Graph

Applications of Graph-Search Algorithms

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Reverse graph

- Given a directed graph $G = (V, E)$
  - Its reverse (or transpose) graph
    - $G^T = (V, E^T)$

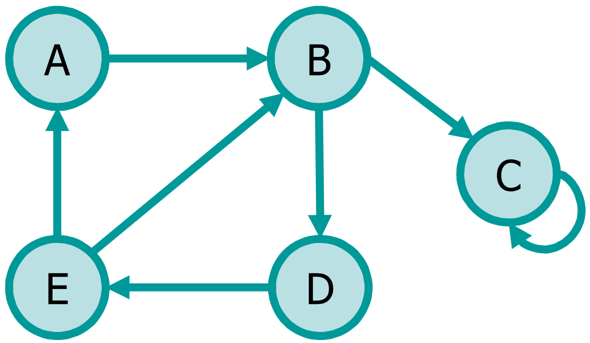
    is such that
    - If $(u, v) \in E$ then $(v, u) \in E^T$

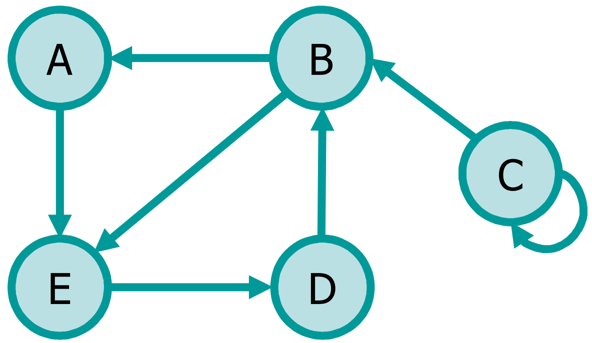
Example

	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;
}
```

Transpose the matrix

Implementation (with adjacency list)

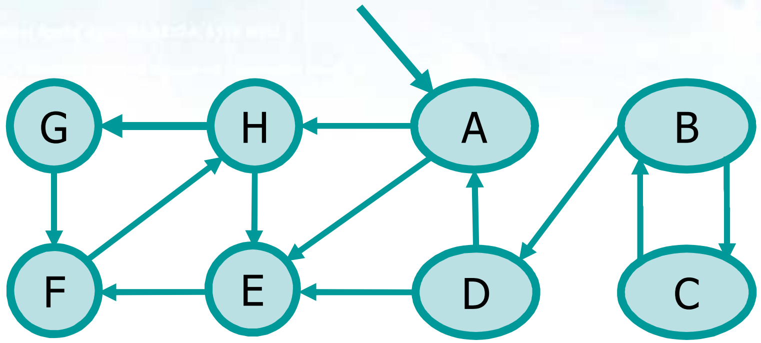
```
graph_t *graph_transpose (graph_t *g) {
  graph_t *h = NULL;
  vertex_t *tmp;
  edge_t *e;
  int i;
  h = (graph_t *) util_calloc (1, sizeof(graph_t));
  h->nv = g->nv;
  for (i=h->nv-1; i>=0; i--)
    h->g = new_node (h->g, i);
  tmp = g->g;
  while (tmp != NULL) {
    e = tmp->head;
    while (e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

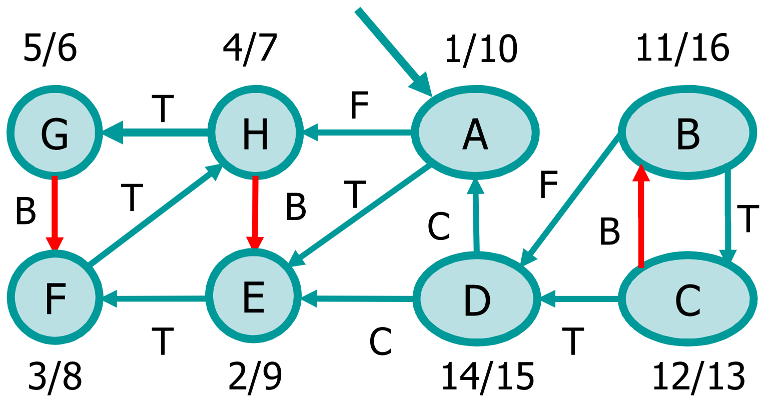
Insert a new edge

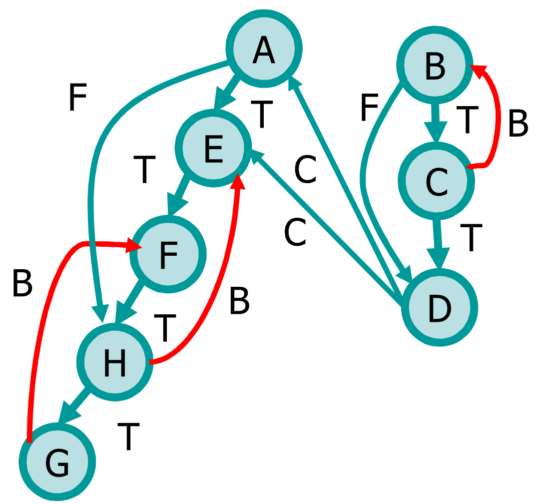
Loop detection

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

Example

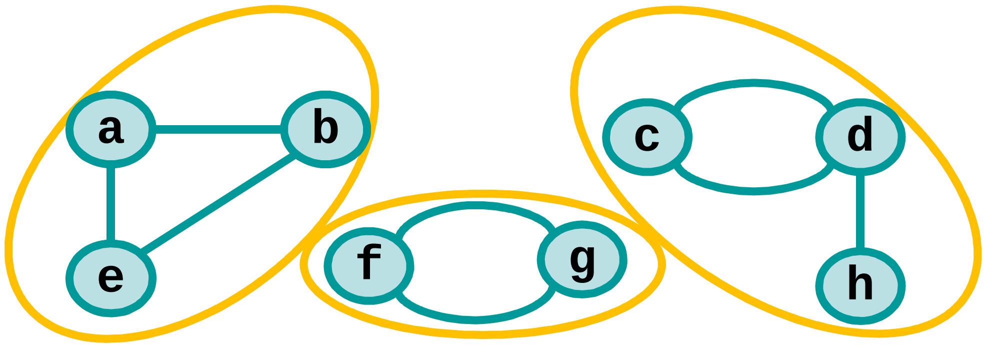






Connection in undirected graphs

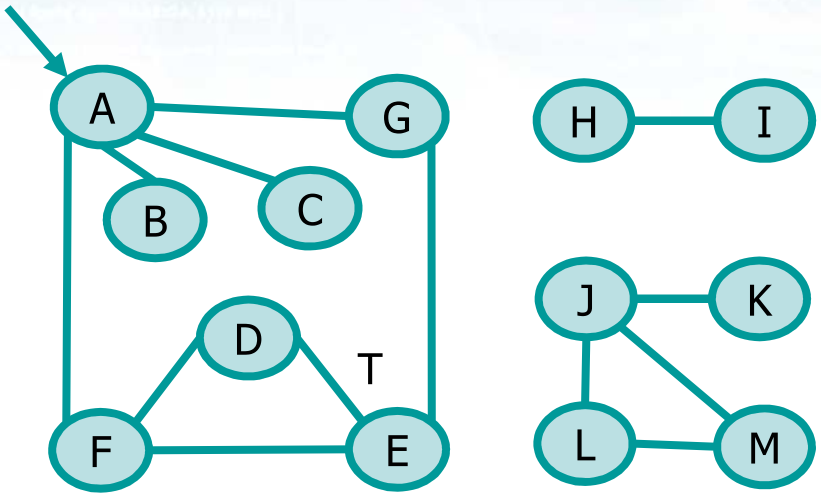
- An undirected graph is said to be connected iff
  - $\forall v_i, v_j \in V$ there exists a path p such that $v_i \rightarrow_p v_j$
- In an undirected graph
  - Connected component
    - Maximal connected subgraph, that is, there is no superset including it which is connected
  - Connected undirected graph
    - Only one connected component



