لتنفيذ خوارزمية Gabow للتحقق من أن البيان متصل بقوة فيJava ، يمكنك اتباع الخطوات التالية. هذه الخوارزمية تعتمد على خوارزمية البحث في العمق (DFS) وتستخدم مكدسين لمتابعة التقدم في الرسم البياني.

فيما يلي الكود الخاص بخوارزمية:Gabow

```
import java.util.*;


public class Main {


    public static void main(String[] args) {
        Digraph G = new Digraph(5);
        G.addEdge(0, 2);
        G.addEdge(2, 1);
        G.addEdge(1, 0);
        G.addEdge(0, 3);
        G.addEdge(3, 4);


        GabowSCC scc = new GabowSCC(G);


        int M = scc.count();
        System.out.println(M + " strong components");


        List<Integer>[] components = (List<Integer>[]) new List[M];
        for (int i = 0; i < M; i++) {
            components[i] = new ArrayList<>();
        }
        for (int v = 0; v < G.V(); v++) {
            components[scc.id(v)].add(v);
        }
        for (int i = 0; i < M; i++) {
```

```
            for (int v : components[i]) {
                System.out.print(v + " ");
            }
            System.out.println();
        }
    }
}

class GabowSCC {
    private int[] id;        // component ids
    private int[] preorder;  // preorder numbers
    private boolean[] marked; // visited vertices
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;
    private int pre;
    private int count;       // number of strongly-connected components

    public GabowSCC(Digraph G) {
        id = new int[G.V()];
        preorder = new int[G.V()];
        marked = new boolean[G.V()];
        stack1 = new Stack<>();
        stack2 = new Stack<>();
        pre = 0;
        count = 0;
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
            }
```

```
    }
}

private void dfs(Digraph G, int v) {
    marked[v] = true;
    preorder[v] = pre++;
    stack1.push(v);
    stack2.push(v);

    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        } else if (id[w] == 0) {
            while (!stack2.isEmpty() && preorder[stack2.peek()] > preorder[w]) {
                stack2.pop();
            }
        }
    }

    if (stack2.peek() == v) {
        stack2.pop();
        int w;
        do {
            w = stack1.pop();
            id[w] = count;
        } while (w != v);
        count++;
    }
}
```

```
    public boolean stronglyConnected(int v, int w) {
        return id[v] == id[w];
    }

    public int count() {
        return count;
    }

    public int id(int v) {
        return id[v];
    }
}

class Digraph {
    private final int V;
    private int E;
    private List<Integer>[] adj;

    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (List<Integer>[]) new List[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new ArrayList<>();
        }
    }

    public void addEdge(int v, int w) {
```

```
        adj[v].add(w);
        E++;
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public int V() {
        return V;
    }

    public int E() {
        return E;
    }
}}
```

\- =-------------------------

1. **GabowSCC**:
   - **id**: معرفات المكونات المتصلة بقوة.
   - **preorder**: أرقام الترتيب المسبق لكل رأس.
   - **marked**: حالة الزيارة لكل رأس.
   - **stack1** و**stack2**: مكدسات لتتبع الرؤوس أثناء البحث في العمق.
   - **dfs**: الدالة الرئيسية لتنفيذ البحث في العمق.
2. **Digraph**:
   - **V**: عدد الرؤوس.
   - **E**: عدد الحواف.
   - **adj**: قائمة الحواف المرتبطة بكل رأس.

2

```
import java.util.ArrayList;
import java.util.List;

public class Graph {
    private final int V;
    private final List<Integer>[] adj;

    public Graph(int V) {
        this.V = V;
        adj = (List<Integer>[]) new List[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new ArrayList<>();
        }
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public int V() {
        return V;
    }
```

```
}
```

JensSchmidtAlgorithm.java

```
import java.util.*;

public class JensSchmidtAlgorithm {
    private boolean[] visited;
    private int[] disc;
    private int[] low;
    private int[] parent;
    private int time;
    private List<Integer> articulationPoints;

    public JensSchmidtAlgorithm(Graph graph) {
        int V = graph.V();
        visited = new boolean[V];
        disc = new int[V];
        low = new int[V];
        parent = new int[V];
        articulationPoints = new ArrayList<>();
        time = 0;

        Arrays.fill(parent, -1);

        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                dfs(graph, i);
            }
```

```
    }
}

private void dfs(Graph graph, int u) {
    int children = 0;
    visited[u] = true;
    disc[u] = low[u] = ++time;

    for (int v : graph.adj(u)) {
        if (!visited[v]) {
            children++;
            parent[v] = u;
            dfs(graph, v);

            low[u] = Math.min(low[u], low[v]);

            if (parent[u] == -1 && children > 1) {
                articulationPoints.add(u);
            }

            if (parent[u] != -1 && low[v] >= disc[u]) {
                articulationPoints.add(u);
            }
        } else if (v != parent[u]) {
            low[u] = Math.min(low[u], disc[v]);
        }
    }
}
```

```
    public boolean hasArticulationPoints() {
        return !articulationPoints.isEmpty();
    }

    public List<Integer> getArticulationPoints() {
        return articulationPoints;
    }

    public static void main(String[] args) {
        Graph graph = new Graph(5);
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(3, 4);

        JensSchmidtAlgorithm algorithm = new JensSchmidtAlgorithm(graph);

        if (algorithm.hasArticulationPoints()) {
            System.out.println("The graph has articulation points: " +
algorithm.getArticulationPoints());
        } else {
            System.out.println("The graph has no articulation points.");
        }
    }

}
```

