EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

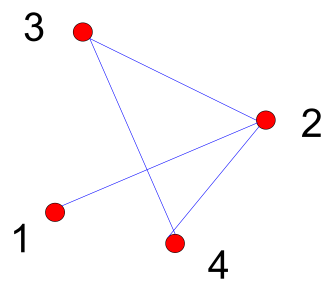
Representations of graphs

- Adjacency matrix:
  - $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

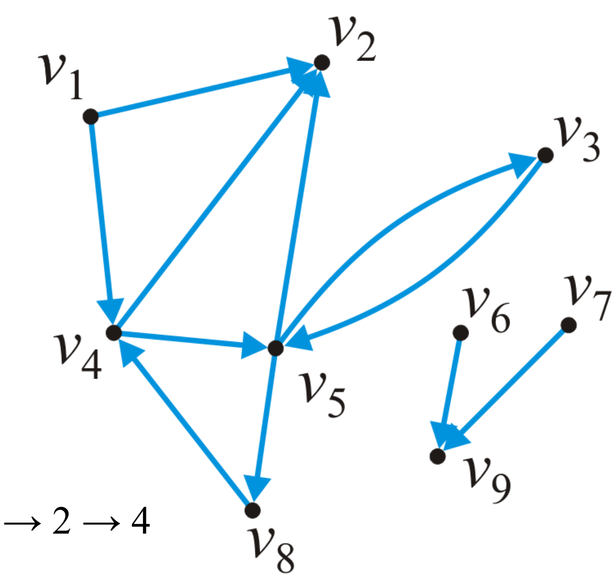
  - Symmetric for undirected graphs

$$A = \begin{array}{cl} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} & \begin{matrix} \cdots 1 \\ \cdots 2 \\ \cdots 3 \\ \cdots 4 \end{matrix} \\ \begin{matrix} \vdots & \vdots & \vdots & \vdots \\ 1 & 2 & 3 & 4 \end{matrix} & \end{array}$$



Adjacency List

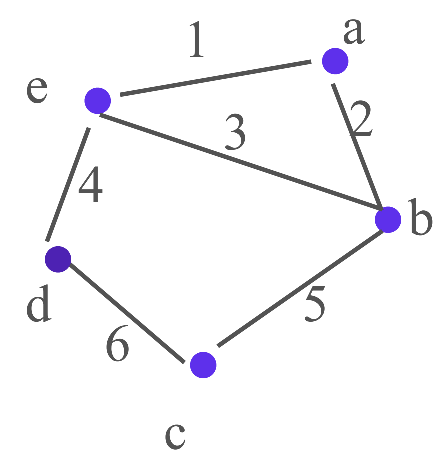
- Each vertex has list of neighbor nodes
  - For directed graphs, out-list
  - Used in MATLAB for sparse matrices
  - Can be implemented as arrays
    - Forward-Star
  - or linked lists
    - |V|+|E| for directed graphs
  - Must store 2 edges for undirected



1 $\bullet \to 2 \to 4$
2 $\bullet$
3 $\bullet \to 5$
4 $\bullet \to 2 \to 5$
5 $\bullet \to 2 \to 3 \to 8$
6 $\bullet \to 9$
7 $\bullet \to 9$
8 $\bullet \to 4$
9 $\bullet$

V vertices
E edges

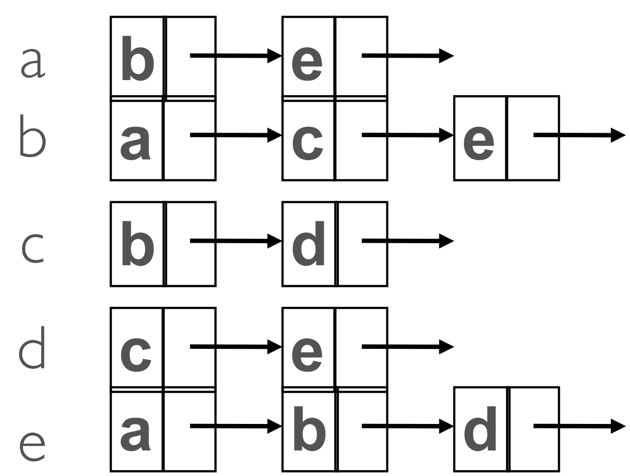
1. Edge list: {a,b}, {b,c}, {c,d},{d,e},{e,a}, {e,b}

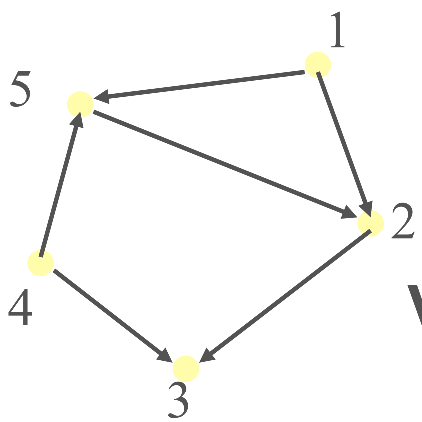


2. Adjacency list

Vertex	Adjacencies
a	b, e
b	a, c, e
c	b, d
d	c, e
e	a, b, d

Linked List Form





Forward Star Representation

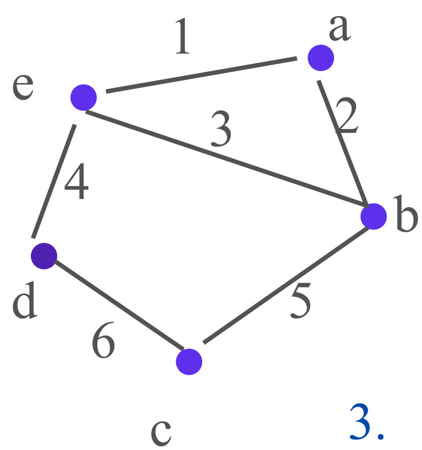
Vertex Array First[I]

i =1	1
i =2	3
i =3	4
i =4	4
i =5	6
i =6	7

Edges[j]

2	j =1
5	j =2
3	j =3
3	j=4
5	j=5
2	j =6
-1	j =7

Just the adjacency list put end to end in the arc array!



3. Adjacency matrix:

A=

	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
d	0	0	1	0	1
e	1	1	0	1	0

4. Incidence matrix:

Matrix of vertex rows, edges columns

M =

	1	2	3	4	5	6
a	1	1	0	0	0	0
b	0	1	1	0	1	0
c	0	0	0	0	1	1
d	0	0	0	1	0	1
e	1	0	1	1	0	0

Directed: -1 in start of edge, 1 in end

Undirected: 1 in both

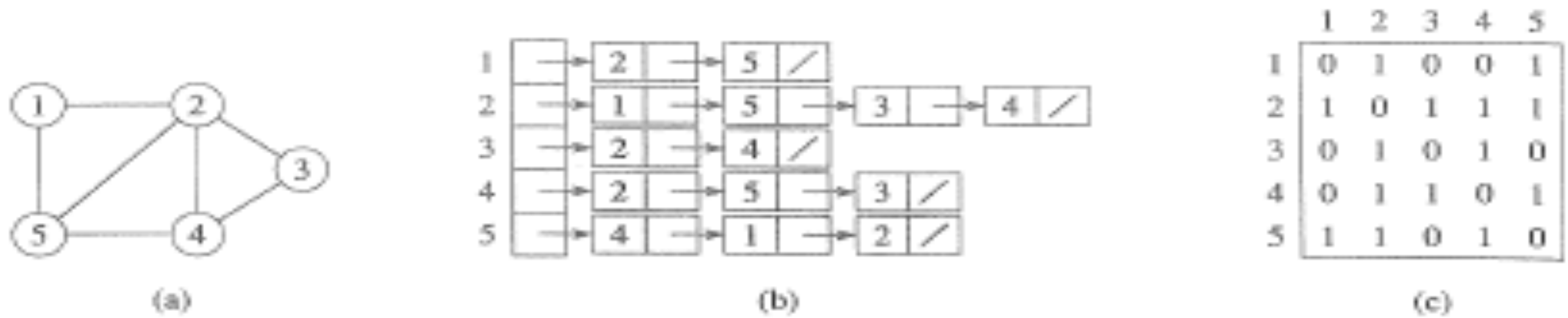


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

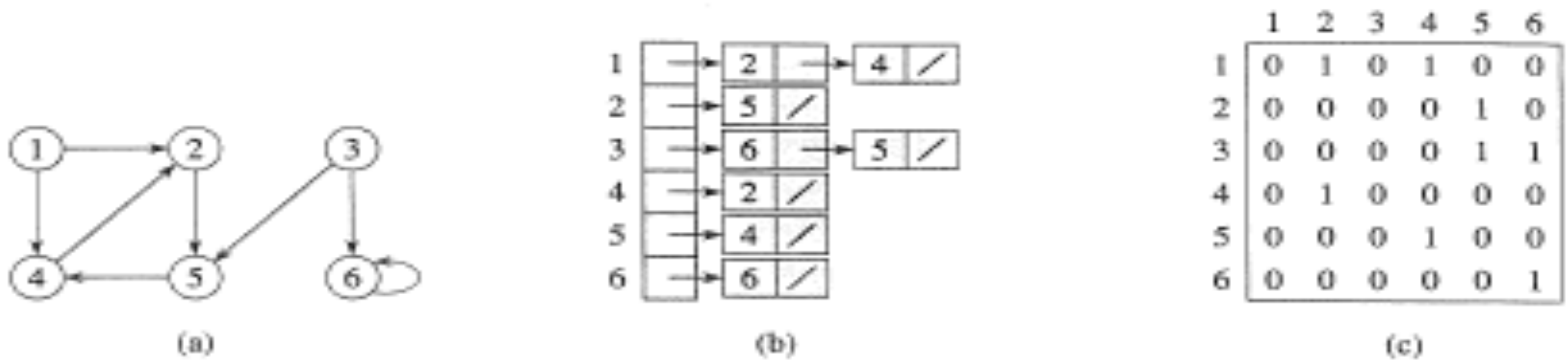


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Graph Traversals

- Traversals of graphs are also called *searches*
- We can use either breadth-first or depth-first traversals
  - Breadth-first requires a queue
  - Depth-first requires a stack
- We each case, we will have to track which vertices have been visited requiring $\Theta(|V|)$ memory
- The time complexity cannot be better than and should not be worse than $\Theta(|V| + |E|)$
  - Connected graphs simplify this to $\Theta(|E|)$
  - Worst case: $\Theta(|V|^2)$

