Algorithms

Graphs:
BFS + DFS

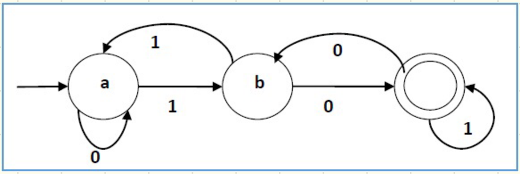
Recap

- Midterms - back Friday
  (+ grades in Blackboard + Banner)
- Reading due this week before class
- HW5 - due next Wed.
- Talk today:
  3pm, 115 Ritter
  (on graph drawing!)

Last Lecture: Graphs

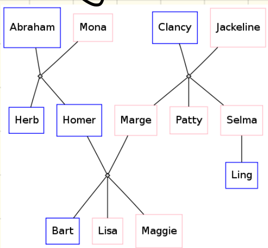
(Because they model everything!)

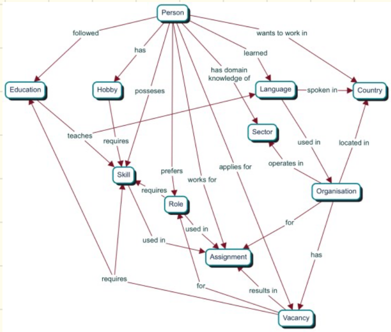
DFA:



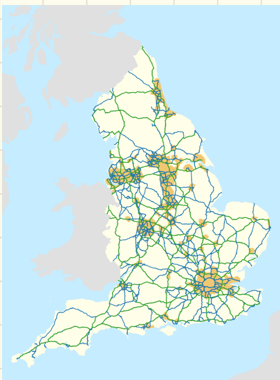
concept map:

lineages:





road network:



Dfns: $G = (V, E)$

$\cancel{n = |V|}$ V

$\cancel{m = |E|}$ E

paths: no repeated edges / verts

cycles: paths $v_1, \ldots v_k$ where $v_1 = v_k$

walks:

degree-sum: $\sum_v d(v) = 2|E|$

connected: if every vertex has a path to every other vert.

if not: (connected) components

Representations: 2 main ways

1

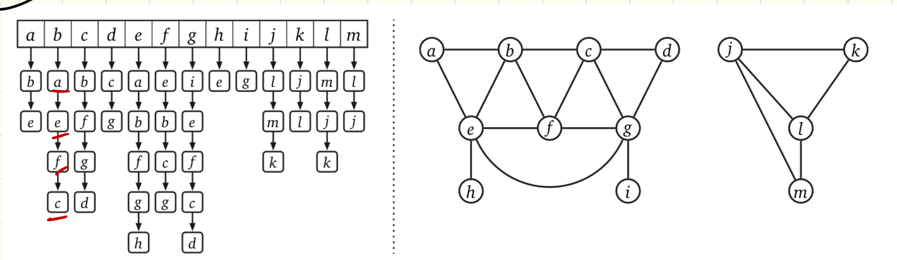


Figure 5.9. An adjacency list for our example graph.

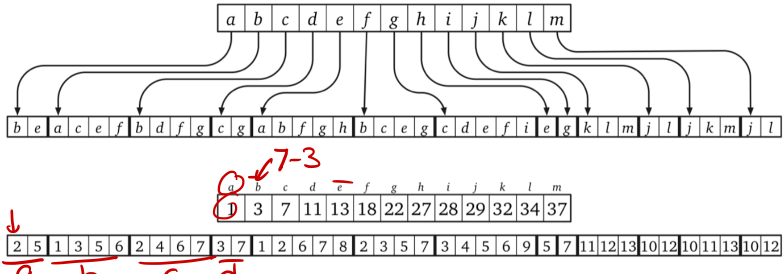


Figure 5.10. An abstract adjacency array for our example graph, and its actual implementation as a pair of integer arrays.

$\boxed{a}$: b → e

b : a → c → e → f

2

	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~~ 10	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

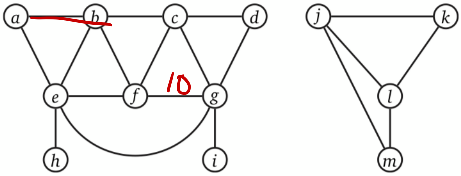


Figure 5.11. An adjacency matrix for our example graph.

End result:

	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

Table 5.1. Times for basic operations on standard graph data structures.

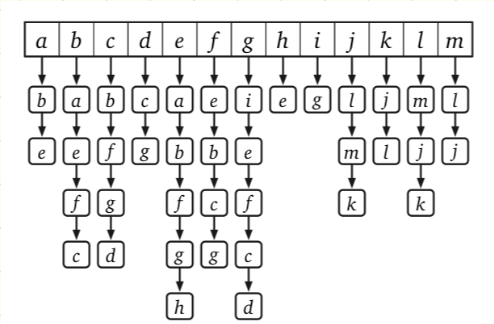
In this class:

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:



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

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

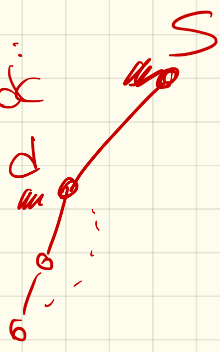
```
WhateverFirstSearch(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

These are essentially just search strategies:

How can we decide if u + v are connected?

(See demo...)

DFS: stack



"bag" a b c ~~d~~

b f g

BFS: queue

Can use this to build a spanning tree:

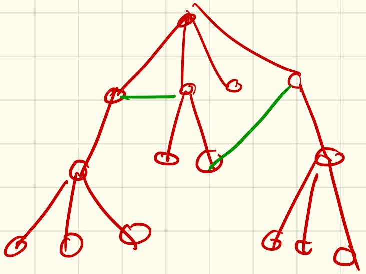
```
WhateverFirstSearch(s):
    put (∅, s) in bag
    while the bag is not empty
        take (p, v) from the bag              (⋆)
        if v is unmarked
            mark v
            parent(v) ← p
            for each edge vw                  (†)
                put (v, w) into the bag       (⋆⋆)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

O(1) to add/remove!

O(V+E)

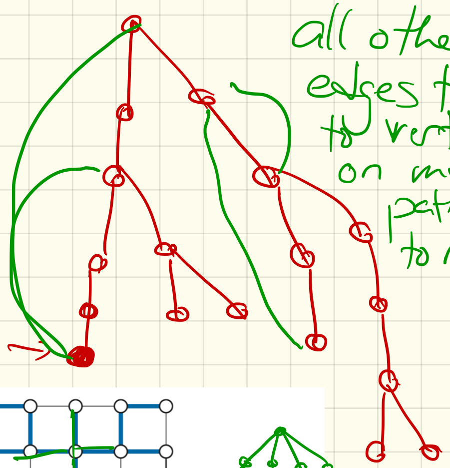
remembering 1st node to reach v

BFS tree:



all other edges go inside a level, or b/t nbr levels

DFS tree:



all other edges to vert. on my path to root

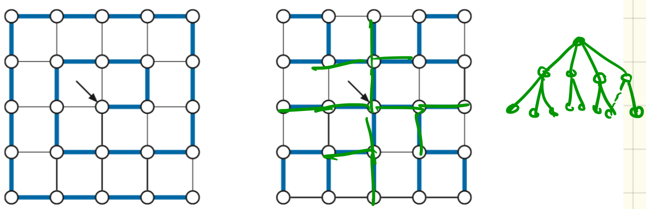
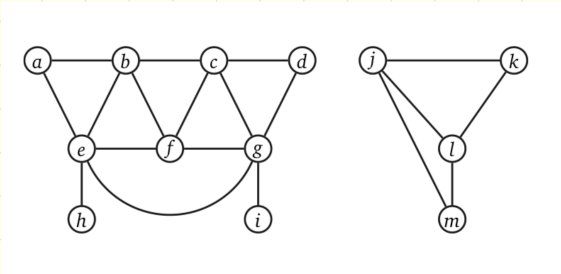


Figure 5.12. A depth-first spanning tree and a breadth-first spanning tree of the same graph, both starting at the center vertex.

In a disconnected graph:

Often want to count or label the <u>components</u> of the graph!

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

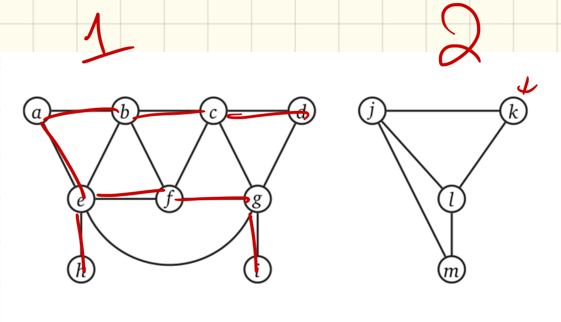


Solution: Call it more than one time!

<u>un</u>mark all vertices
for all vertices v:
  if not marked:
    run WFS(v)

Might want to count the # of components:

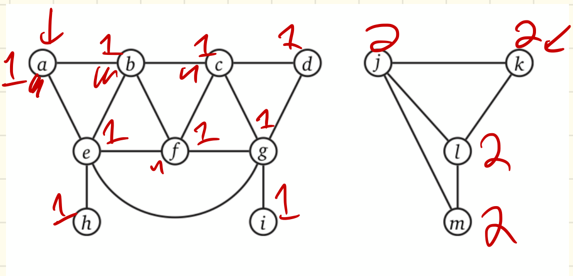
```
COUNTCOMPONENTS(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            WHATEVERFIRSTSEARCH(v)
    return count
```



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
```



Dfn: Reduction

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

We'll see a ton of these!

(Especially common in graphs...)

Key:
- What graph to build
- What algorithm to use

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.

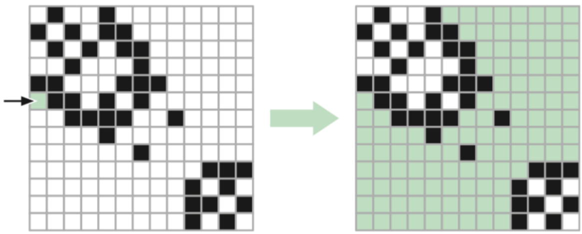


Figure 5.13. An example of flood fill

How?