

INFT13-316: Developing and Deploying Large Systems

Assignment 2 – Hard Copy Submission

Name: Nicholas Webb **SID:** 12946743

```

import java.util.*;

/**
 * HamiltonianSolver.
 *
 * A Hamiltonian cycle is a path from a node to itself, passing through each node in
 the graph exactly once.
 * This program makes use of worklist processing in order to produce all the
 hamiltonian cycles of a
 * graph.
 *
 * Additional rules from a standard worklist process are:
 * - Do not propagate paths longer than number of nodes.
 * - Do not add path to solution if will introduce cycle of length less than number of
 nodes.
 *
 * @author Nicholas Webb
 * @version 1, 7/11/08
 */
public class HamiltonianSolver {

    /*
     * The main method which takes in user input as arguments.
     * If the user has not specified an external graph source,
     * a default one will be provided.
     *
     * The method creates the worklist processing environment and displays both the
     * initial graph and all its hamiltonian cycles once complete.
     */
    public static void main(String [] args) {
        //Responsible for storage of the graph created.
        Graph g;

        if (args.length != 0)
            //Will create the graph based on an external source once constructor is
implemented.
            g = new Graph(args[0]);
        else
            //Otherwise produces the default graph.
            g = new Graph();

        //A constant which represents the full length a hamiltonian cycle should be.
        final int HLENGTH = g.size() + 1;

        //Creates a new worklist using this graph as input.
        WorkList work = new WorkList(g);

        //Contains the current node focused on.
        Node n = new Node("");

```

```

//Continues until the worklist becomes empty.
while (!work.isEmpty()) {

    //Takes the node off the worklist.
    n = (Node)work.remove();

    //Checks whether there are any incorrect paths to be removed
    //and if so, records them in queue for later removal.
    Queue removeq = checkRemove(n, HLENGTH);

    //Append node to each path in solution
    work.appendPaths(n, HLENGTH);

    //Removes any incorrect paths from the solution.
    while (!removeq.isEmpty())
        n.getSolution().remove(removeq.remove());

    //Propagate solution to all connected nodes.
    work.propogate(n);
}
//Prints out an initial display of the graph.
System.out.println("Graph:\nG = [V, E]\n" + g.toString() + "\n");
//Prints out all the hamiltonian cycles of that graph.
System.out.println("Hamiltonian Cycles:\n" + checkHamiltonian(n.getSolution(),
HLENGTH).toString() + "\n");
}

/*
 * The checkRemove method searches through the solution and keeps track of paths
 * to be removed. A queue is returned having stored within it all of these paths.
 */
public static Queue checkRemove(Node n, int hlength) {
    //An iterator for the traversal over a nodes solution.
    Iterator it = n.iterator();
    //Queue that will keep track of paths to be removed.
    Queue removeq = new LinkedList();

    //Searches through each of the paths.
    while (it.hasNext()) {
        Path p = (Path)it.next();

        //If a path does not meet the requirements of a hamiltonian cycle, place it onto
the queue
        //for removal.
        if (!p.isEmpty() && ((p.getLast().equals(n.toString()) && p.size() != hlength)
|| (p.contains(n.toString()) && p.size() < hlength - 1) ||
(!p.getFirst().equals(p.getLast()) && p.size() == hlength)))
            removeq.add(p);
    }
}

```

```

    //Returns the queue of all paths which need to be removed.
    return removeq;
}

/*
 * The checkHamiltonian method makes sure that the final solution does not
contain any paths
 * that fall short of the length requirement of a hamiltonian cycle. That is, visiting
every
 * node exactly once and returning to itself.
 */
public static Set checkHamiltonian(Set h, int hlength) {
    //Create an iterator to traverse over the solution.
    Iterator it = h.iterator();
    //Create the set in which to store correct hamiltonian cycles.
    Set hamiltonian = new HashSet();

    //Checks that each path satisfies a hamiltonian cycle.
    while (it.hasNext()) {
        Path temp = (Path)it.next();
        if (!(temp.size() < hlength))
            hamiltonian.add(temp);
    }

    //Returns the set of all correct hamiltonian cycles.
    return hamiltonian;
}
}

