

Transition DS + Algorithms

BFS &
DFS



Worksheet

10:38 AM Wed Oct 14

CSE 50502

October 12

Fall 2026

wolfchambers.github.io

94%

1 of 1

Graphs

1. Data structures recap: For each of the following claims, explain what is missing or incorrect, or defend why the statement makes sense.

- (a) "Linked lists are best when there are many insertions, because insertion takes $O(1)$ time."
- (b) "Hash tables are always better than AVL trees, because $O(1)$ is better than $O(\log n)$."
- (c) "Arrays are poor choices for changing collections, because every insertion takes $O(n)$ time."

a) Only if have ptr, if no finding
vectors are also $O(1)$ at end
hashing is $O(1)$ expected also.
(c) not if at end!

2. Consider the adjacency matrix and adjacency list representations of a graph. Which choice would be better in each of the following situations? Justify your answer.

- (a) The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible. $\rightarrow$ adj list
- (b) The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.
- (c) You need to answer queries to find if two vertices are adjacent as fast as possible, no matter how much space you use. $\rightarrow$ adj matrix

b) also: Rehashing
Hash is
worst
case $O(n)$.

AVL is
steady worst
case, + sorted

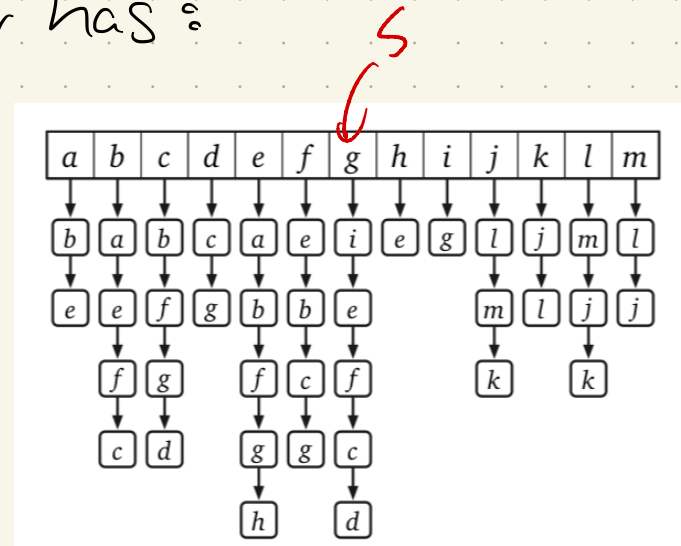
is slow
 $O(n)$

Graph Searching

How can we tell if 2 vertices are connected?

Remember the computer only has:

Traverse lists
& find who
can reach who.



g
m

Bigger question: can we tell if all the vertices are in a single connected component?

Possibly you saw depth first search (DFS)
and breadth first search (BFS)
in data structures:

