Graph

Applications of Graph-Search Algorithms

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Reverse graph

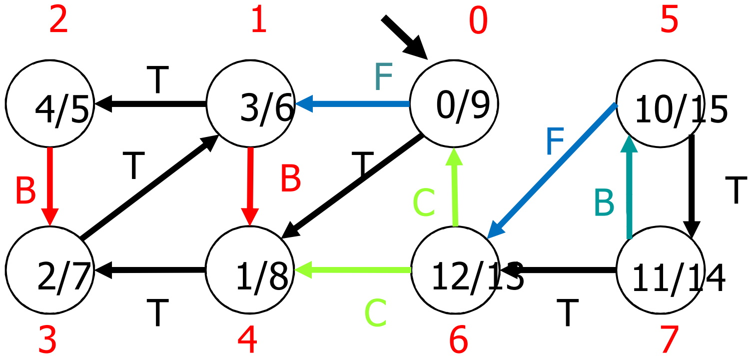
- Given a directed graph G =(V, E)
  - Its reverse (or tanspose) graph $G^T = (V, E^T)$ is such that $(u, v) \in E$ $(v,u) \in E^T$

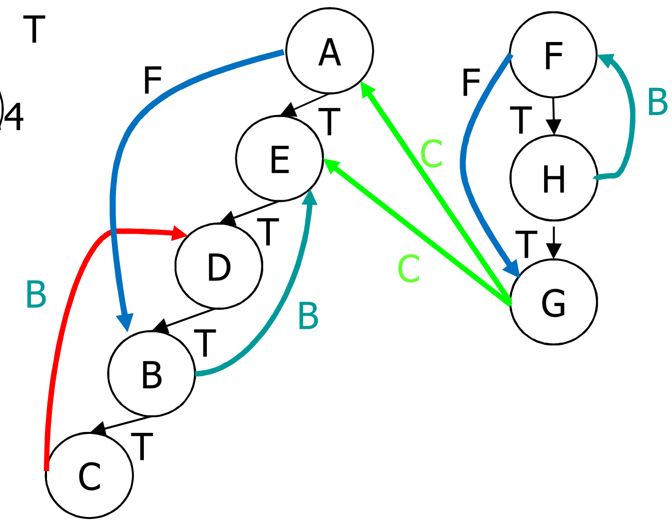
With Adjacency Matrix

```
graph_t *graph_transpose(graph_t *g, int nv) {
  graph_t *t;
  int i, j;
  t = (graph_t *)util_calloc(nv, sizeof(graph_t));
  for (i=0; i<nv; i++) {
    t[i] = g[i];
    t[i].rowAdj = (int *)util_calloc(nv, sizeof(int));
    for (j=0; j<nv; j++) {
      t[i].rowAdj[j] = g[j].rowAdj[i];
    }
  }
  return t;
}
```

Loop detection

- A graph is acyclic if and only if in a DFS there are no edges labelled Backward (B)

