Single-Source Shortest Paths and Strong Connectivity in Dynamic Planar Graphs

Panagiotis Charalampopoulos
Department of Informatics, King’s College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Adam Karczmarz
Institute of Informatics, University of Warsaw, Poland
a.karczmarz@mimuw.edu.pl

Abstract

Efficient algorithms for computing and processing additively weighted Voronoi diagrams on planar graphs have been instrumental in obtaining several recent breakthrough results, most notably the almost-optimal exact distance oracle for planar graphs [Charalampopoulos et al., STOC’19], and subquadratic algorithms for planar diameter [Cabello, SODA’17, Gawrychowski et al., SODA’18]. In this paper, we show how Voronoi diagrams can be useful in obtaining dynamic planar graph algorithms and apply them to classical problems such as dynamic single-source shortest paths and dynamic strongly connected components.

First, we give a fully dynamic single-source shortest paths data structure for planar weighted digraphs with \( \tilde{O}(n^{4/5}) \) worst-case update time and \( O(\log^2 n) \) query time. Here, a single update can either change the graph by inserting or deleting an edge, or reset the source \( s \) of interest. All known non-trivial planarity-exploiting exact dynamic single-source shortest paths algorithms to date had polynomial query time. Further, note that a data structure with strongly sublinear update time capable of answering distance queries between all pairs of vertices in polylogarithmic time would refute the APSP conjecture [Abboud and Dahlgaard, FOCS’16].

Somewhat surprisingly, the Voronoi diagram based approach we take for single-source shortest paths can also be used in the fully dynamic strongly connected components problem. In particular, we obtain a data structure maintaining a planar digraph under edge insertions and deletions, capable of returning the identifier of the strongly connected component of any query vertex. The worst-case update and query time bounds are the same as for our single-source distance oracle. To the best of our knowledge, this is the first fully dynamic strong-connectivity algorithm achieving both sublinear update time and polylogarithmic query time for an important class of digraphs.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Theory of computation → Shortest paths

Keywords and phrases dynamic graph algorithms, planar graphs, single-source shortest paths, strong connectivity

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.31

Funding Panagiotis Charalampopoulos: Partially supported by ERC Starting Grant TOTAL under the EU’s Horizon 2020 Research and Innovation Programme (agreement no. 677651). Adam Karczmarz: Supported by ERC Consolidator Grant 772346 T UgboAAT and the Polish National Science Centre 2018/29/N/ST6/00757 grant.
1 Introduction

The dynamic shortest paths problem seeks for a data structure maintaining a graph under updates and supporting shortest path queries.\footnote{We will identify shortest paths queries with distance queries. Almost all known dynamic shortest paths algorithms (for some exceptions see \cite{74,80}) can also report the actual path in nearly linear (in the number of the path’s edges) time after computing a distance estimate.} Depending on the set of supported updates, we call such a graph data structure \textit{fully dynamic} if both edge insertions and deletions are allowed, \textit{incremental} if only edge insertions (or edge weight decreases) are supported, or \textit{decremental} if only edge deletions (or weight increases) are allowed. In the \textit{all-pairs} variant of dynamic shortest paths problem one has to support shortest path queries between any pair of vertices of the graph. In the \textit{single-source} variant all shortest paths queries have to originate in a fixed distinguished vertex and the only parameter of a query is the target vertex.

For the most general setting where one requires \textit{exact} answers, Demetrescu and Italiano \cite{31} gave a fully dynamic algorithm (improved slightly by Thorup \cite{77}) recomputing the all-pairs shortest paths matrix in nearly optimal $O(n^2)$ amortized time even if real edge weights are allowed. Note that recomputing the distance matrix from scratch takes $O(nm) = O(n^3)$ time \cite{55}. Fully dynamic all-pairs shortest paths data structures with subcubic \textit{worst-case} update bounds are also known \cite{5,46,78}. There exist faster algorithms if the input graph is unweighted and \textit{partially dynamic} (i.e., incremental or decremental) \cite{6,7}. However, none of the known results improves upon a trivial, recompute-from-scratch algorithm with $O(mn)$ update time and $O(1)$ query time if the graph is \textit{sparse}, i.e., $m = O(n)$. For the single-source variant, all known non-trivial exact dynamic shortest paths algorithms \cite{37} are partially dynamic and yield no improvement over the respective recompute-from-scratch algorithm in the sparse case either.

The lack of progress on obtaining an exact fully dynamic single-source shortest paths algorithms with $O(n^{3-\epsilon})$ initialization time, $O(m^{1-\epsilon})$ amortized update time and $O(n^{1-\epsilon})$ query time at the same time can be explained by a matching lower bound conditional on the (static) APSP conjecture \cite{73}. In fact, breaking this barrier even in the partially dynamic setting for undirected unweighted graphs would be a large breakthrough \cite{43}.

As a result, since finding good exact algorithms for general graphs seems hopeless, one needs to look for either approximate solutions or restrict their attention to more structured graph classes. Indeed, a large body of research has been devoted to designing \textit{approximate} dynamic shortest paths algorithms, especially in partially dynamic settings \cite{9-13,15,26,29,43-45,48-50,59,60}, which find many applications, e.g., in various maximum flow related problems \cite{29,67}. Unfortunately, many of the known fastest approximate dynamic shortest path algorithms (e.g. \cite{9,44,48}) suffer from assuming an oblivious adversary, which significantly limits their applicability (cf. e.g., \cite{29}).

Similarly, dynamic shortest paths problems have also been studied for important graph classes like planar graphs \cite{3,4,38,56,63,66}, or low treewidth-graphs \cite{3,58}. The primary reason why faster dynamic shortest paths algorithms in these cases are possible is the existence of non-trivial \textit{distance oracles} for these classes. A distance oracle is a compact representation of the graph’s shortest paths such that the distance (or a distance estimate) between any pair of vertices can be retrieved efficiently. For general graphs, such non-trivial distance oracles exist only for undirected graphs and assuming an approximation ratio of at least $3$ \cite{25,79,82}. On the contrary, for planar graphs many non-trivial exact distance oracles
have been proposed \[19,27,33,38,56,63,69\]. The first exact oracles with polylogarithmic query time and subquadratic space have been obtained only recently \[23,30,41\], following Cabello’s breakthrough of employing Voronoi diagrams for the planar diameter problem \[20\]. Also near-optimal (in terms of query time, construction time, and used space) \((1 + \epsilon)\)-approximate distance oracles have been known for nearly two decades \[62,76\], and a lot of effort has been put to push the known bounds as close to optimal as possible \[22,42,61,84\].

### Dynamic shortest paths in planar graphs.

In this paper our focus is on computing shortest paths in dynamic planar graphs and applications. Klein and Subramaniam were the first to give a planarity-exploiting dynamic shortest paths algorithm \[66\] – their data structure worked for undirected graphs, was fully dynamic, \((1 + \epsilon)\)-approximate and had \(O(n^{2/3})\) update and query time bounds. A data structure with the same bounds (up to polylogarithmic factors), but for exact distances in directed graphs was obtained in the breakthrough work of Fakcharoenphol and Rao \[38\] (later extended and slightly improved in \[24,40,54,56,63\]). Abraham et al. \[4\] gave a faster \((1 + \epsilon)\)-approximate dynamic algorithm for undirected graphs with \(\widetilde{O}(n^{1/2})\) update and query times. Karczmarz \[58\] matched this bound for directed planar graphs, albeit only in the \((1 + \epsilon)\)-approximate decremental setting. Abboud and Dahlgaard \[1\] showed that by the APSP conjecture, one should not expect an exact dynamic all-pairs shortest paths data structure for planar graphs with strongly sublinear product of update time and query time. However, no exact data structure to date has matched this product lower bound while retaining strongly sublinear update time.

The single-source scenario is much less studied for dynamic planar graphs. Karczmarz \[58\] showed a decremental \((1 + \epsilon)\)-approximate single-source shortest paths algorithm for minor-free (and thus also planar) digraphs with \(\widetilde{O}(n^{1/2})\) update time and \(O(1)\) query time. Although not explicitly stated in the literature, the all-pairs data structure of \[63\] can be easily converted to a fully dynamic exact single-source distance oracle with \(\widetilde{O}(n^{2/3})\) update time and \(\widetilde{O}(n^{1/3})\) query time. However, no fully dynamic single-source shortest paths algorithm for planar graphs to date has been able to achieve sublinear update time and polylogarithmic query time, or at least break through the \(\widetilde{O}(n)\) update-query time product barrier, even in the approximate setting.

