<https://efficientcodeblog.wordpress.com/2017/11/28/topological-sort-dfs-bfs-and-dag/>

Today we will talk about **Topological Sorting** of a **Directed Acyclic Graph (DAG)**. But before that let us first refresh our memory about some of the important characteristics of **Depth First Search (DFS)** and **Breadth First Search (BFS)**:

***DFS*** and ***BFS*** are two graph search techniques.

Both ***DFS*** and ***BFS*** find all nodes findable, and nothing more.

Both ***DFS*** and ***BFS***search all findable nodes in **linear time**, i.e., O(E + V), where E = number of edges, V = number of vertices.

**A DFS will only store as much memory on the stack as is required for the longest root to leaf path in the tree. In other words, it space usage is  O(h) where  h is the height of the tree. A BFS on the other hand will queue every node at a fixed depth before visiting the next depth. If, for example, you have a complete binary tree, the number of leaves will be  n/2 = O(n) where  n is the number of nodes in the tree. Each of those  n/2 nodes will be queued up at some point. Many trees you will come across are moderately balanced. Even randomly built binary search trees have expected logarithmic height. In these cases  h = O(log n) < O(n) and DFS will therefore use less memory. So DFS is more memory efficient than BFS.**

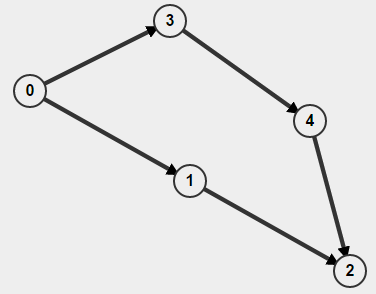
Depth first search is more memory efficient than breadth first search also because you can backtrack sooner. It is also easier to implement if you use the call stack but this relies on the longest path not overflowing the stack.

It’s also worth mentioning that ***BFS*** is often very easy to understand, by ***DFS*** is, in general, more efficient most of the time. But it totally depends on the kind of problem we are trying to solve.

***Topological Sort* is the most important operation on directed acyclic graphs or *DAG*s**. It orders the vertices on a line such that all directed edges go from left to right. Such an ordering cannot exist if the graph is not a ***DAG*** and contains one or more directed cycle(s), because there is no way we can keep going on the right on a line and still return to a vertex already visited (we are talking about **Back Edges**here).

If any of you thinking why I did not mention **cross-edges**along with **back-edges**, then reason for that is, BFS is used only in Undirected Graphs to determine whether there is a cycle or loop in it by examining if there is a cross-edge. Showing that there is a cross-edge while doing a BFS on Directed graph does not prove that the Directed Graph has a cycle.

If your graph is *directed* then you have to not just remember if you have visited a node or not, but also how you got there. Otherwise you might think you have found a cycle but in reality all you have is two separate paths A->B but that doesn’t mean there is a path B->A. For example,



If you do BFS starting from 0, it will detect as cycle is present but actually there is no cycle. With a depth first search you can mark nodes as visited as you descend and unmark them as you backtrack.

**Every DAG has at least one Topological Ordering.**

**Motivation for the Topological Sort:**Topological Sort is useful in scheduling tasks where precedence ordering matters, i.e., one task needs to be done before starting another. Topological sort can sequence tasks while respecting all sequence constraints without any conflict.

**A real world scenario**: In most academic programs there are prerequisite courses for taking a specific course. The prerequisite course(s) needs to be completed before taking the course. Topological sorting can give you the sequence in which different courses can be taken by students so that they can complete the pre-requisite courses before taking a course.

**When Does A Directed Graph Have A Topological Ordering ?**

**There is a very clear necessary condition for a directed graph to have a topological sorting, and that is it better be acyclic. So the graph needs to be a DAG (Directed Acyclic Graph). Put in a different way, one of the ways to prove that a graph is a DAG is to show that it has a topological ordering.**

Graph G has a directed cycle => G has no Topological Ordering

It is because if a graph G has a directed cycle that means it would have a back-edge, which means when trying to form a topological ordering from at least one of the edges we will have a edge back to one of the earlier edges, or even to the starting edge, in some cases, depending on the graph. It has been seen that having a directed cycle is the only obstruction for having a topological sorting. If you don’t have a directed cycle in a graph G then then you are guaranteed to have a topological ordering for the graph G.

