Algorithms (2020)

Graphs
(pt 3)

Recap

- HW 4 (Greed) is up.
  due in 1 week (Wed.)
- Reading as usual
- No office hours on Friday.
  ↳ email me if want a time on Thursday

Last time: Iterative search algorithm (not recursive)

```
WHATEVERFIRSTSEARCH(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

data structure

find if any/every node connects to s

track edges in tree

```
WHATEVERFIRSTSEARCH(s):
    put (∅, s) in bag
    while the bag is not empty
        take (p, v) from the bag            (⋆)
        if v is unmarked
            mark v
            parent(v) ← p
            for each edge vw                (†)
                put (v, w) into the bag     (⋆⋆)
```

edge

demo: parents are stored in an array (once!)

use to find paths to s

```
WHATEVERFIRSTSEARCH(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

✓ base case

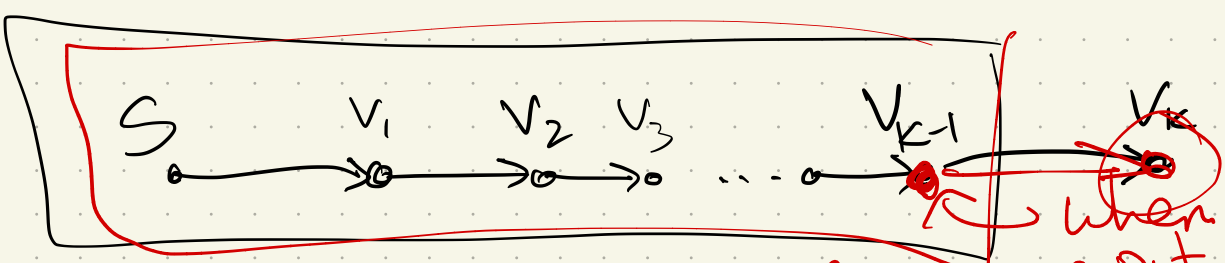
when loop ends, ds "bag" is empty

Correctness:

Need to show it marks all vertices reachable from s, + no others.

Proof: induction!

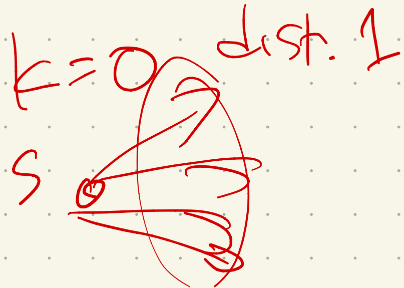
- s is marked
- Assume vertices at distance (# edges) k−1 are marked + show all at distance k will also be marked



when marked, we put nbrs

rec fairy

Ind hyp.



To show it's a tree: (p, w)
trace all those parent "pointers".
There are n-1, one per vertex (other than s).

only updated when unmarked, & only marked once

no cycles & n-1 edges
$\Rightarrow$ a tree.

Why? See dis. math book, or exercise 1 of this chapter.

Other notes:

- "Best-first" search:
  Wait for next chapters - these are a bit more subtle, so we'll spend more time later.
- Directed - See chapter 6.

Well-known variants:
(cool demo last time)

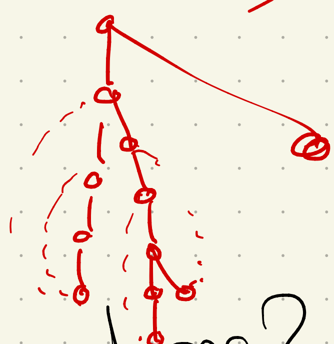
BFS: bag is queue $\Rightarrow$ short & "bushy"

r

all nbrs of r

all dist 2 from r

DFS: bag is stack $\Rightarrow$ tall & "skinny" trees



Other data structures?

these two are fast!

Next – other useful questions this can address...

