Graph

Applications of Graph-Search Algorithms

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Reverse graph

- Given a directed graph $G = (V, E)$
- Its reverse (or transpose or converse) graph $G^T = (V, E^T)$
  - Is another directed graph on the same set of vertices with all the edges reversed compared to the original orientation
  - If G contains an edge (u, v) then G^T contains (v, u)

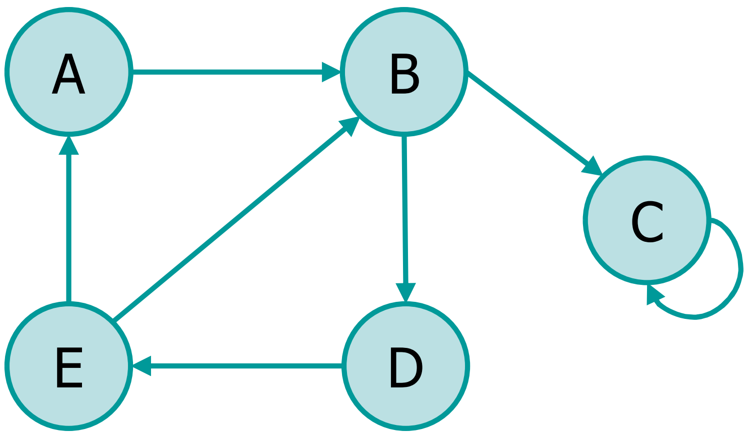
$$\forall\ (u, v) \in E \quad \rightarrow \quad (v, u) \in E^T$$

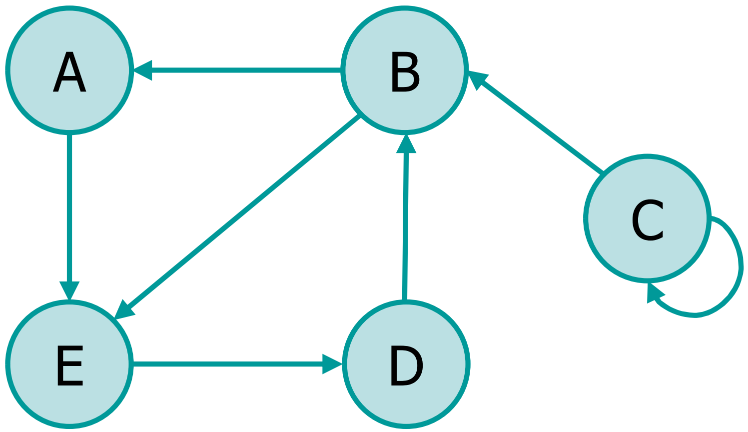
Example

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	1	0	0
D	0	0	0	0	1
E	1	1	0	0	0

$G \Rightarrow G^T$

	A	B	C	D	E
A	0	0	0	0	1
B	1	0	0	0	1
C	0	1	1	0	0
D	0	1	0	0	0
E	0	0	0	1	0





Implementation (with adjacency matrix)

```
graph_t *graph_transpose (graph_t *g) {
  graph_t *h;
  int i, j;

  h = (graph_t *) util_calloc (1, sizeof (graph_t));
  h->nv = g->nv;
  h->g = (vertex_t *) util_calloc (g->nv, sizeof(vertex_t));
  for (i=0; i<h->nv; i++) {
    h->g[i] = g->g[i];
    h->g[i].rowAdj = (int *) util_calloc (h->nv, sizeof(int));
    for (j=0; j<h->nv; j++) {
      h->g[i].rowAdj[j] = g->g[j].rowAdj[i];
    }
  }

  return h;
}
```

Given g it creates and returns h

Transpose the matrix

Implementation (with adjacency list)

```
graph_t *graph_transpose (graph_t *g) {
  graph_t *h = NULL;
  vertex_t *tmp;
  edge_t *e;
  int i;
  h = (graph_t *) util_calloc (1, sizeof(graph_t));
  h->nv = g->nv;
  for (i=h->nv-1; i>=0; i--)
    h->g = new_node (h->g, i);
  tmp = g->g;
  while (tmp != NULL) {
    e = tmp->head;
    while (e != NULL) {
      new_edge (h, e->dst->id, tmp->id, e->weight);
      e = e->next;
    }
    tmp = tmp->next;
  }
  return h;
}
```

Given g it creates and returns h

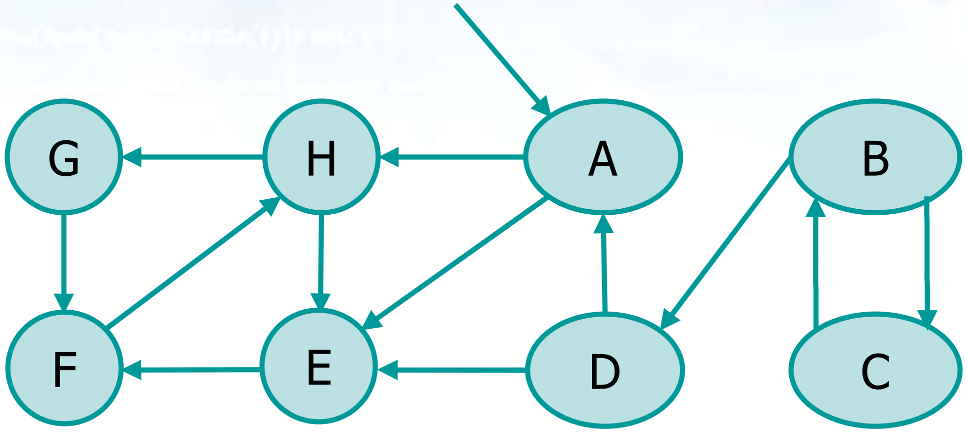
Insert a new edge

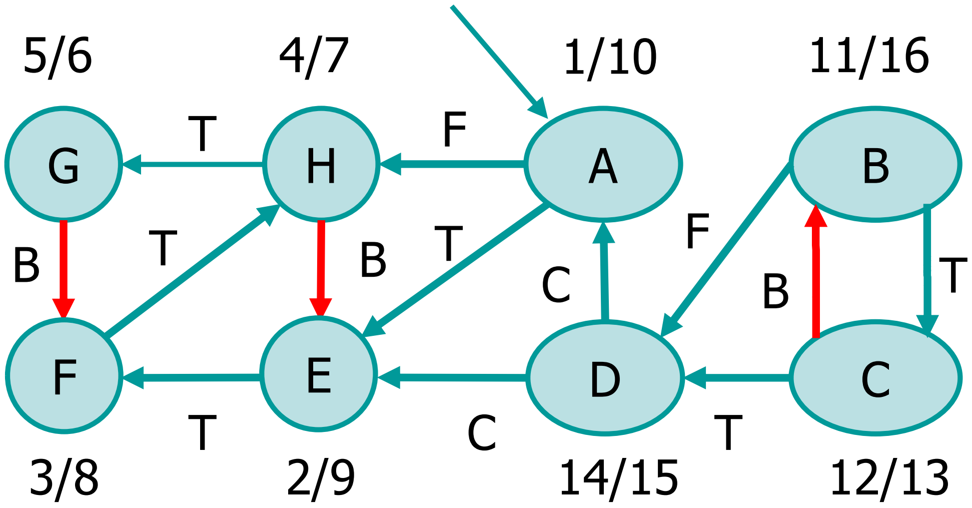
Loop detection

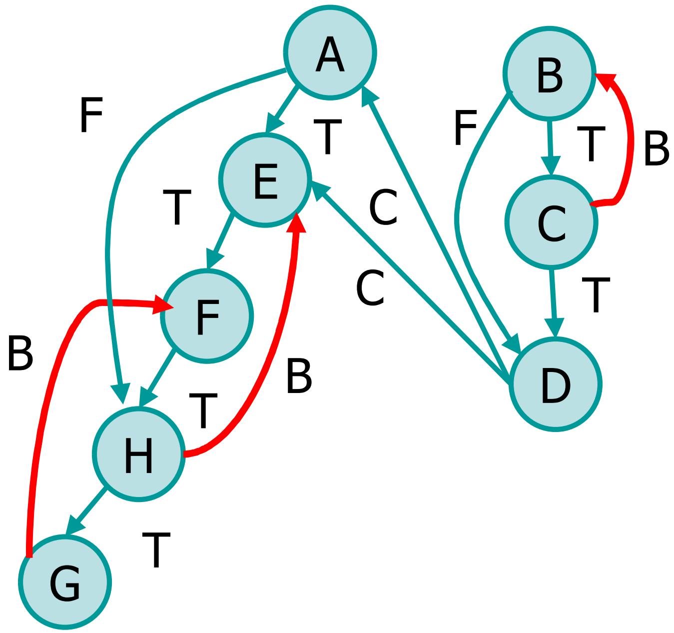
- Given a graph $G = (V, E)$, G is acyclic if and only if in a DFS there are no edges labelled backward (B)

igf there is no cycle there cannot be an infinite path

Example


G
H
A
B
F
E
D
C


5/6
4/7
1/10
11/16
T
F
G
H
A
B
B
T
B
T
C
F
B
T
F
E
D
C
T
C
T
3/8
2/9
14/15
12/13


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

Topological Sort

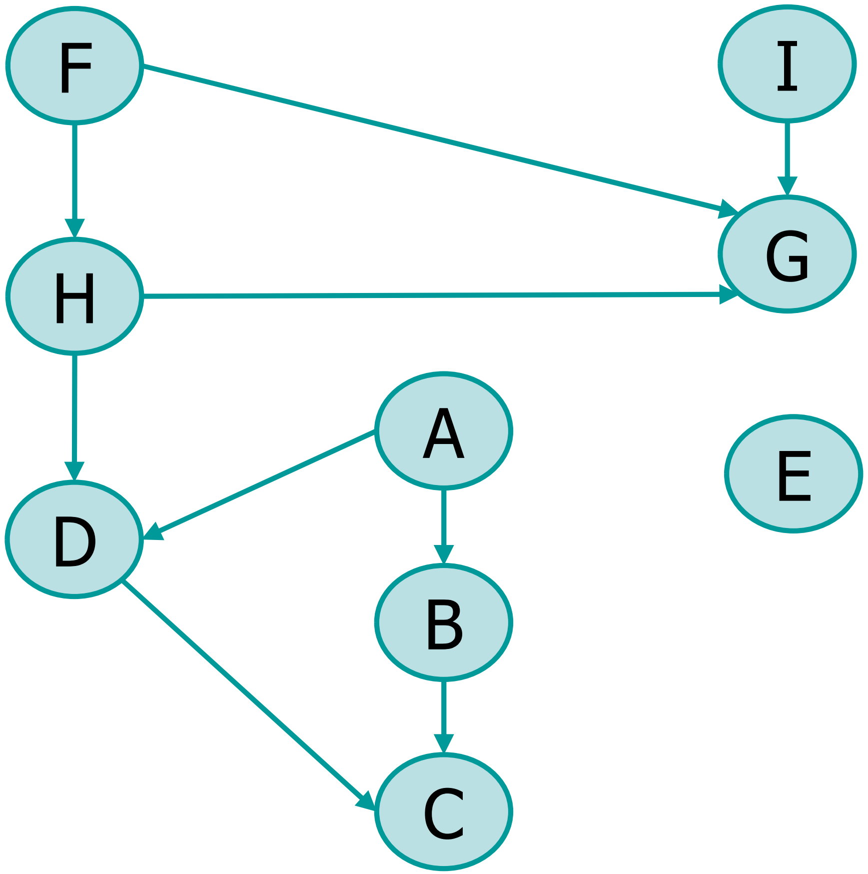
- Given a directed graph a topological sort (or topological ordering) is a linear ordering of its vertices such that
  - For every directed edge (v, u), node v comes before node u in the ordering
- Finding the topological order (reverse) means
  - Reordering the nodes according to a horizontal line, so that if the (v, u) edge exists, node v appears to the left (right) of node u and all edges go from left (right) to right (left)

Topological Sort

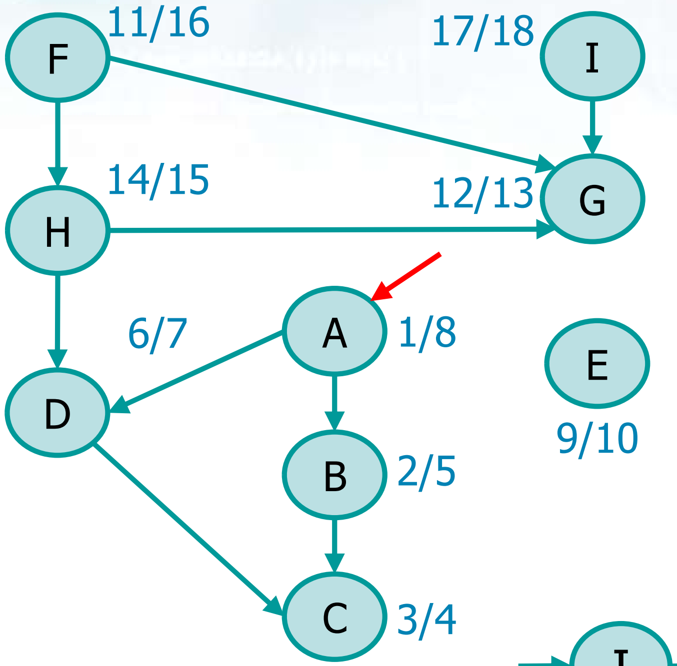
- A topological ordering is possible only if the graph has no directed cycles
  - Each DAG has **at least one** topological ordering
- Algorithm
  - Perform a DFS computing **end-processing** times
  - Order vertices with **descending** end-processing times
- Alternative algorithm
  - Perform a DFS and when assigning end-processing times insert the vertex into a **LIFO** list

Example

- Find the topological ordering and the reverse topological ordering for the following graph G