It has been seen that **having a directed cycle** is the **only obstruction** for having a **topological sort**. If you don’t have a directed cycle in a graph G then then you are guaranteed to have a topological ordering for the graph G.

DFS is a slick and highly efficient way to compute topological sort of a graph in linear time: O(E + V).

**Sink Vertex**: While trying to implement topological sort using DFS it is important to know what ***sink vertices***are. **Sink vertex is the vertex which has no outbound edges**. **A DAG has to have at least one sink vertex**.  If we don’t have a sink vertex then that means that every edge will have at least one outgoing vertex and since we have finite vertices in our graph then by ***pigeon hole principle*** that means that, we will be visiting one of the already visited vertices, which implies we would have a directed  cycle. In fact you must convince yourself that one of the sink vertices must be the end vertex of the topological sort. Sink vertices are the only candidates for the final/last vertex position in the topological ordering.

*Corollary*: A directed graph with no sink vertex must have cycle.

**DFS Implementation  Mechanism for Topological Sort:**

While doing a DFS we will search for the ***sink vertices***, and as we get a sink vertex we will disconnect that vertex from the graph and will then backtrack and search for the next sink vertex along the search path, until all the vertices in the graph are visited. This way we will get all the vertices in totally reverse order of that of what a topological ordering should be. So we would need to reverse the order to get the topological sort.

**Why can’t the DFS result be the same as the *topological sort* and why would we need to find the sink vertices while doing a DFS to have the *topological ordering*?**

This is because in DFS we print the nodes as we see them, which means when we print a node, it has just been discovered but not yet processed, which means it is in the ***Visiting*** state. So DFS gives the order in which the nodes enter the ***Visiting*** state and not the ***Visited*** state. For topological sorting we need to have the order in which the nodes are completely processed, i.e., the order in which the nodes are marked as ***Visited***. Because when a node is marked ***Visited*** then all of its child node have already been processed, so they would be towards the right of the child nodes in the topological sort, as it should be (remember that we are going to reverse the order in which the nodes are marked visited, so that the child nodes go towards the right).

**Note: *'Visiting' is used for backtracking***

**DFS on same path backtrack algorithm:**

public class Vertex {

    public int val;

    public boolean visiting;

    public boolean visited;

    public List childNodes;

}

// returns null if no topological sort is possible

public List topological\_sort\_by\_dfs(List<Vertex> graph) {

    Stack stack = new Stack();

    List<Vertex> result = new ArrayList<>();

    for (Vertex vertex : graph) {

        if (!vertex.visited) {

            boolean dfs\_result = dfs(vertex, stack);

            // if cycle found then there is no topological sort possible

            if (!dfs\_result) {

                return null;

            }

        }

    }

    for (Vertex vertex : stack) {

        result.add(stack.pop());

    }

    return result

}

private boolean dfs(Vertex vertex, Stack<Vertex> stack) {

    vertex.visiting = true;

    for (Vertex childNode : vertex.childNodes) {

        if (!childNode.visited) {

            // check for back-edge, i.e., cycle

            if (childNode.visiting) {

                return false;

            }

            boolean childResult = dfs(childNode, stack);

            if (!childResult) {

                return false;

            }

        }

    }

    // now that you have completely visited all the

    // childNodes of the vertex, push the vertex in the stack

    stack.push(vertex);

    // this vertex is processed (all its descendents, i.e,

    // the nodes that are dependent on

    // this vertex along with this vertex itself have been visited

    // and processed). So mark this vertex as Visited.

    vertex.visited = true;

    // mark vertex as visiting as false since we

    // have completed visiting all the subtrees of

    // the vertex, including itself. So the vertex is no more

    // in visiting state because it is done visited.

