CS2100 - Intro to Graphs

Note Title 12/8/2011

Announcements

- Lab tomorrow (already up)
- HW due Wednesday

Decode HW

```
BinaryTree<          > myTree;
BitStream myfile;
myfile.open("banana.myzip.txt");
int input;
input = myfile.read(1);
if (input != 0) {
    myTree.createRoot();
}
read more bits
//loop to create the tree
while(          ) {

    (move position up & down to
        insert new values)

}
```

```
int letter;
letter = myfile.read(9)
cout << char(letter) << endl;
```

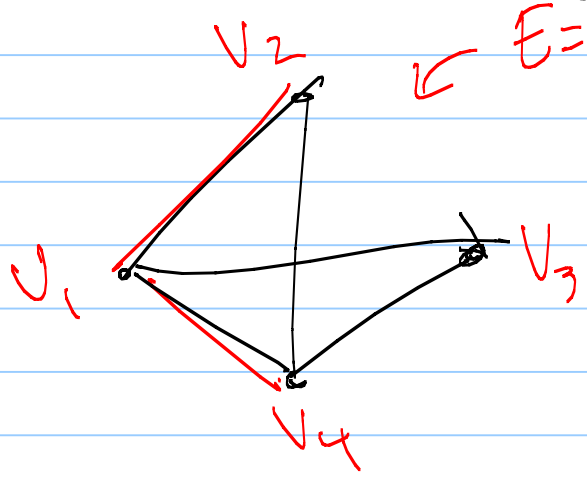
Graphs

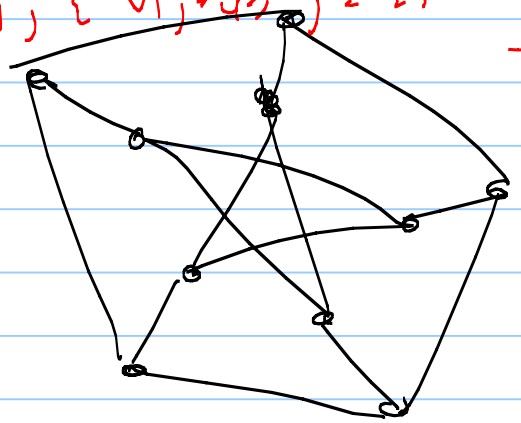
A graph $G = (V, E)$ is a set of 2 sets V & E.

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

E = edges (which are pairs of vertices)

$E = \{\{v_1, v_2\}, \{v_1, v_4\}, \ldots\}$





Why use graphs?

They can model anything!

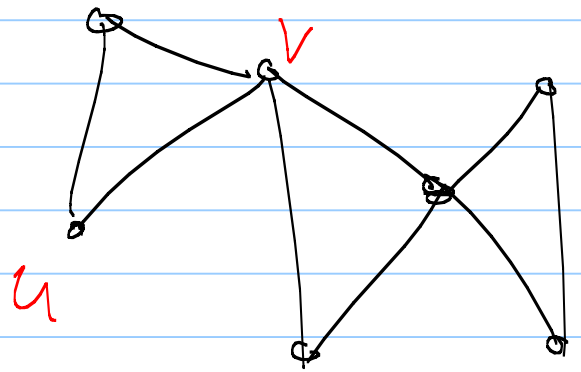
Examples:

- maps
- networks (facebook, pagerank...)
- scheduling
- ⋮

Definitions

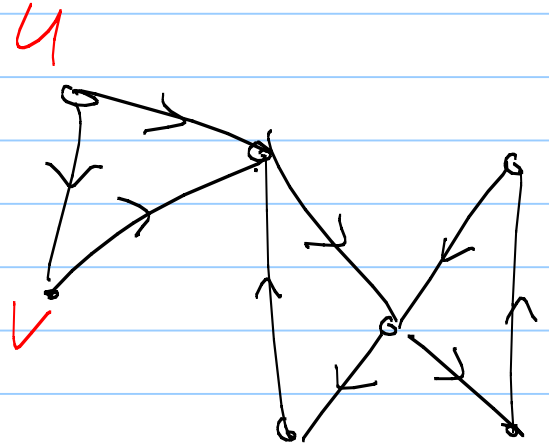
- G is undirected if every edge is an unordered pair

