Algorithms (2020)

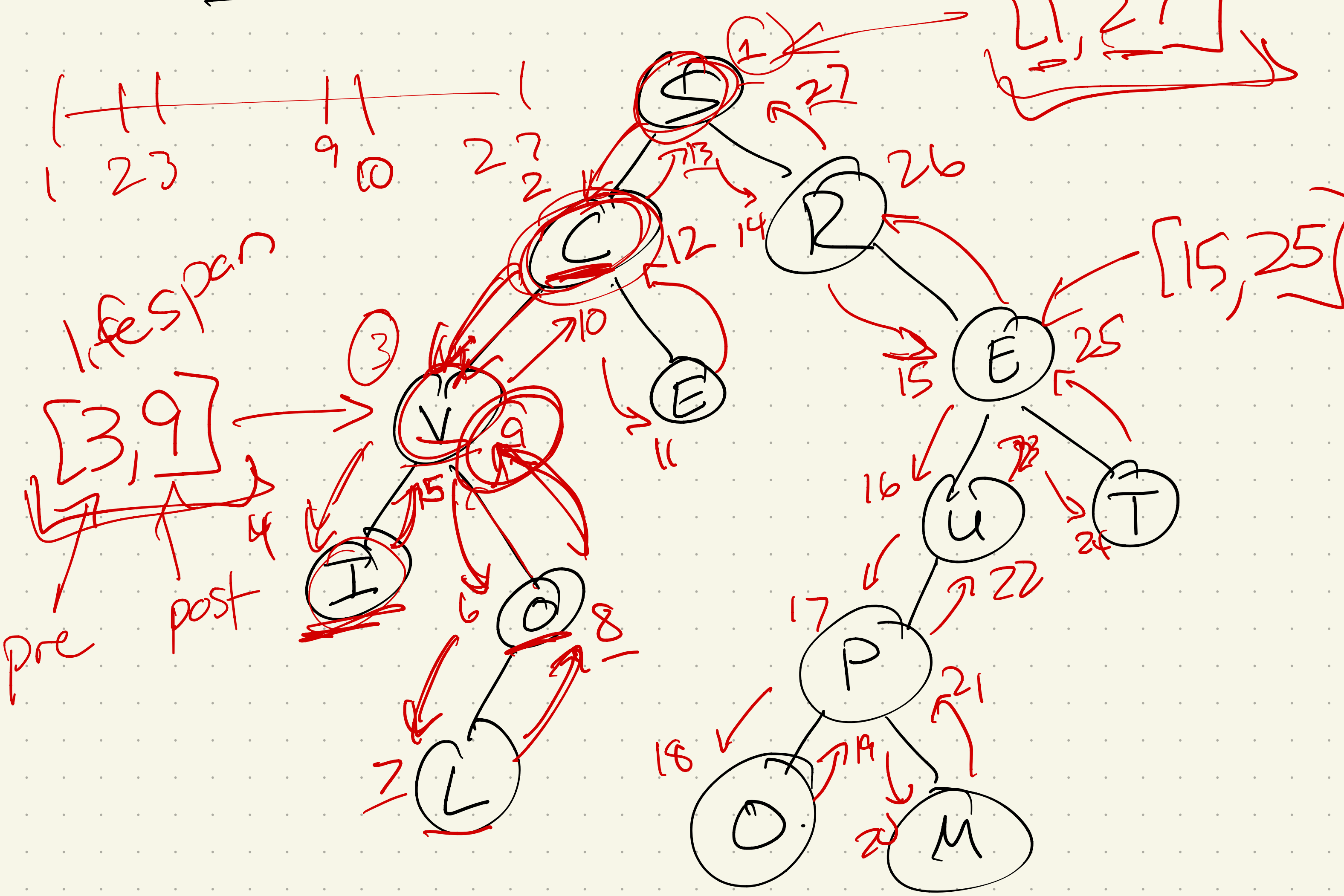
DFS + directed graphs

Recap

- No office hours today
- HW - due next Wed. ←
- Reading on Sunday (as usual)

Searching & directed graphs:

<u>Last time:</u> post order traversal



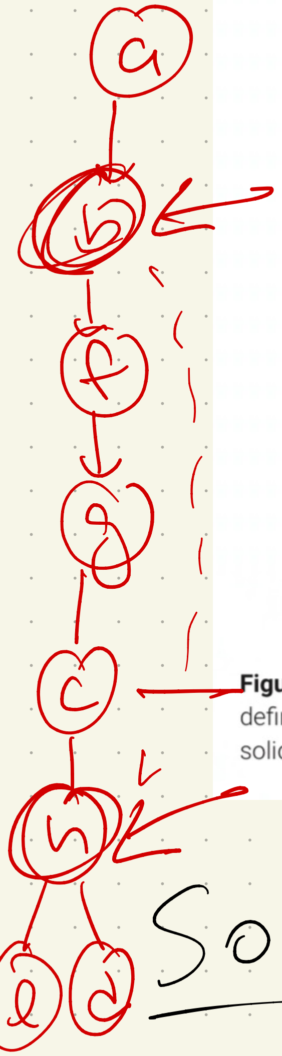
I L O V "time" = 9

- imagine a "clock" incrementing each time an <u>edge</u> is traversed:

activates when marked.

