Lecture 11: Graphs I: Breadth First Search

Lecture Overview

- A Quick Refresher
- Applications of Graph Search
- Graph Representations
- Breadth-First Search

Recall:

Graph $G = (V, E)$

- V = set of vertices (singular *vertex*; these may be arbitrarily labeled)
- E = set of edges, *i.e.*, vertex pairs (s, t). This is a subset of $V \times V$.
  - $|E| = O(V^2)$ (as in CLRS, we're dropping the |.|s inside the O-notation)
  - ordered pair $\implies$ directed edge of graph (directed graphs = digraphs)
  - unordered pair $\implies$ undirected

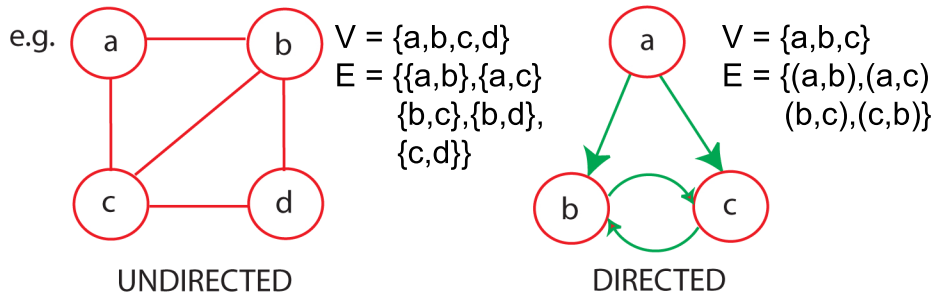


Figure 1: Example to illustrate graph terminology

Graph Search

"Explore a graph", e.g.:

- find a path from start vertex s to a desired vertex
- visit all vertices or edges of graph, or only those reachable from s

Applications:

There are many.

- web crawling (how Google finds pages)
- social networking (Facebook friend finder)
- mapping, flights
- garbage collection
- model checking (finite state machine – these are used in CS, economics, even nuclear security setups)
- solving puzzles and games
- tracking criminals (and other people) – High Country Bandits, NSA CO-TRAVELER

Graph Representations: (data structures)

Let's assume that the vertices V are labeled $\{1, \ldots, n\}$.

Adjacency Matrices:

Adjacency matrix A is a 2D matrix, $[1, \ldots, n, 1, \ldots, n]$.

$$A[i,j] = \begin{cases} 1 \text{ if } (i,j) \in E \\ 0 \text{ if } (i,j) \notin E \end{cases}$$

This is a "dense" representation, and takes $\theta(V^2)$ storage. Undirected edges will give us a symmetric matrix. However, adding and checking for an edge is very easy (takes constant time... and you can do all sorts of cool things by multiplying these matrices)! Visiting all the neighbors of a given edge takes $\theta(n)$ time.

Adjacency lists:

This is a sparse representation, and only takes $\theta(V + E)$ storage. This is basically the best you can hope for, since you need to store all the vertices and all the edges.

Array Adj of $|V|$ linked lists

- for each vertex $u \in V$, $Adj[u]$ stores u's neighbors, i.e., $\{v \in V \mid (u,v) \in E\}$. (u,v) are just outgoing edges if directed. (See Fig. 2 for an example.)
- $|Adj[u]| = degree(u)$ for undirected graphs; this is $outdegree(u)$ for directed graphs
- $\Sigma_{u \in V} |Adj[u]| = \Sigma_{u \in V} degree(u) = 2|E|$ for undirected graphs Handshaking Lemma

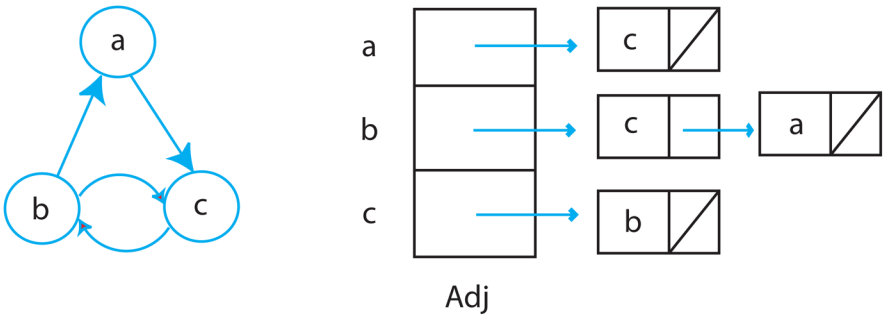


Figure 2: Adjacency List Representation: Space $\Theta(V + E)$

- in Python: Adj = dictionary of list/set values; vertex = any hashable object (e.g., int, tuple)
- Adding an edge takes constant time. Checking the existence of an edge takes O(longest list of neighbors), which is just the maximum degree, IF you're using linked lists – you might achieve better results if you used another hashtable here, etc. Visiting all neighbors is similarly bounded by the number of neighbors.
- advantage: multiple graphs on same vertices

