# Algorithms + Complexity, Spring 26

## Directed graphs

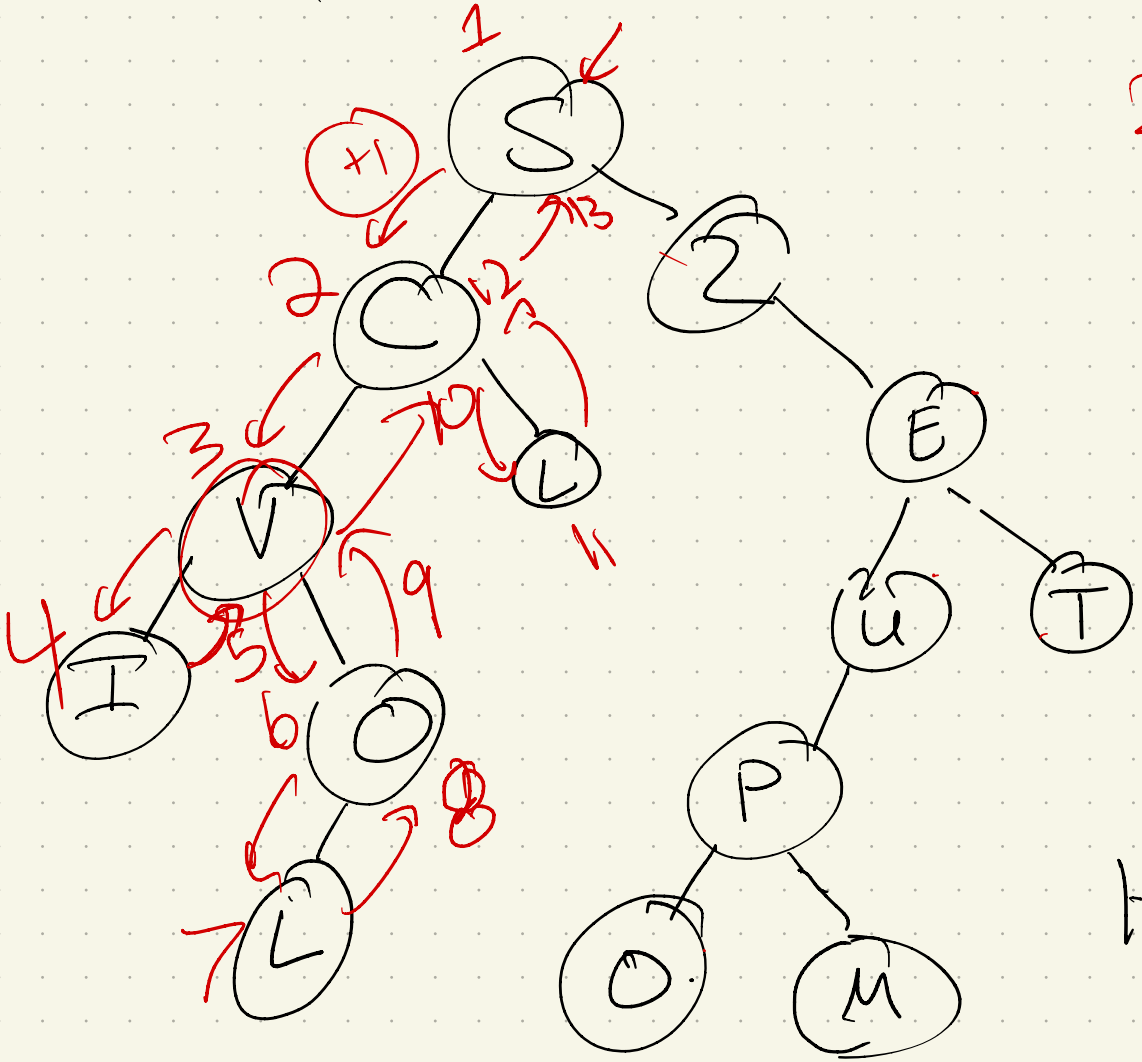
# Recap

- Some midterm caveats:
  - No drops included yet
  - Midterm curve → not quite final but would be my curve if I had to set now.

- HW this week: NP-Completeness, plus some graphs
  - ↳ see skipped Ch. 5 of text for review, or last Thursday's lecture

# Searching a directed graphs:

## Recall: post order traversal




Print: all children & then node

ILOV...

His "clock":
add 1 for every edge traversal

In a larger directed graph, this "lifespan" represents

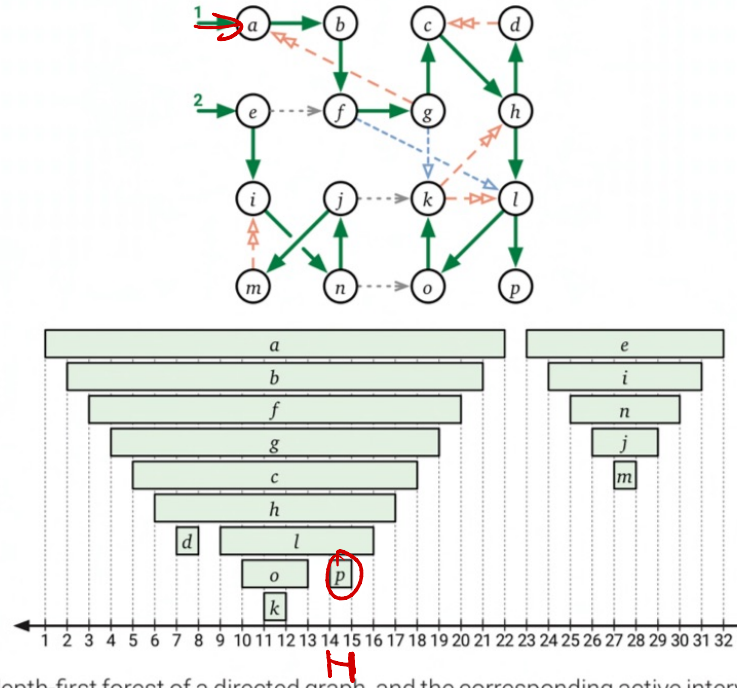



**Figure 6.4.** A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

how long a vertex is on the stack

Notation: [v.pre v.post]