### Our results.

In this paper we show the first exact dynamic single-source shortest paths algorithm for planar graphs with strongly sublinear update time and polylogarithmic query time. Our algorithm, summarized by the following theorem and described in Section 3, is deterministic and can be easily extended to report paths.

\[\textbf{Theorem 1.} \text{ Let } G \text{ be a real-weighted planar digraph with a source } s \in V(G). \text{ There exists an } O(n \log n) - \text{space data structure maintaining } G \text{ under edge insertions, edge deletions, and source changes with } O(n^{4/5} \log^2 n) \text{ worst-case update time that can compute } dist_G(s,v) \text{ for any } v \in V(G) \text{ in } O(\log^2 n) \text{ time. The initialization time is } O(n \log^2 n).\]

To the best of our knowledge, this result constitutes the first known application of additively weighted Voronoi diagrams machinery (first introduced by Cabello \[20\]) in dynamic graph algorithms. More specifically, it is obtained by combining fully dynamic maintenance of \(r\)-divisions \[66,75\], the shortest paths algorithm for dense distance graphs \[38\], the recent efficient construction of dual Voronoi diagrams \[23\] via FR-Dijkstra \[38\], and the efficient point location data structure for Voronoi diagrams \[41\].

We now provide a brief overview of the data structure underlying Theorem 1. We maintain distances in \(G\) from the source vertex \(s\) to each boundary vertex of each piece of an \(r\)-division of \(G\) using FR-Dijkstra. For each piece of the \(r\)-division, we maintain an additively weighted...
Voronoi diagram augmented with a point location data structure, with weights equal to the distances from \( s \). Upon a query for \( \text{dist}_G(s, v) \), we perform a point location query on the Voronoi diagram of a piece of the \( r \)-division that contains \( v \).

It is worth noting that our data structure (and in fact all data structures obtained in this paper) works in the most general model of dynamic planar graphs where we only require that \( G \) remains planar after each update. Some fully dynamic planar graph algorithms assume a weaker \textit{plane} model (e.g., \cite{32,35,54}) where some plane embedding of \( G \) is fixed and we only allow inserting edges connecting vertices that lie on a common face of (that embedding of) \( G \).

We also generalize our single-source data structure to the case when, instead of a single source \( s \), a set of facilities \( F \subseteq V \) is given, and our goal is to locate the closest (i.e., minimizing \( \text{dist}_G(f, v) \)) facility \( f \in F \) for a given query vertex.\(^2\) We show that maintaining such a data structure under edge updates issued to \( G \), or updates to the facilities set \( F \), is possible using \( \tilde{O}(n^{3/4} \cdot |F|^{1/4} + n^{1/5}) \) worst-case update time. The query time remains \( O(\log^2 n) \). Note that even though multiple-source shortest paths or maximum flow problems can be typically easily reduced to the single-source case by adding a super-source, such a reduction does not preserve planarity and indeed handling multiple sources tends to be challenging in planar graphs (cf. e.g., \cite{16,17}). Our generalized data structure handles up to \( O(n^{1/5}) \) sources as efficiently as the single-source case. Moreover, the update time remains strongly sublinear unless the number of facilities is not strongly sublinear.

Surprisingly, we show that the same framework that we use to prove Theorem 1 can be applied to obtain interesting results not directly related to the shortest paths problem. Namely, in Section 4 we show a fully dynamic strong-connectivity algorithm for planar graphs, encapsulated in the following theorem.

\textbf{Theorem 2.} Let \( G \) be a planar digraph. There exists an \( O(n \log n) \)-space data structure maintaining \( G \) under edge insertions and deletions with \( O(n^{4/5} \log^2 n) \) worst-case update time that can compute the identifier of the strongly connected component of any \( v \in V(G) \) in \( O(\log^2 n) \) time. The initialization time is \( O(n \log^2 n) \).

We now sketch the main ideas behind our fully dynamic strong-connectivity algorithm. As in Subramanian’s dynamic all-pairs reachability algorithm \cite{75}, the base of our data structure is a graph \( X \), called a \textit{reachability certificate}, that sparsifies the reachability information between boundary vertices \( \partial R \) of a fully dynamic \( r \)-division \( R \) with few holes. Naively recomputing the strongly connected components of \( X \) gives us the restriction of the strongly connected components of \( G \) to the boundary vertices \( \partial R \). The main challenge, of course, is to compute the identifier of a strongly connected component (SCC) of an arbitrary non-boundary vertex \( v \) of \( G \), internal to some piece \( P \) of the \( r \)-division \( R \). To this end, we use the following observation: suppose \( b_1, \ldots, b_k \) are some vertices of \( G \) lying in distinct strongly connected components of \( G \). Then, \( v \) is strongly connected to some \( b_j \) if and only if \( b_j \) is in the topologically earliest SCC of \( G \) reachable from \( v \) and \( b_j \) is in the topologically latest SCC of \( G \) that can reach \( v \). Roughly speaking, this observation applied to the boundary vertices of \( P \) labeled using the topological order of their respective SCCs in the certificate \( X \), allows us to identify the SCC of \( v \) using two point-location queries on the Voronoi diagram of piece \( P \). Each such point location query, computes, instead of the nearest site of \( v \), the highest (or lowest) priority site that can reach \( v \) (that \( v \) can reach, resp.), and can be simulated using a standard point location query on a Voronoi diagram \cite{41}.

---

\(^2\) One can also view \( F \) as a set of sites of a graphic Voronoi diagram – then the query locates the cell of the Voronoi diagram wrt. \( F \) that a given vertex \( v \) belongs to.
Whereas maintaining strongly connected components is a well-studied problem in partially dynamic settings [8,14,47,53], we are not aware of any non-trivial fully dynamic strongly connected components data structures designed specifically for this problem for any digraph class – note that one could use a fully dynamic transitive closure data structure for this task: for example, the dynamic plane transitive closure data structure of [32] which has $\tilde{O}(n^{1/2})$ update and query time. Such a strongly connected components data structure (i.e., with both update and query bounds $O(n^{1-\epsilon})$) for general graphs is in fact ruled out by a conditional (on SETH) lower bound [2]. As a result, to the best of our knowledge, we obtain the first fully dynamic strongly connected components algorithm to achieve sublinear update-query time product for any important class of digraphs.

The undirected counterpart of the dynamic strongly connected components problem, the dynamic connectivity problem, is very well-studied. Near-optimal deterministic amortized update bounds [51,52,83] and randomized worst-case update bounds [57,81] (see also [71]) are known for fully dynamic general graphs. An almost optimal deterministic worst-case update bound was very recently achieved in [28]. For fully dynamic planar graphs polylogarithmic worst-case update bounds are known to be achievable even deterministically [34].

2 Preliminaries

Throughout the paper we consider as input a simple, directed and weighted planar graph $G$ with $n$ vertices, and no negative weight cycles. We call a planar graph $G$ plane if some embedding of $G$ is assumed. We use $|G|$ to denote the number of vertices of $G$. Since simple planar graphs are sparse, $|E(G)| = O(|G|)$ as well.

We use the terms weight and length for edges and paths interchangeably throughout the paper. For any two vertices $u,v \in V(G)$, we denote by $\text{dist}_G(u,v)$ the length of some shortest $u \to v$ path in the graph $G$.

Multiple-source shortest paths. The multiple-source shortest paths (MSSP) data structure [21,63] represents all shortest path trees rooted at the vertices of a single face $f$ in a weighted plane digraph using a persistent dynamic tree. It can be constructed in $O(n \log n)$ time, requires $O(n \log n)$ space, and can report any distance between a vertex of $f$ and any other vertex in the graph in $O(\log n)$ time. MSSP can be augmented to also return the first edge of this path (and each of its subsequent edges) in $O(\log \log n)$ time (cf. [56]).