Connected components

- 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

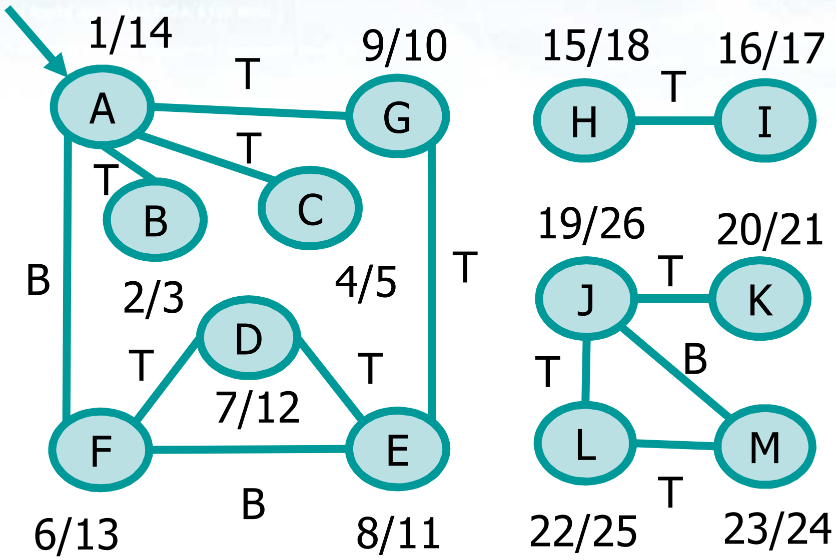
Example

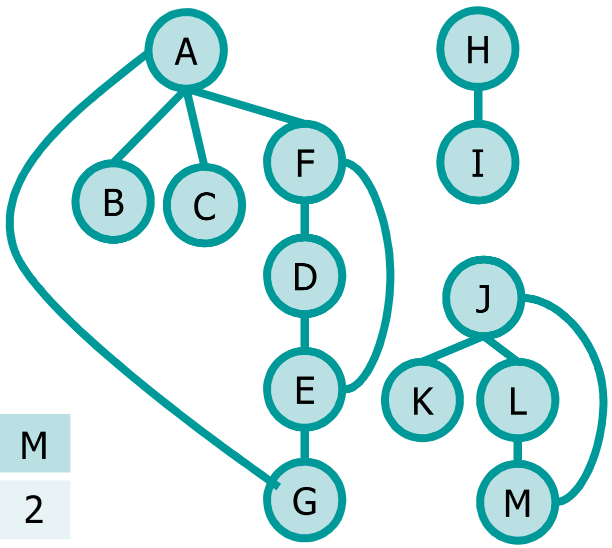


Connected Component Ids

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

Solution



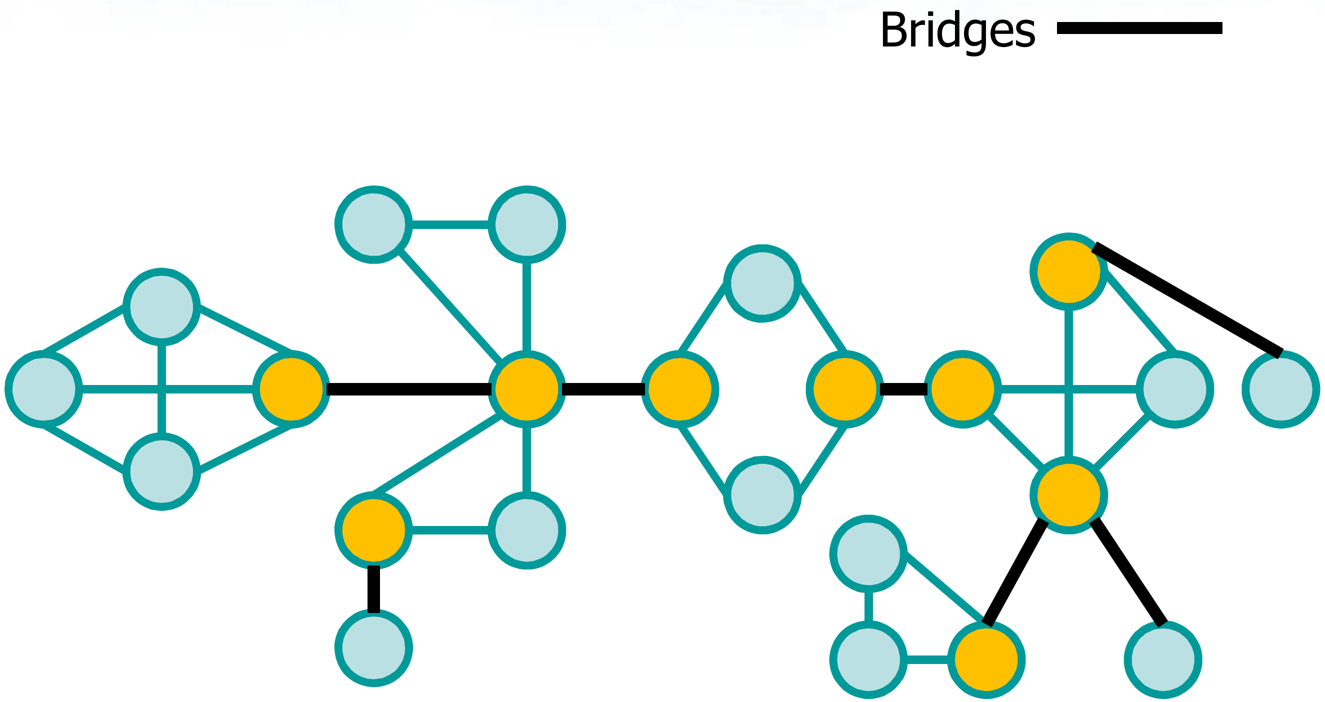


A	B	C	D	E	F	G	H	I	J	K	L	M
0	0	0	0	0	0	0	1	1	2	2	2	2

Bridges

- Given an undirected and connected graph, find out whether the property of being connected is lost because
  - An edge is removed
- Bridge
  - Edge whose removal disconnects the graph

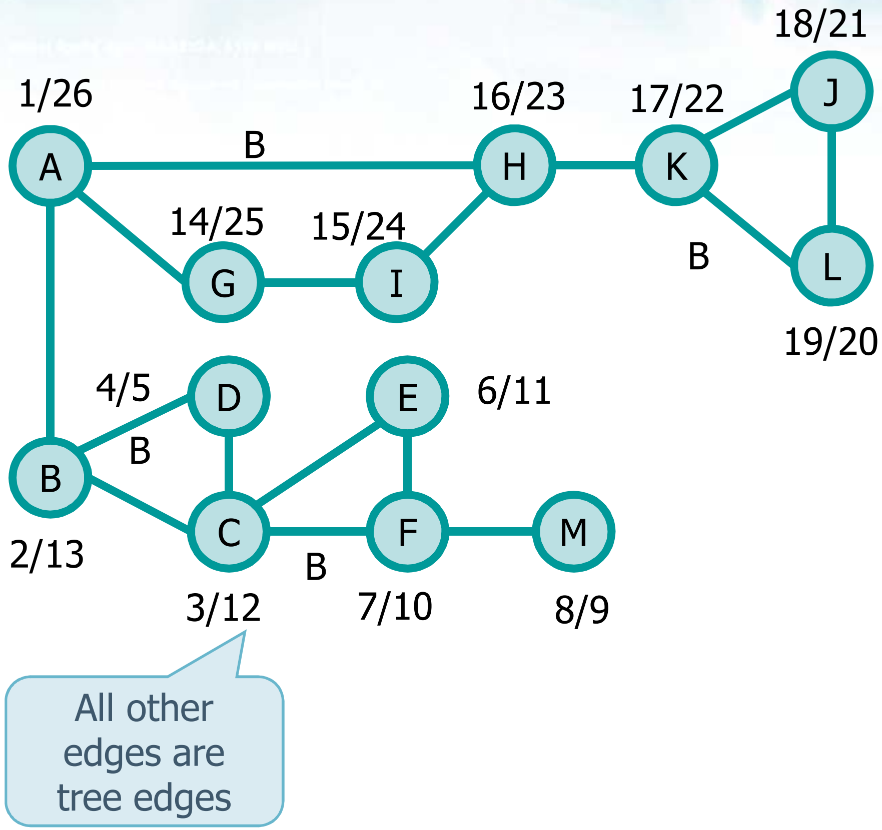
Example

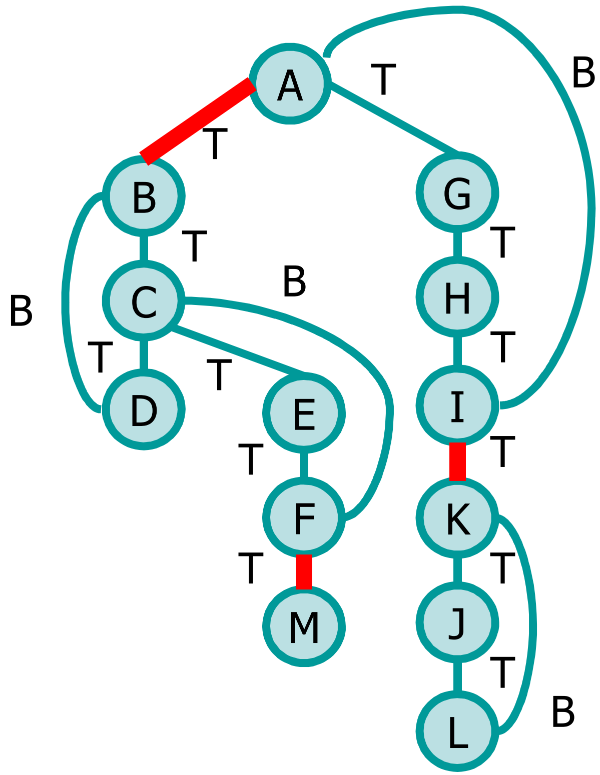


Bridges

- An edge (v,w)
  - Labelled Back (B) cannot be a bridge
    - Nodes v and w are also connected by a path in the DFS tree
  - Labelled Tree (T) is a bridge if and only if there are no edges labelled Back that connect a descendant of w to an ancestor of v in the DFS tree

