Cycle.java

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

```
/******************************************************************************
 *  Compilation:  javac Cycle.java
 *  Execution:    java  Cycle filename.txt
 *  Dependencies: Graph.java Stack.java In.java StdOut.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
 *
 *  Identifies a cycle.
 *  Runs in O(E + V) time.
 *
 *  % java Cycle tinyG.txt
 *  3 4 5 3
 *
 *  % java Cycle mediumG.txt
 *  15 0 225 15
 *
 *  % java Cycle largeG.txt
 *  996673 762 840164 4619 785187 194717 996673
 *
 ******************************************************************************/

/**
 *  The {@code Cycle} class represents a data type for
 *  determining whether an undirected graph has a simple cycle.
 *  The <em>hasCycle</em> operation determines whether the graph has
 *  a cycle and, if so, the <em>cycle</em> operation returns one.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes &Theta;(<em>V</em> + <em>E</em>) time in the
 *  worst case, where <em>V</em> is the number of vertices and
 *  <em>E</em> is the number of edges.
 *  (The depth-first search part takes only <em>O</em>(<em>V</em>) time;
 *  however, checking for self-loops and parallel edges takes
 *  &Theta;(<em>V</em> + <em>E</em>) time in the worst case.
 *  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 <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Cycle {
    private boolean[] marked;
    private int[] edgeTo;
    private Stack<Integer> cycle;

    /**
     * Determines whether the undirected graph {@code G} has a cycle and,
     * if so, finds such a cycle.
     *
```

```
     * @param G the undirected graph
     */
    public Cycle(Graph G) {
        // need special case to identify parallel edge as a cycle
        if (hasParallelEdges(G)) return;

        // don't need special case to identify self-loop as a cycle
        // if (hasSelfLoop(G)) return;

        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v])
                dfs(G, -1, v);
    }


    // does this graph have a self loop?
    // side effect: initialize cycle to be self loop
    private boolean hasSelfLoop(Graph G) {
        for (int v = 0; v < G.V(); v++) {
            for (int w : G.adj(v)) {
                if (v == w) {
                    cycle = new Stack<Integer>();
                    cycle.push(v);
                    cycle.push(v);
                    return true;
                }
            }
        }
        return false;
    }

    // does this graph have two parallel edges?
    // side effect: initialize cycle to be two parallel edges
    private boolean hasParallelEdges(Graph G) {
        marked = new boolean[G.V()];

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

            // check for parallel edges incident to v
            for (int w : G.adj(v)) {
                if (marked[w]) {
                    cycle = new Stack<Integer>();
                    cycle.push(v);
                    cycle.push(w);
                    cycle.push(v);
                    return true;
                }
                marked[w] = true;
            }

            // reset so marked[v] = false for all v
            for (int w : G.adj(v)) {
                marked[w] = false;
            }
        }
        return false;
    }

    /**
     * Returns true if the graph {@code G} has a cycle.
     *
     * @return {@code true} if the graph has a cycle; {@code false} otherwise
     */
```

```
    public boolean hasCycle() {
        return cycle != null;
    }

     /**
     * Returns a cycle in the graph {@code G}.
     * @return a cycle if the graph {@code G} has a cycle,
     *         and {@code null} otherwise
     */
    public Iterable<Integer> cycle() {
        return cycle;
    }

    private void dfs(Graph G, int u, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {

            // short circuit if cycle already found
            if (cycle != null) return;

            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, v, w);
            }

            // check for cycle (but disregard reverse of edge leading to v)
            else if (w != u) {
                cycle = new Stack<Integer>();
                for (int x = v; x != w; x = edgeTo[x]) {
                    cycle.push(x);
                }
                cycle.push(w);
                cycle.push(v);
            }
        }
    }

    /**
     * Unit tests the {@code Cycle} 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);
        Cycle finder = new Cycle(G);
        if (finder.hasCycle()) {
            for (int v : finder.cycle()) {
                StdOut.print(v + " ");
            }
            StdOut.println();
        }
        else {
            StdOut.println("Graph is acyclic");
        }
    }

}
```