# Complexity & Algorithms, Spring 2026

## Intro to Graphs

# Recap

- Posted HW4 (or will be soon)
- Readings → after class
- Today: Ch 5 of Jeff Erickson's book
- HW3 & Midterms → graded by Tuesday

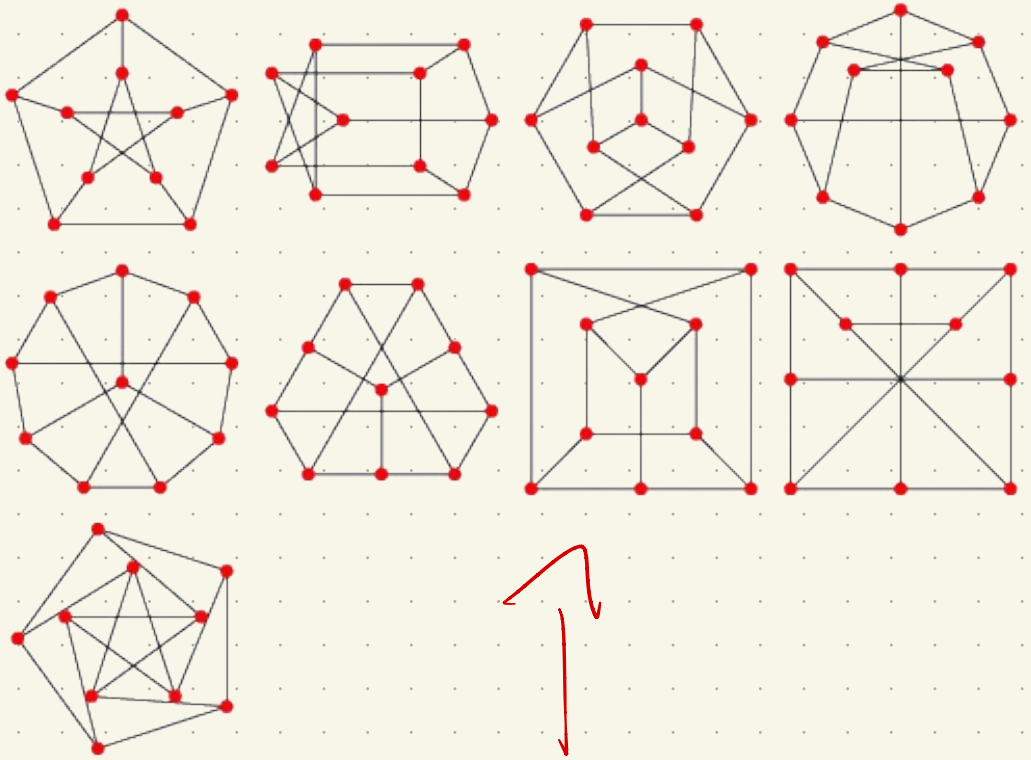
# Graphs

A graph $G = (V, E)$ is an ordered pair of 2 sets:

$V$ = vertices = $\{v_1, \dots, v_n\}$ $\quad |V|$ or $V$

$E$ = edges = $\{\{u, v\} \mid u, v \in V\}$ unordered

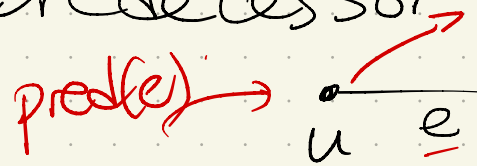
We often draw them, but they do not come with coordinates



# Definitions:

- Vertices (nodes), $V$
- Edges, $E$
- endpoints of an edge $e \in E$: $e = uv$
- head & tail
  - $u \to v$ or $(u, v)$ ← ordered pair $\neq (v, u)$
- simple: no parallel edges or 1-edge loops

- adjacent: two vertices $u \neq v$ are adj if $uv \in E$
- degree(v): # of adjacent vertices
- predecessor & successor

