13a Multiple-Source Shortest Paths, Revisited$^{\alpha}$

In a recent breakthrough, Das, Kipouridis, Probst Gutenberg, and Wulff-Nilsen described an alternative algorithm that solves the planar multiple-source shortest path problem using a relatively simple divide-and-conquer strategy. Their algorithm theoretically runs in $O(n \log h)$ time, where h is the number of vertices on the outer face, which improves the $O(n \log n)$ time of Klein's algorithm when h is small. Moreover, this running time is worst-case optimal as a function of both n and h.

A better expression for the running time is $O(S(n) \log h)$, where $S(n)$ is the time to compute a single-source shortest path tree.

- If we use Dijkstra's algorithm off the shelf, the running time is $O(n \log n \log h)$.
- If we use the $O(n \log \log n)$-time algorithm that we will see in Lecture 15,[1] the running time is $O(n \log h \log \log n)$.
- If we use the $O(n)$-time algorithm of Henzinger et al.,[2] the running time is the optimal $O(n \log h)$.

The new algorithm is simpler in the sense that it uses only black-box shortest-path algorithms, completely avoiding complex dynamic forest data structures that are inefficient in practice, at least for small graphs.[3] On the other hand, the new algorithm requires a subtle divide-and-conquer algorithm with weighted r-divisions, which is *also* inefficient in practice, to achieve its best possible running time $O(n \log h)$. On the gripping hand, Klein's algorithm has been observed to require a sublinear number of pivots for many inputs, so the $O(n \log n)$ time bound, while tight in the worst case, is usually conservative; whereas, the $O(n \log h)$ time bound for the new algorithm is tight for *all* inputs. It would be interesting to experimentally compare Klein (or CCE) using linear-time dynamic trees against the new algorithm using Dijkstra as a black box.

13a.1 Problem formulation

It will be convenient to describe the inputs and outputs of the MSSP problem slightly differently than in the previous lecture.

The input consists primarily of a *directed* planar map $\Sigma = (V, E, F)$ with a distinguished outer face o and a non-negative weight $\ell(u \to v)$ for every directed edge/dart $u \to v$, which could be infinite (to indicate that a directed edge is missing from the graph). The weights are not necessarily symmetric; we allow $\ell(u \to v) \neq \ell(v \to u)$.

Let $s_0, s_1, \ldots, s_{h-1}$ be any subsequence of h vertices in counterclockwise order around the outer face, and let $S = \{s_0, s_1, \ldots, s_{h-1}\}$. Our goal is to compute an implicit representation of the shortest paths from each source s_i to every original vertex of Σ. See Figure 1.

For ease of presentation, I will make a few minor technical assumptions:

[1] The $O(n \log \log n)$-time shortest-path algorithm from Lecture 15 uses the *parametric* MSSP algorithm from the previous lecture as a subroutine. If we instead recursively apply the recursive MSSP strategy described in this lecture, the resulting doubly-recursive MSSP algorithm runs in $O(n \log h \log \log n \log \log \log n \log \log \log \log n \ldots)$ time.

[2] This $O(n)$-time shortest-path algorithm does *not* use MSSP as a subroutine.

[3] David Eisenstat [2] implemented Chambers, Cabello, and Erickson's MSSP algorithm using both efficient dynamic trees and brute-force to find pivots. His experimental evaluation showed that the brute-force implementation was faster in practice for graphs with up to 200000 vertices. More generally, in a large-scale experimental comparison of several dynamic-forest data structures by Tarjan and Werneck [6, 7], brute-force implementation beat all other data structures for trees with depth less than 1000.

