Graphs specify pairwise relationships between objects.

An undirected graph $G = (V, E)$ is a pair of sets:

- $V$ is a set of nodes, aka vertices.
- $E$ is a set of two-elements subset of $V$.
  An element of $E$ is of the form: $e = \{u, v\}$ with $u, v \in V$.

A graph is directed if $E$ is a set of ordered pairs $(u, v)$, $u, v \in V$. 
• **Simple**: at most one edge between every pair of vertices.
• **Complete**: all possible edges are present (denoted: $K_n$)
• **Degree** of a vertex: number of incident edges.
• **Self-loop**: an edge $\{u, u\}$ — often disallowed.
• **Path**: a sequence $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $i = 1 \ldots, k$.
• **Simple path**: a path with no vertex repeated.
• **Closed path**: a path with $v_1 = v_k$. This path is a **cycle** if it is simple.
• A graph is **connected** if there is a path between any pair of vertices.
• **Digraph**: short name for directed graph.
• A graph $G = (V_G, E_G)$ is a subgraph of $H = (V_H, E_H)$ if $V_G \subseteq V_H$ and $E_G \subseteq E_H$.

• (Connected) Component: a maximal connected subgraph.

• Cut-edge (aka bridge): an edge whose removal increases the number of connected components
Graph Modeling Examples

Graphs model *many* concepts naturally:

1. Social networks
2. Geographic adjacency
3. Polyhedra
4. Chemical Molecules
5. Assigning jobs to applicants
6. Food webs
7. Finite-state machines
8. Markov processes
9. Project dependencies
10. World Wide Web
11. Telephone network
12. Roads
Graphs model many concepts naturally:

1. Social networks
2. Geographic adjacency
3. Polyhedra
4. Chemical Molecules
5. Assigning jobs to applicants ← last week
6. Food webs
7. Finite-state machines
8. Markov processes
9. Project dependencies ← today
10. World Wide Web
11. Telephone network
12. Roads
Graph Isomorphism

When are two graphs “the same”? Often, we care only about the structure of the graph, and not any vertex labels.

**Definition (Graph isomorphism)**

Graphs $G$ and $H$ are **isomorphic** if there is a bijection $f : V_G \rightarrow V_H$ such that

\[
\{u, v\} \in E_G \iff \{f(u), f(v)\} \in E_H.
\]

What’s an algorithm to test whether two graphs are isomorphic?
When are two graphs “the same”? Often, we care only about the structure of the graph, and not any vertex labels.

**Definition (Graph isomorphism)**

Graphs $G$ and $H$ are **isomorphic** if there is a bijection $f : V_G \rightarrow V_H$ such that

$$\{u, v\} \in E_G \iff \{f(u), f(v)\} \in E_H.$$ 

What’s an algorithm to test whether two graphs are isomorphic?

No one knows a really good algorithm. This problem is not known to be NP-complete (unlike Independent Set).
Open Problems (Already!)

1. Is there a polynomial time algorithm for Independent Set?

2. Is there a polynomial time algorithm for Graph Isomorphism?
Representing Graphs

Adjacency matrix:

$$
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0
\end{pmatrix}
$$

Adjacency list:

1 → 2 → 3
2 → 1 → 5
3 → 1 → 4 → 5
4 → 3
5 → 2 → 3
Trees are a special type of graph that occur often in algorithms.

**Definition (Tree)**

A graph $G$ is a tree if it is connected and contains no cycles.
Theorem (Characterization of Trees)

The following statements are equivalent:

1. $T$ is a tree.
2. $T$ contains no cycles and $n - 1$ edges.
3. $T$ is connected and has $n - 1$ edges.
4. $T$ is connected and every edge is a cut-edge.
5. Any two nodes in $T$ are connected by exactly 1 path.
6. $T$ is acyclic, and adding any new edge creates exactly one cycle.
Tree Terminology

- A tree is **rooted** if it has a distinguished vertex called the root.
- A tree is **ordered** if it is a rooted tree where the children are assigned an order.
- In a **binary** tree, each node has at most 2 children.
- In a **m-ary** tree, each node has at most \( m \) children.
- A **complete tree** is a \( m \)-ary tree for which each node has \( m \) children and all leaves are at the same level.
A binary search tree is a binary tree where

1. a key $k(u)$ is associated with each node $u$, and
2. the keys in the subtree rooted at $\text{left}(u)$ are all $\leq k(u)$
3. the keys in the subtree rooted at $\text{right}(u)$ are all $> k(u)$.