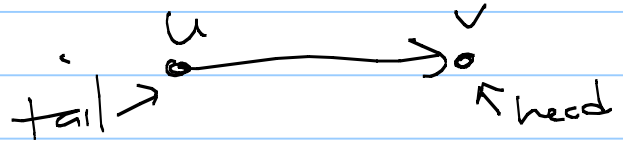
  so $\{u, v\} = \{v, u\}$



- G is directed is every edge is an ordered pair

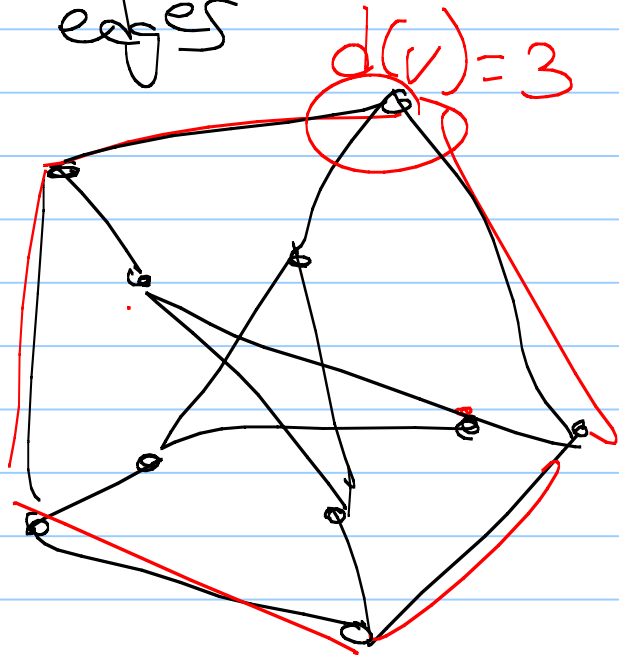
  $e = (u, v) \neq (v, u)$





Dfns

- The degree of a vertex, $d(v)$, is the number of adjacent edges
- A path $P = v_1 \dots v_k$ is a set of vertices with $\{v_i, v_{i+1}\} \in E$
- A path is simple if all vertices are distinct
- A path is a cycle if it is simple except $v_1 = v_k$



Lemma: (degree-sum formula)

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

Why? counting: think of edges in G

every edge connects 2 vertices

count via the vertices:

$d(v_1) + d(v_2) + d(v_3) + \cdots$

Either way, counting vertex-edge incidences

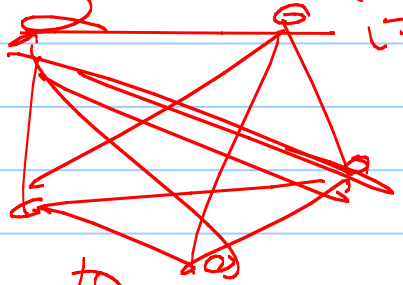
Sizes of $|V|$ & $|E|$

We usually let $n = |V|$ and $m = |E|$.

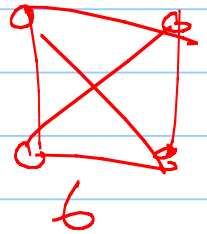
How big can m be?

How many edges can a graph have?

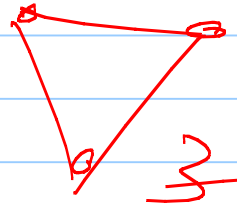
$$\binom{n}{2} = \frac{n(n-1)}{2} = \sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n$$



10



6



3

Complete graph

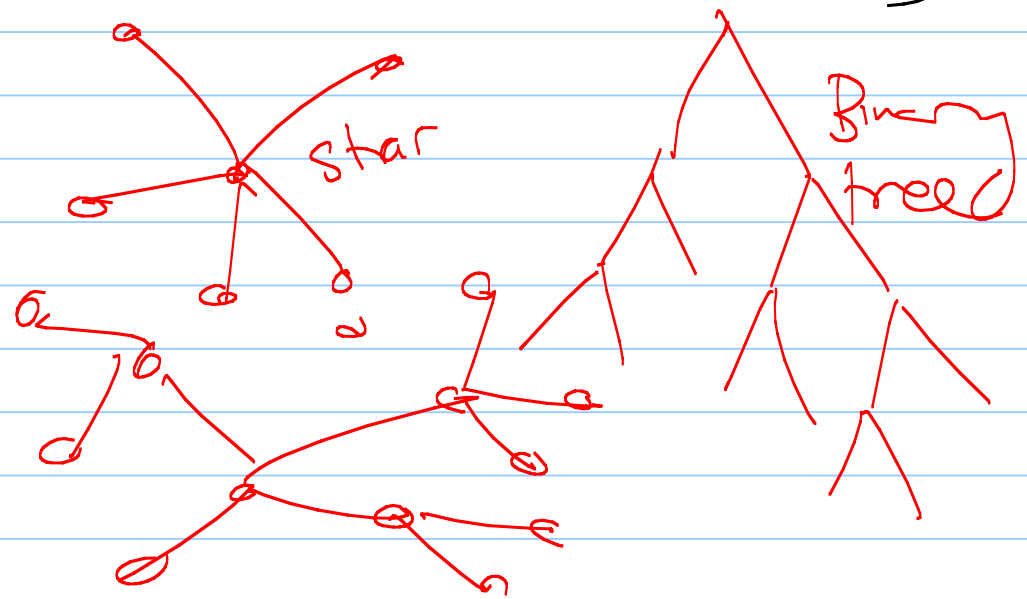
$m = O(n^2)$

Tree: A connected graph with no cycles.

