Universitetet i Oslo
Institutt for Informatikk
I. Yu, D. Karabeg

uiologo

# INF2220: algorithms and data structures
# Series 7

**Topic More on graphs: DFS, Biconnectivity, and strongly connected components (Exercises with hints for solution)**

**Issued: 05. Oct. 2016**

## Classroom

**Exercise 1 (Biconnectivity)** Given the graph of Figure 1,

1. Is the graph biconnected?
2. If not, which node(s) is (are) the articulation point(s) in the graph? Show the depth-first search spanning tree. Indicate *back edge(s)* if applicable. You should also indicate the value of *Low(v)* and *Num(v)* for each node *v* in the graph.

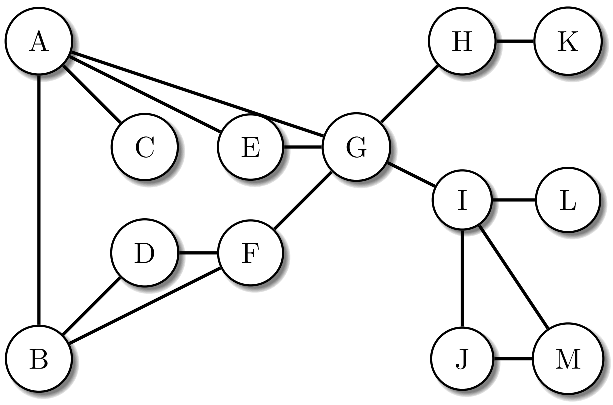


Figure 1: Connected, undirected Graph for Exercise 1

**Solution:** [of Exercise 1]

1. No.
2. Articulation point: A, H G, I. The dfs spanning tree is shown in Figure 2. The edges of the DFS spanning tree are shown as solid arrows. Of course, the shown tree corresponds to one particular run of a dfs, starting at *A*; there are different

runs, which also lead to different pre-order numbering (the "visiting times") and consequently to different *low*-numbers. Independent from the particular run, the articulation points, however, are always the same.

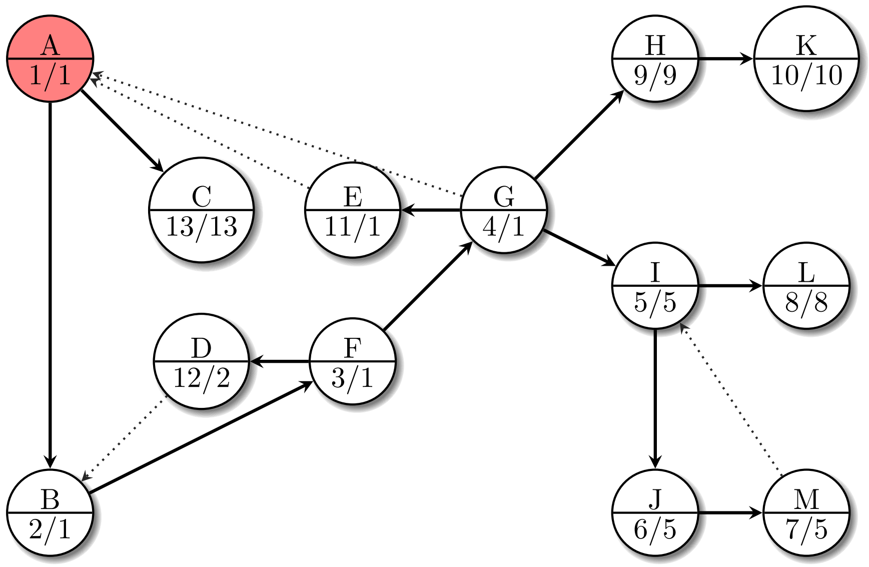


Figure 2: Exercise 1 with *Num(v)/Low(v)* (cf. also Fig. 1

**Exercise 2 (Strongly-connected components)** In the lecture, we learned to compute strongly-conneced components (scc's) for a directed graph.[1] Show how this is done for the graph in Figure3.

