

Algorithms - Spring '25

Directed  
graphs



Recap

# Chapter 6:

All about directed graphs!

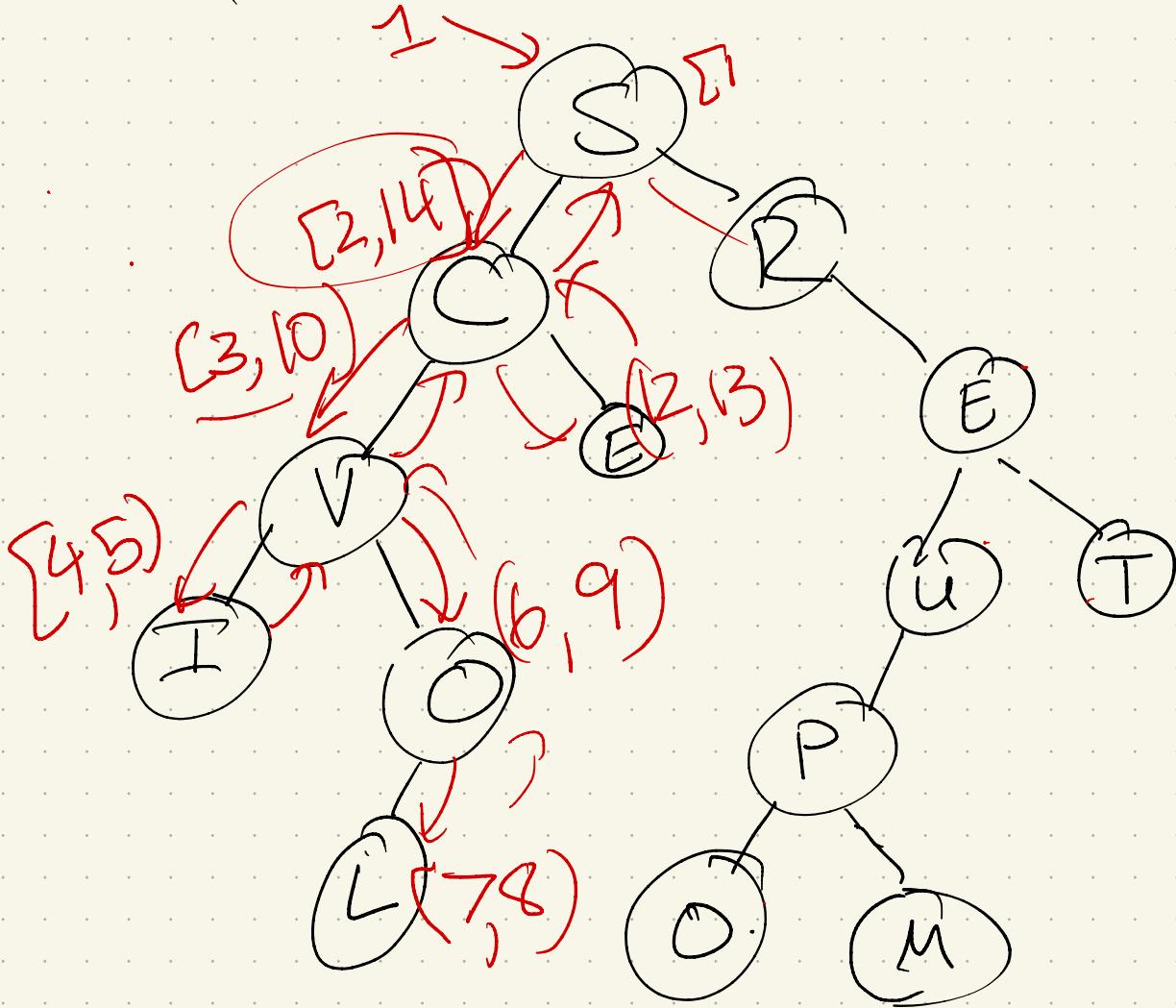
First, though, some things to recall: graph traversals.

- Pre-order: visit  $\checkmark$  children  
visit  $\checkmark$
- Post-order: visit all children  
visit  $\checkmark$



We can use these  
on more general graphs!

