Universitetet i Oslo
Institutt for Informatikk

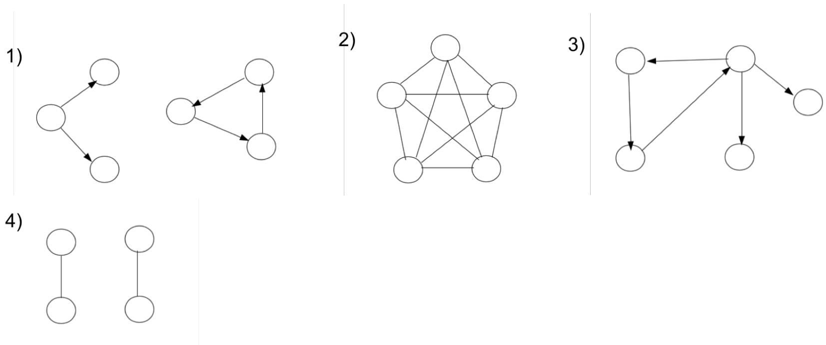
I. Yu, D. Karabeg

INF2220: algorithms and data structures
Series 5

Topic Graphs

Issued: 21. 09. 2016

Classroom



Exercise 1 (Graphs)

1. For each of the graphs 1-4 determine whether the graph is
   (a) Connected?
   (b) Directed?
   (c) Cyclic?
2. What kind of graph is most suitable to represent ...
   (a) Friends at Facebook?
   (b) Flights?
   (c) Subjects you can choose/attend at the University?

Exercise 2 (Terminology, topological sorting)

1. Draw the following directed graph $G = (V, E)$, i.e., give a pictorial/"graphical" representation of G:

$$\begin{aligned} V &= \{1,2,3,4,5,6,7\} \\ E &= \{(1,2),(1,4),(1,3),(4,3),(4,6),(4,7),(3,6),(5,4),(5,7),(2,4),(7,6)\} \end{aligned}$$

2. Is G *strongly connected*? If not, is it *weakly connected*?
3. Write down the *adjacency matrix* and *adjacency list* for the graph.
4. What are the indegree and outdegree for each node in the graph?
5. Does there exists a legal topological ordering/topological sorting of the elements? If so, give such an ordering? If not: why does it not exist?

Exercise 3 (Topological sorting) Draw each of the following graphs. Determine if there exists a topological sorting for each of them; if yes, find it, if no, explain why not.

1. {(A,B),(A,F),(B,D),(E,C),(B,E),(F,E)}
2. {(A,C),(B,E),(C,E),(D,B)}
3. {(A,B),(A,F),(B,C),(C,D),(D,A),(F,C),(F,E),(E,D)}

Exercise 4 (Hamiltonian Path (exam 2013)) A *Hamiltonian path* is a path in a graph that visits each node in the graph **once**. Let graph $G = (V, E)$ be a *directed acyclic graph* (DAG).

Write a method `HamiltonianPath` which, given a graph G as input, returns *true* if G contains a Hamiltonian path and returns *false* otherwise. Your algorithm should have *linear* time complexity. Show that your algorithm satisfies this running time requirement.

Exercise 5 (Terminology, shortest paths)

1. Draw the following directed and weighted graph $G = (V, E)$ defined as follows:

$$\begin{array}{rcl} V & = & \{1,2,3,4,5,6\} \\ E & = & \{(1,5,1),(1,6,2),(2,1,7),(2,6,9),(3,1,2),(3,5,3),(4,1,3),(4,3,2),(4,5,4), \\ & & (5,2,6),(6,2,4),(6,4,4)\} \ . \end{array}$$

   As a weighted graph, the set of edges is not a subset of $V \times V$, but $E \subseteq V \times V \times \mathbb{N}$, i.e., it has a *weight* or *cost* as a third component (here as non-negative natural number). The cost is here the third element in the edge tuples (thus edge triples) above.