Separators and recursive decompositions. Miller [68] showed how to compute, in a triangulated plane graph with $n$ vertices, a simple cycle of size $2\sqrt{2}\sqrt{n}$ that separates the graph into two subgraphs, each with at most $2n/3$ vertices. Simple cycle separators can be used to recursively separate a planar graph until pieces have constant size. The authors of [64] show how to obtain a complete recursive decomposition tree $T(G)$ of a triangulated graph $G$ using cycle separators in $O(n)$ time. $T(G)$ is a binary tree whose nodes correspond to subgraphs of $G$ (pieces), with the root being all of $G$ and the leaves being pieces of constant size. We identify each piece $P$ with the node representing it in $T(G)$. We can thus abuse notation and write $P \in T(G)$. The boundary vertices $\partial P$ of a non-leaf piece $P$ are vertices that $P$ shares with some other piece $Q \in T(G)$ that is not $P$'s ancestor. For convenience we extend the boundary set $\partial L$ of a leaf piece $L$ to its entire vertex set $V(L)$. We assume $P$ to inherit the embedding of $G$. The faces of $P$ that are faces of $G$ are called natural, whereas the faces of $P$ that are not natural are the holes of $P$. The construction of [64] additionally guarantees that for each piece $H \in T(G)$, (a) $H$ is connected, (b) if $H$ is non-leaf, then each natural
face \( f \) of \( H \) is a face of a unique child of \( H \), (c) \( H \) has \( O(1) \) holes containing precisely the vertices \( \partial H \). Throughout, to avoid confusion, we use nodes when referring to \( T(G) \) and vertices when referring to \( G \) or its subgraphs. It is well-known [18, 41, 53, 64] that by suitably choosing cycle separators one can also guarantee that (1) \( \sum_{H \in T(G)} |H| = O(n \log n) \), (2) \( \sum_{H \in T(G)} |\partial H|^2 = O(n \log n) \), and (3) \( |\partial H| = O(\sqrt{n}/c^d) \), where node \( H \) of \( T(G) \) has depth \( d \) and \( c > 1 \) is some constant.

The recursive decomposition algorithm of [64] works with no changes and maintains all the properties of \( T(G) \) even if the initial graph \( G \) has a predefined set of boundary vertices \( \partial G \) of size \( O(\sqrt{|G|}) \) located on \( O(1) \) of \( G \)'s faces. These faces are predefined as holes of \( G \) and are the only faces of \( G \) that are allowed to be non-triangular.

An \( r \)-division [39] \( R \) of a planar graph, for \( r \in [1, n] \), is a decomposition of the graph into \( O(n/r) \) pieces, each of size \( O(r) \), such that each piece \( P \) has \( O(\sqrt{r}) \) boundary vertices (denoted \( \partial P \)), i.e., vertices shared with some other piece of \( R \). We denote by \( \partial R \) the set \( \bigcup_{P \in R} \partial P \). If additionally all pieces are connected, and the boundary vertices of each piece \( P \) of the \( r \)-division \( R \) are distributed among \( O(1) \) faces of \( P \) (also called holes\(^3\) of \( P \)), we call \( R \) an \( r \)-division with few holes.

In [64] it was shown that for every \( r \) larger than some constant, \( T(G) \) admits an \( r \)-division with few holes, i.e., there exists a subset of nodes of \( T(G) \) forming an \( r \)-division with few holes of \( G \). Using this property, it is shown in [64] that an \( r \)-division with few holes of a triangulated graph can be computed in linear time. More generally, given a geometrically decreasing sequence of numbers \( (r_m, r_{m-1}, \ldots, r_1) \), where \( r_1 \) is a sufficiently large constant, \( r_{i+1}/r_i \geq b \) for all \( i \) for some \( b > 1 \), and \( r_m = n \), we can obtain \( r_i \)-divisions with few holes for all \( i \) in time \( O(n) \). For convenience, we define the only piece in the \( r_m \)-division to be \( G \) itself. These \( r \)-divisions satisfy the property that a piece in the \( r \)-division is a – not necessarily strict – descendant (in \( T(G) \)) of a piece in the \( r_j \)-division for each \( j > i \). We also call such sequence of \( r_i \)-divisions obtained from \( T(G) \) a recursive \((r_m, \ldots, r_1)\)-division of \( G \).

We assume for simplicity that all holes we ever encounter are simple cycles. Unfortunately, this is not true in general. However, non-simple holes do not pose a significant obstacle, and can be avoided by suitably extending the graphs, as discussed numerous times in the past, see e.g., [23, 53, 56, 72].

**Dense distance graphs and FR-Dijkstra.** For a plane digraph \( H \) with weights from \( \mathbb{R}_{\geq 0} \cup \{\infty\} \) and a distinguished set \( \partial H \subseteq V(H) \) of boundary vertices lying on \( O(1) \) faces of \( H \), we denote by \( DDG_H \) the complete weighted graph on \( \partial H \) whose edge weights represent distances between all pairs of vertices of \( \partial H \) in \( H \). \( DDG_H \) can be computed in \( O((|H| + |\partial H|^2) \log n) \) time using MSSP [63]. In particular, dense distance graphs for all pieces \( H \in T(G) \) can be computed in \( O(n \log^2 n) \) time.

When \( H = \{H_1, \ldots, H_q\} \) is a collection of plane graphs, we set \( DDG(H) := \bigcup_{H \in H} DDG_H \).

\[\textbf{Lemma 3 (FR-Dijkstra [38, 40]).} \text{ Given all } DDG_{H_i}, \text{ one can compute a single-source shortest paths tree from any source } s \text{ in } DDG(H) \text{ in } O\left(\sum_{i=1}^{q} |\partial H_i| \log^2 n\right) \text{ time, where } n = |V(DDG(H))|\].\(^4\)

\(^3\) This definition is slightly more general than the definition of a hole of a piece \( P \in T(G) \). Namely, the definition of an \( r \)-division does not assume a fixed embedding of the entire \( G \); it only assumes some fixed embeddings of individual pieces.

\(^4\) In particular \( H_i \) may be single-edge. This way, this lemma captures also the case when we compute shortest paths in a collection of DDGs with some auxiliary vertices and edges.
We now state some fairly standard definitions and lemmas about representing distances between some vertices of $G$ of interest using unions of dense distance graphs of a recursive decomposition's (or $r$-division's) pieces (for instance cf. [24]), adapted to our notation. We include their proofs for completeness in Appendix A. For example, Lemma 4 captures the well-known observation of [38] that in order to compute a shortest path between any pair of vertices of $G$, it is enough to compute a shortest path in a union of dense distance graphs from $T(G)$ with only $O(\sqrt{n})$ vertices in total.

Let $L$ be some leaf of $T(G)$. We define the cone of $L$, denoted $\text{cone}_G(L)$, to be the collection of pieces of $T(G)$ containing $L$, all ancestors of $L$, and all siblings of (weak) ancestors of $L$. For some collection $\mathcal{L}$ of leaf pieces of $T(G)$, we define $\text{cone}_G(\mathcal{L}) = \bigcup_{L \in \mathcal{L}} \text{cone}_G(L)$.

**Lemma 4** ([24, 38]). Let $\mathcal{L}$ be some collection of leaf pieces of $T(G)$. Then:
1. For any $u, v \in V(\text{DDG}(\text{cone}_G(\mathcal{L})))$, $\text{dist}_G(u, v) = \text{dist}_{\text{DDG}(\text{cone}_G(\mathcal{L}))}(u, v)$.
2. $\sum_{H \in \text{cone}_G(\mathcal{L})} |\partial H| = O\left(\sqrt{n|\mathcal{L}|}\right)$.

Let $\mathcal{R}$ be an $r$-division with few holes of $G$ and $T(P)$ be a recursive decomposition of $P \in \mathcal{R}$ with the root boundary set to $\partial P$. For any $v \in V(G) \setminus \partial G$, let $L_v$ be some leaf containing $v$ in the unique piece $P_v \in \mathcal{R}$ containing $v$. For any $X \subseteq V(G)$ let us define $\text{cone}_\mathcal{R}(X) = \mathcal{R} \cup \bigcup_{v \in X \setminus \partial \mathcal{R}} \text{cone}_{P_v}(L_v)$.

**Lemma 5** ([24, 38]). Let $X \subseteq V(G)$ be non-empty. Then:
1. For any $u, v \in V(\text{DDG}(\text{cone}_\mathcal{R}(X)))$, $\text{dist}_\mathcal{R}(u, v) = \text{dist}_{\text{DDG}(\text{cone}_\mathcal{R}(X))}(u, v)$.
2. $\sum_{H \in \text{cone}_\mathcal{R}(X)} |\partial H| = O\left(n/\sqrt{r} + \min\left(\sqrt{n \cdot |X|}, |X| \cdot \sqrt{r}\right)\right)$.

**Fully dynamic $r$-divisions.** Many dynamic algorithms for planar graphs maintain $r$-divisions and useful auxiliary data structures under dynamic updates. The exact set of supported updates to $G$ varies; e.g., [38, 56, 63] support only edge weight changes, [54] assumes embedding-preserving insertions, whereas [66, 75] only assume that the graph $G$ remains planar at all times. We stick to the last, most general setting. The core of the construction behind the following theorem is due to Klein and Subramanian [66, 75]; for completeness we give a complete proof in Appendix A.

