CC.java

Below is the syntax highlighted version of CC.java from §4.1 Undirected Graphs.

```
/******************************************************************************
 *  Compilation:  javac CC.java
 *  Execution:    java CC filename.txt
 *  Dependencies: Graph.java StdOut.java Queue.java
 *  Data files:   https://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                https://algs4.cs.princeton.edu/41graph/mediumG.txt
 *                https://algs4.cs.princeton.edu/41graph/largeG.txt
 *
 *  Compute connected components using depth first search.
 *  Runs in O(E + V) time.
 *
 *  % java CC tinyG.txt
 *  3 components
 *  0 1 2 3 4 5 6
 *  7 8
 *  9 10 11 12
 *
 *  % java CC mediumG.txt
 *  1 components
 *  0 1 2 3 4 5 6 7 8 9 10 ...
 *
 *  % java -Xss50m CC largeG.txt
 *  1 components
 *  0 1 2 3 4 5 6 7 8 9 10 ...
 *
 *  Note: This implementation uses a recursive DFS. To avoid needing
 *        a potentially very large stack size, replace with a nonrecursive
 *        DFS ala NonrecursiveDFS.java.
 *
 ******************************************************************************/

/**
 *  The {@code CC} class represents a data type for
 *  determining the connected components in an undirected graph.
 *  The <em>id</em> operation determines in which connected component
 *  a given vertex lies; the <em>connected</em> operation
 *  determines whether two vertices are in the same connected component;
 *  the <em>count</em> operation determines the number of connected
 *  components; and the <em>size</em> operation determines the number
 *  of vertices in the connect component containing a given vertex.

 *  The <em>component identifier</em> of a connected component is one of the
 *  vertices in the connected component: two vertices have the same component
 *  identifier if and only if they are in the same connected component.

 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes &Theta;(<em>V</em> + <em>E</em>) time,
 *  where <em>V</em> is the number of vertices and <em>E</em> is the
 *  number of edges.
 *  Each instance method takes &Theta;(1) time.
 *  It uses &Theta;(<em>V</em>) extra space (not including the graph).
 *  <p>
 *  For additional documentation, see
 *  <a href="https://algs4.cs.princeton.edu/41graph">Section 4.1</a>
 *  of <em>Algorithms, 4th Edition</em> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class CC {
    private boolean[] marked;   // marked[v] = has vertex v been marked?
    private int[] id;           // id[v] = id of connected component containing v
    private int[] size;         // size[id] = number of vertices in given component
    private int count;          // number of connected components

    /**
     * Computes the connected components of the undirected graph {@code G}.
     *
     * @param G the undirected graph
     */
    public CC(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        size = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    /**
     * Computes the connected components of the edge-weighted graph {@code G}.
     *
     * @param G the edge-weighted graph
     */
    public CC(EdgeWeightedGraph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        size = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    // depth-first search for a Graph
    private void dfs(Graph G, int v) {
        marked[v] = true;
        id[v] = count;
        size[count]++;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }

    // depth-first search for an EdgeWeightedGraph
    private void dfs(EdgeWeightedGraph G, int v) {
        marked[v] = true;
        id[v] = count;
        size[count]++;
        for (Edge e : G.adj(v)) {
            int w = e.other(v);
            if (!marked[w]) {
```

```
            dfs(G, w);
        }
    }
}


/**
 * Returns the component id of the connected component containing vertex {@code v}.
 *
 * @param  v the vertex
 * @return the component id of the connected component containing vertex {@code v}
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public int id(int v) {
    validateVertex(v);
    return id[v];
}

/**
 * Returns the number of vertices in the connected component containing vertex {@code v}.
 *
 * @param  v the vertex
 * @return the number of vertices in the connected component containing vertex {@code v}
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public int size(int v) {
    validateVertex(v);
    return size[id[v]];
}

/**
 * Returns the number of connected components in the graph {@code G}.
 *
 * @return the number of connected components in the graph {@code G}
 */
public int count() {
    return count;
}

/**
 * Returns true if vertices {@code v} and {@code w} are in the same
 * connected component.
 *
 * @param  v one vertex
 * @param  w the other vertex
 * @return {@code true} if vertices {@code v} and {@code w} are in the same
 *         connected component; {@code false} otherwise
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 * @throws IllegalArgumentException unless {@code 0 <= w < V}
 */
public boolean connected(int v, int w) {
    validateVertex(v);
    validateVertex(w);
    return id(v) == id(w);
}

/**
 * Returns true if vertices {@code v} and {@code w} are in the same
 * connected component.
 *
 * @param  v one vertex
 * @param  w the other vertex
 * @return {@code true} if vertices {@code v} and {@code w} are in the same
 *         connected component; {@code false} otherwise
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
```

```
     * @throws IllegalArgumentException unless {@code 0 <= w < V}
     * @deprecated Replaced by {@link #connected(int, int)}.
     */
    @Deprecated
    public boolean areConnected(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        return id(v) == id(w);
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
    private void validateVertex(int v) {
        int V = marked.length;
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0 and " + (V-1));
    }

    /**
     * Unit tests the {@code CC} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        CC cc = new CC(G);

        // number of connected components
        int m = cc.count();
        StdOut.println(m + " components");

        // compute list of vertices in each connected component
        Queue<Integer>[] components = (Queue<Integer>[]) new Queue[m];
        for (int i = 0; i < m; i++) {
            components[i] = new Queue<Integer>();
        }
        for (int v = 0; v < G.V(); v++) {
            components[cc.id(v)].enqueue(v);
        }

        // print results
        for (int i = 0; i < m; i++) {
            for (int v : components[i]) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
    }
}
```

Last updated: Sun Nov 27 05:45:28 EST 2022.