Algorithms - Spring '25

Directed graphs

Recap

Next chapter:

All about directed graphs!

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

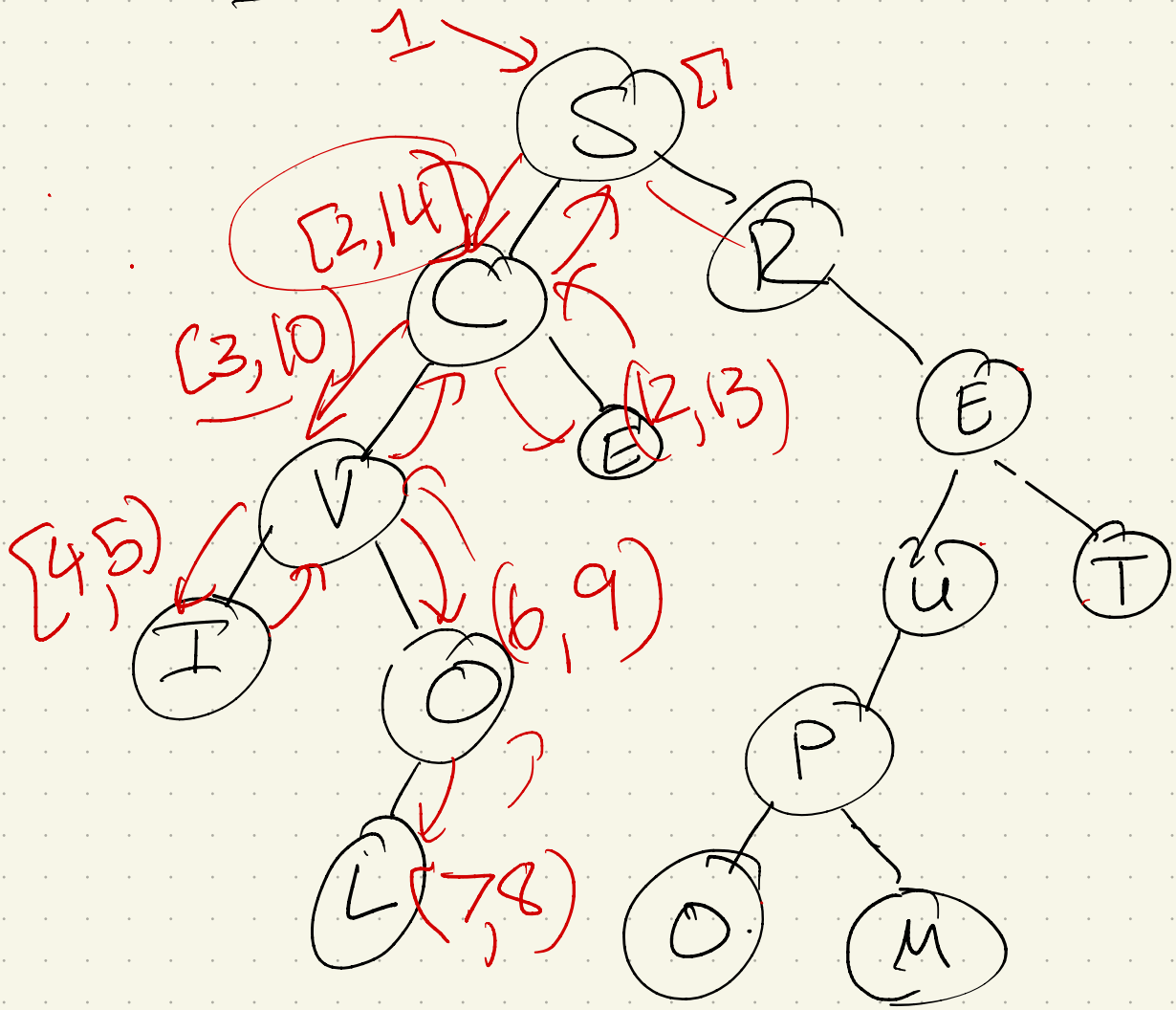
- Pre-order: visit v
  visit children
- Post-order: visit all children
  visit v

- In-order: (binary tree)

left
self
right

Searching a directed graphs:

Recall: post order traversal



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

Result:

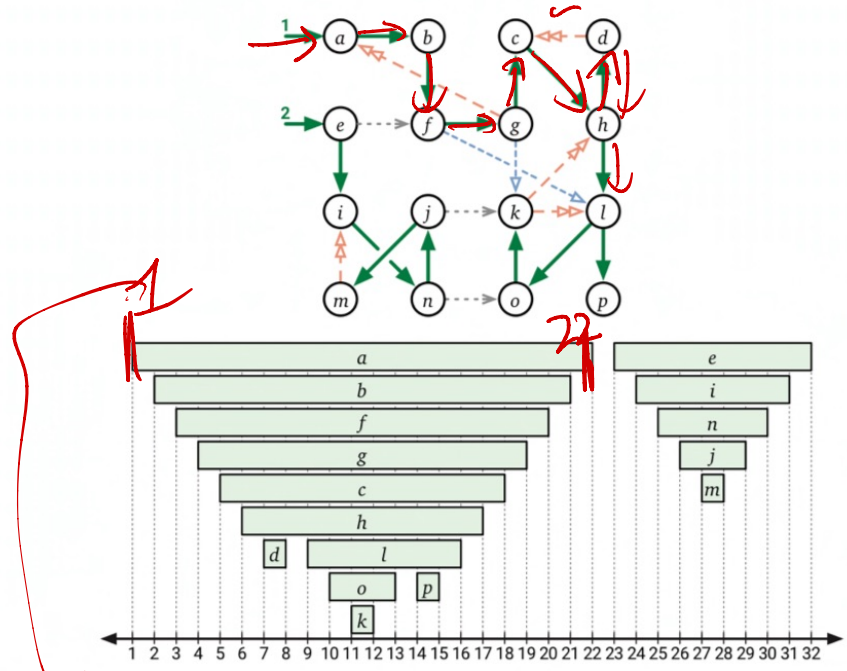


Figure 6.4. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

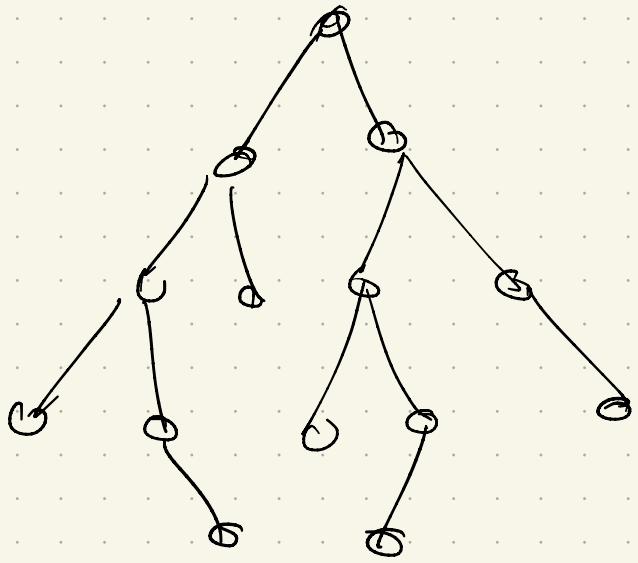
So: in DFS, this "lifespan" represents how long a vertex is on the stack.

Notation:

[v.pre, v.post]

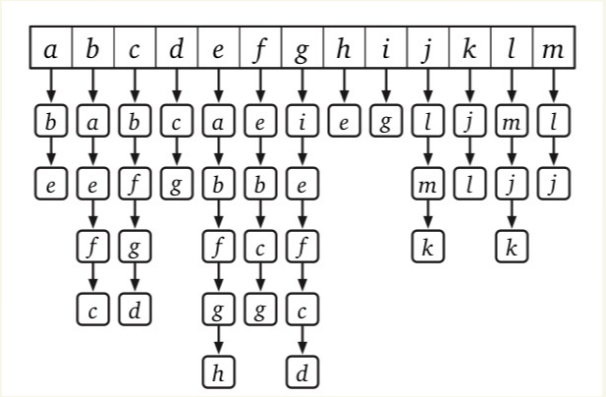
Note: In general graphs, post order traversal is not unique!

