Week 11 — Sheet 10

Algorithms and Data Structures

04.12.2023🎉 — Georg Hasebe

Debriefing of Submissions

On External Resources

- ChatGPT
- In the case of the use of external resources (i.e. resources not from lecture notes, script, exercise sheets) we expect the solution to be complete and explained well! (especially the parts that were not covered!)

Other Remarks

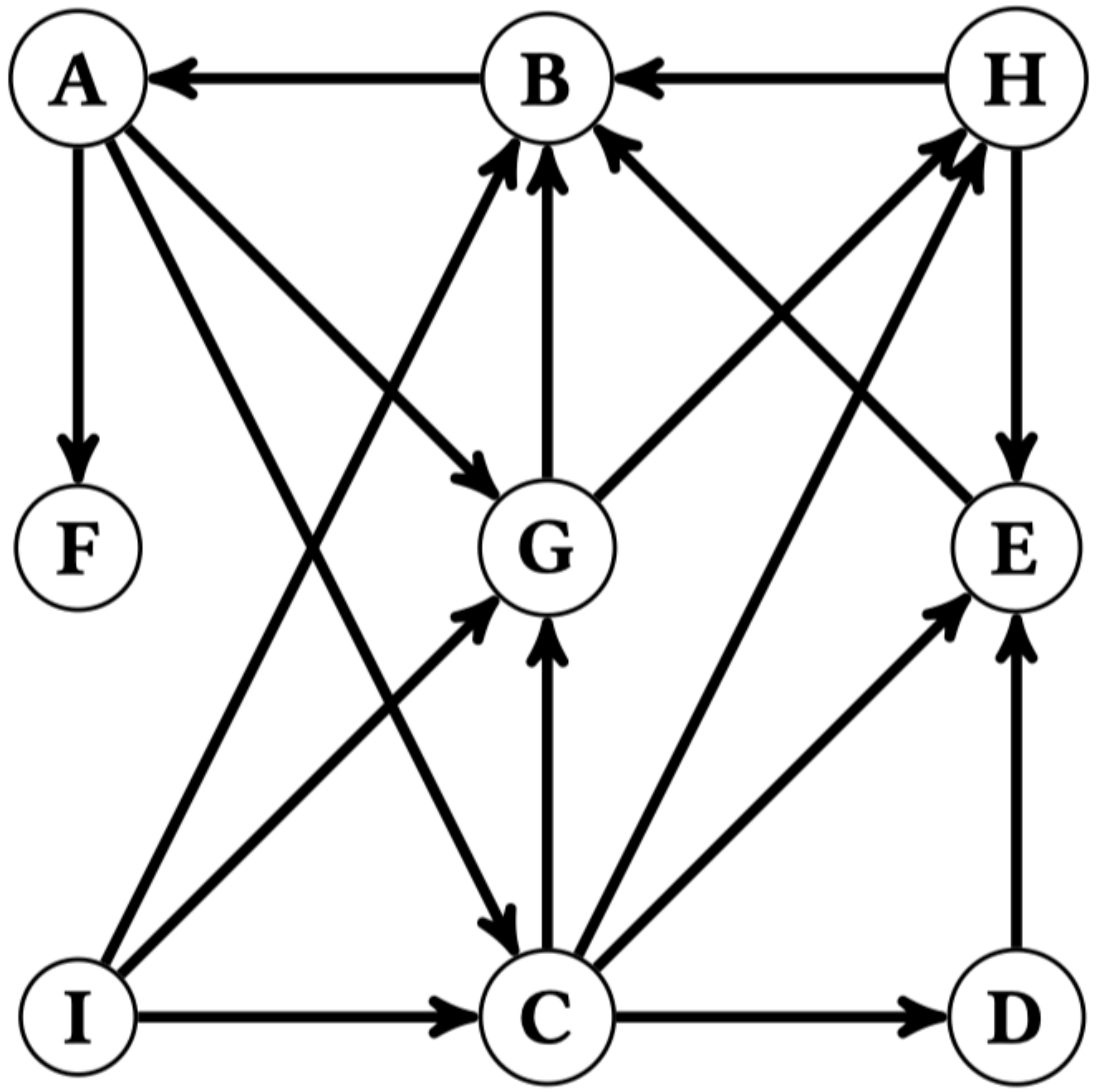
- Calling DFS on all vertices
- Justification for DP recursion

Exercise Sheet 10

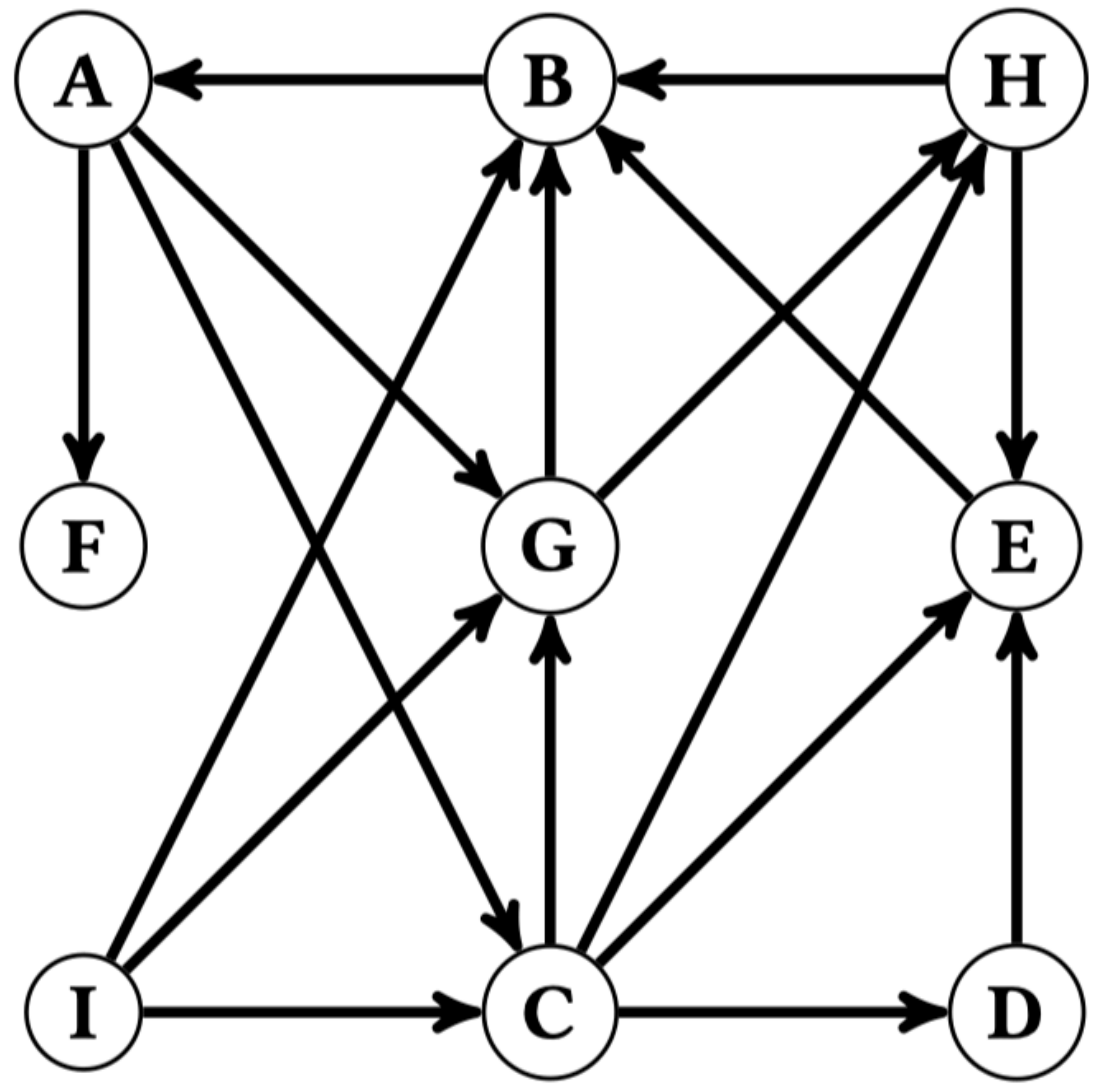
Exercise 10.2

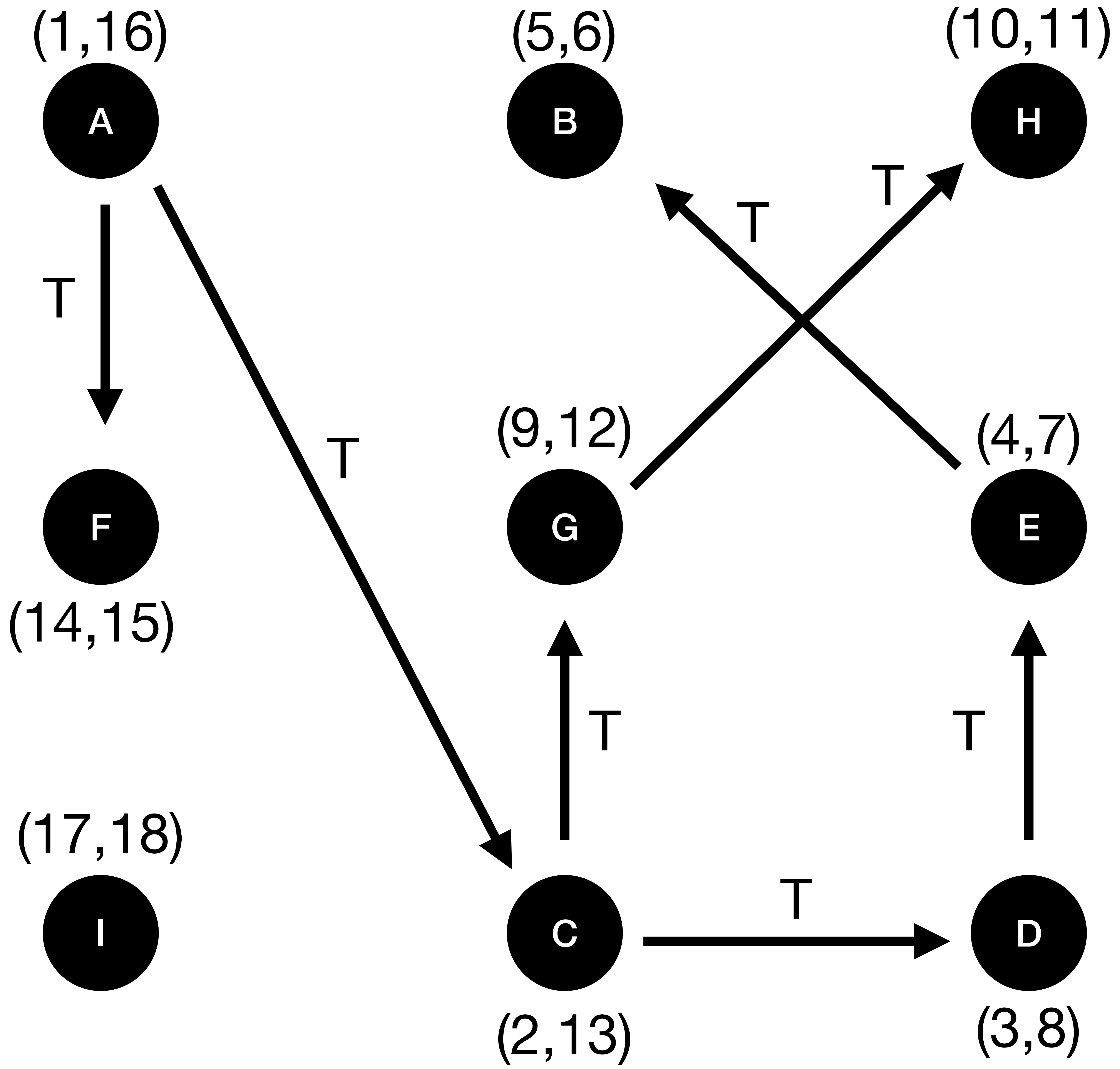
Exercise 10.2 *Depth-first search* **(1 point)**.

