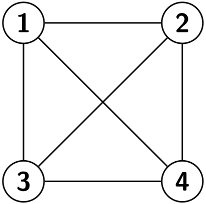
Definition of a Graph

- A graph G consists of a pair of sets V, E denoted by $G = (V, E)$.
- V: vertex set.
  - Each vertex $v \in V$ may represent some records, objects or a piece of information.
- E: edge set.
  - Each edge $e \in E$ links (relates) one pair of distinct vertices $u \neq v \in V$.
  - There is at most one edge which relates two distinct vertices.

Definition of a Graph

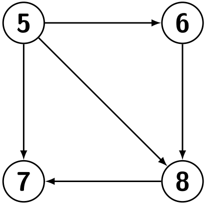
- G is undirected if each edge represents an unordered pair, i.e., $e = (u, v) = (v, u)$.
- In an undirected graph, E may define upto $\binom{|V|}{2}$ relations among vertices.
- If $(u, v) \neq (v, u)$, then the edges are said to be directed:
  - The edge (u, v) is oriented from u to v.
  - The edge (v, u) is oriented from v to u.
- When edges in a graph G are directed, G is known as directed.
- A directed graph may have upto $|V|(|V| - 1)$ edges.

Examples of Graphs



Undirected graph

$$V = \{1, 2, 3, 4\} \text{ and}$$
$$E = \{(1,2), (1,3), (1,4), (2,3), (3,4)\}$$



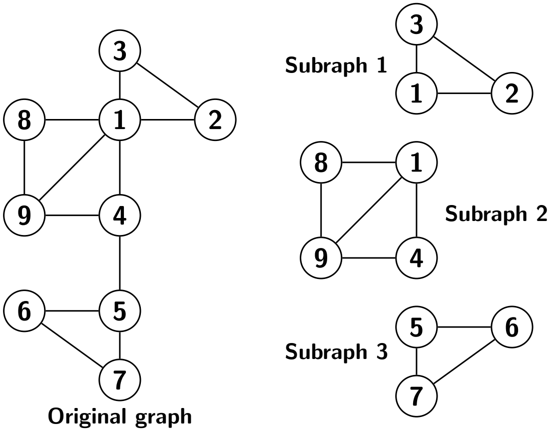
Directed graph

$$V = \{1, 2, 3, 4\} \text{ and}$$
$$E = \{(5,6), (5,7), (5,8), (6,8), (8,6)\}$$

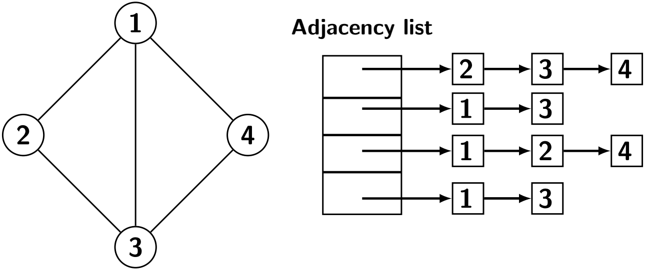
Graph Terminology

- A graph $H = (V_H, E_H)$ is a subgraph of $G = (V_G, E_G)$, if $V_H \subseteq V_G$ and $E_H \subseteq E_G$.
- A simple path in a graph is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ where $(v_i, v_i + 1) \in E$, for $1 \leq i \leq k - 1$.
- A cycle is a simple path in which the start and end vertices are same, i.e., $v_1 = v_k$.
- G is connected if there is a path between any two pair of distinct vertices in G.
- A connected component of a graph G is a maximally connected subgraph of G
- A graph which does not have any cycle is called acyclic.
- An acyclic undirected graph is a tree.

Examples of Subgraph

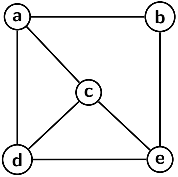


Adjacency List Representation



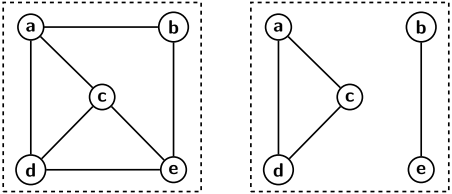
- Each list represents adjacency relations corresponding to a vertex.

