Connected Components in an Undirected Graph

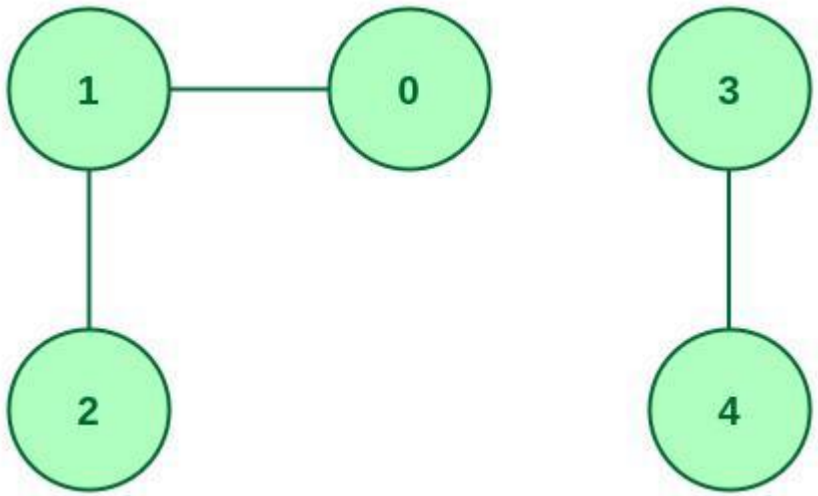
By

Dr GC Jana

Connected Components in an Undirected Graph

Problem Statement: Given an undirected graph, the task is to print all the connected components line by line.

Input: Consider the following graph



Output:
0 1 2
3 4

Explanation: There are 2 different connected components. They are {0, 1, 2} and {3, 4}.

Connected Components for undirected graph using DFS

Finding connected components for an undirected graph is an easier task. The idea is to

Do either BFS or DFS starting from every unvisited vertex, and we get all strongly connected components.

Connected Components for undirected graph using DFS

Follow the steps mentioned below to implement the idea using DFS:

- Initialize all vertices as not visited.
- Do the following for every vertex v:
  - If v is not visited before, call the DFS. and print the newline character to print each component in a new line
    - Mark v as visited and print v.
    - For every adjacent u of v, If u is not visited, then recursively call the DFS.

Connected Components for undirected graph using DFS

```
// Java program to print connected components in
// an undirected graph
import java.util.ArrayList;
class Graph {
    // A user define class to represent a graph.
    // A graph is an array of adjacency lists.
    // Size of array will be V (number of vertices
    // in graph)
    int V;
    ArrayList<ArrayList<Integer> > adjListArray;
```

```
// constructor
  Graph(int V)
  {
      this.V = V;
      // define the size of array as
      // number of vertices
      adjListArray = new ArrayList<>();

      // Create a new list for each vertex
      // such that adjacent nodes can be stored

      for (int i = 0; i < V; i++) {
          adjListArray.add(i, new ArrayList<>());
      }
  }
```

Connected Components for undirected graph using DFS

```
// Adds an edge to an undirected graph
   void addEdge(int src, int dest)
   {
      // Add an edge from src to dest.
      adjListArray.get(src).add(dest);

      // Since graph is undirected, add an edge from dest
      // to src also
      adjListArray.get(dest).add(src);
   }
```

```
void DFSUtil(int v, boolean[] visited)
   {
      // Mark the current node as visited and print it
      visited[v] = true;
      System.out.print(v + " ");
      // Recur for all the vertices
      // adjacent to this vertex
      for (int x : adjListArray.get(v)) {
         if (!visited[x])
            DFSUtil(x, visited);
      }
   }
```

Connected Components for undirected graph using DFS

```
void connectedComponents()
   {
      // Mark all the vertices as not visited
      boolean[] visited = new boolean[V];
      for (int v = 0; v < V; ++v) {
         if (!visited[v]) {
            // print all reachable vertices
            // from v
            DFSUtil(v, visited);
            System.out.println();
         }
      }
   }
```

```
// Driver code
   public static void main(String[] args)
   {
      // Create a graph given in the above diagram
      Graph g = new Graph(5);

      g.addEdge(1, 0);
      g.addEdge(2, 1);
      g.addEdge(3, 4);
      System.out.println(
         "Following are connected components");
      g.connectedComponents();
   }
}
```

Output

Following are connected components
0 1 2
3 4

Connected Components for undirected graph using DFS

Time Complexity: O(V + E) where V is the number of vertices and E is the number of edges.

Auxiliary Space: O(V)

Connected Component for undirected graph using Disjoint Set Union:

The idea to solve the problem using DSU (Disjoint Set Union) is

Initially, declare all the nodes as individual subsets and then visit them. When a new unvisited node is encountered, unite it with the under. In this manner, a single component will be visited in each traversal.

Connected Component for undirected graph using Disjoint Set Union:

Follow the below steps to implement the idea:

- Declare an array **arr[]** of size **V** where **V** is the total number of nodes.
- For every index **i** of array **arr[],** the value denotes who the **parent** of **ith** vertex is.
- Initialize every node as the parent of itself and then while adding them together, **change their parents accordingly**.
- Traverse the nodes from **0 to V**:
  - For each node that is the parent of itself start the DSU.
  - Print the nodes of that disjoint set as they belong to one component.

Connected Component for undirected graph using Disjoint Set Union:

```
import java.util.*;

class ConnectedComponents {
    public static int merge(int[] parent, int x) {
        if (parent[x] == x)
            return x;
        return merge(parent, parent[x]);
    }

    public static int connectedComponents(int n,
List<List<Integer>> edges) {
        int[] parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        for (List<Integer> x : edges) {
            parent[merge(parent, x.get(0))] = merge(parent,
x.get(1));
        }

        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (parent[i] == i) ans++;
        }

        for (int i = 0; i < n; i++) {
            parent[i] = merge(parent, parent[i]);
        }

        Map<Integer, List<Integer>> m = new HashMap<>();
        for (int i = 0; i < n; i++) {
            m.computeIfAbsent(parent[i], k -> new
ArrayList<>()).add(i);
        }
```

Connected Component for undirected graph using Disjoint Set Union:

```
for (Map.Entry<Integer, List<Integer>> it : m.entrySet()) {
        List<Integer> l = it.getValue();
        for (int x : l) {
           System.out.print(x + " ");
        }
        System.out.println();
    }
    return ans;
  }
```

```
public static void main(String[] args) {
      int n = 5;
      List<List<Integer>> edges = new ArrayList<>();
      edges.add(Arrays.asList(0, 1));
      edges.add(Arrays.asList(2, 1));
      edges.add(Arrays.asList(3, 4));

      System.out.println("Following are connected
components:");
      int ans = connectedComponents(n, edges);
   }
}
```

Reference

- https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/