CS 420: Advanced Algorithm Design and Analysis
Spring 2015 – Lecture 11

Department of Computer Science
University of British Columbia

February 10, 2015

Announcements

Assignments...

- Asst4...due Thursday

Readings...

- review material on graph representations and basic graph algorithms
- disjoint set maintenance (UNION-FIND) [Erickson, Chapt. 17; Cormen+, Chapt. 21]
- minimum-cost path problems [Erickson, Chapt 21, 22; Cormen+, Chapt 25 26; ...]

Last class...

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
    - ++++ UNION-FIND data structures

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
- 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)

How slow-growing is $\alpha(n)$?

Suppose that $f_0(i,j) = i + j$ and $f_k(i,j)$, $k > 0$, is defined by the procedure:

```
t ← i
for r ← 1 to j − 1 do
    t ← f_{k−1}(i, t)
end for
```

What function is computed by $f_1(i,j)$? $f_2(i,j)$? $f_3(i,j)$? $f_4(i,j)$?

$\alpha(n)$ is the inverse of Ackerman's function, which grows *faster* than $f_k(i,j)$ for *any* fixed k!

Path Optimization

Let G be an (edge) weighted directed graph, where $c(e)$ (*not necessarily positive*) denotes the weight/cost of edge e. The path $P = \langle v_0, v_1, \ldots, v_k \rangle$ has *length* k and *weight/cost*:

$$c(P) = \sum_{1 \leq i \leq k} c(v_{i-1}, v_i)$$

We denote by $\delta(u, v)$ the cost of the minimum cost path from u to v:

$$\min\{c(P) \mid P \text{ is a path from } u \text{ to } v\}$$

By convention we say that $\delta(u, v) = \infty$ if there is no path from u to v in G.

Properties of minimum cost paths

Algorithms for min cost paths exploit the following properties:

- *existence:* provided G has no negative cost cycle
- *acyclic:* removing a cycle reduces total cost
  ... so min cost paths have at most $n - 1$ edges
- *optimal substructure:* min cost paths are built out of min cost (sub)paths
- *tree representation* min cost paths from a single source s form a *tree* rooted at s.
- *local optimality:*

$$\delta(u, w) = \min_{v \in A^{-1}[w]} \{\delta(u, v) + c(v, w)\}$$

Properties of minimum cost paths

The local optimality property is the basis for the *incremental improvement* of min cost path estimates:

Let $d[u, v]$ be an upper bound on $\delta(u, v)$ (e.g. $d[u, v] = c(u, v)$ and $d[u, u] = 0$).

Then, if $d[u, w] > d[u, v] + c(v, w)$ we can improve the estimate $d[u, w]$ by replacing it with $d[u, v] + c(v, w)$.

This is called a *relaxation* on the edge (v, w).

Properties of edge relaxation

Edge relaxation is fundamental operation in min cost path algorithms. It has the following critical properties:

- *completeness:* When no further improvement by relaxation is possible then $d[u, w] = \delta(u, w)$
  ... (i.e. local optimality implies global optimality)
- *finiteness*
- *restriction*

Algorithms for single-source min-cost paths

Problem: Given G and $s \in V$, determine $\delta(s, v)$, for all $v \in V$.

A. [Bellman-Ford algorithm:]

1. initialize $d[s, v] = c(s, v)$
2. perform rounds of *global relaxation* (relax every edge $(u, v) \in E$)

$$d[s, v] \leftarrow \min\{d[s, v], d[s, u] + c(u, v)\}$$

3. stop when no further improvement by relaxation is possible

Analysis:

- Invariant: after r rounds, $d[s, v] = \delta(s, v)$, for all v whose min-cost path from s has at most r edges
- there are at most $n - 1$ rounds, so the total cost is $O(nm)$
- algorithm works even if there are negative cost edges (but no negative cost cycles)

Algorithms for single-source min-cost paths

Problem: Given G and $s \in V$, determine $\delta(s, v)$, for all $v \in V$.

B. [Dijkstra's algorithm:]

- assumes all edge costs are non-negative
- grows the min-cost path tree rooted at s incrementally (by a *greedy approach*)

$S \leftarrow \{s\}$
$d_S[s, v] =$ min-cost path from s to v, with intermediate vertices in S
while $V - S \neq \emptyset$ **do**
    add v to S if it minimizes $d_S[s, v]$ among all $v \in V - S$
    update d_S (by relaxation on edges out of v)
end while

Maintain d_S values for $v \in V - S$ in a *priority queue*

Complexity of Dijkstra's algorithm

Dijkstra's algorithm uses $n-1$ EXTRACT-MIN operations (line 4) and a total of at most m DECREASE-KEY operations on the underlying heap. Various heap implementations give more-or-less efficient implementations:

Structure	EXTRACT-MIN	DECREASE-KEY	TotalCost
naive heap	$\Theta(n)$	$\Theta(1)$	$\Theta(n^2)$
binary heap	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(m \lg n)$
Fibonacci heap	$\Theta(\lg n)^*$	$\Theta(1)^*$	$\Theta(n \lg n + m)$

(*) denotes *amortized* cost

Bellman-Ford vs. Dijkstra

Note: Both Bellman-Ford and Dijkstra are easily modified to permit min-cost path recovery in the same time. How?

Dijkstra's algorithm has advantages and disadvantages:

- faster: $\Theta(n \lg n + m)$ instead of $\Theta(nm)$
- less general: assumes no negative weight edges
- more *centralized*: less suitable for parallel/distributed implementation

All-pairs min-cost paths

The most obvious approach is to compute shortest paths from all possible sources, using repeated Bellman-Ford (in case of negative weights) or Dijkstra (in case of only non-negative weights):

- cost is $O(n \cdot (nm))$ for repeated Bellman-Ford
- cost is $O(n \cdot (n \lg n + m))$ for repeated Dijkstra

All-pairs min-cost paths

A less obvious approach, particularly suitable for sparse graphs, uses the idea of *reweighting* the edges of G, so that edges become non-negative and Dijkstra's algorithm can be applied.
How can we do it?

- add a suitable constant to each edge?
  No...it alters costs in proportion to path *length*
- need something that treats all paths with same endpoints *equitably*

Coming up...

Min-cost path problems

- all-pairs of endpoints
  - Johnson's algorithm, using *edge re-weighting*
  - algorithms for dense graphs, using dynamic programming
    - applications in string matching