Algoritmiek, Graph Exercise – Example Solution

Tom Schenkels, Delft University of Technology

March 11, 2011

Task A

Since hyperlinks point in one direction they will be represented by directed edges. So we use a directed graph wherein the nodes represent individual webpages. We note that pages may refer to themselves, which is represented by an edge going out of one and entering another node. In real life, multiple hyperlinks can point from one page to another. That could be modeled by weighted edges, but I think that exceeds the scope of this assignment, so I won't implement that. I will represent the directed graph as a double adjacency list as explained in the book (pg. 97).

The number of nodes will always be denoted by n. The upper bound on the number of edges m will arise when all nodes point to all other nodes, are pointed to by all other nodes and also point to themself. The maximum number of edges in an undirected graph is $\binom{n}{2}$ (pg. 87). So when edges are directed and you add the fact that nodes may have an edge pointing to themself, you get $2\binom{n}{2} + n = n^2$.

Task B

All other nodes in the graph can be reached from node p if a directed exploration algorithm starting from p reaches all nodes in the graph. I will use BFS, but DFS would complete the job just as well.

Algorithm 1 BFS from p for directed graphs

```
Set Discovered[p] = true and Discovered[v] = false for all other v
Initialize L[0] to consist of the single element s
Set the layer counter i = 0
while L[i] is not empty do
  Initialize an empty list L[i + 1]
  for each node u in L[i] do
    Consider each outgoing edge (u, v)
    if Discovered[v] = false then
      Set Discovered[v] = true
      Add v to the list L[i + 1]
    end if
  end for
  Increment the layer counter i by one
end while
```

I will use a BFS algorithm similar to the one on page 90. The only difference is in line 7 in Algorithm 1. In the book, the line is "Consider each edge (u, v) incident to node u". I changed it so the algorithm looks only at the outgoing edges of node u. Also, for the task at hand it is

not necessary to construct the BFS tree. If you leave the build up layers intact you have all the information that you need. To complete Task B algorithm 2 is needed.

Algorithm 2 Task B

```
BFS(p)
if Number of nodes discovered equals n then
  return 1
else
  return 0
end if
```

When the search is completed we compare the number of discovered nodes with n. If it is equal, all nodes were reached, so 1 is returned. Otherwise not all nodes could be reached, so 0 is returned.

At most, n^2 edges get looked at in the BFS. For each of them constant time is required. Looking up if all nodes were discovered takes $O(n)$ time. Thus, the tightest upper bound on the running time is $O(m + n)$.

Task C

Algorithm 3 Task C

```
for All nodes do
  count[w] ← amount of incoming edges of node w
end for
s = set of remaining nodes with no incoming edges
while s is not empty do
  remove node v from s
  decrease count[w] for all edges from v to w
  if count[w] hits 0 then
    add w to s
  end if
end while
if count[v] is lower or equal to 0 for all nodes v in graph then
  return 0
else
  return 1
end if
```

I will implement the algorithm for topological ordering which can be found on page 26 of the slides on "DAGs_and_Topological_Ordering". It works by removing nodes which have no incoming edges, thereby decreasing the number of incoming edges for other nodes. If, by following this procedure, all nodes can be removed, the graph has no cycles. Algorithm 3 shows the pseudocode for Task C.

Counting the number of incoming edges for all nodes takes $O(n+m)$ time. The task of removing nodes without incoming edges takes $O(n + m)$ time too. At last, checking if all nodes could be removed takes $O(n)$ time. Also, the running time of the total algorithm is $O(n + m)$.

Why is it correct? If a graph contains a cycle, none of the nodes in the cycle can ever be removed since they all have at least one incoming edge. If this is the case, the algorithm will stop and return 1. On the other hand, if the graph was a DAG, all nodes are removed and the algorithm returns 0.

Task D

In order to find the strongest component, all strong components should be found. Algorithm 4 uses the fact that strong components in graphs are always disjoint, as mentioned on pg. 99 of the book.

Algorithm 4 Task D

```
sizeOfStrongestComponent ← 0
while discovered[v] false for a node v do
  Do BFS(v) on graph g and its reverse g^REV
  discovered[w] ← true if node w was in found strong component
  if size of found strong component > sizeOfStrongestComponent then
    sizeOfStrongestComponent ← size of found component
  end if
end while
return sizeOfStrongestComponent
```

Algorithm 4 discovers all strong components and returns the size of the biggest one. It makes sure that all components are looked at by making sure that all nodes are in a strong component that it has looked at.

Running a BFS takes $O(m + n)$ time. Since all strong components could consist of a single node, $2 * n$ BFS searches need to be done at most. So the total running time of the algorithm is upper bounded by $O(n(m + n))$.

Task E

Algorithm 5 Task E

```
define distance, a 2D array to contain distances between nodes, initialize as 0 for all edges
for all nodes u do
  BFS(u)
  for each layer i do
    for each v ∈ L[i] do
      distance[u, v] ← i
    end for
  end for
end for
return biggest value in distance array
```

Algorithm 5 performs a BFS from every node in the graph. Distances are read from the BFS layers. If node v shows up in L_3 produced by a BFS from node u, the shortest distance from u to v is 3. But not the other way round, that distance is found when BFS is done from v. If a node can't be reached, the distance stays 0. In the end, the biggest distance is returned, the diameter.

