Undirected graphs

**Notation.** \( G = (V, E) \)

- \( V \) = nodes.
- \( E \) = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: \( n = |V|, m = |E| \).

\[
V = \{1, 2, 3, 4, 5, 6, 7, 8\}
\]
\[
E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6, 7-8\}
\]
\[
m = 11, n = 8
\]  

Some graph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>node</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephone, computer</td>
<td>fiber optic cable</td>
</tr>
<tr>
<td>circuit</td>
<td>gate, register, processor</td>
<td>wire</td>
</tr>
<tr>
<td>mechanical</td>
<td>joint</td>
<td>rod, beam, spring</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersection, airport</td>
<td>highway, airway route</td>
</tr>
<tr>
<td>internet</td>
<td>class C network</td>
<td>connection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>social relationship</td>
<td>person, actor</td>
<td>friendship, movie cast</td>
</tr>
<tr>
<td>neural network</td>
<td>neuron</td>
<td>synapse</td>
</tr>
<tr>
<td>protein network</td>
<td>protein</td>
<td>protein-protein interaction</td>
</tr>
<tr>
<td>molecule</td>
<td>atom</td>
<td>bond</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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Graph representation: adjacency lists

**Adjacency lists.** Node indexed array of lists.

- Two representations of each edge.
- Space is $\Theta(m + n)$.
- Checking if $(u, v)$ is an edge takes $O(\text{degree}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.
Paths and connectivity

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

![Path diagram]

Cycles

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

![Cycle diagram]

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

Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

**Importance.** Models hierarchical structure.

---

Tree on $n$ nodes has $n-1$ edges.
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 $t$?

Applications.

• Friendster.
• Maze traversal.
• Kevin Bacon number.
• Fewest number of hops in a communication network.

Breadth-first search

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

BFS algorithm.

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

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 $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then, the level of $x$ and $y$ differ by at most 1.

![BFS tree diagram](image)

(a) (b) (c)

Breadth-first search: analysis

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

**Pf.**

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{v \in V} \deg(u) = 2m$.  

  each edge $(u, v)$ is counted exactly twice
  in sum: once in $\deg(u)$ and once in $\deg(v)$
**Connected component**

**Connected component.** Find all nodes reachable from $s$.

![Graph Image]

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

---

**Connected component**

**Connected component.** Find all nodes reachable from $s$.

---

$R$ will consist of nodes to which $s$ has a path
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
    Add $v$ to $R$
Endwhile

---

**Theorem.** Upon termination, $R$ is the connected component containing $s$.

- BFS = explore in order of distance from $s$.
- DFS = explore in a different way.
Bipartite graphs

**Def.** An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored blue or white such that every edge has one white and one blue end.

**Applications.**
- Stable marriage: men = blue, women = white.
- Scheduling: machines = blue, jobs = white.

![A bipartite graph](image)

Testing bipartiteness

**Many graph problems become:**
- Easier if the underlying graph is bipartite (matching).
- Tractable if the underlying graph is bipartite (independent set).

Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

![A bipartite graph $G$](image)  
![Another drawing of $G$](image)
An obstruction to bipartiteness

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

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

Bipartite graphs

**Lemma.** Let $G$ be a connected graph, and let $L_0, \ldots, L_r$ 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, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

**Algorithm 2-Colour**

Given $G = (V,E)$ (connected)
Given $G = (V,E)$ (connected)

1. Run BFS on $G$
   getting layers $L_0, L_1, L_2, \ldots$

2. For every node $v \in V$
   - color $v = 0$ if $v$ is in even layer $L_{2i}$
   - color $v = 1$ if $v$ is in odd layer $L_{2i+1}$

3. Check if any edge $(u,v) \in E$ has $\text{Color}(u) = \text{Color}(v)$
   If so, output "Not Bipartite".
   Otherwise, output "Bipartite".

**Runtime Analysis**: $O(m+n)$

- (1) BFS: time $O(m+n)$
- (2) Assigning colors to nodes: $O(n)$
- (3) Checking for monochromatic edges: $O(m)$

**Correctness Analysis**:

(1) $G$ not bipartite $\implies$ Algo says "Non-bip".

(2) $G$ bipartite $\implies$ Algo says "Bipartite"

$G$ is not bipartite $\iff$ algo says "Non-bipartite".

We will argue that when the algo says
We will argue that when the algo says “Non-bipartite” on an input graph $G$, then $G$ contains an odd cycle. Hence, $G$ indeed is not bipartite.

Bipartite graphs

**Lemma.** Let $G$ be a connected graph, and let $L_0, ..., L_k$ 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, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

**Pf.** (i)

- Suppose no edge joins two nodes in same layer.
- By BFS property, each edge join two nodes in adjacent levels.
- Bipartition: white = nodes on odd levels, blue = nodes on even levels.

Case (i): no edges in same layer.

This algo 2-colors the graph unless it has an odd cycle.
Bipartite graphs

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

Pf. (ii)
- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_i$.
- Let $z = lca(x, y) =$ lowest common ancestor.
- Let $L_z$ 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. 

The only obstruction to bipartiteness

Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.
Directed graphs

Notation. \( G = (V, E) \).
- Edge \((u, v)\) leaves node \(u\) and enters node \(v\).

Ex. Web graph: hyperlink points from one web page to another.
- Orientation of edges is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

World wide web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another (orientation is crucial).
- Modern search engines exploit hyperlink structure to rank web pages by importance.
Some directed graph applications

<table>
<thead>
<tr>
<th>directed graph</th>
<th>node</th>
<th>directed edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>bank</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>

Graph search

**Directed reachability.** Given a node $s$, find all nodes reachable from $s$.

**Directed $s$-$t$ shortest path problem.** Given two nodes $s$ and $t$, what is the length of the shortest path from $s$ and $t$?