ends when last child recursion ends

Result: b: [2, 21] contains h's: [6, 17]



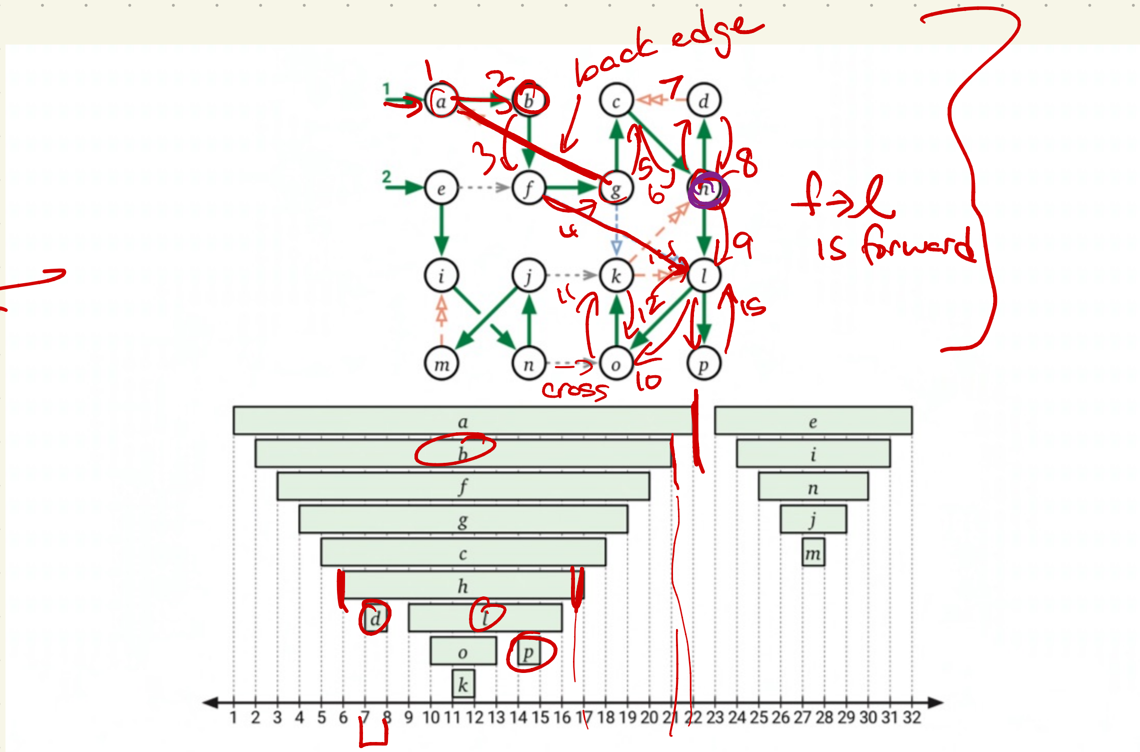


Figure 6.4. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

So: in DFS, this "lifespan" represents how long a vertex is on the stack.

Notation:

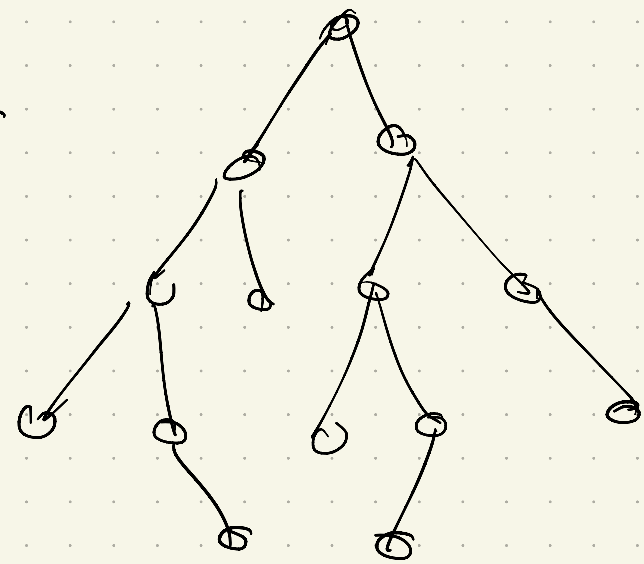
[v.pre, v.post]

If u is "below" v in the DFS tree

Note: In general graphs, post order traversal is not unique!

