6.006 Introduction to Algorithms
Spring 2008

Lecture 12: Searching I: Graph Search and Representations

Lecture Overview: Search 1 of 3

- Graph Search
- Applications
- Graph Representations
- Introduction to breadth-first and depth-first search

Readings

CLRS 22.1-22.3, B.4

Graph Search

Explore a graph e.g., find a path from start vertices 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 $\Longrightarrow$ *directed* edge of graph
  - unordered pair $\Longrightarrow$ *undirected*

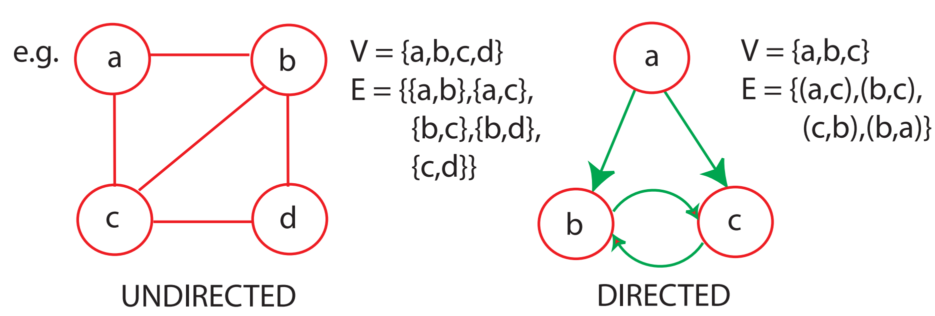


Figure 1: Example to illustrate graph terminology

Applications:

There are many.

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

Pocket Cube:

Consider a $2 \times 2 \times 2$ Rubik's cube

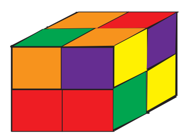


Figure 2: Rubik's Cube

- Configuration Graph:
  - vertex for each possible state
  - edge for each basic move (e.g., 90 degree turn) from one state to another
  - undirected: moves are reversible
- Puzzle: Given initial state s, find a path to the solved state
- $\sharp$ vertices $= 8!.3^8 = 264,539,520$ (because there are 8 cubelets in arbitrary positions, and each cubelet has 3 possible twists)

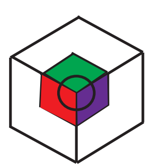


Figure 3: Illustration of Symmetry

- can factor out 24-fold symmetry of cube: fix one cubelet

$$8^{11}.3 \implies 7!.3^7 = 11,022,480$$

- in fact, graph has 3 connected components of equal size $\implies$ only need to search in one

$$\implies 7!.3^6 = 3,674,160$$

"Geography" of configuration graph

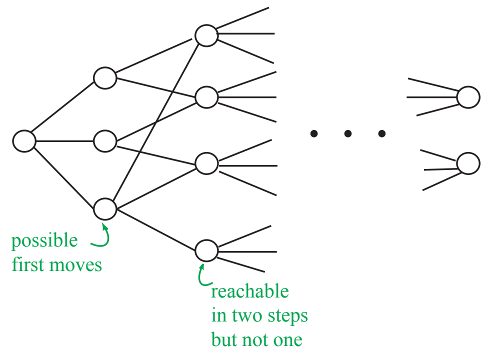


Figure 4: Breadth-First Tree

♯ reachable configurations

distance	90° turns	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
		Wikipedia Pocket Cube

Cf. $3 \times 3 \times 3$ Rubik's cube: ≈ 1.4 trillion states; diameter is unknown! ≤ 26

Representing Graphs: (data structures)

Adjacency lists:

Array *Adj* of $\mid V \mid$ linked lists

- for each vertex $u \epsilon V$, $Adj[u]$ stores u's neighbors, i.e., $\{v \epsilon V \mid (u, v) \epsilon E\}$. colorBlue$(u, v)$ are just outgoing edges if directed. (See Fig. 5 for an example)
- in Python: Adj = dictionary of list/set values vertex = any hashable object (e.g., int, tuple)
- advantage: multiple graphs on same vertices

