**Problem Statement**

Write a program to manipulate an undirected graph using an adjacency matrix.

Initially, the graph is empty, and edges are added to connect pairs of vertices. After creating the graph, display the corresponding adjacency matrix.

Subsequently, remove one edge based on user input and display the updated adjacency matrix.

**Input format :**

The first line of input contains an integer **V,**representing the number of vertices in the graph.

The second line contains an integer**E,** representing the number of edges to be added to the graph.

The following **E** lines each contain two integers **src** and **dest**, representing an edge between vertices src and dest.

The last line is two integers, **src** and **dest** of vertices which are to be removed.

**Output format :**

The first line of output prints "Original Adjacency Matrix:"

Followed by the adjacency matrix representation with each row indicating a vertex and its connections to other vertices.

After a new line, "Adjacency Matrix after removing the edge:" is printed.

Then, the modified adjacency matrix is displayed, showing the effect of removing the specified edge. Again, each row represents a vertex and its connections to other vertices.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

1 ≤ V ≤ 100

0 ≤ E ≤ N\*(N-1)/2 (Maximum possible edges without parallel edges and self-loops)

0 ≤ src, dest < V

**Sample test cases :**

**Input 1 :**

4

5

0 1

0 2

1 2

2 0

2 3

2 3

**Output 1 :**

Original Adjacency Matrix:

0 : 0 1 1 0

1 : 1 0 1 0

2 : 1 1 0 1

3 : 0 0 1 0

Adjacency Matrix after removing the edge:

0 : 0 1 1 0

1 : 1 0 1 0

2 : 1 1 0 0

3 : 0 0 0 0

**Input 2 :**

3

3

0 1

1 2

0 2

1 2

**Output 2 :**

Original Adjacency Matrix:

0 : 0 1 1

1 : 1 0 1

2 : 1 1 0

Adjacency Matrix after removing the edge:

0 : 0 1 1

1 : 1 0 0

**OUTPUT:**

class GraphManipulation {

  public static void main(String[] args) {

      // Static data for testing

      int V = 4;  // Number of vertices

      int E = 5;  // Number of edges

      // Initialize the adjacency matrix

      int[][] adjMatrix = new int[V][V];

      // Static edges

      int[][] edges = {

          {0, 1},

          {0, 2},

          {1, 2},

          {2, 0},

          {2, 3}

      };

      // Add edges to the adjacency matrix

      for (int i = 0; i < E; i++) {

          int src = edges[i][0];

          int dest = edges[i][1];

          adjMatrix[src][dest] = 1;

          adjMatrix[dest][src] = 1; // Since the graph is undirected

      }

      // Display the original adjacency matrix

      System.out.println("Original Adjacency Matrix:");

      printAdjMatrix(adjMatrix, V);

      // Static edge to be removed

      int removeSrc = 2;

      int removeDest = 3;

      adjMatrix[removeSrc][removeDest] = 0;

      adjMatrix[removeDest][removeSrc] = 0; // Since the graph is undirected

      // Display the updated adjacency matrix

      System.out.println("\nAdjacency Matrix after removing the edge:");

      printAdjMatrix(adjMatrix, V);

  }

  private static void printAdjMatrix(int[][] adjMatrix, int V) {

      for (int i = 0; i < V; i++) {

          System.out.print(i + " : ");

          for (int j = 0; j < V; j++) {

              System.out.print(adjMatrix[i][j] + " ");

          }

          System.out.println();

      }

  }

}

**Detailed code:**

import java.util.Scanner;

class GraphManipulation {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the number of vertices (cities)

        int V = scanner.nextInt();

        // Initialize the adjacency matrix with V vertices

        int[][] adjMatrix = new int[V][V];

        // Read the number of edges (direct routes)

        int E = scanner.nextInt();

        // Add edges to the adjacency matrix based on user input

        for (int i = 0; i < E; i++) {

            int src = scanner.nextInt();  // Source vertex of the edge

            int dest = scanner.nextInt(); // Destination vertex of the edge

            adjMatrix[src][dest] = 1;     // Set the value to 1 indicating an edge exists

            adjMatrix[dest][src] = 1;     // Since the graph is undirected, set the symmetric value

        }

        // Display the original adjacency matrix

        System.out.println("Original Adjacency Matrix:");

        printAdjMatrix(adjMatrix, V);

        // Read the edge to be removed

        int removeSrc = scanner.nextInt();  // Source vertex of the edge to be removed

        int removeDest = scanner.nextInt(); // Destination vertex of the edge to be removed

        adjMatrix[removeSrc][removeDest] = 0; // Remove the edge by setting the value to 0

        adjMatrix[removeDest][removeSrc] = 0; // Since the graph is undirected, remove the symmetric edge

        // Display the updated adjacency matrix

        System.out.println("\nAdjacency Matrix after removing the edge:");

        printAdjMatrix(adjMatrix, V);

        scanner.close();

    }

    // Helper method to print the adjacency matrix

    private static void printAdjMatrix(int[][] adjMatrix, int V) {

        for (int i = 0; i < V; i++) {           // Loop through each vertex

            System.out.print(i + " : ");        // Print the vertex number followed by its connections

            for (int j = 0; j < V; j++) {       // Loop through the connections of the vertex