- Every source vertex $s_i \in S$ has out-degree 1 and in-degree 0. We can enforce this assumption if necessary by adding a new artificial source vertex s_i' and a single directed edge $s_i' \to s_i$ with weight 0.
- Σ is simple. We can enforce this condition if necessary by resolving parallel edges and deleting loops in $O(n)$ time using hashing.[4]
- The graph of $\Sigma \setminus S$ is strongly connected. Thus, the shortest path tree rooted at each source vertex s_i includes every non-source vertex. We can enforce this assumption if necessary by adding new infinite-weight edges.
- All shortest paths are unique. If necessary, we can enforce this assumption either by randomly perturbing the edge weights or by choosing leftmost shortest paths, just as in the previous lecture.

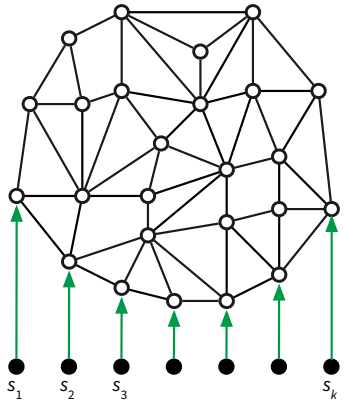

Figure 1: Setup for the recursive MSSP algorithm.

For any index j and any vertex v, let $path_j(v)$ denote the shortest path in Σ from s_j to v, let $dist_j(v)$ denote the length of this shortest path, and let $pred_j(v)$ denote the predecessor of v in this shortest path.

The main recursive algorithm MSSP-Prep preprocesses the map Σ into a data structure that implicitly encodes the single-source shortest path trees rooted at every source s_j. A separate query algorithm MSSP-Query(s_j, v) returns $dist_j(v)$.

13a.2 Overview

The preprocessing algorithm uses a divide-and conquer-strategy. The input to each recursive call MSSP-Prep(H, i, k) consists of the following:

- A planar map H, which is a simple weighted minor of the top-level input map Σ.
- Two indices i and k. To simplify presentation, we implicitly assume that $s_i, s_{i+1}, \ldots, s_k$ are the only source vertices in H.

For each index j, let T_j denote the tree of shortest paths in H from s_j to every other vertex of H. The recursive call MSSP-Prep(H, i, k) computes an implicit representation of all $k - i + 1$ shortest path trees $T_1, T_{i+1}, \ldots, T_k$. The top-level call is MSSP-Prep$(\Sigma, 0, h - 1)$.

MSSP-Prep invokes a subroutine Filter(H, i, k) that behaves as follows:

[4]The algorithms I describe in this note use hashing in multiple places. It is possible to achieve the same running time without hashing, at the expense of simplicity (and probably some efficiency).

- Compute the shortest path trees T_i and T_k rooted at s_i and s_k.
- Identify directed edges that are shared by all shortest path trees T_j with $i \le j \le k$.
- Contract shared edges and update nearby weights to maintain shortest path distances.
- Return the resulting contracted planar map.

Finally, ignoring base cases for now, MSSP-Prep(H, i, k) has four steps:

- Set $H' \leftarrow$ Filter(H, i, k).
- Set $j \leftarrow \lfloor (i+k)/2 \rfloor$.
- Recursively call MSSP-Prep(H', i, j).
- Recursively call MSSP-Prep(H', j, k).

Finally, MSSP-Prep(H, i, k) returns a record storing the following information:

- the indices i and k
- data about each vertex in H computed by Filter
- pointers to the records returned by the recursive calls

Said differently, MSSP-Prep returns a data structure that mirrors its binary recursion tree; every record in this data structure stores information computed by one invocation of Filter.

The time and space analysis of MSSP-Prep hinges on the observation that the total size of all minors H at each level of the resulting recursion tree is only $O(n)$. The depth of the recursion tree is $O(\log h)$, so the total size of the data structure is $O(n \log h)$. Similarly, aside from recursive calls, the time for each subproblem with m vertices is $O(S(m))$, so the overall running time is $O(S(n) \log h)$.

Finally, the query algorithm recovers the shortest-path distance from any source s_j to any vertex v by traversing the recursion tree of MSSP-Prep in $O(\log h)$ time.

