Faster Shortest Paths in Dense Distance Graphs, with Applications[*]

Shay Mozes
IDC Herzliya
smozes@idc.ac.il

Yahav Nussbaum
University of Haifa
yahav.nussbaum@cs.tau.ac.il

Oren Weimann
University of Haifa
oren@cs.haifa.ac.il

Abstract

We show how to combine two techniques for efficiently computing shortest paths in directed planar graphs. The first is the linear-time shortest-path algorithm of Henzinger, Klein, Subramanian, and Rao [STOC'94]. The second is Fakcharoenphol and Rao's algorithm [FOCS'01] for emulating Dijkstra's algorithm on the *dense distance graph* (DDG). A DDG is defined for a decomposition of a planar graph G into regions of at most r vertices each, for some parameter $r < n$. The vertex set of the DDG is the set of $\Theta(nr^{-1/2})$ vertices of G that belong to more than one region (boundary vertices). The DDG has $\Theta(n)$ arcs, such that distances in the DDG are equal to the distances in G. Fakcharoenphol and Rao's implementation of Dijkstra's algorithm on the DDG (nicknamed *FR-Dijkstra*) runs in $O(n \log(n) r^{-1/2} \log r)$ time, and is a key component in many state-of-the-art planar graph algorithms for shortest paths, minimum cuts, and maximum flows. By combining these two techniques we remove the $\log n$ dependency in the running time of the shortest-path algorithm at the price of an additional $\log r$ factor, making it $O(nr^{-1/2} \log^2 r)$.

This work is part of a research agenda that aims to develop new techniques that would lead to faster, possibly linear-time, algorithms for problems in planar graphs such as minimum-cut, maximum-flow, and shortest paths with negative arc lengths. As immediate applications, we show how to compute maximum flow in directed weighted planar graphs in $O(n \log p)$ time, and minimum st-cut in undirected weighted planar graphs in $O(n \log \log p)$ time, where p is the minimum number of edges on any path from the source to the sink. We also show how to compute any part of the DDG that corresponds to a region with r vertices and k boundary vertices in $O(r \log k)$ time, which is faster than has been previously known for small values of k.

[*]The research was supported in part by Israel Science Foundation grants 794/13 and 592/17.

1 Introduction

Finding shortest paths and maximum flows are among the most fundamental optimization problems in graph theory. On planar graphs, these problems are intimately related and can all be solved in linear or near-linear time. Obtaining strictly-linear time algorithms for these problems is one of the main current goals of the planar graphs community [11,22]. Some of these problems are known to be solvable in linear time (minimum spanning tree [33], shortest-paths with non-negative lengths [20], maximum flow with unit capacities [11], undirected unweighted global min-cut [9]), but for many others, only nearly-linear time algorithms are known. These include shortest-paths with negative lengths [13, 31, 36], multiple-source shortest-paths [6, 7, 28], max-flow/min-st-cut [3–5, 12, 22], and global min-cut [8, 32, 34].

Many of the results mentioned above were achieved fairly recently, along with the development of more sophisticated shortest paths techniques in planar graphs. In this paper we show how to combine two of these techniques: the technique of Henzinger, Klein, Rao, and Subramanian for computing shortest paths with non-negative lengths in linear time [20], and the technique of Fakcharoenphol and Rao to compute shortest paths on dense distance graphs in nearly linear time in the number of vertices [13].

The *dense distance graph* (DDG) is an important element in many of the algorithms mentioned above. It is a non-planar graph that represents (exactly) distances among a subset of the vertices of the underlying planar graph G. More precisely, an r-division [14] of an n-vertex planar graph G, for some $r < n$, is a division of G (i.e., a partition of G's edge-set) into $O(n/r)$ subgraphs (called *regions*) $\{G_i\}$, where each region has at most r vertices and $O(\sqrt{r})$ *boundary* vertices (vertices that the region shares with other regions). There exist r-divisions of planar graphs with the additional property that the boundary vertices in each region lie on a constant number of faces (called *holes*) [1, 13, 30, 39]. Another property that we can obtain is that each boundary vertex belongs to a constant number of regions. This can be achieved by first ensuring that each vertex of G has constant degree (say by triangulating the dual graph with zero-length edges) and then using the algorithm of [30].

Consider an r-division of G for some $r < n$. Let K_i be the complete graph on the $O(\sqrt{r})$ boundary vertices of the region G_i, where the length of arc uv is the u-to-v distance in G_i. The graph K_i is called the DDG of G_i. The union $\bigcup_i K_i$ is called the DDG of G (or more precisely, the DDG of the given r-division of G).[1]

DDGs are useful for three main reasons. First, distances in the DDG of G are the same as distances in G. Second, it is possible (FR-Dijkstra [13]) to compute shortest paths in DDGs in time that is nearly linear in the number of vertices of the DDG (i.e., in sublinear time in the number of vertices of G). Finally, DDGs can be computed in nearly linear time either by invoking FR-Dijkstra recursively, or by using a multiple source shortest-path algorithm [7, 28] (MSSP). Until the current work, the latter method was faster in all cases.

Since it was introduced in 2001, FR-Dijkstra has been used creatively as a subroutine in many algorithms. These include algorithms for shortest paths with negative lengths [13], maximum flows and minimum cuts [4, 5, 22, 32], distance oracles [6, 13, 24, 28, 35, 37], and DDG constructions [13]. Improving FR-Dijkstra is therefore an important task with implications to all these problems. For example, consider the minimum st-cut problem in undirected planar graphs. Italiano et. al [22] gave an $O(n \log \log n)$ algorithm for the problem, improving the $O(n \log n)$ algorithms of Frederickson [14]

[1]DDGs are similarly defined for decompositions which are not r-divisions, but our algorithm does not necessarily apply in such cases.

and of Reif [38] (using [20]). Three of the techniques used by Italiano et al. are: (i) constructing an r-division with $r = \text{polylog}(n)$ in $O(n \log r)$ time, (ii) FR-Dijkstra, and (iii) constructing the DDG in $O(n \log r)$ time. In a step towards a linear time algorithm for this fundamental problem, Arge et al. [1], and independently Klein et al [30] gave an $O(n)$ algorithm for constructing an r–division with few holes.[2] This leaves the construction of the DDG as the only current bottleneck for obtaining min-st-cut in $O(n)$ time. The work described in the current paper is motivated by the desire to obtain such a linear time algorithm, and partially addresses techniques (ii) and (iii). It improves the running time of FR-Dijkstra, and, as a consequence, implies faster DDG construction, although for a limited case which is not the one required in [22].

The linear time algorithm for shortest-paths with nonnegative lengths in planar graphs of Henzinger, Klein, Rao, and Subramanian [20] (HKRS) is another landmark result that has been used in many subsequent algorithms. HKRS [20] differs from Dijkstra's algorithm in that the order in which it relaxes the arcs is not determined by the vertex with the current globally minimum label. Instead, it works with a recursive division of the input graph into regions. It handles a region for a limited time period, and then skips to other regions. Within this time period it determines the order of relaxations according to the vertex with minimum label in the *current* region, thus avoiding the need to perform many operations on a large heap. Planarity, or more precisely the existence of small recursive separators, guarantees that local relaxations have limited global effects. Therefore, even though some arcs are relaxed by HKRS more than once, the overall running time can be shown (by a fairly complicated argument) to be $O(n)$. Even though HKRS has been introduced roughly 20 years ago and has been used in many other algorithms, to the best of our knowledge, and unlike other important planarity exploiting techniques, it has always been used as a black box, and was not modified or extended prior to the current work.[3]

1.1 Our results

By combining the framework of Henzinger et al. (HKRS) with a modification of the internal building blocks of Fakcharoenphol and Rao's Dijkstra implementation (FR-Dijkstra), we obtain a faster algorithm for computing shortest paths on dense distance graphs. On the conceptual level, this work is the first to suggest and achieve a combination of these two powerful techniques. Furthermore, it is the first to improve the internal building blocks of FR-Dijkstra, and the first to demonstrate a different use for the framework of HKRS.

Specifically, for a DDG over an r–division of an n-vertex graph, FR-Dijkstra runs in $O(n \log(n) r^{-1/2} \log r)$. We remove the logarithmic dependency on n, and present an algorithm whose running time is $O(nr^{-1/2} \log^2 r)$. This is the first asymptotic improvement over FR-Dijkstra. The improvement is useful in algorithms that use an r-division when r is small (say $r = \text{polylog}(n)$). When r is large (say $r = \text{poly}(n)$) our algorithm has the same complexity as FR-Dijkstra. However, for this case, Gawrychowski and Karczmarz [16] recently presented a followup on our work that achieves a running time of $O(nr^{-1/2} \log^2(n)/ \log^2(\log n))$.

[2]The algorithm in [30] produces a recursive r-division with few holes (i.e., one in which each region is further divided recursively with an r–division until constant sized regions are achieved) in linear time, whereas the one in [1] does not. A recursive r-division is used in this paper and may be essential for obtaining linear time r–division based algorithms.

[3]Tazari and Müller-Hannemann [40] extended [20] to minor-closed graph classes, but that extension uses the algorithm of [20] without change. It deals with the issue of achieving constant degree in minor-closed classes of graphs, which was overlooked in [20].

Our overall algorithm resembles that of HKRS [20]. However, in our algorithm, the bottom level regions are not individual edges (as in HKRS), but hyperedges, which are implemented by a suitably modified version of *bipartite Monge heaps* (the main workhorse of FR-Dijkstra [13]). One of the main challenges in combining the two techniques is that the efficiency of FR-Dijkstra critically relies on the order of relaxations performed by Dijkstra's algorithm, whereas HKRS does not implement Dijkstra's algorithm. In particular, in Dijkstra's algorithm, once a vertex is handled (i.e., extracted from the heap), its distance will never change again. In HKRS this is far from being true. The distance label of a vertex may decrease after it has been handled, causing it to be handled multiple times. A new idea is required to allow the specialized data structure of FR to be used in an algorithm like HKRS. We develop a new kind of *dynamic* Range Minimum Query (RMQ) data structure for Monge matrices (see Section 2.1 for the definition). This data structure can be used to implement some of the functionality of Fakcharoenphol and Rao's bipartite Monge heaps. Our dynamic RMQ data structure is agnostic to the order of relaxations. It allows us to deactivate a vertex when it is handled, and reactivate it if its distance label subsequently decreases.

Another difficulty is that the HKRS algorithm requires accurate distance labels as it shifts its attention between different regions. This is achieved in HKRS by performing explicit relaxations. In our case, however, the relaxations are performed implicitly, causing limited availability of explicit and accurate distance labels. We use a new auxiliary construction and careful coordination in the implementation of the HKRS algorithm to resolve this conflict.

Finally, the analysis of HKRS has to be modified. First, it must be adapted to work with hyperedges. Second, the choice of region sizes in the recursive r–division needs to be carefully tightened. This is because the analysis of HKRS proved linear running-time in the number of arcs of the planar graph G, which is $O(n)$. In our case the number of arcs in the DDG is still $O(n)$, but we want linear running-time in the number of vertices of the DDG, which is only $O(n/\sqrt{r})$.

1.2 Applications

We believe that our fast shortest-path algorithm on dense distance graphs is a step towards optimal algorithms for the important and fundamental problems mentioned in the introduction. In addition, we describe three current applications of our improvement. In these applications we obtain a speedup over previous algorithms by decomposing a region of n vertices using an r-division, and computing distances among the boundary vertices of the region in $O((n/\sqrt{r})\log^2 r)$ time using our fast shortest-path algorithm.

Directed Maximum Flow. In *directed* weighted planar graphs, we show how to compute maximum st-flow (and min-st-cut) in $O(n \log p)$ time if there is some path from s to t with at most p vertices. The parameter p appears in the time bounds of several previous maximum flow and minimum cut algorithms [2,21,23,26]. Our $O(n \log p)$ bound matches the fastest previously known maximum flow algorithms in directed weighted planar graphs for $p = \Theta(1)$ [23] and for $\log p = \Theta(\log n)$ [3,12], and is asymptotically faster than previous algorithms for all other values of p. See Section 5.2.

Undirected Minimum st-Cut. In *undirected* weighted planar graphs, we show how to compute minimum st-cut in $O(n \log\log p)$ time if there is some path from s to t with at most p vertices. This improves the fastest previous algorithms of $O(n \log p)$ [26] and $O(n \log\log n)$ [22] for small values of p. See Section 5.3.

Fast construction of DDGs with few boundary vertices. The current bottleneck in various shortest paths and maximum flow algorithms in planar graphs (e.g., min-st-cut in an undirected graph [22], shortest paths with negative lengths [36]) is computing all boundary-to-boundary distances in a graph with n vertices and $k = O(\sqrt{n})$ boundary vertices that lie on a single face. Currently, the fastest way to compute these k^2 distances is to use the MSSP algorithm [7, 28]. After $O(n \log n)$ preprocessing, it can report the distance from any boundary vertex to any other vertex (boundary or not) in $O(\log n)$ time, so all boundary-to-boundary distances can be found in $O((n+k^2)\log n) = O(n \log n)$ time. We give an algorithm that computes the distances among the k boundary vertices in $O(n \log k)$ time. This algorithm can be used to construct a DDG of a region with n vertices and $k = O(\sqrt{n})$ boundary vertices in $O(n \log k)$ time. In general, this does not improve the $O(n \log n)$ DDG construction time using MSSP since typically $k = \Theta(\sqrt{n})$. However, there is an improvement when k is much smaller. For $k = \text{polylog}(n)$, our algorithm takes $O(n \log\log n)$ time. See Section 5.1.

We conclude this section by discussing the interesting open problem of computing a DDG of a region with n vertices of which $k = O(\sqrt{n})$ are boundary vertices. As we already mentioned, the conventional way of computing a DDG is applying Klein's MSSP algorithm [12, 28], which requires $O(n \log n)$ time regardless of the value of k. The MSSP algorithm implicitly computes all the shortest-path trees rooted at vertices of a face ϕ by going around the face ϕ and for every boundary vertex v of ϕ identifying the *pivots* (arc changes) from the tree rooted at the vertex preceding v on ϕ to the tree rooted at v. Eisenstat and Klein [11] showed an $\Omega(n \log k)$ lower bound on the number of comparisons that any MSSP algorithm requires for identifying all the pivots.[4] Our algorithm can be used to compute the DDG of a region in $O(n \log k)$ time without computing all the pivots. Note, however, that it does not break the $\Omega(n \log k)$ lower bound. The DDG computation problem may be an easier problem than the MSSP problem, since we are interested only in the $O(k^2)$ distances among the boundary vertices and we are not required to compute the pivots. Whether the DDG of a region can be computed in $o(n \log k)$ remains an open problem.

1.3 Roadmap

We begin with some preliminaries. In Section 2.1 we describe Monge matrices and their basic properties. We continue in Section 2.2 with a description of Fakcharoenphol and Rao's FR-Dijkstra algorithm and the Monge heap data structure. This description is essential since we modify the internal structure of the Monge heaps to achieve our results. In Section 3 we give a warmup improvement of FR-Dijkstra by introducing a new RMQ data structure into the Monge heaps. This improvement decouples the logarithmic dependency on n from the logarithmic dependency on r. In Section 4 we present our main result that eliminates the logarithmic dependency on n altogether by combining the shortest-path algorithm of Henzinger et al. with modified Monge heaps similar to those described in the warmup. Finally, the applications of our algorithm to directed maximum flow, undirected minimum st-cut, and DDG computation are described in Section 5.