(Note: No root in this definition!)

How many edges?

$n-1$



Graphs on a computer

How can we construct this data structure?

Linked structure

```
struct Vertex {
    value
    vector or list of nbrs
}
```

Vertex Lists (or Vectors)

+ Value list

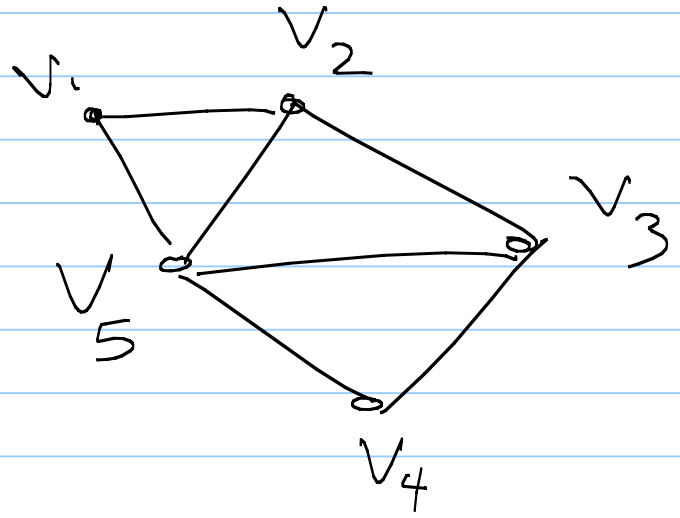
v_1 : v_2, v_5

v_2 : v_1, v_5, v_3

v_3 : v_2, v_4, v_5

v_4 : v_3, v_5

v_5 : v_1, v_2, v_3, v_4



size : $\sum_v d(v) = O(m)$

Check if v_i is neighbor of v_j : $O(n)$

Implementation

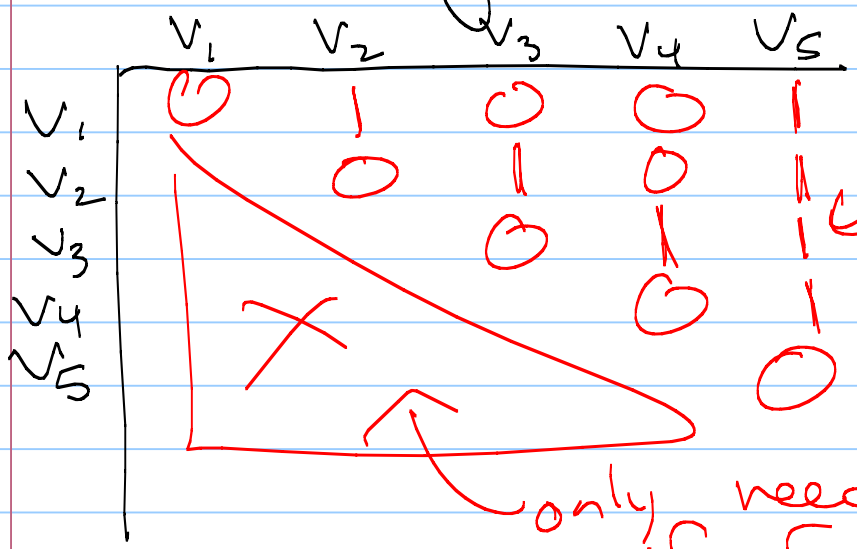
We call these vertex lists, but don't actually need lists.