Example


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

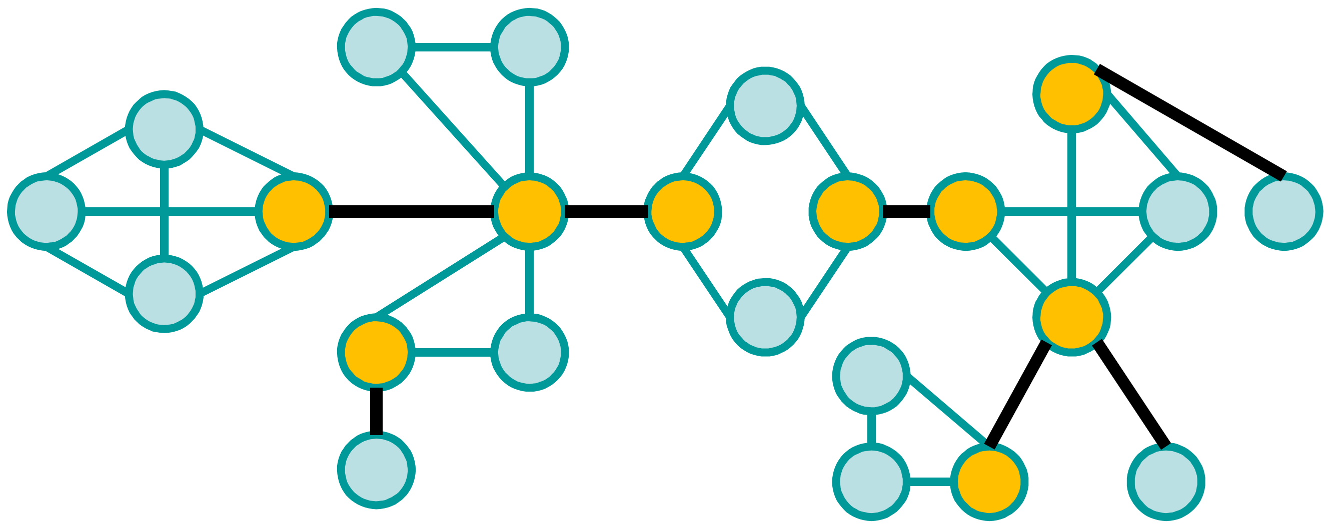

A
T
B
T
B
G
T
T
H
B
C
B
T
T
I
T
D
E
T
F
K
T
T
M
J
T
L
B

Articulation points

- Given an undirected and connected graph, find out whether the property of being connected is lost because
  - A node is removed
- Articulation point
  - Node whose removal disconnects the graph
  - Removing the vertex entails the removal of insisting (incoming and outgoing) edges as well

Example

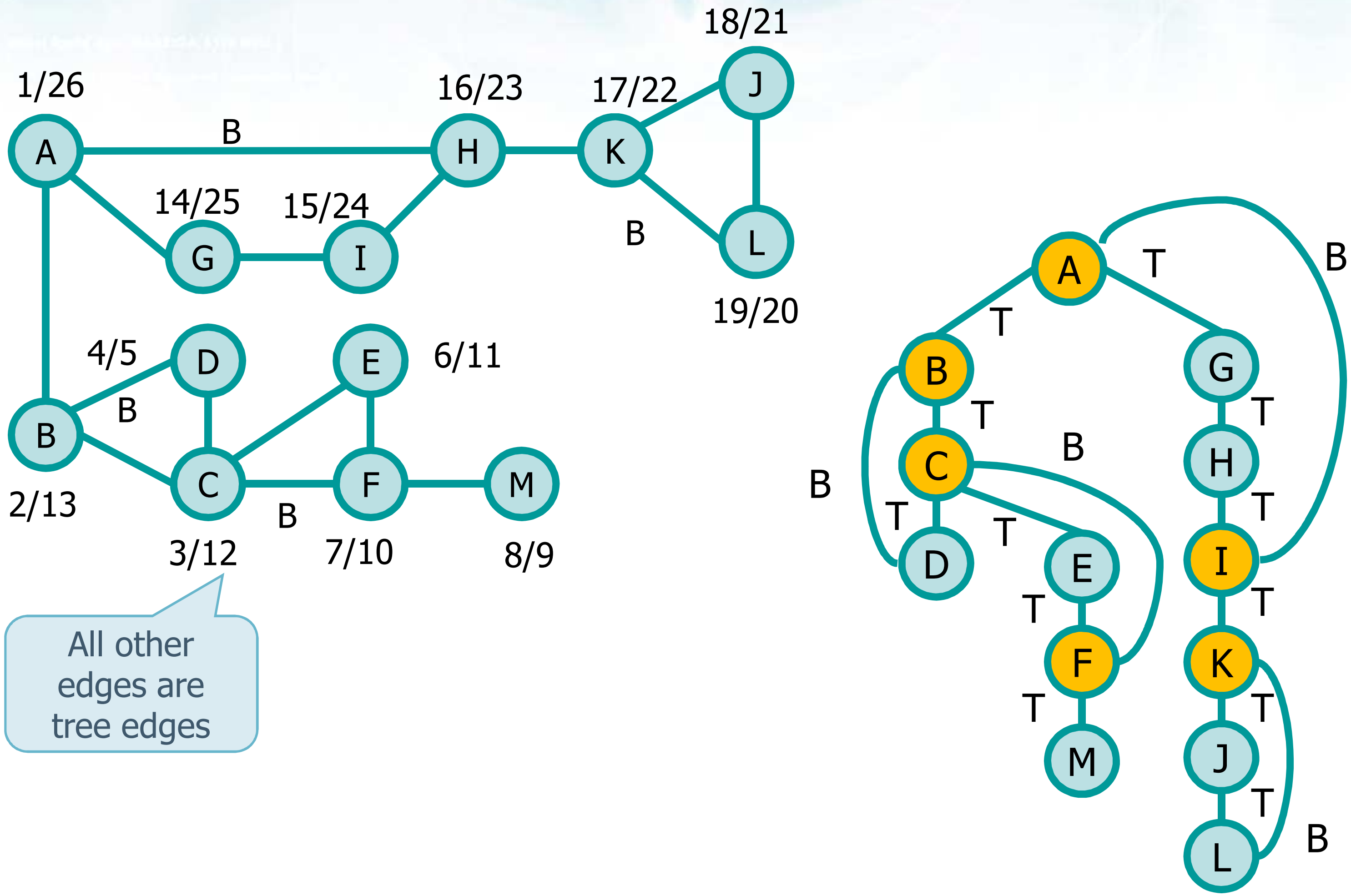
Articulation points



Articulation points

- Given an undirected graph G, 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 s such that there is no edge labelled B from s or from one of its descendants to a proper ancestor of v