In a disconnected graph:

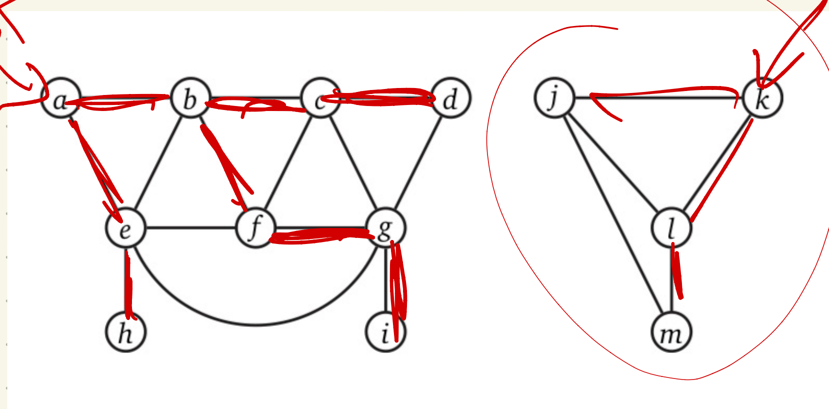
Often want to count or label the components of the graph.

(WFS(v) will only visit the piece that s (the root) belongs to.)

root

root

never visited



Solution: Call it more than one time!

global → unmark all vertices

for all vertices v:

if v is unmarked:

WFS(v)

← marks entire component

Result:

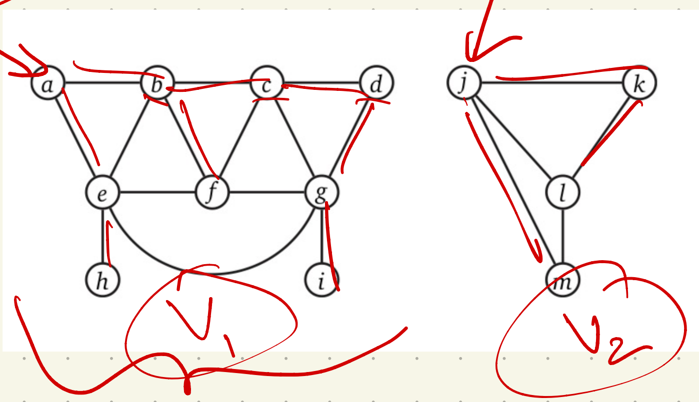
```
CountComponents(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            WhateverFirstSearch(v)
    return count
```

$O(V)$

only happens ≤ 1 per vertex

subroutine (already wrote it!) costs time proportional to size of component

count = ~~1~~ 2



$V = V_1 \cup V_2$

runtime: $O(V) + \underbrace{O(V+E)}$

$= O(V+E)$

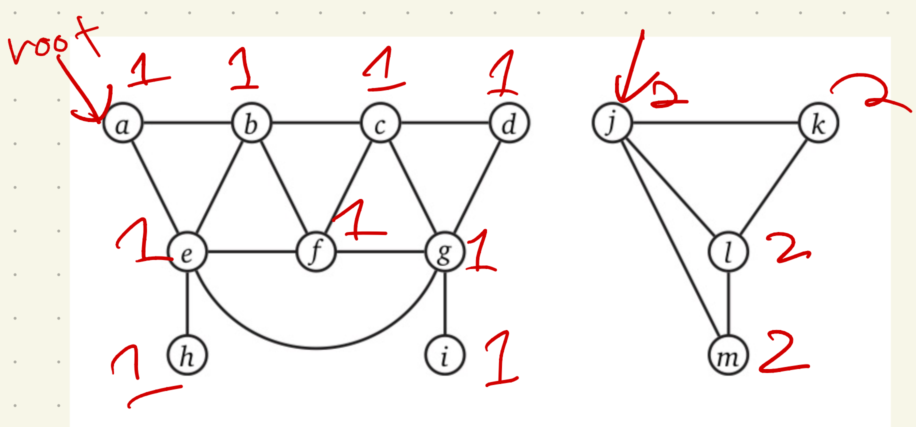
Finally, can even record which component each vertex belongs to:

```
CountAndLabel(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            LabelOne(v, count)
    return count
```

```
⟨⟨Label one component⟩⟩
LabelOne(v, count):
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            comp(v) ← count
            for each edge vw
                put w into the bag
```

basically WFS where count is passed in also.

count = ~~1~~ 2



$O(V+E)$

Dfn: Reduction

A **reduction** is a method of solving a problem by transforming it to another problem.

