CS 163 & CS 265: Graph Algorithms

Week 1: Basics

Lecture 1b: Representation of graphs
Reachability and breadth-first search

David Eppstein
University of California, Irvine

Winter Quarter, 2024

What do we need to represent?

Pagerank algorithm from last time

Translated into pseudocode, and with a little optimization, we get:

```
def approximate_pagerank(G):
    n = #(vertices in G)
    P = { v :  1.0/n for v in G }
    repeat 5 times:
        Q = { v :  0.05/n for v in G }
        for each edge from v to w in G:
            Q[w] += 0.95 * P[v]/#(edges out of v)
        P = Q
    return P
```

Python-like syntax; indentation shows level of nesting

`{x:y for x in z}` makes a dictionary, *x*'s as keys, *y*'s as values

How much time does this take?

Measure in terms of two variables: $n = \#$ vertices, $m = \#$ edges

Typically assume $n - 1 \leq m \leq n(n-1)/2$

(true for connected graphs with no repeated edges)

Use O-notation

Precise mathematical definition of two-variable O-notation not easy
but works ok when we assume $n - 1 \leq m \leq n(n-1)/2$

In practice: just remove all constant factors and lower-order terms

For instance: $3m + 22n \log n + 10n + 42 \Rightarrow O(m + n \log n)$

Graph vs non-graph parts of pagerank algorithm

```
def approximate_pagerank(G):
    n = #(vertices in G)
    P = { v :  1.0/n for v in G }
    repeat 5 times:
        Q = { v :  0.05/n for v in G }
        for each edge from v to w in G:
            Q[w] += 0.95 * P[v]/#(edges out of v)
        P = Q
    return P
```

$5n$ dictionary operations to initialize P and Q; $5m$ arithmetic and dictionary operations in the inner loop, so the non-graph stuff takes time $O(n + m) \Rightarrow O(m)$

To finish the analysis we need to know the time for each red operation
We cannot do this until we understand how graphs are represented and implemented

What do we need to specify?

A graph representation should provide:

The set of operations that it can perform (API)

How to store the information within a computer (data structure)

How to perform the operations (algorithms)

The runtime of each operation (analysis)

Graph operations and desired runtimes

With only these operations, pagerank runs in time $O(m)$:

Count vertices: $O(1)$

Iterate through all vertices: $O(1)$ per vertex

Associate information $P[v]$ with each vertex v: $O(1)$ per get/set

Iterate through all edges: $O(1)$ per edge

Count edges into or out of a vertex ("degree"): $O(1)$

Additional operations, useful for other algorithms:

Count all edges: $O(1)$

Iterate through edges into or out of a vertex: $O(1)$ per edge

Associate information with edges: $O(1)$ per get/set

Test whether two vertices are connected by an edge: $O(1)$

Decorator pattern and adjacency lists

Decorator pattern

Idea: we need to be able to store and retrieve information associated with the vertices

Multiple possible solutions:

- Dictionary, with vertices as keys (Python style)
- Number vertices $0, \ldots n-1$ and index an array by these numbers
- Separate instance variable on vertex objects for each piece of information
- One dictionary instance variable per vertex with a key for each piece of information

Advantages of the first option: doesn't care how vertices are implemented, fewer bugs where different algorithms try to use the same decoration for different purposes

Adjacency list representations

Not really one representation, but a broad class of representations

Main idea: Decorate each vertex with a collection of its neighbors

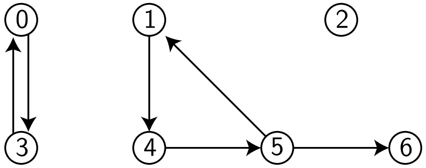
Variations:

- Python (`https://www.python.org/doc/essays/graphs/`):
  directed graph = dictionary with the vertices as its keys,
  and with values = lists of outgoing neighbors of each vertex
- Cormen et al, *Introduction to Algorithms*:
  vertices = numbers from 0 to $n - 1$;
  undirected graph = array of pointers to singly-linked lists of neighbors
- Goodrich & Tamassia, *Algorithm Design*:
  vertices and edges are objects; edges have endpoints as instance variables; vertices have collections of incoming and outgoing edges as instance variables

All can list vertices, edges, or neighbors in $O(1)$ time per item

Example of Python-style adjacency list

The directed graph:



can be represented in Python code as:

```
G = { 0: [3], 1: [4], 2: [ ],
      3: [0], 4: [5], 5: [1, 6], 6: [ ] }
```

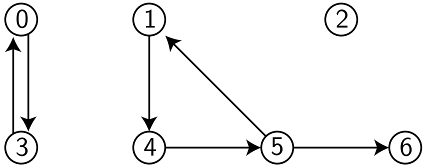
Operations with Python-style representation

