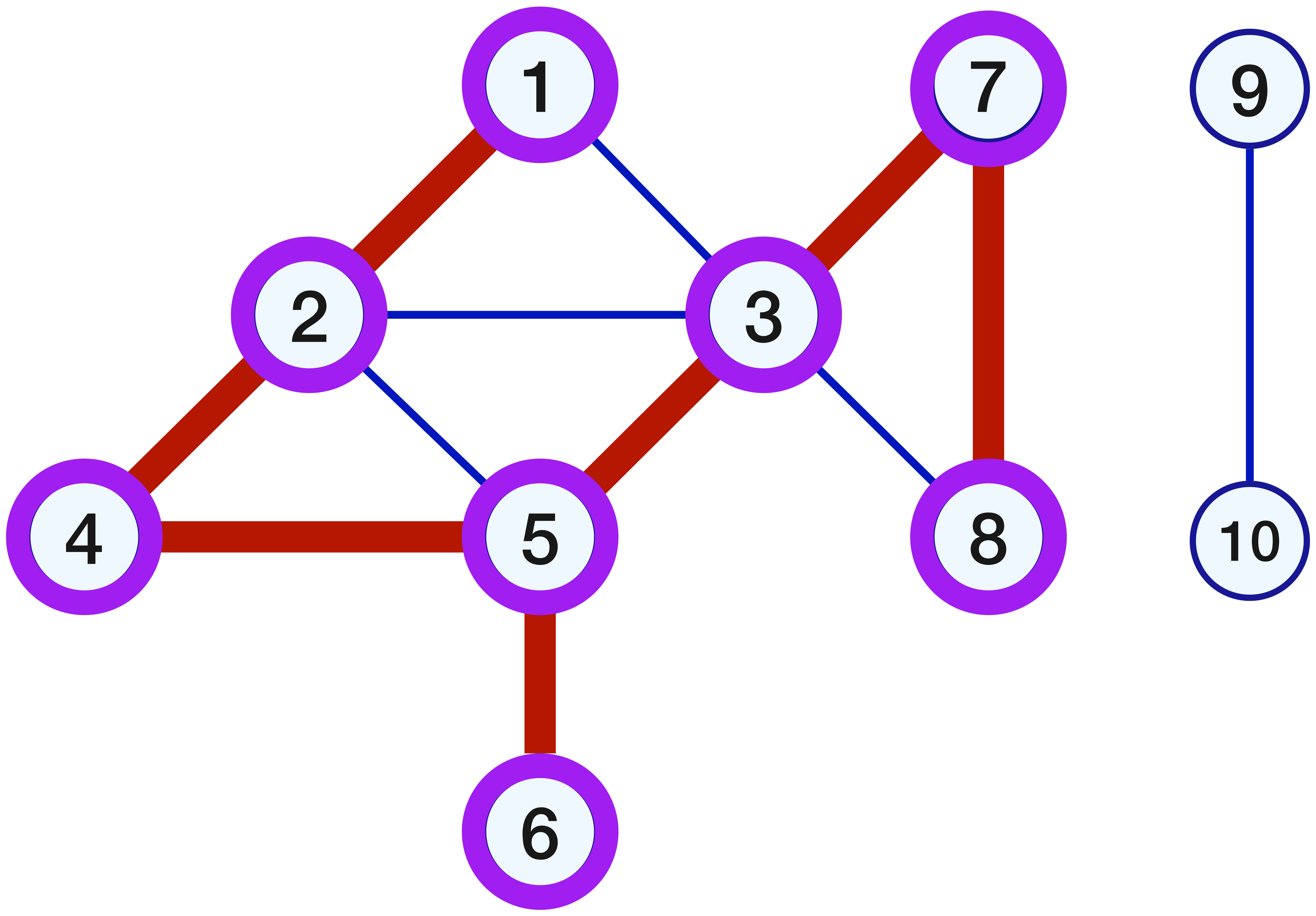
Time = 16

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



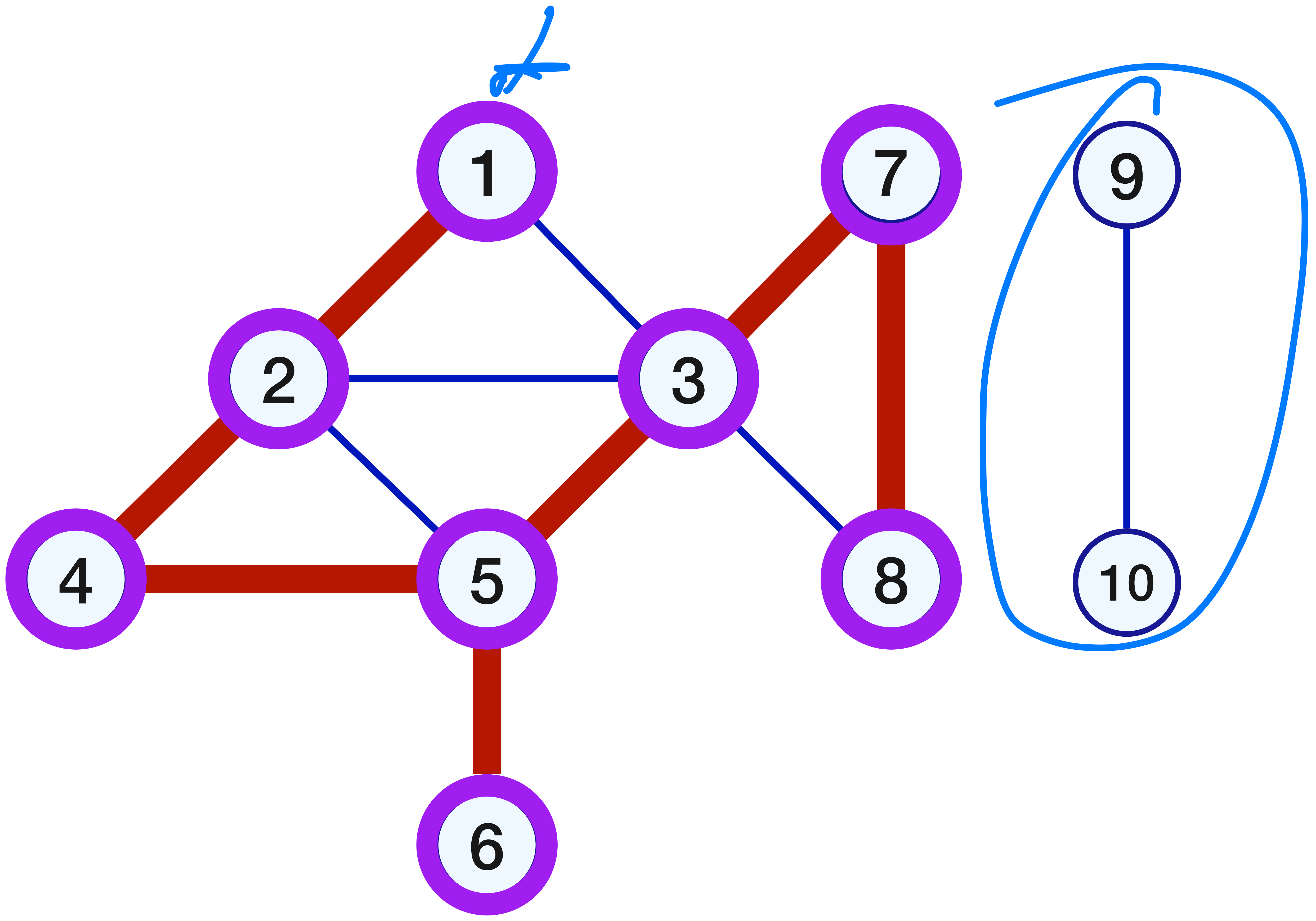
DFS with pre-post numbering

Vertex	*[Pre, Post]*
1	[1,]
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

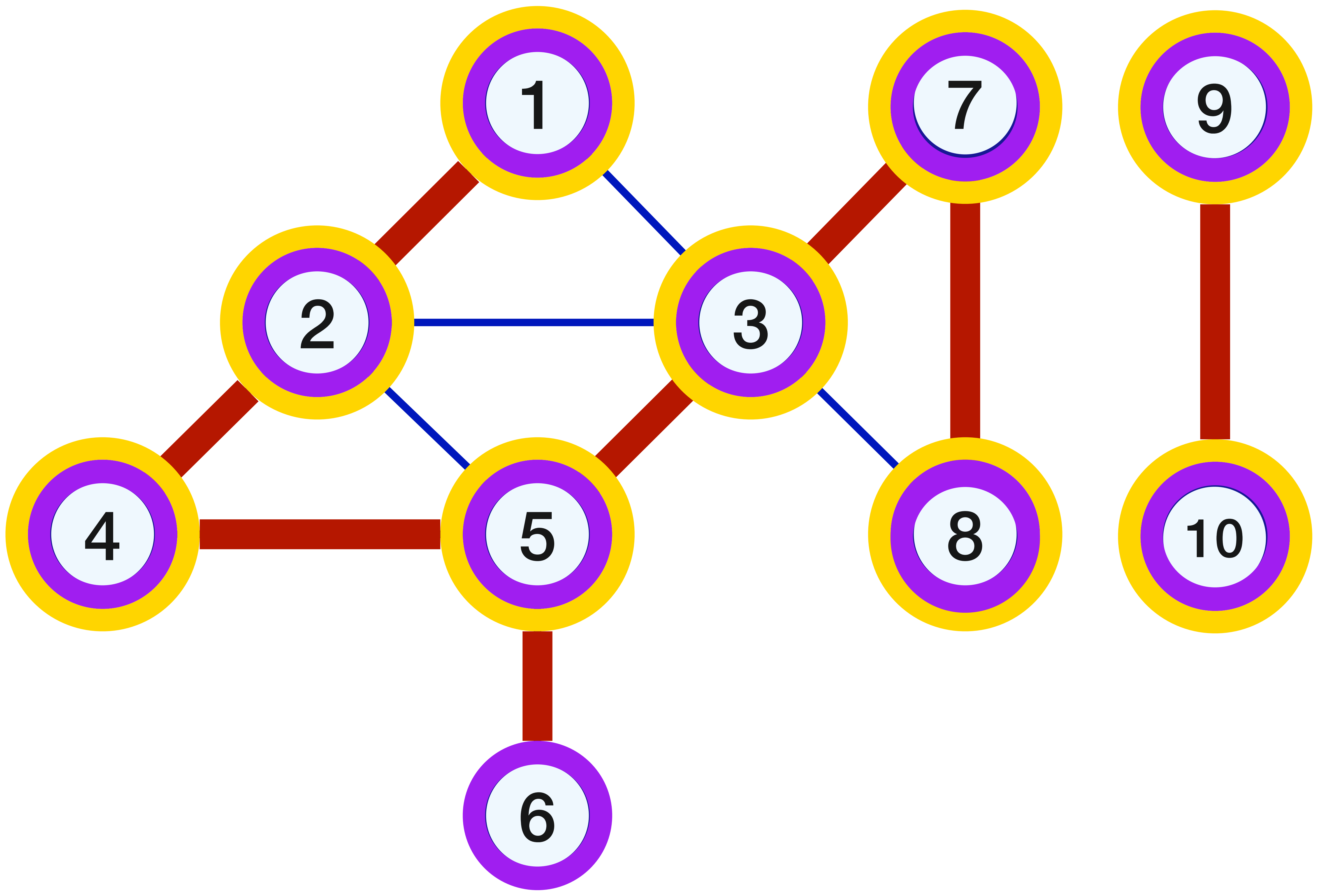
Vertex	*[Pre, Post]*
1	[1, 16]
2	[2, 15]
4	[3, 14]
5	[4, 13]
6	[5, 6]
3	[7, 12]
7	[8, 11]
8	[9, 10]



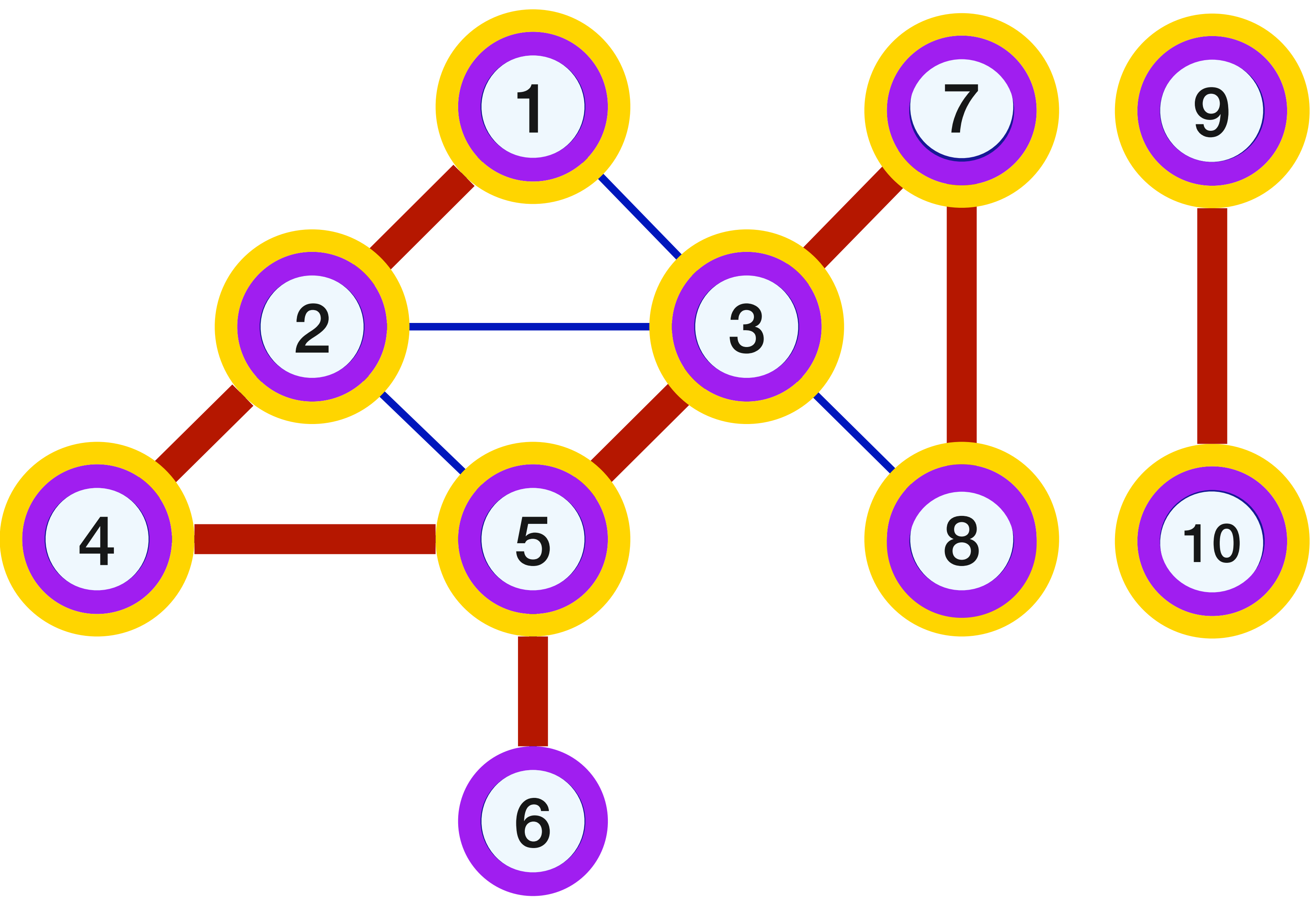
DFS with pre-post numbering

Time = 20 (skipped a few steps)

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]
9	[17, 20]
10	[18, 19]



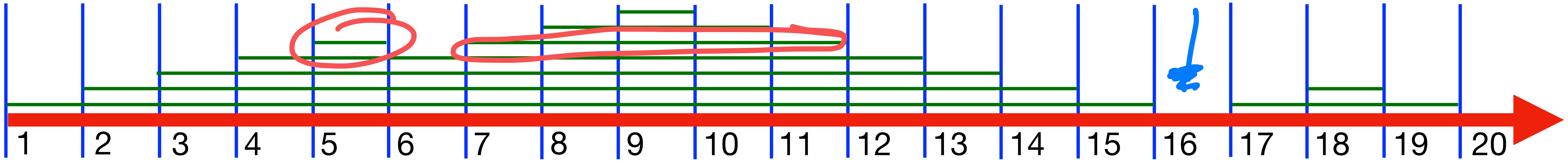
DFS with pre-post numbering



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]
9	[17, 20]
10	[18, 19]

These timestamps can also be thought of as the time interval at which the call DFS(v) was on the runtime stack (ECE 220)

One way to visualize the time stamps

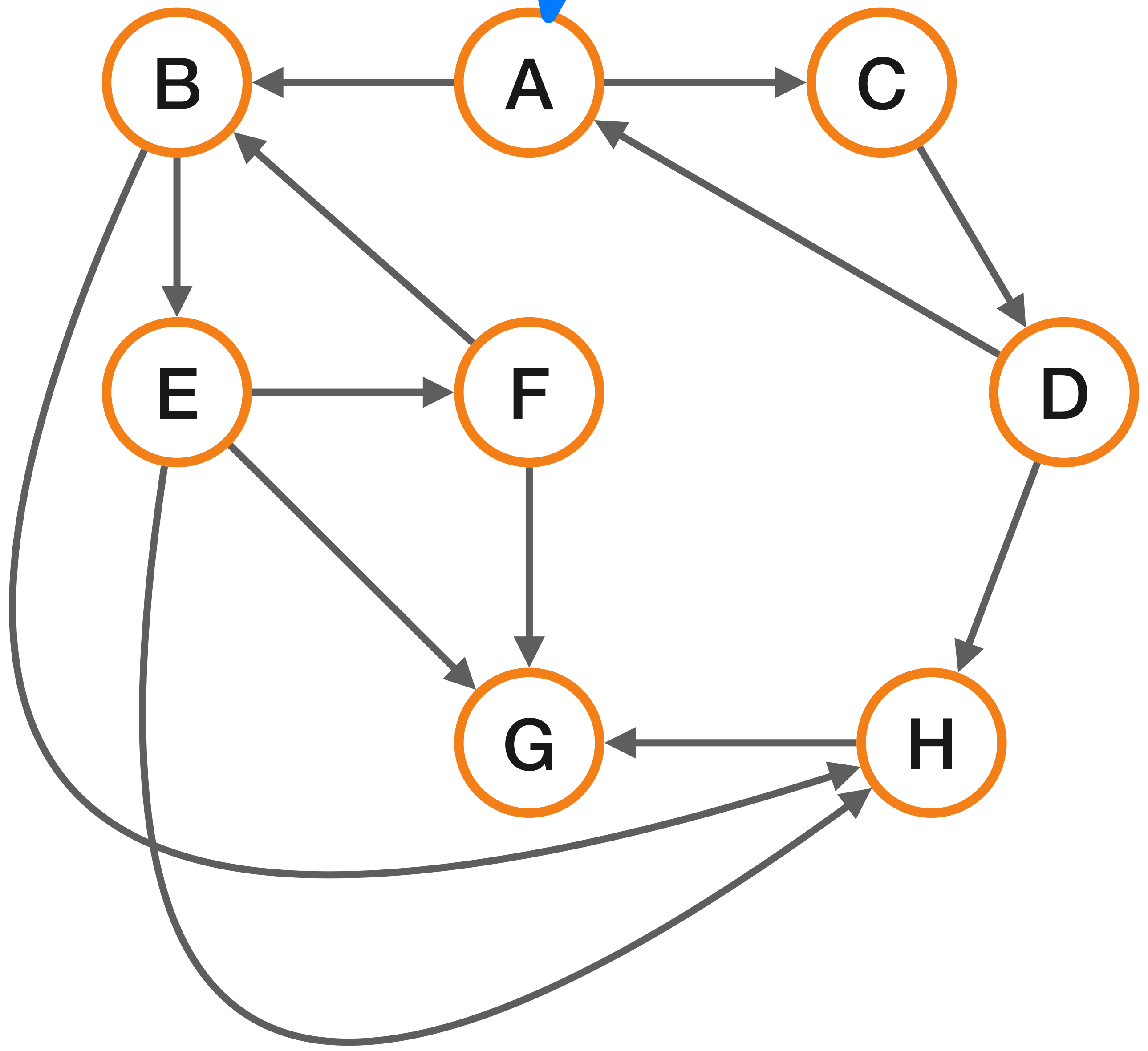


DFS in directed graphs

Exercise - do DFS on this graph and verify search tree

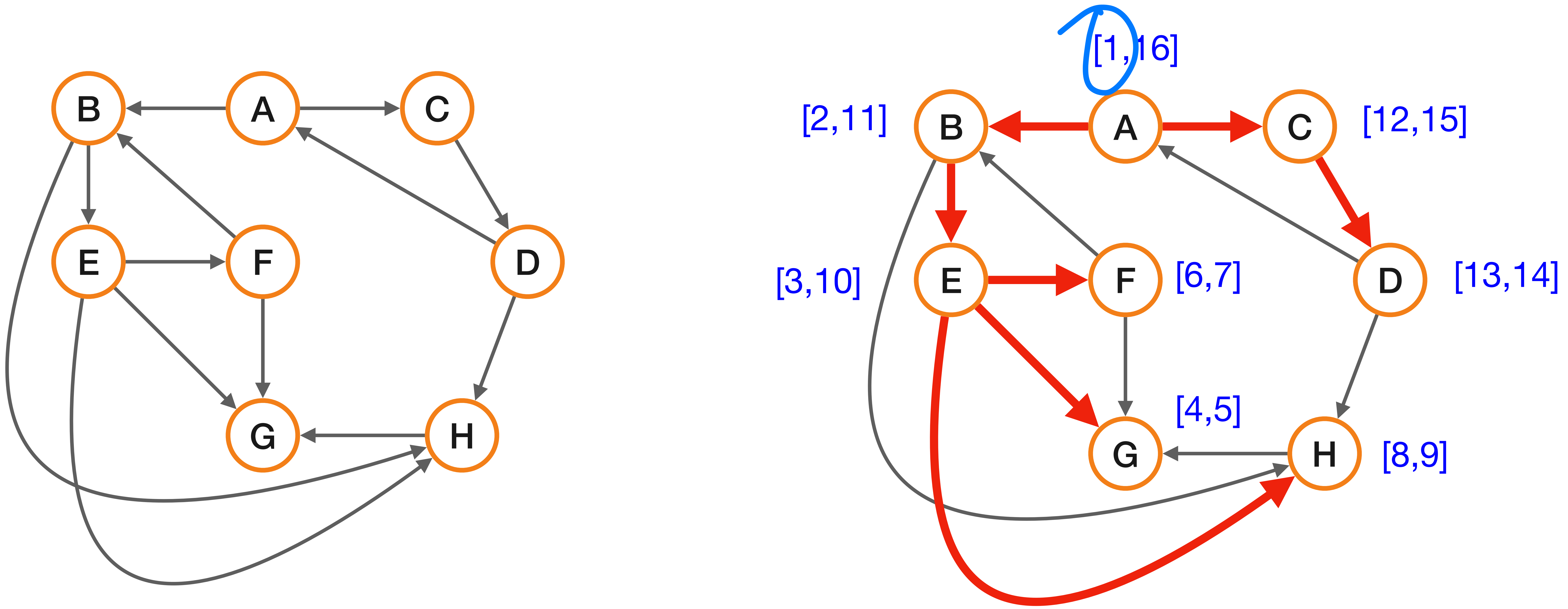
to do at home

start here



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.

Directed DFS with pre/post numbering

- *DFS(G)* takes $O(m+n)$ time.
- Edges added form a *branching*: a forest of **out**-trees.

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.*

Directed DFS with pre/post numbering

Undirected case, DFS(u) will return connected nodes to u.

- *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 $v \in rch(u)$

Directed DFS with pre/post numbering

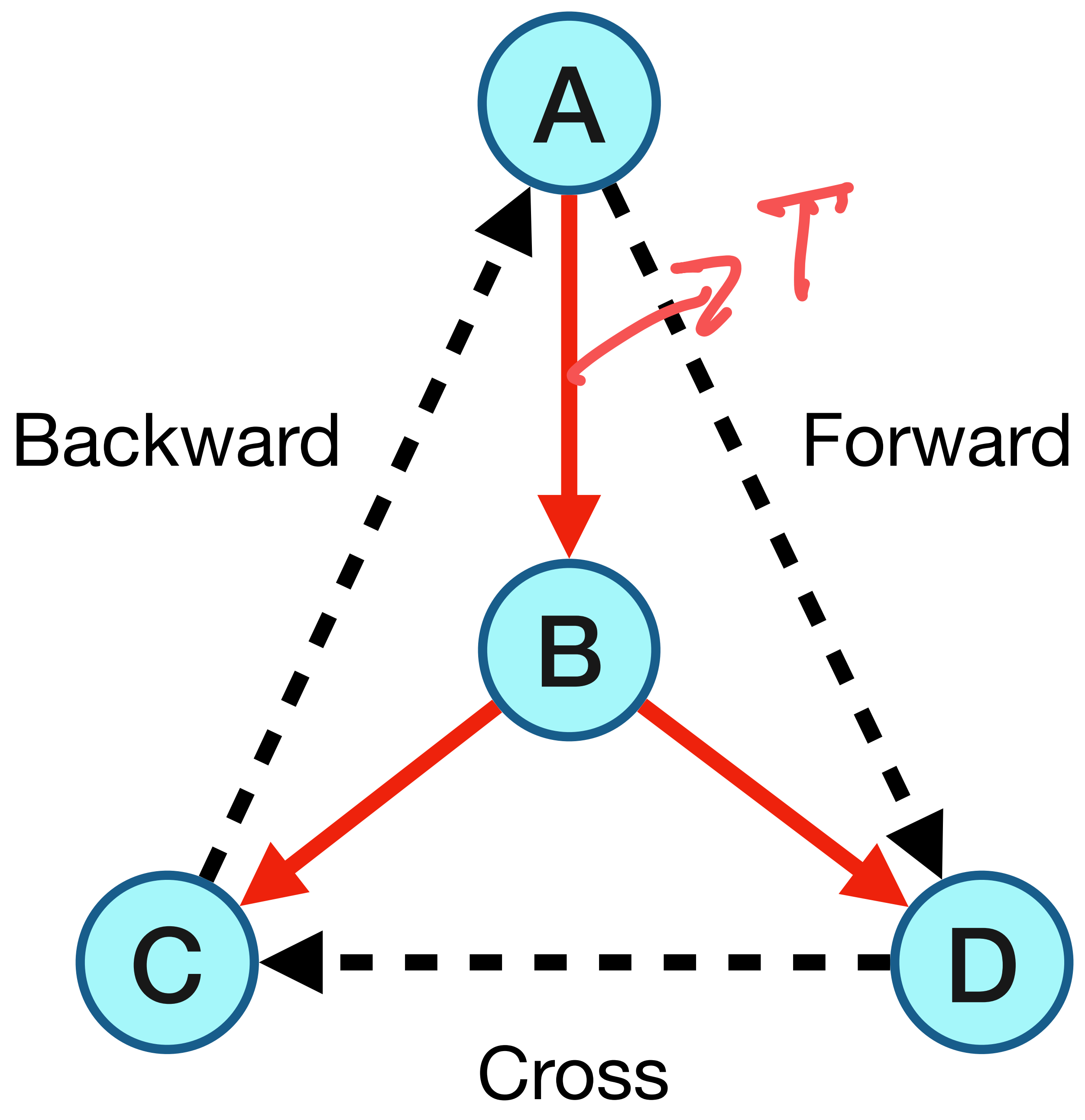
- *DFS(G)* takes $O(m+n)$ time.
- Edges added form a *branching*: a forest of **out**-trees.
  - *Output of DFS(G) depends on the order in which vertices are considered.*
- If u is the first vertex considered by *DFS(G)* then *DFS(u)* outputs a directed out-tree T rooted at u and a vertex v is in T if 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.

↳ think of the runtime stack !!