Searching & directed graphs:  
Recall: post order traversal



- imagine a "clock" incrementing  
each time an edge is traversed:

# General graphs:

DFSALL( $G$ ):

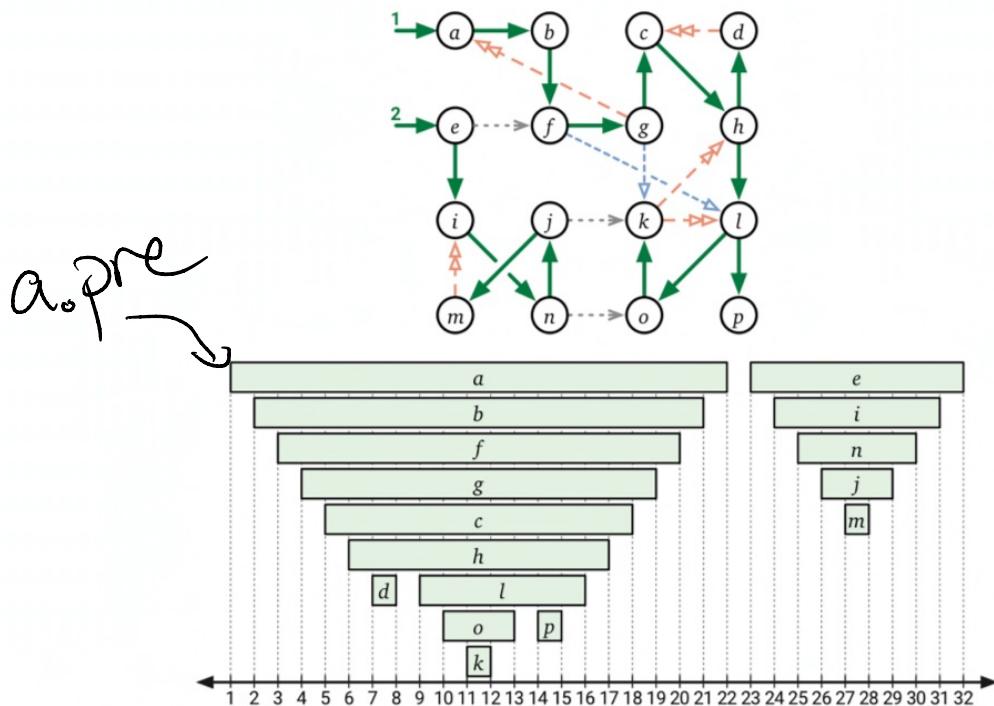
```

 $clock \leftarrow 0$ 
for all vertices  $v$ 
    unmark  $v$ 
for all vertices  $v$ 
    if  $v$  is unmarked
         $clock \leftarrow \text{DFS}(v, clock)$ 
    
```

DFS( $v, clock$ ):

```

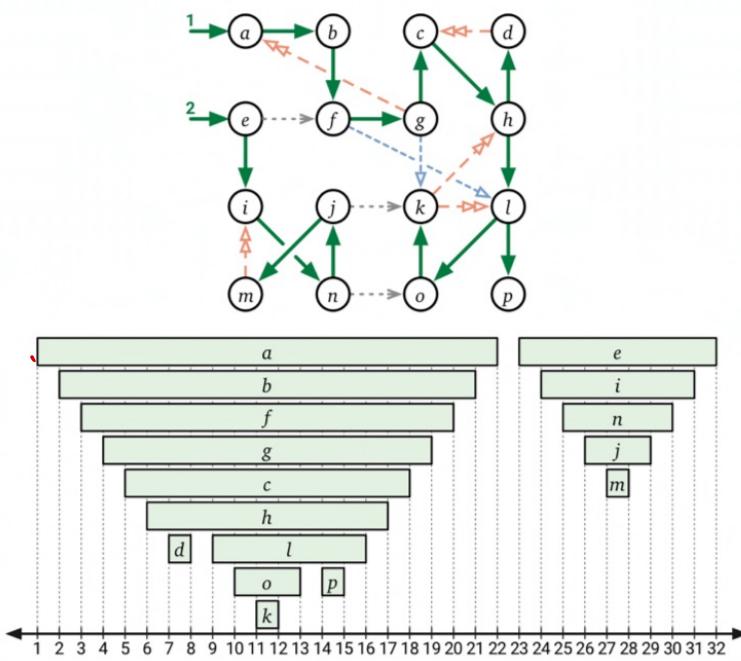
mark  $v$ 
 $clock \leftarrow clock + 1; v.\text{pre} \leftarrow clock$ 
for each edge  $v \rightarrow w$ 
    if  $w$  is unmarked
         $w.\text{parent} \leftarrow v$ 
         $clock \leftarrow \text{DFS}(w, clock)$ 
 $clock \leftarrow clock + 1; v.\text{post} \leftarrow clock$ 
return  $clock$ 
    
```



