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 $\ell : E \rightarrow \mathbb{R}^+$

General Graphs  fastest: Dijkstra’s algorithm $O(m + n \log n)$

Planar Graphs?  can we use $r$–division? recall that $r$–division of $G$ is decomposition into
- $O(n/r)$ edge-disjoint pieces,
- each with $\leq r$ vertices and
- $O(\sqrt{r})$ boundary vertices. $\Leftarrow$ vertices with edges to at least two pieces

assume (temporarily) we can compute $r$–division in $O(n)$.

(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 $\partial P$)
  time required:
  $O\left(\frac{n \log \log n}{\log n} \sqrt{\frac{\log n}{\log \log n}}\right) \cdot O\left(\frac{\log n}{\log \log n} \log \left(\frac{\log n}{\log \log n}\right)\right) = O\left(n \sqrt{\log n \log \log n}\right)$

  let $G'$ denote graph with each piece $P$ replaced by complete graph on $\partial P$
- compute SSSP (Dijkstra) in $G'$
  time required: $n' := |V(G')| = O(n \sqrt{\log \log n / \log n})$ and $m' := |E(G')| = O(n)$, therefore
  $O(n' \log n' + m') = O\left(n \sqrt{\log n \log \log n}\right)$
- FOR EACH piece $P$, FOR EACH boundary node $p \in \partial P$
  compute SSSP in $P$ starting with $d[p] := d_G(s, p)$
  (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

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

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

improvement over $O(n \log n)$

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

**Def.** A $\rho$–clustering of $G$ is a decomposition into

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

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

**Proof.** Problem Set. Hint: DFS tree

**Algorithm**

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

in this process: how many new boundary nodes? how many new pieces? $\Rightarrow$ 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 \( \rightarrow R(G), R(P_1), \ldots R(e) \)
  2. allocate queue \( Q \) per piece
  3. initialize: \( \forall v : d[v] := \infty \)
  4. \( d[s] := 0 \)
  5. FOR EACH edge \( sv \in E(G) \)
  6. \( \text{GLOBALUpdate}(R(sv), sv, 0) \)
  7. WHILE \( \text{MINKey}(Q(R(G))) < \infty \)
  8. \( \text{Process}(R(G)) \)

- **GLOBALUpdate** \( \text{region } R, \text{item } x, \text{value } k \) recursive update value of \( x \) (if needed), queue consistency
  1. \( \text{UPDATEKey}(\text{queue } Q(R), \text{item } x, \text{value } k) \)
  2. IF \( x \) is new \( \text{MINKey}(Q(R)) \)
  3. \( \text{GLOBALUpdate}(\text{Parent}(R), R, k) \)

- **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 \( d[v] > d[u] + \ell(uw) \)
  3. \( d[v] := d[u] + \ell(uw) \)
  4. FOR EACH \( vw \in E : \text{GLOBALUpdate}(R(vw), vw, d[v]) \) note: key of arc \( vw \) is \( d[v] \)
  5. \( \text{UPDATEKey}(Q(R), vw, \infty) \)
  6. ELSE
  7. FOR \( \alpha_{h(R)} \) times AND WHILE \( \text{MINKey}(Q(R)) < \infty \): \( \alpha_i \) defined for recursion height of piece \( R \)
  8. \( R' := \text{EXTRACTMin}(Q(R)) \)
  9. \( \text{Process}(R') \)
  10. \( \text{UPDATEKey}(Q(R), R', \text{MINKey}(Q(R'))) \)

3 Correctness

three properties imply correctness

- **[initialized]** \( d[s] = 0 \)

- **[path-length property]** for each \( v \), \( d[v] \geq d_G(s, v) \) (at least length of some path)

- **[edges relaxed]** for each \( vw \in E : d[w] \leq d[v] + \ell(vw) \)

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 (\( \forall v : d[v] = \infty \)).

  - arc can be **tense** (not relaxed) when labels \( d[\cdot] \) of endpoints change. labels never increase \( \rightarrow \) ok to look at \( d[v] \): might decrease but immediately after we activate edges!

- uv active \( \Rightarrow \) key of \( uv \) is \( d[v] \) (except at step 4)

- queue consistency: for region \( R \), key associated with \( R \) in parent queue \( Q(\text{Parent}(R)) \) is \( \text{MINKey}(Q(R)) \)
  (except for current region and its ancestors)

  corollary: \( \text{MINKey}(Q(R)) \) value is \( \min \) among all active arcs in \( R \)

  \( \Rightarrow \) if \( \text{MINKey}(Q(R(G))) = \infty \), 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 \(\rightarrow\) 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 \(\mathcal{O}(n)\) \(\rightarrow\) each non-truncated invocation on level 1 causes \(\log n\) invocations on level 0 \(\rightarrow\) 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, \(\leq \log n\) 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, \(\leq \log n\) 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 \(\leq s_0/\alpha_1 + \mathcal{O}(n/\log^2 n) = \mathcal{O}(n/\log n)\)
- total level-2 invocations \(s_2\) is \(\leq s_1/\alpha_2 + 1 = s_1 + 1 = \mathcal{O}(n/\log n)\) (only one truncated)

third, analyze running time \(\tau_i\) 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

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

- each call to GlobalUpdate starts at level 0. out-degree \(\leq 2 \rightarrow \mathcal{O}(n)\) calls
- if GlobalUpdate stops on or before level 1: \(\mathcal{O}(n) \cdot \mathcal{O}(\log n)\)
- else
  - if level-0 invocation charged to level-1 or level-2 invocation:
    \(\mathcal{O}(n/\log n)\) of these \(\rightarrow\) 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 \(R(w)\), we update \(d[v]\). 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 $O(m + n \log n)$. If the weights 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].


