```
/**************************************************************************
 * File: EdmondsMatching.java
 * Author: Keith Schwarz (htiek@cs.stanford.edu)
 *
 * An implementation of Edmonds's Maximum Matching algorithm for general
 * graphs.  This algorithm is comparatively old -
it dates back to 1965 - and
 * is not particularly efficient compared to modern algorithms (it runs in
 * O(n^2 m), when O(sqrt(n) m) algorithms now exist).  However, Edmonds's
 * algorithm is extremely theoretically elegant and can be implemented easily
 * using standard, off-the-shelf utility classes.
 *
 * The key concept powering Edmonds's algorithm is the notion of an alternating
 * path.  Given a graph and matching (G, M), consider any path P in G.  Some of
 * the edges in P will also be in M, while others will not be.  Let's call an
 * edge in the path that's also in M "solid," while an edge in the path not in
 * M "dashed."  A path is called alternating if the edges alternate between
 * solid and dashed.  For example, this path is alternating:
 *
 *              * ======== * = = = = = * ========= * = = = = = *
 *
 * As is this one:
 *
 *                            * = = = = = * ========= *
 *
 * Note that this also means that a single edge is automatically an alternating
 * path.
 *
 * The main theorem on which Edmonds's matching algorithm is based is the
 * following:
 *
 * "Given a matching (G, M), M is a maximum matching iff G contains no
 *  alternating paths between exposed vertices."
 *
 * There are several good proofs of this claim.  One direction is easy -
if the
 * graph contains an alternating path between exposed vertices, then we can
 * construct a larger matching by replacing the matched edges in the path with
 * the unmatched edges.  That is, we change
 *
 *          * = = = = * ======= * = = = = * ======= * = = = = *
 *
 * into
 *
 *          * ======= * = = = = * ======= * = = = = * ======= *
 *
 * This matching has edge more than the previous matching, so M isn't maximum.
 * The inverse direction is a bit trickier; it works by considering the set
 * symmetric difference between a maximum matching and any other matching, then
 * looking at the connected components of this graph; one can be shown to be
 * the necessary alternating path.
 *
 * A collection of alternating paths can be formed together to create an
 * "alternating tree."  An alternating tree is a tree defined with respect to
 * a matching (G, M) as follows:
 *
 *  1. The nodes are partitioned into two disjoint groups -
"outer vertices"
 *     and "inner vertices."
 *  2. Each inner vertex has degree two, and exactly one of its incident edges
 *     is in M.
 *  3. Each edge connects exactly one inner node and exactly one outer node.
 *  4. There is a unique outer node that is the root of the tree.
```

```
 *  5. Every path from a vertex to the root is an alternating path.
 *  6. Every path from an outer vertex to the root begins with an edge in M.
 *
 * This is a lot of structure to take in at once, so a more informal
 * presentation is probably in order.  You can think of an alternating tree as
 * a tree formed as the concatenation of several "gadgets" formed of a solid
 * edge (an edge in M) joined to a dashed edge (an edge not in M).  The
 * alternating tree is then a collection of paths formed from these gadgets
 * concatenated together in such a way that the endpoint of each gadget is
 * either a designated root node or the start point of some other gadget.  The
 * "outer vertices" from (1) are then the start nodes of the gadgets, while the
 * "inner vertices" from (1) are the inner nodes.  This automatically
 * guarantees (2) and (3).  The special root node is the unique outer node
 * mentioned in (4), and the construction easily guarantees (5) and (6).
 *
 * The reason that these "alternating trees" are so useful is that they give a
 * great way to locate alternating paths between exposed vertices.  Suppose
 * that we root an alternating tree at an exposed vertex and then find an
 * edge from an outer node to an exposed vertex.  This means that we then have
 * an alternating path from that exposed vertex to the root of the tree, formed
 * by taking the edge from the exposed vertex to the outer node and the path
 * from the outer node to the root of the tree (which is alternating and starts
 * with a matched edge).  The algorithm thus works by growing this alternating
 * tree from an arbitrary exposed node outward until either every node is part
 * of the tree (in which case no alternating path exists and we have a maximum
 * matching), or until an alternating path is found.
