Analysis of Algorithms  
- Elementary graphs algorithms -  

Andreas Ermedahl  
MRTC (Mälardalens Real-Time Research Center)  
andreas.ermedahl@mdh.se  
Autumn 2003

Graphs  
- Graphs are important mathematical entities in computer science and engineering  
- Often used to represent different kinds of relational information  
  - Dependencies between entities  
  - Distance between entities

Graphs  
- In many applications it is interesting to search or traverse the graph  
  - The least number of subway stations to get from Hässelby to Norsborg?  
  - How do I visit all subway stations in an efficient manner?  
- This and following lectures will deal with algorithms for this, and some applications

Representing graphs  
- There are many ways of representing graphs  
  - A graph \( G = (V, E) \) consists of vertices (nodes) \( V \) and edges \( E \)  
  - The representations differ in efficiency depending on the algorithm, whether the graph is dense or sparse, and other factors  
- We will consider two representations:  
  - Adjacency-list representation  
  - Adjacency-matrix representation

Representing graphs  
- Undirected graph

Representing graphs  
- Directed graph
**Adjacency-list representation**

- Adjacency-list representation $G = (V, E)$ represented by an array $Adj$ of $|V|$ lists, one per vertex.
- $Adj[u]$ contains pointers to (ID’s of) all vertices adjacent to $u$.
- Note “adjacent” is different when the graph is directed or undirected.
  - Directed: sum of length of all adjacency lists is $|E|$.
  - Undirected: sum of length of all adjacency lists is $2|E|$.
- Memory required: $\Theta(V + E)$.
- Often efficient representation if $G$ is sparse.
  - $|E|$ not too big compared with $|V|$.
- Easily adapted to weighted graphs.
  - Store weight $w$ on edge in adjacency list.

**Adjacency-matrix representation**

- Adjacency-matrix representation: $G = (V, E)$ represented by a boolean matrix $A$ where:
  
  $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$

  - Size $O(|V|^2)$ (independent of $|E|$), efficient when $|E| = \Theta(|V|^2)$ (and graph size is small).
  - Adjacency matrix symmetric when graph is undirected, then suffices to store one half (still size $O(|V|^2)$, though).
  - Unweighted graph can be stored using one bit per entry.
  - Easily adapted to weighted graphs.
  - Store weight $w$ for edge $(u,v)$ at entry in row $u$ and column $v$.

**Traversing graphs**

- Most basic algorithms on graphs will be applications of graph traversal.
  - Printing or validating each edge/vertex.
  - Copying a graph or converting between representations.
  - Counting the number of edges/vertices.
  - Identifying connected components.
  - Finding paths between two vertices, or cycles.

**Order of exploration**

- The order in which we explore vertices depends on the container used for storing discovered but not finished vertices.
- There are two types of containers used:
  - Queue: leads to so called breadth-first search.
  - Stack: leads to so called depth-first search.
- We will investigate these two graph traversal algorithms in more detail.

**Breadth-first search**

- Simple algorithm for searching a graph.
- Input: A directed/undirected graph with source $s$.
- Output: The shortest distance from $s$.
  - $d[u]$ = shortest distance of node $u$ to $s$.
  - $p[u]$ = predecessor of node $u$ in shortest path to $s$.
  - Represented by a breadth-first tree.
- Only visits vertices reachable from $s$.
- Running time: $O(V + E)$
  - Linear time with respect to adjacency list.

**Traversing graphs**

- Efficiency and correctness.
  - Efficiency: Don’t loop or visit vertices repeatedly.
  - Correctness: Don’t miss any vertex.
- We need to mark vertices as we traverse the graph.
  - Undiscovered (white), the initial state, before we’ve seen it.
  - Discovered (gray), we’ve seen the vertex but not all of its incident edges.
  - Finished (black), all incident edges have been visited.
Breadth-first search

- Given graph G and source vertex s, find all vertices reachable from s by a breadth-first search from s (where s is seen as the root in a tree spanning G)
- Breadth-first means all vertices at “depth” k from s are visited before those at depth k + 1 are
- The algorithm yields the following results:
  - The distance (length of shortest path) from s to any other reachable vertex
  - A s-rooted “breadth-first tree” that consists of the shortest paths from s to all other vertices
- Works on both directed and undirected graphs

The BFS algorithm

```python
BFS(G, s)
for each vertex v in G - {s} do
    d[v] = -
    color[v] = NIL
    d[s] = 0
    color[s] = GRAY
    Q = [s]
while Q ≠ Ø do
    u = DEQUEUE(Q)
    for each v in Adj[u] do
        if color[v] = WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            p[v] = u
            Q = ENQUEUE(Q, v)
```