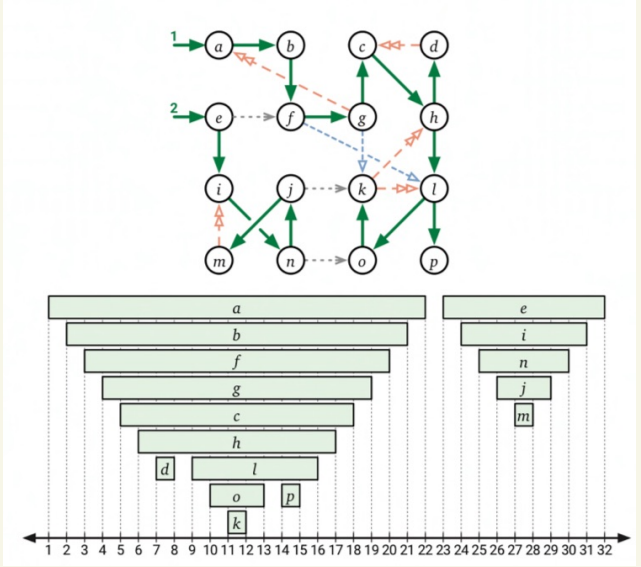
← first push on stack

last removal →

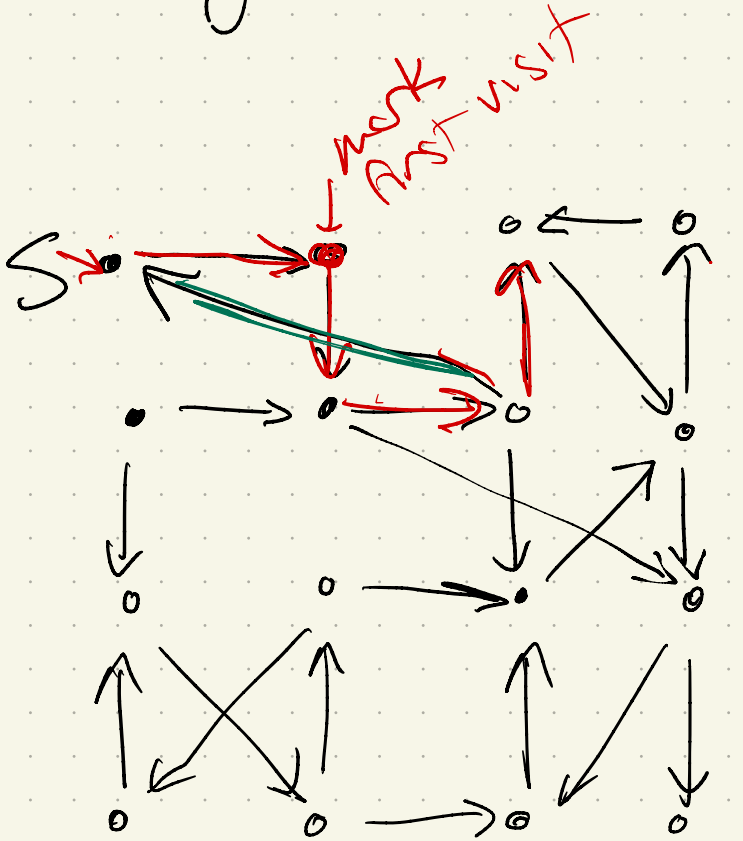
Why not BFS?

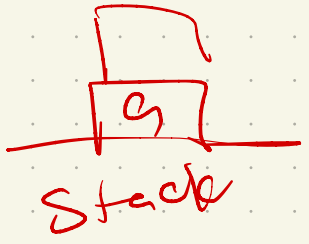
Dfn :- tree edge
- forward edge
- back edge
- cross edge

→ Clock reference:




Picture :



