CS 163 & CS 265: Graph Algorithms

Week 1: Basics

Lecture 2: Representation of graphs

David Eppstein
University of California, Irvine

Fall Quarter, 2022

This work is licensed under a Creative Commons Attribution 4.0 International License

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 10 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 = number of vertices in the graph

m = number of edges in the graph

Use O-notation

Typically assume $n = O(m)$ (true if graph is connected, or even if it is disconnected but has no isolated vertices)
and $m = O(n^2)$ (true if graph has no repeated edges)

Formal definition of two-variable O-notation is a little tricky because different variables can have different limiting behavior, but the assumptions above make it work properly

In practice, we'll use notation like $O(m + n \log n)$ to mean that the total time is always at most some constant times $m + n \log n$

Formula inside O should be as simple as possible
(but should not grow more quickly than the runtime)

Analysis of non-graph parts of the algorithm

Set up a dictionary with n items: $O(n)$

Repeat 10 times: 10 is a constant, won't affect O-notation

Inside the outer 10x loop, set up another dictionary: $O(n)$

Inner loop happens once per edge in G; each step has a constant number of dictionary get/set and arithmetic operations: $O(m)$

Total: $O(m)$

Analysis of graph parts of the algorithm

Count number of vertices in G: ???

Find and loop over all vertices in G: ???

Find and loop over all edges in G: ???

Count number of edges out of a vertex w: ???

We can't complete the analysis without knowing how the graph is represented as a data structure inside the computer

Different choices of representation may have different times for these operations

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

In pagerank:

Count vertices: $O(1)$

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

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

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

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

Other standard operations:

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:

- Pagerank pseudocode used a dictionary, with vertices as keys
- If vertices are numbered $0, \ldots n-1$, information can be stored in an array indexed by these numbers
- In object-oriented style, with vertices as objects, information can be stored on instance variables
- Some textbooks suggest each vertex object has a special "decorator" instance variable, a dictionary with keys = names of decorations and values = the value for that decoration

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

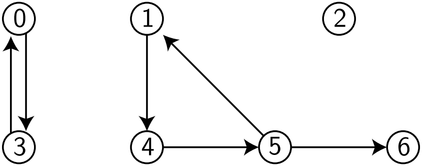
Variations:

- Python (`https://www.python.org/doc/essays/graphs/`):
  Directed graph = dictionary, keys = vertices, 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 instance variables pointing to endpoints; vertices have instance variables pointing to collections of incoming and outgoing edges

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]: ...
  ```

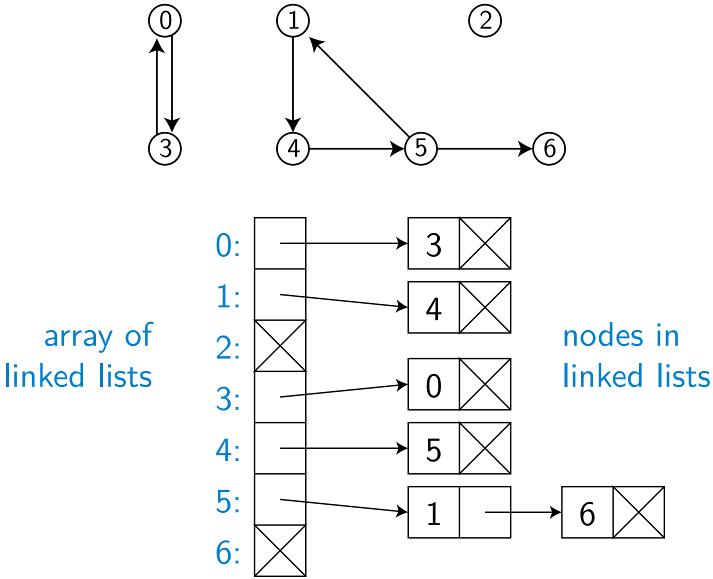
- Test if the graph includes an edge $v \rightarrow w$:

  ```
  w in G[v]
  ```

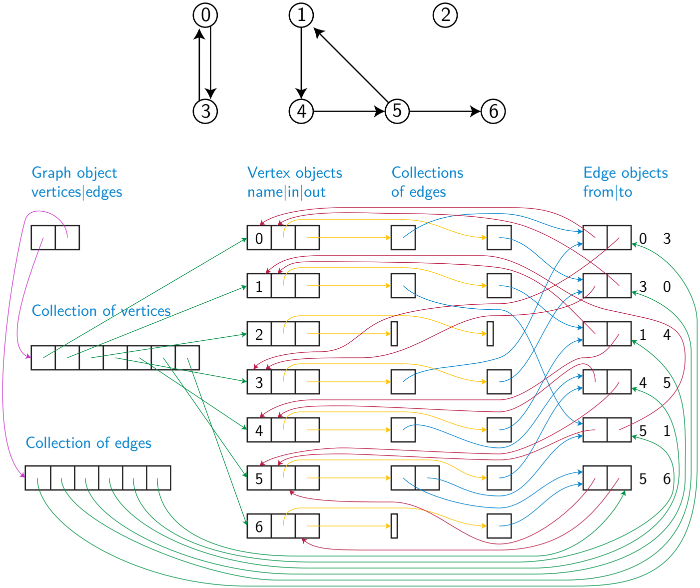
  Slow if neighbors are stored as a list, $O(1)$ if stored as a set

Other operations (like looping through incoming edges) are not directly supported and may be slow

Same example, CLRS-style adjacency list



Same example, object-oriented adjacency list

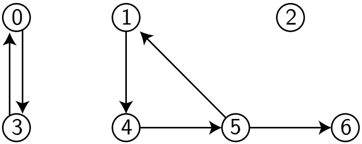


Adjacency matrix

Adjacency matrix representation

Vertices = integers

Two-dimensional array indexed by vertices, with
$A[v, w] = 1$ if there is an edge v to w, zero otherwise



$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The first index is the row, the second is the column
Rows are numbered top-down, starting with row 0
Columns are numbered left-right, starting with column 0

Why adjacency matrices?

In algorithms based on linear algebra

E.g. approximate pagerank = diagonal coefficients of M^{10} for a matrix M derived from the adjacency matrix

For very small graphs, can be a space-efficient way of packing into $n(n-1)/2$ bits (but for large graphs, $O(n^2)$ space is too much)

Allow fast test of edge existence (but so do hash tables of edges)

Usually we will just use adjacency lists

The morals of the story

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

The Python version works well for many purposes but is missing some important operations (especially: looping over incoming edges to a vertex); if needed it can be augmented with extra information (e.g. a second dictionary of incoming neighbors)

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

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