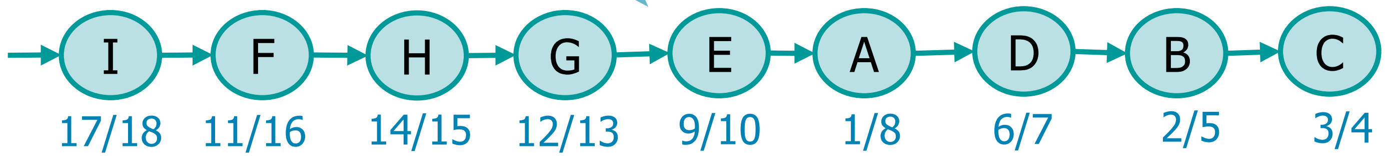
Solution



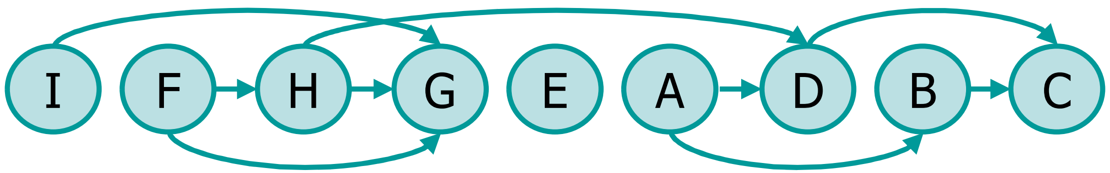
Perform a DFS

Following the alphabetic order

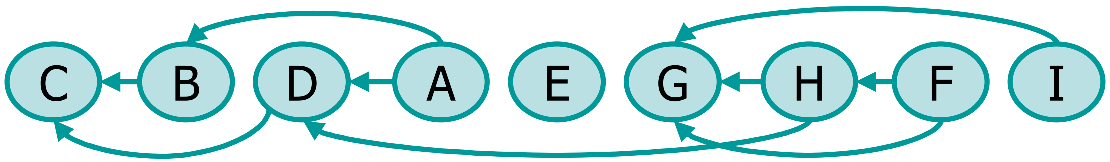
Insert nodes in a LIFO list when labeling them with the finishing time



Topological sort

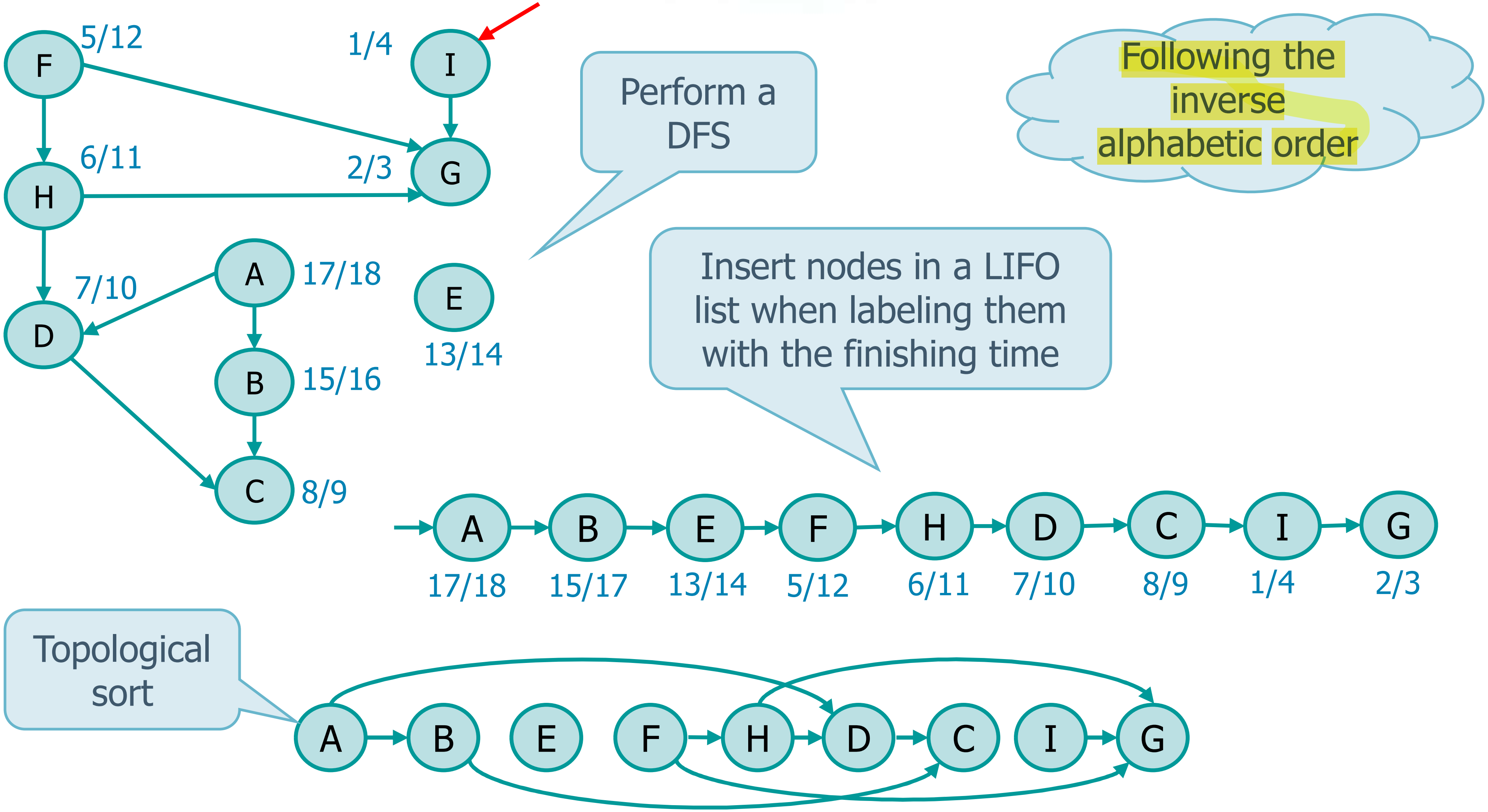


Reverse topological sort



Solution

- A graph may have many topological sortings



Implementation (with adjacency matrix)

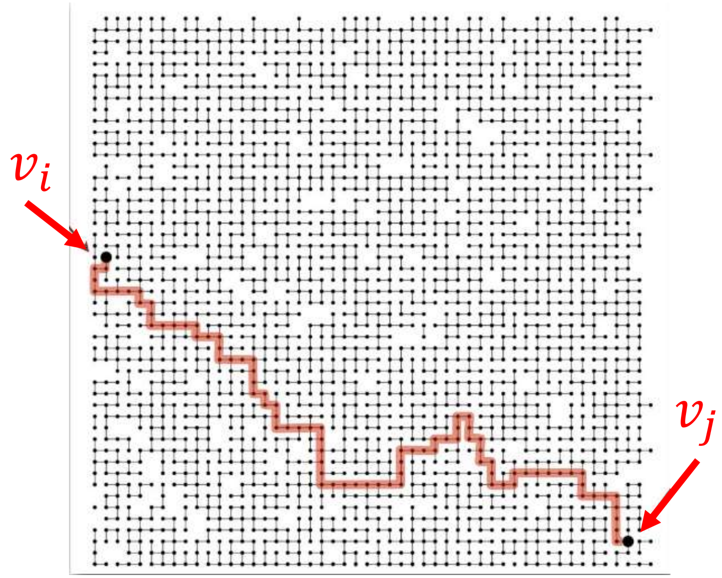
```
void graph_dag (graph_t *g){
  int i, *post, loop=0, timer=0;
  post = (int *)util_malloc(g->nv*sizeof(int));
  for (i=0; i<g->nv; i++) {
    if (g->g[i].color == WHITE) {
      timer = graph_dag_r (g, i, post, timer, &loop);
    }
  }
  if (loop != 0) {
    fprintf (stdout, "Loop detected!\n");
  } else {
    fprintf (stdout, "Topological sort (direct):");
    for (i=g->nv-1; i>=0; i--) {
      fprintf(stdout, " %d", post[i]);
    }
    fprintf (stdout, "\n");
  }
  free (post);
}
```

Implementation (with adjacency matrix)

```
int graph_dag_r(graph_t *g, int i, int *post, int t,
    int *loop) {
  int j;
  g->g[i].color = GREY;
  for (j=0; j<g->nv; j++) {
    if (g->g[i].rowAdj[j] != 0) {
      if (g->g[j].color == GREY) {
         *loop = 1;
      }
      if (g->g[j].color == WHITE) {
         t = graph_dag_r(g, j, post, t, loop);
      }
    }
  }
  g->g[i].color = BLACK;
  post[t++] = i;
  return t;
}
```

Connectivity

- In graph theory, connectivity is one of the basic concepts
  - The connectivity of a graph is an important measure of its resilience as a network
    - It is strictly related to the network flow
  - It asks for the minimum number of elements (nodes or edges) that need to be removed to separate the remaining nodes into two or more isolated graphs



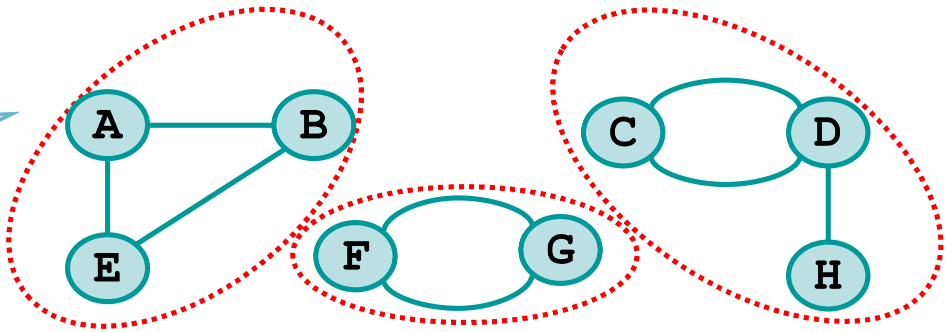
Connectivity: Undirected graphs

- An undirected graph is said to be connected iff

$$\forall v_i, v_j \in V \quad \textit{there exists a path p such that} \quad v_i \rightarrow_p v_j$$

- In an undirected graph a connected component is the maximal connected subgraph
  - There is no superset including it which is connected
- An undirected graph is said to be connected
  - If it includes only one connected component

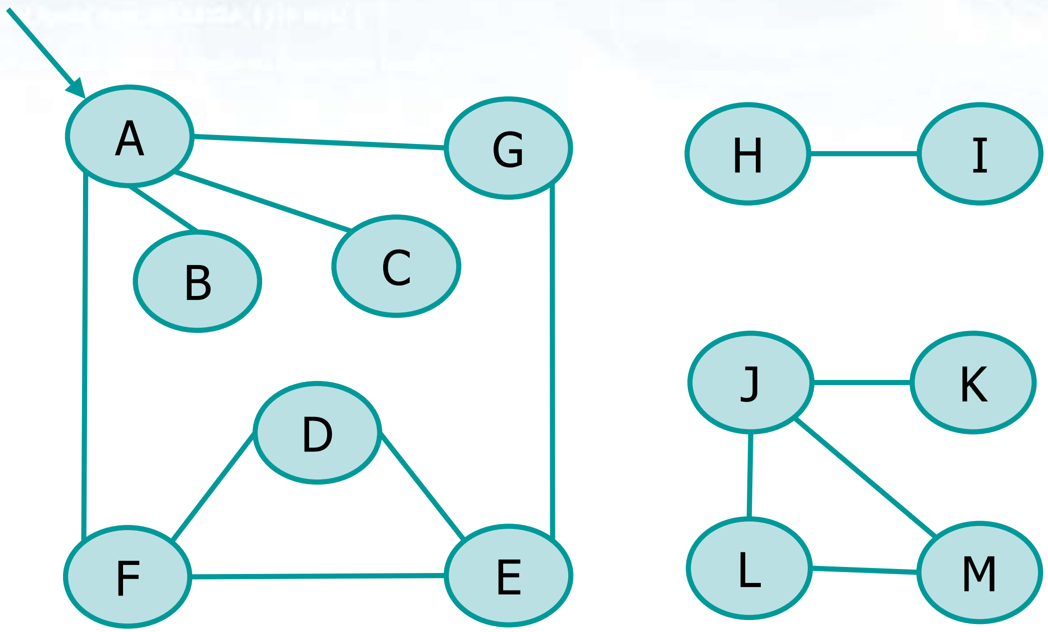
The graph includes 3 connected components



Connectivity: Undirected graphs

- In an undirected graph
  - Each tree of the DFS forest is a connected component
  - Connected component can be represented as an array that stores an integer identifying each connected component
    - Node identifiers serve as indexes of the array