It was in BSTs:



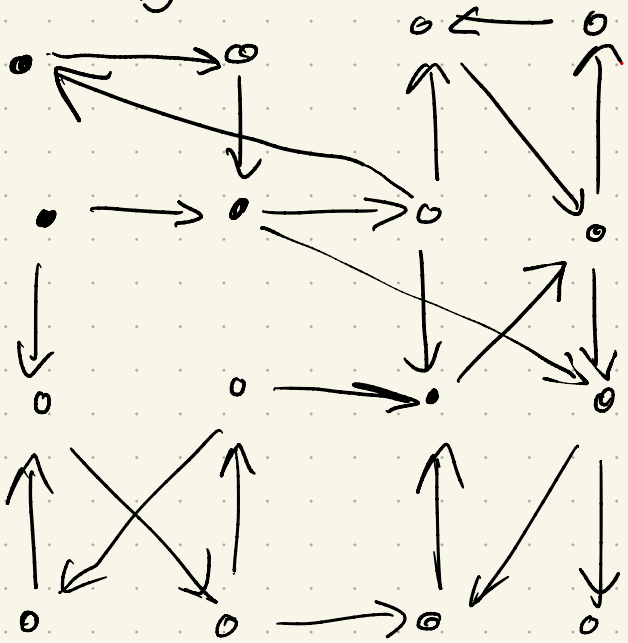
In graphs:
Just use adj. list order.

```
DFS(v):
    if v is unmarked
        mark v
        for each edge v→w
            DFS(w)
```



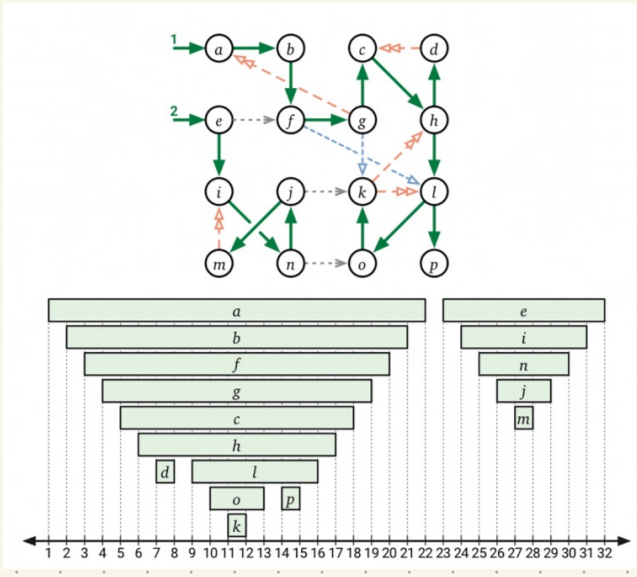
Dfn :- tree edge
- forward edge
- back edge
- cross edge

Picture : S



DFS tree

Clock reference :



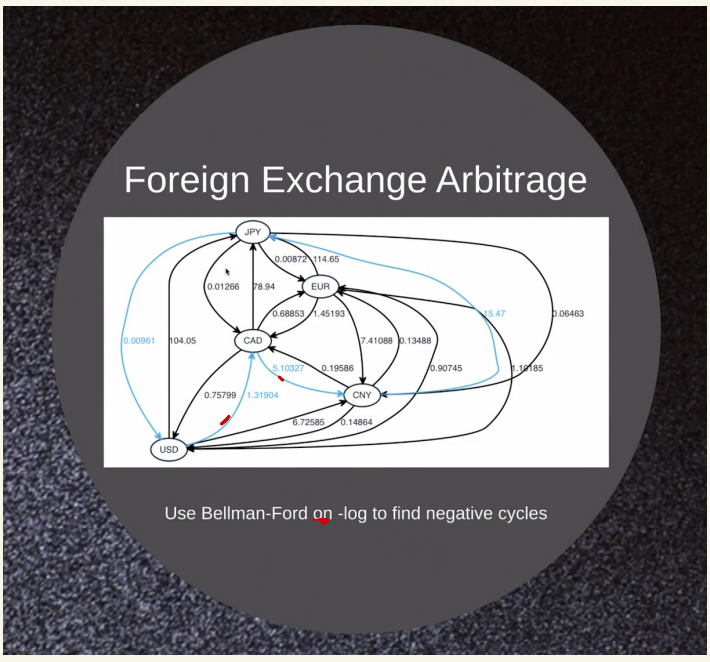
Finding cycles

In general, cycles tend to be important.