In the rest of this note, I'll consider each of the component algorithms in more detail.

13a.3 Properly shared edges

Now I'll describe the filtering algorithm Filter(H, i, k) in more detail. For any index j and any vertex v, define the following:

- T_j is the shortest-path tree in H rooted at source vertex s_j.
- $dist_j(v)$ is the shortest-path distance in H from s_j to v.
- $pred_j(v)$ is the predecessor of v on the shortest path in H from s_j to v.

Our filtering algorithm Filter(H, i, k) begins by computing the distances $dist_i(v)$ and $dist_k(v)$ and predecessors $pred_i(v)$ and $pred_k(v)$ for every vertex v, using two invocations of your favorite shortest-path algorithm. The algorithm also initializes two variables for every vertex v, which will eventually be used by the query algorithm:

- A *representative* vertex $rep(v)$, initially equal to v.
- A non-negative real *offset* $off(v)$, initially equal to 0.

Call any directed edge $u \to v$ *properly shared* by T_i and T_k if it satisfies the following recursive conditions:

- $pred_i(v) = pred_k(v) = u$; in other words, $u \to v$ is an edge in both T_i and T_k.
- If $pred_i(u) = pred_k(u)$, then the edge $pred_i(u) \to u$ is properly shared.

- Otherwise, vertices $pred_i(u)$, v, $pred_k(u)$ are ordered clockwise around u.

We say that a properly shared edge $u \to v$ is *exposed* if $pred_i(u) \neq pred_k(u)$. For example, in Figure 2, both heavy black edges on the left are properly shared, but only the lower edge is exposed; the heavy black edges on the right are not properly shared.

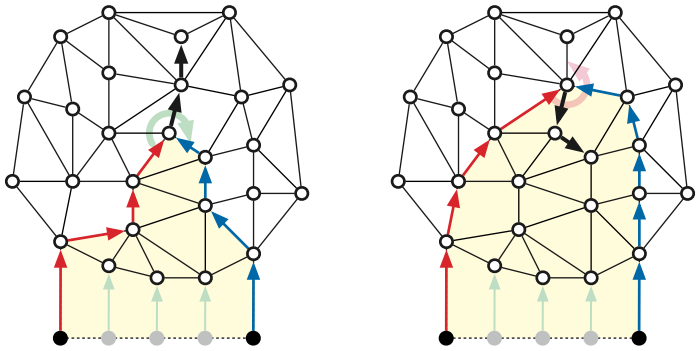

Figure 2: Shortest paths that share two edges. Left: Properly shared. Right: Improperly shared.

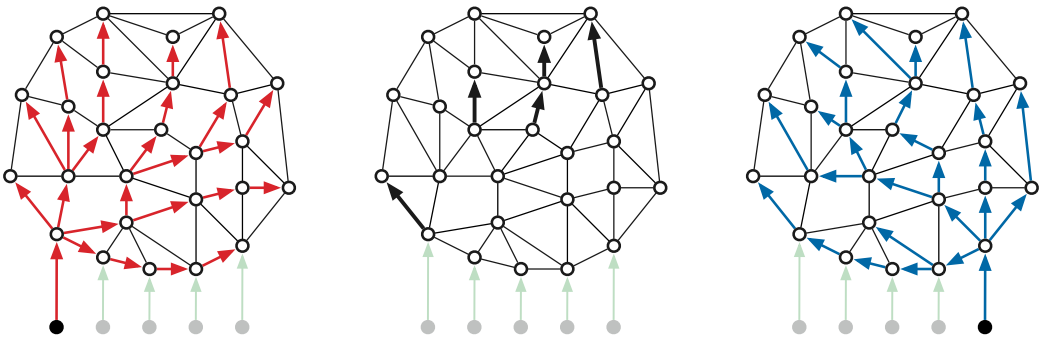

Figure 3: Two shortest path trees with five properly shared edges, four of which are exposed.