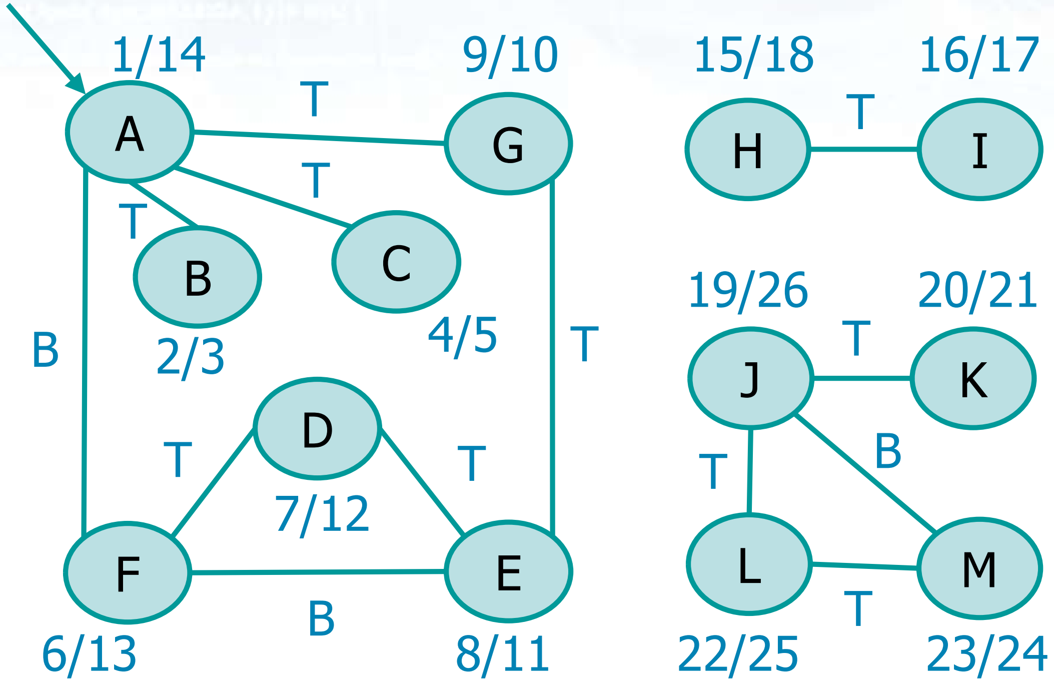
Example



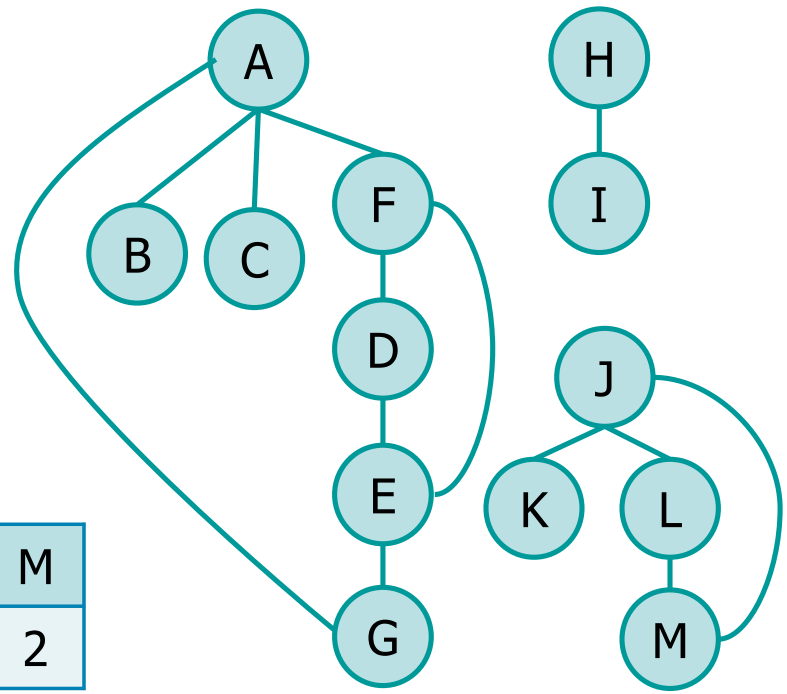
A	B	C	D	E	F	G	H	I	J	K	L	M

Connected Component Ids

Solution



for undirected graphs, much easier than for directed



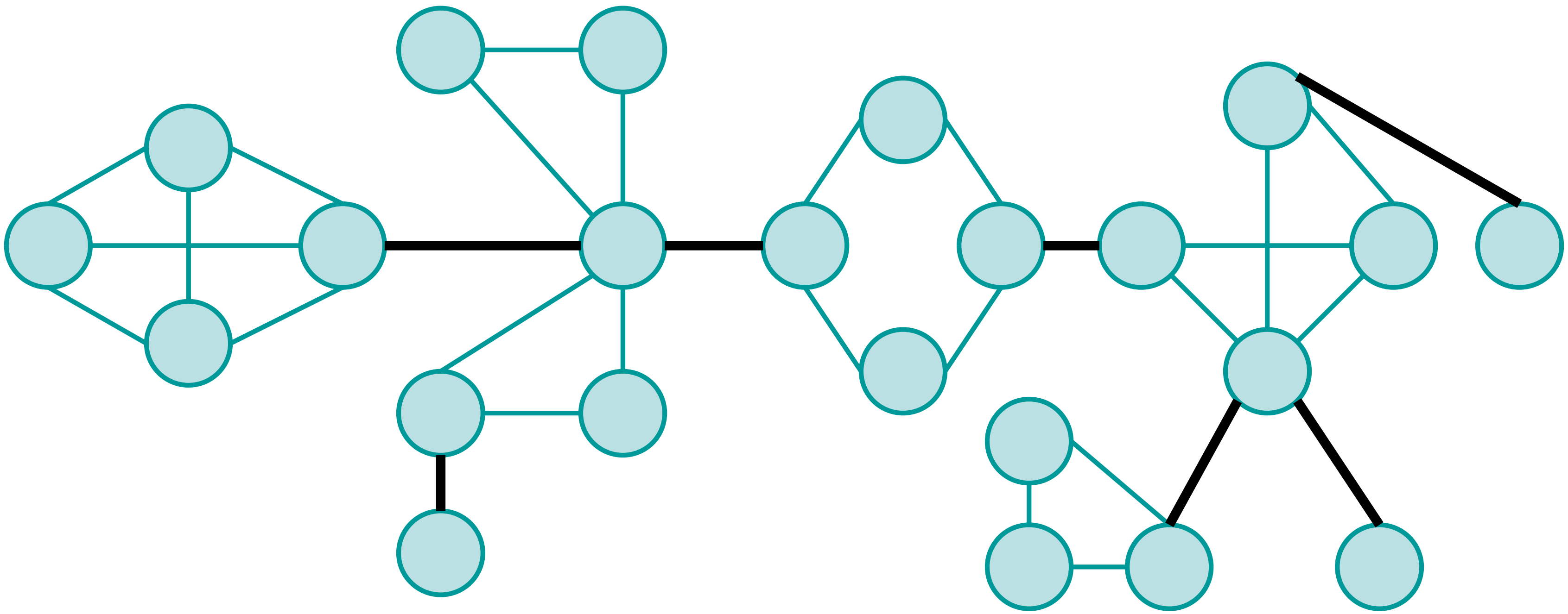
A	B	C	D	E	F	G	H	I	J	K	L	M
0	0	0	0	0	0	0	1	1	2	2	2	2

Connectivity: Bridges

- Given an undirected graph it is important to understand how difficult it is to make it disconnected removing edges
- A bridge (or isthmus or cut-edge) is an edge whose removal increases the number of connected components
  - In **connected** graphs removing a bridge disconnects the graph
  - A graph is said to be bridgeless if it contains no bridges

Example

Bridges

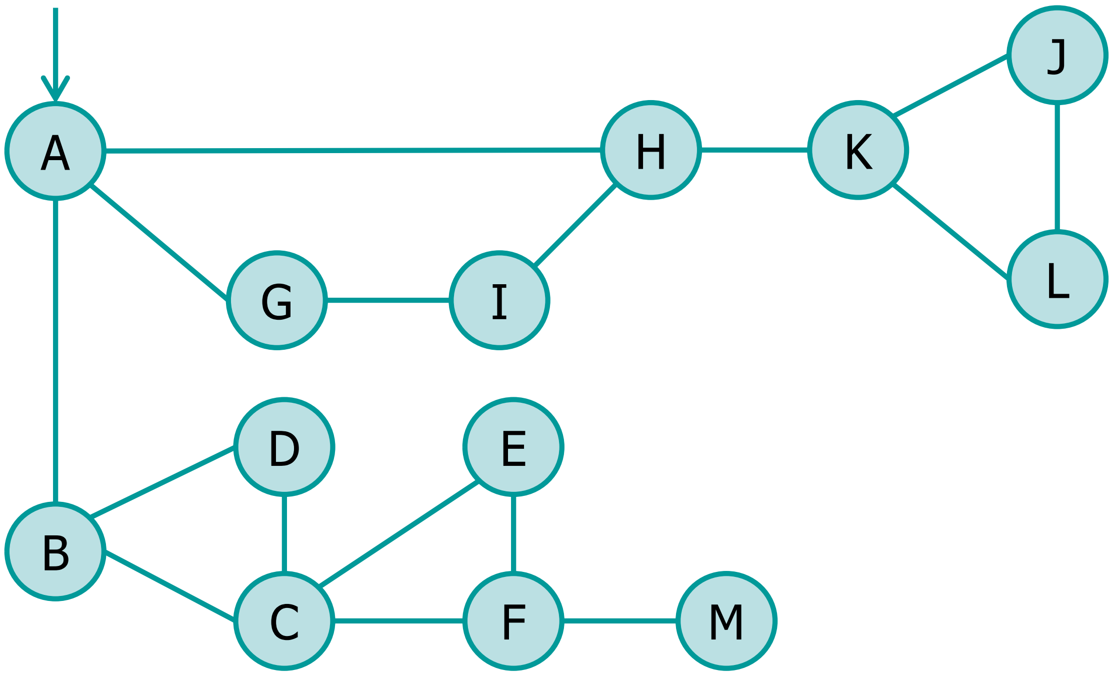


Connectivity: Bridges

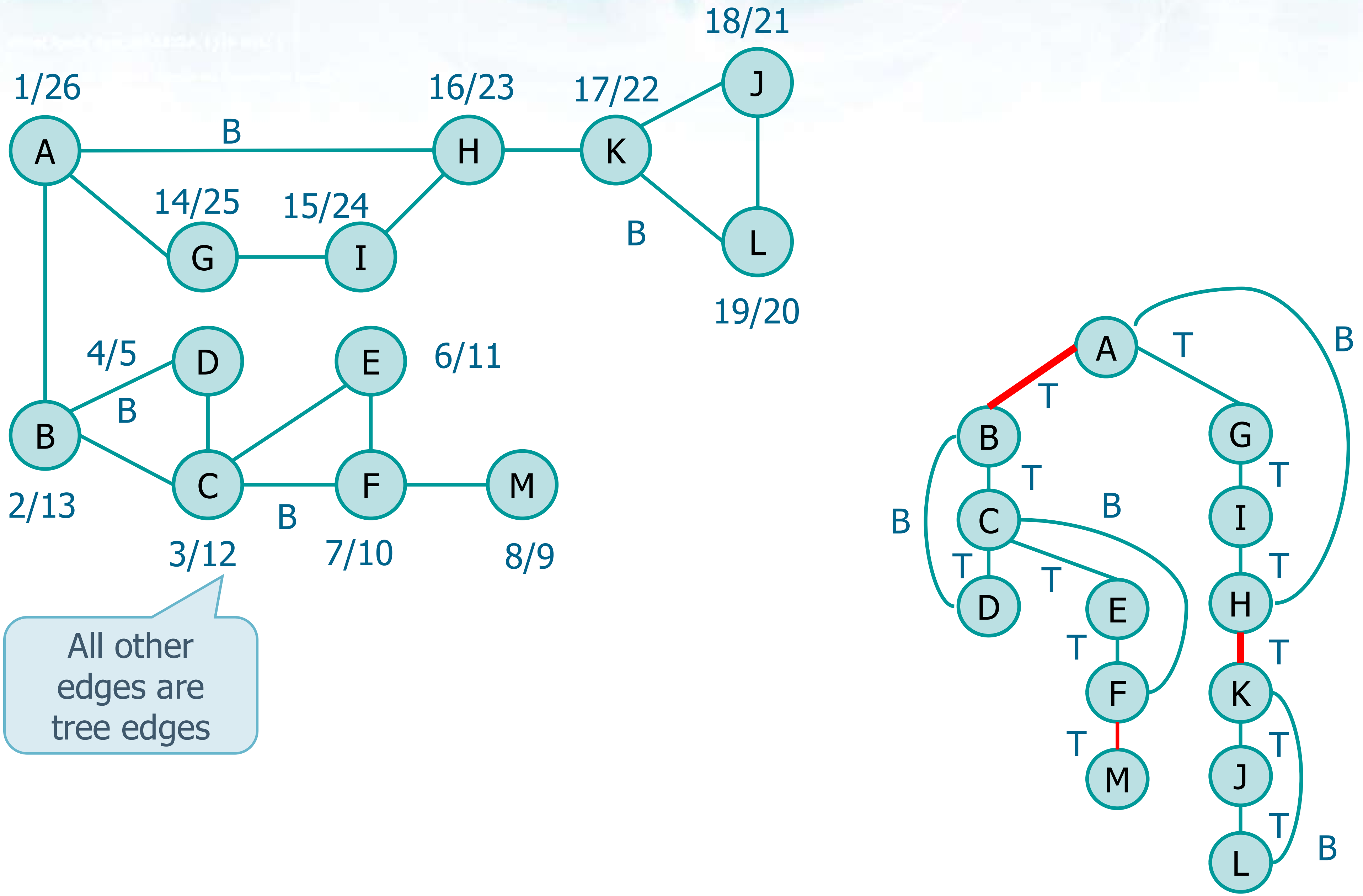
- We can find bridges visiting G in DFS
- An undirected graph includes only Tree (T) and Backward (B) edges
- An edge (v, u)
  - Labelled Back (B) cannot be a bridge
    - Nodes v and u are also connected by a path in the DFS tree
  - Labelled Tree (T) is a bridge if and only if there is **no** edge labelled Back (B) connecting a descendant of u to an ancestor of v in the DFS tree