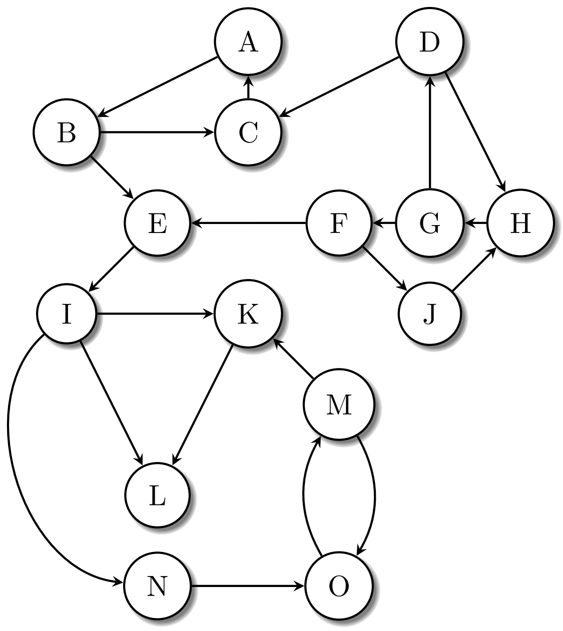


Figure 3: Directed graph for the scc exercise 2

[1]Works as well for undirected graphs, obviously, but the problem there is boring.

**Solution:** This rather smart algo comes from Kosaraju. The first one how solved it was Tarjan, but the way it's done here goes was invented by Kosaraju. Both ways of solving the problem share the main underlying idea: use DFS and the finishing time of the first algorithm to make a second phase. Tarjan's solution does it slightly different than the one presented here. This one is typically considered to be a more "clearly understandable" way of determining SCCs.

The result of one first (iterated) dfs-run is shown in Figure 4.

For the problem of the SCC, the information of the visiting time is not actually needed; the second phase —see below— uses only the finishing time, i.e. the post-order numbering. However, the numbers may give further insight of the nature of dfs.

As far as the pre-order and post-order numbers (visiting, finished), one can make a number of observation. The way that we are numbering, each number is given exactly once, that means, since the example has 15 nodes (and since we are starting by 1) the last finishing time is 30. A further easy observation is that, the way we are numbering, the pair of visiting/finished number consists always of a even and an odd number, and for the roots of the dfs-trees, the visiting number is always uneven.

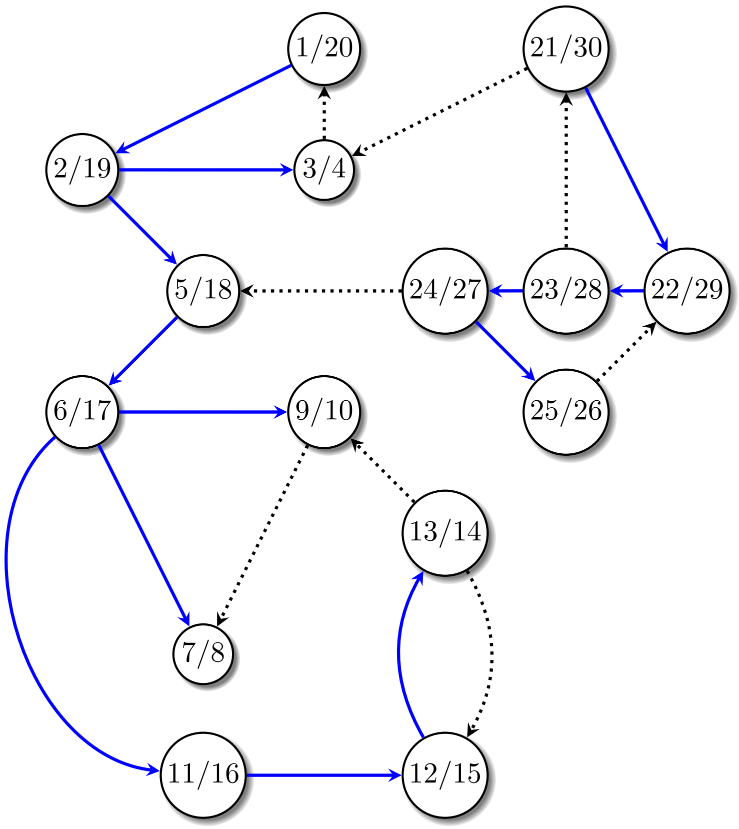


Figure 4: Result of one first dfs run for Exercise 2 (SCC)

After the first run, the graph is reversed/transposed, i.e., the direction of the arrows is reversed. Kosaraju's trick now is not just to run a second (iterated) dfs on the reversed graph; as we know, an iterated dfs is non-deterministic in particular wrt. to to picking an unvisited node as starting node for one recursion, i.e., as root for the dfs-tree.[2] To detect scc's in the second run, the starting points of the iteration, i.e., the roots of the trees of

[2]The other form of non-determinism, inherent in a iterated dfs-run, namely, in which order adjacent nodes are treated, is irrelevant here.

the dfs forest of the second phase,

> must be taken in *decreasing* order as far as the finishing times of the first run is concerned!