                System.out.print(adjMatrix[i][j] + " "); // Print the connection value (0 or 1)

            }

            System.out.println();               // Move to the next line for the next vertex

        }

    }

}

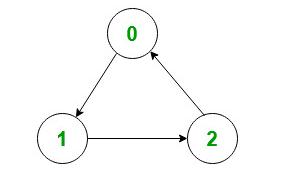
**Detailed Explanation:**

1. **Import Statements:**
   * We import java.util.Scanner to read input from the user.
2. **Main Method:**
   * Scanner scanner = new Scanner(System.in);: Initializes a Scanner object to read user input.
3. **Reading the Number of Vertices:**
   * int V = scanner.nextInt();: Reads the number of vertices (V) from user input.
4. **Initializing the Adjacency Matrix:**
   * int[][] adjMatrix = new int[V][V];: Creates a V x V adjacency matrix initialized to zeros.
5. **Reading the Number of Edges:**
   * int E = scanner.nextInt();: Reads the number of edges (E) from user input.
6. **Adding Edges to the Adjacency Matrix:**
   * The for loop iterates E times to read src (source vertex) and dest (destination vertex) for each edge.
   * adjMatrix[src][dest] = 1; and adjMatrix[dest][src] = 1;: Sets the corresponding entries in the adjacency matrix to 1 to indicate the presence of an edge (since the graph is undirected).
7. **Displaying the Original Adjacency Matrix:**
   * printAdjMatrix(adjMatrix, V);: Calls the printAdjMatrix method to display the adjacency matrix.
8. **Reading the Edge to be Removed:**
   * int removeSrc = scanner.nextInt(); and int removeDest = scanner.nextInt();: Reads the source and destination vertices of the edge to be removed.
   * adjMatrix[removeSrc][removeDest] = 0; and adjMatrix[removeDest][removeSrc] = 0;: Sets the corresponding entries in the adjacency matrix to 0 to remove the edge.
9. **Displaying the Updated Adjacency Matrix:**
   * printAdjMatrix(adjMatrix, V);: Calls the printAdjMatrix method to display the updated adjacency matrix.
10. **Closing the Scanner:**
    * scanner.close();: Closes the Scanner object to free resources.
11. **Helper Method printAdjMatrix:**
    * This method iterates through the adjacency matrix and prints it in a readable format, showing which vertices are connected.

This structure allows for easy modification and understanding of the program's flow. The adjacency matrix clearly shows the connections between vertices before and after removing an edge.

**Problem Statement**

Given the number of vertices and edges of a directed graph, the task is to represent the adjacency list.



**Example**

**Input:**

V = 3, edges[][]= {{0, 1}, {1, 2} {2, 0}}

**Output:**

0 -> 1

1 -> 2

2 -> 0

**Explanation:**

The output represents the adjacency list for the given graph.

**Input format :**

The first line of input consists of two integers representing the number of vertices (V) and edges (E), separated by a space.

The following E lines consist of two space-separated integers each, representing the connection of vertices.

**Output format :**

The output prints the adjacency list representation of the graph, where each line corresponds to a vertex followed by its neighbouring vertices separated by a space.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ V, E ≤ 10

**Sample test cases :**

**Input 1 :**

3 3

0 1

1 2

2 0

**Output 1 :**

0 -> 1

1 -> 2

2 -> 0

**Input 2 :**

4 5

0 1

1 2

1 3

2 3

3 0

**Output 2 :**

0 -> 1

1 -> 2 3

2 -> 3

3 -> 0

**Solution:**

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class DirectedGraphAdjacencyList {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the number of vertices (V) and edges (E)

        int V = scanner.nextInt();

        int E = scanner.nextInt();

        // Initialize the adjacency list

        List<List<Integer>> adjList = new ArrayList<>();

        // Create an empty list for each vertex

        for (int i = 0; i < V; i++) {

            adjList.add(new ArrayList<>());

        }

        // Read the edges and populate the adjacency list

        for (int i = 0; i < E; i++) {

            int src = scanner.nextInt();  // Source vertex of the edge

            int dest = scanner.nextInt(); // Destination vertex of the edge

            adjList.get(src).add(dest);   // Add the destination vertex to the adjacency list of the source vertex

        }

        // Display the adjacency list

        for (int i = 0; i < V; i++) {

            System.out.print(i + " ->"); // Print the vertex number followed by its connections

            for (Integer dest : adjList.get(i)) {

                System.out.print(" " + dest); // Print each connected vertex

            }

            System.out.println(); // Move to the next line for the next vertex

        }

        scanner.close();

