Chapter 3 - Graphs
Undirected Graphs

Undirected graph. $G = (V, E)$

- $V = \text{nodes}$.
- $E = \text{edges between pairs of nodes}$.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|, m = |E|$.

$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6\}$

$n = 8$

$m = 11$
Google maps

Transportation graph.
- **Nodes**: street addresses
- **Edges**: streets/highways
World Wide Web

Web graph.
- Nodes: web pages.
- Edges: hyperlinks.

http://golesystem.blogspot.com/2007/05/worldwide-web-as-seen-by-google.html
Social Networks

Social network graph.
- Node: people.
- Edge: relationship.


http://people.oii.ox.ac.uk/hogan/2010/01/new-pinwheel-network-layout/
A graph of blogosphere links

http://datamining.typepad.com/gallery/blog-map-gallery.html
## Additional Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>Network connections</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence</td>
</tr>
</tbody>
</table>
Graph Representation: Adjacency Matrix

**Adjacency matrix.** n-by-n matrix with $A_{uv} = 1$ if $(u, v)$ is an edge.
- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.
Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.
- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

degree = number of neighbors of $u$
Paths and Connectivity

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

**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 $v_1, v_2, ..., v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

cycle $C = 1-2-4-5-3-1$
Def. An undirected graph is a tree if it is connected and does not contain a cycle.

How many edges does a tree have?

Given a set of nodes, build a tree step wise
- every time you add an edge, you must add a new node to the growing tree, WHY?
- how many edges to connect n nodes?
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.

![Graphs showing rooted trees](image)

- A tree
- The same tree, rooted at 1

- $v$ is a child of $v$
- $v$ is a parent of $v$
Phylogenetic Trees

**Phylogeny.** Describes the evolutionary history of species.

![Phylogenetic Tree](http://www.whozoo.org/mammals/Carnivores/Cat_Phylogeny.htm)

(Redrawn after Johnson, et al, 2006)

http://www.whozoo.org/mammals/Carnivores/Cat_Phylogeny.htm
3.2 Graph Traversal
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?
Breadth First Search

**BFS intuition.** Explore outward from $s$, 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 $L_i$.

**Theorem.** For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
Breadth First Search

**Property.** Let $T$ be a BFS tree of $G$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
**BFS - implementation**

```python
bfs(v) :
    q - queue of nodes to be processed
    q.enqueue(v)
    mark v as explored
    while(q is non empty) :
        u = q.dequeue()
        for (each node v adjacent to u) :
            if v is unexplored :
                mark v as explored
                q.enqueue(v)
```

**Claim:** this implementation explores nodes in order of their appearance in BFS layers
Breadth First Search: Analysis

**Theorem.** The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

**Proof:**

- when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$
- total time processing edges is $\sum_{u \in V} \deg(u) = 2m$ \[\blacksquare\]

Each edge $(u, v)$ is counted exactly twice in sum: once in $\deg(u)$ and once in $\deg(v)$

```python
bfs(v):
    q - queue of nodes to be processed
    q.enqueue(v)
    mark v as explored
while (q is non empty):
    u = q.dequeue()
    for (each node v adjacent to u):
        if v is unexplored:
            mark v as explored
            q.enqueue(v)
```

```
**Connected Components**

*Connected graph.* There is a path between any pair of nodes.

*Connected component of a node s.* The set of all nodes reachable from s.

```
Connected component containing node 1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}.```

![Graph Diagram](image)
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.
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
Connected Components

Given two nodes $s$, and $t$, what can you say about their connected components?
**Connected Components**

A generic algorithm for finding connected components:

\[ R = \{ s \} \quad \text{# the connected component of } s \text{ is initially } s. \]

while there is an edge \((u,v)\) where \(u\) is in \(R\) and \(v\) is not in \(R\):

add \(v\) to \(R\)

**Theorem.** Upon termination, \(R\) is the connected component containing \(s\).

- **BFS:** explore in order of distance from \(s\).
- **DFS:** explore in a different way.
DFS: Depth First Search

Explores edges from the most recently discovered node; backtracks when reaching a dead-end.
DFS: Depth First Search

Explores edges from the most recently discovered node; backtracks when reaching a dead-end.

Recursively:

```
DFS(u):
    mark u as Explored and add u to R
    for each edge (u,v) :
        if v is not marked Explored :
            DFS(v)
```
DFS - nonrecursively

DFS(u):
   mark u as Explored and add u to R
   for each edge (u,v):
      if v is not marked Explored :
         DFS(v)

DFS(v):
   s – stack of nodes to be processed
   s.push(v)
   mark v as Explored
   while(s is non empty) :
      u = s.pop()
      for (each node v adjacent to u) :
         if v is not Explored :
            mark v as Explored
            s.push(v)
Theorem. The above implementation of DFS runs in $O(m + n)$ time if the graph is given by its adjacency list representation.

Proof:

Same as in BFS.