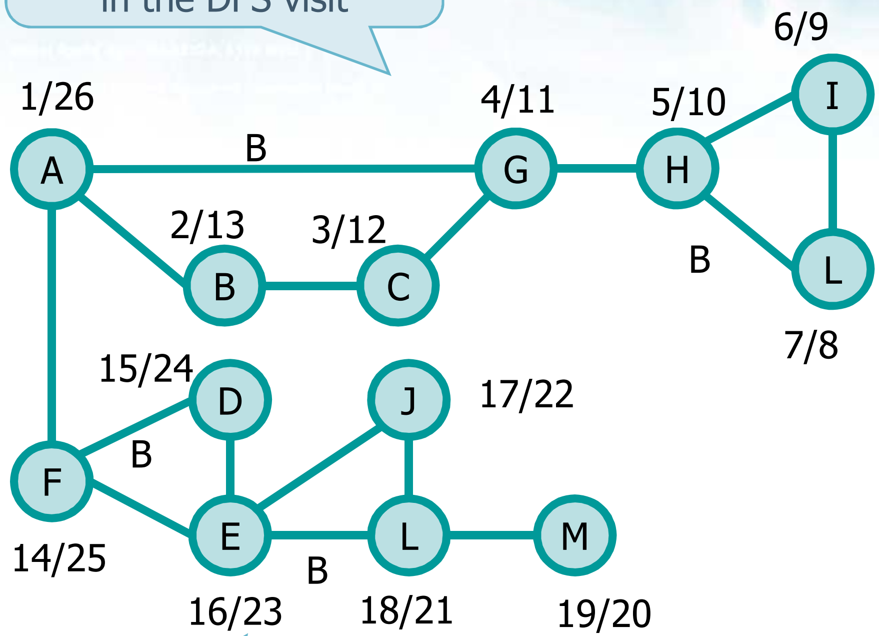
Example



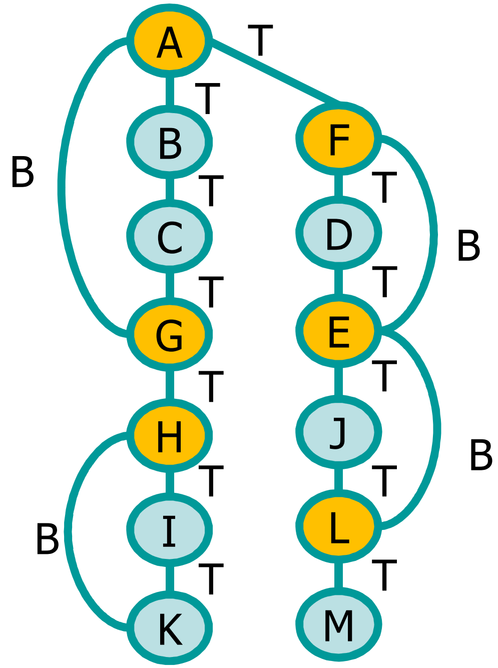
All other edges are tree edges

Example

Same example different ids and order in the DFS visit



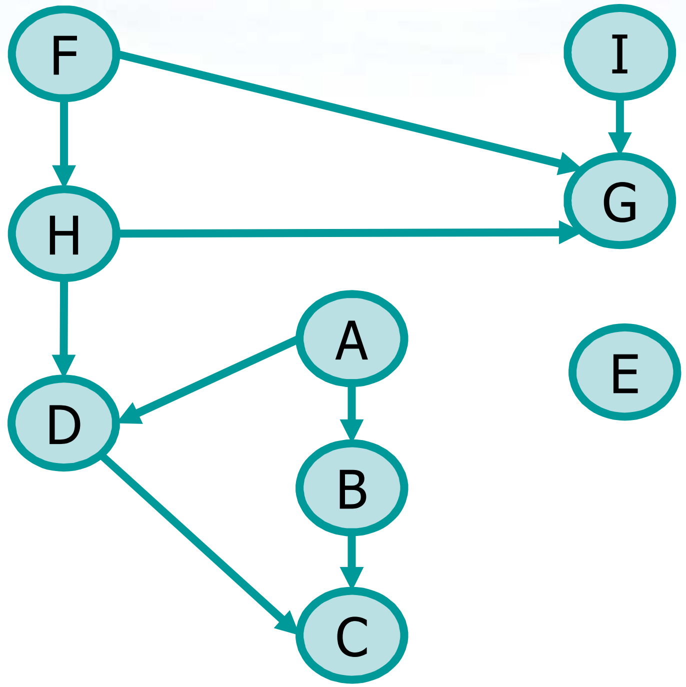
All other edges are tree edges



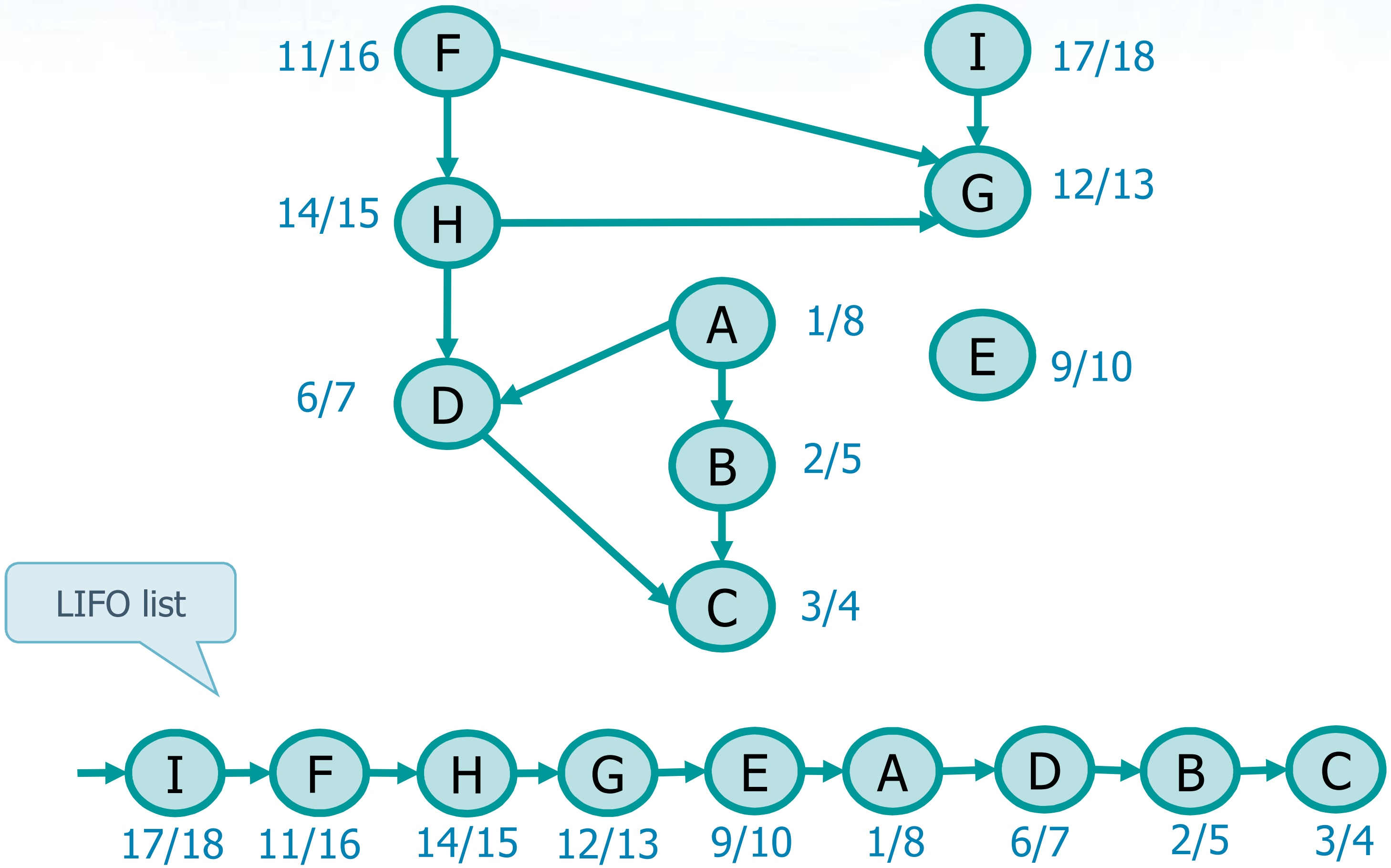
Directed Acyclic Graph (DAG)

- Topological sort (reverse)
  - Reordering the nodes according to a horizontal line, so that if the (u, v) edge exists, node u appears to the left (right) of node v and all edges go from left (right) to right (left)
- 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

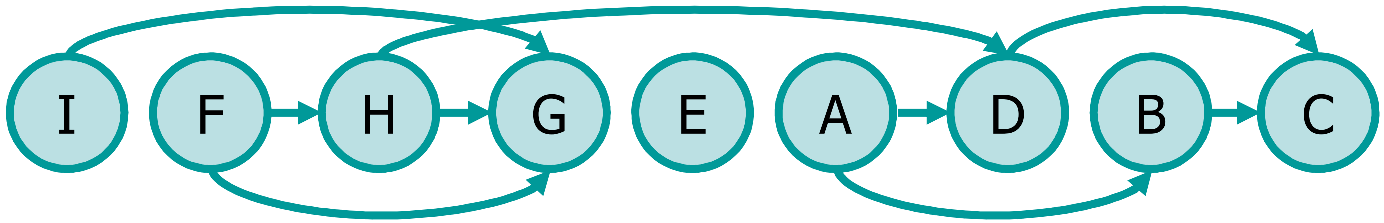


Solution

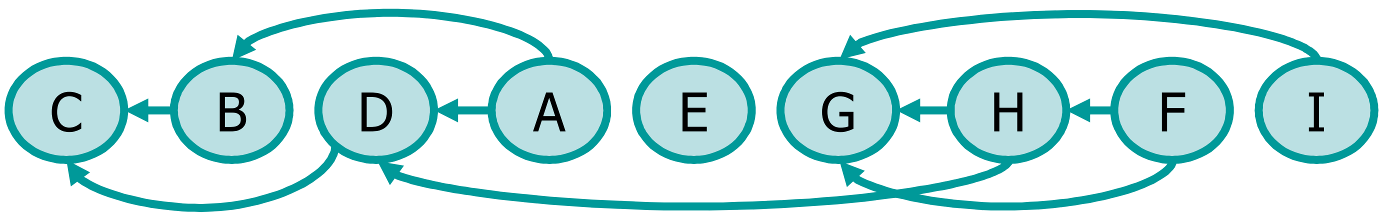

11/16
F
I
17/18
G
12/13
14/15
H
A
1/8
E
9/10
6/7
D
B
2/5
C
3/4
LIFO list
I F H G E A D B C
17/18 11/16 14/15 12/13 9/10 1/8 6/7 2/5 3/4

Topological Sort

- Topological sort
  - With a DAG represented by an adjacency matrix, it is enough to invert references to rows and columns



- Reverse topological sort



Implementation (with adjacency matrix)