"Bag": abstraction
of both
algorithms

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
    
```

"source" $\swarrow$ in back
recursive
vs Iterative $\nearrow$
loop-based

These are essentially just search strategies:
How can we decide if $u \leftrightarrow v$ are connected?

marked
all unmarked
 $\rightarrow$ set to F
in array

F	1
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~
~~F~~	~~1~~

bag: $\neq$ all s's
nbrs

Can use this to build a Spanning tree:

BFS

WHATEVERFIRSTSEARCH(s):

put $(\emptyset, s)$ in bag

while the bag is not empty

take (p, v) from the bag

if v is unmarked

mark v

$\text{parent}(v) \leftarrow p$

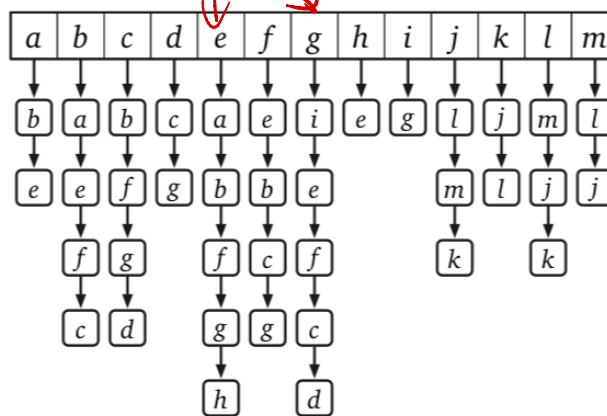
for each edge vw

put (v, w) into the bag

(*)

(†)

(**)