Sometimes bad:

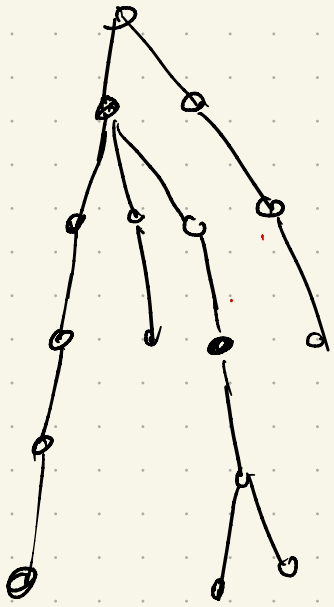
- topological ordering in a DAG (see next slides)
- longer run time
  ↳ see Dyn. Pro.

Sometimes good:



(taken from colloquium last year, 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.



Where can u be?

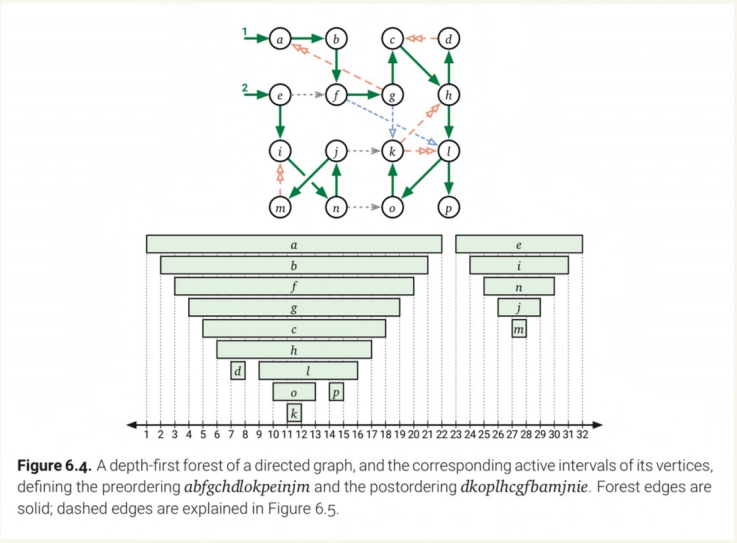


Figure 6.4. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

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

Topological ordering: Why?

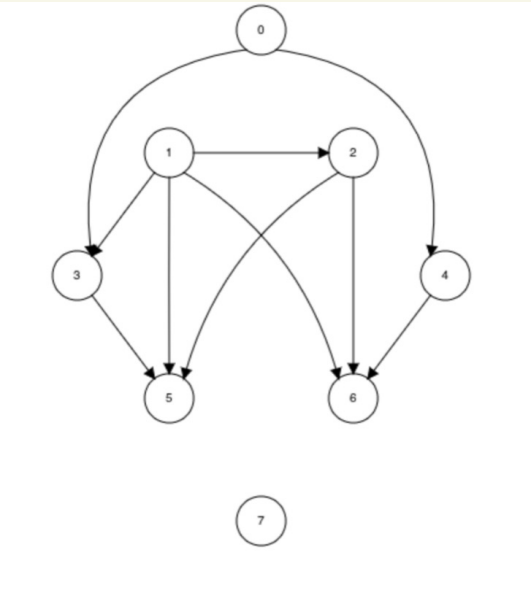
Track dependencies:

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

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

↳ lays out dependences 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?
(b/c DFS)