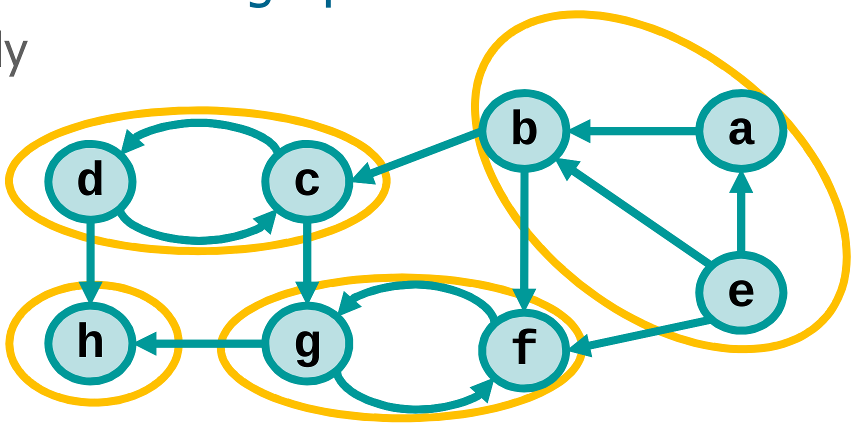
```
void graph_dag (graph_t *g){
  int i, *post, loop=0, timer=0;
  post = (int *)util_malloc(g->nv*sizeof(int));
  for (i=0; i<g->nv; i++) {
    if (g->g[i].color == WHITE) {
      timer = graph_dag_r (g, i, post, timer, &loop);
    }
  }
  if (loop != 0) {
    fprintf (stdout, "Loop detected!\n");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

Connection in directed graphs

- A directed graph is said to be strongly connected iff
  - $\forall v_i, v_j \in V$ there exists two paths p, p' such that $v_i \rightarrow_p v_j$ and $v_j \rightarrow_{p'} v_i$
- In a directed graph
  - Strongly connected component
    - Maximal strongly connected subgraph
  - Strongly connected directed graph
    - Only one strongly connected component

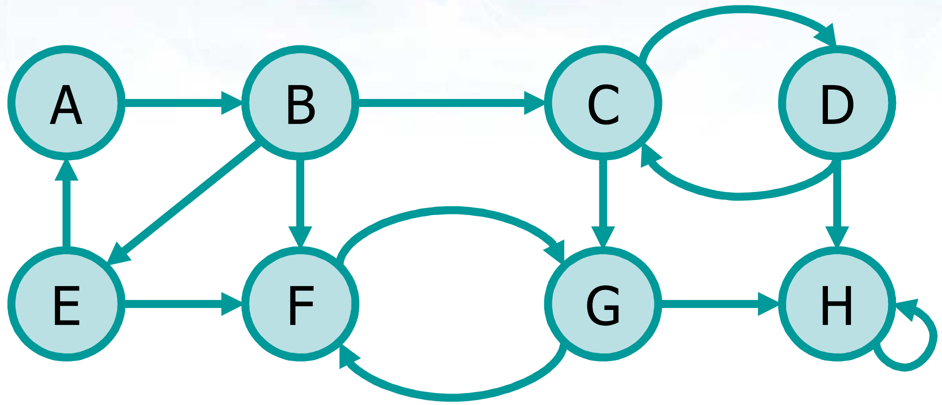


Strongly Connected Component (SCC)

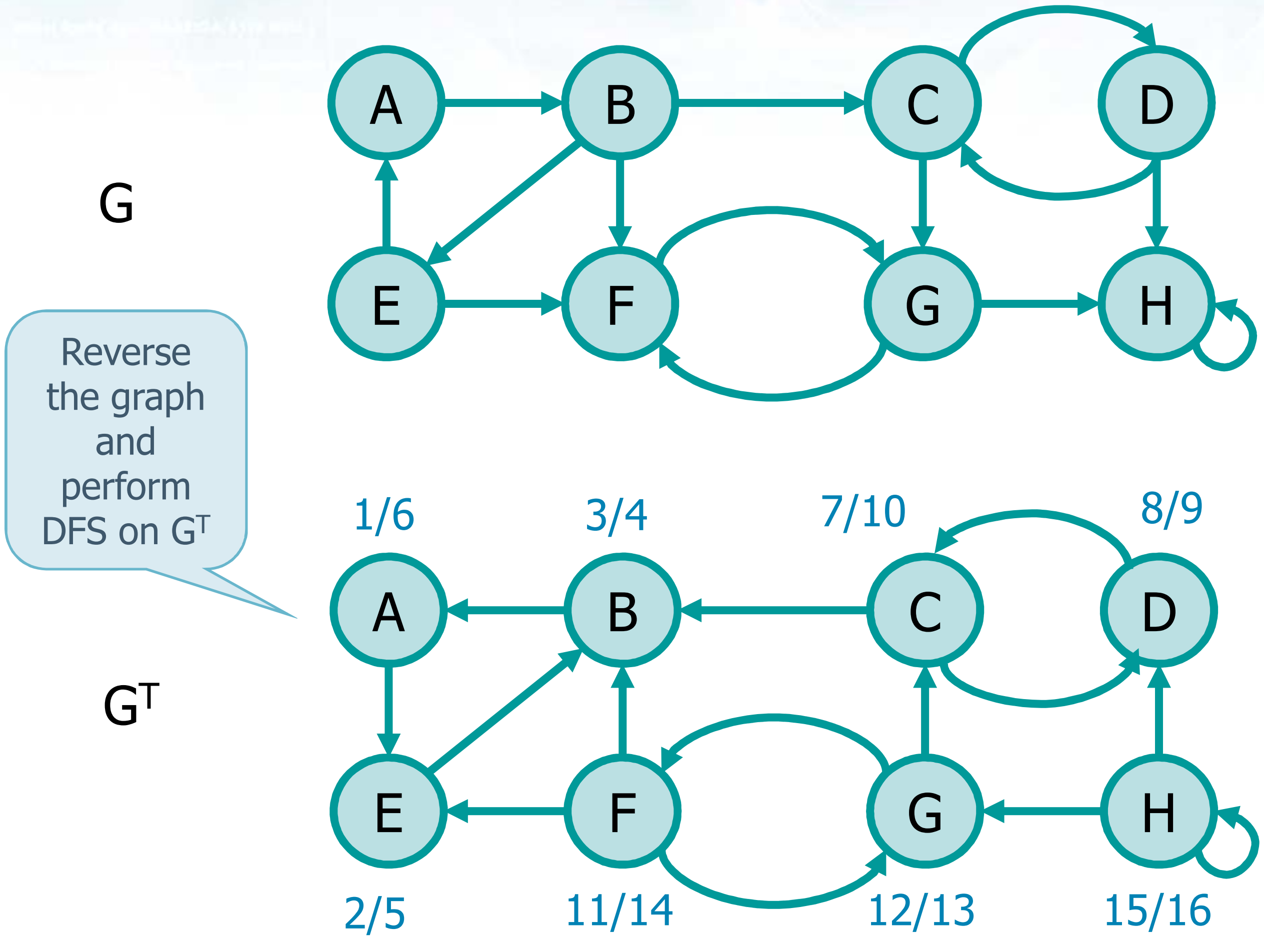
- Kosaraju's algorithm ('80s)
  - Reverse the graph
  - Execute DFS on the reverse graph, computing discovery and end-processing times
  - Execute DFS on the original graph according to **decresasing** end-processing times
  - The trees of this last DFS are the strongly connected components

Example

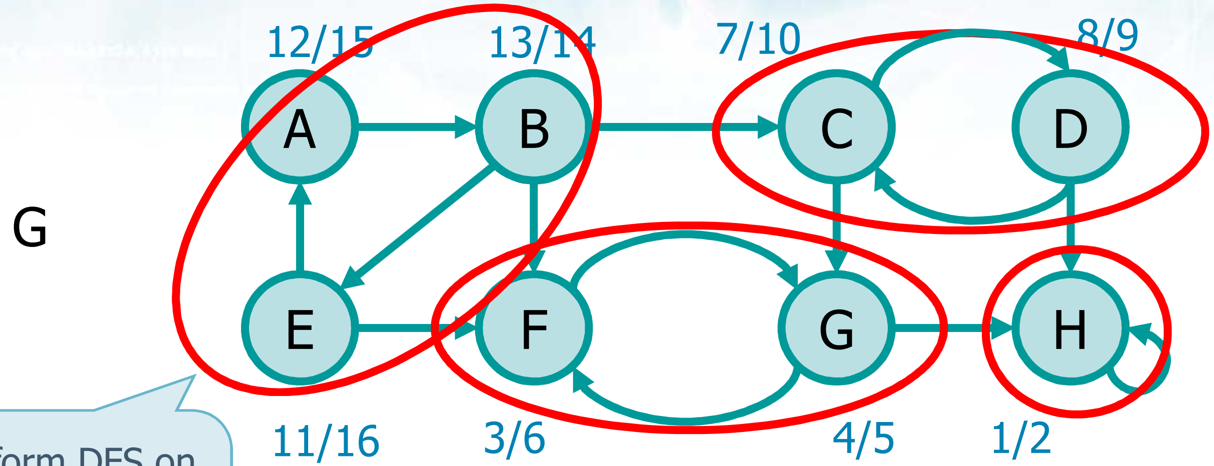
G



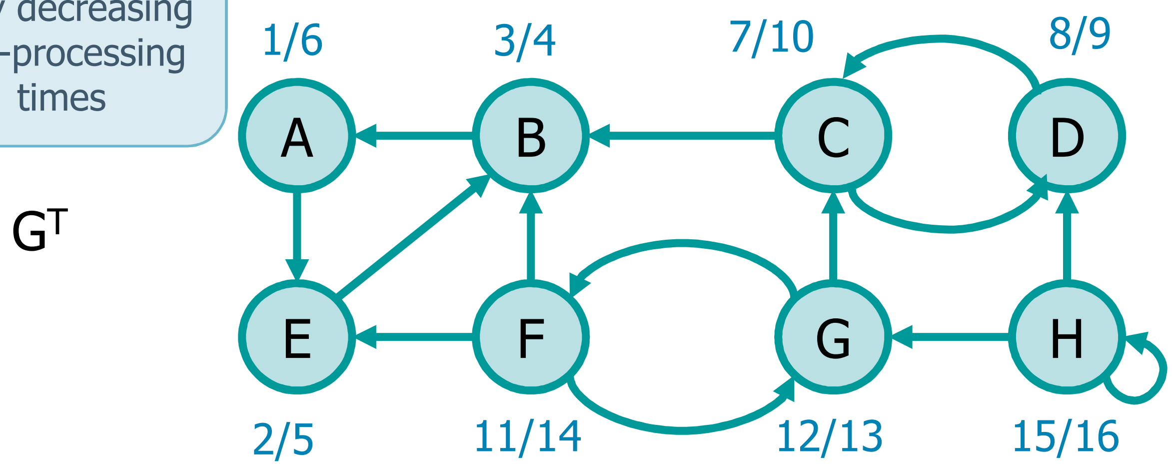
Solution


G
A
B
C
D
E
F
G
H
Reverse the graph and perform DFS on G^T
G^T
1/6
3/4
7/10
8/9
A
B
C
D
E
F
G
H
2/5
11/14
12/13
15/16

Solution

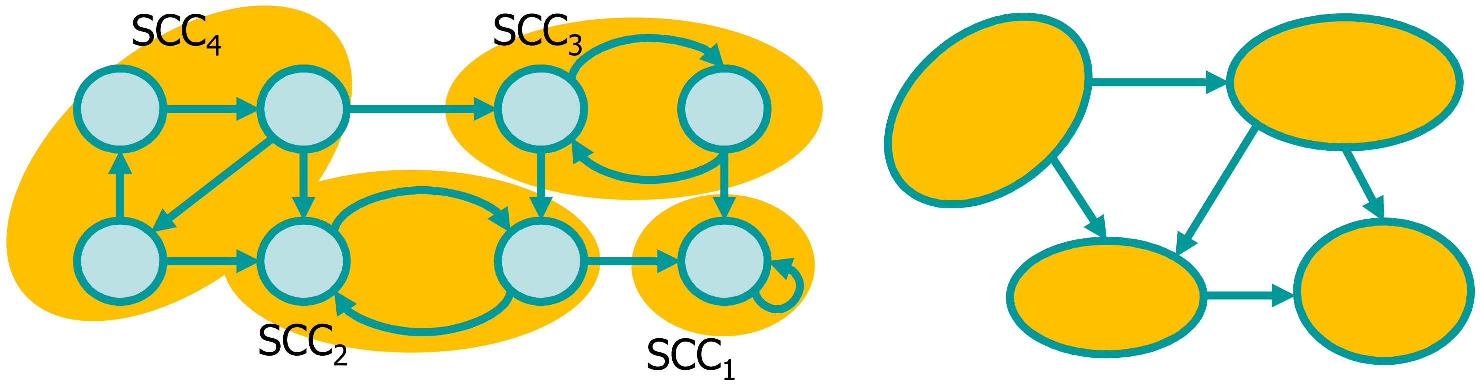


Perform DFS on G by decreasing end-processing times



Considerations

- SCCs are equivalence classes with respect to the mutual reachability property
- 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);
```