**Theorem 6.** Let $G = (V, E)$ be a weighted planar graph. Suppose that adding infinite-weight edges to $G$ does not have effect on any properties of $G$ that we care about. Let $r \in [1, n]$.

There is a data structure maintaining an $r$-division with few holes $\mathcal{R}$ of some $G^+$ such that:
1. $G^+$ is obtained from $G$ by adding infinite-weight edges.
2. Each $P$ has all its faces except its holes triangular and is accompanied with some auxiliary data structures that can be constructed in $T(r)$ time given $P$ and use $S(r)$ space.

The data structure uses $O\left(n + \frac{n}{r} \cdot S(r)\right)$ space and can be initialized in $O\left(n + \frac{n}{r} \cdot T(r)\right)$ time. After each edge deletion and edge insertion (preserving the planarity of $G$), it can be updated in $O(r + T(r))$ worst-case time.

**Additively weighted Voronoi diagrams.** Let $G$ be a directed planar graph of size $n$ with real edge-lengths, and no negative-length cycles. Assume that all faces of $G$ are triangles except, perhaps, a single face $f$. Let $S$ be the set of vertices that lie on $f$, called sites, i.e., $S = V(f)$. Let us assign to each site $s \in S$ a weight $\omega(s) \in \mathbb{R}_{\geq 0} \cup \{\infty\}$. The additively weighted distance $\text{dist}_G^\omega(s, v)$ between a site $s \in S$ and a vertex $v \in V(G)$ is defined as $\omega(s) + \text{dist}_G(s, v)$.
The additively weighted Voronoi diagram of \((S, \omega)\) within \(G\), denoted by \(\text{VD}(S, \omega)\), is a partition of \(V(G)\) into pairwise disjoint sets, one set \(\text{Vor}(s)\) for each site \(s \in S\). The set \(\text{Vor}(s)\), which is called the Voronoi cell of \(s\), contains all vertices in \(V(G)\) that are closer (wrt. \(dist_{\omega}\)) to \(s\) than to any other site in \(S\).

In the following and throughout, whenever we work with Voronoi diagrams we assume that (1) \(G\) is strongly connected, (2) shortest paths in \(G\) are unique, and (3) additively weighted shortest paths in \(G\) are unique, i.e., for each \(v \in V(G)\) there is a unique site \(s\) minimizing \(dist^\omega_G(s, v)\). Note that these assumptions make the Voronoi cells well-defined and simply connected, and guarantee that they indeed form a partition of \(V(G)\). We will explicitly ensure that these requirements are met for \(G, S\) and the weight function whenever we define a Voronoi diagram on \(G\).

There is a dual representation \(\text{VD}^\ast(S, \omega)\) of Voronoi diagram \(\text{VD}(S, \omega)\) as a tree of constant degree with \(O(|S|)\) vertices and edges [41]. An efficient FR-Dijkstra based algorithm for computing \(\text{VD}^\ast(S, \omega)\) was presented by Charalamopoulos et al. [23].

\[\text{Theorem 7} \ (\text{[23]})\] Suppose that we have at hand a recursive decomposition \(T(G)\) of \(G\), with the only hole of \(G\) being \(f\) and \(S = V(f)\). Further suppose that we have \(\text{DDG}_H\) computed for each piece \(H \in T(G)\). Then, we can compute \(\text{VD}^\ast(S, \omega)\) in \(O(\sqrt{n} \cdot |S| \log^2 n)\) time.

\[\text{Remark 8}\] The algorithm underlying Theorem 7 implicitly assumes that \(\text{Vor}(s)\) is non-empty for all \(s \in S\). In Appendix B, we discuss why this assumption is not necessary, relying on [41].

In a point location query for some Voronoi diagram \(\text{VD}(S, \omega)\), we are given a vertex \(v \in V(G)\) and are requested to find the site \(s \in S\) such that \(v \in \text{Vor}(s)\) and also the value of \(\text{dist}^\omega_{VD}(s, v)\). Gawrychowski et al. [41] showed the following result.

\[\text{Theorem 9} \ (\text{[41]})\] Suppose that we have at hand an MSSP data structure for \(G\) with sources from the face \(f\). Given such dual representation \(\text{VD}^\ast(S, \omega)\), we can preprocess it in \(O(|S|)\) time, so that point location queries for \(\text{VD}(S, \omega)\) can be answered in \(O(\log^2 n)\) time.

### 3 Fully Dynamic Single Source Shortest Paths

In this section we show our single-source exact distance oracle for planar graphs with \(O(n^{1/5} \log^2 n)\) update time and \(O(\log^2 n)\) query time and thus prove Theorem 1. For simplicity, let us assume that \(G\) is non-negatively weighted. Negative edges can be handled as in [56] – see Appendix C.

\[\text{Theorem 1}\] Let \(G\) be a real-weighted planar digraph with a source \(s \in V(G)\). There exists an \(O(n \log n)\)-space data structure maintaining \(G\) under edge insertions, edge deletions, and source changes with \(O(n^{1/5} \log^2 n)\) worst-case update time that can compute \(\text{dist}_G(s, v)\) for any \(v \in V(G)\) in \(O(\log^2 n)\) time. The initialization time is \(O(n \log^2 n)\).

The base of our data structure is a dynamic \(r\)-division \(R\) with few holes, as given in Theorem 6. Note that in our shortest-paths problem, indeed adding infinite-weight edges to \(G\) does no harm. Hence, in the following we work with the graph \(G^+\) from Theorem 6 when computing distances, but identify it, without loss of generality, with our original graph \(G\).

For technical reasons, however, we would like to avoid dealing with infinite weights in some of our data structures handling individual pieces. In the real-weighted fully dynamic setting, however, we cannot fix a sufficiently large finite number, larger than all edge weights that will ever appear in the future graph \(G\), beforehand. Instead, we do the following. For
each \( P \in \mathcal{R} \), let \( M_P \) be a sufficiently large finite number, e.g., larger than the sum of finite edge weights in \( P \). Consider \( M_P \) to be an auxiliary data structure of \( P \) as in Theorem 6. We will use \( M_P \) to simulate infinite edge weights in \( P \), and also for detecting paths non-existent in the original graph \( G \) (but having infinite weight in \( G^+ \cap P \)). As a result, below we assume each infinite weight in \( P \) (or any auxiliary data structure related to \( P \)) is replaced by \( M_P \) in all the computations performed locally on the piece \( P \), whereas globally (when performing some computation for many pieces at once, like the shortest paths algorithm of Lemma 3) we treat all edge or path weights in \( P \) that are at least \( M_P \) as infinite.

For each piece \( P \in \mathcal{R} \) we store the following additional data structures.

- We store the recursive decomposition \( T(P) \) with the initial boundary set to \( \partial P \), and also DDGs for all the pieces \( H \in T(P) \).
- For each hole \( h \) of \( P \), let \( P_h \) be the piece \( P \) after applying the following standard augmentations. First, \( P \) is extended into a graph \( P'_h \) using \( \mathcal{O}(r) \) vertices and edges of weight \( M_P \) embedded inside either the piece or other (than \( h \)) holes of \( P \) that would make \( P \) strongly connected and triangulated (except for the hole \( h \)) without changing the distance between any pair of reachable vertices in \( P \). The graph \( P_h \) is in turn obtained from \( P'_h \) by changing \( P'_h \)'s edge weights into \( \mathcal{O}(1) \)-size vectors as described in [36] so that there is a unique shortest path (wrt. the lexicographical order on path weights, defined as the coordinate-wise sum of the path's individual edge weights) between any \( u, v \in V(P_h) \) with cost of the form \( (\text{dist}_{P_h}(u, v), \cdot) \). As proven in [36], one can compute \( P_h \) from \( P \) deterministically in linear time. The \( \mathcal{O}(1) \)-size vector weights, in turn, can be easily packed into usual single-number weights.

For each \( P_h \), we store an MSSP data structure initialized for the hole \( h \). Recall that an MSSP data structure can be computed in \( \mathcal{O}(r \log r) \) time. Moreover, we store a recursive decomposition \( T(P_h) \) of \( P_h \) with the boundary \( \partial P_h \) of the root piece set to \( \partial P \cap V(h) \) of size \( \mathcal{O}(\sqrt{r}) \). For each node (piece) \( H \in T(P_h) \), we also store DDG\(_H\). Since the sum of sizes of all the pieces of \( T(P_h) \) is \( \mathcal{O}(r \log r) \), computing all these dense distance graphs takes \( \mathcal{O}(r \log^2 r) \) time (see Section 2).