DFS(v) :

s - stack of nodes to be processed
s.push(v)
mark v as Explored
while(s is non empty) :
    u = s.pop()
    for (each node v adjacent to u) :
        if v is not Explored :
            mark v as Explored
            s.push(v)
3.4 Testing Bipartiteness
Def. An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.
- Scheduling: machines = red, jobs = blue.
Testing bipartiteness. Given a graph $G$, is it bipartite?

- Many graph problems become tractable if the underlying graph is bipartite (independent set)
- A graph is bipartite if it is 2-colorable

\[ \begin{array}{c}
    v_1 \\
    v_2 \\
    v_3 \\
    v_4 \\
    v_5 \\
    v_6 \\
    v_7 \\
\end{array} \]

\[ \begin{array}{c}
    v_1 \\
    v_2 \\
    v_3 \\
    v_4 \\
    v_5 \\
    v_6 \\
    v_7 \\
\end{array} \]

a bipartite graph $G$

another drawing of $G$
Algorithm for testing if a graph is bipartite

- Pick a node $s$ and color it blue
- Its neighbors must be colored red.
- Their neighbors must be colored blue.
- Proceed until the graph is colored.
- Check that there is no edge whose ends are the same color.
An Obstacle to Bipartiteness

Which of these graphs is 2-colorable?
An Obstacle to Bipartiteness

**Lemma.** If a graph $G$ is bipartite, it cannot contain an odd cycle.

**Proof.** Not possible to 2-color the odd cycle, let alone $G$. 

![Bipartite Graph](image1)

bipartite (2-colorable)

![Not Bipartite Graph](image2)

not bipartite (not 2-colorable)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer. $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer. $G$ contains an odd-length cycle (and hence is not bipartite).

Case (i) 

Case (ii)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer. $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer. $G$ contains an odd-length cycle (and hence is not bipartite).

Proof. (i)
- Suppose no edge joins two nodes in the same layer.
- I.e. all edges join nodes on adjacent layers.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

Case (i)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer. $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer. $G$ contains an odd-length cycle (and hence is not bipartite).

Proof. (ii)

- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) =$ lowest common ancestor.
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd.
Corollary. A graph $G$ is bipartite iff it contains no odd length cycle.
3.5 Connectivity in Directed Graphs
Directed Graphs

**Directed graph.** $G = (V, E)$
- Edge $(u, v)$ goes from node $u$ to node $v$.

**Example.** Web graph - hyperlink points from one web page to another.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Graph Search

**Directed reachability.** Given a node \( s \), find all nodes reachable from \( s \).

**Web crawler.** Start from web page \( s \). Find all web pages linked from \( s \), either directly or indirectly.

BFS and DFS extend naturally to directed graphs.

Given a path from \( s \) to \( t \), not guaranteed there is a path from \( t \) to \( s \).
Strong Connectivity

Def. Nodes u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Proof. ⇒ Follows from definition.

Proof. ⇐ Path from u to v: concatenate u-s path with s-v path.
Path from v to u: concatenate v-s path with s-u path. •
Strong Connectivity: Algorithm

Theorem. Can determine if $G$ is strongly connected in $O(m + n)$ time.

Proof.

- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{rev}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. □
3.6 DAGs and Topological Ordering
Graphs Describing Precedence

Examples:
- prerequisites for a set of courses
- dependencies between programs
- dependencies between jobs

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Want an ordering of the nodes that respects the precedence relation

- Example: An ordering of CS courses

The graph does not contain cycles. Why?
Directed Acyclic Graphs

Def. A Directed Acyclic Graph (DAG) is a directed graph that contains no directed cycles.

Def. A topological order of a directed graph $G$ is an ordering of its nodes as $v_1, v_2, ..., v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$. 

![a DAG](image1.png)  
![a topological ordering](image2.png)
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Proof. (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. □
Directed Acyclic Graphs

Lemma. If $G$ has a topological order, then $G$ is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?
Directed Acyclic Graphs

**Lemma.** If $G$ is a DAG, then $G$ has a node with no incoming edges.

**Proof. (by contradiction)**

- Suppose that $G$ is a DAG and every node has at least one incoming edge.
- Pick any node $v$, and begin following edges backward from $v$.
- Repeat. After $n + 1$ we will have visited a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Proof. (by induction on $n$)

- **Base case:** true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By induction hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; append nodes of $G - \{v\}$ in topological order.

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G - \{v\}$
and append this order after $v$
Topological Sort: Algorithm

Algorithm:

keep track of # incoming edges per node
while (nodes left):
    extract one with 0 incoming
    subtract one from all its adjacent nodes

Running time?
Better way?
Topological Sort: Algorithm Running Time

**Theorem.** Algorithm can be implemented to run in $O(m + n)$ time.

**Proof.**

- Maintain the following information:
  - $\text{count}[w] =$ remaining number of incoming edges
  - $S =$ set of nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - for each edge $(v, w)$: decrement $\text{count}[w]$ and add $w$ to $S$ if $\text{count}[w]$ hits 0
- this is $O(1)$ per edge