    }

}

**Detailed Explanation:**

1. **Import Statements:**
   * import java.util.ArrayList;
   * import java.util.List;
   * import java.util.Scanner;
   * These imports are required for using the ArrayList, List, and Scanner classes.
2. **Main Class and Method:**
   * class DirectedGraphAdjacencyList { ... }
   * public static void main(String[] args) { ... }
   * This is the main class and method where the program execution begins.
3. **Reading Input:**
   * Scanner scanner = new Scanner(System.in);
   * Initializes a Scanner object to read user input.
   * int V = scanner.nextInt();
   * Reads the number of vertices (V) from user input.
   * int E = scanner.nextInt();
   * Reads the number of edges (E) from user input.
4. **Initializing the Adjacency List:**
   * List<List<Integer>> adjList = new ArrayList<>();
   * Creates an adjacency list where each element is a list representing the vertices connected to the corresponding vertex.
   * for (int i = 0; i < V; i++) { adjList.add(new ArrayList<>()); }
   * Initializes an empty list for each vertex in the graph.
5. **Reading Edges and Populating the Adjacency List:**
   * for (int i = 0; i < E; i++) { ... }
   * Iterates E times to read src (source vertex) and dest (destination vertex) for each edge.
   * int src = scanner.nextInt(); and int dest = scanner.nextInt();
   * Reads the source and destination vertices of an edge.
   * adjList.get(src).add(dest);
   * Adds the destination vertex to the list of the source vertex, representing a directed edge from src to dest.
6. **Displaying the Adjacency List:**
   * for (int i = 0; i < V; i++) { ... }
   * Iterates through each vertex to print its adjacency list.
   * System.out.print(i + " ->");
   * Prints the vertex number followed by its connections.
   * for (Integer dest : adjList.get(i)) { System.out.print(" " + dest); }
   * Iterates through the list of each vertex and prints the connected vertices.
   * System.out.println();
   * Moves to the next line for the next vertex.
7. **Closing the Scanner:**
   * scanner.close();
   * Closes the Scanner object to free up resources.

This program reads the number of vertices and edges from the user, constructs the adjacency list for a directed graph based on the provided edges, and then prints the adjacency list in a readable format.

import java.util.ArrayList;

import java.util.List;

class DirectedGraphAdjacencyList {

    public static void main(String[] args) {

        // Static number of vertices (V) and edges (E)

        int V = 5;  // Number of vertices

        int E = 6;  // Number of edges

        // Initialize the adjacency list

        List<List<Integer>> adjList = new ArrayList<>();

        // Create an empty list for each vertex

        for (int i = 0; i < V; i++) {

            adjList.add(new ArrayList<>());

        }

        // Static edge data

        int[][] edges = {

            {0, 1},

            {0, 2},

            {1, 2},

            {2, 0},

            {2, 3},

            {3, 3}

        };

        // Populate the adjacency list with static edge data

        for (int i = 0; i < E; i++) {

            int src = edges[i][0];  // Source vertex of the edge

            int dest = edges[i][1]; // Destination vertex of the edge

            adjList.get(src).add(dest);   // Add the destination vertex to the adjacency list of the source vertex

        }

        // Display the adjacency list

        for (int i = 0; i < V; i++) {

            System.out.print(i + " ->"); // Print the vertex number followed by its connections

            for (Integer dest : adjList.get(i)) {

                System.out.print(" " + dest); // Print each connected vertex

            }

            System.out.println(); // Move to the next line for the next vertex

        }

    }

}

**Explanation:**

1. **Static Number of Vertices and Edges:**
   * int V = 5; and int E = 6;
   * These lines define the number of vertices and edges in the graph.
2. **Initialization of Adjacency List:**
   * List<List<Integer>> adjList = new ArrayList<>();
   * Creates the adjacency list as before.
3. **Creating Empty Lists for Each Vertex:**
   * for (int i = 0; i < V; i++) { adjList.add(new ArrayList<>()); }
   * Initializes an empty list for each vertex.
4. **Static Edge Data:**
   * int[][] edges = { {0, 1}, {0, 2}, {1, 2}, {2, 0}, {2, 3}, {3, 3} };
   * This array represents the edges of the graph in a 2D array format, where each inner array contains the source and destination vertices of an edge.
5. **Populating the Adjacency List:**
   * for (int i = 0; i < E; i++) { ... }
   * Iterates through the static edge data to populate the adjacency list.
6. **Displaying the Adjacency List:**
   * for (int i = 0; i < V; i++) { ... }
   * Iterates through each vertex to print its adjacency list.

This static version of the program initializes the graph with predefined vertices and edges, constructs the adjacency list, and prints it, making it easy to visualize the graph structure without user input.

**Problem Statement**

Guide Bharath in designing a program to manage an undirected graph where you can add edges between vertices and display the adjacency list. After building the graph, remove an edge specified by the user and display the updated adjacency list.

**Input format :**

The first line of input contains an integer **V,**representing the number of vertices in the graph.

The second line contains an integer**E,** representing the number of edges to be added to the graph.

The following **E** lines each contain two integers **src** and **dest**, representing an edge between vertices src and dest.

The last line is two integers, **src** and **dest** of vertices which are to be removed.