    // Another reason is (the critical one) now that all the

    // descendents of the vertex are visited, any future

    // inbound edge to the vertex won't be a back edge anymore.

    vertex.visiting = false;

    return true;

}

Topological Sort by BFS:

*Topological Sort* can also be implemented by *Breadth First Search* as well. In DFS implementation of Topological Sort we focused on sink vertices, i.e., vertices with zero out-going edges, and then at last had to reverse the order in which we got the sink vertices (which we did by using a stack, which is a Last In First Out data structure). In BFS implementation of the Topological sort we do the opposite: **We look for for edges with no inbound edges**. And consequently in BFS implementation we don’t have to reverse the order in which we get the vertices, since we get the vertices in order of the topological ordering. We use First-In-First-Out data structure Queue in this implementation. We just search for the vertices with zero inbound edges and put them in the queue till we have processed all the vertices of the graph. Polling vertices from the queue one by one give the topological sort of the graph.

**BFS indegree=0 Algorithm:**

// The input graph is represented by a list of edges,

// not adjacency matrices.

// the edge goes from edge[0] towards edge[1]

// n\_vertices is the total number of vertices present in the graph,

// marked from 1 to n\_vertices

// The function below return an ordered list of the vertices of the

// given graph

// in topological ordering. Since it is an ordered list it

// will return the

// topological sorted vertices in correct order.

// returns null if no topological sort is possible

public List topological\_sort\_bfs(int[][] graph, int n\_vertices) {

  int[] indegree = new int[n\_vertices + 1]; // we are not using the index 0

  // compute the indegree of all the vertices in the graph

  for (int[] edge : graph) {

        indegree[edge[1]]++;

    }

    Queue currentVerticesWithNoInboundEdge = new LinkedList<>();

    // initialize the queue with all the vertices with no inbound edges

    for (int index = 1; index < n\_vertices; index++) {

        if (indegree[index] == 0) {

            currentVerticesWithNoInboundEdge.add(index);

        }

    }

    List<Integer> result = new ArrayList<>();

    while (!queue.isEmpty()) {

        int vertex1 = queue.poll();

        result.add(vertex1);

        // now disconnect vertex1 from the graph

        // and decrease the indegree of the other end of the edge by 1

        for (int[] edge : graph) {

            if (edge[0] == vertex1) {

                indegree[edge[1]]--;

                if (indegree[edge[1]] == 0) {

                    queue.add(edge[1]);

                }

            }

        }

    }

    // check if the graph had a cycle / loop

    // if the resultset does not contain all the vertices that means

    // there is at least one cycle in the graph

    // that's why the indegree of those vertices participating in the

    // loop could not be made 0 by decrementing.

    if (result.size() != n\_vertices) {

    return null;

    }

    return result;

}

**Refer to Deepseek**

**What is cross-edge and back-edge in graph ?**

In graph theory, **cross-edges** and **back-edges** are concepts that arise when performing a Depth-First Search (DFS) traversal of a graph. These terms are used to classify the edges of the graph based on how they are encountered during the DFS traversal. They are particularly important in understanding the structure of the graph, especially in directed and undirected graphs.

**1. Tree Edge:**

A **tree edge** is an edge that is part of the DFS tree. It is the edge that discovers a new vertex during the DFS traversal.

In other words, if you move from vertex u to vertex v and v is being visited for the first time, then (u, v) is a tree edge.

**2. Back Edge:**

A **back edge** is an edge that connects a vertex to an **ancestor** in the DFS tree (other than its immediate parent).

Back edges indicate the presence of a **cycle** in the graph.

In a directed graph, a back edge is evidence of a cycle.

In an undirected graph, every edge is either a tree edge or a back edge (since there are no cross-edges or forward edges).

**Example**:

A → B → C

↑ ↓

└----D

**If DFS starts at A, the edge D → A is a back edge because A is an ancestor of D in the DFS tree.**

**3. Forward Edge:**

A **forward edge** is an edge that connects a vertex to a **descendant** in the DFS tree (other than its immediate child).

Forward edges are non-tree edges that go from a vertex to a descendant that has already been discovered.