Example: Breadth-first search

- After start node s has been added to Q

```
Q: [s]
```

- After node w has been processed and its neighbours t and x have been added

```
Q: [s, t, w, x]
```

- After node r has been processed and its neighbour v has been added

```
Q: [s, t, w, x, v]
```
Example: Breadth-first search
• After node \( t \) has been processed and its neighbour \( u \) has been added

\[
\begin{array}{c}
\text{BFS}(G, s) \\
2 \quad \text{for each vertex}\ v \in V \setminus \{s\} \\
3 \quad \text{do} \quad \text{color}(v) \leftarrow \text{WHITE} \\
4 \quad d[v] \leftarrow \infty \\
5 \quad n[v] \leftarrow \text{NIL} \\
6 \quad \text{color}(s) \leftarrow \text{GRAY} \\
7 \quad d[s] \leftarrow 0 \\
8 \quad n[s] \leftarrow \text{NIL} \\
9 \quad Q \leftarrow \emptyset \\
10 \quad \text{ENQUEUE}(Q, s) \\
11 \quad \text{while} \ Q \neq \emptyset \\
12 \quad \text{do} \ u \leftarrow \text{DEQUEUE}(Q) \\
13 \quad \text{for each}\ v \in \text{Adj}(u) \\
14 \quad \text{if color}(v) = \text{WHITE} \\
15 \quad \text{then color}(v) \leftarrow \text{GRAY} \\
16 \quad d[v] \leftarrow d[u] + 1 \\
17 \quad \text{ENQUEUE}(Q, v) \\
18 \quad \text{color}[s] \leftarrow \text{BLACK}
\end{array}
\]

Analysis of BFS
First loop takes \( O(|V|) \) time

What about the outer while-loop? What about the inner for-loop?
Analysis of BFS

Outer loop (while):
– Can we bound the time until Q becomes empty?
• Only white vertices are enqueued, and they are always grayed when enqueued
  ⇒ Thus a vertex can be enqueued at most once
• One vertex dequeued for every lap
  ⇒ Thus outer while loop executed O(|V|) times

Whole loop nest
⇒ O(|V|) + O(|E|) = O(|V| + |E|)

Depth-first search

• Simple algorithm which searches “deeper” in the graph whenever possible
• Input: A directed/undirected graph.
• Output:
  – Deep-first forest (composed of depth-first trees)
  – Each vertex \( u \) is time-stamped: discover \( d[u] \) and finish \( f[u] \).
  – Edges \( \{\text{tree}, \text{back}, \text{forward}, \text{or cross edge}\} \).
• Visits all vertices.
• Running time: \( \Theta(V+E) \)
  Linear time with respect to adjacency list.

The DFS algorithm
– \( d[u] \) is timestamp when vertex \( u \) first discovered
– \( f[u] \) is timestamp when search of \( u \)’s adjacency list is completed
– BFS only builds a tree of vertices reachable from some given root vertex

Depth-first search

• Searches graph by recursively exploring the vertices in the adjacency list
• All vertices reachable from a vertex in the adjacency list are recursively searched before next vertex in the list is explored
• DFS constructs a depth-first forest that contains all vertices in the graph
  ⇒ BFS only builds a tree of vertices reachable from some given root vertex
Depth-first search: vertex classification

- Time-stamp vertices when they are discovered/finished:
  - \( d[u] \): when discovered
  - \( f[u] \): when finished
- The vertex colors are equivalent to the following cases:
  - **White**: the initial state, before we've seen it
  - **Gray**: we've seen the vertex but not all of its incident edges
  - **Black**: all incident edges have been visited

\[ \begin{array}{ccc}
 & \text{White} & \text{Gray} & \text{Black} \\
The vertex colors \end{array} \]

Depth-first search: edge classification

- Edges are classified according to the following four cases:
  - **Tree-edge (T)**
  - **Back-edge (B)**
  - \( d[u] < d[v] \): **Forward-edge (F)**
  - \( d[u] > d[v] \): **Cross-edge (C)**

Depth-first search example

1. \( S: \{ \} \)
2. \( S: \{a\} \)
3. \( S: \{a, d\} \)
4. \( S: \{a, d, f\} \)
Depth-first search example