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

clock: 0    a is unmarked;  
 $\text{DFS}(a, \text{clock})$

# Result:



**Figure 6.4.** A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeijnm* and the postordering *dkoplhcgfbamnjie*. 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:

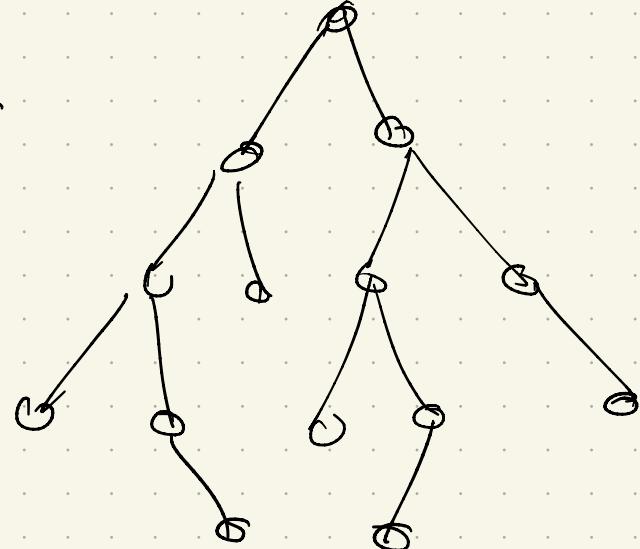
$[v_{\text{pre}}, v_{\text{post}}]$

Pre order visit time

postorder visit time

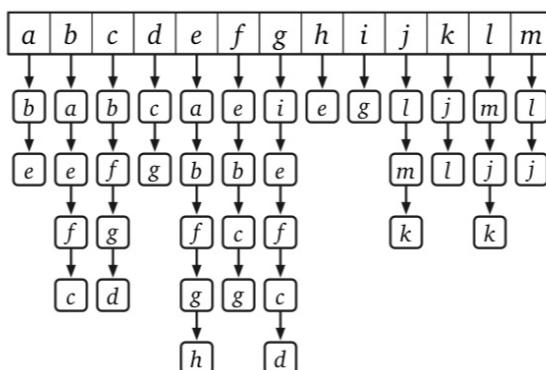
Note: In general graphs,  
post order traversal is  
not unique!

It was in BSTs:



In graphs:

Just use adj. list order



DFSALL( $G$ ):

$clock \leftarrow 0$   
for all vertices  $v$   
    unmark  $v$

for all vertices  $v$   
    if  $v$  is unmarked  
         $clock \leftarrow DFS(v, clock)$

DFS( $v, clock$ ):

mark  $v$

$clock \leftarrow clock + 1; v.pre \leftarrow clock$

for each edge  $v \rightarrow w$

    if  $w$  is unmarked

$w.parent \leftarrow v$

$clock \leftarrow DFS(w, clock)$

$clock \leftarrow clock + 1; v.post \leftarrow clock$

return  $clock$

Dfn :- tree edge : (green)

$u \succ v$

DFS(u) calls DFS(v) &  $v \not\in$  not visited before

- forward edge : (blue)

