Graphs

- A (simple) graph \( G = (V, E) \) consists of
  - \( V \), a nonempty set of vertices and
  - \( E \), a set of unordered pairs of distinct vertices called edges.

Examples

\[
V = \{A, B, C, D, E\}
\]
\[
E = \{ (A, D), (A, E), (B, D), (B, E), (C, D), (C, E) \}
\]
Directed Graphs

- A directed graph (or digraph) $G = (V, E)$ consists of
  - $V$, a nonempty set of vertices and
  - $E$, a set of ordered pairs of distinct vertices called edges.

- Examples
Multigraphs

- A multigraph (directed multigraph) $G = (V, E)$ consists of
  - $V$, a set of vertices,
  - $E$, a set of edges, and
  - a function $f$ from $E$ to $\{\{u, v\} : u \neq v \in V\}$

- Two edges $e_1$ and $e_2$ with $f(e_1) = f(e_2)$ are called multiple edges.

- Put simply, a multigraph $G = (V, E)$ is a graph in which multiple edges are allowed.

- Examples
Weighted Graphs

- A **weighted graph** is a graph (or digraph) with the additional property that each edge \( e \) has associated with it a real number \( w(e) \) called its **weight**.

- A weighted digraph is often called a **network**.

- **Examples**
Psuedographs

- A psuedograph $G = (V, E)$ consists of
  - $V$, a set of vertices,
  - $E$ a set of edges, and
  - a function $f$ from $E$ to $\{\{u, v\} : u, v \in V\}$.

- Psuedo-multigraphs are defined similarly.

- Put another way, a psuedograph is a graph in which we allow loops, that is, edges from a vertex to itself.

Examples
Graph Definitions Summary

- There are several ways to categorize graphs:
  - Directed or undirected edges.
  - Weighted or unweighted edges.
  - Allow multiple edges or not.
  - Allow loops or not.

- Unless specified, you can usually assume a graph does not allow multiple edges and loops. These aren’t that common.

- For clarity, if a graph is not specified as weighted or directed, assume it isn’t.

- The most common graphs we’ll use are graphs, digraphs, weighted graphs, and networks.

- When writing graph algorithms, it is important to know what characteristics the graphs have. For instance, if a graph might have loops, the algorithm should be able to handle it.

What will determine the type of graph we'll see?
Graph Terminology

Let $u$ and $v$ be vertices, and let $e = \{u, v\}$ be an edge in an undirected graph $G$.

- The vertices $u$ and $v$ are said to be **adjacent**
- The edge $e$ is said to be **incident** to $u$ and $v$.
- The edge $e$ is said to **connect** $u$ and $v$.
- The vertices $u$ and $v$ are called the **endpoints** of the edge $e$.
- The **degree** of a vertex, denoted $deg(v)$, in an undirected graph is the number of edges incident to it (where loops are counted twice).

$G_1$, $G_2$, $G_3$
Examples

$G_1$

$G_2$

$G_3$

<table>
<thead>
<tr>
<th>$G_1$</th>
<th>$G_2$</th>
<th>$G_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>deg(u)=3</td>
<td>deg(u)=2</td>
<td>deg(u)=2</td>
</tr>
<tr>
<td>deg(v)=5</td>
<td>deg(v)=3</td>
<td>deg(v)=4</td>
</tr>
<tr>
<td>deg(w)=3</td>
<td>deg(w)=2</td>
<td>deg(w)=3</td>
</tr>
<tr>
<td>deg(x)=2</td>
<td>deg(x)=4</td>
<td>deg(x)=2</td>
</tr>
<tr>
<td>deg(y)=2</td>
<td>deg(y)=3</td>
<td>deg(y)=3</td>
</tr>
<tr>
<td>deg(z)=3</td>
<td></td>
<td>deg(z)=2</td>
</tr>
</tbody>
</table>
More Graph Terminology

- A **subgraph** of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

- A **path** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that consecutive vertices $v_i$ and $v_{i+1}$ are adjacent.
More Graph Terminology

- A **simple path** is a path with no repeated vertices.

- A **cycle** is a simple path whose last vertex is the same as the first vertex.
More Graph Terminology

• A graph is called **connected** if there is a path between every pair of distinct vertices.