**Graph search.** BFS extends naturally to directed graphs.

**Web crawler.** Start from web page $s$. Find all web pages linked from $s$, either directly or indirectly.
Strong connectivity

**Def.** Nodes $u$ and $v$ are **mutually reachable** if there is a both 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.

**Pf.** $\Rightarrow$ Follows from definition.

**Pf.** $\Leftarrow$ Path from $u$ to $v$: concatenate $u \rightarrow s$ path with $s \rightarrow v$ path.
Path from $v$ to $u$: concatenate $v \rightarrow s$ path with $s \rightarrow u$ path.

Strong connectivity: algorithm

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

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^\text{reverse}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.
**Strong components**

**Def.** A **strong component** is a maximal subset of mutually reachable nodes.

![Graph with strong components highlighted]

**Theorem.** [1] In an **O(m + n)** time.

---

**Depth-First Search (DFS)**

\[
\text{DFS} (G = (V, E))
\]

\[
\begin{align*}
\text{for each } v \in V, & \text{ mark } v \text{ not explored} \\
\text{while } \text{not all } v \in V \text{ explored} & \\
\text{for each } v \in V & \\
\text{if } v \text{ is not explored} & \\
& \text{DFS-Visit} (v) \\
\text{end for}
\end{align*}
\]

**DFS-Visit (v)**

\[
\begin{align*}
\text{mark } v \text{ explored} \\
\text{for each edge } u - v & \\
\text{if } v \text{ not explored} & \\
& \text{DFS-Visit} (v)
\end{align*}
\]
for each node $v$
  if $v$ not explored
    then $\text{DFS-Visit}(v)$
end for

$\text{DFS}$

Ex

$\text{DFS}$

Observation:

$\text{DFS-Visit}(u)$

- $v$ explored
- $w$ explored
- exit $\text{DFS-Visit}(u)$

descendants of $u$ in $T$

$\text{DFS Tree Property}$:

$T$ $\text{DFS tree of } G = (V, E)$

$(x, y) \in E$ but not edge of $T$

Then one of $x, y$ is an ancestor
of the other in \( T \).

Proof: Say DFS(\( x \)) is called before y was discovered. Then y is discovered while still within the DFS(\( x \)) call. Hence, x may in the DFS tree.

Directed acyclic graphs

**Def.** A **DAG** is a directed graph that contains no directed cycles.

**Def.** A **topological order** of a directed graph \( G = (V, E) \) is an ordering of its nodes as \( v_1, v_2, \ldots, v_n \) so that for every edge \((v_i, v_j)\) we have \( i < j \).
**Precedence constraints.** Edge \((v_i, v_j)\) means task \(v_j\) must occur before \(v_j\).

**Applications.**
- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\).
- Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).

---

**Directed acyclic graphs**

**Lemma.** If \(G\) has a topological order, then \(G\) is a DAG.

**Pf.** [by contradiction]
- Suppose that \(G\) has a topological order \(v_1, v_2, \ldots, v_n\) and that \(G\) also has a directed cycle \(C\). Let’s see what happens.
- 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 \(v_1, v_2, \ldots, v_n\) 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 entering edges.

**Pf.** [by contradiction]

- Suppose that $G$ is a DAG and every node has at least one entering edge. Let’s see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one entering edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one entering edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.

![Diagram of a directed acyclic graph with a cycle highlighted in red.][diagram]

[diagram: A directed acyclic graph with nodes $w$, $x$, $u$, and $v$. The path $w ightarrow x ightarrow u ightarrow v$ is part of a cycle, indicated by a red dashed line.]}
Directed acyclic graphs

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

Pf. [by induction on $n$]
• Base case: true if $n = 1$.
• Given DAG on $n > 1$ nodes, find a node $v$ with no entering edges.
• $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
• By inductive hypothesis, $G - \{v\}$ has a topological ordering.
• Place $v$ first in topological ordering; then append nodes of $G - \{v\}$
  in topological order. This is valid since $v$ has no entering edges.  

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

Topological sorting algorithm: running time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.
Pf.
• Maintain the following information:
  - $count(w) =$ remaining number of incoming edges
  - $S =$ set of remaining nodes with no incoming edges
• Initialization: $O(m + n)$ via single scan through graph.
• Update: to delete $v$
  - remove $v$ from $S$
  - decrement $count(w)$ for all edges from $v$ to $w$;
    and add $w$ to $S$ if $count(w)$ hits 0
  - this is $O(1)$ per edge  

BFS & DFS Implementations
BFS: use queues

DFS: use stacks

Algo BFS: G = (V,E), s ∈ V

for each v ∈ V
  Explored[v] = False
end for

Explored[s] = True
L[0] = s
i = 0

while L[i] ≠ ∅
  L[i+1] = ∅
  for each u ∈ L[i]
    for each edge (u,v) ∈ E
      if Explored[v] = False
        then Explored[v] = True
          L[i+1] = L[i+1] + v
        end if
      end for
    end for
  end while
T = T + (u,v)
Algorithm XFS (s):

Stack \( S = \langle s \rangle \)

while \( S \neq \emptyset \)

\[ u = \text{pop}(S) \]

if \( \text{explored}(u) = \text{False} \)

\[ \text{explored}(u) = \text{True} \]

for each edge \( (u, v) \in E \)

\[ \text{push}(S, v); \text{parent}(v) = u \]

endfor

endif

endwhile