Recap:

- Normal-ish rest of the week
- HW due Wednesday
- Final worksheet posted
- Review next Monday in class
- Test next Wednesday @ 8am
- Sample finals tomorrow

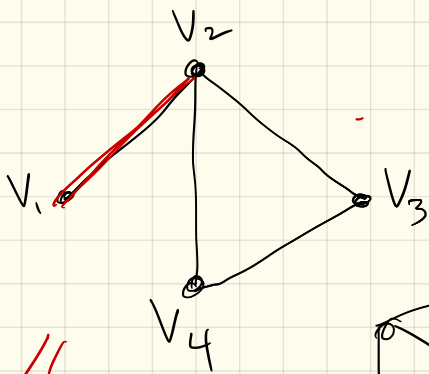
Graphs

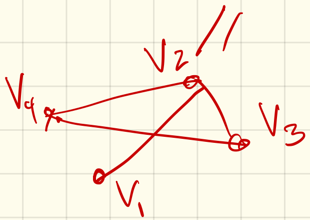
A graph $G = (V, E)$ is an ordered pair of 2 sets:

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

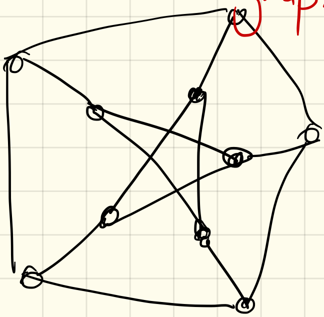
E = edges = $\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_2, v_4\}\}$

View:





Peterson graph

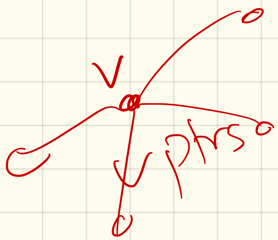


Representing graphs

How do we make this data structure?

- pointers! :)

  ↳ list-like



Adjacency (or vertex) lists:

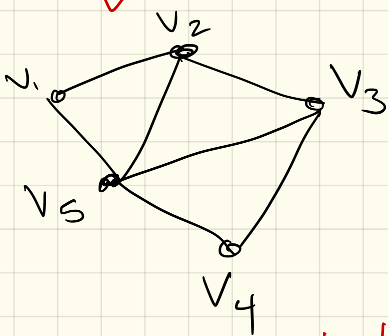
v_1 : v_2, v_5

v_2 : v_1, v_5, v_3

v_3 :

v_4 :

v_5 :



Size: n "lists", each ~~size~~ upper bnd $\le n-1$

$O(n+m)$

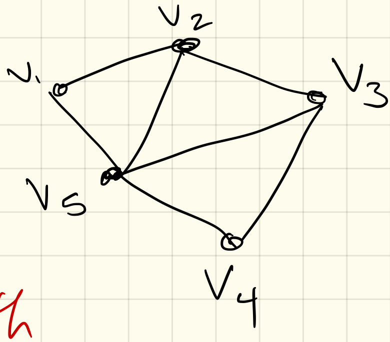
Lookup: Time to check if v_i + v_j are nbrs:

$O(n)$

(or $O(\log n)$)

Adjacency Matrix

	v_1	v_2	v_3	v_4	v_5
v_1	X	1	0	0	1
v_2	1	X	1	0	1
v_3		1	X	1	0
v_4			1	X	1
v_5				1	X



directed: need both "halves" of matrix

space: $O(n^2)$

check nbr: $O(1)$

$A[i][j]$

Which is better?

Depends!

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

$O(n^2)$ space

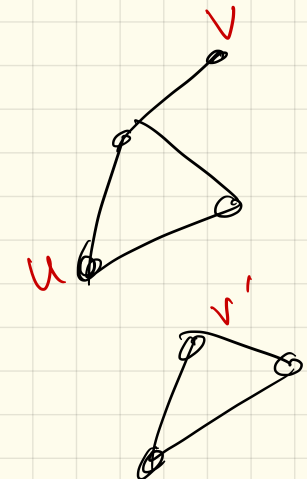
$O(n+m)$

Libraries: Boost, etc.

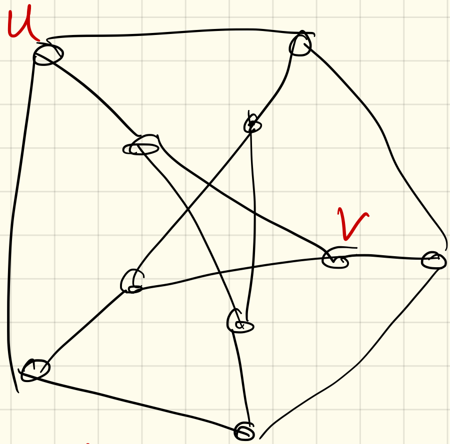
Dfn:

paths

- G is connected if $\forall u,v$, there $\exists$ path from u to v.
- The distance from u to v, $d(u,v)$, is equal to the # of edges on the minimum u,v-path (sum of weights)



$d(u,v') = \infty$
so disconnected



$d(u,v) = 2$

Algorithms on graphs

Basic 1st question:

Given any 2 vertices, are they connected?