Implicit Graphs:

Adj(u) is a function — compute local structure on the fly (e.g., Rubik's Cube). This requires "Zero" Space. Of course, if you need this information later, you might choose to remember it, but that's not necessary. (E.g., Subpoenaed phone numbers.)

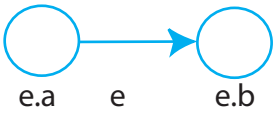
Object-oriented Variations:

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

In other words, this is 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

We want to search a graph, *e.g.*, to find a path from a vertex s to another vertex t, or to find all vertices reachable from s, or to count the hops (or minimum number of hops) needed to get from s to t.

BFS explored a graph level by level from s. At each level, we consider a "frontier" of exploration and visit everything one hop from that frontier until we have explored everything at that level. After each level, we consider these newly discovered vertices our new frontier and discard the old. This can be done in multiple ways and we discuss one of them below:

- level $0 = \{s\}$
- level i = vertices reachable by path of i edges but not fewer

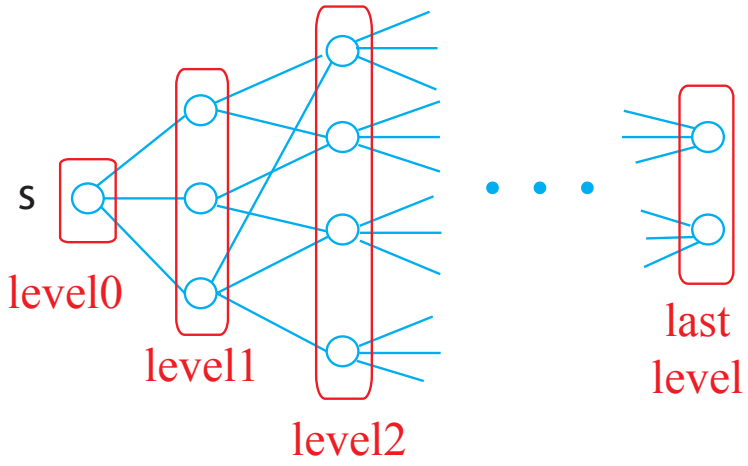


Figure 3: Illustrating Breadth-First Search

- build level $i > 0$ from level $i-1$ by trying all outgoing edges, but ignoring vertices from previous levels

Breadth-First-Search Algorithm

```
BFS (V,Adj,s):                                  See CLRS for queue-based implementation
    level = { s: 0 }
    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

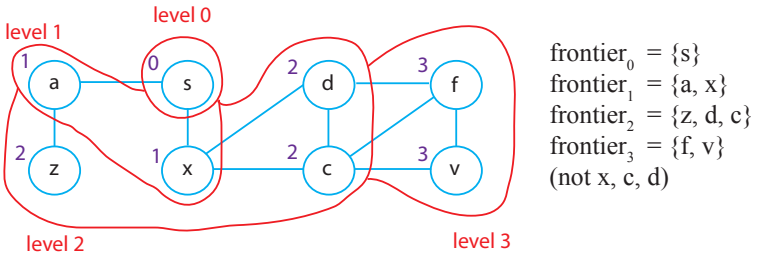


Figure 4: Breadth-First Search Frontier

Analysis:

- vertex V enters next (& then frontier) only once (because level[v] then set)

  base case: $v = s$

- $\implies$ Adj[v] looped through only once

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

- $\implies$ $O(E)$ time

- $O(V + E)$ ("LINEAR TIME") to also list vertices unreachable from v (those still not assigned level)
- Vertices unreachable from s can be considered to have level ∞.

Shortest Paths:

cf. L13-16

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] \text{ if } v \text{ assigned level} \\ \infty \quad \text{else (no path)} \end{cases}$$

- parent pointers form shortest-path tree = union of such a shortest path for each v
  $\implies$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)
- If a shorter path existed, then v would have been discovered at an earlier BFS level.