DFS trees and edge types

Edge classisifcations

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

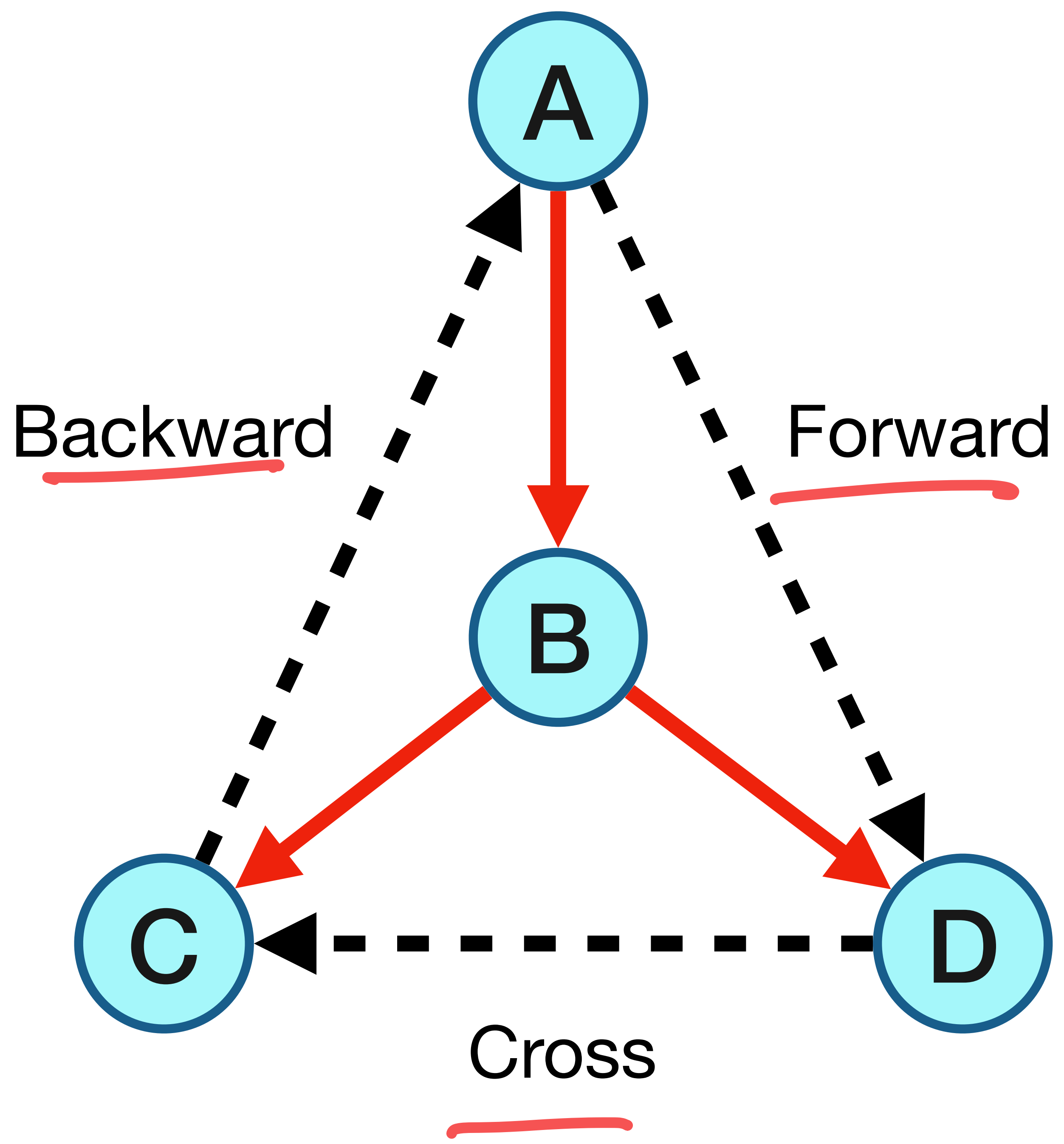


DFS trees and edge types

Edge classisifcations

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

- Tree edges that belong to T

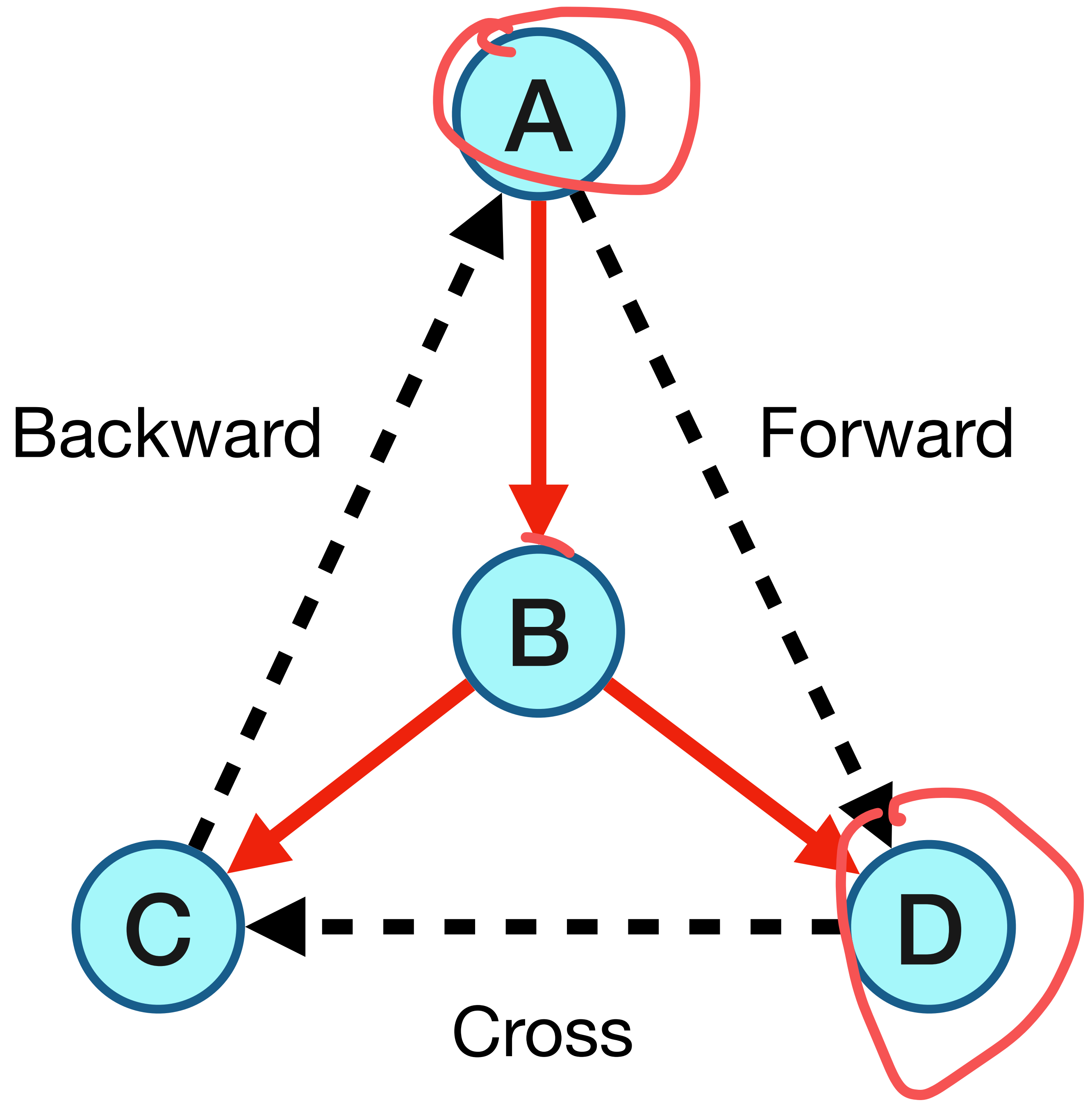


DFS trees and edge types

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 edge (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

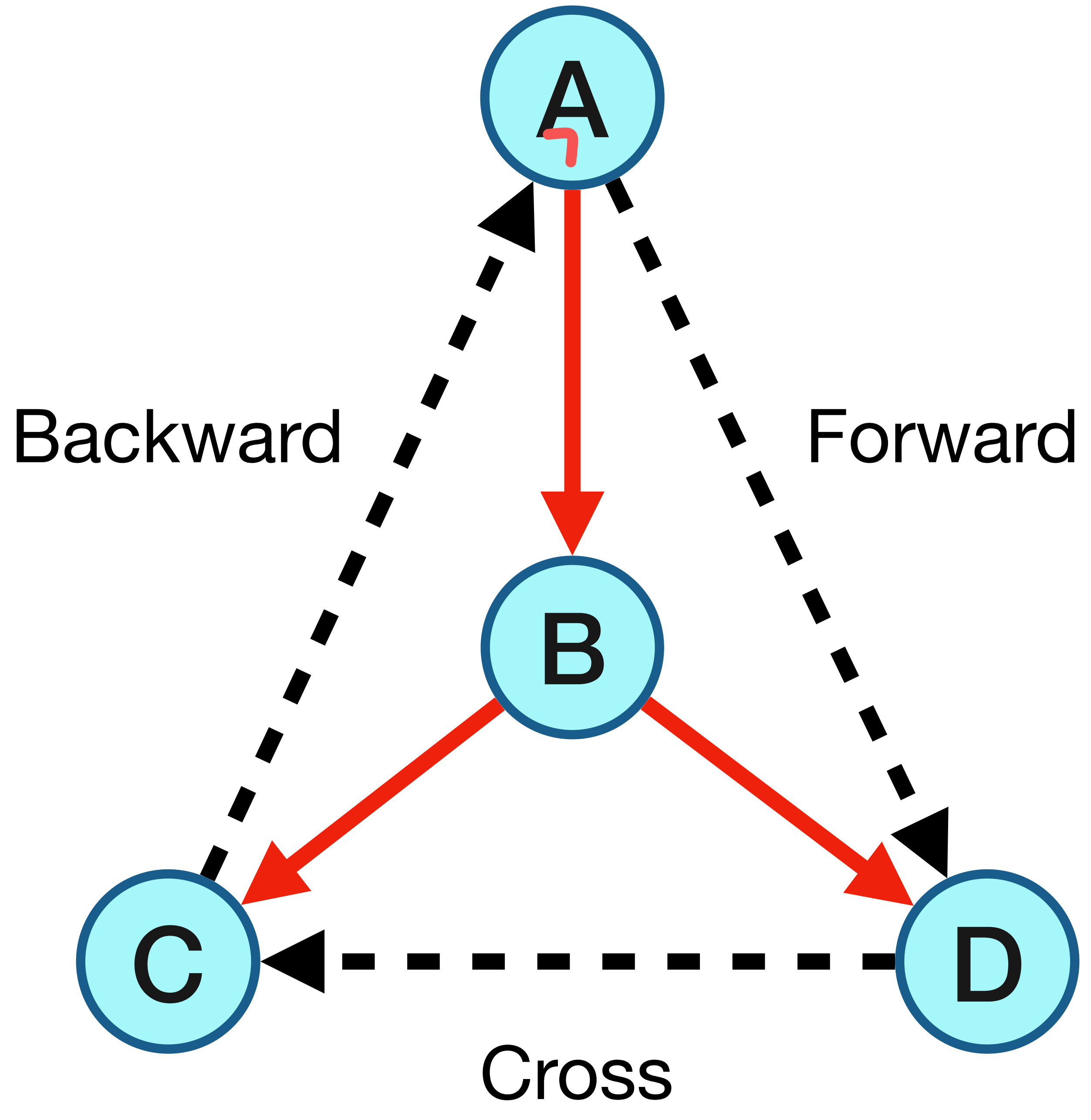


DFS trees and edge types

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 $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

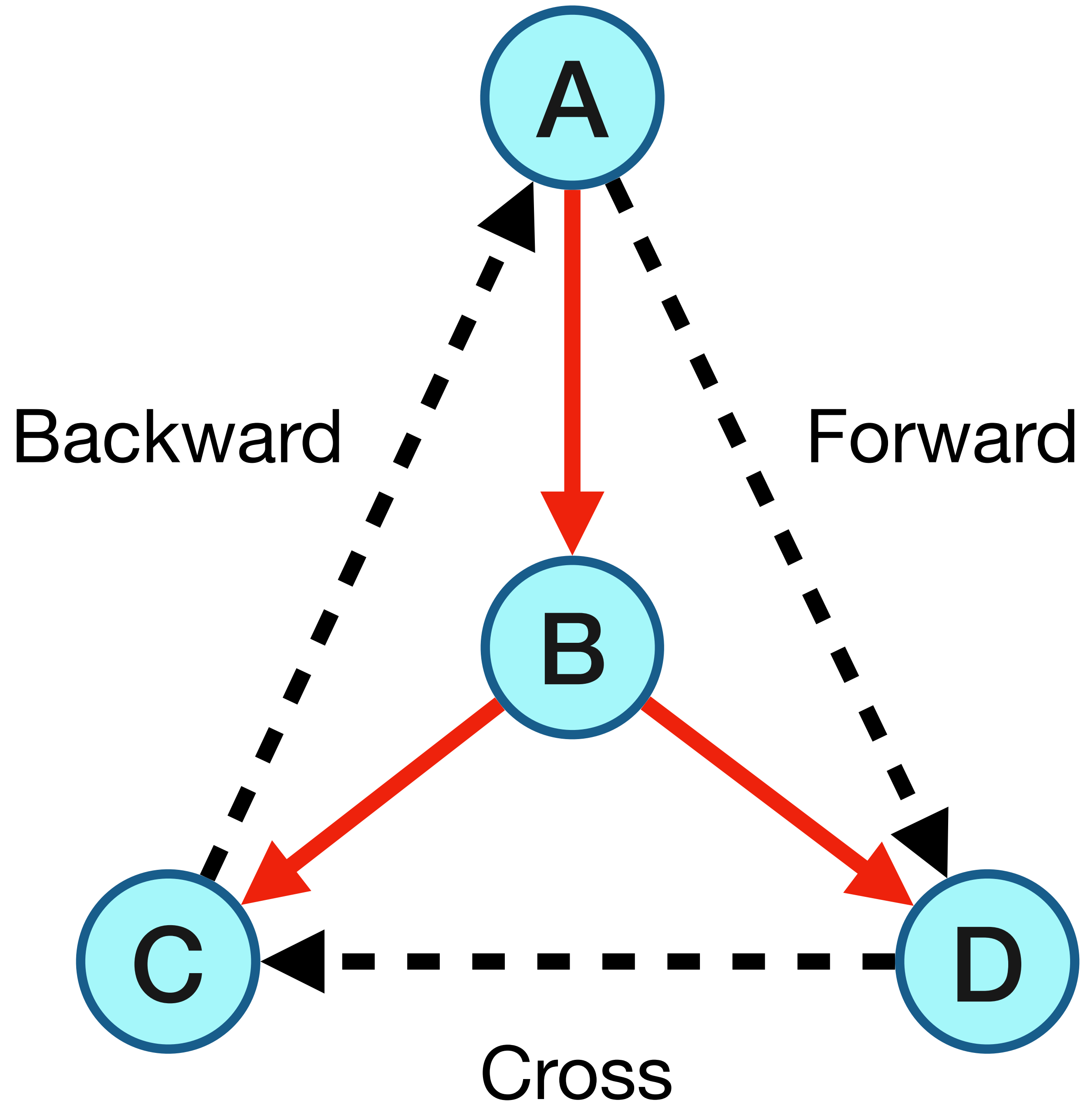


DFS trees and edge types

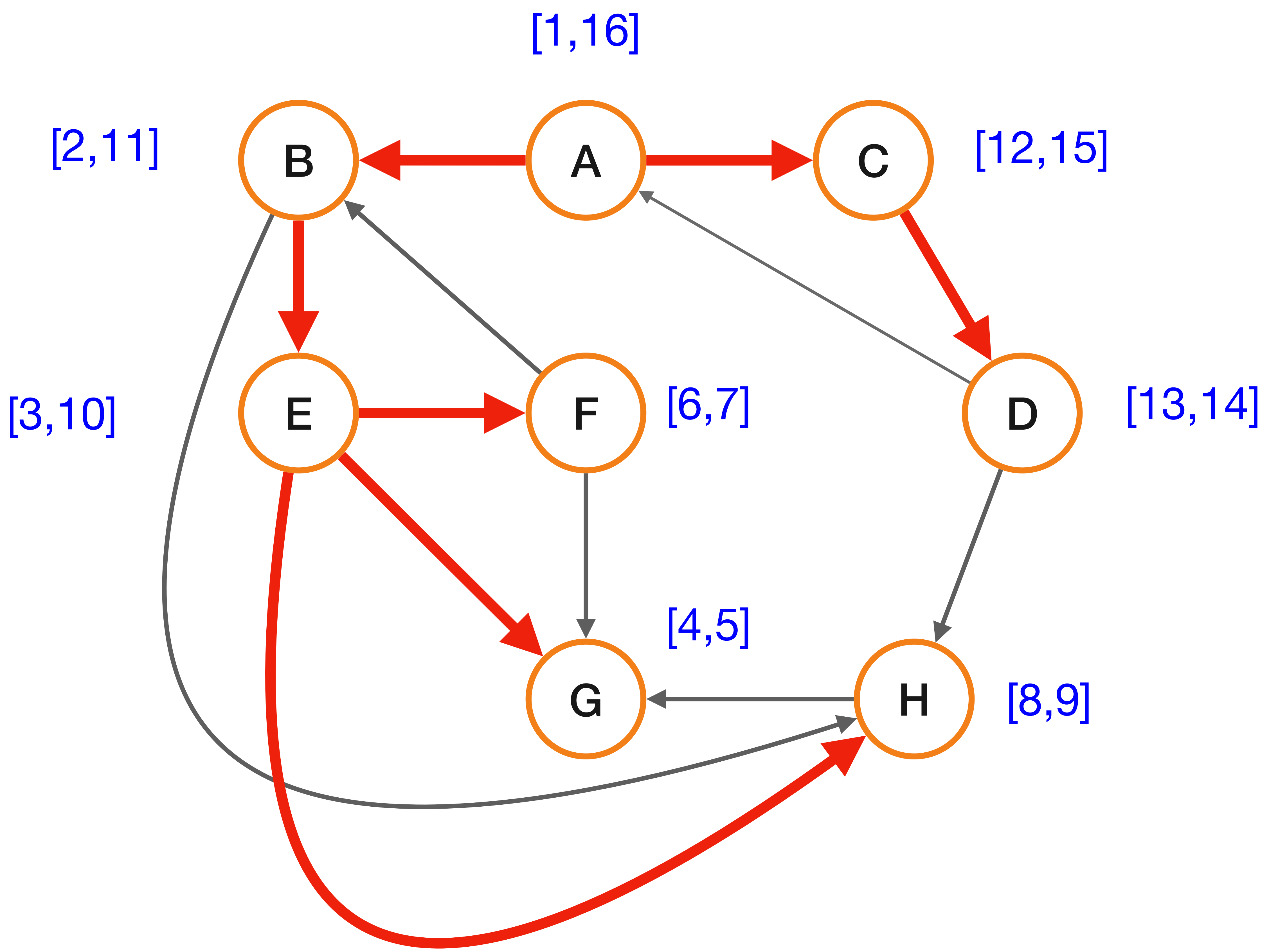
Edge classisifcations

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

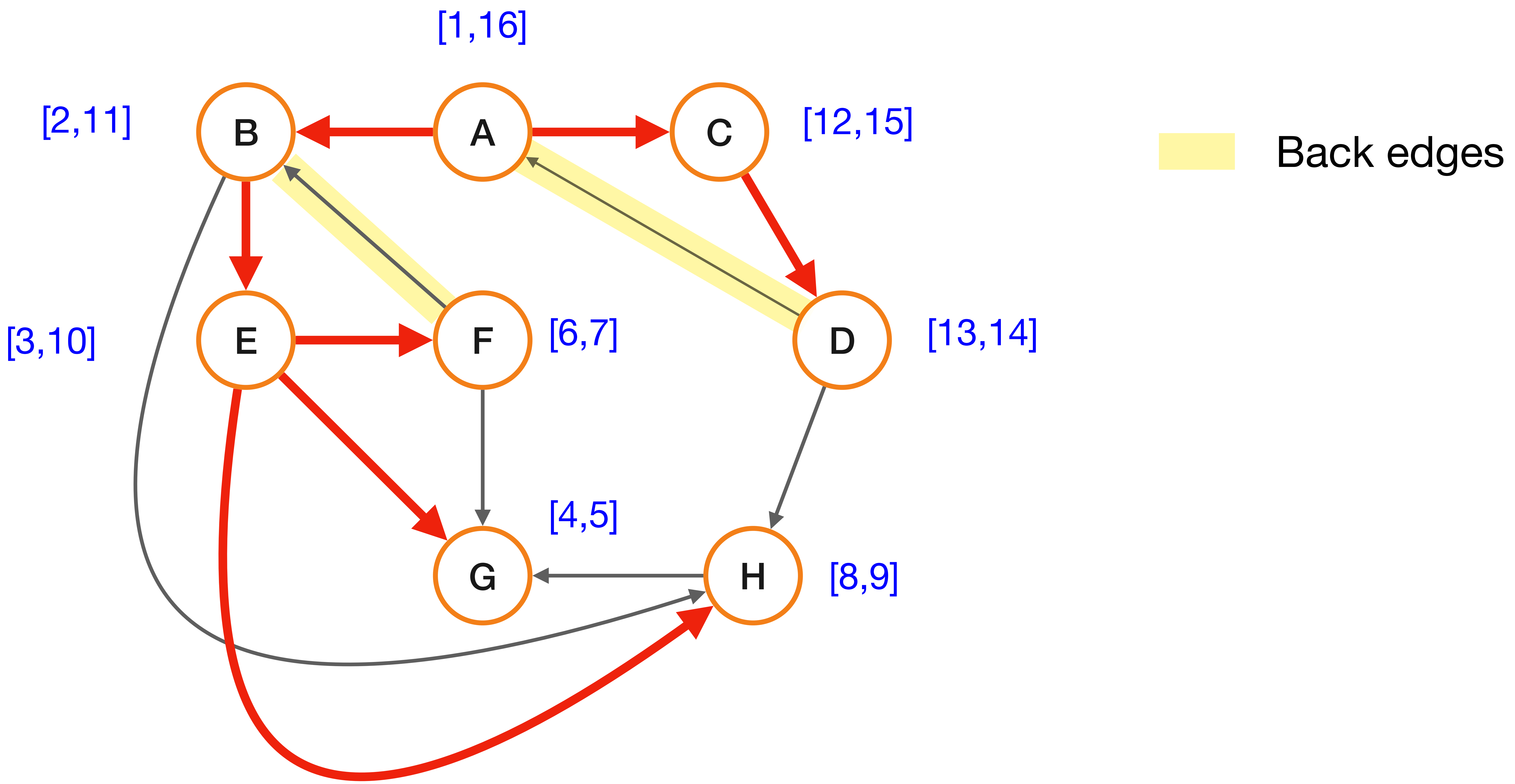
- Tree edges that belong to T
- A forward edge is a non-tree edges (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A backward edge is a non-tree edge (y, x) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A cross edge is a non-tree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



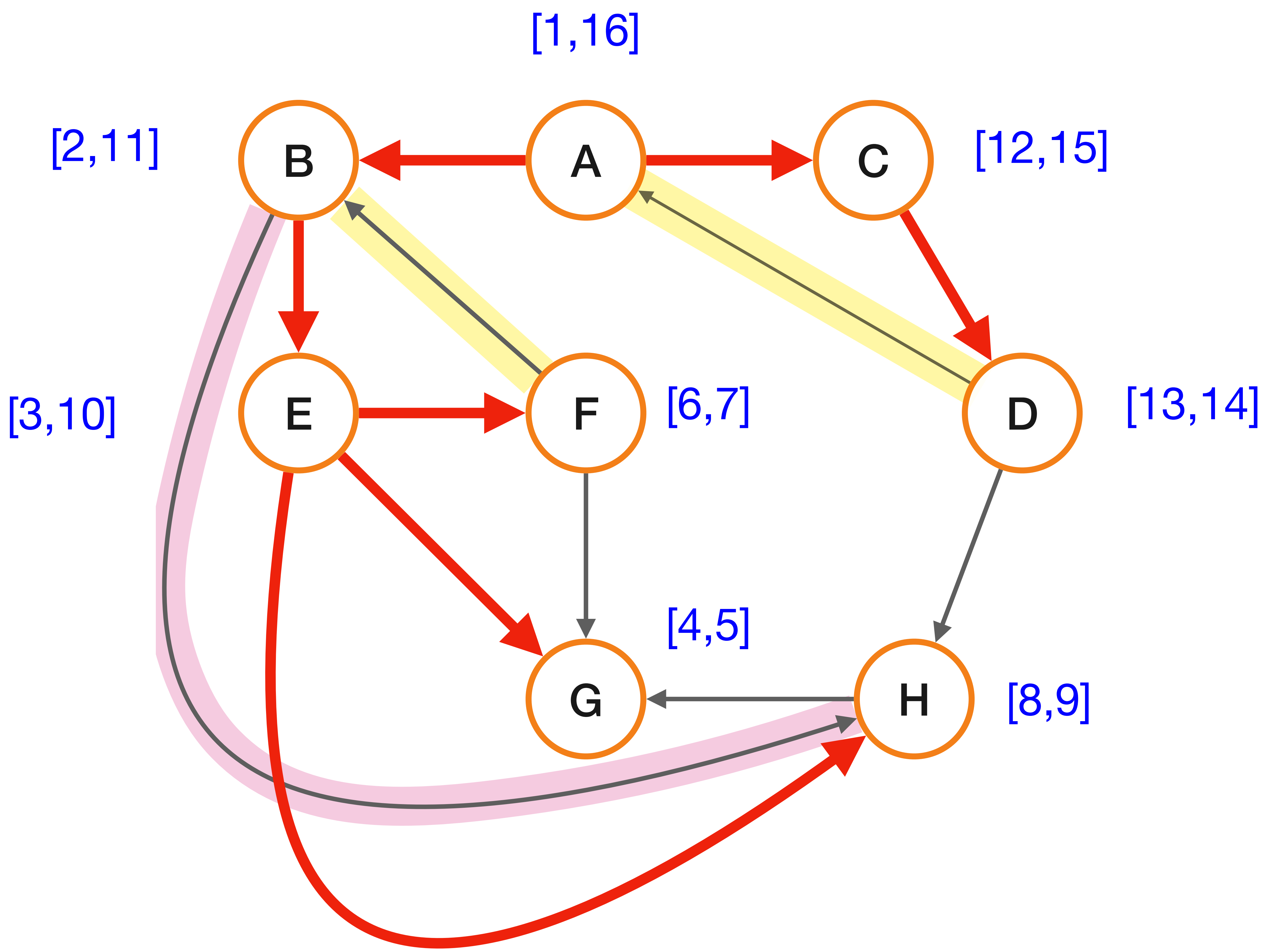
Types of edges



Types of edges



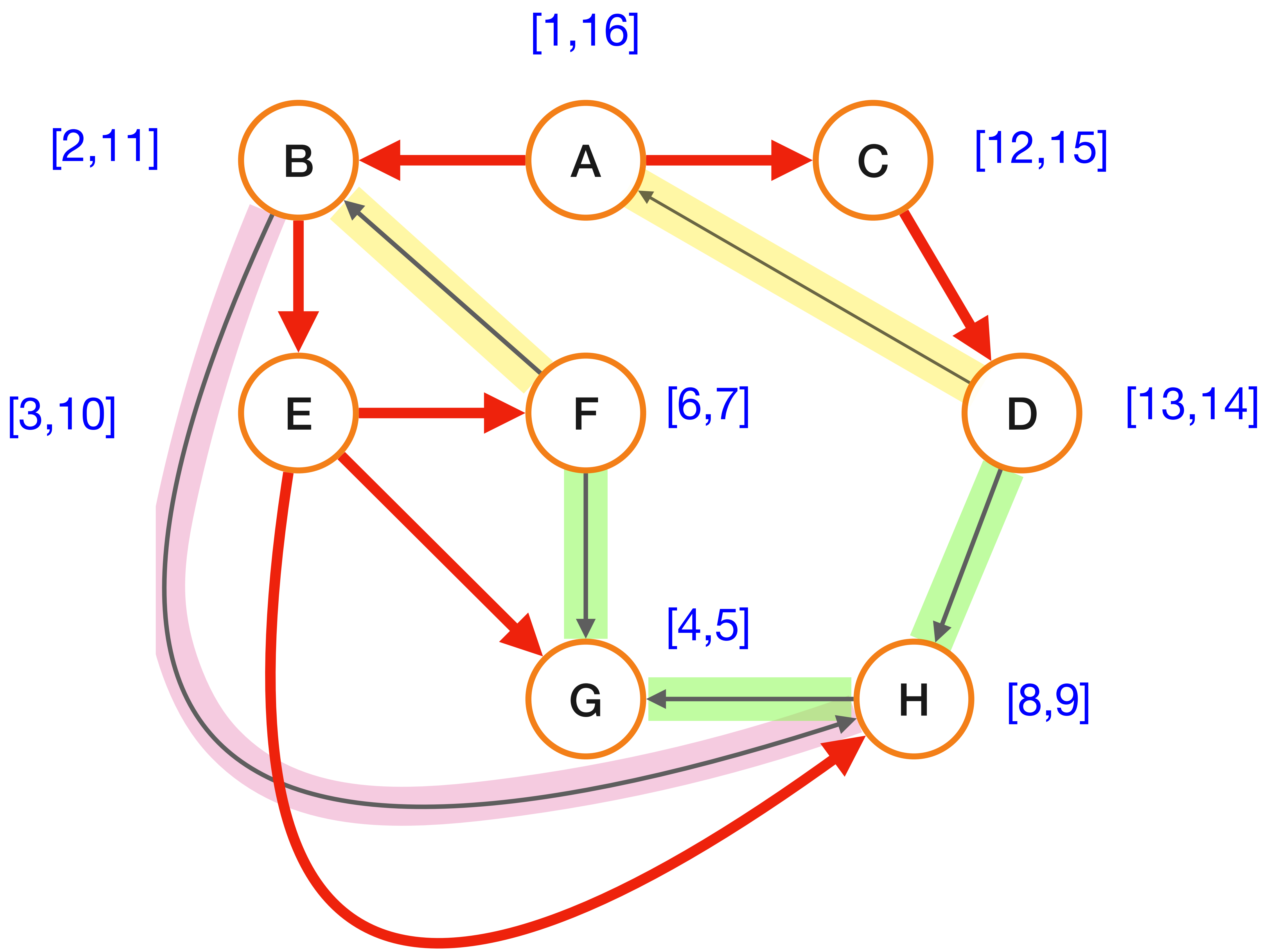
Types of edges



Back edges

Forward edges

Types of edges



Back edges

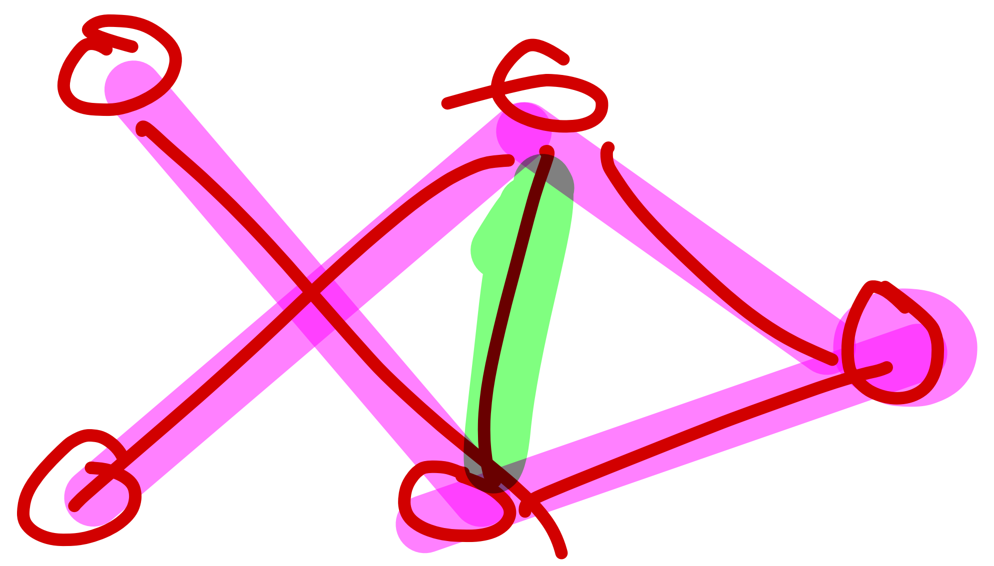
Forward edges

Cross edges

Exercise: confirm slide #25's definition hold for these 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?

Recall T, spans V (set of vertices) ⇒ if an edge is not in T, then there is a cycle

- **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.

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 $DFS(G)$.

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).

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.