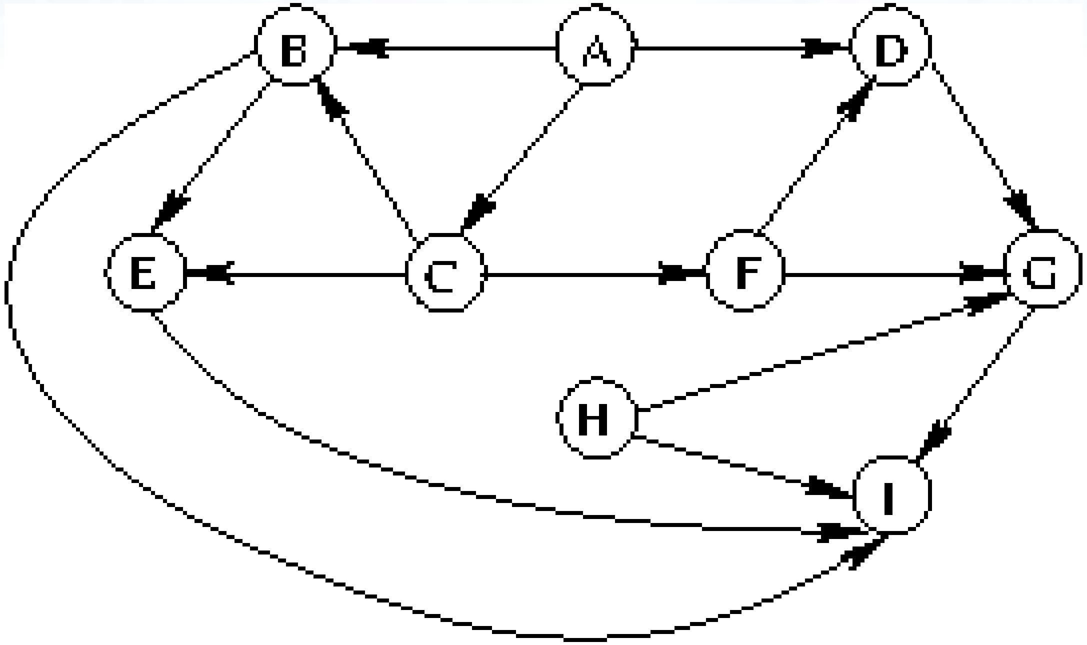
Implementation (with adjacency matrix)

```
  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)

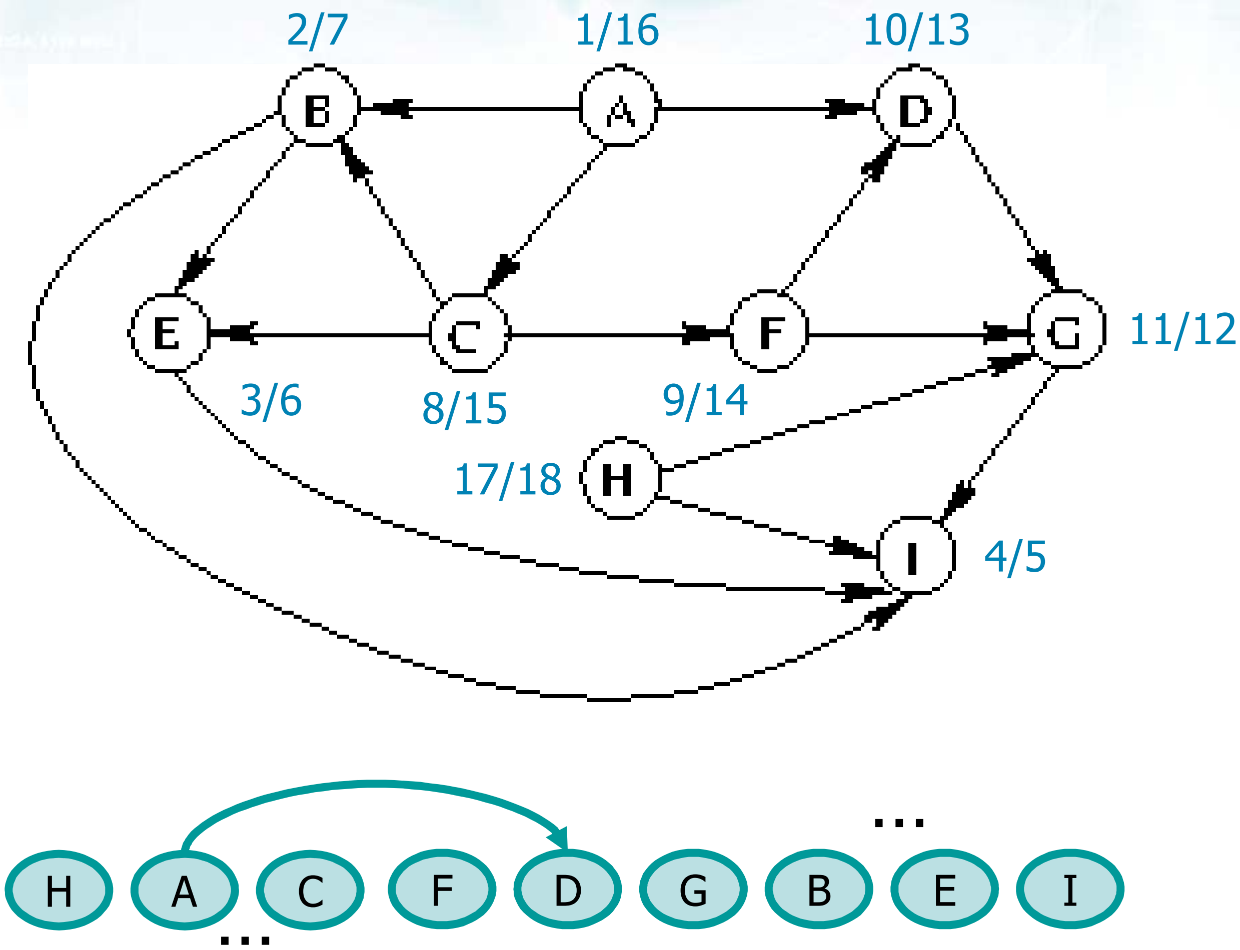
```
int graph_scc_r(
  graph_t *g, int i, int *post, int id, int t
) {
  int j;
  g->g[i].color = GREY;
  g->g[i].scc = id;
  for (j=0; j<g->nv; j++) {
    if (g->g[i].rowAdj[j]!=0 &&
        g->g[j].color==WHITE) {
      t = graph_scc_r (g, j, post, id, t);
    }
  }
  g->g[i].color = BLACK;
  post[t++]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