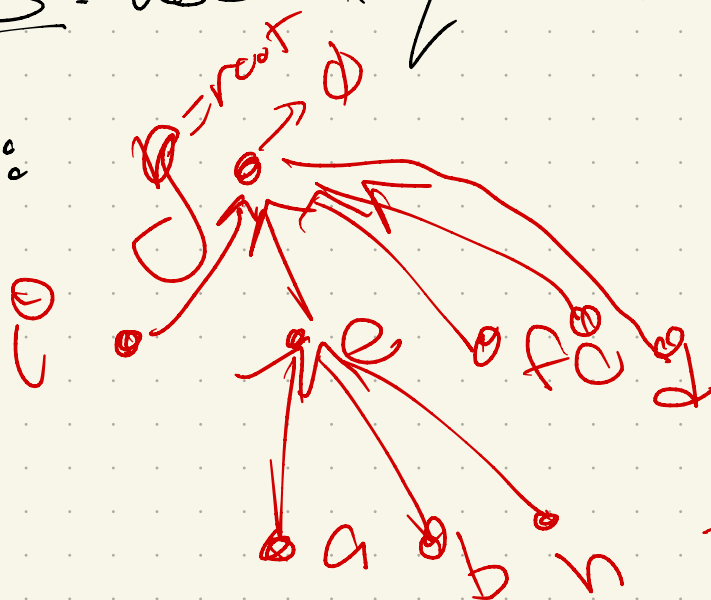


marked

- ~~m~~ a
- ~~m~~ b
- ~~m~~ c
- ~~m~~ d
- ~~m~~ e
- ~~m~~ f
- ~~m~~ g
- ~~m~~ h
- ~~m~~ i
- ~~m~~ j
- ~~m~~ k
- ~~m~~ l
- ~~m~~ m

BFS: use a queue:

Tree:



Queue:

~~$(\emptyset, s)$~~

~~(s, a)~~ ~~(s, b)~~ ~~(s, c)~~ ~~(s, d)~~ ~~(s, e)~~ ~~(s, f)~~ ~~(s, g)~~ ~~(s, h)~~ ~~(s, i)~~ ~~(s, j)~~ ~~(s, k)~~ ~~(s, l)~~ ~~(s, m)~~

~~(e, a)~~ ~~(e, b)~~ ~~(e, c)~~ ~~(e, d)~~ ~~(e, f)~~ ~~(e, g)~~ ~~(e, h)~~ ~~(e, i)~~ ~~(e, j)~~ ~~(e, k)~~ ~~(e, l)~~ ~~(e, m)~~

~~(f, a)~~ ~~(f, b)~~ ~~(f, c)~~ ~~(f, d)~~ ~~(f, e)~~ ~~(f, f)~~ ~~(f, g)~~ ~~(f, h)~~ ~~(f, i)~~ ~~(f, j)~~ ~~(f, k)~~ ~~(f, l)~~ ~~(f, m)~~

~~(g, a)~~ ~~(g, b)~~ ~~(g, c)~~ ~~(g, d)~~ ~~(g, e)~~ ~~(g, f)~~ ~~(g, g)~~ ~~(g, h)~~ ~~(g, i)~~ ~~(g, j)~~ ~~(g, k)~~ ~~(g, l)~~ ~~(g, m)~~

