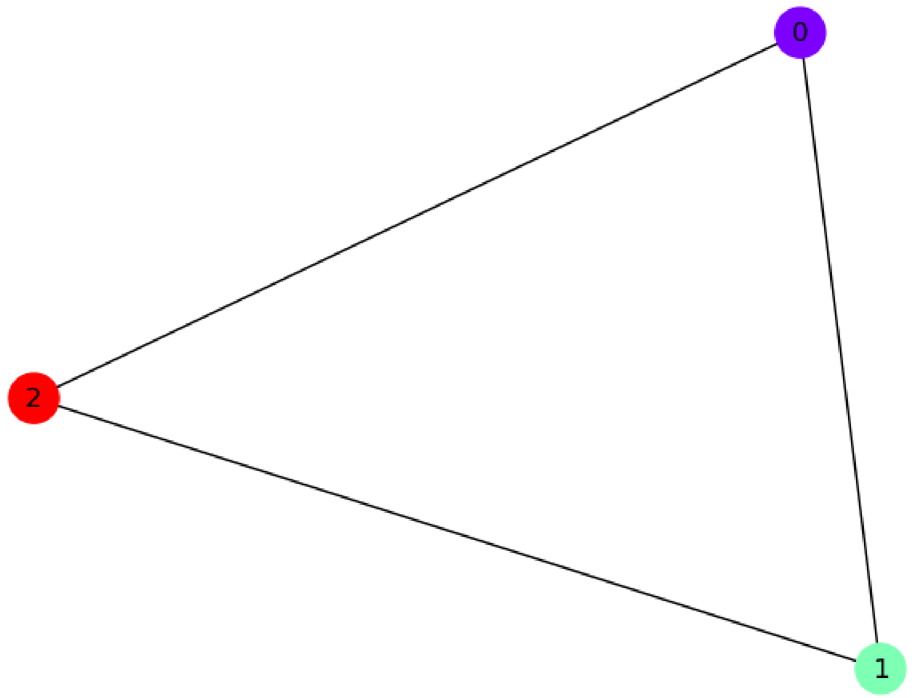
In [40]:

```
import networkx as nx
import matplotlib.pyplot as plt

def is_safe(graph, v, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True

def graph_coloring_backtracking(graph, m, color, v):
    if v == len(graph):
        return True

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

    return False

def branch_and_bound(graph, m, color, v, best_solution):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if is_safe(graph, v, color, c):
            color[v] = c
            if branch_and_bound(graph, m, color, v + 1, best_solution):
                return True
            color[v] = 0

    return False

def solve_graph_coloring(n, edges, m):
    graph = [[0] * n for _ in range(n)]

    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1

    color = [0] * n

    print("Solving using Backtracking...")
    if graph_coloring_backtracking(graph, m, color, 0):
        print("Solution Found:", color)
    else:
        print("No solution exists with given number of colors")

    best_solution = [0] * n
    print("Solving using Branch and Bound...")
    if branch_and_bound(graph, m, best_solution, 0, color):
        print("Solution Found:", best_solution)
    else:
        print("No solution exists with given number of colors")

    return color

def draw_graph(n, edges, color):
    G = nx.Graph()
    for i in range(n):
        G.add_node(i)
    for u, v in edges:
        G.add_edge(u, v)

    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color=color, cmap=plt.cm.rainbow, edge_color='black',
    plt.show()

if __name__ == "__main__":
    n = int(input("Enter number of vertices: "))
    e = int(input("Enter number of edges: "))
    edges = []
    for _ in range(e):
        u, v = map(int, input("Enter edge (u v): ").split())
```

```
        edges.append((u, v))
    m = int(input("Enter number of colors: "))

    color = solve_graph_coloring(n, edges, m)
    draw_graph(n, edges, color)
```

```
Enter number of vertices:  3
Enter number of edges:  3
Enter edge (u v):  0 1
Enter edge (u v):  1 2
Enter edge (u v):  2 0
Enter number of colors:  3

Solving using Backtracking...
Solution Found: [1, 2, 3]
Solving using Branch and Bound...