  return t;
}
```

Exercise

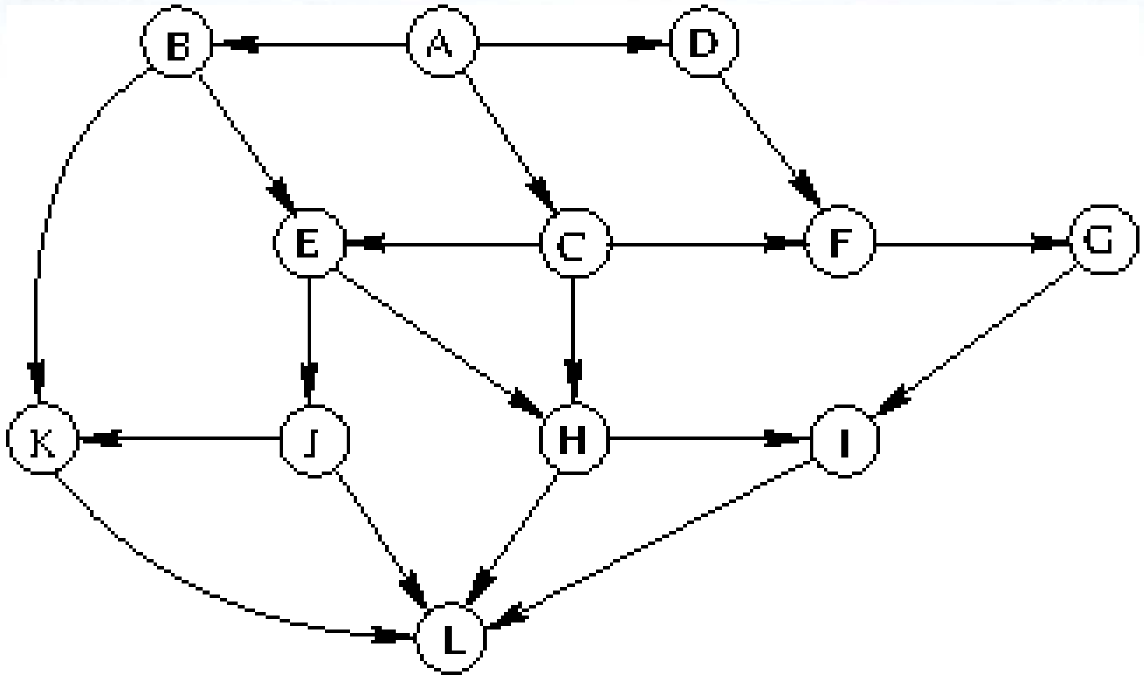


- Given the previous DAG find the topological order of all vertices

Solution

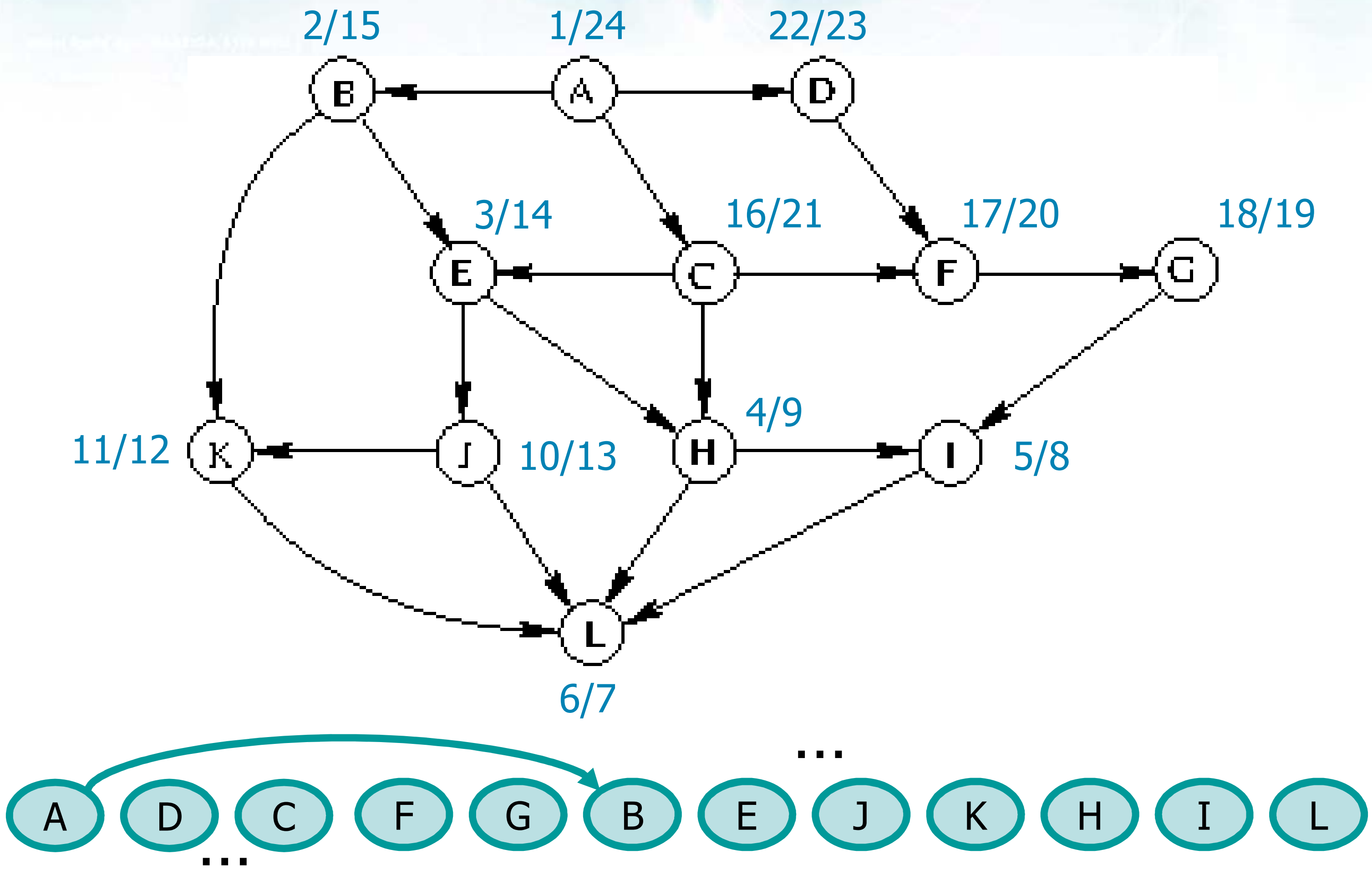


Exercise

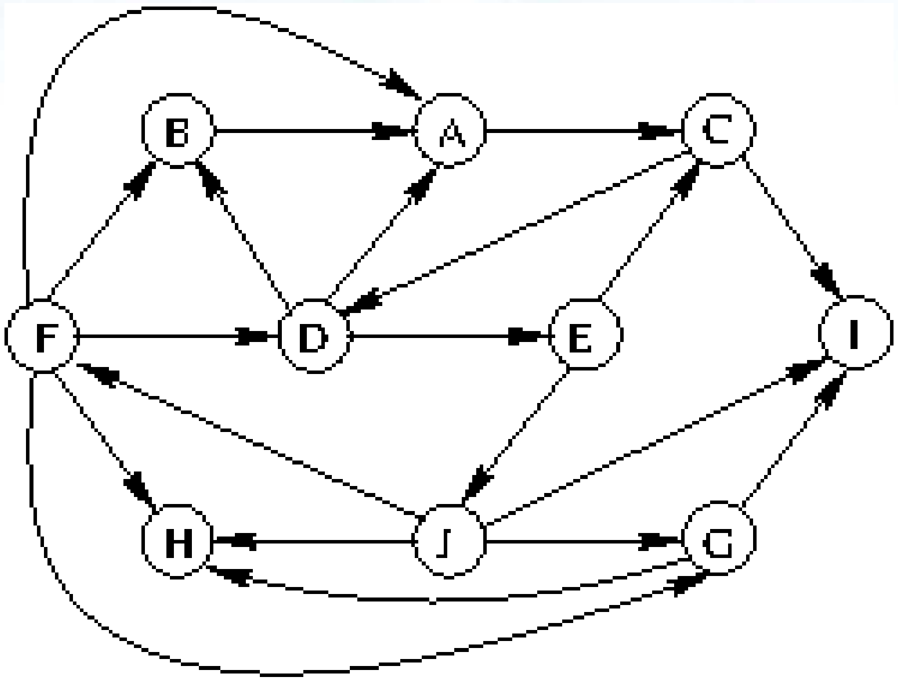


- Given the previous DAG find the topological order of all vertices

Solution

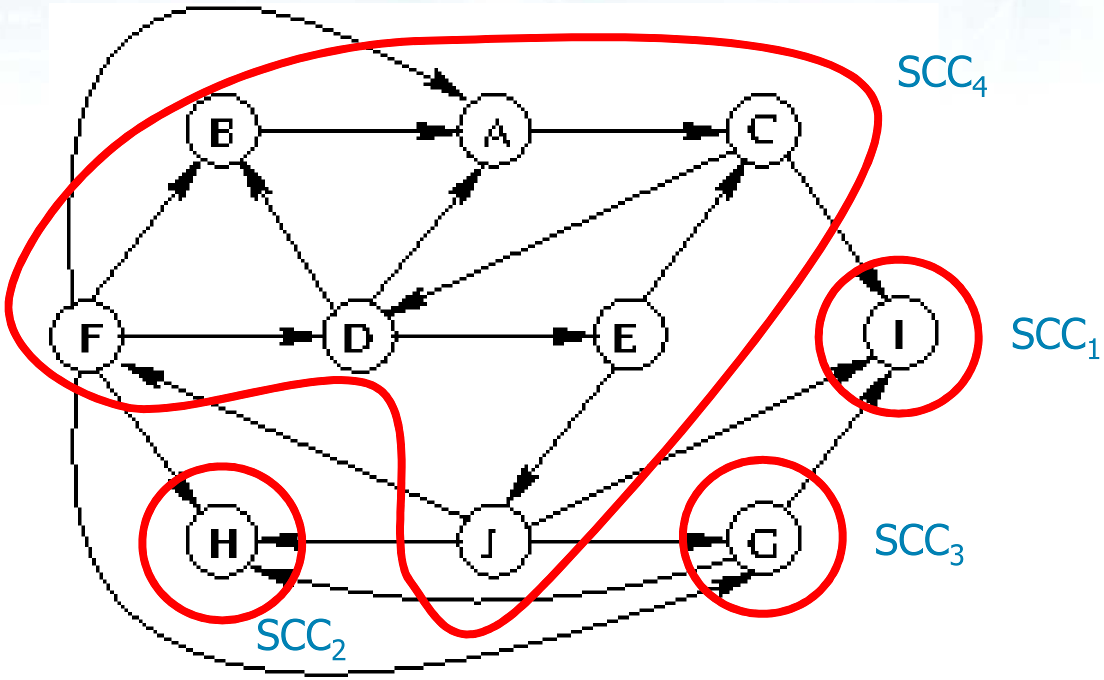

2/15
1/24
22/23
B
A
D
3/14
16/21
17/20
18/19
E
C
F
G
4/9
11/12
K
J
10/13
H
I
5/8
L
6/7
...
A
D
C
F
G
B
E
J
K
H
I
L
...

Exercise



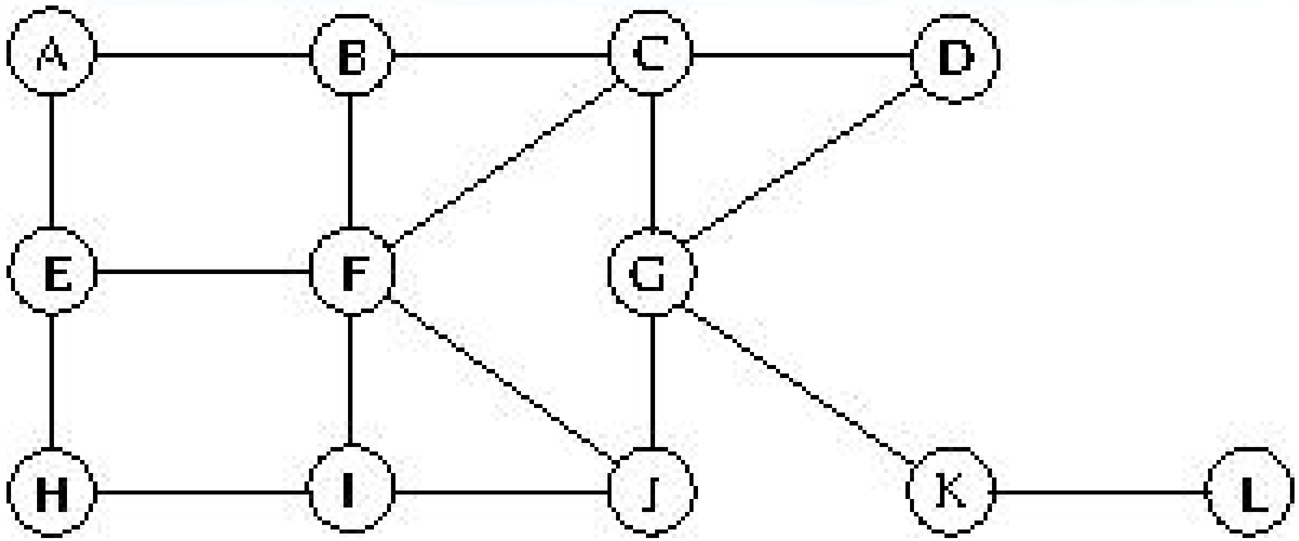