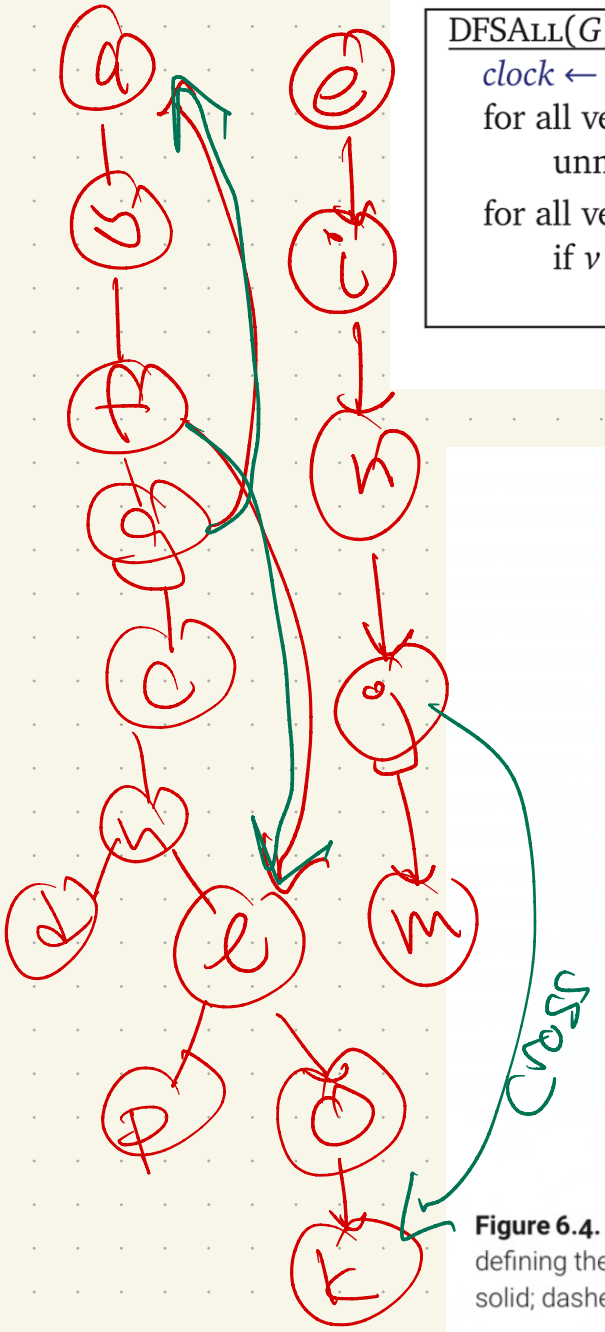



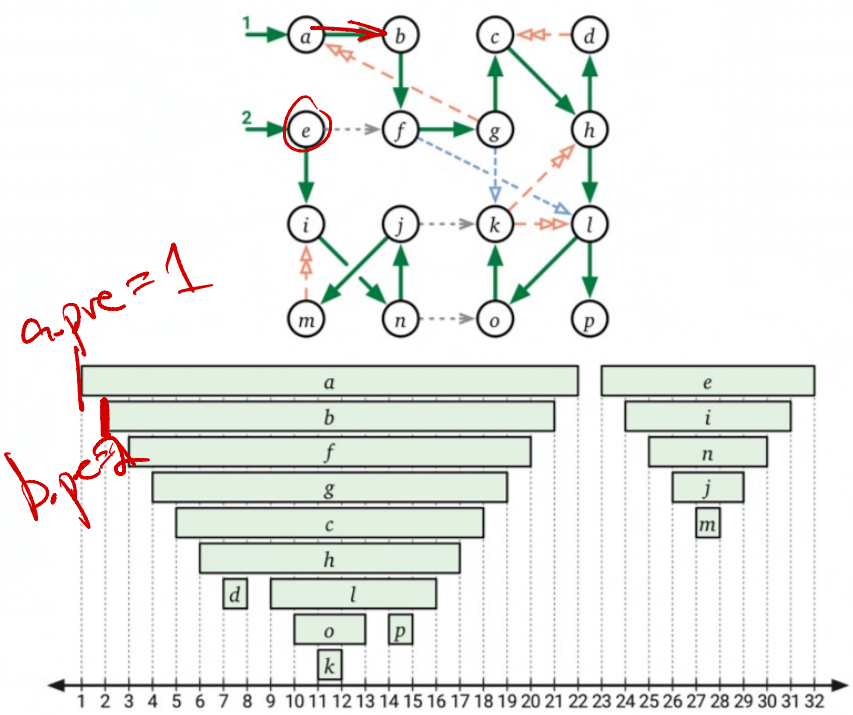
# Code: Still just DFS!

runtime $O(V+E)$

```
DFSALL(G):
    clock ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            clock ← DFS(v, clock)
```

```
DFS(v, clock):
    mark v
    clock ← clock + 1;  v.pre ← clock
    for each edge v→w
        if w is unmarked
            w.parent ← v
            clock ← DFS(w, clock)
    clock ← clock + 1;  v.post ← clock
    return clock
```




edges of G not in DFS tree

**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.

# Finding cycles

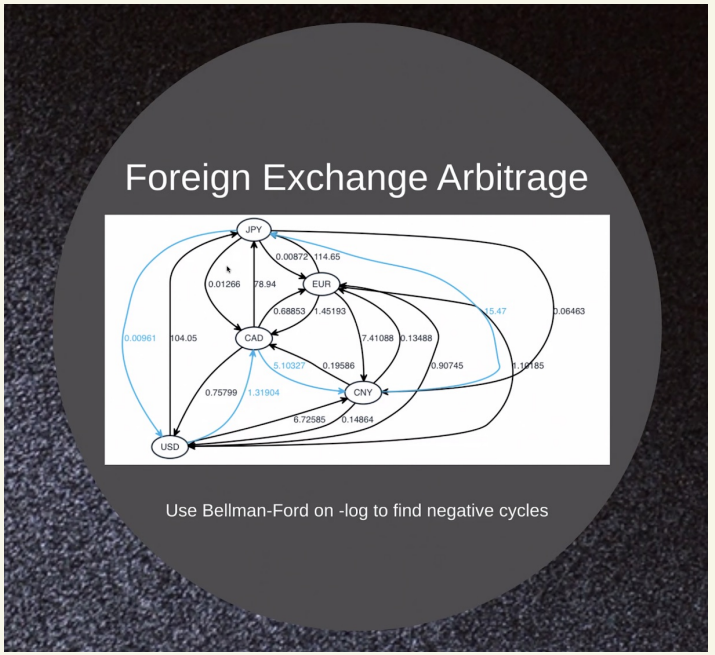
In general, cycles tend to be important.

Sometimes bad:

- topological ordering in a DAG
- checking dependencies
- longer run time
  ↳ see Dyn. Pro

Sometimes good:

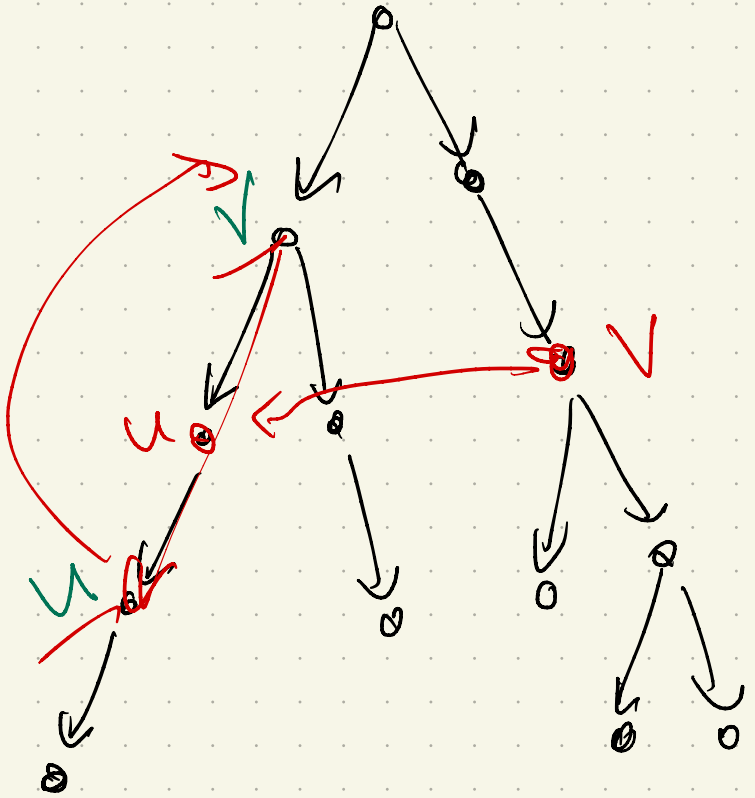
- arbitrage in finance



Suppose $u \to v$, $u.post < v.post$:

u was removed from "active" stack before v.