Top sort DFS: making it more precise

```
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

Unpacking his figure:

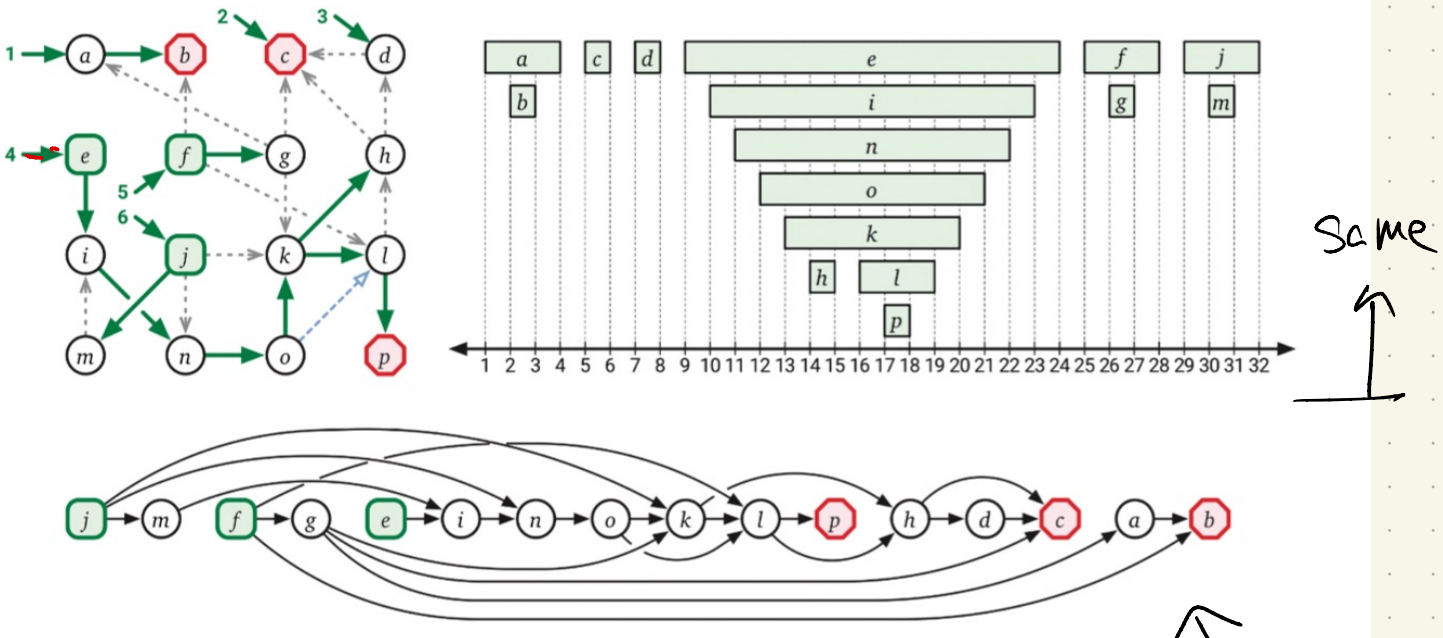


Figure 6.8. Reversed postordering of the dag from Figure 6.6.

Memoization + DP

Nice connection!

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

Why?

Think of the recurrences:

$$T(v) = \max_{\text{(predecessors or successors } u \text{ of } v)} \{ T(u) \text{ lookup} + \text{calculation} \}$$

When will the algorithm get stuck?

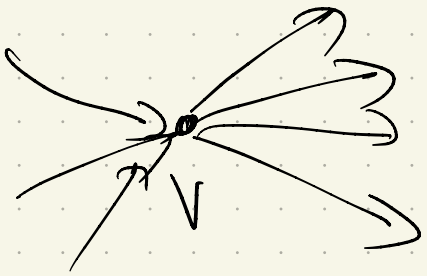
Example: longest path in a DAG.

Usually → very hard.

