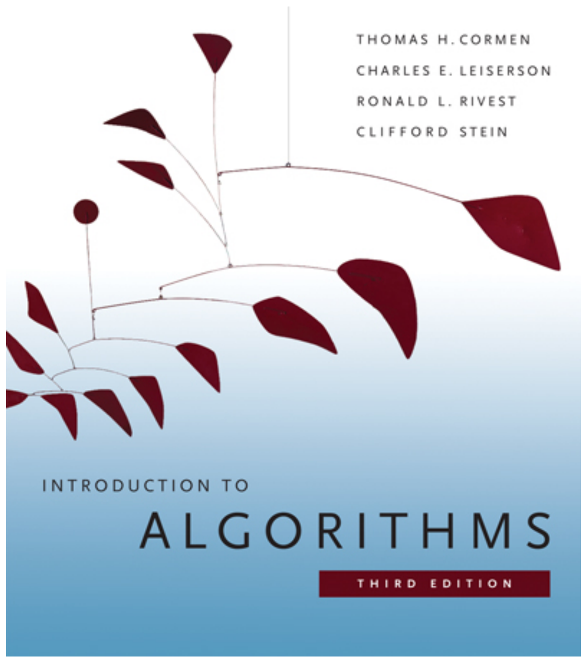
6.006- *Introduction to Algorithms*



Lecture 11

Prof. Patrick Jaillet

Lecture Overview

Searching I: Graph Search and Representations

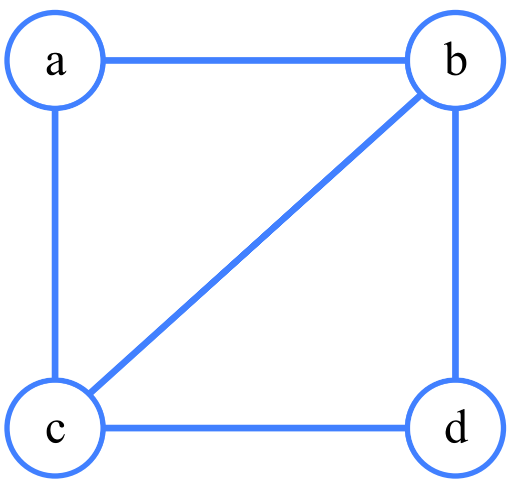
Readings: CLRS 22.1-22.3, B.4

Graphs

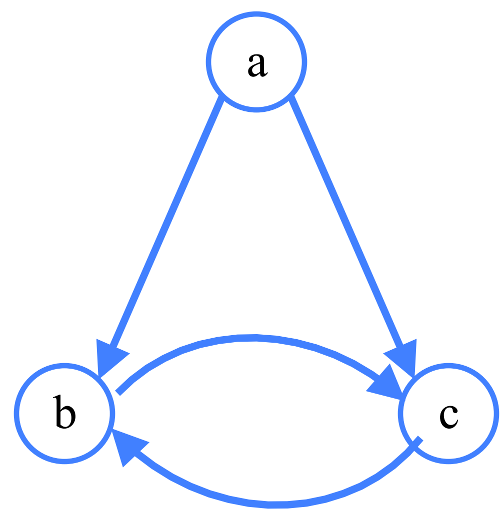
- G=(V,E)
- V a set of vertices
  - usually number denoted by n
- $E \subseteq V \times V$ a set of edges (pairs of vertices)
  - usually number denoted by m
  - note $m < n(n-1) = O(n^2)$
- Flavors:
  - pay attention to order: directed graph
  - ignore order: undirected graph
    - Then only $n(n-1)/2$ possible edges

