Algorithms - Spring '25

Top. order

Dyn Pro.

Recap

- Reading posted for Friday (will be shorter)
- Post next 2 weeks at same time
- Next HW - up later today over Chs 5 & 6
- Exam graded by Monday (hopefully)
- Pass back after break
- Mid Sem. survey check-in

Top sort DFS: making it more precise

O(V+E)

DFS

```
TopologicalSort(G):
    for all vertices v
        v.status ← New
    clock ← V
    for all vertices v
        if v.status = New
            clock ← TopSortDFS(v, clock)
    return S[1 .. V]
```

```
TopSortDFS(v, clock):
    v.status ← Active
    for each edge v→w
        if w.status = New
            clock ← TopSortDFS(v, clock)
        else if w.status = Active
            fail gracefully
    v.status ← Finished
    S[clock] ← v
    clock ← clock − 1
    return clock
```

Figure 6.9. Explicit topological sort

clock: ~~V~~ ~~V-1~~ ~~V-2~~ ~~V-3~~ ~~V-4~~ V-5

S[1 .. h | d c a b]

1 V

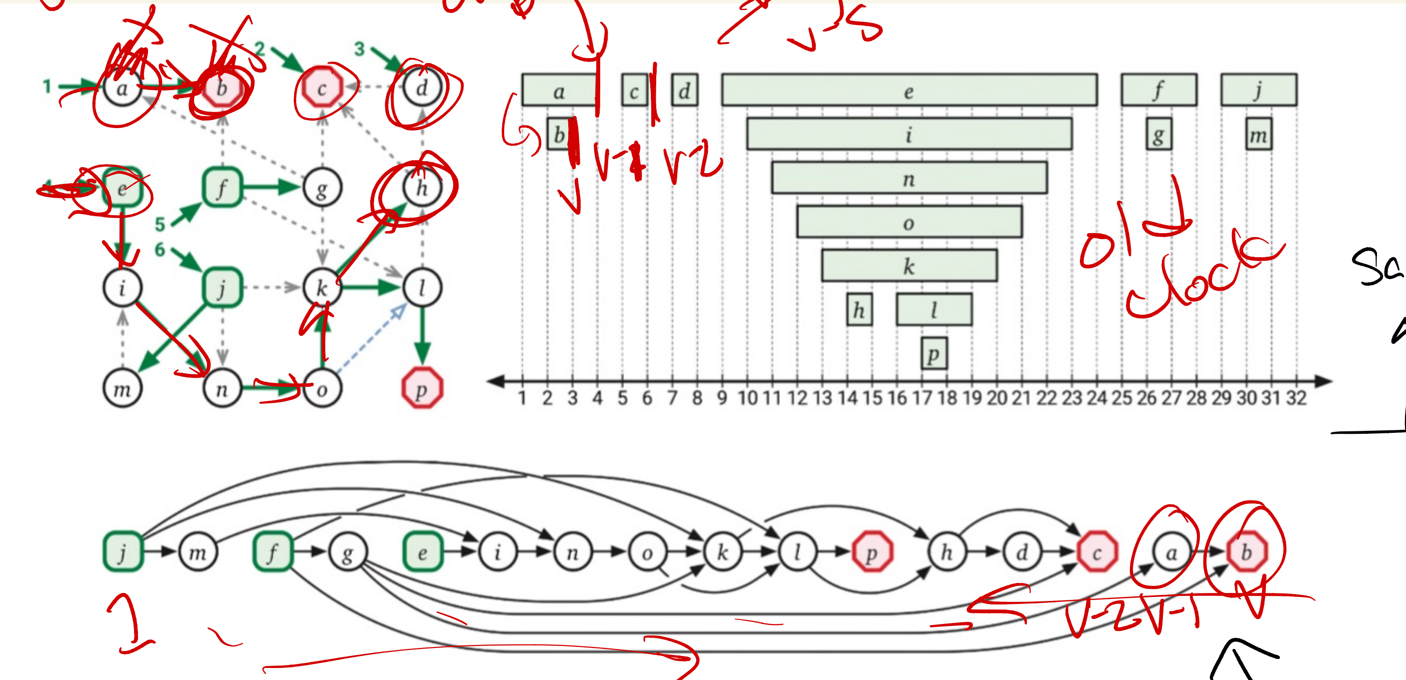
a → adj list   b → adj list   c

a.postorder

V-1 V-2

old clock

same



1

V-2 V-1 V

Figure 6.8. Reversed postordering of the dag from Figure 6.6.