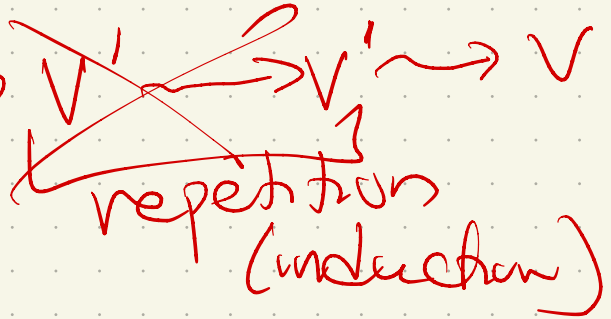



- indegree & outdegree if directed:

# More!

- Subgraph: general: $V' \subseteq V$, $E' \subseteq E$ s.t. all endpts of edges in $E'$ are in $V'$
  - induced: fix subset of $V$, take all possible edges using those vertices
- Walk: $v_0 e_1 v_1 e_2 v_2 \ldots$ s.t. $e_i = \{v_{i-1}, v_i\}$
- Path: no repeated vertices or edges

Note: If you have a walk $u \rightsquigarrow v$ can make a path. How?




- connected: $\forall u, v \in V, \exists$ a $u$ to $v$ path
- closed: end at start vertex
- cycle: path except 1st vertex = last
- tree: acyclic, connected, $n-1$ edges } any 2 suffice

First: some "easy" bounds

**Lemma:** $|E| \leq \frac{V(V-1)}{2}$ for simple + undirected graphs.

pf: no loops or parallel edges, so at most each pair of vertices gets one edge

$$|E| \leq \binom{V}{2} = \frac{V \cdot (V-1)}{2} \Rightarrow E = O(V^2)$$

**Lemma:** $\sum_{v \in V} d(v) = 2E$ "Handshaking lemma"

$= 2 \cdot \deg(v_1) + d(v_2) + \cdots d(v_n)$

proof:

in this sum, every $e \in E$ appears twice: $+1$ for $v$ ($e = uv$), $+1$ for $u$

$\Rightarrow 2 \times$ # edges

First question

Computers don't do well with images!
So pictures won't help them.

We need to store this info (somehow).

Ideas from data structures:

- lists
- vectors
- hash tables
- trees
- ...

tradeoffs!

# Adjacency (or vertex) lists:

$V_1$: $V_2, V_5$ ← $d(v_1)$ ← (u, v)

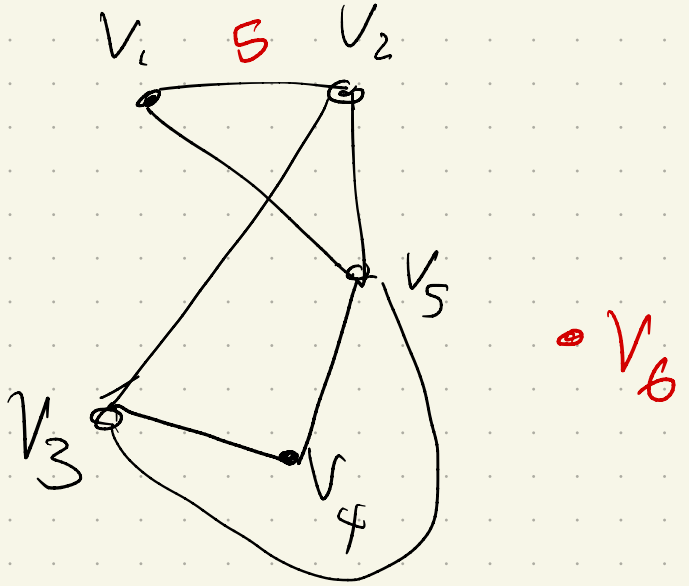
$V_2$: $V_1, V_3, V_5$ ← $d(v_2)$

$V_3$:

$V_4$:

$V_5$:

$V_6$:




Size: $V + \sum_{v \in V} d(v) = \Theta(V+E)$

$\sum_{v \in V} d(v) = 2E$

lookup: time to check $uv \in E$: $O(V)$

cost to lookup in u or v's list

time to loop over all neighbors of v:

$O(d(v))$

# Implementation:

More buried data structures!