Note: you've seen/done this in other classes!

↳ every data structure or object or subroutine.

We'll see a ton of these!

(Especially common in graphs...)

Key: Take whatever input, & convert it to a form that other problem solves.

First example:

Given a pixel map, the flood-fill operation lets you select a pixel & change the color of it & all the pixels in its region.

n

n

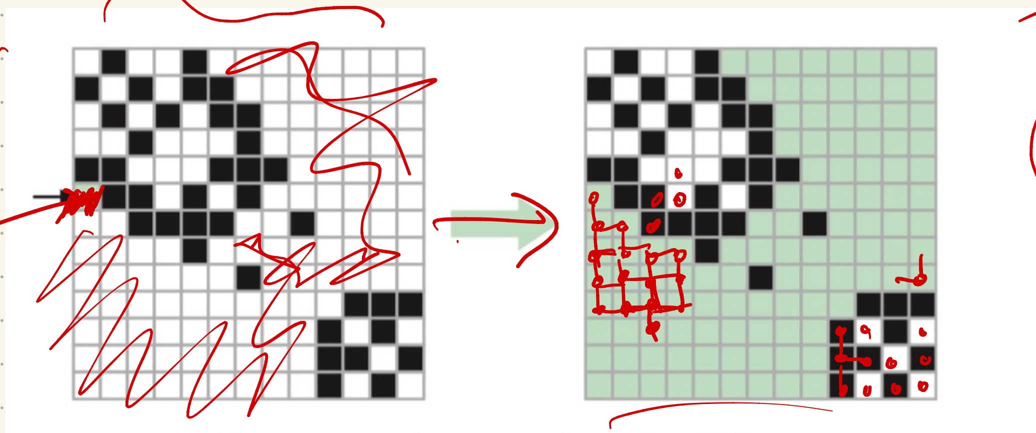


Figure 5.13. An example of flood fill

not a graph

How?

size of G: $O(n^2)$ for $n \times n$ pixel image

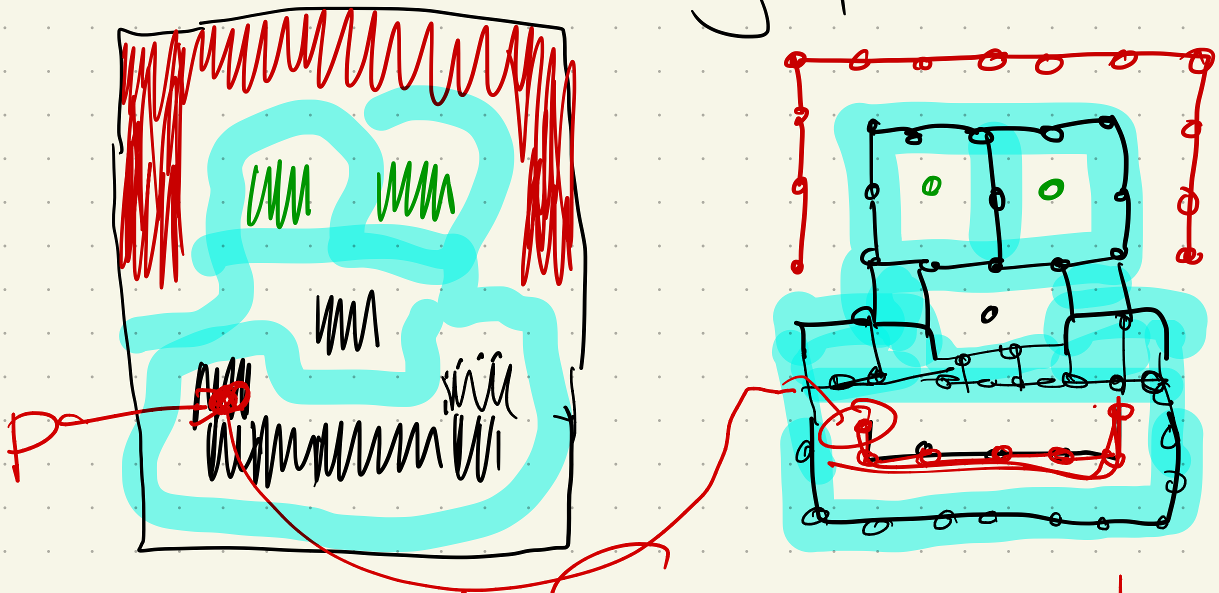
Could do some crazy pixel algorithm.

