6.889 — Lecture 12: Exact Distance Oracles
(a.k.a. Shortest-Path Queries)

Christian Sommer csom@mit.edu
(figures extracted from [Dji96, FR06])

October 24, 2011

**Distance Oracle:** given a graph $G = (V, E)$, preprocess it into a data structure such that we can compute shortest-path distances $d_G(v, w)$ (distance queries) efficiently (and output path if desired). Two algorithms: one algorithm to preprocess the graph, one algorithm to query the data structure.

**Assumption (all of Lecture 12)** planar $G$, non-negative edge lengths $\ell : E \to \mathbb{R}^+$

**Lazy strategy** run SSSP algorithm for every query $d_G(v, w)$
preprocessing time $O(n)$, space $O(n)$ (store the graph), query time $O(n^2)$

**Eager strategy** precompute APSP, complete distance matrix, one table lookup to answer query $d_G(v, w)$
preprocessing time and space $O(n^2)$, query time $O(1)$

**Oracle** something “between” SSSP and APSP? applications: route planning, traffic simulations, etc.
main concern: *preprocessing time* (running time of the first algorithm), *space* consumption of the data structure (size of the output of the first algorithm), and *query time* (running time of the second algorithm) — in particular, *tradeoffs* between these quantities

**Recall: MSSP data structure** preprocessing time and space $O(n \log n)$, query time $O(\log n)$ (queries however restricted to source on single face)

**$r$–division approach** pieces of size $O(r)$ with boundary $O(\sqrt{r})$ per piece (total boundary $O(n/\sqrt{r})$)
precompute APSP for all nodes on the boundary, space $O(n^2/r)$; at query time, explore piece of $v$ (say $P_v$) and piece of $w$ ($P_w$) and find best connection pair $\partial P_v \times \partial P_w$, piece sizes $O(r)$ and $O(\sqrt{r})^2$ connection pairs, total query time $O(r)$ ○ smoothly interpolates between SSSP and APSP (only separators used, extends to minor-free graphs). can we do better?

**Connections between pieces** $O(\sqrt{r})^2$ connection pairs in $\partial P_v \times \partial P_w$, not independent!
(assume boundary $\partial P$ on $O(1)$ cycles, $r$–division with $O(1)$ holes per piece $P$)
Non-crossing property and bipartite Monge search

Main idea store distance from $v$ to all nodes in $\partial P_v$ with respect to $G$ (not just $P_v$). space $O(n\sqrt{r})$.

between boundary nodes, precompute distance in $G \setminus (P_v \cup P_w)$. space $O(n^2/r)$.

at query time, divide and conquer for the best pair in $\partial P_v \times \partial P_w$. For each boundary node $y \in \partial P_w$,

- find the best boundary node $x \in \partial P_v$, minimizing $\min_{x \in \partial P_v} d_G(v, x) + d_G(\setminus (P_v \cup P_w))(x, y)$

if query algo already found pairs $(x_1, y_2), (x_2, y_2)$, the min for $y$ is restricted to all $x$ between $x_1$ and $x_2$

![Figure 1: Non-crossing property](image1)

also known as the Monge property: $\forall u \leq v \forall x \leq y$: $d(u, x) + d(v, y) \leq d(u, y) + d(v, x)$

![Figure 2: Non-crossing and Monge properties](image2)

Bipartite dense distance graph complete bipartite graph on node set $X \cup Y$, edge lengths correspond to shortest-path distances in some graph $H$, i.e. $\ell(x, y) := d_H(x, y)$

Bipartite Monge search given $d(x, y)$ (dense distance graph for $X \times Y$), compute parent $x \in X$ for all $y \in Y$ (directed matching, one-sided, $x$ can be parent of multiple $y$).

If $d(\cdot, \cdot)$ from planar $G$ (Monge) then parent intervals: $x$ parent of $y_i$ and $y_k \Rightarrow x$ parent of $y_j$. $\forall i \leq j \leq k$.