Example

- Given the following graph G, find all bridges



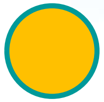
Solution

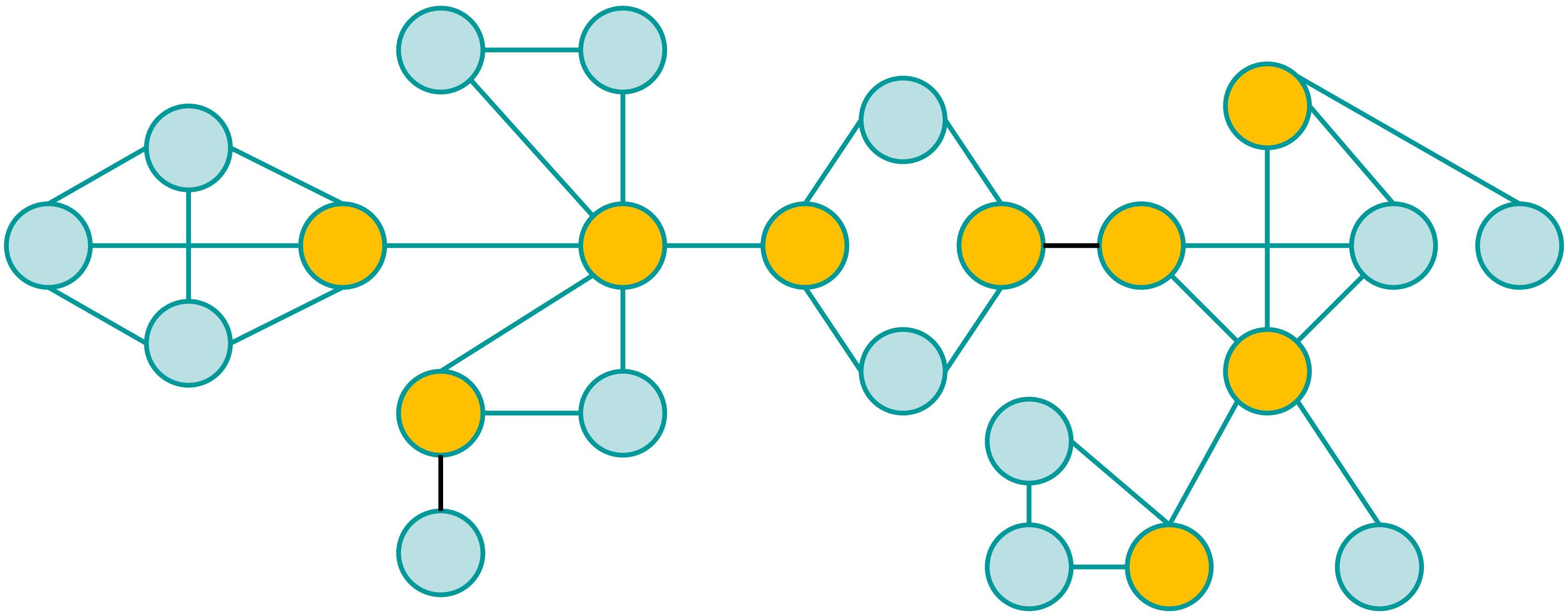

1/26
16/23
17/22
18/21
14/25
15/24
19/20
4/5
6/11
2/13
3/12
7/10
8/9
A
B
C
D
E
F
G
H
I
J
K
L
M
B
T
All other edges are tree edges

Connectivity: Articulation points

- Given an undirected graph it is important to understand how difficult it is to make it disconnected removing nodes
- An articulation point (or cut-vertex or separating-vertex) is a vertex whose removal increases the number of connected components
  - In **connected** graphs removing an articulation point disconnects the graph
  - Removing the vertex entails the removal of insisting (incoming and outgoing) edges as well

Example

Articulation points 

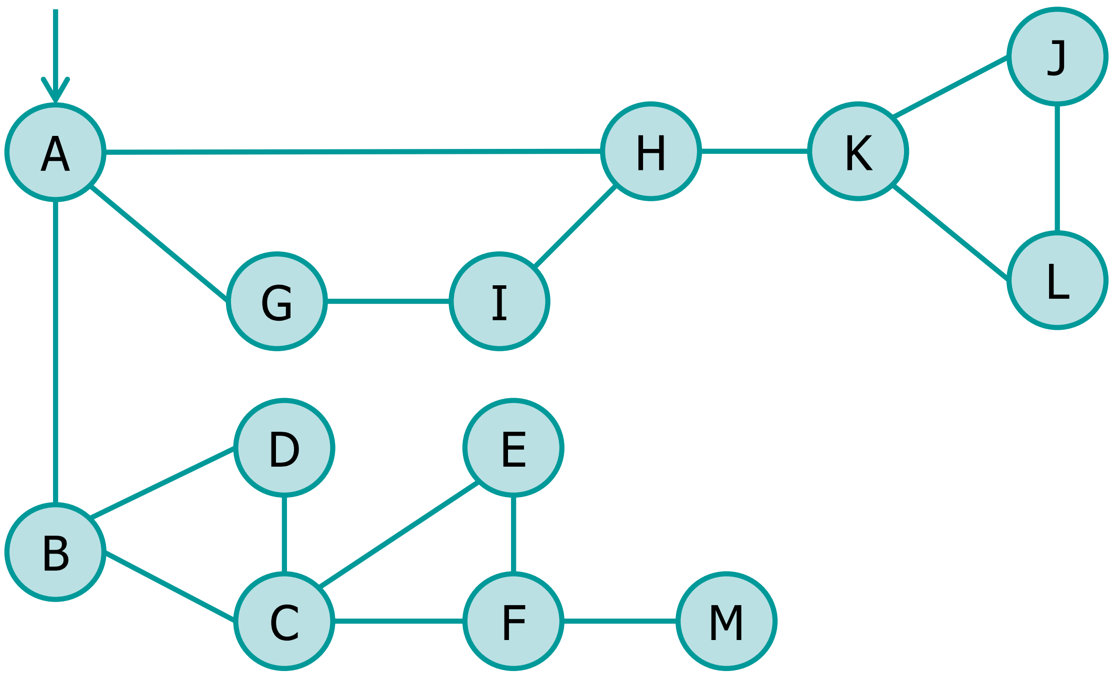


Connectivity: Articulation points

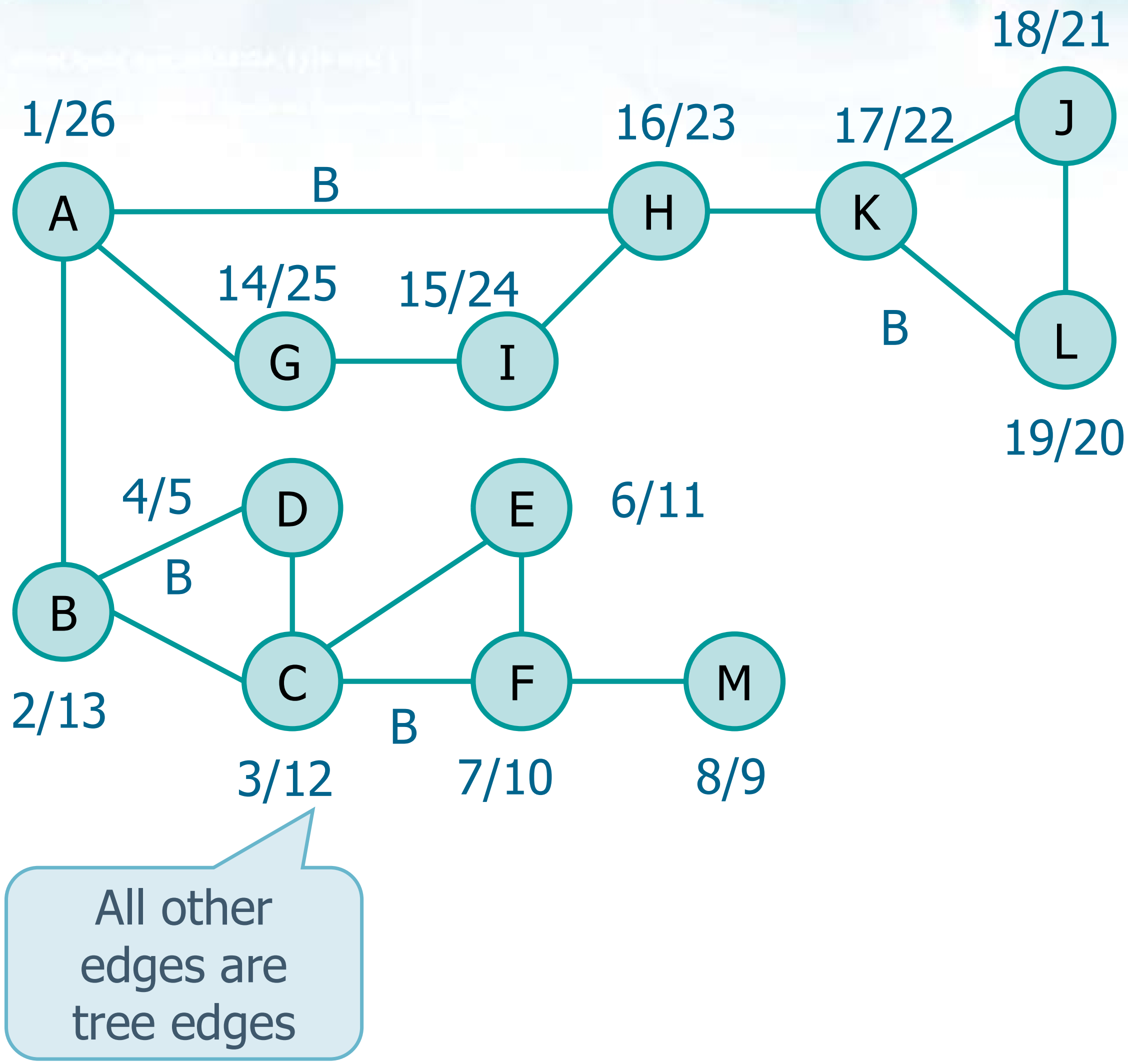
- We can find articulation points visiting G in DFS
- Given the DFS tree G_P
  - The root of G_P is an articulation point if and only if it has at least two children
  - Leaves cannot be articulation points
  - Any internal node v is an articulation point of G if and only if v has at least one child u such that there is no edge labelled B from u or from one of its descendants to a proper ancestor of v

Example

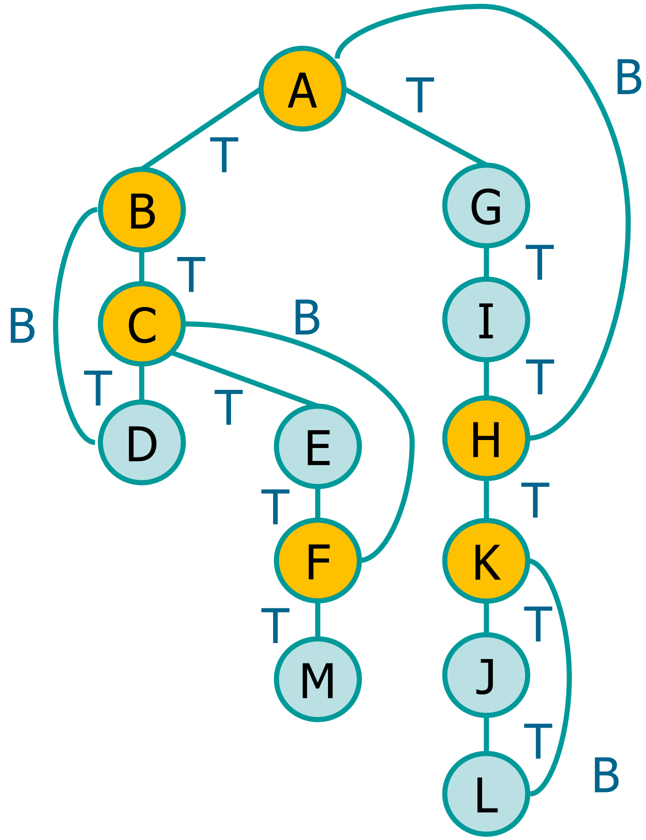
- Given the following graph G, find all articulation points
  - Has the way we perform the DFS some influence on the result?