No!

$\boxed{\text{Build a graph: let each pixel be a vertex. Connect if same color, \& if nbrs}}$: n^2 vertices $= V$

$\leq 4n^2$ edges $= E$

So: Build a graph from pixels:



Runtime; in terms of input nxn pixel array!

Algorithm:

build G ← $O(n^2)$

call WFS ← $O(n^2)$

(plus tell me why!)

If pixel p is selected:

Find corresponding vertex p in graph.

WFS(p, color)

$O(V+E)$

variant where marking instead colors the vertex

Arguably, these reductions are the most important thing in graphs!

Like data structures - you won't usually have to re-code everything.

Instead:

- Set up graph ← takes time
- Call some algorithm

↑ usually based on input size, not V+E

So runtime/correctness:

tracking V+E in terms of input.

graph is built so that algorithm's answer gives correct answer on input.

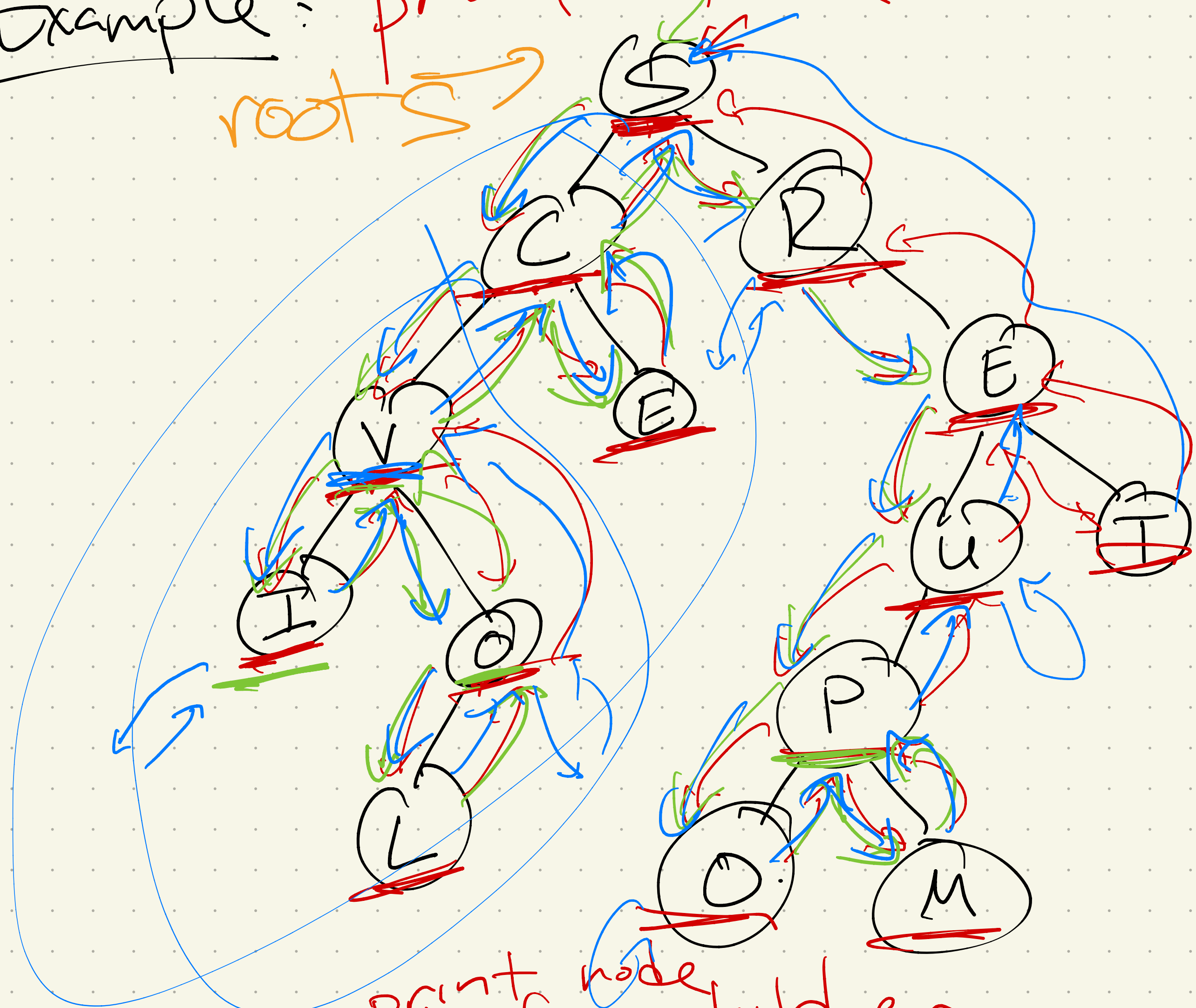
Next chapter:

All about directed graphs!

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

- Pre-order(v):
  - visit v
  - visit all children (if present)
- Post-order:
  - visit all children
  - visit v.
- In-order: (binary trees)
  - visit left child
  - visit self (v)
  - visit right child

Example: print tree



print node before children

Pre-order:

SCVIOLEREUPOMT

Post-order:

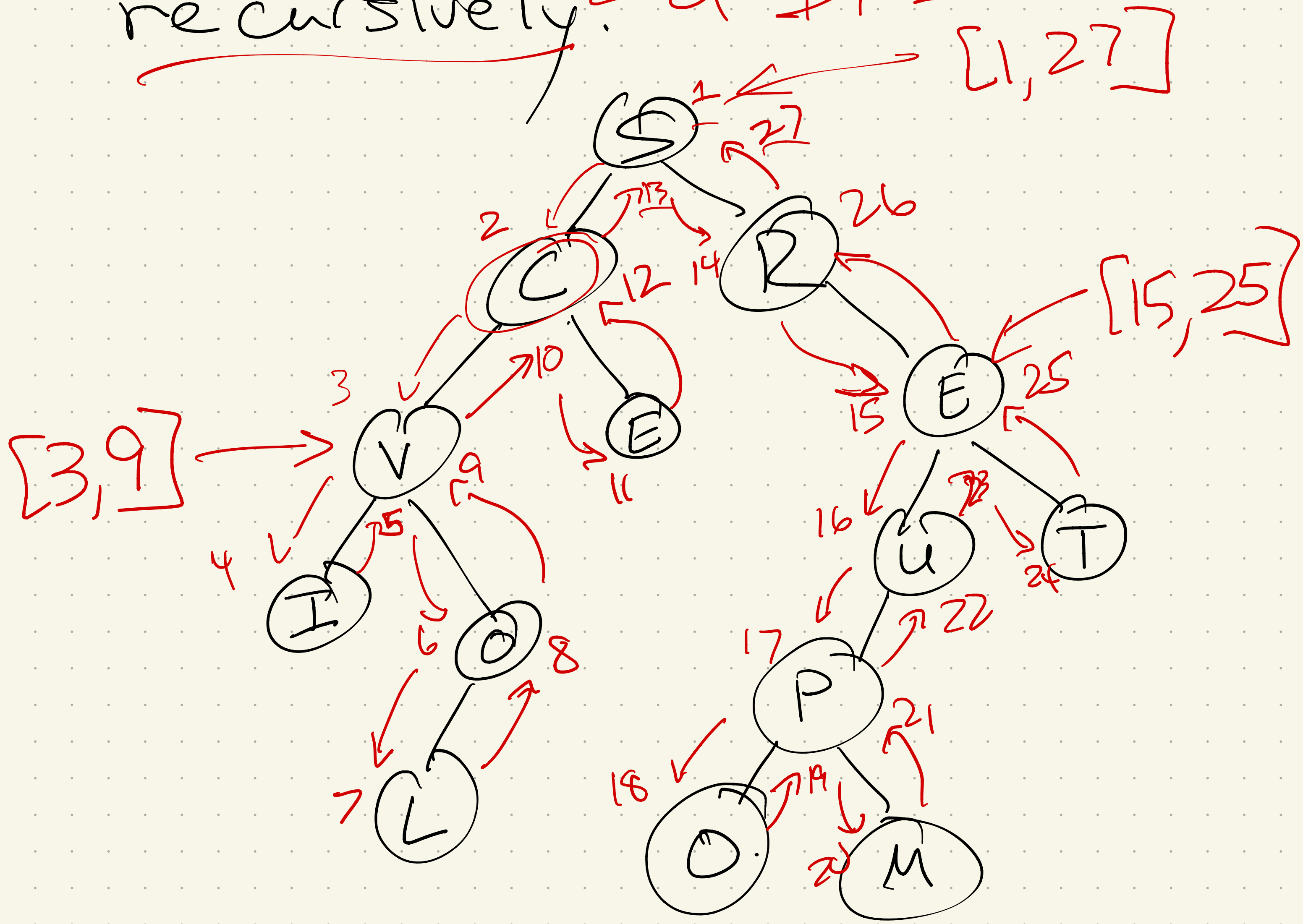