**Output format :**

The output prints the adjacency list of the graph for each vertex after removing the specified edges.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

1 ≤ V ≤ 100

0 ≤ E ≤ N\*(N-1)/2 (Maximum possible edges without parallel edges and self-loops)

0 ≤ src, dest < V

**Sample test cases :**

**Input 1 :**

4

5

0 1

0 2

1 2

2 0

2 3

2 3

**Output 1 :**

Adjacent vertices of vertex 0: 1 2

Adjacent vertices of vertex 1: 0 2

Adjacent vertices of vertex 2: 0 1

Adjacent vertices of vertex 3:

**Input 2 :**

4

4

0 1

0 2

1 2

2 3

0 2

**Output 2 :**

Adjacent vertices of vertex 0: 1

Adjacent vertices of vertex 1: 0 2

Adjacent vertices of vertex 2: 1 3

Adjacent vertices of vertex 3: 2

**Solution:**

import java.util.ArrayList;

import java.util.List;

class UndirectedGraphAdjacencyList {

    public static void main(String[] args) {

        // Static number of vertices (V)

        int V = 5;

        // Initialize the adjacency list

        List<List<Integer>> adjList = new ArrayList<>();

        // Create an empty list for each vertex

        for (int i = 0; i < V; i++) {

            adjList.add(new ArrayList<>());

        }

        // Static edge data

        int[][] edges = {

            {0, 1},

            {0, 4},

            {1, 2},

            {1, 3},

            {1, 4},

            {2, 3},

            {3, 4}

        };

        // Number of edges (E)

        int E = edges.length;

        // Populate the adjacency list with static edge data

        for (int i = 0; i < E; i++) {

            int src = edges[i][0];

            int dest = edges[i][1];

            adjList.get(src).add(dest);

            adjList.get(dest).add(src); // Since the graph is undirected

        }

        // Static edge to be removed

        int removeSrc = 1;

        int removeDest = 4;

        // Remove the edge from the adjacency list

        adjList.get(removeSrc).remove((Integer) removeDest);

        adjList.get(removeDest).remove((Integer) removeSrc); // Since the graph is undirected

        // Display the updated adjacency list

        for (int i = 0; i < V; i++) {

            System.out.print("Adjacent vertices of vertex " + i + ":");

            for (Integer vertex : adjList.get(i)) {

                System.out.print(" " + vertex);

            }

            System.out.println();

        }

    }}

**Explanation:**

1. **Static Number of Vertices and Edges:**
   * int V = 5;
   * Defines the number of vertices in the graph.
2. **Initialization of Adjacency List:**
   * List<List<Integer>> adjList = new ArrayList<>();
   * Creates the adjacency list as before.
3. **Creating Empty Lists for Each Vertex:**
   * for (int i = 0; i < V; i++) { adjList.add(new ArrayList<>()); }
   * Initializes an empty list for each vertex.
4. **Static Edge Data:**
   * int[][] edges = { {0, 1}, {0, 4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {3, 4} };
   * This array represents the edges of the graph in a 2D array format, where each inner array contains the source and destination vertices of an edge.
5. **Populating the Adjacency List:**
   * for (int i = 0; i < E; i++) { ... }
   * Iterates through the static edge data to populate the adjacency list.
6. **Static Edge to be Removed:**
   * int removeSrc = 1;
   * int removeDest = 4;
   * Defines the edge to be removed from the graph.
7. **Removing the Edge:**
   * adjList.get(removeSrc).remove((Integer) removeDest);
   * adjList.get(removeDest).remove((Integer) removeSrc);
   * Removes the specified edge from both vertices' adjacency lists since the graph is undirected.
8. **Displaying the Updated Adjacency List:**
   * for (int i = 0; i < V; i++) { ... }
   * Iterates through each vertex to print its adjacency list.

This static version of the program initializes the graph with predefined vertices and edges, constructs the adjacency list, removes a specified edge, and prints the updated adjacency list, making it easy to visualize the graph structure without user input.

In Java, when working with collections like ArrayList, the get and remove methods are used to retrieve and manipulate elements. The line adjList.get(removeSrc).remove((Integer) removeDest); utilizes these methods, and here's a detailed explanation of each component:

**Breakdown of the Code**

1. **adjList.get(removeSrc)**:
   * adjList is a List<List<Integer>>, which means it's a list of lists. Each inner list represents the adjacency list for a particular vertex.
   * adjList.get(removeSrc) retrieves the list of adjacent vertices for the vertex removeSrc. This list is an ArrayList<Integer>.
2. **remove((Integer) removeDest)**:
   * The remove method is used to remove a specific element from a list.
   * removeDest is an int representing the vertex you want to remove from the adjacency list of removeSrc.
   * To ensure that the remove method interprets removeDest as an Integer object (and not just an int), you use (Integer) removeDest. This is necessary because ArrayList's remove method has two overloads:
     + One that takes an int index.
     + One that takes an Object (in this case, Integer).
   * By casting removeDest to Integer, you are explicitly specifying that you want to remove the Integer object with the value of removeDest, not an element at a specific index.

**Detailed Explanation:**

* **Casting (Integer) removeDest**:
  + **Purpose**: The ArrayList class has two remove methods: one that removes by index and another that removes by object. Casting to Integer tells the method to remove the object with that value.
  + **Reason**: Java's ArrayList can sometimes confuse an int for an index if it has overloaded methods. By casting to Integer, you ensure that the remove method treats the input as the object to be removed, not an index.
* **Usage**:
  + adjList.get(removeSrc).remove((Integer) removeDest);:
    - This line removes the removeDest vertex from the adjacency list of removeSrc. This effectively deletes the edge between removeSrc and removeDest in the undirected graph.
  + adjList.get(removeDest).remove((Integer) removeSrc);:
    - This line removes the removeSrc vertex from the adjacency list of removeDest. This completes the removal of the edge in both directions, ensuring that the graph remains consistent as an undirected graph.

