CS 420: Advanced Algorithm Design and Analysis
Spring 2015 – Lecture 10

Department of Computer Science
University of British Columbia

February 05, 2015

Announcements

Assignments...

- Solutions to Asst 2 and 3 have been posted
- Asst3...back today
- Asst4...out (due next Thursday)

Announcements

Readings...

- review material on graph representations and basic graph algorithms
- minimum-cost path problems [Erickson, Chapt 21, 22; Cormen+, Chapt 25 26; ...]

Last class...

Exploiting non-uniform access patterns

- unknown/changing access probabilities
  ... adaptive (self-organizing) tree-structured dictionaries
  - restructuring primitive...tree rotations
  - restructuring strategies...
    - rotate-to-root is *not* c-competitive
  - splay steps (zig-zag and zig-zig)
  - splay-to-root strategy
    - comparison with rotate-to-root
    - intuition: access path compression
  - amortized analysis of splaying
    - review of amortization using potential functions
    - weight-based potential assignment: weights, authority, rank and potential
    - Access Theorem
    - —- outline of proof, using Access Lemma
    - —- important Corollaries
    - —- dynamic optimality conjecture

Looking ahead...

After the midterm...

- on to graphs, and graph algorithms

Today...

Graph algorithms

- review of basic graph notation/terminology
- review of basic graph representations
- review of basic graph properties
  - paths and connectivity
- review of basic graph algorithms
  - connectivity
    - breadth-first and depth-first search (adjacency lists)
    - testing connectivity using an adjacency matrix
    - connectivity in semi-dynamic graphs

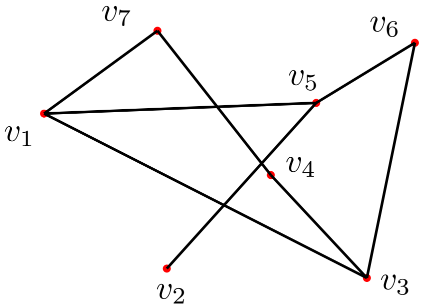
Review of basic graph notation and terminology

A *graph* G is a pair (V, E) where:

- V is a set of *vertices*
- $E \subseteq V \times V$ is a set of *edges*
  - frequently denote $|V|$ by n and $|E|$ by m
  - if the relation E is *symmetric* ($(u, v) \in E$ iff $(v, u) \in E$) then the graph G is *undirected*. Edges in an undirected graph are simply *un-ordered* pairs of vertices. (otherwise *directed*)
  - if E has an associated *weight/cost* function c then G is *(edge) weighted*
  - if E is a multi-set, then G is called a *multi-graph*
  - if E is an arbitrary subset of 2^V, then G is called a *hypergraph*

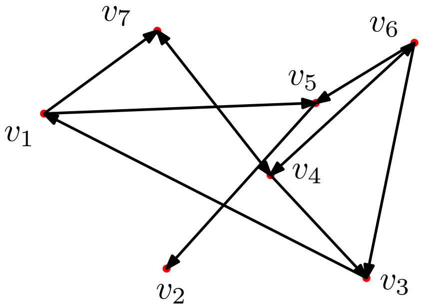
Review of basic graph representations

A *graphical* representation of G consists of n points (depicting vertices) and m arcs/arrows (depicting edges).

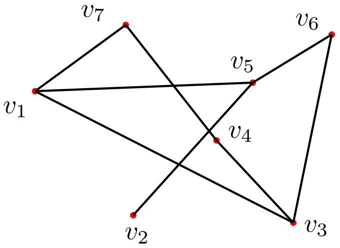


Review of basic graph representations

A *graphical* representation of G consists of n points (depicting vertices) and m arcs/arrows (depicting edges).



Graphical representation



Raises issues:

- visualization of large graphs
- optimization of esthetic considerations; edge crossings, symmetries, clusters, etc.

Review of basic graph representations

An *adjacency matrix* representation of G is the $n \times n$ Boolean ($\{0, 1\}$-valued) array A

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$$

where $a_{i,j} = 1$ if and only if there is an edge from vertex v_i to vertex v_j. Hmmm...reminds me of a direct access table....

