Lectures 21-23
Graphs
• Basic model
• Traversals -- BFS

Adam Smith
**Graphs**

**Definition.** A *directed graph (digraph)* $G = (V, E)$ is an ordered pair consisting of

- a set $V$ of *vertices* (synonym: *nodes*),
- a set $E \subseteq V \times V$ of *edges*
- An edge $e= (u, v)$ goes “from $u$ to $v$” (may or may not allow $u=v$)

- In an *undirected graph* $G = (V, E)$, the edge set $E$ consists of *unordered* pairs of vertices
  - Sometimes write $e=\{u,v\}$

- Sometimes, store additional info: edge lengths, weights
- How many edges can a graph have?
  - In either case, $|E| = O(V^2)$.  

9/3/2008

*A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne*
## Graphs are everywhere

<table>
<thead>
<tr>
<th>Example</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transportation network: airline routes</td>
<td>airports</td>
<td>nonstop flights</td>
</tr>
<tr>
<td>Communication networks</td>
<td>computers, hubs, routers</td>
<td>physical wires</td>
</tr>
<tr>
<td>Information network: web</td>
<td>pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>Information network: scientific papers</td>
<td>articles</td>
<td>references</td>
</tr>
<tr>
<td>Social networks</td>
<td>people</td>
<td>“u is v’s friend”,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“u sends email to v”,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“u’s MySpace page links to v”</td>
</tr>
<tr>
<td>Computer programs</td>
<td>functions (or modules)</td>
<td>“u calls v”</td>
</tr>
<tr>
<td></td>
<td>statement blocks</td>
<td>“v can follow u”</td>
</tr>
</tbody>
</table>

A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne
Paths and Connectivity

• Path = sequence of consecutive edges in $E$
  – $(u,w_1), (w_1,w_2), (w_2,w_3), \ldots, (w_{k-1},v)$
  – Write $u \leftrightarrow v$ or $u \sim v$
  – (Note: in a directed graph, direction matters)
  – Path is simple if no vertex is repeated

• Undirected graph $G$ is **connected** if for every two vertices $u,v$, there is a path from $u$ to $v$ in $G$

• Directed graph?
  – **Strongly connected** if for every pair, $u \sim v$ and $v \sim u$
Puzzles

• Suppose an undirected graph $G$ is connected.
  – True or false? $G$ has at least $n-1$ edges.

• Suppose that an undirected graph $G$ has exactly $n-1$ edges (and no self-loops)
  – True or false? $G$ is connected.
  – What if $G$ has $n-1$ edges and no cycles?
Paths and Connectivity

- Def. A **path** in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$.

- Def. A path is **simple** if all nodes are distinct.

- Def. An undirected graph is **connected** if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 
Cycles

- Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

cycle $C = 1-2-4-5-3-1$
Trees

• Def. An undirected graph is a tree if it is connected and does not contain a cycle.

• Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  – G is connected.
  – G does not contain a cycle.
  – G has n-1 edges.
Rooted Trees

- Rooted tree. Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.
- Importance. Models hierarchical structure.
Phylogeny Trees

- Phylogeny trees. Describe evolutionary history of species.
Exploring a graph

**Classic problem:** Given vertices $s, t \in V$, is there a path from $s$ to $t$?

**Idea:** explore all vertices reachable from $s$

**Two basic techniques:**

- **Breadth-first search (BFS)**
  - Explore children in order of distance to start node

- **Depth-first search (DFS)**
  - Recursively explore vertex’s children before exploring siblings

9/5/2008

*A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne*
Graphs as data structures

• Representation: two common ones
  – adjacency matrix
  – adjacency lists
  – Python: dict of dicts

• Basic operations: traversals
  – breadth-first search
  – depth-first search
The **adjacency matrix** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \ldots n, 1 \ldots n]$ given by

$$A[i, j] = \begin{cases} 
1 & \text{if } (i, j) \in E, \\
0 & \text{if } (i, j) \notin E. 
\end{cases}$$

### Storage:

$\Theta(V^2)$

Good for **dense** graphs.