S: \{a, d\}

Depth-first search example

S: \{a, d, g\}

Depth-first search example

S: \{a\}

Depth-first search example

S: \{a, b\}

Depth-first search example

S: \{a, b, e\}
Depth-first search example

S: \{a, b\}

Depth-first search example

S: \{a\}

Depth-first search example

S: \{\}\n
Depth-first search example

S: \{c\}

Depth-first search example

S: \{c, h\}

Depth-first search example

S: \{c\}
Analysis of DFS

- Running time of DFS:
  - First loop in DFS takes $\Theta(|V|)$ time (each vertex visited exactly once)
  - In second loop with the recursive calls, observe that DFS visit will be called exactly once on each vertex (requires a proof, really . . .)
- For each vertex $v$ where DFS-VISIT($v$) is called, the loop in DFS-VISIT is called $|\text{Adj}[v]|$ times
- Since $\sum_v |\text{Adj}[v]| = \Theta(|E|)$ the total cost of second loop in DFS will be $\Theta(|E|)$
- Thus, running time of DFS is $\Theta(|V| + |E|)$

Topological sort

- A linear order of all nodes in the graph $G$ such that if $G$ contains an edge $(u, v)$ then $u$ appears before $v$ in the ordering
  - Topological sort is only possible if the graph is acyclic
- One application: DAG represents precedence relations between tasks or events
  - Edge between tasks if first task must be performed before second task
  - Then a topological sort gives a possible schedule of the tasks on a single resource
    - One precedence graph might have several possible schedules

Topological sort: algorithm

- We can use DFS to get a topological sort
- Informal description:
  - Call DFS($G$) to compute finishing times $f[u]$ for all vertices $u$
  - Put nodes into list so they are stored in decreasing order w.r.t. finishing time
- A direct way is to first use DFS($G$) and then sort w.r.t. finishing time $f[u]$
  - This yields time $\Theta(|V| + |E|) = O(|V| \log |V| + |V| + |E|)$
  - But easy to modify DFS to compute the sorted list “on the fly”:
    - Just insert each vertex into list immediately when finished
    - Adds no asymptotic complexity to DFS: still $\Theta(|V| + |E|)$
**Strongly connected components**

A strongly connected component (SCC) of directed graph $G = (V,E)$ is a maximal subset of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$ both vertices are reachable from each other.

![Directed graph with SCCs](image)

**Graph Transpose**

The transpose of a directed graph $G=(V,E)$ is the graph $G^T = (V,E^T)$ such that $E^T = \{(v,u) : (u,v) \in E\}$.

Thus, $G^T$ is $G$ with all edges reversed.

- Both graphs contain the same nodes.
- Both graphs have the same SCCs.

$G^T$ can be created in $O(V + E)$ time.

![Original graph G and transposed graph G^T](image)

**Finding SCCs**

We can use algorithms for DFS and graph transposal for finding SCCs of graph $G$.

Steps:
1. Call $DFS(G)$ to compute finishing times $f[u]$ for each vertex $u$ in $G$.
2. Compute transpose graph $G^T$.
3. Call $DFS(G^T)$, but in the main loop, consider the vertices in decreasing $f[u]$ (as computed in line 1).
4. Each depth-first tree found in $G^T$ forms a SCC in $G$.

![Finding SCCs](image)

**Finding SCCs**

1. Call $DFS(G)$ to compute finishing time $f[u]$ for each vertex $u$.

![Finding SCCs with finishing times](image)

Sort nodes on their finishing time:

$\Rightarrow b,e,a,c,d,g,f,h$

2. Compute $G^T$ from original graph.

![Finding SCCs with transposed graph](image)
Finding SCCs

3. Call DFS($G^T$), but in the main loop, consider the vertices in decreasing $f[u]$ (as computed in line 1) $\Rightarrow$ b, e, a, c, d, g, f, h

Deep first trees found: {b, a, e}, {c, d}, {g, f}, {h}

Finding SCCs

4. Each depth-first tree found in $G^T$ forms a SCC in $G$: {b, a, e}, {c, d}, {g, f}, {h}

5. Collapse all SCC to one single node.
   - The result is the acyclic $G^{SCC}$ graph

Analysis of SCC algorithm

1. Call DFS($G$) to compute finishing times $f[u]$ for each vertex $u$ in $G$ $\Rightarrow \Theta(V + E)$
2. Compute transpose graph $G^T$ $\Rightarrow \Theta(V + E)$
3. Call DFS($G^T$), but in the main loop, consider the vertices in decreasing $f[u]$ (as computed in line 1) $\Rightarrow \Theta(V + E)$
4. Collapse each depth-first tree found in $G^T$ to a SCC $\Rightarrow \Theta(V + E)$

Overall total time: $\Theta(V + E)$