Cycle detection in directed graphs

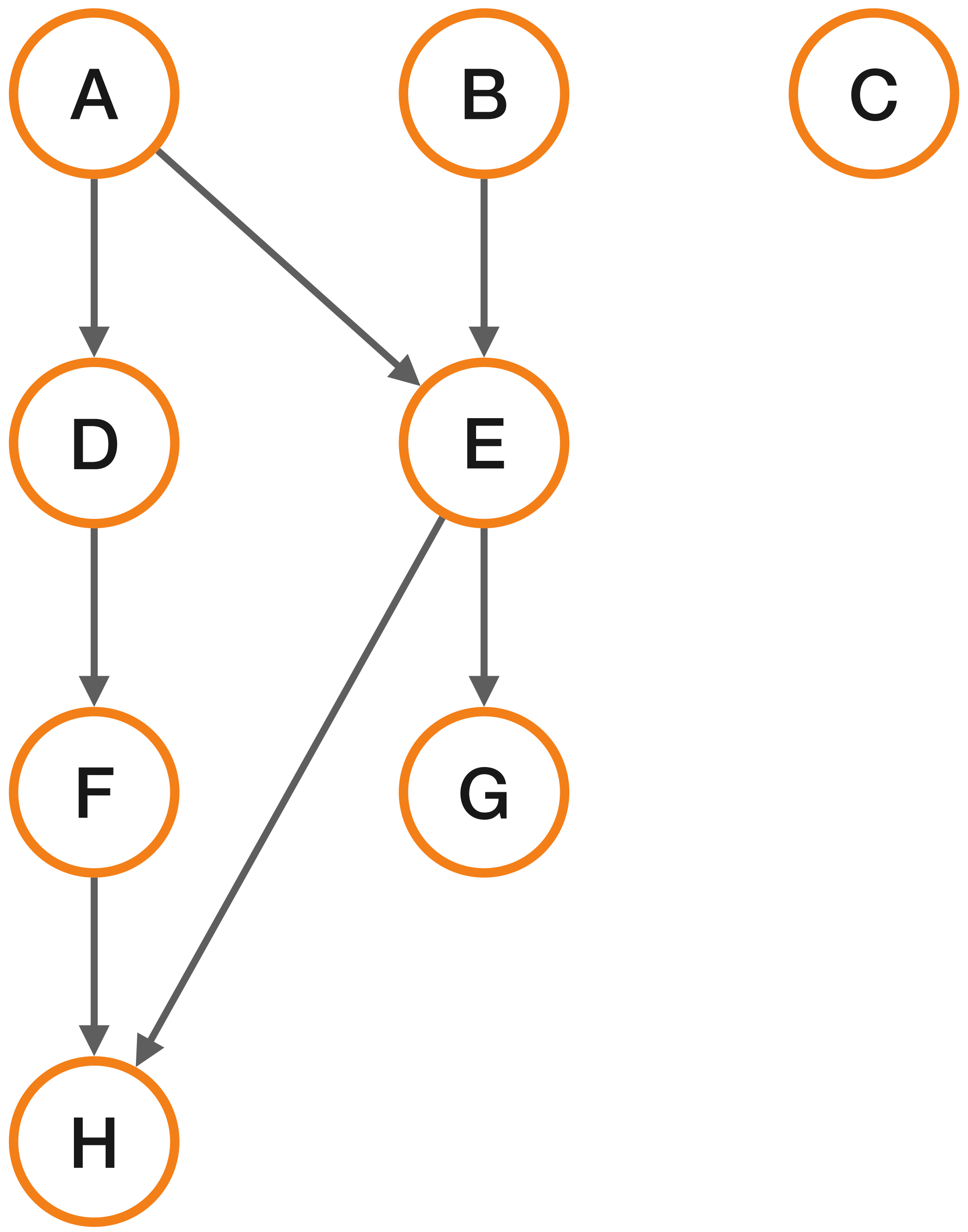
Use topological sorts

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

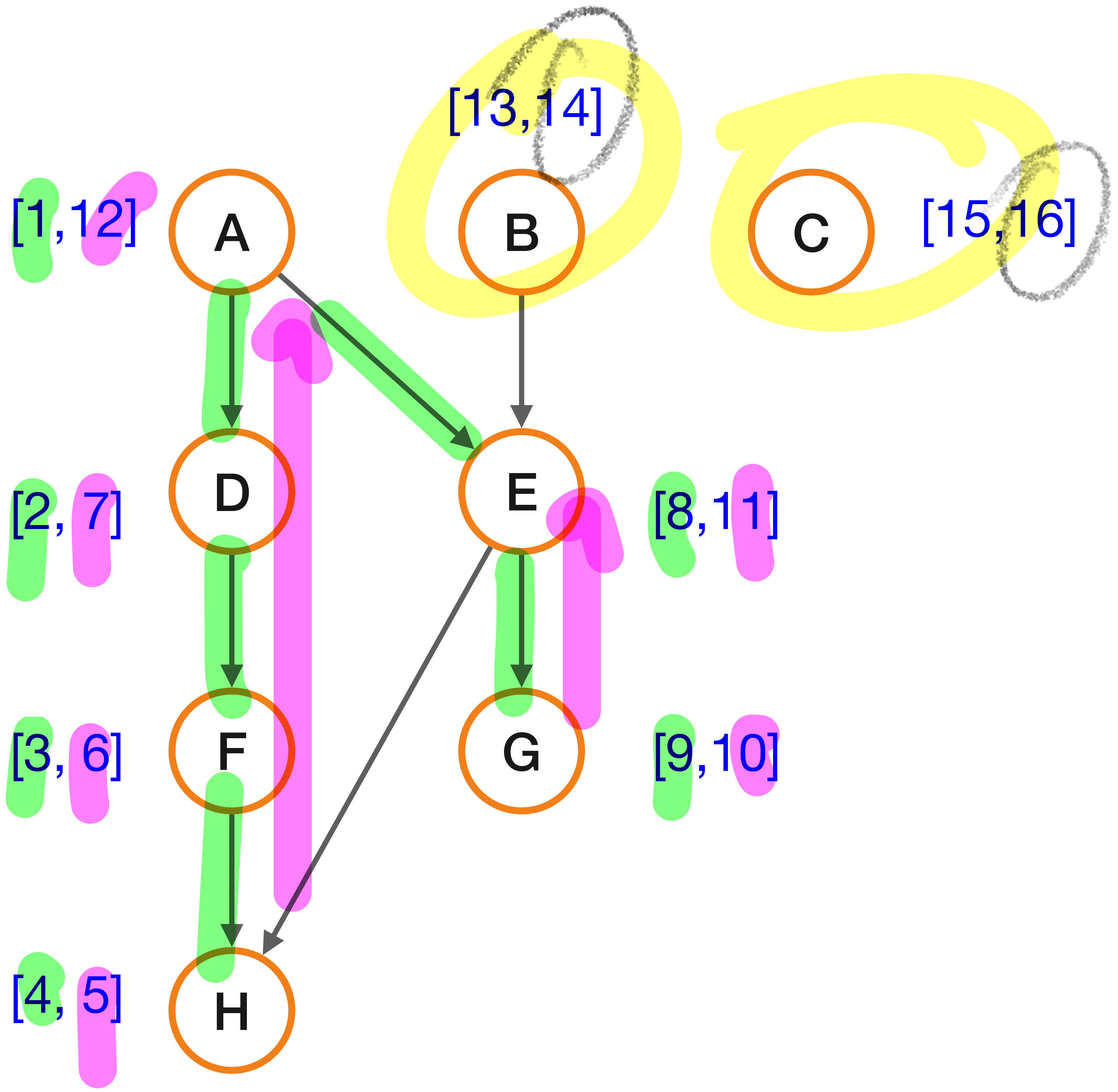
Example



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

Topological sort a graph using DFS

Example

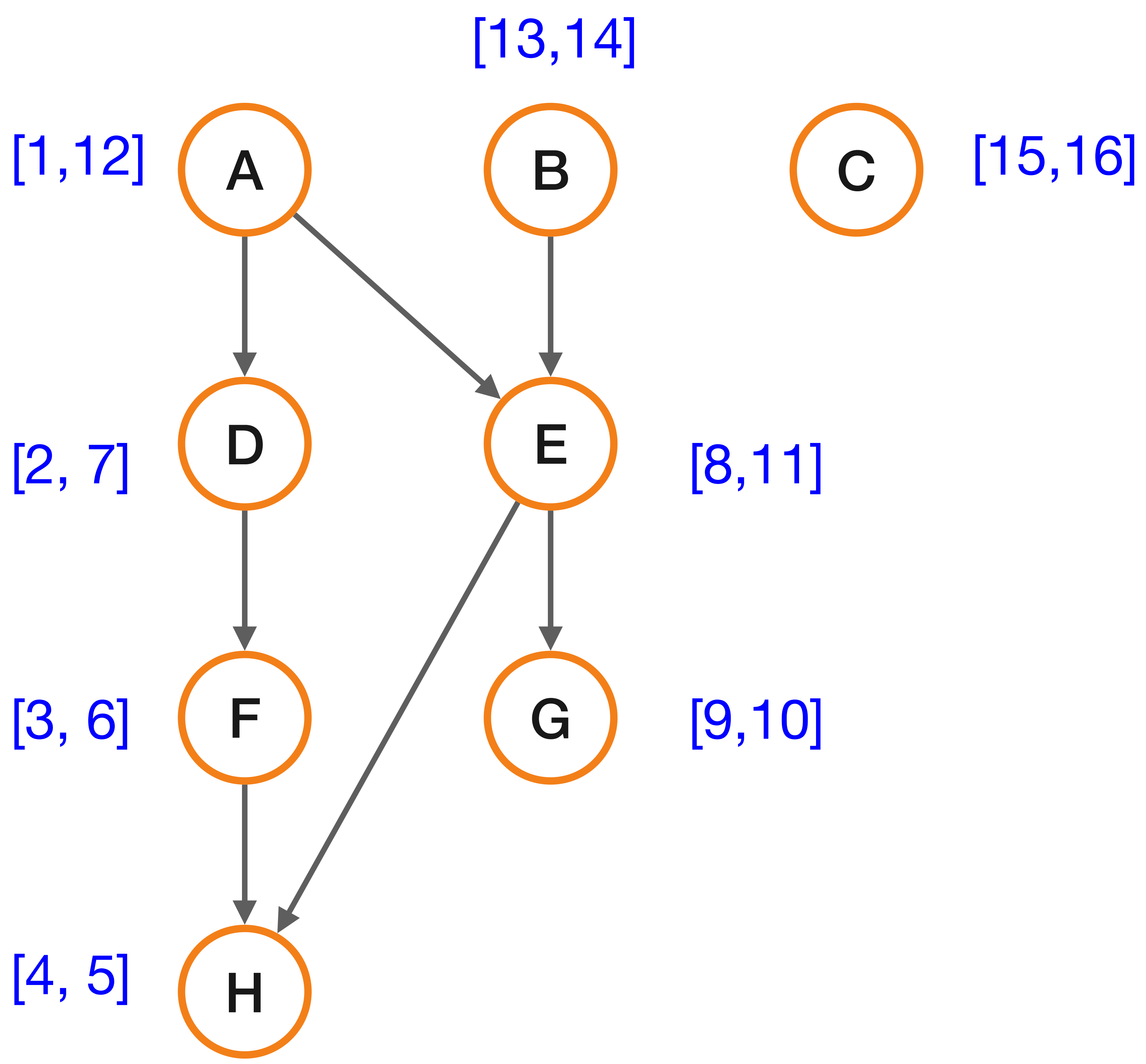


Listing out the vertices in descending order of post-visit numbers gives:

C → B

Topological sort a graph using DFS

Example

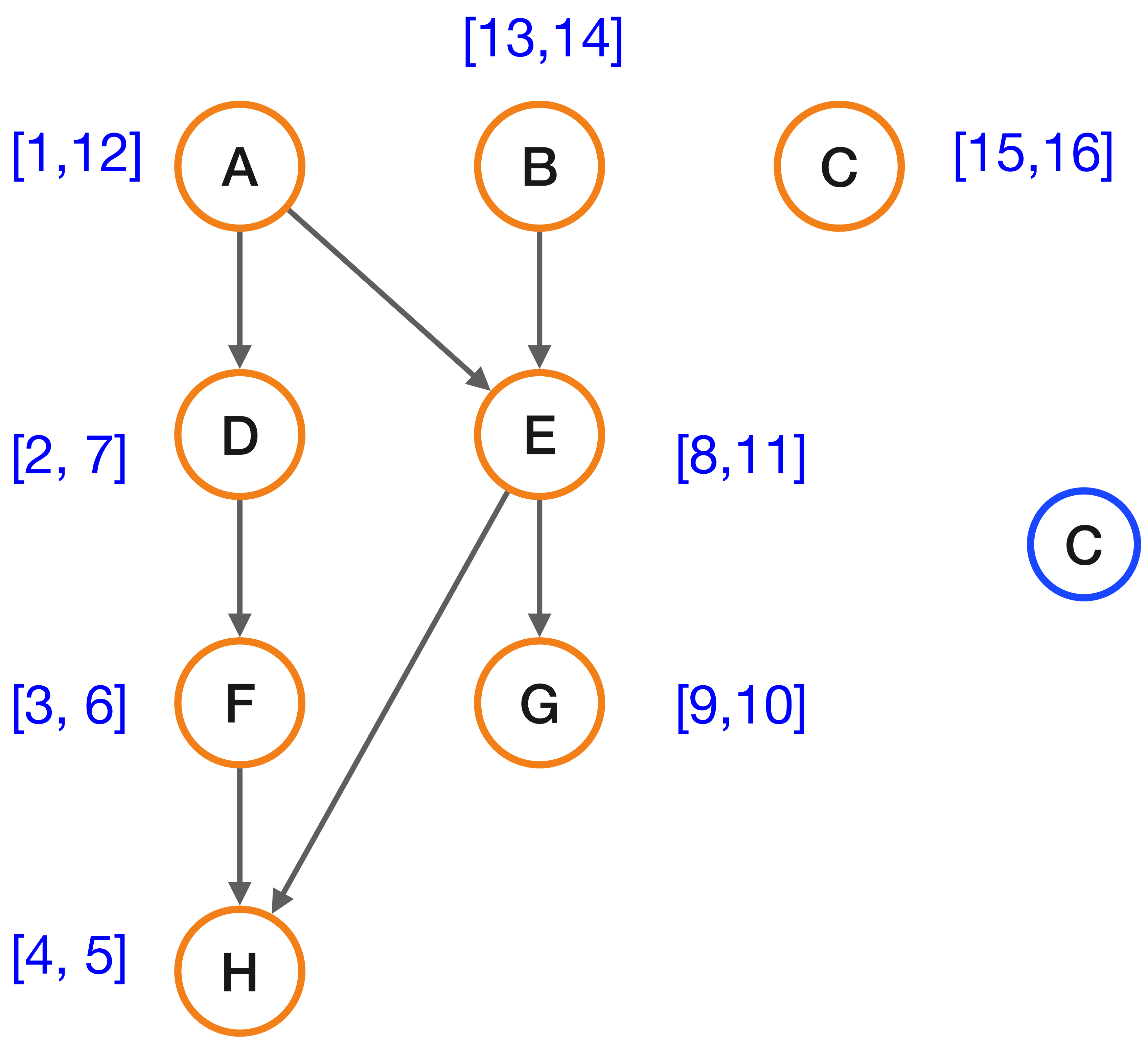


Listing out the vertices in descending order of post-visit numbers gives:

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

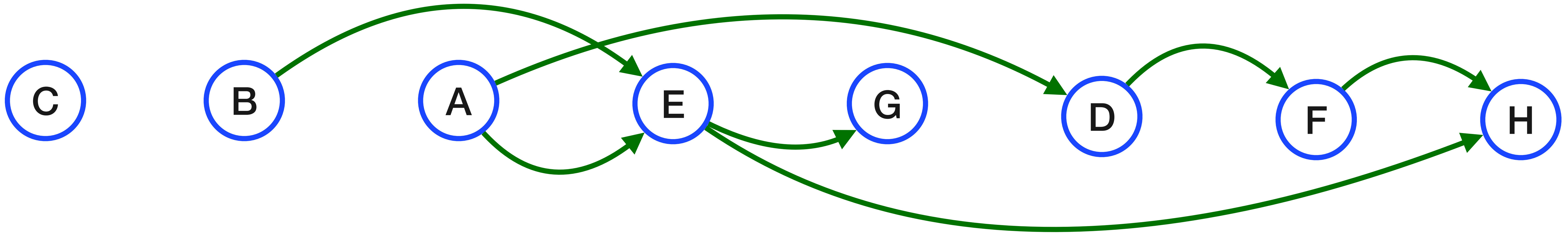
Topological sort a graph using DFS

Example



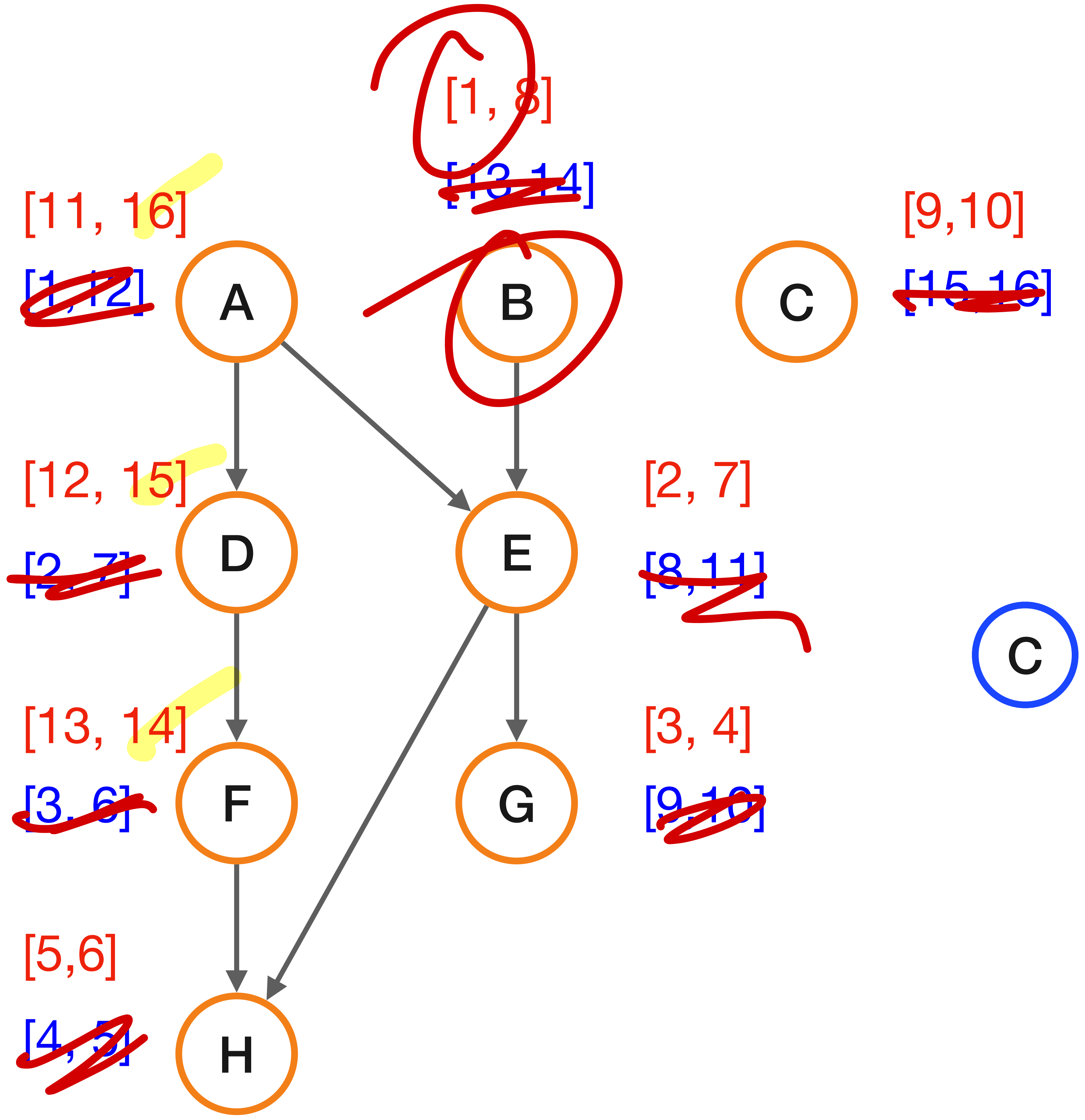
Listing out the vertices in descending order of post-visit numbers gives:

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



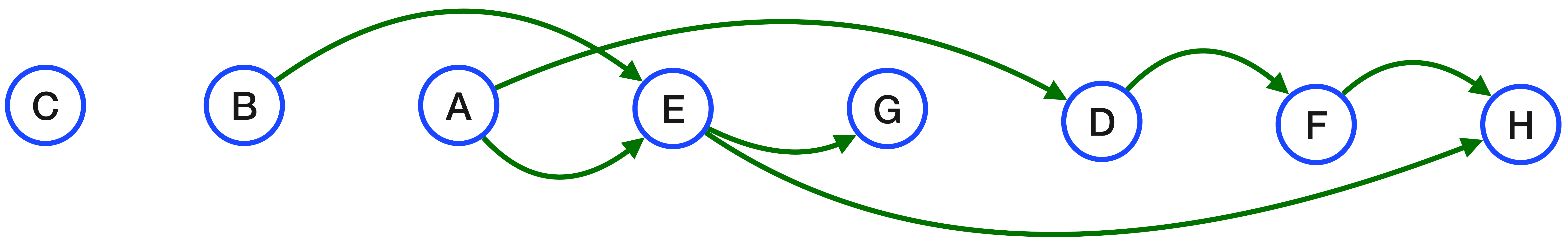
Topological sort a graph using DFS

Example



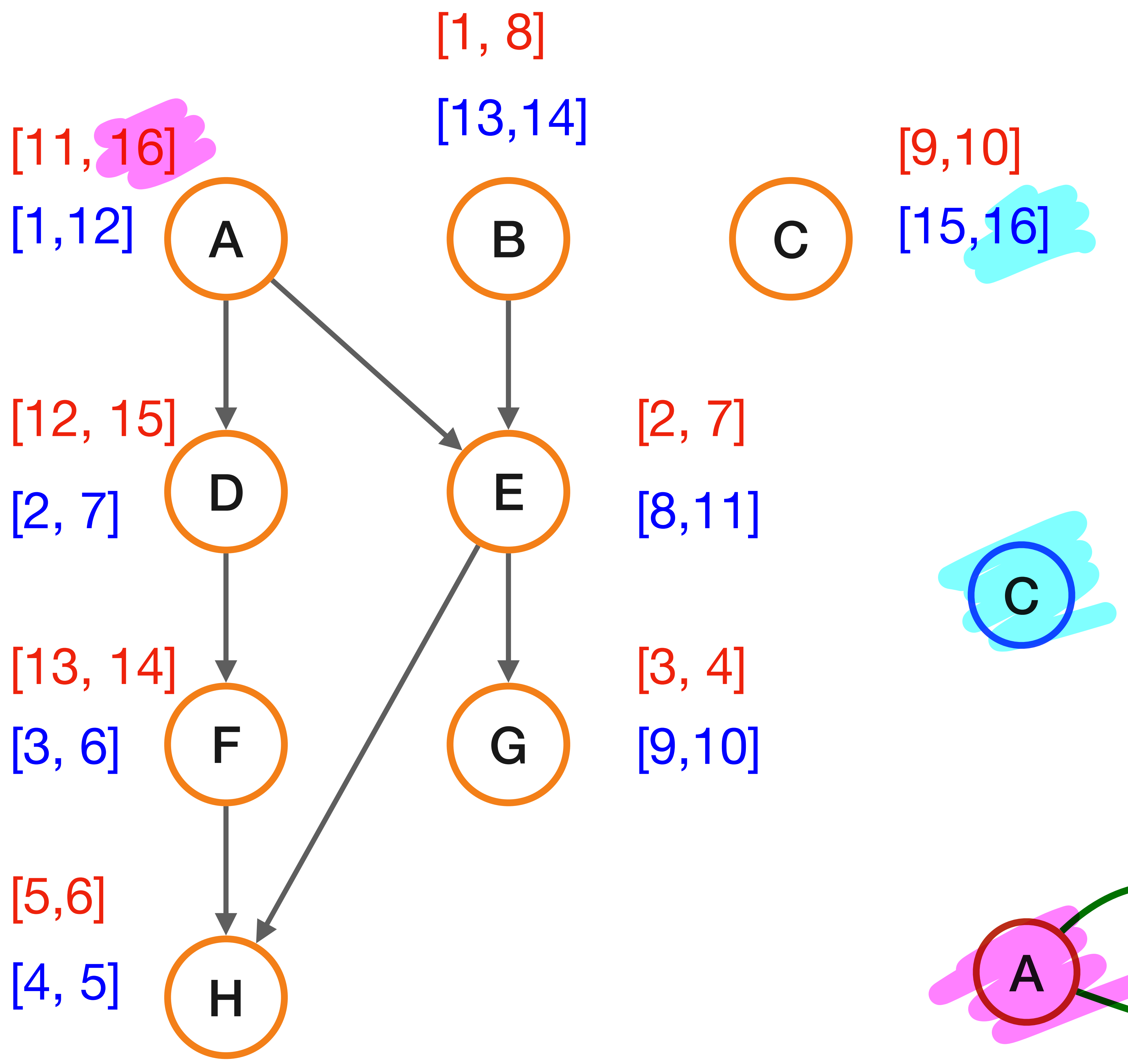
Listing out the vertices in descending order of post-visit numbers gives:

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



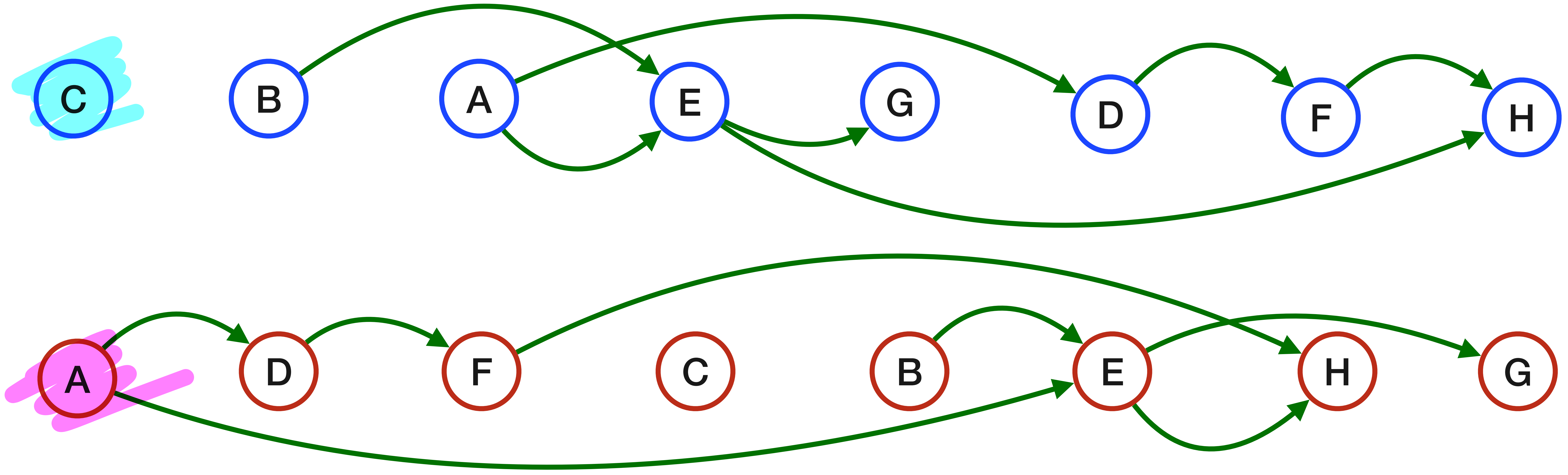
Topological sort a graph using DFS

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).

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).

Back edge and cycles

"if and only if" "necessary and sufficient"

Proposition: G has a cycle $\Longleftrightarrow$ 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 $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$.

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.

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 post$(v) >$ post(u), then $(u \rightarrow v)$ is not in G.

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:

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.

Decreasing post-visit order is a TS

Verify this hold for the graphs on the previous slides

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:

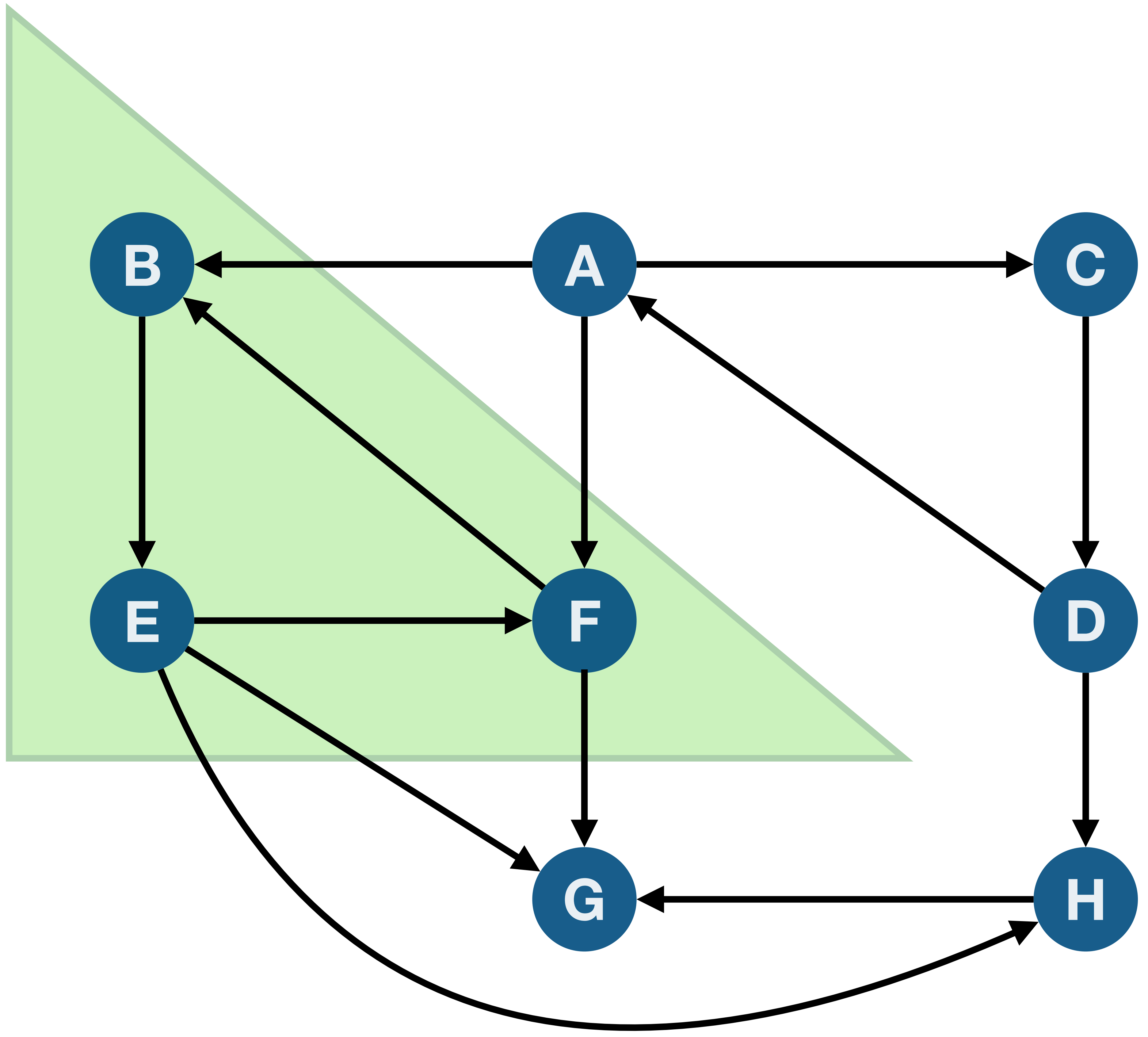
- Case 1: $[\text{pre}(u), \text{post}(u)]$ is contained in $[\text{pre}(v), \text{post}(v)]$. Implies that u is explored during $DFS(v)$ and hence is a descendent of v. Edge (u, v) implies a cycle in G but G is assumed to be DAG.