Execute a depth-first search (*Tiefensuche*) on the following graph. Use the algorithm presented in the lecture. Always do the calls to the function "visit" in alphabetical order, i.e. start the depth-first search from A and once "visit(A)" is finished, process the next unmarked vertex in alphabetical order. When processing the neighbors of a vertex, also process them in alphabetical order.

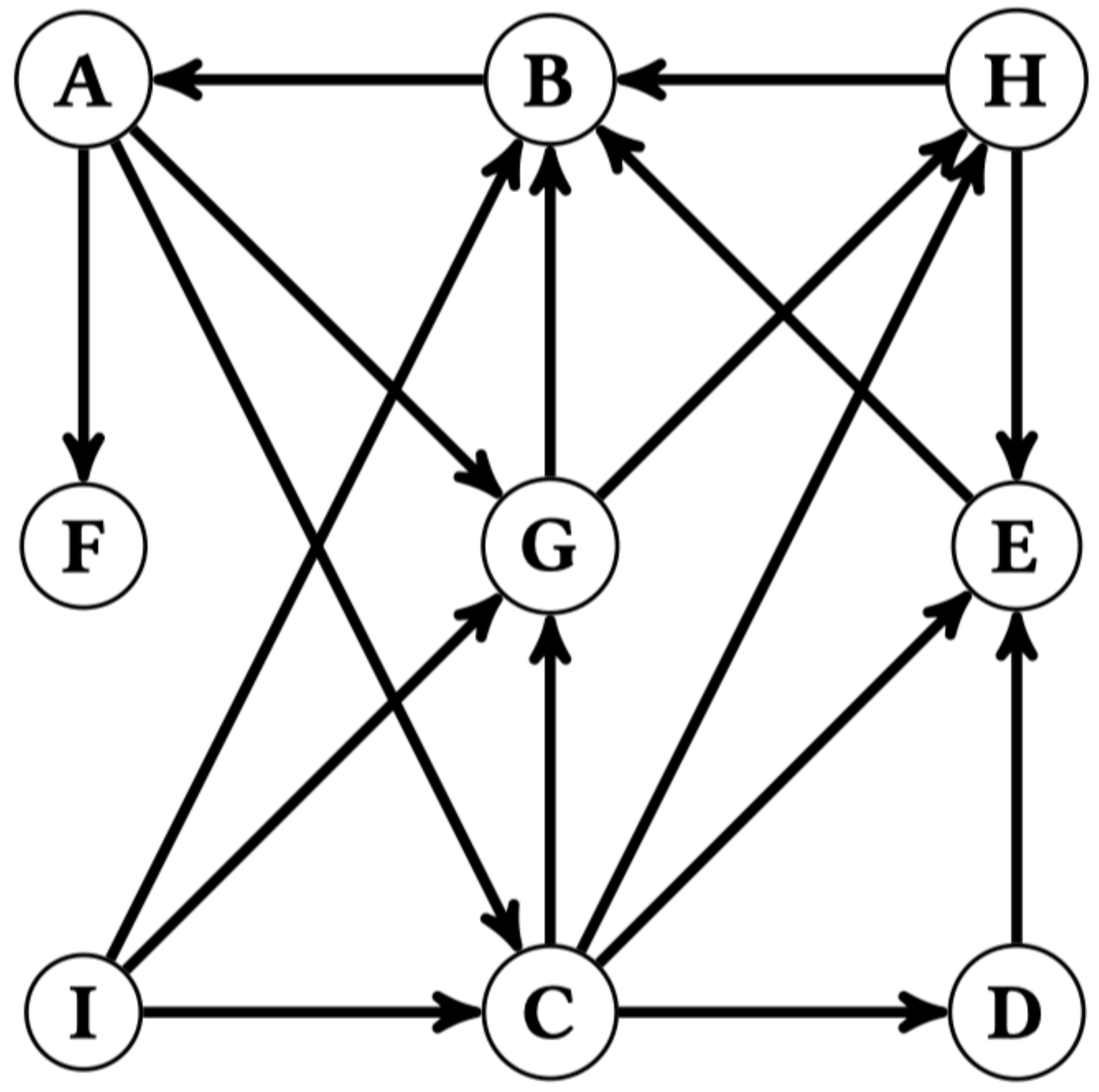


(a) Mark the edges that belong to the depth-first forest (*Tiefensuchwald*) with a “T” (for tree edge).

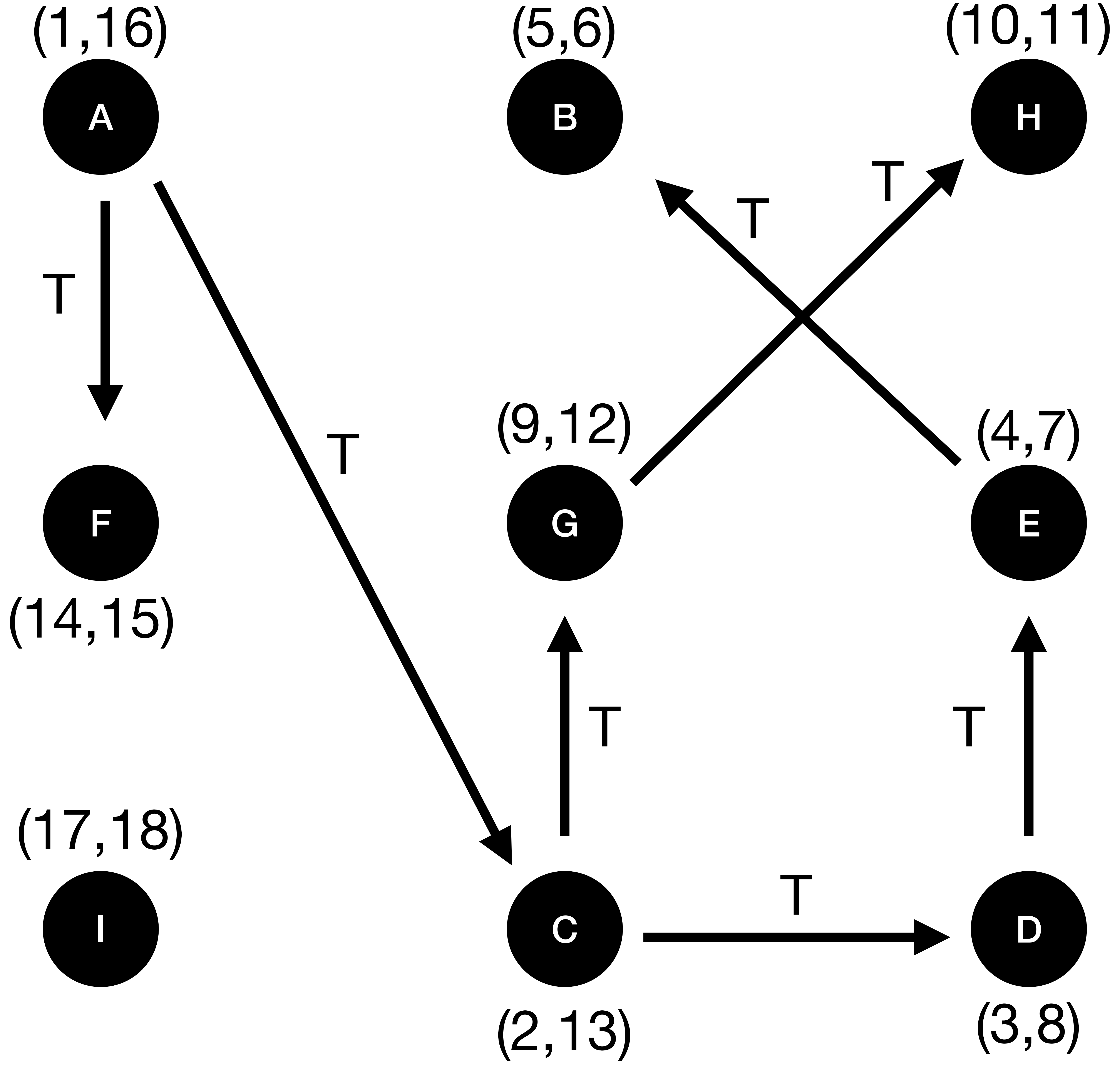




(b) For each vertex in the depth-first forest, give its *pre*- and *post*-number.



