Whatever First Search(s):

marks all vertices reachible from s

finds edges making a spanning tree of s's component

uses a "bag" that you can add edges to & remove them later

different bags give different running times & useful spanning trees

(use a queue for O(V+E) time.)

```
WFSALL(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          WHATEVERFIRSTSEARCH(v)
```

marks vertices component by component (one WFS call per component)

```
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
```

```
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
```

Put v in bag

After running, comp(u) = comp(v) iff u & v are in same component.

If T is the time to add & remove from bag, CountAndLabel takes $O(V + ET)$ time.

Directed Graphs

If you want to mark vertices reachible from s (so there is an s,a-path) call:

```
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 v→w
              put w into the bag
```

$O(V + ET)$ time

Given a pixal map

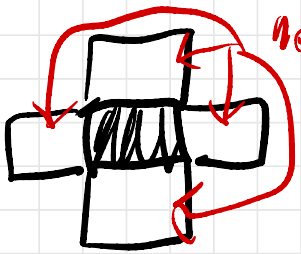
P[1..n, 1..n].

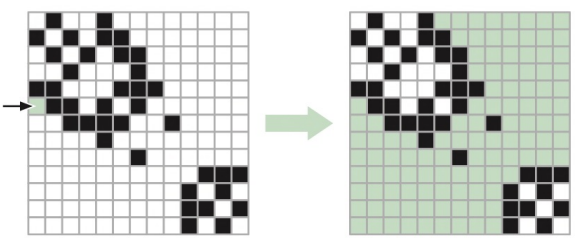
Each entry is called a pixel & its value is the color of the pixel.

Flood fill tool: select a pixel & set it and all pixels in its connected region the same color.

connected region: connected subset of pixels where two

are considered adjacent if they are neighbors & the same color.





Reduces to whatever First Search!

Need to describe graph, input to WFS, & what to do with results.

- build graph $G=(V,E)$

  V: set of pixels

  E: pairs of adjacent pixels

- call WhateverFirstSearch((i,j))

  ↑ selected pixel

- paint the vertices we mark

Give running times in terms of original input size!

If bag is a queue:

$T = O(1)$

So WFS $O(V+E)$

$|V| = n^2$

$|E| \leq 4n^2$

So alg runs in $O(n^2)$

Practical(!) considerations

- don't build graph; work with pixels & neighboring pixels directly
- now takes $O(k)$ time where k: # vertices we color

Use a stack as the bag in WFS...

Depth-first search

Usually written recursively

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  PostVisit(v)
```

```
DFSAll(G):
  Preprocess(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

$O(V+E)$ time

```
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
```

finishing time for v

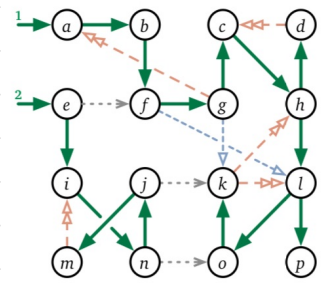
starting time of v

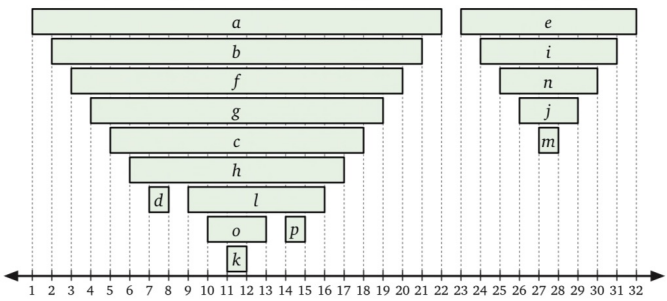
[v.pre, v.post]: active interval for v

active intervals either nest or are disjoint thanks to stack

also, if v.pre < u.pre < w.post < v.post, u is reachible from v

DFS(v) marks all vertices reachible from v that aren't already marked





order by v.pre for a preorder

order by v.post for a postorder

Say we are in the middle of a DFSALL...

Will compare current clock to final pre + post values

Fix some vertex v. v is...

new if clock < v.pre (have not yet called DFS(v))

active if v.pre ≤ clock < v.post

finished if v.post ≤ clock (DFS(v) has returned)

Consider $u \Rightarrow v$ at the moment DFS(u) begins.

If v is new, $u.pre < v.pre < v.post < u.post$.

If DFS(u) calls DFS(v) directly, $u \Rightarrow v$ is tree edge.

Otherwise, $u \Rightarrow v$ is a forward edge.

If v is active,

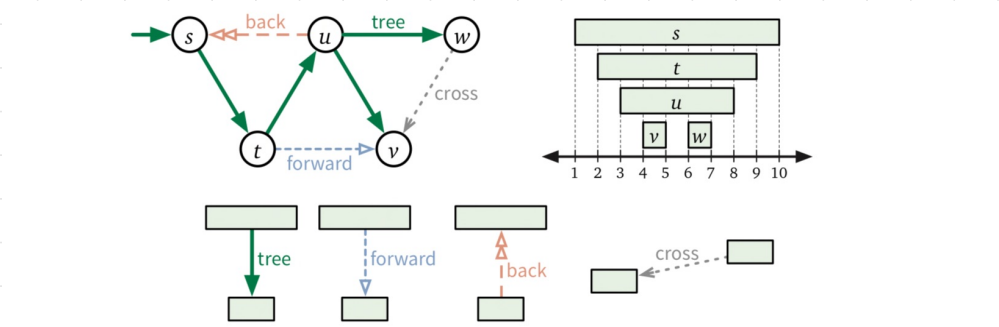
$v.pre < u.pre < u.post < v.post$

$u \Rightarrow v$ is a back edge

If v is finished,

$v.post < u.pre$

$u \to v$ is a cross edge



Edge types & orders depend on order you loop over vertices & edges.