 *
 * There is one last detail to worry about, and that's what happens if there is
 * an unmatched edge in the graph that links together two from two outer nodes
 * of the tree.  If this happens, this means that there is an odd cycle in the
 * graph, and it might be possible for an alternating path to exist in the
 * graph that wouldn't be noticed by the tree (you should draw out a picture
 * here to convince yourself that this is true).  When this happens, the
 * algorithm "contracts" the odd cycle down to a single node, then recursively
 * searches this new graph for an alternating path.  If one exists and doesn't
 * pass through this new pseudonode, then it's an alternating path in the old
 * graph.  If it does pass through the new pseudonode, then the path can be
 * expanded back into a path in the original graph by modifying the matching
 * so that the path through the cycle back to the root is alternating (this
 * doesn't affect the cardinality of the matching), then returning the proper
 * alternating path.
 *
 * The algorithm can make at most O(n) iterations of finding alternating paths,
 * since a maximum matching has cardinality at most n/2.  During each iteration
 * of the tree-
growing step, we need to consider each edge at most once, since
 * it either goes in the tree, finds an augmenting path, or creates a cycle.
 * Whenever a cycle is found, we can create the graph formed by contracting
 * that cycle in O(m) by scanning over the edges.  Finally, we can have at most
 * O(n) contractions, because each contraction removes at least one node.  This
 * gives the algorithm a runtime of O(n^2 m).
 */
import java.util.*; // For ArrayDeque, HashMap, HashSet, LinkedList

public final class EdmondsMatching {
    /**
     * Given an undirected graph, returns a graph containing the edges of a
     * maximum matching in that graph.
     *
     * @param g The graph in which a maximum matching should be found.
     * @return A graph containing a maximum matching in that graph.
     */
    public static <T> UndirectedGraph<T> maximumMatching(UndirectedGraph<T> g) {
        /* Edge case -
```

```
 if the graph is empty, just hand back an empty graph as
         * the matching.
         */
        if (g.isEmpty())
            return new UndirectedGraph<T>();

        /* Construct a new graph to hold the matching.  Fill it with the nodes
         * from the original graph, but not the edges.
         */
        UndirectedGraph<T> result = new UndirectedGraph<T>();
        for (T node: g)
            result.addNode(node);

        /* Now, continuously iterate, looking for alternating paths between
         * exposed vertices until none can be found.
         */
        while (true) {
            /* Look up a path.  If none exists, this function returns null as
             * a sentinel, which we can use as a cue that we're done.
             */
            List<T> path = findAlternatingPath(g, result);
            if (path == null) return result;

            /* If not, update the graph using this alternating path. */
            updateMatching(path, result);
        }
    }

    /**
     * Updates a matching by increasing the cardinality of the matching by
     * flipping the status of the edges in the specified alternating path.  It
     * is assumed that the alternating path is between exposed vertices, which
     * means that the edges [0, 1], [2, 3], ..., [n-2, n-
1] are assumed not to
     * be in the matching.
     *
     * @param path The alternating path linking the exposed endpoints.
     * @param m The matching to update, which is an in/out parameter.
     */
    private static <T> void updateMatching(List<T> path, UndirectedGraph<T> m) {
        /* Scan across the edges in the path, flipping whether or not they're
         * in the matching.  This iteration counts up to the size of the list
         * minus one because each iteration looks at the edge (i, i + 1).
         */
        for (int i = 0; i < path.size() - 1; ++i) {
            /* Check whether the edge exists and react appropriately. */
            if (m.edgeExists(path.get(i), path.get(i + 1)))
                m.removeEdge(path.get(i), path.get(i + 1));
            else
                m.addEdge(path.get(i), path.get(i + 1));
        }
    }

    /* A key step of the algorithm works by building up a collection of
     * forests of alternating trees.  We will need to be able to answer
     * the following queries about the tree structure:
     *
     * 1. For each node, what tree is it a part of?
     * 2. For each node, what is its parent node in the tree?
     * 3. For each node, is it an inner or outer node in its tree?
     *
     * We will represent all of this information by a utility struct that
     * stores this information.  For simplicity, each node will be
     * identified with its root.
```

```
 */
private static final class NodeInformation<T> {
    public final T parent;
    public final T treeRoot;
    public final boolean isOuter; // True for outer node, false for inner.

    /**
     * Constructs a new NodeInformation wrapping the indicated data.
     *
     * @param parent The parent of the given node.
     * @param treeRoot The root of the given node.
     * @param isOuter Whether the given node is an outer node.