$v \in$  interval contained in u's,

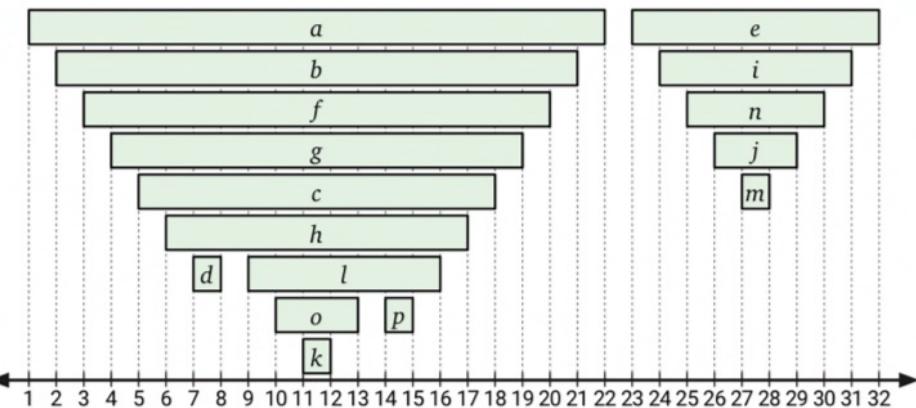
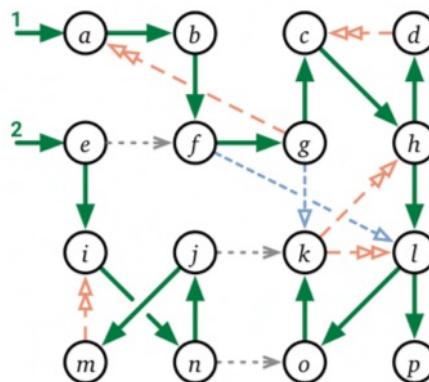
$v$  not visited yet, u doesn't call v

- back edge : (red)

$v$  already active when u is called

- cross edge : (grey)

$v$  is done before u is called



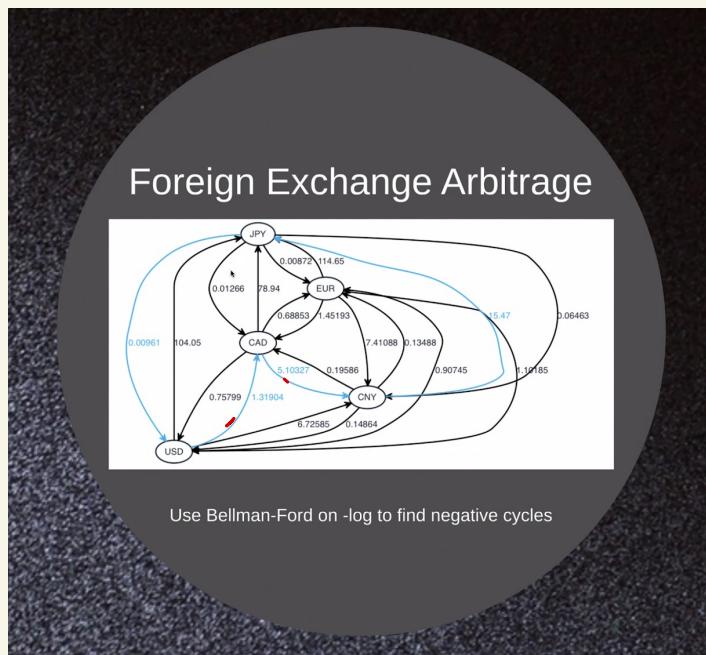
# Finding cycles

In general, cycles tend to be important.