The corresponding root nodes, shown in blue in the figure, are $D, A, E, N, O, K, L$, *in this order.* As usual, the dfs-forest edges are shown solid, the other edges are dotted. In the picture we also make a distinction between two different types of edges: *backward* edges (blue) and *cross edges* (orange). Note that there are *no* forward edges. Note further, that the backward edges connect nodes within the same scc. In contrast, the cross edges bridge here difference scc's. Remember that according to the definition, cross edges are edges of the underlying graph which are non-tree/forest edges and which do *not* connect ancestor-descendants of one tree. So in general, a cross edge can connect 1) two nodes within the same dfs tree, or 2) two nodes of two different dfs trees. Here, however, cross-edges of the type 1) do *not* occur in the second phase of the algo, and neither do *forward* edges. It should be noted that this is not a coincidence of our particular example or run, but a property of Kosaraju's trick of using the visiting time of the first phase to order the iteration in the second phase. That may be "verified" ("verified" in a very lax manner by way of example) when one does the iterated dfs of the second phase *ignoring* the post-order numbers. For instance, just starting with $L$ and exploring $I$ first will give such a situation.

That remark maybe compared to the following observation: in an *undirected* graph (where being weakly connected and being strongly just means the same, and is therefore just called " being connected") there are backward edges only, and the cross-edges are of type 2) only.

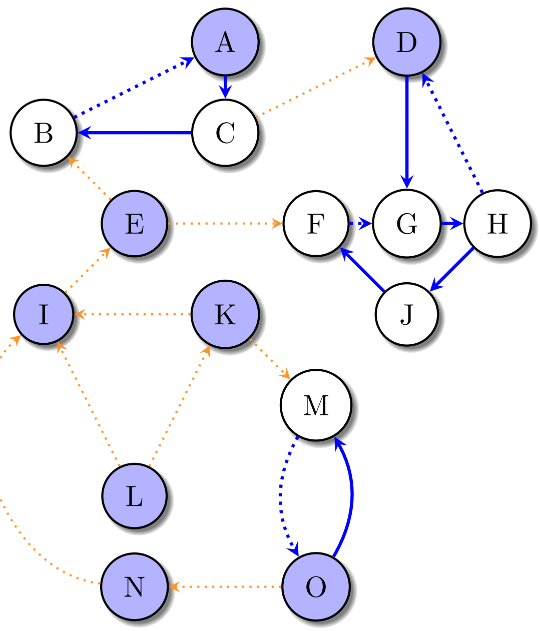
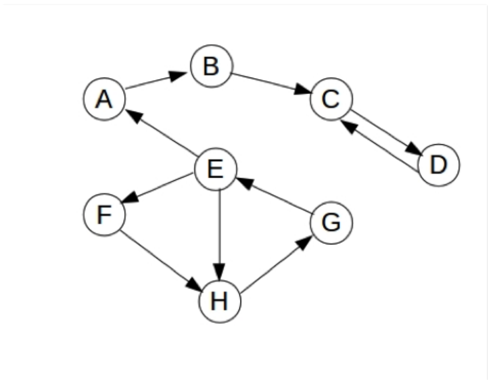
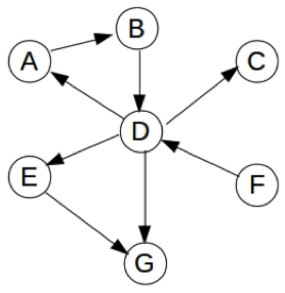


Figure 5: 2nd phase for Exercise 2 (SCC)

**Exercise 3** Find the stronly connected components of the following graphs. Indicate the steps in the algorithm.





**Solution:**

1. $\{C, D\}, \{B\}, \{E, G, H, F\}, \{A\}$
2. $\{A, B, D\}, \{C\}, \{E\}, \{F\}, \{G\}$

**Exercise 4 (Christmas party-2015 exam)** In this problem you will plan the seating arrangement on a Christmas party. You have a list $V$ of guests of type Person:

```
class Person{
 int id;

  ...

}
```

*4.1 Seating arrangement*

Assume that you are also given a lookup table $T$ where $T[u.id]$ for $u \in V$ gives a list of the guests (of type Person) that $u$ knows. If $u$ knows $v$ then $v$ also knows $u$. Your task is to make a seating arrangement so that each guest at a table knows all the others sitting at the same table either directly, or through other guests sitting at that table. For example, if $x$ knows $y$ and $y$ knows $z$, then $x$, $y$ og $z$ can sit at the same table.