Degrees of Vertices



- $V = \{a, b, c, d, e\}$
- $E = \{(a, b), (a, b), (a, d), (b, e),$
  $(c, d), (c, e), (d, e)\}$
- Degree of a vertex v: # edges incident on v.
- $deg(a) = 3$, $deg(b) = 2$, $deg(c) = 3$, $deg(d) = 3$, $deg(e) = 3$,
- # of odd degree vertices is even.

Connectedness of Graphs



Connected graph

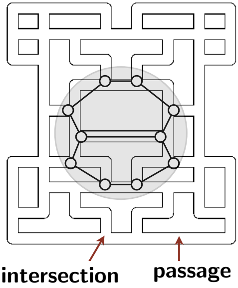
Disconnected graph

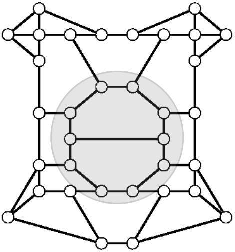
- Connected graphs: $\exists$ a path between any two vertices.
- Disconnected graphs: Having more than one connected subgraphs.

Applications of Graph

- Tremaux was obsessed with problem of finding path out of a maze.
- He came up with technique as follows:
  - Unroll a ball of thread to trace of path that is already traversed.
  - Mark each intersection by putting a mark (color).
  - Retrace back to recent most intersection when no new visit options are present.

Maze to Graph





From Chapter 4 of Robert Sedgewick and Kevin Wayne's Algorithm book.

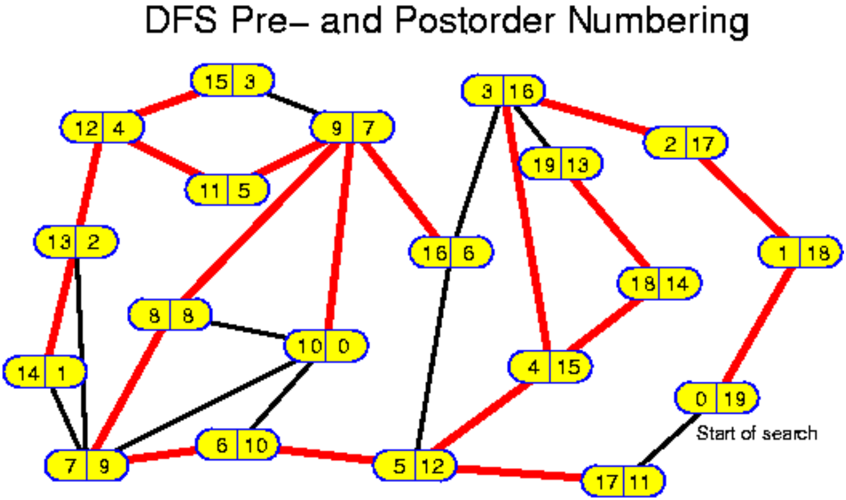
Depth First Search

- Basic form of processing graphs is traversal.
- DFS and BFS are two important traversal techniques.

```
// Initializations
index = 0;
for all (v ∈ V) {
    mark[v] = "unvisited";
    T = Φ; // Tree edges
}
choose(s); // Start vertex
DFS(s, G);
```

```
procedure DFS(G, v) {
    mark[v] = "visited";
    dfn[v] = ++index; // DFS numbers
    for all (w ∈ ADJ_G(v)) {
        if (mark[w]=="unvisited") {
            T = T ∪ {(v, w)}; // Update T
            DFS(G, w); // Recursive call
        }
    }
}
```

Depth First Search Example



Correctness of DFS

Lemma

DFS procedure is called exactly once for each vertex.

Proof.

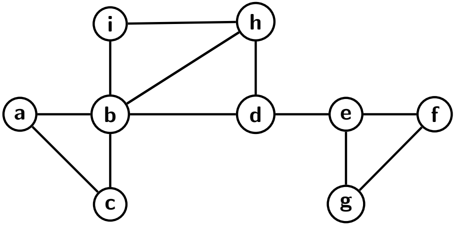
- Once DFS is called for a particular vertex v, it is marked as ”visited”.
- DFS is never called out on ”visited” vertices.