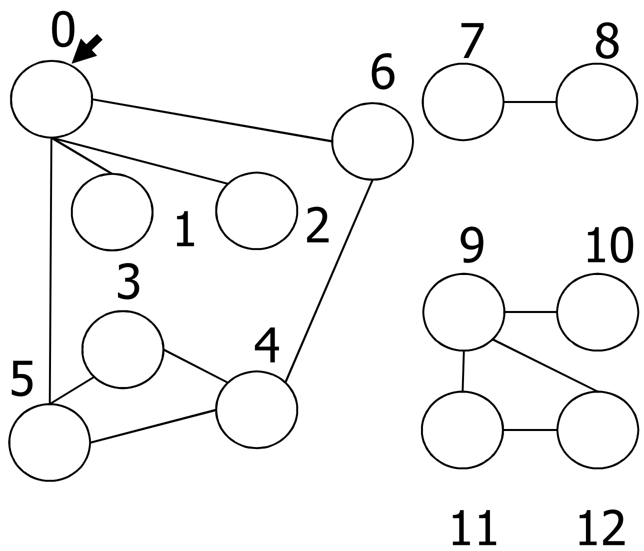




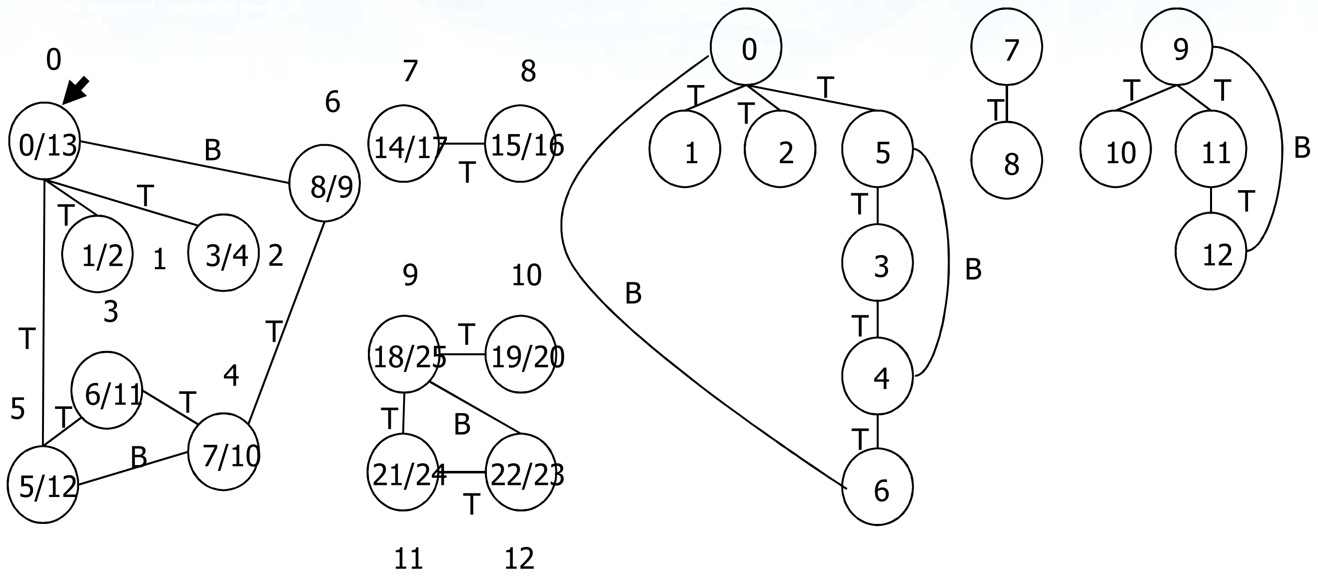
Connected components

- In an undirected graph represented as an adjacency list
  - Each tree of the DFS forest is a connected component
  - CC is an array that stores an integer identifying each connected component. Nodes serve as indexes of the array