2 Preliminaries

2.1 Monge matrices

A matrix M is *Monge* if for any pair of rows $i < j$ and columns $k < \ell$ we have that $M_{ik} + M_{j\ell} \geq M_{i\ell} + M_{jk}$. A *partial* Monge matrix is a Monge matrix with some undefined entries, such that the

[4] In fact they showed an $\Omega(n \log n)$ lower bound by using a face ϕ with $\sqrt{n}$ boundary vertices; generalizing the same proof for a face with k vertices gives the $\Omega(n \log k)$ bound.

defined entries in each row and in each column are contiguous. It is well known (cf. [31]) that the upper and lower triangles of the (weighted) adjacency matrix M of K_i are *partial* Monge matrices.[5]

Consider a Monge martix M with m rows and n columns, where $m \leq n$. Denote $r(j) = i$ if the minimum element of M in column j lies in row i. The *upper envelope* $\mathcal{E}$ of all the rows of the matrix M consists of the n values $r(1), \ldots, r(n)$. An immediate consequence of the Monge property is that $r(1) \leq \ldots \leq r(n)$, and so $\mathcal{E}$ can be implicitly represented in $O(m)$ space by storing only the $r(j)$'s of $O(m)$ columns called *breakpoints*. Breakpoints are the columns j where $r(j) \neq r(j+1)$. The minimum element $r(\pi)$ of an entire column π can then be retrieved in $O(\log m)$ time by a binary search for the first breakpoint column j after π, and setting $r(\pi) = r(j)$.

2.2 FR-Dijkstra [13]

We now describe FR-Dijkstra, which is an emulation of Dijkstra's algorithm on the dense distance graph. Parts of our description deviate from the original description of [13] and were adapted from [29]. We emphasize that this description is not new, and is provided as a basis for the modifications introduced in subsequent sections. Recall Dijkstra's algorithm. It maintains a heap of vertices labeled by estimates $d(\cdot)$ of their distances from the root. At each iteration, the vertex with minimum $d(\cdot)$ is ACTIVATED; It is extracted from the heap, and all its adjacent arcs are relaxed. FR-Dijkstra implements EXTRACTMIN and ACTIVATE efficiently.

Recall that the DDG is the union of complete graphs $\{K_i\}$. The vertices of each K_i correspond to the $O(\sqrt{r})$ boundary vertices of a region of an r–division with a constant number of holes. We assume in our description that each region has a single hole. There is a standard way to handle a constant number of holes using the single hole case with only a constant factor overhead (cf. [13, Section 5], [24, Section 5.2], and [36, Section 4.4]). There is a natural cyclic order on the vertices of K_i according to their order on the single face (hole) of the corresponding region.

To implement EXTRACTMIN and ACTIVATE efficiently, each complete graph K_i in the DDG is decomposed into complete bipartite graphs. The advantage of working with bipartite graphs is that their weighted incidence matrix is Monge (not just partial Monge). The vertices of K_i are first split into two consecutive halves A and B, the complete bipartite graph on A and B is added to the decomposition, and the same process is applied recursively on A and on B. Therefore, the total number of bipartite subgraphs in the decomposition of K_i is $\sqrt{r}$. Each vertex of K_i appears in $O(\log r)$ bipartite subgraphs. See Figure 1.

[5]The order of the rows and columns in M is defined by ordering the boundary vertices of K_i in a clockwise or counterclockwise manner.

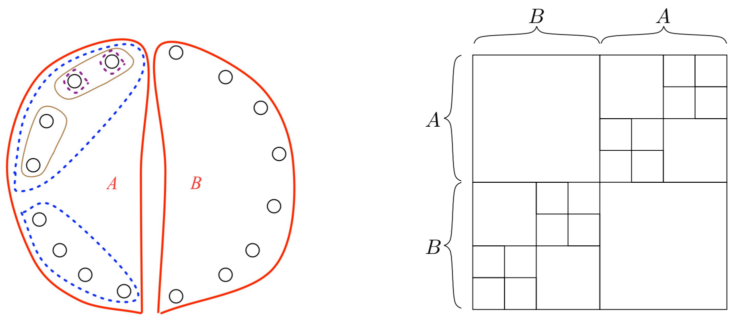

Figure 1: The decomposition of K_i into complete bipartite graphs (left) can also be viewed as a partition of the incidence matrix of K_i into Monge matrices (right).

Let $\mathcal{H} = \{H_j\}$ denote the set of all bipartite graphs in the decompositions of all K_i's. Note that $|\mathcal{H}| = O(n/\sqrt{r})$. The algorithm maintains a data-structure $\mathcal{M}_j$, called a *Monge Heap*, for each bipartite graph $H_j \in \mathcal{H}$.[6] Let A, B be the bipartition of H_j's vertices. The Monge heap supports:

- Activate(a, d) - Sets the label of vertex $a \in A$ to be d, and implicitly relaxes all outgoing arcs from a. This operation may be called at most once per vertex.
- FindMin - Returns the vertex $v \in B$ with minimum label among vertices not yet extracted.
- ExtractMin - Removes the vertex $v \in B$ with minimum label among vertices not yet extracted.

The minimum elements of all the Monge heaps are maintained in a regular global heap $\mathcal{Q}$. In each iteration of FR-Dijkstra, a vertex v with global minimum label is extracted from $\mathcal{Q}$. The vertex v is then extracted from the Monge heap $\mathcal{M}_j$ that contributed v, and the new minimum vertex of $\mathcal{M}_j$ is added to $\mathcal{Q}$. Note that since a vertex v appears in multiple H_j's, v may be extracted as the minimum element of the global heap $\mathcal{Q}$ multiple times, once for each Monge heap it appears in. However, the label $d(v)$ of v is finalized at the first time v is extracted. At that time, and only at that time, the algorithm marks v as finalized, activates v using Activate in *all* Monge heaps such that $v \in A$, and updates the representatives of those Monge heaps in $\mathcal{Q}$.

Analysis of FR-Dijkstra. Since each vertex appears in $O(\log r)$ bipartite graphs H_j in $\mathcal{H}$, the number of times each vertex is extracted from the global heap $\mathcal{Q}$ is $O(\log r)$. Since $\mathcal{Q}$ contains one representative element from each Monge heap $\mathcal{M}_j$, extracting the minimum element from $\mathcal{Q}$ takes $O(\log(n/\sqrt{r})) = O(\log n)$ time. Therefore, the total time spent on extracting vertices from $\mathcal{Q}$ is $O((n/\sqrt{r}) \log n \log r)$.

As for the cost of operations on the Monge heaps, Activate and ExtractMin are called at most once per vertex in each Monge heap, and the number of calls to FindMin is bounded by the number of calls to Activate and ExtractMin. We next show how to implement each of these operations in $O(\log r)$ time. Since each vertex appears in $O(\log r)$ Monge heaps, the total time spent on operations on the Monge heaps is $O((n/\sqrt{r}) \log^2 r)$.

[6] In [13] these are called bipartite Monge heaps. All the bipartite Monge heaps that belong to the same complete graph K_i are aggregated into a single data structure which [13] calls Monge heap. We do not use this aggregation.

Implementing Monge heaps. Let $\mathcal{M}$ be the Monge heap of a bipartite subgraph with columns A and B and a corresponding incidence matrix M. For every vertex $v \in B$ we maintain a bit indicating whether v has already been extracted from $\mathcal{M}$ (i.e., finalized) or not.

The distance label of a vertex $v \in B$ is defined to be $d(v) = \min_{u \in A}\{d(u)+M_{uv}\}$ and is not stored explicitly. Instead, we say that $u \in A$ is the *parent* of $v \in B$ if $d(u)$ is finite and $d(v) = d(u) + M_{uv}$. As the algorithm progresses, the distance labels $d(u)$ of vertices $u \in A$ (hence the parents of vertices $v \in B$) may change. We maintain a binary search tree T of triplets $(a \in A, b_1 \in B, b_2 \in B)$ indicating that a is the current parent of all vertices of B between b_1 and b_2. Note that a vertex $a \in A$ may appear in more than one triplet because, after extracting vertices of B, the non-extracted vertices of which a is a parent might consist of several intervals. The Monge property of M guarantees that if a precedes a' in A then all intervals of a precede all intervals of a'.

Finally, the Monge heap structure also consists of a standard heap Q_B containing, for every triplet $(a, b_1, b_2) \in T$, a vertex b between b_1 and b_2 that minimizes $d(b) = d(a) + M_{ab}$.

The operations on a Monge heap are implemented as follows:

- FINDMIN: Return a vertex b with minimum distance label in Q_B.
- EXTRACTMIN: Extract the vertex b with minimum distance label from Q_B and mark b as extracted. Find the (unique) triplet (a, b_1, b_2) containing b in T. Let b' and b'' be the members of B that precede and follow b, respectively, within this triplet. The algorithm replaces the triplet (a, b_1, b_2) with two triplets (a, b_1, b') and (a, b'', b_2) (if these intervals are defined). In each of these new triplets it finds the vertex b^* that minimizes $d(b^*) = d(a) + M_{ab^*}$ and inserts b^* into Q_B. The vertex b^* is the minimum entry of M in a given row and a range of columns. It is found in $O(\log r)$ time using a standard 1-dimensional static RMQ data structure[7] on each column of M.
- ACTIVATE(a, d): Set $d(a) = d$ and find the children of a in B. If a is the first vertex in the Monge heap for which this operation is applied then all the vertices of B are children of a. Otherwise, we show how to find the children of a whose current parent precedes a in A (the ones whose current parent follows a in A are found symmetrically).

  We traverse the triplets in T one by one backwards starting with triplet $t = (w, f_1, f_2)$ such that w precedes a in A. We continue until we reach a triplet $t' = (u, b_1, b_2)$ such that $d(u) + M_{ub_2} < d(a) + M_{ab_2}$. If $t' = t$ we do nothing. If we scanned all triplets preceding t without finding t' then the first child of a must belong to the last triplet $t'' = (v, g_1, g_2)$ that we scanned. It is found by a binary search on the interval of B between g_1 and g_2. Otherwise, the first child of a must belong to the triplet $t'' = (v, g_1, g_2)$ following t', and can again be found via binary search. Let x (resp., y) be the first (resp., last) child of a in B, as obtained in the preceding step. Note that there are no extracted vertices between x and y in B. This is because we find the distances in monotonically increasing order, and when we extract a vertex b it is the minimum in the global heap $\mathcal{Q}$ so it will never acquire a new parent. We therefore insert a new triplet (a, x, y) into T. We remove from T all other triplets containing vertices between x and y, and remove from Q_B the elements contributed by these triplets. Let (u, b_1, b_2) be the removed triplet that contains x. If $x \neq b_1$ then we insert a new triplet (u, b_1, z_1) where z_1 is the vertex preceding x in B. Similarly, if (w, b'_1, b'_2) is the removed triplet that contains y and $y \neq b'_2$ then we insert a new triplet (w, z_2, b'_2) where z_2 is the vertex following y in B. Finally, we update the three values that the new triplets $(a, x, y), (u, b_1, z_1)$, and (w, z_2, b'_2) contribute to Q_B. We find these values by a range minimum query to the RMQ data structure.

[7] Any balanced search tree can be used for such RMQ data structure.

Analysis of Monge heaps. Clearly, FINDMIN takes $O(1)$ time. Both EXTRACTMIN and ACTIVATE insert a constant number of new triplets to T in $O(\log r)$ time, make a constant number of range-minimum queries in $O(\log r)$ time, and update the representatives of the new triplets in the heap Q_B in $O(\log r)$ time. ACTIVATE(a, d), however, may traverse many triplets to identify the children of a. Since all except at most two of the triplets that it traverses are removed, we can charge their traversal and removal to their insertion in a previous EXTRACTMIN or ACTIVATE.

3 A Warmup Improvement of FR-Dijkstra

In this section we show how to modify FR-Dijkstra from $O((n/\sqrt{r})(\log n \log r))$ to $O((n/\sqrt{r})(\log n + \log^2 r \log\log r))$. This improves FR-Dijkstra for small values of r and is obtained by avoiding the vertex copies in the global heap $\mathcal{Q}$ using a decremental RMQ data structure.

Recall that, at any given time, the global heap $\mathcal{Q}$ of FR-Dijkstra maintains $O(n/\sqrt{r})$ items – one item for each Monge heap. However, each vertex has copies in $O(\log r)$ Monge heaps so overall $O((n/\sqrt{r}) \log r)$ items are extracted from $\mathcal{Q}$. Each extraction takes $O(\log(n/\sqrt{r})) = O(\log n)$ time so the complexity of FR-Dijkstra is $O((n/\sqrt{r}) \log r \log n)$. In our modified algorithm the global heap $\mathcal{Q}$ contains, for each vertex v, a single item whose key is the minimum label over all copies of v. In other words, $\mathcal{Q}$ maintains a total of $O(n/\sqrt{r})$ items throughout the entire execution. An item that corresponds to a vertex v is extracted from $\mathcal{Q}$ only once. The extraction takes $O(\log n)$ time, but may cause $O(\log r)$ calls to DECREASEKEY. Using a Fibonacci heap [15] for $\mathcal{Q}$, DECREASEKEY only takes $O(1)$ time. Note that the total number of operations on $\mathcal{Q}$ is still $O((n/\sqrt{r}) \log r)$.

The main problem with having one copy is that a triplet (a, x, y) in a Monge heap might now contain extracted vertices between x and y in B. The original implementation of FR-Dijkstra uses an elementary RMQ data structure (a binary search tree for each row of the $\sqrt{r} \times \sqrt{r}$ matrix M). This data structure is only queried on intervals $[x, y]$ that have no extracted vertices. To cope with query intervals $[x, y]$ that have extracted vertices we present a *dynamic* RMQ data structure that can handle extractions:

Lemma 1. *Given an $n \times n$ partial Monge matrix M, one can construct in $O(n \log n)$ time a dynamic data structure that supports the following operations: (1) in $O(\log^3 n)$ time, set all entries of a given column as inactive (2) in $O(\log^3 n)$ time, set all entries of a given column as active (3) in $O(\log^2 n)$ time, report the minimum active entry in a query row and a contiguous range of columns.*
For a decremental data structure in which only operations 1 and 3 are allowed, operation 1 can be supported in amortized $O(\log^2 n \log\log n)$ time and operation 3 in worst case $O(\log n \log\log n)$ time.

The proof of Lemma 1 appears below. Note that we stated the above lemma for partial Monge matrices and not full Monge matrices. This is because we apply the RMQ data structure to the upper and lower triangles of the $\sqrt{r} \times \sqrt{r}$ incidence matrix M of K_i (which are partial Monge) as opposed to FR-Dijkstra that uses a separate RMQ for each bipartite subgraph (whose corresponding submatrix is full Monge) of K_i's decomposition. In this section we only deactivate columns of M and never activate columns. Therefore, we use the second data structure in Lemma 1. The first data structure in Lemma 1 is used in Section 4. We now prove Lemma 1.

Proof. It is known that the undefined entries in a partial Monge matrix M can be implicitly replaced so that M becomes fully Monge and each entry M_{ij} can be returned in $O(1)$ time. Such procedure was first shown in [18] but the proof had an error that was fixed in [17]. Note that this procedure does not preserve row minima, as the implicitly replaced entires may be small. Therefore, such a transformation is useless when dealing with (entire) row minimum queries in partial Monge matrices. However, we consider row minimum queries restricted to a contiguous range of columns that does not include any replaced entries. Therefore, we may assume that M is fully Monge. To make the presentation clear (and compatible with [24]), we prove the lemma on the transpose of M (i.e., we activate and deactivate rows and a query is a column and a range of rows). Furthermore, although the lemma states that M is an $n \times n$ matrix, we consider an $m \times n$ matrix for some $m \leq n$.

The static data structure of [24]. We first review the static data structure of [24], and then modify it to support row activation and deactivation.

The tree T_h introduced in [24] is a full binary tree T_h whose leaves are the rows of M. A node u whose subtree contains k leaves (i.e., k rows) stores the $O(k)$ breakpoints of the $k \times n$ matrix M_u defined by these k rows and all n columns of M (each breakpoint also stores its appropriate row index). A leaf represents a single row and requires no computation. For an internal node u, we compute its list of breakpoints by merging the breakpoint lists of its left child u_ℓ and its right child u_r, where u_ℓ is the child whose rows have lower indices.

By the Monge property, u's list of breakpoints starts with a prefix of u_ℓ's breakpoints and ends with a suffix of u_r's breakpoints. Between these there is possibly one new breakpoint j (the *transition* breakpoint). The prefix and suffix parts can be found easily in $O(k)$ time by linearly comparing the relative order of the upper envelopes of M_{u_ℓ} and M_{u_r}. This allows us to find an interval of columns containing the transition column j. Over this interval, the transition row is in one of two rows (a row in M_{u_ℓ} and a row in M_{u_r}), and so j can be found in $O(\log n)$ time via binary search. Summing $O(k + \log n)$ over all nodes of T_h gives $O(n \log n)$ time. The total size of T_h is $O(m \log m)$. A query to T_h is performed as follows.

