Graph II

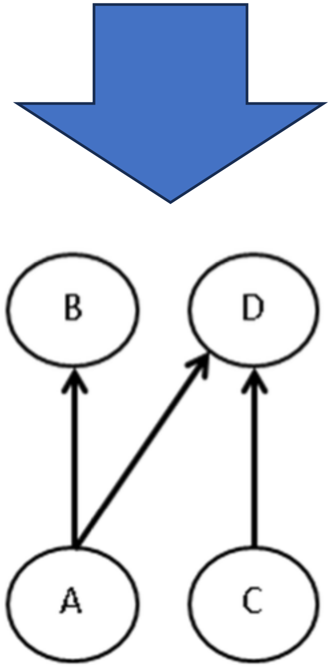
Outline

- Topology sorting
- Transitive closure
- Implementation

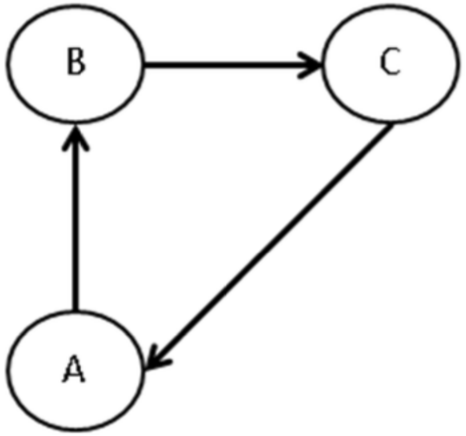
Acyclic graph

- A graph is formed by vertices and by edges connecting pairs of vertices, where the vertices can be any kind of object that is connected in pairs by edges.
- In the case of a directed graph, each edge has an orientation, from one vertex to another vertex.
- A path in a directed graph is a sequence of edges having the property that the ending vertex of each edge in the sequence is the same as the starting vertex of the next edge in the sequence; a path forms a cycle if the starting vertex of its first edge equals the ending vertex of its last edge.
- A directed acyclic graph is a directed graph that has no cycles.

Acyclic graph



A



B

Topological Sorting

- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from vi to vj, then vj appears after vi in the ordering.

Topological Sorting

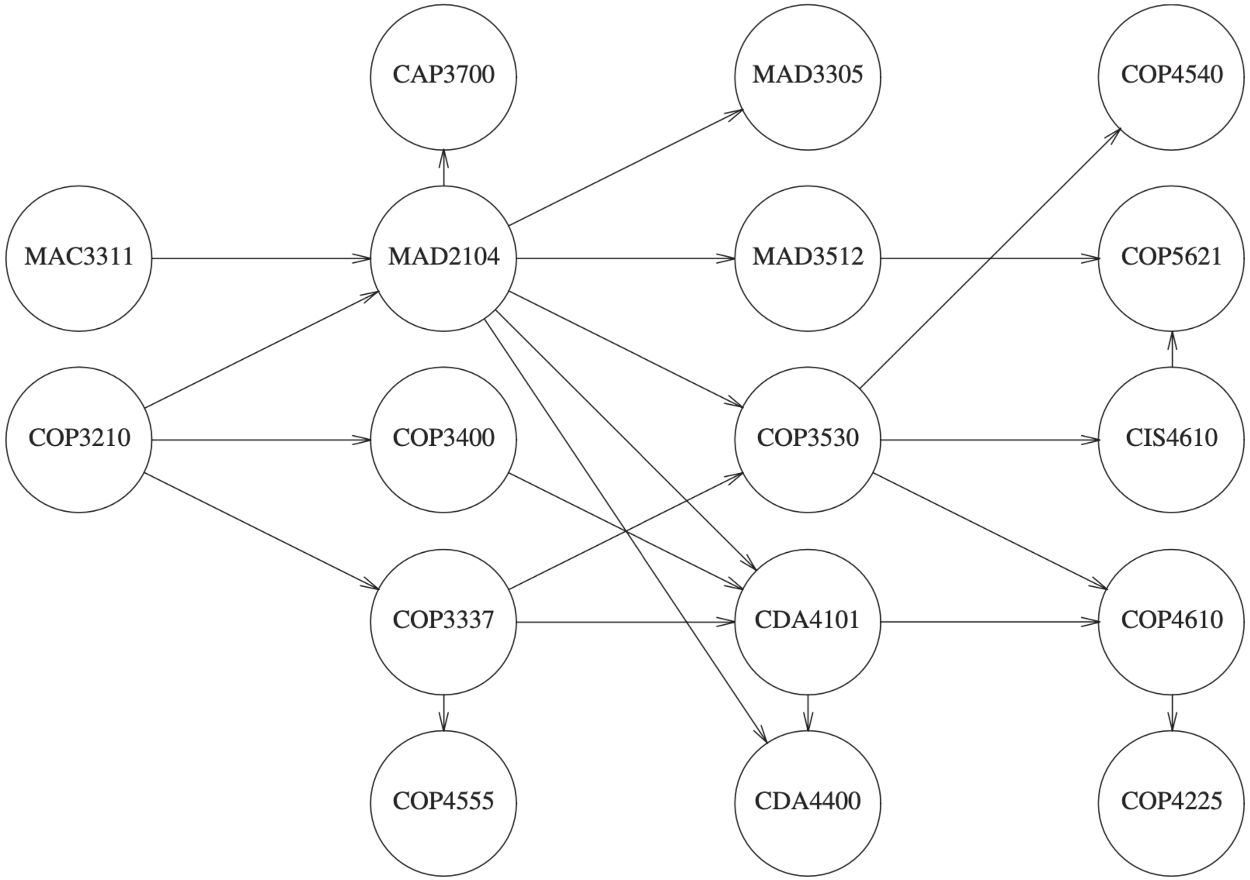


Figure 9.3 An acyclic graph representing course prerequisite structure

Topological Sorting

- A directed edge (v, w) indicates that course v must be completed before course w may be attempted.
- A topological ordering of these courses is any course sequence that does not violate the prerequisite requirement.