**Summary:**

* **get(index)**: Retrieves the list of adjacent vertices for the specified vertex index from the adjacency list.
* **remove(object)**: Removes the first occurrence of the specified object from the list. By casting removeDest to Integer, you ensure that the remove method removes the object with that value, not an index.

This approach ensures that you properly manage the adjacency list when an edge is removed, maintaining the integrity of the graph's structure.

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class UndirectedGraphAdjacencyList {

    public static void main(String[] args) {

        // Create a Scanner object to read user input

        Scanner scanner = new Scanner(System.in);

        // Prompt and read the number of vertices in the graph

        int V = scanner.nextInt();

        // Initialize the adjacency list where each vertex has its own list

        List<List<Integer>> adjList = new ArrayList<>();

        for (int i = 0; i < V; i++) {

            // Create an empty list for each vertex

            adjList.add(new ArrayList<>());

        }

        // Prompt and read the number of edges in the graph

        int E = scanner.nextInt();

        // Read the edges and populate the adjacency list

        //System.out.println("Enter the edges (src dest): ");

        for (int i = 0; i < E; i++) {

            // Read the source and destination of each edge

            int src = scanner.nextInt();

            int dest = scanner.nextInt();

            // Add the destination vertex to the adjacency list of the source vertex

            adjList.get(src).add(dest);

            // Add the source vertex to the adjacency list of the destination vertex (undirected graph)

            adjList.get(dest).add(src);

        }

        // Prompt and read the edge to be removed

        int removeSrc = scanner.nextInt();

        int removeDest = scanner.nextInt();

        // Remove the edge from the adjacency list of the source vertex

        // (casting to Integer to avoid removing by index)

        adjList.get(removeSrc).remove((Integer) removeDest);

        // Remove the edge from the adjacency list of the destination vertex

        // (casting to Integer to avoid removing by index)

        adjList.get(removeDest).remove((Integer) removeSrc);

        // Display the updated adjacency list

        for (int i = 0; i < V; i++) {

            // Print the vertex and its adjacent vertices

            System.out.print("Adjacent vertices of vertex " + i + ":");

            for (Integer vertex : adjList.get(i)) {

                // Print each adjacent vertex

                System.out.print(" " + vertex);

            }

            // Move to the next line after printing all adjacent vertices for the current vertex

            System.out.println();

        }

        // Close the Scanner object to free up resources

        scanner.close();

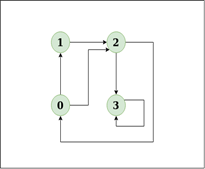
    }

}

**Problem Statement**

Given the root of a directed graph, write a program to check whether the graph contains a cycle or not.

**Examples**



**Input:**

4 6

0 1

0 2

1 2

2 0

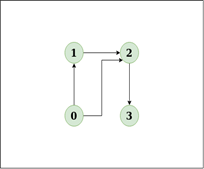
2 3

3 3

**Output:**

Yes, graph contains a cycle

The diagram clearly shows a cycle 0 -> 2 -> 0.



**Input:**

4 4

0 1

1 2

0 2

2 3

**Output:**

No, graph doesn't contain a cycle.

The diagram clearly shows there is no cycle.

**Input format :**

The first line of input consists of two integers **N** and**E**, representing the number of vertices and edges, respectively.

The next **E** lines contain two integers each source and destination, representing a directed edge from source to destination.

**Output format :**

If the graph contains a cycle, print "Yes, graph contains a cycle".

If the graph doesn't contain a cycle, print "No, graph doesn't contain a cycle".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

1 ≤ N ≤ 7

0 ≤ E ≤ 7

0 ≤ source, destination < N (vertices are numbered from 0 to N−1)

**Sample test cases :**

**Input 1 :**

4 6

0 1

0 2

1 2

2 0

2 3

3 3

**Output 1 :**

Yes, graph contains a cycle

**Input 2 :**

4 4

0 1

1 2

0 2

2 3

**Output 2 :**

No, graph doesn't contain a cycle

**Solution:**

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class DirectedGraphCycleDetection {

    public static void main(String[] args) {

        // Create a Scanner object to read user input

        Scanner scanner = new Scanner(System.in);

        // Prompt and read the number of vertices (N) in the graph

        int N = scanner.nextInt();

        // Prompt and read the number of edges (E) in the graph

        int E = scanner.nextInt();

        // Initialize the adjacency list where each vertex has its own list

        List<List<Integer>> adjList = new ArrayList<>();

        for (int i = 0; i < N; i++) {

            // Create an empty list for each vertex

            adjList.add(new ArrayList<>());

        }

        // Read the edges and populate the adjacency list

        for (int i = 0; i < E; i++) {

            int src = scanner.nextInt();

            int dest = scanner.nextInt();

            // Add the destination vertex to the adjacency list of the source vertex

            adjList.get(src).add(dest);

        }

        // Array to keep track of visited vertices

        boolean[] visited = new boolean[N];

        // Array to keep track of vertices in the current recursion stack

        boolean[] recStack = new boolean[N];

        // Variable to track if a cycle is detected

        boolean containsCycle = false;

        // Check for a cycle in each connected component of the graph

        for (int i = 0; i < N; i++) {

            if (dfs(i, adjList, visited, recStack)) {

                containsCycle = true;

                break; // Exit the loop if a cycle is found

            }

        }

        // Output the result

        if (containsCycle) {

            System.out.println("Yes, graph contains a cycle");

        } else {

            System.out.println("No, graph doesn't contain a cycle");

        }

        // Close the Scanner object to free up resources

        scanner.close();

    }

    /\*\*

     \* Depth-First Search (DFS) to detect a cycle in the directed graph.