Examples

- Undirected
- V={a,b,c,d}
- E={{a,b}, {a,c}, {b,c}, {b,d}, {c,d}}



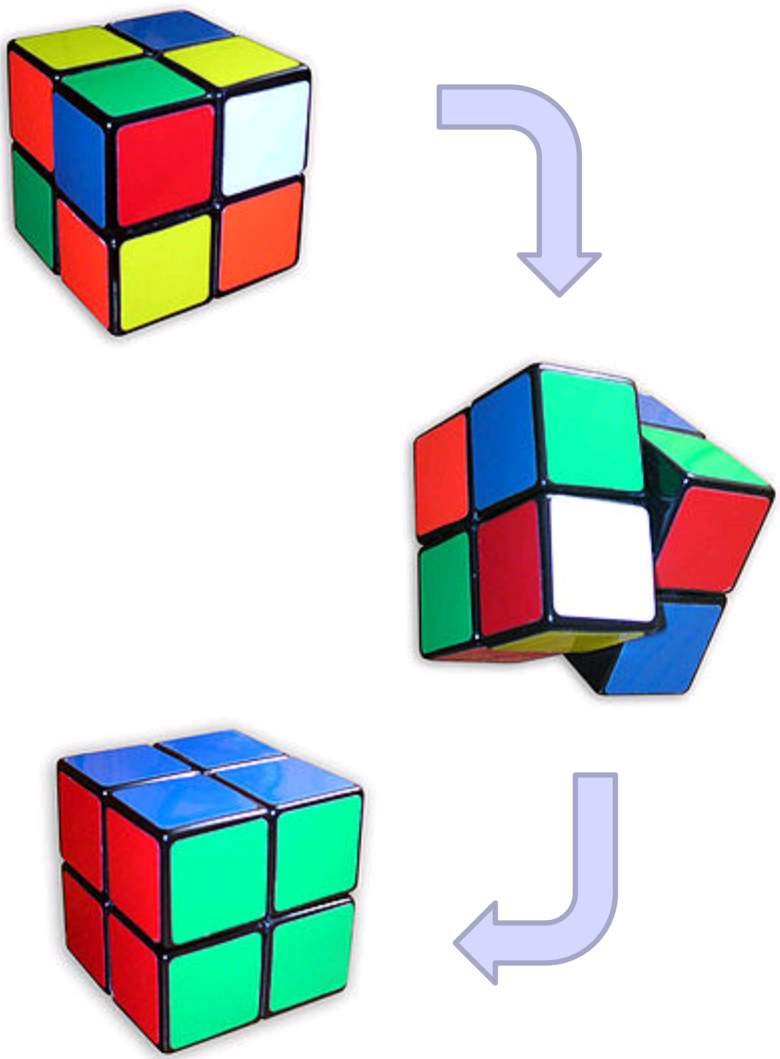
- Directed
- V = {a,b,c}
- E = {(a,c), (a,b) (b,c), (c,b)}



Instances/Applications

- Web
  - crawling
- Social Network
  - friend finder
- Computer Networks
  - internet routing
  - connectivity
- Game states
  - rubik’s cube, chess

Pocket Cube



- $2 \times 2 \times 2$ Rubik's cube
- Start with any colors
- Moves are quarter turns of any face
- "Solve" by making each side one color

Configuration Graph

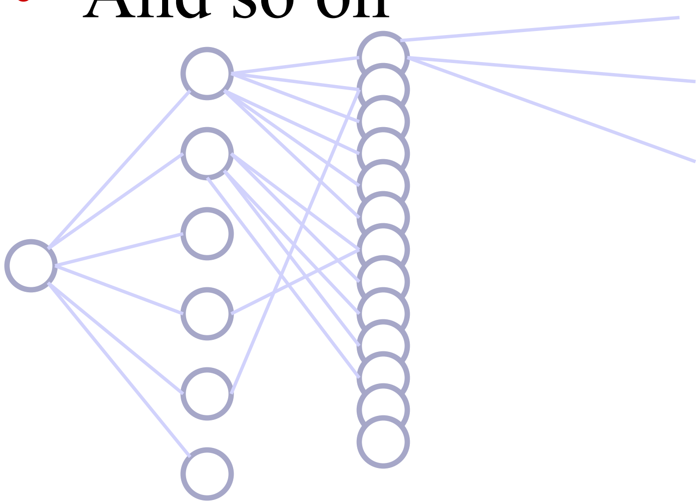
- One vertex for each state
- One edge for each move from a vertex
  - 6 faces to twist
  - 3 nontrivial ways to twist (1/4, 2/4, 3/4)
  - So, 18 edges out of each state
- Solve cube by finding a path (of moves) from initial state (vertex) to "solved" state

Combinatorics

- State for each arrangement and orientation of 8 cubelets
  - 8 cubelets in each position: 8! Possibilities
  - Each cube has 3 orientations: 3^8 Possibilities
  - Total: $8! \times 3^8 = 264{,}539{,}320$ vertices
- But divide out 24 orientations of whole cube
- And there are three separate connected components (twist one cube out of place 3 ways)
- Result: 3,674,160 states to search

GeoGRAPHy

- One start vertex
- 6 others reachable by one 90° turn
- From those, 27 others by another
- And so on



distance	90°	90° and 180°
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,526
8	114,149	870,072
9	360,508	1,887,748
10	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