Lemma: *If $u \to v$ is properly shared by T_i and T_k, then $pred_j(v) = u$ for all $i \leq j \leq k$.*

Proof: First suppose $u \to v$ is properly shared and exposed. Let γ be a simple closed curve obtained by concatenating $path_k(u)$, the reversal of $path_i(u)$, and a simple path from s_i to s_k through the outer face. (The shaded yellow region Figure 2 is the interior of γ.) Each source vertex s_j is inside γ, and v is outside γ. So the Jordan curve theorem implies that $path_j(v)$ must cross γ. Uniqueness of shortest paths implies that $path_j(v)$ cannot cross either $path_i(v)$ or $path_k(v)$. It follows that $path_j(v)$ must contain u, and thus $pred_j(v) = u$.

Now suppose $u \to v$ is properly shared but not exposed. Let p be the first vertex on $path_i(v)$ that is also in $path_k(v)$, and let $p \to q$ be the first edge on the shortest path from p to v in H. Our recursive definitions imply that $p \to q$ is properly shared and exposed, so by the previous paragraph, for any index j, we have $pred_j(q) = p$ for all $i \leq j \leq k$. It follows that T_j contains the entire shortest path from p to v, and in particular, the edge $u \to v$. □

The converse of the previous lemma is not necessarily true; it is possible for $pred_j(v) = u$ for every index j even though $u \to v$ is not properly shared. Consider the reversed shortest path tree $\overline{T}_v$ rooted at v. Let s_l and s_r be the leftmost and rightmost source vertices in the subtree of $\overline{T}_v$ rooted at u. If this subtree contains *every* source vertex s_j, then $l = r + 1 \bmod h$; intuitively, the subtree wraps around $u \to v$ and meets itself at the boundary. See Figure 4 for an example. Edges

of this form are *not* detected by the filtering algorithm.

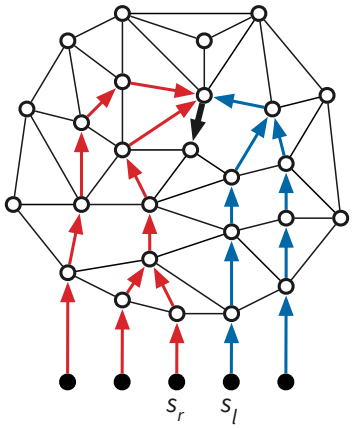

Figure 4: The black edge is shared by all shortest-path trees, but not properly shared by T_i and T_k.

Let m denote the number of vertices in H. We can identify all properly shared edges in H in $O(m)$ time using a preorder traversal of either T_i or T_k. In particular, we can find all *exposed* edges leaving vertex u in $\deg(u)$ time by visiting the darts into u in clockwise order—following the successor permutation—from $pred_i(u)\to u$ to $pred_k(u)\to u$.

13a.4 Contraction

The main work of the filtering algorithm is *contracting* properly shared edges so that they need not be passed to recursive subproblems. Intuitively, we contract the edge $u\to v$ into its tail u, changing the tail of each directed edge $v\to w$ from v to u. Here are the steps in detail:

- Set $rep(v) \leftarrow u$
- Set $off(v) \leftarrow \ell(u\to v)$
- For every edge $w\to v$:
  - Set $\ell(w\to v) \leftarrow \infty$
- For every edge $v\to w$:
  - Set $\ell(v\to w) \leftarrow off(v) + \ell(v\to w)$
  - If $pred_i(w) = v$, set $pred_i(w) \leftarrow u$
  - If $pred_k(w) = v$, set $pred_k(w) \leftarrow u$
- Contract uv to u

The actual edge-contraction (in the second-to-last step) merges the successor permutations of u and v in $O(1)$ time, as described in Lecture 10.