Breadth-First Search

1. Mark all vertices as unvisited, parents as NULL, depth as -1
2. Choose any unvisited vertex, mark it as visited and enqueue it onto queue
3. While the queue is not empty:
    - Dequeue top vertex v from the queue. Do work to be done on that vertex
        - If parent[v] == NULL, set depth to 0; otherwise, set depth to depth[parent[v]] + 1
        - For each vertex adjacent to v (e.g. in out list) that has not been visited: Mark it visited, mark its parent as v, and enqueue it
        - Mark v as done
4. If there are unvisited vertices, choose any unvisited vertex, mark it as visited, enqueue it and repeat step 3

- This can handle graphs that are not connected
    - Marking as visited avoids cycles
    - Complexity: O(#V + #E), reduces to O(#E) if strongly connected
    - Size of queue is O(#V)

Depth-First Search

Recursive implementation:

1. Mark all vertices as unvisited; mark all parents as NULL
2. While there are vertices marked as unvisited:
   - Select unvisited vertex v, mark as visited:
   - Do DFS(vertex)

DFS(vertex):
- For neighbors of vertex
  - If neighbor is unvisited, mark as visited and do DFS(neighbor)

- This can handle graphs that are not strongly connected
  - Marking as visited avoids cycles
  - Complexity: O(#V + #E), reduces to O(#E) if strongly connected
  - Size of queue is O(#V)

Recursive depth-first traversal

A recursive implementation uses the call stack for memory:

```
void Graph::depth_first_traversal( Vertex *first ) const {
    std::unordered_map<Vertex *, int> hash;
    hash.insert( first );

     first->depth_first_traversal( hash );
}

void Vertex::depth_first_traversal( unordered_map<Vertex *, int> &hash ) const {
    // Perform an operation on this

    for ( Vertex *v : adjacent_vertices() ) {
        if ( !hash.member( v ) ) {
            hash.insert( v );
            v->depth_first_traversal( hash );
        }
    }
}
```

Depth-First Search

Iterative implementation: (Similar to Recursive)

1. Mark all vertices as unvisited; mark all parents as NULL
2. While there are vertices marked as unvisited:
   - Choose any unvisited vertex, mark it as visited and push it onto stack
   - While stack is not empty:
     - Let v be the vertex that is on top of stack.
     - If v has no unvisited neighbors, pop v , work on it and mark it done
     - Else select an unvisited neighbor of v, mark neighbor as visited, set parent[neighbor] = v, and push neighbor onto stack

- This can handle graphs that are not strongly connected
  - Marking as visited avoids cycles
  - Complexity: O(#V + #E), reduces to O(#E) if strongly connected
  - Size of queue is O(#V)

Depth-First Search

Alternative Iterative implementation: (Different from Recursive)

1. Mark all vertices as unvisited; mark all parents as NULL
2. While there are vertices marked as unvisited:
   - Choose any unvisited vertex v, mark it as visited and push it onto stack
   - While stack is not empty:
     - Pop v be the vertex that is on top of stack. Work on v. For any unvisited neighbors of v in its out list, mark neighbor as visited, set parent[neighbor] = v, and push it onto stack. Mark v as done

- This can handle graphs that are not strongly connected
  - Marking as visited avoids cycles
  - Complexity: O(#V + #E), reduces to O(#E) if strongly connected
  - Size of queue is O(#V)

Iterative depth-first traversal

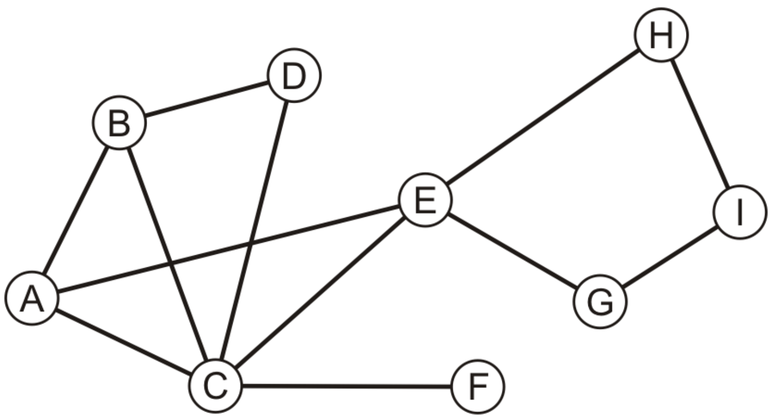
An iterative implementation can use a stack

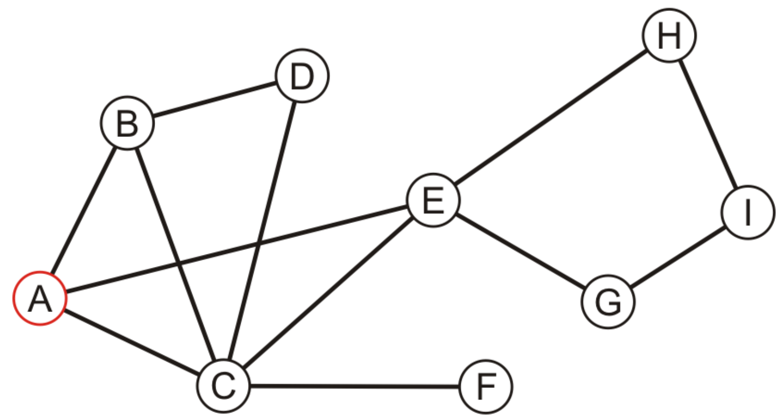
```
void Graph::depth_first_traversal( Vertex *first ) const {
    unordered_map<Vertex *, int> hash;
    hash.insert( first );
    std::stack<Vertex *> stack;
    stack.push( first );

    while ( !stack.empty() ) {
        Vertex *v = stack.top();
        stack.pop();
        // Perform an operation on v

        for ( Vertex *w : v->adjacent_vertices() ) {
            if ( !hash.member( w ) ) {
                hash.insert( w );
                stack.push( w );
            }
        }
    }
}
```

Example: BFS





A					

Queue

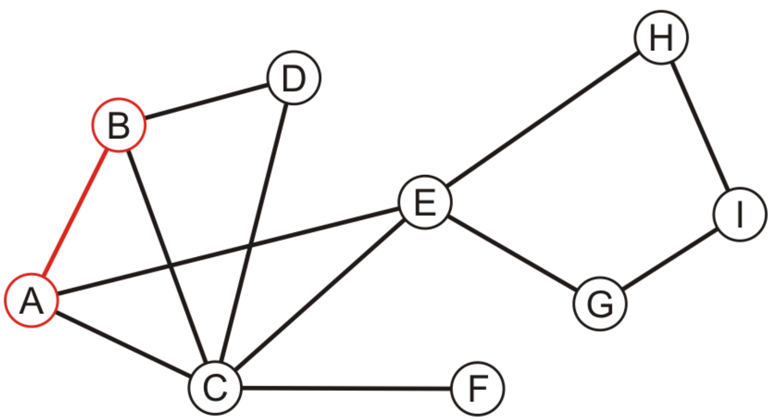
B	C	E			

Example

Performing a breadth-first traversal:

– Pop B and push D

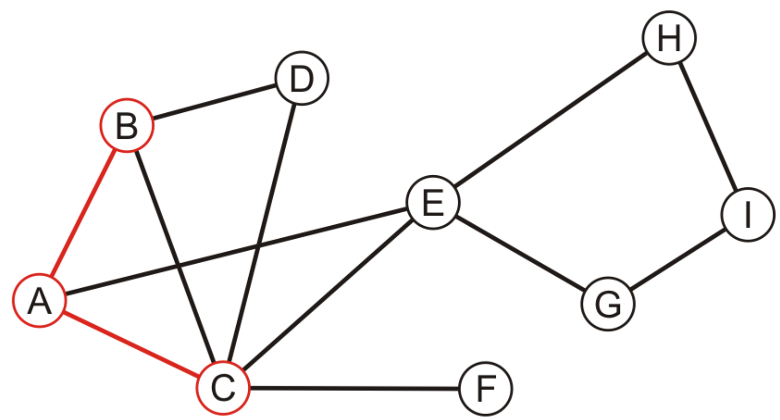
Done: A, B



C	E	D			

Pop C and push F

A, B, C

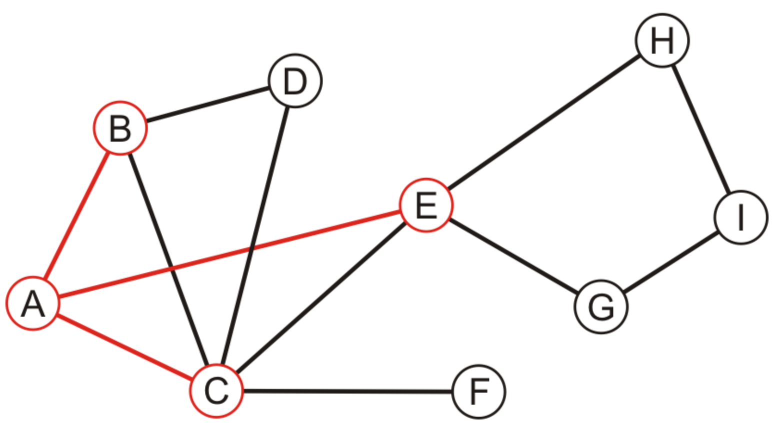


E	D	F			

Example

Performing a breadth-first traversal:

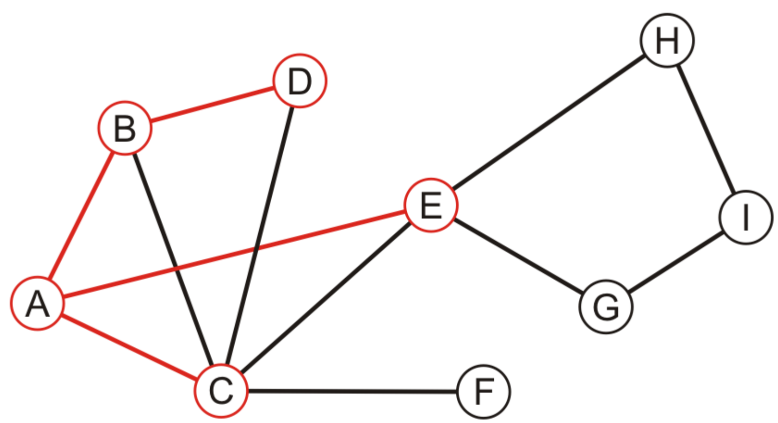
- Pop E and push G,
- Done: A, B, C, E



D	F	G	H		

Pop D

A, B, C, E, D



F	G	H			

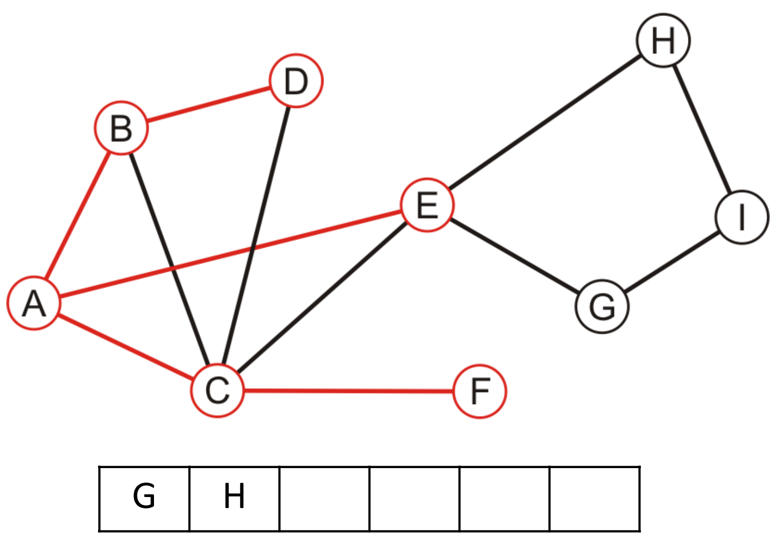
Example

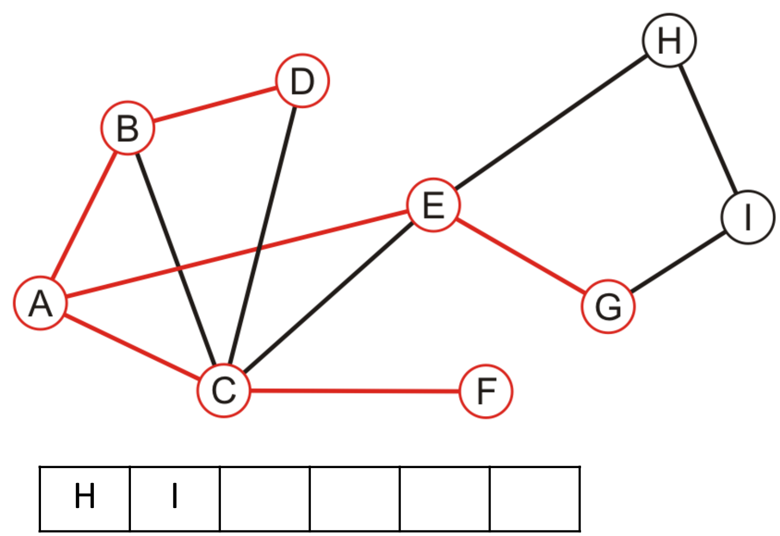
Performing a breadth-first traversal:

- Pop F
- Done: A, B, C, E, D,F

Pop G, push I

A, B, C, E, D, F, G





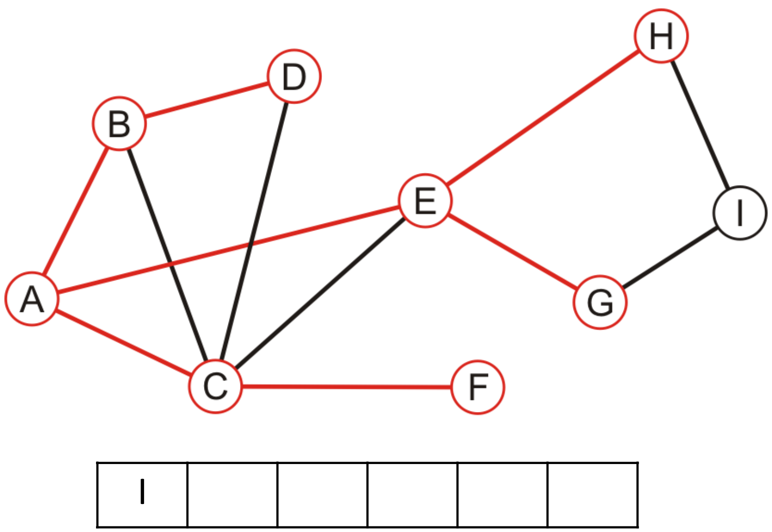
Example

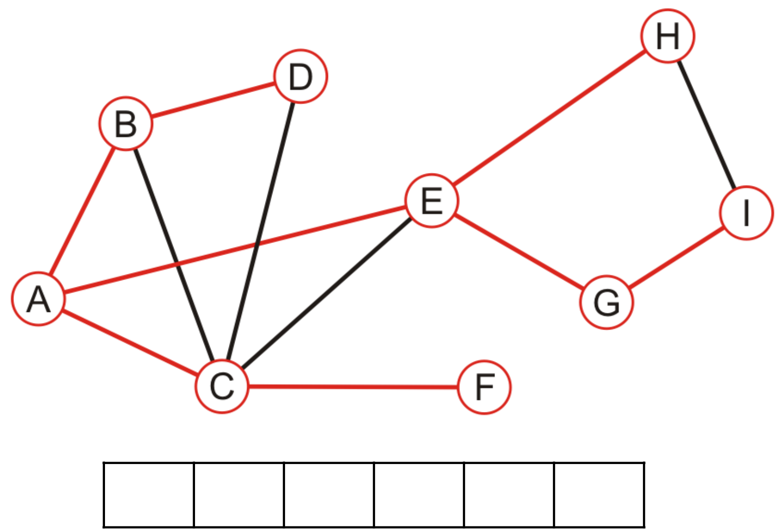
Performing a breadth-first traversal:

- Pop H
- Done: A, B, C, E, D,F, G, H

Pop I

A, B, C, E, D, F, G, H, I





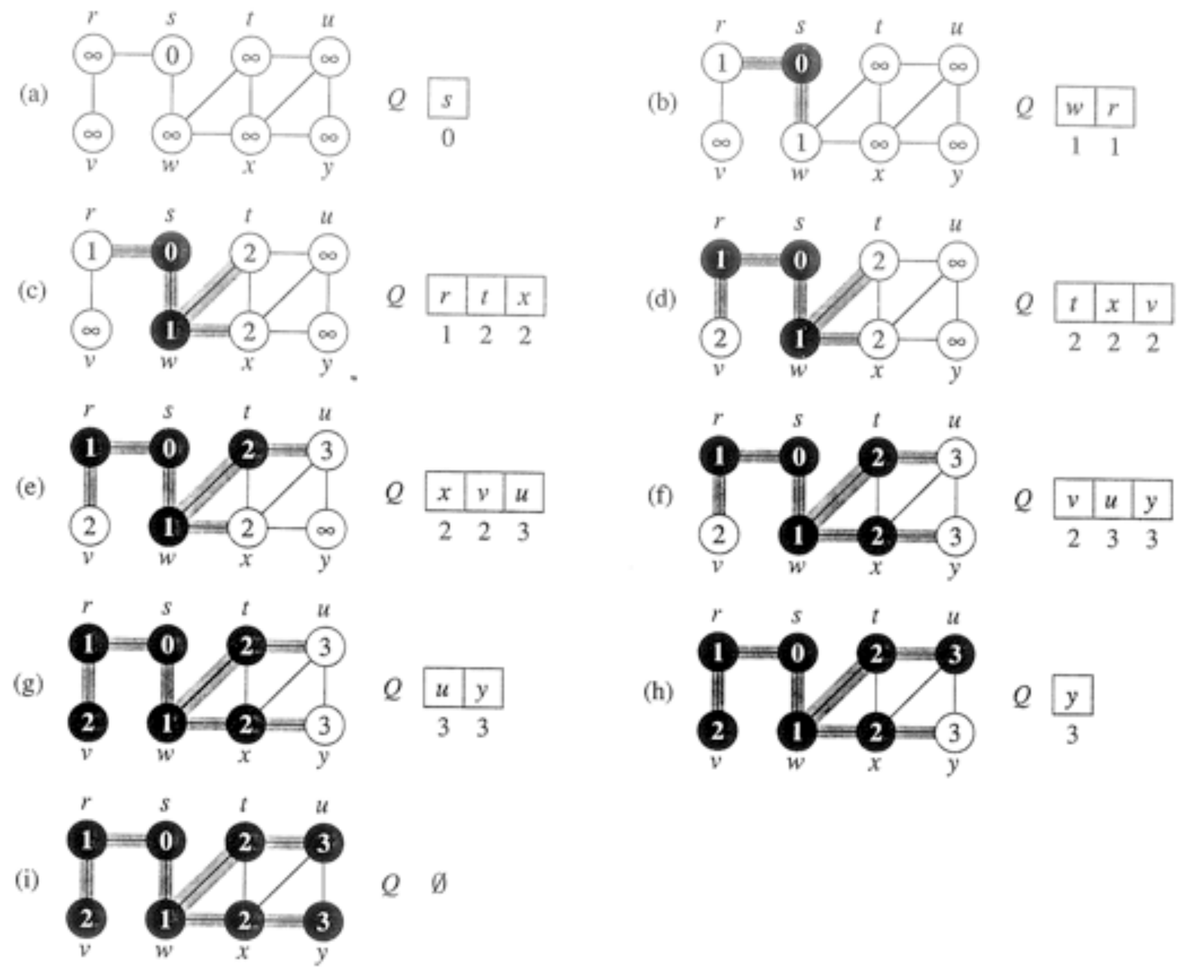


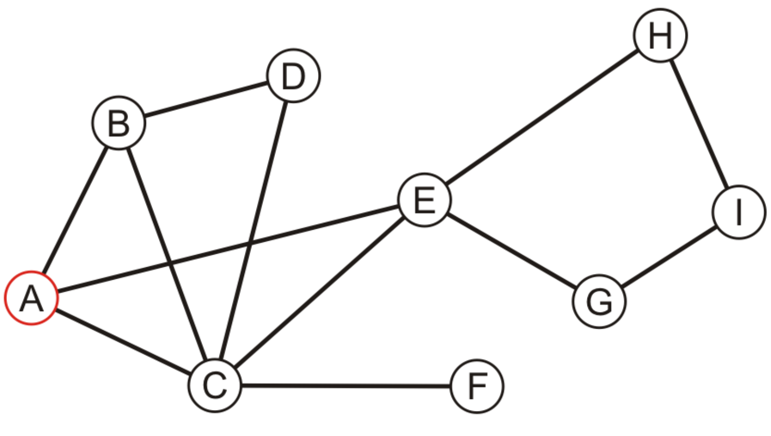
Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