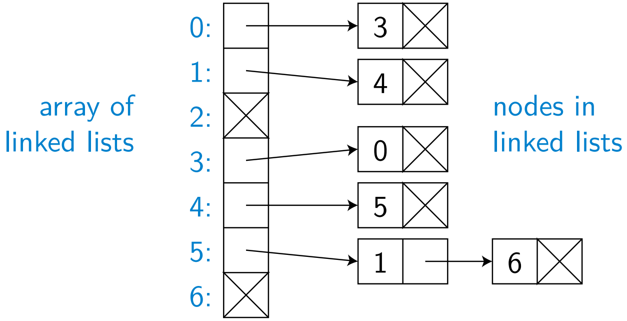
If `G` is a graph represented in this way, then:

- Number of vertices in G: $O(1)$ time
  ```
  len(G)
  ```
- Loop through all vertices: $O(1)$ time per vertex
  ```
  for v in G: ...
  ```
- Number of outgoing neighbors of v: $O(1)$ time
  ```
  len(G[v])
  ```
- Loop through edges out of given vertex v: $O(1)$ time per edge
  ```
  for w in G[v]: ...
  ```
- Test if the graph includes an edge $v \to w$:
  ```
  w in G[v]
  ```
  Slow if neighbors are stored as a list, $O(1)$ if stored as a set

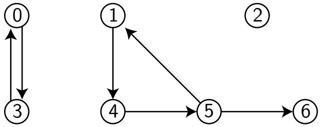
Other operations (looping through incoming edges) not directly supported, maybe slow

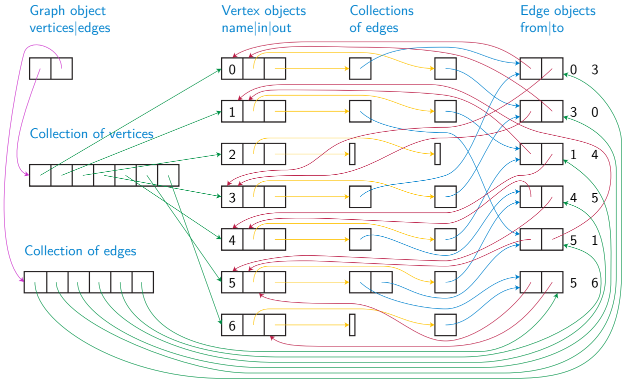
Same example, CLRS-style adjacency list





Same example, object-oriented adjacency list





Reachability and implicit graphs

Spiders

We've discussed how web search engines build indexes of web pages and sort the results of queries, but how did they find those pages in order to build an index of them?

Web crawler or spider

Algorithm that explores the web graph or any unknown graph finding all of the vertices that it can reach by following links from a given starting vertex

Example: find all web pages on the www.ics.uci.edu web site that can be reached from the starting page https://www.ics.uci.edu/

Pages that are not linked from other pages will not be found

Implicit graph representation

Web graph: We do not already have its vertices and edges in our computer (that's the problem we're trying to solve) but they still exist out on the internet

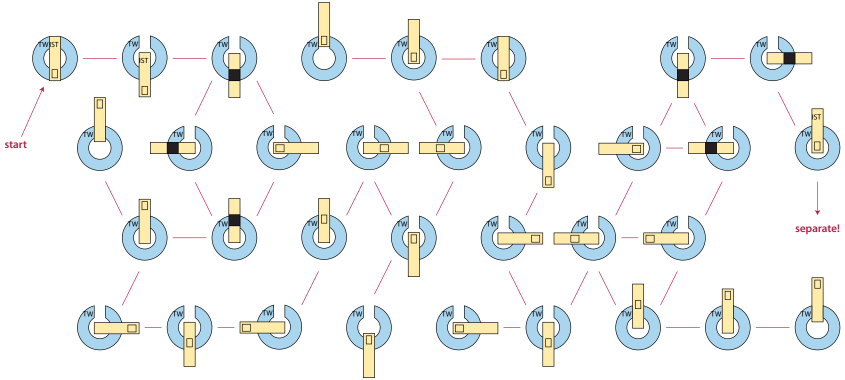
We can represent any vertex by its url, a string

Given any vertex we can find its outgoing edges by reading and parsing the html content at that url

More generally, an implicit graph is any graph that is not already represented in our computer, where we have some way of describing vertices and finding their edges

Another application of implicit graphs: Puzzles

Vertices = states of the puzzle, edges = moves



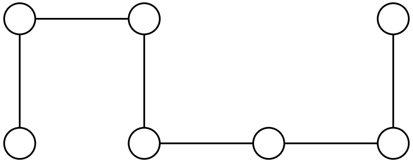
Some puzzles like Rubik's cube have many more states than we can store in computer

