Course:

CS 2100

Intro to graphs
Recap

- HW due Wed.
- Worksheet: coming (today)
- Review: 1 week from today
- Sample final coming tomorrow
- No lab - lecture instead
- Wed next week: 8am final
- No office hours today
A graph $G = (V, E)$ is an ordered pair of two sets:

$$V = \text{vertices} = \{v_1, v_2, v_3, v_4\}$$

$$E = \text{edges} = \{v_1, v_2, v_3, v_4\}$$

- $\{v_1, v_3\}$
- $\{v_1, v_4\}$
- $\{v_2, v_3\}$
- $\{v_2, v_4\}$

View:

- Peterson graph
Why?

They model everything! (non-hierarchical, non-linear)

Examples
- Connections on social media
- Road networks
- Internet
More defns:

$G$ is undirected if edges are unordered pairs so $\exists u, v \in V \implies \exists v, u \in V$.

$G$ is directed if edges are ordered pairs so $(u, v) \neq (v, u)$.

Endpoints

Head

Tail
The degree of a vertex, $d(v)$, is the number of adjacent edges.

A path $P = v_1, ..., v_k$ is a set of vertices with $\{ v_i, v_{i+1} \} \in E$ (or $(v_i, v_{i+1}) \in E$ if directed).

A path is simple if all vertices are distinct.

A cycle is a path which is simple except $v_i = v_k$.

5-cycle $P = u, v, w, y$
Lemma: (degree-sum formula)

\[ \sum_{v \in V} d(v) = 2|E| \]

Proof:

Sum over all vertices of degree of vertex:
\[ d(v_1) + d(v_2) + \ldots + d(v_n) \]

(Sum over each vertex)

Number of edges:
\[ 2 + 2 + \ldots + 2 \]

(Sum over each edge)
Size of $G$:

2 parameters:

$|V| = n$

$|E| = m$

How big can $m$ be in terms of $n$?

$m \leq \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$

$= O(n^2)$

Worst case: $K_n$

Complete graph
Tree:
A connected graph with no cycles
(Note: no root here!)

\[ n-1 \text{ edges} = m \]
Representing graphs

How do we make this data structure?

- pointers! 

> list-like

\[ \text{graph} \]
Adjacency (or vertex) lists:

- $V_1 \circ V_2, V_5$
- $V_2 \circ V_1, V_3, V_5$
- $V_3$
- $V_4$
- $V_5$

**Size**: $n$ "lists", each size $\leq n-1$

**Lookup time to check if $v_i \circ v_j$ are nbrs**: $O(1)$

$O(n)$ (or $O(\log n)$)
Implementation:
We call these vertex lists, but don't have to use lists.

Options:
- Lists
- Vectors

Trade-offs:
Insert/Remove vs. Lookup
Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>v₁</th>
<th>v₂</th>
<th>v₃</th>
<th>v₄</th>
<th>v₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₁</td>
<td>×</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v₂</td>
<td>1</td>
<td>×</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v₃</td>
<td>1</td>
<td>1</td>
<td>×</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>v₄</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>×</td>
<td>0</td>
</tr>
<tr>
<td>v₅</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>×</td>
</tr>
</tbody>
</table>

Directed: need both "halves" of matrix

Space: $O(n^2)$

Check nbr: $O(1)$

$A[i][j]$
Which is better? Depends!

<table>
<thead>
<tr>
<th></th>
<th>Adjacency matrix</th>
<th>Standard adjacency list (linked lists)</th>
<th>Adjacency list (hash tables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\Theta(V^2)$</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V + E)$</td>
</tr>
<tr>
<td>Time to test if $uv \in E$</td>
<td>$O(1)$</td>
<td>$O(1 + \min{\deg(u), \deg(v)}) = O(V)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Time to test if $u \rightarrow v \in E$</td>
<td>$O(1)$</td>
<td>$O(1 + \deg(u)) = O(V)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Time to list the neighbors of $v$</td>
<td>$O(V)$</td>
<td>$O(1 + \deg(v))$</td>
<td>$O(1 + \deg(v))$</td>
</tr>
<tr>
<td>Time to list all edges</td>
<td>$\Theta(V^2)$</td>
<td>$\Theta(V + E)$</td>
<td>$\Theta(V + E)$</td>
</tr>
<tr>
<td>Time to add edge $uv$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)^*$</td>
</tr>
<tr>
<td>Time to delete edge $uv$</td>
<td>$O(1)$</td>
<td>$O(\deg(u) + \deg(v)) = O(V)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

$O(n^2)$ space $\uparrow$ $O(n + m)$
**Dfn:**

- $G$ is **connected** if for all $u, v$, there exists a path from $u$ to $v$.

- The **distance** from $u$ to $v$, $d(u, v)$, is equal to the number of edges on the minimum $u, v$-path.
Algorithms on graphs

Basic 1st question:
Given any 2 vertices, are they connected?
Also: What is their distance?

How to solve?
Suggestion:
Suppose we're in a maze searching for something. What do you do?
Pseudocode: two versions

**RecursiveDFS**($v$):
- if $v$ is unmarked
  - mark $v$
  - for each edge $vw$
    - **RecursiveDFS**($w$)

**IterativeDFS**($s$):
- **Push**($s$)
- while the stack is not empty
  - $v \leftarrow \text{Pop}$
  - if $v$ is unmarked
    - mark $v$
    - for each edge $vw$
    - **Push**($w$)

Really, building a "tree":

DFS tree:
General traversal strategy:

**Traverse(s):**
- put s into the bag
- while the bag is not empty
  - take v from the bag
  - if v is unmarked
    - mark v
    - for each edge vw
      - put w into the bag

Q: Can we use a different "bag"?
**BFS**: use a queue

**Traverse(s):**
- put $s$ into the bag
- while the bag is not empty
  - take $v$ from the bag
  - if $v$ is unmarked
    - mark $v$
    - for each edge $vw$
      - put $w$ into the bag
BFS vs. DFS