Note that computing piecewise auxiliary data structures defined so far takes \( \mathcal{O}(r \log^2 r) \) time. So, by Theorem 6, they can be updated in \( \mathcal{O}(r \log^2 r) \) worst-case time after \( G \) undergoes an update.

After the initialization and each update, once \( \mathcal{R} \) and all auxiliary data structures are updated, we compute for each \( P \in \mathcal{R} \) a point location data structure.

**Lemma 10.** Given a weight function \( \omega : \partial P \to \mathbb{R} \cup \infty \), one can compute in \( \mathcal{O}(r^{3/4} \log^2 r) \) time a data structure \( L(P) \) answering the following queries in \( \mathcal{O}(\log^2 r) \) time: given any \( v \in V(P) \), compute the value \( \min_{b \in \partial P} \{ \omega(b) + \text{dist}_P(b, v) \} \) along with the minimizer \( b \).

**Proof.** Since \( \partial P \) is a union of \( \mathcal{O}(1) \) sets \( \partial P_h \), we can compute the desired minimum over each \( \partial P_h \) separately and then take the minimum over all \( h \).

Let us first note that negative values of \( \omega \) are not a problem. We can turn negative weights into non-negative by adding some common large value to the weights of all sites. We can thus suppose w.l.o.g. that all values of \( \omega \) are non-negative.

If all the weights are infinite, the queries can be answered trivially in \( \mathcal{O}(1) \) time. So in the following assume that there is at least one site whose weight is finite.

Let \( S = \partial P_h = \{ s_1, \ldots, s_k \} \) be the set of sites. In order to guarantee that for each \( v \in V(G) \) there is a unique site \( s \) minimizing \( \text{dist}_P^2(s, v) \), we will break ties by considering \( (\omega(s), i) \) instead of \( \omega(s) \) as the weight of site \( s_i \), adding a second coordinate to each edge weight in \( P_h \), set to 0, and comparing additively weighted distances lexicographically. Clearly, this extension does not break any of the properties of \( P_h \).
Recall that $P_h$ has finite non-negative real weights, is strongly connected, has unique shortest paths, and has a single face $h$ that is possibly non-triangular that contains all the sites $S$. Moreover, the additively weighted distances in $P_h$ are unique. Therefore, we can invoke Theorem 7 to construct the dual representation $VD^\ast(S,\omega)$ of the Voronoi diagram $VD(S,\omega)$. This requires $O\left(\sqrt{n/d_P} \log^2 n\right) = O\left(r^{3/4} \log^2 n\right)$ time.

Then, we construct the point location data structure of Theorem 9 for $VD(S,\omega)$. Note that we have an MSSP data structure for $P_h$ for hole $h$ and hence point location queries, given $VD^\ast(S,\omega)$, can be answered in $O(\log^2 n)$ time.

We invoke Lemma 10 with weight function $\omega(b) := dist_G(a,b)$ in order to construct $L(P)$ for each $P \in \mathcal{R}$. This requires $O\left(n/r^{1/4}, \log^2 n\right)$ time in total. By Lemma 5, the values $dist_G(s,b)$ for all $b \in \partial R$ can be computed in $O\left(n/\sqrt{r} \cdot \log^2 n\right)$ time if we run the single-source shortest paths algorithm of Lemma 3 (FR-Dijkstra) on the graph $DDG(cone_R(s))$. We also compute $dist_{P_s}(s,u)$ for all $u \in P_s$, where $P_s$ is an arbitrary piece containing $s$ using Dijkstra’s algorithm in $O(r \log r)$ time.

Now, we can compute $dist_G(s,v)$ for a query vertex $v$ as follows. If the shortest $s \to v$ path in $G$ does not go through $\partial R$, then it is fully contained in $P_s$ and therefore $v \in P_s$ and $dist_G(s,v) = dist_{P_s}(s,v)$, i.e., we have $dist_G(s,v)$ already computed. Otherwise, let $P_b$ be an arbitrary piece containing $v$. Observe that we have $dist_G(s,v) = \min_{b \in \partial P_s} \{dist_G(s,b) + dist_{P_b}(b,v)\}$ where the minimizer $b$ corresponds to the a boundary vertex of some shortest $s \to v$ path in $G$. So this case can be reduced to a single query to the data structure $L(P_v)$. This takes $O(\log^2 n)$ time.

The worst-case update time is $O(r \log^2 r + \frac{n}{r^{3/4}} \log^2 r)$. By setting $r = n^{4/5}$ we get $O(n^{4/5} \log^2 n)$ worst-case update time. Since the space usage per piece is $O(r \log r)$, we need $O(n \log^2 n)$ space.

Remark 11. Our data structure can be extended to report, following the computation of $dist_G(s,v)$, a shortest $s \to v$ path $Q$ in time nearly linear in the number of edges of $Q$. This follows easily by the fact that the MSSP data structure [63] can report shortest paths efficiently (see e.g., [56] for details). Therefore, we can efficiently expand the used edges of dense distance graphs and the shortest $b \to v$ path into actual edges in $G$.

3.1 A Dynamic Closest Facility Data Structure

We can generalize the dynamic single-source shortest paths data structure as follows. Suppose we replace a single source vertex $s$ with a set of facilities $F \subseteq V$. Given $F$, for a query vertex $v$ we would like to compute $\min_{f \in F} \{dist_G(f,v)\}$, and also possibly $f \in F$ minimizing this expression. A dynamic update would consist of either an edge update or changing the set $F$. In other words, such a problem can be seen as dynamic point location in a Voronoi diagram wrt. $F$, where each update either changes the graph or resets the Voronoi diagram of interest.

In this setting, we consider the following simple generalization of the single-source data structure. Let the update procedure first compute the distances $d(b) = \min_{f \in F} \{dist_G(f,b)\}$ for all $b \in \partial R$. Note that by Lemmas 5 this can be achieved by computing single-source shortest paths in the graph $DDG(D_F)$, where $D_F = cone_R(F)$, extended with 0-weight edges $sf$, where $s$ is an auxiliary super-source. By Lemma 3 this can be done in $O((\sqrt{n|F|} + n/\sqrt{|F|}) \log^2 n) = O((\sqrt{n|F|} + n^{4/5}) \log^2 n)$ time. By using weights $\omega := d$ in the individual point-location data structures $L(P)$, $P \in \mathcal{R}$, a single point location query on $L(P_v)$ (recall that $P_v$ is some piece containing $v$) would compute the desired closest facility $f_v$ minimizing $dist_G(f_v,v)$ unless the sought (weighted) shortest $f_v \to v$ does not go through a boundary.
vertex of $\mathcal{R}$. We could in principle handle such paths by proceeding as in the single-source case and computing shortest paths naively in each piece containing a facility. However, this could take time $\Omega(r \cdot \min(n/r, |F|))$, i.e., linear in $n$ even for moderately large facility set sizes, e.g., $|F| = \Omega(n^{1/5})$.

To improve upon this simple approach, we proceed as follows. Let $(\rho_m, \ldots, \rho_1)$ be such a sequence of integers that $\rho_m = r$, $\rho_1 = O(1)$ and $\rho_{i+1}/\rho_i = 2$ for all $i < m$. For each $P \in \mathcal{R}$ we store a recursive $(\rho_i)$-division consisting of pieces of $T(P)$ (cf. Section 2). Let $\mathcal{R}_{P,i}$ be the $\rho_i$-division of $P$. All $\mathcal{R}_{P,i}$ can be computed in linear time given $T(P)$ [64]. Note that $\mathcal{R}_i$, defined as the union of $\mathcal{R}_{P,i}$ over all pieces $P \in \mathcal{R}_i$ actually forms an $\rho_i$-division with few holes of the entire graph $G$. In particular, we have $\mathcal{R}_m = \mathcal{R}$.

We store the extended pieces $Q_h$ (recall how we obtained extended pieces $P_h$ with unique shortest paths from $P$ in the single-source case) plus their recursive decompositions $T(Q_h)$, DDGs, and an MSSP data structure for all pieces $Q$ of all $T(P)$ instead of just the pieces of $\mathcal{R}_m = \mathcal{R}$ as we did in the single-source case. However, we stress that these auxiliary components for a piece $Q \subseteq P$ where $P \in \mathcal{R}$, are counted as accompanying data structures of the piece $P$. So, we compute $O(1)$ fresh recursive decompositions $T(Q_h)$ for each piece $Q \in T(P)$ – computing each takes $O(|Q| \log^2 n)$ time. As a result, by the bound $\sum_{Q \in T(P)} |Q| = O(|P| \log |P|)$, the time to compute accompanying data structures of piece $P$ increases to $O(r \log^3 n)$.