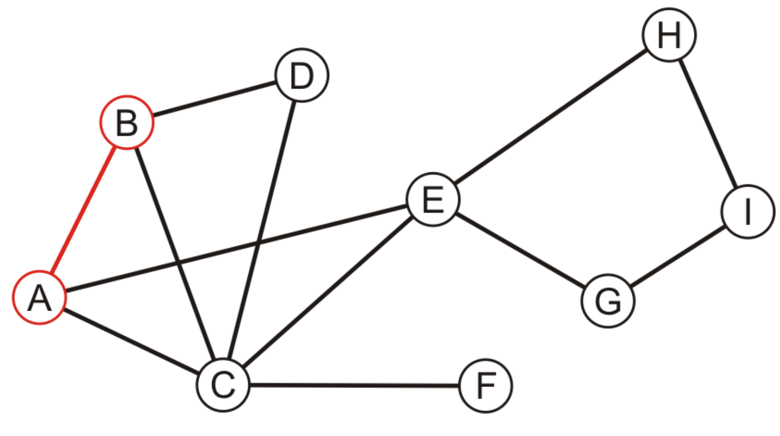
Example

Performing a recursive depth-first traversal:

– Insert A: Visited: A, B
Stack: A, B

Examine B: Visited A, B, C,
Stack A, B, C

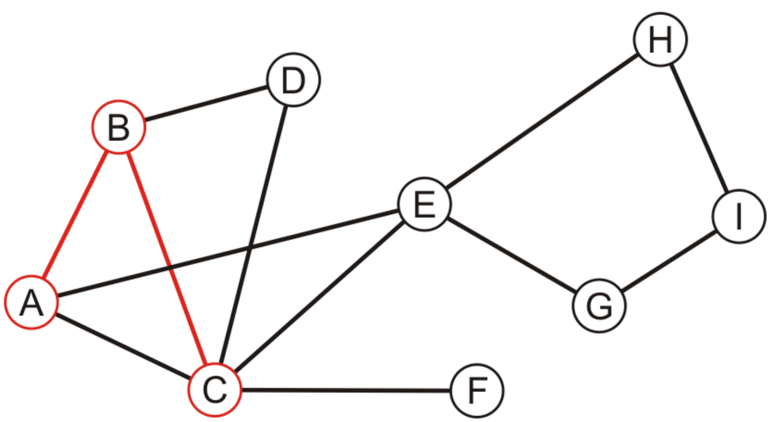




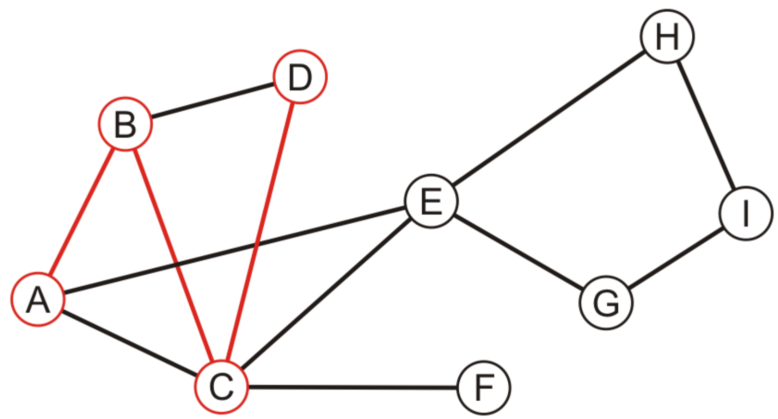
Example

Performing a recursive depth-first traversal:

– Examine C: Visited: A, B,C,D
  Stack: A,B,C,D



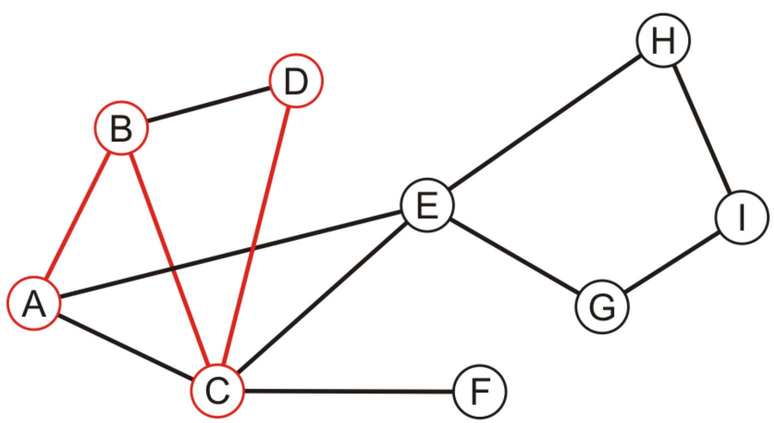
Pop D: Visited A, B, C, D
Stack A,B,C



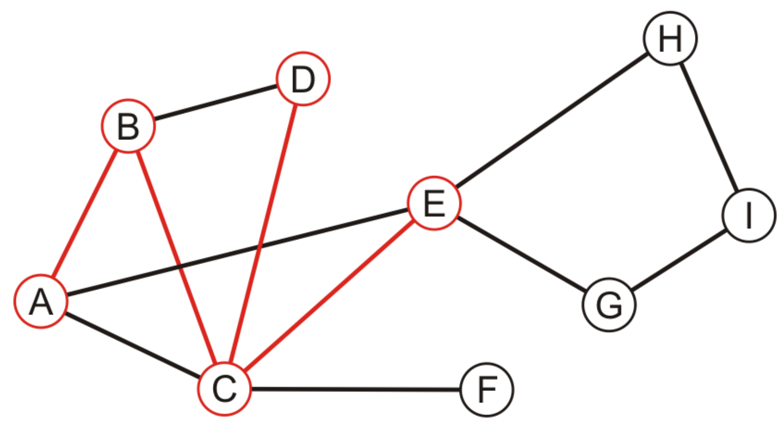
Example

Performing a recursive depth-first traversal:

- Examine C: Visited: A,B,C,D,E
  Stack: A,B,C,E



Examine E: Visited A,B,C,D,E,G
Stack A,B,C,E,G

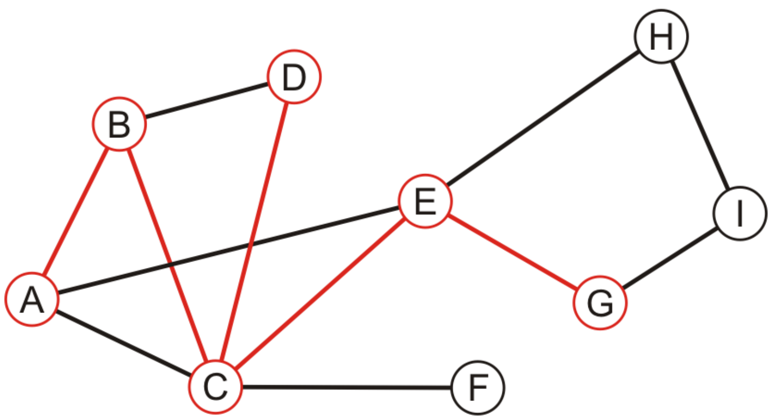


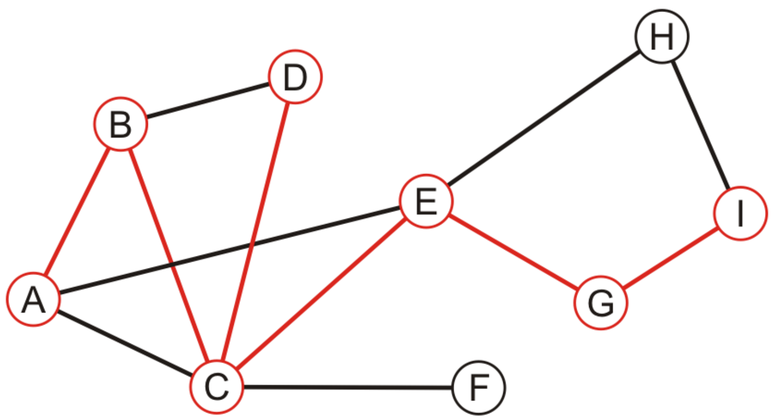
Example

Performing a recursive depth-first traversal:

– Examine G: Visited: A,B,C,D,E,G,I
  Stack: A,B,C,E,G,I

Examine I: Visited A,B,C,D,E,G,I,H
Stack A,B,C,E,G,I,H





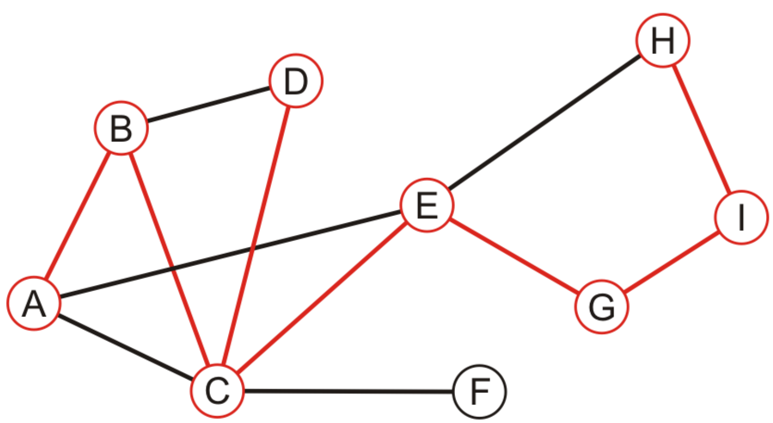
Example

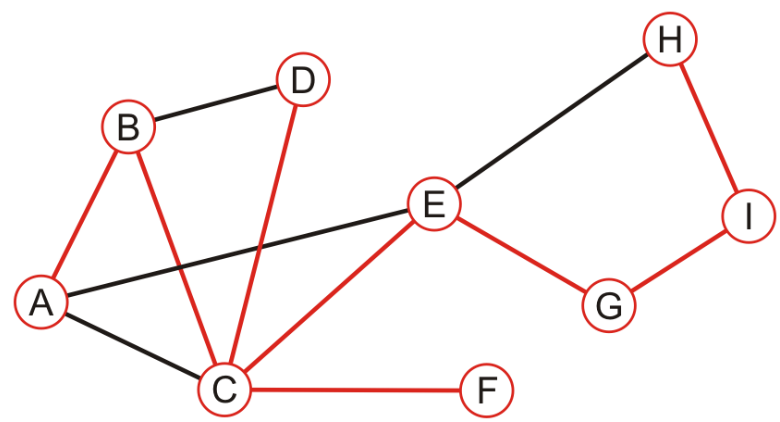
Performing a recursive depth-first traversal:

- Pop H: Visited: A,B,C,D,E,G,I,H
  Stack: A,B,C,E,G,I
  Pop G, then E
  Inspect C: Visited A,B,C,D,E,G,I,H,F
  Stack: A,B,C,F

  Pop I: Visited A,B,C,D,E,G,I,H
  Stack A,B,C,E,G,

  Pop F, C, B, A. Done.



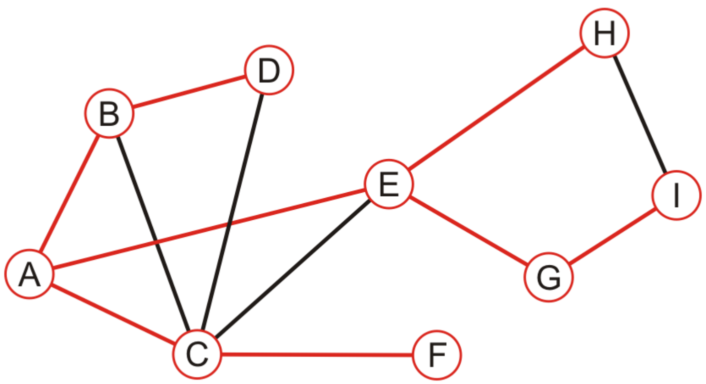


Comparison

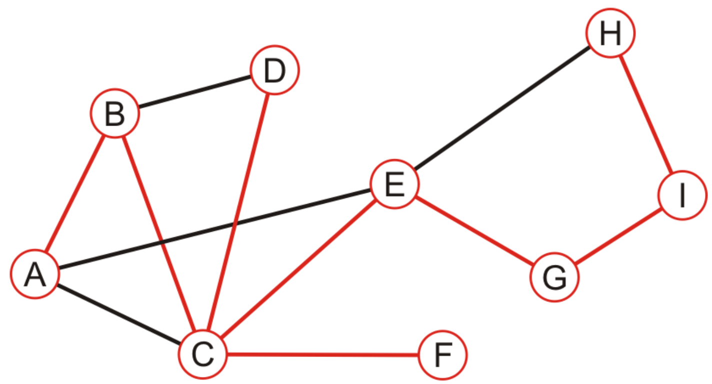
The order in which vertices can differ greatly

– An iterative depth-first traversal may also be different again

A, B, C, E, D, F, G, H, I



A, B, C, D, E, G, I, H, F



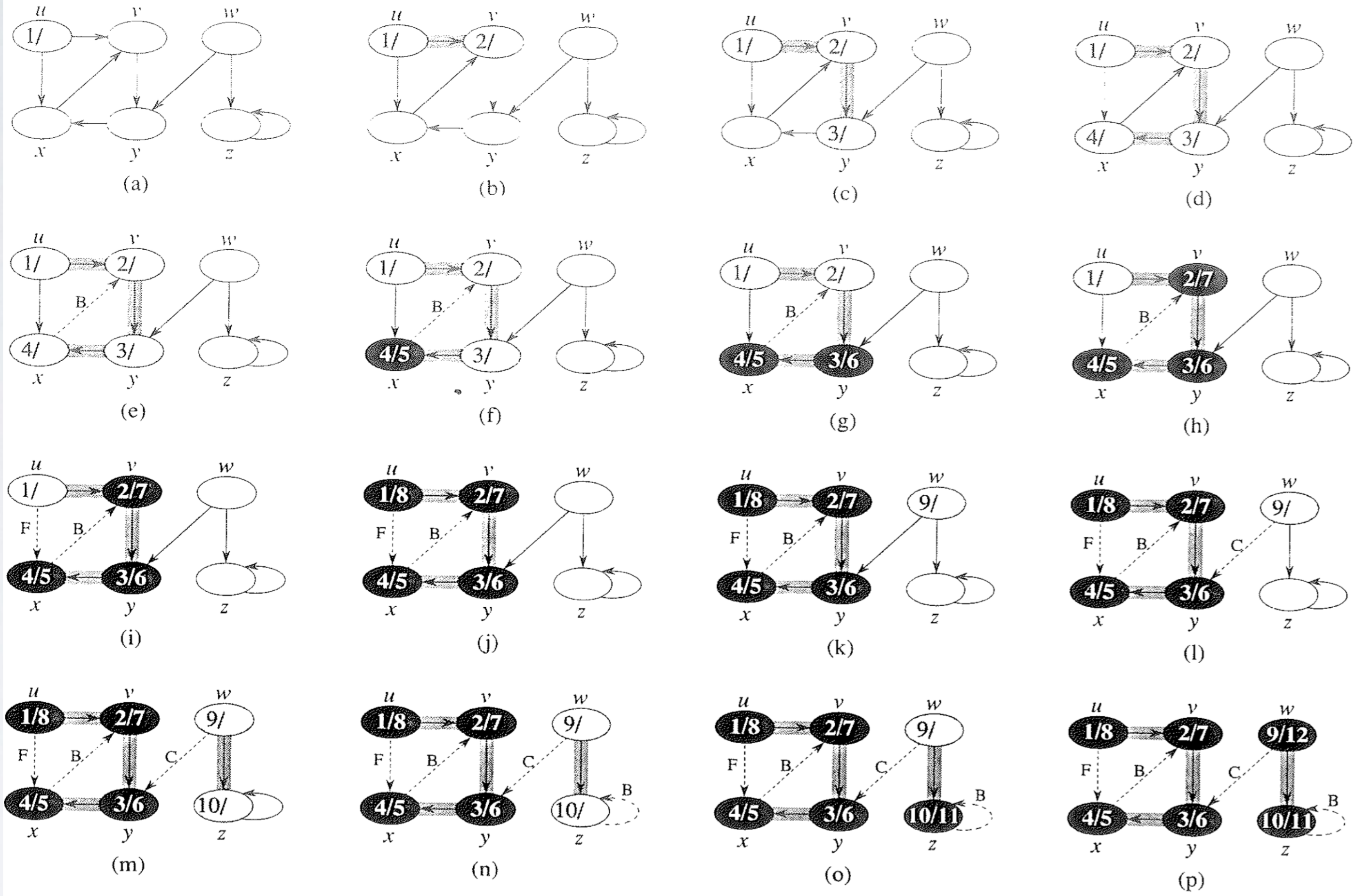


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

Applications

Applications of tree traversals include:

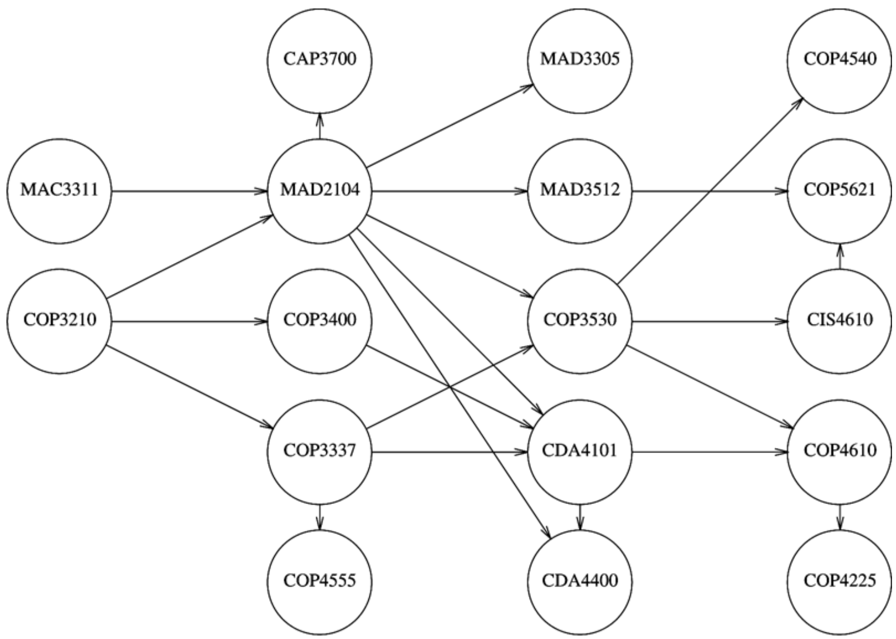
- Determining connectedness and finding connected sub-graphs
- Determining the path length from one vertex to all others
- Testing if a graph is bipartite
- Branch and bound search
- Topological Sort
- …

Topological Sort

- Application
  - Given a number of tasks, there are often a number of constraints between the tasks:
    - task A must be completed before task B can start
  - These tasks together with the constraints form a directed acyclic graph
  - A topological sort of the graph gives an order in which the tasks can be scheduled while still satisfying the constraints

