6.006	Lecture 12	Mar. 18, 2008

Outline: Search I (of 3)

- graph search
- applications
- graph representations
- introduction to breadth-first & depth-first search

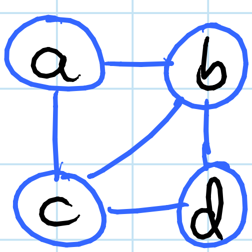
Reading: CLRS 22.1–22.3, B.4

Graph search: explore a graph
e.g. find a path from start vertex s to a desired vertex

Recall: graph $G=(V,E)$

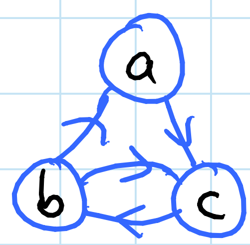
- V = set of vertices (arbitrary labels)
- E = set of edges i.e. vertex pairs (v,w)
  - ordered pair ⇒ directed edge & graph
  - unordered pair ⇒ undirected

e.g.



UNDIRECTED

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



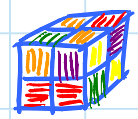
DIRECTED

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

Applications: many

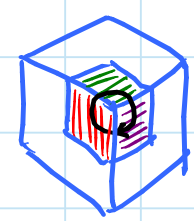
- web crawling (how Google finds pages)
- social networking (Facebook friend finder)
- computer networks: (routing in the Internet)
  shortest paths [next unit]
- solving puzzles & games
- checking mathematical conjectures

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



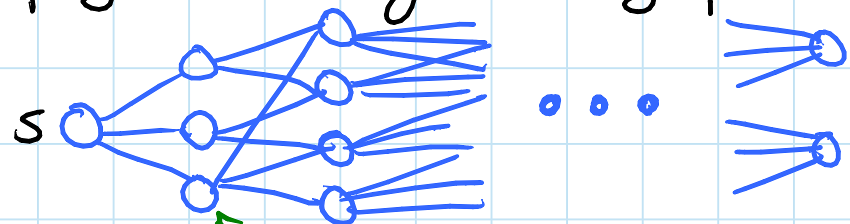
- configuration graph:
  - vertex for each possible state
  - edge for each basic move (e.g., 90° turn) from one state to another
  - undirected: moves are reversible
- puzzle: given initial state s, find a path to the solved state
- $\#\text{vertices} = 8! \cdot 3^8 = 264{,}539{,}520$

  8 cubelet in arbitrary positions — each cubelet has 3 possible twists



- can factor out 24-fold symmetry of cube: fix one cubelet
  $24 = 8 \cdot 3$
  $\Rightarrow 7! \cdot 3^7 = 11{,}022{,}480$
- in fact, graph has 3 connected components of equal size $\Rightarrow$ only need to search in one
  $\Rightarrow 7! \cdot 3^6 = 3{,}674{,}160$

"Geography" of configuration graph:



"breadth-first tree"

possible first moves

reachable in two steps but not one

distance	# reachable configurations: 90° turns	# reachable configurations: 90° & 180° turns
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1,847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,536
8	114,149	870,072
9	360,508	1,887,748
10	930,588	623,800
11	1,350,852	2,644 ← diameter
12	782,536	
13	90,280	
14	276 ← diameter	
	3,674,160	3,674,160

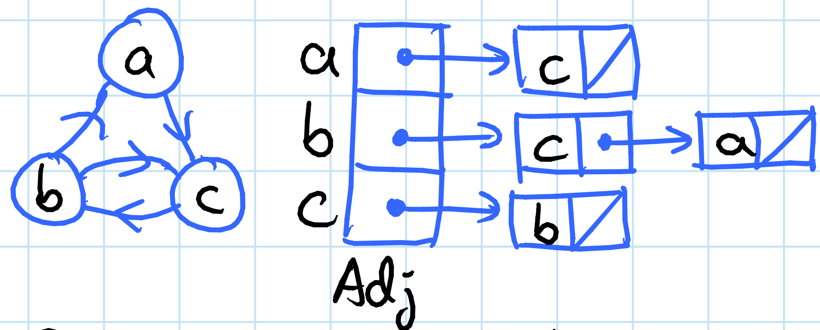
http://en.wikipedia.org/wiki/Pocket_Cube

Cf. 3×3×3 Rubik's cube: ≈1.4 trillion states
diameter unknown! ≤26

Representing graphs: (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

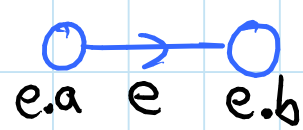
e.g.