Could use:

ptrs for each vertex

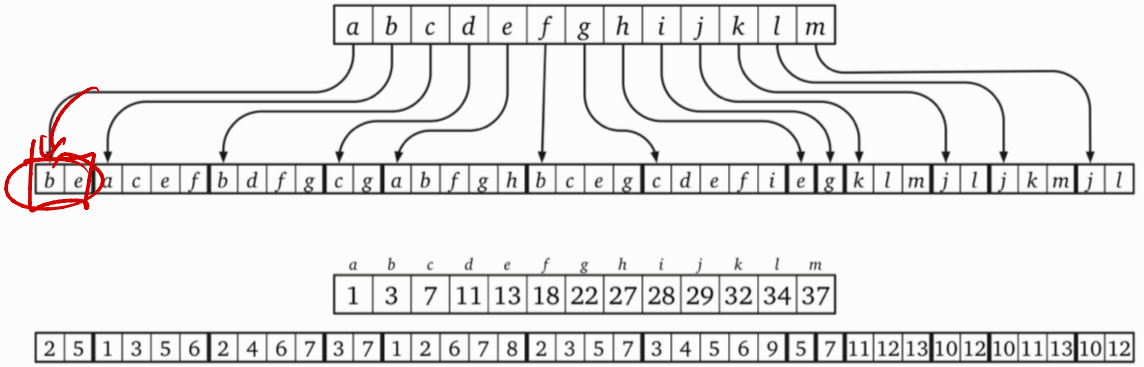



**Figure 5.10.** An abstract adjacency array for our example graph, and its actual implementation as a pair of integer arrays.

array based

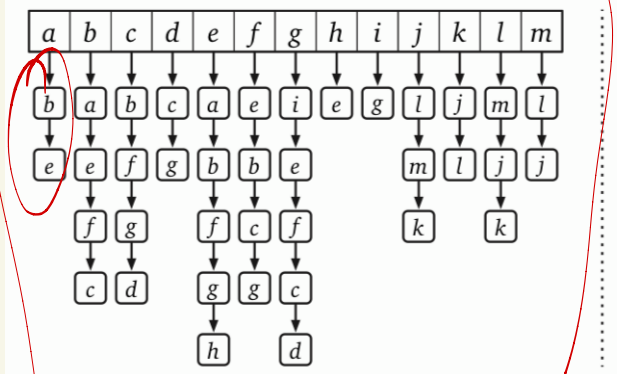
lookup: $O(\log n)$ not $O(n)$

But: slow insertion

linked:

insertion: $O(1)$

lookup: $O(n)$



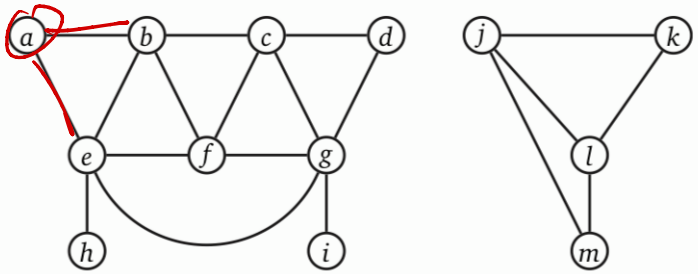
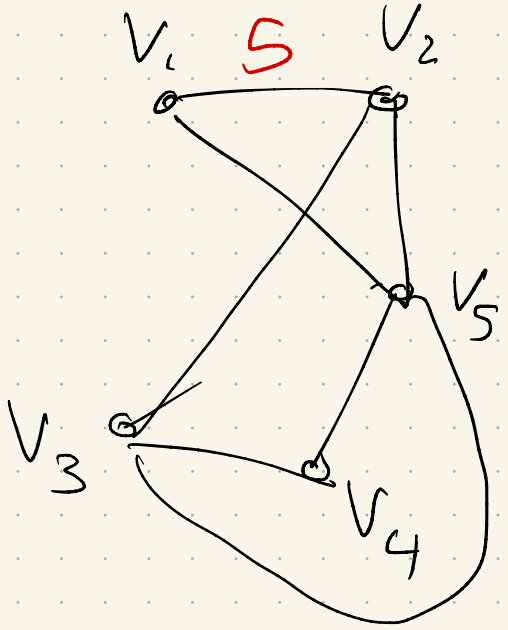




**Figure 5.9.** An adjacency list for our example graph.

# Adjacency Matrix

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
|---|---|---|---|---|---|
| $v_1$ | 5 | | 0 | 0 | 1 |
| $v_2$ | 1 | | | | 1 |
| $v_3$ | | | | | |
| $v_4$ | | | | | |
| $v_5$ | | | | | |




space: $O(n^2)$

check nbr: $O(1)$

loop over neighbors: $O(n)$

(not $O(d(v))$)