```

```

import java.util.*;

/**
 * Graph.
 *
 * A class that represents a mathematical graph.
 *
 * The Graph is defined by a set of vertices (or nodes)
 * and a set of edges (or arcs) between vertices.
 *
 * @author Nicholas Webb
 * @version 1, 7/11/08
 */
public class Graph {
    //A set representing all the nodes in the graph.
    private Set nodes = new HashSet();
    //A set representing all the edges in the graph.
    private Set edges = new HashSet();

    //Constructor for the default graph if no external one
    //is provided.
    public Graph() {
        Node a = new Node("A");
        Node b = new Node("B");
        Node c = new Node("C");
        Node d = new Node("D");

        nodes.add(a);
        nodes.add(b);
        nodes.add(c);
        nodes.add(d);

        edges.add(new Edge(a, b));
        edges.add(new Edge(a, c));
        edges.add(new Edge(a, d));
        edges.add(new Edge(b, a));
        edges.add(new Edge(b, c));
        edges.add(new Edge(b, d));
        edges.add(new Edge(c, a));
        edges.add(new Edge(c, b));
        edges.add(new Edge(c, d));
        edges.add(new Edge(d, a));
        edges.add(new Edge(d, c));
        edges.add(new Edge(d, b));
    }

    //Constructor for a graph where an external source is provided.
    public Graph (String filename) {
        //In here the component to read a graph from an external source
        //can be easily added when the feature becomes available.
    }
}

```

```
}

//Returns an iterator to traverse over all the nodes in the graph.
public Iterator nodesIterator() {
    return nodes.iterator();
}

//Returns an iterator to traverse over all the edges in the graph.
public Iterator edgesIterator() {
    return edges.iterator();
}

//Calculates how large the graph is; the number of nodes it contains.
public int size() {
    return nodes.size();
}

//Produces a String representation of the graph with all of its nodes and edges.
public String toString() {
    return ("V = " + nodes.toString() + "\nE = " + edges.toString());
}
}
```

```

import java.util.*;

/**
 * Worklist.
 *
 * An implementation of a worklist, represented by a queue of nodes.
 *
 * Provides methods for actions used in worklist processing, such as:
 *
 * - Initialising each node information to {} and putting on queue. (Worklist
Constructor)
 * - Removal of first node. (remove() method)
 * - Appending the node to its own solution (appendPaths() method)
 * - Propagation to each connected node. (propagate() method)
 * - If connected node solution changed, add connected node to worklist. (Worklist as
Observer, Nodes as Observable)
 *
 * @author Nicholas Webb
 * @version 1, 7/11/08
 */
public class WorkList implements Observer {
    //Creates a queue which will represent the data structure of the worklist.
    private Queue work = new LinkedList();
    //Stores the graph currently worked with.
    private Graph graph;

    //Constructor for initialising each of the nodes and placement onto the queue.
    public WorkList(Graph g) {
        //Store the given graph for use by the worklist.
        graph = g;
        //An iterator to traverse over all the nodes in the graph.
        Iterator nodeIt = graph.nodesIterator();

        while (nodeIt.hasNext()) {
            Node n = (Node)nodeIt.next();
            //Addition of the worklist as an observer to each node in the graph.
            n.addObserver(this);
            work.add(n);
        }
    }

    //The worklist will add the connected node if its solution has changed.
    public void update(Observable obj, Object arg) {
        work.add(obj);
    }

    //Append node to each path in solution
    public void appendPaths(Node n, int size) {
        n.updateSolution(n.toString(), size);
    }
}

```

```

//Propagate solution to all connected nodes
public void propogate(Node n) {
    Iterator it = graph.edgesIterator();

    while (it.hasNext()) {
        Edge temp = (Edge)it.next();

        if (n.equals(temp.getSource()))
            temp.getDestination().addSolution(temp.getSource().getSolution());
        }
    }

//Removal of the first element off the worklist
public Object remove() {
    return work.remove();
}

//Checks whether the list is empty.
public boolean isEmpty() {
    return work.isEmpty();
}

//Produces a String representation of the worklist.
public String toString() {
    return work.toString();
}
}

