6.889 — Lecture 4: Single-Source Shortest Paths

Christian Sommer csom@mit.edu

September 19 and 26, 2011

Single-Source Shortest Path (SSSP) Problem: given a graph $G = (V, E)$ and a *source* vertex $s \in V$, compute shortest-path distance $d_G(s, v)$ for each $v \in V$ (and encode shortest-path tree)

Assumption (all of Lecture 4) non-negative edge lengths $\ell : E \to \mathbb{R}^+$

General Graphs fastest: Dijkstra's algorithm $\mathcal{O}(m + n \log n)$

Planar Graphs? can we use r–division? recall that r–division of G is decomposition into

- $\mathcal{O}(n/r)$ edge-disjoint *pieces*,
- each with $\leqslant r$ vertices and
- $\mathcal{O}(\sqrt{r})$ *boundary* vertices. $\Leftarrow$ vertices with edges to at least two pieces

assume (temporarily) we can compute r–division in $\mathcal{O}(n)$.

(Simple) Algorithm note that Dijkstra running time not "balanced," nodes are bottleneck. idea: *densify*

- r–division with $r := \log n / \log \log n$ (wlog assume s is on boundary)
- FOR EACH piece P, FOR EACH boundary node $p \in \partial P$
  compute SSSP (Dijkstra) in P, store in $\partial P \times \partial P$ distance matrix (complete graph on ∂P)
  time required:

$$\mathcal{O}\left(\frac{n \log\log n}{\log n}\sqrt{\frac{\log n}{\log\log n}}\right) \cdot \mathcal{O}\left(\frac{\log n}{\log\log n}\log\left(\frac{\log n}{\log\log n}\right)\right) = \mathcal{O}\left(n\sqrt{\log n \log\log n}\right)$$

  let G' denote graph with each piece P replaced by complete graph on ∂P

- compute SSSP (Dijkstra) in G'
  time required: $n' := |V(G')| = \mathcal{O}(n\sqrt{\log\log n/\log n})$ and $m' := |E(G')| = \mathcal{O}(n)$, therefore

$$\mathcal{O}(n' \log n' + m') = \mathcal{O}\left(n\sqrt{\log n \log\log n}\right)$$

- FOR EACH piece P, FOR EACH boundary node $p \in \partial P$
  compute SSSP in P starting with $\mathbf{d}[p] := d_G(s, p)$
  (which computes, for every $p' \in P$, the distance $d_G(s, p')$ and the *last* boundary node $p \in \partial P$ on the shortest path from s to p')

note that output *not* necessarily in sorted order
let's try recursion? issues: $\sqrt{r}$ instances of SSSP per piece, G' is *non-planar* graph

1 Fast r–division

need r–division in many algorithms. first step of SSSP algorithm. want to improve upon $\mathcal{O}(n \log n)$ for SSSP $\rightsquigarrow$ need fast r–division

Lemma. *For planar G, we can compute an r–division in time $\mathcal{O}(n \log r)$.*

improvement over $\mathcal{O}(n \log n)$

Idea do first $\mathcal{O}(\log n)$ recursion levels on smaller graph, then $\mathcal{O}(\log r)$ levels on original graph

Def. *A ρ–*clustering *of G is a decomposition into*

- $\mathcal{O}(n/\rho)$ *vertex-disjoint* connected pieces*,*
- *each with $\Theta(\rho)$ vertices.*

Lemma. *A ρ–clustering can be found in linear time (assuming G has bounded degree).*

Proof. Problem Set. Hint: DFS tree □

Algorithm

- ρ–clustering for $\rho = \sqrt{r}$
- contract each piece into single node, make graph simple $\rightsquigarrow$ G' planar graph on $n' = \mathcal{O}(n/\sqrt{r})$ vertices
- $\mathcal{O}(n' \log n')$ algorithm for r–division on G' in time $\mathcal{O}((n/\sqrt{r}) \log n)$
- expand piece back $\rightsquigarrow$ new pieces have size between r and $r^{3/2}$
- another $\mathcal{O}(\log r)$ levels of recursion

From Small Total Boundary to Small Piece Boundary

- WHILE $\exists$ piece P with boundary $|\partial P| > c\sqrt{r}$ for some constant c
  let $n' = |\partial P|$. apply separator theorem to $G(P)$ with weight $1/n'$ on each $p \in \partial P$, weight 0 otherwise
  $\rightsquigarrow$ splits boundary

in this process: how many new boundary nodes? how many new pieces? $\rightsquigarrow$ see book for details

2 SSSP Algorithm

queue operations are expensive part. work on smaller queues

Idea like Dijkstra SSSP algorithm with *limited attention span*. recursive r–division. spend *some* time on the region with the current minimum. distances not necessarily correct while in region, *speculative* work! but queue operations are cheap!