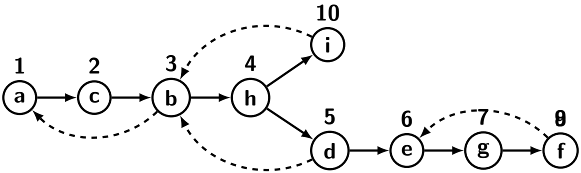
$\square$

Classification of Edges

- DFS gives an orientation to edges of an undirected graph.
- Traversing some edges lead to unvisited vertices.
- While the remaining edges lead to visited vertices.
- If a vertex w is found visited during DFS(v), then w must be an ancestor of v in the DFS tree.
  - DFS(v) must have been called during the time DFS(w) call itself.
  - In other words, DFS(w) is still incomplete when DFS(v) was called.
- So edges are classified into two types: tree edges, and back edges.

Edge Types in DFS of Undirected Graphs

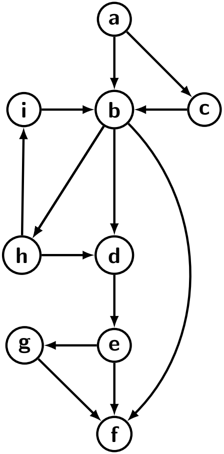


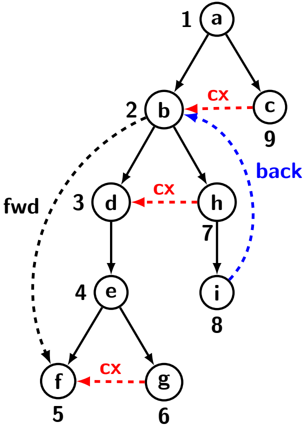


DFS of Directed Graphs

- DFS of directed graphs must explore the edges by respecting the direction of orientation of the edges.
- As usual, tree edges are those edges that always lead to new (unvisited) vertices.
- Remaining edges are partitioned into three other types.
  - Back edges: which lead from a descendant to an ancestor.
  - Forward edges: which lead from a proper ancestor to a descendant
  - Cross edges: connects two unrelated vertices w and v. If the orientation is $v \to w$, then v is visited aftr w.

Edge Types in DFS of Directed Graphs





DFS of Disconnected Graphs

- Algorithm we have presented works for connected graph.
- For DFS of disconnected graphs, we need to change initial calling of procedure a bit.

```
// Initializations
index = 0;
for all (v ∈ V) {
    mark[v] = "unvisited";
    T = Φ; // Tree edges
}
for all (v ∈ V) {
     if (mark[v] == "unvisited")
          DFS(v, G);
}
```

Iterative DFS

- Use a stack to allow for backtracking during DFS.
- Initialize stack by placing a start vertex v.
- As long as stack is nonempty pop the last vertex and mark it visited if it s not visited.
- Then push all the other end vertices of the edges incident on the current vertex.

Iterative DFS

```
IterativeDFS(G, v) {
// Initialization
index = 0;
T = Φ;
makeNull(S);   // Define an empty stack
for all (v ∈ V)
    mark[v] = unvisited;
choose(s); // Start vertex
S.push(s);
// Remaining part in next slide
}
```

Iterative DFS

```
while (!isEmpty(S)) {
    v = S.pop();
    if (marked[v] == "unvisited") {
        mark[v] = "visited";
        dfn[v] = ++index;
        for all w ∈ ADJ_G(v) {
            if (marked[w]=="unvisited") {
                S.push(w);
            }
        }
    }
}
```

Iterative DFS

- There may be multiple copies of vertices on the stack.
- But the total number of iterations of stack loop cannot exceed number edges.
- Thus the size of the stack cannot exceed $|E|$.
- Try out how you can avoid having multiple copies a vertex in the stack.

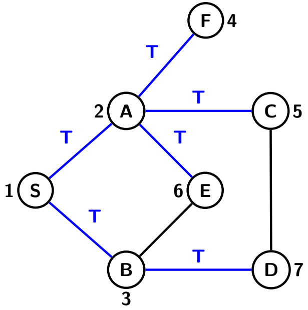
Breadth First Search

```
// Initialization
index = 0;
T = Φ;
Q = NULL;
for all (v ∈ V)
    mark[v] = "unvisited";
choose(s); // Start vertex
bfn[s] = ++index;
ENQUEUE(Q, s);
// Remaining part in next slide
```

Breadth First Search

```
while (!isEmpty(Q)) {
    v = DEQUEUE(Q);
    mark[v] = "visited";
    for all (w ∈ ADJ_G(v)) {
        if (mark[w] == "unvisited") {
            mark[w] = "visited";
            T = T ∪ {(v, w)};
            bfn[v] = ++index;
            ENQUEUE(Q, w);
        }
    }
}
```

Breadth First Search



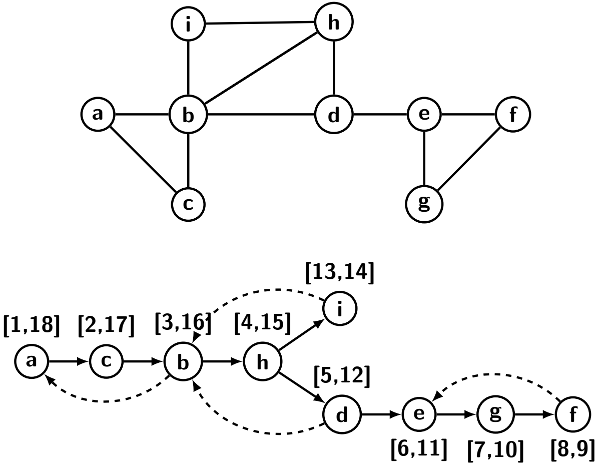
v	w	Action	Queue
-	-	bfn(S) = 1	{S}
S	A	bfn(A) = 1	{A}
	B	bfn(B) = 2	{A,B}
A	F	bfn(F) = 4	{B,F}
	C	bfn(C) = 5	{B,F,C}
	E	bfn(E) = 6	{B,F,C,E}
B	D	BFN(D) = 7	{F,C,E,D}
	E	None	{F,C,E,D}
	S	None	{F,C,E,D}
F	A	None	{C, E, D}
C	A	None	{E, D}
	D	None	{E, D}
E	A	None	{D}
	B	None	{D}
D	C	None	{}
	B	None	{}

Classification of Edges by BFS

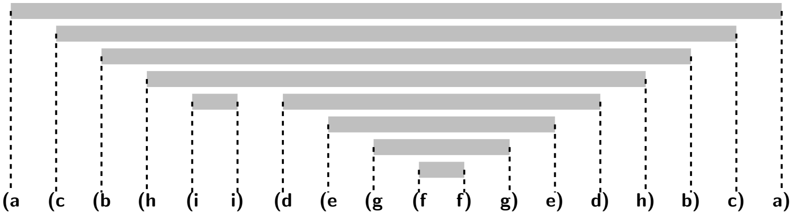
- There can be no back edges or forward edges in BFS of undirected graphs.
- For each tree edge (u, v), dist[v] = dist[u] + 1
- For each cross edge (u, v), dist[u] = dist[v] or dist[v] = dist[u] + 1

Parenthesis Theory

- Consider DFS numbers and reverse DFS numbers generated during DFS of a graph.
- Let u and v be any two vertices in the graph and let d[v] and f[u] respectively denote discovery time and finishing time of DFS then exactly one of the following three conditions hold.
  - If the intervals [d[u],f[u]] and [d[v],f[v]] are disjoint neither u nor v is a descendant of the other in DFS tree/forest.
  - If [d[v],f[v]] completely enclosed within [d[u],f[u]] then v is a descendant of u.
  - If [d[u],f[u]] completely enclosed within [d[v],f[v]] then u is a descendant of v.


i
h
a
b
d
e
f
c
g
[13,14]
[1,18]
[2,17]
[3,16]
[4,15]
i
a
c
b
h
[5,12]
d
e
g
f
[6,11]
[7,10]
[8,9]

Parenthesis Theory



- Opening parenthesis corresponds to discovery time d.
- Closing parenthesis corresponds to finish time f.
- Resulting expression is a valid parenthetical matching string.

Connected Components

- How to obtain connected components?
  - Using DFS/BFS it is possible.
- Outline of the algorithm is as follows:
  - Initialize a connected component number to 0.
  - Inside the second for-loop increment connected component number each time before calling DFS procedure.