If u and v have any common neighbors, contracting v into u creates parallel edges, which we must resolve before passing the contracted map to MSSP-prep. After all properly shared edges are contracted, we perform a global cleanup that identifies and resolves all families of parallel edges. Specifically, for each pair of neighboring vertices u and v in the contracted map, we choose on edge e between u and v, change the dart weights of e to match the lightest darts $u\to v$ and $v\to u$, and then delete all other edges between u and v. If we use hashing to recognize and collect parallel edges, the entire cleanup phase takes linear time.[5]

[5]Efficiently maintaining a *simple* planar graph under arbitrary edge contractions is surprisingly subtle; see Holm et al [2] and Kammer and Meintrup [3]. For this MSSP algorithm, it suffices to resolve only *adjacent* parallel edges and

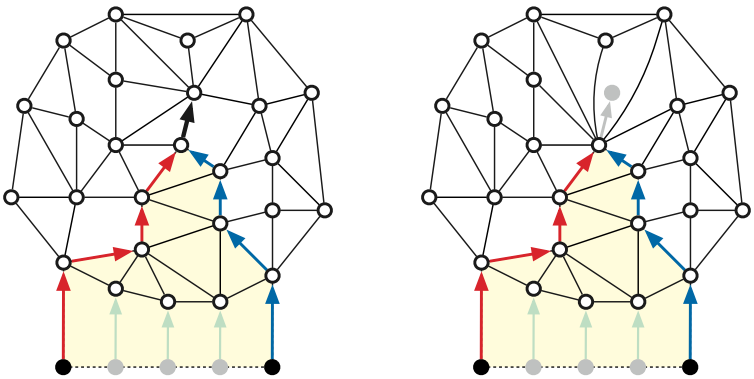

Figure 5: Contracting an exposed properly shared dart.

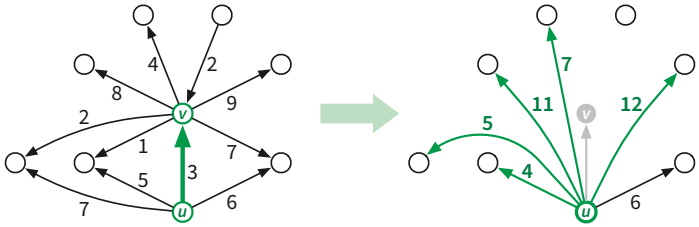

Figure 6: Edge weights before and after contraction and cleanup

Contracting $u \to v$ preserves the shortest-path distance from every source s_j to every other vertex (except the contracted vertex v). Moreover, for every source vertex s_j and every vertex w in the original map H *including* v, contraction also maintains the following invariant, which allows us to recover shortest-path distances during the query algorithm. Let $dist_j(w)$ denote the shortest-path distance from s_j to w in the original map H, and let $dist'_j(w)$ denote the corresponding distance in the current contracted map.

Key Invariant: *For every vertex w of H and for every index j such that $i \le j \le k$, we have $dist_j(w) = dist'_j(rep(w)) + off(w)$.*

When Filter begins, we have $dist_j(w) = dist'_j(w)$ and $rep(w) = w$ and $off(w) = 0$, so the Key Invariant holds trivially.

We contract properly shared edges in the same order they were discovered, following a preorder traversal of T_i. This contraction order conveniently guarantees that we only contract *exposed* edges; contracting one exposed edge $u \to v$ transforms each properly shared edge leaving v into an *exposed* properly shared edge leaving u. This contraction order also guarantees that after contracting $u \to v$, no edge into u will ever be contracted. It follows that we change the tail of each edge (and therefore the predecessors of each vertex) at most once, and the Key Invariant is maintained. We conclude:

Lemma: *Filter(H, i, k) identifies and contracts all properly shared edges in H in $O(S(m) + m)$ time, where $m = |V(H)|$. Moreover, after Filter(H, i, k) ends, the Key Invariant holds.*

delete *empty* loops immediately after each contraction in $O(1)$ time per deleted edge using only standard graph data structures. The resulting planar map is no longer necessarily simple, but every face has degree at least 3, which is good enough.