- **Lookup:** $O(1)$ time
- **List all neighbors:** $O(|V|)$
Adjacency list representation

• An **adjacency list** of a vertex $v \in V$ is the list $\text{Adj}[v]$ of vertices adjacent to $v$.

\[
\begin{align*}
\text{Adj}[1] &= \{2, 3\} \\
\text{Adj}[2] &= \{3\} \\
\text{Adj}[3] &= \{} \\
\text{Adj}[4] &= \{3\}
\end{align*}
\]

For undirected graphs, $|\text{Adj}[v]| = \text{degree}(v)$. For digraphs, $|\text{Adj}[v]| = \text{out-degree}(v)$.

**How many entries in lists?** $2|E|$

**Total** $\Theta(V + E)$ storage — good for **sparse** graphs.
Exercise

• What if I use a hash table to keep track of the adjacency list of each vertex?
• Two possibilities
  – Table of outgoing edges for each vertex
  – Two tables for each vertex, one in, one out.
Implementing Traversals

Generic traversal algorithm

1. \( R = \{s\} \)

2. While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \),
   - Add \( v \) to \( R \)

To implement this, need to choose…

- Graph representation
- Data structures to track…
  - Vertices already explored
  - Edge to be followed next

These choices affect the order of traversal

A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne
Breadth First Search

• BFS intuition. Explore outward from $s$ in all possible directions, adding nodes one "layer" at a time.

• BFS algorithm.
  - $L_0 = \{ s \}$.
  - $L_1 =$ all neighbors of $L_0$.
  - $L_2 =$ all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
  - $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$. 

\[
\begin{align*}
S & \\
L_1 & \\
L_2 & \\
\ldots & \\
L_{n-1} & \\
\end{align*}
\]
Breadth First Search

- Property. Let T be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
BFS example in a directed graph

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{s} \\
\text{e} \\
\text{c} \\
\text{f} \\
\text{g} \\
\text{h} \\
\text{i}
\end{array}
\]

Can show that \( Q \) consists of vertices with \( d \) values.

--

Only 1 or 2 values.

--

If 2, differ by 1 and all smallest are first.

--

Since each vertex gets a finite \( d \) value at most once, values assigned to vertices are monotonically increasing over time.

--

Actual proof of correctness is a bit trickier. See book.

--

BFS may not reach all vertices.

--

Depth-first search

Input: \( G \), directed or undirected. No source vertex given!

Output: 2 timestamps on each vertex: \( d \) (discovery time), \( f \) (finishing time).

These will be useful for other algorithms later on.

Can also compute \( |E| \).

[See book.]

Will methodically explore every edge.

Start over from different vertices as necessary.

As soon as we discover a vertex, explore from it.

Unlike BFS, which puts a vertex on a queue so that we explore from it later.
BFS example in a directed graph

A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne
Application: connected component

- **Connected component**
  - set of nodes all of which can reach each other

- **Every undirected graph can be broken up into disjoint connected components**
  - BFS finds all nodes in connected component of s
BFS with adjacency list rep.

- Level[1 .. n]:
  - array of integers
  - (Or hash table where value = level)
- Queue Q
  - initialized to empty
- Tree T
  - initialized to empty
  - Could be just a set of edges
  - Or could simply store the parent pointer for each vertex

A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne
Theorem: BFS takes $O(m+n)$ time

BFS(s):
1. Initialize Level[$v$] to infinity for all vertices
2. Set Level[$s$]=0
3. Add $s$ to $Q$
4. While (Q not empty)
   a) Dequeue (u)
   b) For each edge (u,v) adjacent to u
      a) If Level[v] = $\infty$ then
         a) Add edge (u,v) to tree T (alt: parent[v] = u)
         b) Add v to Q
         c) Level[v] := Level[u]+1
Implementing graph attributes

No one best way to implement. Depends on the programming language, the algorithm, and how the rest of the program interacts with the graph.