Given the set of facilities $F$, let $j$ be such that $\rho_j = \Theta \left( \min \left( \frac{n}{\sqrt{|F|}}, r \right) \right)$. Redefine $D_F = \text{cone}_{\mathcal{R}}(F)$. Again, let us compute distances $d(b) = \min_{f \in F} \{ \text{dist}(f, b) \}$ (and the closest facilities) for all $b \in \bigcup_{H \in D_F} \partial H$ using FR-Dijkstra on DDG($D_F$) extended with a super-source $s$ and auxiliary edges $s, f \in F$. This takes $O \left( \left( \sqrt{|F|} + n/\sqrt{\rho_j} \right) \log^2 n \right) = O(\sqrt{n|F|} \log^2 n)$ time by Lemmas 3 and 5.

It only remains to show how to handle computation of closest facilities for $v \in V \setminus \bigcup_{H \in D_F} \partial H$. Recall that the pieces $D_F$ cover the entire $G$, no facility $f$ is an internal (non-boundary) vertex of a piece $H \in D_F$, and each $v \in V \setminus \bigcup_{H \in D_F} \partial H$ is clearly an internal vertex of a unique piece $H_v \in D_F$. Consequently, by Lemma 10, the closest facility to $v$ can be found in $O(\log^2 n)$ time using a single query to the data structure $L(H_v)$. For this to be possible, upon update we need to build the data structures $L(H)$ for all $H \in D_F$. By Lemma 10, this takes $O \left( \sum_{H \in D_F} |H| \cdot |\partial H| \log^2 n \right)$ time. Let us now bound this sum. First, let us consider the sum restricted to the pieces $H \in D_F \setminus \mathcal{R}_j$, i.e., the pieces of $r$-division $\mathcal{R}_j$. Since $\rho_j = \Omega(n/|F|)$ or $\rho_j = \Omega(r)$ we get:

$$O \left( \sum_{H \in D_F \setminus \mathcal{R}_j} \sqrt{|H| \cdot |\partial H|} \log^2 n \right) = O \left( \frac{n \cdot \rho_j^{3/4}}{\sqrt{ho_j}} \log^2 n \right) = O \left( n^{3/4} \cdot |F|^{1/4} + \frac{n}{r^{1/4}} \log^2 n \right).$$

On the other hand, if $H \notin D_F \setminus \mathcal{R}_j$, then $H \in \text{cone}_{\mathcal{R}_j}(F)$, where $f \in F \setminus \mathcal{R}_j$, $P_f$ is the unique piece of $\mathcal{R}_j$ containing $f$, and $L_f$ is some leaf of $T(P_f)$ containing $f$. For a fixed $f$, by the definition of $\text{cone}_{\mathcal{R}_j}(F)$, there are $O(\log n)$ pieces $H$ satisfying this, at most two per each level $i$ of $T(P_f)$. Hence, the sum of $|H| \cdot |\partial H|$ over such pieces can be bounded by $\sum_{i=0}^{\infty} \sqrt{|P_f| \cdot |\partial P_f|} / r^i = O(\rho_j^{3/4})$. Summing over all $f$, and using $\rho_j = O(n/|F|)$, we get

$$O \left( \sum_{H \in D_F \setminus \mathcal{R}_j} \sqrt{|H| \cdot |\partial H|} \log^2 n \right) = O \left( |F| \cdot \rho_j^{3/4} \log^2 n \right) = O \left( n^{3/4} \cdot |F|^{1/4} \log^2 n \right).$$
To conclude, the update time is $O\left((n^{3/4} \cdot |F|^{1/4} + \frac{n}{r^{3/5}}) \log^2 n + r \log^3 n\right)$. By setting $r = (n/\log n)^{4/5}$ we obtain the following theorem.

**Theorem 12.** Let $G$ be a real-weighted planar digraph with a set $F \subseteq V$ of facilities. There exists an $O(n \log^2 n)$-space data structure maintaining $G$ under edge insertions, edge deletions, and changes of the facilities set $F$ with $O\left((n^{3/4} \cdot |F|^{1/4} + n^{3/5}) \log^{11/5} n\right)$ worst-case update time that can compute $\min_{f \in F} \{\text{dist}_G(f, v)\}$ along with the respective closest facility of $v$ for any $v \in V(G)$ in $O(\log^2 n)$ time. The initialization time is $O(n \log^3 n)$.

### 4 Fully Dynamic Strongly Connected Components

In this section we show that a strategy similar to that of Section 3 can be used to obtain a fully dynamic strong-connectivity algorithm. Again, we maintain an $r$-division $\mathcal{R}$ and some auxiliary data structures for all the individual pieces. Formally, using a dynamic $r$-division as in Theorem 6 may require introducing new infinite-weight edges to $G$ which in turn may change the reachability relation in $G$. We circumvent this problem by setting the weights of the original edges of $G$ to 0, and all auxiliary edges plus infinity (simulated in the implementation by sufficiently large values $M_P$ inside individual pieces, as in Section 3). This way, $u$ can reach $v$ in $G$ if and only if $\text{dist}_G(u, v) = 0$, and otherwise $\text{dist}_G(u, v) = \infty$. All known properties of reachability in plane graphs also extend to reachability using 0-weight paths (assuming non-negative weights). In the following, whenever we say that $v$ is reachable from $u$, or there exists a $u \rightarrow v$ path, we mean $\text{dist}_G(u, v) = 0$.

As in Section 3, for each piece of $\mathcal{R}$ we store a recursive decomposition, dense distance graphs and MSSP data structures. All these data structures are also maintained for pieces of $\mathcal{R}$ with all edges reversed – for a piece $P$ we call this graph the reverse of $P$ and denote it by $P^{-\text{rev}}$.

Another ingredient is a collection of reachability certificates $X_P$ for all the pieces, as defined in the following lemma due to Subramanian [75], slightly adjusted to certify 0-weight paths.

**Lemma 13** ([75]). Let $P \in \mathcal{R}$ be a piece. There exists a directed graph $X_P$, where $\partial P \subseteq V(X_P)$, of size $O(\sqrt{r} \log r)$ satisfying the following property: for any $u, v \in \partial P$, $\text{dist}_P(u, v) = 0$ if and only if there exists a $u \rightarrow v$ path in $X_P$. The graph $X_P$ can be computed in $O(r \log r)$ time.

We include the reachability certificate in the set of auxiliary piecewise data structures. Since reachability certificates can be computed in $O(r \log r)$ time, maintaining them does not incur any additional asymptotic cost. The following lemma is a direct consequence of Lemma 13.

**Lemma 14.** For any $u, v \in \partial \mathcal{R}$, $u$ can reach $v$ in $G$ if and only if $u$ can reach $v$ in $X = \bigcup_{P \in \mathcal{R}} X_P$.

**Proof.** Let $u, v \in \partial \mathcal{R}$. Since each $X_P$ certifies the reachability between $\partial P$ in $P$, clearly a $u \rightarrow v$ path in $X$ implies an existence of a $u \rightarrow v$ path in $G$. Now suppose there is a $u \rightarrow v$ path $Q$ in $G$. Split $P$ into maximal subpaths $Q_1, \ldots, Q_k$, such that each $Q_i$ is fully contained in a single piece $P_i \in \mathcal{R}$. For each $i$, the endpoints $a, b$ of $Q_i$ are contained in $\partial P_i$ and hence there exists a $a \rightarrow b$ path in $X_P \subseteq X$. Consequently, there exist a $u \rightarrow v$ path in $X$.

To handle an edge update, after $\mathcal{R}$ and auxiliary data structures are updated, we compute the strongly connected components of $X$ (defined as in Lemma 14) in $O(|X|) = O(n/\sqrt{r})$ time using any classical linear-time algorithm. For any $b \in \partial \mathcal{R}$, let $s_X(b)$ denote an integer...
identifier of b’s strongly connected component in X. By additionally sorting the SCCs of X topologically we can further assume that sX satisfies the following property: if a, b ∈ ∂R are not strongly connected, but a can reach b in X then sX(a) < sX(b). By Lemma 14, for a, b ∈ ∂R, we have sX(a) = sX(b) if and only if a and b are strongly connected in G; moreover, if a can reach b in G, then sX(a) ≤ sX(b).