لفحص ما إذا كان الرسم البياني قوي الاتصال ذو عقدتين (2-vertex strongly connected) ، بمعنى إذا حذفنا أي عقدة من الرسم البياني فإنه يبقى قوي الاتصال، يمكننا اتباع الخطوات التالية:

1. **تحقق من أن الرسم البياني قوي الاتصال**: نستخدم خوارزمية Gabow للتحقق من أن الرسم البياني الأصلي قوي الاتصال.
2. **تحقق من أن إزالة أي عقدة لا تؤدي إلى انفصال الرسم البياني**: نستخدم خوارزمية Jens Schmidt للتحقق من عدم وجود عقد مفصلية.

الكود:

لنفترض أننا ندمج كلا الخوارزميتين في كود واحد لفحص الرسم البياني.

Graph.java

```
import java.util.ArrayList;

import java.util.List;


public class Graph {

    private final int V;

    private final List<Integer>[] adj;


    public Graph(int V) {

        this.V = V;

        adj = (List<Integer>[]) new List[V];

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

            adj[v] = new ArrayList<>();

        }

    }


    public void addEdge(int v, int w) {
```

```
        adj[v].add(w);
        adj[w].add(v); // For undirected graph
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public int V() {
        return V;
    }

    public int E() {
        return adj.length;
    }
}
```

GabowSCC.java

```
import java.util.*;

public class GabowSCC {
    private int[] id;        // component ids
    private int[] preorder;  // preorder numbers
    private boolean[] marked; // visited vertices
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;
    private int pre;
    private int count;       // number of strongly-connected components
```

```java
public GabowSCC(Digraph G) {
    id = new int[G.V()];
    preorder = new int[G.V()];
    marked = new boolean[G.V()];
    stack1 = new Stack<>();
    stack2 = new Stack<>();
    pre = 0;
    count = 0;
    for (int v = 0; v < G.V(); v++) {
        if (!marked[v]) {
            dfs(G, v);
        }
    }
}

private void dfs(Digraph G, int v) {
    marked[v] = true;
    preorder[v] = pre++;
    stack1.push(v);
    stack2.push(v);

    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        } else if (id[w] == 0) {
            while (!stack2.isEmpty() && preorder[stack2.peek()] > preorder[w]) {
                stack2.pop();
            }
```

```java
            }
        }

        if (stack2.peek() == v) {
            stack2.pop();
            int w;
            do {
                w = stack1.pop();
                id[w] = count;
            } while (w != v);
            count++;
        }
    }

    public boolean stronglyConnected(int v, int w) {
        return id[v] == id[w];
    }

    public int count() {
        return count;
    }

    public int id(int v) {
        return id[v];
    }
}
```

JensSchmidtAlgorithm.java

```
import java.util.*;

public class JensSchmidtAlgorithm {
    private boolean[] visited;
    private int[] disc;
    private int[] low;
    private int[] parent;
    private int time;
    private List<Integer> articulationPoints;

    public JensSchmidtAlgorithm(Graph graph) {
        int V = graph.V();
        visited = new boolean[V];
        disc = new int[V];
        low = new int[V];
        parent = new int[V];
        articulationPoints = new ArrayList<>();
        time = 0;

        Arrays.fill(parent, -1);

        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                dfs(graph, i);
            }
        }
    }
```

```java
private void dfs(Graph graph, int u) {
    int children = 0;
    visited[u] = true;
    disc[u] = low[u] = ++time;

    for (int v : graph.adj(u)) {
        if (!visited[v]) {
            children++;
            parent[v] = u;
            dfs(graph, v);

            low[u] = Math.min(low[u], low[v]);

            if (parent[u] == -1 && children > 1) {
                articulationPoints.add(u);
            }

            if (parent[u] != -1 && low[v] >= disc[u]) {
                articulationPoints.add(u);
            }
        } else if (v != parent[u]) {
            low[u] = Math.min(low[u], disc[v]);
        }
    }
}

public boolean hasArticulationPoints() {
    return !articulationPoints.isEmpty();
}
```

```
    public List<Integer> getArticulationPoints() {
        return articulationPoints;
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Digraph digraph = new Digraph(5);
        digraph.addEdge(0, 2);
        digraph.addEdge(2, 1);
        digraph.addEdge(1, 0);
        digraph.addEdge(0, 3);
        digraph.addEdge(3, 4);

        GabowSCC scc = new GabowSCC(digraph);

        if (scc.count() == 1) {
            System.out.println("The graph is strongly connected.");
        } else {
            System.out.println("The graph is not strongly connected.");
            return;
        }

        Graph graph = new Graph(5);
        graph.addEdge(0, 1);
```

```
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(3, 4);

        JensSchmidtAlgorithm algorithm = new JensSchmidtAlgorithm(graph);

        if (algorithm.hasArticulationPoints()) {
            System.out.println("The graph has articulation points: " +
algorithm.getArticulationPoints());
        } else {
            System.out.println("The graph has no articulation points.");
        }

        if (algorithm.hasArticulationPoints()) {
            System.out.println("The graph is not 2-vertex strongly connected.");
        } else {
            System.out.println("The graph is 2-vertex strongly connected.");
        }
    }
}
```