3.1 Basic Definitions and Applications

Undirected Graphs

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

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

Some 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>fiber-optic cables</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>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.

node: cnn.com
node: cnet.com
node: time Warner.com
node: hbo.com
node: sorpanos.com

9-11 Terrorist Network

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

Ecological Food Web

Food web graph.
- Node: species.
- Edge: from prey to predator.

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.

Paths and Connectivity

- **Def.** A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, ..., v_k$, 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 $v_1, v_2, ..., v_k$, in which $v_1 = v_k, k > 2$, and the first $k-1$ nodes are all distinct.

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.

![Diagram of a rooted tree]

- $v$: vertex
- $parent$ of $v$: parent of vertex $v$
- $child$ of $v$: child of vertex $v$

**Phylogeny Trees**

*Phylogeny trees.* Describe evolutionary history of species.

![Phylogenetic tree diagram]

**GUI Containment Hierarchy**

*GUI containment hierarchy.* Describe organization of GUI widgets.


**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?

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.**
- $L_0 = \{s\}$
- $L_1$ = 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: 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 $\sum_{u \in V} \deg(u) = 2m$

  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 $s$.

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

**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.

**Depth-First Search: The Code**

```
DFS(G)
{
  for each vertex $u \in G->V$
  {
    Mark $v$ unexplored ;
  }
  time = 0;
  for each vertex $u \in G->V$
  {
    if ($u$ is UNEXPLORED)
    {
      DFS_Visit($u$);
    }
  }
}
```

```
DFS_Visit($u$)
{
  Mark $u$ EXPLORED;
  add $u$ to R;
  for each $v \in u->Adj[]$
  {
    if ($v$ is NOT_EXPLORED)
    {
      DFS_Visit($v$);
    }
  }
}
Depth-First Search: The Code

What will be the running time of DFS?

\[
\text{ DFS(G)} \\
\begin{array}{l}
\text{for each vertex } u \in G \rightarrow V \\
\quad \text{Mark } v \text{ unexplored ; }
\end{array}
\]

time = 0;
for each vertex \( u \in G \rightarrow V \)
\[
\begin{array}{l}
\quad \text{if } (u \text{ is UNEXPLORED}) \\
\quad \quad \text{DFS_Visit}(u);
\end{array}
\]

\[
\text{DFS_Visit}(u) \\
\begin{array}{l}
\quad \text{Mark } u \text{ EXPLORED;} \\
\quad \quad \text{add } u \text{ to } R;
\end{array}
\]

for each \( v \in u \rightarrow \text{Adj}[] \)
\[
\begin{array}{l}
\quad \text{if } (v \text{ is NOT_EXPLORED}) \\
\quad \quad \text{DFS_Visit}(v);
\end{array}
\]

Running time: There is a tighter bound \(O(V+E)\) or \(O(m+n)\)

\[n = |V| \text{ and } m = |E|\]

Depth-First Search: The Code

Running time: \(O(n^2)\) because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as \(|V|\) times

\[
\text{ DFS(G)} \\
\begin{array}{l}
\text{for each vertex } u \in G \rightarrow V \\
\quad \text{Mark } v \text{ unexplored ; }
\end{array}
\]

time = 0;
for each vertex \( u \in G \rightarrow V \)
\[
\begin{array}{l}
\quad \text{if } (u \text{ is UNEXPLORED}) \\
\quad \quad \text{DFS_Visit}(u);
\end{array}
\]

\[
\text{DFS_Visit}(u) \\
\begin{array}{l}
\quad \text{Mark } u \text{ EXPLORED;} \\
\quad \quad \text{add } u \text{ to } R;
\end{array}
\]

for each \( v \in u \rightarrow \text{Adj}[] \)
\[
\begin{array}{l}
\quad \text{if } (v \text{ is NOT_EXPLORED}) \\
\quad \quad \text{DFS_Visit}(v);
\end{array}
\]

Connected Component

Connected component. Find all nodes reachable from \( s \).

The nodes can be reached by BFS, DFS

\( R \) 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 \( R \)

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

- BFS = explore in order of distance from \( s \).
- DFS = explore in a different way.
- For any two node \( s \) and \( t \) their connected components are either identical or disjoint.
Adjacency Matrix vs Lists

\[ n = |V| \text{ and } m = |E| \]

Running time depends on the relationship between \( n \) and \( m \)

- Dense graph \( \left( \frac{n}{2} \right) \approx n^2 \text{ edges} \)
- Connected graph at least \( m \geq n - 1 \)

Space and time considerations depend on the algorithm

Queues vs Stacks

\[ n = |V| \text{ and } m = |E| \]

- Important to maintain an order which elements are visited
- Implemented as doubly linked lists
- FIFO - first in first out QUEUE
- LIFO - last in first out STACK
- BSF Implementation (QUEUE or STACK)
- DFS Implementation (STACK)
- Finding number of Connected Components

Look at the pseudo-code for DFS and BSF in the book

Queues vs Stacks

\[ n = |V| \text{ and } m = |E| \]

- Important to maintain an order which elements are visited
- Implemented as doubly linked lists
- FIFO - first in first out QUEUE
- LIFO - last in first out STACK
- BSF Implementation (QUEUE or STACK)
- DFS Implementation (STACK)
- Finding number of Connected Components

Look at the pseudo-code for DFS and BSF in the book
3.4 Testing Bipartiteness

Definition. 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.
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

Testing Bipartiteness

Testing bipartiteness. Given a graph $G$, is it bipartite?
- 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.

An Obstruction to Bipartiteness

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

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

Bipartite Graphs

A bipartite graph $G$

Another drawing of $G$
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.**

(i) Suppose no edge joins two nodes in the same layer. In BFS either edge joins nodes in the same layer or adjacent layers, since the assumption is (i) then all nodes are in adjacent layers.

(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. □

**Corollary.** A graph $G$ is bipartite iff it contains no odd length cycle.

Obstruction to Bipartiteness

- bipartite (2-colorable)
- not bipartite (not 2-colorable)
3.5 Connectivity in Directed Graphs

Directed Graphs

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

Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- 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 \( s \).
Directed \( s \)-to-\( t \) shortest path problem. Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) 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. Node \( u \) 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.

Pf. \( \Rightarrow \) Follows from definition.

Pf. \( \Leftarrow \) Path from \( u \) to \( v \): concatenate \( u \)-to-\( s \) path with \( s \)-to-\( v \) path.
Path from \( v \) to \( u \): concatenate \( v \)-to-\( s \) path with \( s \)-to-\( u \) path. 

\( \text{ok if paths overlap} \)
**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 \( G^{\text{rev}} \).
- Return true if all nodes reached in both BFS executions.

Correctness follows immediately from previous lemma.

**Example 1 (yes)**

**Example 2 (no)**

---

**3.6 DAGs and Topological Ordering**

**Directed Acyclic Graphs**

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

**Example.** Precedence constraints: edge \( (v_i, v_j) \) means \( v_i \) must precede \( v_j \).

**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**

**Precedence constraints.** Edge \( (v_i, v_j) \) means task \( v_i \) 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 $v_1, \ldots, v_n$, 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 $v_1, \ldots, v_n$ 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.

Pf. (by contradiction)

- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let’s see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming 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. □

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 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 incoming 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 incoming edges. □

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 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.