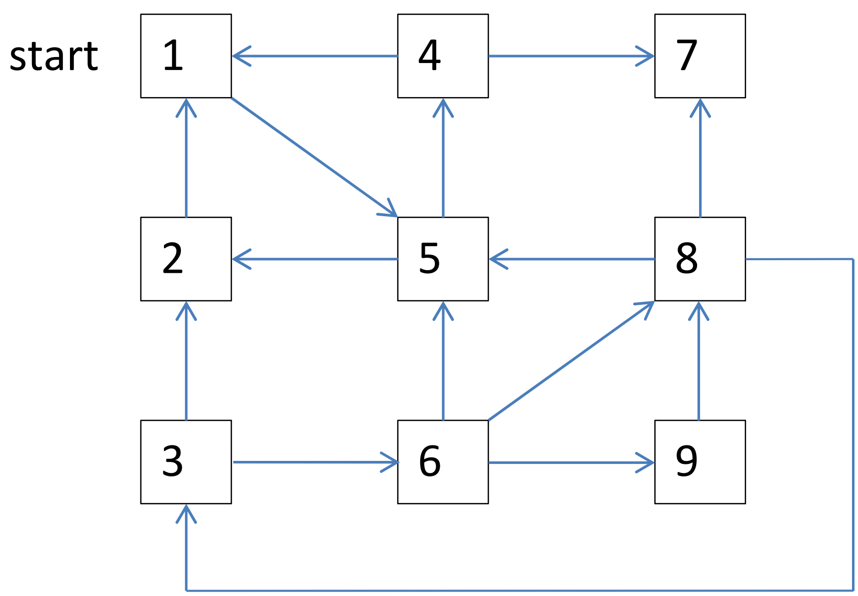
DFS Applications

More applications of Depth-First Search:

- Strong connectedness of directed graph:
  Is the graph strongly connected?
  If not, find all the strongly connected components

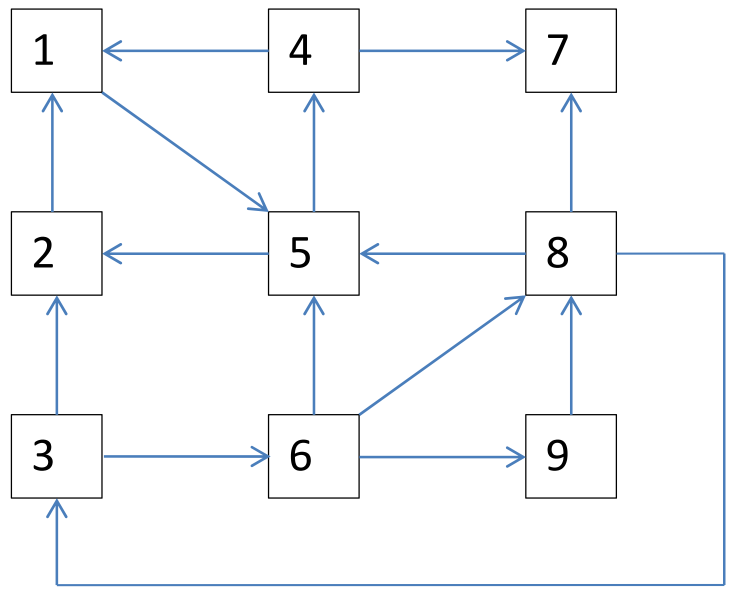


SCC: {1,2,4,5} {7} {3,6,8,9}

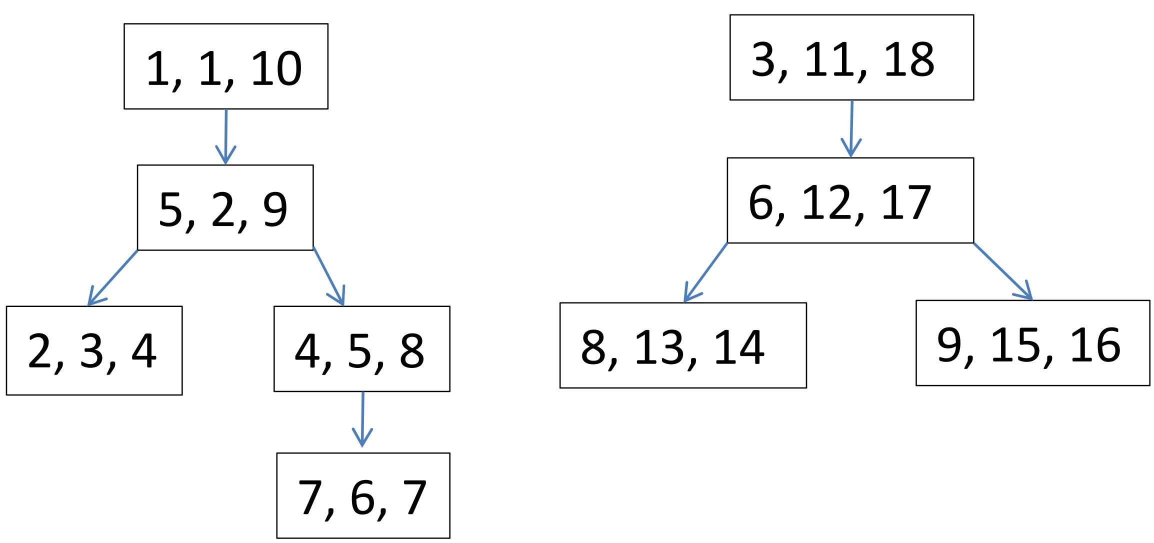
First we modify DFS to compute start and finish timestamps:

```
boolean seen[1…n];
int start[1…n], finish[1…n];            // timestamps
int time = 0;

DFS (Graph G) {
    for (k = 1; k<=n; k++)
        seen[k] = false;
    for (k = 1; k<=n; k++)
        if (! seen[k])
            DFS (G, k);
}

DFS (G, x) {
    seen[x] = true;
    start[x] =  ++ time;                // preVisit(x)
    for each vertex y such that (x,y) is an edge
        if (! seen[y])  {
            //  optionally add edge (x,y) to the DFS tree;
            DFS (G, y);
        }
    finish[x] = ++ time;                // postVisit (x)
}
```

