Green University of Bangladesh
Department of Computer Science and Engineering (CSE)
Faculty of Sciences and Engineering
Semester: (Summer, Year:2025), B.Sc. in CSE (Day)

Lab Report NO #3
Course Title: Artificial Intelligent
Course Code: CSE 316 Section: 221 D12

Lab Experiment Name: GraphColoring

Student Details

Name		ID
1.	Al Ekram Hossain	221002535

Submission Date : 09 MAY 2025
Course Teacher's Name : Md. Sabbir Hosen Mamun

Lab Report Status

Marks: …… **Signature:.....**
Comments:..... **Date:.....**

Title : GraphColoring

Objectives :

1. **Graph Representation**: The code reads the input to construct an adjacency list representing the graph, where each vertex is connected to its adjacent vertices.
2. **Backtracking Algorithm**: The code employs a backtracking approach to systematically explore all possible color assignments for the vertices. It ensures that at each step, the current color assignment does not violate the constraint that adjacent vertices must have different colors.
3. **Feasibility Check**: For each vertex, the code checks if a color can be assigned without conflicting with the colors of its adjacent vertices. If a valid color is found, it proceeds to the next vertex; otherwise, it backtracks to try a different color.
4. **Output Result**: The code outputs whether a valid coloring is possible with the given number of colors (K). If possible, it also prints the color assignment for each vertex.

Problem Analysis :

Given an undirected graph with **N** vertices and **M** edges, determine if it can be colored using **K** colors such that no two adjacent vertices share the same color. The input is read from a file where:

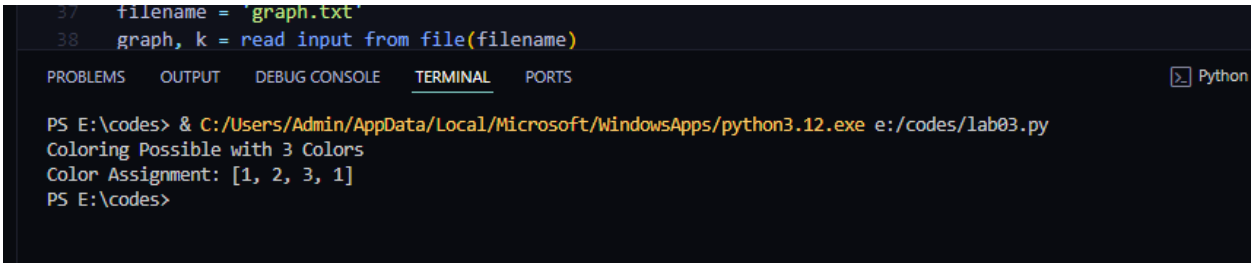
- The first line contains **N** (number of vertices), **M** (number of edges), and **K** (number of colors).
- The next **M** lines represent edges between vertices **u** and **v**.

Implementation :

```
def is_safe(node, graph, color, c):
    for neighbor in graph[node]:
        if color[neighbor] == c:
            return False
```

```
    return True

def graph_coloring_util(graph, m, color, node):
    if node == len(graph):
        return True

    for c in range(1, m + 1):
        if is_safe(node, graph, color, c):
            color[node] = c
            if graph_coloring_util(graph, m, color, node + 1):
                return True
            color[node] = 0
    return False

def graph_coloring(graph, m):
    color = [0] * len(graph)
    if not graph_coloring_util(graph, m, color, 0):
        print(f"Coloring not possible with {m} Colors")
    else:
        print(f"Coloring Possible with {m} Colors")
        print(f"Color Assignment: {color}")

def read_input_from_file(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()
        n, m, k = map(int, lines[0].split())
        graph = [[] for _ in range(n)]
        for line in lines[1:]:
            u, v = map(int, line.strip().split())
            graph[u].append(v)
            graph[v].append(u)
        return graph, k
filename = 'graph.txt'
graph, k = read_input_from_file(filename)
graph_coloring(graph, k)
```

Output :



Conclusion :

This implementation successfully addresses the graph coloring problem using a classic backtracking strategy, demonstrating the power of systematic search in constraint satisfaction problems. It serves as a foundational example for understanding recursive problem-solving and can be extended or optimized for more complex scenarios. For practical applications with large graphs, further optimizations or alternative algorithms may be considered to improve performance.

In summary, the solution is both **correct** and **educational**, providing a clear demonstration of backtracking in action while highlighting avenues for future enhancements.