**Example**:

A → B → C

↓ ↓

D E

If DFS starts at A, the edge A → E is a forward edge because E is a descendant of A in the DFS tree.

**4. Cross Edge:**

A **cross edge** is an edge that connects two vertices that are not ancestors or descendants of each other in the DFS tree.

Cross edges can only occur in **directed graphs** (they do not exist in undirected graphs).

They typically connect vertices in different branches of the DFS tree.

**Example:**

A → B → C

↓

D → E

If DFS starts at A, the edge D → E is a cross edge because D and E are in different branches of the DFS tree and neither is an ancestor of the other.

**Key Differences Between Back Edge and Cross Edge:**

|  |  |  |
| --- | --- | --- |
| Feature | Back Edge | Cross Edge |
| **Direction** | Connects to an ancestor in the DFS tree. | Connects to a node that is neither an ancestor nor a descendant. |
| **Cycle** | Indicates a cycle in the graph. | Does not indicate a cycle. |
| **Graph Type** | Exists in both directed and undirected graphs. | Exists only in directed graphs. |
| **Example** | A → B → C → A  (cycle). | A → B  and  C → D  (no direct relationship). |

**Visualization:**

Consider the following directed graph:

A → B → C

↓ ↓

D → E

Tree Edges: A → B, B → C, A → D, D → E

Back Edge: E → B (if it exists, it creates a cycle)

Forward Edge: A → E (if it exists, E is a descendant of A)

Cross Edge: C → E (if it exists, C and E are in different branches)

**Practical Applications:**

**Cycle Detection:**

**Back edges are used to detect cycles in a graph. If a back edge is found during DFS, the graph contains a cycle.**

**Topological Sorting:**

**Cross edges and back edges are important in topological sorting of directed acyclic graphs (DAGs). A back edge indicates that the graph is not a DAG.**

Strongly Connected Components (SCCs):

Cross edges and back edges are used in algorithms like Tarjan's algorithm to find SCCs in a directed graph.

**Example Code to Classify Edges in DFS:**

Here’s an example of how you can classify edges during a DFS traversal in a directed graph:

import java.util.\*;

class Graph {

private int V;

private List<List<Integer>> adj;

public Graph(int V) {

this.V = V;

adj = new ArrayList<>();

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

adj.add(new ArrayList<>());

}

}

public void addEdge(int u, int v) {

adj.get(u).add(v);

}

public void classifyEdges() {

boolean[] visited = new boolean[V];

int[] discoveryTime = new int[V];

int[] finishTime = new int[V];

int[] parent = new int[V];

Arrays.fill(parent, -1);

int time = 0;

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

if (!visited[i]) {

dfs(i, visited, discoveryTime, finishTime, parent, time);

}

}

}

private void dfs(int u, boolean[] visited, int[] discoveryTime, int[] finishTime, int[] parent, int time) {

visited[u] = true;

discoveryTime[u] = ++time;

for (int v : adj.get(u)) {

if (!visited[v]) {

parent[v] = u;

System.out.println(u + " → " + v + " : Tree Edge");

dfs(v, visited, discoveryTime, finishTime, parent, time);

} else if (discoveryTime[u] > discoveryTime[v] && finishTime[v] == 0) {

System.out.println(u + " → " + v + " : Back Edge");

} else if (discoveryTime[u] < discoveryTime[v]) {

System.out.println(u + " → " + v + " : Forward Edge");

} else {

System.out.println(u + " → " + v + " : Cross Edge");

}

}

finishTime[u] = ++time;

}

}

public class Main {

public static void main(String[] args) {

Graph g = new Graph(6);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 3);

g.addEdge(3, 4);

g.addEdge(4, 1); // Back edge

g.addEdge(4, 5); // Forward edge

g.addEdge(5, 2); // Cross edge

g.classifyEdges();

}

}

**Summary:**

Back Edge: Indicates a cycle; connects to an ancestor.

Cross Edge: Connects to a node that is neither an ancestor nor a descendant.

These concepts are crucial for understanding graph traversal and properties like cycles, topological sorting, and strongly connected components.