- 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.



Strongly connected components (SCCs)

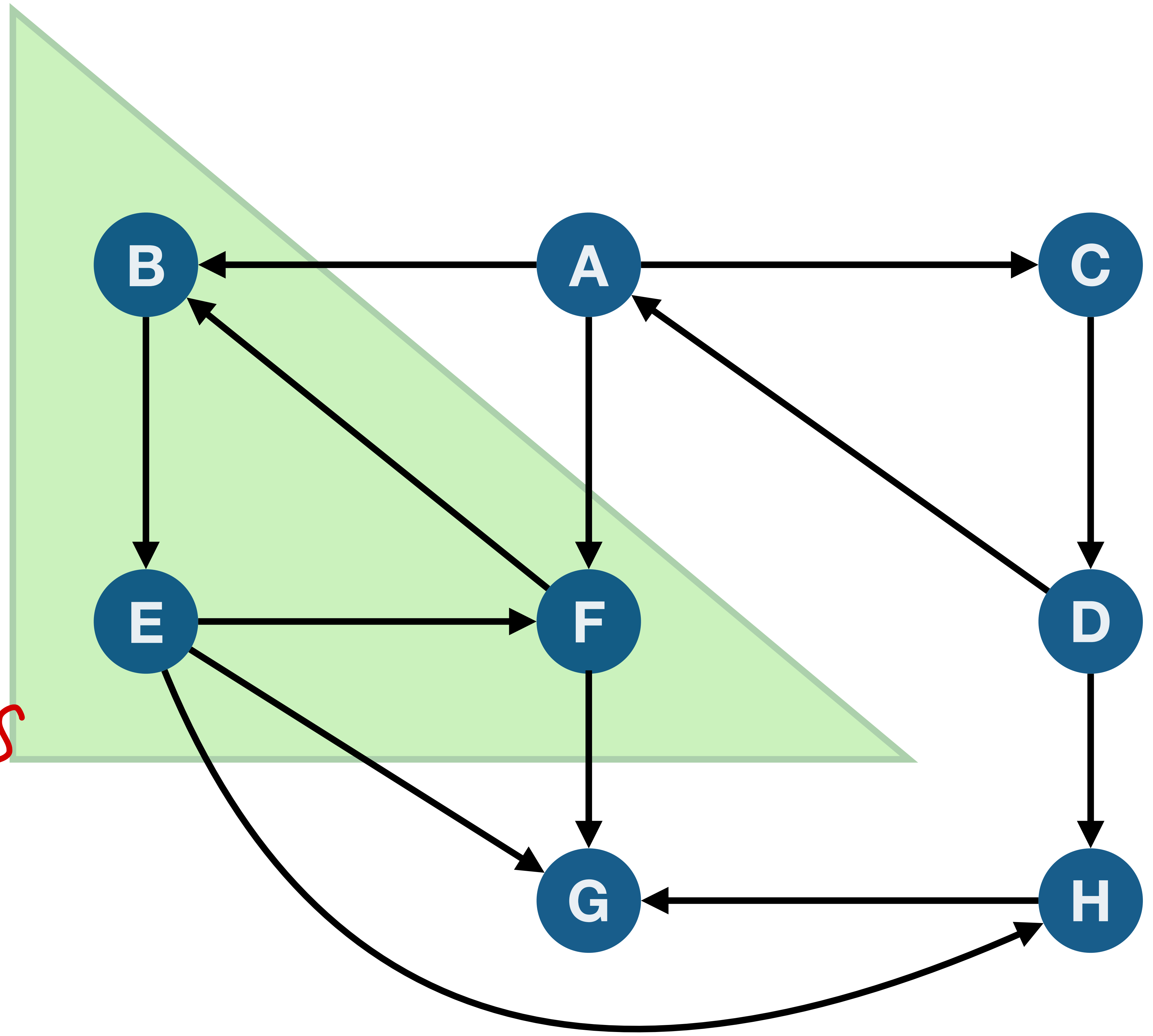
of vertices

Algorithmic problem

Find all SCCs of a given directed graph.

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

# of edges



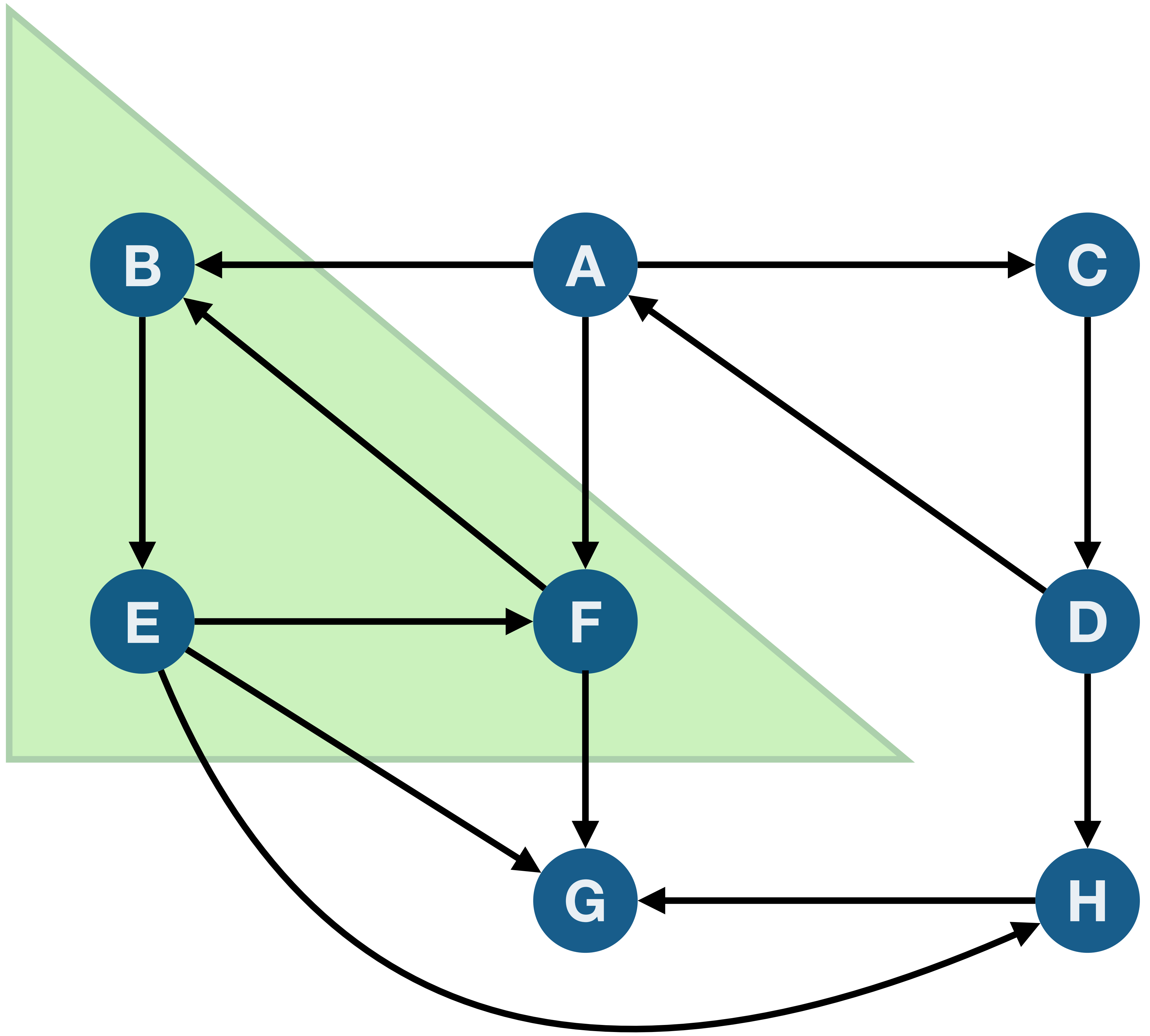
Strongly connected components (SCCs)

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.



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:

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.
```

Discussed last time

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

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.
```

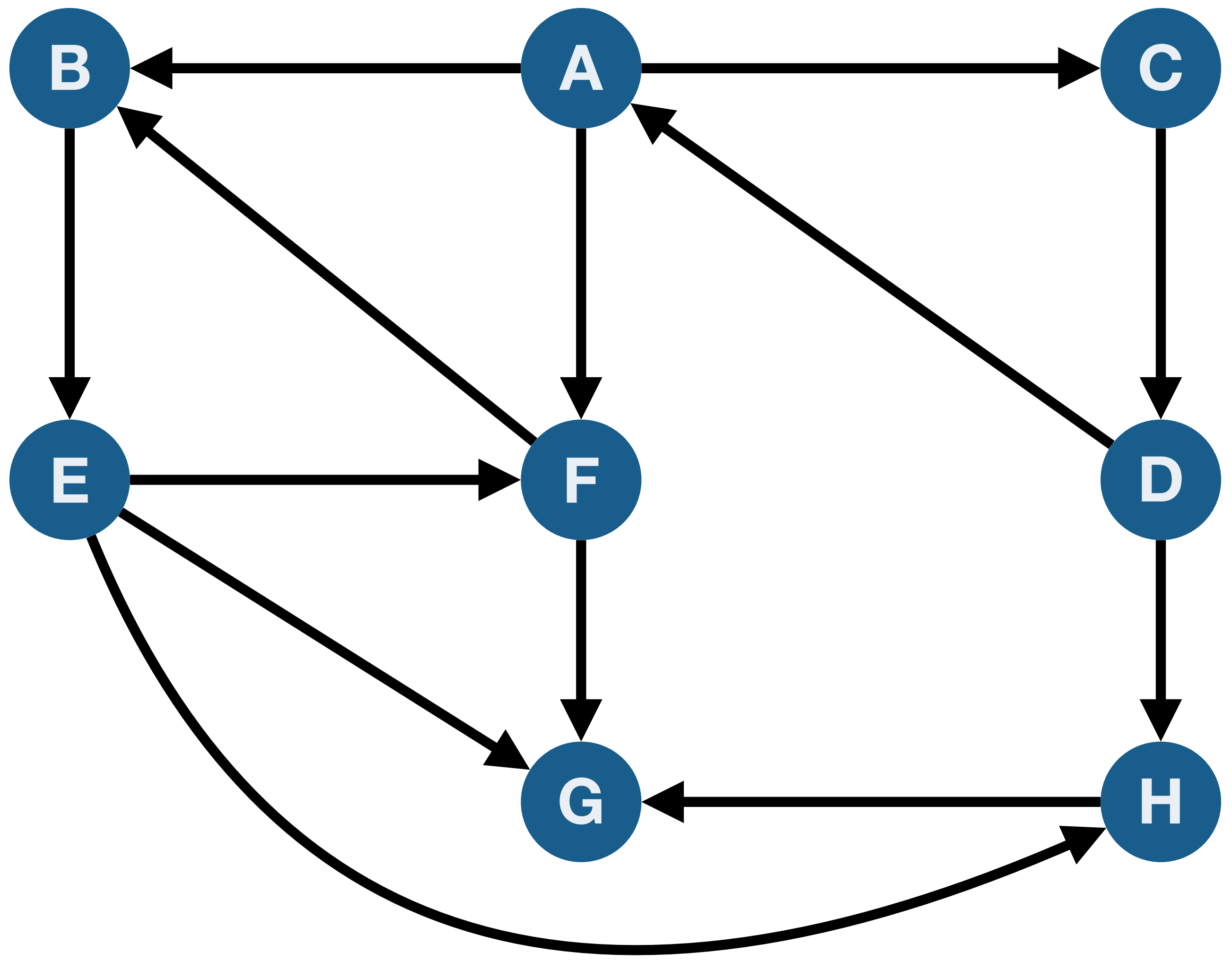
Running time: $O(n(n+m))$

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

Graph of SCCs

Meta-graph of SCCs

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

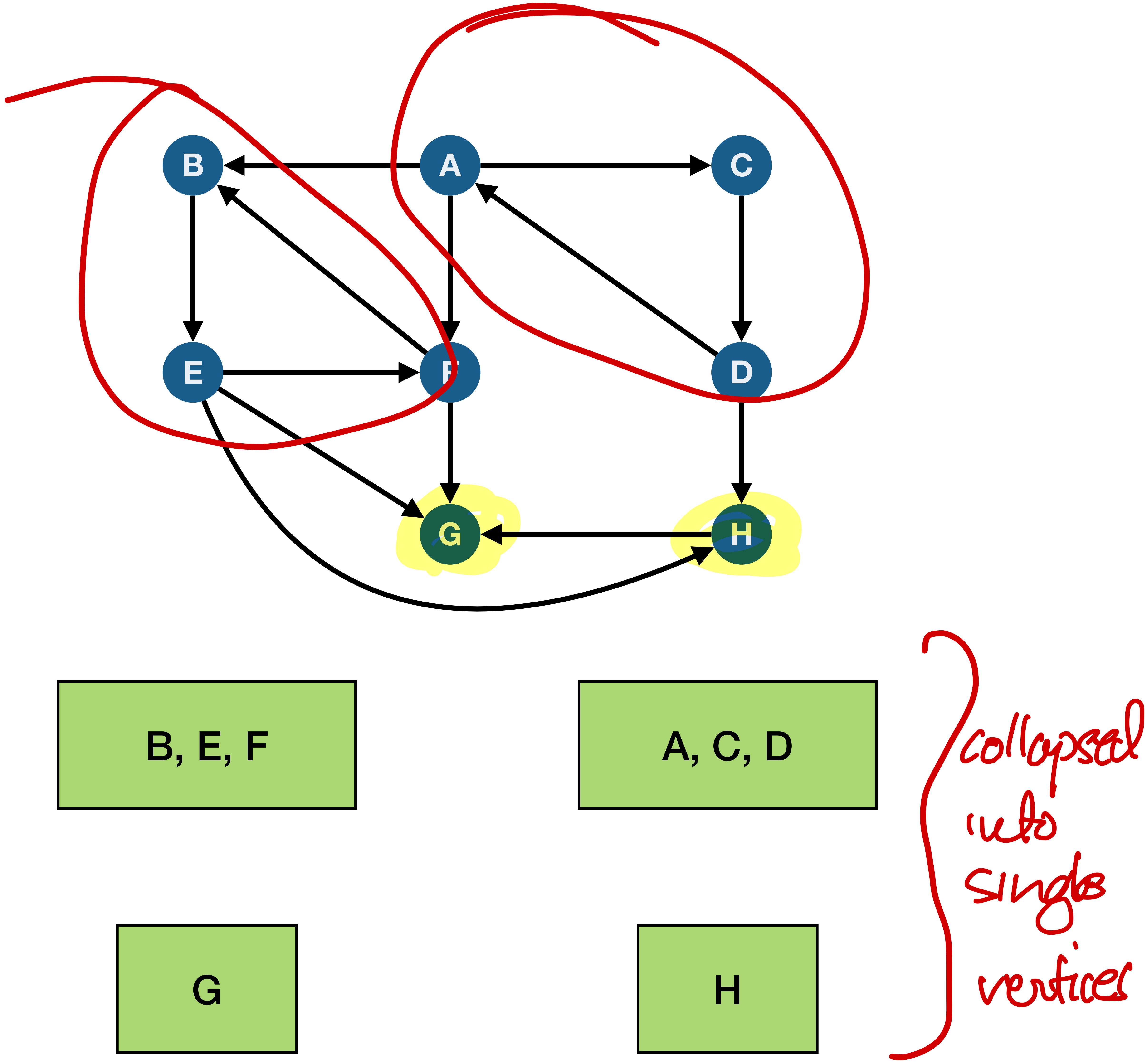


Graph of SCCs

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 as G^{SCC}:

- Vertices of G^{SCC} are $S_1, S_2, \ldots S_k$

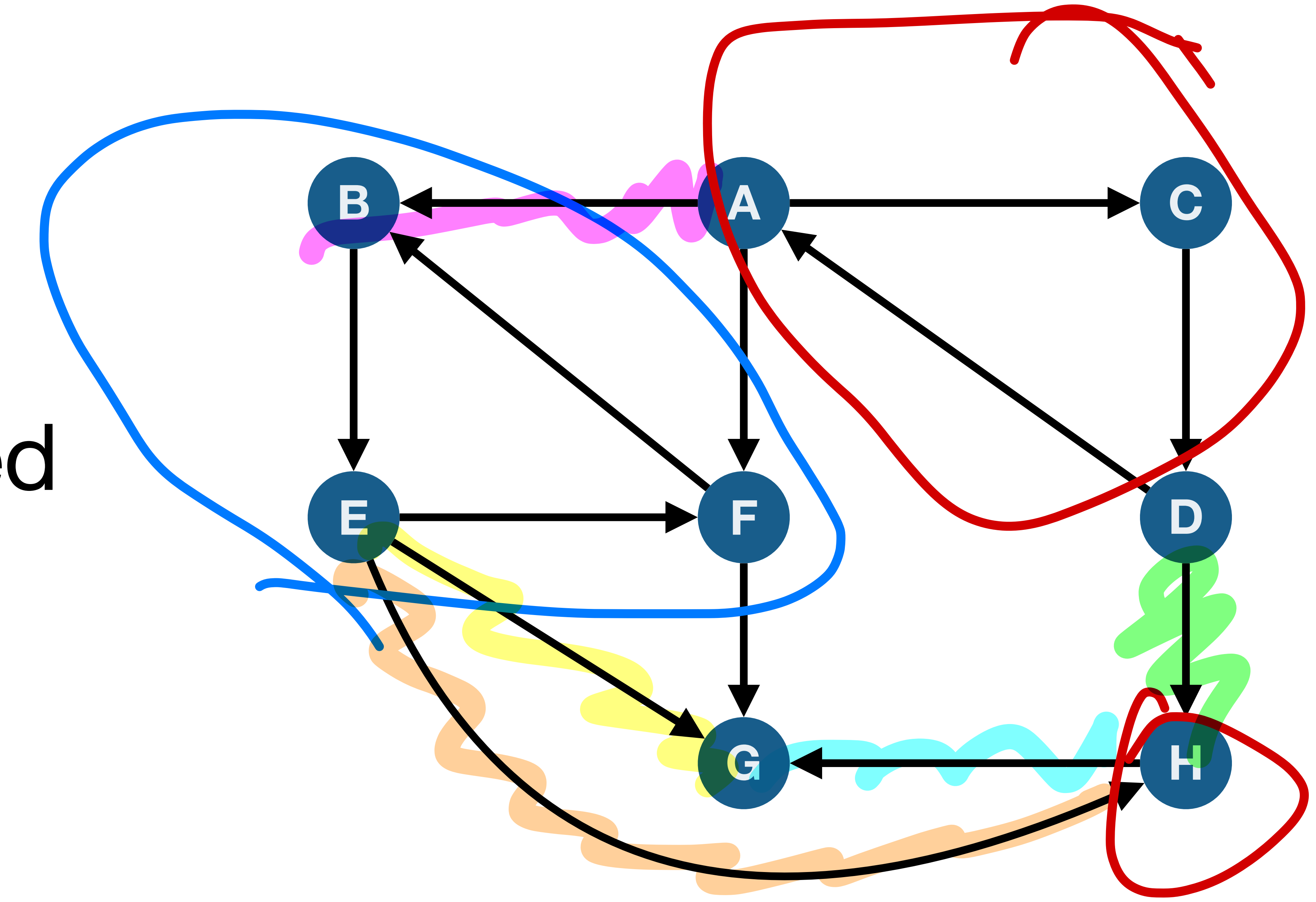


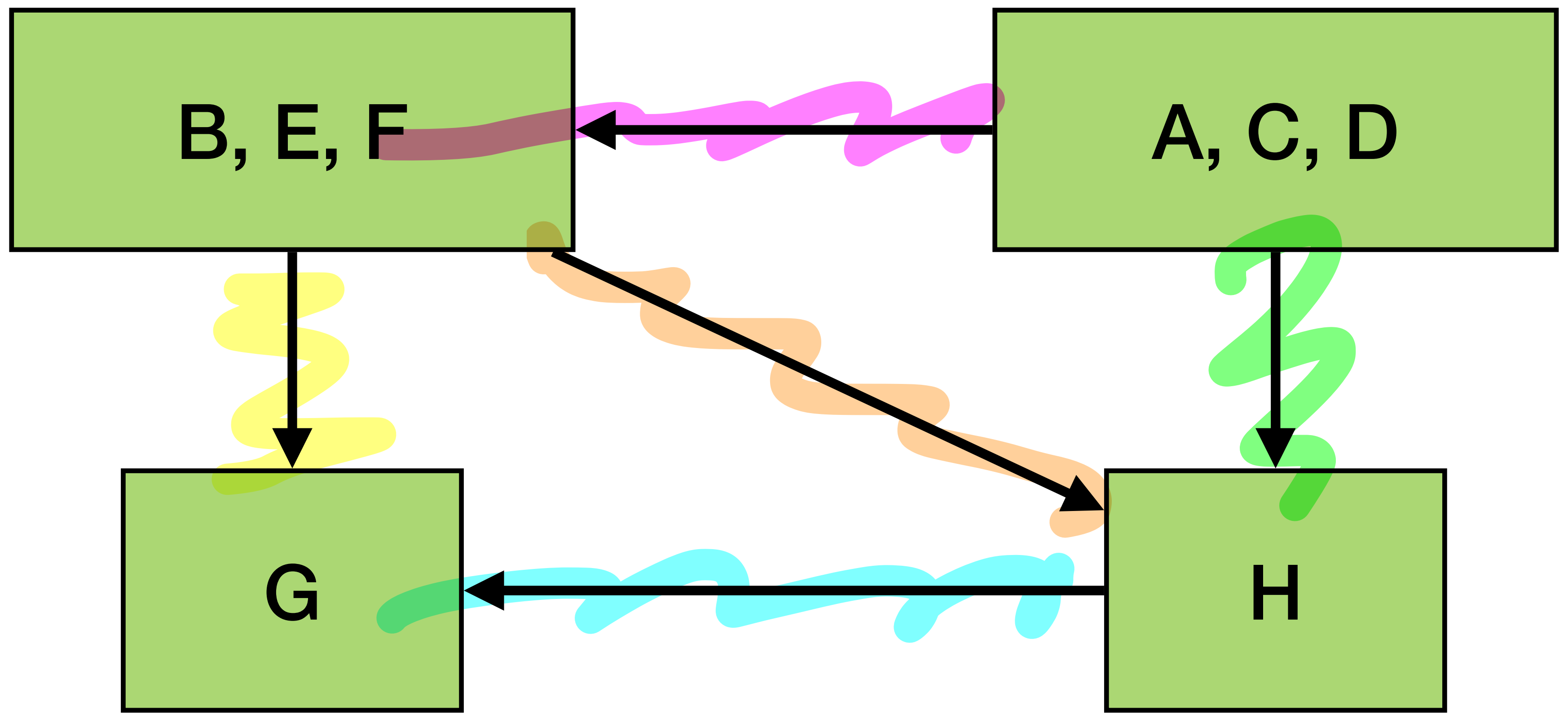
Graph of SCCs

Meta-graph of SCCs

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

- Vertices of G^{SCC} are $S_1, S_2, ... S_k$
- 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.



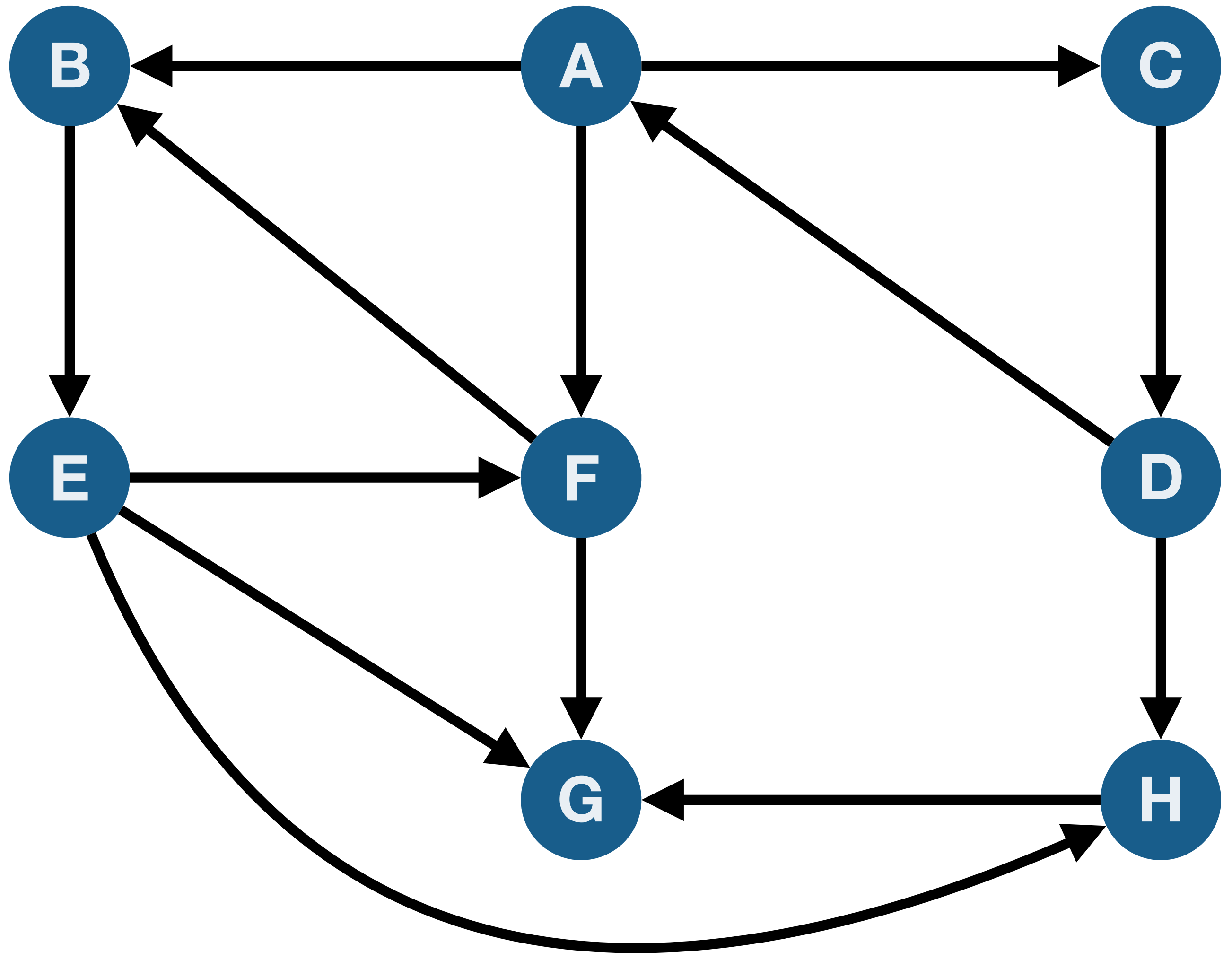


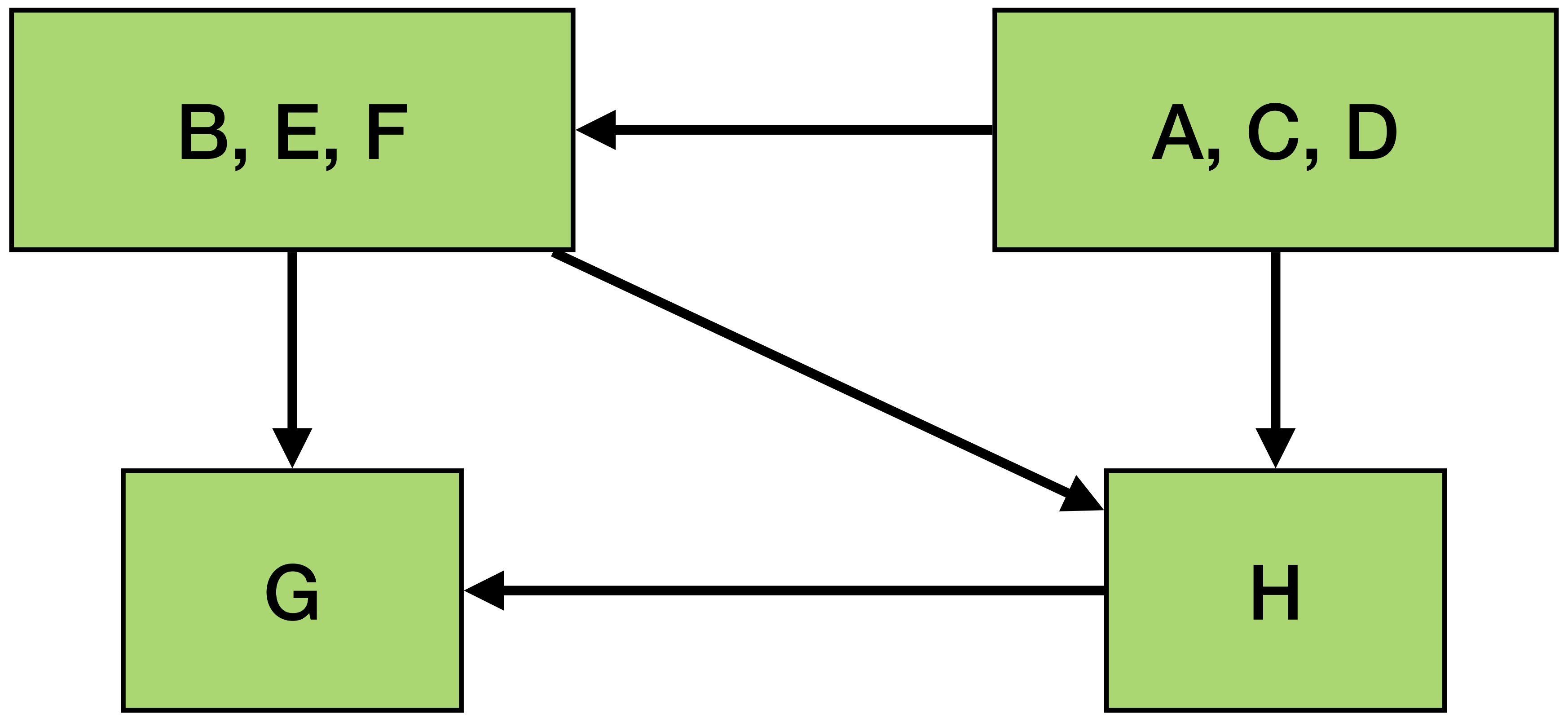
Graph of SCCs

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 as G^{SCC}:

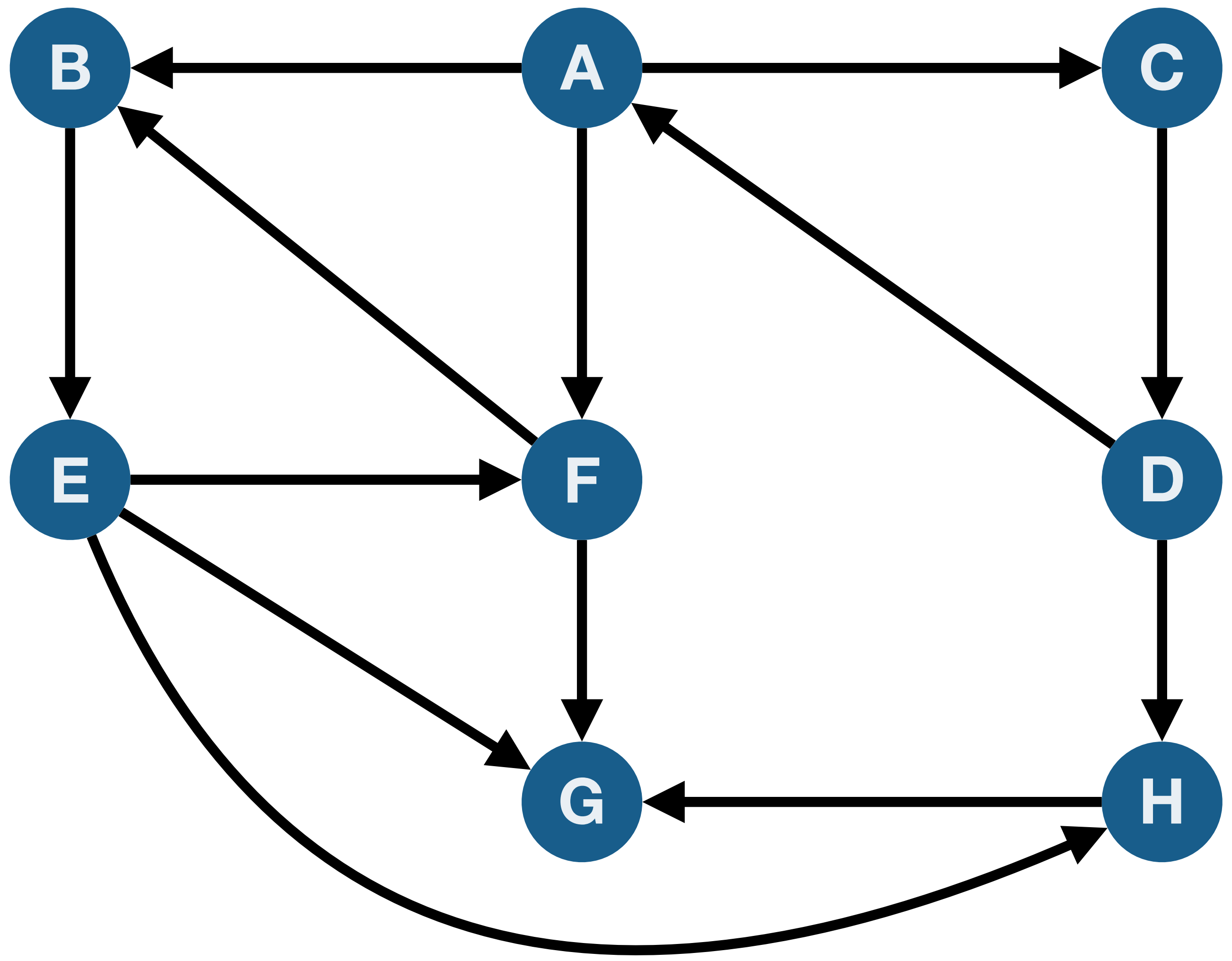
- Vertices of G^{SCC} are $S_1, S_2, \ldots S_k$
- There is an edge (S_i, S_j) if there is some $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge in G.