# Implementation:

## More data structures!

| | a | b | c | d | e | f | g | h | i | j | k | l | m |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| b | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| c | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| d | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| e | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| f | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| h | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| i | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| j | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| k | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| l | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| m | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

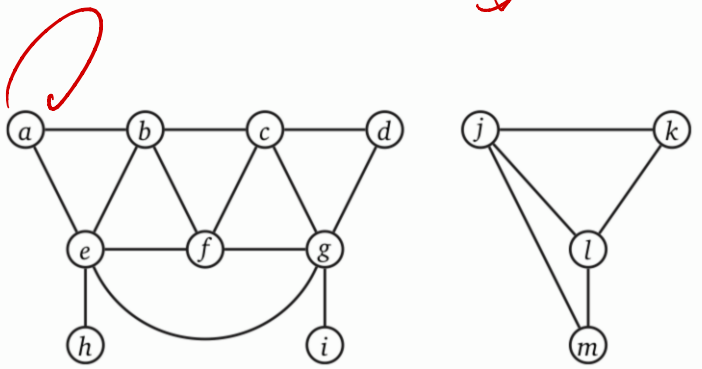



**Figure 5.11.** An adjacency matrix for our example graph.

update: $O(1)$ if add edge

Which is better?

Depends!

| | Adjacency matrix | Standard adjacency list (linked lists) | Adjacency list (hash tables) |
|---|---|---|---|
| Space | $\Theta(V^2)$ | $\Theta(V+E)$ | $\Theta(V+E)$ |
| Time to test if $uv \in E$ | $O(1)$ | $O(1+\min\{\deg(u),\deg(v)\}) = O(V)$ | $O(1)$ |
| Time to test if $u \to v \in E$ | $O(1)$ | $O(1+\deg(u)) = O(V)$ | $O(1)$ |
| Time to list the neighbors of $v$ | $O(V)$ | $O(1+\deg(v))$ | $O(1+\deg(v))$ |
| Time to list all edges | $\Theta(V^2)$ | $\Theta(V+E)$ | $\Theta(V+E)$ |
| Time to add edge $uv$ | $O(1)$ | $O(1)$ | $O(1)^*$ |
| Time to delete edge $uv$ | $O(1)$ | $O(\deg(u)+\deg(v)) = O(V)$ | $O(1)^*$ |

Here:

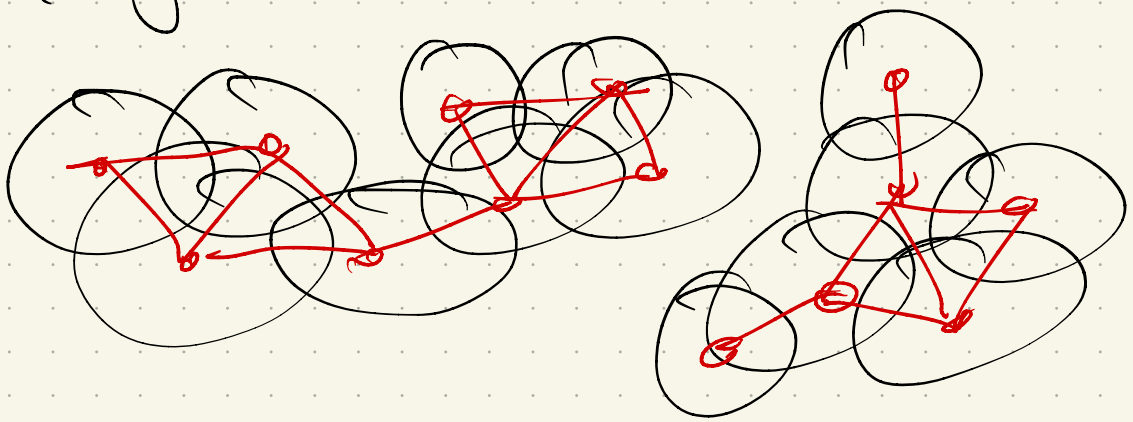
***In the rest of this book, unless explicitly stated otherwise, all time bounds for graph algorithms assume that the input graph is represented by a standard adjacency list.*** Similarly, unless explicitly stated otherwise, when an exercise asks you to design and analyze a graph algorithm, you should assume that the input graph is represented in a standard adjacency list.

