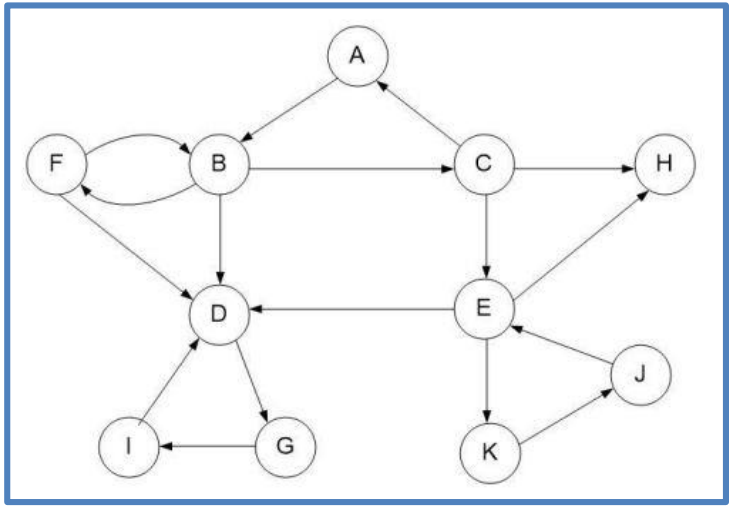
TRAVERSAL OF DIRECTED GRAPHS



Partha P Chakrabarti

Indian Institute of Technology Kharagpur

Directed Graphs

An Undirected Graph G = (V, E) consists of the following:

- A set of Vertices or Nodes V
- A set of DIRECTED Edges E where each edge connects two vertices of V. The edge is an ORDERED pair of vertices

Successor Function: succ(i) = {set of nodes to which node i is connected}

Directed Acyclic Graphs (DAGs): Such Graphs have no cycles (Figure 2)

Weighted Undirected Graphs: Such Graphs may have weights on edges (Figure 3). We can also have Weighted DAGs