Options: vectors, lists, BSTs

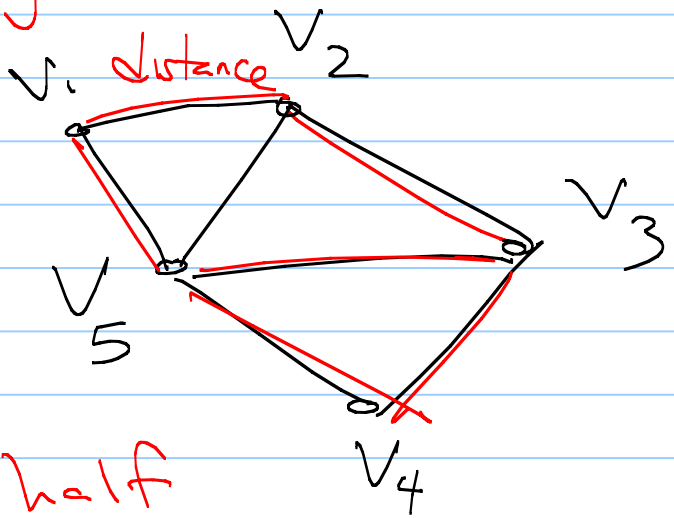
Tradeoffs: insert vs remove vs find

Value list : v_1 is "Erin"
v_2 is "..."

Adjacency Matrix



weight



only need this half if G is directed

Space: $O(n^2)$

check neighbor: $O(1)$ A[i][j]

Which is best?

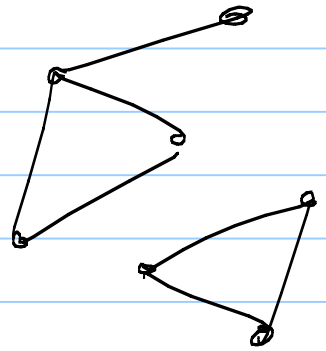
Just depends.

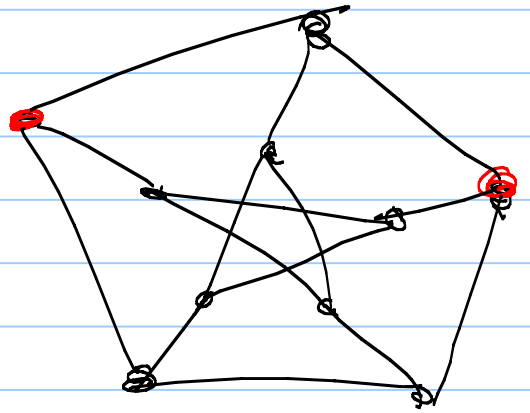
Incidence Matrix

	e_1	...	e_m
v_1			
v_2			
v_3			
v_4			
⋮			
v_n			

Dfns

- G is connected if for all $u \neq v$, there is a path from u to v.
- The distance from u to v, $d(u,v)$, is equal to the length of the minimum u,v-path.





Algorithms on Graphs

Basic Question: Given 2 vertices, are they connected?

How to solve?

Suggestion:

- Suppose we're in a maze, searching for a treasure.

What do you do?

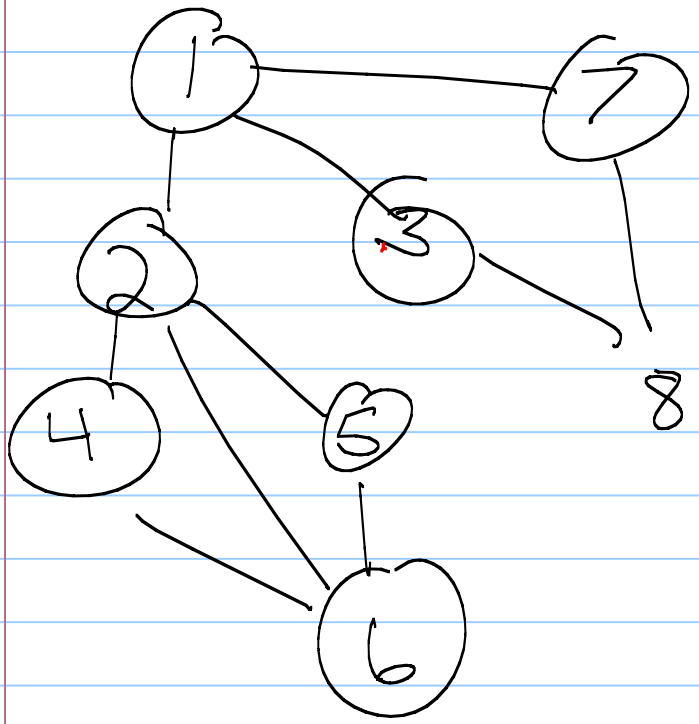
RecursiveDFS(u):

If u is unmarked:
- mark u
- for each edge $\{u, v\} \in E$
  - RecursiveDFS(v)

To check if s & t are connected, call DFS(s).

At end, if t is marked, return true

DFS



Another version of DFS

```
IterativeDFS(u):
    create empty stack S
    S.push(u)

    while S is not empty:
        v ← S.pop
        if v is not marked
            mark(v)
            for each edge vw
                S.push(w)
```