     */
    public NodeInformation(T parent, T treeRoot, boolean isOuter) {
        this.parent = parent;
        this.treeRoot = treeRoot;
        this.isOuter = isOuter;
    }
}

/* A utility struct representing an edge in the graph. */
private static final class Edge<T> {
    public final T start;
    public final T end;

    /**
     * Constructs a new edge between the two indicated endpoints.
     *
     * @param start The edge's starting point.
     * @param end The edge's endpoint.
     */
    public Edge(T start, T end) {
        this.start = start;
        this.end = end;
    }
}

/* One final piece of information that's necessary to get the algorithm
 * working is the ability to locate and navigate odd alternating cycles in
 * the alternating forest.  Once we've found such a cycle, we need to
 * contract it down to a single node, find an alternating path in the
 * contracted graph, then expand the path back into an alternating path in
 * the original graph.  To do this, we need to be able to answer the
 * following questions efficiently:
 *
 * 1. What nodes are in the cycle (blossom)?
 * 2. Starting at the root of the blossom, what order do the edges go in?
 * 3. What node is used to represent the cycle in the contracted graph?
 *
 * This information is encoded in this utility struct.
 */
private static final class Blossom<T> {
    public final T root; // The root of the blossom; also the representative
    public final List<T> cycle; // The nodes, listed in order around the cycle
    public final Set<T> nodes; // The nodes, stored in a set for efficient lookup

    /**
     * Given information about the blossom, constructs a new Blossom
     * holding this information.
     *
     * @param root The root node of the blossom.
     * @param cycle The nodes of the cycle listed in the order in which
     *              they appear in the blossom.
     * @param nodes The nodes in the cycle, stored as a set.
```

```
         */
        public Blossom(T root, List<T> cycle, Set<T> nodes) {
            this.root = root;
            this.cycle = cycle;
            this.nodes = nodes;
        }
    }

    /**
     * Given a graph and a matching in that graph, returns an augmenting path
     * in the graph if one exists.
     *
     * @param g The graph in which to search for the path.
     * @param m A matching in that graph.
     * @return An alternating path in g, or null if none exists.
     */
    private static <T> List<T> findAlternatingPath(UndirectedGraph<T> g,
                                                   UndirectedGraph<T> m) {
        /* We need to maintain as state all of the forests that are currently
         * being considered.  To do this, we'll create a map associating each
         * node with its information.
         */
        Map<T, NodeInformation<T>> forest = new HashMap<T, NodeInformation<T>>();

        /* We also will maintain a worklist of edges that need to be
         * considered.  These will be explored in a breadth-
first fashion.
         * Whenever we add a new node to the forest, we'll add all its
         * outgoing edges to this queue.
         */
        Queue<Edge<T>> worklist = new LinkedList<Edge<T>>();

        /* Begin by adding all of the exposed vertices to the forest as their
         * own singleton trees.
         */
        for (T node: g) {
            /* Check if the node is a singleton by seeing if it has no edges
             * leaving it in the matching.
             */
            if (!m.edgesFrom(node).isEmpty())
                continue;

            /* This node is an outer node that has no parent and belongs in
             * its own tree.
             */
            forest.put(node, new NodeInformation<T>(null, node, true));

            /* Add to the worklist all edges leaving this node. */
            for (T endpoint: g.edgesFrom(node))
                worklist.add(new Edge<T>(node, endpoint));
        }

        /* Now, we start growing all the trees outward by considering edges
         * leaving each tree.
         */
        while (!worklist.isEmpty()) {
            /* Grab the next edge.  We're only growing the tree along edges
             * that are not part of the matching (since we're looking to grow
             * trees with pairs of edges, a non-
matched edge from an outer
             * node to an inner node, followed by a matched edge to some next
             * node.
             */
            Edge<T> curr = worklist.remove();
```

```
            if (m.edgeExists(curr.start, curr.end))
                continue;

            /* Look up the information associated with the endpoints. */
            NodeInformation<T> startInfo = forest.get(curr.start);
            NodeInformation<T> endInfo = forest.get(curr.end);

            /* We have several cases to consider.  First, if the endpoint of
             * this edge is in some tree, there are two options:
             *
             * 1. If both endpoints are outer nodes in the same tree, we have
             *    found an odd-
length cycle (blossom).  We then contract the
             *    edges in the cycle, repeat the search in the contracted
             *    graph, then expand the result.
