CS300

BFS, MST

Announcements

- HW due today
- Next HW: oral grading, end of next week
- Midterm: Wednesday October 18

  review on Monday in class

Last time:

- Graph representations
- Graph traversals: DFS

Idea: determine connectivity - can we reach vertex u from vertex v?

(We're doing undirected, but all can be modified for directed - usually just by making sure edge lists have only outgoing edges.)

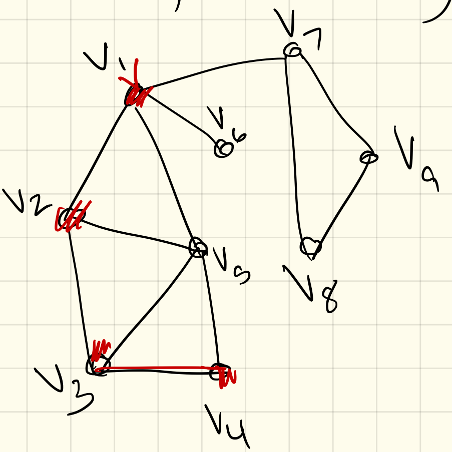
Pseudocode: two versions

```
RecursiveDFS(v):
    if v is unmarked
        mark v
        for each edge vw
            RecursiveDFS(w)
```

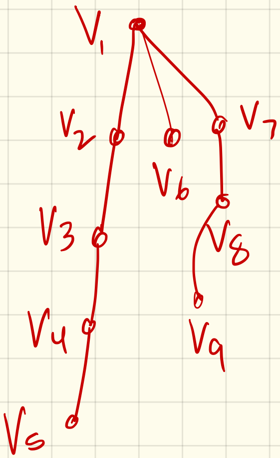
```
IterativeDFS(s):
    Push(s)                          O(1)
    while the stack is not empty
        v ← Pop                      O(1)
        if v is unmarked
            mark v
            for each edge vw
                Push(w)              O(1)
```

$O(m+n)$ total

Really, building a "tree":



DFS tree:



General traversal strategy:

in DFS, bag = stack

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

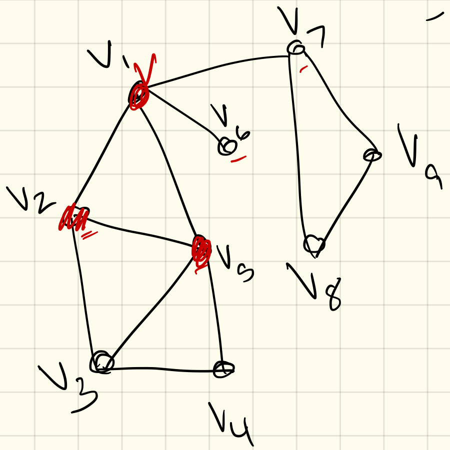
BFS: use a queue

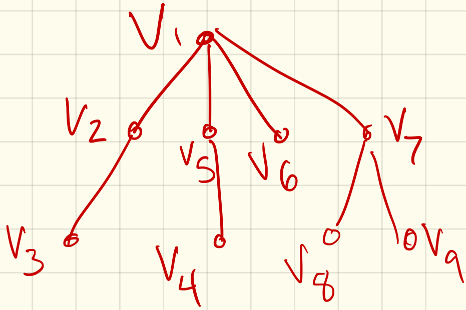
breadth first search

TRAVERSE(s):
    put s into the bag queue Q
    while the ~~bag~~ queue 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

$O(m+n)$

"distance" traversal





BFS vs. DFS.

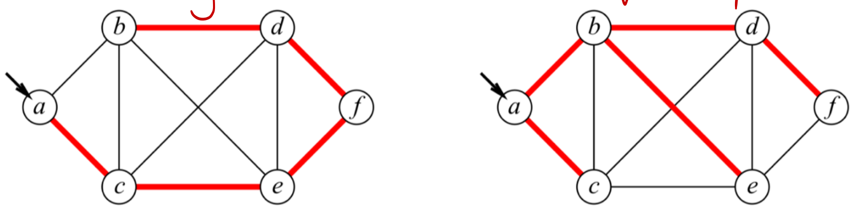
- Both can tell if 2 vertices are connected
- Both can be used to detect cycles.

  How? If revisit an edge, must have some cycle

- Both run in $O(V+E) = O(n+m)$ time.

Difference:

"long & thin" "bushy"

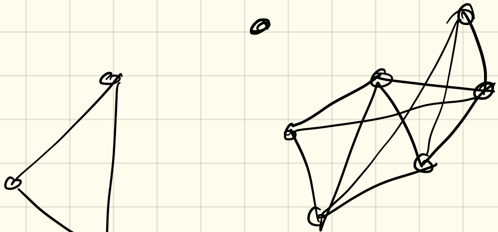


A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex a.

Dfn: A <u>tree</u> is a maximal acyclic graph, always with $n-1$ edges.

(DFS + BFS can both be used to get trees.)

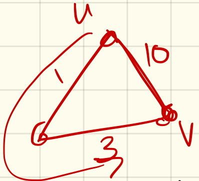
Dfn: A <u>component</u> of a graph is a maximal connected subset of G.



3 components

New setting: a weighted graph

A graph $G=(V,E)$ together with a weight function $w: E \to \mathbb{R}$ that gives a weight $w(e)$ to each edge.



sometimes $\mathbb{R}^+$

In this setting, finding shortest paths is much more interesting!

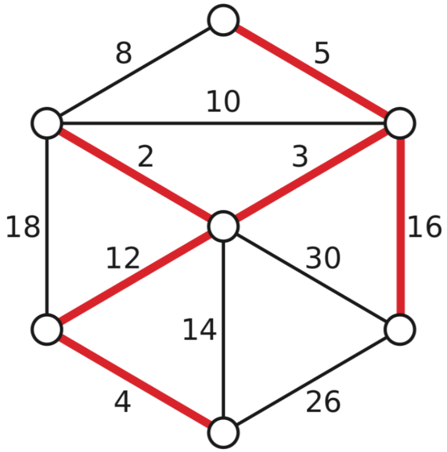
We'll start with a more basic question:

What is the best tree contained in the graph?

acyclic
minimum

Problem: Minimum Spanning Tree

Find a set of edges which connects all vertices & is as small as possible.



A weighted graph and its minimum spanning tree.

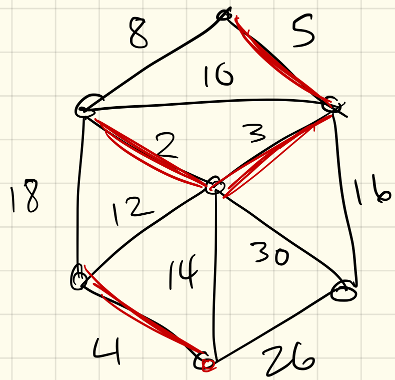
Applications: obvious

Strategy:

- We'll start by assuming edge weights are unique:

so $w(e) \neq w(e')$ $\forall e, e' \in E$

How to get started?

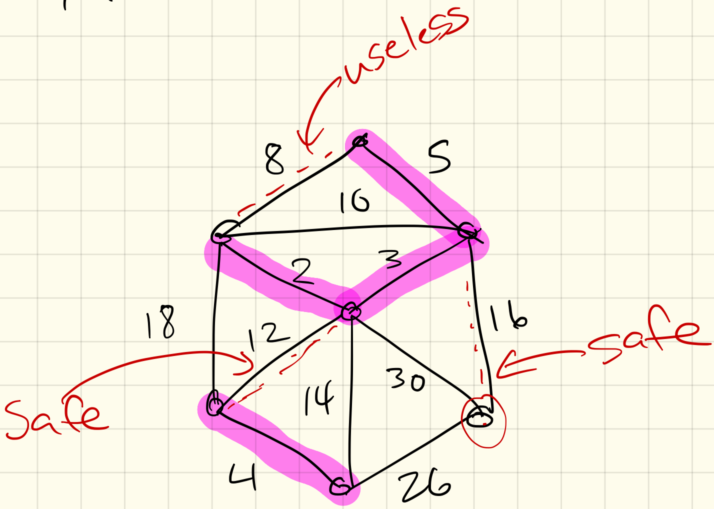


Idea: Choose smallest edge.

(greedy!)

Intermediate stage

Now suppose we have a partial MST (a forest).

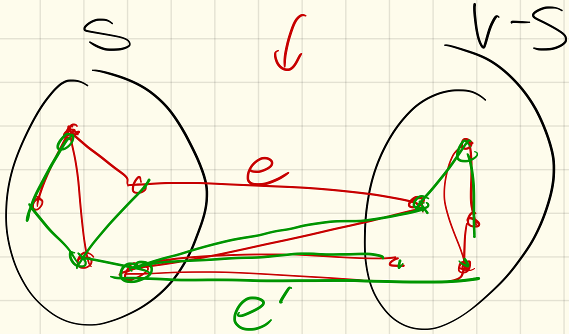


Better: Add min edge if not forming a cycle.

Lemma: Let S be any subset of V ($\neq \emptyset$ or V). Let e be the edge of minimum weight between S and $V-S$.

Then e is in *any* MST of G.

Pf:

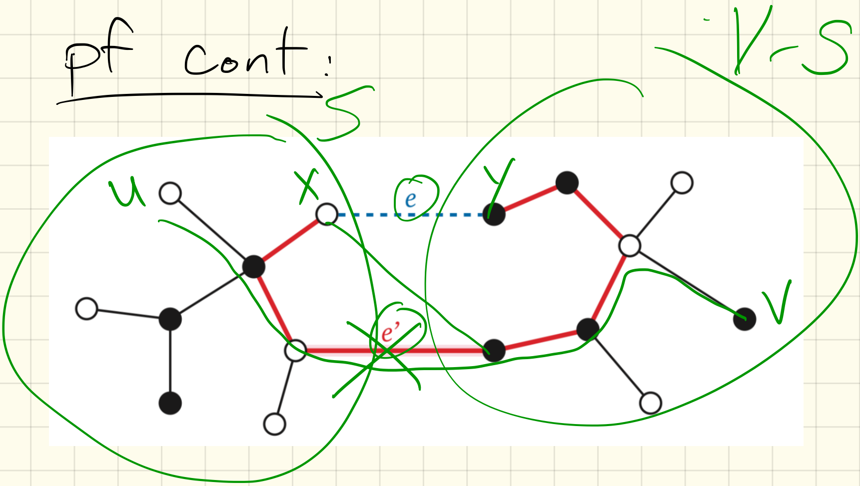


Suppose e is *not* in MST.