One Level

- r–division with $r = \log^4 n$
- REPEAT
  - select region R with current minimum
  - run Dijkstra on R for $\alpha = \log n$ steps (if possible, otherwise *truncated*)

Implementation consists of three procedures (plus implementation of priority queue)

- SSSP(G, s)
  1. recursive r–division $\rightsquigarrow R(G), R(P_i), \dots R(e)$
  2. allocate queue Q per piece
  3. initialize: $\forall v : \mathbf{d}[v] := \infty$
  4. $\mathbf{d}[s] := 0$
  5. FOR EACH edge $sv \in E(G)$
  6.   GLOBALUPDATE$(R(sv), sv, 0)$
  7. WHILE MINKEY$(Q(R(G))) < \infty$
  8.   PROCESS$(R(G))$

- GLOBALUPDATE(region R, item x, value k) recursive update value of x (if needed), queue consistency
  1. UPDATEKEY(queue $Q(R)$, item x, value k)
  2. IF x is new MINKEY$(Q(R))$
  3.   GLOBALUPDATE(PARENT$(R), R, k)$

- PROCESS(R) (for simplified algorithm, let $\alpha_2 = 1, \alpha_1 = \log n, \alpha_0 = 0$)
  1. IF R contains single edge uv
  2.   IF $\mathbf{d}[v] > \mathbf{d}[u] + \ell(uv)$
  3. $\mathbf{d}[v] := \mathbf{d}[u] + \ell(uv)$
  4.   FOR EACH $vw \in E$: GLOBALUPDATE$(R(vw), vw, \mathbf{d}[v])$ $\rightsquigarrow$ note: key of *arc* vw is $\mathbf{d}[v]$
  5.   UPDATEKEY$(Q(R), uv, \infty)$
  6. ELSE
  7.   FOR $\alpha_{h(R)}$ times AND WHILE MINKEY$(Q(R)) < \infty$: (α_i defined for recursion height of piece R)
  8. $R' :=$EXTRACTMIN$(Q(R))$
  9.   PROCESS(R')
  10.   UPDATEKEY$(Q(R), R'$,MINKEY$(Q(R')))$

3 Correctness

three properties imply correctness

- *[initialized]* $\mathbf{d}[s] = 0$
- *[path-length property]* for each v, $\mathbf{d}[v] \geqslant d_G(s, v)$ (at least length of *some* path)
- *[edges relaxed]* for each $vw \in E$: $\mathbf{d}[w] \leqslant \mathbf{d}[v] + \ell(vw)$

first two properties are easy. all edges relaxed?

- say edge uv is *active* iff key of uv in $Q(R(uv))$ is finite. inactive otherwise.
- uv inactive $\Rightarrow$ relaxed (except at step 4)
  - easy at beginning ($\forall v : \mathbf{d}[v] = \infty$).
  - arc can be *tense* (not relaxed) when labels $\mathbf{d}[\cdot]$ of endpoints change. labels never increase $\rightsquigarrow$ ok to look at $\mathbf{d}[v]$: might decrease but immediately after we activate edges!

  uv active $\Rightarrow$ key of uv is $\mathbf{d}[v]$ (except at step 4)
- queue consistency: for region R, key associated with R in parent queue $Q($PARENT$(R))$ is MINKEY$(Q(R))$ (except for current region and its ancestors)
  corollary: MINKEY$(Q(R))$ value is $\min$ among all active arcs in R
  $\rightsquigarrow$ if MINKEY$(Q(R(G))) = \infty$, no arc in G is active, all inactive, all relaxed

4 Running Time [Lec. 4-b]

Main Idea *invocation* of procedure PROCESS. count *truncated* invocations. charge them to boundary nodes. not too many boundary nodes $\rightsquigarrow$ not too many truncated invocations

Non-truncated Invocations assume we can bound total number of invocations on lowest level (level-0 regions contain only one edge) by $\mathcal{O}(n) \rightsquigarrow$ each non-truncated invocation on level 1 causes $\log n$ invocations on level 0 $\rightsquigarrow$ at most $\mathcal{O}(n/\log n)$ non-truncated invocations. enough descendants "pay" for queue operation

Charging Scheme Invariant for any pair (R, v) of region R and boundary node $v \in \partial R$, exists invocation B of PROCESS s.t. all invocations charging to (R, v) are descendants of B. all truncated invocations charge.