New

Memoization + DP

Nice connection!

If the graph is a DAG, can do dynamic programming on it.

Why?

Think of the recurrences:

$$T(v) = \max_{\substack{(\text{predecessors} \\ \text{or successors } u \\ \text{of } v)}} \left\{ \begin{array}{l} T(u) \\ \text{lookup} + \\ \text{calculation} \end{array} \right\}$$

When will the algorithm get stuck?

↳ no base case

↳ infinite path, or cycle

Example: longest path in a DAG.

Usually → very hard.

Think backtracking for a moment, & fix a "target" vertex t.

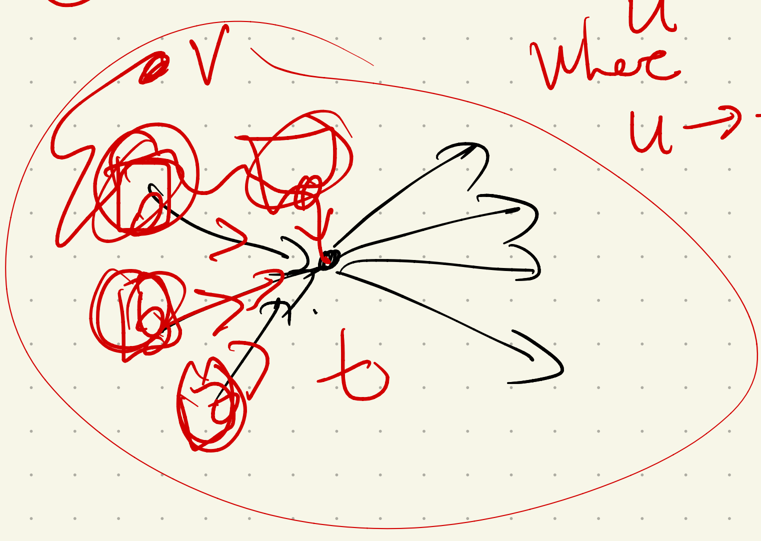
Let $LLP(\cancel{v})$ = longest path from v to t

t

6

$$= \max_{\substack{u \\ \text{where} \\ u \to t}} \left\{ dist(u) + 1 \right.$$

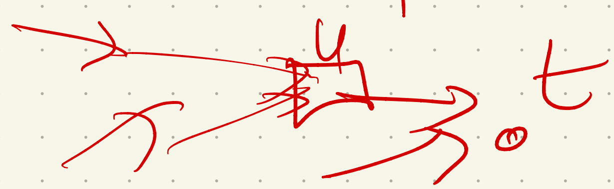
or $w(u \to t)$



Using this recursion:
"memoize" the value LLP:
Add a field to the vertex
& store it.
(Initially, = ∞) ← except v

