Graph \( G = (V, E) \)

\( V \): vertices: an arbitrary finite set of anything

\( E \): edges: pairs of elements of \( V \) (i.e., vertices)
$u \rightarrow v$ (directed edge)
$uv$ (undirected edge)

If edge $e$ contains vertices $u \leftrightarrow v$:

- $u \leftrightarrow v$ are adjacent
- $e$ is incident to $u \leftrightarrow v$
- $u$ is a neighbor of $v$ (vice versa)
degree of \( u \) is \( \# \) neighbors (assume no parallel edges)

If \( u \rightarrow v \) is a directed edge, \( u \) is the tail and \( v \) is the head.

\( u \) is a predecessor of \( v \)
\( v \) is a successor of \( u \).
in-degree: \# predecessors
out-degree: \# successors

Sometimes \( V \) is \# vertices and \( E \) is \# edges

\textit{e.g., an algo runs in } \( O(V+E) \) \textit{time}
Data Structures

adjacency list:
an array indexed by vertices or their label
elements are lists of adjacent vertices
(successors only if \( G \) is directed)
usually uses singly linked lists for adjacent vertices. Each edge $uv$ appears twice if undirected.
Space: $\Theta(V + E)$

Learn neighbors of $u$ in optimal $O(\text{deg}(u))$ time

degree of $u$ (\# neighbors)

Have to check $u$'s whole list to know if $u \rightarrow v$ exists!
Could use hash table lists...
Adjacency matrix: \( |V| \times |V| \) matrix of 0s and 1s, stored as 2D array \( A \) of \( |V| \times |V| \).

Undirected: \( A[u,v] = 1 \) iff \( uv \in E \)

Directed: \( A[u,v] = 1 \) iff \( u \rightarrow v \in E \)
$\Theta(\nu^2)$ space, 
Neighbors of $u$ in $\Theta(\nu)$ time.

Check if $uv$ exists in $\Theta(1)$ time.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Standard adjacency list (linked lists)</th>
<th>Fast adjacency list (hash tables)</th>
<th>Adjacency matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V^2)$</td>
</tr>
<tr>
<td>Test if $uv \in E$</td>
<td>$O(1 + \min{\deg(u), \deg(v)}) = O(V)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(1 + \deg(u)) = O(V)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$\Theta(1 + \deg(v)) = O(V)$</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V)$</td>
</tr>
<tr>
<td></td>
<td>$O(1)$</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(1)$</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>List v’s (out-)neighbors</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert edge $uv$</td>
<td>$O(1)$</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete edge $uv$</td>
<td>$O(1)$</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(1)^*$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume we're using an adjacency list.
Graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

walk: a sequence of edges where successive edges share a common vertex

path: a walk with no repeated vertices

$G$ is connected if there is a walk between any pair of vertices
Components: maximal connected subgraphs of $G$

Given vertex $s$, a vertex $u$ is reachable from $s$, if there is an $su$-walk.

Given $s$, what is reachable?
Whatever-first search uses a "bag" data structure - supports adding objects, removing added objects (where we came from) on objects

**WHATEVERFIRSTSEARCH(s):**

- put \((\emptyset, s)\) in bag
- while the bag is not empty
  - take \((p, v)\) from the bag
  - if \(v\) is unmarked
    - mark \(v\)
    - \(parent(v) \leftarrow p\)
    - for each edge \(v\)w
      - put \((v, w)\) into the bag
cycle: a walk that repeats only its first/last vertex

Tree: a connected graph that has no cycles

spanning tree of $G$: a subgraph of $G$ that is a tree & contains every vertex
Lemma: Whatever First Search(s) marks exactly the vertices reachable from s.

The set of pairs \((v, \text{parent}(v))\) where \(\text{parent}(v) \geq 0\) form a spanning tree of the component containing s.

Proof: Each vertex marked at most once.
Show each reachable vertex is marked by induction on shortest path length from s.

S is marked right away.

If $v \neq s$ is reachable.

Let $s \rightarrow \cdots \rightarrow u \rightarrow v$ be shortest path to v.

$u$ is reachable & closer to $s$ than $v$.

IH implies $u$ is marked.
So we add \((u,v)\) to \(f\). Guaranteed \(v\) is marked after \((u,v)\) is removed.

We only mark reachable vertices.\(s\) is marked & reachable.

If we mark \(v \neq s\). \(\text{Pair } (\text{parent}(v), v)\) is an edge.

So we marked \(\text{parent}(v)\) first.

By induction on order we mark vertices, \(\text{parent}(v)\) is reachable.
There is a walk
\[ S \rightarrow \ldots \rightarrow \text{parent}(v) \rightarrow v \]

Finally there \( (\text{parent}(v), v) \)

edges spanning the component of \( s \).

All marked vertices except \( s \) has a parent, so one fewer edge than \( A \) vertices in component.

\[ \implies \text{the edges make a tree} \]
**WhateverFirstSearch(s):**

- put \((\emptyset, s)\) in bag
- while the bag is not empty
  - take \((p, v)\) from the bag
  - if \(v\) is unmarked
    - mark \(v\)
    - \(parent(v) \leftarrow p\)
    - for each edge \(vw\)
      - put \((v, w)\) into the bag

\[\text{Time} \leq 2|E| + 1 \times\]

- \(T\): time to add or remove from bag

\[O(V + E)\text{ time}\]

- put \& pull from bag
Which bag?

Stack: depth-first spanning tree

- long & skinny

Queue: breadth-first (unweighted) spanning tree

Shortest paths!

$O(V + E)$
priority queue: depending on priorities:
  - Prim's algo for MST
  - Dijkstra
  - "widest" paths with min heap:
    \[ O(V + E \log E) = O(V + E \log V) \]