Example



Example

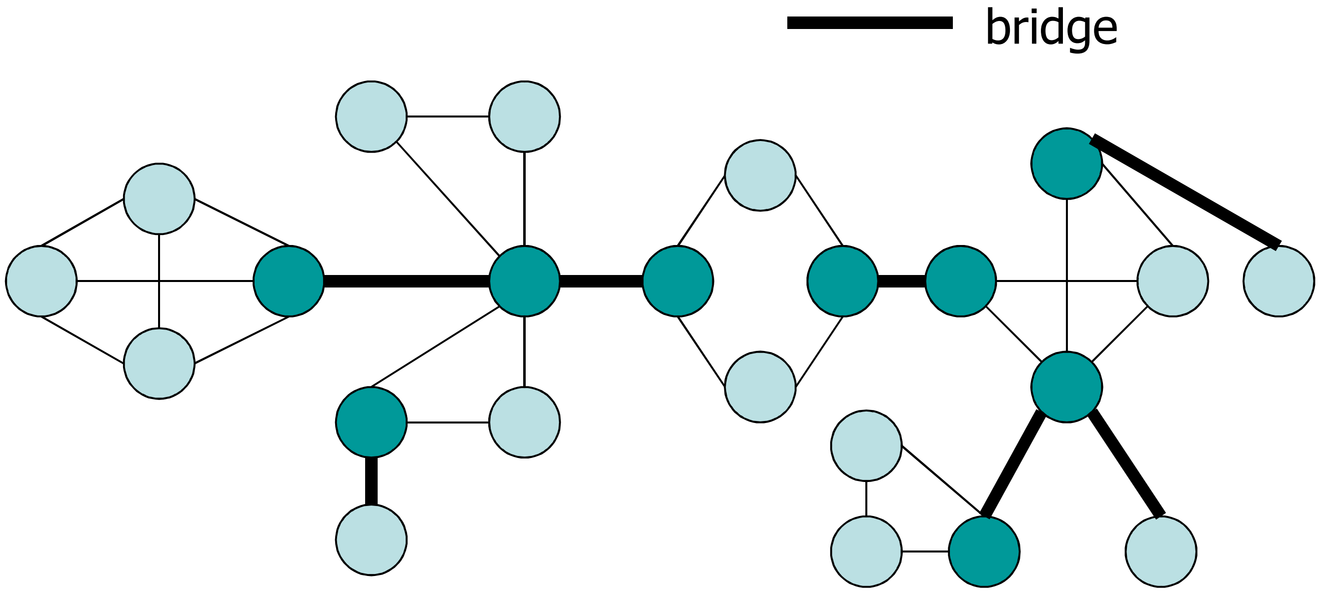


	0	1	2	3	4	5	6	7	8	9	10	11	12
cc	0	0	0	0	0	0	0	1	1	2	2	2	2

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

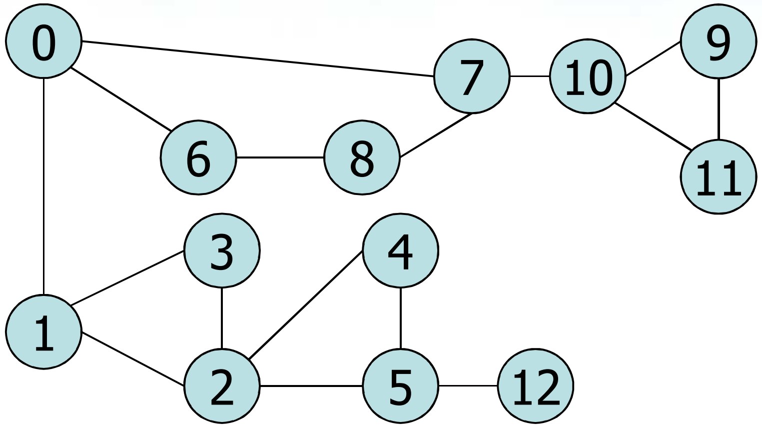
Example

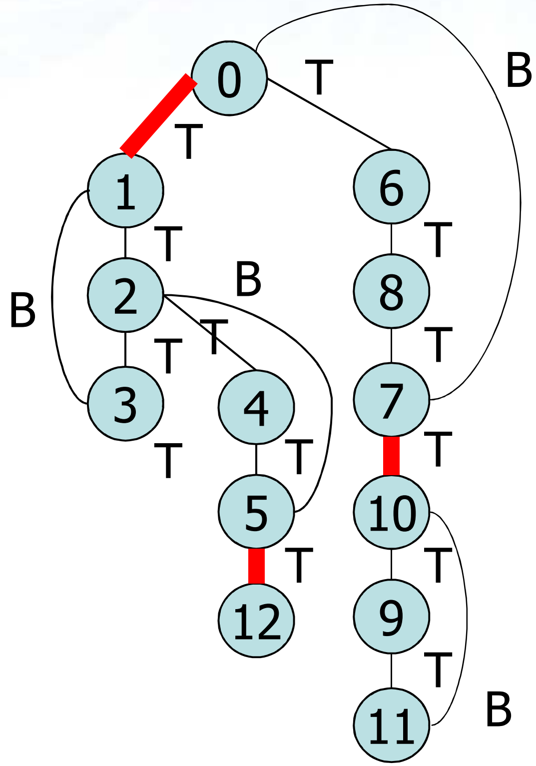


Bridges

- An edge (v,w)
  - Labelled Back (B) can’t be a brigdge
    - 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

