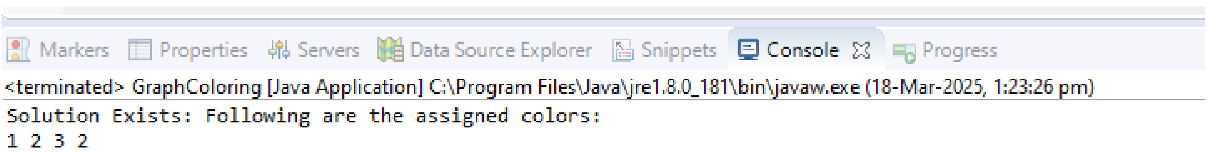
UNIT-5 Backtracking and Branch and Bound

- Try assigning a color to a vertex.
- Check if it's safe (i.e., no adjacent vertex has the same color).
- If valid, proceed to the next vertex.
- If a conflict occurs, backtrack and try another color.

Program 17- Write a program to implement Graph coloring problems using Backtracking

```
class GraphColoring {
    final int V = 4; // Number of vertices
    int graph[][] = {
        { 0, 1, 1, 1 },
        { 1, 0, 1, 0 },
        { 1, 1, 0, 1 },
        { 1, 0, 1, 0 }
    };

    int colors[]; // Stores the color assigned to each vertex

    // Function to check if it's safe to assign a color to vertex v
    boolean isSafe(int v, int c) {
        for (int i = 0; i < V; i++) {
            if (graph[v][i] == 1 && colors[i] == c) {
                return false; // If adjacent vertex has same color, return false
            }
        }
        return true;
    }

    // Backtracking function to solve m-coloring problem
    boolean solveGraphColoring(int v, int m) {
        if (v == V) {
            return true; // All vertices are assigned colors successfully
        }

        for (int c = 1; c <= m; c++) {
            if (isSafe(v, c)) {
                colors[v] = c; // Assign color

                if (solveGraphColoring(v + 1, m)) {
                    return true; // Recur for next vertex
                }

                colors[v] = 0; // Backtrack if no valid color found
```

```java
        }
    }

    return false; // If no color can be assigned, return false
}

// Function to print the solution
void printSolution() {
    System.out.println("Solution Exists: Following are the assigned colors:");
    for (int i = 0; i < V; i++) {
        System.out.print(colors[i] + " ");
    }
    System.out.println();
}

// Main function to check if solution exists
boolean graphColoring(int m) {
    colors = new int[V]; // Initialize color array with 0

    if (!solveGraphColoring(0, m)) {
        System.out.println("Solution does not exist.");
        return false;
    }

    printSolution();
    return true;
}

// Driver Code
public static void main(String args[]) {
    GraphColoring g = new GraphColoring();
    int m = 3; // Number of colors
    g.graphColoring(m);
}
}
```



This means:

```
  (0)
 / | \
(1)-(2)
 \ | /
  (3)
```

- **Vertex 0** → Color `1`
- **Vertex 1** → Color `2`
- **Vertex 2** → Color `3`
- **Vertex 3** → Color `2`

Explanation:

Function Breakdown:

- isSafe(int v, int c): Checks if color c can be assigned to vertex v.
- solveGraphColoring(int v, int m): Tries to color the graph using m colors with backtracking.
- graphColoring(int m): Initializes the coloring process.
- printSolution(): Displays the assigned colors for each vertex.

Step 1: Initial Function Call

int m = 3;
g.graphColoring(m);

- colors[] = {0, 0, 0, 0} (Initially, no colors assigned)
- Calls solveGraphColoring(0, 3), meaning we start coloring **vertex 0**.

Step 2: Coloring Vertex 0
solveGraphColoring(0, 3)

- **Loop Iteration for c = 1 (Trying Color 1)**
  - isSafe(0, 1) → **Yes, it's safe!**
  - Assign colors[0] = 1
  - **Recursive Call:** solveGraphColoring(1, 3)

✅ **Current Color Assignment:**
colors[] = {1, 0, 0, 0}

Step 3: Coloring Vertex 1

solveGraphColoring(1, 3)

- **Loop Iteration for c = 1 (Trying Color 1)**
  - isSafe(1, 1) → ❌ **Not Safe** (Vertex 1 is adjacent to Vertex 0, which already has color 1)
- **Loop Iteration for c = 2 (Trying Color 2)**
  - isSafe(1, 2) → ✅ **Safe**
  - Assign colors[1] = 2
  - **Recursive Call:** solveGraphColoring(2, 3)

✅ **Current Color Assignment:**
colors[] = {1, 2, 0, 0}

Step 4: Coloring Vertex 2
solveGraphColoring(2, 3)

- **Loop Iteration for c = 1 (Trying Color 1)**
  - isSafe(2, 1) → ❌ **Not Safe** (Vertex 2 is adjacent to Vertex 0, which has color 1)
- **Loop Iteration for c = 2 (Trying Color 2)**
  - isSafe(2, 2) → ❌ **Not Safe** (Vertex 2 is adjacent to Vertex 1, which has color 2)
- **Loop Iteration for c = 3 (Trying Color 3)**
  - isSafe(2, 3) → ✅ **Safe**
  - Assign colors[2] = 3
  - **Recursive Call:** solveGraphColoring(3, 3)