![Diagram showing connected and not connected graphs]

• A **connected component** of a graph is a maximal connected subgraph. e.g. the graph below has 3 connected components.

![Diagram showing three connected components]
Trees

- A tree (or unrooted tree, or free tree) is a connected acyclic graph. That is, a graph with no cycles.
- A forest is a collection of trees.

- These trees are not to be confused with rooted trees.
Spanning Tree

- A **spanning tree** of $G$ is a subgraph which is a tree and contains all of the vertices of $G$. 

![Graph G and its spanning tree](image)
Some Special Graphs

- $K_n$: The complete graph on $n$ vertices.

- $C_n$: The cycle of length $n$. 
- $Q_n$: The $n$-cube.

\[ Q_1 \quad Q_2 \quad Q_3 \quad Q_4 \]
• **Bipartite Graphs:** A simple graph $G$ is called bipartite if the vertex set $V$ can be partitioned into two disjoint nonempty sets $V_1$ and $V_2$ such that every edge connects a vertex in $V_1$ to a vertex in $V_2$.

Put another way, no edges in $V_1$ are connected to each other, and no edges in $V_2$ are connected to each other.
Some Theorems

- **Theorem 1**: Let $G = (V, E)$ be an undirected graph with $|E|$ edges. Then

$$2 |E| = \sum_{v \in V} \deg(v).$$

- **Proof**: Let $X = \{(e, v) : e \in E, v \in V, e \text{ and } v \text{ are incident}\}$. We will compute $|X|$ in two ways. Each edge $e \in E$, is incident with exactly 2 vertices. Thus,

$$|X| = 2 |E|$$

Also, each vertex $v \in V$ is incident with $\deg(v)$ edges. Thus, we have that

$$|X| = \sum_{v \in V} \deg(v).$$

Setting these equal, we have the result.

- **Corollary 2**: An undirected graph has an even number of vertices of odd degree.
Examples

<table>
<thead>
<tr>
<th>Graph</th>
<th>$G_1$</th>
<th>$G_2$</th>
<th>$G_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>E</td>
<td>$</td>
<td>9</td>
</tr>
<tr>
<td>$\sum_{v \in V} deg(v)$</td>
<td>18</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
Directed Graph Terminology

Let \( u \) and \( v \) be vertices in a directed graph \( G \), and let \( e = (u, v) \) be an edge in \( G \).

- \( u \) is said to be adjacent to \( v \).
- \( v \) is said to be adjacent from \( u \).
- \( u \) is called the initial vertex of \((u, v)\).
- \( v \) is called the terminal or end vertex of \((u, v)\).
- The in-degree of \( u \), denoted by \( \text{deg}^- (u) \), is the number of edges in \( G \) which have \( u \) as their terminal vertex.
- The out-degree of \( u \), denoted by \( \text{deg}^+ (u) \), is the number of edges in \( G \) which have \( u \) as their initial vertex.

- **Theorem 3**: Let \( G = (V, E) \) be a directed graph. Then
  \[
  \sum_{v \in V} \text{deg}^- (v) = \sum_{v \in V} \text{deg}^+ (v) = |E|.
  \]
Examples

\[ G_4 \]

\[ G_5 \]

\[ G_6 \]

<table>
<thead>
<tr>
<th></th>
<th>( G_4 )</th>
<th>( G_5 )</th>
<th>( G_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \deg^{-}(u) = 2 )</td>
<td>( \deg^{+}(u) = 4 )</td>
<td>( \deg^{-}(u) = 1 )</td>
<td>( \deg^{+}(u) = 1 )</td>
</tr>
<tr>
<td>( \deg^{-}(v) = 2 )</td>
<td>( \deg^{+}(v) = 2 )</td>
<td>( \deg^{-}(v) = 1 )</td>
<td>( \deg^{+}(v) = 2 )</td>
</tr>
<tr>
<td>( \deg^{-}(w) = 1 )</td>
<td>( \deg^{+}(w) = 1 )</td>
<td>( \deg^{-}(w) = 1 )</td>
<td>( \deg^{+}(w) = 2 )</td>
</tr>
<tr>
<td>( \deg^{-}(x) = 2 )</td>
<td>( \deg^{+}(x) = 3 )</td>
<td>( \deg^{-}(x) = 1 )</td>
<td>( \deg^{+}(x) = 1 )</td>
</tr>
<tr>
<td>( \deg^{-}(y) = 3 )</td>
<td>( \deg^{+}(y) = 0 )</td>
<td>( \deg^{-}(y) = 2 )</td>
<td>( \deg^{+}(y) = 2 )</td>
</tr>
<tr>
<td>( \deg^{-}(z) = 1 )</td>
<td>( \deg^{+}(z) = 1 )</td>
<td>( \deg^{-}(z) = 2 )</td>
<td>( \deg^{+}(z) = 2 )</td>
</tr>
</tbody>
</table>
Graph Representation