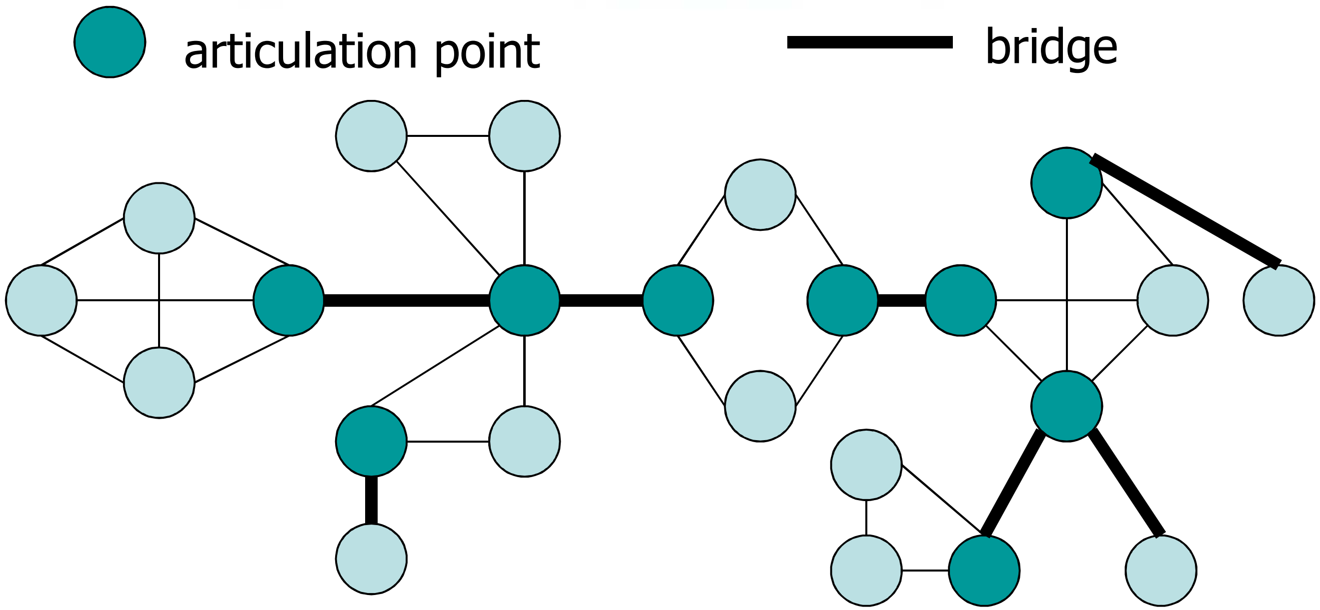




Articulation points

- 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

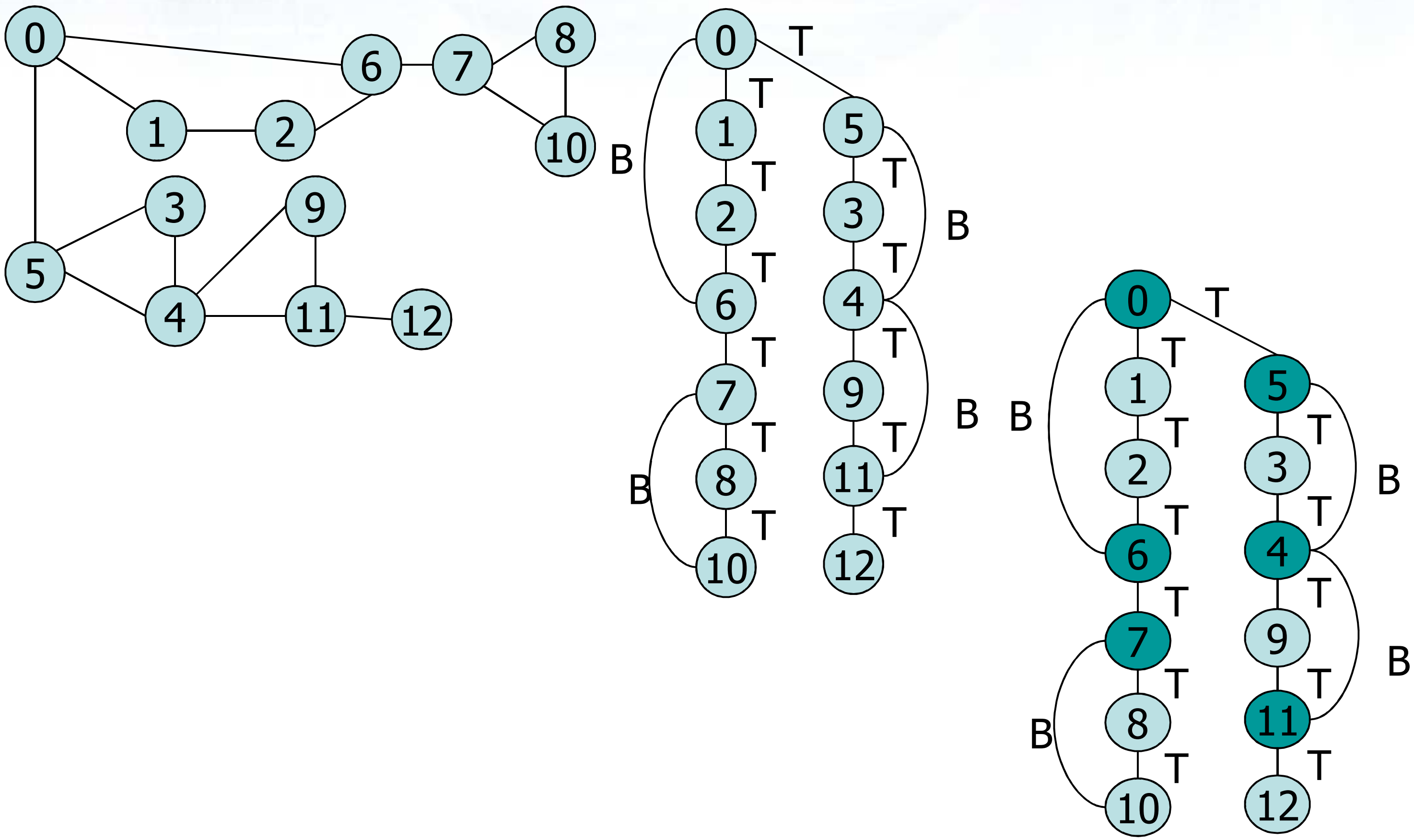
Example



Articulation points

- 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
  - Levaes 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



