Chapter 3 - Graphs

**Undirected Graphs**

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

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

**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.

Examples

- Social network graph
  - Node: person
  - Edge: undirected relationship

- World Wide web
  - Node: web page
  - Edge: directed, link

- A graph of blogosphere links

- Google maps
  - Transportation graph
    - Nodes: street addresses
    - Edges: streets/highways
Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</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.** An n-by-n matrix with \( A_{uv} = 1 \) if (u, v) is an edge.
- For undirected graphs, each edge is represented twice.
- Space proportional to \( n^2 \).
- Checking if (u, v) is an edge takes \( \Theta(1) \) time.
- Identifying all outgoing edges from a node takes \( \Theta(n) \) time.

```
1 0 1 0 0 0 0 0
0 1 0 1 1 0 0 0
1 1 0 1 0 0 0 1
0 1 1 0 0 1 0 1
0 0 1 0 0 0 1 0
```

Graph Representation: Adjacency List

**Adjacency list.** Node indexed array of lists.
- For undirected graphs, each edge represented twice.
- Space proportional to \( m + n \).
- Checking if (u, v) is an edge takes \( O(deg(u)) \) time.
- Identifying all outgoing edges from a node takes \( O(deg(u)) \) time.
- Identifying all edges takes \( \Theta(m + n) \) time.

```
1
2 3
4 2
5 2
6
7 3
8 1
```

Which Implementation

Which implementation best supports common graph operations:
- Is there an edge between node i and node j?
- Find all nodes adjacent to node j

Which best uses space?

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 \( G \).

**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 \( v_1, v_2, \ldots, v_k, v_1 \) in which \( v_1 = v_k \), \( k > 2 \), and the first \( k-1 \) nodes are distinct.

Example cycle: 1-2-4-5-3-1
Trees

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

How many edges does a tree with $n$ nodes have?

Rooted Trees

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

Models hierarchical structure.

Phylogenetic Trees

Phylogeny. Describes the evolutionary history of species.

http://www.whozoo.org/mammals/Carnivores/Cat_Phylogeny.htm

Traversing a Rooted Binary Tree

Pre order
- visit the node
- go left
- go right

In order
- go left
- visit the node
- go right

Post order
- go left
- go right
- visit the node

Level order / breadth first
- for $d = 0$ to height
- visit nodes at level $d$
**Traversal Examples**

- **Pre order**: ABDGHCFEI
- **In order**: GDHBAECFI
- **Post order**: GHDEIFCA
- **Level order**: ABCDEFGHI

**Traversal Implementation**

- **Recursive implementation of preorder**
  - The steps:
    - Visit node
    - Preorder(left child)
    - Preorder(right child)
  - For in-order, post-order just change the order of function calls

  How would you implement level order?

---

**3.2 Graph Traversal**

**Connectivity**

- **s-t connectivity problem**: Given two nodes s and t, is there a path between s and t?
- **s-t shortest path problem**: Given two nodes s and t, what is the length of the shortest path between s and t?

**Graph Traversal**

- What makes it different from rooted tree traversal?

  What makes it different from rooted tree traversal:
  - Cycles - more than one way to get to a node
  - What to do about it?
Graph Traversal

What makes it different from rooted tree traversal:
- cycles

What to do about it?
- mark the nodes

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.

BFS - implementation

Claim: this implementation explores nodes in order of their appearance in BFS layers

BFS - Example

How would you compute distances using this implementation?
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 \(\text{deg}(u)\) incident edges \((u, v)\)
- total time processing edges is \(\sum_{u \in V} \text{deg}(u) = 2m\)
  - each edge \((u, v)\) is counted exactly twice in sum: once in \(\text{deg}(u)\) and once in \(\text{deg}(v)\)

```
bfs(v):
  q = queue of nodes to be processed
  mark v as explored
  q.enqueue(v)
  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.

Example:
- Connected component of node 1: \{1, 2, 3, 4, 5, 6, 7, 8\}
- Connected component of node 2: \{2, 1, 7\}
- Connected component of node 3: \{3\}
- Connected component of node 4: \{4\}
- Connected component of node 5: \{5\}
- Connected component of node 6: \{6\}
- Connected component of node 7: \{7\}
- Connected component of node 8: \{8\}

Connected Components

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

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

The facebook graph

- 721 million active accounts
- 68.7 billion friendship edges (median number of friends = 99)
- The largest connected component of facebook users contains 99.9% of the users
- Average distance between any pair of users: 4.7


Connected components using BFS

How will you use BFS to find the connected component of a node s?

How will you use BFS to find all the connected components of a graph?
Connected Components
A generic algorithm for finding connected components:

Upon termination, R is the connected component containing s.
- BFS: explore in order of distance from s.
- DFS: explores edges from the most recently discovered node; backtracks when reaching a dead-end.

DFS: Depth First Search

DFS - Example

DFS - Analysis

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.
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 end and one blue end.

Applications.
- Scheduling: machines = red, jobs = blue.

Algorithm for testing if a graph is bipartite
1. Pick a node $s$ and color it blue.
2. Its neighbors must be colored red.
3. Their neighbors must be colored blue.
4. Proceed until the graph is colored.
5. Check that there is no edge whose ends are the same color.

Testing Bipartiteness

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

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 $G$. 

Which of these graphs is 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).

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.

Proof. (ii)
- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) = \text{lowest common ancestor}$.
- 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.

3.5 Connectivity in 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. But:
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.

Directed Acyclic Graphs

Def. A Directed Acyclic Graph (DAG) is a directed graph that contains no directed cycles.
Def. A topological order (also called a topological sort) of a directed graph G is an ordering of its nodes as v1, v2, ..., vn so that for every edge (vi, vj) we have i < j.

3.6 DAGs and Topological Ordering

Is the topological order unique?

Examples: Graphs Describing Precedence
- prerequisites for a set of courses
- dependencies between programs
- dependencies between jobs
- order of putting your clothes on

Precedence constraints. Edge (vi, vj) means task vi must occur before vj.
Topological ordering: an ordering of the nodes that respects the precedence relation
- Example: An ordering of CS courses
- Graphs describing precedence must not contain cycles.

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

Proof. (by contradiction)

- Suppose that $G$ has a topological order $v_1, ..., v_n$ 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 $v_1, ..., v_n$ is a topological order, we must have $j < i$, a contradiction. □

Directed Acyclic Graphs

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

Why is this interesting?

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 $n + 1$ steps 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 - contradiction. □
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 a DAG with $n > 1$ 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.

Topological Ordering: 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 Ordering: Algorithm Running Time

Theorem. Algorithm can 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: pick a node $v$ in $S$
  - 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

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 $v$