Solution

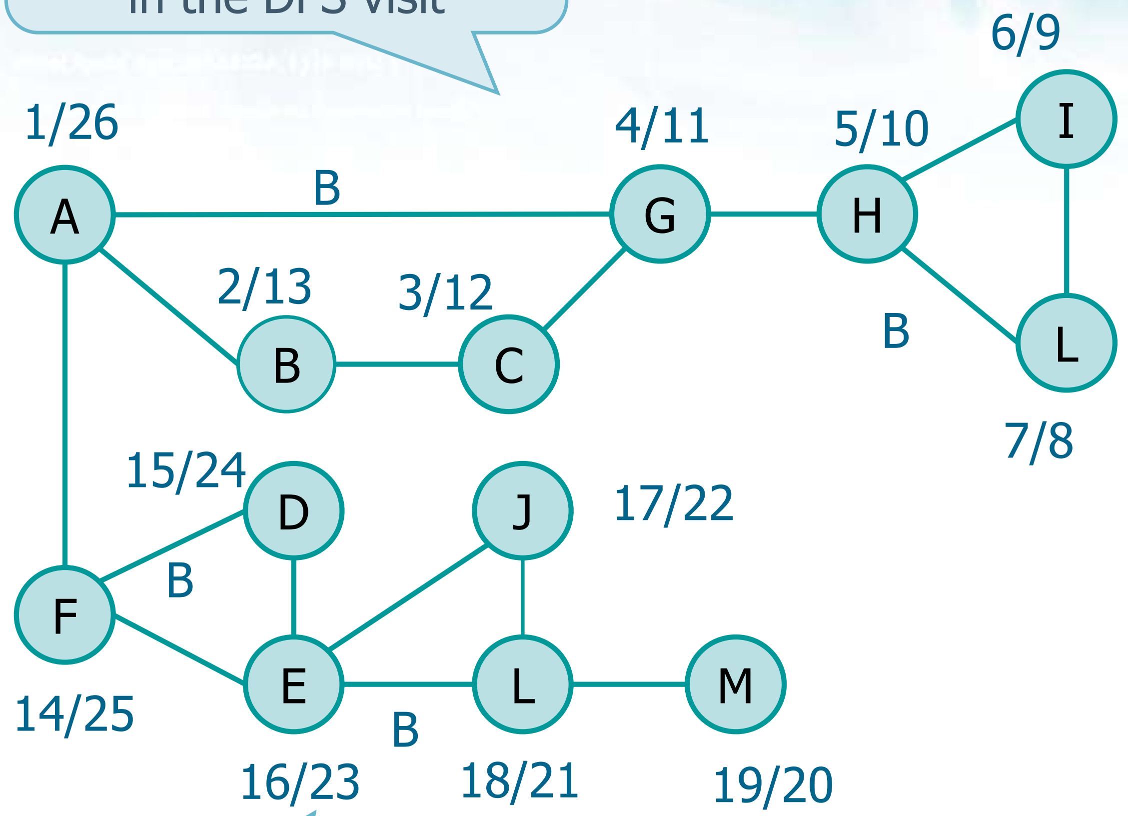


Following the alphabetic order



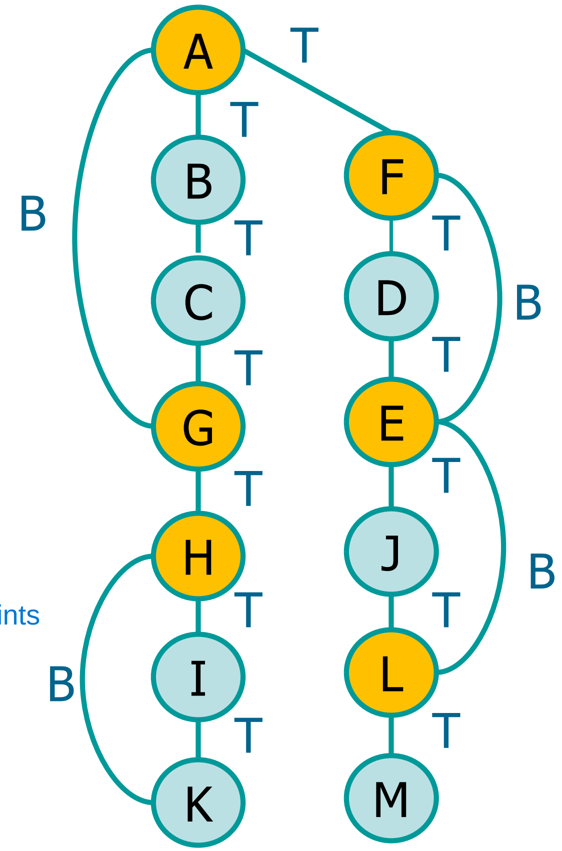
Example

Same example different ids and order in the DFS visit



All other edges are tree edges

Following the alphabetic order with a different labeling



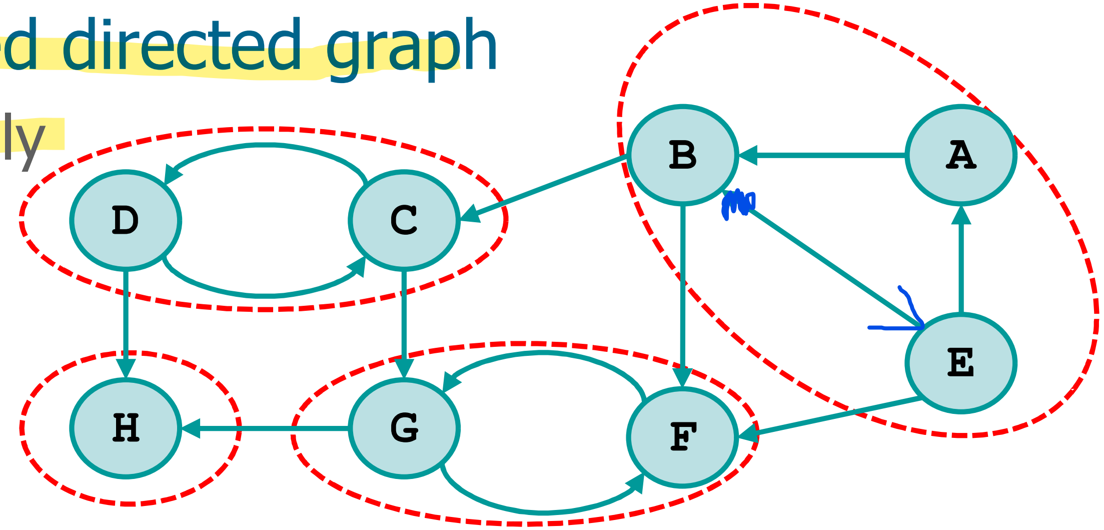
The way we visit the graph
does not affect which are the articulation points

Connectivity: Directed graphs

- A directed graph is said to be strongly connected iff

$$\forall v_i, v_j \in V \quad \textit{there exists two paths } p \textit{ and } p' \textit{ such that}$$
$$v_i \rightarrow_p v_j \quad \textit{and} \quad v_j \rightarrow_{p'} v_i$$

- In a directed graph
  - Strongly connected component
    - Maximal strongly connected subgraph
  - Strongly connected directed graph
    - Only one strongly connected component



Connectivity: Directed graphs

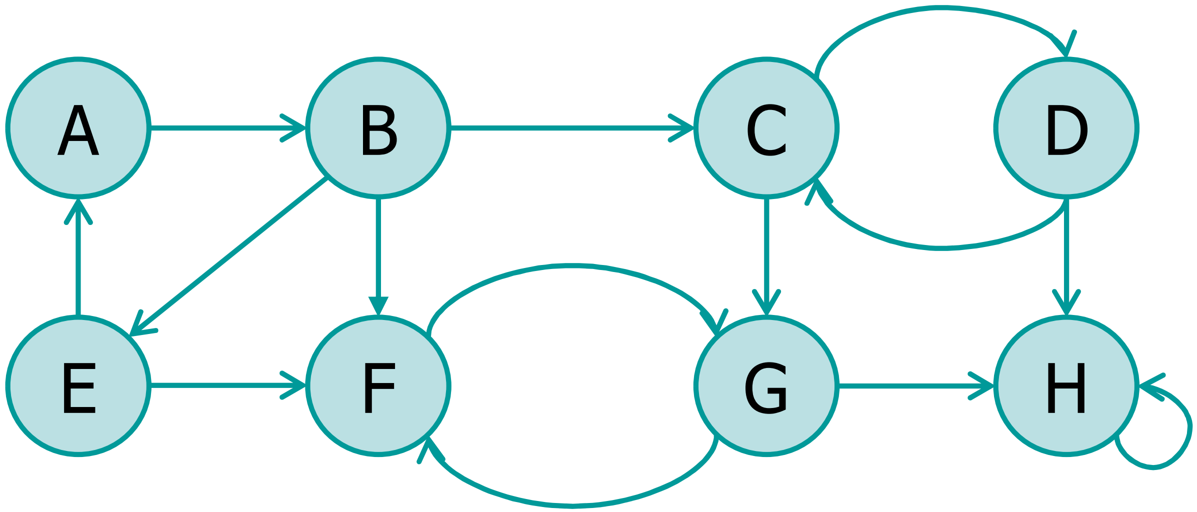
- Strongly Connected Component (or SCC) can be found using the Kosaraju's algorithm (1978)
- It makes use of the fact that the transpose gaph has exactly the same strongly SCCs

Connectivity: Directed graphs

- The Kosaraju's algorithm is based on two DFSs done in sequence
  - Given the graph G
  - Reverse the graph finding G^T
  - Execute a DFS on G^T and compute discovery and end-processing times for all nodes
  - Execute a DFS on G starting from nodes having a **decreasing** end-processing times
  - The trees of the latter DFS are the strongly connected components of G

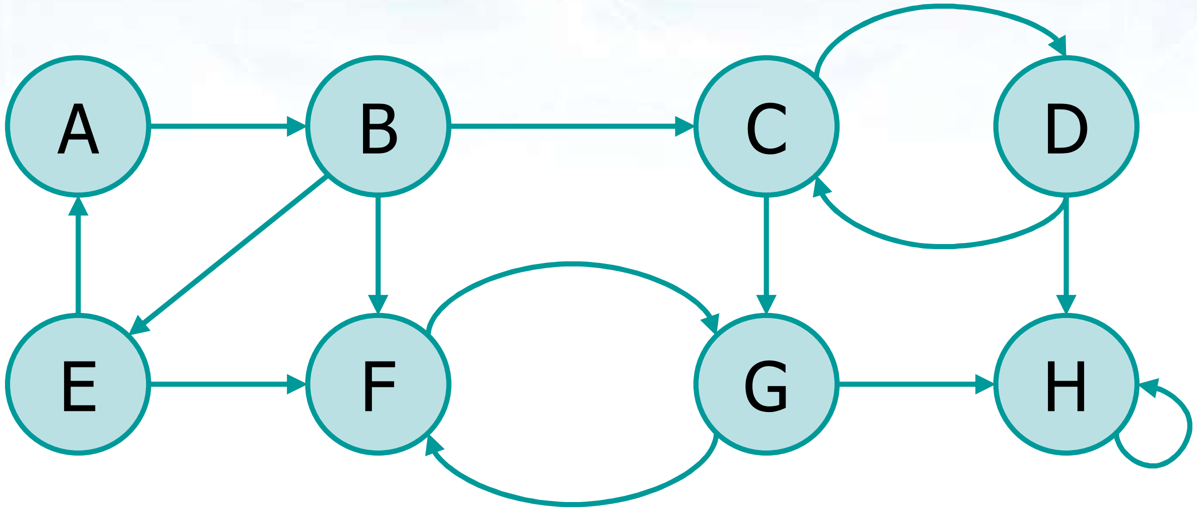
Example

- Given the following graph G, find its SCCs using the Kosaraju's algorithm



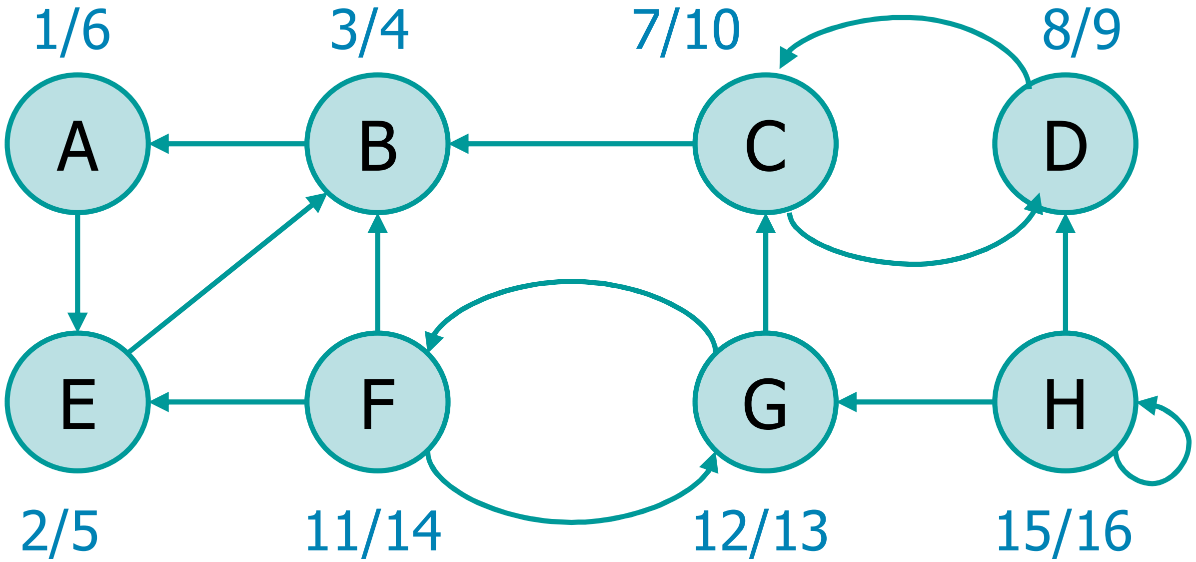
Solution

G



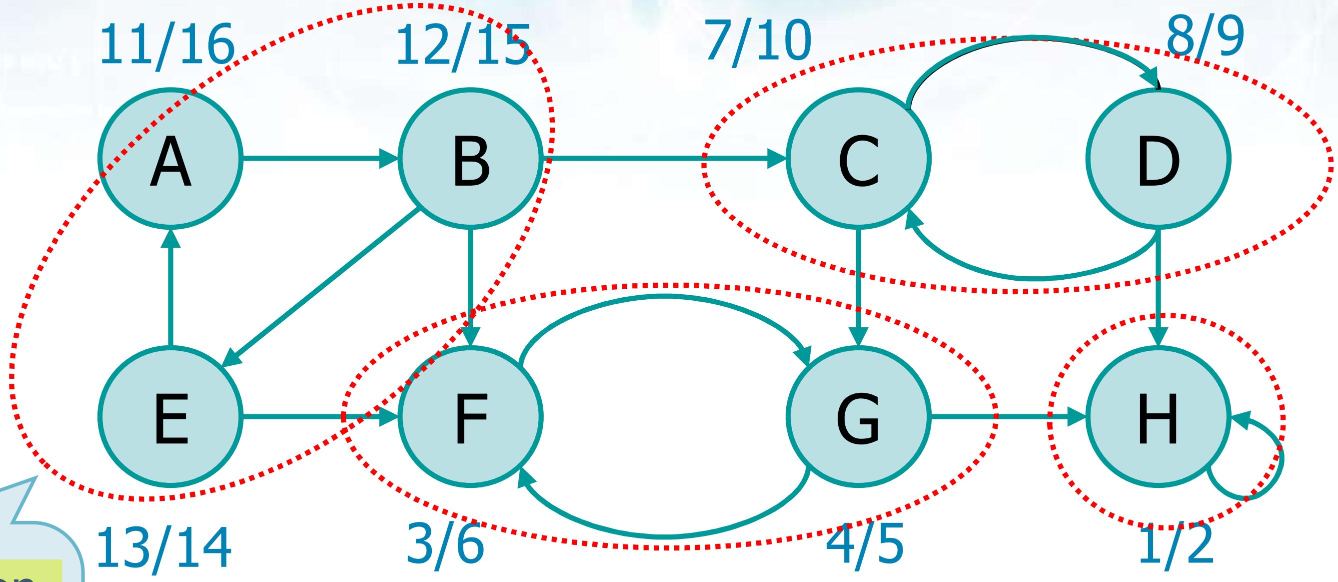
Reverse the graph and perform DFS on G^T