✅ **Current Color Assignment:**
colors[] = {1, 2, 3, 0}

Step 5: Coloring Vertex 3
solveGraphColoring(3, 3)

- **Loop Iteration for c = 1 (Trying Color 1)**
  - isSafe(3, 1) → ❌ **Not Safe** (Vertex 3 is adjacent to Vertex 0, which has color 1)
- **Loop Iteration for c = 2 (Trying Color 2)**
  - isSafe(3, 2) → ✅ **Safe**
  - Assign colors[3] = 2
  - **Recursive Call:** solveGraphColoring(4, 3)

✅ **Current Color Assignment:**
colors[] = {1, 2, 3, 2}

Detailed Loop Execution for Each Vertex

Vertex 0

Color	isSafe()	Assigned?
1	✅ Safe	✅ Yes
2	-	-
3	-	-

Vertex 1

Color	isSafe()	Assigned?
1	❌ Not Safe	❌ No
2	✅ Safe	✅ Yes
3	-	-

Vertex 3

Color	isSafe()	Assigned?
1	❌ Not Safe	❌ No
2	✅ Safe	✅ Yes
3	-	-

Program 18- Write a program to implement the Hamiltonian cycle using Backtracking

- **Graph Representation**:
  - The graph is stored as an adjacency matrix (`graph[][]`).
  - A `path[]` array is used to store the Hamiltonian cycle.
- **Backtracking Approach**:
  - Start at `path[0] = 0` (vertex 0).
  - Try adding vertices **one by one** to the cycle while ensuring:
    - The vertex is **connected** to the previous vertex.
    - The vertex is **not already in the path**.
- **Recursive Exploration (`cycleFound()`)**:
  - If all vertices are added (`k == NODE`), check if the last vertex connects to the first.

- If valid, return **true** (cycle found). Otherwise, **backtrack** by removing the last added vertex and trying the next one.

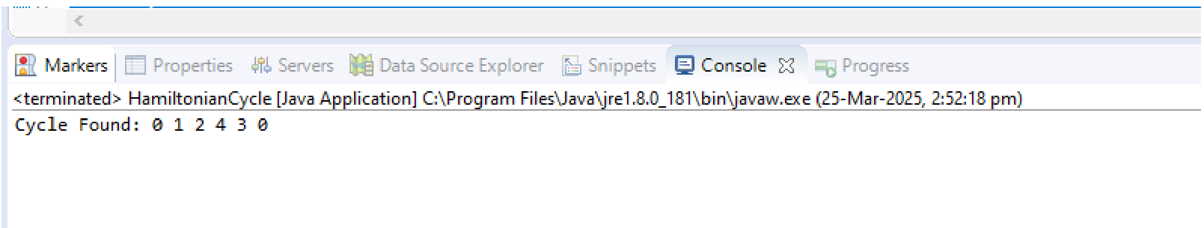
- **Final Check & Output (`hamiltonianCycle()`)**:

  - If a cycle is found, print the path.
  - If no solution exists, print `"Solution does not exist"`.

Code:

```
public class HamiltonianCycle {

        static final int NODE = 5;
        static int[][] graph = {
          {0, 1, 0, 1, 0},
          {1, 0, 1, 1, 1},
          {0, 1, 0, 0, 1},
          {1, 1, 0, 0, 1},
          {0, 1, 1, 1, 0}
        };
        static int[] path = new int[NODE];
        // method to display the Hamiltonian cycle
        static void displayCycle() {
          System.out.print("Cycle Found: ");
          for (int i = 0; i < NODE; i++)
            System.out.print(path[i] + " ");
          // Print the first vertex again
          System.out.println(path[0]);
        }
        // method to check if adding vertex v to the path is valid
        static boolean isValid(int v, int k) {
          // If there is no edge between path[k-1] and v
          if (graph[path[k - 1]][v] == 0)
            return false;
          // Check if vertex v is already taken in the path
          for (int i = 0; i < k; i++)
            if (path[i] == v)
              return false;
          return true;
        }
        // method to find the Hamiltonian cycle
        static boolean cycleFound(int k) {
          // When all vertices are in the path
          if (k == NODE) {
            // Check if there is an edge between the last and first vertex
            if (graph[path[k - 1]][path[0]] == 1)
              return true;
            else
              return false;
```

```
        }
        // adding each vertex (except the starting point) to the path
        for (int v = 1; v < NODE; v++) {
          if (isValid(v, k)) {
            path[k] = v;
            if (cycleFound(k + 1))
              return true;
              // Remove v from the path
              path[k] = -1;
          }
        }
        return false;
      }
      // method to find and display the Hamiltonian cycle
      static boolean hamiltonianCycle() {
        for (int i = 0; i < NODE; i++)
          path[i] = -1;
        // Set the first vertex as 0
        path[0] = 0;
        if (!cycleFound(1)) {
          System.out.println("Solution does not exist");
          return false;
        }
        displayCycle();
        return true;
      }
      public static void main(String[] args) {
        hamiltonianCycle();
      }
    }
```



19 Write a program to implement Travelling Salesman using branch and bound. (Assignment)