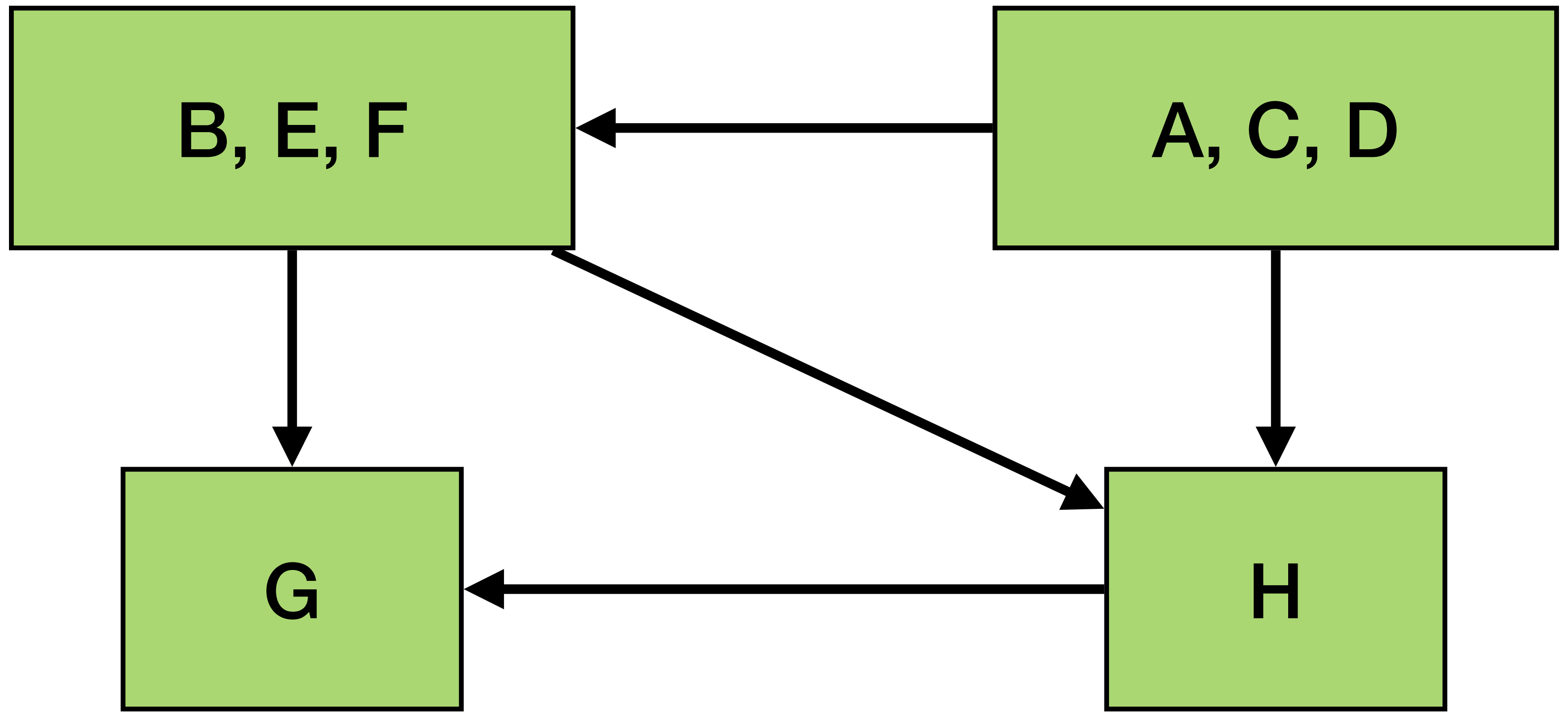




For any graph G, the graph G^{SCC} has no directed cycle! ← Fact! Proof? Exercise.

Connected structure of a directed graph





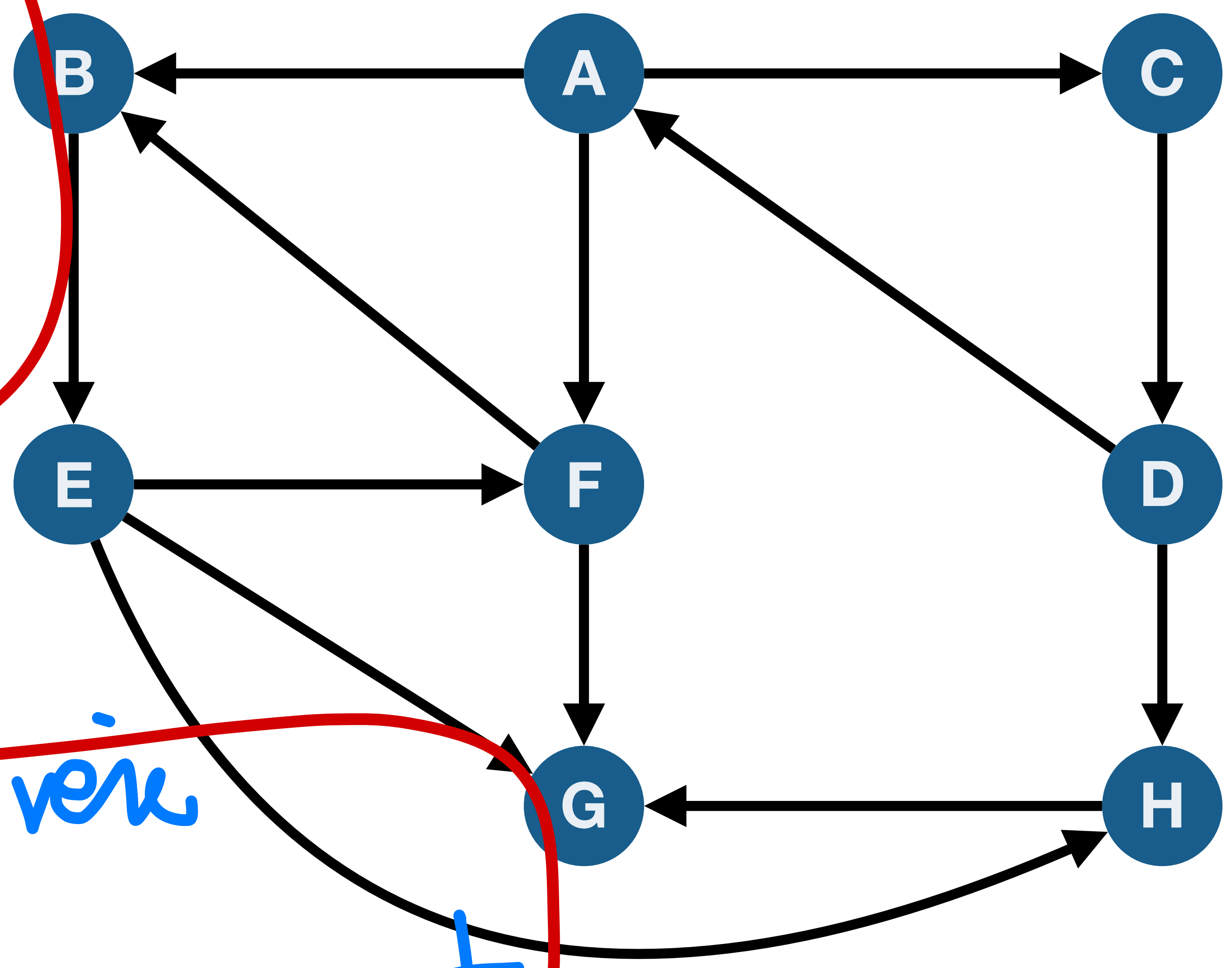
Reminder G^{SCC} is created by collapsing every strong connected component to a single vertex.

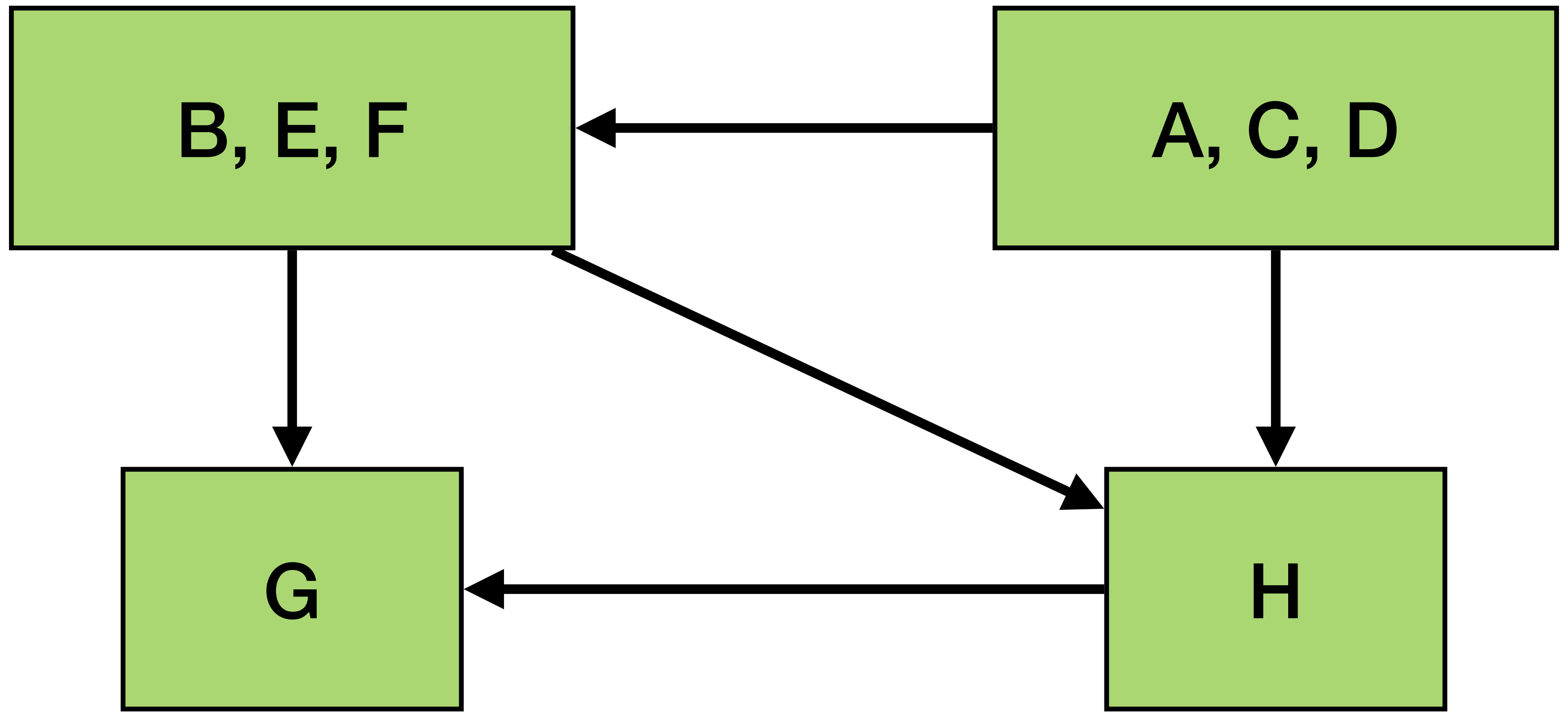
Connected structure of a directed graph

if DAG $\Downarrow$ $\exists$ TopSort

+

TopSort given by DFS w/ pre-post





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

Idea

Wishful thinking algorithm

- Let u be a vertex in a *sink* SCC of G^{SCC}.

Linear-time Algorithm for SCCs

Idea

Wishful thinking algorithm

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

Linear-time Algorithm for SCCs

Idea

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.

Linear-time Algorithm for SCCs

Idea

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.

Linear-time Algorithm for SCCs

Idea

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!

Linear-time Algorithm for SCCs

Idea

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 $SCC(u)$.

Linear-time Algorithm for SCCs

Idea

How to find ??

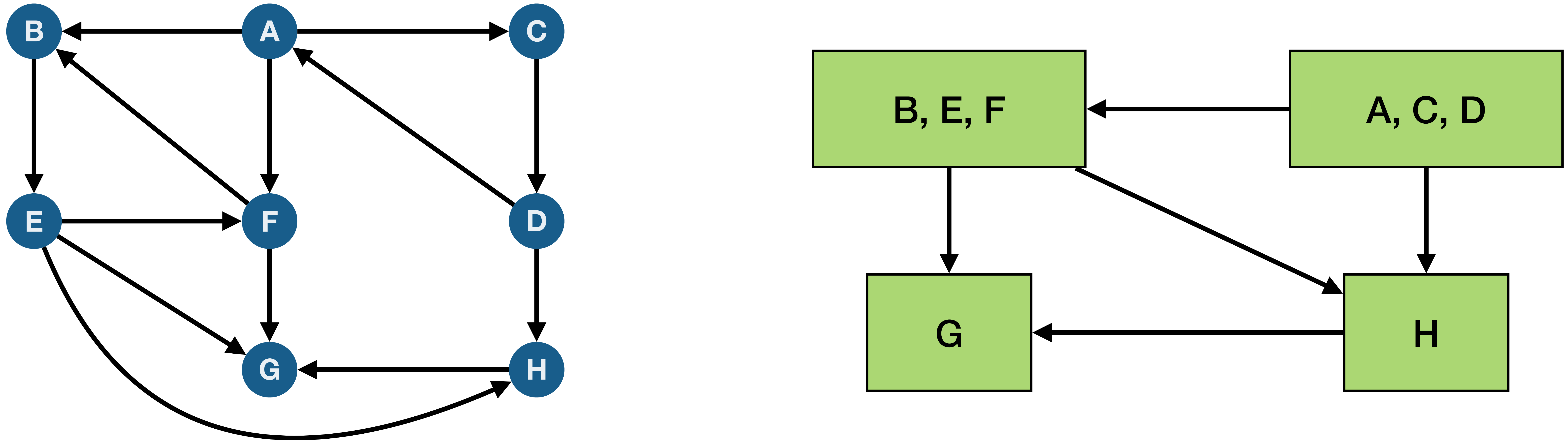
Wishful thinking algorithm

- 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

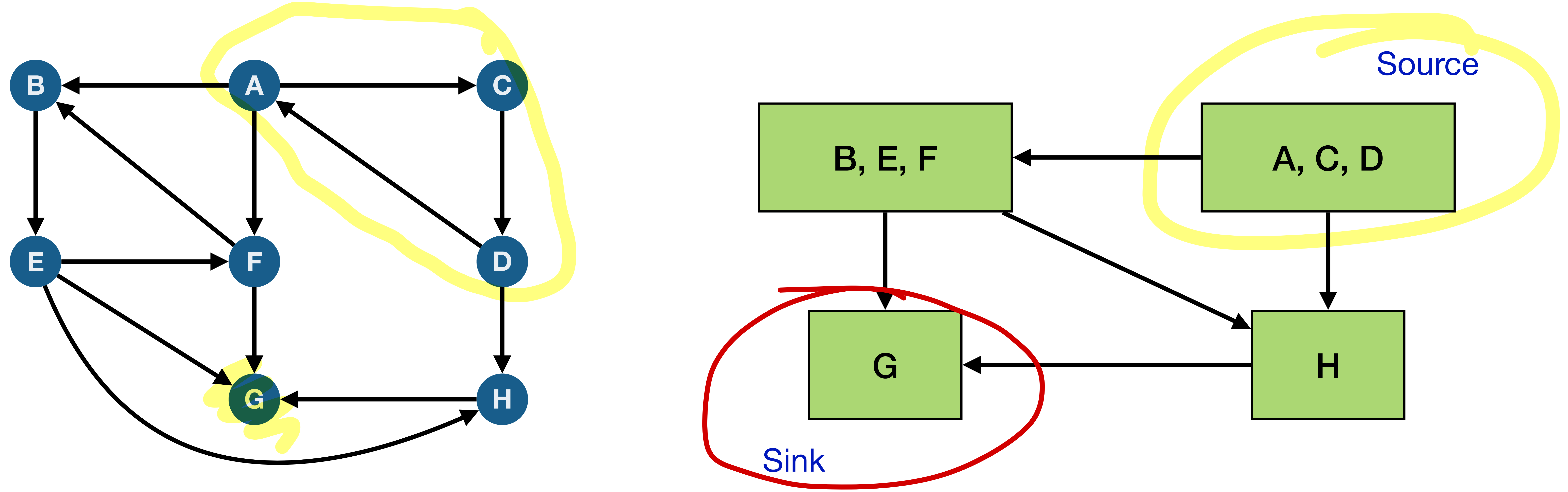
- $DFS(u)$ only visits vertices (and edges) in $SCC(u)$ since there are no edges coming out of a sink!
- $DFS(u)$ takes time proportional to size of $SCC(u)$.
- Therefore, total time $O(n + m)$!

Connected structure of a directed graph



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

Connected structure of a directed graph



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

On the right the SCC $\{G\}$ is a sink and the SCC $\{A, C, D\}$ is a source.

Questions

Okay but …

Think: a chicken or egg problem.

Question: 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}?

encoded in details.

Questions

Okay but …

Question: 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!

Questions

Okay but …

Question: 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!

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}.

Questions

Okay but …

Question: 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}?

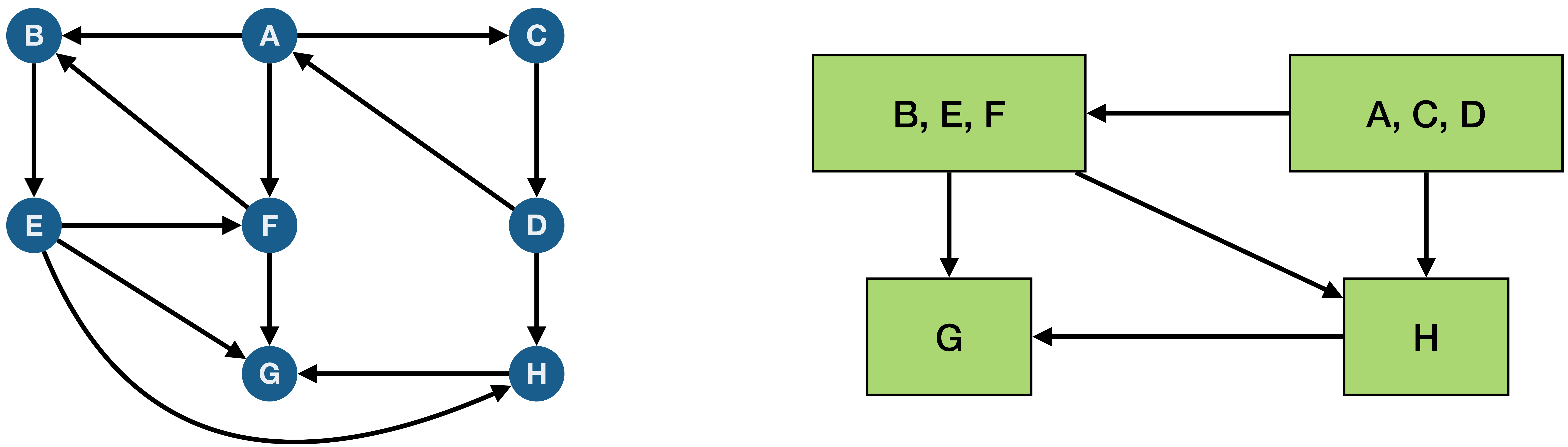
Answer: $DFS(G)$ gives some information!

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}.

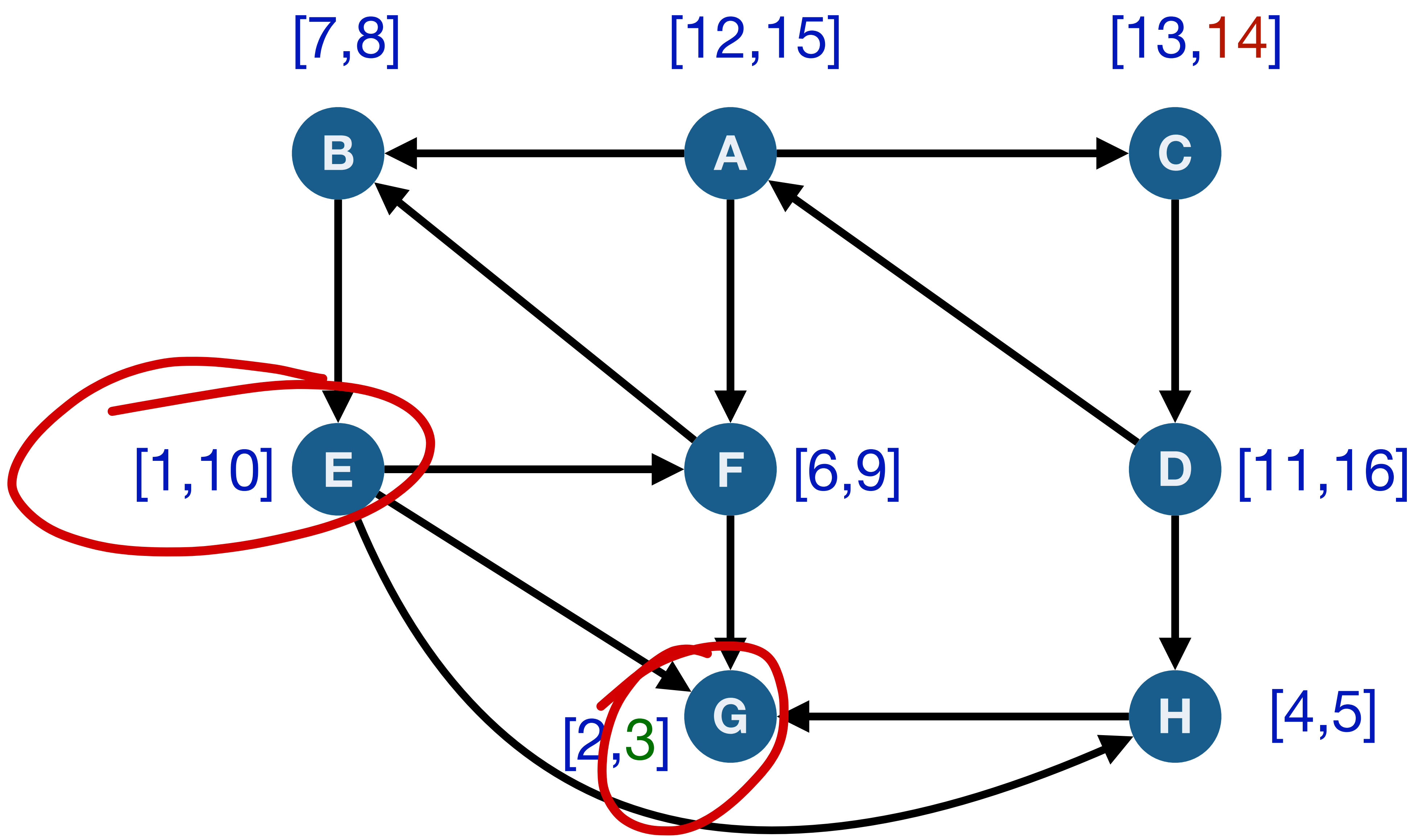
↳ See piazza about why G^{rev} is necessary.