Think DFS tree:



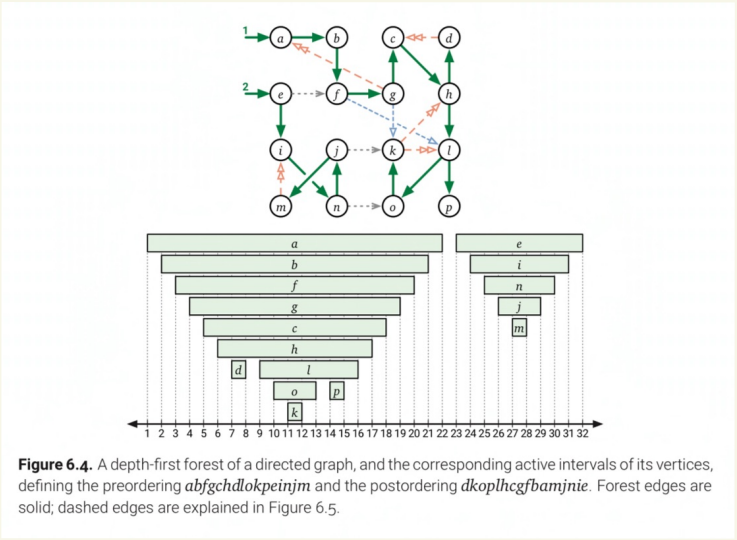




**Figure 6.4.** A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

Detect cycles:

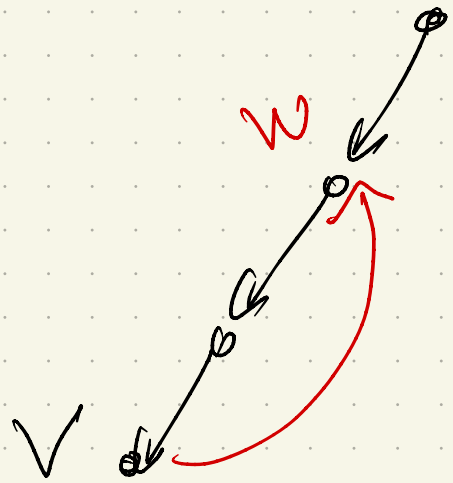
# The code: Recursive DFS modification

```
IsAcyclic(G):
    for all vertices v
        v.status ← New
    for all vertices v
        if v.status = New
            if IsAcyclicDFS(v) = False
                return False
    return True
```

```
IsAcyclicDFS(v):
    v.status ← Active
    for each edge v→w
        if w.status = Active
            return False
        else if w.status = New
            if IsAcyclicDFS(w) = False
                return False
    v.status ← Finished
    return True
```

have directed cycle

**Figure 6.7.** A linear-time algorithm to determine if a graph is acyclic.




Runtime? O(V+E)

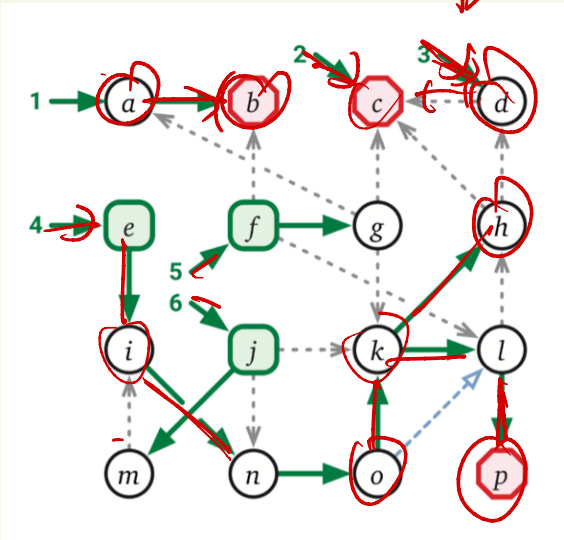
# Topological ordering: Why?

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 orders dependences

An example:




DFS tree

Post-ordering:

visit children

↳ then mark self

Here: b a c d p l h k o ...

Note: this puts a vertex "after" anything it can reach!

So, to get a topological ordering:

doing post-order print of DFS tree, then reversing

The code:

```
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

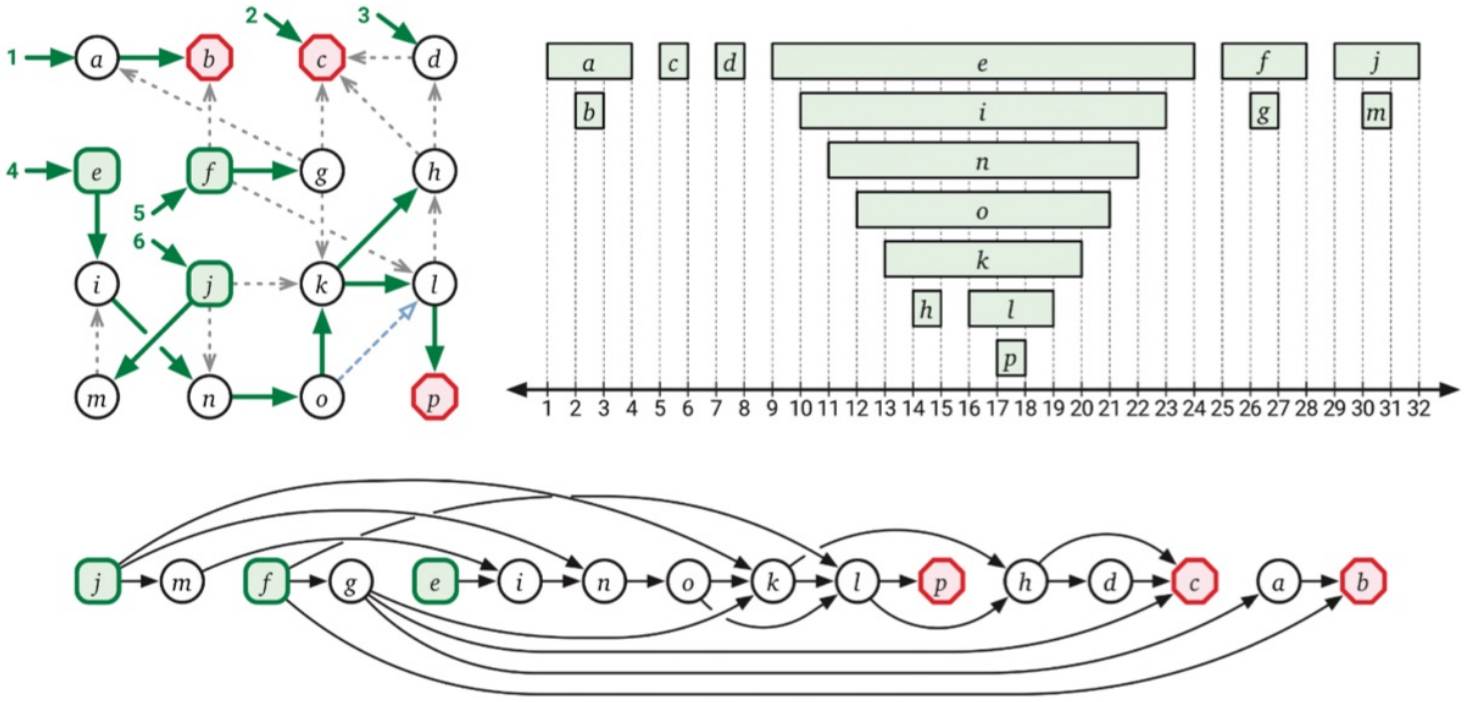
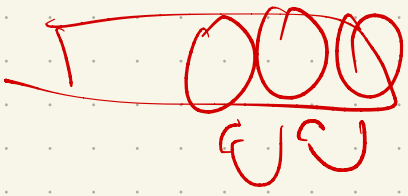
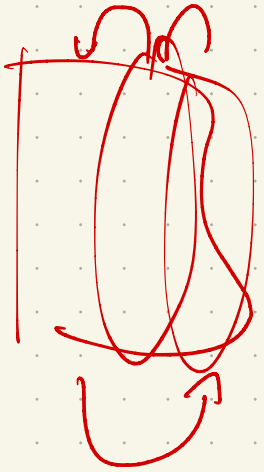



**Figure 6.8.** Reversed postordering of the dag from Figure 6.6.

# Memoization + DP





Nice connection!

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

Why? Think of the recurrences:

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

When do we get stuck?

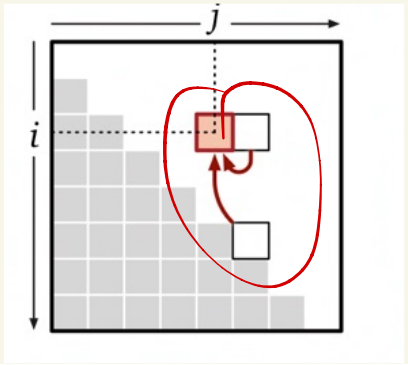
↳ when cyclic dependencies

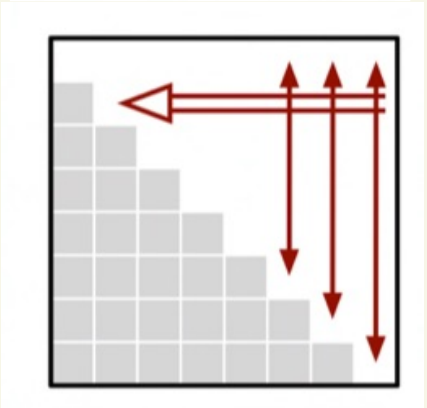
In principle, every DP we saw is working on a dependency graph of subproblems!

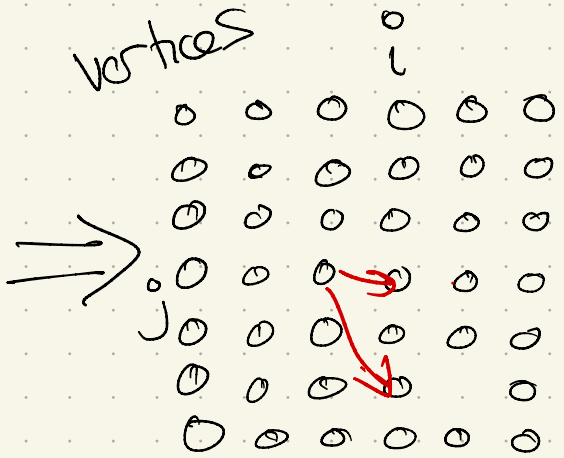
## Recall: Longest Inc. Subsequence

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

$n \times n$





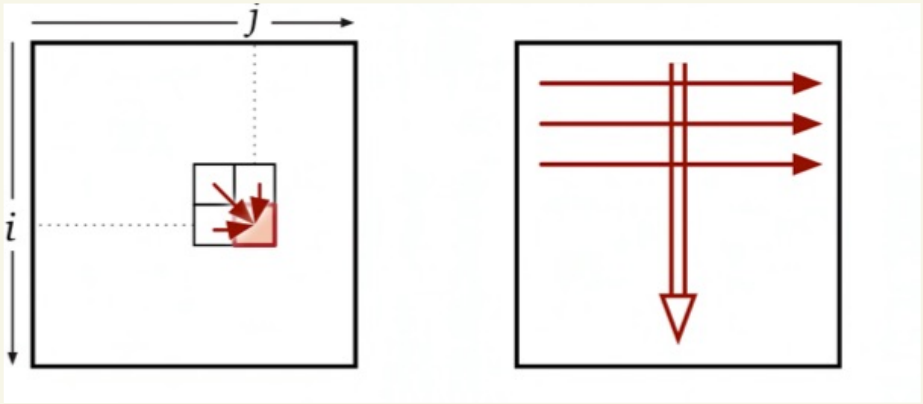

edges:

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

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

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

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



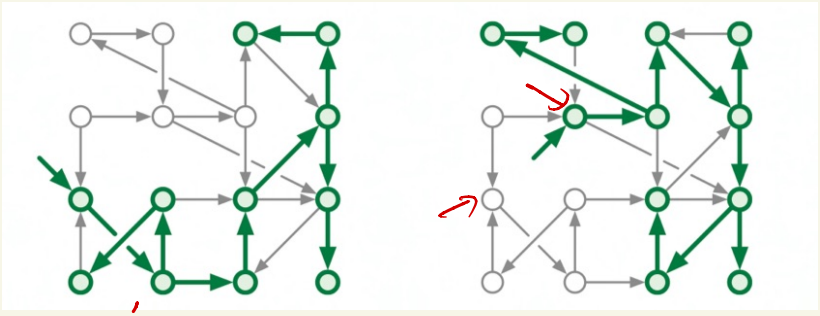