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

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

$$= \max \left\{ \right.$$



Using this recursion:

"memoize" the value LLP:

Add a field to the vertex & store it.

(Initially, =)

GetLongest(v):

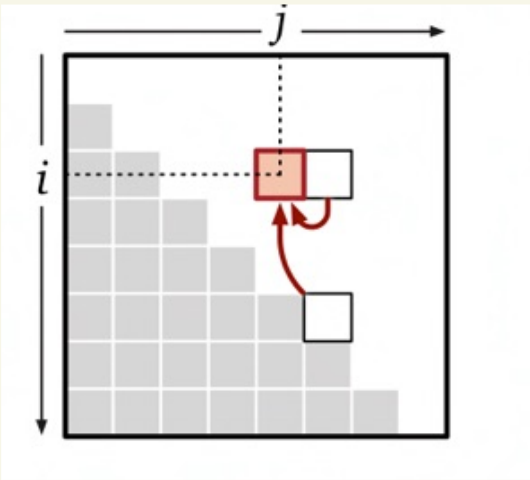
if v = t:

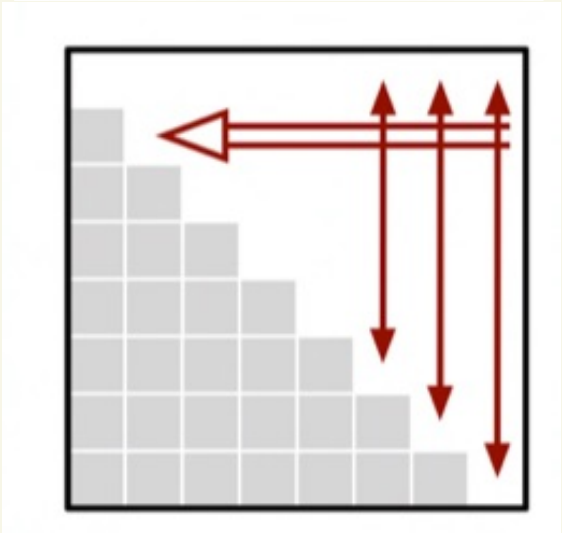
otherwise:

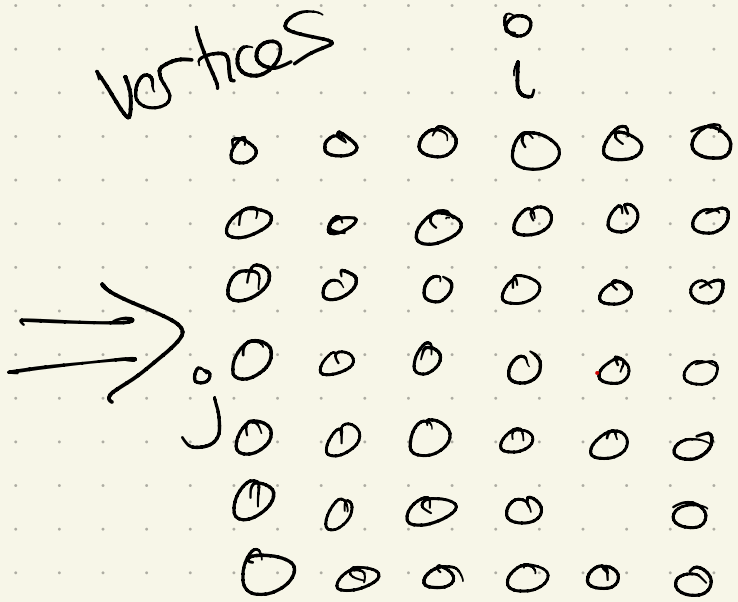
In principle, every DP we saw is working on a dependency graph of subproblems!

Recall: Longest Inc. Subsequence

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







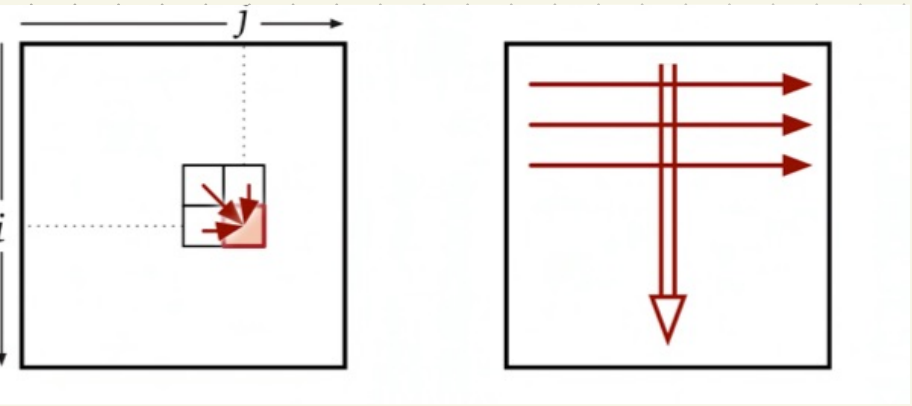
edges:

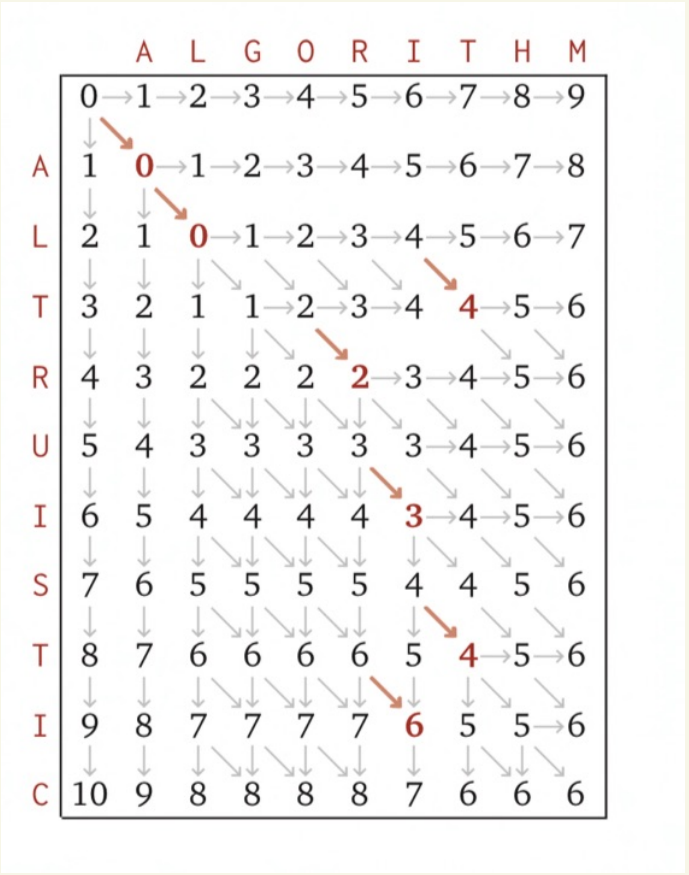