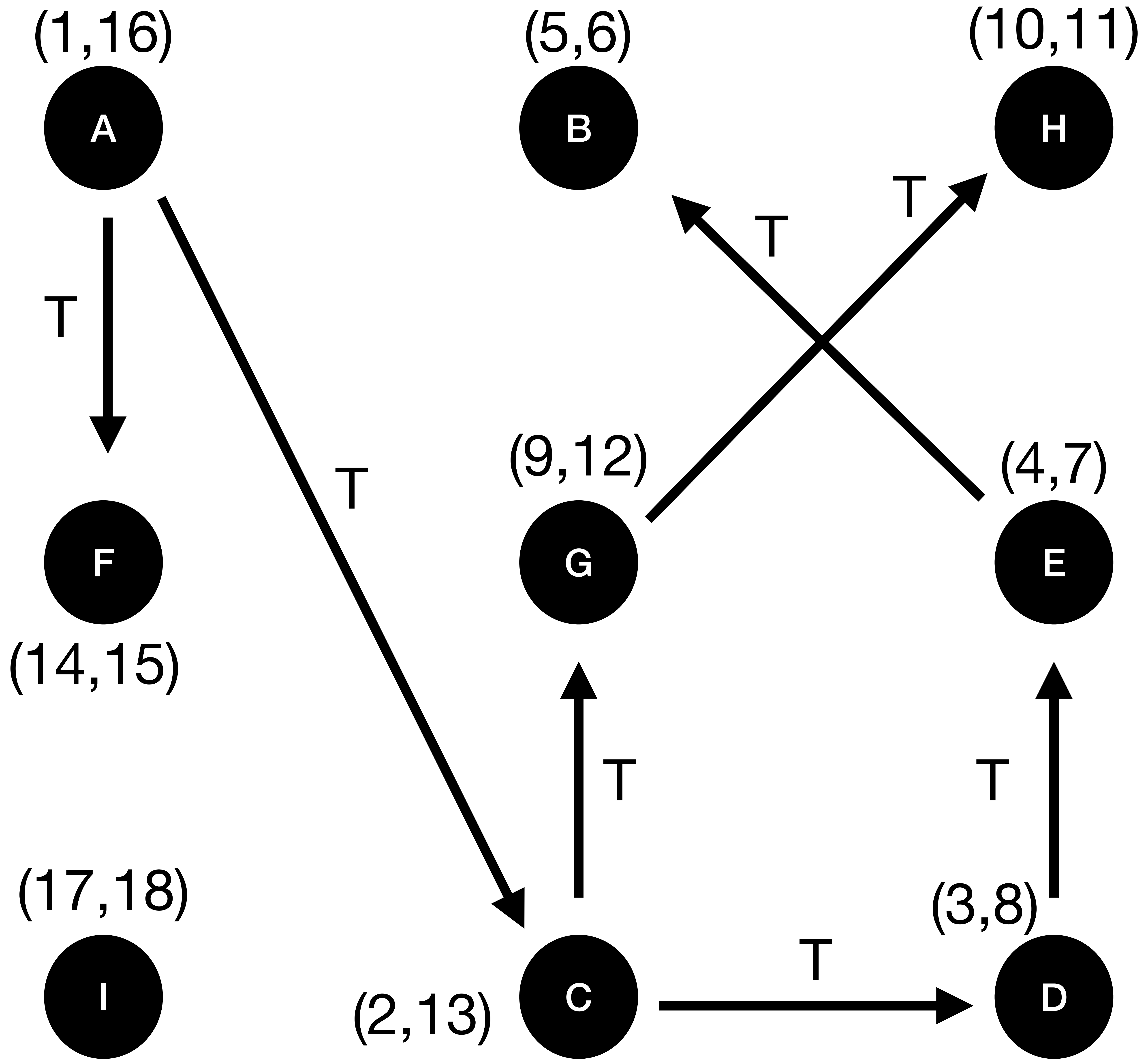
A(1,16) B(5,6) C(2,13) D(3,8) E(4,7) F(14,15) G(9,12) H(10,11) I(17,18).



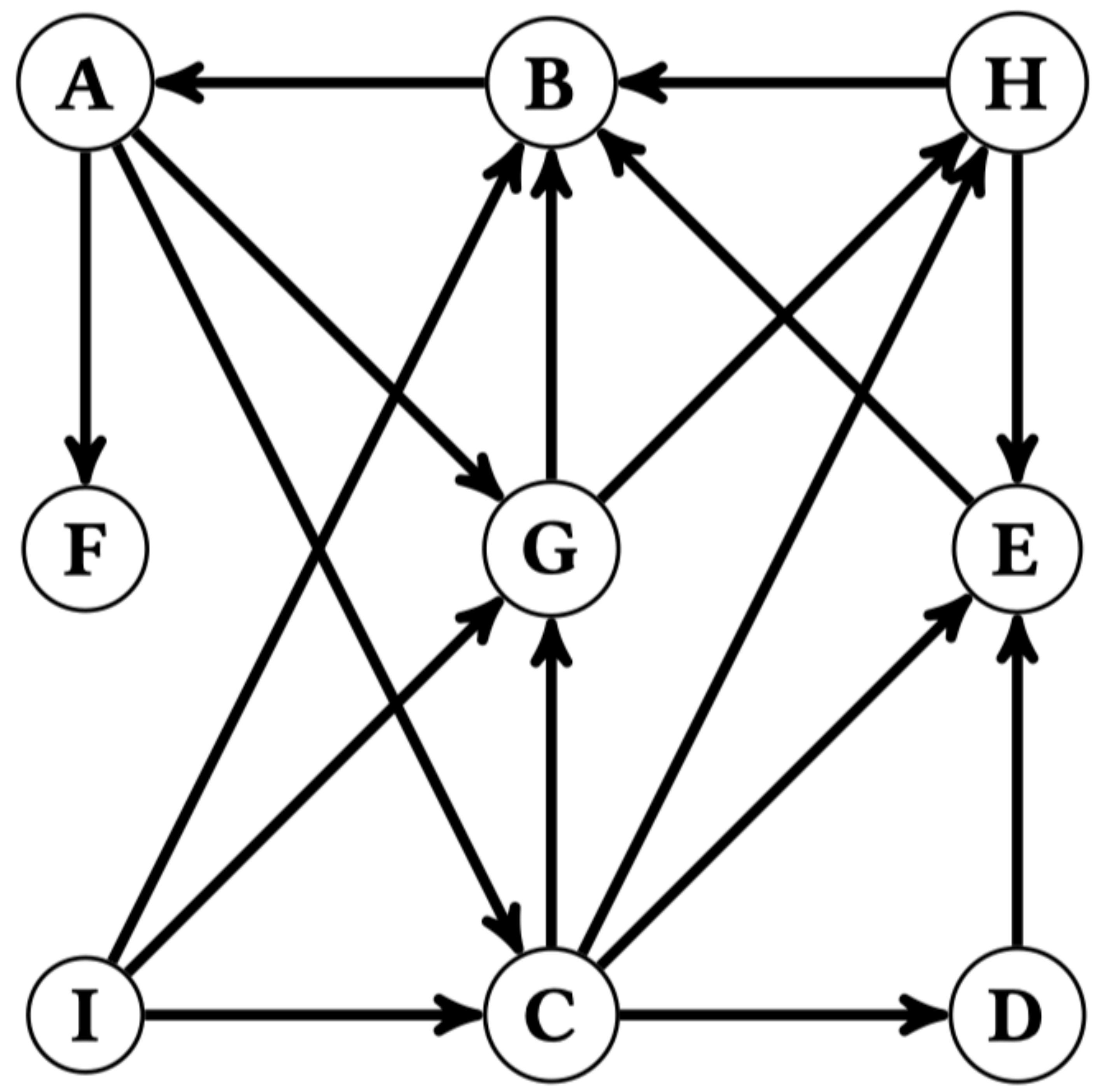
(c) Give the vertex ordering that results from sorting the vertices by pre-number. Give the vertex ordering that results from sorting the vertices by post-number.

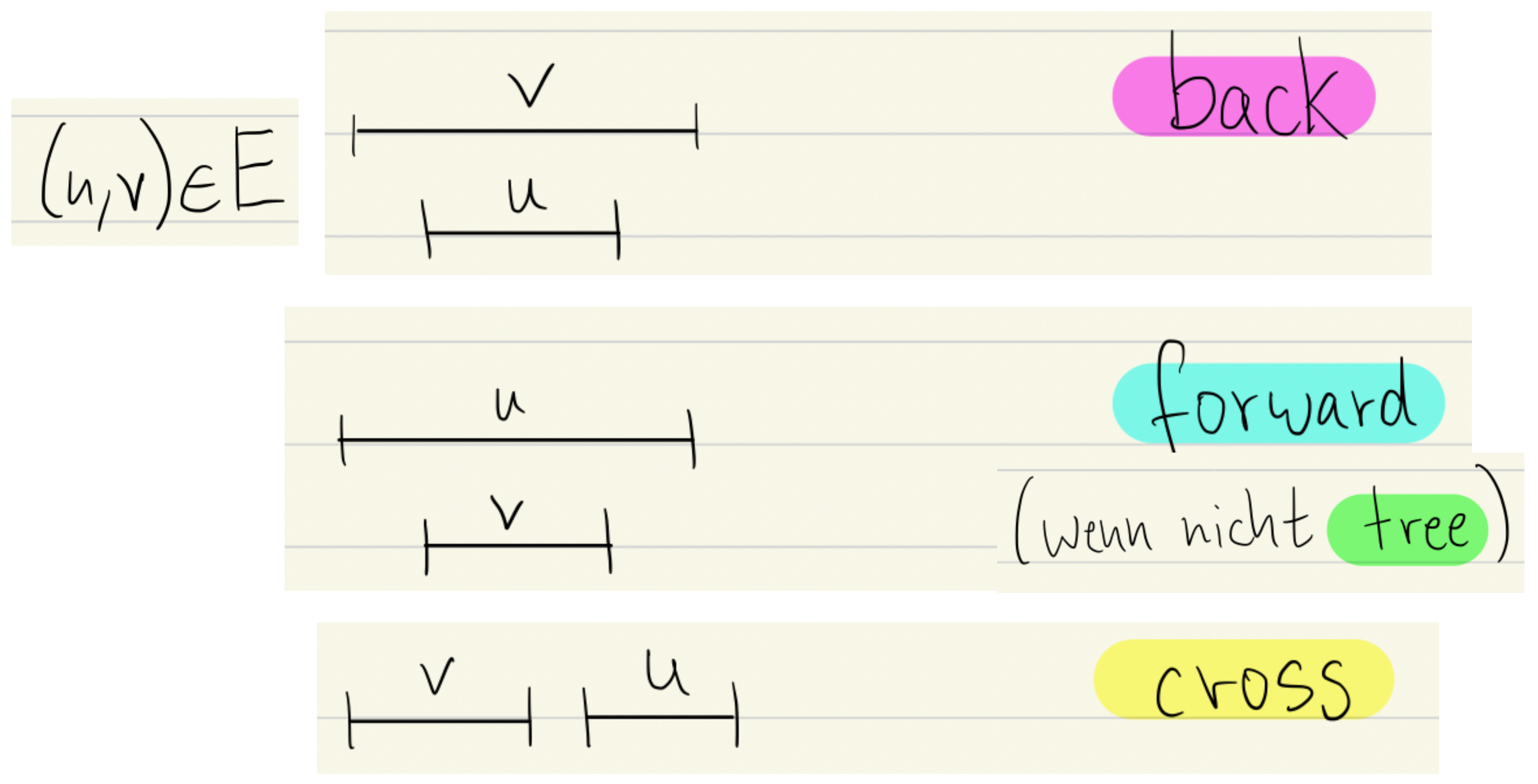
Pre-ordering: A, C, D, E, B, G, H, F, I.