There are two common ways of representing $G$. Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges.

**Adjacency Lists**

- For each vertex $v$ in $G$, we store a list of vertices adjacent to $v$.
- This is often implemented using linked lists.

![Diagram of graph representation](image)

- For weighted graphs, an additional field can be stored in each node.
- The space required for storage is
  \[ \Theta(n + 2m) = \Theta(n + m) \]
  for graphs, and
  \[ \Theta(n + m) = \Theta(n + m) \]
  for digraphs.

*Why the constant 2 in graphs and not digraphs?*
Adjacency Matrix

- Number the vertices 1, 2, \ldots, n in some arbitrary order.
- We use a n by n matrix $M$ defined as
  
  $M(i, j) = \begin{cases} 
  1 & \text{if } (i, j) \text{ is an edge} \\
  0 & \text{if } (i, j) \text{ is not an edge}
  \end{cases}$

- If $G$ is weighted, we store the weights in the matrix. For non-adjacent vertices, we store $\infty$, or $\text{MAX\_INT}$.

- It is clear that this representation requires $\Theta(n^2)$ space.

Has some nice uses via matrix multiplication
Some Basic Graph Problems

- Is there a path from A to B?
- CYCLES: Does the graph contain a cycle?
- CONNECTIVITY (SPANNING TREE): Is there a way to connect the vertices?
- BICONNECTIVITY: Will the graph become disconnected if one vertex is removed?
- PLANARITY: Is there a way to draw the graph without edges crossing?
- SHORTEST PATH: What is the shortest way from A to B (weighted or unweighted)?
- LONGEST PATH: What is the longest way from A to B (weighted or unweighted)?
- MINIMAL SPANNING TREE: What is the best way to connect the vertices?
- TRAVELING SALESMAN: What is the shortest route to connect the vertices without visiting the same vertex twice?
(Rooted) Trees

- A **rooted tree** is a tree which has a specially designated vertex called the *root*.

- In rooted trees, vertices are called *nodes*.

- Each node contains some information and one or more links to other nodes further down the hierarchy. (Similar to nodes in a linked list.)

- For convenience, we can think of trees as acyclic digraphs in which every edge “points away from” the root.
Rooted Trees Terminology

- A node that is adjacent from $v$ is a child of $v$.
- A node that is adjacent to $v$ is a parent of $v$.
- Two nodes who have the same parent are siblings.
- A leaf or external node is a node with degree zero. (i.e. a node with no children)
- An internal node is a nonleaf node. (i.e. a node with at least one child)
- An ancestor of a node is any node on the path from the root to the node.
- A descendant of a node $v$ is any node which has $v$ as an ancestor.
Rooted Trees Terminology

- The **degree of a node** is the number of children the node has.
- The **depth** of a node is the length of the path from the root to the node.
- The **height** of a node is the maximum length of a path from the node to a leaf.
- The **height or depth** of a tree is the maximum height of any node in the tree.
- The **subtree rooted at** $x$ is the subtree consisting of $x$ and all of its descendents.
Binary Tree

- A **binary tree** is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the *left subtree* and the *right subtree*

- Put another way, a **binary tree** is a rooted tree such that each node has
  - no children,
  - a *right child*,
  - a *left child*, or
  - both a *right child* and a *left child*.

![Binary Tree Diagram](Image)
Binary Trees: Definitions

- A **full binary tree** is one in which internal nodes completely fill every level, except possibly the last.