Solution Found: [1, 2, 3]
```

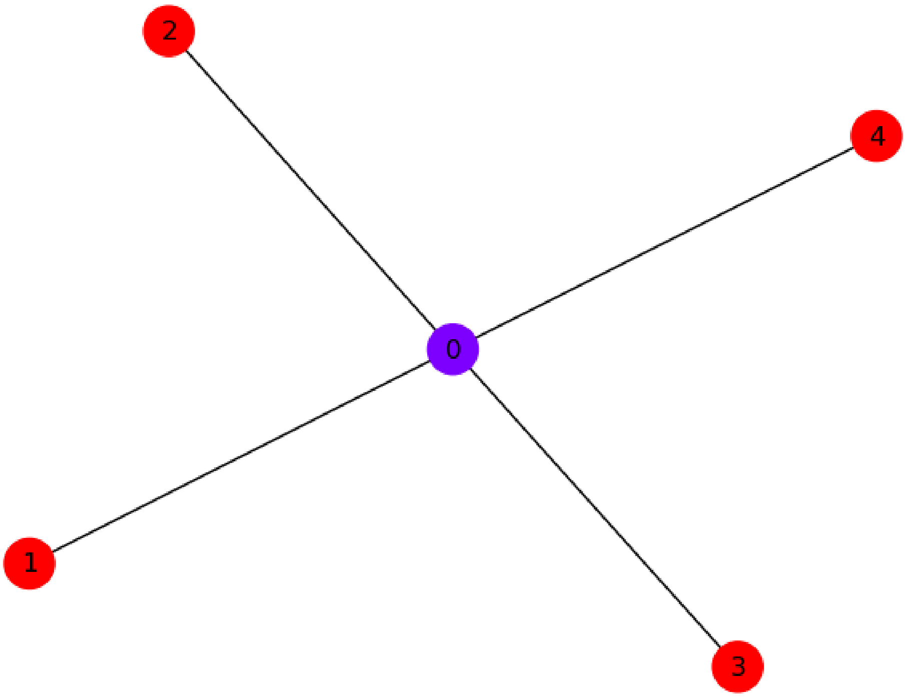


In [30]:

```python
import networkx as nx
import matplotlib.pyplot as plt

def is_safe(graph, v, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True

def graph_coloring_backtracking(graph, m, color, v):
    if v == len(graph):
        return True

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

    return False

def branch_and_bound(graph, m, color, v, best_solution):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if is_safe(graph, v, color, c):
            color[v] = c
            if branch_and_bound(graph, m, color, v + 1, best_solution):
                return True
            color[v] = 0

    return False

def solve_graph_coloring(n, edges, m):
    graph = [[0] * n for _ in range(n)]

    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1

    color = [0] * n

    print("Solving using Backtracking...")
    if graph_coloring_backtracking(graph, m, color, 0):
        print("Solution Found:", color)
    else:
        print("No solution exists with given number of colors")

    best_solution = [0] * n
    print("Solving using Branch and Bound...")
    if branch_and_bound(graph, m, best_solution, 0, color):
        print("Solution Found:", best_solution)
    else:
        print("No solution exists with given number of colors")

    return color

def draw_graph(n, edges, color):
    G = nx.Graph()
    for i in range(n):
        G.add_node(i)
    for u, v in edges:
        G.add_edge(u, v)

    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color=color, cmap=plt.cm.rainbow, edge_color='black',
    plt.show()

if __name__ == "__main__":
    n = int(input("Enter number of vertices: "))
    e = int(input("Enter number of edges: "))
    edges = []
    for _ in range(e):
        u, v = map(int, input("Enter edge (u v): ").split())
```

```
        edges.append((u, v))
    m = int(input("Enter number of colors: "))

    color = solve_graph_coloring(n, edges, m)
    draw_graph(n, edges, color)
```

```
Enter number of vertices:  5
Enter number of edges:  4
Enter edge (u v):  0 1
Enter edge (u v):  0 2
Enter edge (u v):  0 3
Enter edge (u v):  0 4
Enter number of colors:  2

Solving using Backtracking...
Solution Found: [1, 2, 2, 2, 2]
Solving using Branch and Bound...