It was in BSTs:

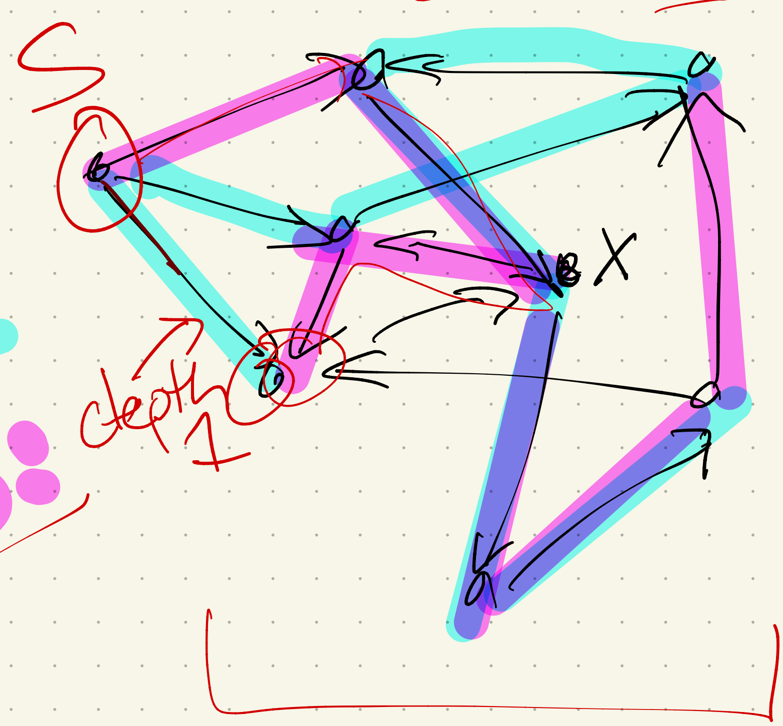
left, right, self



In graphs: each vertex has an unordered list

DFS tree

DFS:



Dfn: - tree edge
- forward edge
- back edge
- cross edge

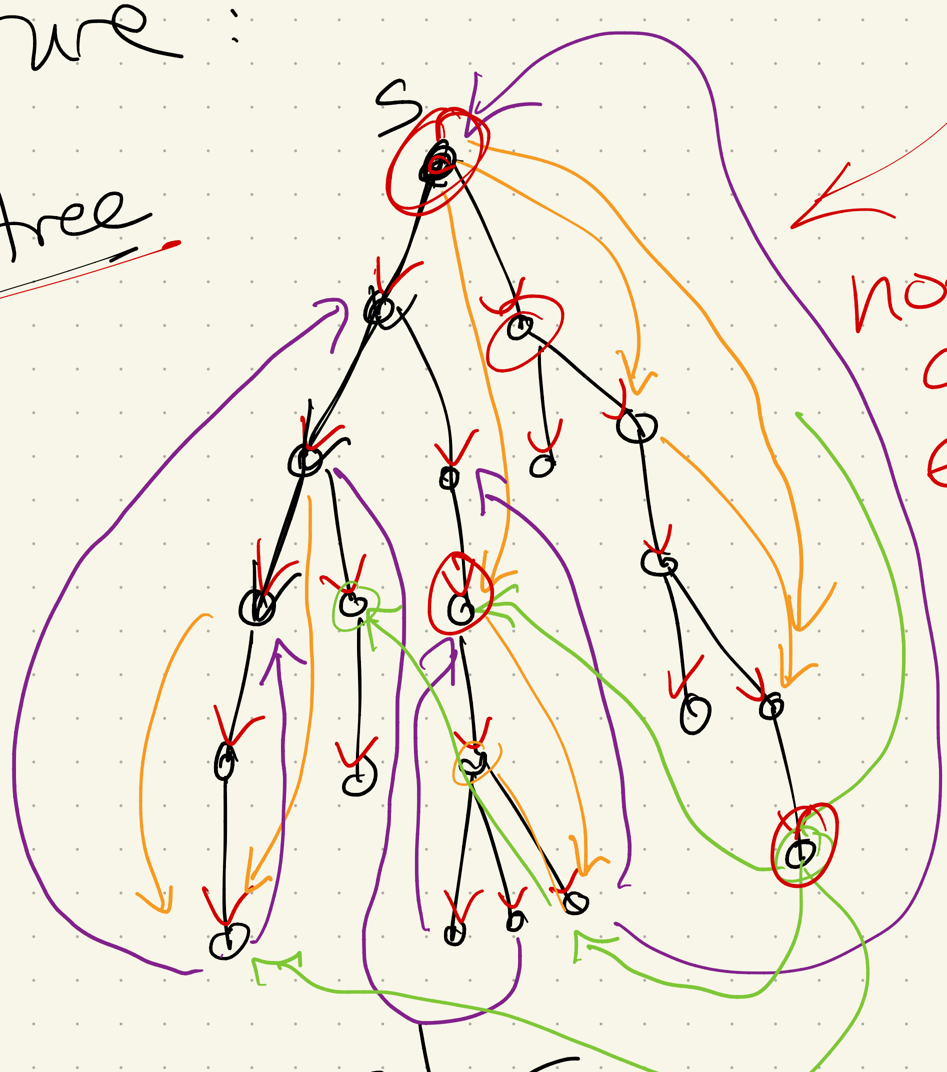
in your subtree

up & "left"

fix a tree

Picture:

DFS tree



not shown: other edges in G

Why? pre'd post order

In order

Topological ordering: Why?