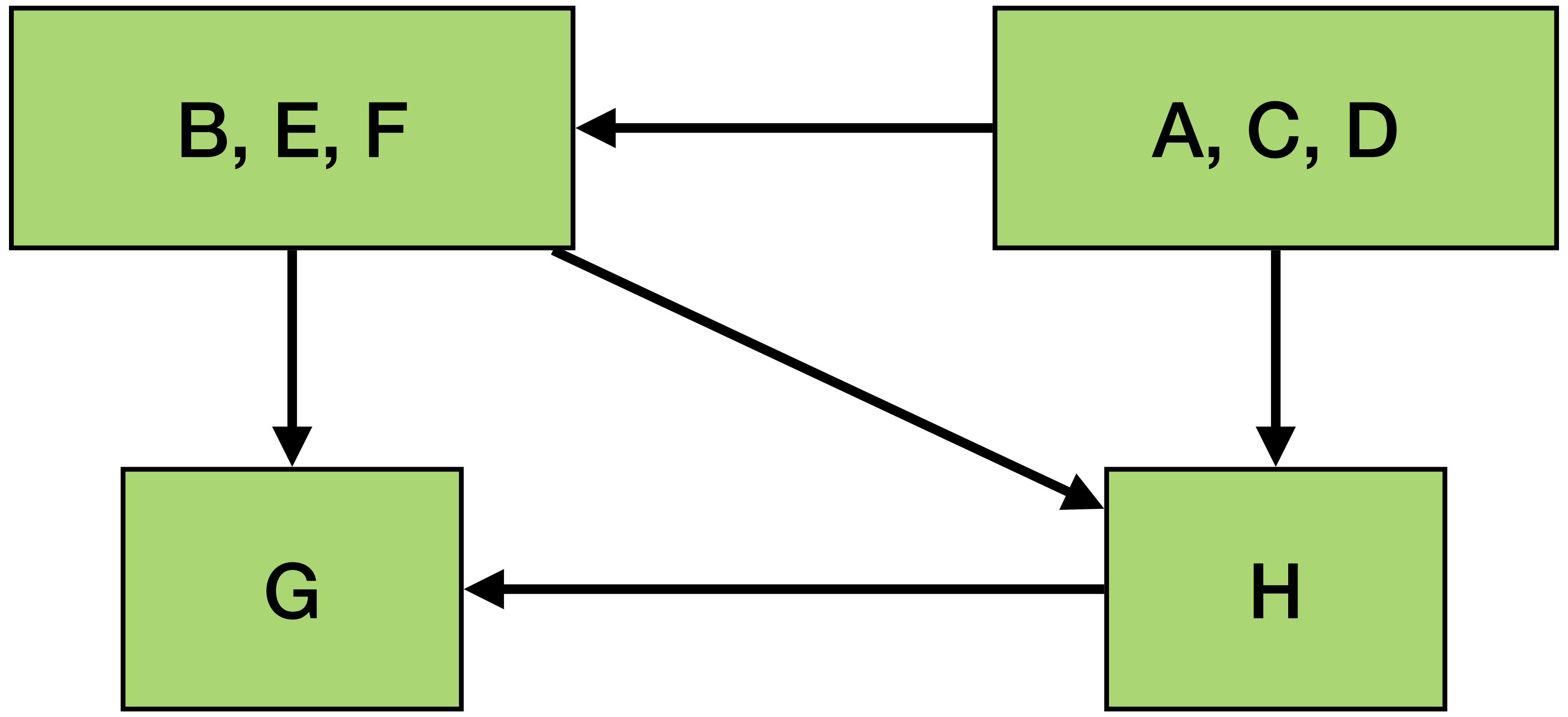
Connected structure of a directed graph



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

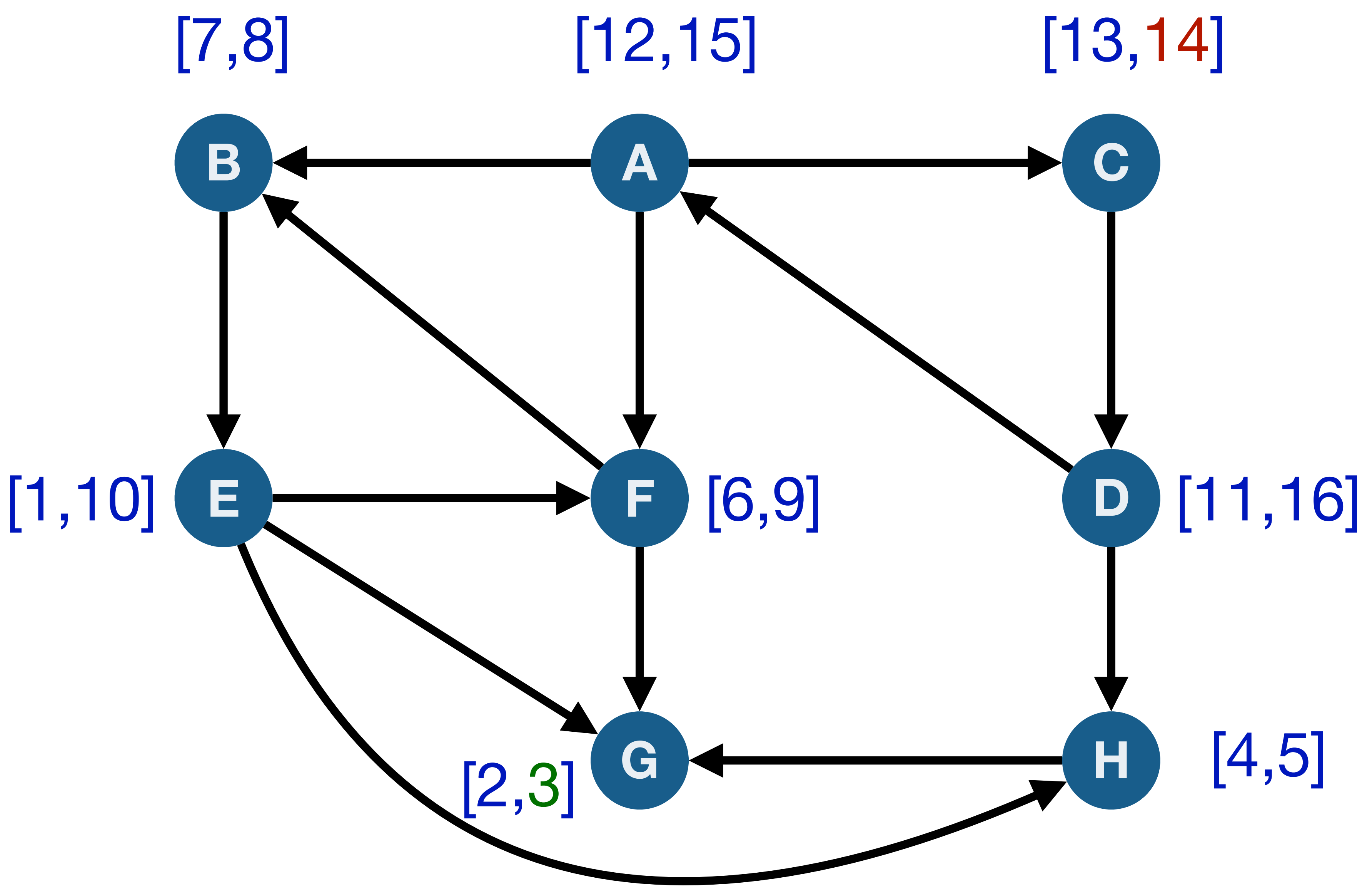
Connected structure of a directed graph

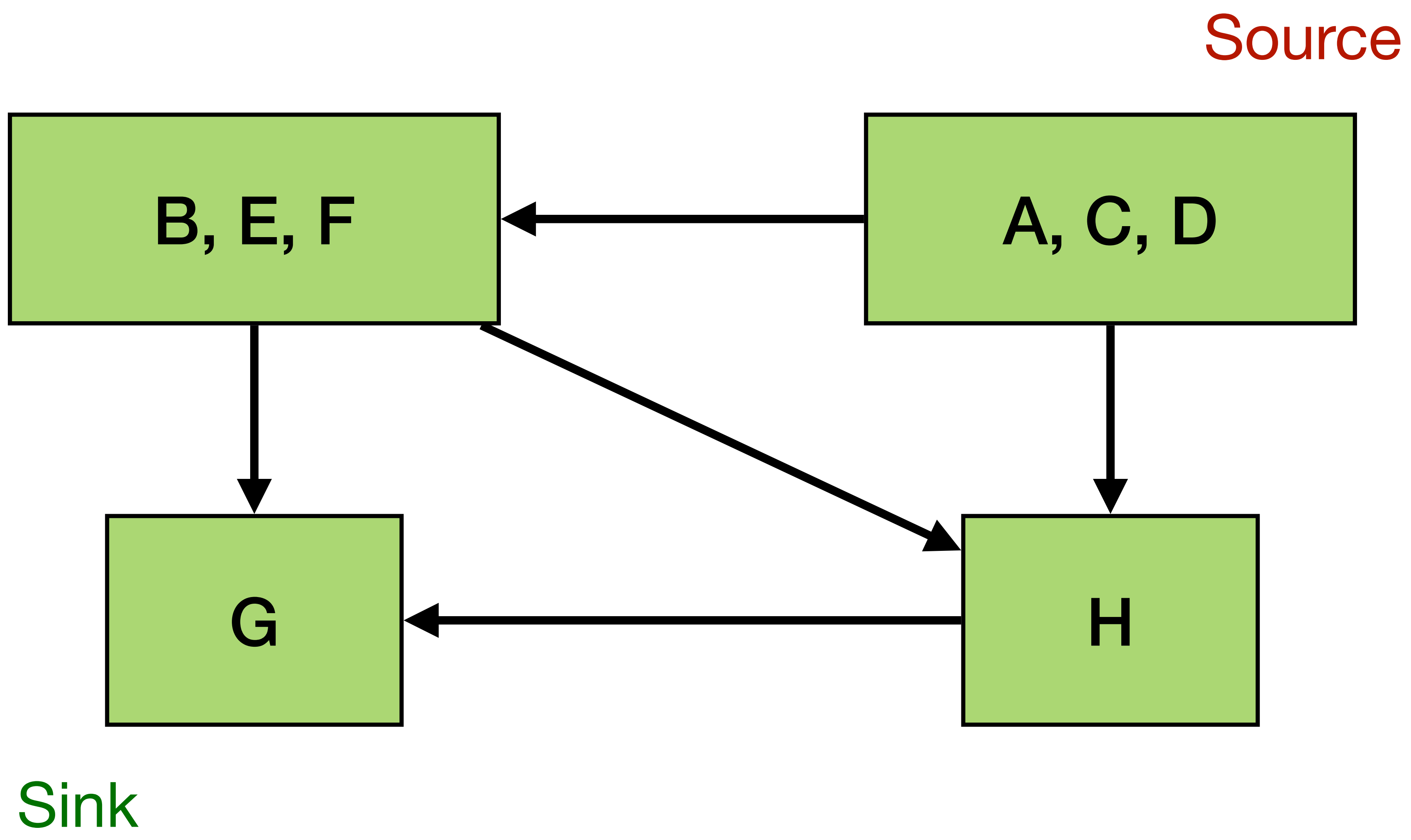




Reminder G^{SCC} is created by collapsing every strong connected component to a single vertex.

Connected structure of a directed graph





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

On the right the SCC $\{G\}$ is a sink and the SCC $\{A, C, D\}$ is a source.

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
```

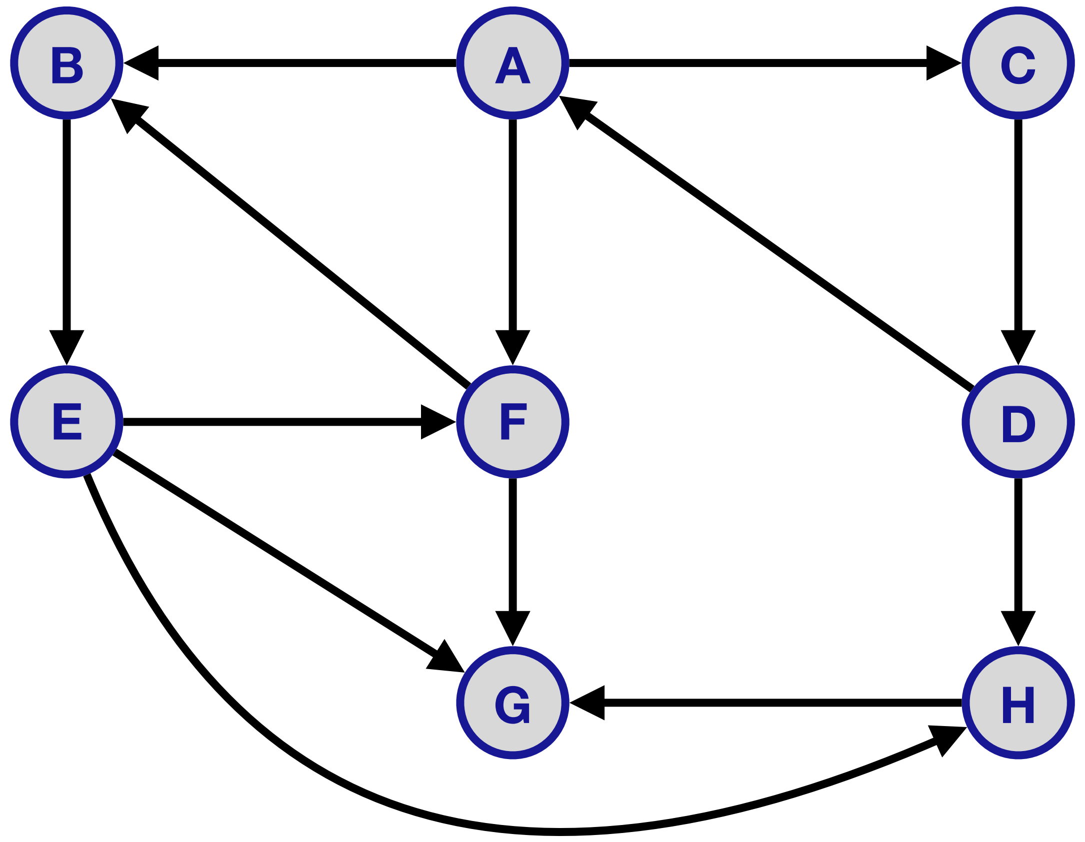
Linear Time SCC Algorithm

```
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.