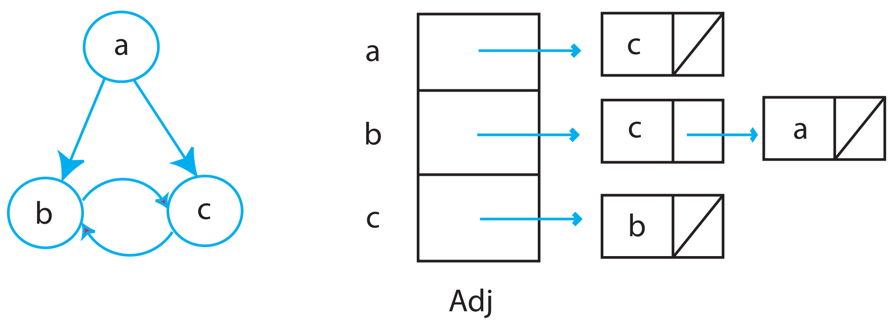


Figure 5: Adjacency List Representation

Object-oriented variations:

- object for each vertex u
- u.neighbors = list of neighbors i.e., $Adj[u]$

Incidence Lists:

- can also make edges objects (see Figure 6)
- u.edges = list of (outgoing) edges from u.
- advantage: storing data with vertices and edges without hashing

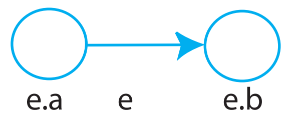


Figure 6: Edge Representation

Representing Graphs: contd.

The above representations are good for for sparse graphs where $| E | \ll (| V |)^2$. This translates to a space requirement $= \Theta(V + E)$ (Don't bother with $| . |$'s inside O/Θ).

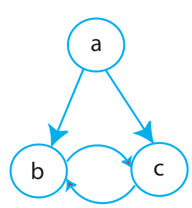
Adjacency Matrix:

- assume $V = \{1, 2, \ldots, |v|\}$ (number vertices)
- $A = (a_{ij}) = |V| \times |V|$ matrix where i = row and j = column, and

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \ \epsilon \text{ E} \\ \phi & \text{otherwise} \end{cases}$$

  See Figure 7.

- good for dense graphs where $| E | \approx (| V |)^2$
- space requirement $= \Theta(V^2)$
- cool properties like A^2 gives length-2 paths and Google PageRank $\approx A^\infty$
- but we'll rarely use it Google couldn't; $| V | \approx$ 20 billion $\implies (| V |)^2 \approx 4.10^{20}$ [50,000 petabytes]



$$A = \begin{array}{cc} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ & \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{array}{c} 1 \\ 2 \\ 3 \end{array} \end{array}$$

Figure 7: Matrix Representation

Implicit Graphs:

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

High level overview of next two lectures:

Breadth-first search

Levels like "geography"

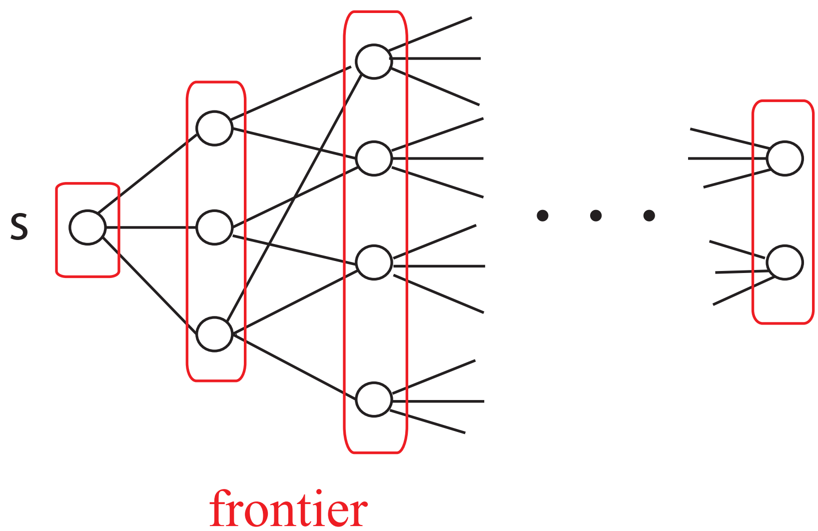


Figure 8: Illustrating Breadth-First Search

- 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

This is like exploring a maze.

- e.g.: (left-hand rule) - See Figure 9
- follow path until you get stuck
- backtrack along breadcrumbs until you reach an unexplored edge

- recursively explore it
- careful not to repeat a vertex

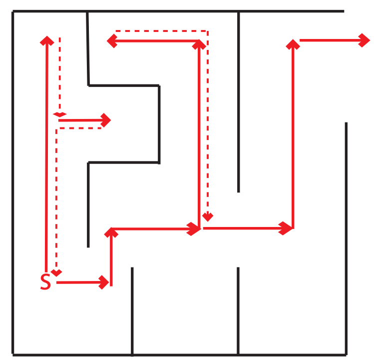


Figure 9: Illustrating Depth-First Search