Solution Found: [1, 2, 2, 2, 2]
```

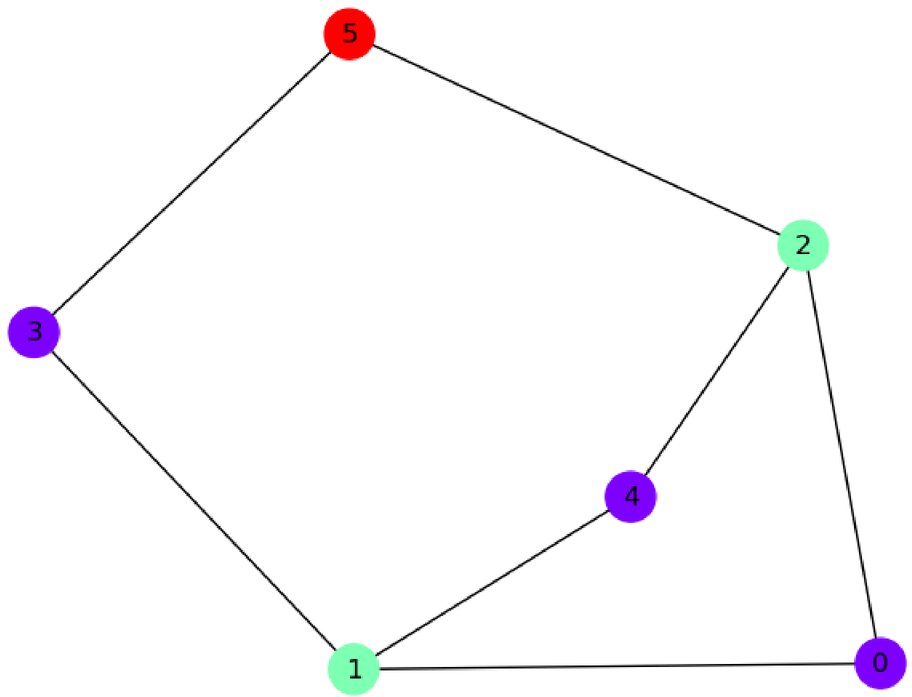


In [32]:

```
import networkx as nx
import matplotlib.pyplot as plt

def is_safe(graph, v, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True

def graph_coloring_backtracking(graph, m, color, v):
    if v == len(graph):
        return True

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

    return False

def branch_and_bound(graph, m, color, v, best_solution):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if is_safe(graph, v, color, c):
            color[v] = c
            if branch_and_bound(graph, m, color, v + 1, best_solution):
                return True
            color[v] = 0

    return False

def solve_graph_coloring(n, edges, m):
    graph = [[0] * n for _ in range(n)]

    for u, v in edges:
        graph[u][v] = 1
        graph[v][u] = 1

    color = [0] * n

    print("Solving using Backtracking...")
    if graph_coloring_backtracking(graph, m, color, 0):
        print("Solution Found:", color)
    else:
        print("No solution exists with given number of colors")

    best_solution = [0] * n
    print("Solving using Branch and Bound...")
    if branch_and_bound(graph, m, best_solution, 0, color):
        print("Solution Found:", best_solution)
    else:
        print("No solution exists with given number of colors")

    return color

def draw_graph(n, edges, color):
    G = nx.Graph()
    for i in range(n):
        G.add_node(i)
    for u, v in edges:
        G.add_edge(u, v)

    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color=color, cmap=plt.cm.rainbow, edge_color='black',
    plt.show()

if __name__ == "__main__":
    n = int(input("Enter number of vertices: "))
    e = int(input("Enter number of edges: "))
    edges = []
    for _ in range(e):
        u, v = map(int, input("Enter edge (u v): ").split())
```

```
        edges.append((u, v))
    m = int(input("Enter number of colors: "))

    color = solve_graph_coloring(n, edges, m)
    draw_graph(n, edges, color)
```

```
Enter number of vertices:  6
Enter number of edges:  7
Enter edge (u v):  0 1
Enter edge (u v):  0 2
Enter edge (u v):  1 3
Enter edge (u v):  1 4
Enter edge (u v):  2 4
Enter edge (u v):  2 5
Enter edge (u v):  3 5
Enter number of colors:  3

Solving using Backtracking...
Solution Found: [1, 2, 2, 1, 1, 3]
Solving using Branch and Bound...