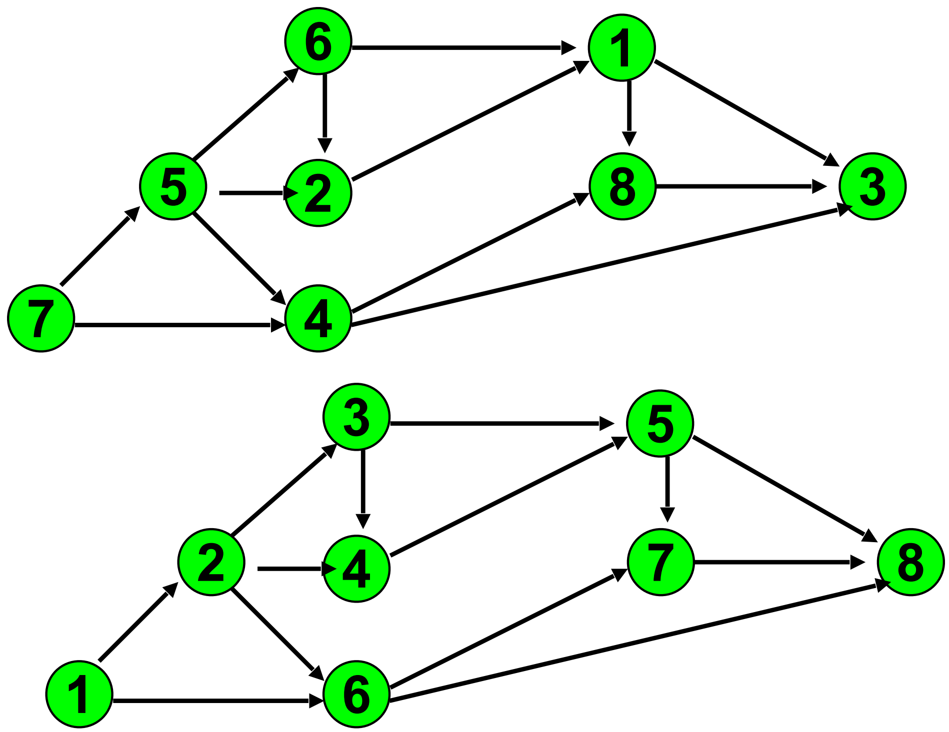
Topological Sort

- Course prerequisite structure represented in an acyclic graph



Topological Sort

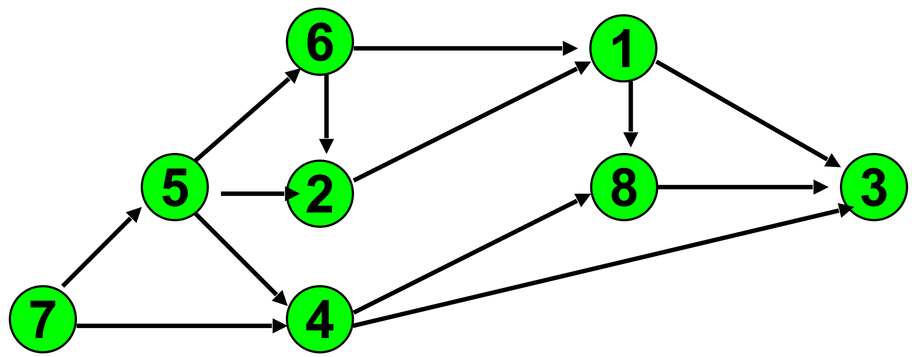
- Topological Sort (Relabel): Given a directed acyclic graph (DAG), relabel the vertices such that every directed edge points from a lower-numbered vertex to a higher numbered one



Algorithm Using Priority

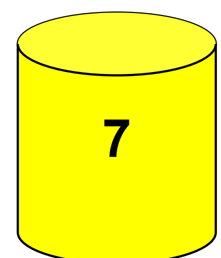
Determine the indegree of each vertex

List is set of vertices with indegree 0



next 0

Vertex	1	2	3	4	5	6	7	8
Indegree	2	2	3	2	1	1	0	2



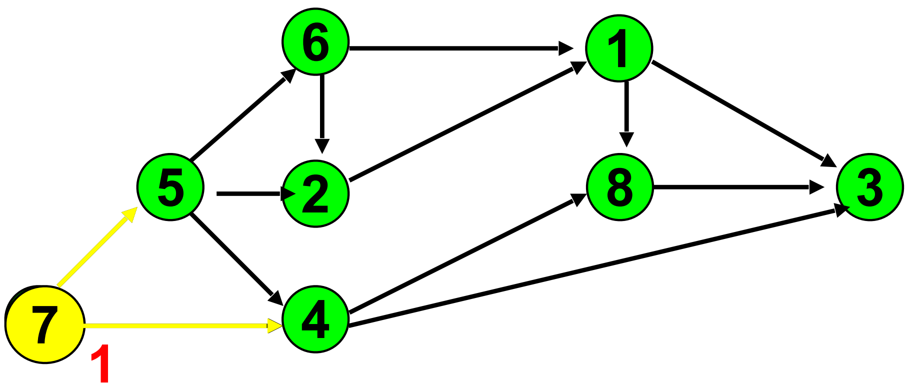
Heap

Select a node from LIST

Select a vertex from LIST and delete it.

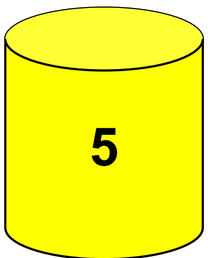
For all vertices in outlist of vertex, reduce in degree by 1

update LIST



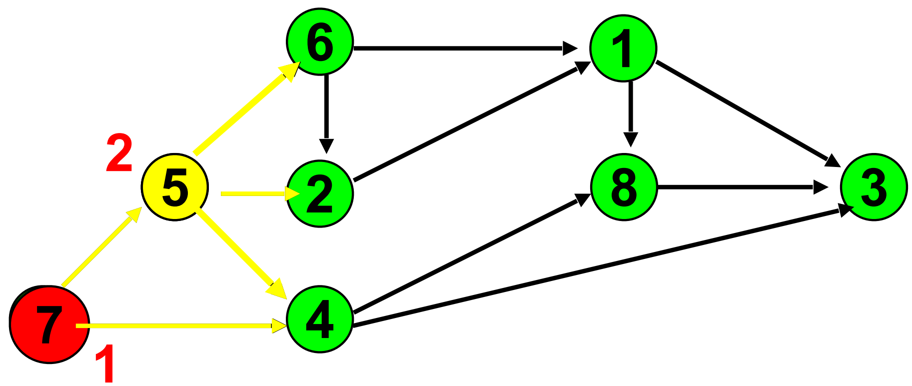
next 1

Node	1	2	3	4	5	6		8
Indegree	2	2	3	1	0	1		2



LIST

Delete 5



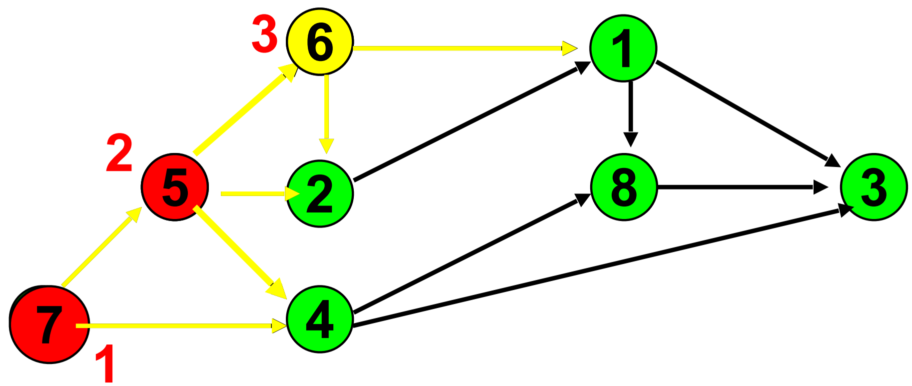
next 2

Node	1	2	3	4	6	8
Indegree	2	1	3	0	0	2

4
6

LIST

Delete 6



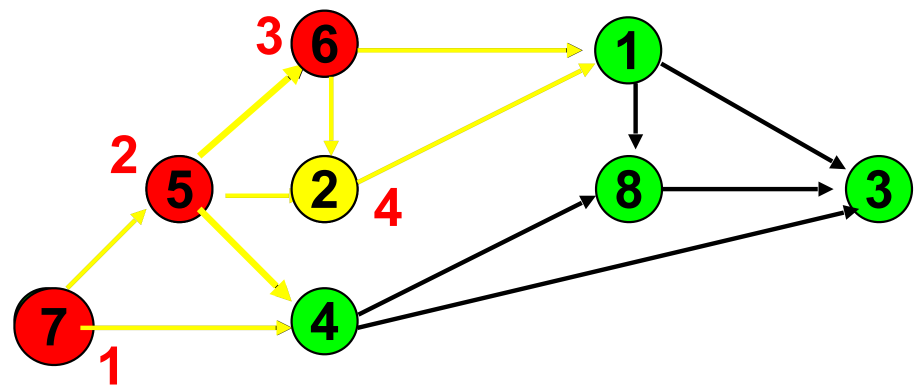
next 3

Node	1	2	3	4	8
Indegree	1	0	3	0	2

4
2

LIST

Delete 2



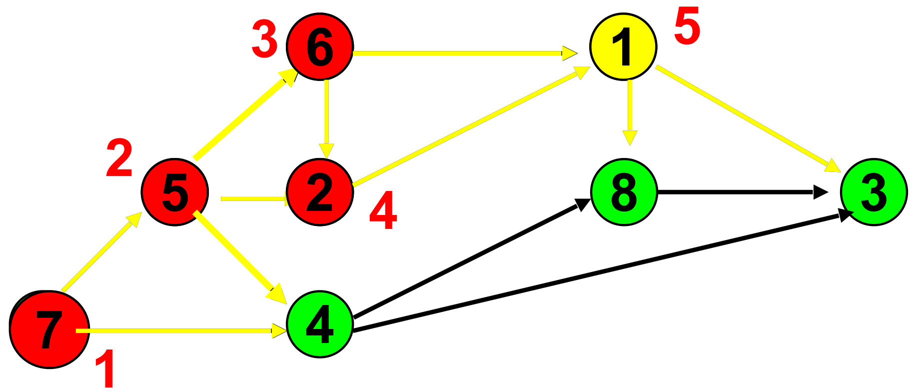
next 4

Node	1	3	4	8
Indegree	0	3	0	2

4
1

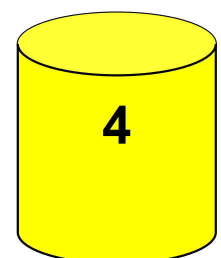
LIST

Delete 1



next **5**

Node	3	4	8
Indegree	2	0	1



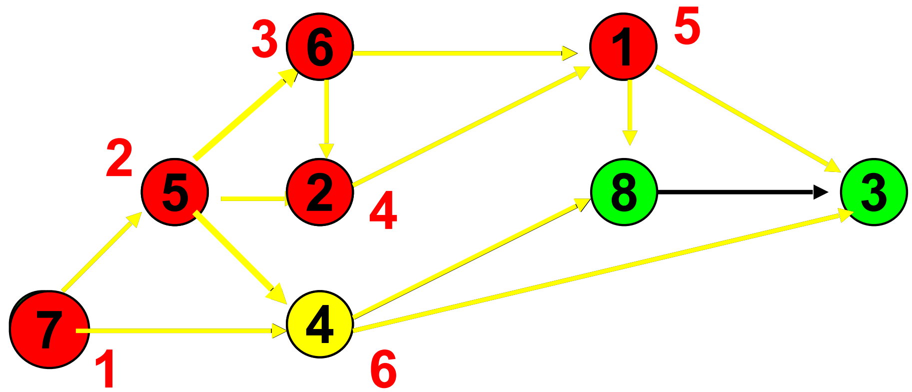
LIST

Delete 4

next := next +1
order(i) := next;