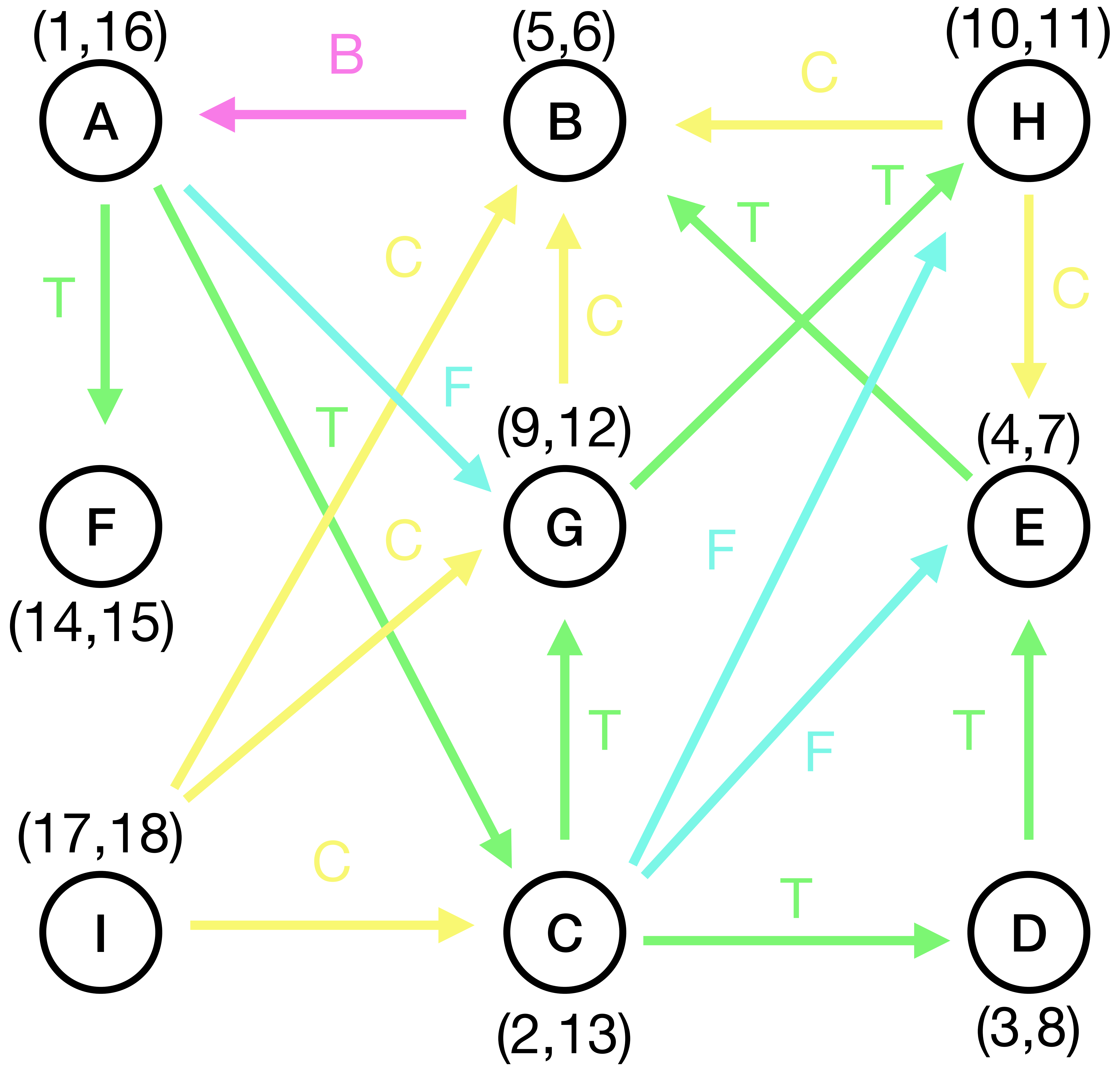
Post-ordering: B, E, D, H, G, C, F, A, I.



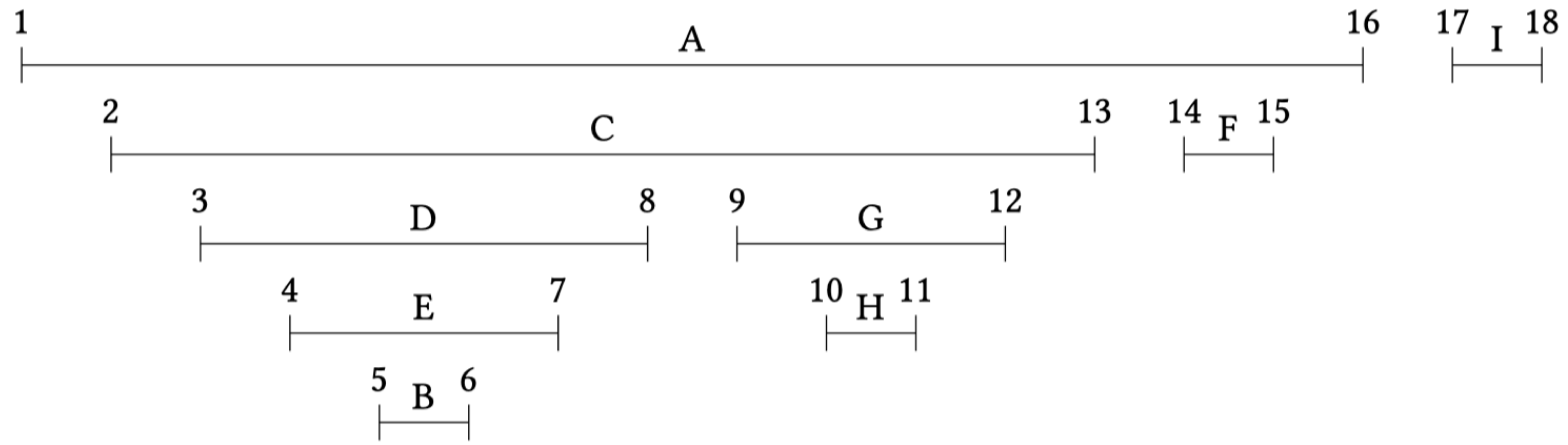
(d) Mark every forward edge (*Vorwärtskante*) with an “F”, every backward edge (*Rückwärtskante*) with a “B”, and every cross edge (*Querkante*) with a “C”.







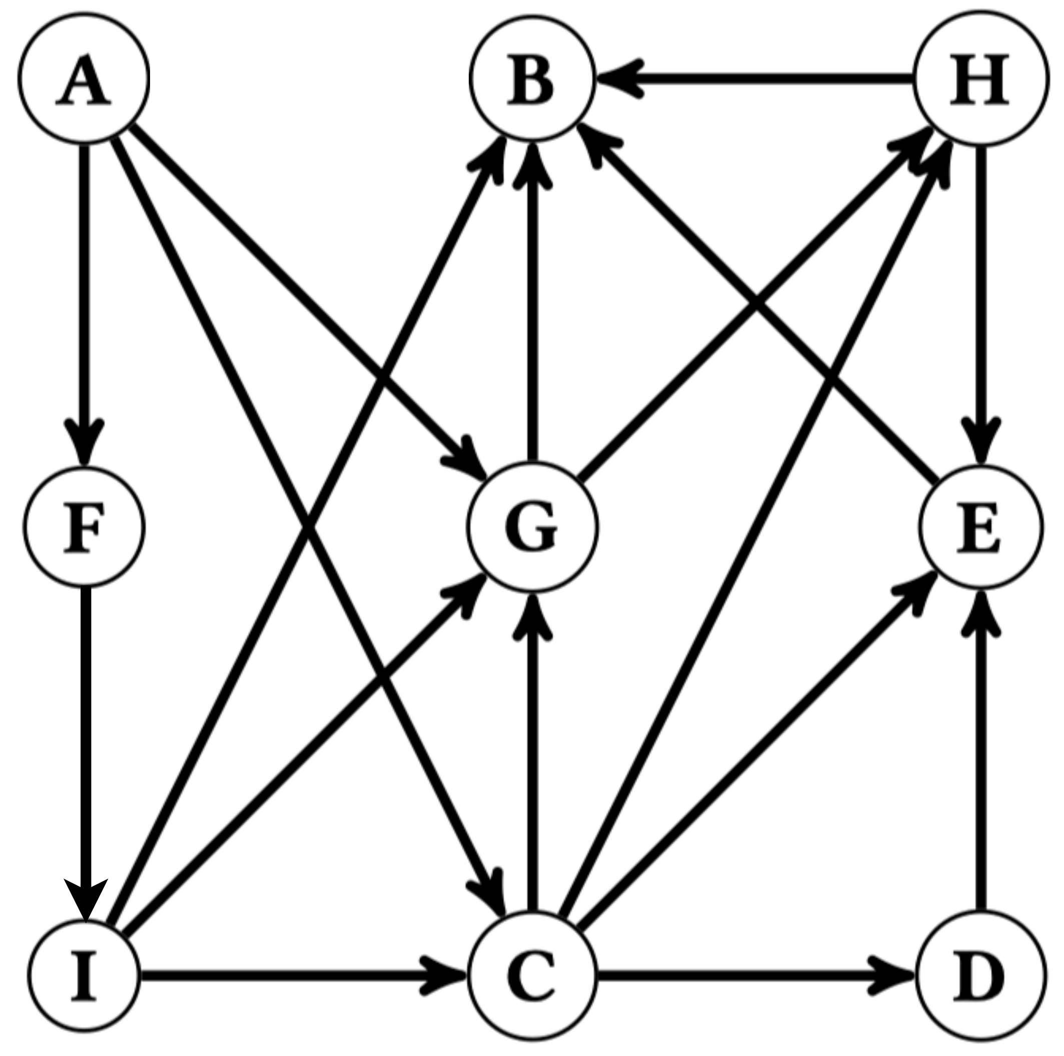
(f) Draw a scale from 1 to 18, and mark for every vertex v the interval I_v from pre-number to post-number of v. What does it mean if $I_u \subset I_v$ for two different vertices u and v?



If $I_u \subset I_v$ for two different vertices u and v, then u is visited during the call of visit(v).

