**Topological Sorting:**

**Basic code:**

// A C++ program to print topological sorting of a DAG

#include<iostream>

#include <list>

#include <stack>

using namespace std;

// Class to represent a graph

class Graph

{

int V; // No. of vertices'

// Pointer to an array containing adjacency listsList

list<int> \*adj;

// A function used by topologicalSort

void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints a Topological Sort of the complete graph

void topologicalSort();

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

// A recursive function used by topologicalSort

void Graph::topologicalSortUtil(int v, bool visited[],

stack<int> &Stack)

{

// Mark the current node as visited.

visited[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

topologicalSortUtil(\*i, visited, Stack);

// Push current vertex to stack which stores result

Stack.push(v);

}

// The function to do Topological Sort. It uses recursive

// topologicalSortUtil()

void Graph::topologicalSort()

{

stack<int> Stack;

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for (int i = 0; i < V; i++)

visited[i] = false;

// Call the recursive helper function to store Topological

// Sort starting from all vertices one by one

for (int i = 0; i < V; i++)

if (visited[i] == false)

topologicalSortUtil(i, visited, Stack);

// Print contents of stack

while (Stack.empty() == false)

{

cout << Stack.top() << " ";

Stack.pop();

}

}

// Driver program to test above functions

int main()

{

// Create a graph given in the above diagram

Graph g(6);

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

g.addEdge(3, 1);

cout << "Following is a Topological Sort of the given graph \n";

g.topologicalSort();

return 0;

}

**Note:** Now, though, we can start from an index with in degree 0, a topological sorting can be started from any vertex.

**Kahn’s Algorithm For Topological Sorting:**

**A DFS based solution to find a topological sort has already been discussed.**

**In this article we will see another way to find the linear ordering of vertices in a directed acyclic graph (DAG). The approach is based on the below fact :**

**A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0.**

**Proof: There’s a simple proof to the above fact is that a DAG does not contain a cycle which means that all paths will be of finite length. Now let S be the longest path from u(source) to v(destination). Since S is the longest path there can be no incoming edge to u and no outgoing edge from v, if this situation had occurred then S would not have been the longest path**

**=> indegree(u) = 0 and outdegree(v) = 0**

**Algorithm:**

Steps involved in finding the topological ordering of a DAG:

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

Increment count of visited nodes by 1.

Decrease in-degree by 1 for all its neighboring nodes.

If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If count of visited nodes is not equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

**Print All Topological Sorting:**

We can go through all possible ordering via backtracking , the algorithm step are as follows :

Initialize all vertices as unvisited.

Now choose vertex which is unvisited and has zero indegree and decrease indegree of all those vertices by 1 (corresponding to removing edges) now add this vertex to result and call the recursive function again and backtrack.

After returning from function reset values of visited, result and indegree for enumeration of other possibilities.

**// C++ program to print all topological sorts of a graph**

**#include <bits/stdc++.h>**

**using namespace std;**

**class Graph**

**{**

**int V; // No. of vertices**

**// Pointer to an array containing adjacency list**

**list<int> \*adj;**

**// Vector to store indegree of vertices**

**vector<int> indegree;**

**// A function used by alltopologicalSort**

**void alltopologicalSortUtil(vector<int>& res,**

**bool visited[]);**

**public:**

**Graph(int V); // Constructor**

**// function to add an edge to graph**

**void addEdge(int v, int w);**

**// Prints all Topological Sorts**

**void alltopologicalSort();**

**};**

**// Constructor of graph**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**// Initialising all indegree with 0**

**for (int i = 0; i < V; i++)**

**indegree.push\_back(0);**

**}**

**// Utility function to add edge**

**void Graph::addEdge(int v, int w)**

**{**

**adj[v].push\_back(w); // Add w to v's list.**

**// increasing inner degree of w by 1**

**indegree[w]++;**

**}**

**// Main recursive function to print all possible**

**// topological sorts**

**void Graph::alltopologicalSortUtil(vector<int>& res,**

**bool visited[])**

**{**

**// To indicate whether all topological are found**

**// or not**

**bool flag = false;**

**for (int i = 0; i < V; i++)**

**{**

**// If indegree is 0 and not yet visited then**

**// only choose that vertex**

**if (indegree[i] == 0 && !visited[i])**

**{**

**// reducing indegree of adjacent vertices**

**list<int>:: iterator j;**

**for (j = adj[i].begin(); j != adj[i].end(); j++)**

**indegree[\*j]--;**

**// including in result**

**res.push\_back(i);**

**visited[i] = true;**

**alltopologicalSortUtil(res, visited);**

**// resetting visited, res and indegree for**

**// backtracking**

**visited[i] = false;**

**res.erase(res.end() - 1);**

**for (j = adj[i].begin(); j != adj[i].end(); j++)**

**indegree[\*j]++;**

**flag = true;**

**}**

**}**

**// We reach here if all vertices are visited.**

**// So we print the solution here**

**if (!flag)**

**{**

**for (int i = 0; i < res.size(); i++)**

**cout << res[i] << " ";**

**cout << endl;**

**}**

**}**

**// The function does all Topological Sort.**

**// It uses recursive alltopologicalSortUtil()**

**void Graph::alltopologicalSort()**

**{**

**// Mark all the vertices as not visited**

**bool \*visited = new bool[V];**

**for (int i = 0; i < V; i++)**

**visited[i] = false;**

**vector<int> res;**

**alltopologicalSortUtil(res, visited);**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**// Create a graph given in the above diagram**

**Graph g(6);**

**g.addEdge(5, 2);**

**g.addEdge(5, 0);**

**g.addEdge(4, 0);**

**g.addEdge(4, 1);**

**g.addEdge(2, 3);**

**g.addEdge(3, 1);**

**cout << "All Topological sorts\n";**

**g.alltopologicalSort();**

**return 0;**

**}**

**Some Real Life Application Of Topological Sorting:**

One real time application I can think of is usage of topological sort for maven dependency resolution.

In maven build system , we provide dependencies of different modules in pom.xml. While resolving these dependency , it automatically does transitive dependency resolution also.

Let say ,

You have a module A . In its pom , you specify dependency of B and C.

Module B has dependency of D.

So it would look like A->b , A->c , b->d.

You can visualize each module as a node in graph , and their dependency as a directed edge. while build , maven must be using topological sort on these modules to correctly resolve dependencies.