c's nbr, d's nbrs, a's, b's

DFS: use a stack

WHATEVERFIRSTSEARCH(s):

put $(\emptyset, s)$ in bag

while the bag is not empty

take (p, v) from the bag

if v is unmarked

mark v

$parent(v) \leftarrow p$

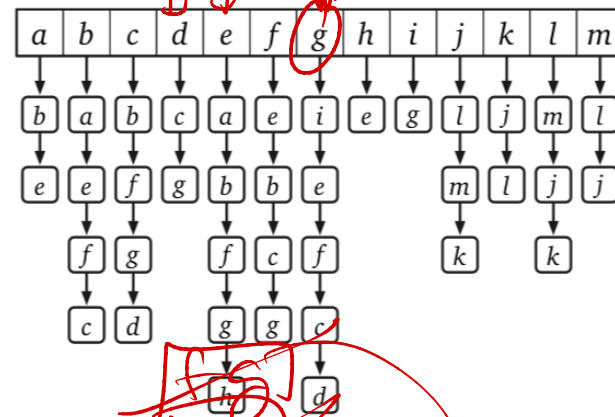
for each edge vw

put (v, w) into the bag

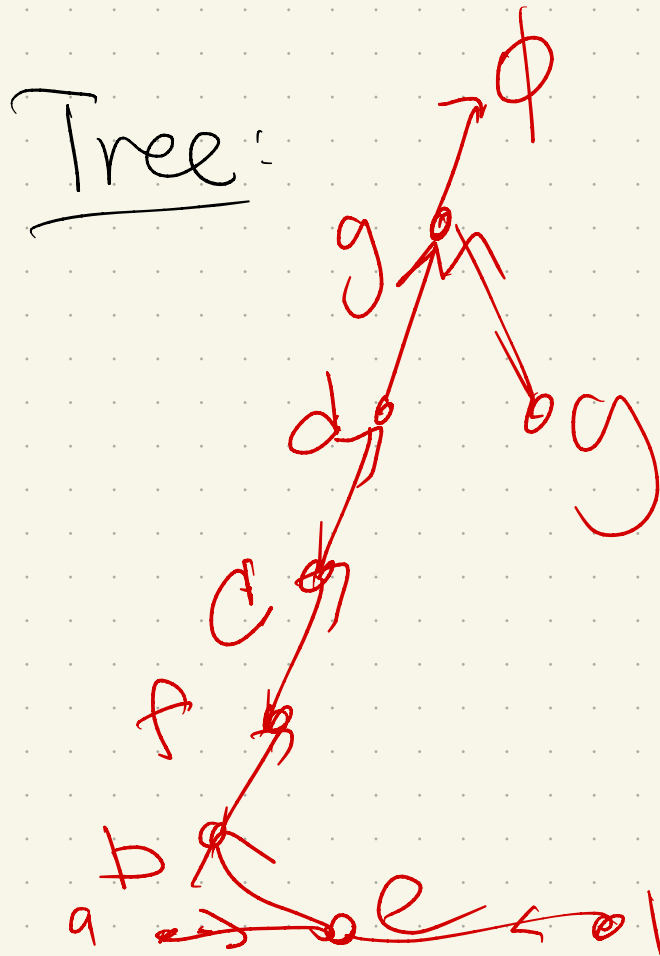
(*)

(†)

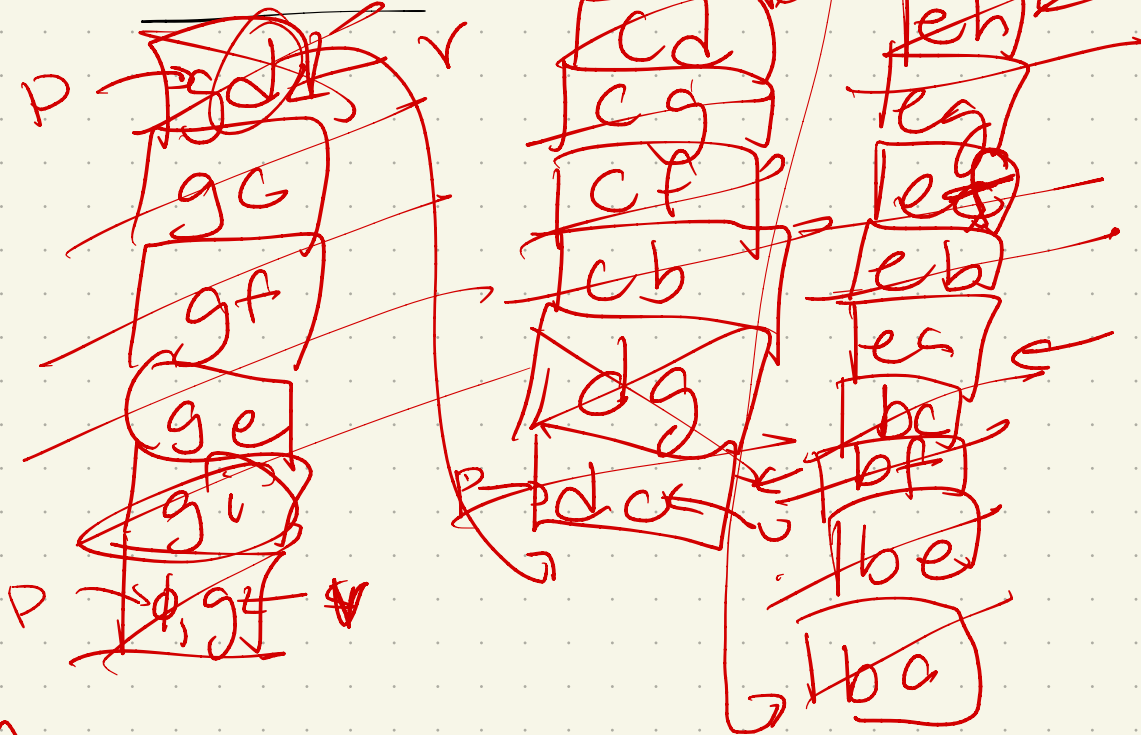
(**)



~~un a~~
~~un b~~
~~un c~~
~~un d~~
~~un e~~
~~un f~~
~~un g~~
~~un h~~
~~un i~~
~~un j~~
~~un k~~
~~un l~~
~~un m~~



Stack:



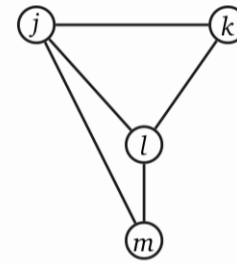
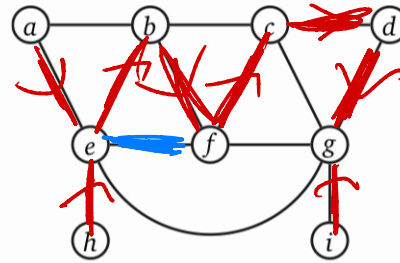
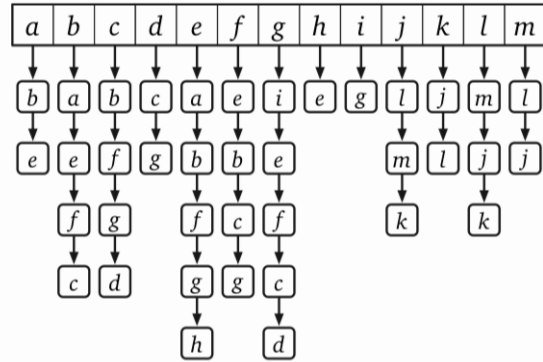
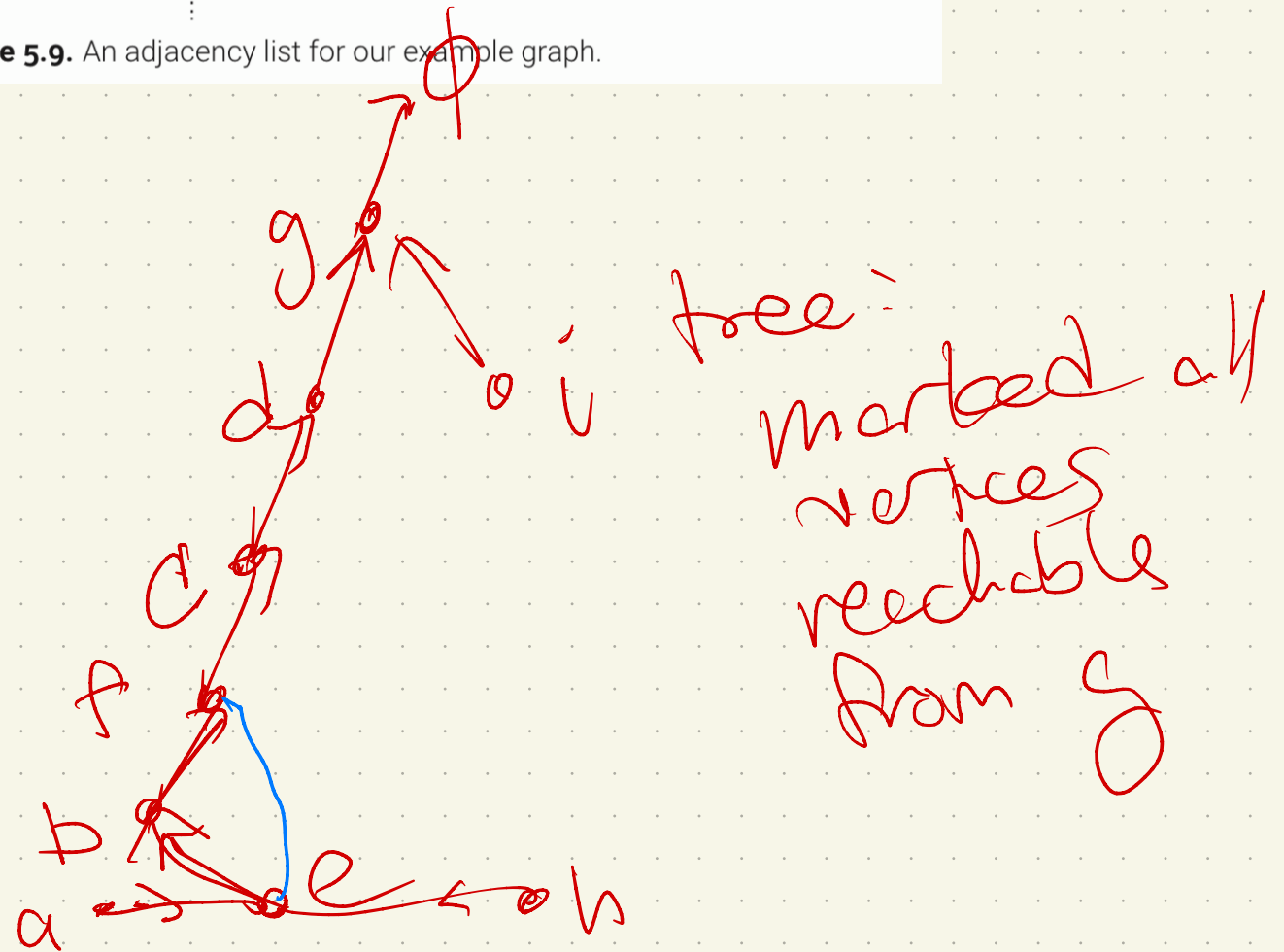