Track dependencies:

- class prereqs
- compilers & #includes
- ordering evaluations of cells in a spreadsheet
- data analysis pipelines

⋮

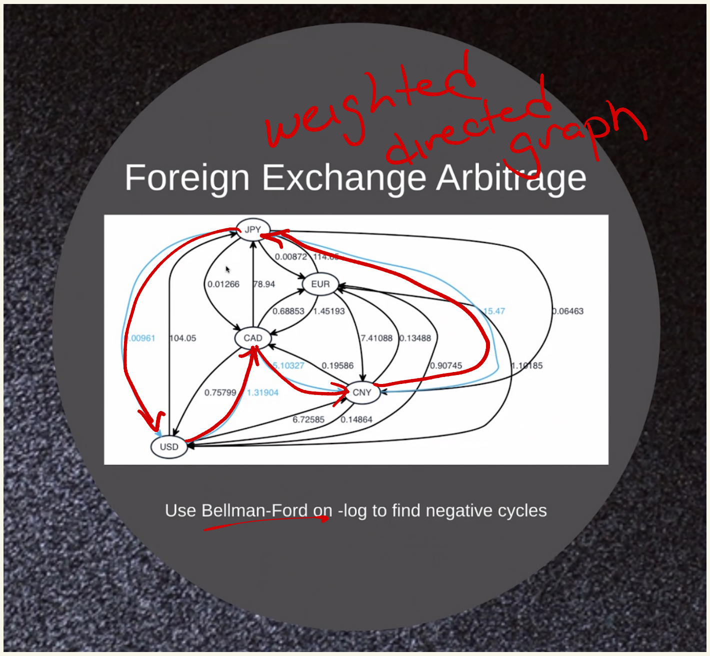
DAG: directed acyclic graph.

In general, cycles tend to be important.

Sometimes bad:

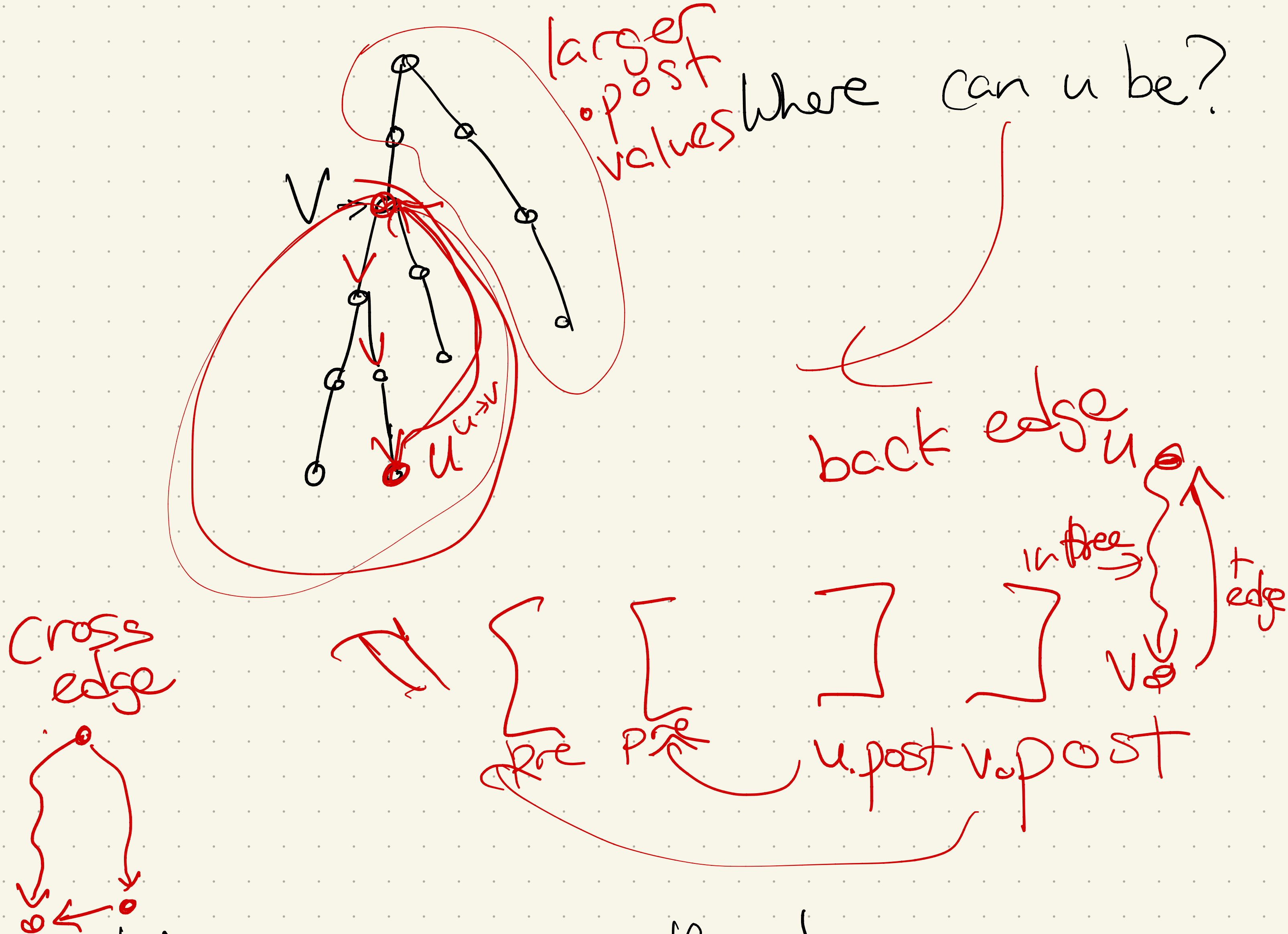
- topological ordering in a DAG
- longer run time