The minimum entry in a query column π and a contiguous range of rows R is found by identifying $O(\log m)$ *canonical nodes* of T_h. A node u is canonical if u's set of rows is contained in R but the set of rows of u's parent is not. For each such canonical node u, the minimum element in column π amongst all the rows of u is found in $O(\log m)$ time via binary search on u's list of breakpoints. The output is the smallest of these. The total query time is thus $O(\log^2 m)$.

A dynamic data structure. The above T_h data structure was used in [24] in a static version, which allowed the query time to be improved from $O(\log^2 m)$ to $O(\log m)$ using fractional cascading [10]. In order to allow activating/deactivating of a row we slightly modify this data structure and do not use fractional cascading. Instead of storing all its breakpoints, every node u of T_h will now only store its single transition breakpoint. In order to find the minimum element in column π of M_u, we need to find the predecessor breakpoint of π in M_u. This is done in $O(\log m)$ time by following a path in u's subtree starting from u and ending in a leaf. At each node v on this path we decide whether to visit v's left or right child according to whether π is smaller than v's transition breakpoint or not. Given a query column and a contiguous range of rows we again identify $O(\log m)$ canonical nodes. Each of them now requires following a path of length $O(\log m)$ for a total of $O(\log^2 m)$ time.

In order to deactivate an entire row r, we mark this row's leaf in T_h as inactive and then fix the breakpoints bottom-up on the entire path from r to the root of T_h. When we handle a node u on this path, the transition breakpoints for all descendants of u are correct and we find the new transition breakpoint j of u as follows: If u has left child u_ℓ and right child u_r then j should be placed between a breakpoint j_ℓ of M_{u_ℓ} and a breakpoint j_r of M_{u_r}. Here, M_{u_ℓ} and M_{u_r} correspond to the *active* rows in u_ℓ's and u_r's subtrees. If there are no active rows in u_ℓ's or u_r's subtrees then finding j is trivial. If both have active rows then we now show how to find j_ℓ, finding j_r is done similarly.

To find j_ℓ, we follow a path from u_ℓ downwards (i.e., toward the leaves) until we reach a leaf or a node whose subtree contains no active rows as follows. Recall that the node u_ℓ stores a single breakpoint (a row index i' and a column index j'). Since (i', j') is the minimum entry in column j' of M_{u_ℓ} we want to compare it with (i'', j') = the minimum entry in column j' of M_{u_r}. If $M[i', j'] < M[i'', j']$ then we know that $j > j'$ and so we can proceed from u_ℓ to u_ℓ's right child. Similarly, if $M[i', j'] \geq M[i'', j']$ then $j \leq j'$ and we proceed from u_ℓ to u_ℓ's left child. The problem is we do not know i''. To find it, we need to find the minimum entry in column j' of M_{u_r}. We have already seen how to do this in $O(\log m)$ time by following a downward path from u_r. We therefore get that finding j_ℓ (and similarly j_r) can be done in $O(\log^2 m)$ time. After finding j_ℓ and j_r we find where j is between them in $O(\log n)$ time via binary search. The overall time for deactivating an entire row r is thus $O(\log^3 m + \log m \log n) = O(\log^3 n)$.

In order to activate an entire row r we mark its leaf in T_h as active and fix the breakpoints from r upwards using a similar process as above. This completes the proof of the dynamic data structure.

A decremental data structure. We now proceed to proving the decremental data structure in which inactive rows never become active. We take advantage of this fact by maintaining, for each node u of T_h, not only its transition breakpoint but all the active breakpoints of M_u. Since breakpoints are integers bounded by n, we can maintain u's k active breakpoints in an $O(k)$-space $O(\log \log n)$ amortized query-time predecessor data structure such as Y-fast-tries [41]. Given a query column and a contiguous range of rows, we identify $O(\log m)$ canonical nodes in T_h and then only need to do a predecessor search on each of them in total $O(\log m \log \log n)$ time.

In order to deactivate a row r, we mark r's leaf in T_h as inactive and fix the predecessor data structures of r's ancestors. For each node u on the r-to-root path we may need to change u's transition breakpoint as well as insert (possibly many) new breakpoints into u's predecessor data structure Y_u. We show how to do this assuming u's left child u_ℓ is the ancestor of r and we've already fixed its predecessor data structure Y_ℓ. The case where u's right child u_r is the ancestor of r is handled symmetrically. Note that, if r is a descendent of u_ℓ, then the transition breakpoint and the predecessor data structure Y_r of u_r undergo no changes.

We first check whether u's old transition breakpoint has row index r. If not then we don't need to change u's transition breakpoint. If it is, then we delete u's old transition breakpoint from Y_u and we find the new transition breakpoint j of u via binary search. The binary search takes $O(\log n \log \log n)$ time since each comparison requires to query Y_ℓ and Y_r in $O(\log \log n)$ time. After finding the new transition breakpoint j we insert it into Y_u. Finally, we need to insert to Y_u (possibly many) new breakpoints, each in $O(\log \log n)$ time. These breakpoints are precisely all the breakpoints in Y_ℓ that are to the left of j and to the right of predecessor(j) in Y_u as well as all the breakpoints in Y_r that are to the right of j and to the left of successor(j) in Y_u.

We claim that the above procedure requires $O(\log^2 n \log \log n)$ amortized time. First note that finding the new transition breakpoints of all $O(\log m)$ nodes on the r-to-root path takes $O(\log m \log n \log \log n)$ time. Inserting these new breakpoints into the predecessor data structures and deleting the old one takes $O(\log m \log \log n)$ time. Next, we need to bound the overall number of breakpoints that are ever inserted into predecessor data structures. In the beginning, all rows are active and the total number of breakpoints in all predecessor data structures is $O(m \log m)$. Then, whenever we deactivate a row, each node u on the leaf-to-root path in T_h may insert a new transition breakpoint j into Y_u. This breakpoint j might consequently be inserted into every Y_v where v is an ancestor of u. Let $top(j)$ denote the depth of the highest ancestor of u to which we will consequently insert the breakpoint j (the depth of the root is 0). Note that we might also insert to Y_u other breakpoints but that only means that their *top* value would decrease. Since every *top* value is at most $O(\log m)$ and since top values only decrease (there are only deactivations, no activations), this means that over all the m row deactivations only $O(m \log^2 m)$ breakpoints are ever inserted. This means that all insertions take $O(m \log^2 m \log\log n)$ time. □

3.1 Modifying FR-Dijkstra.

We already mentioned that we want $\mathcal{Q}$ to be a Fibonacci heap and to include one item per boundary vertex v. We also change the Monge heaps so that Q_B now contains, for every triplet (a, b_1, b_2), a vertex that minimizes $d(b) = d(a) + M_{ab}$ among vertices b between b_1 and b_2 that were *not yet extracted from* $\mathcal{Q}$. If all vertices between b_1 and b_2 were already extracted then the triplet has no representative in Q_B.

As in FR-Dijkstra, in each iteration a minimum item, corresponding to a vertex v, is extracted from $\mathcal{Q}$ in $O(\log n)$ time. In FR-Dijkstra v is then extracted from the (unique) Monge heap $\mathcal{M}_j$ that contributed v and the new minimum vertex of $\mathcal{M}_j$ is added to $\mathcal{Q}$. In our algorithm, v's extraction from $\mathcal{Q}$ affects all the $O(\log r)$ $\mathcal{M}_j$'s that include v in their B. Before handling any of them we apply operation (1) of the RMQ data structure of Lemma 1 in $O(\log^2 r \log\log r)$ time. Then, for each such $\mathcal{M}_j$, if v was the minimum of some triplet (a, b_1, b_2) then we apply the following operations: (I) remove v from Q_B in $O(\log r)$ time, (II) query in $O(\log r \log\log r)$ time the RMQ data structure of Lemma 1 for vertex $b^* = RMQ(b_1, b_2)$, (III) insert b^* into Q_B in $O(\log r)$ time. Note that we do not replace the triplet (a, b_1, b_2) with two triplets as done in ExtractMin in [13]. Finally, if v was also the minimum of Q_B then let w be the new minimum of Q_B. If w's value in $\mathcal{Q}$ is larger than in Q_B, then we update it in $\mathcal{Q}$ using DecreaseKey in constant time.

Next, we need to activate v in all $\mathcal{M}_j$'s that have v in their A. This is done exactly as in FR-Dijkstra by removing (possibly many) triplets from T and inserting three new triplets. We now find the minimum in each of these three triplets by querying the RMQ data structure of Lemma 1. Finally, we update Q_B and if this changes the minimum element w of Q_B and w's value in $\mathcal{Q}$ is larger than in Q_B, then we update it in $\mathcal{Q}$ using DecreaseKey in constant time.

To summarize, each vertex v of the $O(n/\sqrt{r})$ vertices is extracted once from $\mathcal{Q}$ in $O(\log n)$ time. For each such extraction we do a single RMQ column deactivation in $O(\log^2 r \log\log r)$ time, we do $O(\log r)$ RMQ queries in total $O(\log^2 r \log\log r)$ time incurring $O(\log r)$ DecreaseKey operations on $\mathcal{Q}$ in total $O(\log r)$ time, we activate v in $O(\log^2 r)$ time, and finally we do $O(\log r)$ updates to Q_B in total $O(\log^2 r)$ time incurring $O(\log r)$ DecreaseKey operations on $\mathcal{Q}$ in total $O(\log r)$ time. The overall complexity is thus $O((n/\sqrt{r})(\log n + \log^2 r \log\log r))$.

4 The Main Algorithm

In this section we describe our main result: Combining the modified FR-Dijkstra from the previous section with HKRS [20] to obtain a shortest-path algorithm that runs in $O((n/\sqrt{r}) \log^2 r)$ time.

Algorithm 1 PREPROCESS(G, r)

```
Require: $G$ is an $n$-vertex directed planar graph with non-negative arc lengths and degree at most 3. $r < n$.
1: let $\vec{r} = (r_1, r_2, \ldots, r_k)$ be such that $r_1 = r$ and $r_i = r_{i-1}^2$ for every $i > 1$.
2: compute a recursive $\vec{r}$–division of $G$ with a constant number of holes
3: for each region $R$ of the $r_1$–division do
4:     compute the dense distance graph of $R$
5:     initialize all bipartite Monge heaps of $R$
6: end for
```

Preprocessing. Our algorithm first performs the preprocessing steps outlined in Algorithm 1, and described here in more detail. A recursive $\vec{r}$–division with $\vec{r} = (r_1, r_2, \ldots, r_k)$ is a decomposition tree of G in which the root corresponds to the entire graph G and the nodes at height i correspond to regions that form an r_i–division of G. Throughout the paper we use r-divisions with a constant number of holes. Our algorithm differs from HKRS [20] in our choice of $r_1 = r$ and $r_i = r_{i-1}^2$ (in HKRS $r_1 = \theta(1)$, and $r_i = 2^{O(r_{i-1})}$). This results in a decomposition tree of height roughly $\log \log n$ (in HKRS it is roughly $\log^* n$). For $i \geq 1$, the regions of the r_i–division are called height-i regions. The height of a vertex v is defined as the largest integer i such that v is a boundary vertex of a height-i region. Our algorithm computes the recursive $\vec{r}$–division in linear time using the algorithm of [30]. We assume the input graph G has degree at most three (this can be obtained in linear time by triangulating the dual graph with zero-length edges). With this assumption the algorithm of [30] can be easily modified to return an $\vec{r}$–division in which, for every i, each vertex v belongs to at most three height-i regions.

We next describe the height-0 regions. In [20], height-0 regions consist of individual arcs. We define height-0 regions differently. We begin by describing an auxiliary construction and then define height-0 regions of two types. Consider a height-1 region R (that is, R is a region with $r_1 = r$ edges and $O(\sqrt{r})$ boundary vertices). The DDG of R is a complete weighted graph over the boundary vertices of R, so it has $O(\sqrt{r})$ vertices and $O(r)$ edges. Consider the set of Monge Heaps (MHs) of R. Each MH h corresponds to a bipartite graph with vertex sets A_h and B_h. Each boundary vertex v of R appears in $O(\log r)$ MHs. For each occurrence of v on the B_h side of some MH h, we create a copy v_h of v. Vertices of G such as v are called *natural vertices*, and v_h is called a *copy* of v. Each natural vertex has $O(\log r)$ copies.

We add a zero length arc from every v_h to v. Each of these newly added arcs is the single element in a distinct height-0 region of R. The label of such an arc (and hence the label of the corresponding height-0 region) is defined (as in [20]) to be the label of its tail. It is initialized to ∞.

Let h be a MH. For every vertex $v \in A_h$, we construct a hyperarc e (directed hyperedge) whose tail is v and whose heads are $\{w_h : w \in B_h\}$. We associate e with the MH h. Each hyperarc e is the single element in a distinct height-0 region of R. The label of this height-0 region is defined to be the label of the tail of e, and it is initialized to ∞. Note that the arcs of the DDG do not belong to any region R in the algorithm. Rather, an arc of the DDG is represented in the algorithm in two ways; by a hyperarc, and by the Monge heap to which it belongs. Therefore, from now on we may refer to an arc vw of the DDG as an arc vw_h, and to a vertex $w \in B_h$ as the copy w_h.

To summarize, there are two types of height-0 regions. The first type contains a single arc from a copy v_h of a natural vertex v to v. The second type contains a single hyperarc whose tail is a natural vertex v and that represents all the arcs emanating from v in a MH h.

4.1 Modified Monge Heaps

We now describe a modification of Monge Heaps. Let h be a MH. The arcs of the DDG represented by h all have their tails in A_h and heads in B_h. Let $d(v)$ denote the current label of vertex v, and let $\ell(vw)$ denote the length of the v-to-w arc in the DDG. At any given time a vertex $w_h \in B_h$ is the child of exactly one vertex $v \in A_h$ (the only exception is that upon initialization, no vertex is assigned a parent). We define a state of a vertex. A vertex $w_h \in B_h$ can be either *active* or *inactive*. Initially all vertices are active. Our modified Monge Heaps support the following operations:

- MH-Relax(h, v) - Implicitly relax all arcs of MH h emanating from vertex $v \in A_h$. Internally, adds to the set of children of v all the vertices in B_h whose labels decreased because of these relaxations. This operation activates any of the newly acquired children of v that were inactive.
- MH-GetMinChild(h, v) - Returns the active child w_h of v in MH h minimizing $\ell(vw_h)$.
- MH-Extract(v_h) - Makes vertex v_h inactive in MH h. Let u be the parent of v_h in MH h. Returns the active child w_h of u in h minimizing $\ell(uw_h)$.

The idea for implementing the modified Monge heaps is to replace the rigid mechanism that handles vertex extraction in Fakcharoenphol and Rao's Monge heaps by dynamic RMQ data structures that support column deactivations and reactivations.

We discuss two implementations of the modified Monge heaps. These implementations are summarized in the following lemma.

Lemma 2. *There exist two implementations of the above modified MH with the following construction time (for all MHs of all regions in an r–division of an n-vertex graph), and operation times:*

1. $O((n/\sqrt{r}) \log^2 r)$ *construction time and* $O(\log^3 r)$ *time per operation.*
2. $O(n \log r)$ *construction time and* $O(\log r)$ *time per operation.*

In some applications, in particular in those that work with *reduced lengths* (such as the maximum flow algorithm of section 5.2), shortest path computations on the DDG are performed multiple times with slightly different DDGs. In each time, the new DDG can be obtained quickly but the RMQ data structure needs to be built from scratch. For such applications, the fast construction time of the first implementation is crucial. For other applications, which perform multiple shortest path computations on the same DDG such as the application in Section 5.1, the second implementation is appropriate, and yields slightly faster shortest paths computations. The $O(n \log r)$ construction is not a bottleneck of such applications since it is performed once, and matches the currently fastest known construction of the DDG. In what follows we denote the time to perform an MH operation as $O(\log^{c_q} r)$ where the constant c_q is either 3 or 1 depending on the implementation choice.

