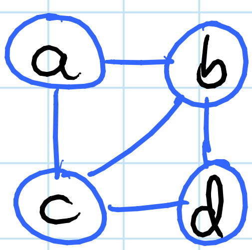
6.006	Lecture 13	Oct. 27, 2011

TODAY: Graphs I: BFS (I of 2)
- applications of graph search
- graph representations
- breadth-first search

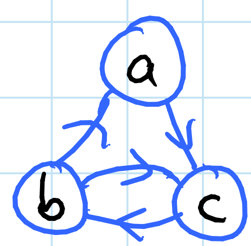
Recall: graph $G = (V, E)$
- V = set of vertices (arbitrary labels)
- E = set of edges i.e. vertex pairs (v, w)
  - ordered pair ⇒ directed edge & graph
  - unordered pair ⇒ undirected

e.g.



UNDIRECTED

$V = \{a, b, c, d\}$
$E = \{\{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{c,d\}\}$



DIRECTED

$V = \{a, b, c\}$
$E = \{(a,c), (b,c), (c,b), (b,a)\}$

Graph search: "explore a graph"
- e.g. find a path from start vertex s to a desired vertex
- e.g. visit all vertices or edges of graph, or only those reachable from s

Applications: many

- web crawling (how Google finds pages)
- social networking (Facebook friend finder)
- network broadcast routing
- garbage collection
- model checking (finite state machine)
- checking mathematical conjectures
- solving puzzles & games

Pocket Cube: $2\times2\times2$ Rubik's cube

- configuration graph:
  - vertex for each possible state
  - edge for each basic move (e.g., 90° turn) from one state to another
  - undirected: moves are reversible

11 for $2\times2\times2$ ← diameter ("God's Number")

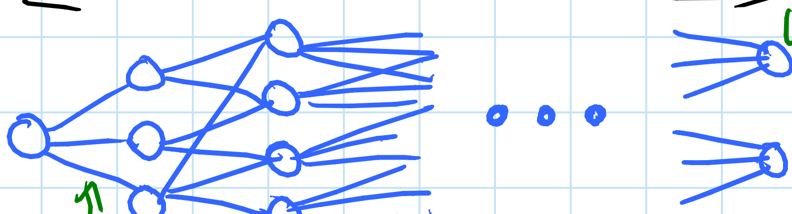
20 for $3\times3\times3$

$\Theta(n^2/\lg n)$ for $n\times n\times n$

[Demaine, Demaine, Eisenstat, Lubiw, Winslow 2011]

"hardest configs."

solved



possible first moves

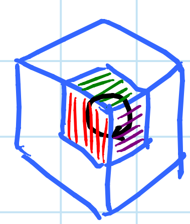
reachable in two steps but not one

"breadth-first tree"

- # vertices $= 8! \cdot 3^8 = 264{,}539{,}520$

8 cubelet in arbitrary positions

each cubelet has 3 possible twists



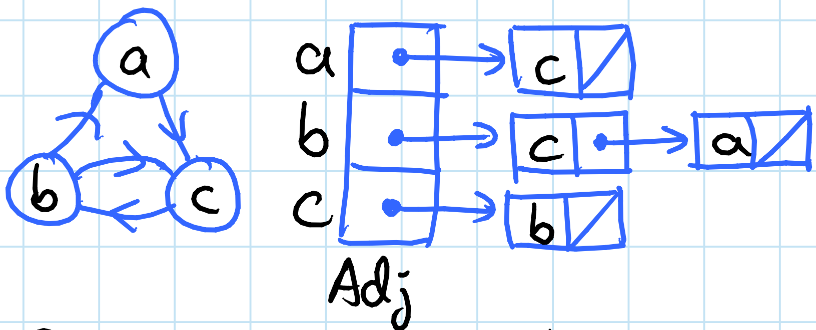
$\times\ 1/24$ if we remove cube symmetries

$\times\ 1/3$ actually reachable (3 conn. components)

Graph representation: (data structures)

Adjacency lists: array Adj of $|V|$ linked lists
- for each vertex $u \in V$, Adj[u] stores u's neighbors, i.e. $\{v \in V \mid (u,v) \in E\}$
  just outgoing edges if directed

e.g.