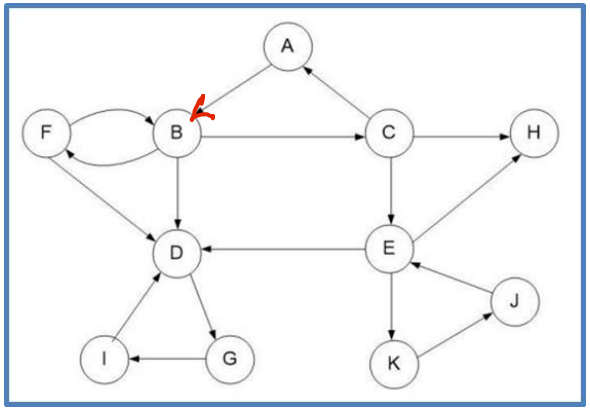


Figure 1

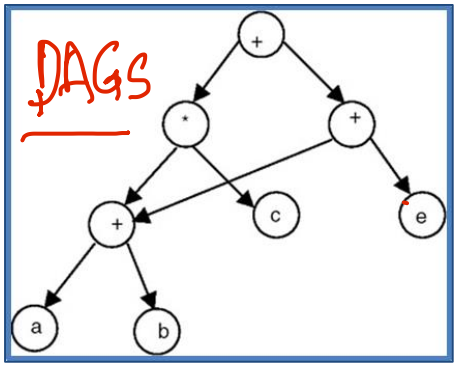


Figure 2

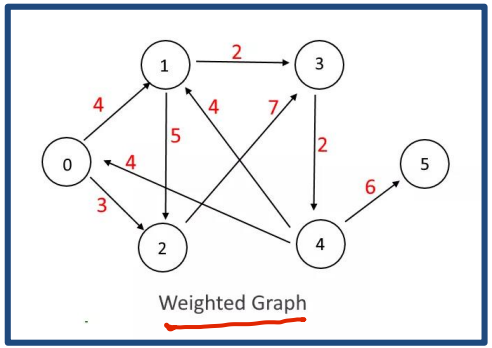
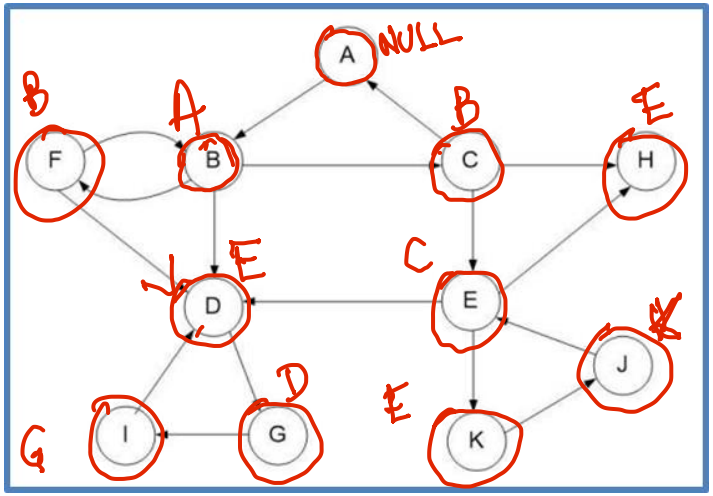


Figure 3

Basic Traversal Algorithm (Depth First Search)

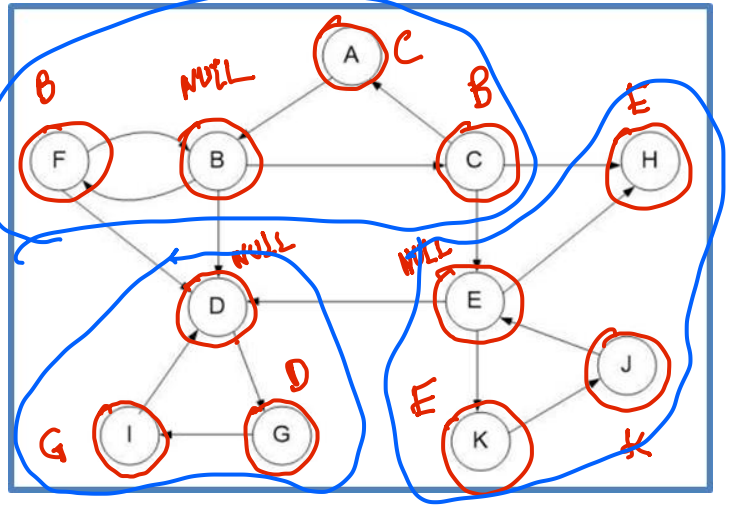
```
Global Data: G = (V,E)
visited [i] indicates if node i is visited. / initially 0 /
Parent[i] = parent of a node in the Search / initially NULL /
succ(i) = {set of nodes to which node i is connected}
Dfs(node) {
    visited[node] = 1;
    for each j in succ(node) do {
        if (visited [j] ==0) { Parent[j] = node;
                               Dfs(j) }
    }
}
```



Parent[i] = j if there is a traversal from j to i and i was not visited earlier

Traversing the Complete Graph by DFS

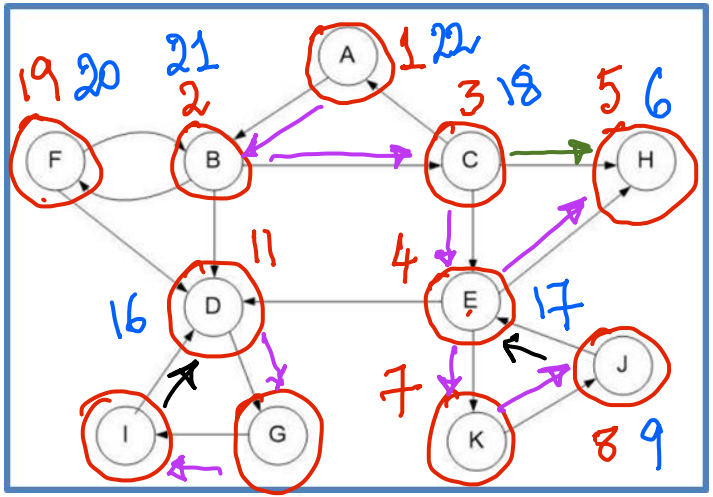
```
Global Data: G = (V,E)
visited [i] indicates if node i is visited. / initially 0 /
Parent[i] = parent of a node in the Search / initially NULL /
succ(i) = {set of nodes to which node i is connected}
Dfs(node) {
   visited[node] = 1;
   for each j in succ(node) do {
      if (visited [j] ==0) { Parent[j] = node;
                             Dfs(j) }
   }
}
```



for each node i if (visited[i]==0)
Dfs(i);