$(i,j) \rightarrow$

$(i,j) \rightarrow$

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

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



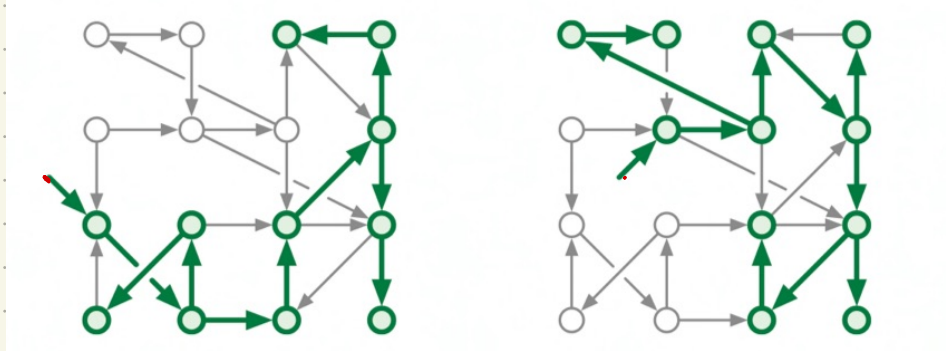


		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

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:

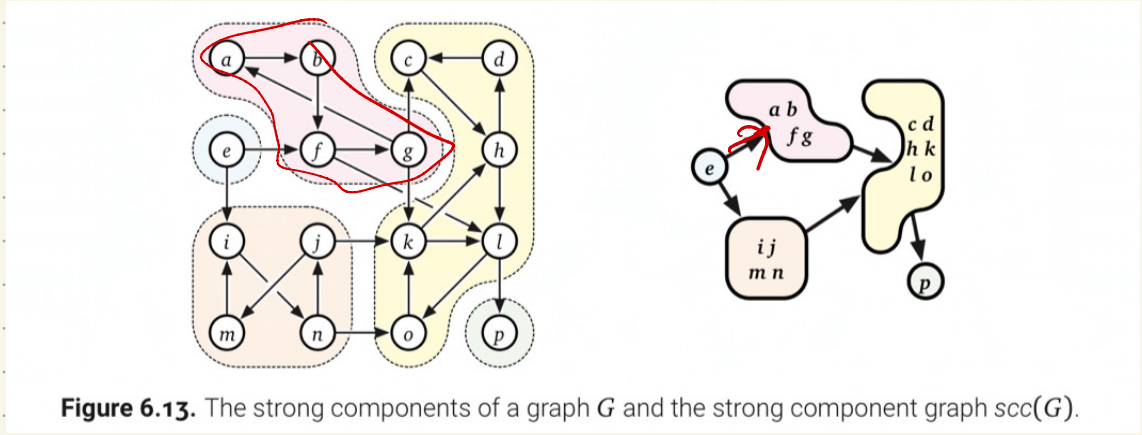


Figure 6.13. The strong components of a graph G and the strong component graph $scc(G)$.

How?

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

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 $W: E \to \mathbb{R}$ (or $\mathbb{R}^+$)

picture: