CS 170 DISCUSSION 4

GRAPHS AND PATHS

Raymond Chan
UC Berkeley Fall 17

DEPTH FIRST SEARCH

- Explore all and only nodes reachable from current node.

```
recursive_DFS (G, v):
  previsit(v)
  mark v as visited
  for all v's neighbors w:
    if vertex w has not been visited:
      recursive_DFS (G, w)
  postvisit(v)
```

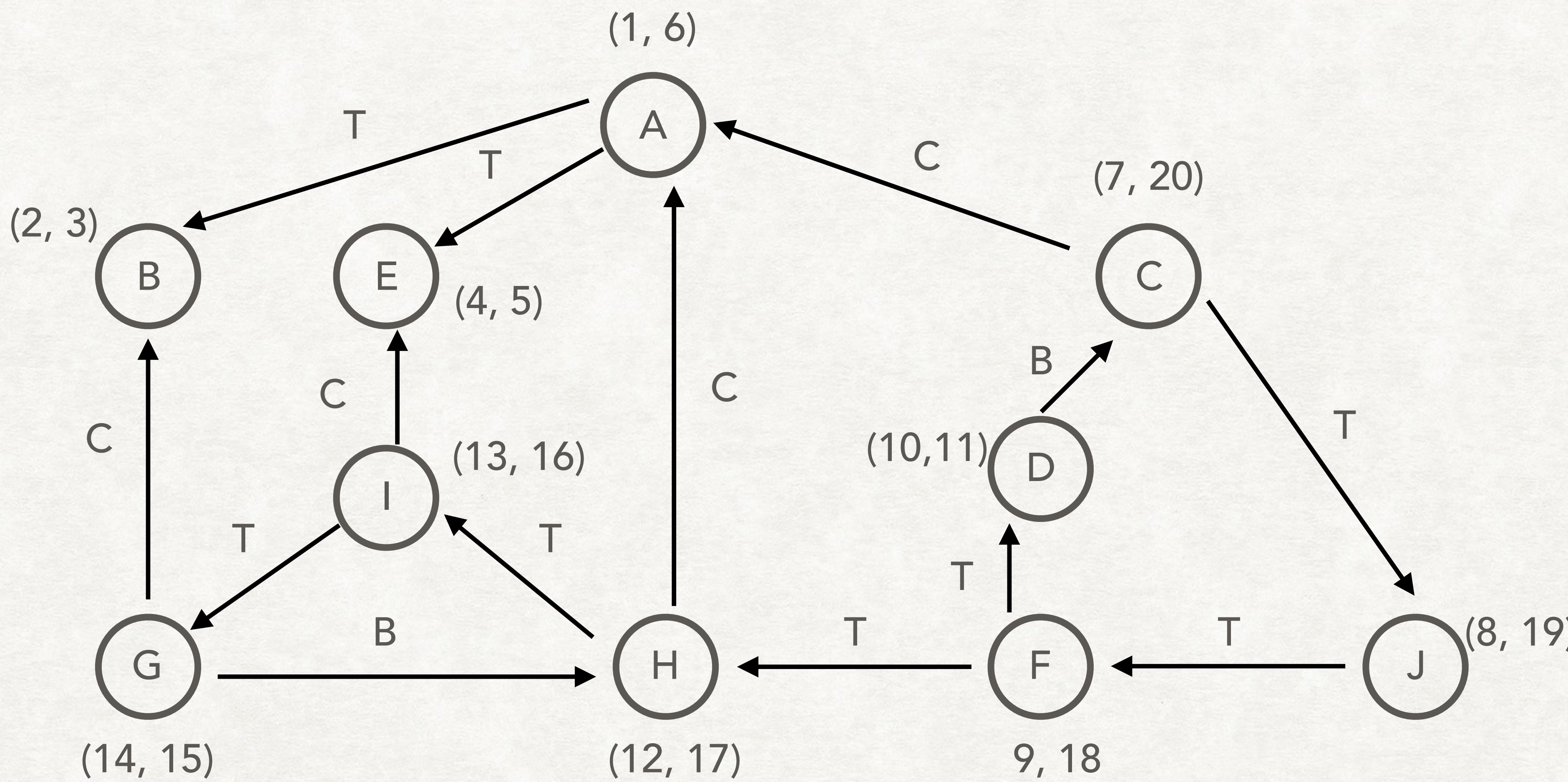
```
iterative_DFS (G, v):
  stack.push(v)
  while stack not empty:
    v = pop from stack
    if v not visited
      mark v as visited:
      for all v's neighbors w:
        push vertex w to stack
```

- Visits all vertices once. Uses all edges once.

- $O(|V| + |E|)$

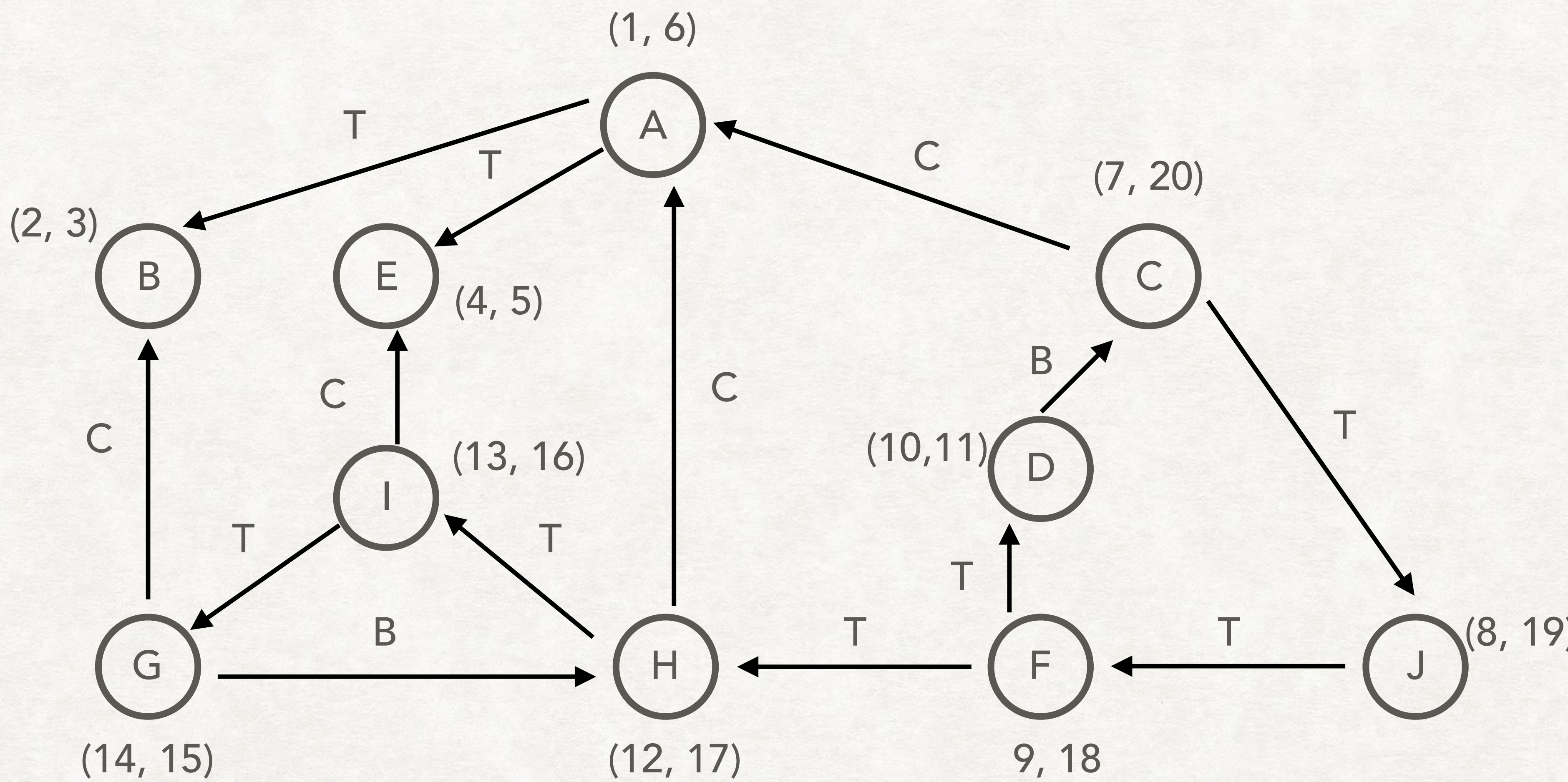
DEPTH FIRST SEARCH

- From last discussion



PREVISIT & POSTVISIT

- Previsit() and Postvisit() increments a global count



PREVISIT & POSTVISIT T/F

- If (u, v) is an edge in an indirect graph and during DFS, post(v) < post(u), then u is an ancestor of v in the DFS tree.

PREVISIT & POSTVISIT T/F

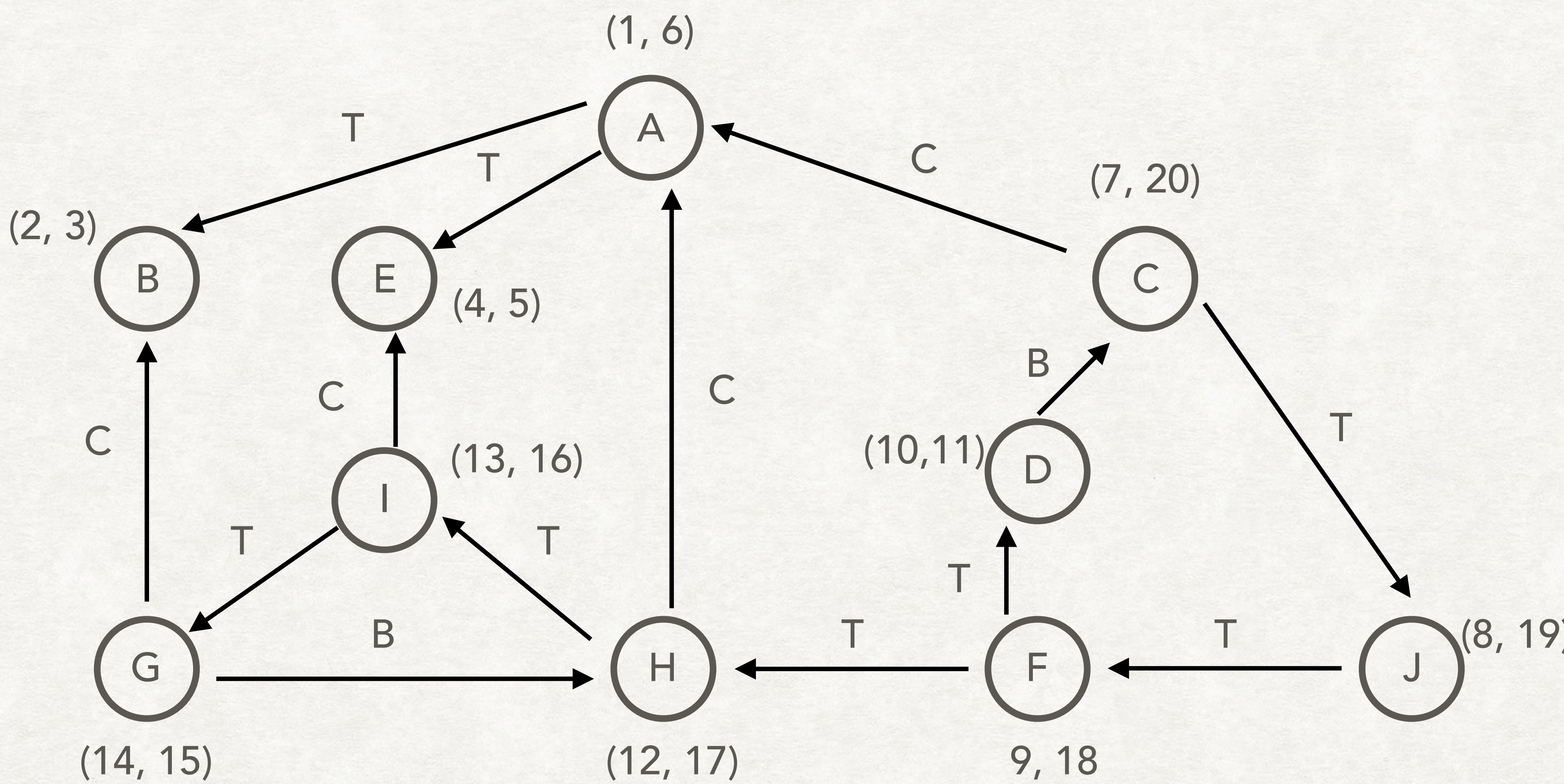
- If (u, v) is an edge in an indirect graph and during DFS, post(v) < post(u), then u is an ancestor of v in the DFS tree.
- 2 cases (since pre < post):
  - pre(u) < pre(v) < post(v) < post(u)
    - u is an ancestor v. Explore u's neighbors before postvisiting u

PREVISIT & POSTVISIT T/F

- If (u, v) is an edge in an indirect graph and during DFS, post(v) < post(u), then u is an ancestor of v in the DFS tree.
- 2 cases (since pre < post):
  - pre(u) < pre(v) < post(v) < post(u)
    - u is an ancestor v. Explore u's neighbors before postvisiting u
  - pre(v) < post(v) < pre(u) < post(u)
    - Looks at all of v's neighbors before looking at u.
    - Contradiction since there is an edge.
- True

PREVISIT & POSTVISIT

- For any two nodes, u, v, [pre(u), post(u)] and [pre(v), post(v)] are either disjoint or one is contained in the other.

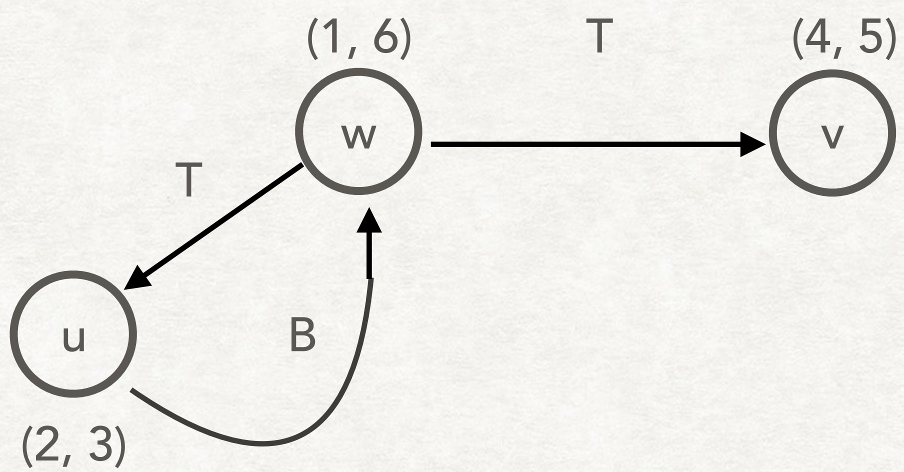


PREVISIT & POSTVISIT T/F

- In a directed graph, if there is a path from u to v and pre(u) < pre(v) then u is an ancestor of v in the DFS tree.

PREVISIT & POSTVISIT T/F

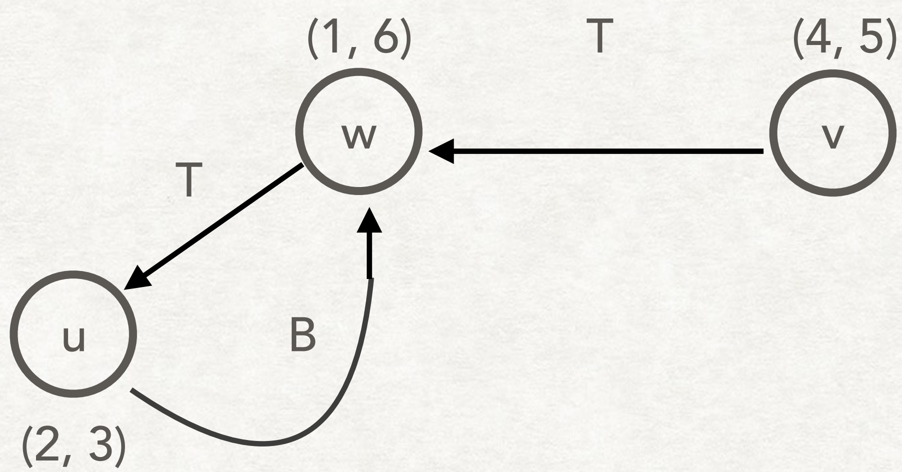
- In a directed graph, if there is a path from u to v and pre(u) < pre(v) then u is an ancestor of v in the DFS tree.
- Consider the case when u and v are a common ancestor's direct children.



PREVISIT & POSTVISIT T/F

- In a directed graph, if there is a path from u to v and pre(u) < pre(v) then u is an ancestor of v in the DFS tree.

- Consider the case when u and v are a common ancestor's direct children.



- u gets visited first via w, who then visits v.

- Need information about post visit number

- False

PREVISIT & POSTVISIT T/F

- In any connected undirected graph G, there is a vertex whose removal leaves G connected.

PREVISIT & POSTVISIT T/F

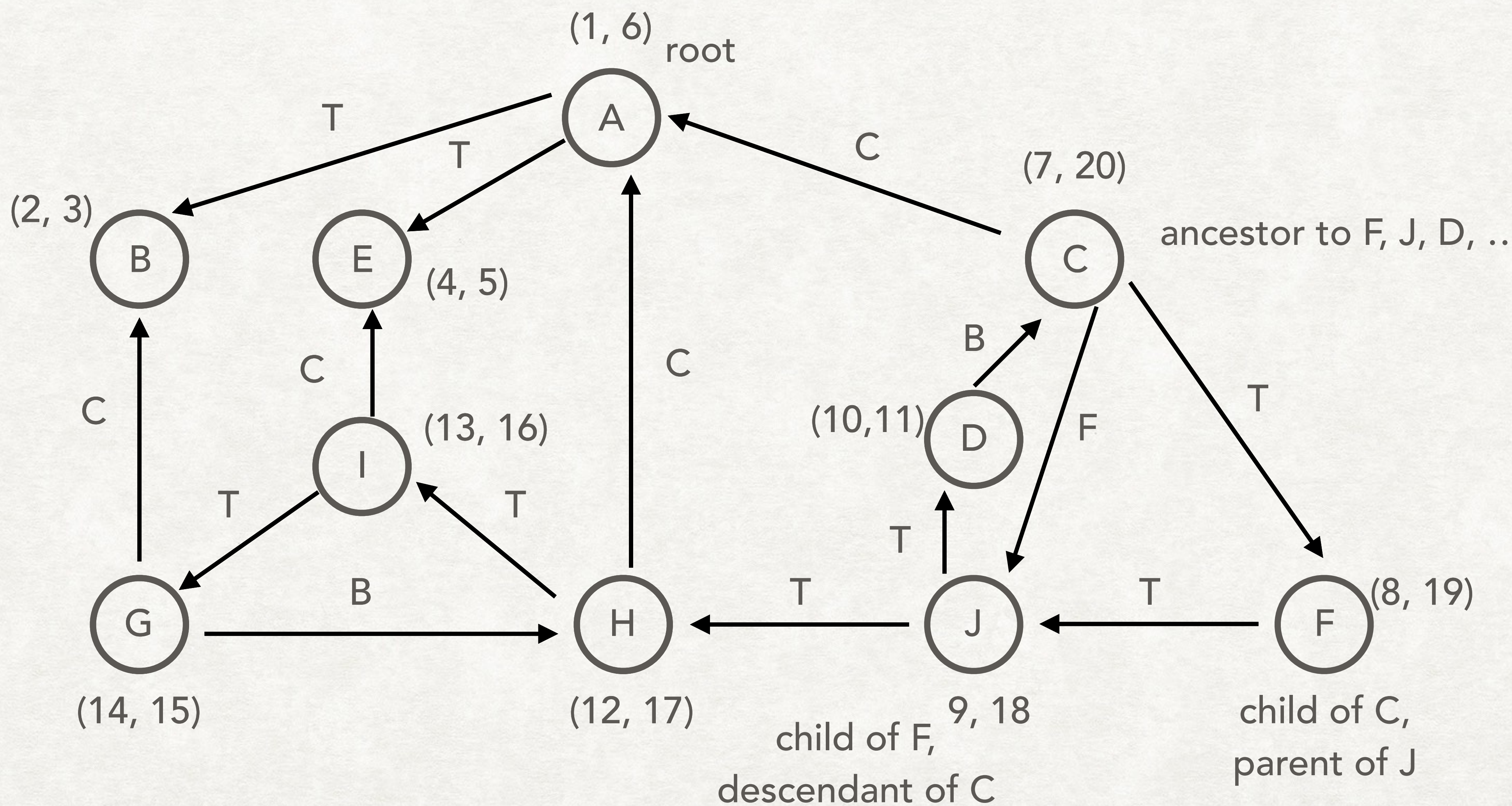
- In any connected undirected graph G, there is a vertex whose removal leaves G connected.
- True

PREVISIT & POSTVISIT T/F

- In any connected indirect graph G, there is a vertex whose removal leaves G connected.
- True
- Removing any leaf from a DFS tree of the graph.
- These leaves have only one neighbor, otherwise they won't be leaves.
- Neighbor is connected to at least one other node.

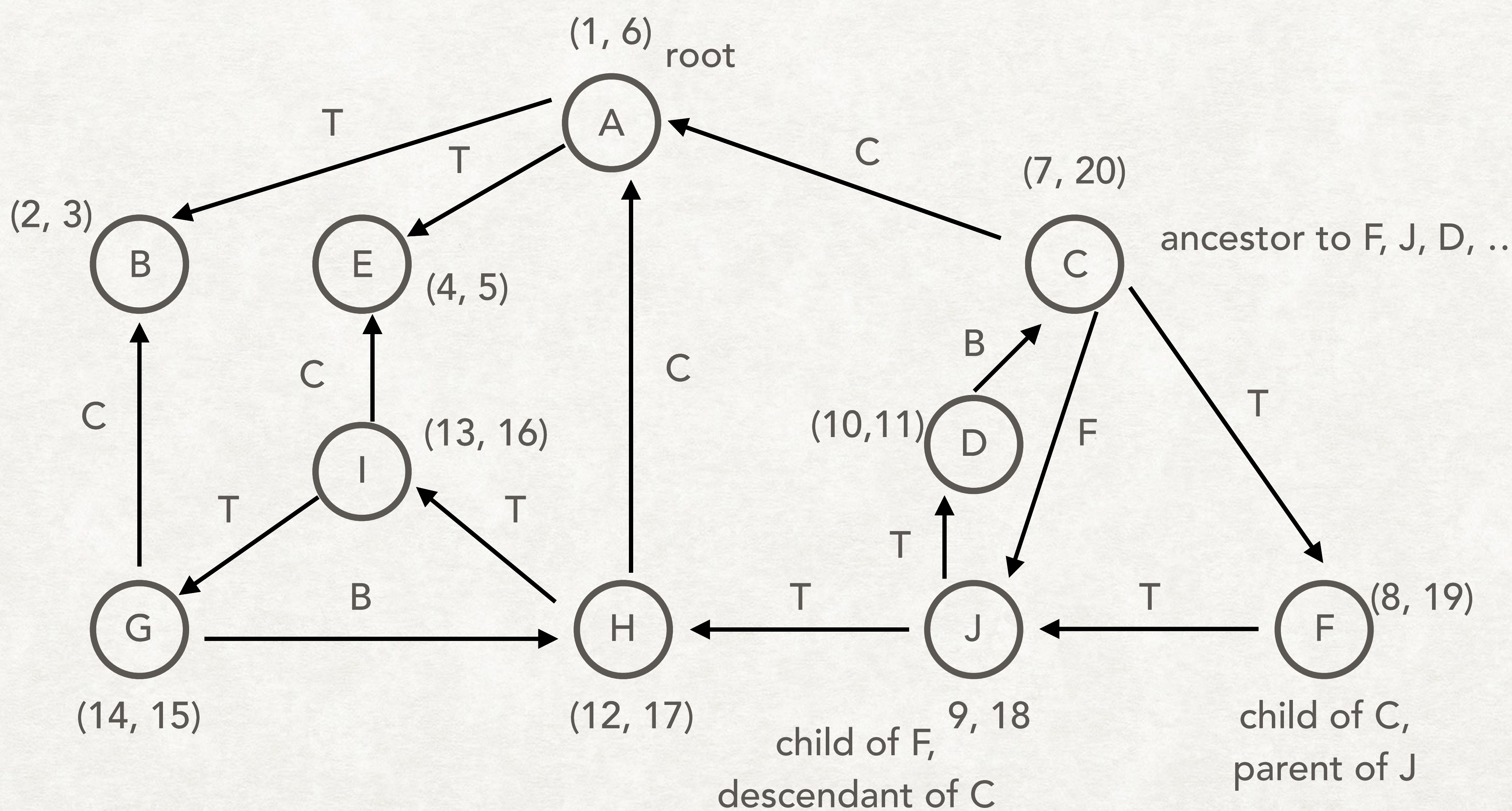
DEPTH FIRST SEARCH

- Tree edge (u, v): part of DFS
    - pre(u) < pre(v) < post(v) < post(u)



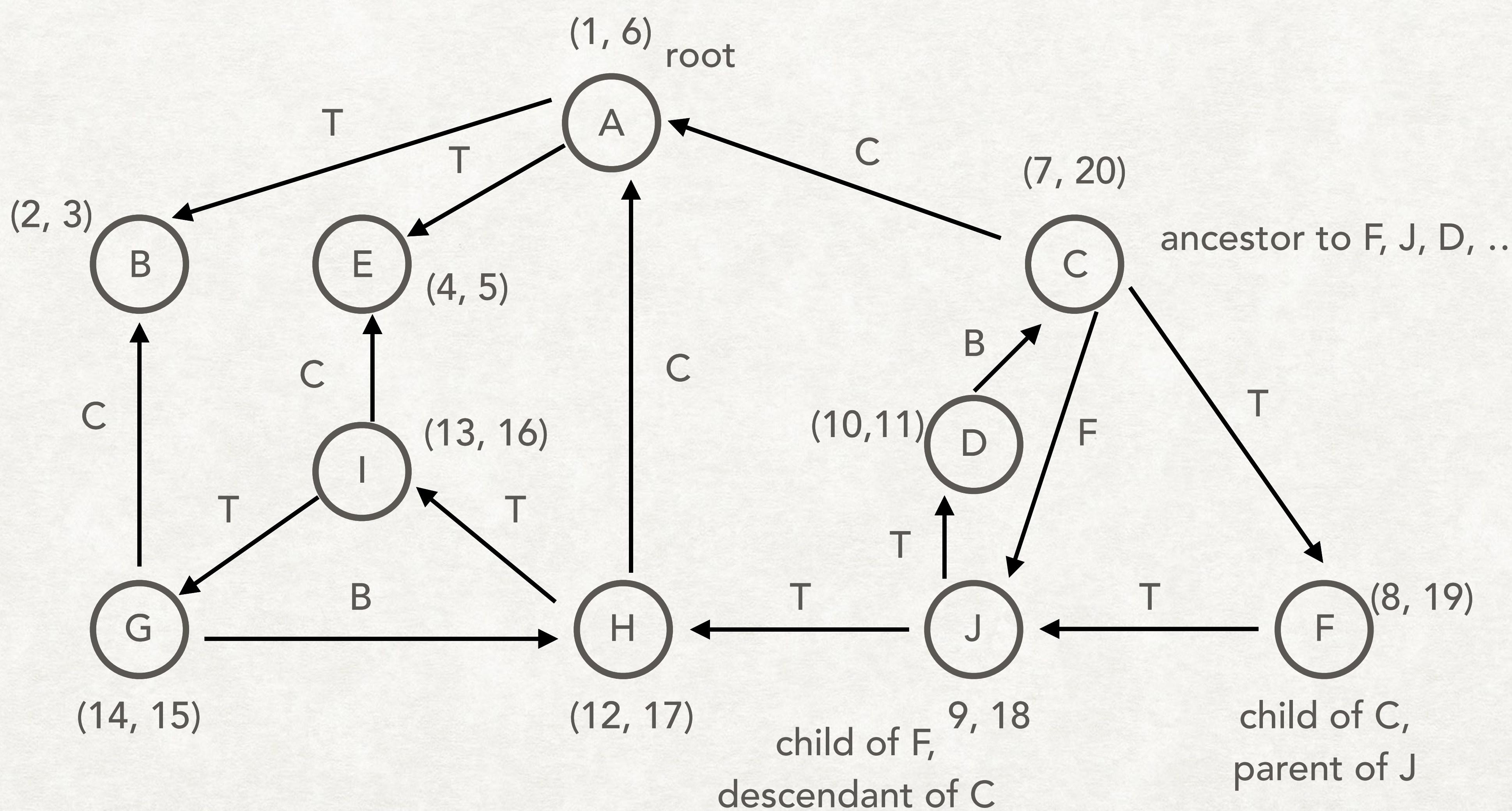
DEPTH FIRST SEARCH

- Forward edge (u, v): leads non-child descendant
    - pre(u) < pre(v) < post(v) < post(u)



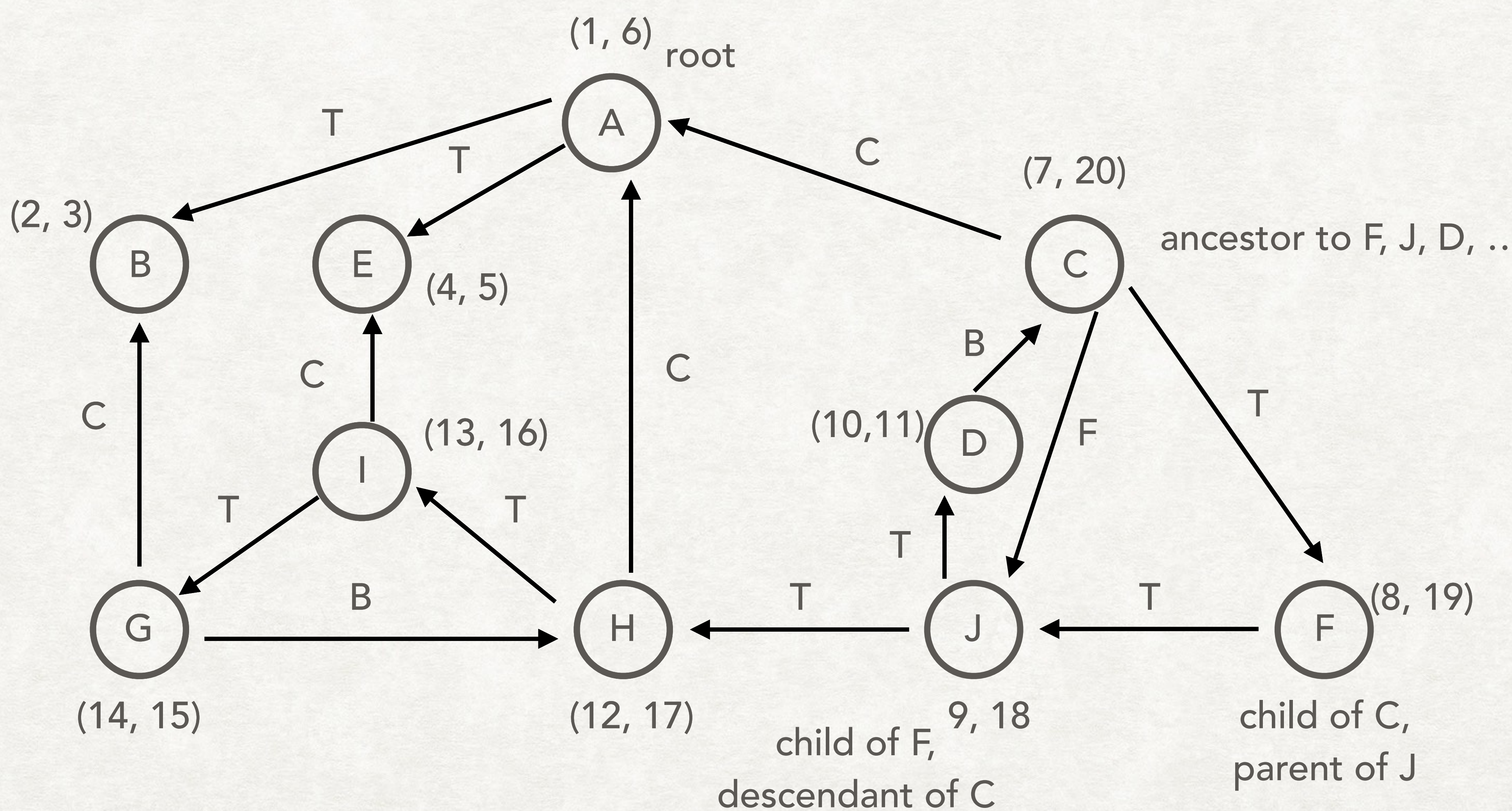
DEPTH FIRST SEARCH

- Cross edge (u, v): leads neither defendant nor ancestor
    - pre(v) < post(v) < pre(u) < post(u)



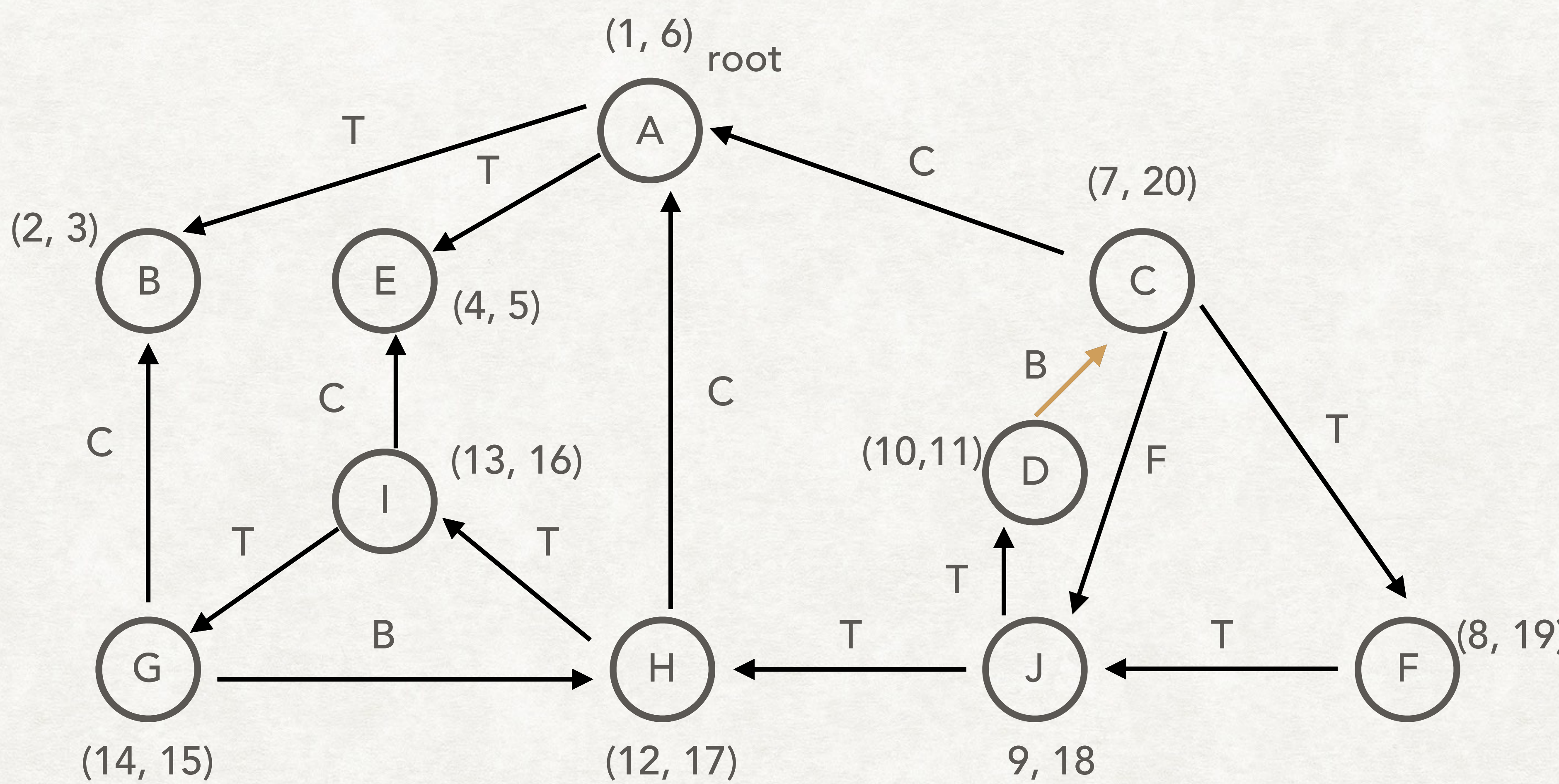
DEPTH FIRST SEARCH

- Back edge (u, v): leads to ancestor
  - pre(v) < pre(u) < post(u) < post(v)



CYCLE DETECTION

- There is a cycle if and only if there is a back edge.
- Run DFS.

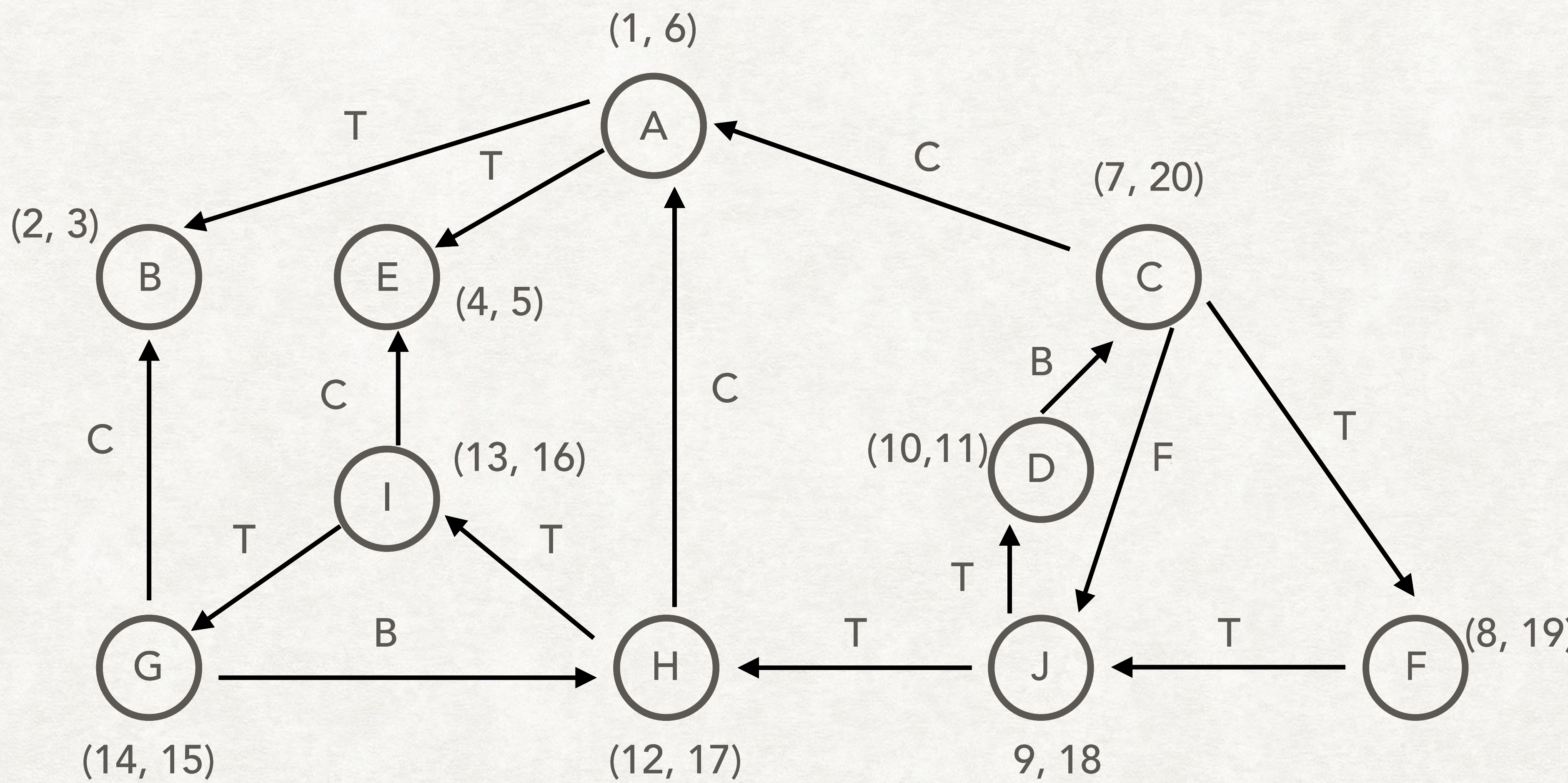


STRONGLY CONNECTED COMPONENTS

- Nodes u and v are strongly connected if there is a path from u to v and there is a path from v to u.
- Strongly connected components are a set of nodes that are strongly connected.
- Can visit every other node in the set.
- Only applies to directed graphs.
- Single case: single node.

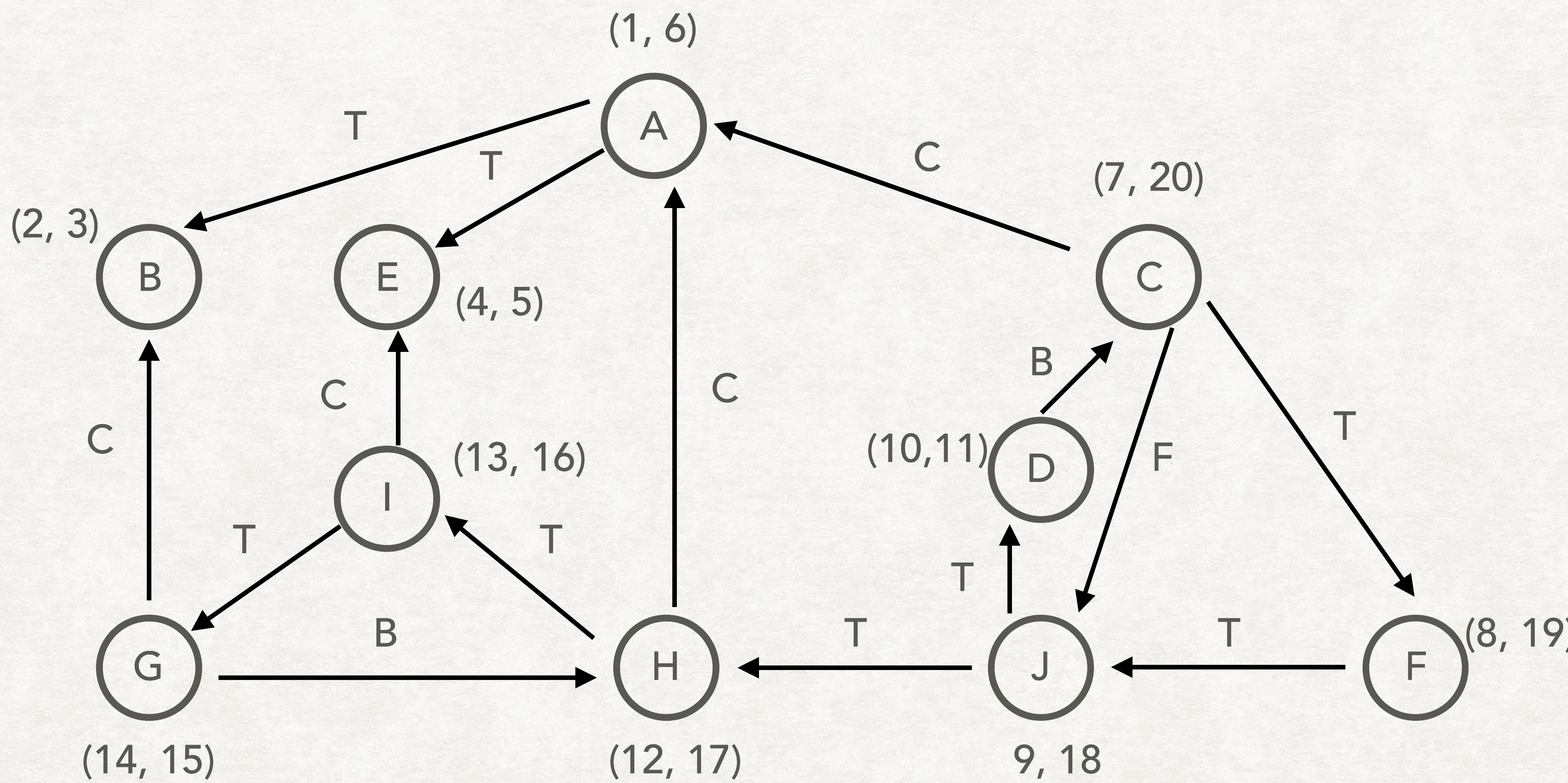
STRONGLY CONNECTED COMPONENTS

- Find the strongly connected components.



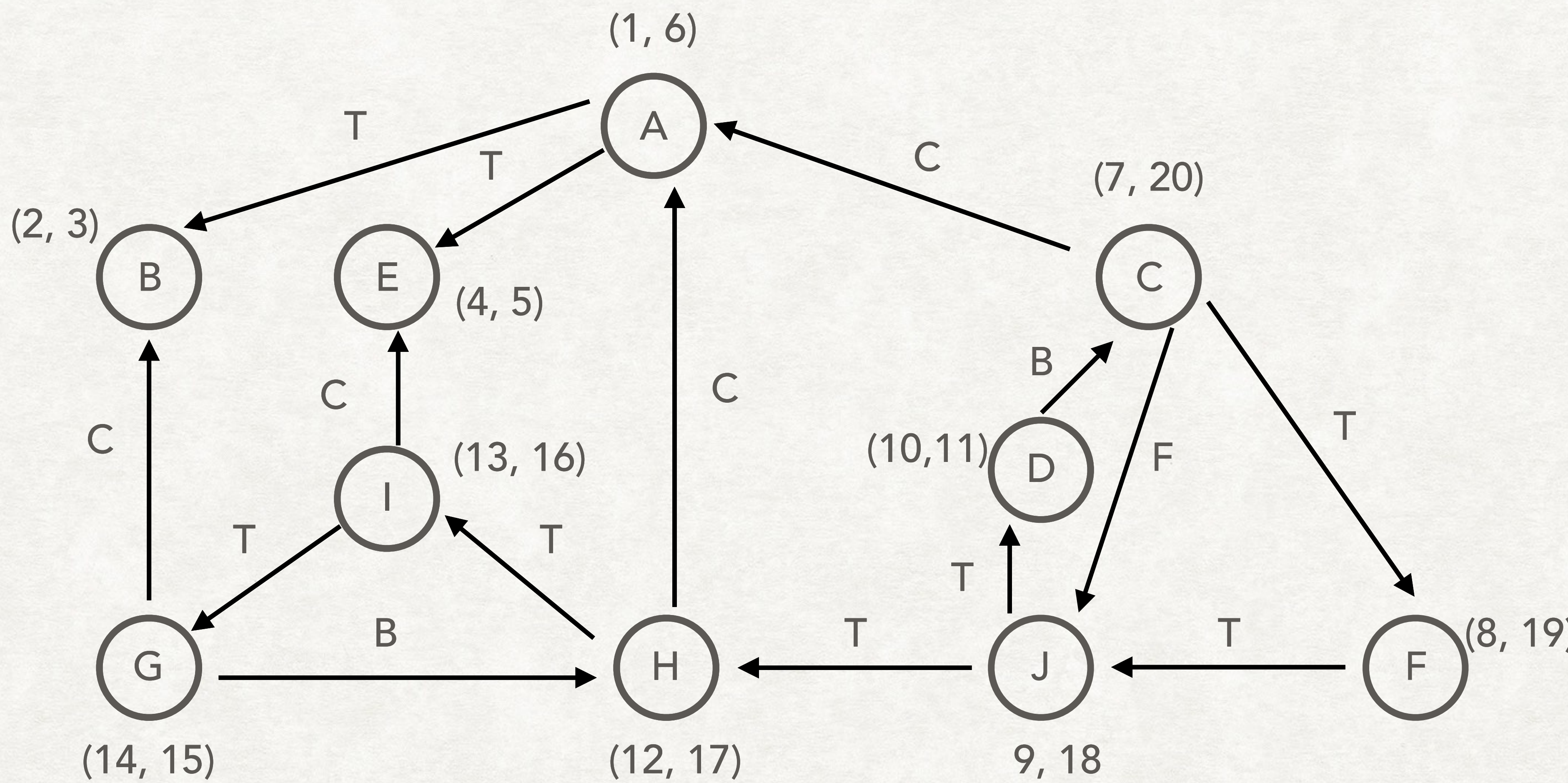
STRONGLY CONNECTED COMPONENTS

- Find the strongly connected components.
- {A}, {B}, {E}, {C, F, J, D}, {G, I, H}



STRONGLY CONNECTED COMPONENTS

- Visualize it
  - https://www.cs.usfca.edu/~galles/JavascriptVisual/ConnectedComponent.html



TOPOLOGICAL SORT

- Source and Sink in a Directed Acyclic Graph (DAG)
- Source:
  - Node has no incoming edges
  - Highest post order number
  - First nodes in topological ordering
- Sink
  - No outgoing edges
  - Lowest post oder number
  - Last nodes in topological ordering
- Every DAG has at least one source and one sink

