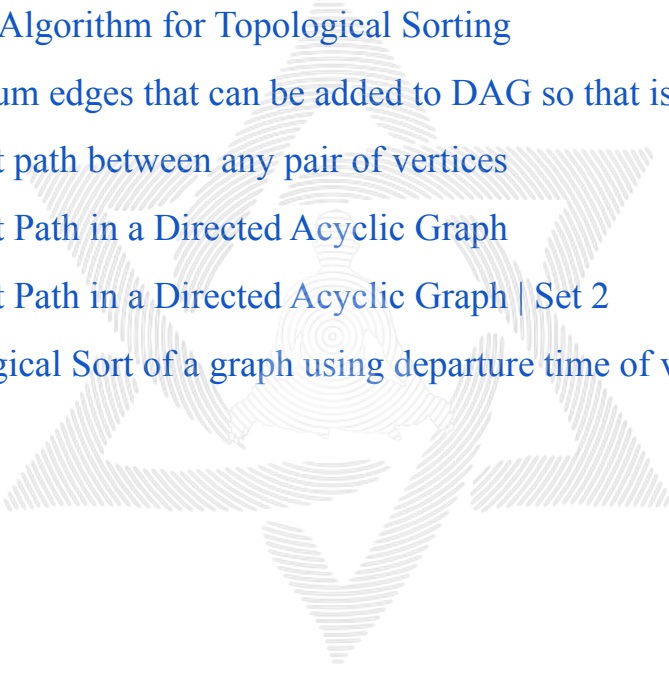


Graphs

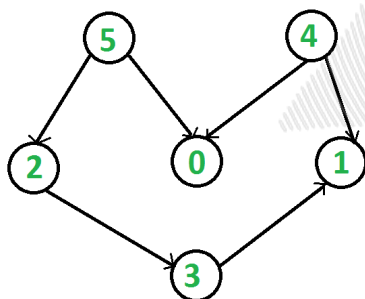
Lesson 6

1. Topological Sorting
 2. All topological sorts of a Directed Acyclic Graph
 3. Kahn's Algorithm for Topological Sorting
 4. Maximum edges that can be added to DAG so that it remains DAG
 5. Longest path between any pair of vertices
 6. Longest Path in a Directed Acyclic Graph
 7. Longest Path in a Directed Acyclic Graph | Set 2
 8. Topological Sort of a graph using departure time of vertex
- 

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge $u \rightarrow v$, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, the topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with an in-degree as 0 (a vertex with no incoming edges).



In [DFS](#), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike [DFS](#), the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, the DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sort.

Algorithm to find Topological Sorting

We can modify DFS to find the Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print the contents of the stack. Note that a vertex is pushed to the stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

DFS approach/recursive:-<https://sapphireengine.com/@/g5hhp2>

Complexity Analysis:

Time Complexity: $O(V+E)$.

Auxiliary space: $O(V)$.

The extra space is needed for the stack.

```
1. void top(int src , vector<int>&vis , vector<int> adj[] , vector<int>&v)
2. {
3.     if(vis[src])
4.         return;
5.     vis[src]=1;
6.     for(int i=0;i<adj[src].size();i++)
7.     {
8.         if(!vis[adj[src][i]])
9.             top(adj[src][i] , vis , adj , v);
10.    }
11.    v.push_back(src);
```

```
12.     }
13.     //Function to return list containing vertices in Topological order.
14.     vector<int> topoSort(int V, vector<int> adj[])
15.     {
16.         vector<int> v(V), vis(V,0);
17.         for(int i=0; i<V; i++)
18.         {
19.             if(!vis[i])
20.             {
21.                 top(i, vis, adj, v);
22.             }
23.         }
24.         reverse(v.begin(), v.end());
25.         return v;
26.         // code here
27.     }
```

Kahn's algorithm for Topological Sorting

Compute in-degree for each of the vertices present in the DAG and initialize the count of visited nodes as 0. Pick all the vertices with in-degree as 0 and add them into a queue. Remove a vertex from the queue and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighbouring nodes.
3. If the in-degree of a neighbouring node is reduced to zero, then add it to the queue.