G^T



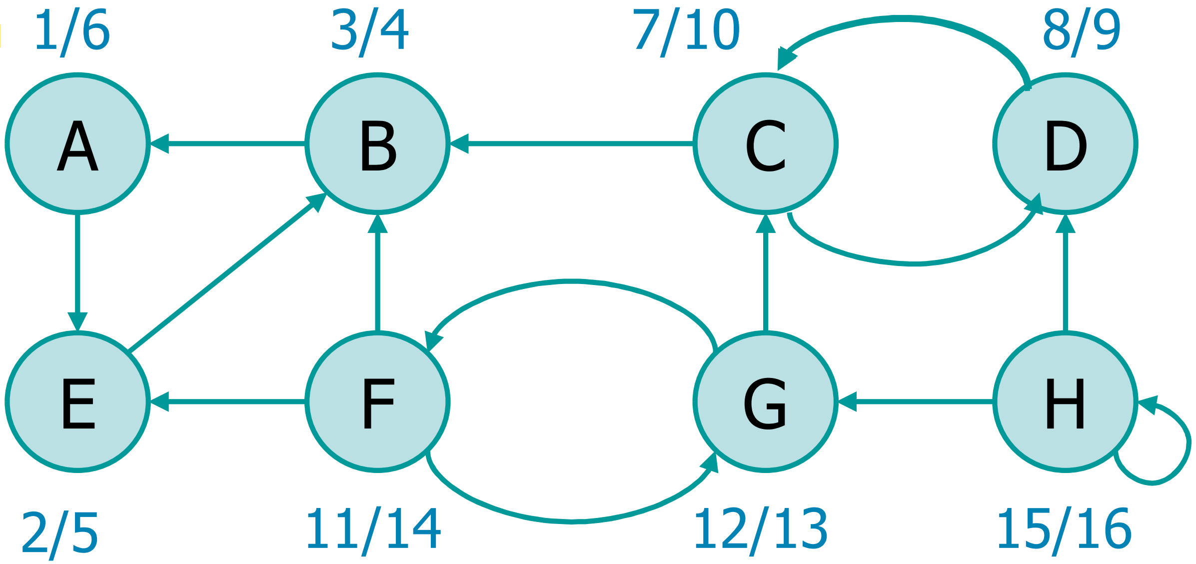
Solution

G



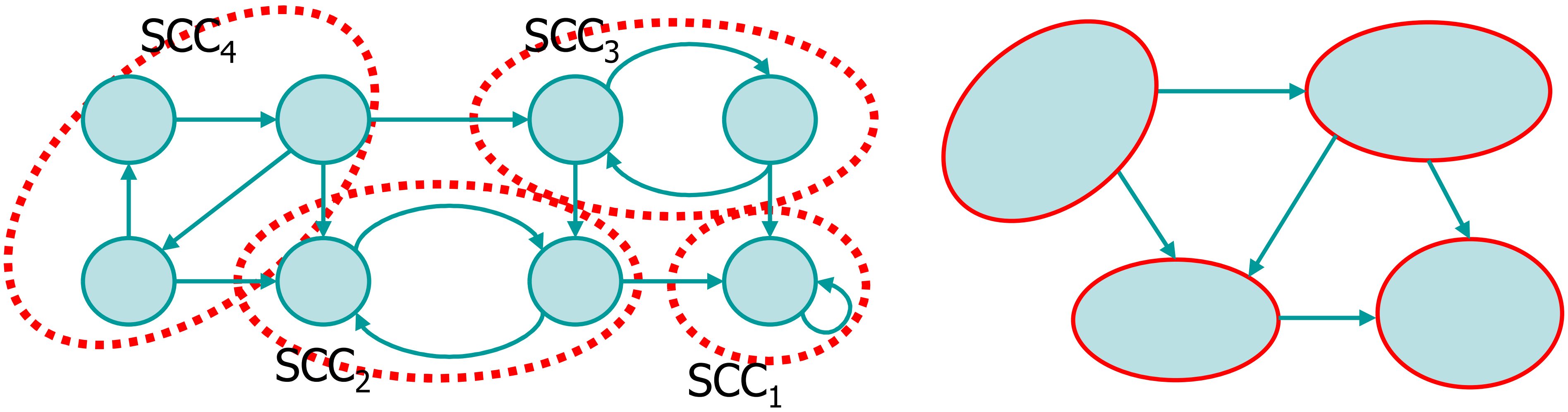
Perform DFS on G by decreasing end-processing times

G^T



Connectivity: Directed graphs

- SCCs are equivalence classes with respect to the property of mutual reachability
  - Given G and its SCC, we can "extract" a reduced graph G' considering 1 node as representing each equivalence class
  - The reduced graph G' is a DAG



Implementation (with adjacency matrix)

Client (code extract)

```
g = graph_load (argv[1]);

sccs = graph_scc (g);

fprintf (stdout, "Number of SCCs: %d\n", sccs);
for (j=0; j<sccs; j++) {
  fprintf (stdout, "SCC%d:", j);
  for (i=0; i<g->nv; i++) {
    if (g->g[i].scc == j) {
      fprintf (stdout, " %d", i);
    }
  }
  fprintf (stdout, "\n");
}

graph_dispose (g);
```

Implementation (with adjacency matrix)

```
int graph_scc (graph_t *g) {
  graph_t *h;
  int i, id=0, timer=0;
  int *post, *tmp;

  h = graph_transpose (g);
  post = (int *) util_malloc (g->nv*sizeof(int));
  for (i=0; i<g->nv; i++) {
    if (h->g[i].color == WHITE) {
      timer = graph_scc_r (h, i, post, id, timer);
    }
  }
  graph_dispose (h);
```

Implementation (with adjacency matrix)

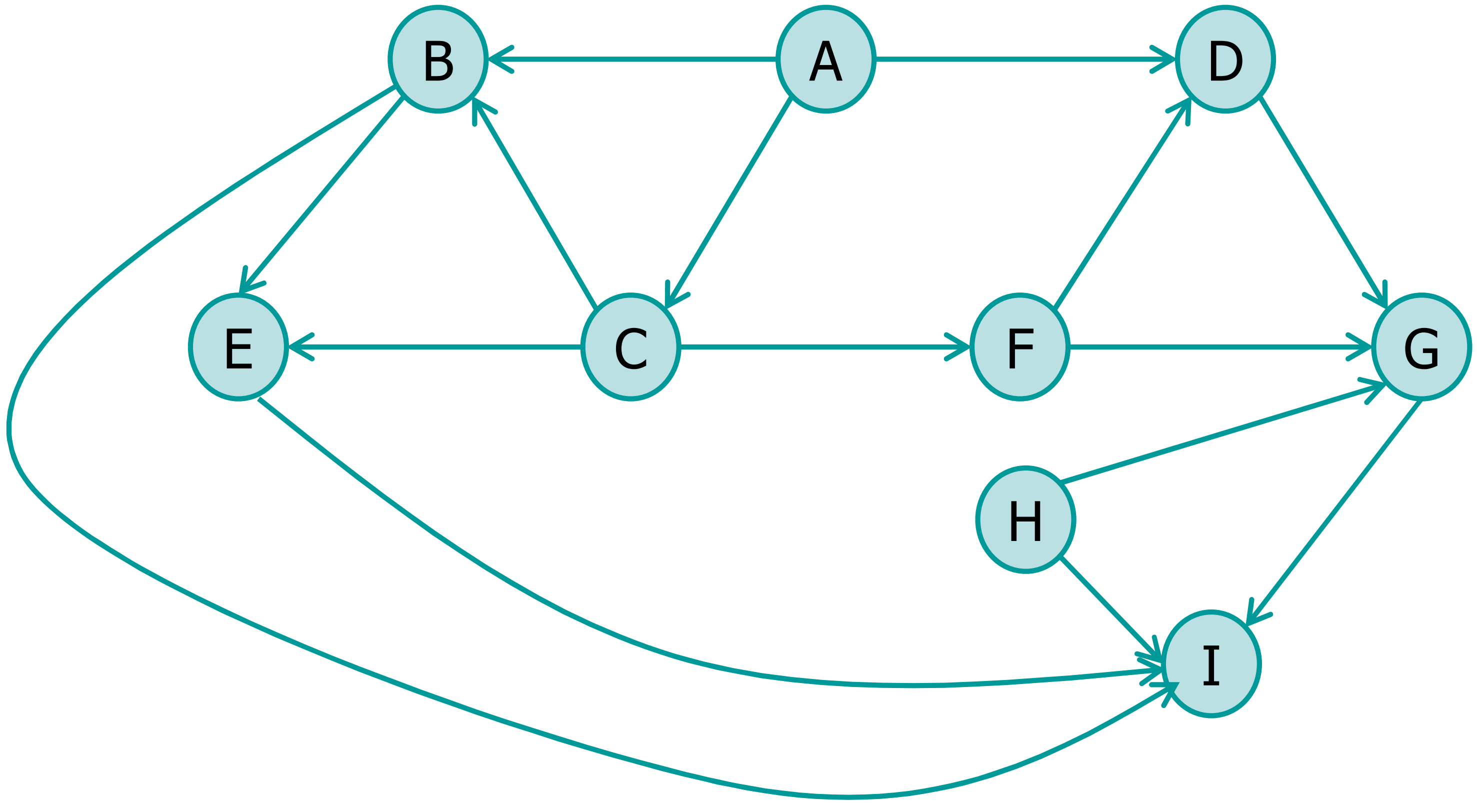
```
  id = timer = 0;
  tmp = (int *) util_malloc (g->nv * sizeof(int));
  for (i=g->nv-1; i>=0; i--) {
    if (g->g[post[i]].color == WHITE) {
      timer=graph_scc_r(g, post[i], tmp, id, timer);
      id++;
    }
  }

  free (post);
  free (tmp);

  return id;
}
```

Implementation (with adjacency matrix)

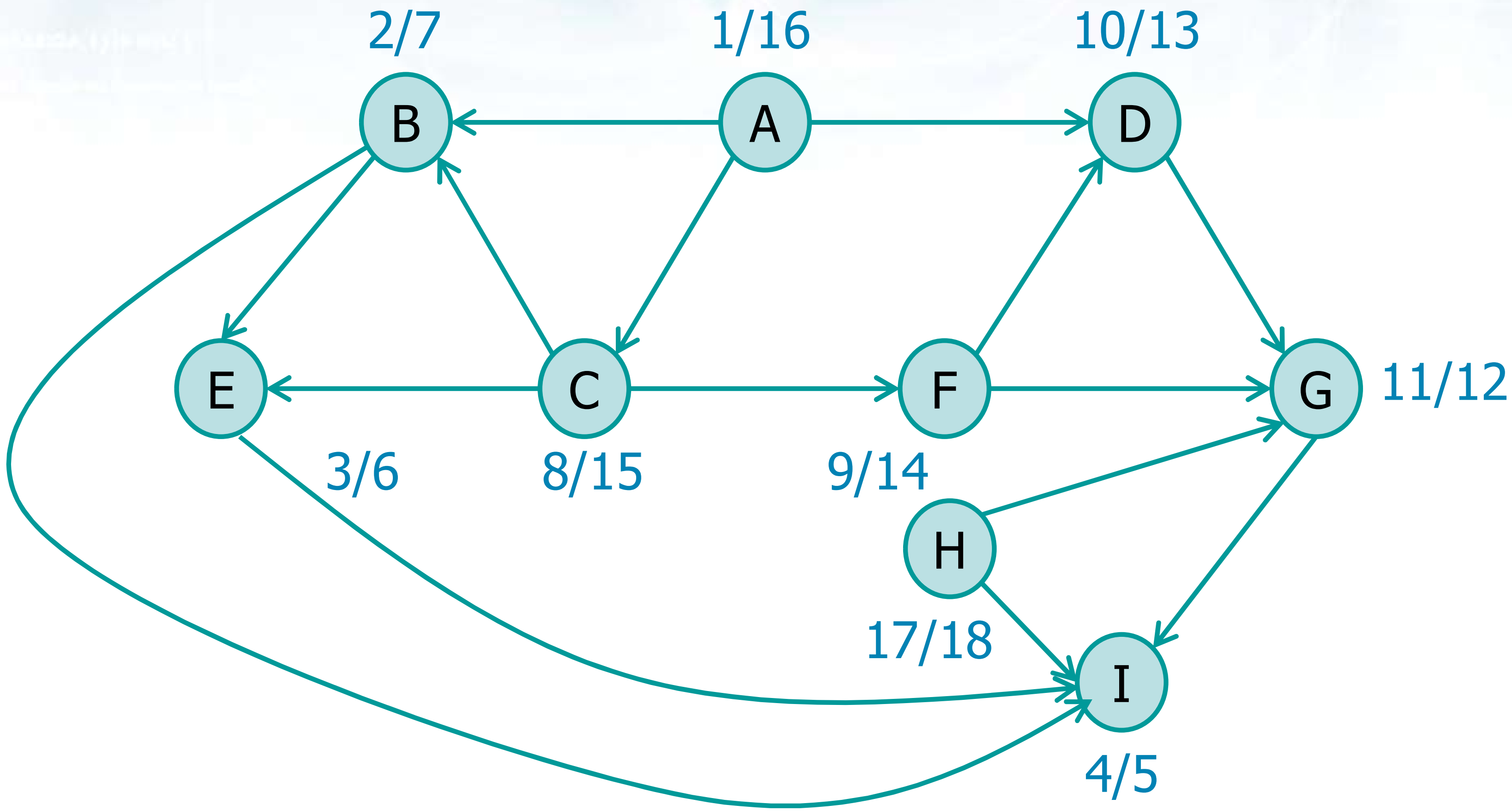
```
int graph_scc_r(
  graph_t *g, int i, int *post, int id, int t
) {
  int j;
  g->g[i].color = GREY;
  g->g[i].scc = id;
  for (j=0; j<g->nv; j++) {
    if (g->g[i].rowAdj[j]!=0 &&
        g->g[j].color==WHITE) {
      t = graph_scc_r (g, j, post, id, t);
    }
  }
  g->g[i].color = BLACK;
  post[t++] = i;

  return t;
}
```