❖ Given the previous graph, find its SCC

Solution



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

Exercise

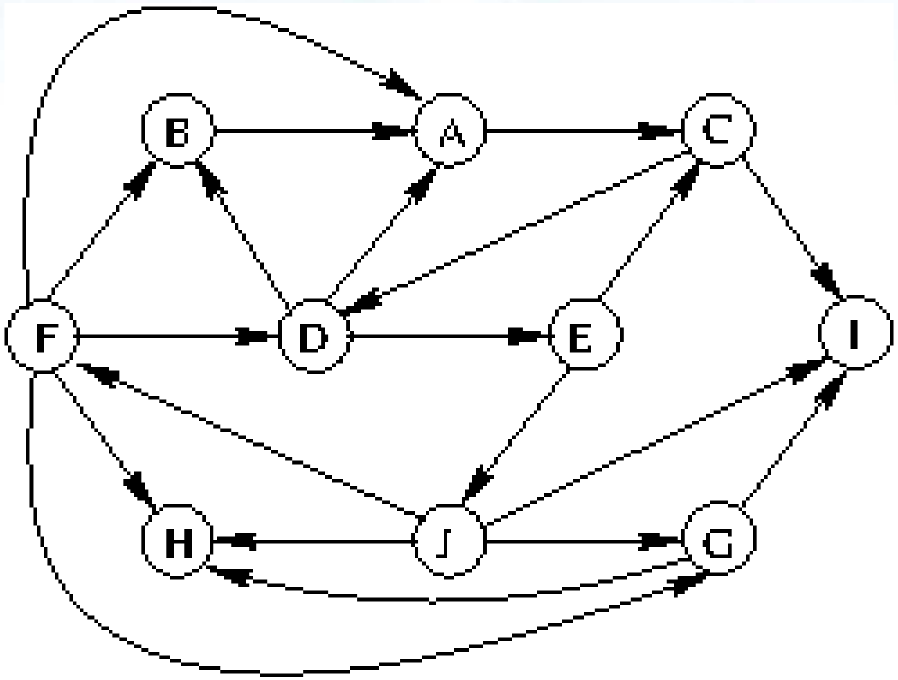


- Given the previous graph find articulation points

Solution

- Articulation points
  - G and K

Exercise



- Given the previous graph, transform it into an undirected graph and find articulation points, bridges, and connected components

Solution

- Articulation points
  - None
- Bridges
  - None
- Connected Components
  - One with all vertices, {A, B, C, D, E, F, G, H, I, J}