Really - might depend on input!

- size of graph
- freq. of changes
- representation: usually, some "word problem" is handed to you!

You'll have to build the graph

Ex: Given a set of overlapping circles, find the largest set where no 2 intersect.



Build G'

↓

ind set!

Even more:

- Space available
- language used
- previous "legacy" code
- other developers

⋮

To reiterate:

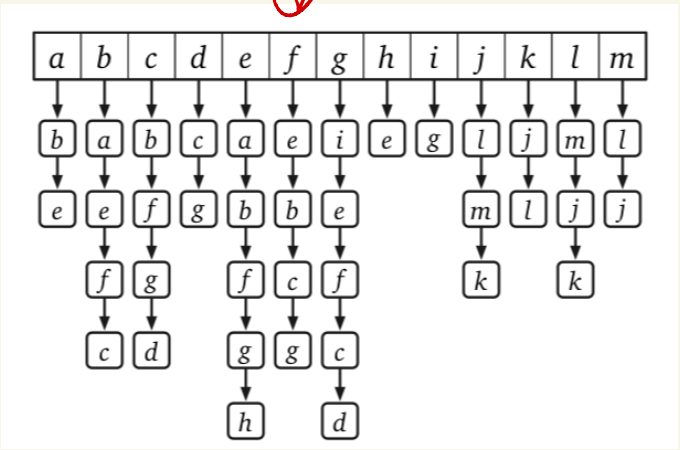
***In the rest of this book, unless explicitly stated otherwise, all time bounds for graph algorithms assume that the input graph is represented by a standard adjacency list.*** Similarly, unless explicitly stated otherwise, when an exercise asks you to design and analyze a graph algorithm, you should assume that the input graph is represented in a standard adjacency list.

# Graph Searching

How can we tell if 2 vertices are connected?

Remember the computer only has:

Is i connected to a?

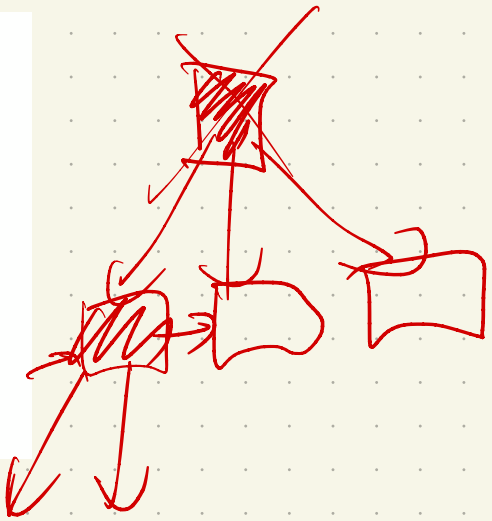



Bigger question: can we tell if all the vertices are in a single connected component?

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

"bag":
stack
or queue

```
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
```

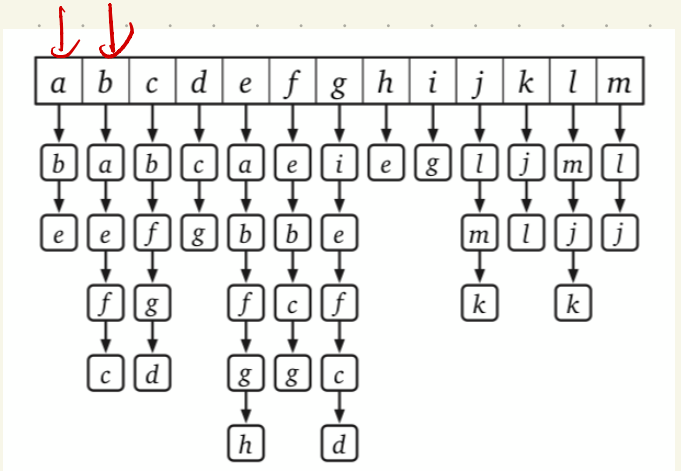


These are essentially just search strategies:
How can we decide if u ↔ v are connected?

