Experiment No. 7

Title: Implementation of Topological Sorting

Batch: B-4 **Roll No: 16010422234** **Name: Chandana Ramesh Galgali**

Experiment No.:7

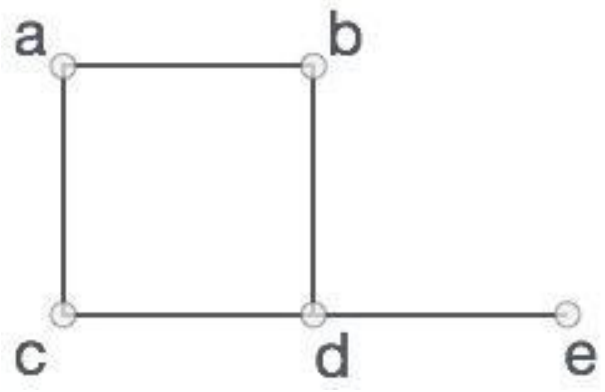
Aim: To study Graph Theory and graph traversal for implementation of problem statements that are based on BFS, DFS & topological sort and verify given test cases.

Resources needed: Text Editor, C/C++ IDE

Theory:

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

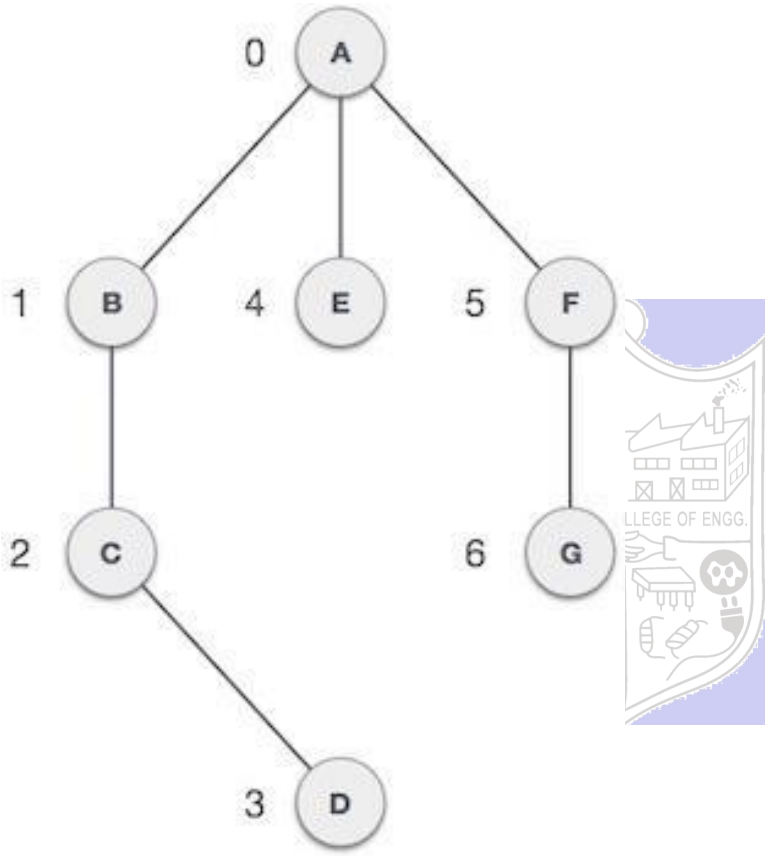
Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We

can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



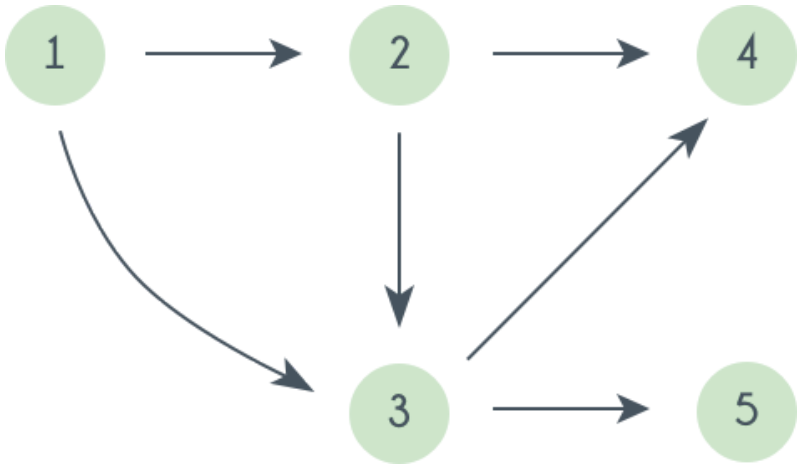
Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph.