We also define and maintain similar SCC-identifiers sP for individual pieces P, i.e., for u, v ∈ V(P), sP(u) = sP(v) implies u, v are strongly connected in P, whereas sP(u) < sP(v) implies there is no v → u path in P. Clearly, the identifiers sP can be recomputed in O(ν) time given P, so we also include them into the set of auxiliary per-piece data structures.

For any Q ∈ {X} ∪ R, let SQ be the set of used identifiers of the form sQ(v). Observe that we can easily guarantee that the sets SQ are pairwise disjoint, e.g., by using disjoint integer ranges for different sets SQ.

The final component of our data structure, is, again a collection of per-piece point location data structures. For each P ∈ R, we have two point location data structures L(Pνν) and L(P) of Lemma 10. After each edge update, L(Pνν) is computed for Pνν with weight function ω = sX. On the other hand, L(P) is initialized with weight function ω = −sX. As in Section 3, all these point location data structures are recomputed in O(n/ν1/4 · log2 n) time (over all pieces).

We now describe how our data structure handles a query for an SCC identifier of a vertex v. The returned identifier always comes from the set ∪Q∈{X}∪R SQ. Let Pv be some piece containing v. Let smin be the value computed by L(Pννv) for vertex v. Let smax be minus the value computed by L(Pv) for v. If either of smin, smax equals ±∞ or smin ̸= smax holds, we return sPv(v). Otherwise, we return smin ∈ SX. The following lemma establishes the correctness of this query procedure.

**Lemma 15.** Let u, v ∈ V(G) and let su, sv be the respective identifiers returned by the query procedure. Then, su = sv if and only if u and v are strongly connected in G.

**Proof.** Suppose su = sv. If su ∈ SP for some piece P, then su = sv implies that u and v are strongly connected in P and thus also in G. So suppose su ∈ SX. Take any b ∈ ∂R such that sv = sX(b). We now prove that v and b are strongly connected in G. Similarly we prove that u and b are strongly connected in G. By transitivity it will follow that u and v are indeed strongly connected.

Let Pν, smin, smax be defined as in the query procedure’s description. Recall that sv ∈ SX implies that smin, smax are finite and sv = smin = smax. Since all edges of Pν have weight 0, and smax is finite, smax in fact represents the maximum value sX(a) among those a ∈ ∂Pν such that a path a → v exists in Pν. Similarly, observe that smin represents the minimum value sX(c) among those c ∈ ∂Pν such that a path c → v exists in Pνν, i.e., such that a path v → c exists in Pν. Let us denote by a and c the respective vertices of ∂Pν attaining the maximum and minimum values of sX. Since sX(a) = sX(c), there exists a path c → a in G. However, by the definition of a and c, paths a → v and v → c also exist in G, and hence a, v, and c are strongly connected in G. Since a is clearly strongly connected to b by sX(a) = sX(b), indeed v and b are strongly connected in G.

Now let us move to proving the “⇒” direction. Suppose u and v are strongly connected in G. First consider the case when there exists some vertex b ∈ ∂R located in the same strongly connected component of G as u and v. In this case we prove that sv = sX(b). An analogous proof that su = sX(b) will establish su = sv. Since v and b are strongly connected, there exist some paths Q1 = v → b and Q2 = b → v in G. Let v1 be the first vertex on Q1 such that v1 ∈ ∂Pνv1 — note that v1 necessarily exists since b ∈ ∂R. Similarly set v2 to be
the last vertex on \( Q_2 \) such that \( v_2 \in \partial P_v \). Observe that the subpaths \( v \rightarrow v_1 \) and \( v_2 \rightarrow v \) of \( Q_1 \) and \( Q_2 \) respectively lie entirely inside \( P_v \). Hence, \( s_{\text{min}} \) and \( s_{\text{max}} \) are finite and we have \( s_{\text{max}} = s_X(v_2) \) and \( s_{\text{min}} = s_X(v_1) \). Recall that there exists a walk \( v \rightarrow v_1 \rightarrow b \rightarrow v_2 \rightarrow v \), so in fact \( v_1, v_2, b \) are strongly connected, i.e., \( s_X(v_1) = s_X(v_2) = s_X(b) \). Thus, we obtain \( s_{\text{min}} \leq s_X(b) \leq s_{\text{max}} \).

On the other hand, let \( a \in \partial P_v \) be such that a path \( a \rightarrow v \) exists in \( P_v \) and \( s_{\text{max}} = s_X(a) \) (\( a \) exists by \( s_{\text{max}} \neq \pm \infty \)). Similarly, let \( c \in \partial P_v \) be such that a path \( v \rightarrow c \) exists in \( P_v \) and \( s_{\text{min}} = s_X(c) \). Since a path \( a \rightarrow c \) through \( v \) exists in \( P_v \) (so also in \( G \)), we have that \( s_{\text{max}} \leq s_{\text{min}} \) by the fact that the identifiers \( S_X \) respect the topological order of the SCCs of \( X \). Recall that we have already proved \( s_{\text{min}} \leq s_X(b) \leq s_{\text{max}} \) so in fact we have \( s_{\text{min}} = s_{\text{max}} = s_X(b) \), and consequently \( s_v = s_X(b) \).

Finally, suppose there is no vertex of \( \partial R \) in the SCC of \( G \) containing \( u \) and \( v \). First, this implies that \( u, v \notin \partial R \) and all \( u \rightarrow v \) and \( v \rightarrow u \) paths are contained in a single, unique piece \( P \). This implies that \( s_P(u) = s_P(v) \). Hence it is sufficient to prove \( s_v = s_P(u) \) and \( s_v = s_P(v) \). We prove the latter equality; proving the former is analogous. Recall that \( s_v \) is not set to \( s_P(v) \) only if both \( s_{\text{min}}, s_{\text{max}} \) are finite and \( s_{\text{min}} = s_{\text{max}} \). This can only happen if there exists vertices \( a, c \in \partial P \) such that \( a \) can reach \( v \) in \( P \), \( v \) can reach \( c \) in \( P \) and \( s_X(c) = s_{\text{min}} = s_{\text{max}} = s_X(a) \), i.e., \( a \) and \( c \) are strongly connected in \( G \). But this implies that \( a, c \) and \( v \) are strongly connected in \( G \), which contradicts the fact that the SCC of \( v \) in \( G \) does not contain vertices of \( \partial R \).

The running time analyses of both the update and query procedures are identical to the analyses of Section 3. Hence, we have proved the following theorem.

\textbf{Theorem 2.} Let \( G \) be a planar digraph. There exists an \( O(n \log n) \)-space data structure maintaining \( G \) under edge insertions and deletions with \( O(n^{4/3} \log^2 n) \) worst-case update time that can compute the identifier of the strongly connected component of any \( v \in V(G) \) in \( O(\log^2 n) \) time. The initialization time is \( O(n \log^2 n) \).

\textbf{References}


Single-Source Shortest Paths and Strong Connectivity in Dynamic Planar Graphs


Then for any $u,v$ we get $H_{\text{dist}}$ trivial if $H_{\partial H} \subseteq \mathcal{Q}$. We prove this claim by induction on the level $\ell$ of piece $H$. For $\ell = 0$, we get $H \in \mathcal{Q}$, so clearly $\partial H \subseteq V(H) \subseteq V(\text{DDG}(Q))$. Suppose $\ell \geq 1$. The statement is trivial if $H \in \mathcal{Q}$. Otherwise, consider the children $H_1, \ldots, H_k$ of $H$. By induction we get $\partial H_i \subseteq V(\text{DDG}(Q))$. So in fact we have $\partial H \subseteq \bigcup_{i=1}^{k} \partial H_i \subseteq V(\text{DDG}(Q))$.

Since the edges of each $\text{DDG}_H$ encode lengths of some paths in $G$, we have that $\text{dist}_{\text{DDG}(Q)}(u,v) \geq \text{dist}_G(u,v)$. We now prove that there is a path of length at most $\text{dist}_G(u,v)$ in $\text{DDG}(Q)$. Let $P$ be some shortest $u \rightarrow v$ path in $G$. Let $H$ be a piece of $\mathcal{T}(G)$ of minimum level that contains $P$. We prove our claim by induction on the level $\ell$ of $H$. 