Topological Sorting

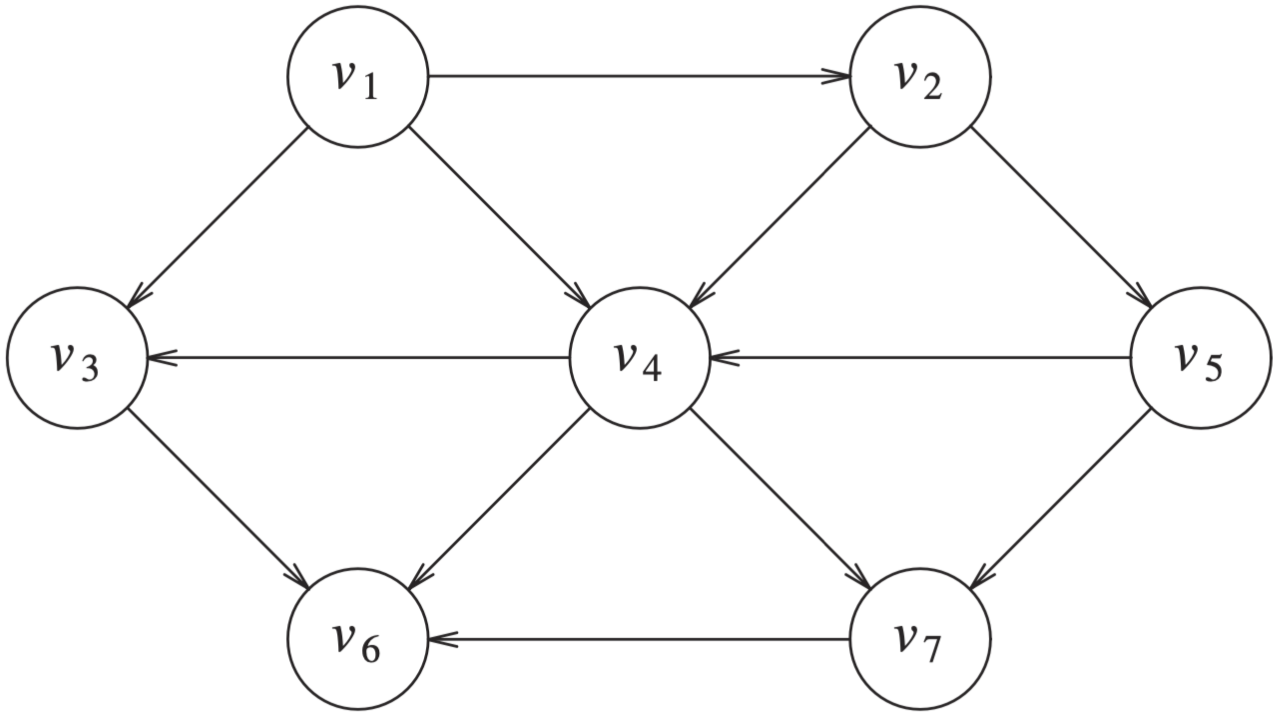
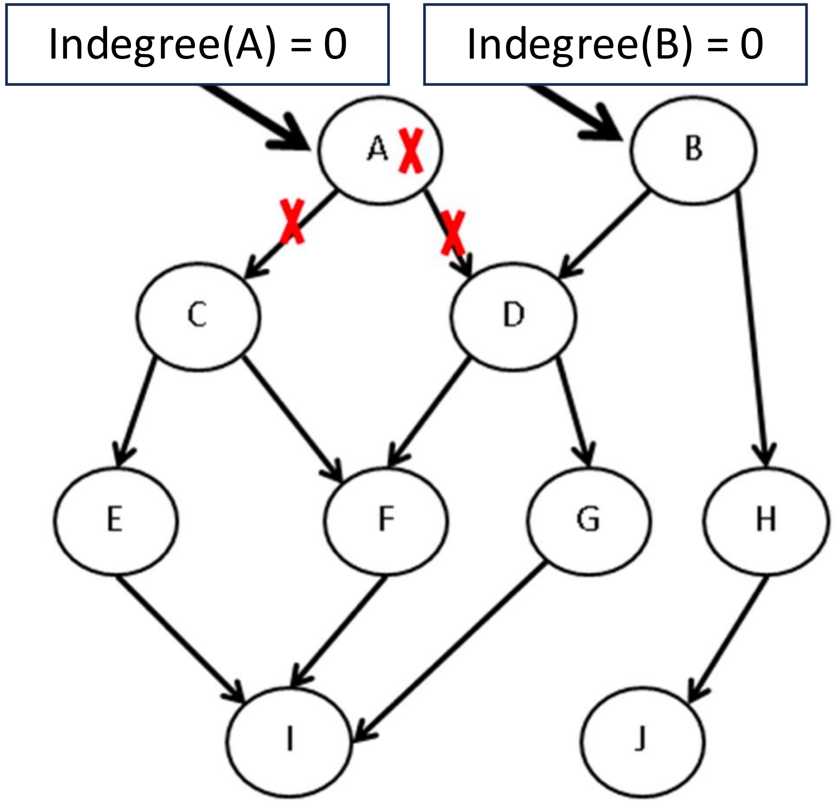