| | | A | L | G | O | R | I | T | H | M |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| A | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| L | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| T | 3 | 2 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 |
| R | 4 | 3 | 2 | 2 | 2 | 2 | 3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

Topological ordering

# Strong connectivity

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

Not true in directed case:



So 2 notions:

$u \rightsquigarrow v$ are:

weak connectivity: underlying undirected graph is connected: $u \rightsquigarrow v$ or $v \rightsquigarrow u$ a path

strong connectivity: $u \rightsquigarrow v$ and $v \rightsquigarrow u$ via directed paths

Can actually order the strongly connected pieces of a graph:

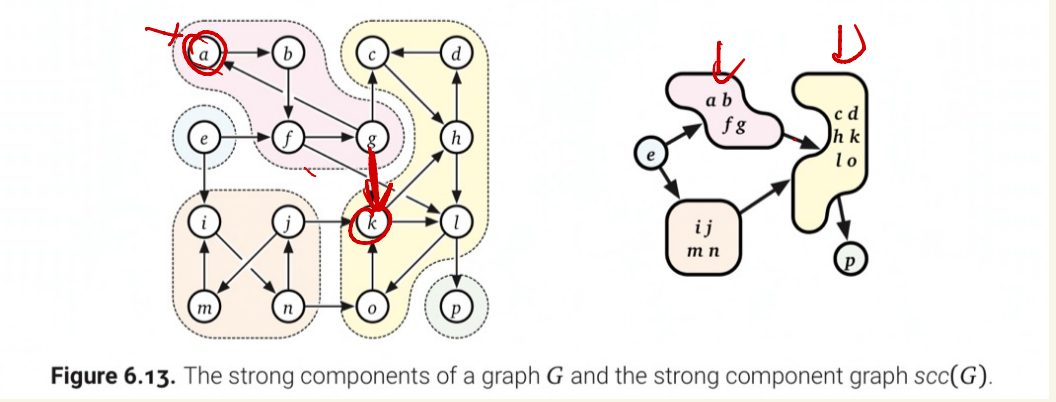
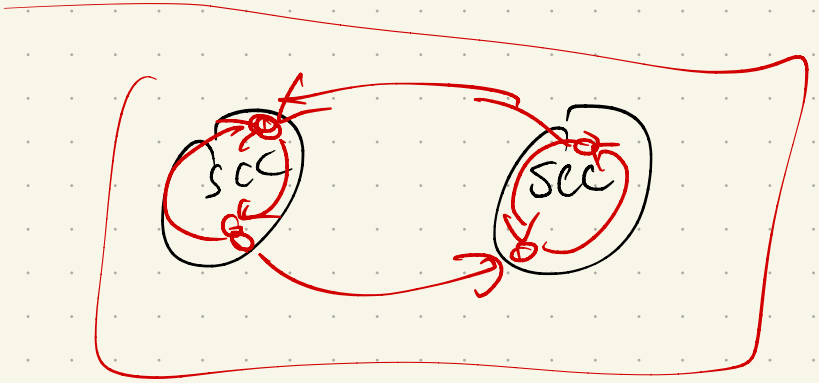



**Figure 6.13.** The strong components of a graph $G$ and the strong component graph $scc(G)$.

How?

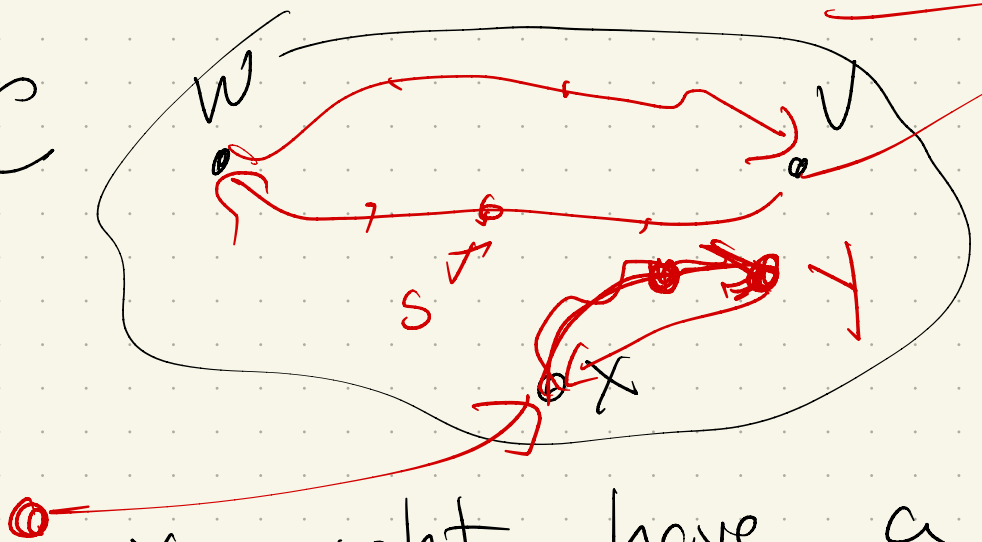
- Well, two components either aren't connected, or only have 1-way edges. Why?



More formally:

Every strongly cc has exactly one vertex with no parent in the component.

Proof: Consider $u \leadsto w$ in SCC $C$.




Let $x$ be first vertex in clock-order in $C$.

$x.pre < w.pre$
$x.pre < v.pre$

$x$ might have a parent — where?

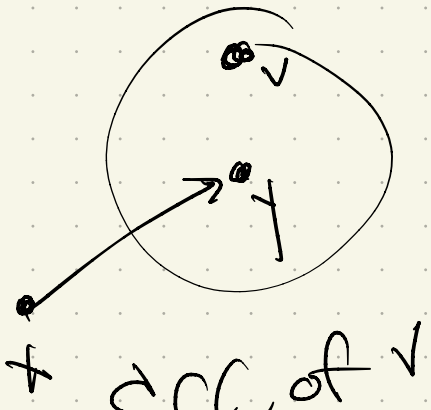
in another component only