Entry-Exit Numbering

```
Global Data: G = (V,E)
visited [i] indicates if node i is visited. / initially 0 /
Parent[i] = parent of a node in the Search / initially
NULL /
Entry[i] = node entry sequence / initially 0 /
Exit[i] = node exit sequence / initially 0 /
succ(i) = {set of nodes to which node i is connected}
numb = 0; /Global Time Stamp/
Dfs(node) {
   visited[node] = 1; numb = numb+1;
   Entry[node] = numb;
   for each j in succ(node) do
      if (visited [j] ==0) { Parent[j] = node;
                             Dfs(j) }
    numb = numb + 1;
    Exit[node] = numb;
 }
```



Tree Edge, Back Edge, Forward Edge, Cross Edge

Global Data: G = (V,E)

visited [i] indicates if node i is visited. / initially 0 /

Parent[i] = parent of a node in the Search / initially NULL /

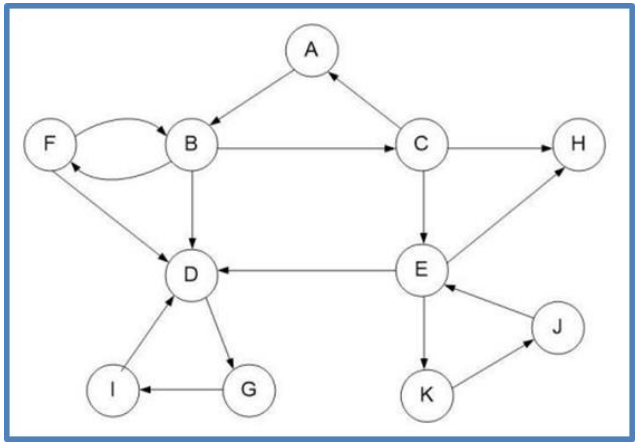
Entry[i] = node entry sequence / initially 0 /

Exit[i] = node exit sequence / initially 0 /

succ(i) = {set of nodes to which node i is connected}

numb = 0;

```
Dfs(node) {
   visited[node] = 1; numb = numb+1;
   Entry[node] = numb;
   for each j in succ(node) do
      if (visited [j] ==0) { Parent[j] = node;
                             Dfs(j) }
    numb = numb + 1;
    Exit[node] = numb;
 }
```



Edge (u,v) is

Tree Edge or Forward Edge: if & only if

Entry[u] < Entry[v] < Exit[v] < Exit[u]

Back Edge: if & only if

Entry[v] < Entry [u] < Exit [u] < Exit [v]

Cross Edge: if & only if

Entry [v] < Exit [v] < Entry [u] < Exit [u]

Reachability, Paths, Cycles, Components

Global Data: G = (V,E)

visited [i] indicates if node i is visited. / initially 0 /

Parent[i] = parent of a node in the Search / initially NULL /

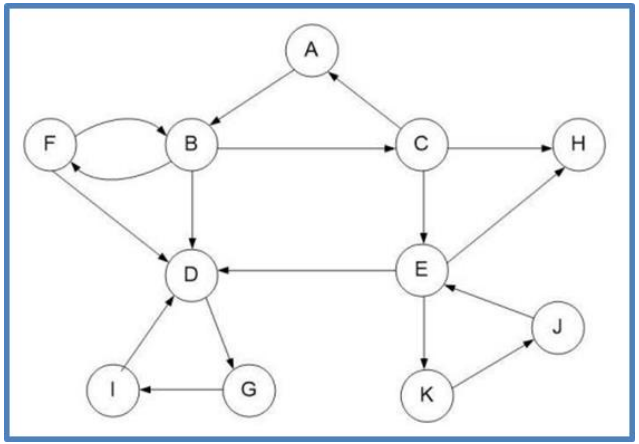
Entry[i] = node entry sequence / initially 0 /

Exit[i] = node exit sequence / initially 0 /

succ(i) = {set of nodes to which node i is connected}

numb = 0;

```
Dfs(node) {
   visited[node] = 1; numb = numb+1;
   Entry[node] = numb;
   for each j in succ(node) do
      if (visited [j] ==0) { Parent[j] = node;
                             Dfs(j) }
    numb = numb + 1;
    Exit[node] = numb;
 }
```