     \* @param vertex The current vertex to explore.

     \* @param adjList The adjacency list representation of the graph.

     \* @param visited Array to keep track of visited vertices.

     \* @param recStack Array to keep track of vertices in the current recursion stack.

     \* @return true if a cycle is detected, false otherwise.

     \*/

    private static boolean dfs(int vertex, List<List<Integer>> adjList, boolean[] visited, boolean[] recStack) {

        // If the vertex is already in the recursion stack, a cycle is detected

        if (recStack[vertex]) {

            return true;

        }

        // If the vertex is already visited and not in the recursion stack, no cycle from this vertex

        if (visited[vertex]) {

            return false;

        }

        // Mark the vertex as visited and add it to the recursion stack

        visited[vertex] = true;

        recStack[vertex] = true;

        // Explore all neighbors of the current vertex

        for (int neighbor : adjList.get(vertex)) {

            // If the DFS traversal from the neighbor detects a cycle, return true

            if (dfs(neighbor, adjList, visited, recStack)) {

                return true;

            }

        }

        // Remove the vertex from the recursion stack before backtracking

        recStack[vertex] = false;

        return false;

    }

}

**Detailed Explanation:**

1. **Imports**:
   * **import java.util.ArrayList;** and **import java.util.List;**: These imports are used for working with lists to store the adjacency list.
   * **import java.util.Scanner;**: This import is used to read user input from the console.
2. **Main Method**:
   * **Scanner scanner = new Scanner(System.in);**: Initializes a Scanner object to read input from the user.
   * **int N = scanner.nextInt();**: Reads the number of vertices in the graph.
   * **int E = scanner.nextInt();**: Reads the number of edges in the graph.
   * **List<List<Integer>> adjList = new ArrayList<>();**: Initializes an adjacency list where each vertex has an associated list of adjacent vertices.
   * **for (int i = 0; i < N; i++) { adjList.add(new ArrayList<>()); }**: Initializes an empty adjacency list for each vertex.
3. **Populating Adjacency List**:
   * **for (int i = 0; i < E; i++) { int src = scanner.nextInt(); int dest = scanner.nextInt(); adjList.get(src).add(dest); }**: Reads each edge and populates the adjacency list. Each edge is directed from src to dest.
4. **Cycle Detection**:
   * **boolean[] visited = new boolean[N];**: Array to keep track of whether a vertex has been visited during the DFS traversal.
   * **boolean[] recStack = new boolean[N];**: Array to keep track of vertices currently in the recursion stack (used to detect cycles).
   * **boolean containsCycle = false;**: Variable to indicate if a cycle has been detected.
5. **DFS Traversal**:
   * **for (int i = 0; i < N; i++) { if (dfs(i, adjList, visited, recStack)) { containsCycle = true; break; } }**: Iterates through all vertices to ensure that all connected components are checked for cycles.
6. **DFS Method**:
   * **if (recStack[vertex]) { return true; }**: If the current vertex is in the recursion stack, a cycle is detected.
   * **if (visited[vertex]) { return false; }**: If the vertex has been visited and is not in the recursion stack, no cycle is detected from this vertex.
   * **visited[vertex] = true; recStack[vertex] = true;**: Marks the vertex as visited and adds it to the recursion stack.
   * **for (int neighbor : adjList.get(vertex)) { if (dfs(neighbor, adjList, visited, recStack)) { return true; } }**: Recursively explores all neighbors of the current vertex.
   * **recStack[vertex] = false;**: Removes the vertex from the recursion stack before backtracking.
7. **Output**:
   * **if (containsCycle) { System.out.println("Yes, graph contains a cycle"); } else { System.out.println("No, graph doesn't contain a cycle"); }**: Outputs whether a cycle was detected or not.
8. **Closing Scanner**:
   * **scanner.close();**: Closes the Scanner object to release resources.

This code effectively detects cycles in a directed graph using depth-first search (DFS) and recursion, employing a recursion stack to track vertices and detect cycles.

**Static data:**

import java.util.ArrayList;

import java.util.List;

public class DirectedGraphCycleDetection {

    public static void main(String[] args) {

        // Static data for testing

        int N = 4; // Number of vertices

        int E = 4; // Number of edges

        // Initialize the adjacency list

        List<List<Integer>> adjList = new ArrayList<>();

        for (int i = 0; i < N; i++) {

            adjList.add(new ArrayList<>());

        }

        // Static edges

        int[][] edges = {

            {0, 1}, // Edge from vertex 0 to vertex 1

            {1, 2}, // Edge from vertex 1 to vertex 2

            {2, 0}, // Edge from vertex 2 to vertex 0 (creates a cycle)

            {2, 3}  // Edge from vertex 2 to vertex 3

        };

        // Populate the adjacency list with static edges

        for (int i = 0; i < E; i++) {

            int src = edges[i][0];

            int dest = edges[i][1];

            adjList.get(src).add(dest);

        }

        // Check if the graph contains a cycle

        boolean containsCycle = false;

        boolean[] visited = new boolean[N];

        boolean[] recStack = new boolean[N];

        for (int i = 0; i < N; i++) {

            if (dfs(i, adjList, visited, recStack)) {

                containsCycle = true;

                break;

            }

        }

        // Output the result

        if (containsCycle) {

            System.out.println("Yes, graph contains a cycle");

        } else {

            System.out.println("No, graph doesn't contain a cycle");

        }

    }

    /\*\*

     \* Depth-First Search (DFS) to detect a cycle in the directed graph.

     \* @param vertex The current vertex to explore.

     \* @param adjList The adjacency list representation of the graph.

     \* @param visited Array to keep track of visited vertices.

     \* @param recStack Array to keep track of vertices in the current recursion stack.

     \* @return true if a cycle is detected, false otherwise.

     \*/

    private static boolean dfs(int vertex, List<List<Integer>> adjList, boolean[] visited, boolean[] recStack) {

        if (recStack[vertex]) {

            return true; // Cycle detected

        }

        if (visited[vertex]) {

            return false; // No cycle detected from this vertex

        }

        // Mark the vertex as visited and add it to the recursion stack

        visited[vertex] = true;

        recStack[vertex] = true;

        // Explore all neighbors of the current vertex

        for (int neighbor : adjList.get(vertex)) {

            if (dfs(neighbor, adjList, visited, recStack)) {

                return true; // Cycle detected

            }

        }

        // Remove the vertex from the recursion stack before backtracking

        recStack[vertex] = false;

        return false; // No cycle detected

    }

}

**Problem Statement**

Sanjay is writing a program to manage an undirected graph using an adjacency list representation. Users will initially input the vertices to create the edges. The program will display the adjacency list of the graph.

Assist Sanjay in creating the program.

**Input format :**

The input consists of 5 nodes.

Edge information: ab ae bc bd be cd de.

**Output format :**

The output prints the adjacency list representation of the graph.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

0 ≤ node values ≤ 10

**Sample test cases :**

**Input 1 :**

0 1 2 3 4

**Output 1 :**

vertex 0: head -> 4 -> 1

vertex 1: head -> 4 -> 3 -> 2 -> 0

vertex 2: head -> 3 -> 1

vertex 3: head -> 4 -> 2 -> 1

vertex 4: head -> 3 -> 1 -> 0

**Input 2 :**

1 2 3 4 0

**Output 2 :**

vertex 0: head -> 4 -> 2 -> 1

vertex 1: head -> 0 -> 2

vertex 2: head -> 0 -> 4 -> 3 -> 1

vertex 3: head -> 4 -> 2

vertex 4: head -> 0 -> 3 -> 2

import java.util.LinkedList;

import java.util.Scanner;

class AdjListNode {

    int dest;

    AdjListNode next;

    public AdjListNode(int dest) {

        this.dest = dest;

        this.next = null;

    }

}

class AdjList {

    AdjListNode head;

}

class Graph {

    int V;

    AdjList[] array;

    public Graph(int V) {

        this.V = V;

        array = new AdjList[V];

        for (int i = 0; i < V; ++i) {

            array[i] = new AdjList();

            array[i].head = null;

        }

    }

    public void addEdge(int src, int dest) {

        AdjListNode newNode = new AdjListNode(dest);

        newNode.next = array[src].head;

        array[src].head = newNode;

        newNode = new AdjListNode(src);

        newNode.next = array[dest].head;

        array[dest].head = newNode;

    }

    public void printGraph() {

        for (int v = 0; v < V; ++v) {

            AdjListNode pCrawl = array[v].head;

            System.out.print("vertex " + v + ": head");

            while (pCrawl != null) {

                System.out.print(" -> " + pCrawl.dest);

                pCrawl = pCrawl.next;

            }

            System.out.println();

        }

    }

}

public class Main {

    public static void main(String[] args) {

        int V = 5;

        Scanner sc = new Scanner(System.in);

        int a = sc.nextInt();

        int b = sc.nextInt();

        int c = sc.nextInt();

        int d = sc.nextInt();

        int e = sc.nextInt();

        sc.close();

        Graph graph = new Graph(V);

        graph.addEdge(a, b);

        graph.addEdge(a, e);

        graph.addEdge(b, c);

        graph.addEdge(b, d);

        graph.addEdge(b, e);

        graph.addEdge(c, d);

        graph.addEdge(d, e);

        graph.printGraph();

    }

}

**Problem Statement**

Imagine you are working on a project to visualize the relationships between various cities in a transportation network. Each city is represented as a vertex, and the direct routes between cities are represented as edges in an undirected graph.

Your task is to write a program that reads the number of cities (vertices) and the direct routes (edges), constructs the adjacency matrix for the graph, and then prints this matrix to help in the analysis of the connectivity between the cities.

**Input format :**

The first line of input consists of an integer**v,**representing the number of cities(vertices).

The second line consists of an integer **e,** representing the direct routes(edges).

The following e consists of the edge information.