Sometimes good:



(taken from Monday's colloquium, by a person who works in high frequency trading)

Suppose $u \to v$, + $u.post < v.post$:
u was removed from "active" stack before v.



We can use this!
To detect cycles, + order (if not present).

Top sort DFS (by picture):

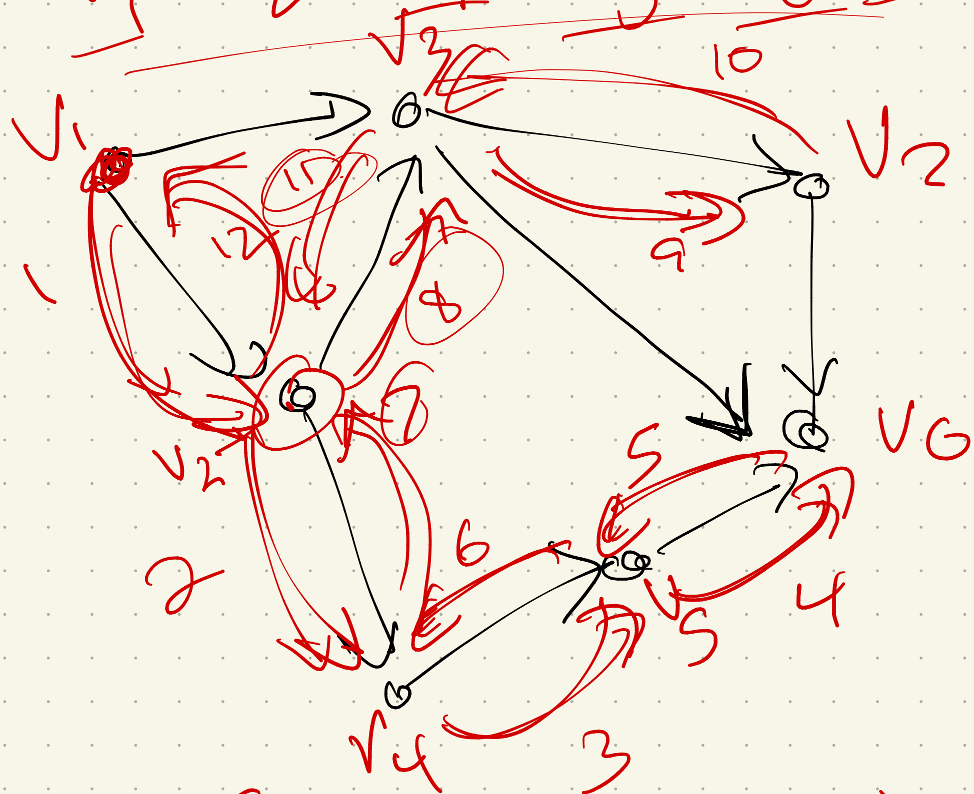
```
TopologicalSort(G):
    for all vertices v
        v.status ← New
    clock ← V
    for all vertices v
        if v.status = New
            clock ← TopSortDFS(v, clock)
    return S[1..V]
```

```
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

S is "Sort"

Order V_1 V_2 V_4 V_5 V_6 V_3 V_7



S: [1 2 6 3 4 5 7]

1 2 3 4 5 6 7

Memoization + DP

Nice connection!

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

Why?

backtracking

Think of the recurrences:

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

When will the algorithm get stuck?

cycles!

Example: longest path in a DAG.

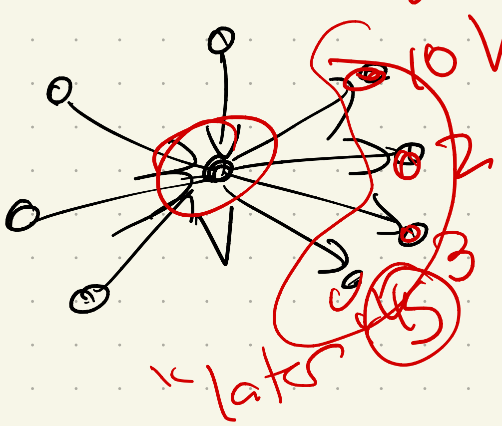
Usually → very hard.

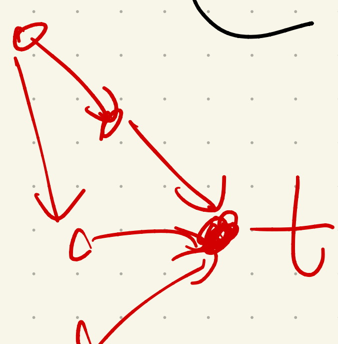
Think backtracking for a moment, & fix a "target" vertex t.

do this for every possible t, could solve

Let $LLP(v)$ = longest path from v to t

$$= \max_{\substack{\text{all nbrs} \\ w \text{ with} \\ v \to w}} \begin{cases} \text{if } v = t, & 0 \\ 1 + LLP(w) & \end{cases}$$





Using this recursion:

"memoize" the value LLP:

Add a field to the vertex & store it.

(Initially, $= -\infty$)

GetLongest(v):

if $v = t$:

return 0

otherwise:

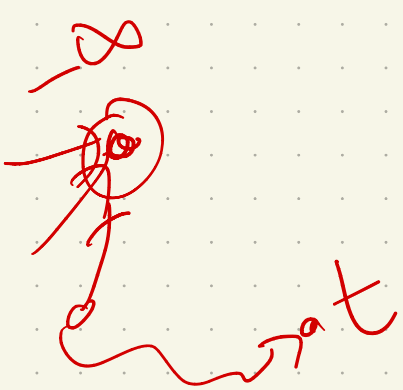
maxnbr $\leftarrow -\infty$

for every edge $u \to w$

if (Getlongest(w) + 1 > maxnbr)

maxnbr $\leftarrow$

return maxnbr.

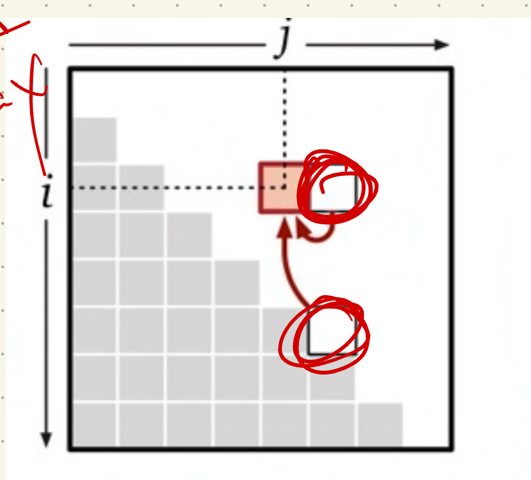


In principle, every DP we saw is working on a dependency graph of subproblems!