WFS(u)
check if v is marked at end.

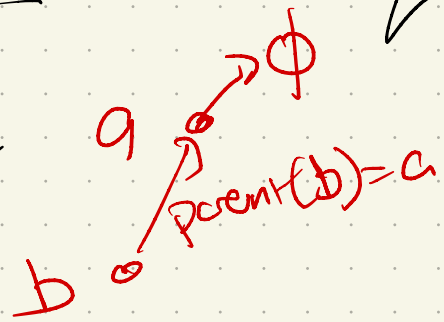
Can use this to build a spanning tree:

```
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         (⋆⋆)
```

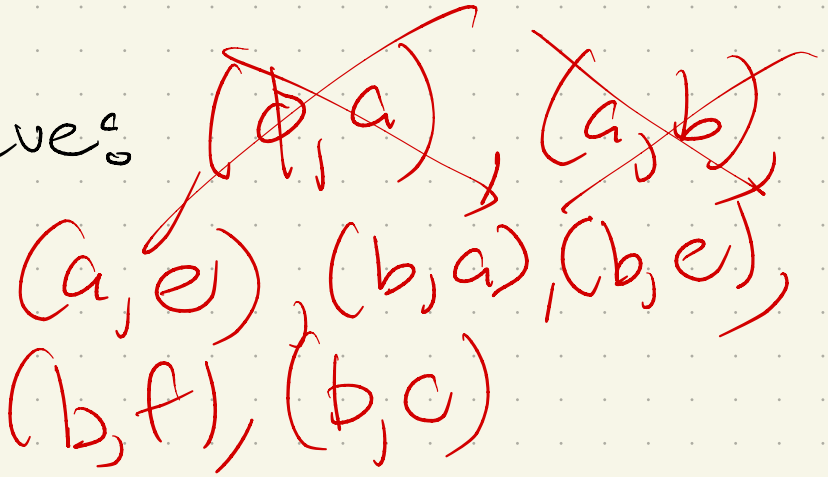



BFS: use a queue:

Tree:

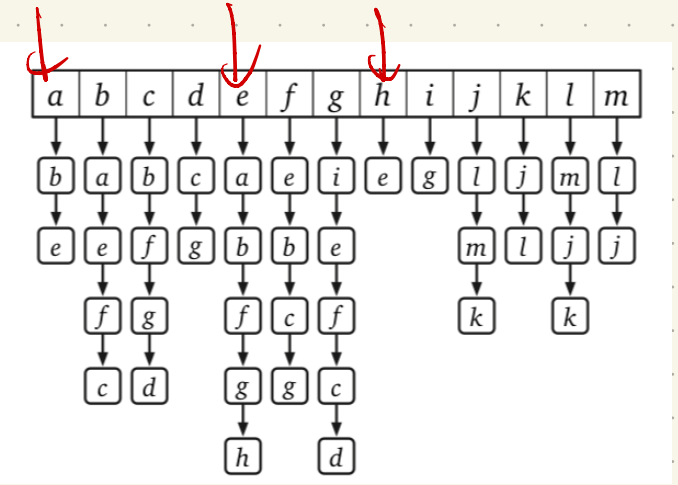


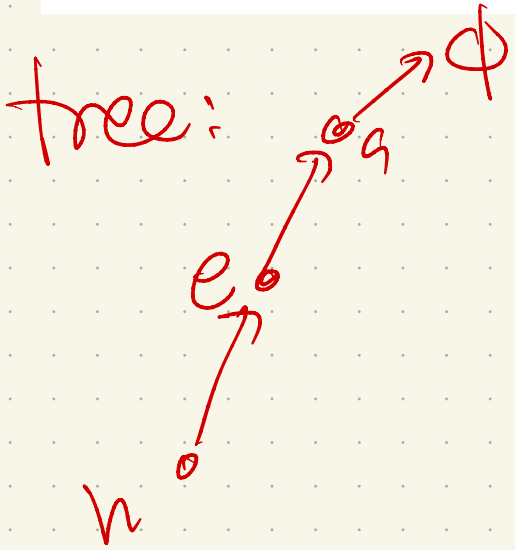

Queue:



# DFS: use a stack

```
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         (⋆⋆)
```




stack:

~~(ϕ, a)~~, (a, b), ~~(a, e)~~

(e, a), (e, b), (e, f)

(e, g), ~~(e, h)~~, ~~(h, e)~~

Just remember: different!