Figure 5.9. An adjacency list for our example graph.



tree:
marked all
vertices
reachable
from g

Just remember: different!

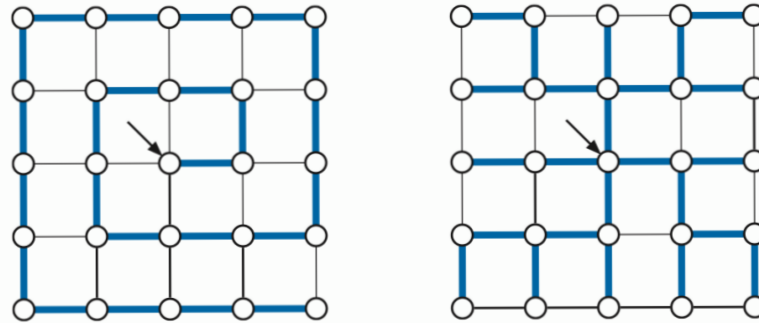
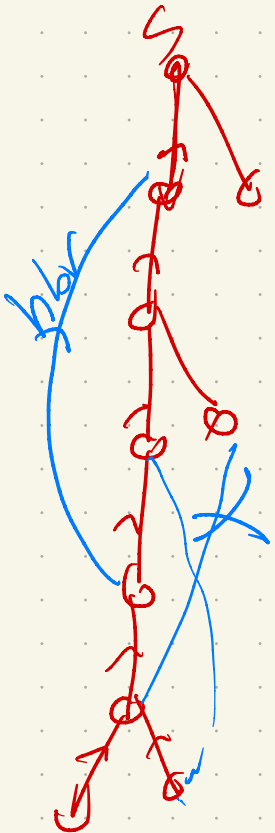


Figure 5.12. A depth-first spanning tree and a breadth-first spanning tree of the same graph, both starting at the center vertex.

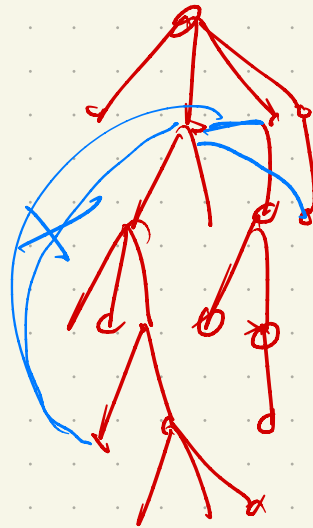
DFS:

All non-tree edges must connect a vertex to an ancestor in the tree



BFS:

All non-tree edges must connect vertices either at the same level, or 1 level apart



Correctness:

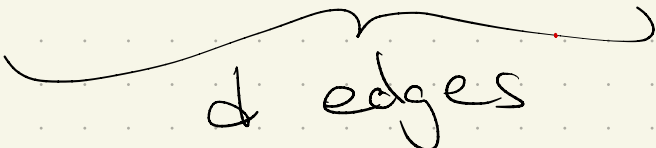
Claim: BFS will mark all reachable vertices.

Pf: induction on distance to the source:

$d=0$:

$d > 0$: Consider v at distance d in G , so