In the remainder of this subsection we prove Lemma 2.

Implementation 1 This implementation in the statement of the lemma is similar to that of section 3 and differs only in the use of RMQ. In section 3 we used one RMQ data structure for all MHs of a region R. Here, each MH requires an RMQ data structure of its own. This means that for $i = 1, \dots, \log(\sqrt{r})$, a region R requires 2^i RMQ data structures for $\sqrt{r}/2^i \times \sqrt{r}/2^i$ matrices. Furthermore, in section 3 we only deactivated vertices and so we could use the decremental RMQ of Lemma 1. Here, an inactive vertex can become active again and so we use the (slower) dynamic RMQ of Lemma 1.

To implement MH-GetMinChild(h, v) we simply query the RMQ data structure of h in $O(\log^2 r)$ time. To implement MH-Extract(v_h) we first deactivate v_h in the RMQ of h in $O(\log^3 r)$ time (notice that the deactivation is done only in h as opposed to section 3 where a deactivation applies to all MHs that contain the vertex). Then, we find w_h by querying the RMQ of h in $O(\log^2 r)$ time. To implement MH-Relax(h, v) we first perform the same procedure as in section 3 that creates one new triplet and destroys possibly many triplets. This takes amortized $O(\log r)$ time since destroying a triplet is paid for by that triplet's creation (note that as opposed to section 3 here a triplet of vertex v can be destroyed and then created again multiple times). Then, activating all the new children of v that were inactive takes amortized $O(1)$ time since we can charge the activation of a vertex to the time it was deactivated (in a prior call to MH-Extract).

Constructing the dynamic RMQ of Lemma 1 for an $x \times x$ matrix takes $O(x \log x)$ time. Therefore all RMQ data structures of a region R are constructed in $\sum_{i=1}^{\log(\sqrt{r})} 2^i \cdot \sqrt{r}/2^i \cdot \log(\sqrt{r}/2^i) = O(\sqrt{r} \log^2 r)$ time, and so over all regions we need $O((n/\sqrt{r}) \log^2 r)$ time.

Implementation 2 The second implementation in the statement of Lemma 2 is based on the following simple RMQ.

Observation 1. *Given an $n \times n$ Monge matrix M, one can construct in $O(n^2 \log n)$ time and $O(n^2)$ space a data structure that supports the following operations in $O(\log n)$ time: (1) mark an entry as inactive (2) mark an entry as active (3) report the minimum active entry in a query row and a contiguous range of columns (or ∞ if no such entry exists).*

Proof. The data structure is a very naïve one. We simply store, for each row of M, a separate balanced binary search tree keyed by column number where each node stores the minimum value in its subtree. Deactivating (resp. activating) an entry is done by deletion (resp. insertion) into the tree, and a query is done by taking the minimum value of $O(\log n)$ canonical nodes identified by the query's endpoints. □

Notice that the above naïve RMQ data structure activates and deactivates single entries in M while the specification of MH-Extract seems to require activating and deactivating and entire columns of M (The specification is to make v_h inactive in the entire MH h). However, activating and deactivating a single element suffices. This is because each vertex $v_h \in B_h$ appears in exactly one triplet at any given time. Therefore, whether it is active or inactive is only important in the RMQ of the vertex $u \in A_h$ whose triplet contains v_h. Whenever an inactive v_h changes parents, we make v_h active by reintroducing the corresponding element into the RMQ if it was previously deleted, and charge the activation to the time v_h was deactivated in a prior call to MH-Extract.

Constructing the naïve data structure on all MHs of a region R takes $\sum_{i=1}^{\log(\sqrt{r})} 2^i(\sqrt{r}/2^i)^2 \cdot \log(\sqrt{r}/2^i) = O(r \log r)$ time, and so over all regions we need $O(n \log r)$ time. Each operation on the RMQ now takes only $O(\log r)$ time. This means that MH-Relax, MH-GetMinChild, and MH-Extract can all be done in $O(\log r)$ amortized time.

4.2 Computing Shortest Paths on the DDG

The pseudocode of the algorithm is given below. Parts of the pseudocode appear in black and parts in blue. The black parts describe our algorithm, and at this point it is enough to focus only on them.

The blue parts are additions to the algorithm that should not (and in fact cannot) be implemented. They are used for the analysis only.

The algorithm maintains a label $d(v)$ for each vertex v. Initially all labels are infinite, except the label of the source s, which is 0. For each height-i region R, the algorithm maintains a heap $Q(R)$ (implemented using Fibonacci heaps [15]) containing the height-$(i-1)$ subregions of R. For a height-0 region R, $Q(R)$ contains the single element (arc or hyperarc) e comprising the region R. If v is the tail of e, then the key of this single element is $d(v)$ if e is not relaxed, and infinity otherwise. For a height-i region R ($i \geq 1$), the key of a subregion R' of R in $Q(R)$ is $\mathit{minKey}(Q(R'))$. Such heaps were also used in [20] but they were not implemented by Fibonacci heaps. The heaps support the following operations: *getKey*, *minKey*, *minItem*, *decreaseKey*, *increaseKey*. The first four operations take constant amortized time, and the fifth takes logarithmic amortized time.

As in [20], the main procedure of the algorithm is the procedure PROCESS, which processes a region R. The algorithm repeatedly calls PROCESS on the region R_G corresponding to the entire graph, until $\mathit{minKey}(Q(R_G))$ is infinite, at which point the labels $d(\cdot)$ are the correct shortest path distances. As in [20], processing a height-i region R for $i \geq 1$ consists of calling PROCESS on at most α_i subregions R' of R (Line 29). Processing a region R' may change $\mathit{minKey}(Q(R'))$, so the algorithm updates the key of R' in $Q(R)$ in Line 30.[8]

Our algorithm differs from [20] in processing height-0 regions. Let us first recall how [20] processes height-0 regions. In [20], each height-0 region R corresponds to a single arc uv, and processing R consists of relaxing uv. If the relaxation decreases $d(v)$, then all arcs whose tail is v become unrelaxed. For every such arc e' the key of the corresponding height-0 region R' corresponding to e' needs to be updated to $d(v)$. Doing so may require updating the keys of ancestor regions of R'. The procedure UPDATE performs this chain of updates.

In our case, the height-0 regions may correspond to single arcs or to single hyperarcs. If a height-0 region R consists of a single hyperarc e whose tail is v, then processing R implicitly relaxes all the arcs of the DDG represented by e by calling MH-RELAX (Line 4). We can only afford to update explicitly the label of a single child of v. We do so for the child w_h of v with minimum label (Line 7). Since w_h is a copy vertex of a natural vertex w, there is exactly one arc whose tail is w_h (namely, $w_h w$), and no hyperarcs whose tail is w_h. The arc $w_h w$ is now unrelaxed, so the key of the height-0 region corresponding to $w_h w$ needs to be updated by a call to UPDATE (Line 9).

If a height-0 region R consists of a single arc $v_h v$, then, as in [20], processing R explicitly relaxes $v_h v$. Changing the label of v requires updating the labels of all elements (hyperarcs in this case) whose tail is v. This is done by calling UPDATE on the heaps of the corresponding height-0 regions and their ancestor regions (Lines 13– 15). Similarly to the implementation of FR-Dijkstra, since at this point the vertex v_h has no outgoing unrelaxed arcs, the algorithm extracts v_h from the Monge heap h to which it belongs (so v_h becomes inactive). Among the children of the parent of v_h there is now a new minimum active child w_h (v_h may have been the previous minimum, but it has just become inactive). The algorithm handles w_h, as in the previous case, by updating the key of the height-0 region corresponding to $w_h w$ with a call to UPDATE (Line 22).

[8]Since edge lengths are non-negative, and since the labels $d(\cdot)$ (and hence keys) only change by relaxations or by setting to infinity, $\mathit{minKey}(Q(R'))$ may only increase when processing R'. Therefore updating the key of R' in $Q(R)$ is done using *increaseKey*.

Algorithm 2 UPDATE(R, x, k, v)

```
Require: $R$ is a region, $x$ is an item of $Q(R)$, $k$ is a key value, $v$ is a boundary vertex of $R$.
1: $decreaseKey(Q(R), x, k, v)$
2: if the $decreaseKey$ operation reduced the value of $minKey(Q(R))$ then
3:     entry$(R) \leftarrow v$
4:     return 1+UPDATE(parent$(R), R, k, v)$
5: else
6:     return 1
7: end if
```

4.3 Correctness

In this section we prove the correctness of our algorithm by considering a variant of the algorithm of [20] on the graph constructed by our algorithm.

Note that distances in the input graph and in the graph constructed by our algorithm are identical since duplicating vertices and adding zero-length arcs between the copies does not affect distances. Also note that the correctness of the algorithm in [20] does not depend on the choice of attention span (the parameters α_i). These parameters only affect the running time. In particular, a variant of that algorithm in which the attention span is not fixed would yield the correct distances upon termination.

We argue the correctness of our algorithm by considering such a variant of the algorithm of [20] where each hyperarc is represented by the individual arcs forming it. We refer to this variant as algorithm H. In what follows we still refer to hyperarcs to make the grouping of the individual arcs clear. We say that an arc vw_h belongs to hyperarc e if vw_h is one of the arcs forming e.

Let e be a hyperarc whose tail is v. Let R be the height-1 region containing e. Consider a time in the execution of algorithm H when R is processed and chooses to process a height-0 region corresponding to a single arc vw_h that belongs to e. This implies that the label of vw_h is the minimum among all height-0 regions (edges) in R. Observe that at this time all other arcs of e have the same label as that of vw_h (they might have infinite label if they have already been processed). Furthermore, since lengths are non-negative, processing vw_h cannot decrease the label of w_h below that of v. We define algorithm H to process all arcs corresponding to e one after the other, regardless of the attention span. The above discussion implies that algorithm H terminates with the correct distance labels for all vertices in the graph.

We now show that algorithm H and our algorithm relax exactly the same arcs, and in the same order. The only difference between algorithm H and our algorithm is that our algorithm performs implicit relaxations using Monge heaps, while algorithm H performs all relaxations explicitly. In particular, when relaxing a hyperarc e whose tail is v, our algorithm only updates the label of the minimum child of v (Lines 5–7), whereas algorithm H updates the labels of all children of v. We call a vertex w_h *latent* if its label does not reflect prior relaxations of arcs vw_h incident to it (because those relaxations were implicit). A vertex that is not latent is *accurate*. Note that whenever the label of a vertex w_h is updated by our algorithm (i.e., w_h becomes accurate), the label of the height-0 region consisting of the unique arc $w_h w$ is also updated (Lines 5–9).

To establish that the order of relaxations is the same in both algorithms, it suffices to establish the following lemma:

Lemma 3. *The following invariants hold:*

Algorithm 3 PROCESS(R, debt)

```
1: if $R$ is a height-0 region that contains a single hyperarc $e$ then
2:     let $v$ be the tail of $e$
3:     let $h$ be the MH to which $e$ belongs
4:     MH-RELAX$(h, v)$
5:     $w_h \leftarrow$ MH-GETMINCHILD$(h, v)$
6:     debt $\leftarrow$ debt $+ c_3 \log^{c_q} r_1$
7:     $d(w_h) \leftarrow d(v) +$ length of arc $vw$ in the DDG of $h$
8:     let $R'$ be the height-0 region consisting of the arc $w_h w$
9:     debt $\leftarrow$ debt $+$ UPDATE$(R', w_h w, d(w_h), w_h)$
10:    $increaseKey(Q(R), e, \infty)$
11: else if $R$ is a height-0 region that contains a single arc $v_h v$ then
12:    if $d(v) > d(v_h)$ then
13:        $d(v) \leftarrow d(v_h)$
14:        for each hyperarc $e$ whose tail is $v$ do
15:            debt $\leftarrow$ debt $+$ UPDATE$(R(e), e, d(v), v)$
16:        end for
17:    end if
18:    $w_h \leftarrow$ MH-EXTRACT$(v_h)$
19:    debt $\leftarrow$ debt $+ c_3 \log^{c_q} r_1$
20:    $d(w_h) \leftarrow d(v) +$ length of arc $vw$ in the DDG of $h$
21:    let $R'$ be the height-0 region consisting of the arc $w_h w$
22:    debt $\leftarrow$ debt $+$ UPDATE$(R', w_h w, d(w_h), w_h)$
23:    $increaseKey(Q(R), v_h v, \infty)$
24: else
25:    upDebt $\leftarrow 0$ ; credit $\leftarrow 0$
26:    for $\alpha_{height(R)}$ times or until $minKey(Q(R))$ is infinity do
27:        $R' \leftarrow minItem(Q(R))$
28:        credit $\leftarrow$ credit $+$ debt$/\alpha_{height(R)}$
29:        upDebt $\leftarrow$ upDebt $+$ PROCESS$(R'$, debt$/\alpha_{height(R)} + \log |Q(R)|)$
30:        $increaseKey(Q(R), R', minKey(Q(R')))$
31:    end for
32:    debt $\leftarrow$ debt $+$ upDebt $-$ credit
33:    if $minKey(Q(R))$ will decrease in the future then
34:        return debt
35:    else (this invocation is stable)
36:        pay off debt from the account of $(R$, entry$(R))$
37:        return 0
38: end if
```

1. *Every latent vertex is active.*
2. *Let h be an MH, and let v be a vertex in A_h. Let S_v be the set of active children of v with minimum label. If S_v is non-empty then there is a vertex in S_v that is accurate.*
3. *If the key of a height-0 region consisting of a single arc $w_h w$ is finite then w_h is active.*

Proof. The proof is by induction on the calls to PROCESS on height-0 regions performed by our algorithm. Initially there are no latent vertices, all the vertices on the B side of any MH are active, and all height-0 regions consisting of a single arc have infinite labels, so all invariants trivially hold. The inductive step consists of two cases.

Suppose first that the height-0 region being processed consists of a single hyperarc e. This hyperarc is implicitly relaxed in Line 4. All the vertices whose label should be changed by the relaxation are guaranteed to become active (by the specification of MH-RELAX). Among these vertices, w_h, the vertex with minimum label, is explicitly relaxed (Lines 5–7), so it is accurate. All the others are latent. This establishes that the first two invariants are maintained. In Line 9 the key of the height-0 region R' consisting of the unique arc whose tail is w_h is set to a finite value. Since w_h has just become active, the third invariant is also maintained.

Suppose now that the height-0 region being processed consists of a single arc $v_h v$. By the third invariant, v_h is active. By the second invariant, v_h must be accurate. The arc $v_h v$ is explicitly relaxed, which does not create any new latent vertices. In Line 18, v_h is deactivated. This is the only case where a vertex is deactivated. Since v_h is accurate, invariant 1 is maintained. Let u be the parent of v_h in MH h. By the specification of MH-EXTRACT, it returns an active child w_h of u whose label is minimum among all of u's active children. The label of w_h is explicitly updated in Line 20, so invariant 2 is maintained. The height-0 region whose label is set to a finite value in Line 22 consists of the single arc $w_h w$. Since w_h is active, invariant 3 is maintained. □

Note that invariants 1 and 2 guarantee that in every Monge heap h, if there exists a latent vertex w_h then there exists an accurate vertex u_h whose label is at most the correct label of w_h (by correct label we mean the label that w_h would have gotten had we performed the relaxations explicitly). This implies that as long as a vertex is latent, its label does not affect the running of the algorithm. To see this, let R be the height-1 region that contains both w_h and u_h. By the following observation, neither w_h or u_h are boundary vertices of R.

Observation 2. *The height of every copy vertex v_h is 0.*

Proof. The construction is such that all arcs and hyperarcs incident to a vertex v_h correspond to height-0 regions of the same height-1 region R. Therefore, v_h is not a boundary vertex of R, so the height of v_h is 0. □

Since the label of u_h is smaller than that of w_h no arc whose tail is w_h would become the minimum of the heap $Q(R)$ as long as w_h is latent. This implies, by a trivial inductive argument, that for any $i \geq 1$, for any height-i region R, the minimum elements in the heap $Q(R)$ in our algorithm and in algorithm H have the same label, and in fact correspond to the same arc (or to a hyperarc e in our algorithm and to one of the arcs that belong to e in algorithm H). It follows that the two algorithms perform the same relaxations and in the same order. Hence our algorithm produces the same labels as algorithm H, and is therefore correct.