             * 2. If both endpoints are outer nodes in different trees, then
             *    we've found an augmenting path from the root of one tree
             *    down through the other.
             * 3. If one endpoint is an outer node and one is an inner node,
             *    we don't need to do anything.  The path that we would end
             *    up taking from the root of the first tree through this edge
             *    would not end up at the root of the other tree, since the
             *    only way we could do this while alternating would direct us
             *    away from the root.  We can just skip this edge.
             */
            if (endInfo != null) {
                /* Case 1: Do the contraction. */
                if (endInfo.isOuter && startInfo.treeRoot == endInfo.treeRoot) {
                    /* Get information about the blossom necessary to reduce
                     * the graph.
                     */
                    Blossom<T> blossom = findBlossom(forest, curr);

                    /* Next, rebuild the graph using the indicated pseudonode,
                     * and recursively search it for an augmenting path.
                     */
                    List<T> path = findAlternatingPath(contractGraph(g, blossom),
                                                       contractGraph(m, blossom));

                    /* If no augmenting path exists, then no augmenting path
                     * exists in this graph either.
                     */
                    if (path == null) return path;

                    /* Otherwise, expand the path out into a path in this
                     * graph, then return it.
                     */
                    return expandPath(path, g, forest, blossom);
                }
                /* Case 2: Return the augmenting path from root to root. */
                else if (endInfo.isOuter && startInfo.treeRoot != endInfo.treeRoot) {
                    /* The alternating path that we'll be building consists of
                     * the path from the root of the first tree to the first
                     * outer node, followed by the edge, followed by the path
                     * from the second outer node to its root.  Our path info
                     * is stored in a fashion that makes it easy to walk up
                     * to the root, and so we'll build up the path (which
                     * we'll represent by a deque) by walking up to the tree
                     * roots and creating the path as appropriate.
                     */
                    List<T> result = new ArrayList<T>();

                    /* Get the path from the first node to the root.  Note
                     * that this path is backwards, so we'll need to reverse
```

```
                 * it afterwards.
                 */
                for (T node = curr.start; node != null; node = forest.get(node).parent)
                    result.add(node);

                /* Turn the path around. */
                result = reversePath(result);

                /* Path from edge end to its root. */
                for (T node = curr.end; node != null; node = forest.get(node).parent)
                    result.add(node);

                return result;
            }
            /* Case 3 requires no processing. */
        }
        /* Otherwise, we have no info on this edge, which means that it
         * must correspond to a matched node (all exposed nodes are in a
         * forest).  We'll thus add that node to the tree containing the
         * start of the endpoint as an inner node, then add the node for
         * its endpoint to the tree as an outer node.
         */
        else {
            /* The endpoint has the edge start as a parent and the same
             * root as its parent.
             */
            forest.put(curr.end, new NodeInformation<T>(curr.start,
                                                        startInfo.treeRoot,
                                                        false));

            /* Look up the unique edge that is matched to this node.  Its
             * endpoint is the node that will become an outer node of this
             * tree.
             */
            T endpoint = m.edgesFrom(curr.end).iterator().next();
            forest.put(endpoint, new NodeInformation<T>(curr.end,
                                                        startInfo.treeRoot,
                                                        true));

            /* Add all outgoing edges from this endpoint to the work
             * list.
             */
            for (T fringeNode: g.edgesFrom(endpoint))
                worklist.add(new Edge<T>(endpoint, fringeNode));
        }
    }

    /* If we reach here, it means that we've constructed a maximum forest
     * without finding any augmenting paths.
     */
    return null;
}

/**
 * Given a forest of alternating trees and an edge forming a blossom in
 * one of those trees, returns information about the blossom.
 *
 * @param forest The alternating forest.
 * @param edge The edge that created a cycle.
 * @return A Blossom struct holding information about then blossom.
 */
private static <T> Blossom<T> findBlossom(Map<T, NodeInformation<T>> forest,
                                          Edge<T> edge) {
    /* We need to locate the root of the blossom, which is the lowest
```

```
         * common ancestor of the two nodes at the endpoint of each edge.  To
         * do this, we'll walk up from each node to the root, storing the
         * paths we find.  We will then look for the last node that's common
         * to both paths.