**Output format :**

The output prints the adjacency matrix of the graph.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ v, e ≤ 20

**Sample test cases :**

**Input 1 :**

5

3

1 2

2 4

3 5

**Output 1 :**

0 1 0 0 0

1 0 0 1 0

0 0 0 0 1

0 1 0 0 0

0 0 1 0 0

**Input 2 :**

6

4

1 2

2 3

3 4

3 5

**Output 2 :**

0 1 0 0 0 0

1 0 1 0 0 0

0 1 0 1 1 0

0 0 1 0 0 0

0 0 1 0 0 0

0 0 0 0 0 0

import java.util.Scanner;

class AdjacencyMatrixGraph {

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the number of vertices (cities) in the graph

        int v = scanner.nextInt();

        // Read the number of edges (direct routes) in the graph

        int e = scanner.nextInt();

        // Initialize the adjacency matrix with size v x v, initially filled with zeros

        // This matrix will represent the graph where adjMatrix[i][j] is 1 if there is an edge from vertex i to vertex j, otherwise 0

        int[][] adjMatrix = new int[v][v];

        // Read each edge and populate the adjacency matrix

        for (int i = 0; i < e; i++) {

            // Read the source and destination of the edge

            int src = scanner.nextInt() - 1; // Subtract 1 to convert from 1-based to 0-based index

            int dest = scanner.nextInt() - 1; // Subtract 1 to convert from 1-based to 0-based index

            // Update the adjacency matrix for the undirected edge

            adjMatrix[src][dest] = 1; // Set entry to 1 to indicate an edge from src to dest

            adjMatrix[dest][src] = 1; // Set entry to 1 to indicate an edge from dest to src (since the graph is undirected)

        }

        // Print the adjacency matrix

        // Each row of the matrix represents the edges connected to a particular vertex

        for (int i = 0; i < v; i++) {

            for (int j = 0; j < v; j++) {

                // Print the value of adjMatrix[i][j] which is either 0 or 1

                // 1 indicates an edge exists between vertex i and vertex j

                System.out.print(adjMatrix[i][j] + " ");

            }

            // Print a new line after each row of the matrix

            System.out.println();

        }

        // Close the Scanner object to prevent resource leaks

        scanner.close();

    }

}

**Detailed Explanation:**

1. **Import Statement**:

java

Copy code

import java.util.Scanner;

This imports the Scanner class from the java.util package, which is used to read input from the console.

1. **Class Declaration**:

java

Copy code

class AdjacencyMatrixGraph {

Defines the class AdjacencyMatrixGraph.

1. **Main Method**:

java

Copy code

public static void main(String[] args) {

The entry point of the Java application.

1. **Scanner Object Creation**:

java

Copy code

Scanner scanner = new Scanner(System.in);

Creates a Scanner object to read input from the console.

1. **Reading Number of Vertices**:

java

Copy code

int v = scanner.nextInt();

Reads the number of vertices (v) from the user input.

1. **Reading Number of Edges**:

java

Copy code

int e = scanner.nextInt();

Reads the number of edges (e) from the user input.

1. **Initializing Adjacency Matrix**:

java

Copy code

int[][] adjMatrix = new int[v][v];

Initializes a 2D array adjMatrix of size v x v, where all elements are initially set to 0. This matrix will represent the graph.

1. **Populating the Adjacency Matrix**:

java

Copy code

for (int i = 0; i < e; i++) {

int src = scanner.nextInt() - 1;

int dest = scanner.nextInt() - 1;

adjMatrix[src][dest] = 1;

adjMatrix[dest][src] = 1;

}

* + Reads e pairs of integers from the input, where each pair represents an edge between vertices.
  + Converts 1-based indices to 0-based indices (by subtracting 1).
  + Updates the matrix to reflect the presence of an undirected edge.

1. **Printing the Adjacency Matrix**:

java

Copy code

for (int i = 0; i < v; i++) {

for (int j = 0; j < v; j++) {

System.out.print(adjMatrix[i][j] + " ");

}

System.out.println();

}

* + Iterates through each cell of the matrix and prints it.
  + Each row corresponds to a vertex, and each cell in a row shows whether there is an edge to another vertex.

1. **Closing the Scanner**:

java

Copy code

scanner.close();

Closes the Scanner object to release resources.

**Static Data**

class AdjacencyMatrixGraph {

  public static void main(String[] args) {

      // Static data for testing

      int v = 5; // Number of vertices (cities)

      int e = 4; // Number of edges (direct routes)

      // Initialize the adjacency matrix with size v x v, initially filled with zeros

      int[][] adjMatrix = new int[v][v];

      // Static edge information

      int[][] edges = {

          {1, 2},

          {2, 3},

          {3, 4},

          {4, 5}

      };

      // Populate the adjacency matrix with the static edge data

      for (int i = 0; i < e; i++) {

          int src = edges[i][0] - 1; // Convert from 1-based to 0-based index

          int dest = edges[i][1] - 1; // Convert from 1-based to 0-based index

          adjMatrix[src][dest] = 1; // Set entry to 1 to indicate an edge from src to dest

          adjMatrix[dest][src] = 1; // Set entry to 1 to indicate an edge from dest to src (since the graph is undirected)

      }

      // Print the adjacency matrix

      System.out.println("Adjacency Matrix:");

      for (int i = 0; i < v; i++) {

          for (int j = 0; j < v; j++) {

              System.out.print(adjMatrix[i][j] + " ");

          }

          System.out.println();

      }

  }

}