4.4 Analysis

The analysis of the original HKRS algorithm [20, 29] proved that the running time is linear in the number of arcs of the planar graph G, which is also linear in the number of vertices of G. In our case however, the number of arcs in the DDG is $O(n)$, but we want a running time bound that is linear in the number of vertices of the DDG, which is $O(n/\sqrt{r})$. To obtain this stronger bound we need a rather technical analysis that follows the analysis in [29] with a careful tighter choice of region sizes and attention span, and with appropriately handling the level-0 regions. The final bounds are summarized by the following theorem.

Theorem 1. *A shortest path computation on the dense distance graph of an r–division of an n-vertex graph can be done in $O((n/\sqrt{r})\log^2 r)$ time after $O(n \log r)$ preprocessing, or in $O((n/\sqrt{r})\log^4 r)$ time after $O((n/\sqrt{r})\log^2 r)$ preprocessing.*

In the remainder of this subsection we prove Theorem 1. The analysis follows that of HKRS [20]. We use the more recent description of the charging scheme in [29], which differs from the original one in [20] in the organization and presentation of the charging scheme, but in essence is the same.

Our algorithm differs from that of [20] in the implementation of height-0 regions. The following lemma bounds the number of such regions.

Let c_1, c_2 be the constants such that an r-division of an n-vertex graph has at most $c_1 n/r$ regions, each having at most $c_2\sqrt{r}$ boundary vertices. Let c_5 be the constant such that there are at most $c_5 \log r_1$ height-0 regions involving v.

Lemma 4. *The number of height-0 regions is at most $c_1 c_2 c_5 \frac{n}{\sqrt{r_1}} \log r_1$*

Proof. There are at most $c_1 \frac{n}{r_1}$ regions R of height-1. A height-1 region R has at most $c_2\sqrt{r_1}$ boundary vertices. Each boundary vertex v of R appears in $O(\log r_1)$ MHs, so it has $O(\log r_1)$ copies. Hence there are $O(\log r_1)$ height-0 regions that correspond to individual arcs $v_h v$, and $O(\log r_1)$ height-0 regions that correspond to individual hyperarcs whose tail is v. There are at most $c_5 \log r_1$ height-0 regions involving v. The total number of height-0 regions is therefore at most $c_1 c_2 c_5 \frac{n}{\sqrt{r_1}} \log r_1$. □

Apart from the implementation of height-0 regions, our analysis differs from the original analysis of HKRS in the choice of parameters (namely, the choice of region sizes r_i). However, large parts of the original analysis of HKRS, and in particular parts that do not rely on the choice of parameters, remain valid without any change. Since our analysis relies on the details of the analysis of HKRS, we include some of the definitions and statement of lemmas from [29] and only prove the lemmas that are different.

We define an *entry vertex* of a region R (denoted entry(R)) as follows. The only entry vertex of the region R_G (the region consisting of the entire graph G) is s (the source) itself. For any other region R, v is an entry vertex if v is a boundary vertex of R. We define the height of a vertex v to be the largest integer i such that v is an entry vertex of a height-i region.

When the algorithm processes a region R, it finds shorter paths to some of the vertices of R and so reduces their labels. Suppose one such vertex v is a boundary vertex of R. The result is that the shorter path to v can lead to shorter paths to vertices in a neighboring region R' for which v is an entry vertex. In order to preserve the property that the minimum key of $Q(R')$ reflects the labels of vertices of R' the algorithm might need to update $Q(R')$. Updating the queues of neighboring regions and their ancestors is handled by the UPDATE procedure. The reduction of *minKey*($Q(R')$)

(which can only occur as a result of a reduction in the label of an entry vertex v of R') is a highly significant event for the analysis. We refer to such an event as a *foreign intrusion of region R' via entry vertex v.*

Because of the recursive structure of PROCESS, each initial invocation PROCESS(R_G) is the root of a tree of invocations of PROCESS and UPDATE, each with an associated region R. Parents, children, ancestors, and descendants of recursive invocations are defined in the usual way.

For an invocation A of PROCESS on region R, we define $\mathrm{start}(A)$ and $\mathrm{end}(A)$ to be the values of *minKey*($Q(R)$) just before the invocation starts and just after the invocation ends, respectively.

To facilitate the analysis we augment the pseudocode of PROCESS and UPDATE to keep track of the costs of the various operations. These additions to the pseudocode are shown in blue. Note that these additions are purely an expository device for the purpose of analysis; the additional blue code is not intended to be actually executed. In fact, Line 33 of PROCESS cannot be executed since it requires knowledge of the future!

Amounts of cost are passed around by UPDATE and PROCESS via return values and arguments. We think of these amounts as *debt obligations*. The running time of the algorithm is dominated by the time for priority-queue operations and for operations on the Monge Heaps. New debt is generated for every such call in Lines 6, 19, and 29 of PROCESS, and in Lines 4 and 6 of UPDATE. These debt obligations travel up and down the forest of invocations. An invocation passes down some of its debt to its children in hope they will pay this debt. A child, however, may pass unpaid debt (inherited from the parent or generated by its descendants) to its parent. Debts are eventually charged by invocations of PROCESS to pairs (R, v) where R is a region and v is an entry vertex of R.

We say an invocation A of PROCESS is *stable* if, for every invocation B that starts after A started, the start key of B is at least the start key of A. If an invocation is stable, it pays off the debt by withdrawing the necessary amount from an account, the account associated with the pair (R, v) where v is the value of $\mathrm{entry}(R)$ at the time of the invocation. This value is set by UPDATE whenever a foreign intrusion occurs (Line 3). Initially the entries in the table are undefined. However, for any region R, the only way that *minKey*($Q(R)$) can become finite is by an intrusion. We are therefore guaranteed that, at any time at which *minKey*($Q(R)$) is finite, $\mathrm{entry}(R)$ is an entry vertex of R.

Any invocation whose region is the whole graph is stable because there are no foreign intrusions of that region. Therefore, such an invocation never tries to pass any debt to its nonexistent parent. We are therefore guaranteed that all costs incurred by the algorithm are eventually charged to accounts.

The following theorem, called the Payoff Theorem (the charging scheme invariant [20, Lemma 3.15]), is the main element in the analysis of HKRS. The theorem, as well as its original proofs in [20, 29], applies to our algorithm. This is because the proof only depends on processing elements with (locally) minimum labels, and on the fact that processing a region R only decreases labels of vertices that belong to R.

Theorem 2 (Payoff Theorem (Lemma 3.15 in [20]))**.** *For each region R and entry vertex v of R, the account (R, v) is used to pay off a positive amount at most once.*

The remainder of the analysis bounds the total debt charged from all accounts. The total debt depends on the parameters $\boldsymbol{r} = (r_1, r_2, \ldots)$ of the recursive $\boldsymbol{r}$-division and on the parameters $\alpha_0, \alpha_1, \ldots$ that govern the number of iterations per invocation of PROCESS. Recall that $r_1 = r$ and $r_i = r_{i-1}^2$ for $i > 1$. Define $\alpha_0 = 1$ and $\alpha_i = \frac{4 \log r_{i+1}}{3 \log r_i} = \frac{8}{3}$ for $i > 0$.

Lemma 5. *Each invocation at height i inherits at most $4 \log r_{i+1}$ debt.*

Proof. The proof is by reverse induction on i. In the base case, each top-height invocation clearly inherits no debt at all. Suppose the lemma holds for i, and consider a height-i invocation on a region R. By the inductive hypothesis, this invocation inherits at most $4 \log r_{i+1}$ debt. An α_i fraction of this debt is passed down to each child. In addition, each child R' inherits a debt of $\log(|Q(R)|) = \log r_i$ to cover the cost of the *minItem* and *increaseKey* operations on $Q(R)$ required for invoking the call to Process on R'. Overall we get that each child inherits $\frac{4 \log r_{i+1}}{\alpha_i} + \log r_i$ which by the choice of α_i is $4 \log r_i$. □

We next wish to bound the number of descendant invocations incurred by an invocation of Process. Define $\beta_{ij} = \alpha_i \alpha_{i-1} \dots \alpha_{j+1}$ for $i > j$. Define β_{ij} to be 1 for $i = j$, and 0 for $i < j$. By definition,

$$\beta_{ij} = \prod_{k=j+1}^{i} \frac{4 \log r_{k+1}}{3 \log r_k} = \left(\frac{4}{3}\right)^{i-j} \frac{\log r_{i+1}}{\log r_{j+1}}.$$

Lemma 6. *A* Process *invocation of height i has at most β_{ij} descendant* Process *invocations of height j.*

The debt incurred by the algorithm by a step of Process is called the *process debt.* Note that process debt is only generated when a recursive call to process is made in Line 29, or at height-0 invocations in Lines 6 and 19. Let c_3 be a constant such that a Monge heap operation takes $c_3 \log^{c_q} r_1$ time.

Lemma 7. *For each height-i region R and boundary vertex x of R, the amount of process debt payed off by the account (R, x) is at most $c_3 \beta_{i0} \log^{c_q} r_1 + \sum_{j \leq i} 4\beta_{ij} \log r_{j+1}$.*

Proof. By the Payoff Theorem, the (R, x) account is used at most once. Let A be the invocation of Process that withdraws the payoff from that account. Each dollar of process debt paid off by A was sent back to A from some descendant invocation who inherited or incurred that dollar of process debt. To account for the total amount of process debt paid off by A we consider each of its descendant invocations. By Lemma 6, A has β_{ij} descendants of height j, each of which inherited process debt of at most $4 \log r_{j+1}$ dollars, by Lemma 5. This accounts for all the process debt generated in executions of Line 29. In addition, each of the β_{i0} descendant height-0 invocations incurs process debt of at most $c_3 \log^{c_q} r_1$. □

Recall that c_1, c_2 are the constants such that an r-division of an n-vertex graph has at most $c_1 n/r$ regions, each having at most $c_2 \sqrt{r}$ boundary vertices.

Lemma 8. *Let n be the number of edges of the input graph. For any nonnegative integer i, there are at most $c_1 c_2 n/\sqrt{r_i}$ pairs (R, x) where R is a height-i region and x is an entry vertex of R.*

Combining this lemma with Lemma 7, we obtain

Corollary 1. *The total process debt is at most*

$$c_1 c_2 \sum_i \frac{n}{\sqrt{r_i}} \left(c_3 \beta_{i0} \log^{c_q} r_1 + \sum_{j \leq i} 4\beta_{ij} \log r_{j+1} \right)$$

The cost incurred by the algorithm by a step of Update is called *update debt.* The event (in Lines 7, 13, and 20 of Process) of reducing a vertex x's label $d(x)$ initiates a chain of calls to Update in Lines 9, 15, and 22, respectively, for each outgoing arc xy. We say the debt incurred is *on behalf of x.*

Lemma 9. *([29, Lemma 6.5.6]) If* Update *is called on the parent of region R during an invocation A of* Process *then R is not the region of A.*

Recall that for a vertex v, we defined

$$\text{height}(v) = \max\{j : v \text{ is a boundary vertex of a height-}j \text{ region}\}$$

Lemma 10. *A chain of calls to* Update *initiated by the reduction of the label of a natural vertex v has total cost $O(1 + \mathit{height}(v))$.*

A chain of calls to Update *initiated by the reduction of the label of a copy vertex w_h has total cost $O(1)$.*

Proof. Let A_0 be the invocation of Process during which the initial call to Update was made, and let R be the (height-0) region of A_0.

For the first case, the call to Update is in line 15. In this case, R corresponds to a single arc v_hv. Consider the chain of calls to Update, and let $R_0, R_1, \dots, R_p$ be the corresponding regions. R_0 corresponds to a single hyperarc e whose tail is v. Note that $\text{height}(R_j) = j$. The cost of each call is $O(1)$ since we use Fibonacci heaps. Since R_{p-1} contains e but (by Lemma 9) does not contain v_hv, we get that v is a boundary vertex of R_{p-1}, so $p \le \text{height}(v) + 1$.

For the second case, the call is Update$(R', w_hw, d(w_h))$, either in Line 9, or in Line 22. In the former case R corresponds to a single hyperarc incident to w_h. In the latter case R corresponds to a single arc v_hv. In both cases $\text{parent}(R)$ and $\text{parent}(R')$ is the same height-1 region R_1, and the key of R' is at least the key of R. Since at the time the call to Update is made $\mathit{minKey}(Q(R_1)) = R$, the call Update$(R', w_hw, d(w_h))$ does not decrease $\mathit{minKey}(Q(R_1))$, so this chain of calls to Update consists of exactly two calls. Since we use Fibonacci heaps, each call costs $O(1)$. □

Lemma 11. *Let i, j be nonnegative integers, and let R be a region of height i. The total amount of update debt incurred on behalf of vertices of height at most j and paid off from accounts $\{(R, x) : x$ is an entry vertex of $R\}$ is at most*

$$3c_2\sqrt{r_i}\beta_{i0}(3c_5 \log r_1 + j).$$

Proof. The number of entry vertices x of R is at most $c_2\sqrt{r_i}$. For each, the Payoff Theorem ensures that all the debt paid off from account (R, x) comes from descendants of a single invocation A of Process. The number of height-0 descendants of A is β_{i0}. Consider such a height-0 descendant A_0 whose region is R_0. If R_0 consists of a single hyperarc, then the height of the vertex w_h on whose behalf the call to Update is made in Line 9 is 0 (see Observation 2). By Lemma 10 the cost of this chain of calls is at most 2. If R_0 consists of a single arc v_hv, then v has height at least 1. Suppose the height of v is at most j. In each of the three height-1 regions to which v belongs, it participates in at most $c_5 \log r_1$ Monge heaps. Hence, in each of these regions there are at most $c_5 \log r_1$ hyperarcs whose tail is v for which a call to Update is made in Line 15. Since all of these calls are made with the same key, exactly one of them may cause a chain of update whose length is at most j. By Lemma 10 the cost of such a chain is at most $j + 2$. The other calls only cause a chain of updates whose length is 2. Hence, all calls made in Line 15 on behalf of v cost at most $3(2c_5 \log r_1 + j + 2)$. There is an additional call to UPDATE in Line 22. As before, since this call is made on behalf of a height-0 vertex, its cost is at most 2.

We have therefore established that the total amount of update debt incurred on behalf of vertices of height at most j and paid off from accounts $\{(R, x) : x \text{ an entry vertex of } R\}$ is at most $c_2\sqrt{r_i}\beta_{i0}(2 + 3(2c_5 \log r_1 + j + 2))$, which is at most $3c_2\sqrt{r_i}\beta_{i0}(3c_5 \log r_1 + j)$. ☐

Lemma 12. *The total update debt is at most*

$$\begin{aligned}
&2c_1c_2c_5 n r_1^{-1/2} \log r_1 \quad + && (1)\\
&9c_1c_2c_5 n \sum_{i>0} r_i^{-1/2}\beta_{i0} \log r_1 \quad + && (2)\\
&3c_1c_2 n \sum_{i>0} r_i^{-1/2}\beta_{i0} i \quad + && (3)\\
&27c_1c_2^2c_5 n \sum_{j\geq 2} r_j^{-1/2} \sum_{1\leq i<j} r_i^{1/2}\beta_{i0} \log r_1 \quad + && (4)\\
&9c_1c_2^2 n \sum_{j\geq 2} r_j^{-1/2} \sum_{1\leq i<j} r_i^{1/2}\beta_{i0} j \quad + && (5)\\
&27c_1c_2^2c_5^2 n \sum_{j\geq 1} r_j^{-1/2} \log^2 r_1 \quad + && (6)\\
&9c_1c_2^2c_5 n \sum_{j\geq 1} r_j^{-1/2} j \log r_1 && (7)
\end{aligned}$$

Proof. To each unit of update debt, we associate two integers: i is the height of the region R such that the debt is paid off from an account (R, x), and j is the height of the vertex v on whose behalf the debt was incurred. If $i \geq j$, we refer to the debt as *type 1* debt, and if $i < j$, we refer to it as *type 2* debt.