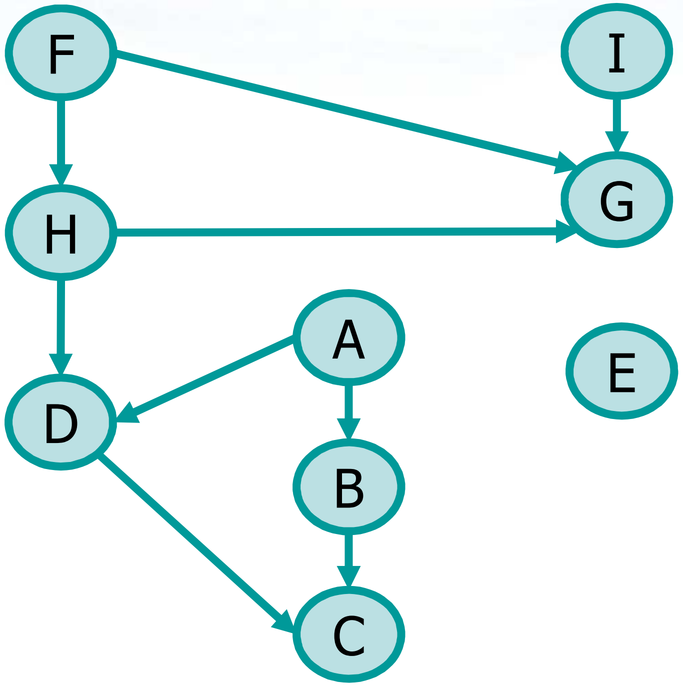
Directed Acyclic Graph (DAG)

- DAG
  - implicit models for partial orderings used in scheduling problems
- Scheduling
  - Given tasks and precedence constraints
  - How can we schedule tasks so that they are all executed satisfying the constraints