If representing the graph with adjacency lists, can represent vertex attributes in additional arrays that parallel the `Adj` array, e.g., `d[1 : |V|]`, so that if vertices adjacent to `u` are in `Adj[u]`, store `u:d` in array entry `d[u]`.

But can represent attributes in other ways. Example: represent vertex attributes as instance variables within a subclass of a `Vertex` class.

### Breadth-first search

**Input:** Graph `G = (D, V; E)`

- either directed or undirected
- and source vertex `s ∈ V`.

**Output:**

- `d ∈ D` distance (smallest # of edges) from `s` to `v`, for all `v ∈ V`.

In book, also `v` such that `(u, v)` is last edge on shortest path `s` to `v`. `u` is `v`'s predecessor. Set of edges `f (u, v): v ∈ G \ v \ u \ u \ v \ u \ g forms a tree.`

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple. Compute only `d`, not `v` such that `(u, v)` is last edge on shortest path. [See book for `v` such that `(u, v)`.

Omitting colors of vertices. [Used in book to reason about the algorithm. We'll skip them here.]

**Idea**

Send a wave out from `s`.

First hits all vertices 1 edge from `s`.

From there, hits all vertices 2 edges from `s`.

Etc.

Use FIFO queue `Q` to maintain wavefront.

`BFS(V, E, s)`

```plaintext
for each u ∈ V - {s}
    u.d = ∞

s.d = 0
Q = ∅
ENQUEUE(Q, s)

while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.d == ∞
            v.d = u.d + 1
            ENQUEUE(Q, v)
```

(S. Raskhodnikova and A. Smith. Based on notes by E. Demaine and C. Leiserson)
Analyzing graph algorithms

- Two important parameters
  - \( n = |V| \) = number of vertices
  - \( m = |E| \) = number of edges
Theorem: BFS takes $O(m+n)$ time

BFS(s):
1. Initialize Level[v] to infinity for all vertices
2. Set Level[s]=0
3. Add s to Q
4. While (Q not empty)
   a) Dequeue (u)
   b) For each edge (u,v) adjacent to u
      a) If Level[v]=∞ then
         a) Add edge (u,v) to tree T
         b) Add v to Q
      c) Level[v] := Level[u]+1

O(1) time, run once overall.
O(1) time, run once per vertex
O(1) time per execution, run at most twice per edge

Total: $O(m+n)$ time
(linear in input size)

9/5/2008

A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova, K. Wayne
Properties

• If \( s \) is the root of BFS tree, then for every vertex \( u \),
  – path in BFS tree from \( s \) to \( u \) is a shortest path in \( G \)
  – depth in BFS tree = distance from \( u \) to \( s \)

• Proof of BFS correctness: see CLRS, Chap 22.
Flood Fill

- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.
Flood Fill

- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.

recolor lime green blob to blue
BFS Review

• If G is directed, can I have a BST tree with
  – Cross edges?
  – Back edges? (How many levels back?)
  – Forward edges?

• How do the answers change if G is undirected?

• In any graph, edges go at most one level forward.
  – Cor: In undirected graph, go at most one level back
Application: Garbage Collection

• How do garbage collected languages work?
• Set up a graph:
  – Vertices = allocated memory blocks
  – Edges = Pointers
• Active vertices are reachable from a pointer in one of the environments in the current call stack
• Periodically, the runtime environment
  – Traverses the graph to mark all active blocks
  – Copies them to a fresh piece of memory
  – Frees up all unallocated blocks
• This is also how hard disk defragmentation works
Exploring a maze

- Vertices = intersections
- Edges = corridors
- Traverse to find the exit!
  - Use pebbles / chalk to mark where you’ve been
  - Typical strategy more like “DFS” than BFS

S. Raskhodnikova and A. Smith. Based on notes by E. Demaine and C. Leiserson