WS 2007/2008

# Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

# 1. Depth First Search

## 1.1 Application of Depth First Search: topological sorting

Finished last week !

## 1.2 Application of Depth First Search: determining biconnected components

### Definition 1

Let $G = (V, E)$ be a connected undirected graph. A vertex $a$ is said to be an *articulation point* of $G$ if there exist vertices $v$ and $w$, and every path between $v$ and $w$ contains the vertex $a$.

Stated another way, $a$ is an articulation point of $G$ if removing $a$ splits $G$ into two or more parts.

### Definition 2

The graph $G = (V, E)$ is called biconnected if for every distinct triple of vertices $v$, $w$, and $a$ there exist a path between $v$ and $w$ not containing $a$.

## Example 3

Consider $G = (V, E)$ such that

$$V = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9)$$

and

$$E = (\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_5\}, \{v_4, v_5\},$$
$$\{v_4, v_6\}, \{v_6, v_9\}, \{v_6, v_8\}, \{v_6, v_7\}, \{v_7, v_8\}, \{v_8, v_9\})$$

Articulation nodes: $v_2, v_4, v_6$. Biconnected components:
$E_1 = (\{v_4, v_6\})$, $V_1 = (v_4, v_6)$; $V_2 = (v_1, v_2, v_3)$,
$E_2 = (\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\})$; $V_3 = (v_2, v_4, v_5)$,
$E_3 = (\{v_2, v_4\}, \{v_2, v_5\}, \{v_4, v_5\})$; $V_4 = (v_6, v_7, v_8, v_9)$
$E_4 = (\{v_6, v_7\}, \{v_6, v_8\}, \{v_6, v_9\}, \{v_9, v_8\}, \{v_7, v_8\})$.

### Example 4

Consider the electric power net suppling a city. The failure at the articulation point of the net leads to power blackout of some parts of the city. To locate the crash - find the articulated vertices of the power net graph. To design the safe power supply - check the biconnectivity of the power net graph.

**Theorem 5**

*If $\{u, v\}$ is a back edge, then in the DFS forest $u$ is an ancestor of $v$ or vice versa.*

**Proof.**

Easy. Homework. □

## Theorem 6

*Vertex $a$ is an articulation point of $G$ if and only if either*

(1) *$a$ is the root and $a$ has more than one son, or*

(2) *$a$ is not the root, and for some son $s$ of $a$ there is no back edge between any descendant of $s$ (including $s$ itself) and an ancestor of $a$*

## Proof

- First, show that a root is an articulation point iff $a$ has more than one son.
  - Easy. Homework.
- (2) is true $\Rightarrow$ a (not the root) is an articulation point
  - Let $f$ be a father of $a$. According to Theorem 1 any back edge from a descendant $v$ of $s$ goes to the ancestor of $v$. By (2) the back edge can not go to the ancestor of $a$. Thus, every path from $s$ to $f$ contains $a$ implying that $a$ is an articulation point.

## Proof (Cont.)

- $a$ (not the root) is an articulation point $\Rightarrow$ (2) is true
  - Let $x$, $y$ be distinct vertices other than $a$. $x$ or $y$ (or both) is a descendant of $a$ (otherwise the path between $x$ and $y$ avoiding $a$ would exist, and $a$ would not be an articulation point). Two cases are possible (try to present them graphically !):
    1. Without loss of generality let $x$ be a descendant of $a$ and $y$ not. If in contradiction to (2) a back edge goes to the descendant of $a$, then this edge allows the way from $x$ to $y$ avoiding $a$. Contradiction to the hypothesis that $a$ is an articulation point.
    2. Let $x$ and $y$ be descendants of $a$. Let $x$ be a descendant of $s$ (perhaps $x = s$). Surely, $y$ is not the descendant of $s$ (otherwise the path avoiding $a$ would exist). Let $\tilde{s}$ be the son of $a$ such that $y$ is the descendant of $\tilde{s}$. The existence of a back edge from some descendant of $\tilde{s}$ would allow the path avoiding $a$. Contradiction to the hypothesis that $a$ is an articulation point.