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

Object-oriented variations:
- object for each vertex u
- u.neighbors = list of neighbors i.e. Adj[u]

"Incidence lists:"
- can also make edges objects



- u.edges = list of (outgoing) edges from u
- advantage: storing data with vertices & edges without hashing

Representing graphs: (cont'd)

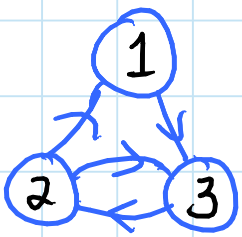
above representations are good for sparse graphs where $|E| \ll |V|^2$ — space requirement $= \Theta(V+E)$

don't bother with $|\cdot|$'s inside O/Θ

Adjacency matrix:

- assume $V = \{1, 2, \dots, |V|\}$ (number vertices)
- $A = (a_{ij}) = |V| \times |V|$ matrix
  where $a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ \emptyset & \text{otherwise} \end{cases}$

i = row
j = column

e.g. 

$$A = \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} \emptyset & \emptyset & 1 \\ 1 & \emptyset & 1 \\ \emptyset & 1 & \emptyset \end{pmatrix} \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}$$

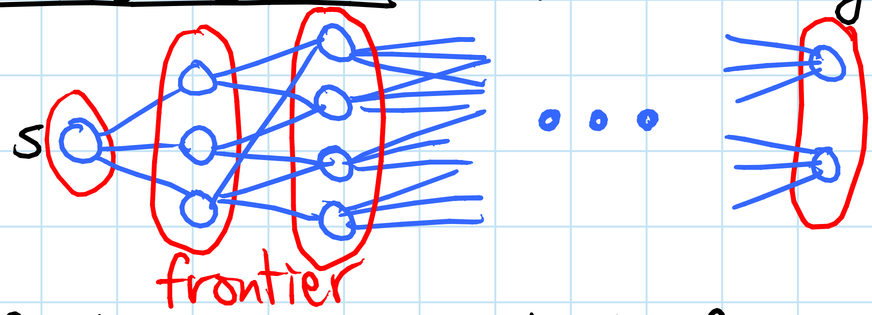
- good for dense graphs where $|E| \approx |V|^2$
- space requirement $= \Theta(V^2)$
- cool properties like A^2 gives length-2 paths & Google PageRank $\approx A^\infty$
- but we'll rarely use it
  (Google couldn't: $|V| \approx 20$ billion $\Rightarrow |V|^2 \approx 4 \cdot 10^{20}$)
  [50,000 petabytes]

Implicit graphs:

Adj(u) is a function
or u.neighbors/edges is a method
$\Rightarrow$ "no space" (just what you need now)

High-level overview of next two lectures:

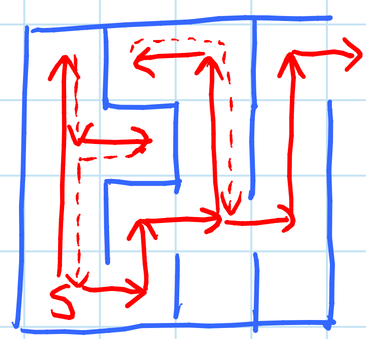
Breath-first search: levels like "geography"



- frontier = current level
- initially $\{s\}$
- repeatedly advance frontier to next level, careful not to go backwards to previous level
- actually find shortest paths
  i.e. fewest possible edges

Depth-first search: like exploring a maze

- e.g.: (left-hand rule)



- follow path until you get stuck
- backtrack along breadcrumbs
  until you reach an unexplored edge
- recursively explore it
- careful not to repeat a vertex