$$S \rightarrow v_1 \rightarrow v_2 \cdots \rightarrow v_{d-1} \rightarrow v_d$$


 d edges

By IH:

Runtime!

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

Claim: marked v's + parents form a spanning tree.

proof:

WHATEVERFIRSTSEARCH(s):

put $(\emptyset, s)$ in bag

while the bag is not empty

take (p, v) from the bag $(*)$

if v is unmarked

mark v

$\text{parent}(v) \leftarrow p$

for each edge vw $(\dagger)$

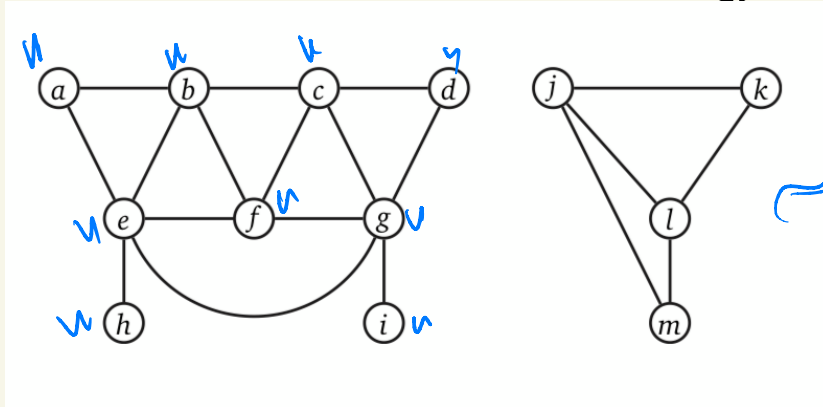
put (v, w) into the bag $(**)$

For each marked vertex:

In a disconnected graph:

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

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



Solution: while unmarked vertices remain
repeat the WFS

Modification: Might want to count the # of connected components:

COUNTCOMPONENTS(G):

count $\leftarrow 0$

for all vertices v

unmark v

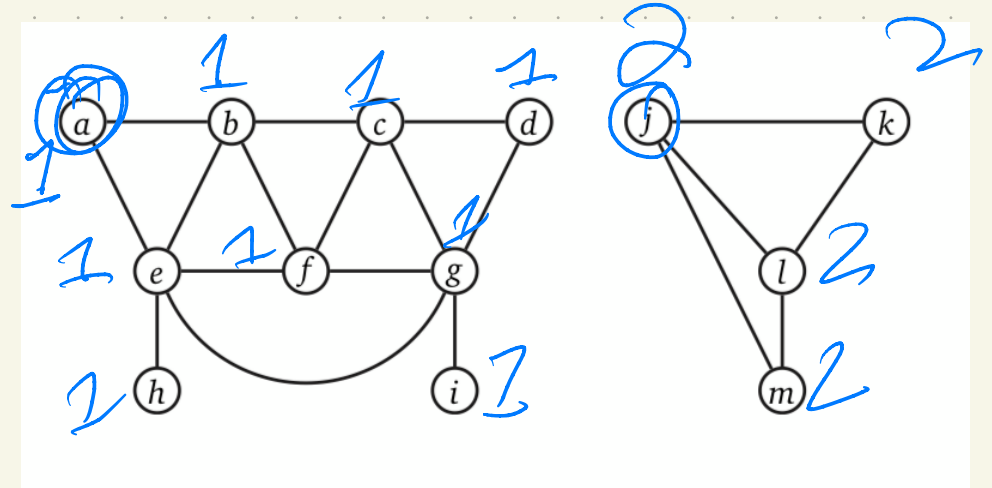
for all vertices v

if v is unmarked

count $\leftarrow$ **count** + 1

WHATEVERFIRSTSEARCH(v)

return count



earlier
code

return 2

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

COUNTANDLABEL(G):

$count \leftarrow 0$

for all vertices v

unmark v

for all vertices v

if v is unmarked

$count \leftarrow count + 1$

$LABELONE(v, count)$

return $count$

⟨⟨Label one component⟩⟩

$LABELONE(v, count)$:

while the bag is not empty

take v from the bag

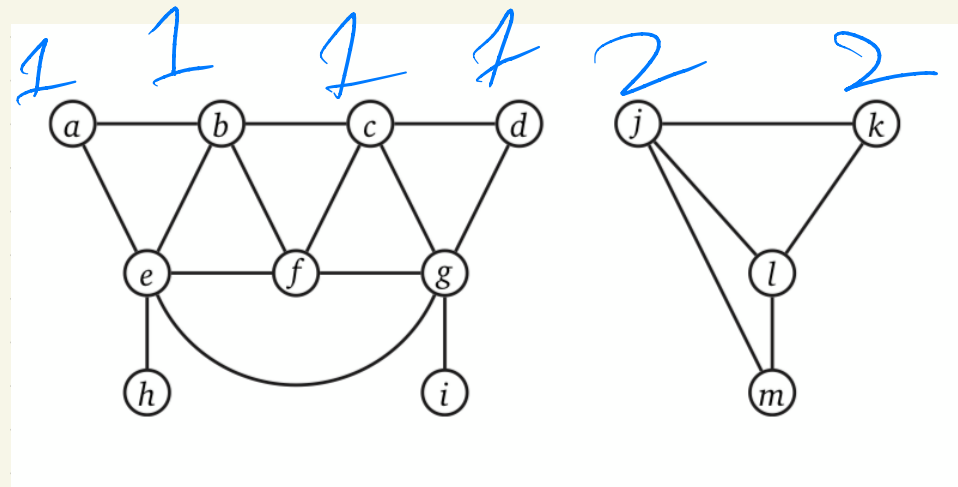
if v is unmarked

mark v

$comp(v) \leftarrow count$

for each edge vw

put w into the bag



Dfn: Reduction