update indegrees

update LIST



next 6

Node	3	8
Indegree	1	0

8

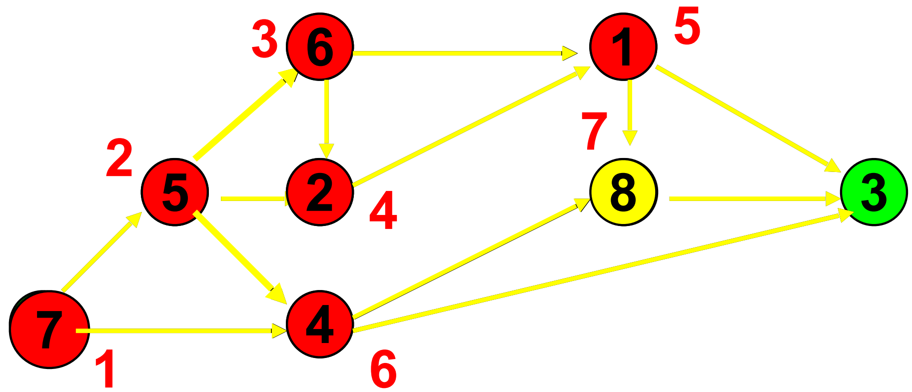
LIST

Delete 8

next := next +1
order(i) := next;

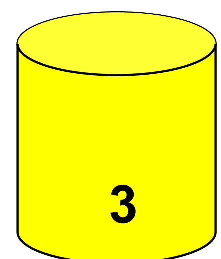
update
indegrees

update LIST



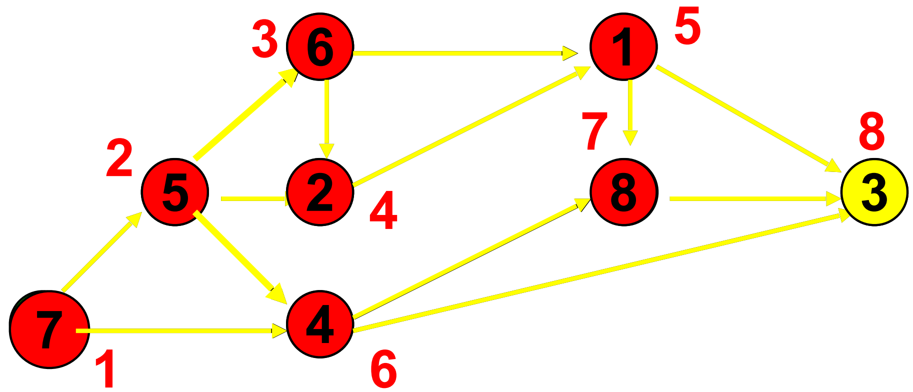
next 7

Node	3
Indegree	0



LIST

Delete 3



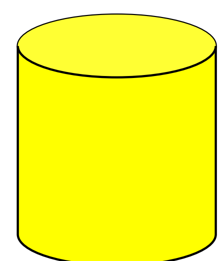
next 8

List is empty.

The algorithm terminates with a topological order of the nodes

Node

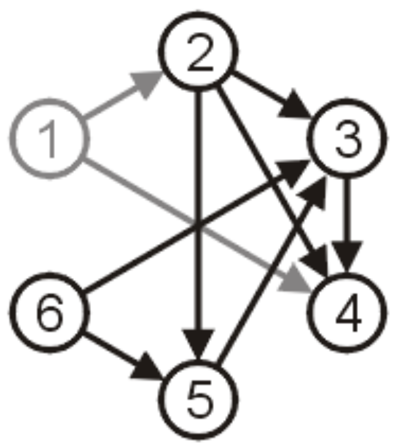
Indegree



LIST

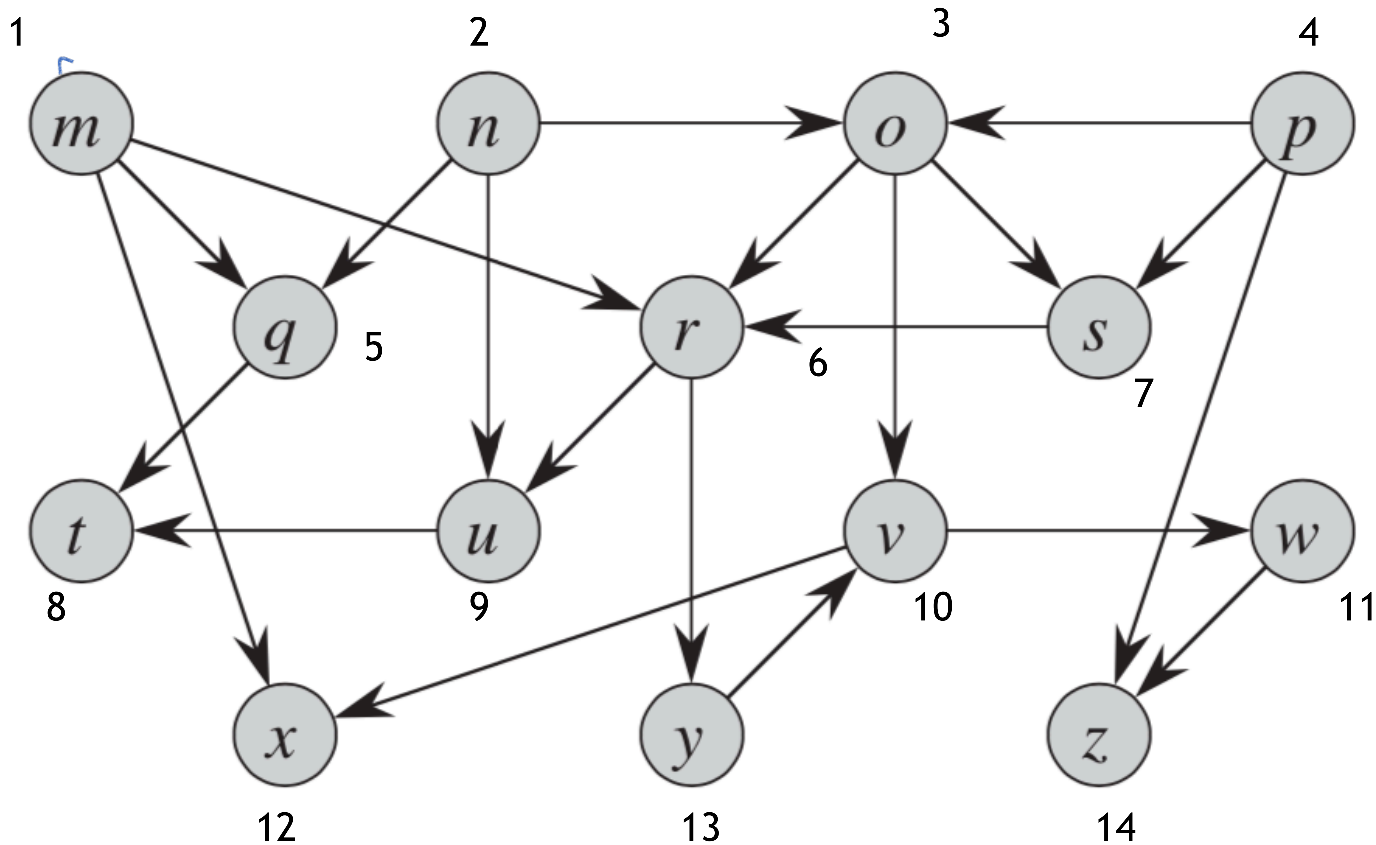
Complexity of Topological Sort

- Computing indegrees: O(#E)
  - Recursively updating indegrees: One operation per edge, O(#E)
  - So, algorithm is O(#E), using an array of O(#V)
- Alternative algorithm: **recursive** DFS (O(#E))
  - Order in which vertices are completed is reverse order of a topological sort!