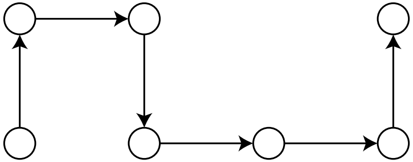
Real-world application in planning sequences of actions

Paths and walks

Path in an undirected graph: alternating sequence of vertices and edges, without repetitions, starting and ending at vertices, each edge between its two endpoints

In a directed graph: outgoing edge endpoint first, then edge, then incoming endpoint





Walk: allow repeated vertices and edges

Unfortunately some sources use different terminology: "simple path" for no repetitions, or "path" allowing repetitions

A single vertex by itself is a path (very short alternating sequence)

Formulation of the web spider problem

Given an implicit graph G and a starting vertex s

Find and return the set of all vertices that can be reached by paths from s

Algorithm doesn't have to know that the graph is implicit, but in the web spider application some graph operations are limited: for instance we cannot easily find incoming edges to vertices

Web spiders just need the reachable set
but some other applications (puzzles) might want the paths

Breadth-first search

Breadth-first search

A good choice for web spiders:

- Linear time, $O(m)$
- Explores vertices in order by # steps from start, spreading load among multiple servers
- In infinite graphs with finite # links per vertex (the web?) will still eventually reach every vertex

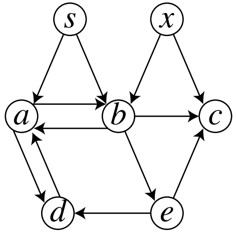
Intuitive idea:

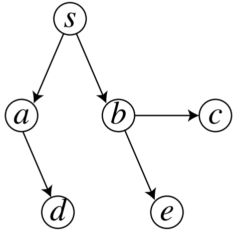
- Queue (first-in first-out) data structure of vertices to explore
- Explore vertices in queue order
- When we find an edge to an unreached vertex add it to the queue

Breadth-first reachability pseudocode

```
def reachable(G,s):
    // arguments are an implicit graph G
    // and the starting vertex s

    reached = new set containing s
    queue = new queue containing s

    while queue is not empty:
        remove first item from queue; call it v
        for each edge in G from v to w:
            if w is not already in reached:
                add w to both reached and queue

    return reached
```

Breadth-first reachability example





queue: s; reached: s

remove s
add unreached neighbors a,b
queue: a,b; reached: s,a,b

remove a
add unreached neighbor d
queue: b,d
reached: s, a, b, d

remove b
add unreached neighbors c, e
queue: d, c, e
reached: s, a, b, d, c, e

remove d, c, e
(no more unreached neighbors)

Tree in bottom shows which *v* was the first to find each vertex *w*, but it is also a tree in the graph, with shortest paths from *s* to other vertices

It is called the BFS tree

The algorithm can be modified to construct and return it

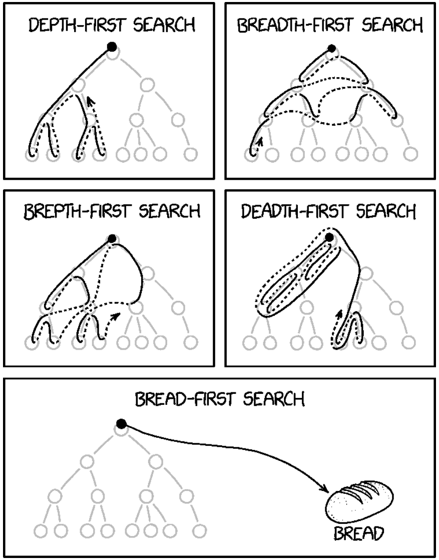
Analysis of breadth-first search

Each vertex is only added to the queue when not in reached, and at the same time is added to reached, so added once per vertex $\Rightarrow$ also removed once per vertex

Each edge is looped over only when we remove one of its endpoints from the queue $\Rightarrow$ once or twice per edge

Constant work for each vertex or edge that we loop over
$\Rightarrow O(m)$ time

Other exploration orders are also possible



https://xkcd.com/2407/

The morals of the story

You need to know about the representation when you're implementing graph algorithms

Adjacency lists allow most operations you want to perform in constant time per step

(Python version is simple and works well but is missing some important operations)

The web graph is already in a graph representation,
an implicit graph, even before we start exploring it

(So are the state spaces of puzzles and planning problems)

Breadth-first search can find short paths to reachable vertices in linear time

For the rest of this class we will mostly just assume an adjacency list representation
and not talk about the details any more