Experiment No. 6

Title: Implementation of problem based on Graph Theory

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

Experiment No.:6

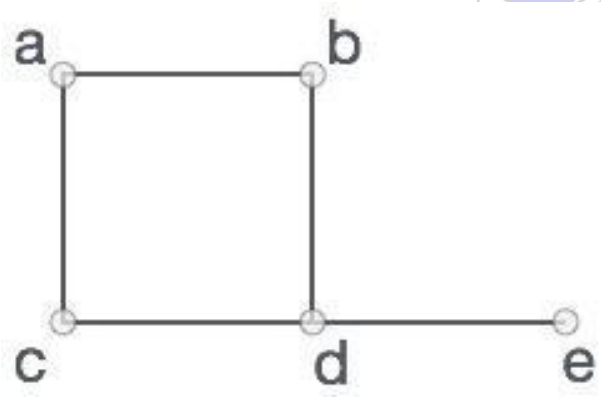
Aim: To study Graph Theory and graph traversal for implementation of problem statements that are based on BFS, DFS & topological sort and verify given test cases.

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}

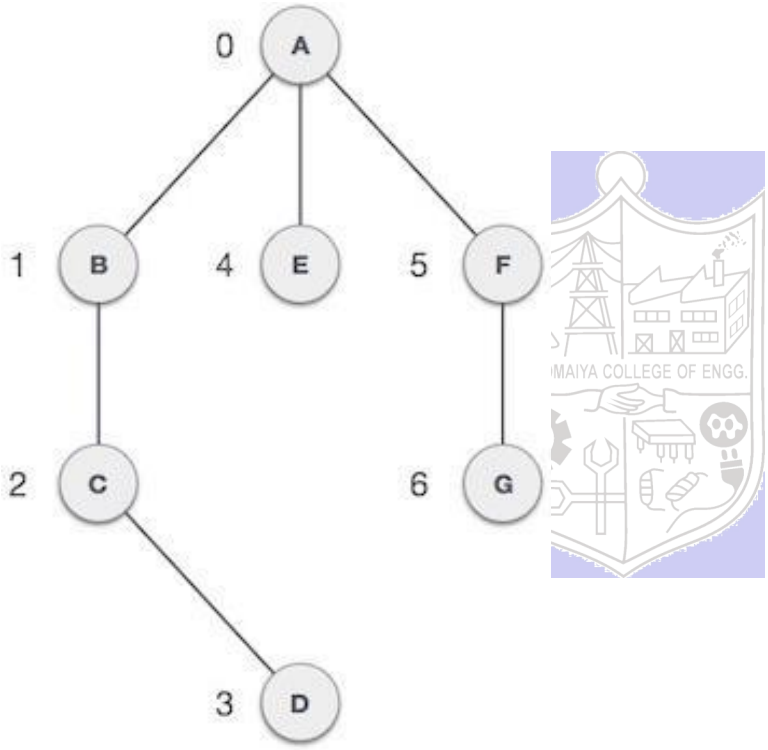
E = {ab, ac, bd, cd, de}

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.

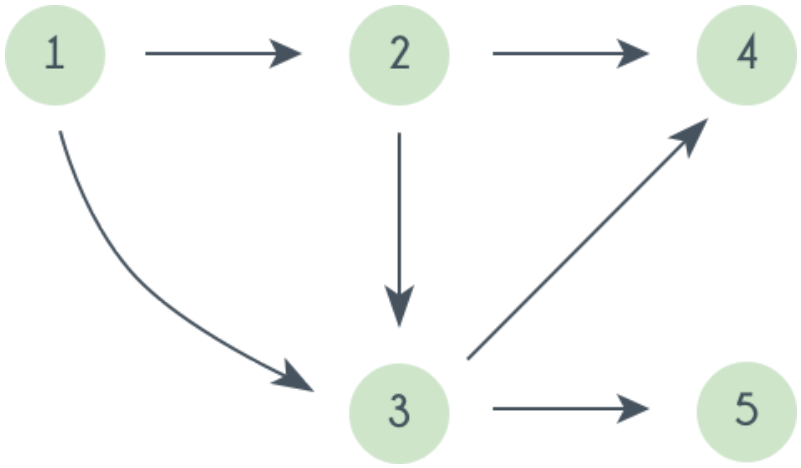


Basic Operations

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.