2. Is G *strongly connected*? If not, is it *weakly connected*?
3. Write down the *adjacency matrix* and *adjacency list* for the graph.
4. What are the indegree and outdegree for each node in the graph?
5. Give the *shortest paths* and the respective *costs* from node `1` to each of the other nodes in the graph.

Lab

Exercise 6 For the topological sorting algorithm, one needs a table of indegrees of nodes in a graph. Write a method that collects the indegrees of each node in a graph. Assume that you have an adjacency list `L[]` that represents the graph. Furthermore, assume that `ListNode` of an adjacency list contains fields

```
public int destination
public ListNode next
```

which one may use in the code. The method signature should be in this form:

```
public int[] calculateIndegrees(ListNode L[])
```

The method should return the table of indegrees.

Exercise 7 Implement the method `shortestPathFrom`, this method should return the nodes with their `distance` variable set according to their actual shortest distance from the input identifier.

```
import java.util.HashMap;
import java.util.LinkedList;

class DistanceGraph{

    HashMap<String, Node> graph;

    DistanceGraph(){
        graph = new HashMap<String, Node>();
    }

    void addVertex(String id){
        graph.put(id, new Node(id));
    }

    boolean addEdge(String from, String to){

        Node fromNode = graph.get(from);
        Node toNode   = graph.get(to);

        if(fromNode == null || toNode == null){
            System.err.printf(" %s : %s",
                              "DistanceGraph.addEdge",
                              "could not find both nodes\n");
            return false;
        }

        return fromNode.addEdge(toNode);
    }

    private void removeVisitedFlags(){
        for(Node n : graph.values()){
            n.visited = false;
        }
    }


    public LinkedList<Node> shortestPathFrom(String id){

        // first we have to clear all flags..
        removeVisitedFlags();

        // make a que and a done list
        LinkedList<Node> que = new LinkedList<Node>();
        LinkedList<Node> done = new LinkedList<Node>();

        // TODO

        return done;
    }


    // testing it with the graph from slide 27 last lecture

    public static void main(String[] args){
```

```
        DistanceGraph dg = new DistanceGraph();

        for(int i = 1; i < 8; i++){
            dg.addVertex(String.format("V%d",i));
        }

        dg.addEdge("V1", "V2");
        dg.addEdge("V1", "V4");
        dg.addEdge("V2", "V5");
        dg.addEdge("V2", "V4");
        dg.addEdge("V3", "V6");
        dg.addEdge("V3", "V1");
        dg.addEdge("V4", "V3");
        dg.addEdge("V4", "V6");
        dg.addEdge("V4", "V7");
        dg.addEdge("V4", "V5");
        dg.addEdge("V5", "V7");
        dg.addEdge("V7", "V6");


        for(Node n : dg.shortestPathFrom("V3")){
            System.out.println(n);
        }

        /* should produce something similar to this..
         Node id: V3, distance 0
         Node id: V6, distance 1
         Node id: V1, distance 1
         Node id: V2, distance 2
         Node id: V4, distance 2
         Node id: V5, distance 3
         Node id: V7, distance 3
         */

        System.out.println(" -------------------------");

        for(Node n : dg.shortestPathFrom("V4")){
            System.out.println(n);
        }

        /* should produce somthing like this
         Node id: V4, distance 0
         Node id: V3, distance 1
         Node id: V6, distance 1
         Node id: V7, distance 1
         Node id: V5, distance 1
         Node id: V1, distance 2
         Node id: V2, distance 3
         */
    }
}


class Node{

    String id;
    LinkedList<Node> neighbours;
    boolean visited;
    int distance;

    Node(String _id){
        id         = _id;
        neighbours = new LinkedList<Node>();
        visited    = false;
    }

    public boolean addEdge(Node to){
        return neighbours.add(to);
    }
```

```
    public LinkedList<Node> getNeighbours(){
        return neighbours;
    }

    public String toString(){
        return String.format(" Node id: %2s, distance %d ",id,distance);
    }
}
```