# W4231: Analysis of Algorithms

10/21/1999 (revised 10/25)

- Definitions for graphs
- Breadth First Search and Depth First Search



## Graphs

A graph $G$ is given by a set of vertices $V$ and a set of edges $E$.

Normally we call $n = |V|$ and $m = |E|$.

- In a **directed** graph, an edge is an **ordered pairs** of vertices $(u, v)$. The edge goes **from** $u$ **to** $v$ and is represented using an arrow.
- In an **undirected** graph, an edge is a **set** (**unordered pair**) of two vertices $\{u, v\}$.



## Expressive power

A graph can be used to represent a communication network, a hierarchy of classes, the topology of a maze, relationships between people, a subway map, a finite-state automaton, the web . . .

Each application motivates a series of computational problems.

We will see efficient solutions to the most basic ones:

- Connectivity and Shortest Paths.
- Cuts, Flows, Matching.



## Representation

There are two simple ways of representing a directed graph $G = (V, E)$. Assume $V = \{1, \ldots, n\}$.

- **Adjacency List.** For every node $u$ we maintain a list of all the nodes $v$ such that $(u, v) \in E$.
- **Adjacency Matrix.** A $n \times n$ Boolean matrix $M[\cdot, \cdot]$ is maintained, where

$$M[u, v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$



## Comparison

- An adjacency list representation uses $O(n + m)$ space: we have an array of $n$ pointers and the sum of the number of elements in all the lists is $m$.

  Deciding whether $(u, v) \in E$ takes $O(n)$ time in the worst case.



- An adjacency matrix uses $O(n^2)$ space.

  Deciding whether $(u, v) \in E$ takes $O(1)$ time in the worst case.

Assuming names of vertices and pointers use 2 bytes each, adjacency list requires $2n + 4m$ bytes of space ($2n + 8m$ for undirected graphs), adjacency matrix $n^2/8$.

## Terminology — Undirected Graph

- $u$ and $v$ are **adjacent** (or **neighbors**) if $\{u, v\} \in E$.
- The **degree** of $u$ is the number of its neighbor (the size of its adjacency list).
- A **path** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that any two consecutive vertices are adjacent. The **length** of the path is $k-1$. A path is **simple** if no vertex is duplicated.
- A **cycle** is a path $v_1, v_2, \ldots, v_k$ where $v_1 = v_k$. A cycle is **simple** if $v_1, \ldots, v_{k-1}$ are all different.



- Two vertices $s$ and $t$ are **connected** if there is a path

$$s = v_1, v_2, \ldots, v_k = t$$

- The equivalence relation "being connected to" among vertices partitions the set of vertices into **connected components**.
- A graph is **connected** if any two vertices are connected. (I.e. the whole graph is a single connected component.)

It is possible to test whether a graph is connected in optimal $O(n+m)$ time.



## Terminology — Directed Graph

- Path, simple path, cycle, simple cycle, as before.
- Two vertices $s$ and $t$ are **strongly connected** if there is a directed path from $s$ to $t$ **and** a directed path from $t$ to $s$.
- The relation "being strongly connected to" partitions the set of vertices into **strongly connected components**. A graph is strongly connected if all its vertices are in the same strongly connected component.

It is possible to test whether a graph is strongly connected in optimal $O(n+m)$ time. (No proof)



## Search

Several graph algorithms use a procedure that "searches" the graph "visiting" all edges.

The two main methods to search a graph are

- **Breadth-first search**
- **Depth-first search**



## Breadth First Search

Start from a vertex, then visit all vertices at distance one, then visit all vertices at distance two, . . .



## Implementation

We use a queue $Q$ and a vector of $n$ "colors", one for each vertex.

```
BFS (s, G = (V, E))
begin
   Initialize Q;
   for all u ∈ V do Initialize col(u) := white
   col(s) := gray; enqueue (s, Q)
   while Q is not empty
      u := dequeue (Q); col(u) := black
      for all v such that (u, v) ∈ E and col(v) = white do
         col(v) := gray
         enqueue(v, Q)
end
```

## Analysis

- Using adjacency list, running time is $O(n+m)$.
- We do $O(1)$ operations on every vertex, and $O(1)$ operations on every edge.
- At the end, the black vertices are precisely those in the connected component of $s$ (for undirected graphs).



## Depth First Search

We follow a direction, as far as possible, and then we backtrack.

Optimal strategy to get out of a maze (BFS is also optimal, but DFS is more natural).



## Recursive Implementation — Simple Version

Basic idea (works for undirected connected graphs):

DFS $(s, G = (V, E))$
    **for all** $u \in V$ **do** Initialize $col(u) := white$
    DFS-R $(s, G)$
**end**

DFS-R $(s, G = (V, E))$
    $col(s) := black$;
    **for all** $v$ such that $(s, v) \in E$ and $col(v) = white$ **do**
        DFS-R $(v, G)$



## Non-recursive Implementation

Non-recursive implementation is similar to BFS but uses a **stack** instead of a **queue**.



## Recursive Implementation — General Version

time is a **global** variable.

DFS $(G = (V, E))$
    **for all** $u \in V$ **do** $col(u) := white$
    $time := 0$
    **for all** $u \in V$ **do if** $col(u) = white$ **then** DFS-R $(u, G)$

DFS-R $(s, G)$
    $time := time + 1$; $d(s) := time$; $col(s) = gray$
    **for all** $v$ such that $(s, v) \in E$ **do if** $col(v) = white$ **then**
        DFS-R $(v, G)$
    $col(s) := black$
    $time := time + 1$; $f(s) = time$



## Discovery Time and Finish Time

The algorithm assigns to every vertex $u$ a **discovery** time $d(u)$ and a **finish** time $f(u)$.

A "clock" is maintained during the execution of the algorithm in the variable $time$. Each vertex is "time-stamped" the first time that it is seen, and the last time that it is dealt with.

# Building a DFS Tree

By a further modification of the procedures DFS and DFS-R, we can also build a tree (or rather a forest).

The roots of the forest are the nodes on which we call DFS-R from within DFS.

The edges in the forest are the edges of the form $(s, v)$ where $s$ is the parameter in a call of DFS$(s, G)$ and $v$ is white, and $DFS(v, G)$ is the resulting procedure call.

The forest represents the way the recursive calls "unfold" during the computation.



# Edges in the DFS Tree

An edge $(u, v)$ is a

- **Tree edge** if it is part of the forest.
- **Back edge** if $v$ is an ancestor of $u$ in the tree.
- **Forward edge** if $v$ is a descendant of $u$ in the tree.
- **Cross edge** otherwise.

In a the DFS forest of an undirected graph, there is no difference betrween forward and back edges, and there are no cross edges.



# Acyclic Graphs

An **acyclic** graph is a directed graph without cycles.

Acyclic graphs represent hierarchical structures, e.g. precedence constraints (as in the `make` command, or in course prerequisites).



# Topological Sort

Suppose $V$ is a set of **actions** that we have to perform, and $(u, v) \in E$ iff action $u$ has to be done **before** action $v$.

We want to find a schedule $v_1, \ldots, v_n$ of the actions such that if $(v_i, v_j) \in E$ then $i < j$.

If the graph contains a cycle we are not going to be able to do that.

If the graph is acyclic we can always find a feasible schedule, and we can do so efficiently.