- fast check for presence of a specified edge
- space requirement? $\Theta(n^2)$ bits?
- initialization?

Review of basic graph representations

An *adjacency list* representation of G consists of n lists $\text{Adj}[v_1], \ldots, \text{Adj}[v_n]$, one for each vertex in $v_i \in V$, specifying the vertices v_j for which $(v_i, v_j) \in E$.

- requires space proportional to $n + m$
- requires more work to check for presence of a specified edge...
- but makes it easy to check for the *next out-going edge* (a critical step in many algorithms)

Review of basic graph representations

An *embedded graph* is a graph together with an assignment of locations, within some embedding space, to all vertices.
An embedded graph is a *planar map* if it has no edge crossings. A graph that can be represented as a planar map is said to be *planar*.

Review of basic graph properties

A *path* in a graph G is a sequence of vertices $u_1, u_2, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$, for $1 \leq i < k$.

- path goes from u_1 to u_k (not necessarily symmetric in a digraph)
- path is *simple* if no vertex is repeated
- path is a *cycle* if $u_1 = u_k$

A graph G is *connected* if there is a path from u to v, for every pair of vertices u, v.

- the property is called *strongly connected* in digraphs
- the equivalence classes of the relation "is joined by a path" are called the *connected components* of G

Review of basic graph algorithms

Testing *connectivity* of a graph G or, more generally, finding the connected components of G, can be done in $O(n+m)$ time by two different techniques, both of which use an adjacency list representation of G (see Kleinberg/Tardos Chapter 3):

- breadth-first search
  - queue-directed (FIFO); explore uniformly expanding *wavefront*
- depth-first search
  - stack-directed (LIFO)
  - also identifies strongly connected components in $O(n+m)$ time

Review of basic graph algorithms

How hard is it to test if a given graph G is connected (or just if there is a path from vertex v_i to vertex v_j), when the graph is represented as an adjacency matrix A?

Claim: $\Omega(n^2)$ (a fixed fraction) of the entries of A must be probed, in the worst case, in order to determine if G is connected.

- proof follows from an *adversary strategy*
- the same claim holds for most natural (non-trivial) graph properties
- $\Omega(n^{5/4})$ probes are required on a randomized model

Review of basic graph algorithms

How hard is it to test if a given graph G is connected (or just if there is a path from vertex v_i to vertex v_j), when the graph is represented as an adjacency matrix A?

On the other hand (as we shall see)

- with sufficient preprocessing A can be converted into a matrix A^* (the *transitive closure* of A) that captures the relation "is connected by a path".
- using A^* path existence queries can be answered in $O(1)$ time

Testing connectivity in (semi)-dynamic graphs

What happens if the graph is changing over time?

- A graph G is *semi-dynamic* if edges are added to (but not deleted from) G over time
  - recall Kruskal's minimum spanning tree algorithm
- the connected components of G are naturally maintained by a *disjoint set* data structure:
  - maintain each connected component as a set (of vertices)
  - operations include MAKE-SET, FIND-SET, and UNION

Strategies for disjoint-set maintenance

Two simple strategies suggest themselves:

- associate an explicit component number with each vertex, and a set of vertices with each component number
  - FIND operation takes $O(1)$ time
  - UNION operation takes $O(\lg n)$ amortized time (weighted union)
- maintain each component as a tree and store the component number at the root
  - UNION operation links the smaller tree to the larger tree (at the root): $O(1)$ time
  - FIND operation involves walking to the tree root: $O(\lg n)$ time in the worst case, since the height of a tree with k elements never exceeds $\lg k$

Strategies for disjoint-set maintenance

A (slightly)more involved, more efficient and more interesting strategy involves *path compression*:

- maintain each component as a tree and store the component number at the root
  - UNION operation links the smaller tree to the larger tree (at the root): $O(1)$ time
  - FIND operation involves walking to the tree root
    - this is followed by *path compression*, which makes every node on the access path an immediate child of the root
    - the amortized cost of FIND is reduced to $O(\alpha(n))$, where α is an *extremely* slow growing function (constant, for all practical purposes)