ILOVECOMPUTERS

In-order:

middle

IVLOCESROPMUET

He will bring up "lifespan" of a node, if implemented recursively. ← of DFS



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

activates when marked.

ends when last child recursion ends

Result:

clock & "pre"

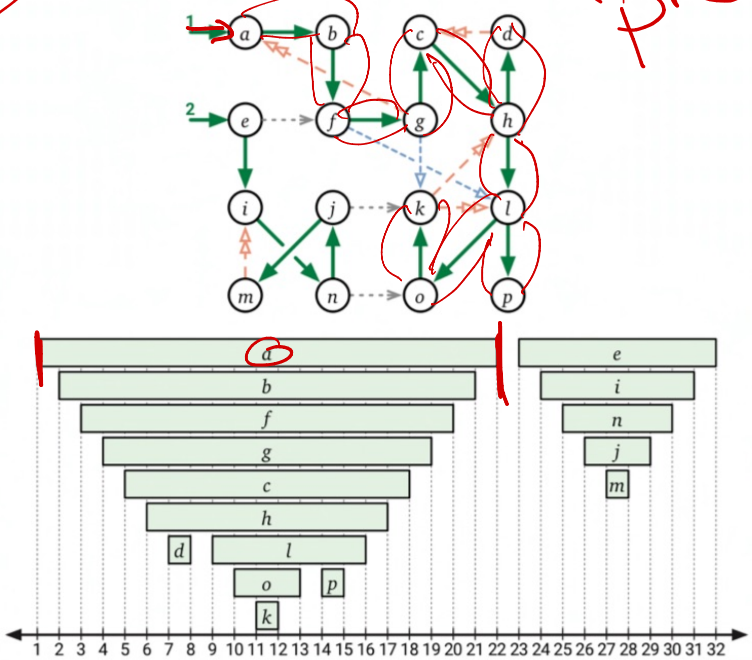


Figure 6.4. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

not binary!
Same idea