TOPOLOGICAL SORT

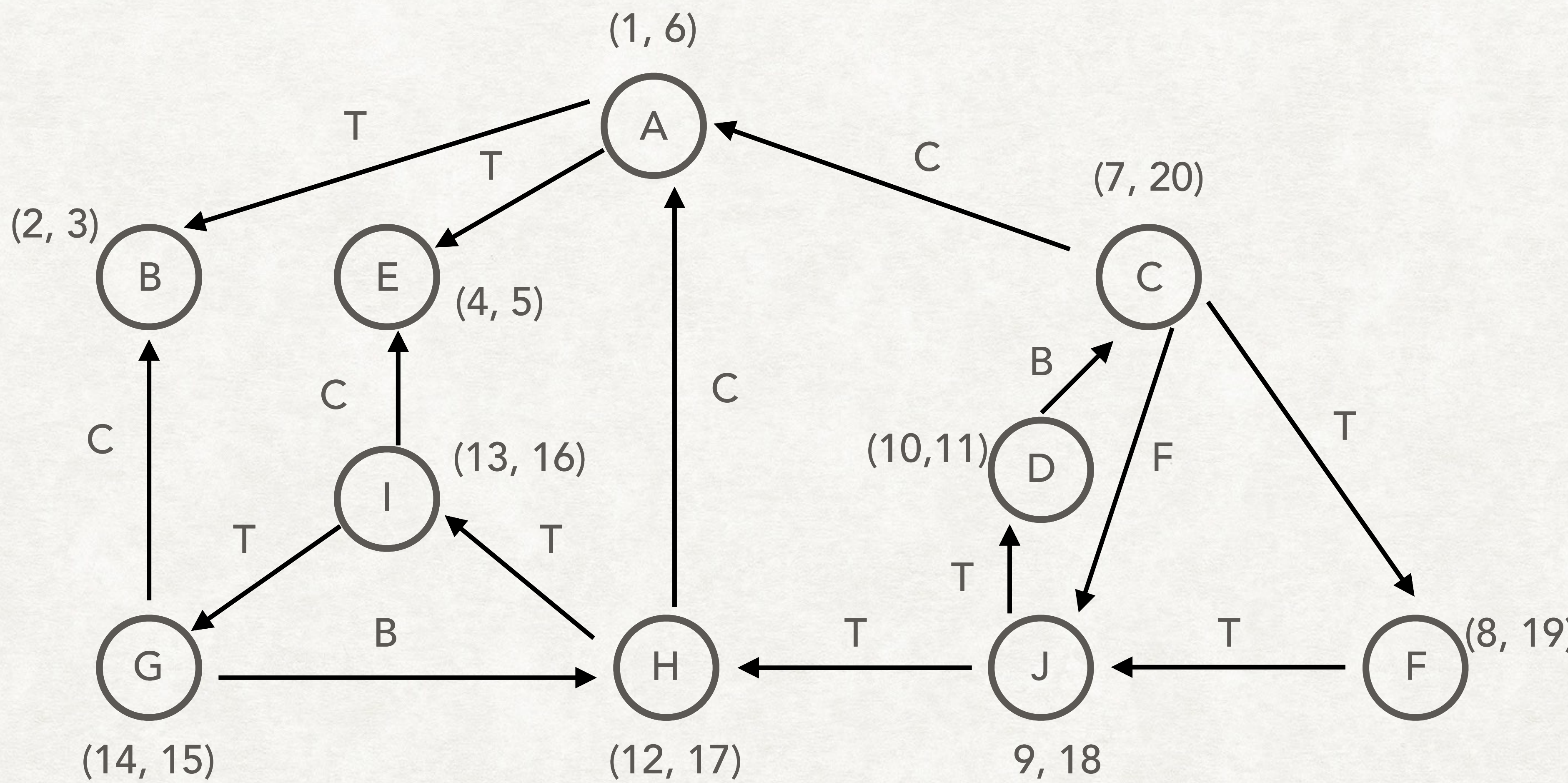
- Ordering of a directed graph's nodes v_1, v_2, … v_n such that for every edge (v_i, v_j), $i < j$.

- Edge arrows go one direction.

- Application: scheduling jobs if order is required.

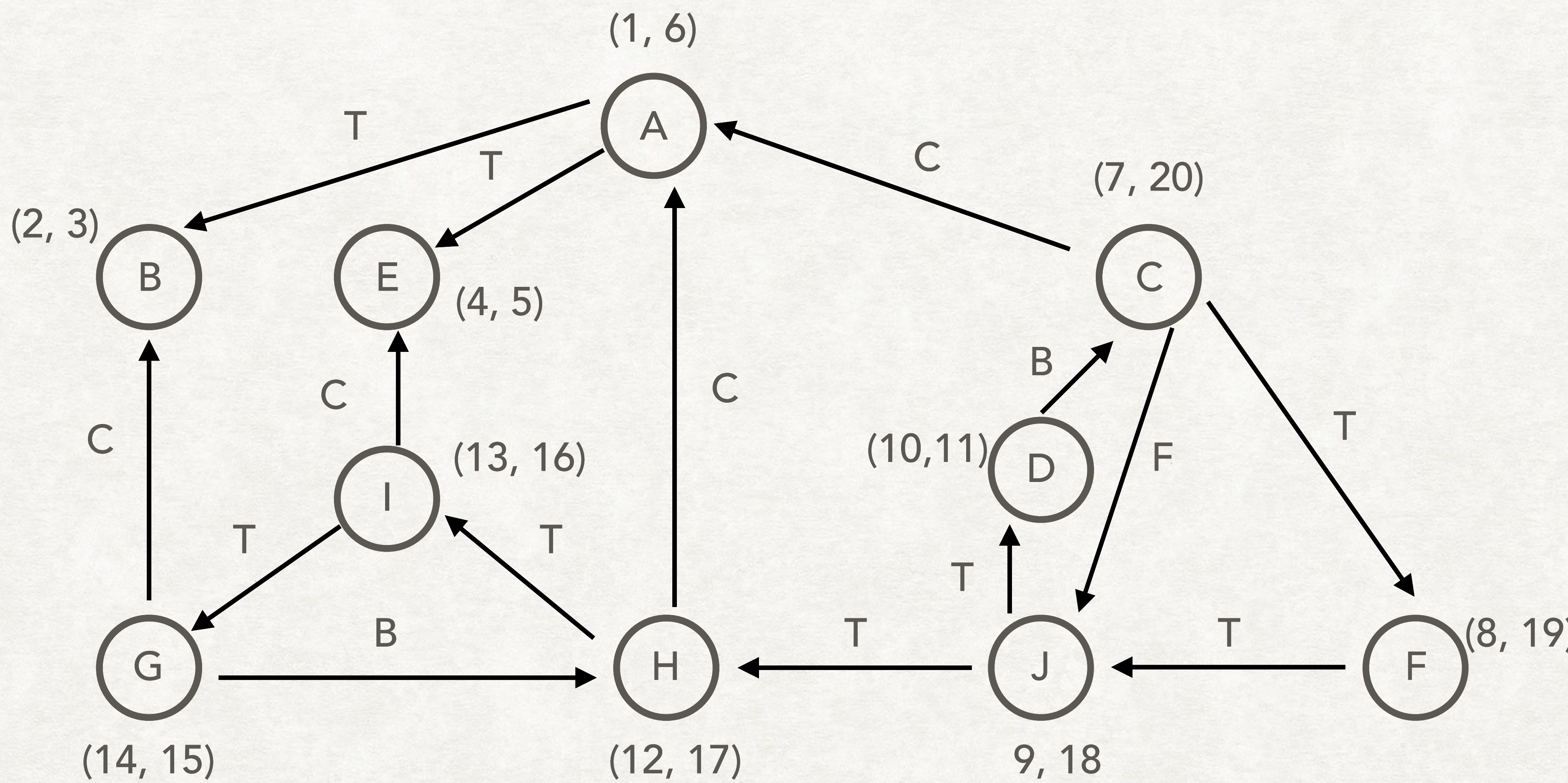
TOPOLOGICAL SORT

- Sort the Strongly Connected Components into a DAG



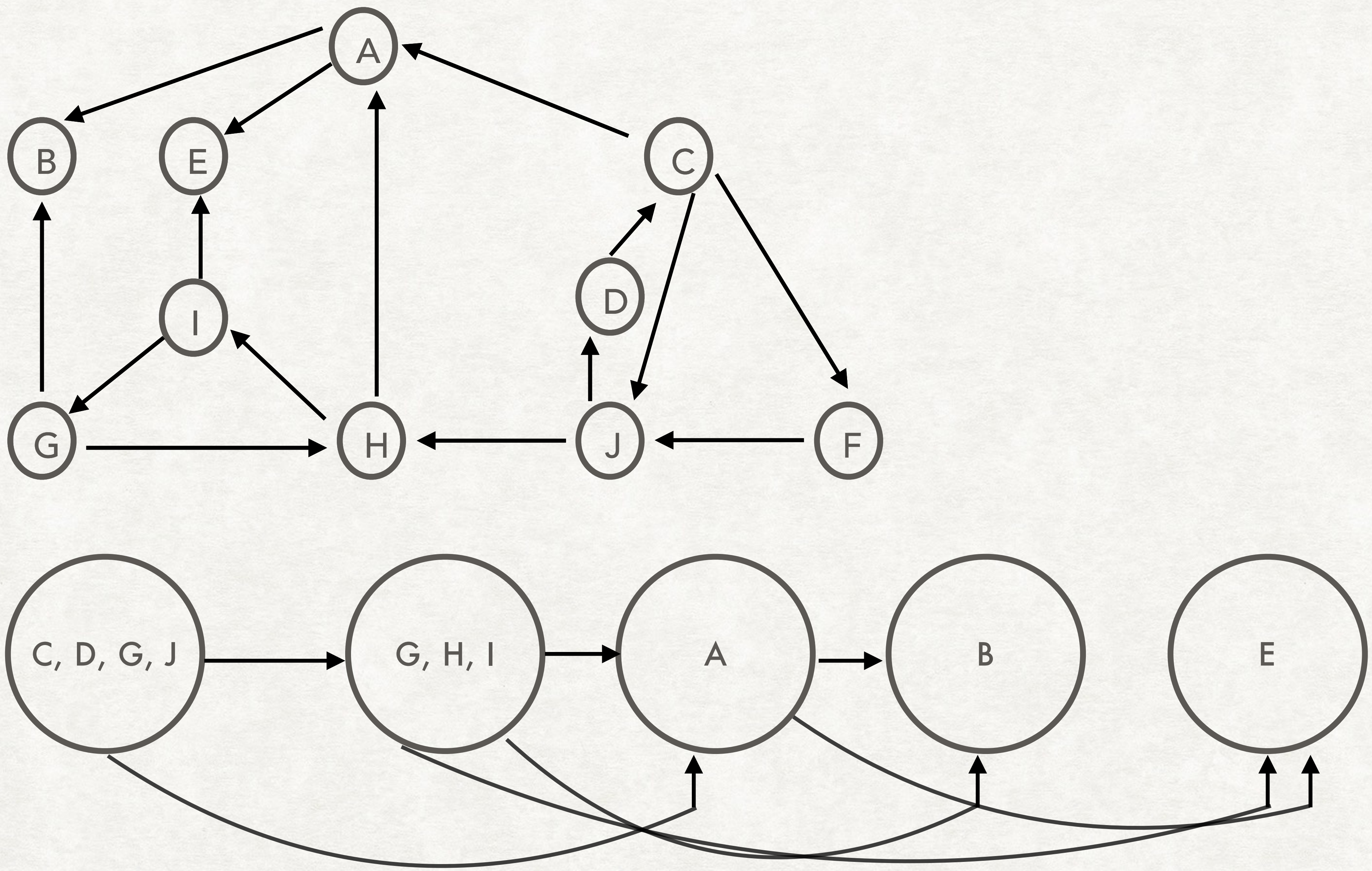
TOPOLOGICAL SORT

- Sort the Strongly Connected Components into a DAG
- {A}, {B}, {E}, {C, F, J, D}, {G, I, H}



TOPOLOGICAL SORT

- {A}, {B}, {E}, {C, F, J, D}, {G, I, H}



ALGORITHMS

```
SCC(G):
  Reverse edges of graph -> G^R
  DFS on G^R
  Run DFS on G in reverse post
    order number from G^R
```

- Reverse graph prevents crossing SCCs
- Tarjan’s SCC algorithm

```
Linearize (G, v):
  push all sources to v
  while stack not empty:
    v = pop from stack
    for all v’s neighbors w:
      remove edge (v, w)
      if w becomes a sources:
        push vertex w to stack
    Add v to result
```

```
Sort by reverse post order number
```

BREADTH FIRST SEARCH

- Explore node u’s neighbors, then all vertices that are adjacent to u’s neighbors, and so on.
- Can keep distance values to find how many edges a node is away from the root.
- Visits all vertices once.
- Uses all edges once.
- O(|V| + |E|)

```
iterative_BFF (G, v):
  d[v] = 0
  queue.enqueue(v)
  while queue not empty:
    v = pop from queue
    for all v’s neighbors w:
      if w not visited
        mark w as visited:
        d[w] = d[v] + 1
        queue.enqueue(w)
```

DIJKSTRA'S SHORTEST PATH

- Find shortest path from s to all other vertices.
- Once we have computed the shortest path for a vertex, we don't revisit it again.

```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```

DIJKSTRA'S SHORTEST PATH

- Keep fringe vertices.
- Look at neighbors and update distance if there is a better path.
- When we pop off a node from Priority Queue, the distance is the shortest path from s to this node so far.

```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```

DIJKSTRA'S SHORTEST PATH

Demo

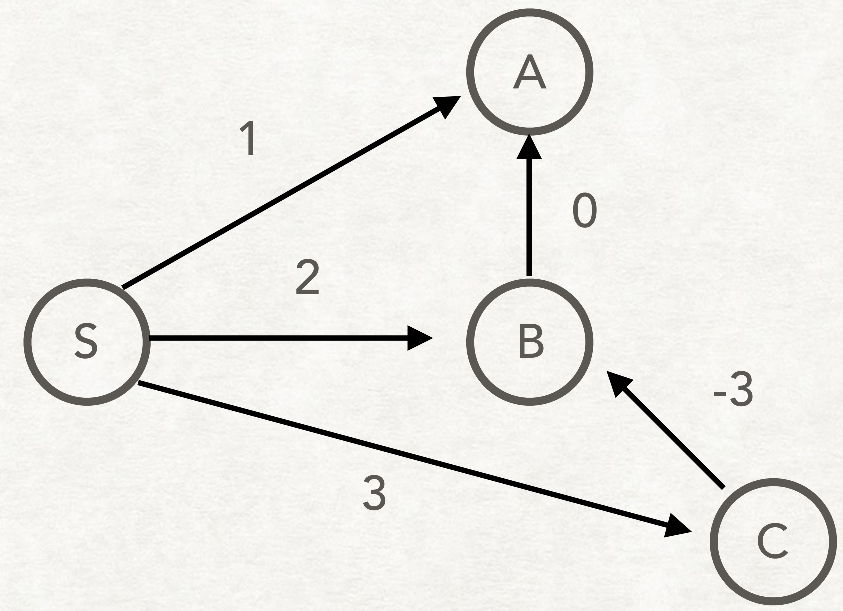
```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```

O(|V|) * DeleteMin + O(|E|)*DecreaseKey

Typically use binary heap for priority queue

O(|V|+|E|) * log(|V|)

NEGATIVE EDGE WEIGHTS




```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```

NEGATIVE EDGE WEIGHTS

Start at S

Process A. d[a] = 1

Process B. d[b] = 2

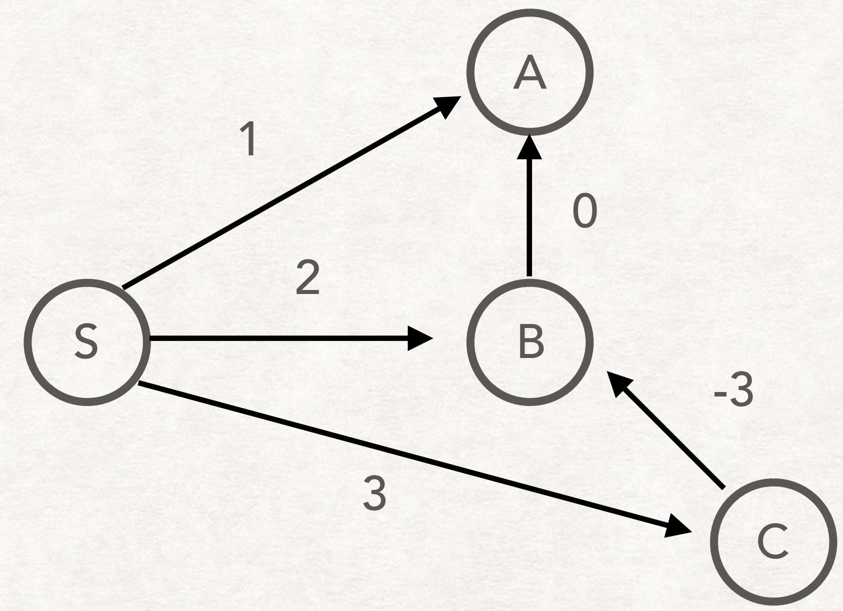
Process C. d[c] = 3

Process A. d[a] = 1.

Process B. d[b] = 2. d[a] = 1

Process C. d[c] = 3. d[b] = 0

New distance for B, but B not in PQ




```
dijkstra (G, s):
  d[v] = infinity
  d[s] = 0
  prev[s] = s
  PQ.add(G.V, infinity)
  PQ.add(s,0)
  while PQ not empty:
    u = PQ.DeleteMin()
    for edge (u, v):
      if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w[u, v]
        prev[v] = u
        PQ.DecreaseKey(v, d[v])
```