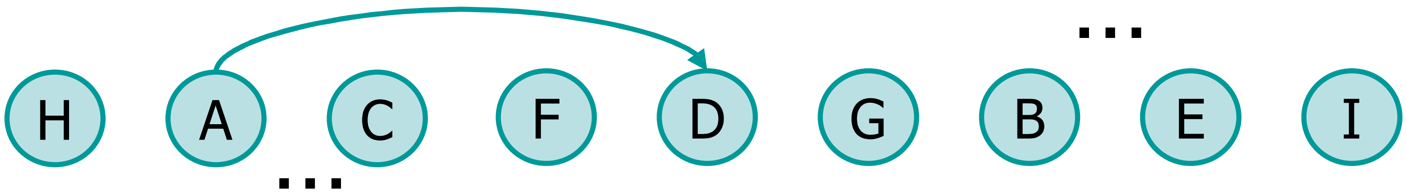
Exercise

- Given the following DAG G, find a topological order of all vertices



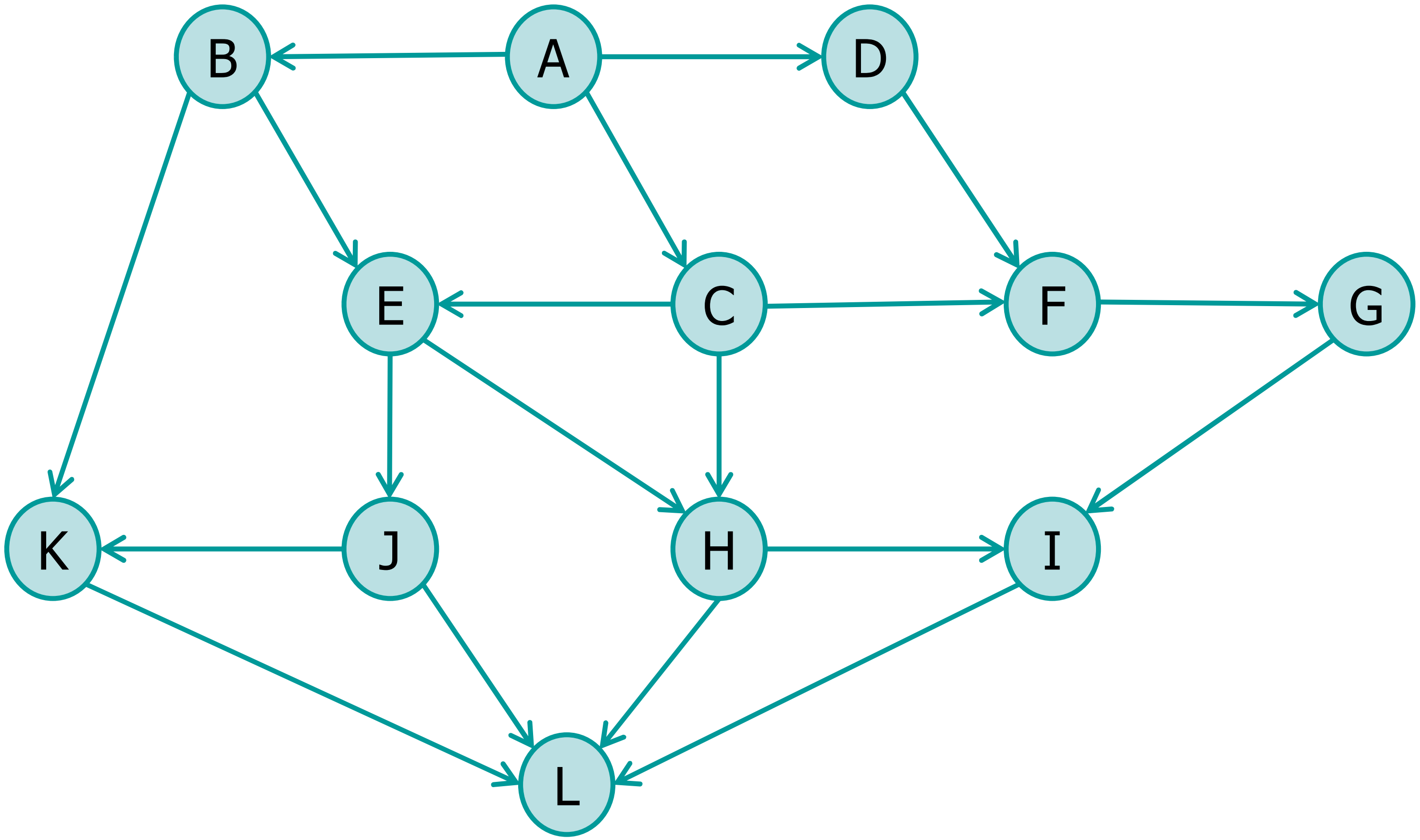
Solution



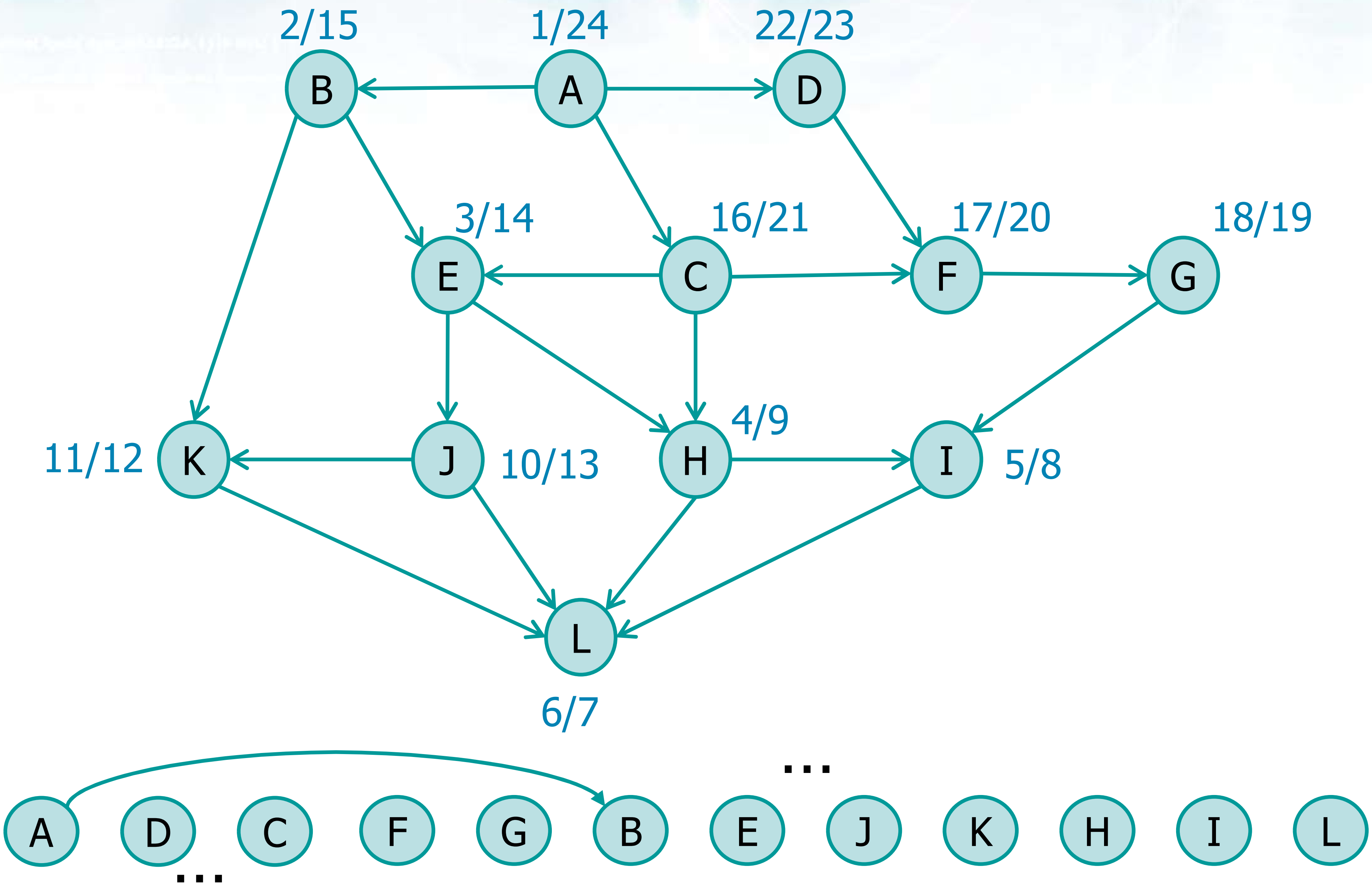


Exercise

- Given the following DAG G, find a topological order of all vertices



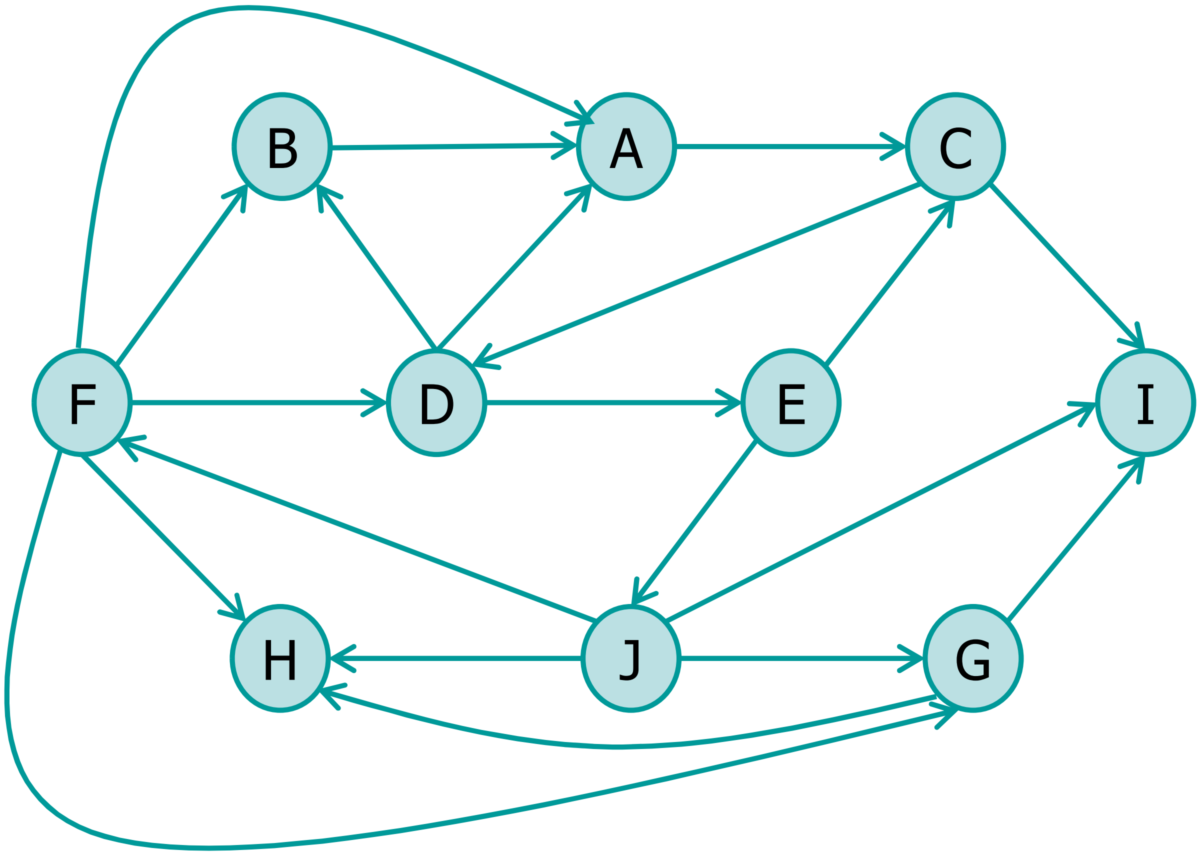
Solution



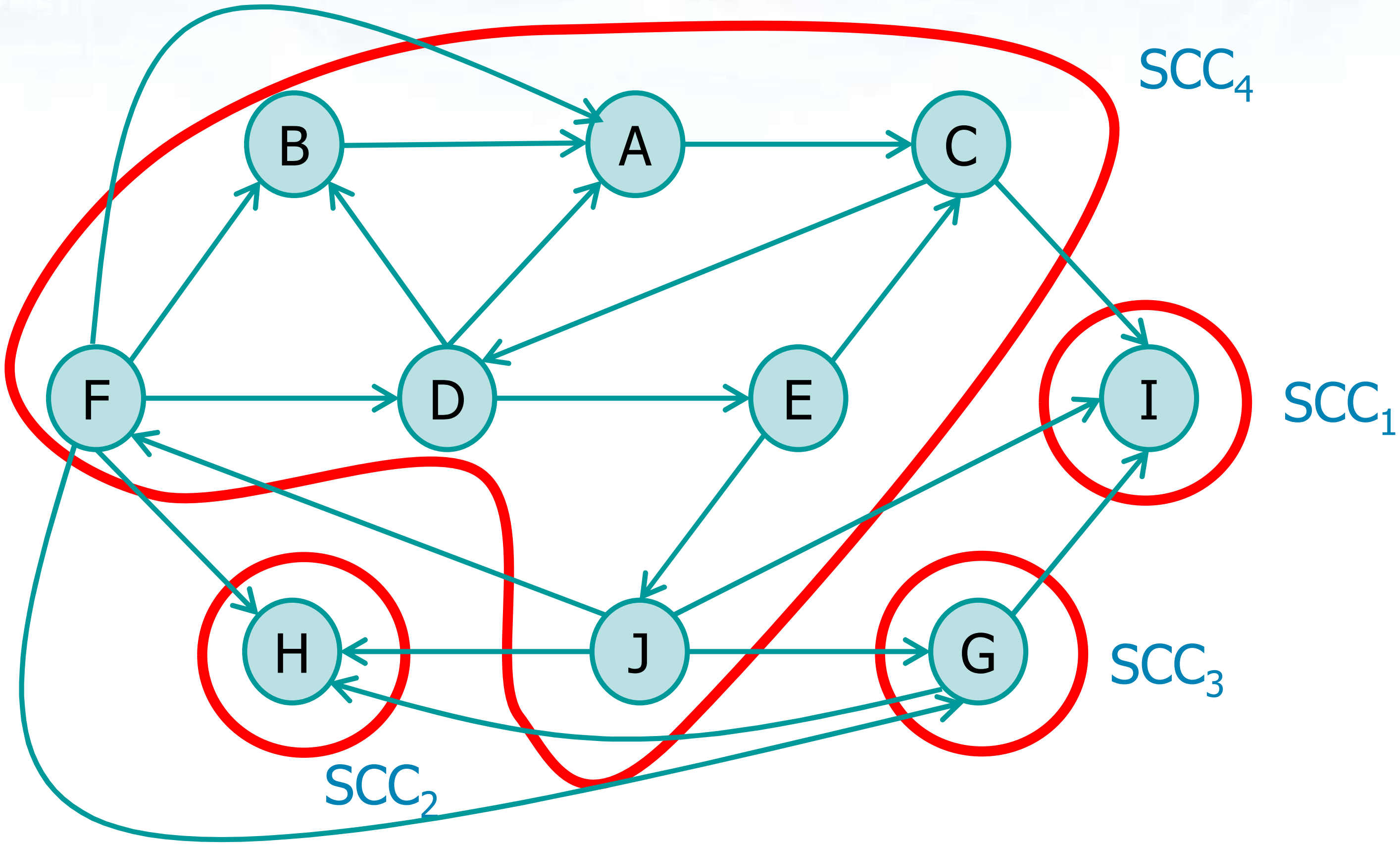
2/15
1/24
22/23
B
A
D
3/14
16/21
17/20
18/19
E
C
F
G
4/9
11/12
K
J
10/13
H
I
5/8
L
6/7
...
A
D
C
F
G
B
E
J
K
H
I
L
...