Space: $\Theta(V+E)$

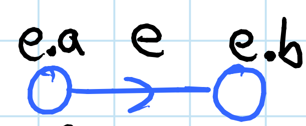
- in Python: Adj = dictionary of list/set values
  vertex = any hashable object (e.g., int, tuple)
- advantage: multiple graphs on same vertices

Implicit graphs: Adj(u) is a function
- compute local structure on the fly
  e.g. Rubik's Cube

"zero" space

Object-oriented variations:
- object for each vertex u
- u.neighbors = list of neighbors i.e. Adj[u]
  (or method for implicit graphs)

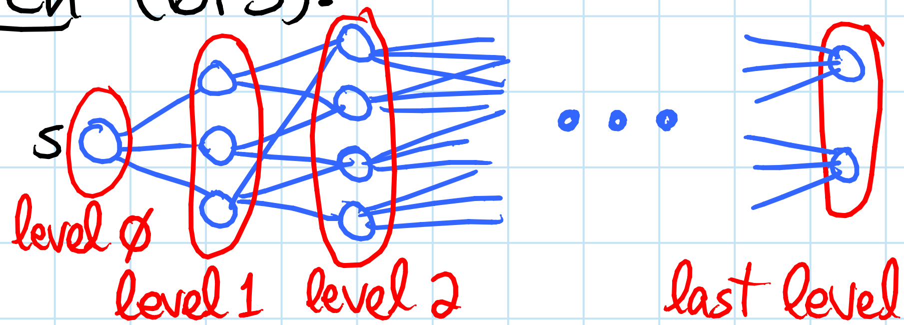
"Incidence lists:"
- can also make edges objects



- u.edges = list of (outgoing) edges from u
- advantage: store edge data without hashing

Breadth-first search (BFS):

explore graph
level by level
from s

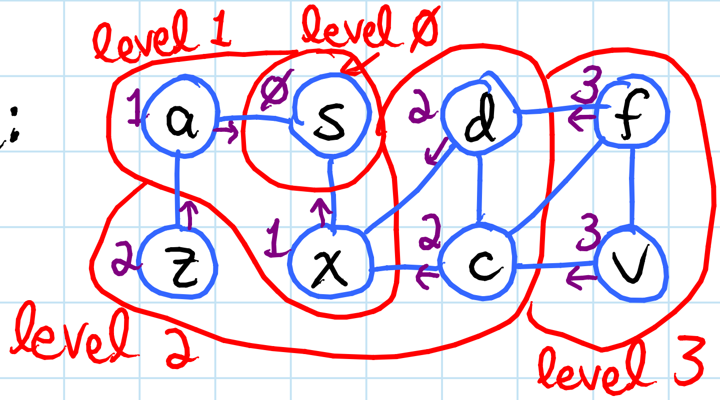


- level ∅ = {s}
- level i = vertices reachable by path of i edges but not fewer
- build level i>∅ from level i−1
  by trying all outgoing edges,
  but ignoring vertices from previous levels

[see CLRS for queue-based implementation]

```
BFS(s, Adj):
    level = {s: ∅}
    parent = {s: None}
    i = 1
    frontier = [s]          # previous level, i-1
    while frontier:
        next = []           # next level, i
        for u in frontier:
            for v in Adj[u]:
                if v not in level:     # not yet seen
                    level[v] = i       # = level[u]+1
                    parent[v] = u
                    next.append(v)
        frontier = next
        i += 1
```

Example:



$frontier_\emptyset = \{s\}$
$frontier_1 = \{a, x\}$
$frontier_2 = \{z, d, c\}$
$frontier_3 = \{f, v\}$
(not x, c, d)

Analysis:

- vertex v enters next (& then frontier) only once (because level[v] then set)
  - base case: $v = s$

$\Rightarrow$ Adj[v] looped through only once

- time $= \sum_{v \in V} |Adj[v]| = \begin{cases} |E| & \text{for directed graphs} \\ 2|E| & \text{for undirected graphs} \end{cases}$

$\Rightarrow O(E)$ time

- $O(V+E)$ to also list vertices unreachable from v (those still not assigned level)

"LINEAR TIME"

Shortest paths: [cf. L15-18]

- for every vertex v, fewest edges to get from s to v is $\begin{cases} \text{level}[v] & \text{if } v \text{ assigned level} \\ \infty & \text{else (no path)} \end{cases}$
- parent pointers form shortest-path tree = union of such a shortest path for each v

$\Rightarrow$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)