and any other vertex is reachable from $x$, so its parent also is.

# Kosaraju + Sharir:

Do DFS → fix post-ordering.

Let v be last vertex.




Claim: v must be a source

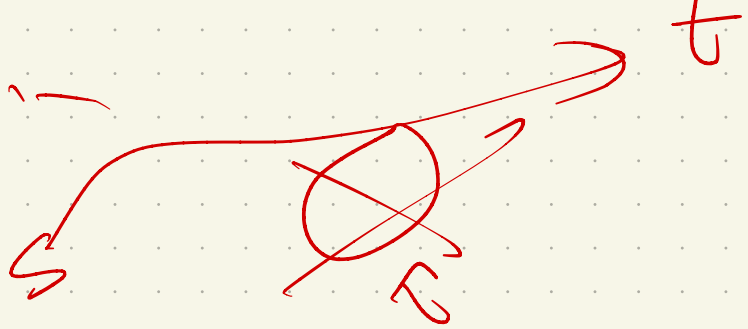
Suppose not: then some edge $x \to y$ into it exists.

But: we did DFS, + hit x earlier!

# Next time: Shortest paths

Goal: given $s, t \in V$, compute the shortest path from $s$ to $t$.

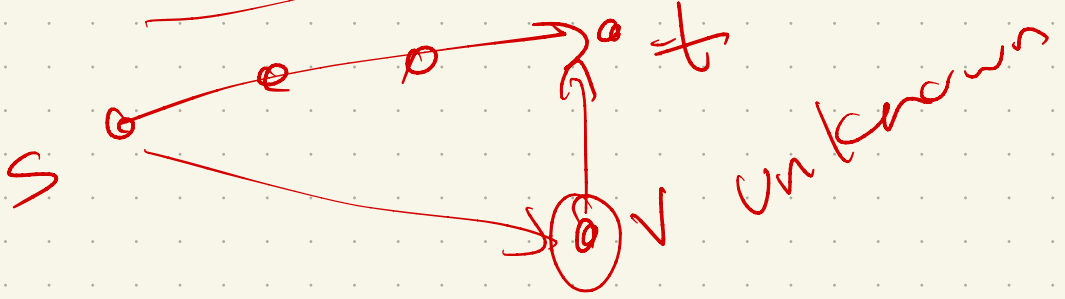
Motivation: roads, routing, ...




To solve this, we need to solve a more general problem:

find shortest paths from $s$ to every vertex.

Why?



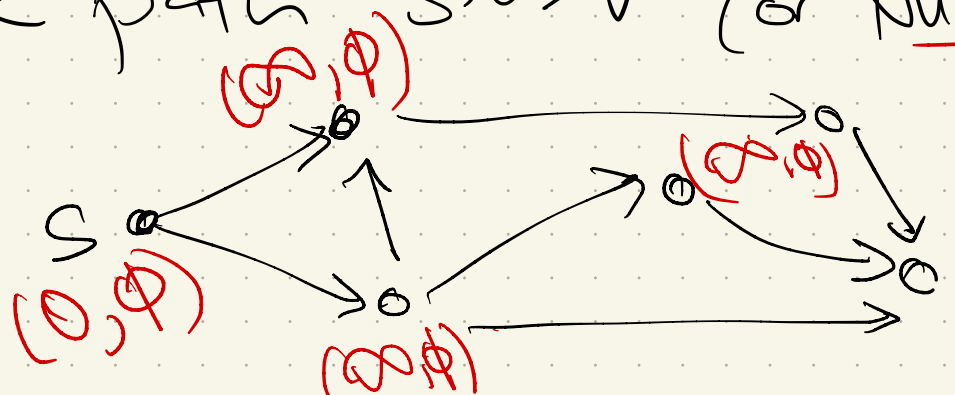
# Computing a SSSP.

(Ford 1956 & Dantzig 1957)

Each vertex will store 2 values.
(Think of these as tentative shortest paths.)

- dist(v) is length of tentative shortest path $s \rightsquigarrow v$ (or $\infty$ if don't have an option yet)
- pred(v) is the predecessor of v on that tentative path $s \rightsquigarrow v$ (or NULL if none)

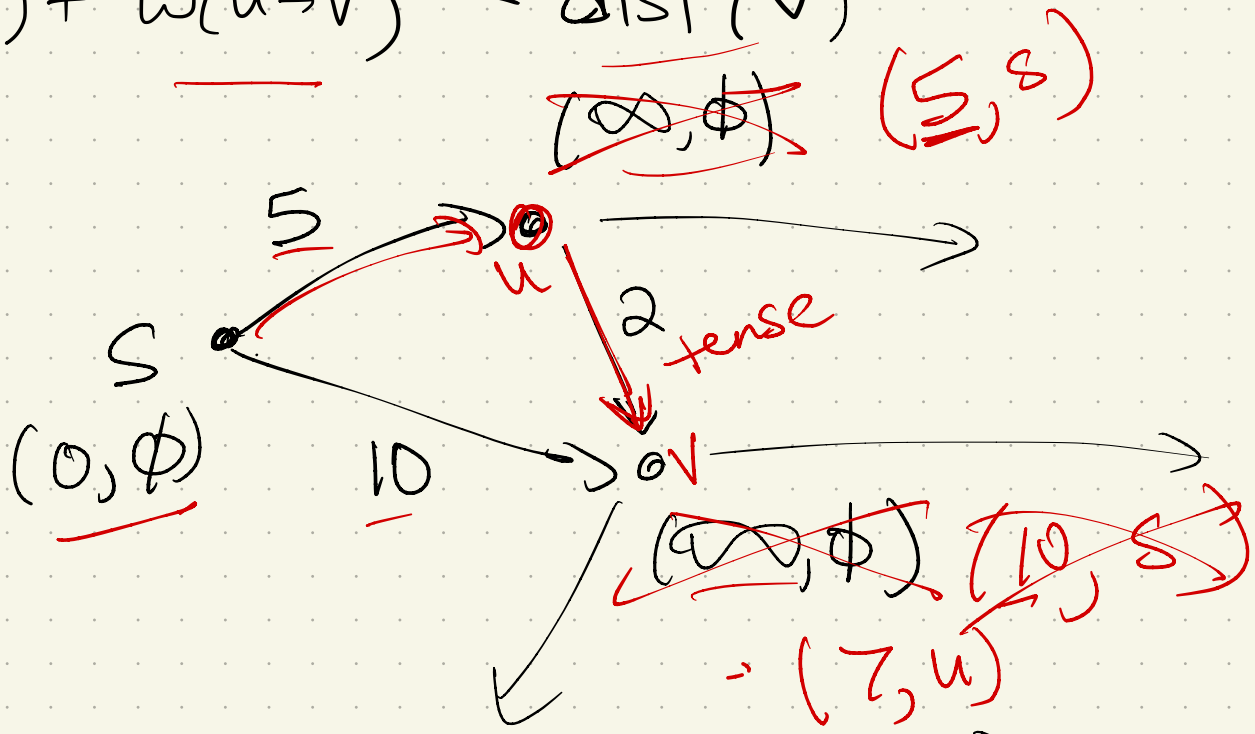
Initially:



We say an edge $\overrightarrow{uv}$ is tense if

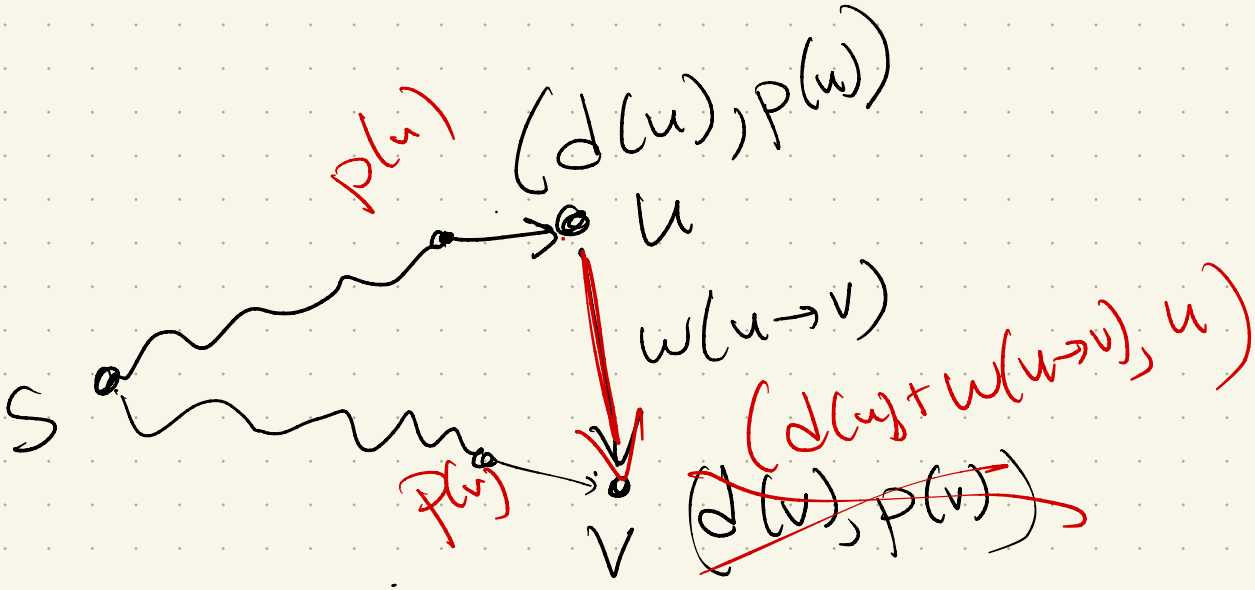
$$dist(u) + w(u \to v) < dist(v)$$

Initially:




Here:

In general:



Key idea of all algorithms:
Find tense edges & relax them:

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

Then:

```
INITSSSP(s):
  dist(s) ← 0
  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← NULL
```

```
GENERICSSSP(s):
  INITSSSP(s)
  put s in the bag
  while the bag is not empty
      take u from the bag
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              put v in the bag
```

Claim: At any point in time, dist(v) is either $\infty$ or the length of some $s \rightsquigarrow v$ walk.

Proof: Induction on while loop iterations.

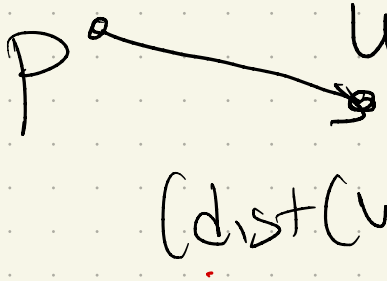
Base case: loop iteration 1

at beginning, s has dist = 0 + all others = $\infty$

at end, s has dist = 0 still, + all neighbors u now have dist(u) = w(s→u), which is a length 1 walk. (Others are $\infty$)

Ind hyp:
In iteration k-1, the claim is true

Ind Step:
In iteration k: At beginning, we take out some vertex u.




By IH, dist(u) is the weight of some s⇝u walk.

At end, all nbrs v of u are either unchanged (& so by IH are still either ∞ or length of s⇝v walk)

or $u \to v$ was tense, so

now $dist(v) = dist(u) + w(u \to v)$.

Since $dist(u)$ is a $s \rightsquigarrow u$ walk, then $dist(v)$ is weight of the walk $(s \rightsquigarrow u) + (u \to v)$, which is a walk with one more edge at end.

(All other vertices are unchanged, so by IH are still $\infty$ or a $s \rightsquigarrow v$ walk.) ∎

So why all the shortest path algorithms?

How best to relax?

Also → negative edges are bad.

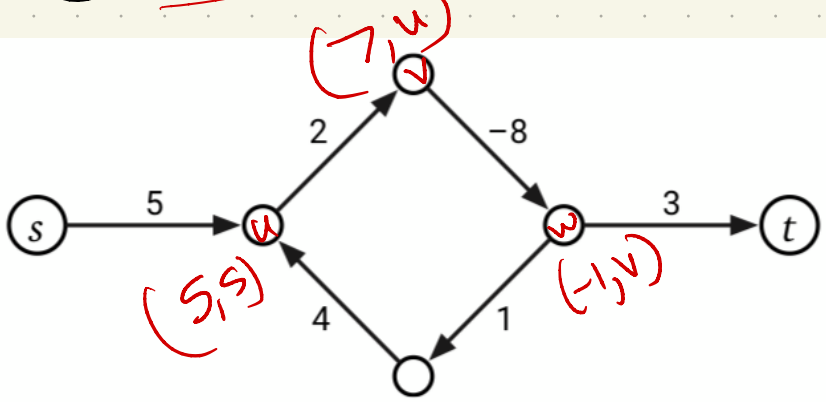



**Figure 8.3.** There is no shortest walk from $s$ to $t$.