Recall: Longest Inc. Subsequence

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

2d array

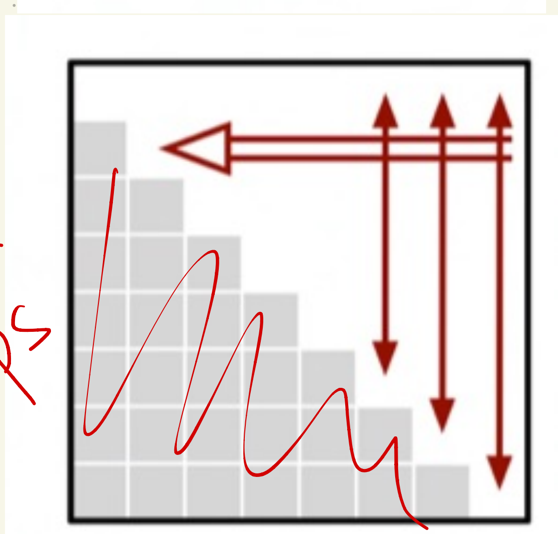


n^2 vertices

$\frac{n^2}{2}$ vertices

graph

for loops



edges:

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

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

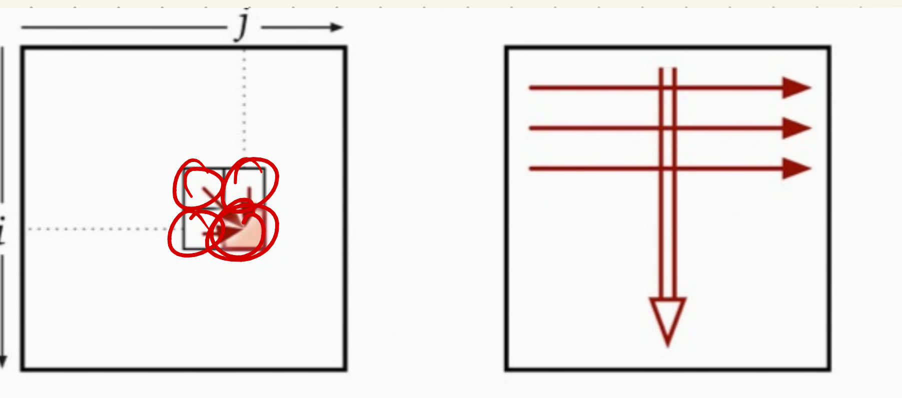
$2n^2$ edges

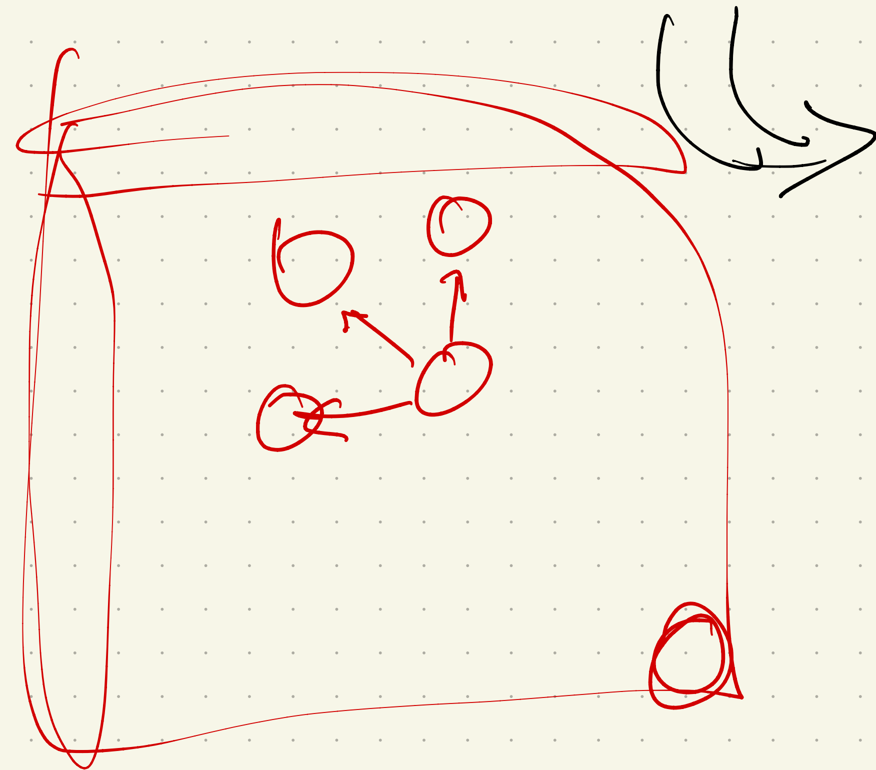
Edit distance:
he actually (sort of)
showed the graph!

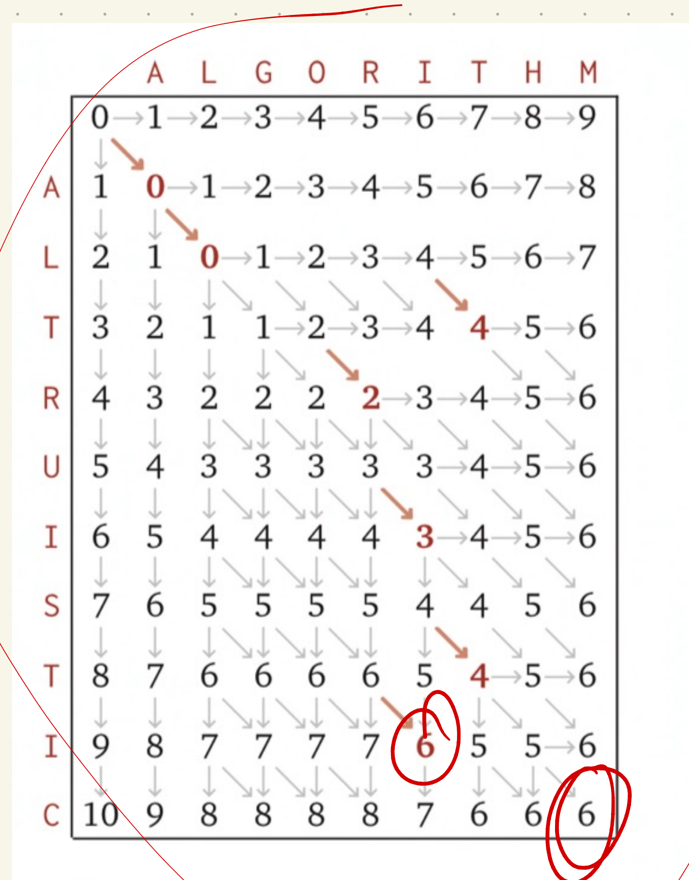
mn vertices

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

3 edges per "node"





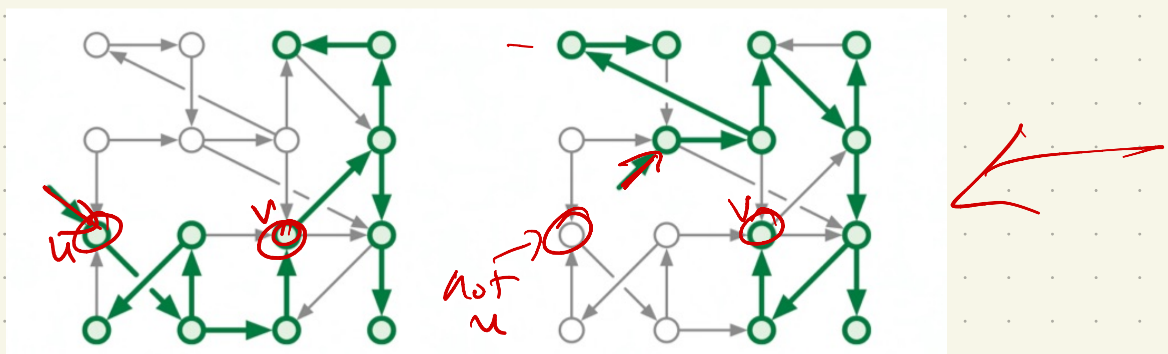


		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Strong connectivity

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

Not true in directed case:



So 2 notions:

weak connectivity:
$\forall$ pairs u, v, either $u \rightsquigarrow v$ or $v \rightsquigarrow u$

