CSP: GRAPH-COLORING

```
def is_safe(graph, colors, vertex, color):
    for neighbor in graph[vertex]:
        if colors[neighbor] == color:
            return False
    return True


def graph_coloring(graph, m):
    n = len(graph)
    colors = [-1] * n  # -1 indicates no color assigned

    def backtrack(vertex):
        if vertex == n:
            return True  # All vertices are colored

        for color in range(m):
            if is_safe(graph, colors, vertex, color):
                colors[vertex] = color
                if backtrack(vertex + 1):
                    return True
                colors[vertex] = -1  # Backtrack
        return False

    if backtrack(0):
        return colors
    else:
        return None
```

 # Example usage

```
if __name__ == "__main__":
    # Graph represented as adjacency list
    # Example: A triangle graph (3 vertices, each connected to the other two)
    graph = {
        0: [1, 2],
        1: [0, 2],
        2: [0, 1]
    }

    m = 3  # Number of colors

    result = graph_coloring(graph, m)

    if result:
        print(f"Graph successfully colored using {m} colors:")
        for vertex, color in enumerate(result):
            print(f"Vertex {vertex} → Color {color}")
    else:
        print(f"Cannot color the graph using {m} colors.")
```

```
Graph successfully colored using 3 colors:
Vertex 0 → Color 0
Vertex 1 → Color 1
Vertex 2 → Color 2
```

Possible Viva Questions on This Practical:

1. **What is the Graph Coloring Problem?**
    - The Graph Coloring Problem involves assigning colors to the vertices of a graph in such a way that no two adjacent vertices share the same color. The goal is to minimize the number of colors used.
2. **What is Backtracking and how does it apply to this problem?**
    - Backtracking is a method of solving problems incrementally, where a solution is built step by step and if at any point a partial solution is found to be invalid, it is abandoned (backtracked). In this problem, backtracking is used to explore all possible color assignments and abandon those that lead to invalid configurations.
3. **What is the base case in the graph_coloring_util function?**
    - The base case is when node == len(graph), meaning all the nodes have been successfully colored. This indicates a valid solution.
4. **What does the function is_safe do?**
    - The function is_safe checks whether it is safe to assign a given color to a node. It ensures that the color does not conflict with any of the adjacent nodes (i.e., no two adjacent nodes share the same color).
5. **What is the time complexity of this backtracking approach?**
    - The time complexity is **O(m^n)**, where n is the number of vertices in the graph and m is the number of colors. This is because the algorithm may try all m colors for each of the n nodes.
6. **How does the algorithm handle backtracking?**
    - If placing a color on a node results in an invalid configuration (i.e., a conflict with an adjacent node), the algorithm "backtracks" by removing the color from the node and trying the next possible color.
7. **Can this algorithm handle large graphs?**
    - The algorithm can handle small to moderately sized graphs, but it may not be efficient enough for very large graphs due to its exponential time complexity. For larger graphs, heuristic or approximation algorithms might be more suitable.
8. **What would happen if the number of colors (m) is less than the chromatic number of the graph?**

- If the number of colors is less than the chromatic number of the graph, the algorithm will not be able to find a valid coloring, and the message "No solution exists with the given number of colors" will be printed.

9. **What is the chromatic number of a graph?**
    - The chromatic number of a graph is the minimum number of colors required to color the vertices of the graph such that no two adjacent vertices have the same color.
10. **What is the significance of using an adjacency matrix in this solution?**
    - The adjacency matrix is used to represent the graph. It allows for efficient checking of whether two nodes are adjacent by simply checking if graph[i][j] == 1 for nodes i and j.