Figure 9.4 An acyclic graph

Topological Sorting

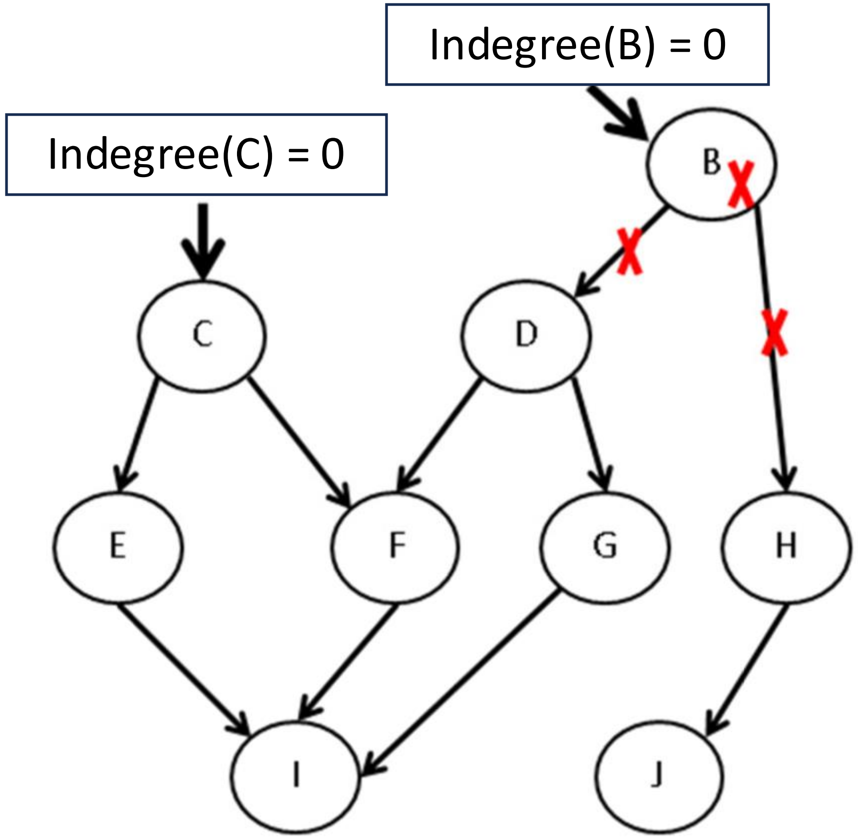
- A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges.
- We can then print this vertex, and remove it, along with its edges, from the graph.
- Then we apply this same strategy to the rest of the graph.