Balanced binary search trees try to avoid long, stringy trees so that when searching we can eliminate about half the remaining elements at each step.

E.g. AVL trees let us find any element in $O(\log n)$ time, and we can efficiently update them.
Graph Traversals
Breadth-First Search

Breadth-first search explores the nodes of a graph in increasing distance away from some starting vertex \( s \).

It decomposes the component into layers \( L_i \) such that the shortest path from \( s \) to each of nodes in \( L_i \) is of length \( i \).

Breadth-First Search:

1. \( L_0 \) is the set \( \{ s \} \).
2. Given layers \( L_0, L_1, \ldots, L_j \), then \( L_{j+1} \) is the set of nodes that are not in a previous layer and that have an edge to some node in layer \( L_j \).
A BFS traversal of a graph results in a breadth-first search tree:

Can we say anything about the non-tree edges?
A BFS traversal of a graph results in a breadth-first search tree:

Can we say anything about the non-tree edges?
Theorem

Choose \( x \in L_i \) and \( y \in L_j \) such that \( \{x, y\} \) is an edge in undirected graph \( G \). Then \( i \) and \( j \) differ by at most 1.

In other words, edges of \( G \) that do not appear in the tree connect nodes either in the same layer or adjacent layer.

Proof.

Suppose not, and that \( i < j - 1 \).

All the neighbors of \( x \) will be found by layer \( i + 1 \).

Therefore, the layer of \( y \) is less than \( i + 1 \), so \( j \leq i + 1 \), which contradicts \( i < j - 1 \).
Depth-First Search

DFS keeps walking down a path until it is forced to backtrack. It backtracks until it finds a new path to go down.

Think: Solving a maze.

It results in a search tree, called the depth-first search tree. In general, the DFS tree will be very different than the BFS tree.
Can we say anything about the non-tree edges?
Can we say anything about the non-tree edges?
A property of Non-DFS-Tree Edges

**Theorem**

Let $x$ and $y$ be nodes in the DFS tree $T_G$ such that $\{x, y\}$ is an edge in undirected graph $G$. Then one of $x$ or $y$ is an ancestor of the other in $T_G$.

**Proof.**

Suppose, wlog, $x$ is reached first in the DFS.

All the nodes that are marked explored between first encountering $x$ and leaving $x$ for the last time are descendants of $x$ in $T_G$.

When we reach $x$, node $y$ must not yet have been explored.

It must become explored before leaving $x$ for the last time (otherwise, we should add $\{x, y\}$ to $T_G$). Hence, $y$ is a descendent of $x$ in $T_G$. 

□
Implementing BFS & DFS

- BFS: When you see a new, unexplored node, put it on a queue. Will process nodes in the order you first see them.

- DFS: When you see a new, unexplored node, put it on a stack. Will immediately process a new node.

When you have to backtrack, unexplored node closest to the top of the stack will be a child of the last node you visited where you had a choice.
We can think of BFS and DFS (and several other algorithms) as special cases of tree growing:

- Let $T$ be the current tree $T$, and
- Maintain a list of frontier edges: the set of edges of $G$ that have one endpoint in $T$ and one endpoint not in $T$:

$$
\begin{align*}
\text{v} & \quad \text{v} \\
\text{v} & \quad \text{v} \\
\text{v} & \quad \text{v} \\
\text{v} & \quad \text{v} \\
\end{align*}
$$

- Repeatedly choose a frontier edge (somehow) and add it to $T$. 

Tree Growing

TreeGrowing(graph G, vertex v, func nextEdge):
    T = (v, ∅)
    S = set of edges incident to v
    While S is not empty:
        e = nextEdge(G, S)
        T = T + e // add edge e to T
        S = updateFrontier(G, S, e)
    return T

• The function nextEdge(G, S) returns a frontier edge from S.
• updateFrontier(G, S, e) returns the new frontier after we add edge e to T.
These algorithms are all special cases of Tree Growing, with different versions of `nextEdge`:

1. Depth-first search
2. Breadth-first search
3. Prim’s minimum spanning tree algorithm
4. Dijkstra’s shortest path
What’s `nextEdge` for DFS?

What’s `nextEdge` for BFS?
BFS & DFS as Tree Growing

What’s nextEdge for DFS?