strong connectivity:
both $u \rightsquigarrow v$ & $v \rightsquigarrow u$

related: SCCs

Can actually order the strongly connected pieces of a graph:

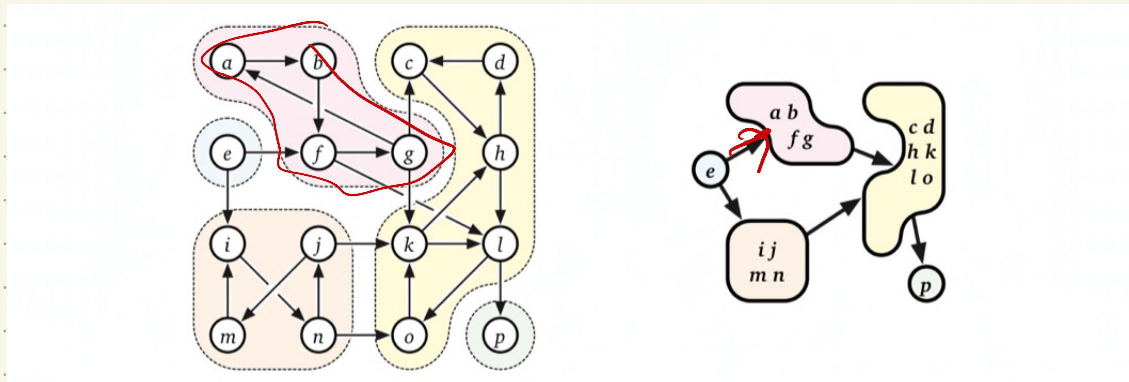
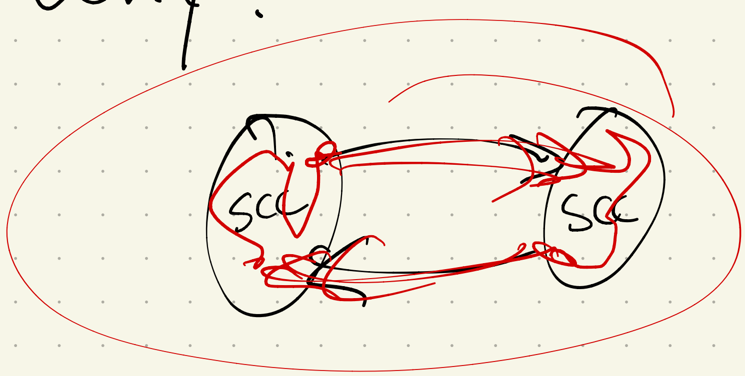


Figure 6.13. The strong components of a graph G and the strong component graph $scc(G)$.

How?

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



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

Sorry - did not assign this one!

But feel free to read anyway. :)

modify DFS again

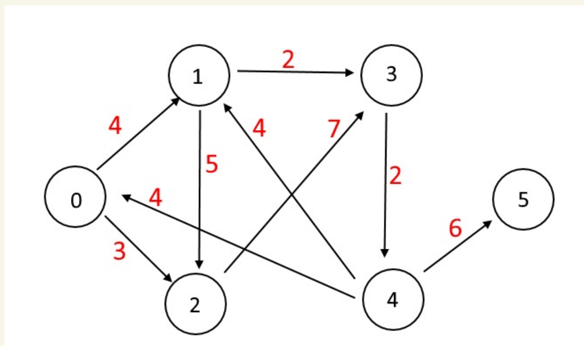
Next module:

Minimum Spanning trees

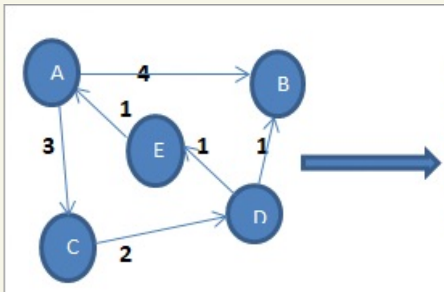
& shortest paths.

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

<u>picture</u>:



Weighted Graph



Weighted Graph

	A	B	C	D	E
A	0	4	3	0	0
B	0	0	0	0	0
C	0	0	0	2	0
D	0	1	0	0	1
E	1	0	0	0	0

Adjacency matrix