CS 312: Project 3 Improvement
Duane Johnson

For this improvement, I modified the original "graph" class and created a new "Graph" class that uses several of the more recently introduced language features of C# 3.5.

For example, this new Graph class uses the HashSet template for Edge objects and a List template for Node objects. This compact type declaration permits the use of iterating and indexing without the use of cumbersome type overrides. To be specific, the new templating system allows for this:

```
Node n = nodes[0];
```

Instead of this:

```
Node n = (Node)nodes[0];
```

Also as part of this improvement, I wrote an iterator class that uses C#'s continuations (see "depthFirstIterator" in the code on page 3). While this iterator would be insufficient as a replacement to the one-pass "findArticulationPoints" method, it does provide a simple and fast way to traverse the entire graph (the previsit and postvisit methods cannot be handled well by a simple iterator). Here is an example of a call to the iterator:

```
// Show the articulation points in color:
 foreach (Node n in graph.depthFirstIterator)
 {
    if (n.separating)
    {
       visualizer.UpdateVertexColor(n.vertex.index, Color.Yellow);
    }
    else
    {
       visualizer.UpdateVertexColor(n.vertex.index, Color.Gray);
    }
 }
```

See the following page for the "Graph" class listing.

```
class Graph
 {
    public List<Node> nodes;
    public HashSet<Edge> edges;
    public Visualizer vis;

    public Graph(Visualizer visualizer)
    {
       Console.WriteLine("Create graph");

       vis = visualizer;

       nodes = new List<Node>();
       edges = new HashSet<Edge>();

       int i = 0;
       foreach (Vertex v in visualizer.Vertices)
       {
          Node n = new Node(v);
          nodes.Add(n);
          v.index = i++;
       }

       Console.WriteLine("Indexed nodes...");

       foreach (Line l in visualizer.Lines)
       {
          Edge e1 = new Edge(nodes[l.V2.index], l);
          Edge e2 = new Edge(nodes[l.V1.index], l);
          nodes[l.V1.index].edges.Add(e1);
          nodes[l.V2.index].edges.Add(e2);
          edges.Add(e1);
          edges.Add(e2);
       }
       Console.WriteLine("Done creating graph");
    }

    public void assignLows()
    {
       HashSet<Node> considered = new HashSet<Node>();
       Stack<HashSet<Node>> next = new Stack<HashSet<Node>>();
       Stack<Node> stack = new Stack<Node>();

       // ... snip ...
    }

    public List<BiconnectedComponent> findBiconnectedComponents()
    {
       List<BiconnectedComponent> components = new List<BiconnectedComponent>();
       HashSet<Node> visited = new HashSet<Node>();
       Stack<Node> nodeStack = new Stack<Node>();
       Stack<Line> edgeStack = new Stack<Line>();

       // ... snip ...

       return components;
    }
```

```
    public IEnumerable<Node> depthFirstIterator
    {
        get
        {
            HashSet<Node> considered = new HashSet<Node>();
            Stack<Node> stack = new Stack<Node>();

            stack.Push(nodes[0]);
            considered.Add(nodes[0]);

            int visit = 0;
            while (stack.Count > 0)
            {
                Node current = stack.Pop();
                foreach (Node n in current.neighbors)
                    if (!considered.Contains(n))
                    {
                        n.parent = current;
                        stack.Push(n);
                        considered.Add(n);
                    }

                yield return current;
            }
        }
    }

}
```