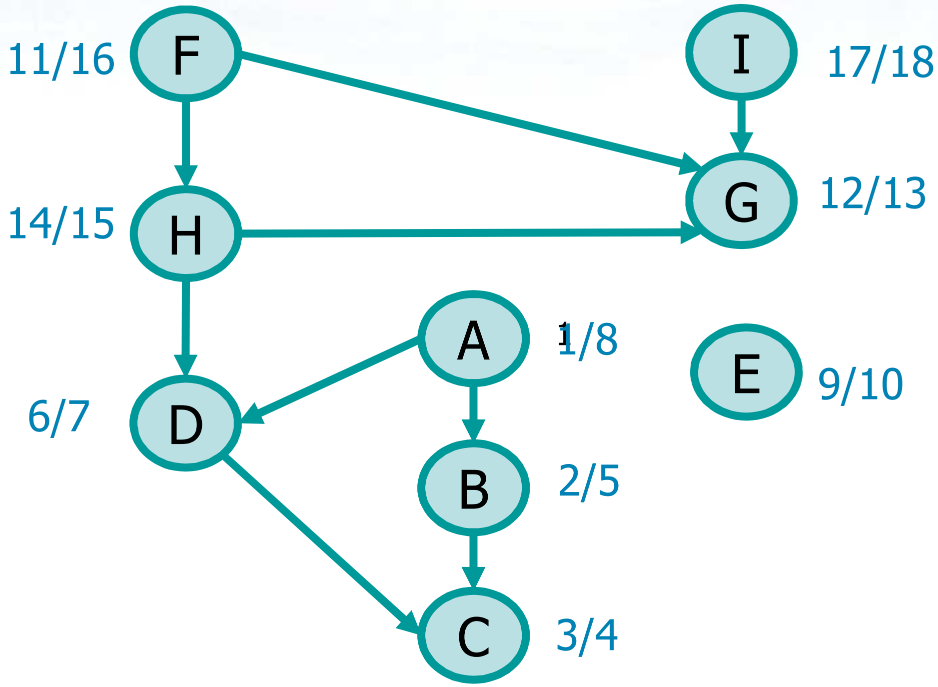
Directed Acyclic Graph (DAG)

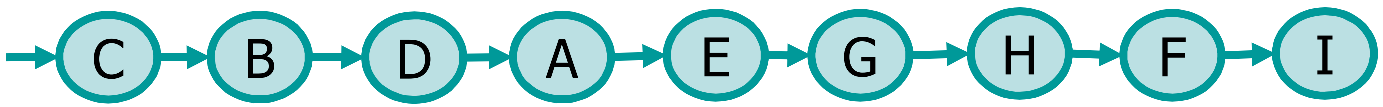
- Topological sort (reverse)
  - Reordering the nodes according to a horizontal line, so that if the (u, v) edge exists, node u appears to the left (right) of node v and all edges go from left (right) to right (left)
- Algorithm
  - Perform a DFS computing **end-processing** times
  - Order vertices using the end-processing times
- Alternative algorithm
  - Perform a DFS and when assigning end-processing times insert the vertex into a LIFO list