Sometimes bad:

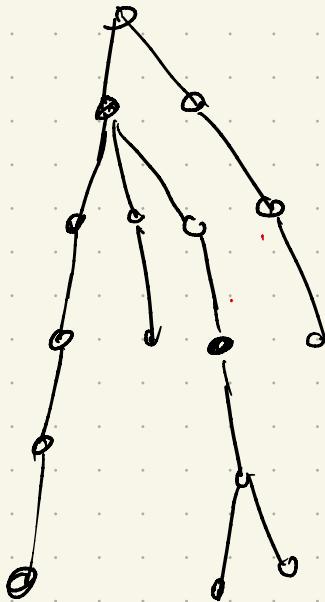
- topological ordering in a DAG (see next slide)
- longer run time
  - ↳ see Dyn. Pro

Sometimes good:

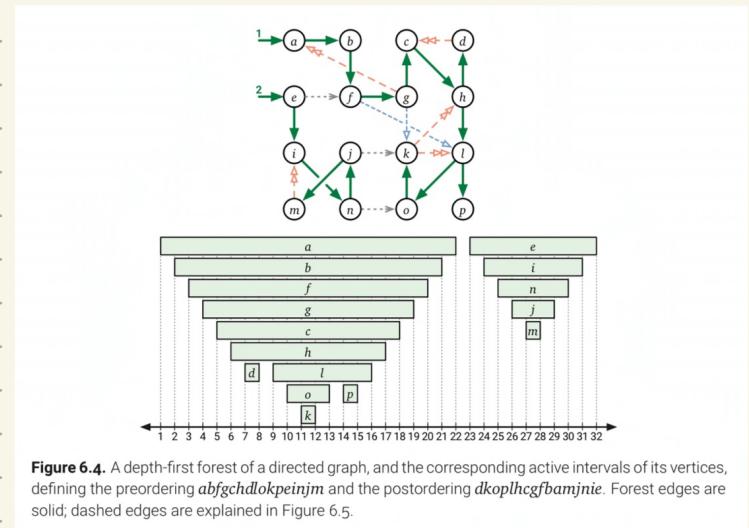


(Taken from  
a talk I saw,  
by a person  
who works in  
high frequency  
trading)

Suppose  $u \rightarrow v$ ,  $u.\text{post} < v.\text{post}$ :  
 $u$  was removed from "active stack" before  $v$ .



Where can  $u$  be?



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

# Topological ordering: Why?

Track dependencies:

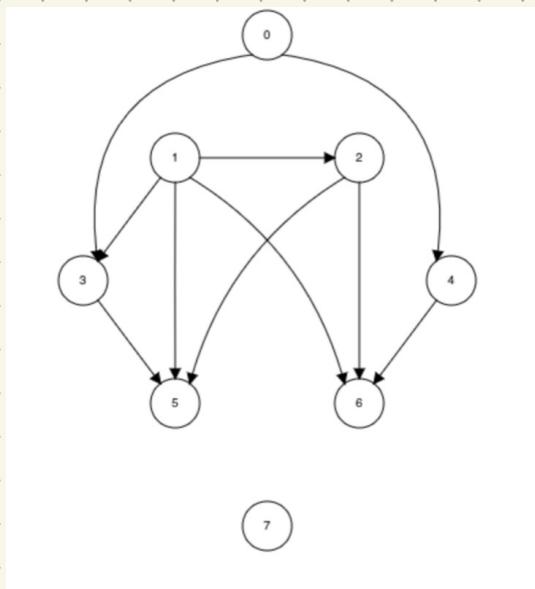
- class prereqs
- compilers + #includes
- ordering evaluations of cells in a spreadsheet
- data analysis pipelines

o  
e  
a

Often, in all these settings, the goal is to find a processing order that works.

↳ lays out dependencies so precursor is evaluated first!

# A simple example



Post-ordering:  
visit children  
then mark self

Here:

Note: this puts a vertex "after" anything it can reach!  
So, to get a post-ordering:  
reverse it!