Topological Sorting



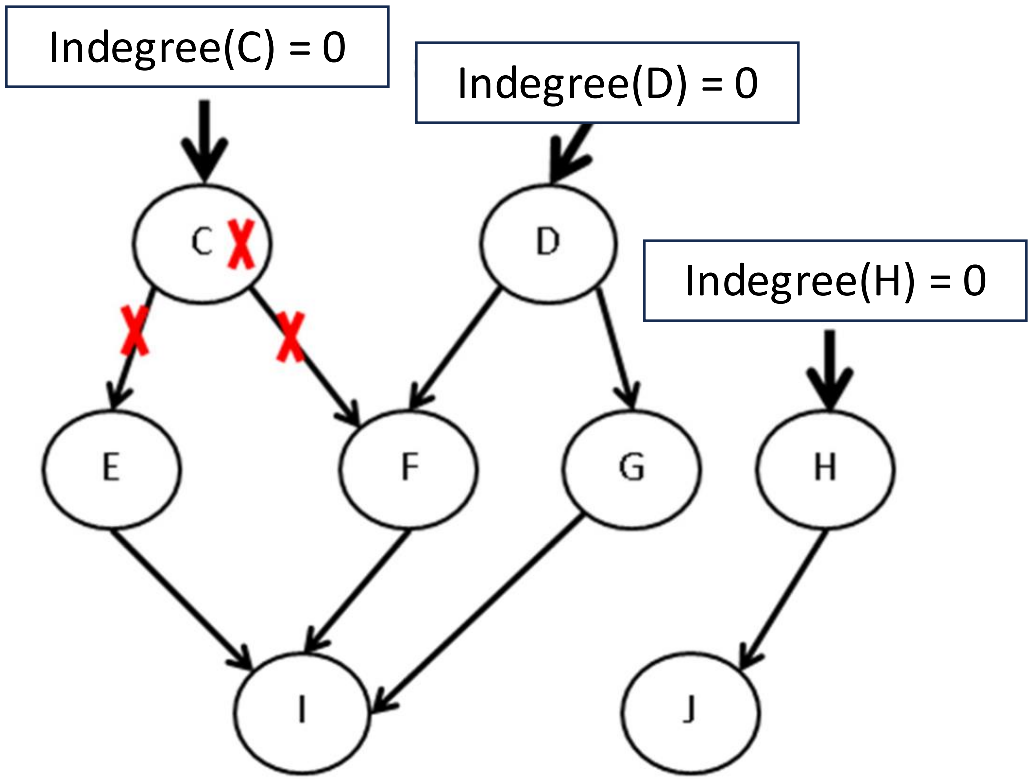
A

Topological Sorting



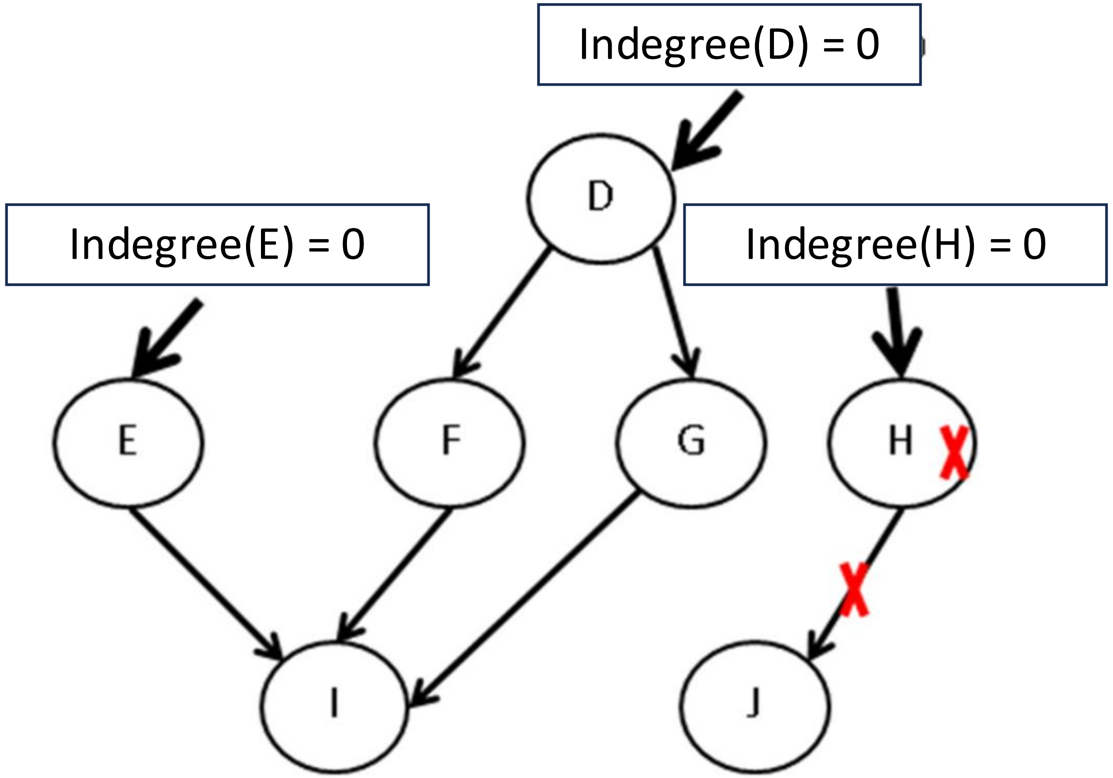
B

Topological Sorting