Solution Found: [1, 2, 2, 1, 1, 3]
```



In [24]:

```python
def print_solution(board):
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print()

def is_safe(board, row, col, n):
    for i in range(row):
        if board[i][col]:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False

    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j]:
            return False

    return True

def solve_n_queens_backtrack(board, row, n):
    if row == n:
        print_solution(board)
        return

    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1
            print(f"Placing queen at ({row}, {col})")
            solve_n_queens_backtrack(board, row + 1, n)
            board[row][col] = 0  # Backtrack
            print(f"Backtracking from ({row}, {col})")

def branch_and_bound_n_queens(n):
    board = [[0] * n for _ in range(n)]
    cols = [False] * n
    diag1 = [False] * (2 * n - 1)
    diag2 = [False] * (2 * n - 1)

    def solve(row):
        if row == n:
            print_solution(board)
            return

        for col in range(n):
            if not cols[col] and not diag1[row - col + n - 1] and not diag2[row + col]:
                board[row][col] = 1
                cols[col] = diag1[row - col + n - 1] = diag2[row + col] = True
                print(f"Branching: Placing queen at ({row}, {col})")
                solve(row + 1)
                board[row][col] = 0
                cols[col] = diag1[row - col + n - 1] = diag2[row + col] = False
                print(f"Branching: Backtracking from ({row}, {col})")

    solve(0)

def n_queens(n):
    print("Solving with Backtracking:")
    board = [[0] * n for _ in range(n)]
    solve_n_queens_backtrack(board, 0, n)

    print("\nSolving with Branch and Bound:")
    branch_and_bound_n_queens(n)

n = int(input("Enter the number of queens: "))
n_queens(n)

```

```
Enter the number of queens:  4
```

```
Solving with Backtracking:
Placing queen at (0, 0)
Placing queen at (1, 2)
Backtracking from (1, 2)
Placing queen at (1, 3)
Placing queen at (2, 1)
Backtracking from (2, 1)
Backtracking from (1, 3)
Backtracking from (0, 0)
Placing queen at (0, 1)
Placing queen at (1, 3)
Placing queen at (2, 0)
Placing queen at (3, 2)
. Q . .
. . . Q
Q . . .
. . Q .

Backtracking from (3, 2)
Backtracking from (2, 0)
Backtracking from (1, 3)
Backtracking from (0, 1)
Placing queen at (0, 2)
Placing queen at (1, 0)
Placing queen at (2, 3)
Placing queen at (3, 1)
. . Q .
Q . . .
. . . Q
. Q . .

Backtracking from (3, 1)
Backtracking from (2, 3)
Backtracking from (1, 0)
Backtracking from (0, 2)
Placing queen at (0, 3)
Placing queen at (1, 0)
Placing queen at (2, 2)
Backtracking from (2, 2)
Backtracking from (1, 0)
Placing queen at (1, 1)
Backtracking from (1, 1)
Backtracking from (0, 3)

Solving with Branch and Bound:
Branching: Placing queen at (0, 0)
Branching: Placing queen at (1, 2)
Branching: Backtracking from (1, 2)
Branching: Placing queen at (1, 3)
Branching: Placing queen at (2, 1)
Branching: Backtracking from (2, 1)
Branching: Backtracking from (1, 3)
Branching: Backtracking from (0, 0)
Branching: Placing queen at (0, 1)
Branching: Placing queen at (1, 3)
Branching: Placing queen at (2, 0)
Branching: Placing queen at (3, 2)
. Q . .
. . . Q
Q . . .
. . Q .

Branching: Backtracking from (3, 2)
Branching: Backtracking from (2, 0)
Branching: Backtracking from (1, 3)
Branching: Backtracking from (0, 1)
Branching: Placing queen at (0, 2)
Branching: Placing queen at (1, 0)
Branching: Placing queen at (2, 3)
Branching: Placing queen at (3, 1)
. . Q .
Q . . .
. . . Q
. Q . .

Branching: Backtracking from (3, 1)
```

```
Branching: Backtracking from (2, 3)
Branching: Backtracking from (1, 0)
Branching: Backtracking from (0, 2)
Branching: Placing queen at (0, 3)
Branching: Placing queen at (1, 0)
Branching: Placing queen at (2, 2)
Branching: Backtracking from (2, 2)
Branching: Backtracking from (1, 0)
Branching: Placing queen at (1, 1)
Branching: Backtracking from (1, 1)
Branching: Backtracking from (0, 3)
```