Edge (u,v) is

Tree Edge or Forward Edge: if & only if

Entry[u] < Entry[v] < Exit[v] < Exit[u]

Back Edge: if & only if

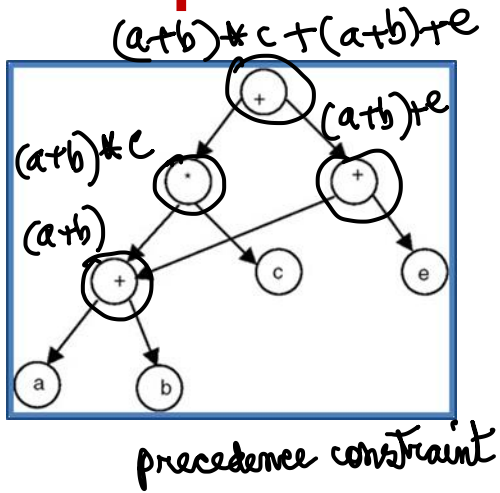
Entry[v] < Entry [u] < Exit [u] < Exit [v]

Cross Edge: if & only if

Entry [v] < Exit [v] < Entry [u] < Exit [u]

Directed Acyclic Graphs

```
Global Data: G = (V,E)
visited [i] indicates if node i is visited. / initially 0 /
Parent[i] = parent of a node in the Search / initially
NULL /
Entry[i] = node entry sequence / initially 0 /
Exit[i] = node exit sequence / initially 0 /
succ(i) = {set of nodes to which node i is connected}
numb = 0;
Dfs(node) {
   visited[node] = 1; numb = numb+1;
   Entry[node] = numb;
   for each j in succ(node) do
      if (visited [j] ==0) { Parent[j] = node;
                             Dfs(j) }
    numb = numb + 1;
    Exit[node] = numb;
 }
```



precedence constraint

NO BACK EDGE

ROOT(S)

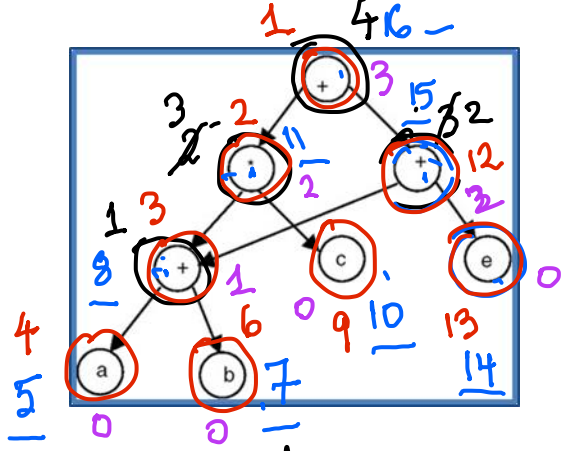
LEAVES

Topological Ordering, Level Values

```
Global Data: G = (V,E)
visited [i] indicates if node i is visited. / initially 0 /
Parent[i] = parent of a node in the Search / initially
NULL /
Entry[i] = node entry sequence / initially 0 /
Exit[i] = node exit sequence / initially 0 /
succ(i) = {set of nodes to which node i is connected}
numb = 0; numb1 = 0;
Dfs(node) {
   visited[node] = 1; numb = numb+1;
   Entry[node] = numb;
   for each j in succ(node) do
      if (visited [j] ==0) { Parent[j] = node;
                             Dfs(j) }
    numb1 = numb1 + 1;
    Exit[node] = numb1;
 }
```