Topological sorting of vertices of a **Directed Acyclic Graph** is an ordering of the vertices v1,v2,...vn in such a way, that if there is an edge directed towards vertex vj from vertex vi, then vi comes before vj.

For example consider the graph given below:



A topological sorting of this graph is: 1 2 3 4 5. There are multiple topological sorting possible for a graph. For the graph given above one another topological sorting is: 1 2 3 5 4

In order to have a topological sorting the graph must not contain any cycles. In order to prove it, let's assume there is a cycle made of the vertices v1,v2,v3...vn. That means there is a directed edge between vi and vi+1 ($1 \leq i < n$) and between vn and v1. So now, if we do topological sorting then vn must come before v1 because of the directed edge from vn to v1. Clearly, vi+1 will come after vi, because of the directed from vi to vi+1, that means v1 must come before vn. Well, clearly we've reached a contradiction, here. So topological sorting can be achieved for only directed and acyclic graphs.

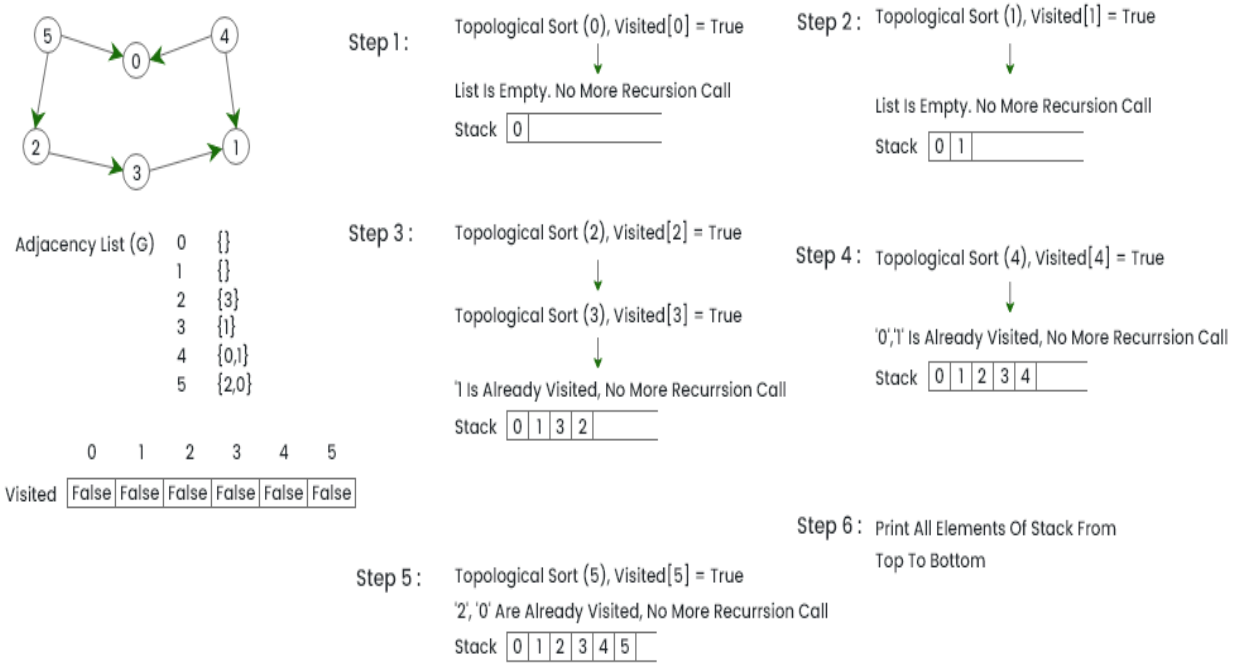
Mechanism behind Topological Sorting Graph:

Le'ts see how we can find a topological sorting in a graph. So basically we want to find a permutation of the vertices in which for every vertex vi, all the vertices vj having edges coming out and directed towards vi comes before vi. We'll maintain an array T that will denote our topological sorting. So, let's say for a graph having N vertices, we have an array in degree[] of size N whose ith element tells the number of vertices which are not already inserted in T and there is an edge from them incident on vertex numbered i. We'll append vertices vi to the array T, and when we do that we'll decrease the value of in degree[vj] by 1 for every edge from vi to vj. Doing this will mean that we have inserted one vertex having edge directed towards vj. So at any point we can insert only those vertices for which the value of in degree[] is 0.