Time $\mathcal{O}(n \log\log n)$ *IF* Invariant Holds first, consider pairs (R, v) we can charge truncated invocations to:

- if R has level 0 (one edge in R): at most one invocation
  $\mathcal{O}(n)$ pairs (R, v) on level 0
- if R has level 1: one level-1 invocation, $\leqslant \log n$ level-0 invocations
  $\mathcal{O}(n/\log^4 n) \cdot \mathcal{O}(\log^2 n)$ pairs (R, v) on level 1
- if R has level 2 (let $\alpha_2 = 1$): one level-2 invocation, one level-1 invocation, $\leqslant \log n$ level-0 invocations
  one such pair: $(R(G), s)$

second, sum non-truncated and truncated

- total level-0 invocations s_0 is $\mathcal{O}(n)$ (since each level-0 invocation is truncated)
- total level-1 invocations s_1 is $\leqslant s_0/\alpha_1 + \mathcal{O}(n/\log^2 n) = \mathcal{O}(n/\log n)$
- total level-2 invocations s_2 is $\leqslant s_1/\alpha_2 + 1 = s_1 + 1 = \mathcal{O}(n/\log n)$ (only one truncated)

third, analyze running time τ_i per level (except GLOBALUPDATE)

- for level-0 invocations: one operation on queue with one element
- for level-1 invocations: $\log n$ operations on queue with $\log^4 n$ elements
- for level-2 invocations: one operation on queue with $\mathcal{O}(n/\log^4 n)$ elements

Level i	Calls α_i	Time per Inv. $\tau_i = \alpha_i \log \lvert Q_i \rvert$	# Pairs (R, v) p_i	Charge c_i	# Trunc. Chargers t_i	Tot. # Inv. $s_i \leqslant s_{i-1}/\alpha_i + t_i$	Total Time $s_i \tau_i$
l. 2	1	$\mathcal{O}(\log n)$	1	$1 + 1 + \log n$	1	$\mathcal{O}(n/\log n)$	$\mathcal{O}(n)$
l. 1	$\log n$	$\mathcal{O}(\log n \log\log n)$	$\mathcal{O}(n/\log^2 n)$	$1 + \log n$	$\mathcal{O}(n/\log^2 n)$	$\mathcal{O}(n/\log n)$	$\mathcal{O}(n \log\log n)$
l. 0	0	$\mathcal{O}(1)$	$\mathcal{O}(n)$	1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
$\sum$							$\mathcal{O}(n \log\log n)$

forth (not covered in class), time for GLOBALUPDATE. assume in- and out-degree $\leqslant 2$ [using planarity here!]

- each call to GLOBALUPDATE starts at level 0. out-degree $\leqslant 2 \rightsquigarrow \mathcal{O}(n)$ calls
- if GLOBALUPDATE stops on or before level 1: $\mathcal{O}(n) \cdot \mathcal{O}(\log\log n)$
- else
  - if level-0 invocation charged to level-1 or level-2 invocation:
    only $\mathcal{O}(n/\log n)$ of these $\rightsquigarrow$ time $\mathcal{O}(n/\log n) \cdot \mathcal{O}(\log n)$
  - else (level-0 invocation charged to itself, at most once (by invariant))
    say for region $R(uv)$, we update $\mathbf{d}[v]$. since we go up to level 2, v must be boundary node of level 1 (cannot be minimum otherwise). at most $\mathcal{O}(n/\log^2 n)$ of those. by in-degree 2, at most two level-2 invocations for v

Linear-Time Algorithm similar analysis with rather involved recursion

References [SSSP, non-negative lengths]

For general graphs with non-negative edge lengths, the fastest known algorithm is Dijkstra's [Dij59] using Fibonacci heaps [FT87]. It runs in time $\mathcal{O}(m + n \log n)$. If the weighs are restricted to non-negative *integer* or *floating-point* values, there are linear-time algorithms for undirected graphs [Tho99, Tho00] and almost-linear-time algorithms for directed graphs [Hag00].

Improvements for planar graphs were first found by Frederickson [Fre87]. The linear-time algorithm is by Henzinger, Klein, Rao, and Subramanian [HKRS97].

[Dij59] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Fre87] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. Announced at FOCS 1984.

[Hag00] Torben Hagerup. Improved shortest paths on the word RAM. In *27th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 61–72, 2000.

[HKRS97] Monika Rauch Henzinger, Philip Nathan Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. Announced at STOC 1994.

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999. Announced at FOCS 1997.

[Tho00] Mikkel Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35(2):189–201, 2000. Announced at STACS 1998.