If $d(\cdot, \cdot)$ satisfies Monge property, divide and conquer computes matching in time $O((|X| + |Y|) \log(|X| + |Y|))$. Significantly faster than $\Omega(|X| \cdot |Y|)$, algorithm does not read all of $d(\cdot, \cdot)$. Can even do $O(|X| + |Y|)$.

Note: still Monge with initialization $D(\cdot)$ for each $x \in X$ (think of $D(\cdot)$ as $v$-to-$\partial P_v$ shortest-path distance)

Oracle Problem need to store $v$-to-$\partial P_v$ distances, requires $O(n\sqrt{r})$ space (dominates space for $r > n^{2/3}$)

Use MSSP? know how to compute (and compactly represent) distances in $P_v$ and $G \setminus P_v$, but not in $G$ shortest paths may use several nodes of $\partial P_v$, leaving and re-entering $P_v$ arbitrarily!
On-line bipartite Monge search

Important subroutine in efficient implementation of Dijkstra’s algorithm running on dense distance graphs. The algorithm computes shortest paths by combining several instances of the following problem:

**On-line bipartite Monge search** given \(d(x, y)\) (dense distance graph for \(X \times Y\)), maintain parent \(x \in X\) for all \(y \in Y\), while initialization \(D(\cdot)\) for each \(x \in X\) is revealed on-line, one node at a time. Can compute matching on-line in overall time \(O((|X|+|Y|) \log(|X|+|Y|))\) (same time as divide and conquer).

**Maintaining a matching** means (herein) managing
- set of active nodes \(A \subseteq X\) and
- growing set of matched nodes \(M \subseteq Y\) (more importantly: shrinking set of yet unmatched nodes \(Y \setminus M\))

while supporting three operations (to be used by Dijkstra’s algorithm):
- \(\text{FindMin}()\) returns the \(\min\) unmatched node \(y \in Y \setminus M\), functions as priority queue for right-hand side
- \(\text{ExtractMin}()\) adds the current \(\min\) to \(M\)
- \(\text{ActivateLeft}(x, \delta)\) reveals initialization \(D(x) = \delta\) for some \(x \in X\) (and updates preliminary matches!)

**Efficient implementation** use heap and intervals in ordered set \(Y\). Assume that, for each \(x \in X\), precomputed (together with dense distance graph), data structure supporting queries \(\min_{1 \leq i \leq t} d(x, y_i)\) for any \(i_-, i_+\) (note: data structure independent of \(D(\cdot)\), query using LCA in \(O(1)\)).

Maintain binary search tree for active nodes \(x \in A\).

For active \(x \in A\), maintain interval \([i_-(x), i_+(x)]\) of children \(y \in Y\) (\(x\) is current parent of \(y\), may change). Maintain priority queue (heap) containing, for each active \(x \in A\), its shortest edge to unmatched \(y \in Y \setminus M\).
- \(\text{FindMin}()\) returns \(\min\) from heap, \(O(1)\)
- \(\text{ExtractMin}()\) adds the current \(\min\) to \(M\) (found by \(\text{FindMin}()\)); suppose the \(\min\) was \(y_j \in Y \setminus M\) with parent \(x \in A\) need to insert second-shortest edge from \(x\) into heap; instead, create two dummy nodes \(x', x''\) spanning intervals \([i_-(x), j]\) and \([j+1, i_+(x)]\), respectively. find \(\min\) in these intervals using above LCA data structure (for \(x\)) and insert into heap (time \(O(\log |Y|)\)).

how many new dummy nodes? at most two for each \(\min\) in \(Y\) extracted from heap, \(O(|X|+|Y|)\) overall

- \(\text{ActivateLeft}(x, \delta)\): new node \(x \in X\) is activated. compute its interval \([i_-(x), i_+(x)]\). all of \(Y\) if first \(x\), otherwise
  - “walk up” in \(A\) (non-empty intervals only) until first \(x'\) whose \(d(x', y_{i_-(x')}) < d(x, y_{i_-(x')})\),
  - binary search for \(i_-(x)\) in the range \([i_-(x'), i_+(x')]\) (time \(O(\log |Y|)\))
  - analogously, “walk down” in \(A\) (non-empty int.) until first \(x''\) w. \(d(x'', y_{i_+(x'')-1}) < d(x, y_{i_+(x'')-1})\),
  - binary search for \(i_+(x)\) in the range \([i_-(x''), i_+(x'')]\) (time \(O(\log |Y|)\))

update other intervals and heap
- for nodes in \(A\) between \(x\) and \(x'\) (and between \(x\) and \(x''\)), interval becomes empty, “deactivate” and remove corresponding \(\min\) in \(Y \setminus M\) from heap (note that sequential search (“walk up/down”) visits such nodes at most once, amortized \(O(\log |Y|)\) per \(x \in X\)
- for \(x'\) and \(x''\), interval may shrink to \([i_-(x'), i_-(x)]\) and \([i_+(x)+1, i_+(x'')]\), respectively (could also become empty), update corresponding \(\min\) in \(Y \setminus M\) in heap (time \(O(\log |Y|)\))
Figure 3: On-line bipartite Monge search
**Efficient Dijkstra implementation**

called *FR-Dijkstra* due to Fakcharoenphol and Rao

**Dense distance graph**  complete graph on node set $X$, edge lengths correspond to shortest-path distances in some graph $H$, i.e. $\ell(x, x') := d_H(x, x')$. Often $X$ is the boundary $\partial P_u$ of a piece $P_u$, and distances are with respect to $P_u$.

**Reduction to bipartite case**  recursively “cut” $X$ into halves, two bipartite graphs (one from left to right, another one from right to left), each node in $\leq 2\lceil\log_2 |X|\rceil$ bipartite graphs

![Figure 4: Reduction to bipartite Monge](image)

**Note**  Nodes are on the left-hand-side of some bipartite graphs, right-hand-side of other bipartite graphs. Each arc (directed edge) is in exactly one bipartite graph.

**Algorithm**  compute SSSP tree on $n$ nodes of a planar graph (connected by dense distance graphs) in time $O(n \log^2 n)$ (note that there may be $\Omega(n^2)$ edges)

The *FR-Dijkstra* algorithm first converts each dense distance graph (DDG) into $2\lceil\log n\rceil$ bipartite DDGs $(X_i, Y_i)$; then it runs on-line bipartite Monge searches for all the instances.

These instances are combined using a global priority queue with one element per instance $(X_i, Y_i)$. Each instance runs in time $O((|X_i| + |Y_i|) \log(|X_i| + |Y_i|))$, the overall time is thus $O(n \log^2 n)$.

<table>
<thead>
<tr>
<th>DIJKSTRA($G, s$)</th>
<th>FR-DIJKSTRA($G, s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall v \in V(G) : d(v) = \infty; d(s) = 0; S = \emptyset$</td>
<td>$\forall v \in V(G) : d(v) = \infty; d(s) = 0; S = \emptyset$</td>
</tr>
<tr>
<td>maintain global heap with respect to $d(\cdot)$</td>
<td>convert each DDG into bipartite DDGs $(X_i, Y_i)$</td>
</tr>
<tr>
<td>WHILE $S \neq V(G)$</td>
<td>FOR EACH $(X, Y)$ s.t. $s \in Y$</td>
</tr>
<tr>
<td>$u \leftarrow EXTRACTMIN(V \setminus S)$</td>
<td>$(X, Y) \leftarrow GLOBAL\cdot EXTRACTMIN()$</td>
</tr>
<tr>
<td>FOR EACH $e = (u, v) \in E(G)$</td>
<td>$u \leftarrow (X, Y)\cdot EXTRACTMIN(Y \setminus M)$</td>
</tr>
<tr>
<td>$d(v) \leftarrow \min{d(v), d(u) + \ell(u, v)}$ (relax $e$)</td>
<td>IF $u \notin S$ (could have been settled in $(\hat{X}, \hat{Y})$)</td>
</tr>
<tr>
<td>$S = S \cup {u}$</td>
<td>FOR EACH $(X', Y')$ s.t. $u \in X'$ “relax” edges in DDGs</td>
</tr>
<tr>
<td></td>
<td>$(X', Y') \cdot ACTIVATELEFT(u, d(u))$</td>
</tr>
<tr>
<td></td>
<td>$v \leftarrow (X', Y')\cdot FINDMIN(Y' \setminus M')$</td>
</tr>
<tr>
<td></td>
<td>$GLOBAL\cdot DECREASEKEY((X', Y'), d(v))$</td>
</tr>
<tr>
<td></td>
<td>$S = S \cup {u}$</td>
</tr>
<tr>
<td></td>
<td>$v \leftarrow (X, Y)\cdot FINDMIN(Y \setminus M)$</td>
</tr>
<tr>
<td></td>
<td>$GLOBAL\cdot DECREASEKEY((X, Y), d(v))$</td>
</tr>
</tbody>
</table>
Distance oracles

Quasi-linear space

Preprocessing  recursively apply cycle separator and compute dense distance graph for inside and outside of cycle (using MSSP). Time per level is \( O(n \log n) \), recursion depth is \( O(\log n) \); overall preprocessing time \( O(n \log^2 n) \) and space requirement \( O(n \log n) \).

Query  given query pair \((v, w)\), run efficient implementation of Dijkstra’s algorithm on dense distance graphs for all cycles enclosing \( v \) or \( w \). Cycles on \( O(\sqrt{n}) \) nodes, cycle lengths decrease geometrically with recursion, total \( O(\sqrt{n}) \) nodes, overall query time \( O(\sqrt{n} \log^2 n) \).

“Arbitrary” space

Preprocessing  for any \( r \in [1, n] \), compute \( r \)-division, and, for each piece \( P \),

- compute and store MSSP data structure for inside and outside of \( P \), respectively,
  (at the same time: compute and store dense distance graphs \( DDG(P) \) and \( DDG(\bar{P}) \) for inside and outside of \( P \), respectively), and

- preprocess quasi-linear-space distance oracle for each piece.

Overall preprocessing \( O(n/r) \cdot O(n \log n) + O(n/r) \cdot O(r \log^2 r) \) and space \( O((n^2/r) \log n) \).

Query  given query pair \((v, w)\),

- if in the same piece, query distance oracle in time \( O(\sqrt{r} \log^2 r) \)
- in any case (even if in the same piece \( P \), shortest path could leave and re-enter \( P \)),
  - query MSSP of \( P_v \) for \( v \)-to-\( \partial P_v \) distances (in \( P_v \)),
  - query MSSP of \( \bar{P}_v \) for \( \partial P_v \)-to-\( w \) distances (in \( \bar{P}_v := (G \setminus P_v) \cup \partial P_v \)),
  - run efficient Dijkstra implementation on \( \{v\} \times \partial P_v, \partial P_v \times \{w\} \), \( DDG(P_v) \), and \( DDG(\bar{P}_v) \)

Overall query time \( O(\sqrt{r} \log^2 r + \sqrt{r} \log n) \).
Exact distance oracles and shortest-path queries for planar graphs have been investigated by several researchers [FMS91, Dj96, ACC⁺96, CX00, FR06, Cab06, Nus11, MS12]. Djidjev [Dji96] first exploited the non-crossing property for pairs of boundary nodes. The efficient implementation of Dijkstra’s algorithm (using on-line Monge search) is by Fakcharoenphol and Rao [FR06] (named Monge search due to [Mon81]). The quasi-linear-space construction is by [FR06]; the construction for arbitrary space can be found in [MS12].