level

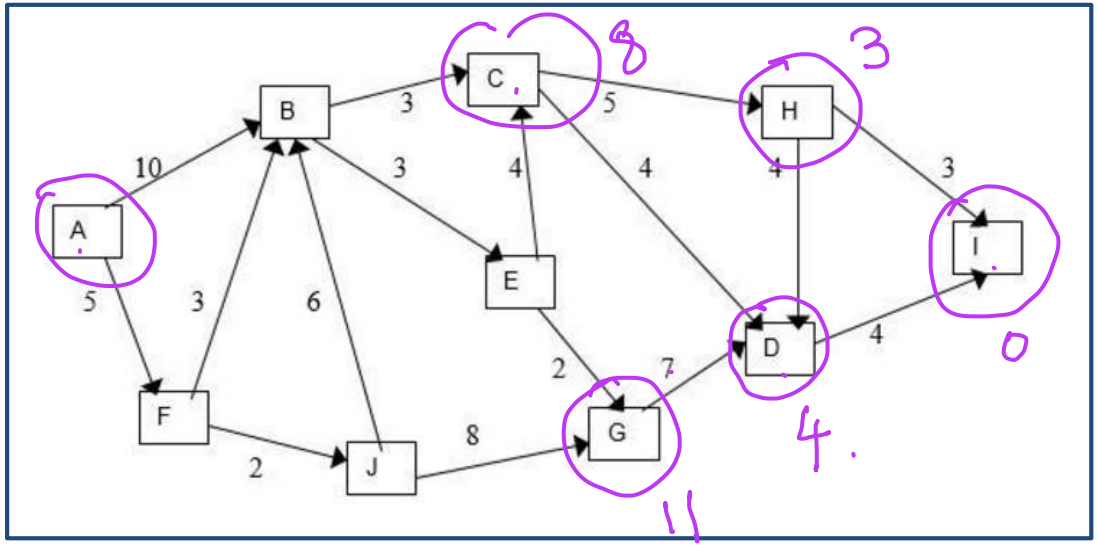
level =



exit values

to get a topological order

Shortest Cost Path in Weighted DAGs



$sp(s, g)$

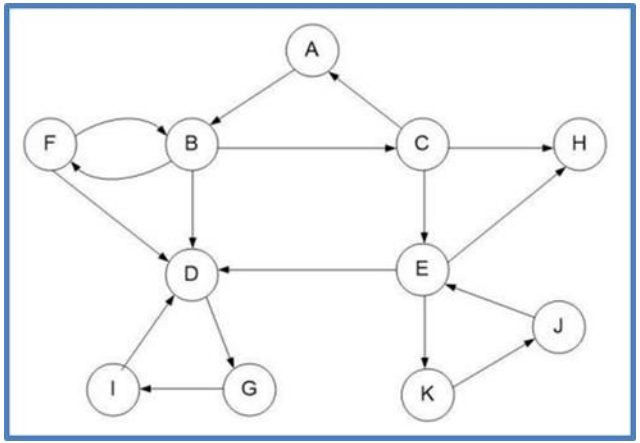
$= 0$ if $s = g$

$= \min_{\forall n_i \in succ(s)} \{ sp(n_i, g) + c(s, n_i) \}$

$O(|V| + |E|)$ submit as a homework assignment

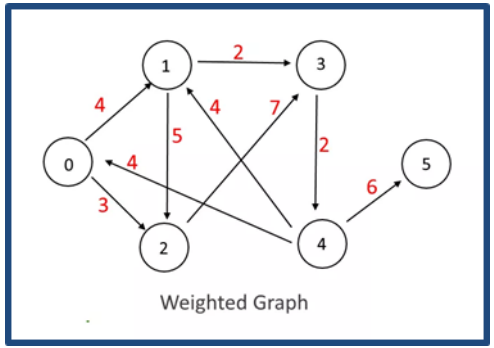
Breadth-First Search

```
Global Data: G = (V,E)
Visited[i] all initialized to 0
Queue Q initially {}
 BFS(k) {
  visited [k] = 0; Q = {k};
  While Q != {} {
    j = DeQueue (Q);
    if visited[j] == 0 {
    visited [j] = 1;
    For each k in succ (j) {
          if (visited[k]==0) EnQueue(Q,k); }
    }
    }
/Parent links, Shortest Length Path Finding in
unweighted directed graphs/
```



Pathfinding in Weighted Directed Graphs

```
Global Data: G = (V,E)
Visited[i] all initialized to 0,
Cost[j] all initialized to INFINITY
Ordered Queue Q initially {}
 BFSW(k) {
 visited [k] = 0; cost [k] = 0; Q = {k};
 While Q != {} {
   j = DeQueue (Q);
   if visited[j] == 0 {
   visited [j] = 1;
   For each k in succ (j) {
   if cost[k] > cost[j] + c[j,k]
          cost[k] = cost[j] + c[j,k];
   EnQueue(Q,k);}
  }
}
```



Dijkstra's SP Algorithm

Thank you