Initializing the array takes $O(n^2)$ time. Performing a BFS for every node $O(n * (n + m))$, and finding the biggest value in the array $O(n^2)$. So the total time can be bound by $O(n * (n + m))$.

```
// Auteur:      Tom Schenkels
// Datum:       17 feb 2010


public class Funda2
{

    public static void main(String[] args)
    {
        String f = args[0];

        Graph g = new Graph(f);
        Graph gRev = new Graph(f).reverseGraph();

        // If no nodes were read, print correct output since the answers of
the output set are questionable
        if(g.n == 0)
        {
            System.out.println("1\n0\n1\n0\n");
            System.exit(0);
        }

        // Task B
        // execute bfs, save discovered nodes
        boolean[] discoveredNodes = (boolean[])g.bfs(g.startingPoint).get(0);
        int nDiscoveredNodes = 0;
        for(int i=0; i<g.n; i++)
            if(discoveredNodes[i])
                nDiscoveredNodes++;
        int res = 0;
        if(nDiscoveredNodes == g.n)
            res = 1;
        System.out.println(res);

        // Task C
        System.out.println(g.hasCycle());

        // Task D
        System.out.println(g.sizeOfStrongestComponent(g, gRev));

        // Task E
        System.out.println(g.diameter());

    }
}
```

```
// Auteur:      Tom Schenkels
// Datum:       17 feb 2010

import java.io.*;
import java.util.*;

public class Graph
{
    public String name;
    public Integer startingPoint;
    public int n;

    public ArrayList<String> nodes = new ArrayList<String>();
    public HashMap<Integer, ArrayList<Integer>> leaving = new
HashMap<Integer, ArrayList<Integer>>();
    public HashMap<Integer, ArrayList<Integer>> entering = new
HashMap<Integer, ArrayList<Integer>>();

    // Reads graph in from file file
    public Graph(String file){
        Scanner sc;
        try
        {
            sc = new Scanner(new File(file));
        }
        catch (FileNotFoundException e)
        {
            // Auto-generated catch block
            e.printStackTrace();
            sc = null;
        }

        sc.next(); // read 'digraph'
        name = sc.next(); // read name
        sc.next(); // read '{'

        String sNext = sc.next();
        while(!sNext.equals("}"))
        {
            // read from and to node
            String from = sNext;
            sc.next(); // read '->'
            String to = sc.next().replace(";", "");

            // check if we have read them already, if not add them to the
```



```
adjacency lists
            if (!nodes.contains(from))
            {
                nodes.add(from);
                Integer iFrom = nodes.indexOf(from);
                leaving.put(iFrom, new ArrayList<Integer>());
                entering.put(iFrom, new ArrayList<Integer>());
            }
            Integer iFrom = nodes.indexOf(from);

            if (!nodes.contains(to))
            {
                nodes.add(to);
                leaving.put(nodes.indexOf(to), new ArrayList<Integer>());
                entering.put(nodes.indexOf(to), new ArrayList<Integer>());
            }
            Integer iTo = nodes.indexOf(to);

            // add edges to adjacency lists
            leaving.get(iFrom).add(iTo);
            entering.get(iTo).add(iFrom);

            sNext = sc.next();

        }

        String sStartingPoint = sc.next();
        if (!nodes.contains(sStartingPoint))
        {
            nodes.add(sStartingPoint);
            Integer iStartingPoint = nodes.indexOf(sStartingPoint);
            leaving.put(iStartingPoint, new ArrayList<Integer>());
            entering.put(iStartingPoint, new ArrayList<Integer>());
        }
        startingPoint = nodes.indexOf(sStartingPoint);

        // save total amount of nodes
        n = nodes.size();
    }

    // Does an outward BFS starting from iStartingPoint
    // Returns ArrayList with discovered[boolean] and found layers l
    public ArrayList bfs(Integer iStartingPoint)
    {
        // Initialize discovered. True for starting point, false for others.
```

```
        boolean[] discovered = new boolean[this.n];
        discovered[iStartingPoint.intValue()] = true;

        // Initialize L0 to consist of one point only; the starting point
        ArrayList<ArrayList<Integer>> l = new
ArrayList<ArrayList<Integer>>();
        ArrayList<Integer> l0 = new ArrayList<Integer>();
        l0.add(iStartingPoint);
        l.add(l0);

        // Set layer counter i = 0
        int bfsLayer = 0;

        // While current list is not empty
        while(!l.get(bfsLayer).isEmpty()){


            // Initialize an empty list L[i+1]
            l.add(new ArrayList<Integer>());

            ArrayList<Integer> currentL = l.get(bfsLayer);
            ArrayList<Integer> nextL = l.get(bfsLayer + 1);

            // For each node u in L[i] (currentL)
            for(int nodeNumber = 0; nodeNumber<currentL.size();
nodeNumber++){

                // Consider each edge (u,v) leaving/entering u
                // for all leaving edges
                for(int edgeNumber = 0; edgeNumber <
this.leaving.get(currentL.get(nodeNumber)).size(); edgeNumber++)
                {