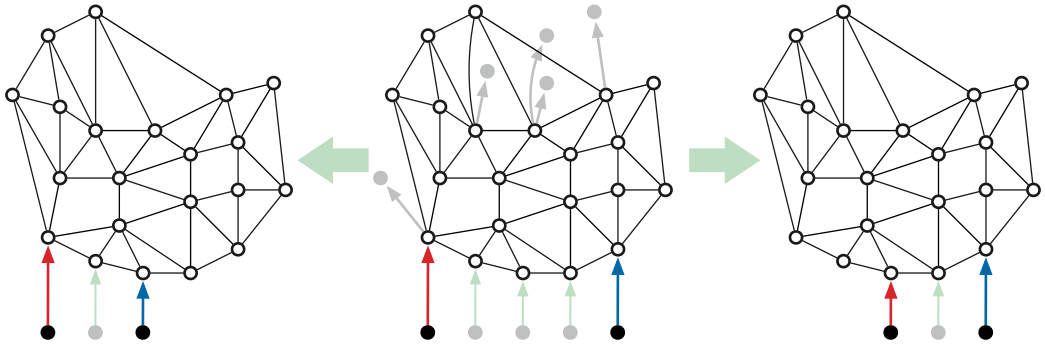

Figure 7: Contracting all properly shared directed edges and recursing.