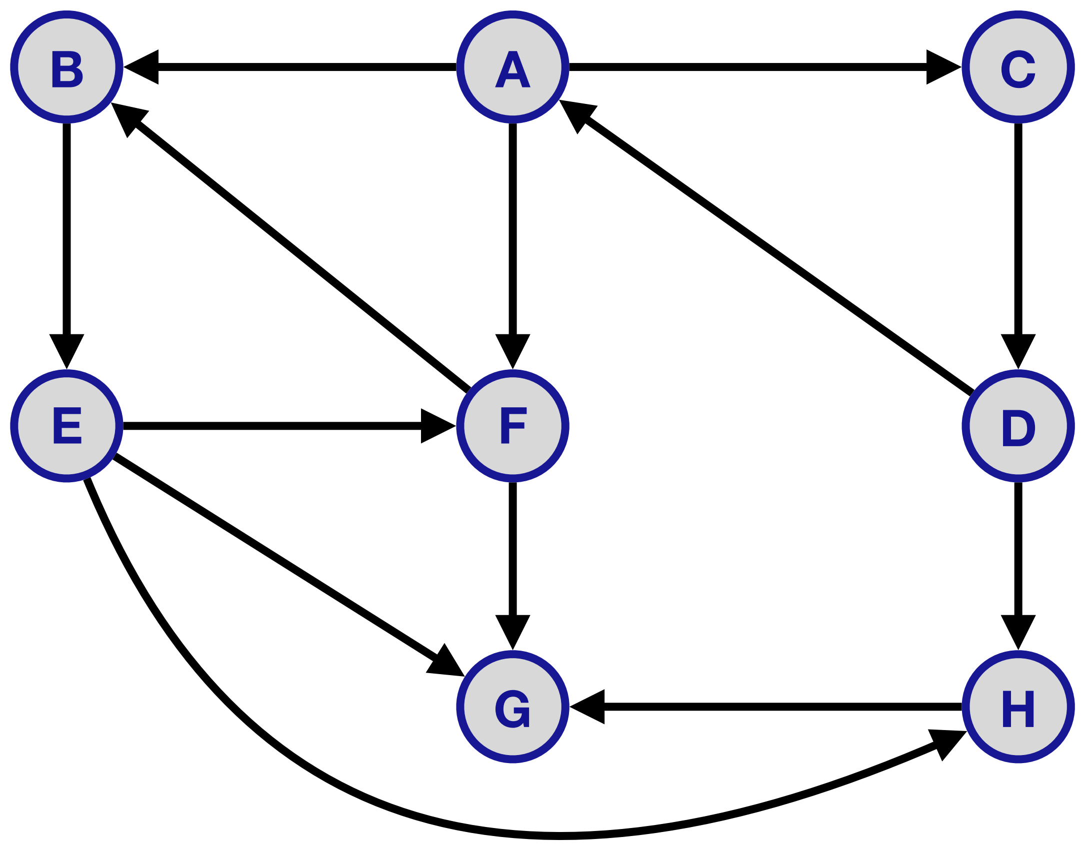
Linear Time Algorithm - An Example

Graph G

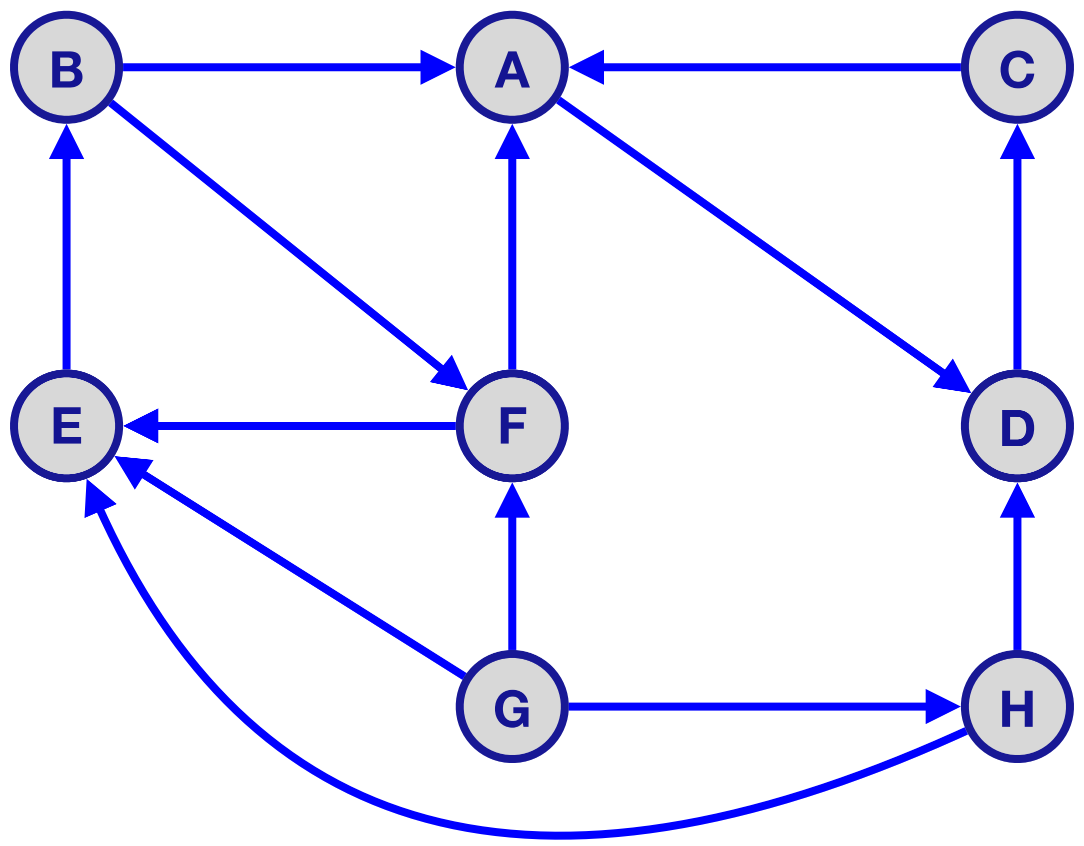


Linear Time Algorithm - An Example

Graph G

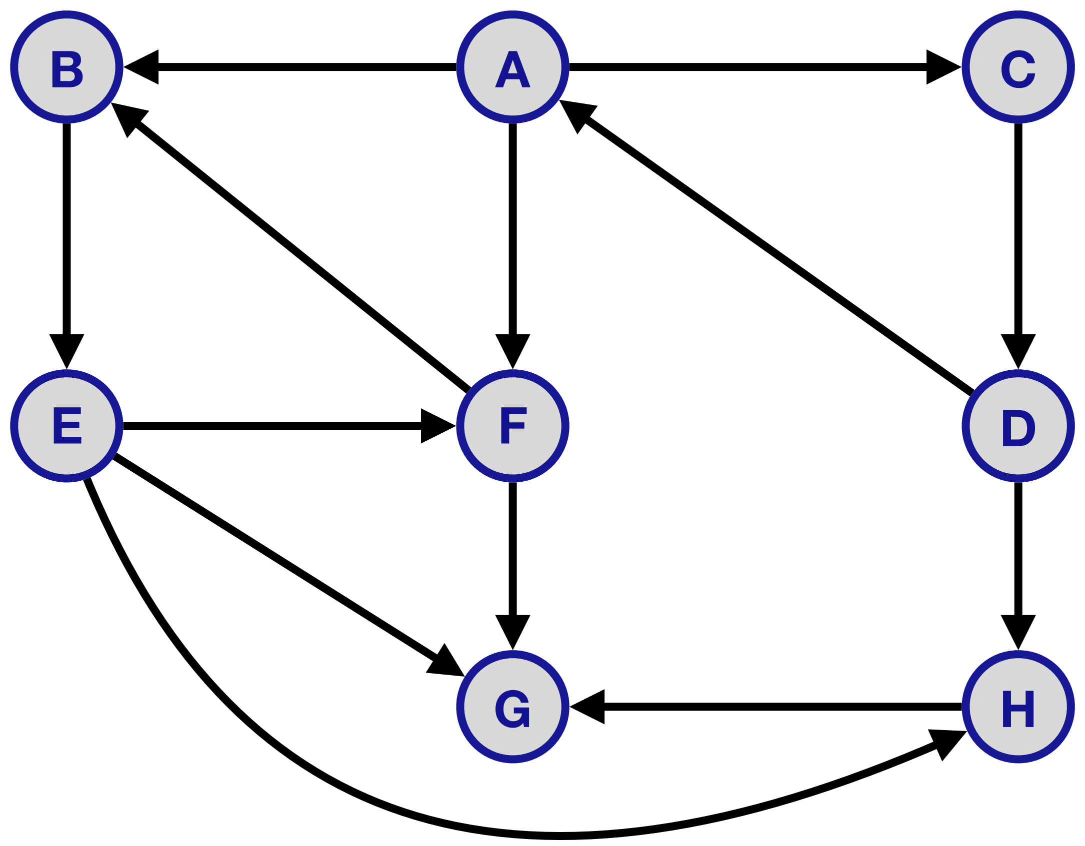


Reverse Graph G^{rev}

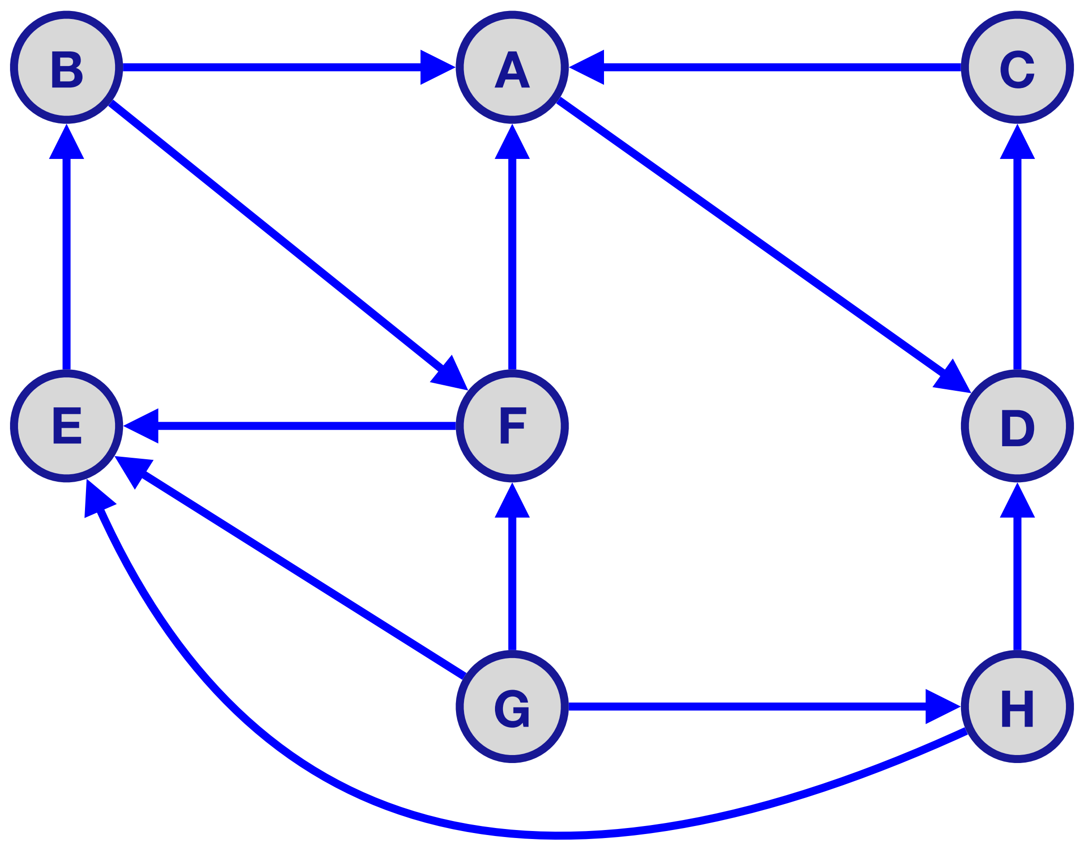


Linear Time Algorithm - An Example

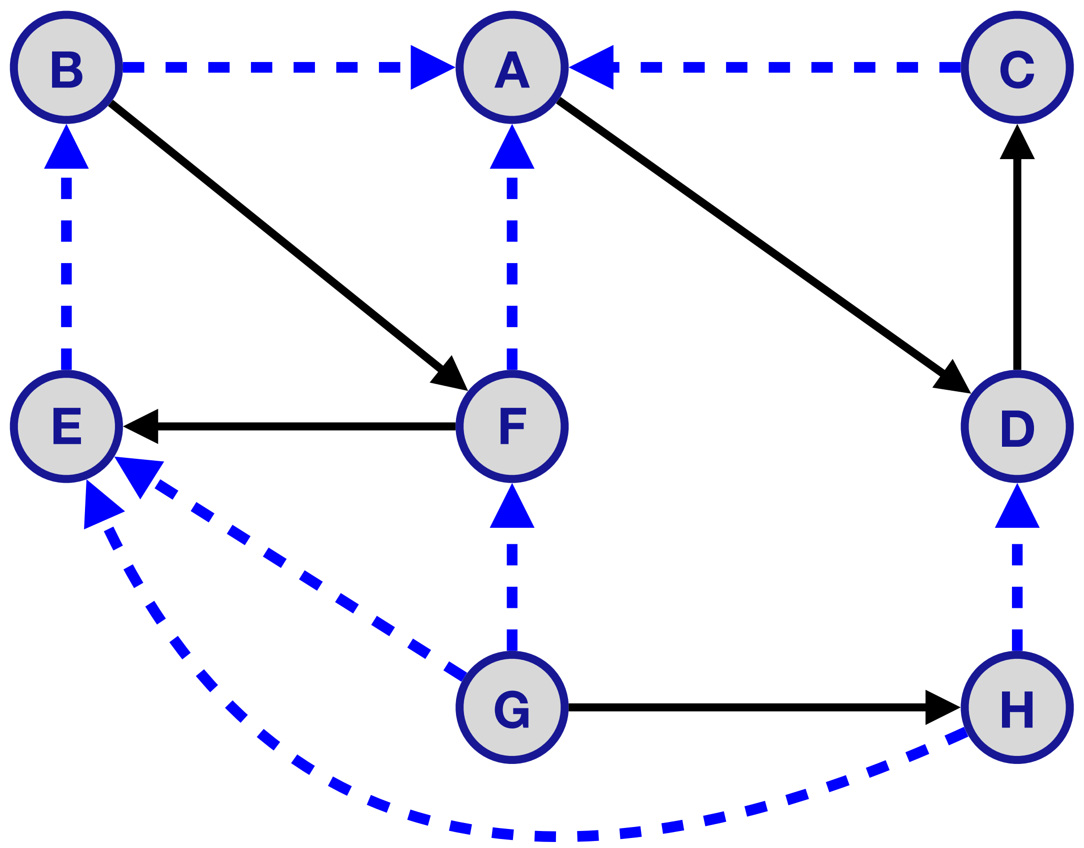
Graph G



Reverse Graph G^{rev}

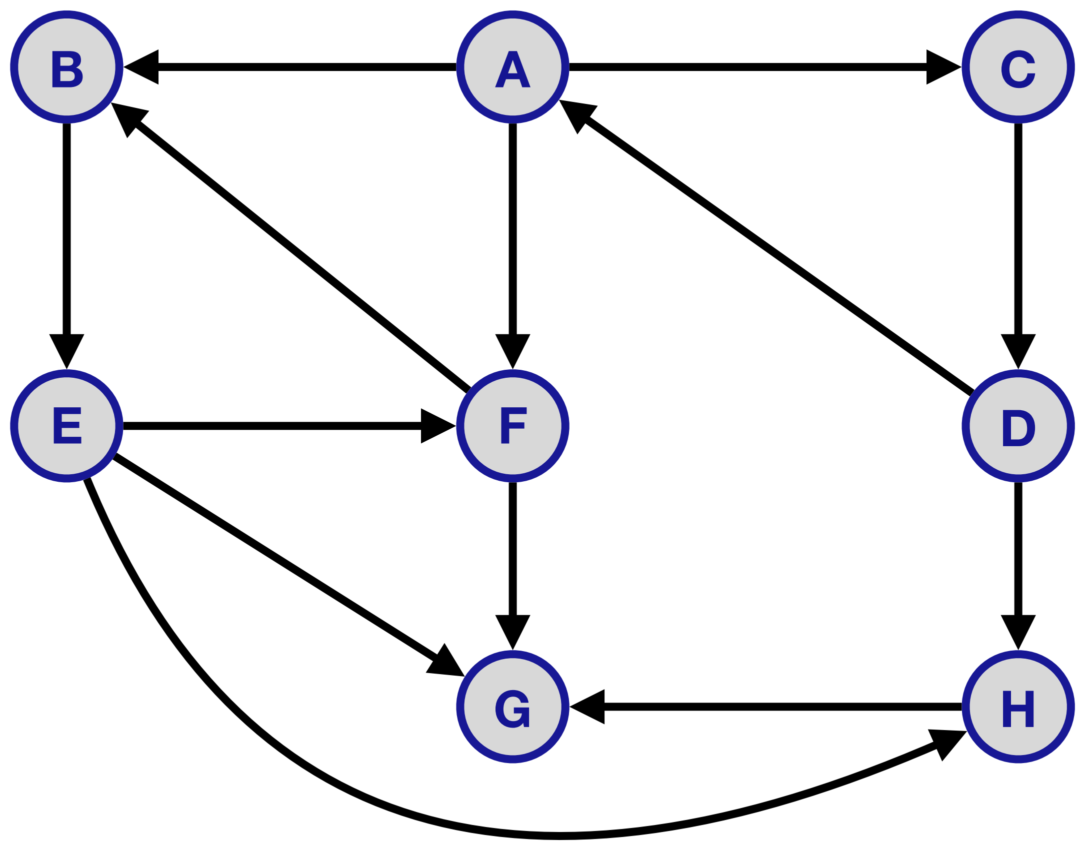


DFS of reverse graph

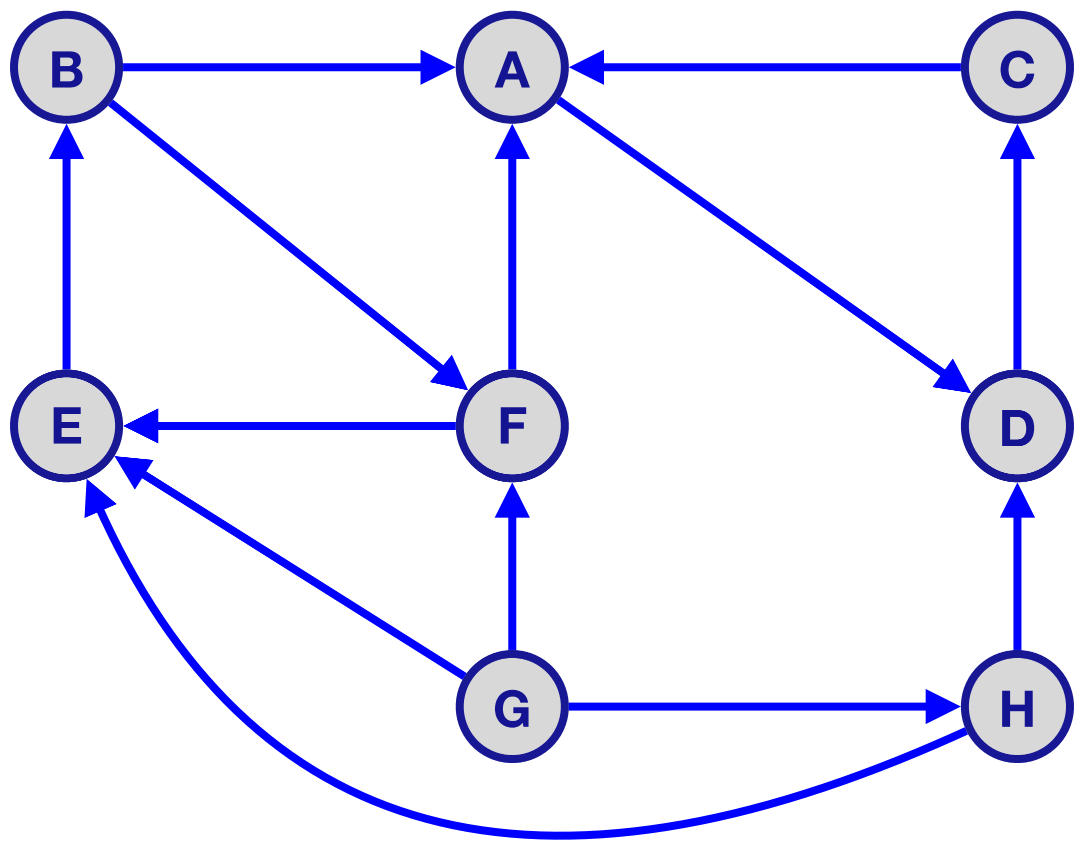


Linear Time Algorithm - An Example

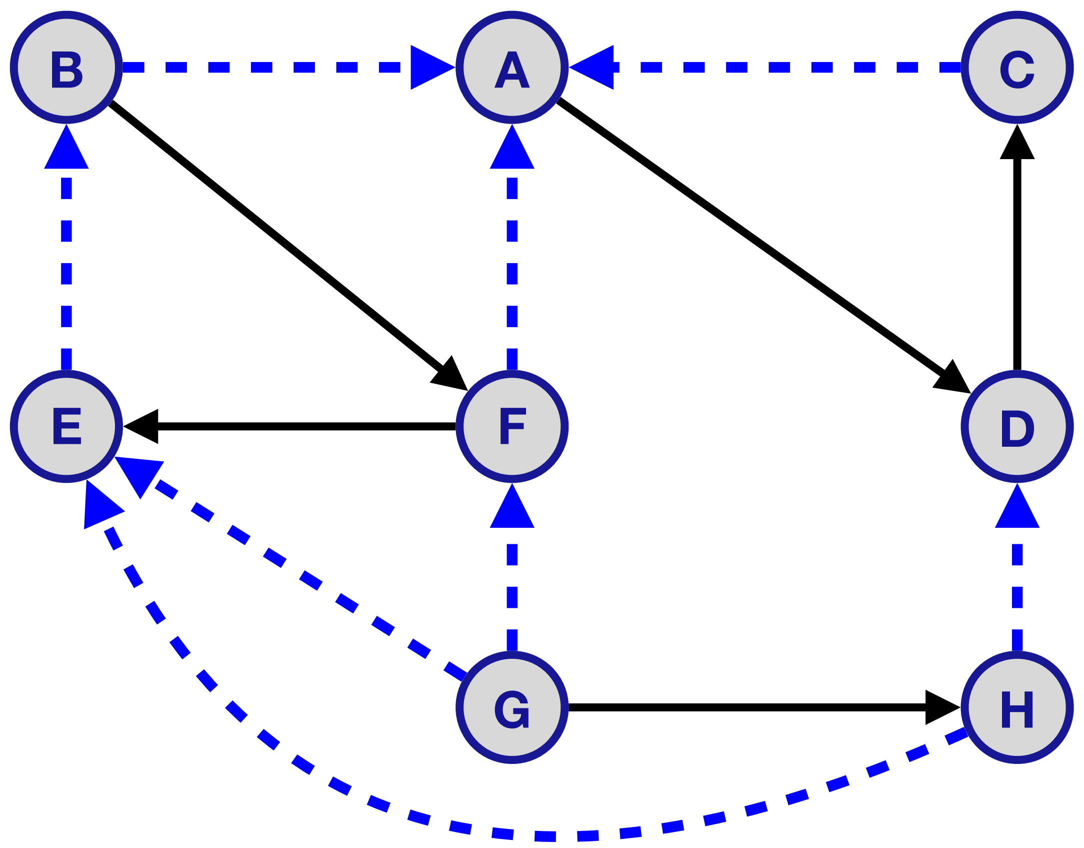
Graph G



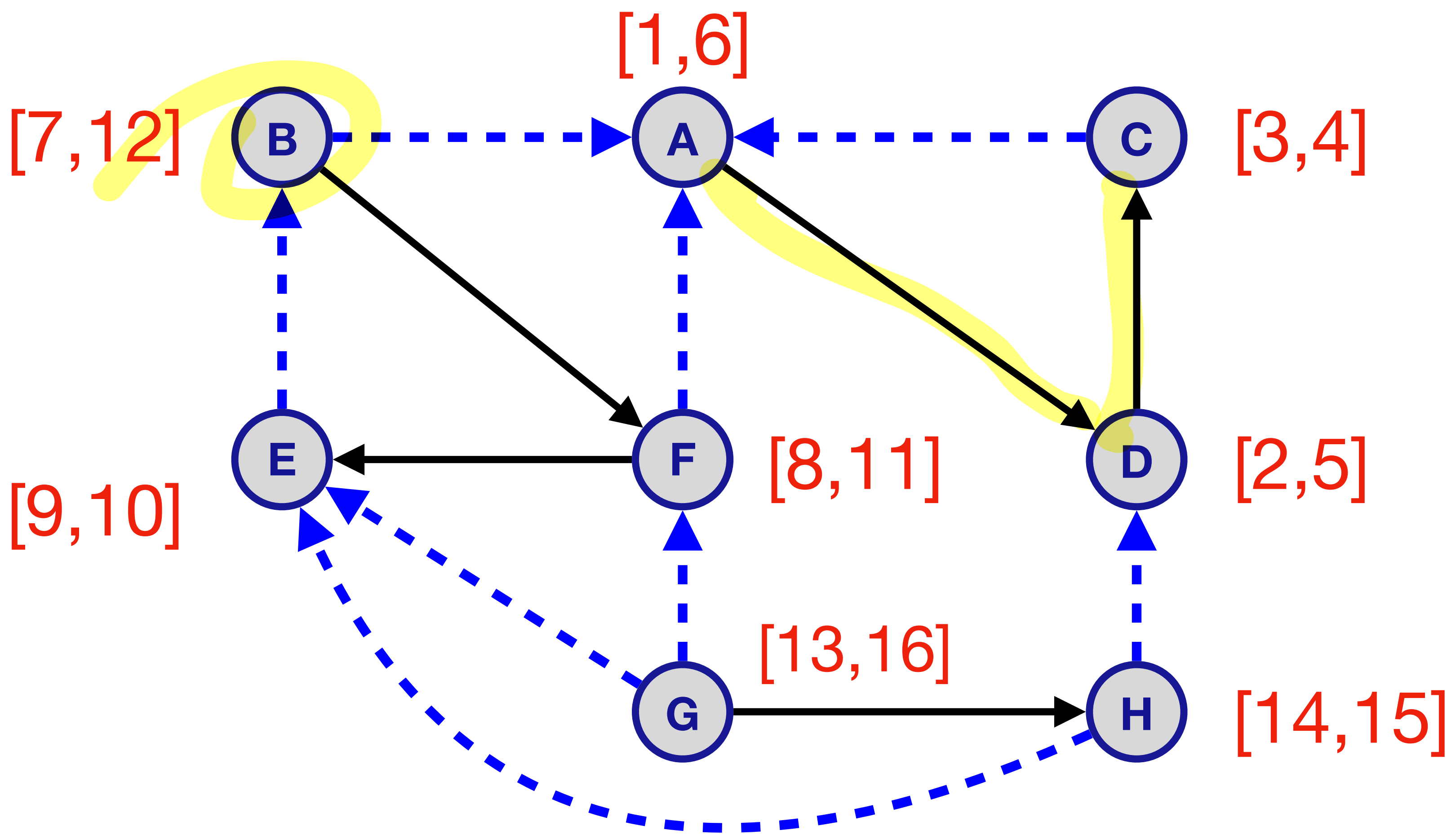
Reverse Graph G^{rev}



DFS of reverse graph

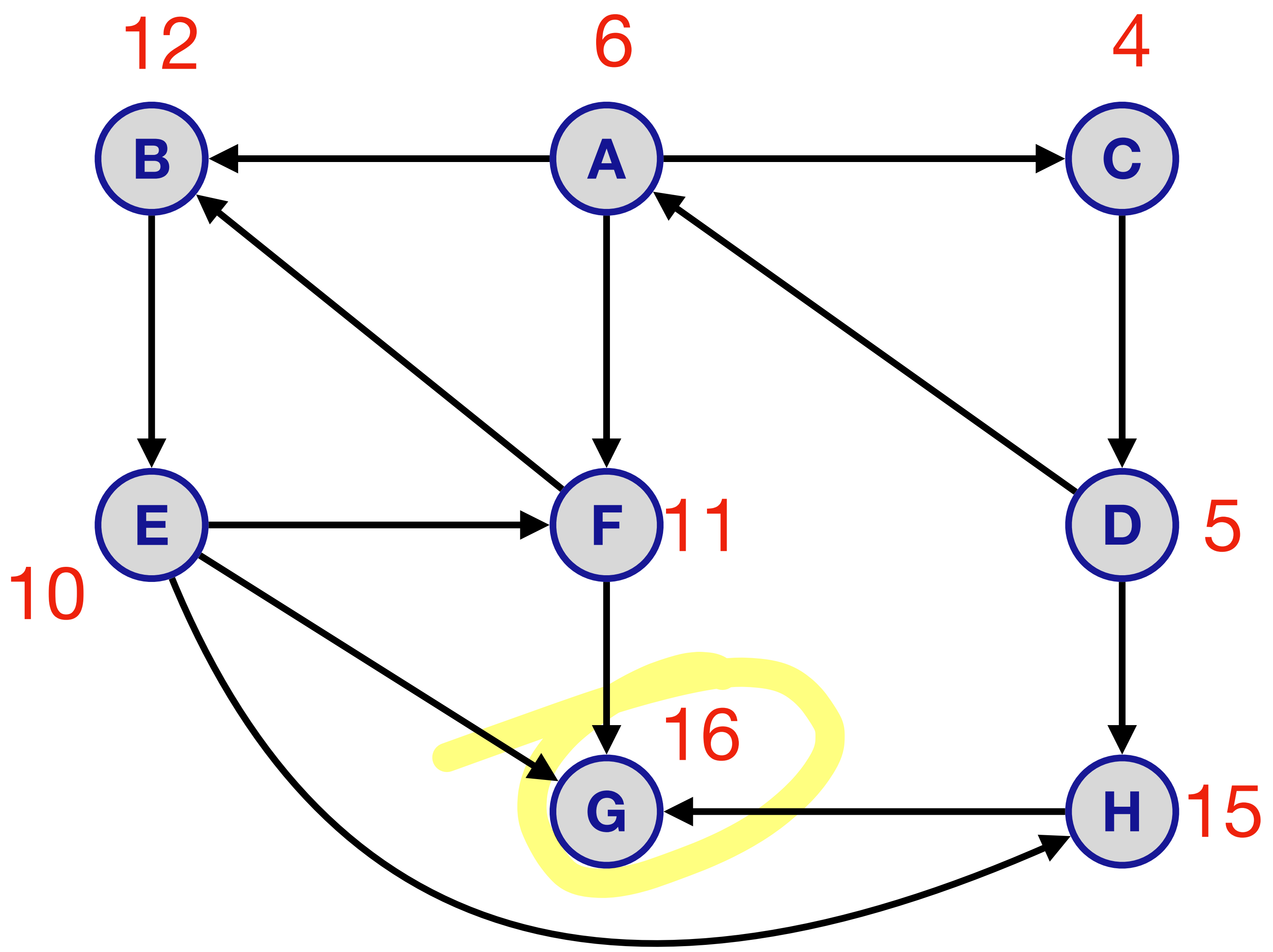


Pre/Post **DFS** numbering
of reverse graph



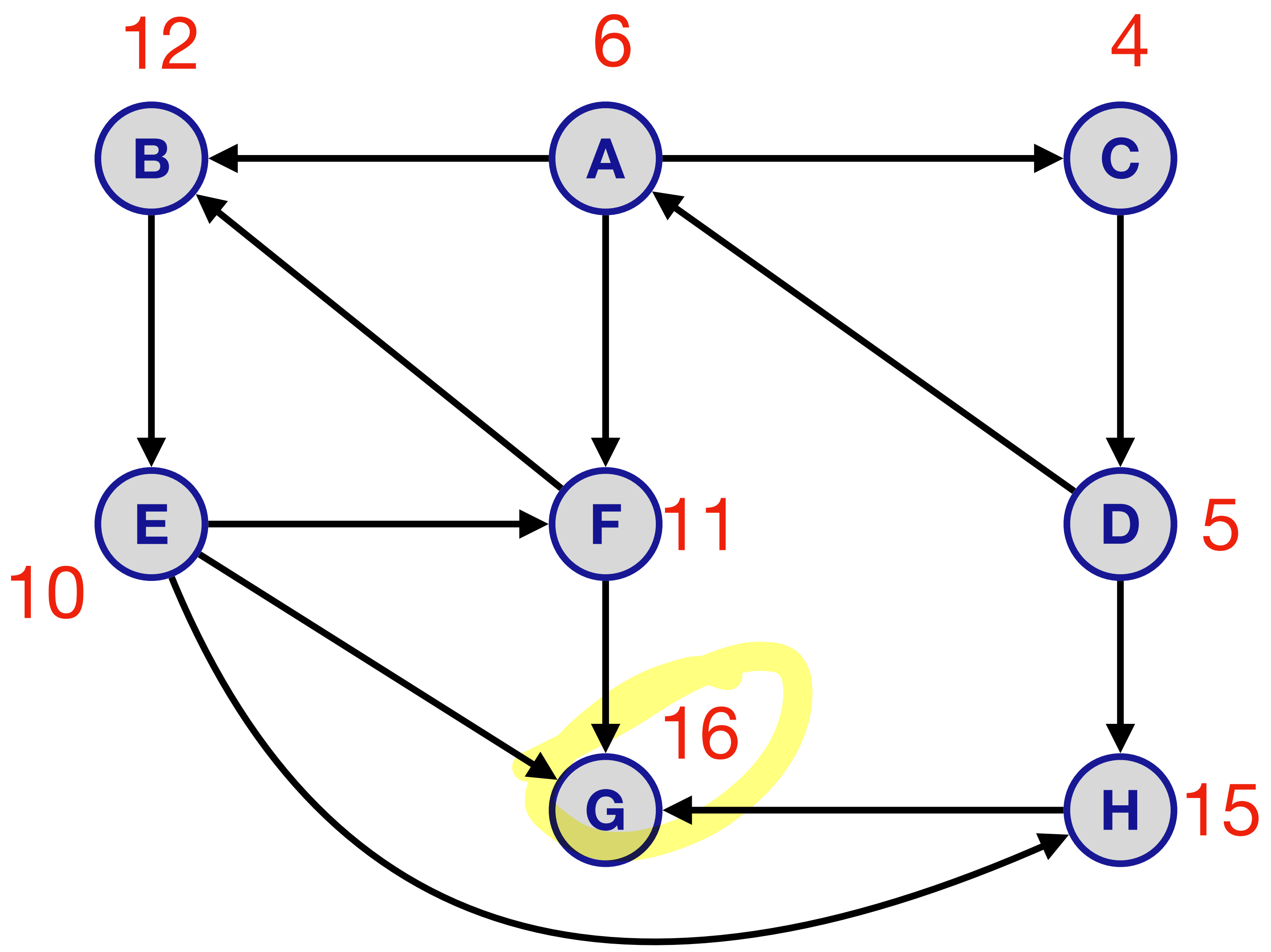
Linear Time Algorithm - An Example

G annotated with G^{rev}'s post numbers



Linear Time Algorithm - An Example

G annotated with G^{rev}'s post numbers

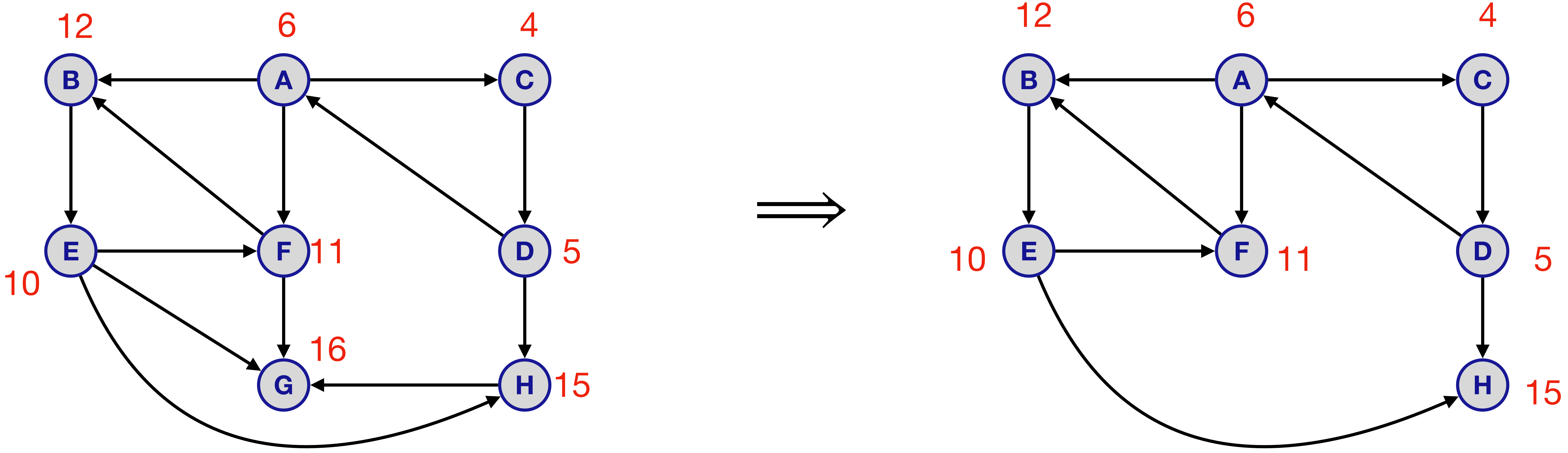


Do **DFS** from vertex G and remove it

Linear Time Algorithm - An Example

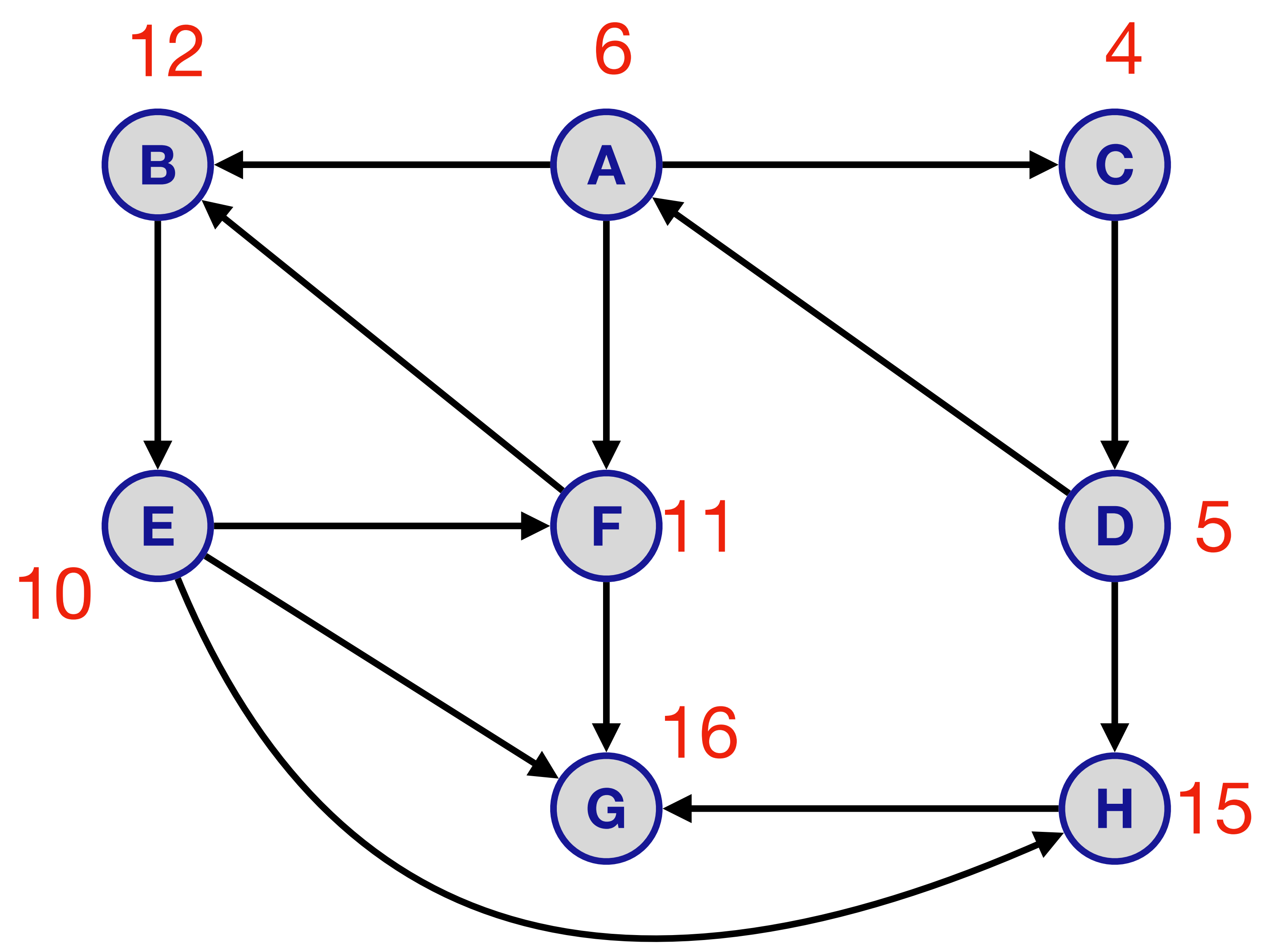
G annotated with G^{rev}'s post numbers

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



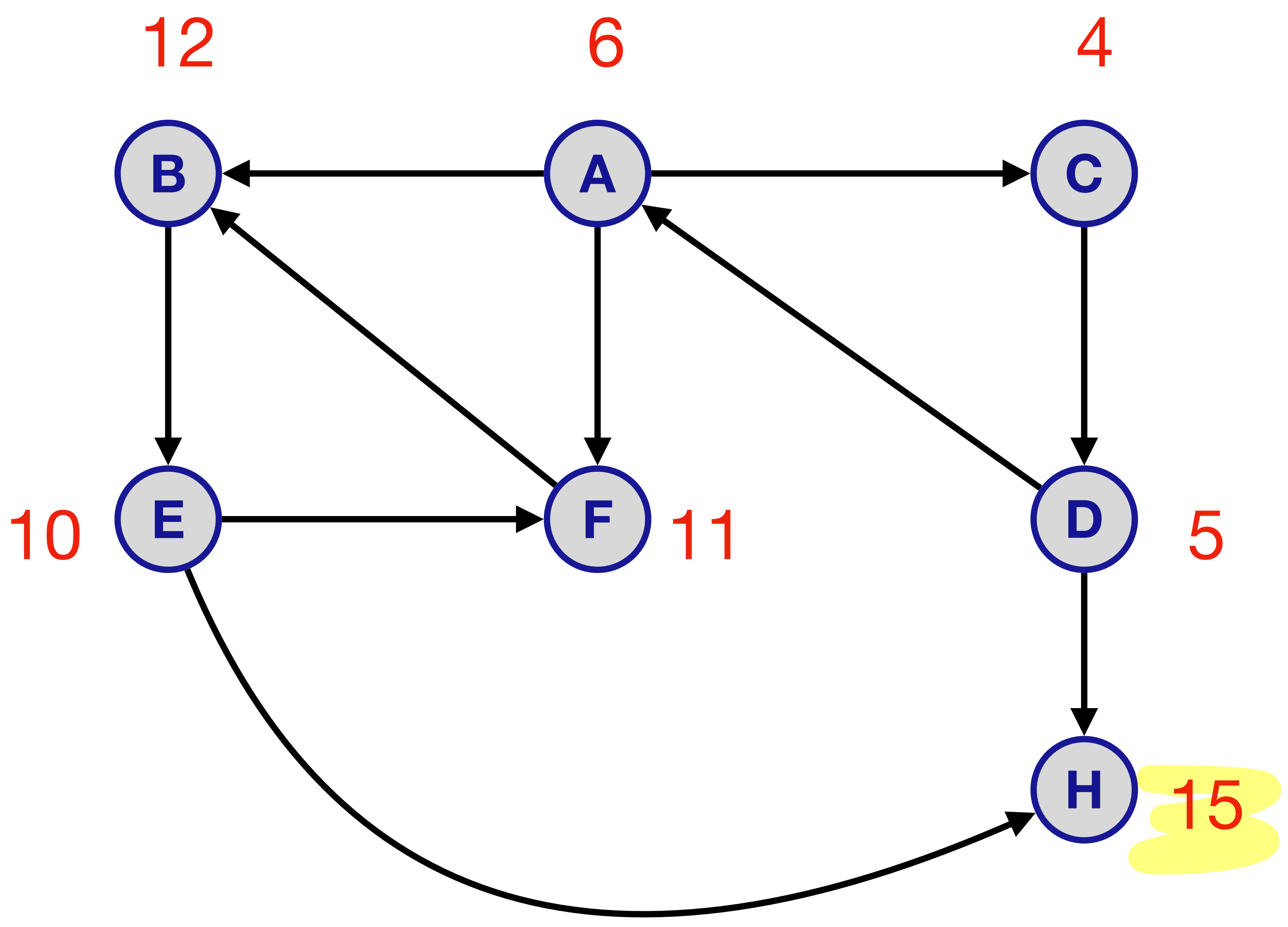
Linear Time Algorithm - An Example

G annotated with G^{rev}'s post numbers



$\Longrightarrow$

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

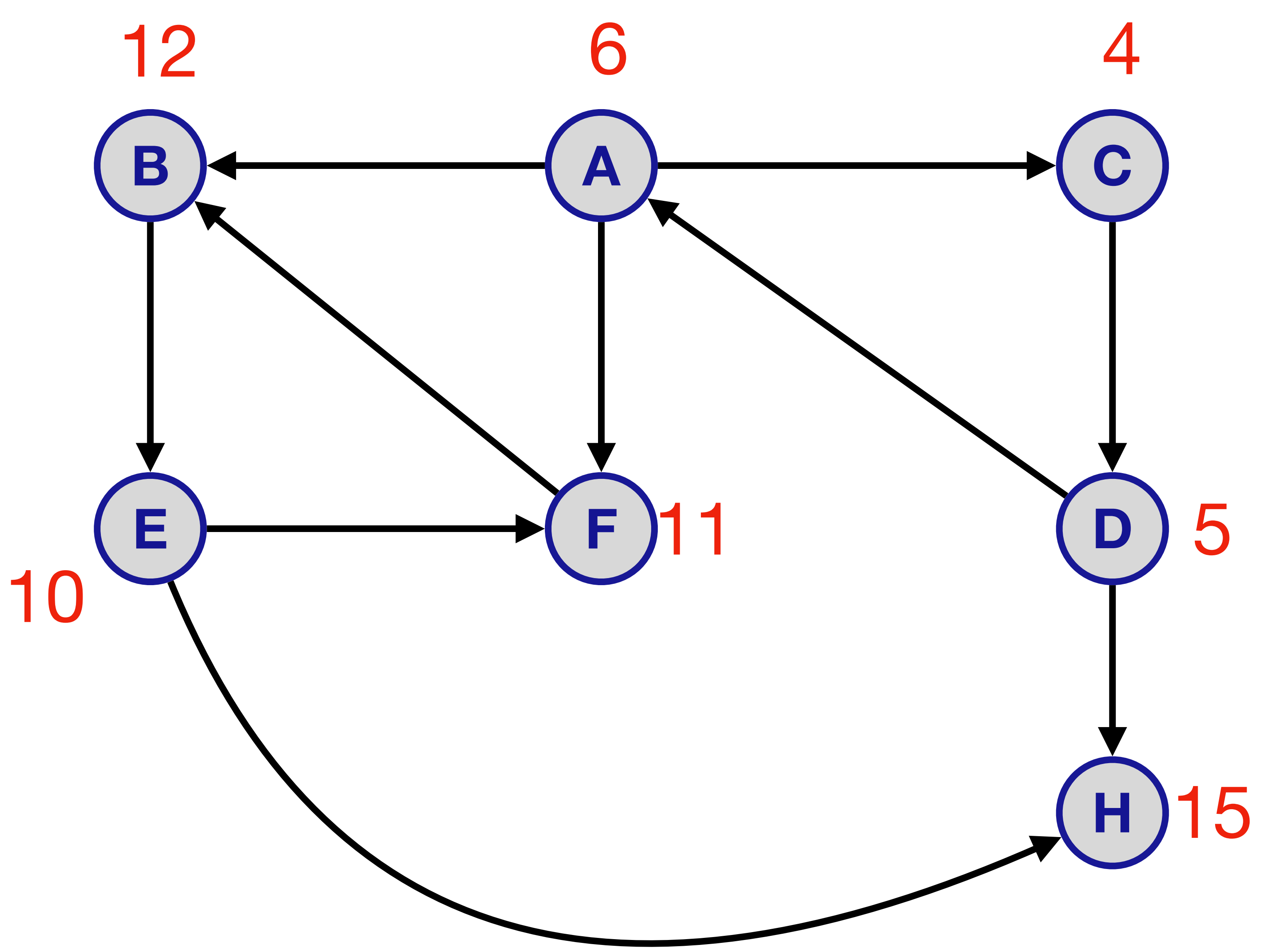


SCC computed:

{G}

Linear Time Algorithm - An Example

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

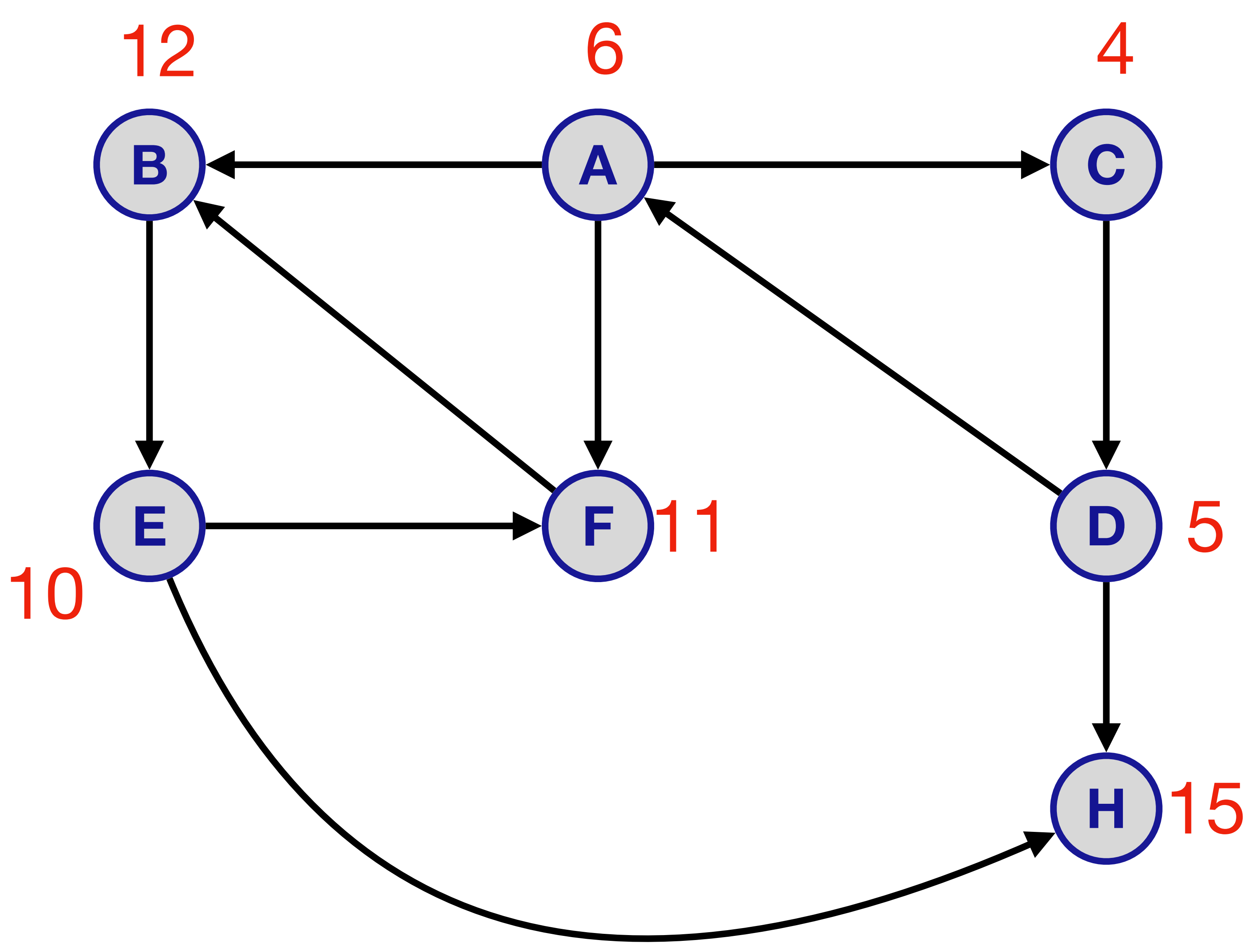


SCC computed:

{G}

Linear Time Algorithm - An Example

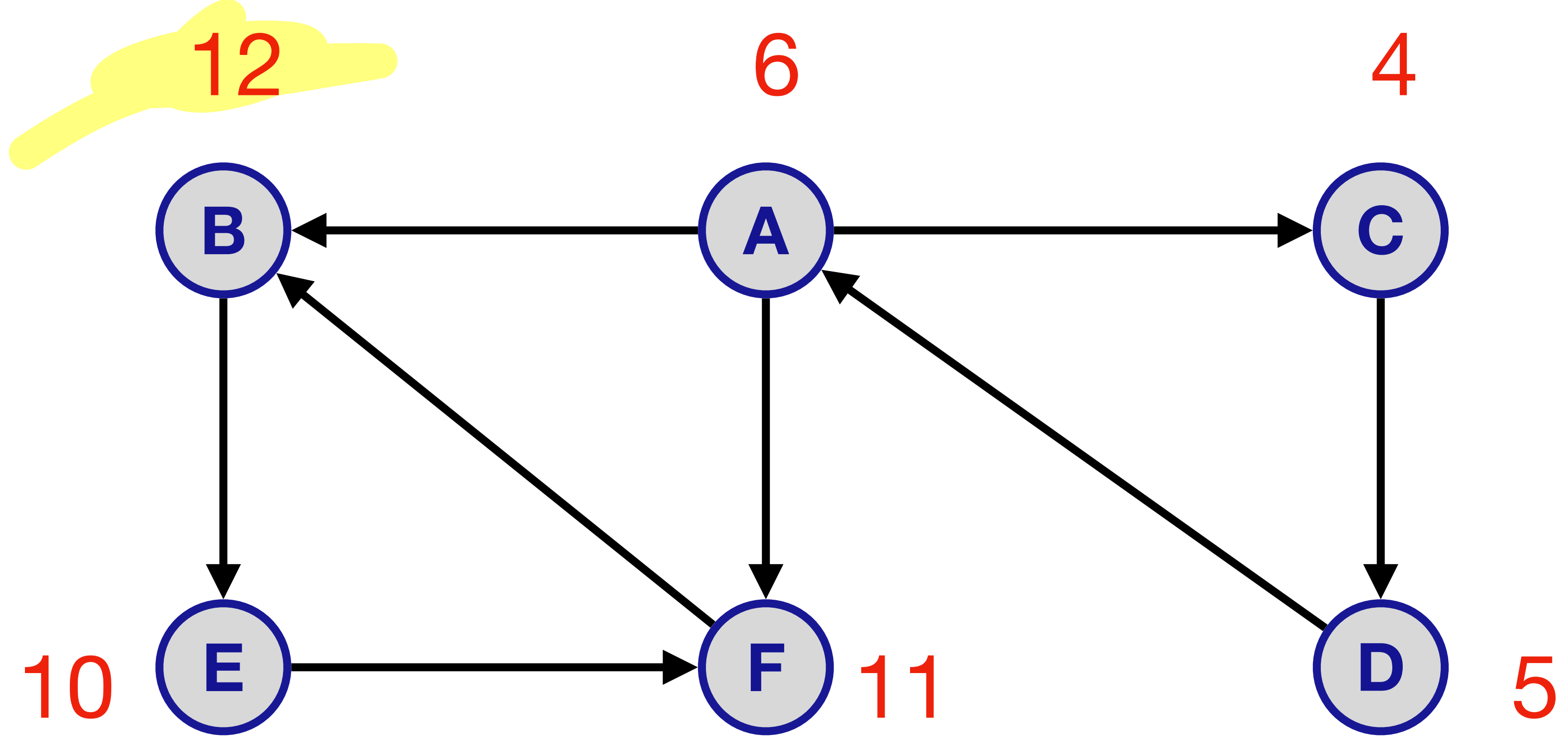
Do **DFS** from vertex H and remove it



SCC computed:

{G}

$\Longrightarrow$

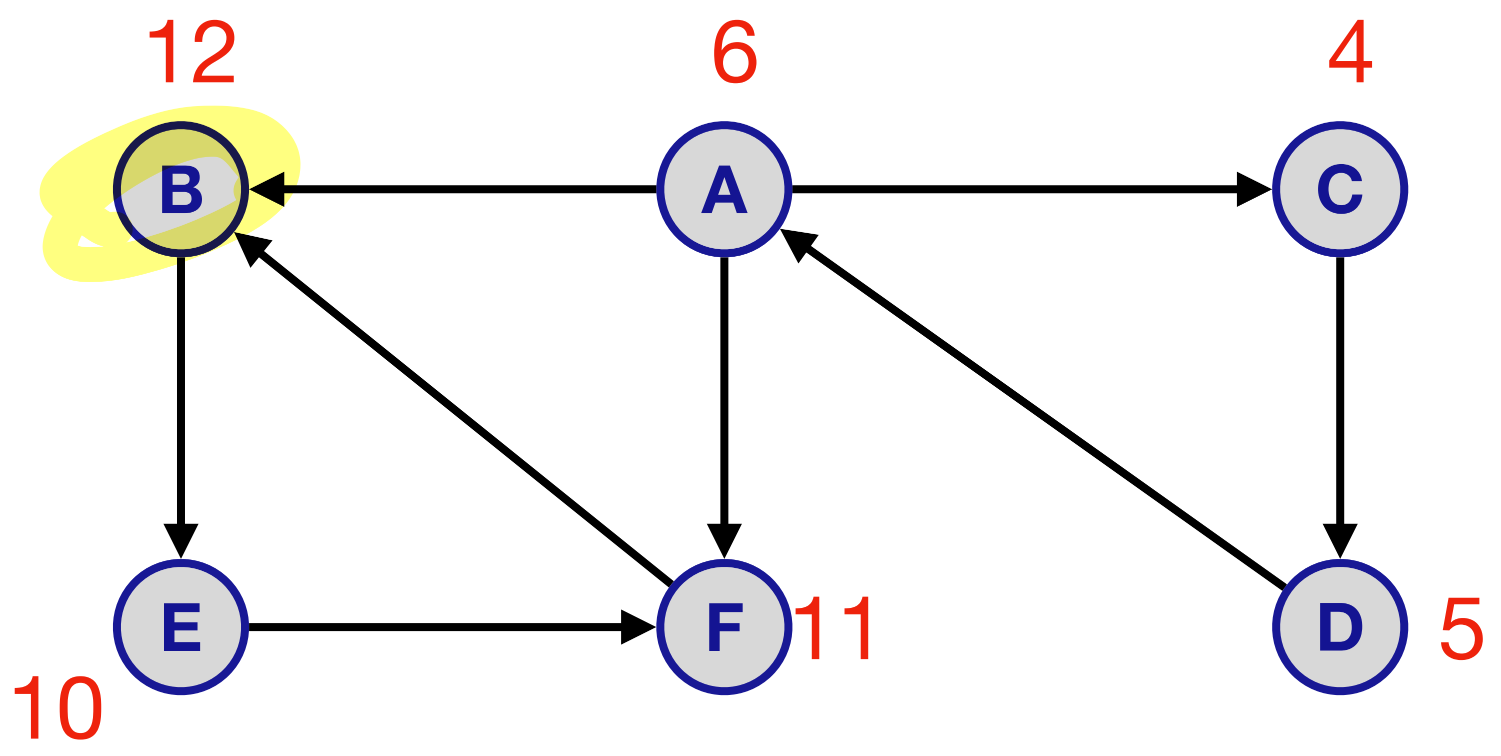


SCC computed:

{G}, {H}

Linear Time Algorithm - An Example

Do **DFS** from vertex B and remove "it"

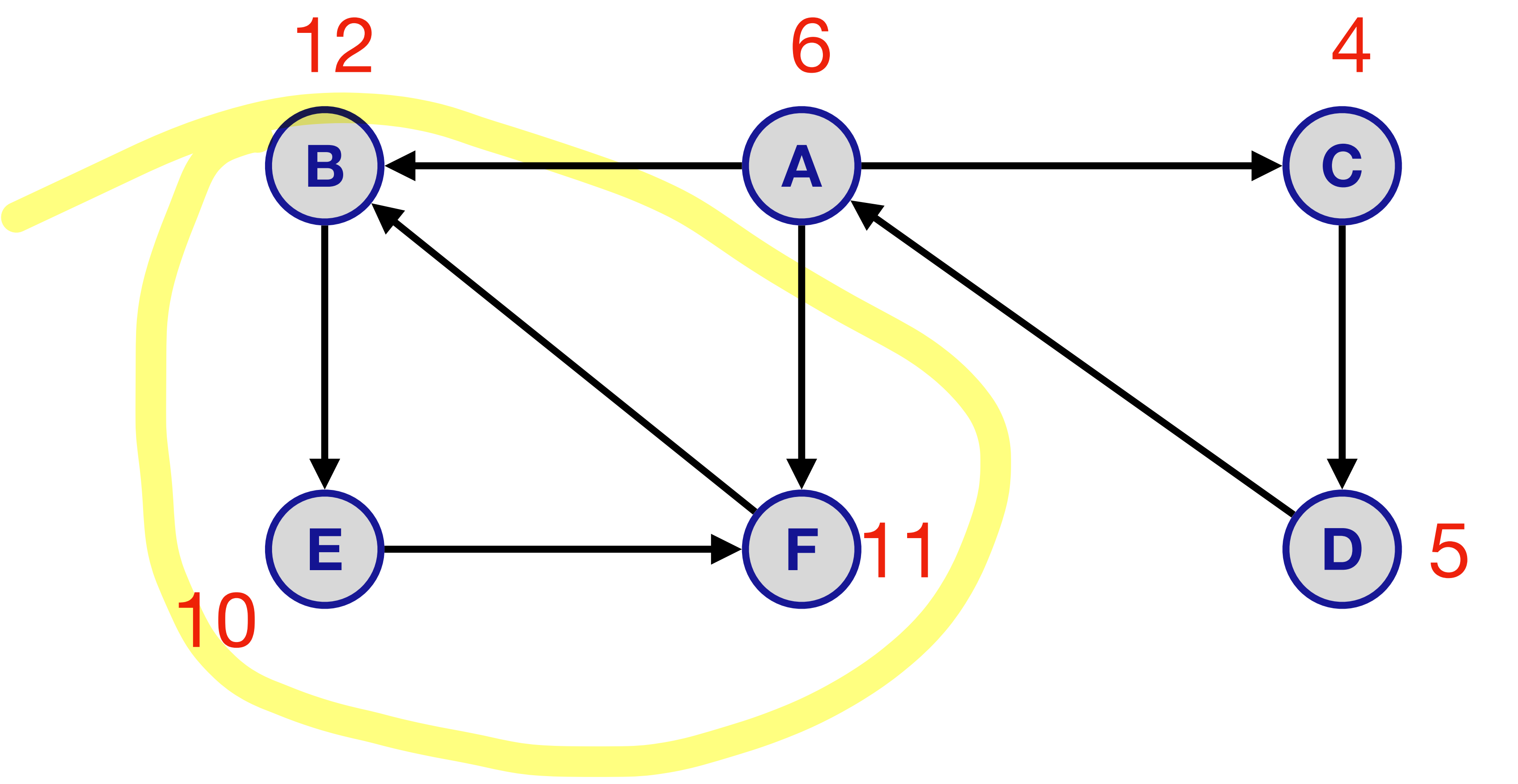


SCC computed:

{G}, {H}

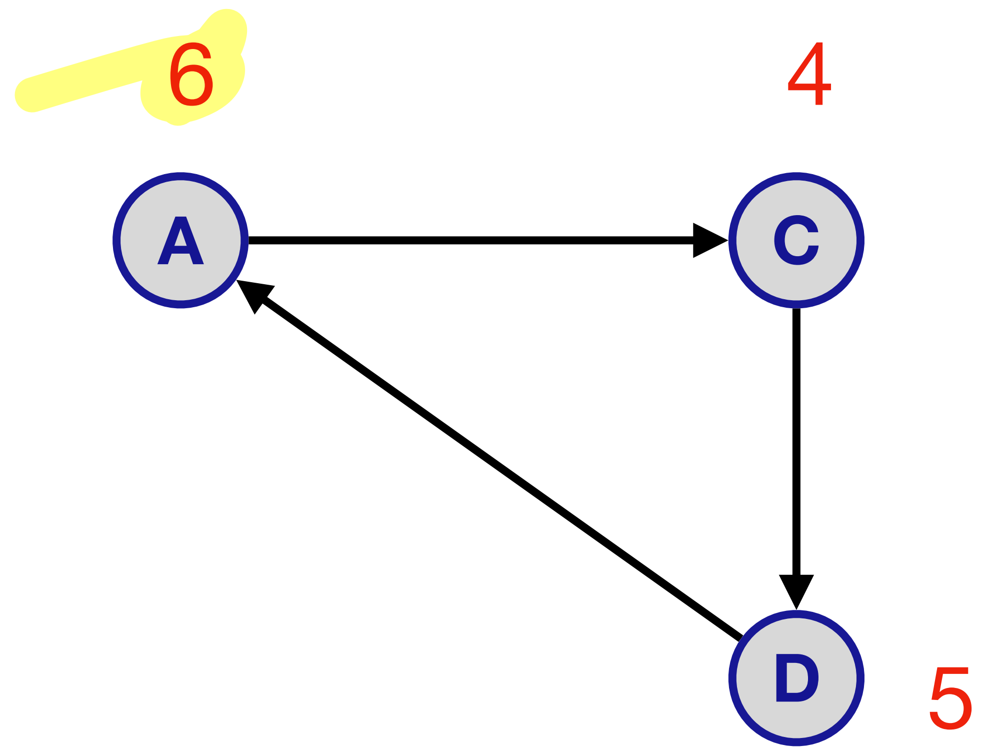
Linear Time Algorithm - An Example

Do **DFS** from vertex B and remove "it"



$\Longrightarrow$

Remove visited vertices: {F, B, E}.



SCC computed:

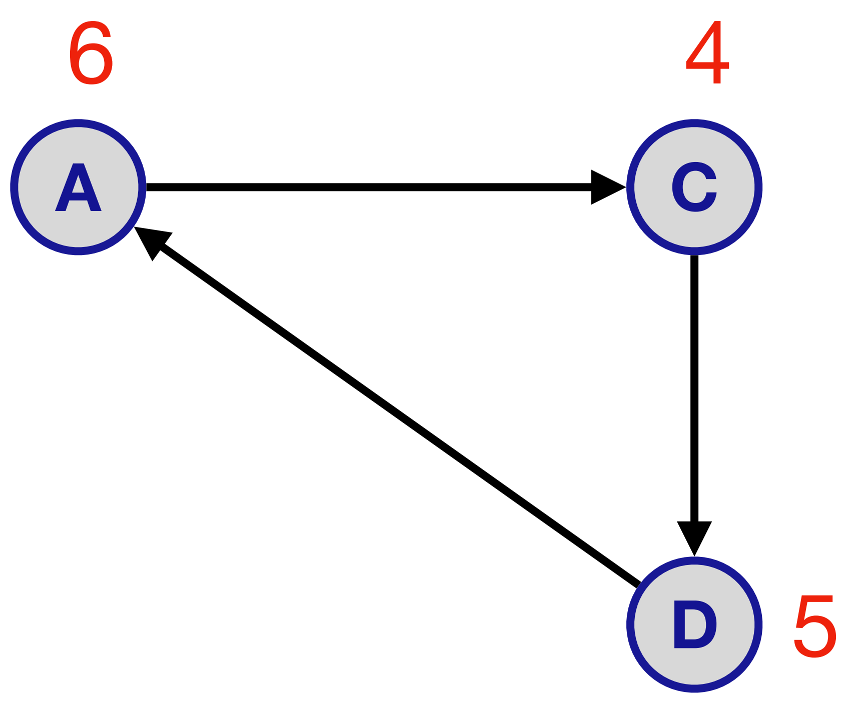
{G}, {H}

SCC computed:

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

Linear Time Algorithm - An Example

Do **DFS** from vertex A and remove "it".

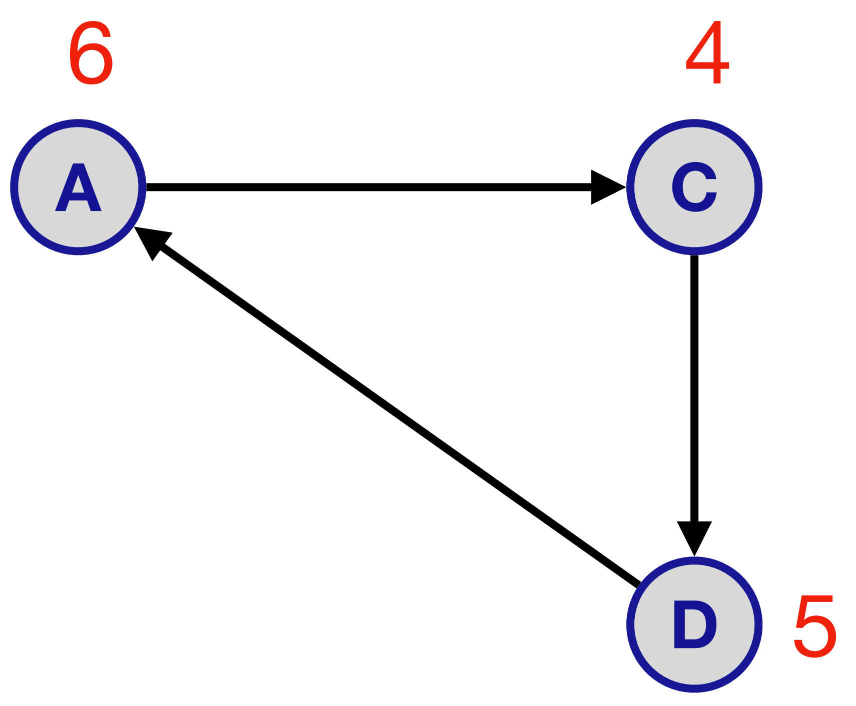


SCC computed:

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

Linear Time Algorithm - An Example

Do **DFS** from vertex A and remove "it".



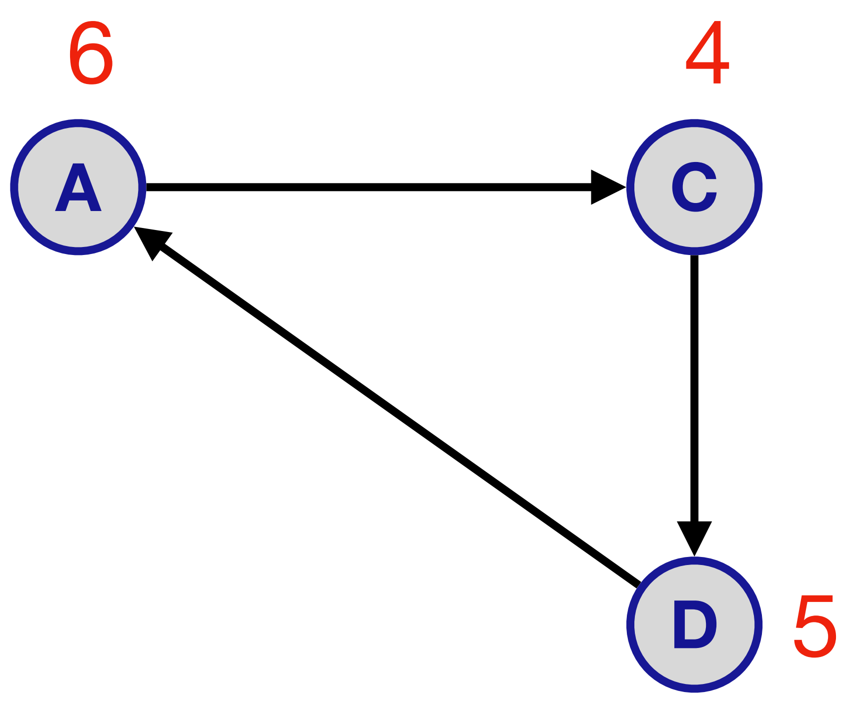
Remove visited vertices: {A, C, D}.

SCC computed:

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

Linear Time Algorithm - An Example

Do **DFS** from vertex A and remove "it".



$\Longrightarrow$

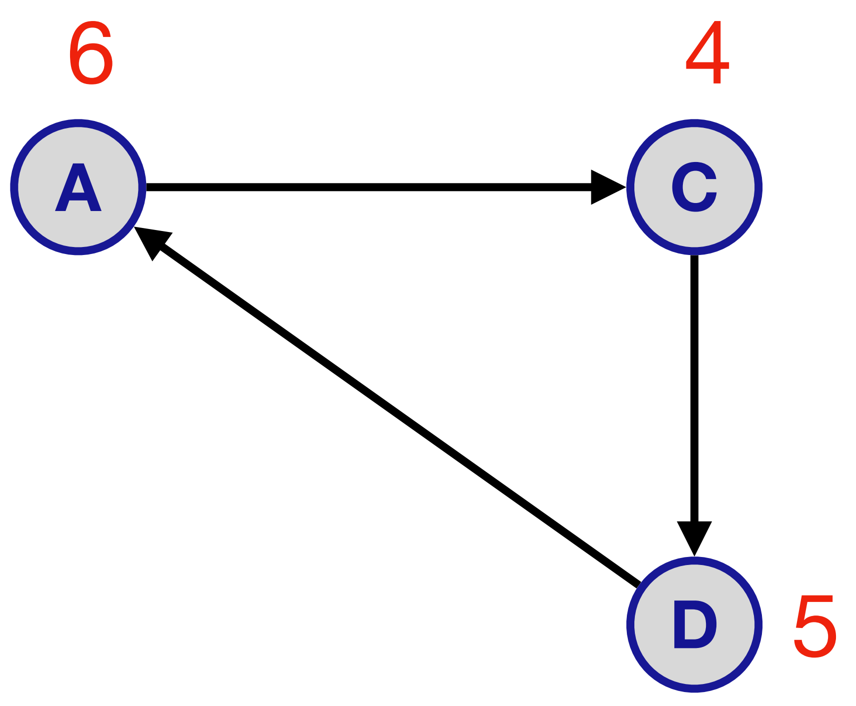
Remove visited vertices: {A, C, D}.

SCC computed:

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

Linear Time Algorithm - An Example

Do **DFS** from vertex A and remove "it".



$\Longrightarrow$

Remove visited vertices: {A, C, D}.

Done

SCC computed:

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

SCC computed:

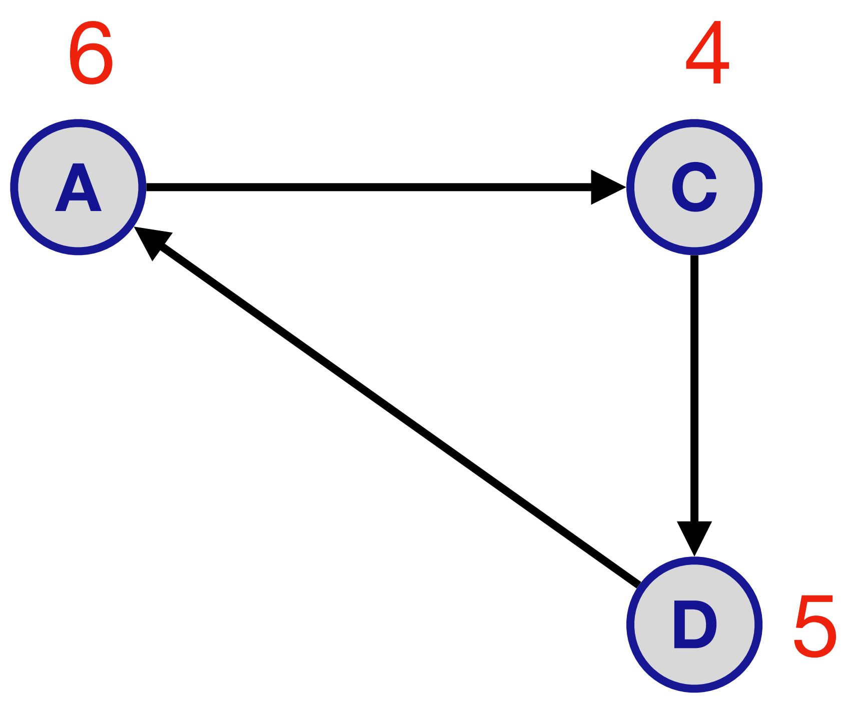
{G}, {H}, {F, B, E}, {A,C,D}

Robert Tarjan's (see Jeff's book)

Kosaraju. presented today (essentially)

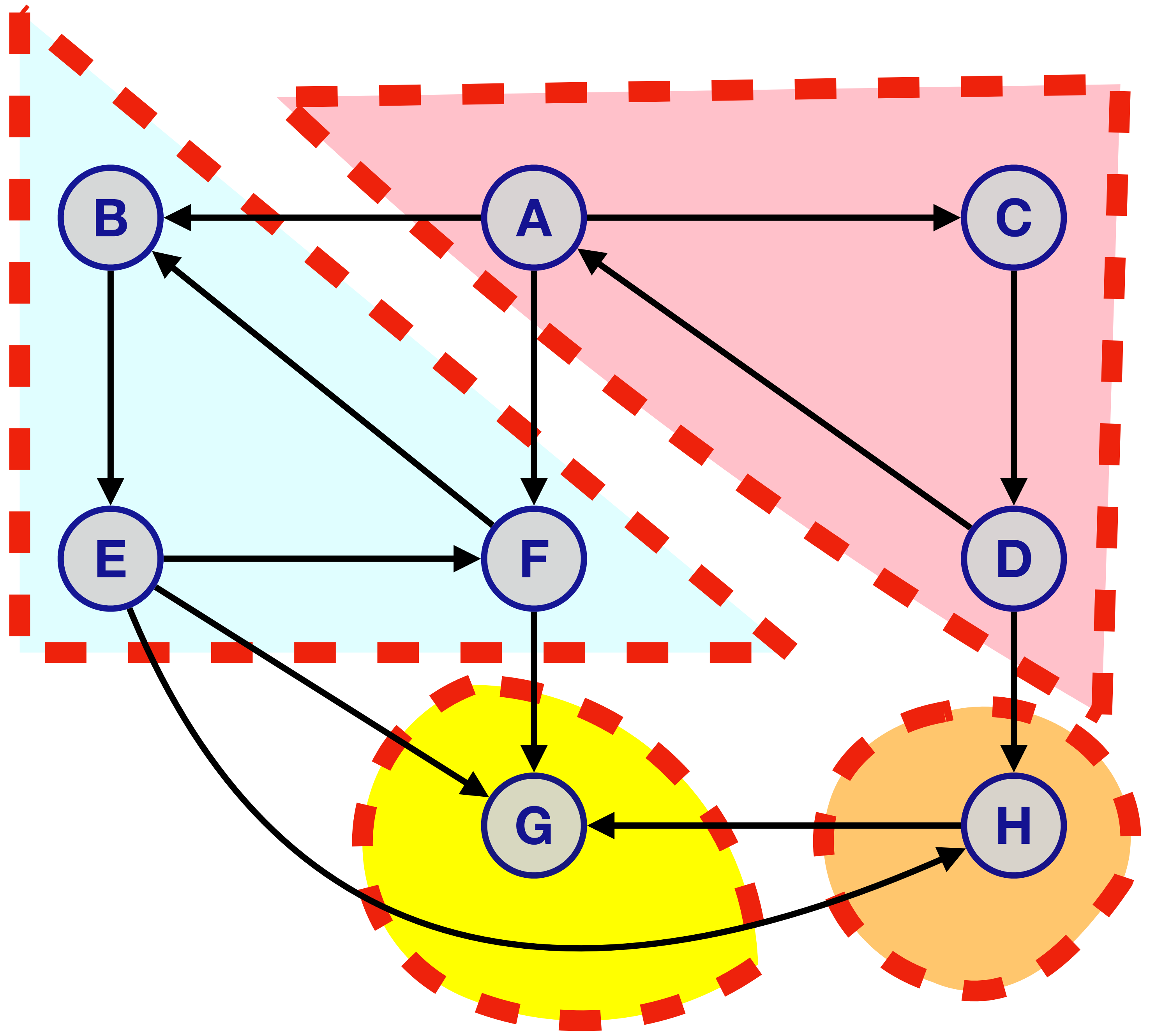
Linear Time Algorithm - An Example

Do **DFS** from vertex A and remove "it".



$\Longrightarrow$

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.

Summary

Take away points

- DAGs and topological orderings.
- **DFS** with pre/post numbering.

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.

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.

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!