Select a frontier edge whose tree endpoint was discovered most recently.

Why? We can use a stack to implement DFS.

What’s nextEdge for BFS?
What’s nextEdge for DFS?

Select a frontier edge whose tree endpoint was discovered most recently.

Why? We can use a stack to implement DFS.

What’s nextEdge for BFS?

Select a frontier edge whose tree endpoint was discovered earliest.

Why? We can use a queue to implement BFS.
An Application of BFS
Testing Bipartiteness

Problem
Determine if a graph $G$ is bipartite.

Bipartite graphs can’t contain odd cycles:
How can we test if $G$ is bipartite?

• Do a BFS starting from some node $s$.
• Color even levels "blue" and odd levels "red."
• Check each edge to see if any edge has both endpoints the same color.
Bipartite Testing

How can we test if $G$ is bipartite?

- Do a BFS starting from some node $s$.
- Color even levels “blue” and odd levels “red.”
- Check each edge to see if any edge has both endpoints the same color.
Proof of Correctness for Bipartite Testing

One of two cases happen:

1. There is no edge of $G$ between two nodes of the same layer. In this case, every edge just connects two nodes in adjacent layers. But adjacent layers are oppositely colored, so $G$ must be bipartite.

2. There is an edge of $G$ joining two nodes $x$ and $y$ of the same layer $L_j$. Let $z \in L_i$ be the least common ancestor of $x$ and $y$ in the BFS tree $T$. $z - x - y - z$ is a cycle of length $2(j - i) + 1$, which is odd, so $G$ is not bipartite.
An Application of DFS
A directed, acyclic graph (DAG) is a graph that contains no directed cycles. (After leaving any node $u$ you can never get back to $u$ by following edges along the arrows.)

DAGs are very useful in modeling project dependencies: Task $i$ has to be done before task $j$ and $k$ which have to be done before $m$. 
Given a DAG $D$ representing dependencies, how do you order the jobs so that when a job is started, all its dependencies are done?
Theorem

Every DAG contains a vertex with no incoming edges.
Theorem

*Every* DAG *contains a vertex with no incoming edges.*

Proof.

Suppose not.

Then keep following edges backward and in fewer than \( n + 1 \) steps you’ll reach a node you’ve already visited.

This is a directed cycle, contradicting that the graph is a DAG.
Theorem

*Every DAG contains a vertex with no incoming edges.*

Proof.

Suppose not.

Then keep following edges backward and in fewer than \( n + 1 \) steps you’ll reach a node you’ve already visited.

This is a directed cycle, contradicting that the graph is a DAG.

How can we turn this into an algorithm?
Topological sort:

1. Let $i = 1$
2. Find a node $u$ with no incoming edges, and let $f(u) = i$
3. Delete $u$ from the graph
4. Increment $i$

Implementation: Maintain

- $\text{Income}[w] =$ number of incoming edges for node $w$
- a list $S$ of nodes that currently have no incoming edges.

When we delete a node $u$, we decrement $\text{Income}[w]$ for all neighbors $w$ of $u$. If $\text{Income}[w]$ becomes 0, we add $w$ to $S$. 
DFS can be used to associate 2 numbers with each node of a graph $G$:

- **discovery time**: $d[u] = \text{the time at which } u \text{ is first visited}$
- **finishing time**: $f[u] = \text{the time at which all } u \text{ and all its neighbors have been visited.}$

Clearly $d[u] \leq f[u]$. 

\[ 
\begin{align*}
&d[u] \quad f[u] \\
&d[v] \quad v \\
&f[v] \\
&d[u] \quad f[u] \\
&u \quad v
\end{align*} \]
Let $(u, v)$ be an edge of a DAG $D$. What can we say about the relationship between $f[u]$ and $f[v]$?
Let \((u, v)\) be an edge of a DAG \(D\). What can we say about the relationship between \(f[u]\) and \(f[v]\)?

Back edges and Left-Right edges cannot occur

\[ \implies f[v] < f[u] \text{ if } (u, v) \in D. \]
Topological Sort Via Finishing Times:

Every edge \((u, v)\) in a DAG has \(f[v] < f[u]\).

If we list nodes from largest \(f[u]\) to smallest \(f[u]\) then every edge goes from left to right.

Exactly a topological sort.

So: as each node is finished, add it to the front of a linked list.