1. If you should represent this as a graph problem, what kind of graph will that be? And what will be represented by the vertices and the edges in that graph?
2. Implement an efficient graph algorithm that, given $V$ and $T$ as input, returns the smallest number of tables that are necessary to satisfy that requirement.
3. What is the running time of your algorithm? Justify briefly.

*4.2 Enemies*

Assume that there are only 2 tables, and that you are given another lookup table $S$ where $S[u.id]$ for $u \in V$ gives a list of guests that have a bad relationship with $u$. If $v$ is in a bad relationship with $u$, then $u$ is also in a bad relationship with $v$. Your task is to make a seating arrangement so that no guests sitting at the same table are in a bad relationship with each another. (In this problem we will not take into account whether the guests know each other.)

Figure 6 is showing two graphs where the guests are represented as vertices and an edge between two vertices means that the corresponding guests are in a bad relationship

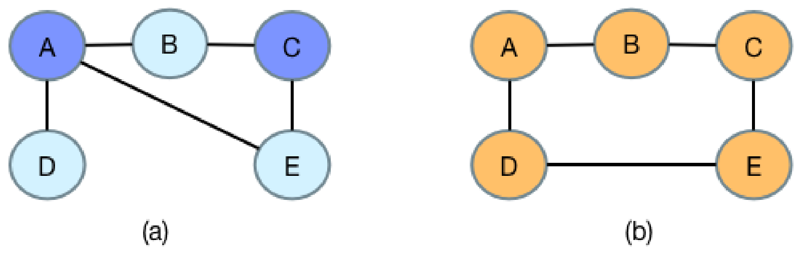


Figure 6:

with each other. For graph **(a)** we see that it is possible to have $A$ and $C$ share one table and $B$, $D$ and $E$ share another table, while for **(b)** we see that this is impossible.

Implement an efficient algorithm that given lists $V$ and $S$ as input returns *true* if we can place all the guests at two tables, and returns *false* otherwise.

**Solution:** of 4.1

1. An undirected graph. Guests as vertices and edges between two vertices means the two guests know each other. Table T represents the adjacency lists for the vertices.

2. If we start from one vertex s and search the graph using breadth-first search (BFS) or depth-first search (DFS), all the guests that are reachable from s can sit at the same table, and additional tables are needed for vertices that are unreachable from s . Hence, to find the minimum number of tables, we can iterate through s ∈ V . If s is not visited, increment the number of tables needed and call dfs-visit( s,T) or BFS(s, T) , marking vertices as visited during the traversal. Return the number of tables needed after iterating through all the vertices. This problem is equivalent to finding the number of connected components in the graph. Here is the pseudocode:

```
  int num−tables(V, T){
   int n = 0
   for s $\in$ V
     if (s.visited == false){
          n = n + 1
          s.visited = true
          dfs−visit(s, T)
     }
   return n
 }

void dfs−visit(u, T){
  for v $\in$ T[u.id]
     if (v.visited == false){
        v.visited = true
        dfs−visit (v, T)
     }
}
```

3. if using DFS as above the running time is O(V + E) because every vertex or edge is visted exactly once.

**Solution:** of 4.2 Note that like the previous task, the graph may not be connected.

```
bool two−tables(V,S){
  int white = 0
  for s $\in$ V
    if (s.visited == false) // s is not visited
```

```
      if (dfs-visit(s, S, white)== false){
         return false
      }
  return true
}

bool dfs-visit(u, S,color-to-apply){
 if (u.visited == false){
    u.visited = true
    u.color = color-to-apply
    for v $\in$ S[u.id]
       if (dfs-visit(v,S,1-color-to-apply) == false)
           return false
 }
 else if (u.color =/= color-to-apply)
    return false

 return true
}
```

## Lab

**Exercise 5 (DFS)** Implement an *iterated* DFS; it should work for directed and indirected graphs. Do the DFS in such a way, that visiting times and finishing times are remembered in the nodes; and printed as well (together with the corresponding node name). The printing can be done during the traversal ("on-the-fly").

In the second half: use the implementation to solve one of the following, or, if you have time and energy, both:

1. *Topological sorting.* The input is a directed gragh. In case the graph is *cyclic*, give back an appropriate message to the user.

2. Biconnectivity, where the input is an undirected graph.