longest v ⇝ t path

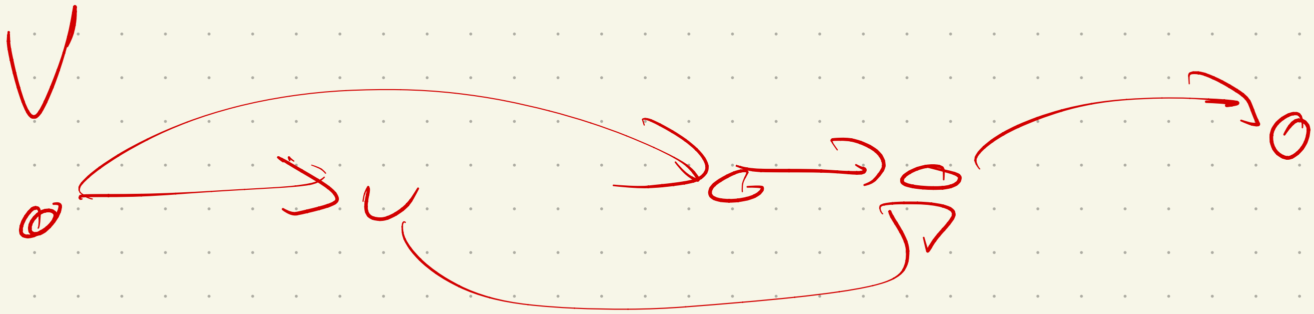
GetLongest(t):



if v = t:

v.length ← 0

otherwise:
max ← 0
for each edge u → t
GetLongest(u)
for each edge u → t
m ← max(m, u.length + 1)



compute top ordering

for each vertex (going in top order)

array S

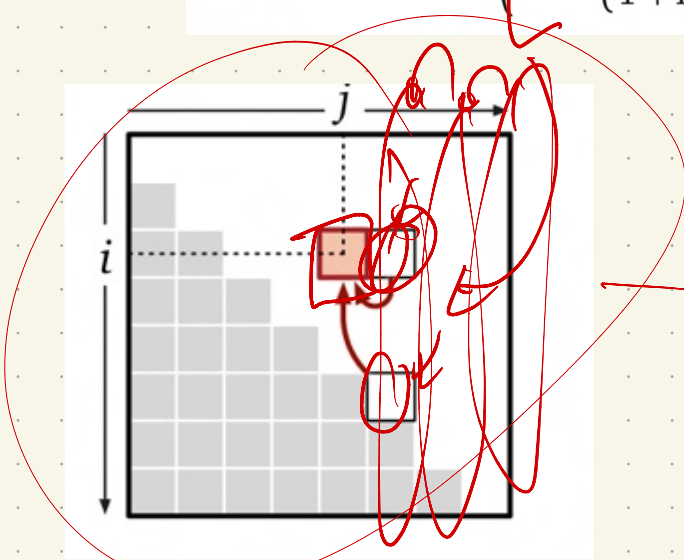
In principle, every DP we saw is working on a dependency graph of subproblems!

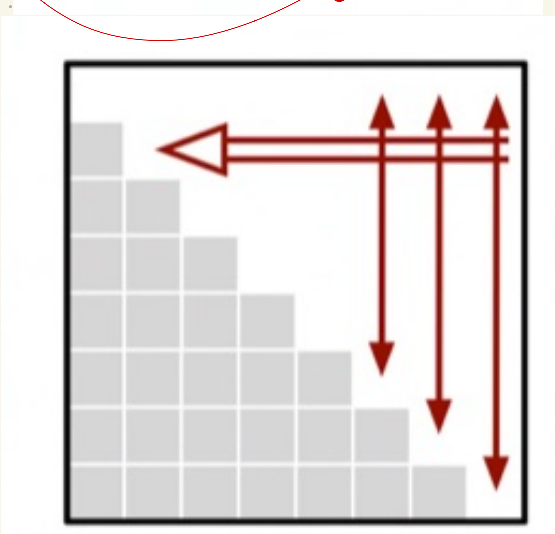
Recall: Longest Inc. Subsequence

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \ge A[j] \\ \max \begin{Bmatrix} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

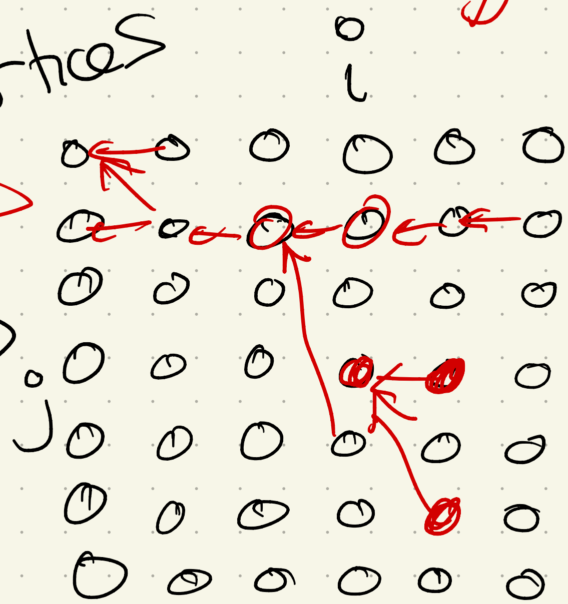
$E \le 2n^2$

$V = n^2$





vertices



edges:

$(j, j+1) \to (i,j)$ +1

$(i, j+1) \to (i,j)$ +0

do top ordering

$V = n^2$

$E \leq 2n^2$

$O(V+E)$ for top sort

↳ $O(n^2 + 2n^2)$

$= O(n^2)$

(same as nested for loops

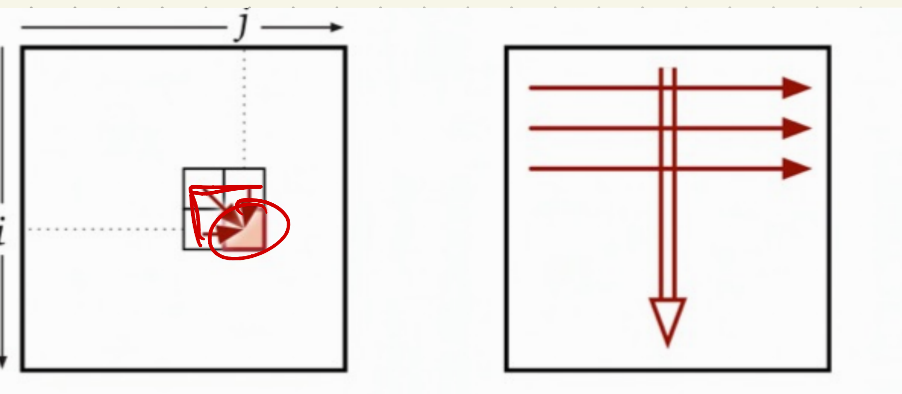
↳ these give $\leq$ top ordering)

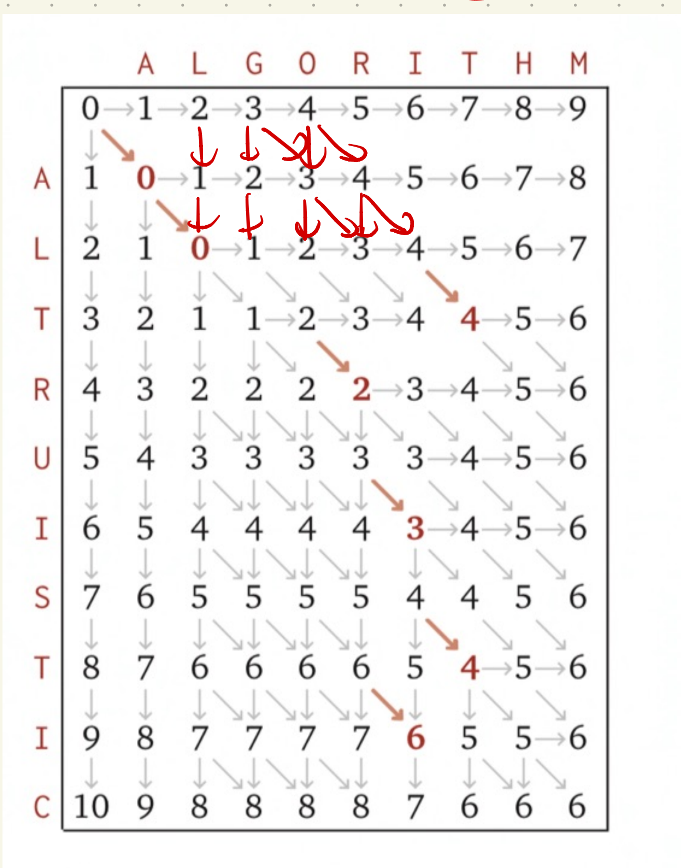
Edit distance:
he actually (sort of)
showed the graph!

insert
delete

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ \underline{Edit(i-1, j-1)} + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

maybe +1
if don't match



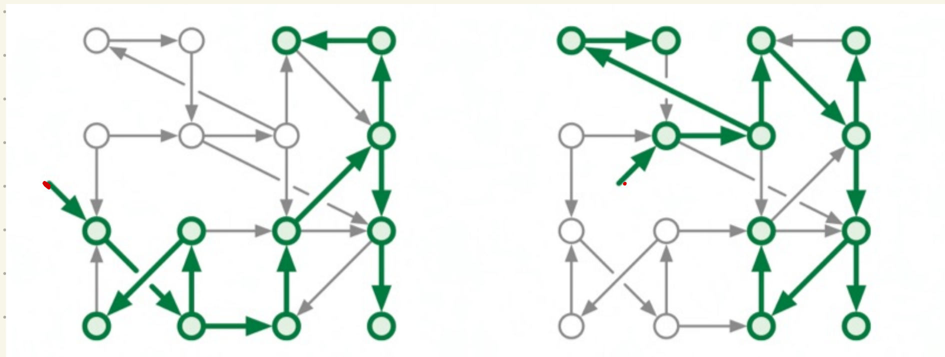


		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Strong connectivity

In an undirected graph, if $u \rightsquigarrow v$, then $v \rightsquigarrow u$.

Not true in directed case:



So 2 notions:

weak connectivity:

strong connectivity:

related: SCCs

Can actually order the strongly connected pieces of a graph:

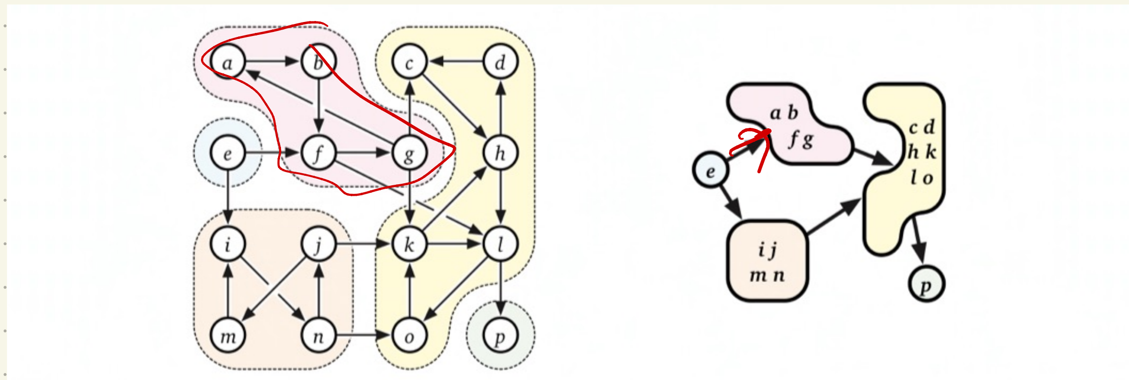


Figure 6.13. The strong components of a graph G and the strong component graph $scc(G)$.

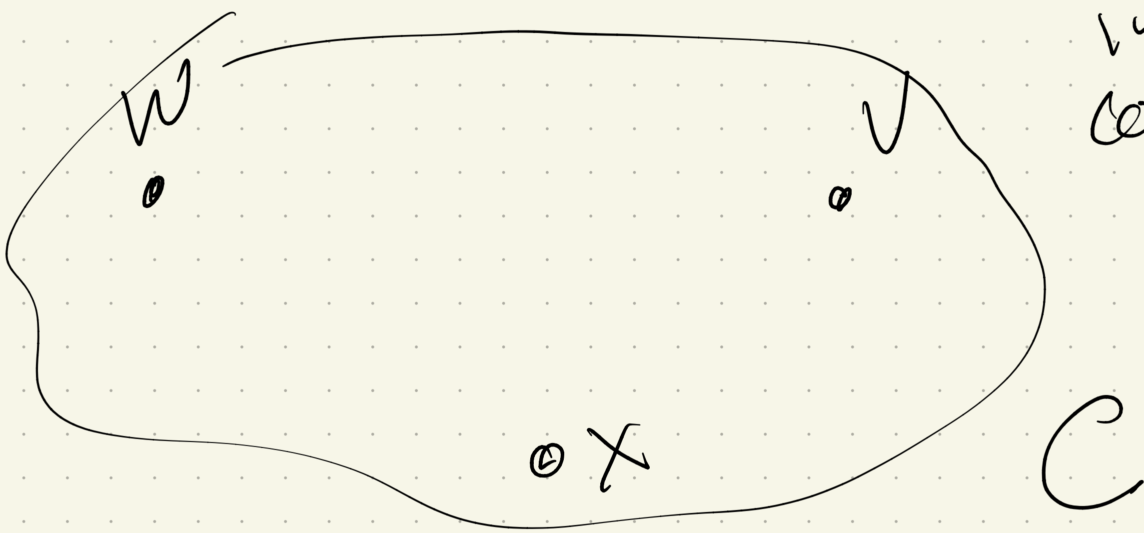
How?

- Well, each component either isn't connected, or only has 1-way edges. Why?

More formally:

Every strong cc must have at least one vertex with no parent.

Proof: Consider two vertices in component



Let x be first vertex in clock-order in scc:

Possible to compute SCCs
in $O(V+E)$ time.

Need good sinks!

DFS (rev(G))

↳ find sinks

Then, reverse back to
G & run DFS from
them.

(See book for details)

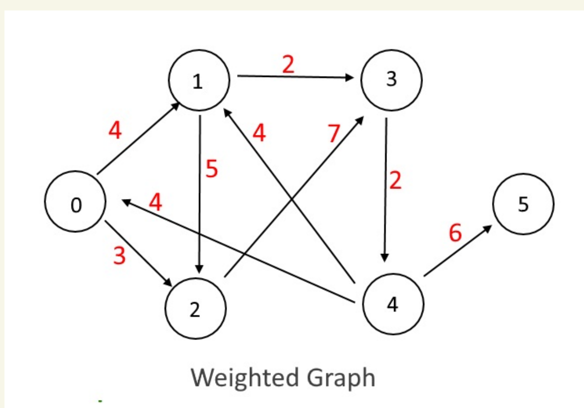
Next module:

Minimum Spanning trees

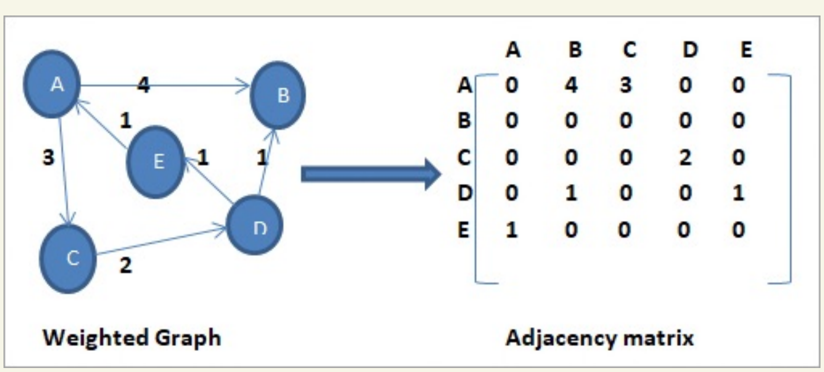
& shortest paths.

Both are on weighted graphs - so $G=(V,E)$, plus $w: E \to \mathbb{R}$ (or $\mathbb{R}^+$)

picture:



Weighted Graph



Weighted Graph

Adjacency matrix

Minimum Spanning Trees

Goal: Given a weighted Graph G, $w: E \to \mathbb{R}$ the weight function, find a Spanning tree T of G that minimizes:

$$w(T) = \sum_{e \in T} w(e)$$

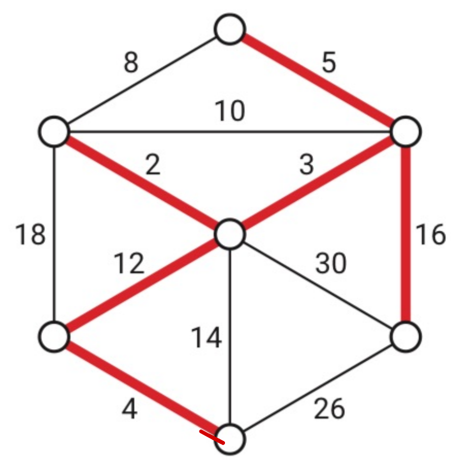


Figure 7.1. A weighted graph and its minimum spanning tree.

Motivation:

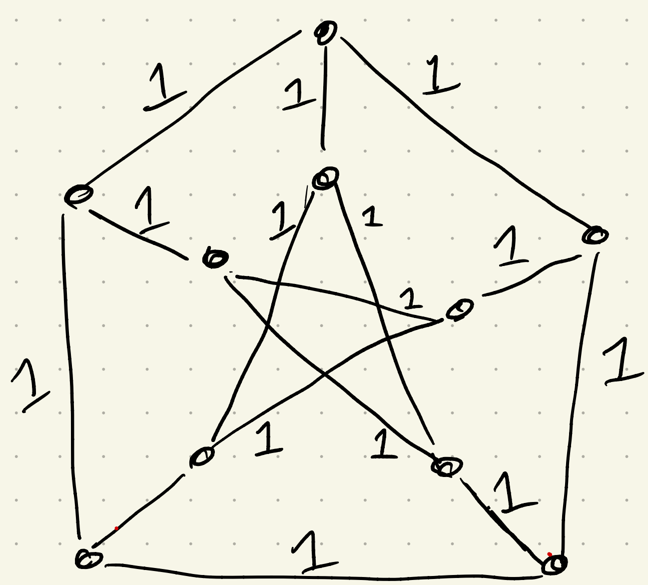
First:

Does it have to be a tree?

Second:

These are obviously not unique!

Ex:



tree?

Things will be cleaner if we have unique trees. So:

<u>Lemma</u>: Assuming all edge weights are distinct, then MST is unique.

<u>Pf</u>: By contradiction:

Suppose T & T' are both MSTs, with $T \neq T'$

- $T \cup T'$ contains a cycle
- That cycle must have 2 edges of equal weight

$\Rightarrow$ Contradiction!

Now, what if weights aren't unique?

Just need a way to consistently break ties.

$\underline{\text{ShorterEdge}(i, j, k, l)}$	
if $w(i, j) < w(k, l)$	then return (i, j)
if $w(i, j) > w(k, l)$	then return (k, l)
if $\min(i, j) < \min(k, l)$	then return (i, j)
if $\min(i, j) > \min(k, l)$	then return (k, l)
if $\max(i, j) < \max(k, l)$	then return (i, j)
⟨⟨*if max(i,j) > max(k,l)*⟩⟩	return (k, l)

So, takeaway:

Can assume unique MST.

Next: an algorithm.

The magic truth of MSTs:

You can be SUPER greedy.

Almost any natural idea will work!

This is highly unusual, & there's a reason for it:

these are a (rare) example of something called a matroid

(Way beyond this class...)