diameter

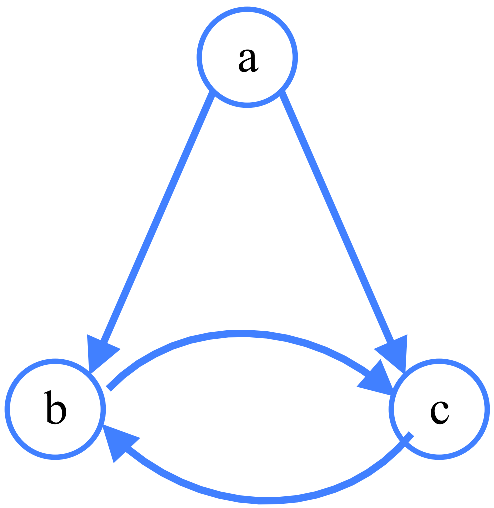
Representation

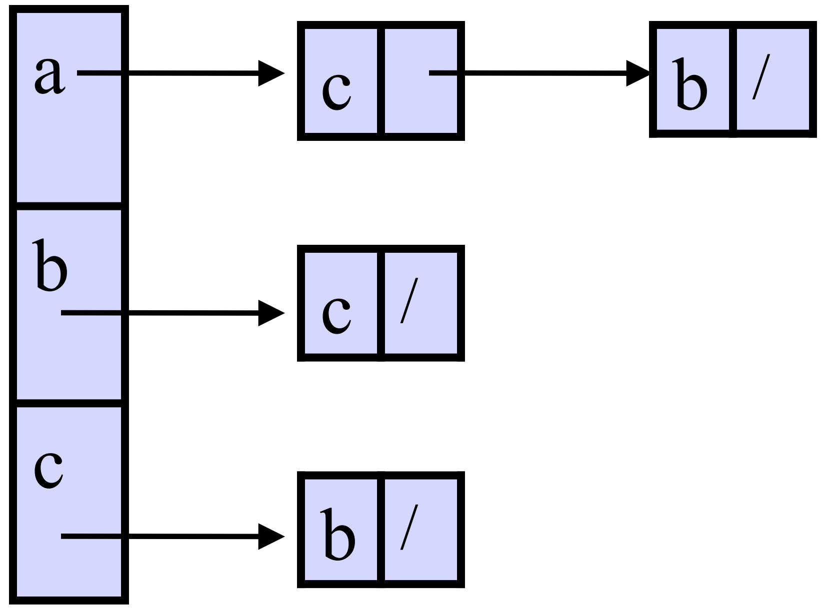
- To solve graph problems, must examine graph
- So need to represent in computer
- Four representations with pros/cons
  - Adjacency lists (of neighbors of each vertex)
  - Incidence lists (of edges from each vertex)
  - Adjacency matrix (of which pairs are adjacent)
  - Implicit representation (as neighbor function)

Adjacency List

- For each vertex v, list its neighbors (vertices to which it is connected by an edge)
  - Array A of |V| linked lists
  - For $v \in V$, list A[v] stores neighbors $\{u \mid (v,u) \in E\}$
  - Directed graph only stores outgoing neighbors
  - Undirected graph stores edge in two places
- In python, A[v] can be hash table
  - v any hashable object

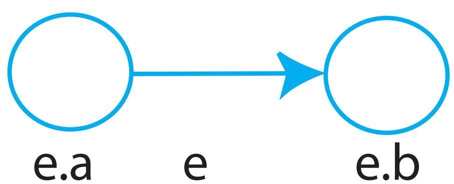
Example





(Object Oriented Variants)

- object for each vertex u
  - u.neighbors is list of neighbors for u
- incidence list: object for each edge e
  - u.edges = list of outgoing edges from u
  - e object has endpoints e.head and e.tail

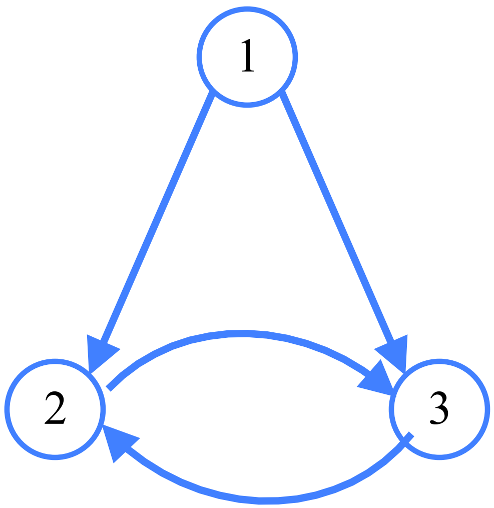


- can store additional info per vertex or edge without hashing

Adjacency Matrix

- assume $V=\{1, \ldots, n\}$
- matrix $A=(a_{ij})$ is $n \times n$
  - row i, column j
  - $a_{ij} = 1$ if $(i,j) \in E$
  - $a_{ij} = 0$ otherwise
- (store as, e.g., array of arrays)

Example



1	2	3	
0	1	1	1
0	0	1	2
0	1	0	3

Graph Algebra

- can treat adjacency matrix as matrix
- e.g., A^2 = length-2 paths between vertices ..
- [note: A^∞ gives pagerank of vertices..]
- undirected graph → symmetric matrix
- [eigenvalues useful for many things, but---
  rarely used in graph algorithms]