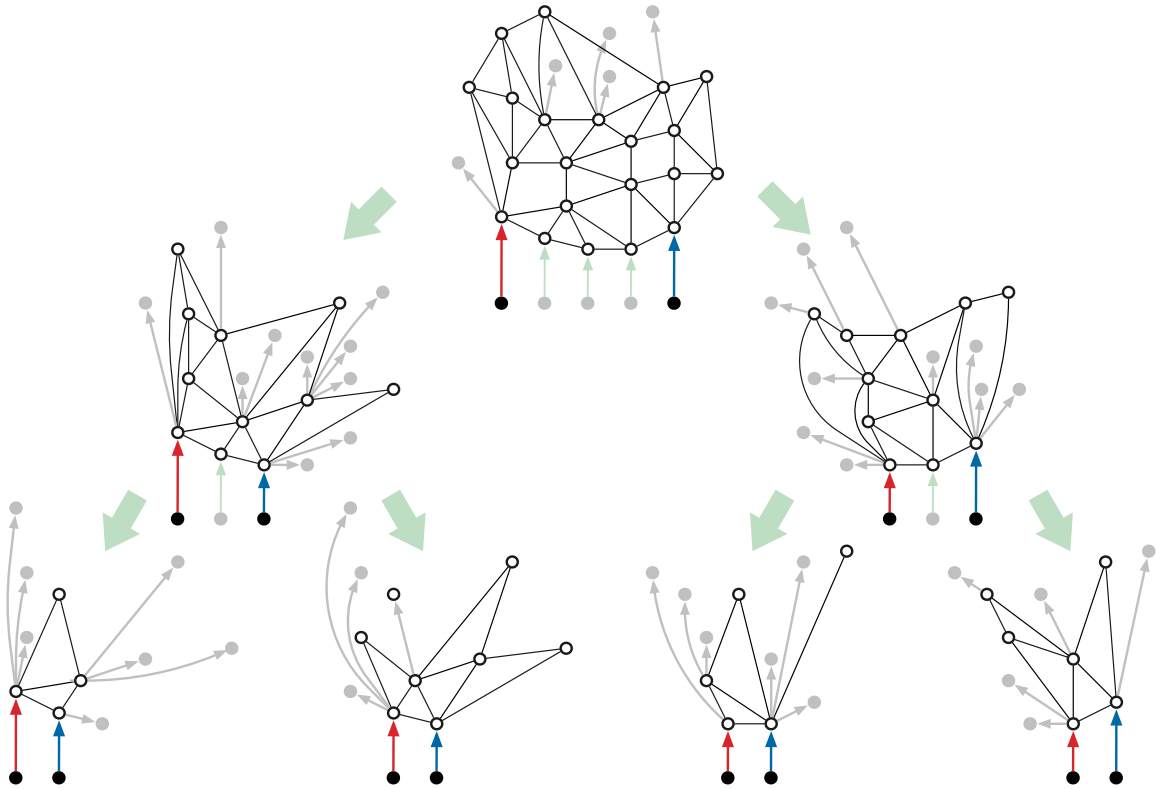

Figure 8: Recursive subproblems after contraction become more and more birdlike.

13a.5 Distance Queries

Each call to Filter(H, i, k) creates a record storing the following information:

- indices i and k
- for each vertex v of the input map H:[6]
  - shortest-path distances $dist_i(v)$ and $dist_k(v)$
  - the representative vertex $rep(v)$
  - the offset $off(v)$.

The recursive calls to MSSP-Prep assemble these records into a binary tree, mirroring the tree of recursive calls, connected by *left* and *right* pointers.

The query algorithm MSSP-query(Rec, j, v) takes as input a recursive-call record *Rec*, a source index j, and a vertex v, satisfying two conditions:

- $Rec.i \le j \le Rec.k$
- v is a vertex of the input map H to the recursive call to MSSP-Prep that created *Rec*.

The output of MSSP-query(Rec, j, v) is the shortest-path distance from s_j to v in Σ. The query algorithm follows straightforwardly from the Key Invariant:

- if $j = i$, return $Rec.dist_i[v]$
- else if $j = k$, return $Rec.dist_k[v]$
- else if $j \le Rec.left.k$, return MSSP-query$(Rec.left, j, Rec.rep[v]) + Rec.off[v]$
- else return MSSP-query$(Rec.righ, j, Rec.rep[v]) + Rec.off[v]$

Because the recursion tree has depth $O(\log h)$, the query algorithm runs in $O(\log h)$ time.

13a.6 Space and Time Analysis

It remains only to bound the size of our data structure and the running time of MSSP-Prep. The key claim is that the total size of all input maps at any level of the recursion tree is $O(n)$.

Contraction sharing lemma: *Contracting one properly shared edge neither creates nor destroys other properly shared edges.*

Proof: Fix a map H and source indices i and k. Let $u \to v$ be an edge in H that is properly shared by T_i and T_k. Let $H' = H/u \to v$, with dart weights adjusted as described above, and let T_i' and T_k' denote the shortest path trees rooted at s_i and s_k in H'.

First, because contraction preserves shortest paths, we can easily verify that $T_i' = T_i/u \to v$ and $T_k' = T_k/u \to v$. It follows that an edge in H is shared by T_i and T_k if and only if the corresponding edge in H' is shared by T_i' and T_k'.

Now consider any edge $x \to y \in T_i \cap T_k$ that is not $u \to v$. We must have $y \neq v$, because each vertex has only one predecessor in any shortest-path tree. Let w be the first node on the shortest path from s_i to x in H that is also on the shortest path from s_k to x, so the entire shortest path from w to y is shared by T_i and T_k. Consider three paths:

- α = the shortest path from s_i to w

[6] To keep the space usage low, we store this vertex information in four hash tables, each of size linear in the number of vertices of H. Alternatively, we can avoid hash tables by compacting the incidence-list structure of H' during the cleanup phase of Filter, and storing the index in the filtered map H' of each vertex of the input map H.