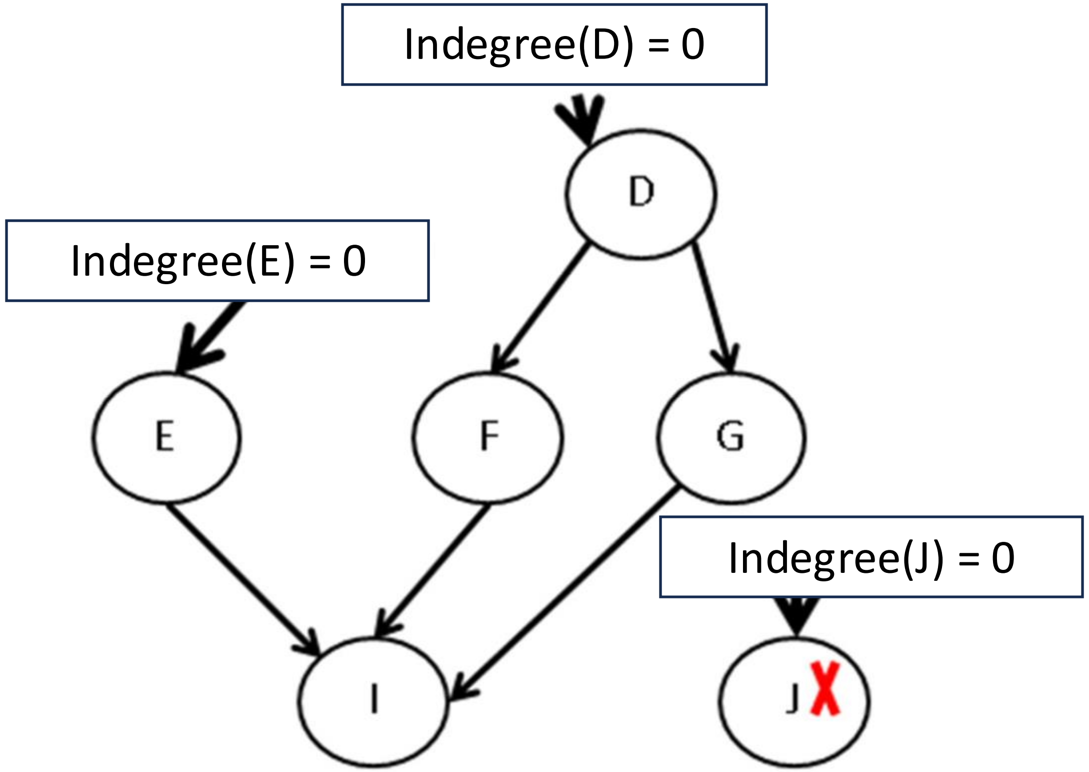
C

Topological Sorting



D

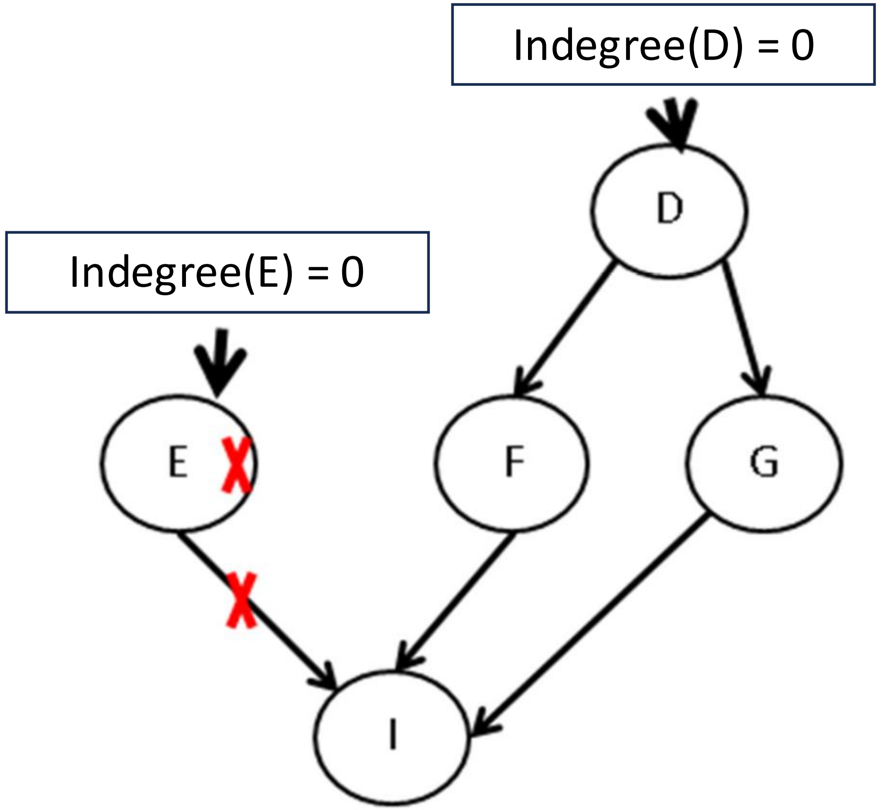
Topological Sorting



E

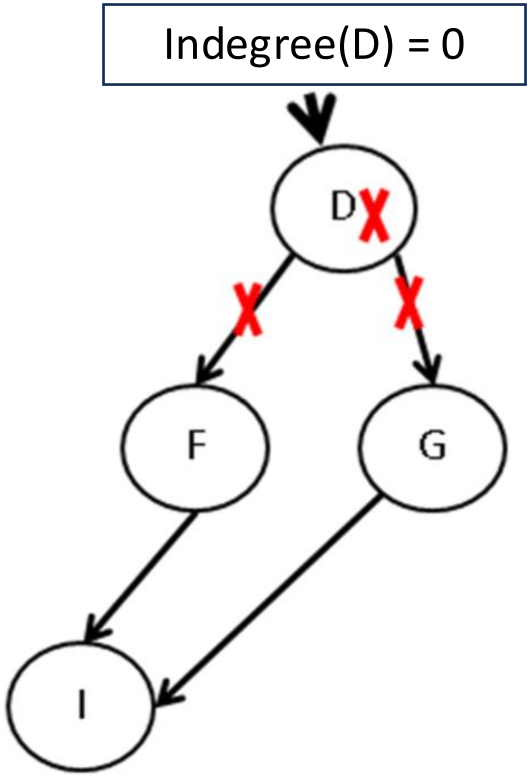
.