Let T be MST not containing e.

Some edge of T must leave S, say e'.

pf cont:



Prove $T - e' + e$ is better:

- minimum is clear:

$$w(e) < w(e')$$

T was a tree: $\forall u, v$, there was a path in T from u to v.

If path didn't use e', still there.

If it did: let $e = xy$.

Use u-to-x path, + e, + y to v path,

so $T - e' + e$ is still a tree. ☒

A bit further: Take a forest F:

Define a safe edge for any component of F as the minimum weight edge with only one endpoint in that component.

A useless edge is one not in F & with both endpoints in the same component.

Note: Prior lemma says any safe edge can be added to the MST!

Algorithm:

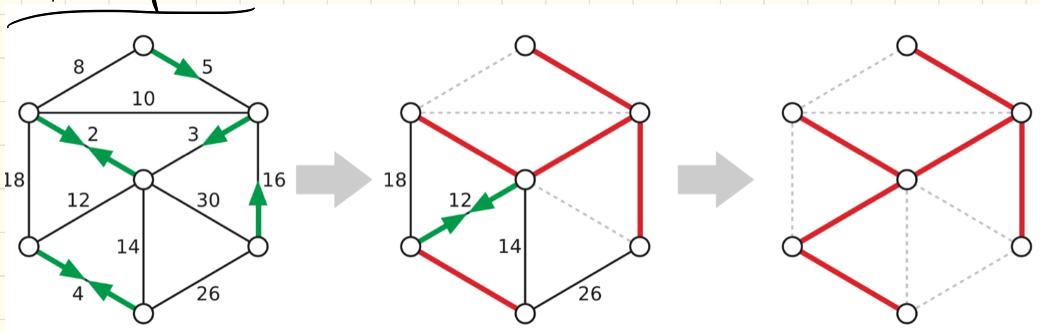
Start with n vertices.

Compute the safe edges.

Add them.

Recurse on new forest.

Example:



This is Borůvka's algorithm, from 1926.

(Also others - often called Sollin's algorithm.)

Pseudocode:

```
BORVKA(V, E):
    F = (V, ∅)
    count ← COUNTANDLABEL(F)
    while count > 1
        ADDALLSAFEEDGES(E, F, count)
        count ← COUNTANDLABEL(F)
    return F
```

```
ADDALLSAFEEDGES(E, F, count):
    for i ← 1 to count
        S[i] ← NULL        ⟨⟨sentinel: w(NULL) := ∞⟩⟩
    for each edge uv ∈ E
        if label(u) ≠ label(v)
            if w(uv) < w(S[label(u)])
                S[label(u)] ← uv
            if w(uv) < w(S[label(v)])
                S[label(v)] ← uv
    for i ← 1 to count
        if S[i] ≠ NULL
            add S[i] to F
```

Essentially:

Find min nbr for each vertex.

Label each component: use DFS/BFS

Find min edge leaving & add to F

repeat

Runtime:

Sort edges (once):

At each stage, get
$\frac{1}{2}$ many components
(worst case)

stages:

$T(n) = T(\frac{n}{2}) + 1$

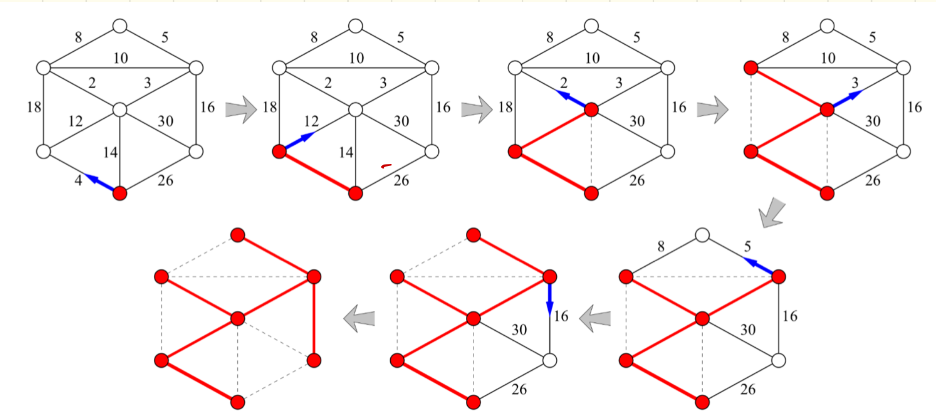
$O(\log n)$ stages

$O(m \log n)$
algorithm

Other algorithms:

Prim's algorithm: add a safe edge, one at a time

(Really Jarník's from 1929)



How to implement?