Stack: 1 2 4 Completed: 4
Stack: 1 2 3 Completed: 3
Stack: 1 2 5 Completed: 5
Stack: 1 2 Completed: 2
Stack: 1 Completed: 1
Stack: 6 Completed: 6
Reverse Order: 6,1,2,5,3,4

DAG Topological Sort (Execution order)



Biconnectivity

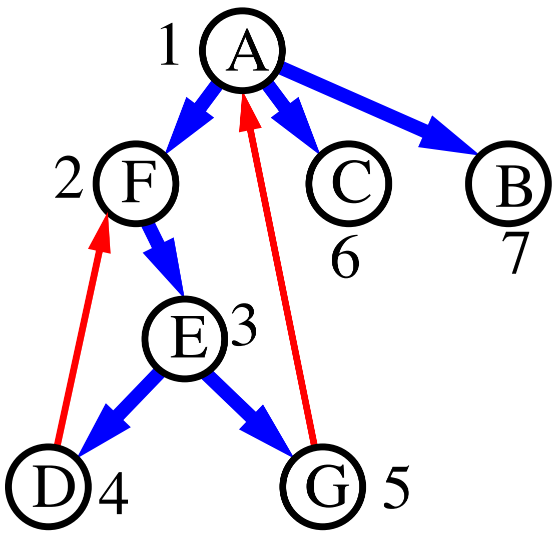
- Connected **undirected** graph is **biconnected** if there are no nodes whose removal disconnects the graph
- Nodes whose removal disconnect the graph are known as **articulation points**
- DFS can be used to find articulation points:
  - Algorithm: Number nodes in Depth First Search order, in the order in which they are inserted into the execution stack of the recursive Depth-First Search. This creates a spanning tree in the graph. Call this number NUM(n) for node n

Biconnectivity

- Algorithm:
  - Number vertices in DFS order, in the order in which they are marked as visited in DFS. Call this number NUM(n) for vertex n
  - For each vertex n, compute the lowest numbered vertex which is reachable from node n by following down the tree 0 or more steps and using a single edge which is not on the tree. Call that number LOW(n)
  - Find articulation points as follows:
    - a root of a DFS tree is an articulation point if and only if it has more than one child.
    - Any vertex n is an articulation point if and only if there is a child m of n in the DFS tree such that LOW(m) $\geq$ NUM(n)

Biconnectivity

- For each vertex n, compute the lowest numbered vertex which is reachable from node n by following down the tree 0 or more steps and using a single edge which is not on the tree. Call that number LOW(n)

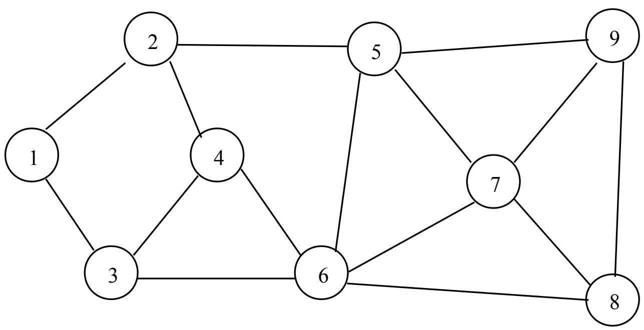


Low(A) = 1
Low(B) = 7
Low(C) = 6
Low(F) = 1 (path F-E-G-A)
Low(E) = 1
Low(D) = 2 (can't use 2 back edges)
Low(G) = 1

Biconnectivity

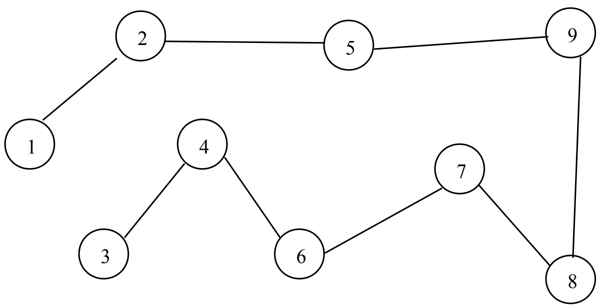
- Why does this work?
  - **Property** 1: The lowest numbered vertex reachable from n with one backward arc must be an ancestor of n in DFS tree
    - DFS algorithm guarantees that edges not included in the spanning tree cannot cross between branches.
  - **Property** 2: If, for all children n1 of n in DFS tree, LOW(n1) < NUM(n), then there is path from each n1 to an ancestor of n which bypasses n (because it is not in the DFS tree) ==> removing n leaves its ancestors and successors connected, so n cannot be an articulation point
  - **Property** 3: If there is a child n1 of n for which LOW(n1) >= NUM(n), then there is no path from n1 to any ancestor of node n which does not go through n ==> removing n disconnects n1 ==> n is articulation point

Example



NUM(1) = 1 LOW(1) = 1
NUM(2) = 2 LOW(2) = 1
NUM(3) = 9 LOW(3) = 1
NUM(4) = 8 LOW(4) = 1
NUM(5) = 3 LOW(5) = 1
NUM(6) = 7 LOW(6) = 1
NUM(7) = 6 LOW(7) = 1
NUM(8) = 5 LOW(8) = 1
NUM(9) = 4 LOW(9) = 1

DFS tree



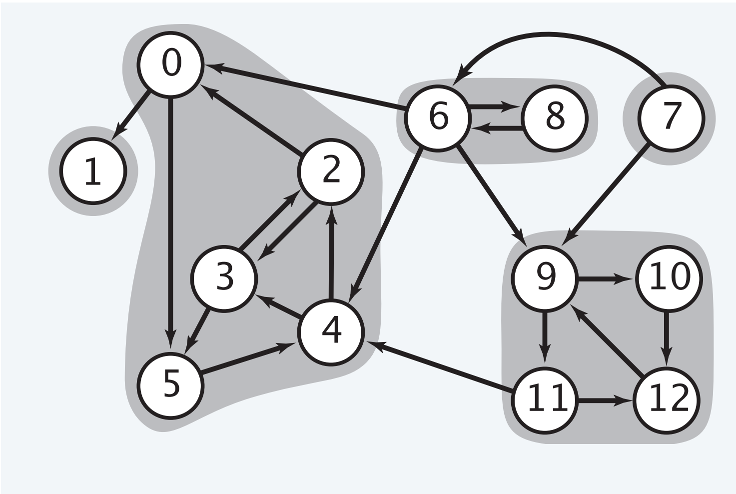
NO ARTICULATION POINTS

Strong Connectivity

- A graph is strongly connected if every vertex can be reached from every other vertex
  - Would like to detect if a graph is strongly connected
- Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.
- Algorithm: Pick any s.
  - Run BFS from s and verify all vertices can be reached
  - Reverse all edges in E, run BFS and verify all vertices can be reached
  - Complexity O(#E + #V) (2 BFS).

Strongly-Connected Components

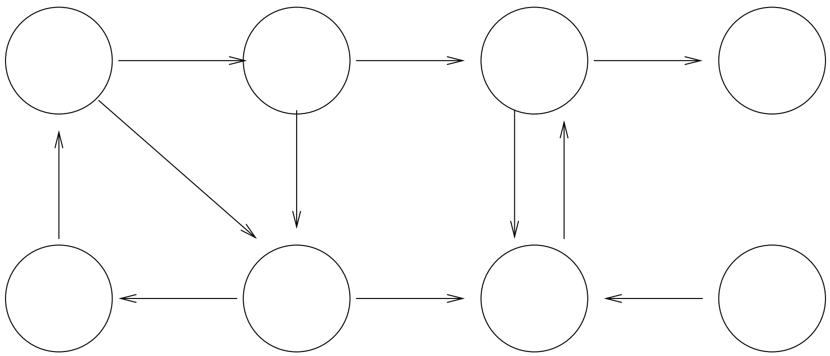
- A graph is strongly connected if every vertex can be reached from every other vertex
  - A strongly-connected component of a graph is a subgraph that is strongly connected
  - Would like to identify strongly-connected components of a graph
  - Can be used to identify weaknesses in a network
  - General approach: Perform two DFSs

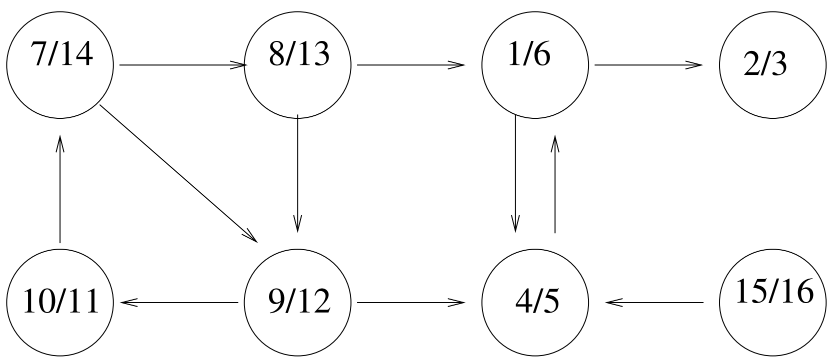


Strongly-Connected Components

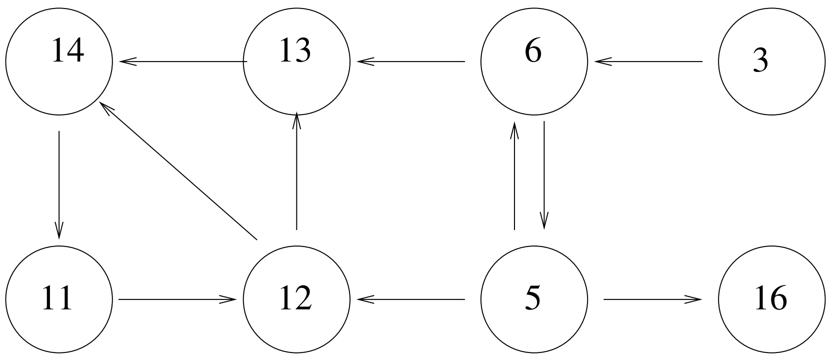
- Kosaraju's Algorithm
  - Perform DFS on graph G = (V, E),
    - Number vertices according to their finishing time in DFS of G
  - Perform DFS on Gr = (V,Er), where Er are reverse of edges in E, selecting nodes in decreasing order of finishing time in previous DFS
  - Strongly connected components = reachable trees obtained in last DFS

Example

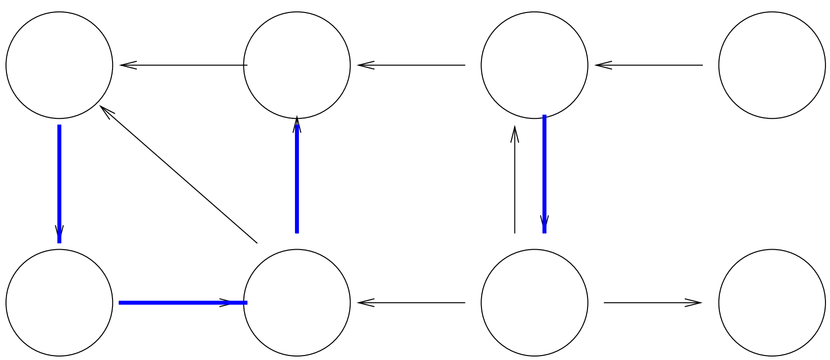




Example



Reverse graph with distance labels



Reverse graph reachable trees

Strongly-Connected Components

- Correctness
  - If v and w are in a strongly-connected component
  - Then there is a path from v to w and a path from w to v
  - Therefore, there will also be a path between v and w in G and Gr

- Running time
  - Two executions of DFS
  - O(|E|+|V|)