Topological Sorting



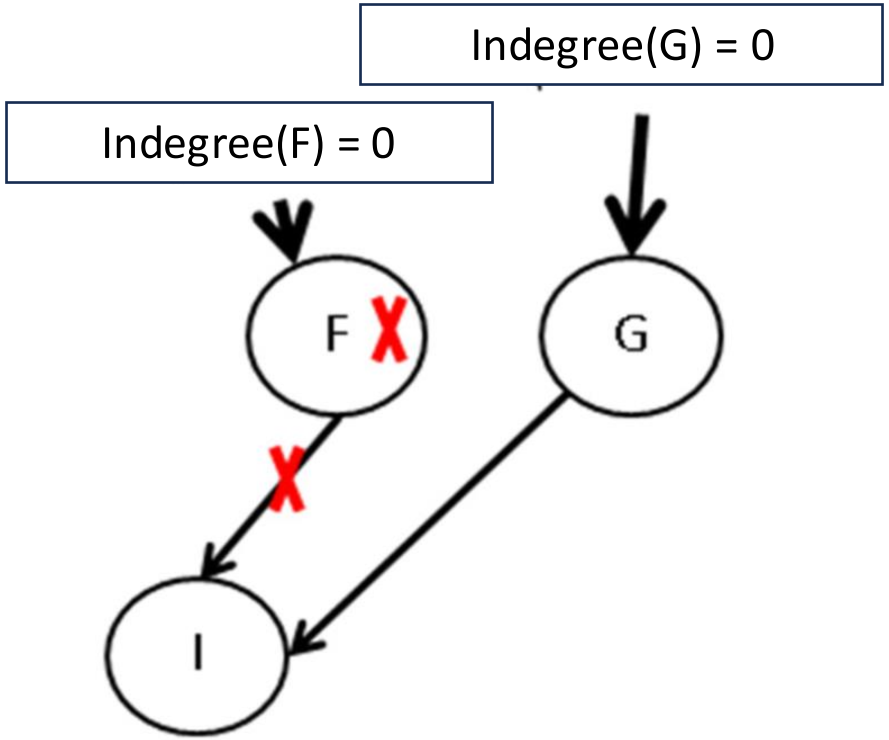
F

Topological Sorting



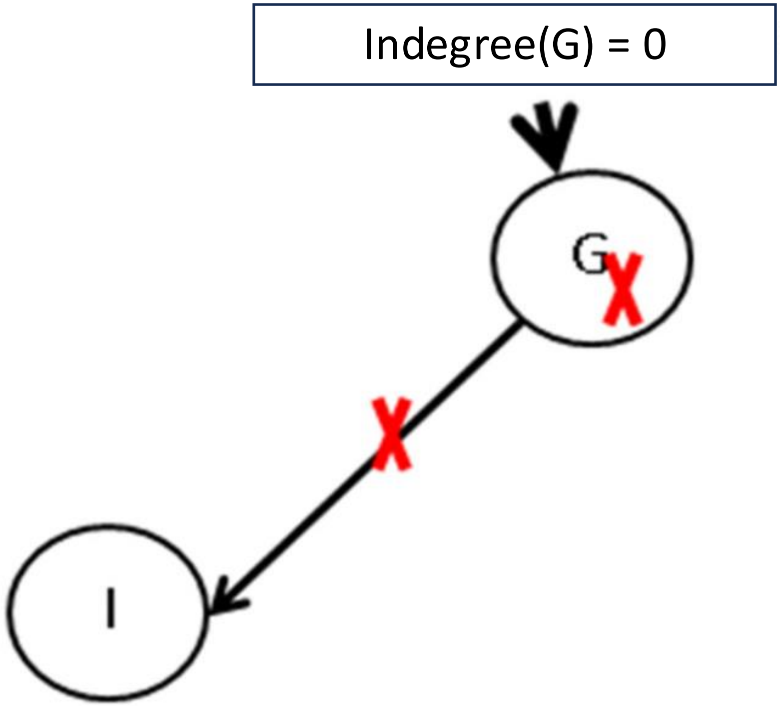
G

Topological Sorting



H

Topological Sorting



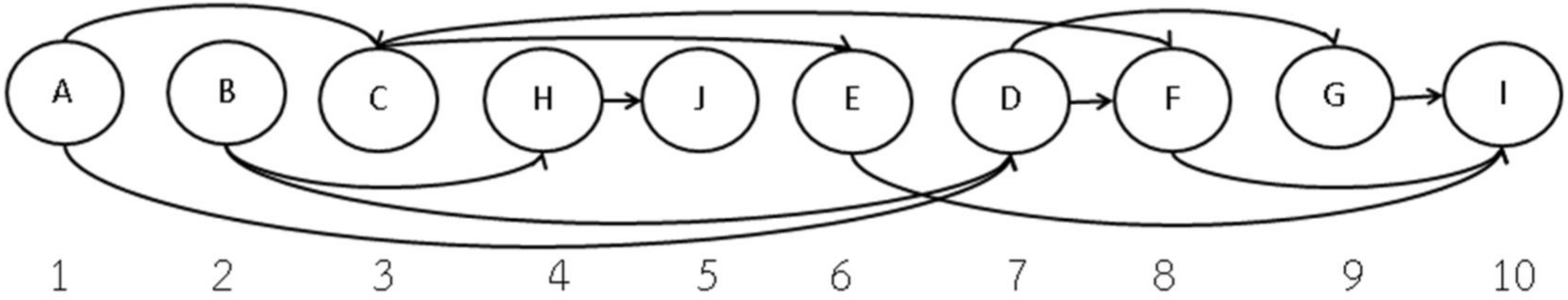
I

Topological Sorting

Indegree(I) = 0

J

Topological Sorting

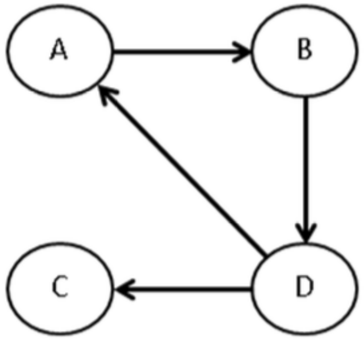


K

Transitive Closure

- Transitive Closure it the reachability matrix to reach from vertex u to vertex v of a graph.
- One graph is given, we have to find a vertex v which is reachable from another vertex u, for all vertex pairs (u, v).