Example:



DFS forest: each node shows (x, start[x], finish[x])



Note: all nodes of each SCC must be in same DFS tree, but all nodes of each DFS tree are not necessarily in same SCC

Kosaraju’s Algorithm for Strongly Connected Components

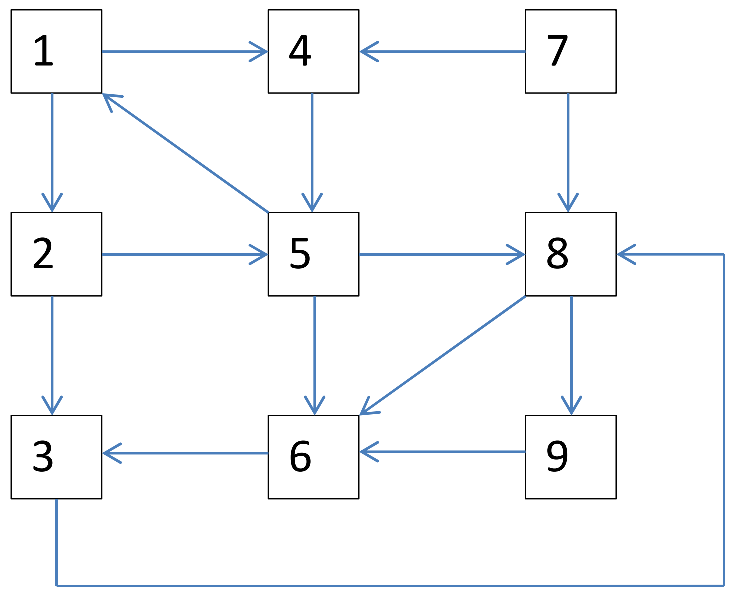
- Run DFS on graph G, compute start and finish timestamps
- Sort vertices in ***descending*** order by finish timestamps, using counting sort or bin sort
- Reverse all the edges of graph G to obtain graph G’
- Run DFS on graph G’, choosing start vertices for each DFS tree in the new sorted order
- Each DFS tree of G’ is a SCC (in both G and G’)

	1	2	3	4	5	6	7	8	9
S	1	3	11	5	2	12	6	13	16
F	10	4	18	8	9	15	7	14	17

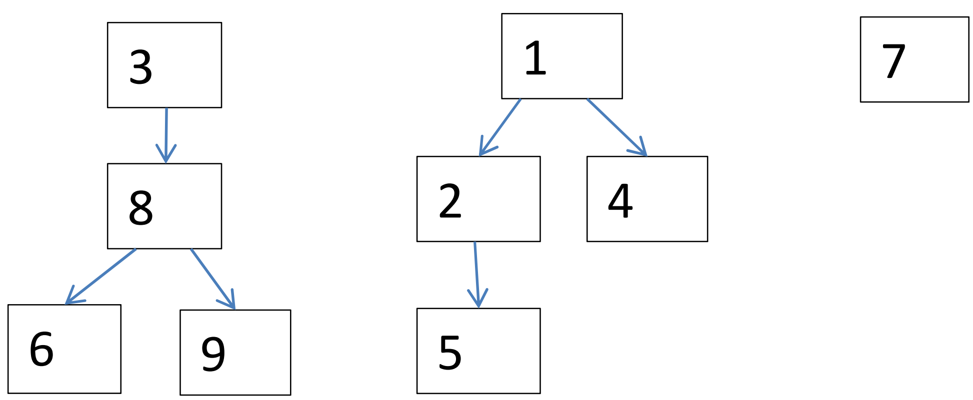
Sort vertices in descending order by finish times:

3, 9, 6, 8, 1, 5, 4, 7, 2

Reverse edges of graph G to obtain graph G':



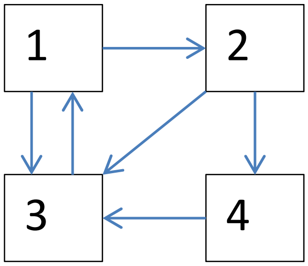
DFS forest on reversed graph G':



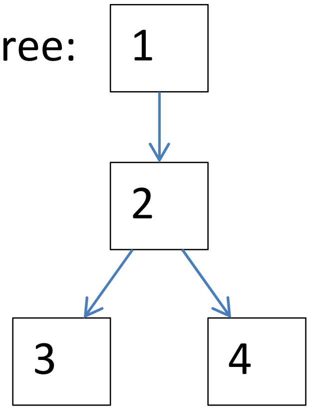
Strongly connected components: {3,6,8,9} {1,2,4,5} {7}

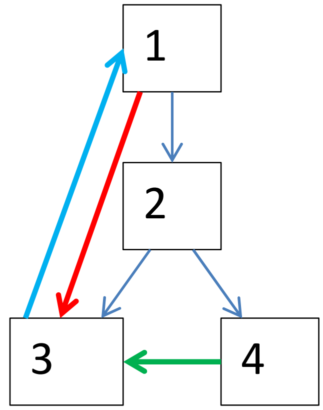
Running time of Kosaraju's algorithm for SCC:
$\theta(n+m)$ time, where n=|V| and m=|E|

Tree edges, Back edges, Forward edges, Cross edges:



DFS tree:





Tree edges: 1→2 and 2→3 and 2→4

Back edge: **3→1** (points from descendant to ancestor)

Forward edge: **1→3** (points from ancestor to descendant, and is not a tree edge)

Cross edge: **4→3** (points from right to left, any edge that's not a tree edge or back edge or forward edge)

Applications of DFS:

- Detecting a cycle (in undirected or directed graph)

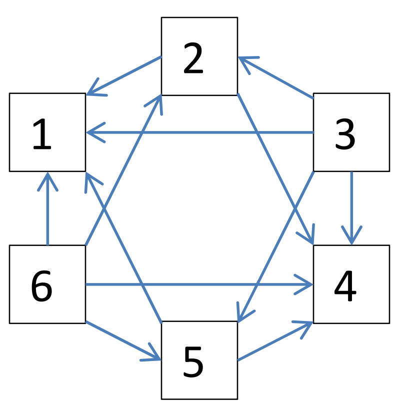
  Check if any back edges are encountered during DFS

  If back edge exists, then it completes a cycle

  If no back edges, then no cycles (graph is acyclic)

- Topological sort of a Directed Acyclic Graph:
  a linear ordering of the vertices so that whenever edge $x \to y$ exists, vertex x must precede vertex y

  Example: this graph is a DAG



Topological sorts (not unique):

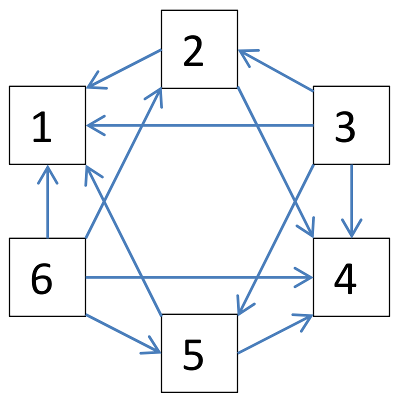
3, 6, 5, 2, 1, 4

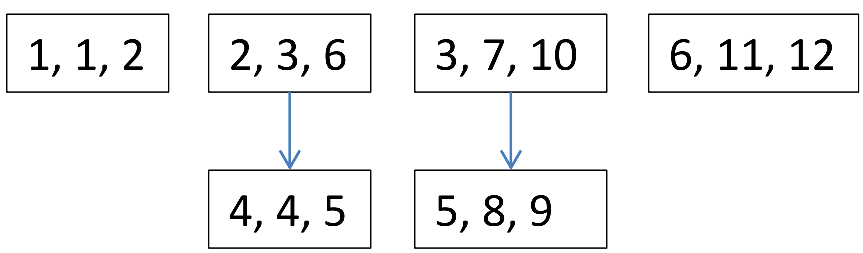
6, 3, 2, 5, 4, 1

also several others

Algorithm for finding a topological sort of a DAG
(same as first two steps of Kosaraju's algorithm for SCC):

- Run DFS on the DAG, compute start and finish timestamps
- Sort vertices in <u>*descending*</u> order by finish timestamps, using counting sort or bin sort





	1	2	3	4	5	6
S	1	3	7	4	8	11
F	2	6	10	5	9	12

Sort vertices in descending order by finish times:

6, 3, 5, 2, 4, 1

This is a topological sort for the given DAG