Example



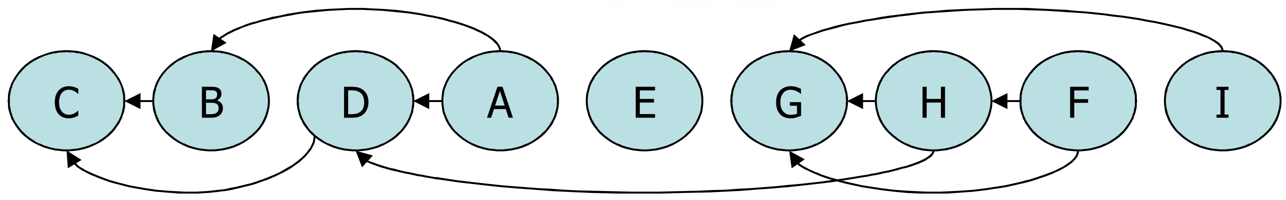
Example



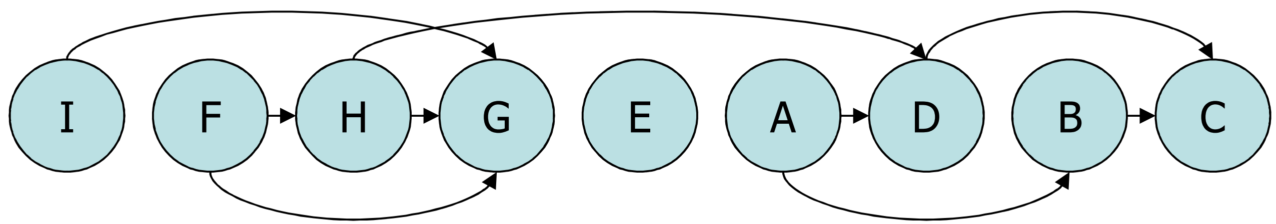


Topological Sort

- Reverse topological sort



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



Graph library: Topological Sort

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

Graph library: Topological Sort

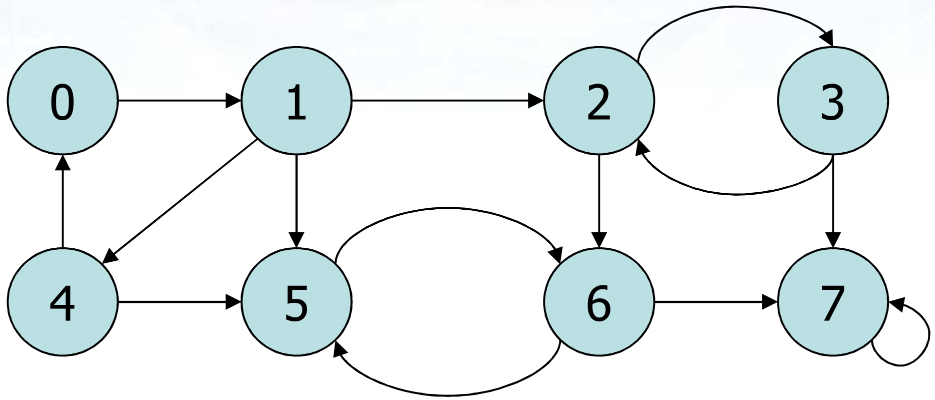
```
int graph_dag_r(
    graph_t *g, int nv, int i, int *post, int t,
    int *loop) {
  int j;
  g[i].color = GREY;
  for (j=0; j<nv; j++) {
    if (g[i].rowAdj[j] != 0) {
      if (g[j].color == GREY) {
        *loop = 1;
      }
      if (g[j].color == WHITE) {
        t = graph_dag_r(g, nv, j, post,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

Strongly Connected Component (SCC)

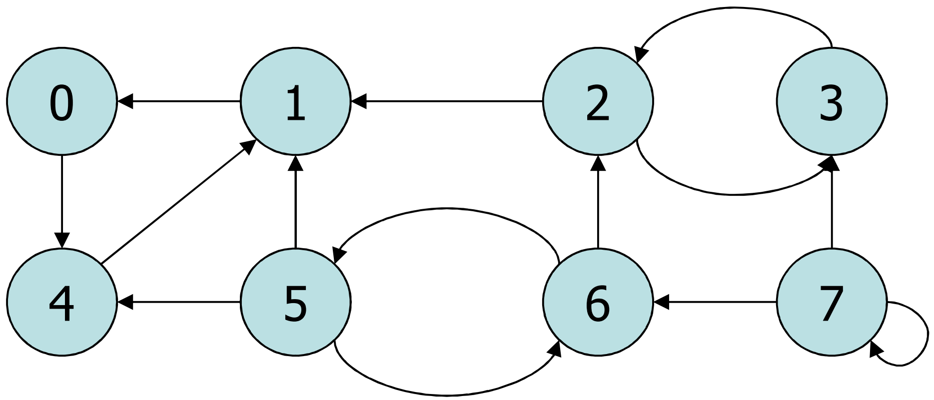
- Kosaraju's algorithm ('80s)
  - Reverse the graph
  - Execute DFS on the reverse graph, computing Discovery/Endprocessing times
