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. Graph Algorithms

Definition 1

Let $G = (V, E)$ be an undirected graph. Select two nodes v, w, and two edges $e, \tilde{e}$.

- v, w are called adjacent iff $\{v, w\} \in E$
- v, e are called incident iff $v \in E$
- e, $\tilde{e}$ are called adjacent iff $|e \cap \tilde{e}| \geq 1$
- e of the form $\{v, v\} = \{v\}$ is called *loop*

Lemma 2

Any undirected graph without loops contains at most $\binom{n}{2} = \frac{n(n-1)}{2}$ *edges,* $|V| = n$*. Any undirected graph with loops contains at most* $\binom{n+1}{2} = \frac{n(n+1)}{2}$ *edges,* $|V| = n$*.*

Proof.

Easy. Homework. Hint: Use $\binom{n+1}{2} = \binom{n}{2} + n$ □

Lemma 2

Any undirected graph without loops contains at most $\binom{n}{2} = \frac{n(n-1)}{2}$ *edges,* $|V| = n$. *Any undirected graph with loops contains at most* $\binom{n+1}{2} = \frac{n(n+1)}{2}$ *edges,* $|V| = n$.

Proof.

Easy. Homework. Hint: Use $\binom{n+1}{2} = \binom{n}{2} + n$ □

Lemma 2

Any undirected graph without loops contains at most $\begin{pmatrix} n \\ 2 \end{pmatrix} = \frac{n(n-1)}{2}$ *edges,* $|V| = n$. *Any undirected graph with loops contains at most* $\begin{pmatrix} n+1 \\ 2 \end{pmatrix} = \frac{n(n+1)}{2}$ *edges,* $|V| = n$.

Proof.

Easy. Homework. Hint: Use $\begin{pmatrix} n+1 \\ 2 \end{pmatrix} = \begin{pmatrix} n \\ 2 \end{pmatrix} + n$ □

Definition 3

Let $G = (V, E)$ be an undirected graph. Select $v \in V$. Define the neighborhood of v to be $N(v) = \{w \in V : \{v, w\} \in E\}$.

- $deg(v) = |N(v)|$
- $\delta(G) = min\{deg(v) : v \in V\}$
- $\Delta(G) = max\{deg(v) : v \in V\}$

Lemma 4

For any undirected $G = (V, E)$ the following is satisfied:

$$\sum_{v \in V} deg(v) = 2 \cdot |E|$$

Proof.

$\sum_{v \in V} deg(v)$ counts every edge twice. □

Lemma 4

For any undirected $G = (V, E)$ the following is satisfied:

$$\sum_{v \in V} deg(v) = 2 \cdot |E|$$

Proof.

$\sum_{v \in V} deg(v)$ counts every edge twice. □

Definition 5

Let $G = (V, E)$ be an undirected graph. Select $v \in V$. Define the neighborhood of v to be $N(v) = \{w \in V : \{v, w\} \in E\}$.

- $deg(v) = |N(v)|$
- $\delta(G) = min\{deg(v) : v \in V\}$
- $\Delta(G) = max\{deg(v) : v \in V\}$

2. Representation of graphs

2.1 Adjacency matrix

Definition 6

An *adjacency matrix* for $G = (V, E)$, $V = |n|$ is a $(n \times n)$-matrix $A = (a_{i,j})$, $n \geq i, j \geq n$ such that

- Case 1: G is undirected
- $a_{i,j} = \begin{cases} 1, & \{i, j\} \in E \\ 0, & \{i, j\} \notin E \end{cases}$
- Case 2: G undirected
- $a_{i,j} = \begin{cases} 1, & (i, j) \in E \\ 0, & (i, j) \notin E \end{cases}$

- Required space for adjacency matrix for $|V| = n$ is $\Theta(n^2)$.
- The adjacency matrix for an undirected graph is symmetric.
- The adjacency matrix for a directed graph is symmetric iff for every directed edge the antiparallel edge exists.
- The adjacency matrix for a directed graph has diagonal elements $\neq 0$ if there are loops.

2.2 Adjacency lists

Definition 7

An *adjacency list* is an array consisting of $|V|$ lists, which store the adjacent vertices for every $v \in V$.

- The order in which the adjacent vertices are stored can be chosen arbitrary
- For directed graphs two adjacency lists are introduced: for ancestors and for successors

3. Seaching in Graphs

3.1 Depth-First-Search

3.1.1 Recursive Version

- For every vertex $v \in V$ let us define its *DFS-number* to be the number of the step at which v is visited (initialized with 0)
- Let $v_0 \in V$ be an arbitrary start vertex
- Let *counter* be a global variable initialized with 1.

Algorithm:

void DFS(vertex v){
  $v.dfsnum$:= counter++;
  foreach $(w|(v,w) \in E\ (\{v,w\} \in E)\ )$ do
    if ($w.dfsnum$=0) then DFS(w);
  od }

3. Seaching in Graphs

3.1 Depth-First-Search

3.1.1 Recursive Version

- For every vertex $v \in V$ let us define its *DFS-number* to be the number of the step at which v is visited (initialized with 0)
- Let $v_0 \in V$ be an arbitrary start vertex
- Let *counter* be a global variable initialized with 1.

Algorithm:

void DFS(vertex v){
  $v.dfsnum$:= counter++;
  foreach $(w|(v,w) \in E$ $(\{v,w\} \in E)$) do
    if ($w.dfsnum$=0) then DFS(w);
  od }

3. Seaching in Graphs

3.1 Depth-First-Search

3.1.1 Recursive Version

- For every vertex $v \in V$ let us define its *DFS-number* to be the number of the step at which v is visited (initialized with 0)
- Let $v_0 \in V$ be an arbitrary start vertex
- Let *counter* be a global variable initialized with 1.

Algorithm:

void DFS(vertex v){
  $v.dfsnum$:= counter++;
  foreach $(w|(v, w) \in E \ (\{v, w\} \in E)\ )$ do
    if ($w.dfsnum$=0) then DFS(w);
  od }

3. Seaching in Graphs

3.1 Depth-First-Search

3.1.1 Recursive Version

- For every vertex $v \in V$ let us define its *DFS-number* to be the number of the step at which v is visited (initialized with 0)
- Let $v_0 \in V$ be an arbitrary start vertex
- Let *counter* be a global variable initialized with 1.

Algorithm:

```
void DFS(vertex v){
  v.dfsnum:= counter++;
  foreach (w|(v,w) ∈ E ({v,w} ∈ E) ) do
    if (w.dfsnum=0) then DFS(w);
  od }
```

The call

counter:=1;
DFS(v_0);

leads to visiting all verteces, which are reachable from v_0. Thus:

Algorithm:

void DepthFirstSearch(graph G){
counter:=1;
foreach $(v \in V)$ do $v.dfsnum := 0$ od
while $\exists v_0 \in V : v_0.dfsnum = 0$ do DFS(v_0) od }

Complexity: $O(n + m)$ (every vertex is visited plus every edge is visited (≤ 2 times)

3.1.2 Iterative version

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 ∃v_0 ∈ 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) ∈ E ({v,w} ∈ E) ) do
        push(w);
      od
    fi
  od }
```

3.2 Classification of edges:

DFS performs the partition of edges into four classes:

- **Tree edges** – edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
- **Back edges** – edge (u, v) connecting a vertex u to an ancestor v in a depth-first tree.
- **Forward edges** – nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- **Cross edges** – are all other edges.