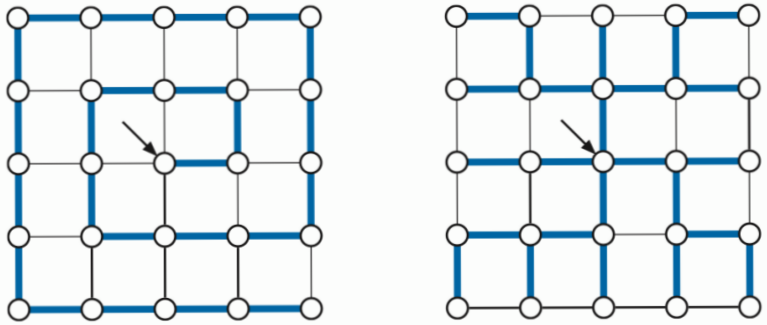
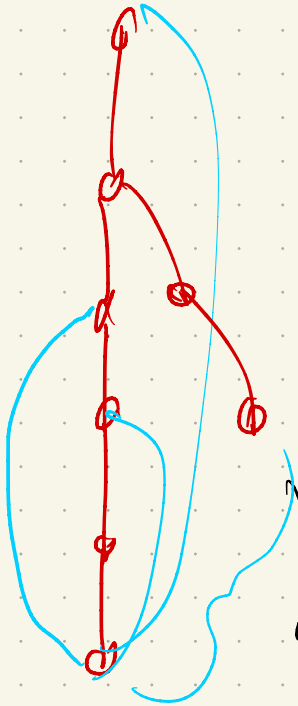


**Figure 5.12.** A depth-first spanning tree and a breadth-first spanning tree of the same graph, both starting at the center vertex.

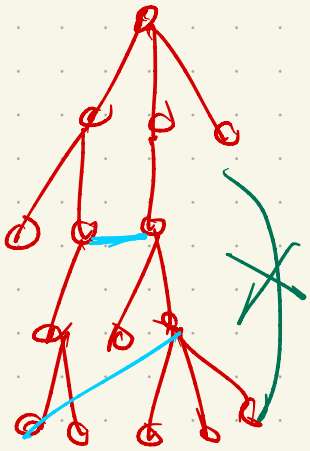
DFS:

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:** WFS will mark all reachable vertices. (from $\subseteq$)

**Pf:** induction on distance to the source:

**$d=0$:** know we mark $\subseteq$ ✓

Base case

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

$$s \to v_1 \to v_2 \cdots \to v_{d-1} \to v_d$$

($d$ edges)

**By IH:** mark all vertices at distance $d-1$

when $v_{d-1}$ marked, added $v_d$ to stack or queue $\rightsquigarrow$ will be marked before DFS is empty

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
```

Stack or queue: $O(1)$

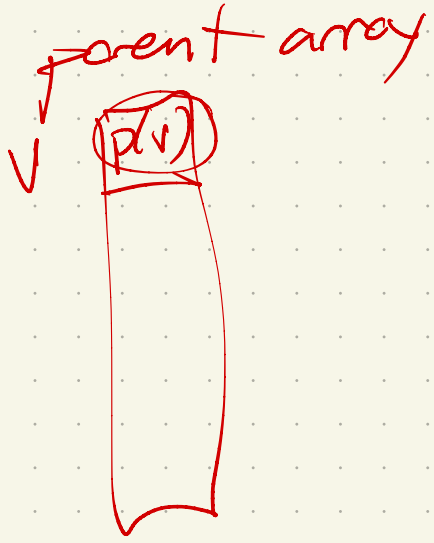
- Each vertex can be added to the ~~stack~~ bag $d(v)$ times
- each vertex is marked once

↳ $O(V+E)$ or $O(m+n)$

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

**proof:**