Exercise

- Given the following DAG G, find its SCCs



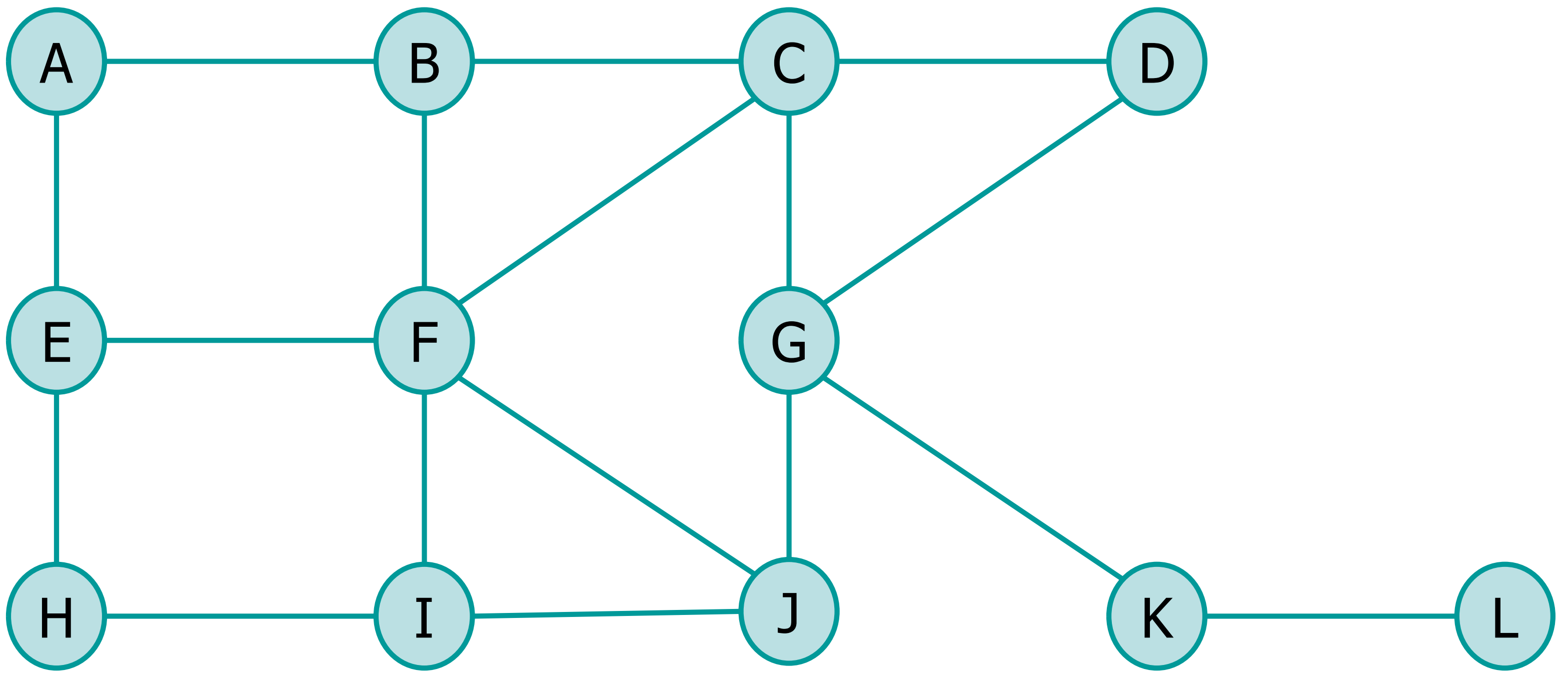
Solution



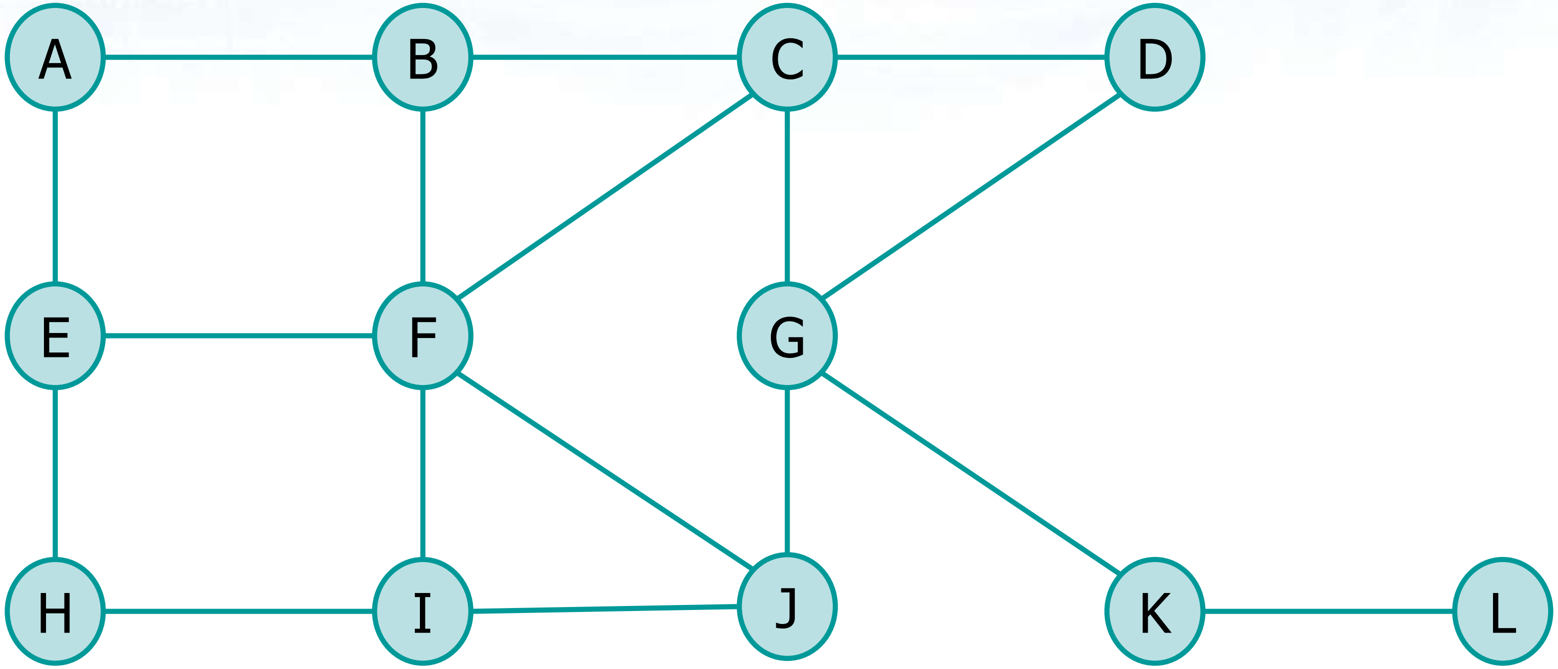
$$SCCs: \{I\}, \{H\}, \{G\}, \{A, B, C, D, E, F, J\}$$

Exercise

- Given the following graph G, find its bridges and articulation points



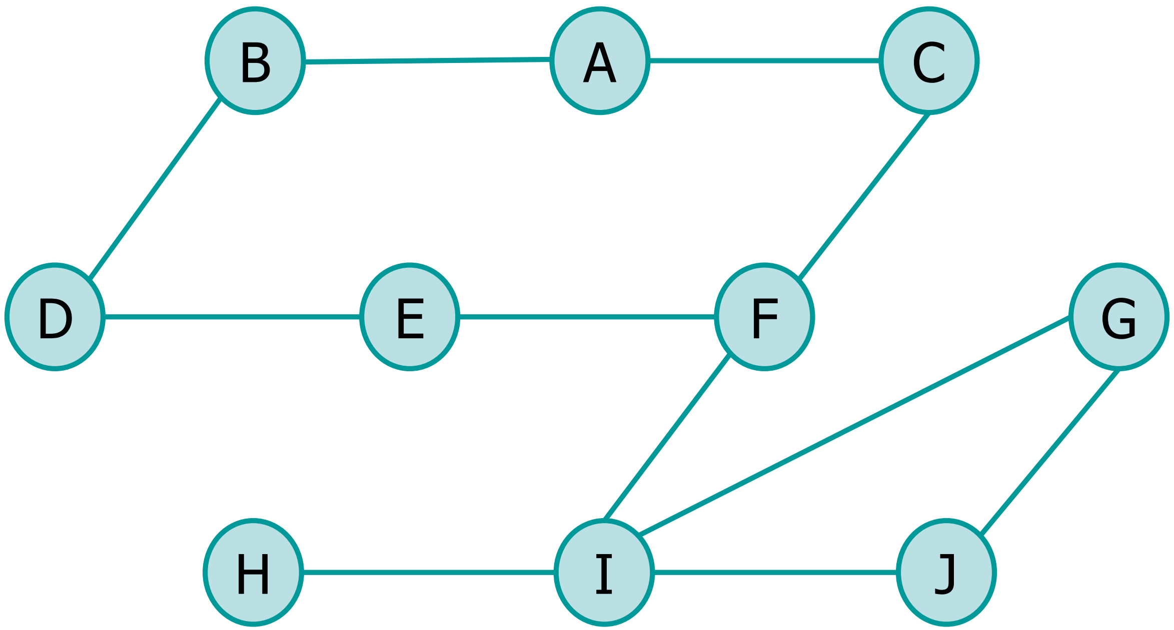
Solution



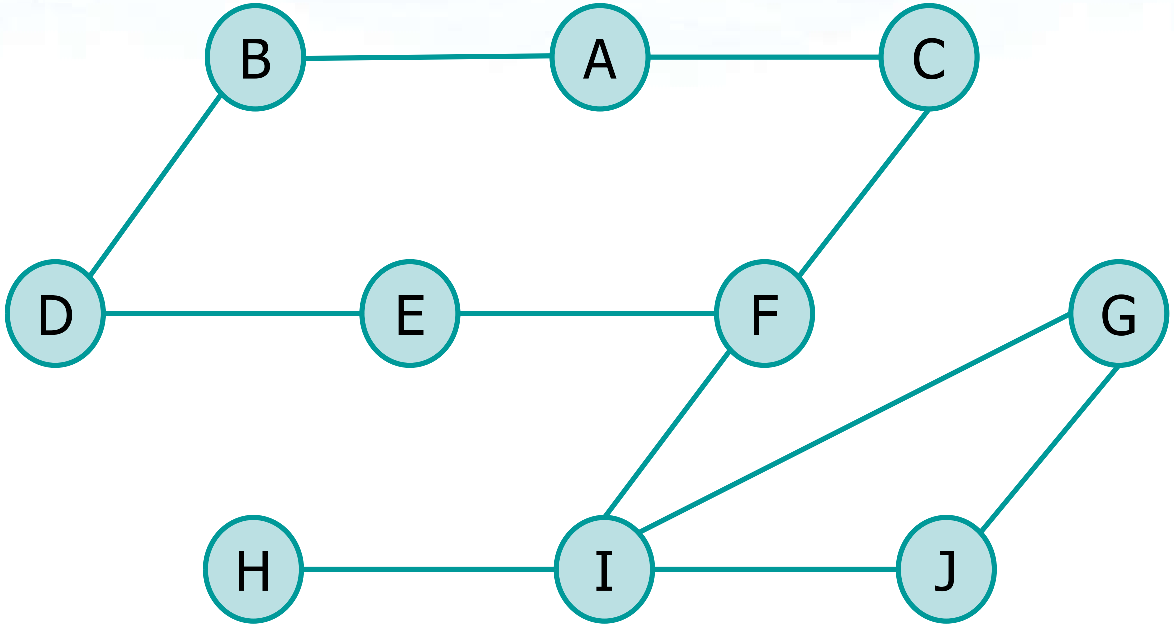
$$Bridges: \ \{G, K\} \ and \ \{K, L\}$$
$$Articulation\ points: \ G \ and \ K$$

Exercise

- Given the following graph G, find articulation points, bridges, and connected components
  - Find the connected component once removing the articulation points



Solution



$$Bridges: \ \{HI, FI\}$$
$$Articulation\ points: \ \{F, I\}$$
$$CC: \ one\ with\ all\ vertices$$
$$CC\ (after\ removing\ F\ and\ I): \ \{ABCDE, GJ, H\}$$