                    Integer toNode =
this.leaving.get(currentL.get(nodeNumber)).get(edgeNumber);

                    if(discovered[toNode] == false){
                        discovered[toNode] = true;
                        nextL.add(toNode);
                    }
                }
            }
            bfsLayer++;
        }

        // We will return an ArrayList with discovered at index 0 and the
```



```
layers at index 1
        ArrayList res = new ArrayList();
        res.add(discovered);
        res.add(l);
        return res;
    }

    // Returns 1 if graph contains cycle
    public Integer hasCycle()
    {
        // count[w] = remaining number of incoming edges in w
        Integer[] count = new Integer[this.n];
        // s = set of remaining nodes with no incoming edges
        ArrayList<Integer> s = new ArrayList<Integer>();

        // Initialization:  O(m + n) via single scan through graph.
        for(int node = 0; node < this.n; node++)
        {
            // Save number of incoming edges
            ArrayList<Integer> incomingEdges = this.entering.get(node);
            count[node] = incomingEdges.size();
            // If 0, add to s
            if(count[node] == 0)
                s.add(node);
        }

        // While there is a node with no incoming edges
        while(!s.isEmpty())
        {
            // Update:  to delete v
            Integer v = s.remove(0);

            // decrement count[w] for all edges from v to w, and add w to S
if count[w] hits 0
            for(int w = 0; w < leaving.get(v).size(); w++)
            {
                Integer nodeW = leaving.get(v).get(w);
                if(--count[nodeW] == 0)
                    s.add(nodeW);
            }
        }

        // Count number of remaining nodes
        int nRemaining = 0;
        for(int node = 0; node < this.n; node++)
```

```
        {
            if(count[node] > 0)
                nRemaining++;
        }

        if(nRemaining == 0)
            // All nodes could be removed. So it is a DAG, so no cycles, so
return 0.
            return 0;
        else
            // Not all nodes could be removed. So it has cycles.
            return 1;
    }

    // Returns size of Strongest Component
    public int sizeOfStrongestComponent(Graph g, Graph gRev)
    {
        // Remember which nodes were in a strong component
        boolean[] inStrongComponent = new boolean[n];
        // Remember how big the strongest component was
        int sizeOfStrongestComponent = 0;

        while(hasFalse(inStrongComponent))
        {
            Integer currentNode = findFirstFalse(inStrongComponent);

            // Initiate bfs on g from currentNode
            boolean[] outward = (boolean[])g.bfs(currentNode).get(0);
            // Initiate bfs on g^rev from currentNode
            boolean[] inward = (boolean[])gRev.bfs(currentNode).get(0);

            // Calculate size of strong component
            int sizeOfStrongComponent = 0;
            for(int node = 0; node < n; node++)
            {
                // if node was in strong component
                if(outward[node] && inward[node])
                {
                    sizeOfStrongComponent++;
                    inStrongComponent[node] = true;
                }
            }

            // Save new max size if it is bigger than what we've found before
            if(sizeOfStrongComponent > sizeOfStrongestComponent)
```



```
                sizeOfStrongestComponent = sizeOfStrongComponent;
        }

        return sizeOfStrongestComponent;
    }

    public int diameter()
    {
        // Initialize 2D array to hold distances from and to nodes
        int[][] distanceFromTo = new int[n][n];

        // for each node u
        for(int u = 0; u < n; u++)
        {
            // Initiate bfs and save layers
            ArrayList<ArrayList<Integer>> l = (ArrayList<ArrayList<Integer>>)
bfs(u).get(1);
            // for each layer
            for(int bfsLayer = 0; bfsLayer < l.size(); bfsLayer++)
            {
                //for each node v in layer
                for(int v = 0; v < l.get(bfsLayer).size(); v++)
                {
                    // save distince from u to v in array
                    distanceFromTo[u][v] = bfsLayer;
                }
            }

        }

        // Find biggest distance in array
        int maxDistance = 0;
        for(int x=0; x<n; x++)
            for(int y=0; y<n; y++)
                if(distanceFromTo[x][y]>maxDistance)
                    maxDistance = distanceFromTo[x][y];

        // And return it
        return maxDistance;
    }

    public Graph reverseGraph()
    {
        HashMap<Integer, ArrayList<Integer>> tmp = this.entering;
        this.entering = this.leaving;
```

```
        this.leaving = tmp;

        return this;
    }
    private boolean hasFalse(boolean[] inStrongComponent)
    {
        return findFirstFalse(inStrongComponent) != -1;
    }

    private Integer findFirstFalse(boolean[] inStrongComponent)
    {
        for(int i = 0; i<n; i++)
            if(inStrongComponent[i] == false)
                return i;
        return -1;
    }

}
```