         */
        LinkedList<T> onePath = new LinkedList<T>(), twoPath = new LinkedList<T>();
        for (T node = edge.start; node != null; node = forest.get(node).parent)
            onePath.addFirst(node);
        for (T node = edge.end; node != null; node = forest.get(node).parent)
            twoPath.addFirst(node);

        /* Now that we have that paths, continue walking forward in them until
         * we find a mismatch.
         */
        int mismatch = 0;
        for (; mismatch < onePath.size() && mismatch < twoPath.size(); ++mismatch)
            if (onePath.get(mismatch) != twoPath.get(mismatch))
                break;

        /* At this point, we know that the mismatch occurs at index right
         * before mismatch.  Because both nodes are part of the same tree, we
         * know that they have the same root, and so the mismatch index is
         * nonzero.
         *
         * From here, we can recover the cycles by walking down from the root
         * to the first node, across the indicated edge, and then back up the
         * path from the second node to the cycle.
         */
        List<T> cycle = new ArrayList<T>();
        for (int i = mismatch - 1; i < onePath.size(); ++i)
            cycle.add(onePath.get(i));
        for (int i = twoPath.size() - 1; i >= mismatch - 1; --
i)
            cycle.add(twoPath.get(i));

        return new Blossom<T>(onePath.get(mismatch -
 1), cycle, new HashSet<T>(cycle));
    }

    /**
     * Given a graph and a blossom, returns the contraction of the graph
     * around that blossom.
     *
     * @param g The graph to contract.
     * @param blossom The set of nodes in the blossom.
     * @return The contraction g / blossom.
     */
    private static <T> UndirectedGraph<T> contractGraph(UndirectedGraph<T> g,
                                                        Blossom<T> blossom) {
        /* The contraction of the graph is the modified graph where:
         * 1. All nodes in the blossom are removed.
         * 2. There is a new node, the pseudonode.
         * 3. All edges between nodes in the blossom are removed.
         * 4. All edges between nodes out of the blossom and nodes in the
         *    blossom are replaced by an edge to the pseudonode.
         */
        UndirectedGraph<T> result = new UndirectedGraph<T>();

        /* Begin by adding all nodes not in the blossom. */
        for (T node: g) {
            if (!blossom.nodes.contains(node))
                result.addNode(node);
        }
```

```
    /* Add the pseudonode in. */
    result.addNode(blossom.root);

    /* Add each edge in, adjusting the endpoint as appropriate. */
    for (T node: g) {
        /* Skip nodes in the blossom; they're not included in the
         * contracted graph.
         */
        if (blossom.nodes.contains(node)) continue;

        /* Explore all nodes connected to this one. */
        for (T endpoint: g.edgesFrom(node)) {
            /* If this endpoint is in the blossom, pretend that it's now
             * an edge to the pseudonode.
             */
            if (blossom.nodes.contains(endpoint))
                endpoint = blossom.root;

            /* Add the edge to the graph, accounting for the fact that it
             * might now be transformed.
             */
            result.addEdge(node, endpoint);
        }
    }

    return result;
}

/**
 * Given an alternating path in a graph formed by the contraction of
 * a blossom into a pseudonode, along with the alternating forest in the
 * original graph, returns a new alternating path in the original graph
 * formed by expanding the path if it goes through a pseudonode.
 *
 * @param path The path in the contracted graph.
 * @param g The uncontracted graph.
 * @param forest The alternating forest of the original graph.
 * @param blossom The blossom that was contracted.
 * @param pseudonode The pseudonode representing the blossom in the
 *                   contracted graph.
 * @return An alternating path in the original graph.
 */
private static <T> List<T> expandPath(List<T> path,
                                      UndirectedGraph<T> g,
                                      Map<T, NodeInformation<T>> forest,
                                      Blossom<T> blossom) {
    /* Any path in the contracted graph will have at most one instance of
     * the blossom's pseudonode in it.  This node will have at most one
     * dotted (unmatched) edge incident to it and one solid (matched)
     * edge.  When expanding the node, the solid edge will end up incident
     * to the root of the blossom, and the dotted edge will leave the
     * blossom through some edge.  The challenge of this function is as
     * follows: given a path that may or may not pass through this
     * blossom, find some edge leaving the blossom to the next node in the
     * chain, then route the path through the blossom in such a way that
     * the path is alternating.