- A **complete binary tree** is a full binary tree where the internal nodes on the bottom level all appear to the left of the external nodes on that level.

- **Example:** A full binary tree
Binary Tree Examples

- **Example:** A complete binary tree

```
1
 /   \
2     3
 / \
4   5 / \
8   9 10 11 12
```

- **Example:** A totally complete binary tree

```
1
 /   \
2     3
 / \
4   5 / \
8   9 10 11 12 13 14 15
```
Properties of Binary Trees

• The maximum number of nodes on level $i$ of a binary tree is $2^i$, $i \geq 1$.

• The maximum number of nodes in a binary tree of depth $k$ is $2^{k+1} - 1$, $k \geq 1$.

• There is exactly one path connecting any two nodes in a tree.

• A tree with $n$ nodes has $n - 1$ edges.

• The height of a full binary tree with $n$ internal nodes is about $\log_2 n$. 
Binary Tree Representation: Arrays

Let \( G \) be a tree of height \( \log_2 n \) with \( m \) nodes, where \( m \leq n \). We can represent \( G \) with an array \( A \) of size \( n \). \( A[1] \) is the root, and given a node with index \( i \), we can find the index of parents and children as follows:

- \( \text{parent}(i) = \begin{cases} \lfloor i/2 \rfloor & \text{if } i \neq 1 \\ \text{undefined} & \text{if } i = 1 \end{cases} \)
- \( \text{left}(i) = \begin{cases} 2i & \text{if } 2i \leq n \\ \text{undefined} & \text{if } 2i > n \end{cases} \)
- \( \text{right}(i) = \begin{cases} 2i + 1 & \text{if } 2i + 1 \leq n \\ \text{undefined} & \text{if } 2i + 1 > n \end{cases} \)
Binary Tree Representation: Linked Lists

- We define a tree node by
  ```
  struct treenode{
    int data;
    treenode *left_child;
    treenode *right_child;
  };
  ```

- We find children by following the pointers.

- Parents are harder to find, unless we use doubly linked lists.
Binary Tree Traversals

• When we visit each node in the tree exactly once, we say we have **Traversed** the tree.

• A full traversal produces a linear order of the information in a tree.

• There are several ways to traverse a tree.
  – **preorder**: visit a node, then traverse its left subtree, and then traverse its right subtree.
  – **inorder**: traverse the left subtree, visit the node and then traverse its right subtree
  – **postorder**: first traverse the left subtree, traverse the right subtree, and then visit the node.

• There is a natural correspondence between these traversals and producing the prefix, infix and postfix form of an expression.
Tree Traversal

- **preorder**: visit a node, then traverse its left subtree, and then traverse its right subtree.

- **inorder**: traverse the left subtree, visit the node and then traverse its right subtree.

- **postorder**: first traverse the left subtree, traverse the right subtree.

Preorder: + * * / A B C D E

Inorder: A / B * C * D + E

Postorder: A B / C * D * E +
Tree Traversal: Implementation

• Assume we have used a linked list to implement a tree.

```
struct tree_node{
    int data;
    tree_node *left_child;
    tree_node *right_child;
};
```

• Assume we have a pointer to the root node.

• From this, we can traverse the tree with any of the methods.
**Preorder Traversal**

Visit a node, then traverse its left subtree, and then traverse its right subtree.

```cpp
void PreOrderTraversal(treenode *ptr) {
    if (ptr!=0) {
        cout << ptr->data;
        PreOrderTraversal(ptr->left);
        PreOrderTraversal(ptr->right);
    }
}
```
Postorder Traversal

First traverse the left subtree, traverse the right subtree, and finally visit the node.

```c
void PostOrderTraversal(treenode *ptr) {
    if (ptr!=0) {
        PostOrderTraversal(ptr->left);
        PostOrderTraversal(ptr->right);
        cout << ptr->data;
    }
}
```
Inorder Traversal

Traverse the left subtree, visit the node and then traverse its right subtree.

```c
void InOrderTraversal(treenode *ptr) {
    if (ptr!=0) {
        InOrderTraversal(ptr->left);
        cout << ptr->data;
        InOrderTraversal(ptr->right);
    }
}
```