Repeat until the queue is empty. If the 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.

Time Complexity: $O(V+E)$.

<https://sapphireengine.com/@/ktoaan>

```
1.  vector<int> topoSort(int V, vector<int> adj[])
2.      {
3.          int indegree[V]={0};
4.          vector<int>vis;
5.          for(int i=0;i<V;i++)
6.              {
7.                  for(int j=0;j<adj[i].size();j++)
8.                      {
9.                          indegree[adj[i][j]]++;
10.                     }
```

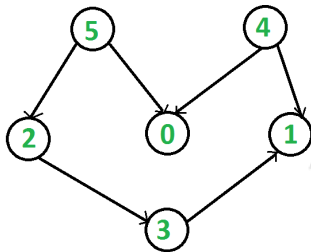
```
11.     }
12.
13.     queue<int>q;
14.     for(int i=0;i<V;i++)
15.     {
16.         if(indegree[i]==0)
17.             q.push(i);
18.     }
19.     int cnt=0;
20.     while(!q.empty())
21.     {
22.         int x=q.front();
23.         q.pop();
24.         vis.push_back(x);
25.         for(int i=0;i<adj[x].size();i++)
26.         {
27.             indegree[adj[x][i]]--;
28.             if(indegree[adj[x][i]]==0)
29.                 q.push(adj[x][i]);
30.         }
31.         cnt++;
32.     }
33.     if(cnt!=V)
34.     {
35.         vis.clear();
36.         return vis;
37.     }
38.     return vis;
39. }
```

All Topological Sorts of a Directed Acyclic Graph

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Given a DAG, print all topological sorts of the graph.

For example, consider the below graph.



All topological sorts of the given graph are:

4 5 0 2 3 1

4 5 2 0 3 1

4 5 2 3 0 1

4 5 2 3 1 0

5 2 3 4 0 1

5 2 3 4 1 0

5 2 4 0 3 1

5 2 4 3 0 1

5 2 4 3 1 0

5 4 0 2 3 1

5 4 2 0 3 1

5 4 2 3 0 1

5 4 2 3 1 0

In a Directed acyclic graph many times we can have vertices which are unrelated to each other because of which we can order them in many ways. These various topological sorting is important in many cases, for example if some relative weight is also available between the vertices, which is to minimize then we need to take care of relative ordering as well as their relative weight, which creates the need of checking through all possible topological ordering.

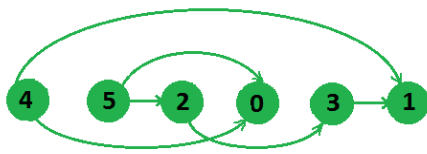
We can go through all possible ordering via backtracking , the algorithm step are as follows :

1. Initialize all vertices as unvisited.
2. Now choose a vertex which is unvisited and has zero indegree and decrease the indegree of all those vertices by 1 (corresponding to removing edges) . Now add this vertex to the result and call the recursive function again and backtrack.
3. After returning from function reset values of visited, result and indegree for enumeration of other possibilities.

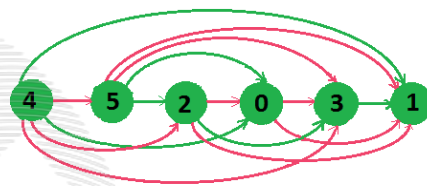
<https://sapphireengine.com/@/j923v2>

Maximum edges that can be added to DAG so that it remains DAG

A DAG is given to us, we need to find the maximum number of edges that can be added to this DAG, after which the new graph still remains a DAG that means the reformed graph should have a maximal number of edges, adding even a single edge will create a cycle in the graph.



Input DAG



DAG with maximum edge addition

The Output for above example should be following edges in any order. 4-2, 4-5, 4-3, 5-3, 5-1, 2-0, 2-1, 0-3, 0-1

<https://sapphireengine.com/@jm6dv2>