- β = the reverse of the shortest path from w to y
- γ = the shortest path from s_k to w

Then $x \to y$ is properly shared if and only if (the last edges of) α, β, and γ are incident to w in clockwise order. The definition of properly shared implies $v = w$, so w is also a vertex in H'. Contracting $u \to v$ might shorten one of the three paths to w, but it cannot change their cyclic order around w. We conclude that $x \to y$ is properly shared in H if and only if $x \to y$ (or $u \to y$ if $x = v$) is properly shared in H'. □

The contraction sharing lemma implies by induction that every call to $\mathsf{Filter}(H, i, k)$ outputs the same contracted map as $\mathsf{Filter}(\Sigma, i, k)$. In particular, an edge $u \to v$ in H is properly shared by two shortest-oath trees in H if and only if the corresponding edge in Σ (which may have a different tail vertex) is properly shared by the corresponding trees in Σ. So from now on, "properly shared" always implies "in the top level map Σ".

Corollary: *For all indices $i \le i' < k' \le k$, if $u \to v$ is properly shared by T_i and T_k, then $u \to v$ is properly shared by $T_{i'}$ and $T_{k'}$.*

Corollary: *The vertices of $\mathsf{Filter}(\Sigma, i, k)$ are precisely the vertices v such that no edge into v is properly shared by T_i and T_k.*

Fix any vertex v of Σ. We call an index j *interesting* if $pred_j(v) \to v$ is *not* properly shared by T_j and T_{j+1}.

Lemma: *Every vertex v of Σ has at most $\deg(v)$ indices.*

Proof: Equivalently, j is interesting to v if either of the following conditions holds:

- $pred_j(v) \neq pred_{j+1}(v)$.
- $pred_0(v) = pred_1(v) = \cdots = pred_{h-1}(v) = u$ and the paths $path_j(u)$ and $path_{j+1}(u)$ "wrap around" $u \to v$.

The Disk-Tree Lemma implies that the first condition holds for at most $\deg(v)$ indices j. If the first condition never holds (that is, if $pred_j(v)$ is the same for index j), then the second condition holds for exactly one index j; otherwise the second condition never holds. □

Lemma: *Each vertex v appears in at most $2\deg(v)$ subproblems at each level of the recursion tree.*

Proof: The children *Rec.left* and *Rec.right* of any recursion record *Rec* store information about v and if and only if at least one index j such that $Rec.i \le j \le Rec.k$ is interesting to v. □

Theorem: *MSSP-Prep$(\Sigma, 0, h-1)$ builds a data structure of size $O(n \log h)$ in $O(S(n) \log h)$ time.*

Proof: The total number of vertices in all subproblems at the same level of the recursion tree is at most $\sum_v 2\deg(v) \le 4 \cdot |E(\Sigma)| \le 4(3n-6) = 12n-24$ by Euler's formula, since Σ is a simple planar map. Each recursion record uses $O(1)$ space per vertex, so the total space used at any level is $O(n)$. Similarly, the time spent in any subproblem is at most $O(S(n)/n)$ per vertex, so the total time spent in each level of the recursion tree is $O(S(n))$.

Finally, the recursion tree has $O(\log h)$ levels. □

13a.7 References

1. Debarati Das, Evangelos Kipouridis, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. A simple algorithm for multiple-source shortest paths in planar digraphs. *Proc. 5th Symp. Simplicity in Algorithms*, 1–11, 2022.

2. David Eisenstat. *Toward Practical Planar Graph Algorithms*. Ph.D. thesis, Comput. Sci. Dept., Brown Univ., May 2014.

3. Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently. *Proc. 25th Ann. Europ. Symp. Algorithms*, 50:1–50:15, 2017. Leibniz Int. Proc. Informatics 87, Schloss Dagstuhl–Leibniz-Zentrum für Informatik. arXiv:1706.10228.

4. Frank Kammer and Johannes Meintrup. Succinct planar encoding with minor operations. Preprint, January 2023. arXiv:2301.10564.

5. Robert E. Tarjan and Renato F. Werneck. Dynamic trees in practice. *J. Exper. Algorithmics* 14:5:1–5:21, 2009.

6. Renato Werneck. *Design and Analysis of Data Structures for Dynamic Trees*. Ph.D. thesis, Dept. Comput. Sci., Princeton Univ., April 2006. Tech. Rep. TR-750-06.