Connected Components

```
index = 0;
count = 1; // initialize
for all (v ∈ V) {
    mark[v] = "unvisited";
}
for all (v ∈ V) {
    if (mark[v]=="unvisited"){
        DFS(G, v);
        increment(count); // Component number
    }
}
```

Depth First Search

```
DFS(G, v) {
    mark[v] = "visited";
    cID[v] = get(count);  // Component ID
    dfn[v] = ++index;  // DFS number
    for all (w ∈ ADJ_G(v)) {
        if (mark[w]=="unvisited") {
            parent[w] = v;
            DFS(G, w);
        }
    }
}
```

Topological Sorting

Definition

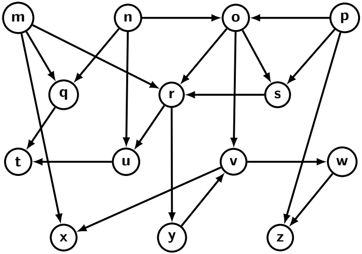
A linear total ordering of the vertices of directed graph, such that for each edge $u \to v$, u appears before v in the list.

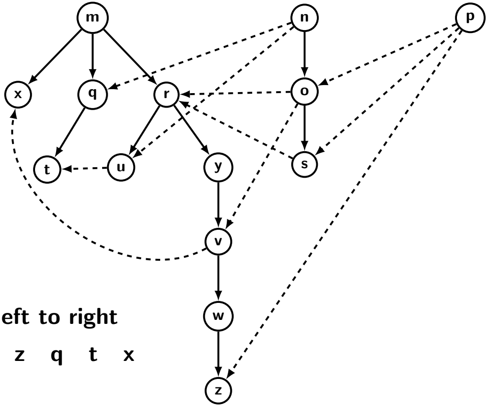
- Scheduling constraints between lectures.
- Pre-requisites of your B. Tech degree.
- Various stages or tasks related to completion of projects.

Topological Sorting

- Just call DFS to compute reverse DFS number.
- As numbering to a vertex get assigned insert it to the front of an initally empty linked list.
- Linked list gives the topological sorted sequence in decreasing order of finish time of the task.

Topological Sorting

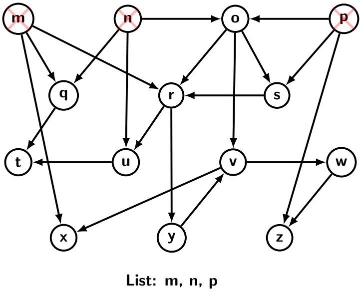


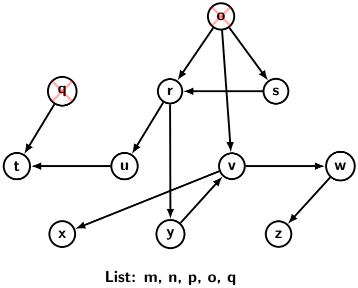


Topological Sort: all edges from left to right

p n o s m r u y v w z q t x

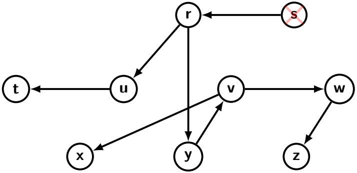
Topological Sorting



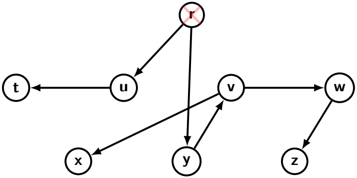


- Another simple way to get topological sort is as follows:
  - List out all vertices with no incoming edges.
  - Remove these vertices and keep repeating two step until all vertices as listed.

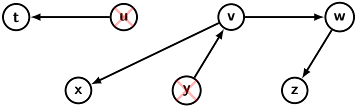
Topological Sorting



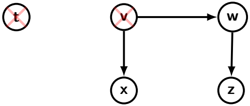
List: m, n, p, o, q, s



List: m, n, p, o, q, s, r

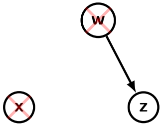


List: m, n, p, o, q, s, r, u, y



List: m, n, p, o, q, s, r, u, y, t, v

Topological Sorting



List: m, n, p, o, q, s, r, u, y, t, v, w, x

Final List:

m, n, p, o, q, s, r, u, y, t, v, w, x, z

- After deleting w and x only z is left out.
- Just append z to the list.
- As we can check the list orders that nodes such that edges alway directed from left to right.