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

Why?

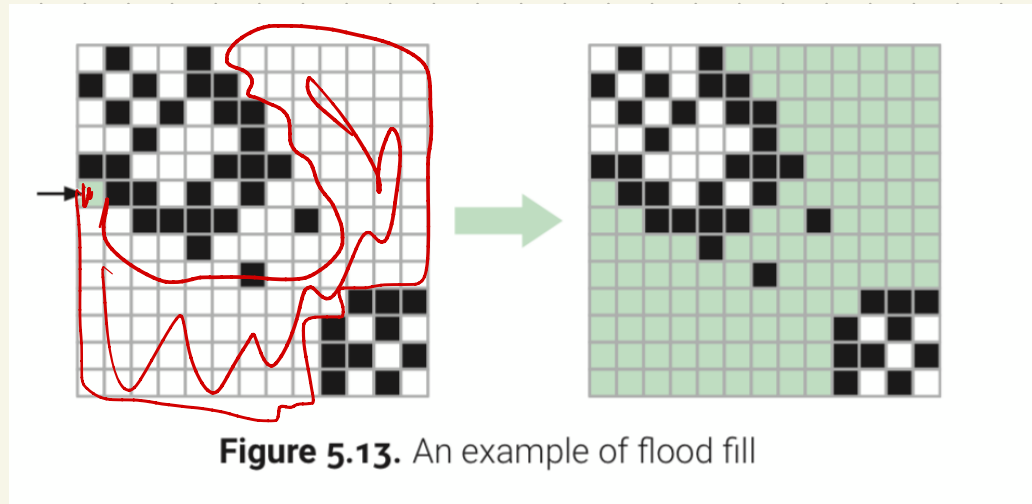
Reuse code!

conn comps

Used WFS

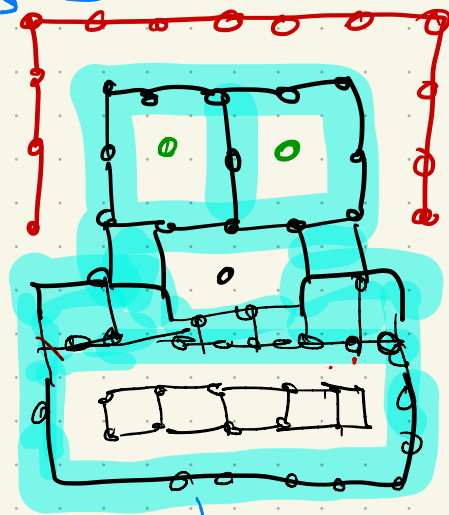
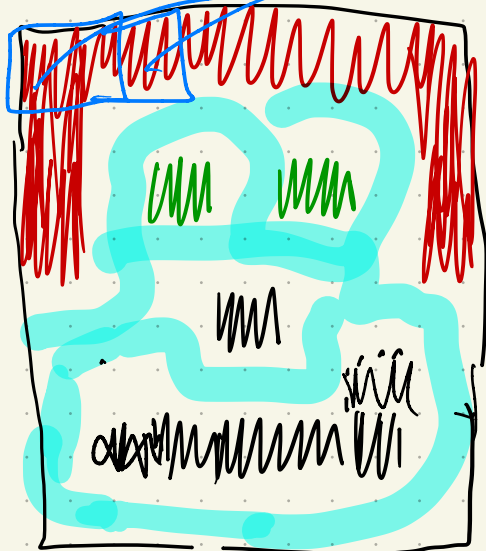
Reduction graph example:

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



How to implement?
use WFS

So: Build a graph from pixels:



each pixel
↳ vertex in G
add edge b/t
any two vertices
whose pixels
are in
same region

Algorithm

$n \times n$ → graph with n^2 vertices

If pixel p is selected:

Find connected comp of p 's
equiv. vertex in graph
recolor anything in that comp

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 Construct V or E

- Call some algorithm

↳ WFS = (BFS or DFS)
connected comps

⋮