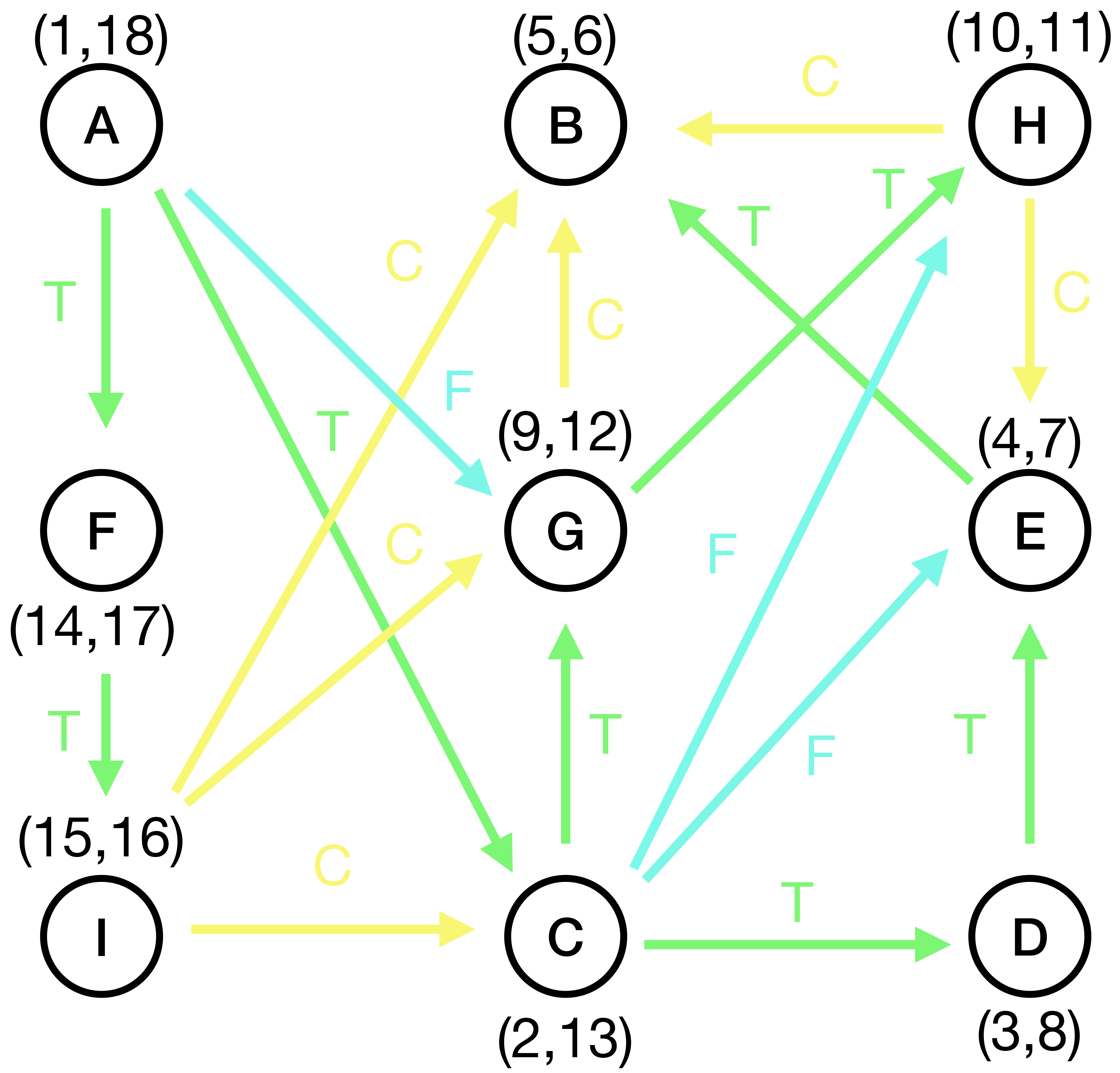
(g) Consider the graph above where the edge from B to A is removed and an edge from F to I is added. How does the execution of depth-first search change? Does the graph have a topological ordering? If yes, write down the topological ordering we get from the execution of depth-first search; if no, argue how we can use the execution of depth-first search to find a directed cycle. If you sort the vertices by *pre-number*, does this give a topological sorting?

Remove B to A, add F to I



Now the graph is a DAG, thus reverse post-ordering gives a topological ordering:

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



Exercise 10.4.a.

Exercise 10.4.b.-d.

```
KOSARAJU($G$, $n$)
    mark all $v \in V$ as unvisited
    $L \leftarrow \emptyset$    // empty list
    for unvisited $v \in V$
        DFS_1($v$, $L$)

    mark all $v \in V$ as unvisited
    reverse $L$
    $C \leftarrow \emptyset$    // list of components
    create $G^{-1} = (V, E^{-1})$ from $G$
    for unvisited $v \in L$   // in order of $L$
        $T \leftarrow \emptyset$    // single component
        DFS_2($v$, $T$)
        add $T$ to $C$

    return $C$   // contains all components
```

```
DFS_1($v$, $L$)
    mark $v$ as visited
    for unvisited $u \in N_G(v)$ // neighbors of $v$ in $G$
        DFS_1($u$)

    add $v$ to back of $L$
```

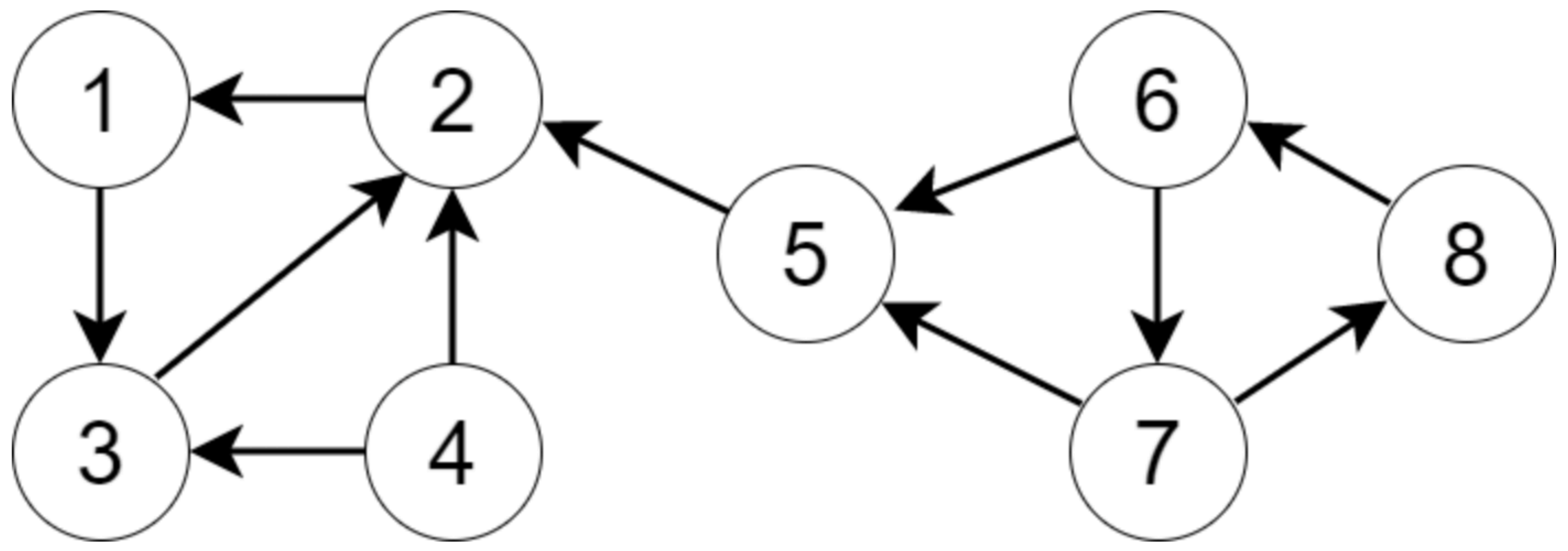
```
DFS_2($v$, $T$)
    mark $v$ as visited
    add $v$ to component $T$
    for unvisited $u \in N_{G^{-1}}(v)$ // neighbors of $v$ in $G^{-1}$
        DFS_2($u$)
```

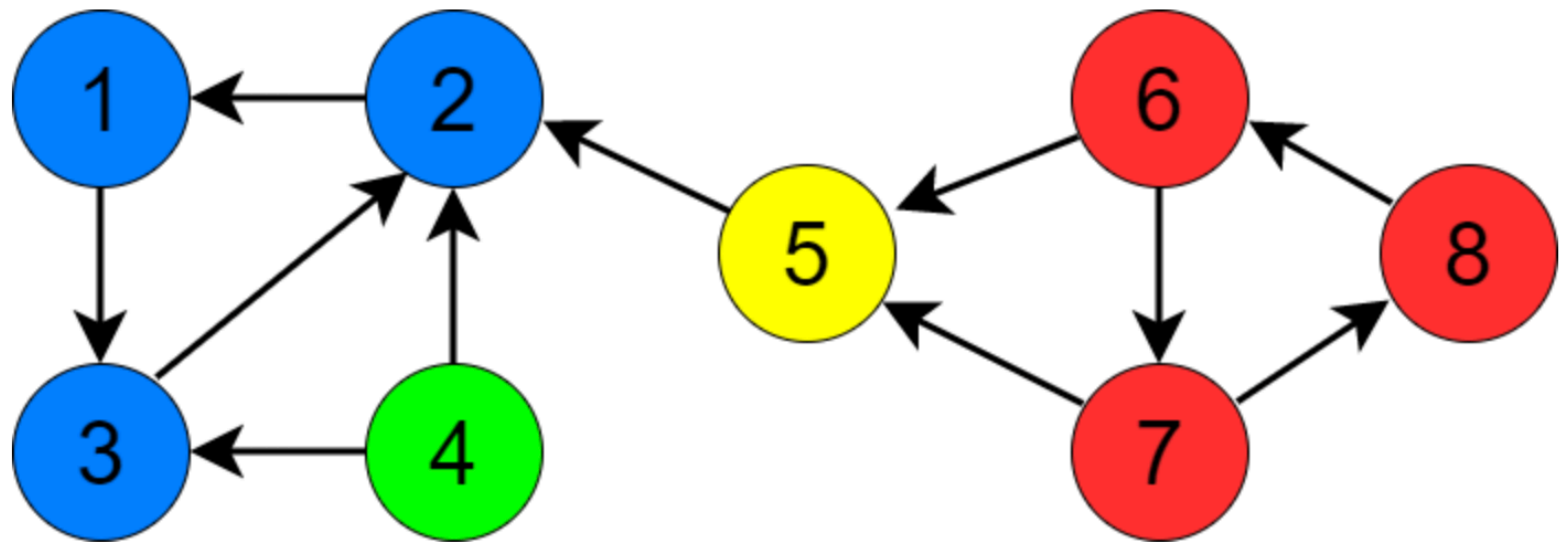
Runtime:

- calls to DFS_1, DFS_2 are in $O(n+m)$
- reversing L is in $O(n)$
- inverting G (G^{-1}) is in $O(n+m)$