```

```
/**
 * Edge.
 *
 * A class which represents an edge which connects two nodes together.
 *
 * The direction of an edge is signified by the join of the source to
 * the destination node, however not the other way.
 *
 * @author Nicholas Webb
 * @version 1, 7/11/08
 */
public class Edge {
    //Nodes which represent the source and the destination of an edge.
    private Node source, destination;

    //Constructor to join the two nodes together.
    public Edge(Node s, Node d) {
        source = s;
        destination = d;
    }

    //Returns the source node of the edge.
    public Node getSource() {
        return source;
    }

    //Returns the destination node of the edge.
    public Node getDestination() {
        return destination;
    }

    //Produces a String representation of the edge from source to destination.
    public String toString() {
        return "(" + source.toString() + ", " + destination.toString() + ")";
    }
}
```

```

import java.util.*;

/**
 * Node.
 *
 * A class which represents a node containing a solution made up of paths.
 *
 * Provides methods for maintenance of this node.
 *
 * @author Nicholas Webb
 * @version 7/11/08
 */
public class Node extends Observable {
    //Represents the name of the node.
    private String name;
    //A set containing all of the paths within the solution.
    private Set solution = new HashSet();

    //A node constructor giving it a name and initialising it with a blank path.
    public Node(String n) {
        name = n;
        solution.add(new Path());
    }

    //Update a solution by appending the node to each of its paths in the solution.
    public void updateSolution(String s, int size) {
        //An iterator to traverse over all the paths in a solution.
        Iterator it = solution.iterator();

        //Updates each path in the solution with the node name.
        while (it.hasNext())
            ((Path)it.next()).updatePath(s, size);
    }

    //Propagate the paths from another solution to this one.
    public void addSolution(Set sol) {
        //An iterator to traverse over all the paths of the given solution.
        Iterator it = sol.iterator();
        //Keeps track whether a change has occurred, as we only need to signify
        //the solution change once.
        boolean changed = false;

        while (it.hasNext()) {
            Path temp = (Path)it.next();

            if (checkPath(temp)) {
                solution.add(new Path(temp.getPath()));
                if (!changed) {
                    this.setChanged();
                    this.notifyObservers();
                }
            }
        }
    }
}

```

```

        changed = true;
    }
}
}
}

//Checks whether a path is able to be added to the solution.
public boolean checkPath(Path p) {
//An iterator to traverse over the solution.
Iterator it = solution.iterator();

    while (it.hasNext()) {
        Path temp = (Path)it.next();
        if(temp.equals(p))
            return false;
    }
    return true;

}

//Returns the current state of the solution.
public Set getSolution() {
    return solution;
}

//Produces a String representation of the node.
public String toString() {
    return name;
}

//Checks whether one node is equal to another.
public boolean equals(Node n) {
    return name.equals(n.toString());
}

//Provides an iterator to traverse over the solution.
public Iterator iterator() {
    return solution.iterator();
}
}

```

```

import java.util.*;

/**
 * Path.
 *
 * A class which represents a path in a graph.
 *
 * Provides the interface for interacting with paths.
 *
 * @author Nicholas Webb
 * @version 7/11/08
 */
public class Path {
    //A linkedlist structure representing the path.
    private LinkedList path;

    //A path constructor creating a blank path.
    public Path() {
        path = new LinkedList();
    }

    //A path constructor for filling additional paths from another node.
    public Path(LinkedList p) {
        path = new LinkedList();
        Iterator it = p.iterator();

        while (it.hasNext())
            path.add(it.next());
    }

    //Returns the path in its current state.
    public LinkedList getPath() { return path; }

    //Updating a path by adding a new value to the end of it.
    //Does not insert if a size parameter is present, -1 signifies
    //no restriction on size.
    public void updatePath(String p, int size) {
        if (path.size() < size || size == -1)
            path.add(p);
    }

    //Returns an iterator for traversal over a path.
    public Iterator iterator() {
        return path.iterator();
    }

    //Produces a String representation of the path.
    public String toString() {
        return path.toString();
    }
}

```

```
//Checks whether one path is equal to another.
public boolean equals(Object p) {
    return path.equals(((Path)p).getPath());
}

//Returns the current size of the path.
public int size() {
    return path.size();
}

//Checks whether the path is empty.
public boolean isEmpty() {
    return path.isEmpty();
}

//Get the last element in the path.
public String getLast() {
    return path.getLast().toString();
}

//Get the first element in the path.
public String getFirst() {
    return path.getFirst().toString();
}

//Checks whether the path contains an element.
public boolean contains(Object o) {
    return path.contains(o);
}
}
```