□

### Exercise 1

*Homework: Modify the DFS algorithm to check the biconnectivity of a given graph. Hint: use Theorem 6 to check the existence of back edges.*

Iterative version of the DFS algorithm: Consider the data structure called stack. The following operations have to be supported:

- **void push(int)** – insert the element into the stack
- **in pop()** – delete the element into the stack

Properties:

- LIFO (Last Input First Output)
- The elements are inserted in the same order **push** is called
- The element deleted from the stack using **pop** is the one most recently inserted

## DepthFirstSearch:

void DepthFirstSearch(vertex $v$){
  initialize the empty stack; // global variable
  foreach ($v \in V$ ) do $v.dfsnum := 0$; od
  while $\exists v_0 \in V : v_0.dfsnum = 0$ do DFS($v_0$) od
  od }

## DFS:

void DFS(vertex $v$){
  push($v$);
  while (stack not empty) do
    $v :=$ pop();
    if ($v.dfsnum = 0$) then
      $v.dfsnum$:=counter++;
      foreach ($w | (v, w) \in E$ ($\{v, w\} \in E$) ) do
        push($w$);
      od
    fi
  od }

## DepthFirstSearch:

```
void DepthFirstSearch(vertex v){
  initialize the empty stack; // global variable
  foreach (v ∈ V ) do v.dfsnum := 0; od
  while ∃v0 ∈ V : v0.dfsnum = 0 do DFS(v0) od
  od }
```

## DFS:

```
void DFS(vertex v){
  push(v);
  while (stack not empty) do
    v:= pop();
    if (v.dfsnum = 0) then
      v.dfsnum:=counter++;
      foreach (w|(v, w) ∈ E ({v, w} ∈ E) ) do
        push(w);
      od
    fi
  od }
```

## 2. Breadth first search (BFS)

Consider the data structure called queue. The following operations have to be supported:

- **void enqueue(int)** – insert the element into the stack
- **int dequeue()** – delete the element into the stack

Properties:

- FIFO (First Input First Output)
- The elements are inserted in the same order **enqueue** is called
- The element deleted from the stack using **dequeue** is the first inserted

BreadthFirstSearch:

void BreadthFirstSearch(vertex $v$){
  initialize the empty stack; // global variable
  foreach $(v \in V\ )$ do $v.bfsnum := 0$; od
  while $\exists v_0 \in V : v_0.bfsnum = 0$ do DFS($v_0$) od
  od }

BFS:

void BFS(vertex $v$){
  enqueue($v$);
  while (stack not empty) do
    $v$:= dequeue();
    if $(v.bfsnum = 0)$ then
      foreach $(w|(v,w) \in E\ (\{v,w\} \in E)\ )$ do
        enqueue($w$);
        $w.bfsnum = v.bfsnum + 1$
      od
    fi
  od }

## BreadthFirstSearch:

void BreadthFirstSearch(vertex $v$){

  initialize the empty stack; // global variable

  foreach ($v \in V$ ) do $v.bfsnum := 0$; od

  while $\exists v_0 \in V : v_0.bfsnum = 0$ do DFS($v_0$) od

  od }

## BFS:

void BFS(vertex $v$){

  enqueue($v$);

  while (stack not empty) do

    $v$:= dequeue();

    if ($v.bfsnum = 0$) then

      foreach ($w | (v, w) \in E$ ($\{v, w\} \in E$) ) do

        enqueue($w$);

        $w.bfsnum = v.bfsnum + 1$

      od

    fi

  od }

Recall the classification of edges introduced last week:

- **Tree edges** – edge $(u, v)$ is a tree edge if $v$ was first discovered by exploring edge $(u, v)$ ($v.dfsnum = 0$).
- **Back edges** – edge $(u, v)$ connecting a vertex $u$ to an ancestor $v$ in a depth-first tree ($v.dfsnum < u.dfsnum$, and DFS($v$) is not finished).
- **Forward edges** – non-tree edges $(u, v)$ connecting a vertex $u$ to a descendant $v$ in a depth-first tree ($v.dfsnum > u.dfsnum$).
- **Cross edges** – are all other edges ($u.dfsnum > v.dfsnum$, and DFS($v$) is finished).

## Lemma 7

*In a breads first search of an undirected graph $G$, every edge of $G$ is either a tree edge, or a cross edge. Furthermore:*

- *for each tree edge $(u, v)$: $v.bfsnum = u.bfsnum + 1$*
- *for each cross edge $(u, v)$: $v.bfsnum = u.bfsnum + 1$ or $v.bfsnum = u.bfsnum$*

## Lemma 8

*In a breads first search of a directed graph $G$, every edge of $G$ is either a tree edge, or a cross edge, or back edge. Furthermore:*

- *for each tree edge* $(u, v)$*:* $v.bfsnum = u.bfsnum + 1$
- *for each cross edge* $(u, v)$*:* $v.bfsnum \leq u.bfsnum + 1$
- *for each back edge* $(u, v)$*:* $0 \leq v.bfsnum < u.bfsnum$