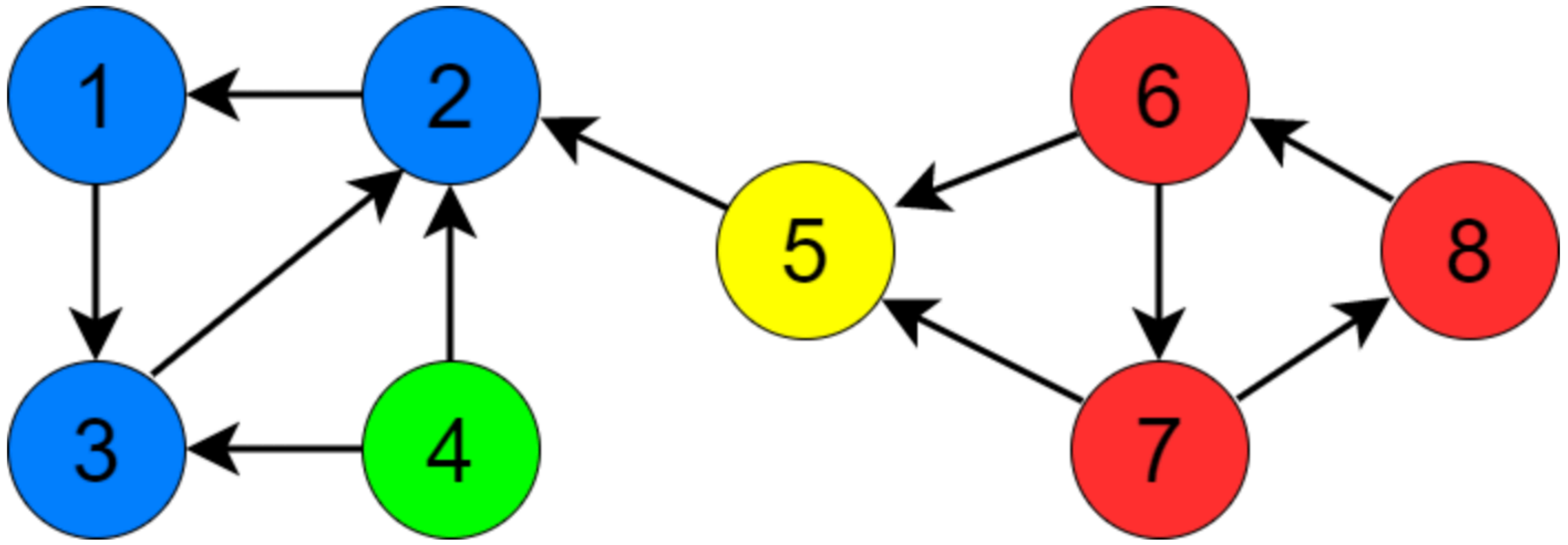
$\Rightarrow O(n+m)$

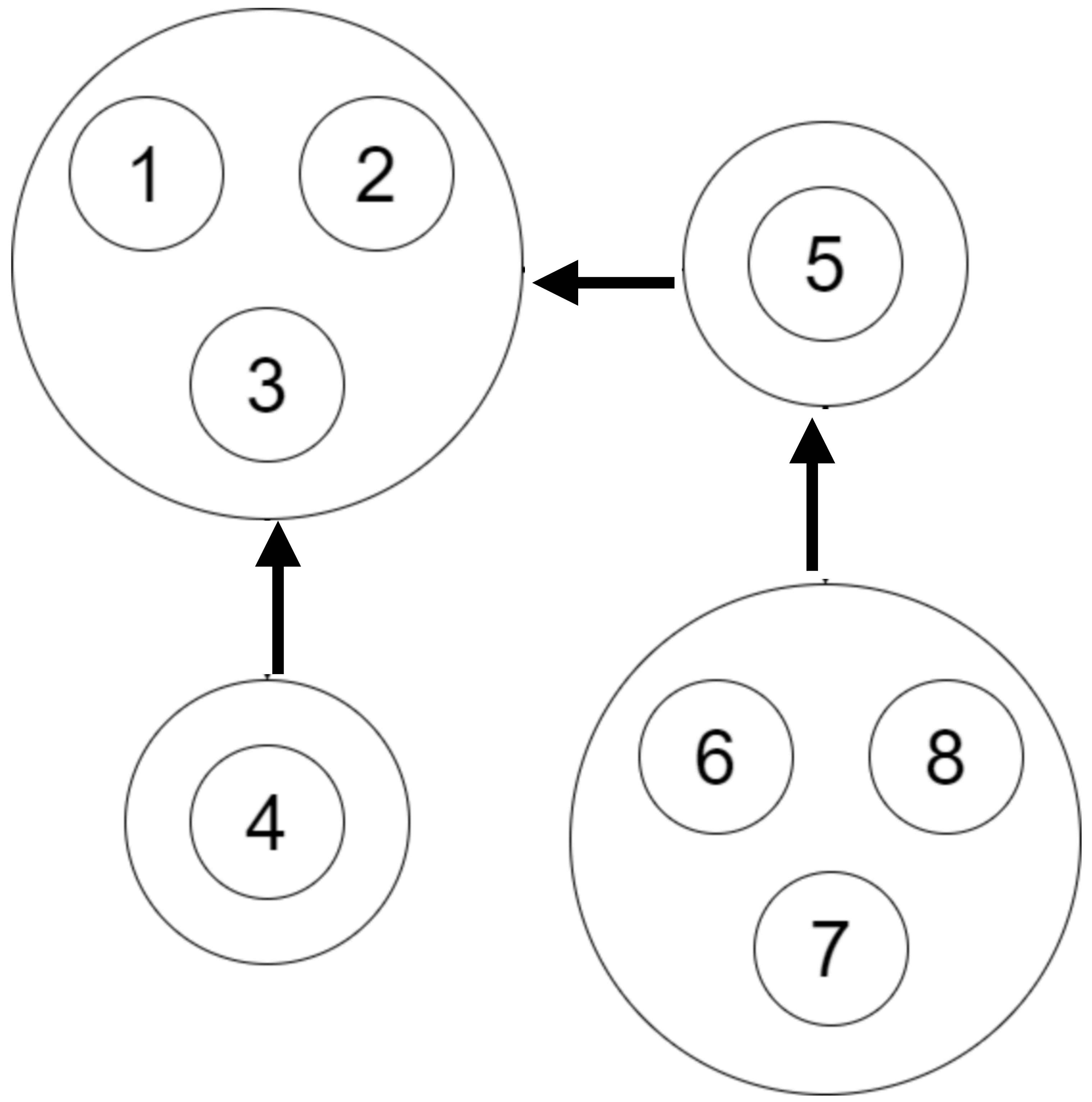
Example





Condensation Graph





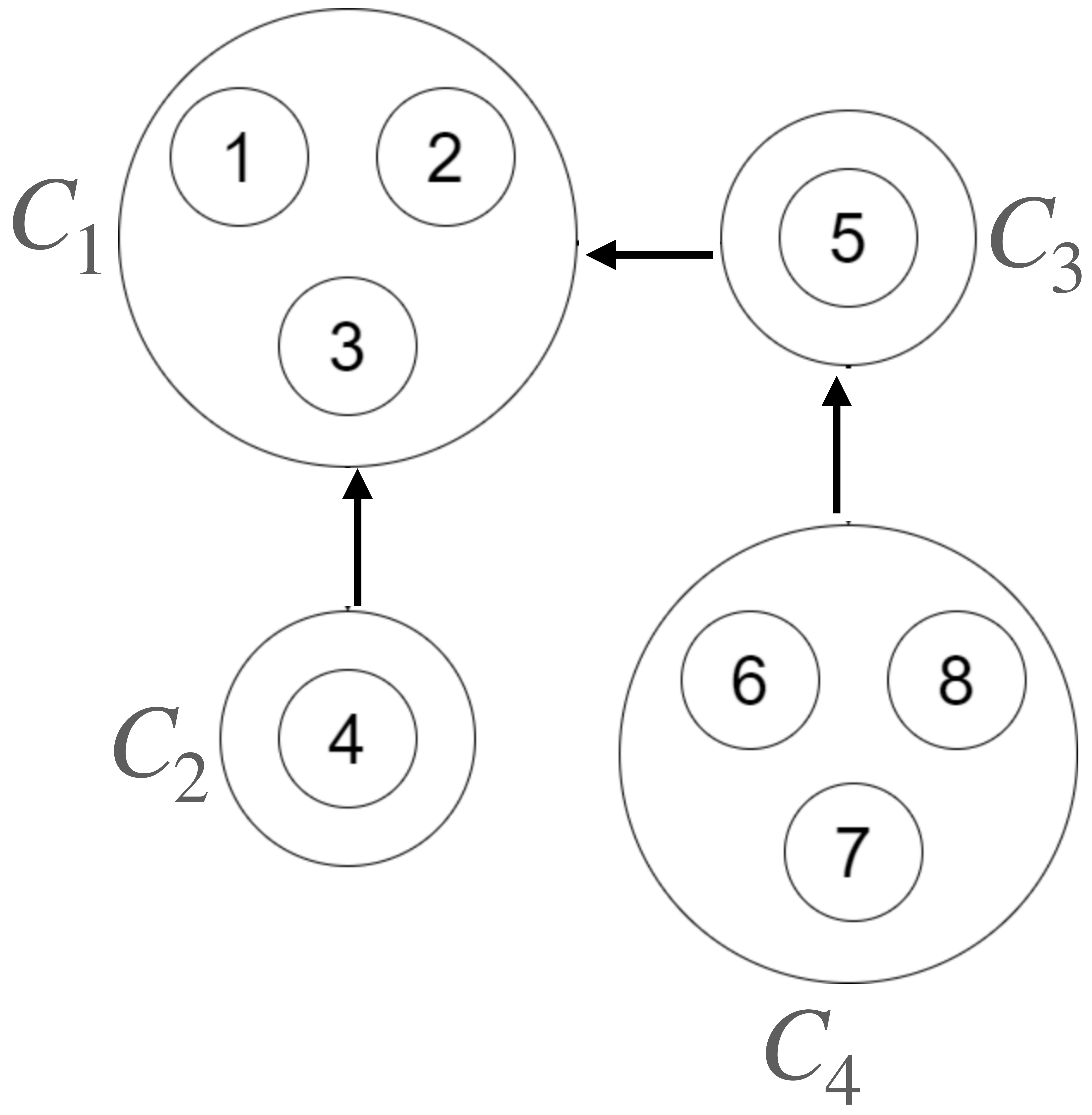
Condensation Graph

Let post[C_i] be the maximum of the post number of all vertices in C_i.

If there is an edge from C_i to C_j we must have post[C_i] > post[C_j]

Proper Proof:
https://cp-algorithms.com/graph/strongly-connected-components.html

Going through the vertices in reverse post order means going through the components in reverse order of post[C_i]

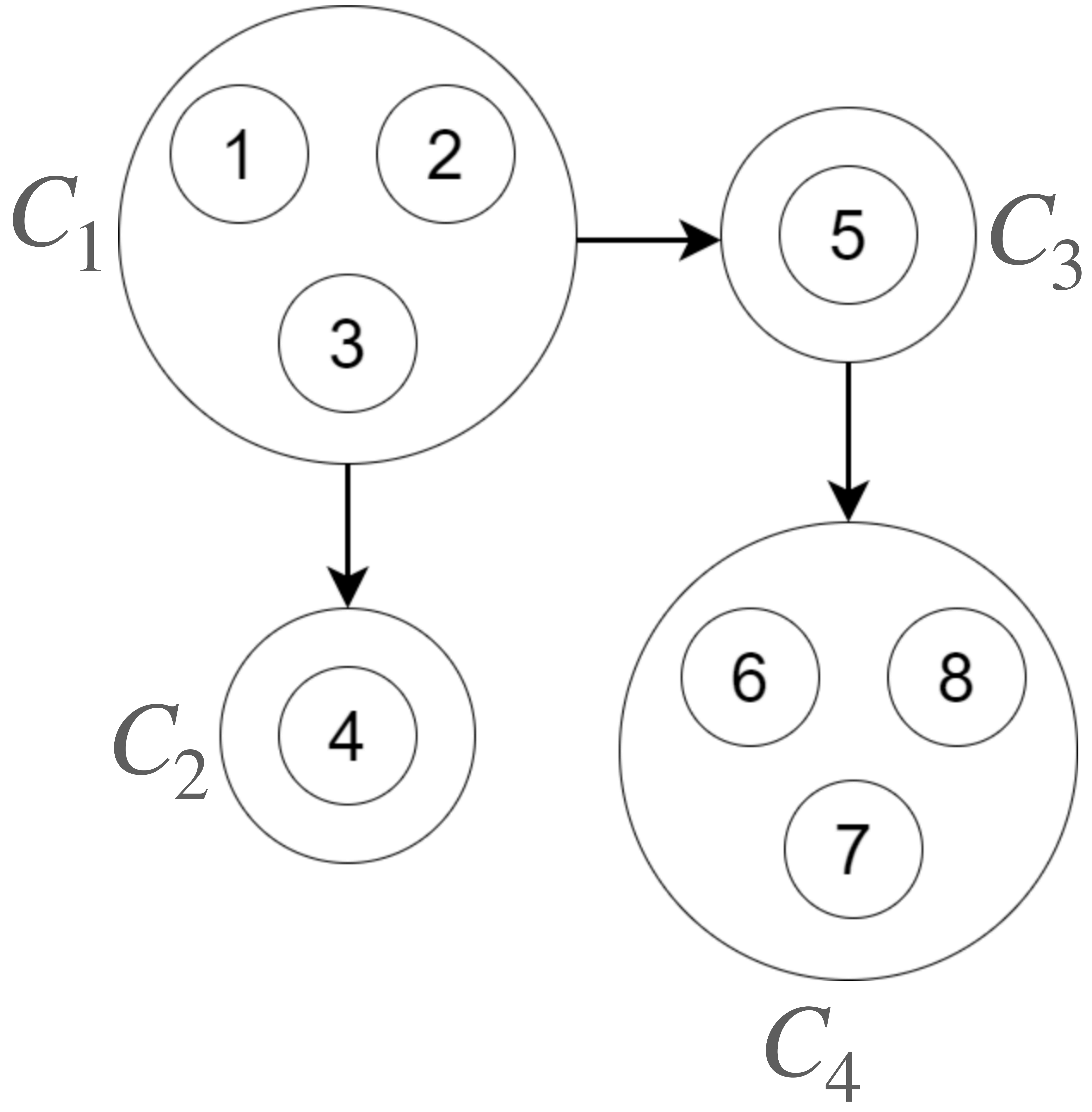


Condensation Graph

Going through the vertices in reverse post order means going through the components in reverse order of post[C_i]

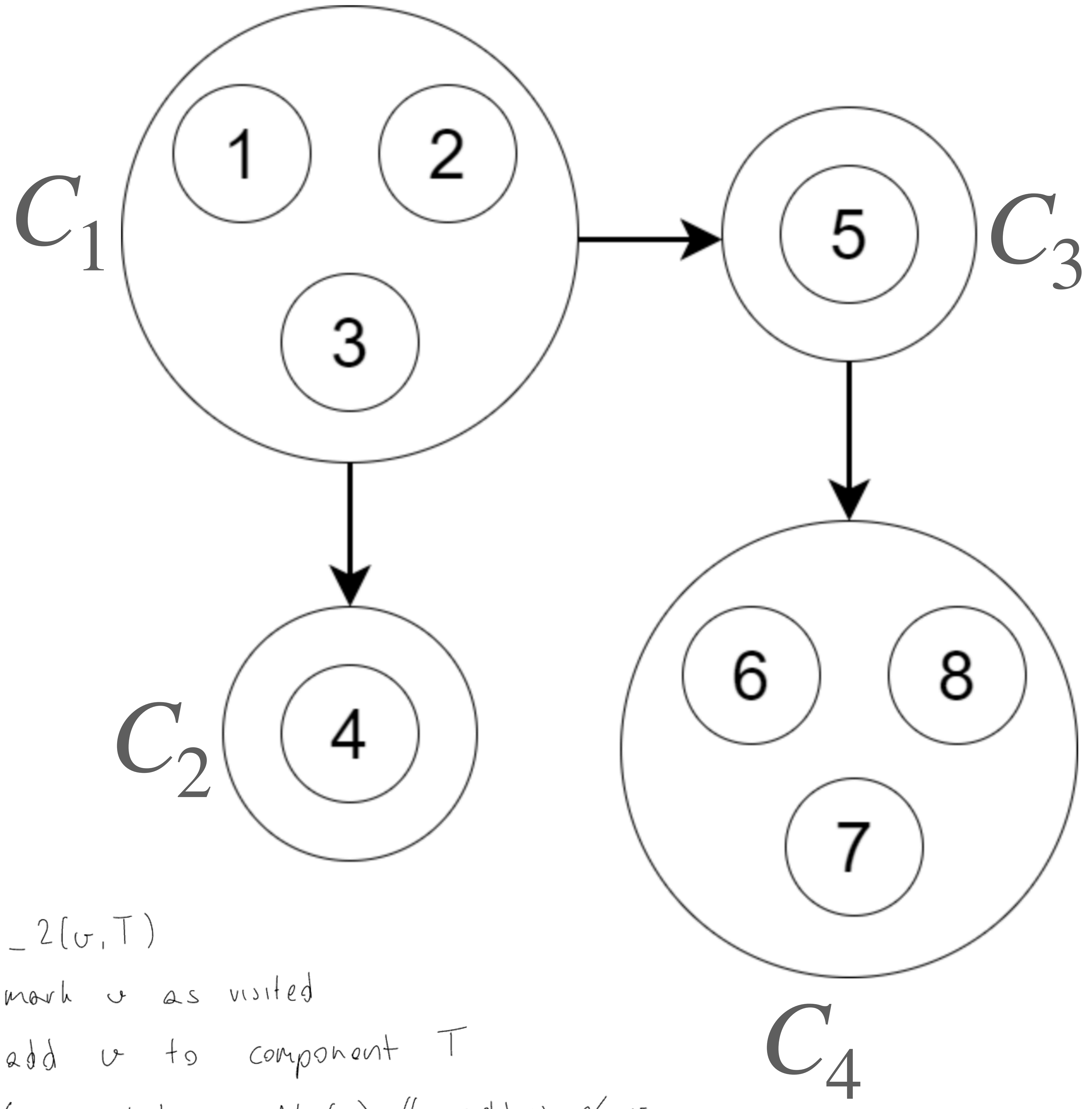
Now reverse the condensation graph.

Now going through vertices in reverse post order ensures we go through the components one by one.



Condensation Graph

Now going through vertices in reverse post order ensures we go through the components one by one.



```
mark all $v \in V$ as unvisited
reverse L
$C \leftarrow \emptyset$    // list of components
create $G^{-1} = (V, E^{-1})$ from $G$
for unvisited $v \in L$ // in order of L
    $T \leftarrow \emptyset$    // single component
    DFS_2(v, T)
    add T to C
```

```
DFS_2(v, T)
    mark v as visited
    add v to component T
    for unvisited $u \in N_{G^{-1}}(v)$ // neighbors of v in $G^{-1}$
        DFS_1(u)
```

(b)* Let $L = [v_1, v_2, \ldots, v_n]$ be a list containing the vertices of G in the *reversed* post-order of a DFS. Show that L has the following property:

(c) Let $\overleftarrow{G} = (V, \overleftarrow{E})$ be the directed graph obtained by inverting all edges in G. Let v_1 be the first element of L. Let $W \subseteq V$ be the set of vertices w for which there is a directed path from v_1 to w in $\overleftarrow{G}$. Show that W is a strongly connected component of G.

(d) Describe an algorithm that outputs all strongly connected components of G. The runtime of your algorithm should be at most $O(n + m)$. Prove that your algorithm is correct, and achieves the desired runtime.

Debriefing of Exercise Sheet 10

Theory Recap

Bellman Ford’s Algorithm

Bellman Ford

Single-source shortest path for weighted graph (now also negative weights!)

- Like Dijkstra, Bellman Ford is an algorithm for finding the shortest paths from a single source s.
- Like Dijkstra, Bellman Ford works with edge relaxation
- Unlike Dijkstra, Bellman Ford works on graphs with negative weights as well (and detects negative cycles!)
- Unlike Dijkstra, Bellman Ford doesn't (greedily) relax edges using a priority queue, but relaxes all edges each iteration.

Bellman Ford

Idea

- A path in a graph $G = (V, E)$ is maximally of length $|V| - 1$ (recall graph exercises!)
- If we relax all edges $|V| - 1$ times, the values for the shortest path from s to any $v \in V$ shouldn't change anymore.
- **Unless**, there is a negative cycle! In this case, the algorithm simply reports the finding of a negative cycle and terminates.
- Using these ideas we do the following: relax all edges $|V| - 1$ times. Relax all edges once more, if distances decrease, we have would have a path of length $|V|$, meaning a negative cycle exists in the graph.

Bellman Ford

Pseudocode

```
BELLMAN-FORD($G = (V, E), s$)

for each $v \in V \setminus \{s\}$ do                     ▷ Initialisiere für alle Knoten die
    $d[v] \leftarrow \infty$; $p[v] \leftarrow$ null                 ▷ Distanz zu s sowie Vorgänger
$d[s] \leftarrow 0$; $p[s] \leftarrow$ null                          ▷ Initialisierung des Startknotens
for $i \leftarrow 1, 2, \dots, |V| - 1$ do                ▷ Wiederhole $|V| - 1$ Mal
    for each $(u, v) \in E$ do                        ▷ Iteriere über alle Kanten $(u, v)$
        if $d[v] > d[u] + w((u, v))$ then               ▷ Relaxiere Kante $(u, v)$
            $d[v] \leftarrow d[u] + w((u, v))$                 ▷ Berechne obere Schranke
            $p[v] \leftarrow u$                              ▷ Speichere u als Vorgänger von v
for each $(u, v) \in E$ do                            ▷ Prüfe, ob eine weitere Kante
    if $d[u] + w((u, v)) < d[v]$ then                   ▷ relaxiert werden kann
        Melde Kreis mit negativem Gewicht
```

Bellman Ford

Runtime

- Relax all edges $|V| - 1$ times.
- Relax all edges once more.
- Edge relaxation is in $O(1)$.
- We have a total runtime of $O(|V||E|)$.

Bellman Ford

Java Implementation

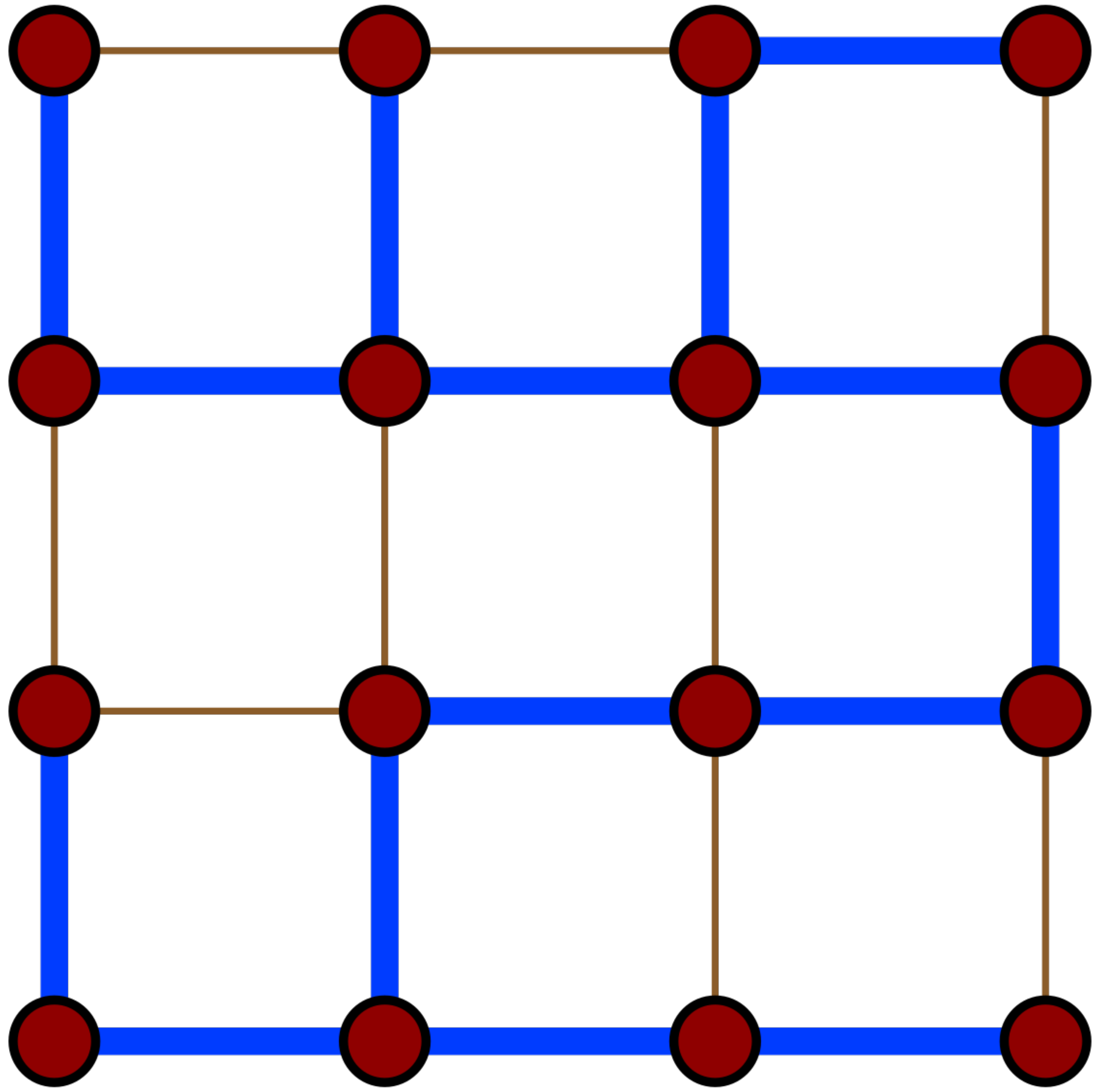
Notice how we handle a potential overflow!

```java
public static int[] bellmanFord(ArrayList<Edge> E, int n, int m, int s) {
    int[] P = new int[n]; // store predecessors of nodes
    int[] D = new int[n]; // store distance from start to nodes

    for (int i = 0; i < n; ++i) D[i] = Integer.MAX_VALUE;
    D[s] = 0;

    for (int i = 1; i < n; ++i) {
        for (Edge e : E) {
            if (D[e.from] != Integer.MAX_VALUE && D[e.to] > D[e.from] + e.weight) {
                D[e.to] = D[e.from] + e.weight;
                P[e.to] = e.from;
            }
        }
    }

    for (Edge e : E) {
        if (D[e.from] != Integer.MAX_VALUE && D[e.to] > D[e.from] + e.weight) {
            System.out.println("negative cycle detected.");
            return null;
        }
    }

    return D;
}
```

Prim’s Algorithm

What is a Spanning Tree?

- A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all the vertices of G.
- Example:
- Can there be multiple?
- Yes!
- A minimum spanning tree (MST) is a spanning tree T, where the sum of all edges in T is minimal. Can there be multiple? Yes!



Prim

Finding the Minimum Spanning Tree (MST)

- Like Dijkstra, Prim greedily selects edges using a priority queue. (cheaper means higher priority)
- Unlike Dijkstra, the priority queue used in Prim only stores the vertex v and the cheapest edge leading to it (from any vertex that is neighbor of v). The edge weight is the priority.

Prim

Idea

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph (we start from here).

2. Grow the tree by one edge: Of the edges that connect the tree to the vertices not yet in the tree, find the minimum-weight edge, and add the edge (including the vertex it connects to) to the tree.

3. Repeat step 2 until the tree contains all vertices (i.e. until the tree is a spanning tree)

Prim

Implementation

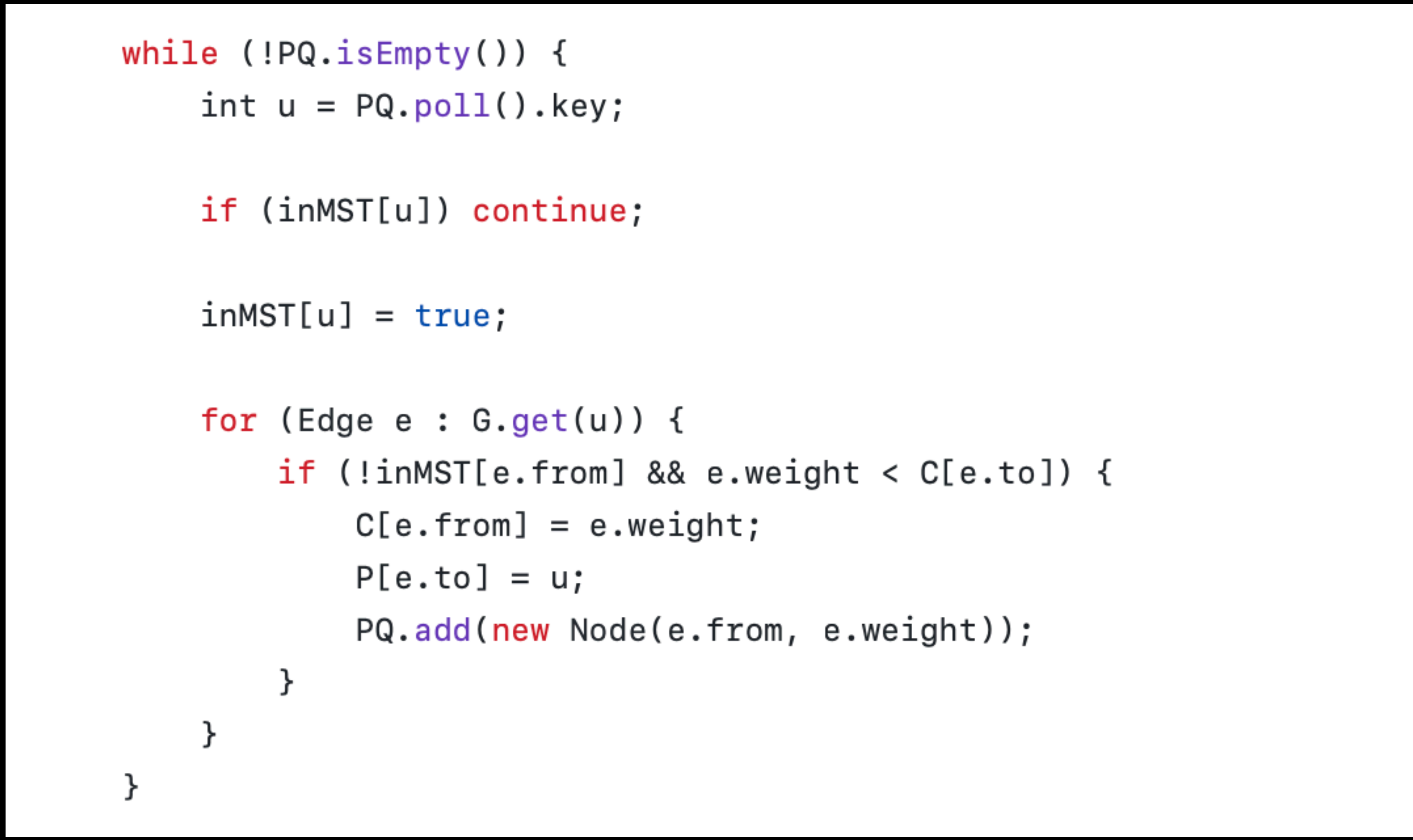
```java
public static int[] prim(List<List<Edge>> G, int n, int source) {
    int[] C = new int[n]; // store cheapest connection cost
    int[] P = new int[n]; // store predecessors of nodes

    for (int i = 0; i < n; ++i) {
        C[i] = Integer.MAX_VALUE;
        P[i] = -1;
    }
    C[source] = 0;

    PriorityQueue<Node> PQ = new PriorityQueue<>();
    for (int i = 0; i < n; ++i) {
        PQ.add(new Node(i, Integer.MAX_VALUE));
    }
    PQ.add(new Node(source, C[source]));

    boolean[] inMST = new boolean[n]; // store if node is in MST

    while (!PQ.isEmpty()) {
        int u = PQ.poll().key;

        if (inMST[u]) continue;

        inMST[u] = true;

        for (Edge e : G.get(u)) {
            if (!inMST[e.from] && e.weight < C[e.to]) {
                C[e.from] = e.weight;
                P[e.to] = u;
                PQ.add(new Node(e.from, e.weight));
            }
        }
    }

    return P;
}
```

Prim

Runtime

- Using a binary heap as a priority queue we can achieve a runtime of $O((|V| + |E|)\log|V|)$ (like Dijkstra).

Example on Blackboard



```java
while (!PQ.isEmpty()) {
    int u = PQ.poll().key;

    if (inMST[u]) continue;

    inMST[u] = true;

    for (Edge e : G.get(u)) {
        if (!inMST[e.from] && e.weight < C[e.to]) {
            C[e.from] = e.weight;
            P[e.to] = u;
            PQ.add(new Node(e.from, e.weight));
        }
    }
}
```