Algorithm –Topological Sorting

1. Create a stack to store the nodes. We use a temporary stack.
2. Initialize visited array of size N to keep the record of visited nodes.
   We don't print the vertex immediately
3. We first recursively call topological sorting for all its adjacent vertices, and then push it to a stack.
4. Run a loop from 0 till N :
    1. if the node is not marked True in visited array then call the recursive function for topological sort and perform the following steps:
    2. Mark the current node as True in the visited array.
    3. Run a loop on all the nodes which has a directed edge to the current node
    4. if the node is not marked True in the visited array:
       Recursively call the topological sort function on the node
    5. Push the current node in the stack.
5. Print all the elements in the stack.

Illustration of Topological Sorting of Graph



Activity:
Consider the following problem statement and other information provided along with it:

Problem Statement: Topological Sorting

You are given a directed acyclic graph (DAG) with N nodes and M edges. Your task is to perform a topological sorting of the graph.

Input:
The first line contains two integers N and M representing the number of nodes and edges in the graph, respectively. $(1 \leq N \leq 10^5,\ 0 \leq M \leq 10^5)$. The next M lines each contain two integer's u and v indicating a directed edge from node u to node v.
Expected:
Defines a topological sort function that takes the graph as input and returns a list of nodes in topological order if it's possible, otherwise, it returns "IMPOSSIBLE". The dfs function performs the depth-first search, marking nodes as visited and backtracking if it detects a cycle.

Output:
If the graph has a valid topological sorting, output a single line containing N space-separated integers representing the nodes in a topologically sorted order.
If the graph has no valid topological sorting (i.e., it contains cycles), output "IMPOSSIBLE".

Constraints:
All node numbers are in the range 1 to N
.

Example:
Input:
6 6
1 2
2 3
2 4
3 5
4 5
6 4

Output:
1 2 3 4 5 6

Program:

```
def topological_sort(n, edges):
    from collections import defaultdict, deque

    # Graph initialization
    graph = defaultdict(list)
    in_degree = [0] * (n + 1)
    visited = [False] * (n + 1)
    stack = []

    # Building the graph
```

```
    for u, v in edges:
        graph[u].append(v)
        in_degree[v] += 1

    # Function to perform DFS
    def dfs(v):
        visited[v] = True
         stack.append(v)  # Use stack to hold the current path for cycle
detection
        for neighbor in graph[v]:
            if neighbor in stack:  # Cycle detection
                return True
            if not visited[neighbor]:
                if dfs(neighbor):
                    return True
        stack.pop()
           result.append(v)  # Append to result if all descendants are
processed
        return False

    result = []

    # Check each node if not already visited
    for v in range(1, n + 1):
        if not visited[v]:
            if dfs(v):
                return "IMPOSSIBLE"  # Cycle detected

       return result[::-1]  # Reverse the result to get the correct
topological order

# Sample input directly within the script
n, e = map(int,input().split())
```

```
edges = []
for i in range(0, e):
    ui, vi =  map(int, input().split())
    edges.append((ui, vi))

# Perform topological sorting
sorted_order = topological_sort(n, edges)
if isinstance(sorted_order, list):
    print(" ".join(map(str, sorted_order)))
else:
    print(sorted_order)
```

Output:

```
PS C:\Users\chand\Downloads\IV SEM\CPL\LAB>
s/IV SEM/CPL/LAB/exp7.py"
6 6
1 2
2 3
2 4
3 5
4 5
6 4
6 1 2 4 3 5
PS C:\Users\chand\Downloads\IV SEM\CPL\LAB>
```

Outcomes: Understand the Graphs, related algorithms, efficient implementation of those algorithms and applications

---

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

In this experiment, we delved into Graph Theory, mastering graph traversal techniques such as BFS, DFS, and particularly topological sorting, which is pivotal for scheduling and dependency tasks. By implementing and verifying these algorithms, we enhanced our

understanding and skills in handling directed acyclic graphs and cycle detection, crucial for robust algorithm design. This experiment bolstered our theoretical knowledge, improved our problem-solving capabilities, and prepared us for complex real-world applications and further advanced studies in computer science.

References:

1. https://www.hackerearth.com/practice/algorithms/graphs/topological-sort/practice-problems/algorithm/lonelyisland-49054110/
2. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
3. Antti Laaksonen, “Guide to Competitive Programming”,Springer,2018
4. Gayle Laakmann McDowell,” Cracking the Coding Interview”,CareerCup LLC,2015
5. Steven S. Skiena Miguel A. Revilla,”Programming challenges, The Programming Contest Training Manual”, Springer, 2006
6. Antti Laaksonen, “Competitive Programmer’s Handbook”, Hand book, 2018
7. Steven Halim and Felix Halim, “Competitive Programming 3: The Lower Bounds of Programming Contests”, Handbook for ACM ICPC