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.

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

Problem Statement: Lonely Island
There are many islands that are connected by one-way bridges, that is, if a bridge connects islands a and b, then you can only use the bridge to go from a to b but you cannot travel back by using the same. If you are on island a, then you select (uniformly and randomly) one of the islands that are directly reachable from a through the one-way bridge and move to that island.

You are stuck on an island if you cannot move any further. It is guaranteed that after leaving any island it is not possible to come back to that island.
Find the island that you are most likely to get stuck on. Two islands are considered equally likely if the absolute difference of the probabilities of ending up on them is $<=10^{-9}$.

Input format

First line: Three integers n (the number of islands), m (the number of one-way bridges), and r (the index of the island you are initially on)

Next m lines: Two integers ui and vi representing a one-way bridge from island ui to vi.

Output format

Print the index of the island that you are most likely to get stuck on. If there are multiple islands, then print them in the increasing order of indices (space separated values in a single line).

Input Constraints

1≤n≤200000
1≤m≤500000
1≤ui,vi,r≤n+

Sample Input	Sample Output
5 7 1 1 2 1 3 1 4 1 5 2 4 2 5 3 4	4

Program:

```
from collections import defaultdict
n, m, r = map(int,input().split())
edges = []
for i in range(0, m):
    ui, vi =  map(int, input().split())
    edges.append((ui, vi))
```

```
graph = defaultdict(list)
stuck_probability = defaultdict(lambda: 0)
for u, v in edges:
    graph[u].append(v)
def calculate_probabilities(node, probability):
    if len(graph[node]) == 0:
        stuck_probability[node] += probability
    else:
        next_prob = probability / len(graph[node])
        for next_node in graph[node]:
            calculate_probabilities(next_node, next_prob)
calculate_probabilities(r, 1)
max_prob = max(stuck_probability.values())
precision_limit = 1e-9
most_probable_sinks = [node for node, prob in stuck_probability.items()
                       if abs(prob - max_prob) < precision_limit]
most_probable_sinks.sort()
print(" ".join(map(str, most_probable_sinks)))
```

Output:

```
PS C:\Users\chand\Downloads\IV SEM\CPL\LAB>
s/IV SEM/CPL/LAB/exp6.py"
5 7 1
1 2
1 3
1 4
1 5
2 4
2 5
3 4
4
```

Test Result:

Score	Time (sec)	Memory (KiB)	Language
0	24.0758	92040	Python 3.8

Input	Result	Time (sec)	Memory (KiB)	Score	Your output	Correct output	Diff
Input #1	Accepted	0.828309	75216	4			
Input #2	Accepted	0.822502	73964	4			
Input #3	Accepted	0.650254	62760	4			
Input #4	Accepted	0.899831	71660	4			
Input #5	Accepted	0.860669	78984	4			
Input #6	Accepted	0.932076	70528	4			
Input #7	Accepted	0.738183	67944	4			
Input #8	Accepted	0.640905	62496	4			
Input #9	Accepted	1.720025	70028	4			
Input #10	Accepted	1.01312	82548	4			
Input #11	Accepted	0.960842	75500	4			
Input #12	Accepted	1.101474	92040	4			
Input #13	Accepted	0.944561	91272	4			
Input #14	Accepted	0.788004	73944	4			
Input #15	Accepted	0.948006	74476	4			
Input #16	Accepted	0.956896	74332	4			
Input #17	Accepted	0.714237	68456	4			
Input #18	Accepted	0.89228	81132	4			
Input #19	Accepted	0.786988	74200	4			
Input #20	Accepted	0.885873	70524	4			

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

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

The experiment effectively demonstrated the practical applications of Graph Theory through the implementation of BFS, DFS, and Topological Sort algorithms, affirming their unique strengths in graph-based problem-solving. By applying these traversal methods to various test cases, we validated their effectiveness in scenarios ranging from pathfinding to task scheduling. The outcomes reinforced the importance of selecting the appropriate traversal strategy to efficiently solve complex problems, encapsulating the experiment's aim and showcasing the versatility of graph algorithms in computational tasks.

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