Why does it work?  
(blc DFS)

# Top sort DFS : making it more precise

TOPLOGICALSORT( $G$ ):

```

for all vertices  $v$ 
     $v.status \leftarrow NEW$ 
     $clock \leftarrow V$ 
for all vertices  $v$ 
    if  $v.status = NEW$ 
         $clock \leftarrow \text{TOPSORTDFS}(v, clock)$ 
return  $S[1..V]$ 

```

TOPSORTDFS( $v, clock$ ):

```

 $v.status \leftarrow ACTIVE$ 
for each edge  $v \rightarrow w$ 
    if  $w.status = NEW$ 
         $clock \leftarrow \text{TOPSORTDFS}(v, clock)$ 
    else if  $w.status = ACTIVE$ 
        fail gracefully
     $v.status \leftarrow FINISHED$ 
 $S[clock] \leftarrow v$ 
 $clock \leftarrow clock - 1$ 
return  $clock$ 

```

Figure 6.9. Explicit topological sort

# Unpacking his figure:

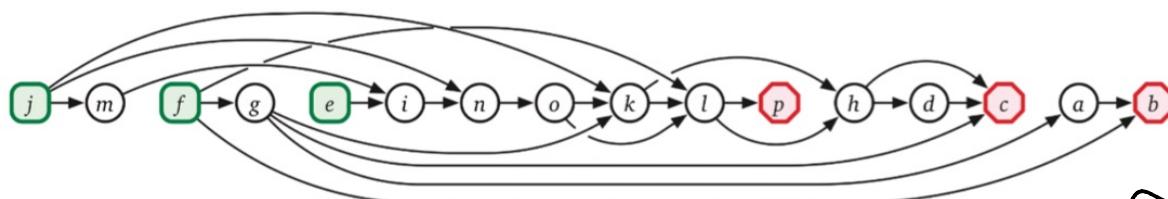
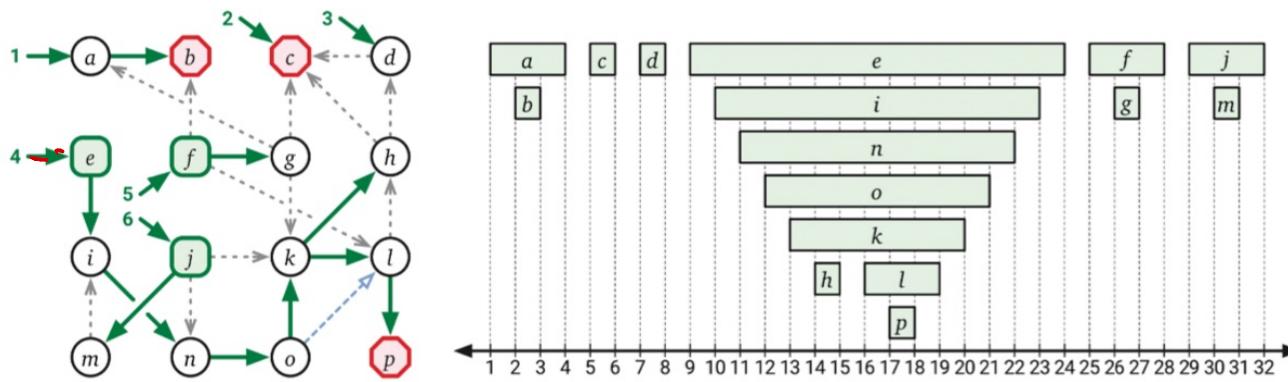


Figure 6.8. Reversed postordering of the dag from Figure 6.6.

New

Same

# Memoization & DP

Nice connection!

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

Why?

Think of the recurrences:

$$T(v) = \max_{\substack{(\text{predecessors} \\ \text{or successors } u \\ \text{of } v)}} \left\{ T(u) \right\}$$

lookup +  
calculation

When will the algorithm  
get stuck?

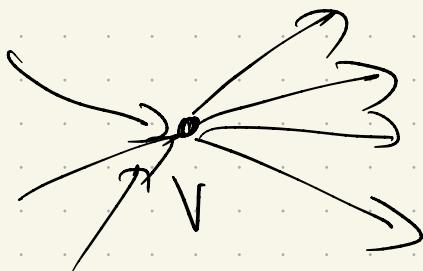
Example: longest path in  
a DAG.

Usually  $\rightarrow$  very hard.

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

Let  $LLP(v) = \underbrace{\text{longest path}}_{\text{from } v \text{ to } t}$

$\equiv \max$



Using this recursion:

"memoize" the value LLP:

Add a field to the vertex  
& store it.