First we bound the type-1 debt. For each integer $i > 0$, there are at most $c_1\frac{n}{r_i}$ regions R of height i. By Lemma 11, these contribute to the total type-1 debt $\sum_{i>0} c_1\frac{n}{r_i}3c_2\sqrt{r_i}\beta_{i0}(3c_5 \log r_1 + i)$.

By Lemma 4, the number of height 0 regions is $c_1c_2c_5\frac{n}{\sqrt{r_1}} \log r_1$, each with a single entry vertex. An invocation on such a region makes at most one call to UPDATE that is on behalf of a height-0 vertex. By Lemma 10 the cost of such a call is at most 2. Therefore, the contribution of height-0 regions to type-1 debt is $2c_1c_2c_5\frac{n}{\sqrt{r_1}} \log r_1$. We get that the total type-1 debt is at most

$$2c_1c_2c_5\frac{n}{\sqrt{r_1}} \log r_1 + \sum_{i>0} 3c_1c_2\frac{n}{\sqrt{r_i}}\beta_{i0}(3c_5 \log r_1 + i).$$

Now we bound the type-2 debt (i.e., $j > i$). For each integer j (since $j > i$, j must be at least 1), the number of regions of height j is at most $c_1\frac{n}{r_j}$ and each has at most $c_2\sqrt{r_j}$ entry vertices, so the total number of vertices of height j is at most $c_1c_2\frac{n}{\sqrt{r_j}}$. Consider each such vertex v. We handle separately the cases $i > 0$ and $i = 0$.

- For any $i > 0$, every vertex (and in particular v) belongs to at most 3 height-i regions. Hence, by Lemma 11, the total type-2 debt for $i \geq 1$ is

$$\sum_{j\geq 2} c_1c_2\frac{n}{\sqrt{r_j}}3 \sum_{1\leq i<j} 3c_2\sqrt{r_i}\beta_{i0}(3c_5 \log r_1 + j).$$

- We next consider the case $i = 0$. For $j \geq 1$, there are $c_1 c_2 \frac{n}{\sqrt{r_j}}$ vertices of height j. Each such vertex v belongs to at most 3 height-1 regions, and hence to at most $3c_5 \log r_1$ height-0 regions. Hence, by Lemma 11, the total type-2 debt for $i = 0$ is

$$\sum_{j \geq 1} c_1 c_2 \frac{n}{\sqrt{r}_j} 3c_5 \log r_1 3c_2 (3c_5 \log r_1 + j) = \sum_{j \geq 1} 9 c_1 c_2^2 c_5 \frac{n}{\sqrt{r}_j} \log r_1 (3c_5 \log r_1 + j)$$

The total update debt is thus as claimed in the lemma's statement. □

We are now prepared to bound the total running time of our algorithm to $O((n/\sqrt{r}) \log^2 r)$. The following two lemmas bound the total process debt to $O(n/\sqrt{r}) \log^{c_q+1} r)$ and the total update debt to $O(n/\sqrt{r}) \log^2 r)$. Using the naive RMQ data structure with $c_q = 1$ we obtain the claimed running time.

Lemma 13. *The total process debt is $O(\frac{n}{\sqrt{r}} \log^{c_q+1} r)$.*

Proof. Recall that $r_{i+1} = r_i^2$, so $r_i = r_1^{2^{i-1}}$ and $\log r_{i+1} = 2 \log r_i$. Assuming $r_1 \geq 2$, we have $\log r_{i+1} > 2^i > \left(\frac{4}{3}\right)^i$. It follows that for any i

$$\left(\frac{4}{3}\right)^i \log r_{i+1} \leq 4 \log^2 r_i$$

Observe that

$$\begin{aligned} r_i^{-1/2} \sum_{j \leq i} \beta_{ij} \log r_{j+1} &= \\ r_i^{-1/2} \sum_{j \leq i} \left(\frac{4}{3}\right)^{i-j} \frac{\log r_{i+1}}{\log r_{j+1}} \log r_{j+1} &= \\ \left(\frac{4}{3}\right)^i r_i^{-1/2} \log r_{i+1} \sum_{j \leq i} \left(\frac{3}{4}\right)^j &< \\ 4 r_i^{-1/2} \left(\frac{4}{3}\right)^i \log r_{i+1} &< \\ 16 r_i^{-1/2} \log^2 r_i & \end{aligned} \tag{8}$$

And that

$$\begin{aligned} r_i^{-1/2} \beta_{i0} \log^{c_q} r_1 &= \\ r_i^{-1/2} \left(\frac{4}{3}\right)^i \frac{\log r_{i+1}}{\log r_1} \log^{c_q} r_1 &< \\ 4 r_i^{-1/2} \log^2 r_i \log^{c_q - 1} r_1 & \end{aligned} \tag{9}$$

Substituting (8) and (9) into Corollary 1, the total process debt is at most

$$c_1 c_2 \sum_i \frac{n}{\sqrt{r_i}} \left(c_3 \beta_{i0} \log^{c_q} r_1 + \sum_{j \leq i} 4 \beta_{ij} \log r_{j+1} \right) <$$
$$c_1 c_2 n \sum_i \left(4 c_3 r_i^{-1/2} \log^2 r_i \log^{c_q - 1} r_1 \ + 64 r_i^{-1/2} \log^2 r_i \right)$$

Since the r_i's increase doubly exponentially, we have that $\sum_{i>0} r_i^{-1/2} \log^2 r_i = O(r_1^{-1/2} \log^2 r_1)$ and so the total process debt is bounded by $O(\frac{n}{\sqrt{r}} \log^{c_q+1} r)$. □

Lemma 14. *The total update debt is* $O(\frac{n}{\sqrt{r_1}} \log^2 r_1)$.

Proof. We need to bound every term in the bound stated in Lemma 12.

- The first term is clearly $O(\frac{n}{\sqrt{r_1}} \log r_1)$.
- By (9), the second term is
$$36c_1c_2c_5n \sum_{i>0} r_i^{-1/2} \log^2 r_i.$$
Recall that $r_{i+1} = r_i^2$, so $r_i = r_1^{2^{i-1}}$. Since the r_i's increase doubly exponentially, we have that $\sum_{i>0} r_i^{-1/2} \log^2 r_i = O(r_1^{-1/2} \log^2 r_1)$ and so the term is bounded by $O(\frac{n}{\sqrt{r_1}} \log^2 r_1)$.
- Similarly to the derivation of (9),
$$\begin{aligned} r_i^{-1/2}\beta_{i0} i &= \\ r_i^{-1/2} \left(\frac{4}{3}\right)^i \frac{\log r_{i+1}}{\log r_1} i &< \\ 4ir_i^{-1/2} \log^2 r_i \log^{-1} r_1 & \end{aligned} \tag{10}$$
Substituting (10) into the third term we get $3c_1c_2n \sum_{i>0} ir_i^{-1/2} \log^2 r_i \log^{-1} r_1$. Since the r_i's increase doubly exponentially, we have that $\sum_{i>0} ir_i^{-1/2} \log^2 r_i = O(r_1^{-1/2} \log^2 r_1)$ and so the term is bounded by $O(\frac{n}{\sqrt{r_1}} \log r_1)$.
- We need to bound the fourth term, which is
$$27c_1c_2^2c_5n \sum_{j\geq 2} r_j^{-1/2} \sum_{1\leq i<j} r_i^{1/2} \beta_{i0} \log r_1.$$
We first focus on the second sum.
$$\begin{aligned} \sum_{1\leq i<j} r_i^{1/2}\beta_{i0} &= \\ \sum_{1\leq i<j} r_i^{1/2} \left(\frac{4}{3}\right)^i \frac{\log r_{i+1}}{\log r_1} &< \\ 4 \sum_{1\leq i<j} r_i^{1/2} \log^2 r_i \log^{-1} r_1 &< \\ c_6 r_{j-1}^{1/2} \log^2 r_{j-1} \log^{-1} r_1 & \end{aligned} \tag{11}$$

In the last step we again used the exponential increase of the r_i's. The fourth term is therefore bounded by

$$\begin{aligned}27c_1c_2^2c_5n\sum_{j\geq 2} r_j^{-1/2}c_6r_{j-1}^{1/2}\log^2 r_{j-1} &= \\ 27c_1c_2^2c_5c_6n\sum_{j\geq 2} r_j^{-1/2}r_j^{1/4}\left(\frac{1}{2}\log r_j\right)^2 &= \\ \frac{27}{4}c_1c_2^2c_5c_6n\sum_{j\geq 2} r_j^{-1/4}\log^2 r_j&\end{aligned}$$

Again, this sum is dominated by the leading term, so it is $O(nr_2^{-1/4}\log^2 r_2) = O(\frac{n}{\sqrt{r_1}}\log^2 r_1)$.

- The fifth term we need to bound is

$$9c_1c_2^2n\sum_{j\geq 2} r_j^{-1/2}\sum_{1\leq i<j} r_i^{1/2}\beta_{i0}j.$$

Using (11), it is bounded by

$$\begin{aligned}9c_1c_2^2n\sum_{j\geq 2} r_j^{-1/2}jc_6r_{j-1}^{1/2}\log^2 r_{j-1}\log^{-1} r_1 &< \\ \frac{9}{4}c_1c_2^2c_6n\log^{-1} r_1\sum_{j\geq 2} jr_j^{-1/4}\log^2 r_j &< \\ O(nr_2^{-1/4}\log^2 r_2\log^{-1} r_1) &= \\ O(\frac{n}{\sqrt{r_1}}\log r_1)&\end{aligned}$$

- In the sixth term $27c_1c_2^2c_5^2n\sum_{j\geq 1} r_j^{-1/2}\log^2 r_1$, the sum is also bounded by its leading term, so we get $O(\frac{n}{\sqrt{r_1}}\log^2 r_1)$.

- Similarly, in the seventh term $9c_1c_2^2c_5n\sum_{j\geq 1} r_j^{-1/2}j\log r_1$, the sum is bounded by its leading term so we get $O(\frac{n}{\sqrt{r_1}}\log r_1)$.

□

Combining the total $O(\frac{n}{\sqrt{r}}\log^{c_q+1} r)$ process debt, and the total $O(\frac{n}{\sqrt{r}}\log^2 r)$ update debt, we get that the total running time of the algorithm is $O(\frac{n}{\sqrt{r}}\log^{c_q+1} r)$.

By Lemma 2, The parameter c_q is either 1 or 3, thus proving Theorem 1

5 Applications

In this section we give the details of three applications of our fast shortest-path algorithm. In these applications, we obtain a speedup over previous algorithms by decomposing a region of n vertices using an r-division and computing distances among the boundary vertices of the region in $O((n/\sqrt{r})\log^2 r)$ time using our fast shortest-path algorithm.

5.1 All-pair distances among boundary vertices of a small face

Let G be a directed planar graph, and let ϕ be a face of G with k vertices on its boundary. We consider the problem of computing the $O(k^2)$ distances among all k boundary vertices of ϕ. There are two previously known solutions for this problem. First, it is possible to compute the shortest-path tree from each of the k boundary vertices. This solution takes $O(nk)$ time using the shortest-path algorithm of Henzinger et al. [20]. Second, we can compute the distances by applying Klein's multiple-source shortest-path algorithm (MSSP) [28] in $O((n+k^2)\log n)$ time.

We use our fast shortest-path to prove the following Theorem.

Theorem 3. *Let G be a planar embedded directed graph with n vertices. Let f be a face in G with $k < \sqrt{n}/\log n$ vertices. One can compute the all-pair distances among the k vertices of f in $O(n \log k)$ time.*

Note that, for $k > \sqrt{n}/\log n$, the MSSP solution is upper bounded by $O((n+k^2)\log k)$ since $\log n = O(\log k)$. Therefore, we get an upper bound of $O((n+k^2)\log k)$ for computing the all-pair distances among the k boundary vertices of a given face, for any value of k.

We consider the entire graph G as a single region of an n-division of some other, larger graph (the large graph itself is not relevant for our algorithm). We define the face ϕ to be a hole of the region G, and the k vertices of ϕ to be boundary vertices of the region G. Note that our definition of the boundary of G is valid for a region in an n-division with a constant number of holes since we assume that $k < \sqrt{n}$.

We further decompose the region G using an r-division, for some value of r to be defined later. This decomposition maintains the property that the k vertices of ϕ are boundary vertices of the regions of the r-division that contain them. We compute this r-division in linear time using the algorithm of Klein et al. [30]. Then, we compute the DDG of the r-division in $O(n \log r)$ time, using the MSSP algorithm of Klein [28].

The k boundary vertices of ϕ are all vertices of the DDG. We apply our fast shortest-path algorithm from each of the k vertices, and retrieve the required all-pair distances among them. Applying our algorithm k times requires $O(k \cdot (n/\sqrt{r}) \log^2 r)$ time.

The total running time of our algorithm is $O(n \log r + k(n/\sqrt{r}) \log^2 r)$. We choose $r = k^2 \log^2 k$; note that since $k < \sqrt{n}/\log n$ we have that $r < n$, and an r-division of G is indeed defined. We get that the running time of our algorithm is $O(n \log k)$, as required.

5.2 Maximum flow when the source and the sink are close

In this section, we use our fast shortest-path algorithm to prove the following theorem.

Theorem 4. *Let G be a planar embedded directed flow network with source s and sink t. Let p be the minimum number of faces that a curve C from s to t passes through. One can compute a maximum st-flow in G in $O(n \log p)$ time.*

The parameter p was first introduced by Itai and Shiloach [21]. It can be equivalently defined as the minimum number of edges on a path in the dualg graph G^* a dual vertex incident to the face s^* to a dual vertex incident to the face t^*. Note that the parameter p depends on the specific embedding of the planar graph.

If $p = 1$ then the graph is *st-planar*. In this case, the maximum flow algorithm of Hassin [19], with the improvement of Henzinger et al. [20] runs in $O(n)$ time. For $p > 1$, Itai and Shiloach [21] gave an $O(np \log n)$ time maximum flow algorithm when the value of the flow is known. Johnson and Venkatesan [23] obtained the same running time without knowing the flow value in advance. Using the shortest-path algorithm of Henzinger et al. [20], it is possible to implement the algorithm of Johnson and Venkatesan in $O(np)$ time. Borradaile and Harutyunyan [2] gave another $O(np)$ time maximum flow algorithm for planar graphs. For undirected planar graphs, Kaplan and Nussbaum [26] showed how to find the minimum st-cut in $O(n \log p)$ time.

Our $O(n \log p)$ time bound matches the fastest previously known maximum flow algorithms for $p = \Theta(1)$ [23] and for $\log p = \Theta(\log n)$ [3], and is asymptotically faster than previous algorithms for other values of p. We assume that $p < \sqrt{n}/\log^3 n$, for larger values of p the $O(n \log n)$ maximum flow algorithm of Borradaile and Klein [3] already runs in $O(n \log p)$ time.

Preliminaries. A planar flow network consists of: (1) a directed planar graph G; (2) two vertices s and t of G designated as a *source* and a *sink*, respectively; and (3) a *capacity* function c defined over the arcs of G. We assume that for every arc uv, the arc vu is also in the graph (if this is not the case, then we add vu with capacity 0), such that the two arcs are embedded in the plane as a single edge. A *flow* function f assigns a flow value to the arcs of G such that: (1) for every arc uv, $f(vu) = -f(uv)$; (2) for every arc uv, $f(uv) \leq c(uv)$; and (3) for every vertex $v \neq s, t$, the amount of flow that enters v equals to the amount of flow that leaves v. A *maximum flow* is a flow in which the total amount of flow that enters the sink is maximal. In a *preflow* some of the vertices of G (in addition to t) may have more incoming flow than outgoing flow (an *excess*). A *maximum preflow* is a preflow in which the total amount of flow that enters the sink is maximal. Finally, we *add* a flow f' to a flow f by setting $f(e) \leftarrow f(e) + f'(e)$ for every arc e.

The *dual graph* G^* of G is is a planar graph such that every face ϕ of G has a vertex ϕ^* in G^*, and every arc e of G with a face ϕ_ℓ to its left and a face ϕ_r to its right has an arc $e^* = \phi_\ell^* \phi_r^*$ in G^*.

