A graph \( G = (V, E) \) is a pair of sets. Vertices: any set you want Edges: If \( G \) is undirected, each edge is an unordered pair of vertices. May denote edge containing \( u \) and \( v \) as \( uv \). If \( G \) is directed, edges are ordered pairs \( (u, v) \). I may write it as \( u \rightarrow v \).
Sometimes edges have a value called the weight or cost.

\( u \rightarrow v \) are adjacent if \( uv \in E \).

(If \( G \) is directed, say "\( v \) is adjacent to \( u \)" if \( u \rightarrow v \in E \).)

Edge \( uv \) is incident to \( u \) and vice versa.
V can be any set.

Social network graphs:
vertices are the set of people or users
edges for connects
(u v : u & v are friends)
(u & v : u follows v)

dependency graph:
classes are vertices
u & v : class u is a prereq.
for class v
Representing graphs

will discuss directed graphs mostly, and directed edges \( uv \) treated as a pair \( u \rightarrow v \) or \( v \leftarrow u \)

Two main representations:

Adjacency matrix:

A \( |V| \times |V| \) boolean matrix \( A \)

\( A[u \rightarrow v] \) is true iff \( u \rightarrow v \in E \).
Could use edge weights or # parallel edges instead of true/false

O(D) time to check if $u \geq v \in E$

Uses $\Theta(|V|^2)$ space.

If graph is dense that's fine ($|E| = \Theta(|V|^2)$)

Sparse: $|E| = \Theta(|V|)$
$\Theta(nm)$ time to find incident edges for $u$.

**Adjacency List:**

For each vertex $u$, keep a list of vertices adjacent to $u$.

Each edge represented once (twice if $G$ undirected).

$\Theta(nm + |E|)$ space. $\checkmark$
To find $k$ vertices adjacent to $u$, walk $u$'s list in $\Theta(k + 1)$ time.

$\Theta(k)$ time to check if $u \ni v \in E$. **Rarely comes up!**
We'll usually use an adjacency list.

Example:

**Vertex class.**
Instances in a doubly-linked list.

**Edge class.**
All in a doubly-linked lists.

Edges had a couple extra instance variables for weights, "marks," etc.

Vertices had lists of links to incident Edges.
Walk: a sequence of vertices $v_1, v_2, \ldots, v_k$ s.t.

$$v_i \neq v_{i+1}, \forall i \leq k$$

Path: A walk with no repeated vertices.

(Text calls these "paths" + "simple paths")

Undirected graph is connected if there is a path between any pair of vertices.
Directed graph is strongly connected if there is a path from $u$ to $v$ and $(u,v) \in V \times V$.

Only weakly connected if we have to ignore edge direction to find these paths.

$u$ can reach $v$ if there is a path from $u$ to $v$. 

\[ \begin{array}{c}
\text{u} \\
\rightarrow \\
\circ \\
\rightarrow \\
\circ \\
\rightarrow \\
\text{v} \\
\end{array} \]
Given \( G = (V,E) \) and some vertex \( s \in V \),

What vertices can be reached from \( s \)?

\( s \) can reach itself with path \( <s> \).

If we can reach \( u \) and \( u \in V \in E \), we can add one more edge to \( su \) walk to get an \( s, v \) walk.

Idea: keep a queue of reachable vertices.
While queue not empty, dequeue u & add all adjacent v to queue.

**Cyclo** Path except \( v_i = v_k \)

Infinite loop!

So, mark vertices when we visit them. Only enqueue unmarked vertices.
Let's use a stack instead.

**Depth-First search (DFS)**

```java
void dfs(Vertex v) {
    v.visited = true;
    for each Vertex w adjacent to v {
        if (!w.visited) {
            dfs(w);
        }
    }
}
```

Call `dfs(s)` to mark vertices reachable from `s`.

Runs in $O(V + E)$

(loops over each edge at most once)

If you can only reach `k` edges, then $O(V + k)$ time.

need to unmark first
The reached edges $v \rightarrow w$ where $w$ is not yet marked form a tree rooted at $s$.

depth-first search tree
void dfsAll() {
    for each Vertex v {
        if (!v.visited) {
            dfs(v);
        }
    }
}

Edges to unmarked vertices form a depth-first spanning forest.

Edges outside forest:

Forward edges: \( u \rightarrow v \) where \( u \) was an ancestor of \( v \) in forest

Cross edges: \( u \rightarrow v \) s.t. neither \( u \) nor \( v \) is an ancestor of each other.
Back edges: \( u \rightarrow v \) where \( v \) is an ancestor of \( u \).

\[
G \text{ has directed cycles iff all } \text{ dfs tree edges have at least one back edge.}
\]
dfs(v) eventually reaches u so \( u \to v \) is a back edge.

Back edge \( u \to v \) means \( \text{dfs}(v) \) started while \( \text{dfs}(u) \) was running.

Add an extra visiting bit to detect back edges and cycles.
void detectCycle(Vertex v) {
    v.visiting = true;
    for each Vertex w adjacent to v {
        if (!w.visited) {
            if (w.visiting) {
                return true;
            }
            detectCycle(w);
        }
    }
    v.visiting = false;
    v.visited = true;
}

boolean detectCycle() {
    for each Vertex v {
        if (!v.visited) {
            detectCycle(v);
        }
    }
    return false;
}