     *
     * To do this, we'll make a simplifying assumption that the path is
     * oriented such that as we start from the front and walk toward the
     * end, we enter the blossom's pseudonode through a solid edge and
     * leave through a dotted edge.  Dotted edges are at indices (0, 1),
     * (2, 3), ..., (2i, 2i + 1), and so we're going to want the
     * pseudonode to appear at an even index.  Because the path is
     * alternating, there's an odd number of edges in the path, and
```

```
         * therefore our path has an even number of nodes.  Consequently, if
         * the pseudonode exists in the path, then either it's at an even
         * index (in which case our assumption holds), or its at an odd index
         * (in which case we need to reverse our list, converting the odd
         * index into an even index).
         */
        int index = path.indexOf(blossom.root);

        /* If the node doesn't exist at all, our path is valid. */
        if (index == -1) return path;

        /* If the node is at an odd index, reverse the list and recompute the
         * index.
         */
        if (index % 2 == 1)
            path = reversePath(path);

        /* Now that we have the pseudonode at an even index (the start of a
         * dotted edge), we can start expanding out the path.
         */
        List<T> result = new ArrayList<T>();
        for (int i = 0; i < path.size(); ++i) {
            /* Look at the current node.  If it's not the pseudonode, then add
             * it into the resulting path with no modifications.
             */
            if (path.get(i) != blossom.root) {
                result.add(path.get(i));
            }
            /* Otherwise, we are looking at the pseudonode, which must be at
             * the start of a dotted edge.  We need to look at the next node
             * in the path to determine how to expand this pseudonode.  In
             * particular, we need to find some node in the cycle that the
             * next node in the path connects to, then need to route the path
             * around the cycle to that node.
             */
            else {
                /* Add the blossom's root to the path, since any path entering
                 * the blossom on a solid edge must come in here.
                 */
                result.add(blossom.root);

                /* Find some node in the cycle with an edge to the next node
                 * in the path.
                 */
                T outNode = findNodeLeavingCycle(g, blossom, path.get(i + 1));

                /* Look up where this node is in the cycle. */
                int outIndex = blossom.cycle.indexOf(outNode);

                /* When expanding out this path around the cycle, we need to
                 * ensure that the path we take ends by following a solid
                 * edge, since the next edge that we'll be taking is dashed.
                 * There are two possible ways to navigate the cycle -
one
                 * going clockwise and one going counterclockwise.  If the
                 * index of the outgoing node is even, then the path through
                 * the cycle in the forward direction will end by following a
                 * solid edge.  If it's odd, then the path through the cycle
                 * in the reverse direction ends with an outgoing edge.  We'll
                 * choose which way to go accordingly.
                 *
                 * The cycle we've stored has the root of the blossom at
                 * either endpoint, and so we'll skip over it in this
                 * iteration.
```

```
                 */
                int start = (outIndex % 2 == 0)? 1 : blossom.cycle.size() -
2;
                int step  = (outIndex % 2 == 0)? +1 : -1;

                /* Walk along the cycle accordingly. */
                for (int k = start; k != step; k += step)
                    result.add(blossom.cycle.get(k));
            }
        }
        return result;
    }

    /**
     * Given a path, returns the path formed by reversing the order of the
     * nodes in that path.
     *
     * @param path The path to reverse.
     * @return The reverse of that path.
     */
    private static <T> List<T> reversePath(List<T> path) {
        List<T> result = new ArrayList<T>();

        /* Visit the path in the reverse order. */
        for (int i = path.size() - 1; i >= 0; --i)
            result.add(path.get(i));

        return result;
    }

    /**
     * Given a graph, a blossom in that graph, and a node in that graph,
     * returns a node in the blossom that has an outgoing edge to that node.
     * If multiple nodes in the blossom have such an edge, one of them is
     * chosen arbitrarily.
     *
     * @param g The graph in which the cycle occurs.
     * @param blossom The blossom in that graph.
     * @param node The node outside of the blossom.
     * @return Some node in the blossom with an edge in g to the indicated
     *         node.
     */
    private static <T> T findNodeLeavingCycle(UndirectedGraph<T> g,
                                              Blossom<T> blossom,
                                              T node) {
        /* Check each node in the blossom for a matching edge. */
        for (T cycleNode: blossom.nodes)
            if (g.edgeExists(cycleNode, node))
                return cycleNode;

        /* If we got here, something is wrong because the node in question
         * should have some edge into it.
         */
        throw new AssertionError("Could not find an edge out of the blossom.");
    }
}
```