Transitive Closure



A

Destination

Source

	A	B	C	D
A	1	0	0	0
B	0	0	0	1
C	0	0	0	0
D	1	0	1	0

Adjacency matrix

B

Destination

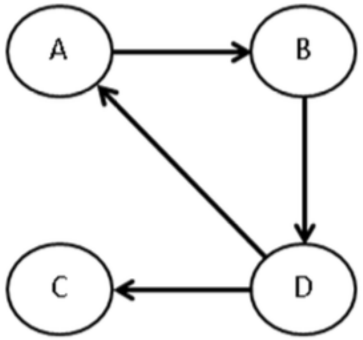
Source

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	0	0
D	1	1	1	1

Transitive closure matrix

C

Transitive Closure



A

Destination

	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	0	0	0	0
D	1	0	1	0

Source

Adjacency matrix

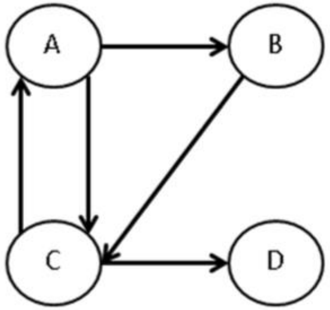
Destination

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	0	0
D	1	1	1	1

Source

Transitive closure matrix

Transitive Closure: DFS



A

Start at A

	A	B	C	D
A	0	0	0	0
B	0	0	0	0
C	0	0	0	0
D	0	0	0	0

Matrix

B

Start at B

	A	B	C	D
A	1	1	1	1
B	0	0	0	0
C	0	0	0	0
D	0	0	0	0

Matrix

C

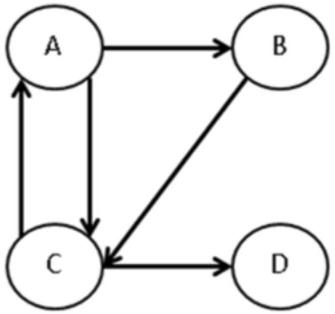
Start at C

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	0	0
D	0	0	0	0

Matrix

D

Transitive Closure: DFS



A

Start at D

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	1	1	1	1
D	0	0	0	0

Matrix

E

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	1	1	1	1
D	0	0	0	0

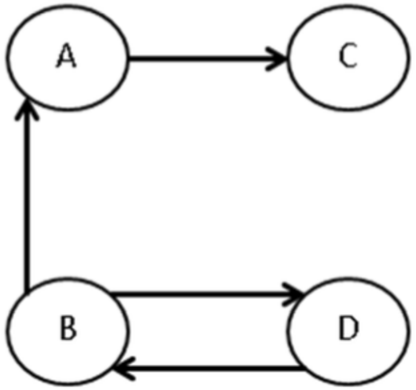
Transitive closure matrix

F

Transitive Closure: Wallshall Algorithm

- Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix.

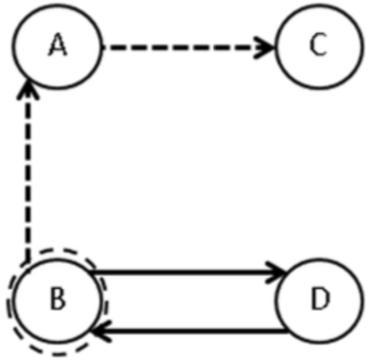
Transitive Closure: Wallshall Algorithm



	A	B	C	D
A	0	0	1	0
B	1	0	0	1
C	0	0	0	0
D	0	1	0	0

A

Transitive Closure: Wallshall Algorithm



B->A A->C B->C

A->C

B->A

	A	B	C	D
A	0	0	1	0
B	1	0	0	1
C	0	0	0	0
D	0	1	0	0

Adjacency matrix

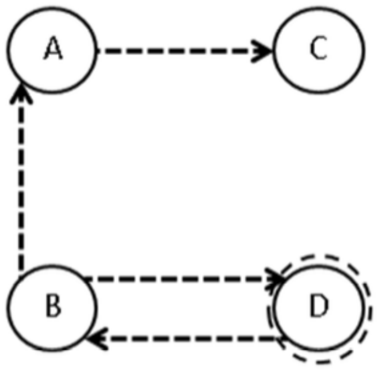
	A	B	C	D
A	0	0	0	0
B	0	0	$\boxed{1}$	0
C	0	0	0	0
D	0	0	0	0