We now describe our maximum flow algorithm. We first describe an $O(np)$-time algorithm and then show how to improve it to $O(n \log p)$ using ideas of Borradaile et al. [4] together with our fast shortest-path algorithm.

An $O(np)$ algorithm. Without loss of generality, we may assume that the curve C from s to t crosses G only at vertices, say $v_0, v_1, v_2, \ldots, v_p$, where $v_0 = s$ and $v_p = t$. Therefore, we can embed in the graph a path P between s and t consisting of p edges $v_{i-1}v_i$ for $1 \leq i \leq p$ (note that some of the edges of P may be parallel to original edges of the graph). We set the capacities of the arcs of P to 0, so the value of the maximum flow is not affected by adding P to the graph.

Intuitively, our algorithm iterates over the vertices of P, from $v_0 = s$ until the last vertex before t, v_{p-1}. At each iteration, the algorithm pushes forward the excess flow of the current vertex of P to the vertices of P that follow it, where the excess of s is ∞, and the excess of any other vertex v_i of P is the excess flow remaining at v_i following the previous iterations. This is a special case of an algorithm of Borradaile et al. for balancing flow excesses and flow deficits of vertices along a path [4]. We obtain a speedup over the algorithm of Borradaile et al. by using our fast shortest-path algorithm, in a way similar to the one in Section 5.1, for computing distances among boundary vertices of a face.

Our algorithm works as follows. For every arc $v_i v_{i-1}$, for $1 \leq i \leq p$, we set $c(v_i v_{i-1}) = \infty$. We initialize a flow f by setting $f(e) = 0$ for every arc of the graph. Then, at iteration $i = 1, \ldots p$, we send a flow from v_{i-1} to v_i in the residual graph of f. The flow we send is as large as possible, but not larger than the excess of v_{i-1}. For $i = 1$, the excess of $v_0 = s$ is ∞. For $i > 1$, the excess of v_{i-1} is the value of the flow that we sent in the previous iteration from v_{i-2} to v_{i-1}. In fact, at iteration i we actually send as much flow as possible from v_{i-1} to *all* vertices of P that follow it. This is because each such vertex v_k is connected to v_i with a path of infinite capacity so any flow that we can send to v_k is routed to v_i. Note that each iteration is actually a flow computation in an st-planar graph, so we can implement in $O(n)$ time using Hassin's algorithm [19, 20]. After we compute the flow from v_{i-1} to v_i, we add it to f and continue to the next iteration.

Following the last iteration, f is a maximum preflow from s to t. Note that only vertices of P may have excess. We discard the arcs of P from the graph. Then, we convert the maximum preflow f to a maximum flow using the following procedure of Johnson and Venkatesan. First, we make the flow acyclic using the algorithm of Kaplan and Nussbaum [25] in $O(n)$ time. Then, we find a topological ordering of the vertices such that if an arc uv carries a positive amount of flow, then u precedes v in the ordering. Finally, in $O(n)$ time, we return the excess from vertices with an excess flow to s backwards along this topological ordering.

Since each iteration takes $O(n)$ time, the total running time of the algorithm is $O(np)$. The bottleneck of the algorithm is applying Hassin's maximum flow algorithm for st-planar graphs p times. We next use a technique of Borradaile et al. to execute Hassin's algorithm implicitly.

An $O(n \log p)$ algorithm. We start by reviewing Hassin's maximum flow algorithm for st-planar graphs. The input for the algorithm is a flow network G with a source s and a sink t, such that s and t are on the boundary of a common face. We embed an arc ts with infinite capacity in G. Let ϕ be the face of G on the left-hand side of ts. We compute shortest path distances in G^* from ϕ^*, where the length of a dual arc equals the capacity of the corresponding primal arc.

Let $\pi(\psi)$ denote the distance in G^* from ϕ^* to ψ^*. The function π, which we call a *potential function*, defines a *circulation* f_π in G as follows. For an arc e with a face ψ_ℓ to its left and a face ψ_r to its right, the flow along e is $f_\pi(e) = \pi(\psi_r) - \pi(\psi_\ell)$. Hassin showed that after we drop the extra arc ts, we remain with a maximum flow f_π from s to t in the original graph.

Following Borradaile and Harutyunyan we apply Hassin's algorithm with two modifications. First, we compute a maximum flow with a prescribed bound x on its value. We obtain such a flow by setting the capacity of ts to x rather than to ∞ [27]. Second, we do not wish to add and remove the arc ts, since we do not want to modify the embedding of the graph. In our case, the arc ts already exists in the graph, this is the arc $v_i v_{i-1}$ with capacity 0. Instead of adding a new arc, with capacity x, from the sink to the source, we set the capacity of $v_i v_{i-1}$ to x. Instead of removing the arc following the flow computation, we set the *residual capacity* of $v_i v_{i-1}$ and $v_{i-1} v_i$ back to 0, by setting their capacities to $f(v_i v_{i-1})$ and $-f(v_i v_{i-1})$, respectively.

We do not maintain a flow f throughout our algorithm, we rather use the potential function π to represent the flow f_π. This gives us the implementation of our maximum flow algorithm in Algorithm 4 below.

If we implement the shortest path computation at Line 6 using the algorithm of Henzinger et al., then the running time of our algorithm is $O(np)$. Borradaile et al. noticed that we do not have to compute and maintain the value of the potential function π for all faces of G prior to Line 11. Until this point, it suffices to compute the value of the potential function only for the faces incident to the arcs of P. This way, Borradaile et al. computed the distances in G^* using FR-Dijkstra on a DDG that contains the dual vertices of all of these faces, and executed Line 6 faster. We follow this approach, and compute the distances at Line 6 in a way similar to the algorithm of Section 5.1. Only after the last iteration of the loop that computes π, we compute explicitly the value of π for

Algorithm 4 MaximumFlow(G, s, t, c)

```
1: Embed the path $P = (v_0, v_1, \dots, v_p)$, where $v_0 = s$ and $v_p = t$, with $c(v_i v_{i-1}) = \infty$ and
   $c(v_{i-1} v_i) = 0$ for every $1 \le i \le p$
2: $\pi(\psi) \leftarrow 0$ for every face $\psi$ of $G$    // implicitly initialize $f_\pi(e) = 0$ for every arc $e$
3: *excess* $\leftarrow \infty$    // the excess of $s$ is $\infty$
4: **for** $i = 1$ to $p$ **do**
5:     $c(v_i v_{i-1}) \leftarrow$ *excess*    // bounds the amount of flow we send from $v_{i-1}$ to $v_i$
6:     $\pi_i(\psi) \leftarrow$ the distance from $\phi_i^*$ to $\psi^*$ in $G^*$, for every face $\psi$, where $\phi_i$ is the face to the left of
       $v_i v_{i-1}$, and for every arc $e$ the length of $e^*$ is the residual capacity of $e$ with respect to $f_\pi$ //
       implements Hassin's maximum flow algorithm
7:     $\pi(\psi) \leftarrow \pi(\psi) + \pi_i(\psi)$, for every face $\psi$    // adds the new flow to $f_\pi$
8:     $c(v_i v_{i-1}) \leftarrow f_\pi(v_i v_{i-1})$; $c(v_{i-1}, v_i) \leftarrow -f_\pi(v_i v_{i-1})$ // set the residual capacity of both arcs to
       0
9:     *excess* $\leftarrow f_\pi(v_i v_{i-1})$    // this is the value of the flow
10: **end for**
11: Discard the path $P$ from the graph
12: Convert the maximum preflow $f_\pi$ into a maximum flow $f$ // the implementation of this step is
    described above
13: **return** $f$
```

all faces of the graph. We now give the details of this improvement.

Consider the graph G^*, the dual graph of G (including the arcs of P). Let X^* be the set of vertices of G^* that correspond to faces of G incident to arcs of P. The set X^* is the set of vertices for which we compute distances at Line 6. We define the region R^* to be the region of G^* that contains the dual arcs of the original arcs of G, excluding the arcs of P. The boundary vertices of R^* are exactly the vertices of X^*, and they all lie on a single face of R^* (which may have other vertices on its boundary). Let P^* be the region of G^* that contains the arcs dual to arcs of P, these are the arcs of G^* that are not in R^*. See Figure 2.

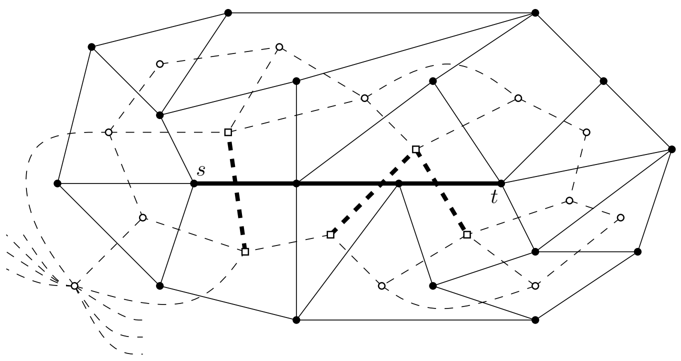

Figure 2: A flow network G (*solid* edges) and the dual network G^* (*dashed* edges). The path P from s to t is *bold*, the region P^* is *dashed and bold*. The set of dual vertices X^* (*boxes*) separates between P^* and R^* (the dual edges that are not in P^*), they are all on the boundary of a single face of R^*. To make the illustration simpler, some of the arcs incident to the dual vertex of the infinite face are omitted.

We decompose R^* using an r-division, similar to the way we do in Section 5.1, such that the vertices of X^* remain boundary vertices of every region containing them. We choose $r = p^2 \log^6 p$ for reasons that will become clear later. Note that $r < n$, so an r-division is well defined. The region P^* has $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices (actually it has much less vertices than r and much less boundary vertices than $\sqrt{r}$). Therefore, the regions of the r-division of R^* together with the region P^* are an r-division of G^*, in which all vertices of X^* are boundary vertices. We compute a DDG for this r-division in $O(n \log r)$ time.

Initially, the lengths of the arcs of R^* are defined by the capacities of the corresponding primal arcs. At each invocation of Line 7, we update the potential function π, and thereby also the residual capacities of the arcs of G. Since the length of a dual arc equals to the residual capacity of the corresponding primal arc, we update the lengths of the arcs of R as well. As Borradaile et al. noticed, we can use π also as a *price function* that represents the change in the lengths of the arcs of R^* and defines *reduced lengths*. That is, if the distance from $\psi_x^* \in X^*$ to $\psi_y^* \in X^*$ with respect to the original capacity of the graph is d, then the distance from ψ_x^* to ψ_y^* with respect to the residual capacities of f_π is $d - (\pi(\psi_y) - \pi(\psi_x))$. We conclude that we compute the DDGs of the regions that compose R^* only once and use π as a price function for these DDGs.

At Line 8 of the algorithm, we update the lengths of $v_i v_{i-1}$ and $v_{i-1} v_i$. This update causes a change in the DDG of P^*. Since P^* contains only p vertices, we can recompute the DDG in $O(p^2 \log p) = O(r)$ time.

We implement Line 6 by applying our fast shortest-path algorithm to the DDG of G^*. We use the variant of our algorithm that allows a price function and supports reduced lengths (i.e, the variant with $c_q = 3$ whose preprocessing time is $O((n/\sqrt{r}) \log^2 r)$ running time is $O((n/\sqrt{r}) \log^4 r)$, see Theorem 1 and Section 4). Thus, the total time for running the loop is $O(p(n/\sqrt{r}) \log^4 r)$. Since we set $r = p^2 \log^6 p$, the running time of the loop is $O(n \log p)$.

It remains to show how to obtain the value of the potential function π for all faces of G prior to Line 11. Recall that, for a face ψ of G, $\pi(\psi)$ is the distance in G^* from the dual vertex of the face ϕ_p to the left of $v_p v_{p-1}$ to ψ^*, where the length of each dual arc is defined by the capacity of the corresponding primal arc. Therefore, we can compute π using a shortest-path algorithm. We do not apply a shortest-path algorithm for the entire graph G^*, since the arcs dual to arcs of P may have negative lengths (which set the residual capacities of the primal arcs with respect to f_π to 0). Instead, we initialize the labels of the vertices of X^* according to the values of π that we have already computed, and apply the shortest-path algorithm of Henzinger et al. to R^*, which contains only non-negative length arcs. This is analogous to the way Borradaile et al. handle the same issue.

We conclude that the total running time of our algorithm is $O(n \log p)$ as required. The correctness of our algorithm follows from the correctness of the algorithm of Borradaile et al., since our algorithm is a special case of it. The differences between our algorithm and the one of Borradaile et al. are that we decompose the region R^* using an r-division and that we apply our fast shortest-path algorithm.

5.3 Minimum st-cut in undirected planar graphs

An *st-cut* in a flow network is a set of arcs whose removal disconnects t from s. A *minimum st-cut* is an st-cut with minimum total capacity. An *undirected* flow network is a flow network in which for every arc uv, the capacities $c(uv)$ and $c(vu)$ are equal. Prior to our work, the fastest algorithm for computing minimum st-cut in undirected planar graphs are the $O(n \log p)$ time algorithm of Kaplan and Nussbaum [26] and the $O(n \log \log n)$ time maximum flow algorithm of Italiano et al. [22]. The parameter p is the same parameter as in the previous section, the minimum number of faces that a curve C from s to t passes through. We combine ideas from both of these algorithms, together with our fast shortest-path algorithm to prove the following theorem.

Theorem 5. *Let G be a planar embedded undirected flow network with source s and sink t. Let p be the minimum number of faces that a curve C from s to t passes through. One can compute a minimum st-cut in G in $O(n \log \log p)$ time.*

Consider a planar graph G with a dual planar graph G^*. As in the previous section, we define the length of a dual arc in G^* to be equal to the capacity of the corresponding primal arc in G. A minimum st-cut in G corresponds to a shortest cycle in G^* that separates between the faces s^* and t^*. Such a cycle in G^* is called a *cut-cycle*. Itai and Shiloach [21] used cut-cycles to show the following minimum st-cut algorithm. Let Q be a shortest path in G^* from some vertex of the face s^* to some vertex of the face t^*. Since Q is a shortest path in G^*, the shortest cut-cycle in G^* cannot cross it more than once. Itai and Shiloach found, for every vertex q_i of Q, the shortest cut-cycle C_i that contains q_i. This is done by finding the shortest path that starts at q_i, leaves Q from its left side, returns to Q from its right side, and ends at q_i, without crossing the path Q. We implement the algorithm of Itai and Shiloach as follows. We make an *incision* along Q, that is we replace the path Q with two copies Q^ℓ and Q^r, every arc that was adjacent to a vertex q_i of Q becomes adjacent to the copy q_i^ℓ of q_i in Q^ℓ if the arc was adjacent to q_i from the left side of Q or to the copy q_i^r of q_i in Q^r if the arc was adjacent to q_i from the right side of Q. See Figure 3. For every vertex q_i of Q, we compute the shortest path from q_i^ℓ to q_i^r, this path corresponds to the cut-cycle C_i. The shortest among these cut-cycles is the shortest cut-cycle in the graph, which is dual to the minimum st-cut. The algorithm of Itai and Shiloach requires $|Q|$ shortest path computations in G^*.

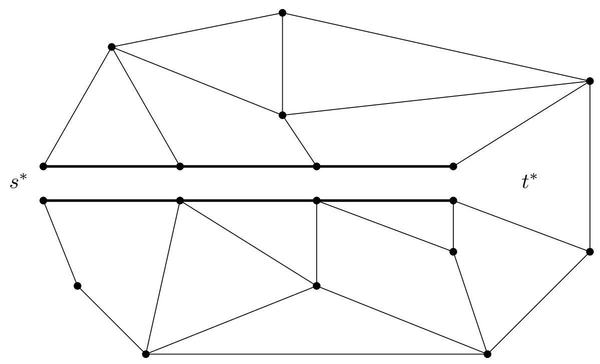

Figure 3: An *incision* along the *bold* path between s^* (the infinite face) and t^*. The incision splits the path into two paths, and merges s^* and t^* into a single face. All the vertices of the path are on the bounday of the new face.