Also: What is their distance?

How to solve?

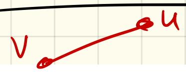
Suggestion:

Suppose we're in a maze, seaching for something.
What do you do?

depth first search
- go as far as you can

breadth first search
check nbrs,
then their nbrs, etc.

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 ← POP
        if v is unmarked
            mark v
            for each edge vw
                PUSH(w)
```

O(1)

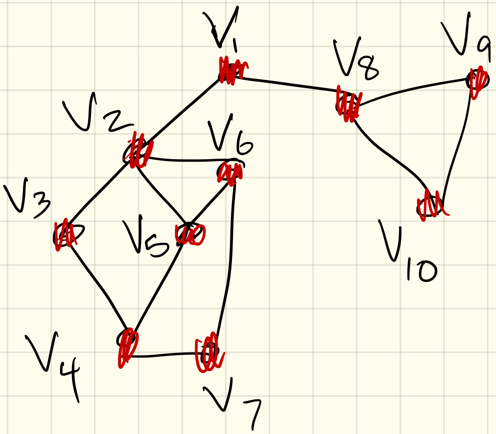
ds to hold boolean

O(1)

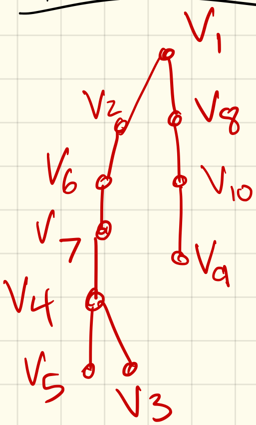
O(1)

$O(m+n)$

Really, building a "tree":



DFS tree:



Stack: ~~V_1~~ ~~V_2~~ ~~V_8~~ ~~V_1~~ ~~V_9~~ ~~V_{10}~~ ~~V_8~~ ~~V_9~~
V_1 V_3 V_5 ~~V_6~~ V_2 V_5 ~~V_7~~ ~~V_4~~ ~~V_6~~
~~V_3~~ ~~V_5~~ ~~V_7~~ ~~V_2~~ ~~V_4~~ ~~V_6~~ ~~V_7~~

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"?

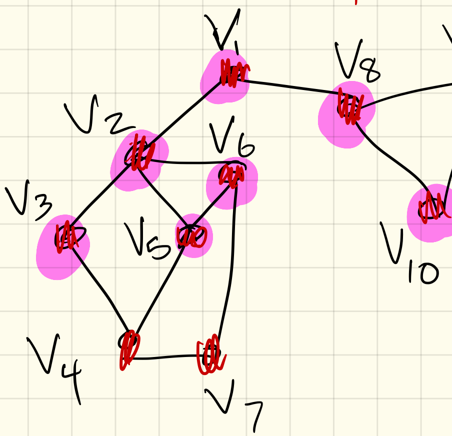
queue → $O(m+n)$

BFS: use a queue

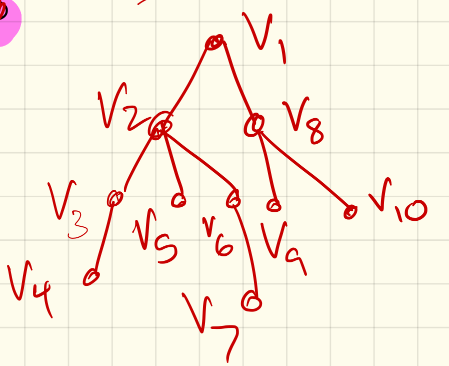
TRAVERSE(s):
- put s into the ~~bag~~ Q queue = Q
- while the ~~bag~~ Q is not empty
  - take v from the ~~bag~~ Q
  - if v is unmarked
    - mark v
    - for each edge vw
      - put w into the ~~bag~~ Q

Q: $\cancel{V_1}$ $\cancel{V_2}$ $\cancel{V_8}$ $\cancel{V_1}$ $\cancel{V_3}$ $\cancel{V_5}$ $\cancel{V_6}$ $\cancel{V_4}$ $\cancel{V_9}$ V_{10}

V_2 V_4 V_2 V_4 V_6 $\underbrace{V_2\ V_5\ V_7}_{V_6\text{'s push}}$ ...



BFS tree:



BFS vs. DFS:

- Both do connectivity
- Both are $O(m+n)$ time
  (w/ either graph rep)
- Difference:
  What you are optimizing tree for.

Next time:

- directed searching
- weighted graphs