A

	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	1	0	0	0

M_A

Transitive Closure: Wallshall Algorithm



B->A → D->A
D->B
B->C → D->C
B->D → D->D

B->A B->C B->D

	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	0	1	0	0

M_A

+

D->B

	A	B	C	D
A	0	0	0	0
B	0	0	0	0
C	0	0	0	0
D	1	0	1	1

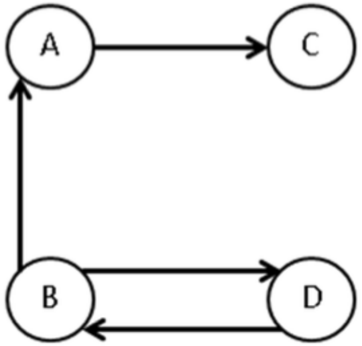
B

=

	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	1	1	1	1

M_B

Transitive Closure: Wallshall Algorithm



	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	1	1	1	1

M_B

A->C

B->C

D->C

	A	B	C	D
A	0	0	0	0
B	0	0	0	0
C	0	0	0	0
D	0	0	0	0

C

	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	1	1	1	1

M_C

Transitive Closure: Wallshall Algorithm

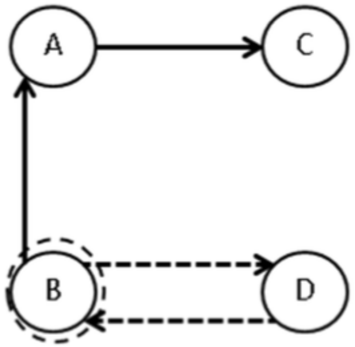
D->A D->B D->C D->D

B->D

D->D

D->A
D->B
D->C
D->D

B->A
D->A
B->B
D->B
B->C
D->C
B->D
D->D



	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	1	1	1	1

M_C

B->D

D->D

	A	B	C	D
A	0	0	0	0
B	1	1	1	1
C	0	0	0	0
D	1	1	1	1

D

	A	B	C	D
A	0	0	1	0
B	1	0	1	1
C	0	0	0	0
D	1	1	1	1

Transitive closure matrix

Implementation

```
#include<bits/stdc++.h>
using namespace std;
class graph
{
    public:
    int edges[100][100];
    int s_v;
    graph(int n)
    {
        s_v = n;
        for(int i=0;i<s_v;i++)
        {
            for(int j=0;j<s_v;j++)
            {
                edges[i][j] = 0;
            }
        }
    }
    void add_edge(int x,int y,int w)
    {
        edges[x][y] = w;
    }
    void print()
    {
        for(int i=0;i<s_v;i++)
        {
            cout<<i<<" : ";
            for(int j=0 ; j <= s_v ; j++ )
            {
                if(edges[i][j] > 0 )
                {
                    cout<<j<<","<<edges[i][j]<<"  ";
                }
            }
            cout<<endl;
        }
    }
    void bft(int start)
    {
        bool visited_bft[100];
        for(int i=0;i<100;i++)
        {
            visited_bft[i] = 0;
        }
        visited_bft[start] = 1;
        vector<int> q;
        q.push_back(start);
        while(q.empty() == 0)
        {
            start = q.front();
            cout << start << " ";
            q.erase(q.begin());
            for(int y=0 ; y<s_v ; y++)
            {
```

```
    void sub_graph()
    {
        int num_subgraph  = 1;
        for(int i=0;i<100;i++)
        {
            visited_dft[i] = 0;
        }
        for(int y=0;y<s_v;y++)
        {
            if( visited_dft[y]==0 )
            {
                cout<<"\nsub graph = "<<num_subgraph<<" : ";
                sub_dft(y);
                num_subgraph = num_subgraph + 1;
            }
        }
    }
    int n_in_degree[100];
    int t_edges[100][100];
    void in_degree()
    {
        for(int i=0;i<s_v;i++)
        {
            n_in_degree[i] = 0;
            for(int j=0;j<s_v;j++)
            {
```

Implementation

```
            for(int k=0;k<s_v;k++)
            {
                if( t_edges[j][k] == 1 )
                {
                    n_in_degree[k]++;
                }
            }
        }
    }
}
void topologicalsort()
{
    bool visited[100];
    int t_s_v = 0;
    for(int i=0;i<s_v;i++)
    {
        visited[i] = 0;
        for(int j=0;j<s_v;j++)
        {
            t_edges[i][j] =  edges[i][j];
        }
    }
    while(t_s_v < s_v)
    {
        in_degree();
        for(int i=0;i<s_v;i++)
        {
            if(n_in_degree[i] == 0 && visited[i] == 0)
            {
                visited[i] = 1;
                cout<<i<<" ";
                for(int j=0;j<s_v;j++)
                {
                    t_edges[i][j] = 0;
                }
                t_s_v++;
                break;
            }
        }
    }
}
bool tc[100][100];
int  start_vertex;
bool first_access;
void sub_transitive_closure_dft(int start)
{
    if(first_access > 0)
    {
        visited_dft[start] = 1;
        tc[start_vertex][start] = 1;
    }
    first_access = 1;
    for(int y=0;y<s_v;y++)
    {
        if( visited_dft[y] == 0 && edges[start][y] > 0 )
        {
            sub_transitive_closure_dft(y);
        }
    }
}
void transitive_closure_dft()
{
    for(int i=0;i<s_v;i++)
    {
        for(int j=0;j<s_v;j++)
        {
            tc[i][j] = 0;
        }
    }
    for (int i=0;i<s_v;i++)
    {
        for (int j=0;j<s_v;j++)
        {
            visited_dft[j] = false;
        }
        first_access = 0;
        start_vertex = i;
        sub_transitive_closure_dft(start_vertex);
    }
    for (int i=0;i<s_v;i++)
    {
        for (int j=0;j<s_v;j++)
        {
            cout<<tc[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

Implementation

```
void warshall()
{
    for(int i=0;i<s_v;i++)
    {
        for(int j=0;j<s_v;j++)
        {
            tc[i][j] = edges[i][j];
        }
    }
    for (int k=0;k<s_v;k++)
    {
        for (int i=0;i<s_v;i++)
        {
            for (int j=0;j<s_v;j++)
            {
                tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
            }
        }
    }
    for (int i=0;i<s_v;i++)
    {
        for (int j=0;j<s_v;j++)
        {
            cout<<tc[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

https://www.tutorialspoint.com

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