Reif [38] improved the running time of the algorithm of Itai and Shiloach using a divide-and-conquer approach. Let q_i be the middle vertex of Q. We find the cut-cycle C_i as before, by computing a shortest path from q_i^ℓ to q_i^r. Reif observed that a cut-cycle C_j for any vertex q_j of Q with $j \neq i$ cannot cross C_i. This way, we can divide the problem of finding the shortest cut-cycle into two subproblems. In one subproblem we take the subgraph in the side of C_i that contains s^*, and find the shortest cut-cycle that separates between s^* and C_i. In the other subproblem we take the subgraph in the side of C_i that contains t^* and find the shortest cut-cycle that separates between C_i and t^*. The shortest cut-cycle in the graph is the shortest among C_i and the two shortest cut-cycles that we find for the two subproblems. Using the linear time shortest-path algorithm of Henzinger et al. [20], the running time of Reif's algorithm is $O(n \log |Q|)$. Note that with our algorithm from Section 5.1 we can implement the algorithm of Itay and Shiloach in $O(n \log |Q|)$ time as well, since the paths Q^ℓ and Q^r define a face in the graph, so we can compute the distances from every vertex of Q^ℓ to the corresponding vertex of Q^r in $O(n \log |Q|)$ time.

Reif's algorithm was improved in two ways. Kaplan and Nussbaum [26] gave an algorithm similar to Reif's algorithm that can use any (not necessarily shortest) path from s^* to t^* instead of the shortest path Q. In particular, we can use a path P of p edges from s^* to t^* in G^*, compute shortest cut-cycles for vertices of P, and get an $O(n \log |P|) = O(n \log p)$ time minimum st-cut algorithm. This algorithm works as follows. Since P is not a shortest path, it is not longer true that the shortest cut-cycle containing a vertex p_i of P, which we also denote by C_i, crosses P only once. However, it is true that C_i crosses P an odd number of times. We make an incision along P as before. We create an additional copy of the graph G^* following the incision and denote it by $\bar{G}^*$. For every vertex p_i of P, we identify the vertex p_i^ℓ in the path P_ℓ in G^* and the vertex $\bar{p}_i^r$ of the copy $\bar{P}_r$ of P_r in $\bar{G}^*$. Similarly, we identify the vertex p_i^r with $\bar{p}_i^\ell$. See Figure 4. The shortest cut-cycle C_i containing p_i corresponds to the shortest path from p_i^ℓ to $\bar{p}_i^r$. Indeed, the projection of the path to the original graph crosses P an odd number of times since every crossing of P corresponds to a move from G^* to $\bar{G}^*$ or from $\bar{G}^*$ to G^*. As in Reif's algorithm, after we find C_i, we divide the problem into two subproblems, one in each side of the cycle. In Reif's algorithm, one subproblem contains the vertices q_j with $j < i$, and the other subproblem contains the vertices q_j with $j > i$. Here this property does not hold, since C_i may cross P several times. However, Kaplan and Nussbaum showed that in the side of C_i that contains s^* we are interested only in finding C_j for $j < i'$, where

i' is the smallest index such that $p_{i'}$ is a vertex of C_i. Similarly, in the side of C_i that contains t^* we are interested only in finding C_j for $j > i''$, where i'' is the largest index such that $p_{i''}$ is a vertex C_i. See Figure 5. Hence, the *relevant* part of P in each subproblem has at most half of the vertices of P, and we get a recursive depth of $O(\log p)$ and an $O(n \log p)$ time minimum st-cut algorithm.

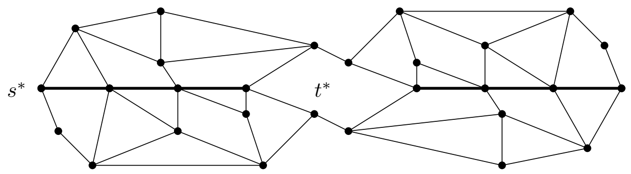

Figure 4: Two copies of the graph from Figure 3, G^* from above and $\bar{G}^*$ below. We identify the left copy of the *bold path* between s^* and t^* in G^* with the right copy of the path in $\bar{G}^*$ and vice versa. This construction is called a *cyclic double cover*.

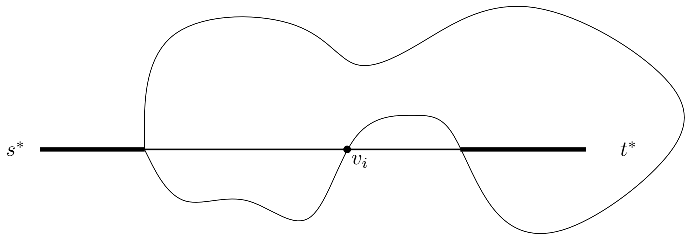

Figure 5: The shortest cut-cycle containing v_i, C_i, crosses the path P between s^* and t^* *three* times. In the two recursive call, in each side of C_i, only the *bold* part of P is relevant.

The other improvement of Reif's algorithm is by Italiano et al. [22]. They improved the running time of Reif's algorithm to $O(n \log \log n)$ by using FR-Dijkstra instead of Dijkstra's algorithm to compute some of the cut-cycles. The algorithm begins by computing an r-division with $r = \log^6 n$. This value of r was chosen so that computing the DDG of the r-division takes $O(n \log \log n)$ time, and applying FR-Dijkstra to it takes $O(n/ \log n)$ time. Then, we make an incision along Q, the shortest path from s^* to t^*. For every vertex q_i of Q that was a boundary vertex of the r-division prior to the incision, we define both of its copies q_i^ℓ and q_i^r to be boundary vertices. We compute the DDG of the r-division following the incision. We apply Reif's algorithm in two phases. In the first phase, called the *coarse* phase, we apply the algorithm to the DDG. That is, for every vertex q_i of Q that is a boundary vertex, we find the shortest path from q_i^ℓ to q_i^r using FR-Dijkstra. Since applying FR-Dijkstra takes $O(n/ \log n)$ time, and the recursion depth is $O(\log n)$, we compute all these cut-cycles in $O(n)$ time. In the second phase, called the *refined* phase, we use the set of cut-cycles that we found in the first phase to divide the graph into smaller problems. Let q_i and q_j be two consecutive boundary vertices in Q. For every q_k with $i < k < j$, the cut-cycle C_k is bounded by the cut-cycles C_i and C_j, which we found in the coarse phase. This way, q_i and q_j define a subgraph to which we apply Reif's standard algorithm. There is also a subgraph that is bounded by the cut-cycle of the first boundary vertex of Q and a subgraph bounded by the cut-cycle of the last boundary vertex of Q, we apply Reif's algorithm to these two subgraphs as well. Since the size of each piece is at most $r = \log^6 n$ vertices, the subpath of Q in each subgraph has at most $\log^6 n$ vertices. Therefore, the total time for applying the refined phase of the algorithm is $O(n \log\log n)$.

We now describe our $O(n \log\log p)$ time algorithm, which combines ideas from the algorithms described above. Let P be a path of p edges from s^* to t^*, as in the algorithm of Kaplan and Nussbaum. We find an r-division of G^* with $r = \log^6 p$. This value of r allows us to apply our fast shortest path algorithm to the DDG in $O(n/\log p)$ time. This is where the minimum st-cut algorithm benefits from our result. Similarly to the algorithm of Italiano et al., we make an incision along P, and for every vertex p_i of P that was a boundary vertex prior to the incision, we define both of its copies p_i^ℓ and p_i^r to be boundary vertices. We compute the DDG of the r-division, denote it by H. We create an additional copy $\bar{H}$ of H, and for every vertex p_i of P we identify p_i^ℓ with $\bar{p}_i^r$ and p_i^r with $\bar{p}_i^\ell$, where $\bar{p}_i^r$ and $\bar{p}_i^\ell$ are the copies of p_i^r and p_i^ℓ respectively in $\bar{H}$ (this is similar to Figure 4). The first phase of our algorithm is the coarse phase, in which we find the cut-cycles of the vertices of P that are boundary vertices. Let p_i be the middle vertex among these boundary vertices. We find the shortest path from p_i^ℓ to $\bar{p}_i^r$ using our fast shortest path algorithm in $O(n/\log p)$ time. This shortest path corresponds to the cut-cycle C_i. We use C_i to divide the DDG H into two subgraphs. One is on the side of the cycle that contains s^*, in which we look for C_j only for $j < i'$, where i' is the smallest index such that $p_{i'}$ is a vertex of C_i. The other is on the side of C_i that contains t^*, in which we look for C_j only for $j > i''$, where i'' is the largest index such that $p_{i''}$ is a vertex of C_i (this is similar to the situation in Figure 5). Note that we do not use the shortest path from p_i^ℓ to $\bar{p}_i^r$ to divide the union of H and $\bar{H}$, this path does not divide the union of the DDG into two separate subgraphs. Instead, we use the projection of this path in H, which is the cut-cycle C_i, to divide H, and maintain the property that $\bar{H}$ is a copy of H. We divide the DDG as in the algorithm of Italiano et al. This takes a total of $O(n \log\log p)$ for all division of the DDG.[9] The depth of the recursion of the coarse phase of the algorithm is $O(\log p)$, since the relevant part of P in each subproblem has at most half of the vertices of P, and we conclude that the total time for the coarse phase is $O(n \log\log p)$.

We now continue to the refined phase of our algorithm. The set of cut-cycles that we found in the coarse phase divides the input graph G into smaller subgraphs. We can obtain this division into subgraphs in $O(n)$ time, as described by Italiano et al. [22]. We apply the algorithm of Kaplan and Nussbaum to each of these subgraphs. In the algorithm of Italiano et al. we knew that there are at most r vertices of the path Q (where r is the parameter of the r-division, and Q is a shortest path from s^* to t^*) in each subgraph, since we computed a cut-cycle for every vertex of Q that is a boundary vertex. In our algorithm, in contrast, there may be more than r vertices of P in each subgraph, since we do not compute a cut-cycle for every boundary vertex of P in the coarse phase. However, for two consecutive cut-cycles from the first phase, C_i and C_j, all the vertices of P that remain relevant, that is vertices p_k with $i < k < j$ such that we want to compute C_k, are consecutive along P. Therefore, there are at most $r = \log^6 p$ relevant vertices in each subproblem in the refined step. Hence, the total time for the refined step is $O(n \log\log p)$. We conclude that the total running time of our algorithm is $O(n \log\log p)$ as required.

[9]Italiano et al. omitted the details of this step from the conference version of their result [22]. This step is implemented by identifying the edges of the DDG that cross the cycle that divides the DDG. For each piece that the cut-cycle crosses, we label the two sides of the cycle in two different labels concurrently, and halt when one of the sides is fully labeled. The edges that cross the cycle are those that their two endpoint have different labels (or that only one of the endpoints is labeled). We divide the DDG by setting the length of these edges to ∞. We charge the labelling on the smaller part of the piece. Since the initial size of the piece is $r = \log^6 p$, the total times for all divisions is $O(n \log r) = O(n \log\log p)$.

6 Acknowledgements

We thank Philip Klein for discussions and for allowing us to use large parts of his description [29] of the algorithm of Henzinger et al. [20] as the basis for our description in section 4.

References

[1] L. Arge, F. van Walderveen, and N. Zeh. Multiway simple cycle separators and I/O-efficient algorithms for planar graphs. In *Proceedings of the 24th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 901–918, 2013.

[2] G. Borradaile and A. Harutyunyan. Maximum st-flow in directed planar graphs via shortest paths. In *Combinatorial Algorithms*, pages 423–427. Springer, 2013.

[3] G. Borradaile and P. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *Journal of the ACM*, 56(2):1–30, 2009.

[4] G. Borradaile, P. N. Klein, S. Mozes, Y. Nussbaum, and C. Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM J. Comput.*, 46(4):1280–1303, 2017.

[5] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen. Min st-cut oracle for planar graphs with near-linear preprocessing time. *ACM Trans. Algorithms*, 11(3):16:1–16:29, 2015.

[6] S. Cabello. Many distances in planar graphs. *Algorithmica*, 62(1-2):361–381, 2012. Preliminary version in SODA 2006.

[7] S. Cabello, E. Chambers, and J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.

[8] P. Chalermsook, J. Fakcharoenphol, and D. Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In *Proceedings of the 14th annual Symposium On Discrete Algorithms (SODA)*, pages 828–829, 2004.

[9] H.-C. Chang and H.-I. Lu. Computing the girth of a planar graph in linear time. *SIAM Journal on Computing*, 42(3):1077–1094, 2013.

[10] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[11] D. Eisenstat and P. N. Klein. Linear-time algorithms for max flow and multiple-source shortest paths in unit-weight planar graphs. In *Proceedings of the 45th ACM Symposium on Theory of Computing, (STOC)*, pages 735–744, 2013.

[12] J. Erickson. Maximum flows and parametric shortest paths in planar graphs. In *Proceedings of the 21st annual Symposium On Discrete Algorithms (SODA)*, pages 794–804, 2010.

[13] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72:868–889, 2006.

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

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[16] P. Gawrychowski and A. Karczmarz. Improved bounds for shortest paths in dense distance graphs. arXiv:1602.07013, 2016.

[17] P. Gawrychowski, S. Mozes, and O. Weimann. Improved submatrix maximum queries in Monge matrices. arXiv:1307.2313, 2013.

[18] P. Gawrychowski, S. Mozes, and O. Weimann. Improved submatrix maximum queries in Monge matrices. In *Proceedings of the 41st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 525–537, 2014.

[19] R. Hassin. Maximum flow in (s,t) planar networks. *Inform. Process. Lett.*, 13(3):107, 1981.

[20] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[21] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.

[22] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, (STOC)*, pages 313–322, 2011.

[23] D. Johnson and S. M. Venkatesan. Partition of planar flow networks. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 259–264, 1983.

[24] H. Kaplan, S. Mozes, Y. Nussbaum, and M. Sharir. Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications. *ACM Trans. Algorithms*, 13(2):26:1–26:42, 2017.

[25] H. Kaplan and Y. Nussbaum. Maximum flow in directed planar graphs with vertex capacities. *Algorithmica*, 61(1):174–189, 2011.

[26] H. Kaplan and Y. Nussbaum. Minimum s-t cut in undirected planar graphs when the source and the sink are close. In *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 117–128, 2011.

[27] S. Khuller, J. Naor, and P. Klein. The lattice structure of flow in planar graphs. *SIAM J. Discret. Math.*, 6(3):477–490, 1993.

[28] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 146–155, 2005.

[29] P. N. Klein and S. Mozes. *Optimization Algorithms for Planar Graphs.* http://planarity.org/, 2014.

[30] P. N. Klein, S. Mozes, and C. Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Proceedings of the 44th ACM Symposium on Theory of Computing, (STOC)*, pages 505–514, 2012.

[31] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. *ACM Transactions on Algorithms*, 6(2):1–18, 2010. Preliminary version in SODA 2009.

[32] J. Łącki and P. Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Proceedings of the 19th annual European Symposium on Algorithms (ESA)*, pages 155–166, 2011.

[33] T. Matsui. The minimum spanning tree problem on a planar graph. *Discrete Applied Mathematics*, 58(1):91–94, 1995.

[34] S. Mozes, C. Nikolaev, Y. Nussbaum, and O. Weimann. Minimum cut of directed planar graphs in $O(n \log \log n)$ time. In *Proceedings of the 29th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, 2018. To appear.

[35] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *Proceedings of the 23rd ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 571–582, 2012.

[36] S. Mozes and C. Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$. In *Proceedings of the 18th annual European Symposium on Algorithms (ESA)*, pages 206–217, 2010.

[37] Y. Nussbaum. Improved distance queries in planar graphs. In *Proceedings of the 14th International Workshop on Algorithms and Data Structures (WADS)*, pages 642–653, 2011.

[38] J. Reif. Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM Journal on Computing*, 12:71–81, 1983.

[39] S. Subramanian. *Parallel and Dynamic Shortest-Path Algorithms for S parse Graphs*. PhD thesis, Brown University, 1995. Available as Brown University Computer Science Technical Report CS-95-04.

[40] S. Tazari and M. Müller-Hannemann. Shortest paths in linear time on minor-closed graph classes, with an application to steiner tree approximation. *Discrete Applied Mathematics*, 157(4):673–684, 2009.

[41] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.