(Initially,  $=$ )

Get Longest( $V$ ):

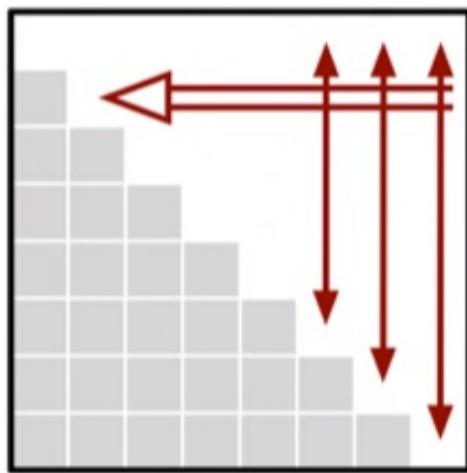
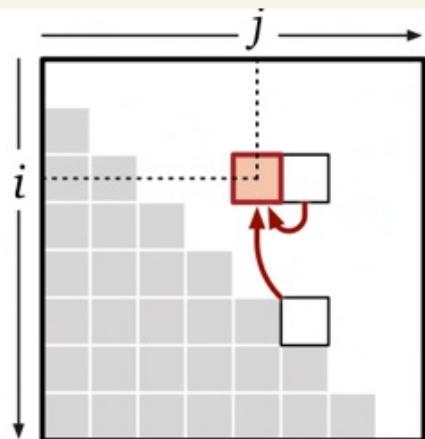
If  $V = t$ :

otherwise:

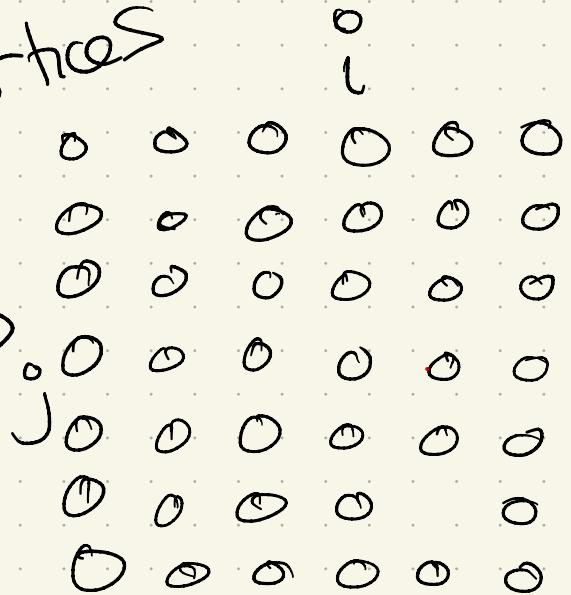
In principle, every DP we saw is working on a dependency graph of subproblems!

Recall: Longest Inc Subsequence

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



vertices



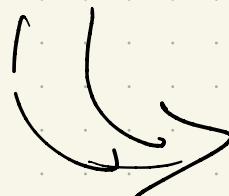
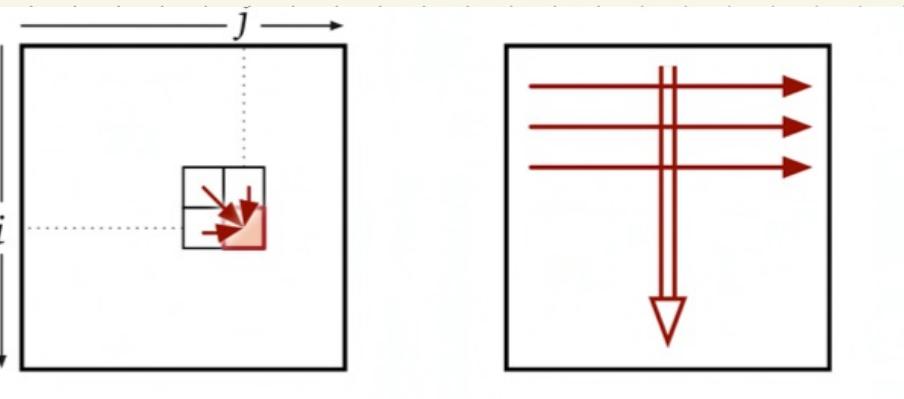
edges:

$$(i, j) \rightarrow$$

$$(i, j') \rightarrow$$

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

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

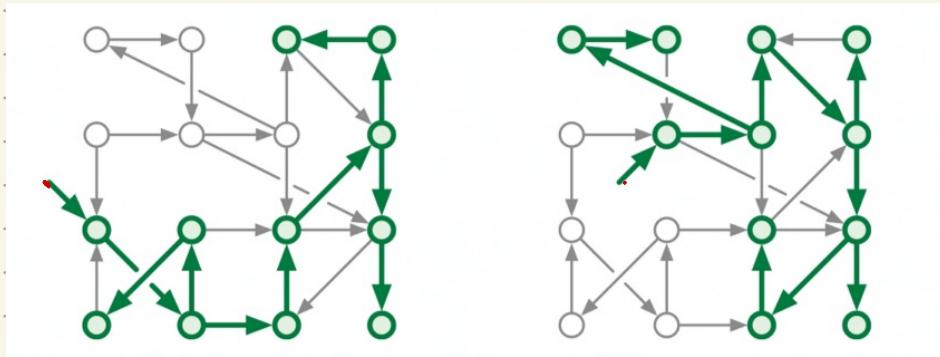


	A	L	G	O	R	I	T	H	M	
0	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	
2	1	0	1	2	3	4	5	6	7	
3	2	1	1	2	3	4	4	5	6	
4	3	2	2	2	2	3	4	5	6	
5	4	3	3	3	3	3	4	5	6	
6	5	4	4	4	4	3	4	5	6	
7	6	5	5	5	5	4	4	5	6	
8	7	6	6	6	6	5	4	5	6	
9	8	7	7	7	7	6	5	5	6	
10	9	8	8	8	8	7	6	6	6	

# Strong connectivity

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

Not true in directed case:



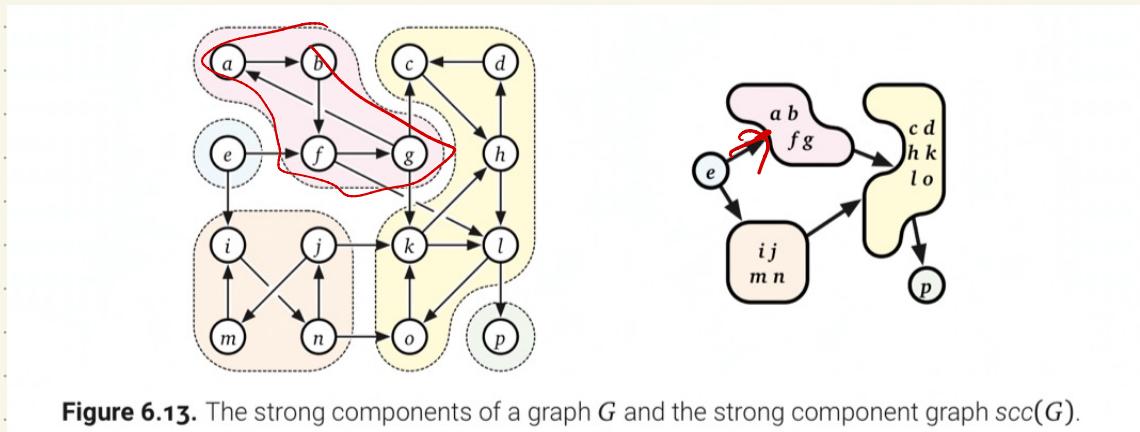
So 2 notions:

weak connectivity:

strong connectivity:

related: SCCs

Can actually order the  
Strongly connected pieces  
of a graph:



How?

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

( $scc$ )

( $scc$ )

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

Sorry - did not assign  
this one!

But feel free to read  
anyway. :)

Next module:

# Minimum Spanning trees

& shortest paths.

Both are on weighted

graphs - so  $G = (V, E)$ ,  
plus  $w: E \rightarrow \mathbb{R}$  (or  $\mathbb{R}^+$ )

Picture:

