15 Shortest Paths in Planar Graphs

15.1 Dense distance graphs

One of the most important applications of separators and $r$-divisions in planar graphs is faster algorithms to compute shortest paths. Most of these faster algorithms rely on an implicit representation of shortest-path distances called the dense distance graph, first proposed by Jittat Fakcharoenphol and Satish Rao in 2001.

Let $\Sigma$ be a simple planar map with weighted edges; for now we'll assume that all edge weights are non-negative. Add zero-weight edges to obtain a simple triangulation $\Delta$. Recall that a good $r$-division of $\Sigma$ is a subdivision of $\Sigma$ into $O(n/r)$ pieces $R_1, R_2, \ldots$ satisfying three conditions:

- Each piece has $O(r)$ vertices.
- Each piece has $O(\sqrt{r})$ boundary vertices (that is, vertices that are shared with other pieces).
- Each piece has $O(1)$ holes (faces of the piece that are not faces of $\Sigma$).

Fix a good $r$-division $\mathcal{R}$. For each piece $R_i \in \mathcal{R}$, let $X_i$ be a clique over the boundary vertices of $R_i$, where each edge is weighted by the shortest-path distance in $R_i$ between its endpoints. The dense distance graph is the union of these $O(n/r)$ weighted cliques. Altogether, the dense distance graph has $O(n/r)$ vertices—only the boundary vertices of the pieces of the $r$-division—and $O(n)$ weighted edges.

Assuming all edge weights are non-negative, we can compute all $O(r)$ boundary-to-boundary shortest-path distances in each piece $R_i$ in $O(r \log r)$ time, by running the multiple-source shortest-path algorithm once for each hole in $R_i$, using Dijkstra's algorithm to compute the initial shortest-path tree. Thus, the overall time to compute the dense-distance graph is $O(n \log r)$.

15.2 Beating Dijkstra

**Theorem:** Given any planar map $\Sigma$ with non-negative lengths on its edges, we can compute the shortest path between any two vertices of $\Sigma$ in $O(n \log \log n)$ time.

**Proof:** Triangulate $\Sigma$ in $O(n)$ time, build a good $r$-division for the resulting triangulation in $O(n)$ time, and then build the dense distance graph for the resulting $r$-division in $O(n \log r)$ time, for some parameter $r$ to be determined. In the top-level recursive call to build the $r$-division, we artificially declare $s$ and $t$ to be boundary vertices, so that they survive as vertices in the dense-distance graph. We can compute the shortest-path distance from $s$ to every other vertex of the dense distance graph (including $t$) using Dijkstra's algorithm, in time $O(V \log V + E) = O((n/\sqrt{r}) \log n + n)$. If we set $r = O(\log^2 n)$, the overall running time of the algorithm becomes $O(n \log r + (n/\sqrt{r}) \log n) = O(n \log \log n)$. \hfill \Box

15.3 Repricing

Depending on which textbook you read, Dijkstra's algorithm is either no longer correct or no longer efficient when some edges of the graph have negative-weight weights. The standard shortest-paths algorithm for graphs with negative edges is Bellman-Ford, which runs in $O(V E)$ time; for simple planar graphs with $n$ vertices, Bellman-Ford takes $O(n^2)$ time. But just as we beat Dijkstra's, we can beat Bellman-Ford when the underlying graph is planar.

Our more efficient algorithms all use a standard repricing technique first\(^1\) proposed independently.

\(^1\)At least, first explicitly proposed. Arguably the repricing technique is already implicit in Jacobi's early 19th-century

Suppose each vertex \( v \) has an associate price \( \pi(v) \). We can assign a new edge-length function \( \ell' \) as follows:

\[
\ell'(u \rightarrow v) := \pi(u) + \ell(u \rightarrow v) - \pi(v).
\]

Then for any path \( s \rightarrow t \) in \( G \), we have a telescoping sum

\[
\ell'(s \rightarrow t) := \pi(s) + \ell(s \rightarrow t) - \pi(t).
\]

Because the length of every path from \( s \) to \( t \) changes by the same amount, the shortest paths from \( s \) to \( t \) with respect to \( \ell \) and \( \ell' \) coincide! Thus, if we can find a pricing function that makes all new edge lengths \( \ell'(u \rightarrow v) \) non-negative, we can compute shortest-path distances with respect to \( \ell' \) using Dijkstra’s algorithm in \( O(n \log n) \) time, or its more efficient planar replacement in \( O(n \log \log n) \) time, and then recover distances with respect to \( \ell \) as follows:

\[
dist(s, t) := dist'(s, t) - \pi(s) + \pi(t).
\]

For example, suppose \( \pi(v) = dist(s, v) \) for some fixed source vertex \( s \), where \( dist \) denotes shortest-path distance with respect to \( \ell \). Then we have

\[
\ell'(u \rightarrow v) := dist(s, u) + \ell(u \rightarrow v) - dist(s, v).
\]

Ford’s formulation of shortest paths implies that the expression on the right is non-negative. Thus, once we’ve computed shortest paths from one source, we can efficiently compute shortest paths from any other source in near-linear time.

15.4 **Beating Bellman-Ford: Nested dissection**

One of the earliest applications of planar separators was an algorithm to compute shortest paths in planar graphs in subquadratic time. Unlike most algorithms in this class, this algorithm deals with abstract planar graphs rather than planar maps. Fortunately, Lipton and Tarjan’s original separator algorithm does not require an embedding.

Let \( G \) be a simple planar graph (sic) with \( n \) vertices, where every edge \( e \) has a real length \( \ell(e) \), and let \( s \) be a vertex of \( G \). Suppose we want to compute the shortest-path distance from \( s \) to every other vertex in \( G \). To simplify the presentation, I’ll assume that no cycle in \( G \) has negative total length, so that these shortest-path distances are well-defined.

First we compute a balanced separator \( S \) for \( G \). This separator splits \( G \) into two subgraphs \( A \) and \( B \), each with at most \( 3n/4 \) vertices, such that \( A \cap B = S \) and any path from \( A \setminus S \) to \( B \setminus S \) contains at least one vertex in \( S \). In light of the previous section, we can assume without loss of generality that \( s \in S \).

Let me write \( dist_P(X, Y) \) to mean the set of all shortest-path distances in subgraph \( P \) from vertices in \( X \) to vertices in \( Y \). The algorithm proceeds in four stages.

1. Recursively compute \( dist_A(s, A) \) and \( dist_B(s, B) \). How the Recursion Fairy does this is none of your business.
2. Compute $\text{dist}_A(S, S)$ and $\text{dist}_B(S, S)$. Assign each vertex $v$ in $A$ the price $\pi_A(v) = \text{dist}_A(s, v)$, and then for each vertex $s \in S$, compute $\text{dist}_A(s, S)$ via Dijkstra's algorithm. Computing $\text{dist}_B(S, S)$ is symmetric. (Yes, vertices in $S$ may get different prices in $A$ and $B$.) The total time for this stage is $O(n^{3/2} \log n)$.

3. Compute $\text{dist}_G(s, S)$. Build a complete directed graph $H$ with vertices $S$, where each edge $u \rightarrow v$ has length $\min\{\text{dist}_A(u, v), \text{dist}_B(u, v)\}$. The graph $H$ has $O(\sqrt{n})$ vertices and $O(n)$ edges, so we can compute $\text{dist}_G(s, t) = \text{dist}_H(s, S)$ using Bellman-Ford in $O(n^{3/2})$ time.

4. Finally, compute $\text{dist}_G(s, V)$ by brute force. For each vertex $v \in A$, we have $\text{dist}_G(s, v) = \min_{t \in S} (\text{dist}_G(s, t) + \text{dist}_A(t, v))$, and similarly for vertices in $B$. This final stage takes $O(n^{3/2})$ time.

The overall running time of the algorithm satisfies the recurrence

$$T(n) \leq T(n_1) + T(n_2) + O(n^{3/2} \log n)$$

where $n_1 + n_2 = n + O(\sqrt{n})$ and $\max\{n_1, n_2\} \leq 3n/4 + O(\sqrt{n})$. We conclude that the algorithm runs in $O(n^{3/2} \log n)$ time; the bottleneck is stage 2, followed closely by stages 3 and 4.

Figure 1: Computing planar shortest paths by nested dissection

### 15.5 Easy Improvements for Planar Maps

We can make several straightforward improvements to this basic algorithm, especially if the underlying graph $G$ is embedded in the plane.

1. See the sign on the Recursion Fairy’s door that says “Do Not Disturb”? They mean it.

2. If the graph $G$ is embedded, we can add infinite-length edge to transform the embedding into a simple triangulation, and then compute a balanced cycle separator $S$. Then we can compute $\text{dist}_A(S, S)$ and $\text{dist}_A(S, S)$ using the multiple-source shortest-path algorithm, in $O(n \log n)$ time, by treating $S$ as the boundary of the outer face of $A$ and $B$, respectively.

3. We’ll see how to improve Bellman-Ford shortly.

4. Instead of brute force, we can rely on Johnson’s repricing trick. We construct a graph $H$ from the disjoint union $A \cup B$ as follows First, we add an artificial source vertex $\hat{s}$. Then, for each vertex $v \in S$, we add edges $\hat{s} \rightarrow v_A$ and $\hat{s} \rightarrow v_B$ to the copies of $v$ in $A$ and $B$, both with length $\text{dist}_G(s, v)$. Now for any target vertex $t$, we have $\text{dist}_G(s, t) = \text{dist}_H(\hat{s}, t)$. Now define prices for the vertices of $H$ as follows:

$$\pi(v) = \begin{cases} 
\text{dist}_A(s, v) & \text{if } v \text{ is a vertex of } A \\
\text{dist}_B(s, v) & \text{if } v \text{ is a vertex of } B \\
\infty & \text{if } v = \hat{s}.
\end{cases}$$
where $\infty$ is a symbolic placeholder for some sufficiently large value.\textsuperscript{2} Straightforward
calculation implies that edges in $H$ have non-negative repriced length. $H$ has $O(n)$ vertices
and edges, so we can compute shortest paths in $H$ in $O(n \log n)$ time via Dijkstra’s algorithm,
or in $O(n \log \log n)$ time using our earlier planar shortest-path algorithm.

Our improvements to stages 2 and 4 decrease the running time of our algorithm to $O(n^{3/2})$. Any
further improvements require speeding up Bellman-Ford, which is exactly what we’re going to
do next!

15.6 Monge arrays and SMAWK

A two-dimensional array $M$ is Monge if

$$M[i, j] + M[i', j'] \leq M[i, j'] + M[i', j]$$

for all array indices $i < i'$ and $j < j'$. Monge arrays are named after the French
geometer and civil engineer Gaspard Monge, who described an equivalent geometric condition in his 1781 Mémoire
sur la Théorie des Déblais et des Remblais. Monge observed that if $A, B, b, a$ are the vertices of a
convex quadrilateral in cyclic order, the triangle inequality implies that $|Aa| + |Bb| < |Ab| + |aB|$.

Figure 2: Monge’s observation: Non-crossing paths are shorter

Monge Structure Lemma: The following arrays are Monge:

(a) Any array with constant rows.
(b) Any array with constant columns.
(c) Any array that is all 0s except for an upper-right rectangular block of 1s.
(d) Any array that is all 0s except for an lower-left rectangular block of 1s.
(e) Any positive multiple of any Monge array.
(f) The sum of any two Monge arrays.
(g) The transpose of any Monge array.

In 1987, Alok Aggarwal, Maria Klawe, Shlomo Moran, Peter Shor, and Robert Wilber described
an elegant recursive algorithm that finds the minimum element in every row of an $n \times n$ Monge
array in $O(n)$ time, now usually called the SMAWK algorithm after the suitably-permuted initials

\textsuperscript{2}If you're uncomfortable with symbolic infinities, it suffices to set

$$\pi(\hat{s}) = \max \{\text{dist}_A(s, u) - \text{dist}_G(s, u_A), \text{dist}_B(s, u) - \text{dist}_G(s, u_B) \mid u \in S\}$$

2
of its authors. In 1990, Maria Klawe and Daniel Kleitman described an extension to the SMAWK algorithm that finds row-minima in partial Monge matrices, where some entries are undefined, but the Monge inequality holds whenever all four entries are defined. Klawe and Kleitman’s algorithm runs in $O(n \alpha(n))$ time, where $\alpha(n)$ is the slowly-growing inverse Ackermann function.

A description of these algorithms is beyond the scope of this class, but you can find a complete description and analysis of the SMAWK algorithm in my algorithms lecture notes.

15.7 Planar distance matrices are (almost) Monge

In the same 2001 paper where they defined dense distance graphs, Jittat Fakcharoenphol and Satish Rao described how to use SMAWK to compute shortest paths in planar maps more quickly.

Let $\Sigma$ be a planar map with weighted edges. Let $s_1, s_2, \ldots, s_k$ be the sequence of vertices on the boundary of the outer face of $\Sigma$, in cyclic order. (If the outer face boundary is not a simple cycle, the same vertex may appear multiple times in this list.) Let $D$ be the $k \times k$ array where $D[i, j] = \text{dist}_\Sigma(s_i, s_j)$.

**Lemma:** The distance array $D$ can be decomposed into two partial Monge matrices.

**Proof:** Fix four vertices $u, v, w, x$ in cyclic order around the boundary of the outer face of $\Sigma$. The Jordan curve theorem implies that the shortest paths from $u$ to $w$ and from $v$ to $x$ must cross; let $z$ be any vertex in the intersection of these two shortest paths. The triangle inequality implies

$$\text{dist}(u, w) + \text{dist}(v, x) = (\text{dist}(u, z) + \text{dist}(z, w)) + (\text{dist}(v, z) + \text{dist}(z, x))$$

$$= (\text{dist}(u, z) + \text{dist}(z, x)) + (\text{dist}(v, z) + \text{dist}(z, w))$$

$$\leq \text{dist}(u, x) + \text{dist}(v, w)$$

(omitting subscript $\Sigma$’s everywhere).

It follows that the Monge inequality

$$M[i, j] + M[i', j'] \leq M[i, j'] + M[i', j];$$

holds for any indices $i, i', j, j'$ that appear in that cyclic order (possibly with ties) modulo $k$. In particular, the Monge inequality holds whenever $i \leq i' \leq j \leq j'$, which implies that the portion of $M$ on or below the main diagonal is Monge. Symmetrically, the portion of $M$ on or above the main diagonal is Monge. These two partial Monge matrices cover $M$. □

Perhaps a better way to express this analysis is that the $k \times 2k$ partial array defined by

$$D[i, j] := \begin{cases} \text{dist}_G(s_i, s_j \mod k) & \text{if } i \leq j \leq i + k \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a single partial Monge array.

15.8 Beating nested dissection

Now recall that the third phase of our nested-dissection algorithm computes the distances $\text{dist}_H(s, S)$. It will be more convenient to think of $H$ as the overlay of two directed cliques $H_A$.
Figure 3: For any planar map, the array of boundary-to-boundary distances both splits into two partial Monge arrays (left) and unrolls into a single partial Monge array (right)

and $H_B$ over the vertices $S$. For each pair of vertices $u,v \in S$, both $H_A$ and $H_B$ contain an edge $u \rightarrow v$, with lengths $\ell_A(u \rightarrow v) = \operatorname{dist}_A(u,v)$ and $\ell_B(u \rightarrow v) = \operatorname{dist}_B(u,v)$, respectively.

This portion of the algorithm invokes Bellman-Ford, which has the following simple structure. After initializing $\operatorname{dist}_{ij}[s] = 0$ and $\operatorname{dist}_{ij}[v] = \infty$ for all $v \neq S$, the algorithm repeatedly identifies all tense edges in $H$, and then relaxes them all. The algorithm terminates after $O(k) = O(\sqrt{n})$ relaxation phases.

Here is some pseudo-Python for a single relaxation phase:

```python
for i in range(k):
    for j in range(k):
        if dist[j] < dist[i] + l[i,j]:
            dist[j] = dist[i] + l[i,j]
```

As written, this block of code runs in $O(k^2)$ time. Because the order that we scan the edges doesn't matter, let's first scan all edges in $H_A$ and then all edges in $H_B$:

```python
for i in range(k):
    for j in range(k):
        if dist[j] < dist[i] + lA[i,j]:
            dist[j] = dist[i] + lA[i,j]

for i in range(k):
    for j in range(k):
        if dist[j] < dist[i] + lB[i,j]:
            dist[j] = dist[i] + lB[i,j]
```

Now I'm going to do something a little weird to each of these loops. For each vertex $v$, I'll first figure out the minimum value of $\operatorname{dist}[u] + lA[i,u]$ and only compare that minimum value to $\operatorname{dist}[v]$ at the end.

```python
for j in range(k):
    bestcost = math.inf
    for i in range(k):
        if dist[i] + lA[i,j] < bestcost:
            best[i] = i
            bestcost = dist[i] + lA[i,j]

for j in range(k):
    if dist[j] < bestcost:
        dist[j] = bestcost
```
The first (outer) for-loop is choosing the minimum element in every column of a $k \times k$ matrix $M$, where

$$M[i,j] := \text{dist}(s_i) + \text{dist}_A(s_i, s_j)$$

$M$ is the sum of a matrix with constant columns (which is Monge) and the boundary-to-boundary distance matrix in $A$. Thus, $M$ can be split into two partial Monge arrays, and therefore so can its transpose. It follows that Klawe and Kleitman's algorithm can compute $\text{best}[j]$ for all $j$ in only $O(k\alpha(k))$ time.

With this optimization in place, Bellman-Ford computes all distances $\text{dist}_H(s,S)$ in $O(k^2\alpha(k)) = O(n\alpha(n))$ time. (This accelerated version of Bellman-Ford is now commonly called “FR-Bellman-Ford” after Fakcharoenphol and Rao, because nobody can spell Fakcharoenphol.)

With all these improvements in place, the running time of our shortest-path algorithm satisfies the recurrence

$$T(n) \leq T(n_1) + T(n_2) + O(n \log n) + O(n\alpha(n)) + O(n \log \log n)$$

where (after a suitable domain transformation) $n_1 + n_2 = n$ and $\max\{n_1, n_2\} \leq 3n/4$. We conclude that the algorithm runs in $O(n \log^2 n)$ time; our invocation of MSSP in stage 2 is (just barely) the bottleneck.

This faster algorithm was described by Philip Klein, Shay Mozes, and Oren Weimann in 2009. One year later, Shay Mozes and Christian Wulff-Nilsen described a further improvement using nice $r$-divisions that runs in $O(n \log^2 n/\log \log n)$.

### 15.9 Aptly Named Sir Not

- Shortest paths in $O(n)$ time.