  - Execute DFS on the original graph according to decresasing Endprocessing times
  - The trees of this last DFS are the strongly connected components

Example

G

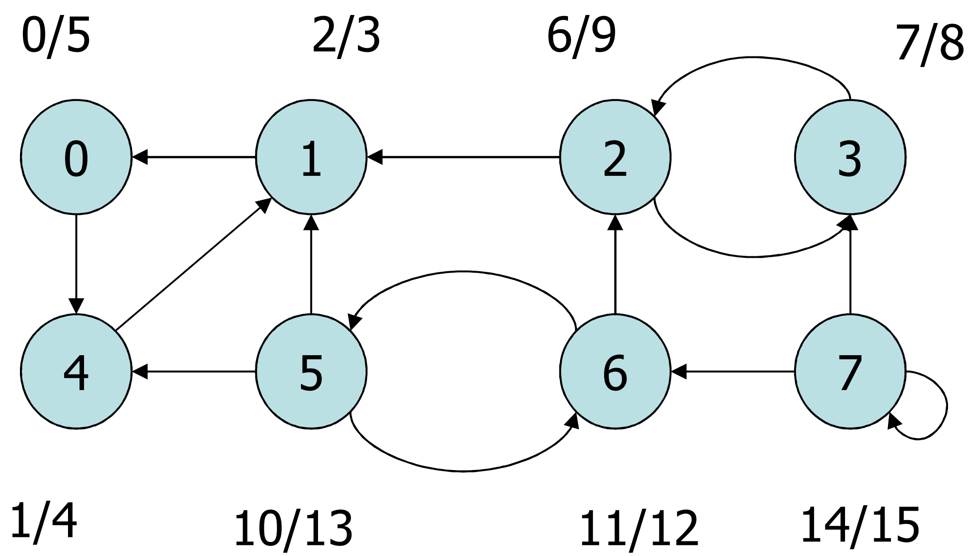


G^T



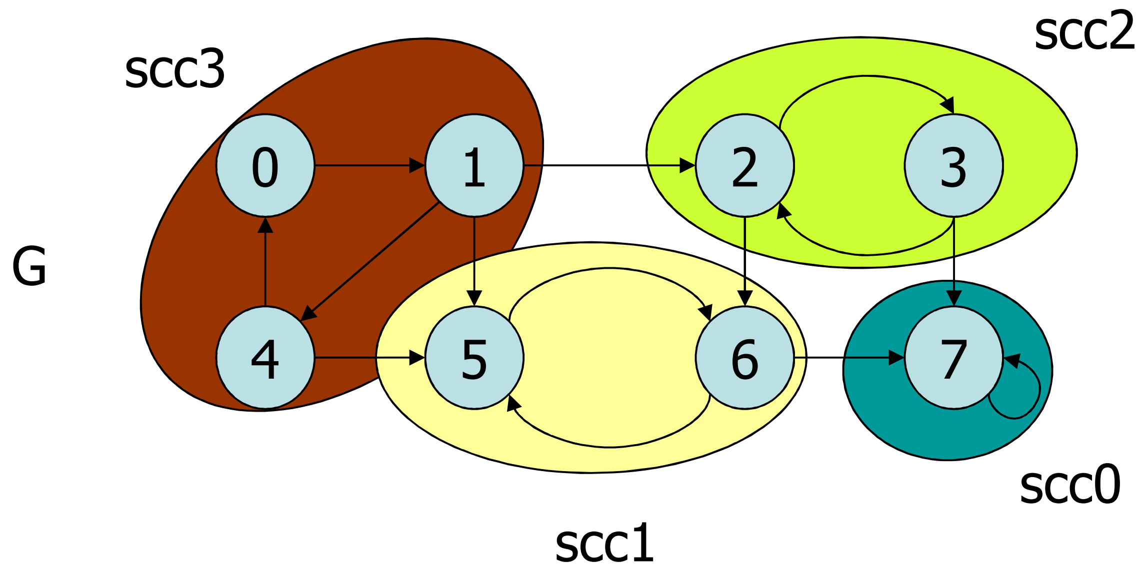
Algorithm

- DFS of the reverse graph G^T (time stamps pre[v] and post[v] are displayed, though only post[v] time stamps are used)



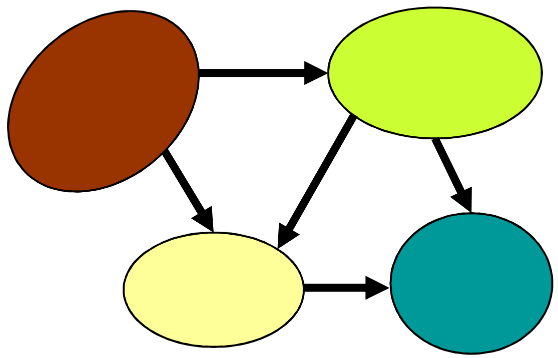
Algorithm

- DFS of the original graph according to decreasing endprocessing times computed in the DFS of the reverse graph G^T



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



Graph library: SCC

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

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

Graph library: SCC

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

  free(post);
  free(tmp);
  return id;
}
```

Graph library: SCC

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

  return t;
}
```