Tradeoff: Space

- Adjacency lists use one list node per edge
  - And two machine words per node
  - So space is $\Theta(mw)$ bits (m=#edges, w=word size)
- Adjacency matrix uses n^2 entries
  - But each entry can be just one bit
  - So $\Theta(n^2)$ bits
- Matrix better only for very dense graphs
  - m near n^2
  - (Google can't use matrix)

Tradeoff: Time

- Add edge
  - both data structures are O(1)
- Check “is there an edge from u to v”?
  - matrix is O(1)
  - adjacency list must be scanned
- Visit all neighbors of v (very common)
  - adjacency list is O(neighbors)
  - matrix is $\Theta(n)$
- Remove edge
  - like find + add

Implicit representation

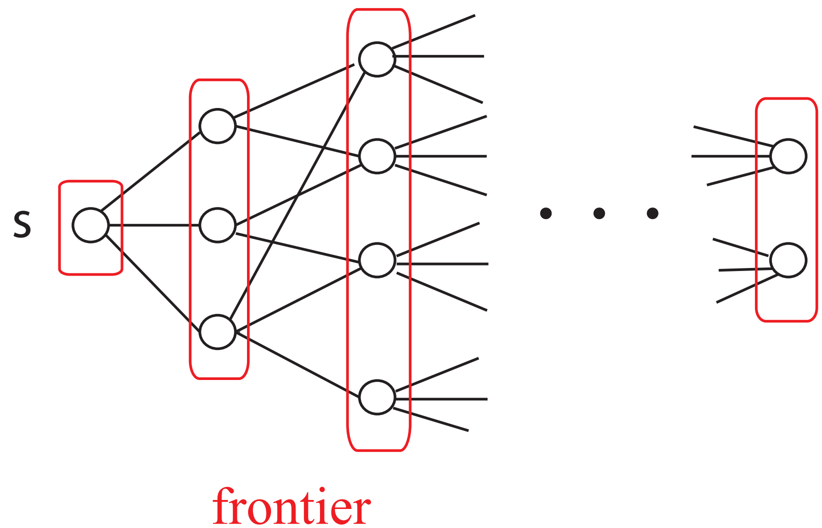
- Don't store graph at all
- Implement function Adj(u) that returns list of neighbors or edges of u
- Requires no space, use it as you need it
- And may be very efficient
- e.g., Rubik's cube

Searching Graph

- We want to get from current Rubik state to "solved" state
- How do we explore?

Breadth First Search

- start with vertex v
- list all its neighbors (distance 1)
- then all their neighbors (distance 2)
- etc.

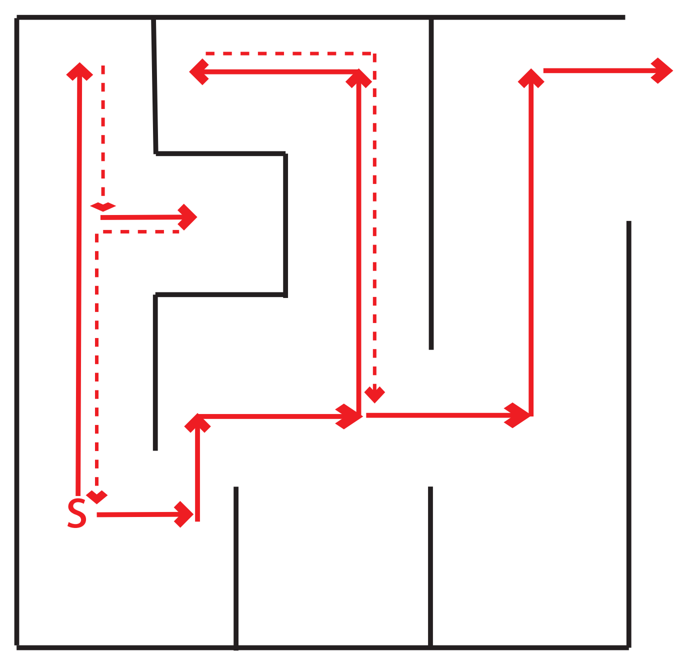


- algorithm starting at s:
  - define frontier F
  - initially F={s}
  - repeat F=all neighbors of vertices in F
  - until all vertices found

Depth First Search

- Like exploring a maze
- From current vertex, move to another
- Until you get stuck
- Then backtrack till you find a new place to explore

- e.g "left-hand" rule



Problem: Cycles

- What happens if unknowingly revisit a vertex?
- BFS: get wrong notion of distance
- DFS: go in circles
- Solution: mark vertices
  - BFS: if you've seen it before, ignore
  - DFS: if you've seen it before, back up

Conclude

- Graphs: fundamental data structure
  - Directed and undirected
- 4 possible representations
- Basic methods of graph search

- Next time:
  - Formalize BFS and DFS
  - Runtime analysis
  - Applications