```
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         (⋆⋆)
```




For each marked vertex:

add a single parent except for source

$\Rightarrow$ $n-1$ edges in this graph

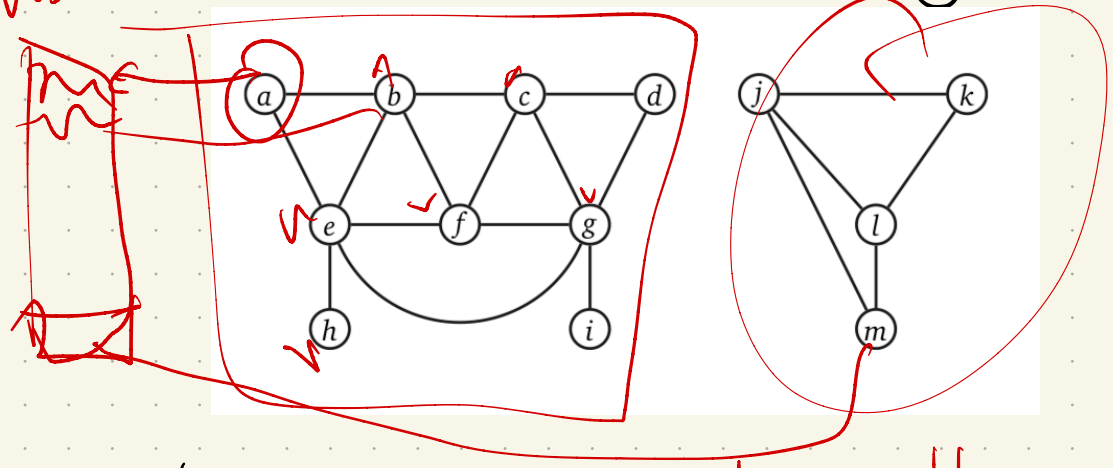
If G is connected, mark every vertex

$\Rightarrow$ tree,

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.)

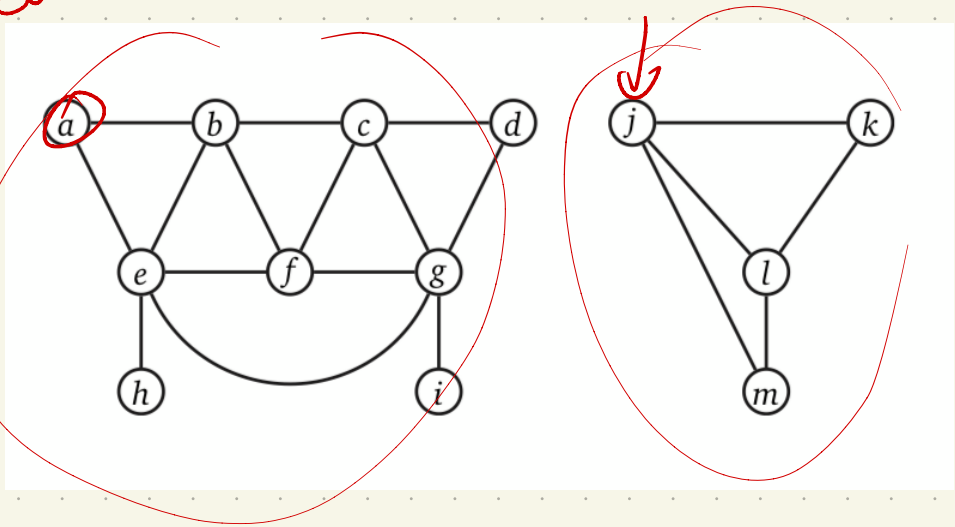



Solution: unmark all vertices
while some v is unmarked
WFS(v)

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

count = 2

```
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
```




→ 2

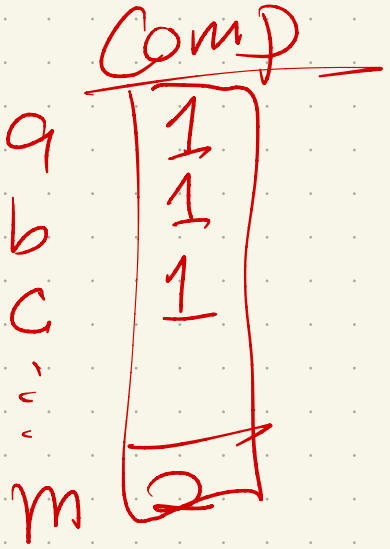
$O(V+E)$

is h connected to m?

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
```




Check if u is connected to v:
is comp(u) = comp(v)?