Definitions for graphs
- Breadth First Search and Depth First Search
- Topological Sort.

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.

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.

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 $\{u, v\}$.

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

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

- 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

- 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

- 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, \ldots

Implementation

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

```plaintext
BFS (s, G = (V, E))
begin
 Initialize Q;
 for all $u \in V$ 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 $(u, v) \in E$ and col(v) = white do
     col(v) := gray
     enqueue(v, Q)
 end
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).

Rationale

Whenever a new (white) vertex is found, it is reached through a shortest path from $s$.

Will prove later.

We maintain a vector of distances $d[\cdot]$, where $d[u]$ is the distance from $s$ to $u$.

Distance

Say that the distance between $s$ and $t$ is the smallest $k$ such that there is a path of length $k$ connecting $s$ to $t$. (Distance is undefined, or $\infty$, is $s$ and $t$ are not connected.)

BFS can be modified to find the shortest path between $s$ and every other vertex.

Initially, $d[s] = 0$ and $d[u] = \infty$ for $u \neq s$.

Inductively, it will always be true that all vertices in the queue have the right entry in the $d[\cdot]$ vector.

When we are looking at the neighbours of $u$, the white ones will be at distance $d[u] + 1$ from $s$.

Modified BFS

```plaintext
BFS (s, G = (V, E))
- Initialize Q;
- for all u ∈ V do Initialize col(u) := white
- for all u ∈ V do Initialize d[u] := ∞
- col(s) := gray; d[s] := 0
- 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;
    - d[v] := d[u] + 1
    - enqueue(v, Q)
```

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):

\[
\text{DFS} \ (s, G = (V,E)) \\
\text{ for all } u \in V \ \text{do} \ \text{Initialize} \ col(u) := \text{white} \\\n\text{DFS-R} \ (s,G)
\]

\[
\text{DFS-R} \ (s, G = (V, E)) \\
\text{col}(s) := \text{black}; \\
\text{for all } v \text{ such that } (u,v) \in E \ \text{and} \ \text{col}(v) = \text{white} \ \text{do} \\
\text{DFS} \ (v, G)
\]

Recursive Implementation — General Version

time is a global variable.

\[
\text{DFS} \ (G = (V,E)) \\
\text{ for all } u \in V \ \text{do} \ \text{Initialize} \ col(u) := \text{white} \\
\text{time} := 0 \\
\text{for all } u \in V \ \text{do} \ \text{if} \ col(u) = \text{white} \ \text{then} \ \text{DFS-R} \ (u,G)
\]

\[
\text{DFS-R} \ (s, G) \\
\text{time} := \text{time} + 1; \ \text{d}(s) := \text{time}; \ \text{col}(s) := \text{gray} \\
\text{for all } v \text{ such that } (s,v) \in E \ \text{if} \ \text{col}(v) = \text{white} \ \text{then} \\
\text{DFS} \ (v, G) \\
\text{col}(s) := \text{black} \\
\text{time} := \text{time} + 1; \ \text{f}(s) = \text{time}
\]

Non-recursive Implementation

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

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 \( \text{time} \). 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 between 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.

One Algorithm for “Topological Sort”

1. Find a node $v$ with out-degree zero; make $v$ be the last element of the schedule.
2. Delete $v$ and its incident edges from the graph. Schedule recursively the remaining vertices.

Time: $O(n(n+m))$ with careless implementation.
Correctness: exercise.

The Optimal and Surprising Algorithm

**Algorithm:**

- Do DFS; schedule the vertices by decreasing values of $f()$. (Latest finish first)

**Claim:** if the graph is acyclic, the nodes in the list are ordered in the right way.

Analysis

- **Running time:** $O(m + n)$. We can modify DFS-R so that every time we are finished with a vertex we put it on top of an initially empty linked list.
- **Correctness:** by the following two results:
  - $G$ is acyclic $\Leftrightarrow$ there are no back edges in the DFS forest.
    - We only need $\Rightarrow$
  - Cross edges and forward edges always go from nodes with higher finish time to nodes with lower finish time.

First Step

**Lemma 1.** If $G$ is acyclic then the DFS forest of $G$ has no back edge.

**PROOF:** If there is a back edge then there is a cycle.
The analysis works

Theorem 2. If $G$ is acyclic, the order of discovery in DFS is a good topological sort.

PROOF: We want to show that if there is an edge $(u, v)$ then $f(u) > f(v)$. When $(u, v)$ is considered:

- $v$ is not gray, otherwise $u$ would be a descendent of $v$ and $(u, v)$ be a back edge.
- If $v$ is white, $v$ becomes a child of $u$, and $f(u) > f(v)$.
- If $v$ is black, then $f(v) < f(u)$ too.

A Converse to Lemma 1

Lemma 3. If the DFS forest of $G$ has no back edge then $G$ is acyclic.

PROOF: If there is a cycle, let $v$ be the first discovered vertex of the cycle, and let $u$ be the predecessor of $v$ in the cycle. $v$ is discovered before $u$, and there is a path (made by all white vertices) from $v$ to $u$. It follows that $u$ is a descendent of $v$ in the DFS tree (this is quite obvious, but we better prove it later).

Then $(u, v)$ is a back edge.

To complete the argument

Theorem 4. For any two vertices $u$ and $v$, exactly one of the following cases hold:

1. The intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ are disjoint.
2. $[d(u), f(u)]$ contains $[d(v), f(v)]$ and $v$ is a descendent of $u$ in the same DFS tree.
3. $[d(v), f(v)]$ contains $[d(u), f(u)]$ and $u$ is a descendent of $v$ in the same DFS tree.

Theorem 5. [White Path Theorem] If at time $d(u)$ there is a path of white vertices going from $u$ to $v$ (v included) then $v$ will become a descendent of $u$ in the DFS forest.

PROOF: Suppose not. Then assume that all the other vertices in the $u \rightarrow v$ path become a descendant of $u$, except $v$. (Otherwise repeat the argument using instead of $v$ the closest element to $u$ in the path that does not become a descendant.) Then let $w$ be the predecessor of $v$, then

$$d(u) \leq d(v) \leq f(w) \leq f(v)$$

Then the interval $[d(v), f(v)]$ is contained in $[d(u), f(u)]$ and so $v$ is a descendant of $u$. 