### Lemma 16

Let $\mathcal{Q}$ be some collection of pieces from $\mathcal{T}(G)$ such that:

1. for each leaf piece $L \in \mathcal{T}(G)$, either $L$ or some ancestor of $L$ is in $\mathcal{Q}$,
2. for each $H \in \mathcal{Q}$, if some ancestor of $H$ is in $\mathcal{Q}$, then the parent of $H$ is also in $\mathcal{Q}$.

Then for any $u,v \in V(\text{DDG}(\mathcal{Q}))$, $\text{dist}_{\text{DDG}(\mathcal{Q})}(u,v) = \text{dist}_G(u,v)$.
First note that by property 1 of $Q$, if $H \notin Q$ and no descendant of $H$ belongs to $Q$, then $H$ has a nearest ancestor $H^*$ such that $H^* \in Q$. Observe that $V(\text{DDG}(Q)) \cap V(H) \subseteq \partial H^*$. Hence, $u, v \in \partial H^*$ and thus $\text{dist}_{\text{DDG}}(u, v) \leq \text{dist}_{\text{DDG}}(u, v) = \text{dist}_{H^*}(u, v) \leq \text{dist}_{G}(u, v)$.

So we can assume that either $H \in Q$ or some descendant of $H$ belongs to $Q$. Then by property 2 of $Q$, we have that either $H \in Q$ or no ancestor of $H$ belongs to $Q$. In either of these cases we have $\partial H \subseteq V(\text{DDG}(Q))$. Suppose first $\ell = 0$ — then $H$ is a leaf piece and thus $H \in Q$. So clearly $P \subseteq H \subseteq \text{DDG}(Q)$, i.e., $\text{dist}_{\text{DDG}}(u, v) \leq \text{dist}_{G}(u, v)$. On the other hand, if we assume $\ell \geq 1$ then we can split $P$ into maximal subpaths $P_1, \ldots, P_k$ such that each $P_i = u_i \rightarrow v_i$ is entirely contained in a single child of $H_P$. Then for each $i$ we have $\{u_i, v_i\} \subseteq \{u, v\} \cup \partial H \subseteq V(\text{DDG}(Q))$ so by induction we get that $\text{dist}_{\text{DDG}}(u_i, v_i) \leq \text{dist}_{G}(u_i, v_i)$ which implies $\text{dist}_{\text{DDG}}(u, v) \leq \text{dist}_{G}(u, v)$.

**Lemma 4** ([24, 38]). Let $\mathcal{L}$ be some collection of leaf pieces of $\mathcal{T}(G)$. Then:
1. For any $u, v \in V(\text{DDG}(\text{cone}_G(\mathcal{L})))$, $\text{dist}_G(u, v) = \text{dist}_{\text{DDG}(\text{cone}_G(\mathcal{L}))}(u, v)$.
2. $\sum_{H \in \text{cone}_G(\mathcal{L})} |\partial H| = O\left(\sqrt{|\mathcal{L}|}\right)$.

**Proof.** To obtain item 1 it is enough to note that $\text{cone}_G(\mathcal{L})$ satisfies the requirements posed on the collection $Q$ in Lemma 16.

Let $A$ be the set containing all ancestors of all the leaf pieces $L \in \mathcal{L}$. We show item 2 by bounding the sum $X = \sum_{H \in A} |\partial H|$. Since the number of boundary vertices of a piece is bounded by the sum of numbers of boundary vertices of its parent and its sibling, the sum $\sum_{H \in \text{cone}_G(\mathcal{L})} |\partial H|$ of our interest can be larger from $X$ only by a constant factor.

Recall that $\mathcal{T}(G)$ admits an $r$-division for any $r \in [1, n]$, i.e., there exists such an $r$-division $\mathcal{R}$ that $\mathcal{R} \subseteq \mathcal{T}(G)$ and the boundary of each piece of $\mathcal{R}$ equals the boundary of that piece in $\mathcal{T}(G)$. Let us split $A$ into two parts: let $A_1$ contain those $H \in A$ that are descendants of some piece $P \in \mathcal{R}$, and let $A_2 = A \setminus A_1$. Let $X_i = \sum_{H \in A_i} |\partial H|$.

Since each $L \in \mathcal{L}$ is a descendant of a unique piece $P \in \mathcal{R}$, we now bound the sum of $|\partial H|$ over all ancestors of $L$ that are descendants of $P$. Recall (Section 2) that if $H$ is a piece in a recursive decomposition of an $n$-vertex graph, then $|\partial H| = O(\sqrt{n} / c^d)$ where $d$ is the depth of $H$ in that decomposition. Consider the subtree of $\mathcal{T}(G)$ rooted at $P$ — it forms a recursive decomposition $\mathcal{T}(P)$ of $P$ with some initial boundary vertex set $|\partial P|$ of size $O(\sqrt{r})$. So the sum of $|\partial H|$ over all ancestors of $L$ that are descendants of $P$ is actually equal to the sum of $|\partial H|$ over all ancestors $H$ of $L$ in $\mathcal{T}(P)$. Since each ancestor has distinct integral depth, this sum is $O(\sum_{i=0}^{\infty} \sqrt{|P_i|} / c^d) = O(\sqrt{|P|}) = O(\sqrt{r})$. Hence $X_1 = O(|\mathcal{L}| \cdot \sqrt{r})$. On the other hand, $X_2 \leq \sum_{P \in \mathcal{R}} |\partial P| = O(n / \sqrt{r})$. So we obtain $X = X_1 + X_2 = O(n / \sqrt{r} + |\mathcal{L}| \cdot \sqrt{r})$. By choosing $r = \sqrt{n / \sqrt{|\mathcal{L}|}}$, we obtain $X = O(\sqrt{n \cdot |\mathcal{L}|})$ as desired.

**Lemma 5** ([24, 38]). Let $X \subseteq V(G)$ be non-empty. Then:
1. For any $u, v \in V(\text{DDG}(\text{cone}_R(X)))$, $\text{dist}_G(u, v) = \text{dist}_{\text{DDG}(\text{cone}_R(X))}(u, v)$.
2. $\sum_{H \in \text{cone}_R(X)} |\partial H| = O\left(\sqrt{n} + \text{min}\left(\sqrt{n \cdot |X|}, |X| \cdot \sqrt{r}\right)\right)$.

**Proof sketch.** The proof is completely analogous to that of items 2 and 3 of Lemma 4. It is enough to glue the individual decompositions $\mathcal{T}(P)$ into a single decomposition $\mathcal{T}(G)$ such that the root has $O(n / r)$ children instead of just 2: the individual pieces of $\mathcal{R}$. Then item 1 follows by Lemma 16. Let $Y = \sum_{H \in \text{cone}_R(X)} |\partial H|$. Note that $Y = O(n / \sqrt{r} + |X| \cdot \sqrt{r})$. If we have $r = n / |X| \leq r$, then we can obtain the bound $Y = O(\sqrt{n \cdot |X|})$ in the proof of Lemma 4. Otherwise, $|X| \leq n / r$, so $Y = O(n / \sqrt{r} + |X| \cdot \sqrt{r}) = O(n / \sqrt{r})$. So $Y = O(n / \sqrt{r} + \sqrt{n \cdot |X|})$ in all cases as well.

\[ [31:20] \]
Theorem 6. Let $G = (V, E)$ be a weighted planar graph. Suppose that adding infinite-weight edges to $G$ does not have any properties of $G$ that we care about. Let $r \in [1, n]$.

There is a data structure maintaining an $r$-division with few holes $R$ of some $G^+$ such that:
1. $G^+$ is obtained from $G$ by adding infinite-weight edges.
2. Each $P$ has all its faces except its holes triangular and is accompanied with some auxiliary data structures that can be constructed in $T(r)$ time given $P$ and use $S(r)$ space.

The data structure uses $O\left(n + \frac{n}{r} \cdot S(r)\right)$ space and can be initialized in $O\left(n + \frac{n}{r} \cdot T(r)\right)$ time. After each edge deletion and edge insertion (preserving the planarity of $G$), it can be updated in $O(r + T(r))$ worst-case time.

Proof. On initialization, we first connect and triangulate $G$ using infinite-weight edges, thus obtaining $G^+$. Then, we compute an $r$-division $R$ with few holes of $G^+$ in linear time [64] and subsequently initialize the auxiliary data structures. Let $h \geq 2$ and $c \geq 8$ be constants such that a single piece of the computed $r$-division has at most $c\sqrt{r}$ boundary vertices distributed over $h$ holes.

We will guarantee at all times that for any single piece $P$, $|\partial P| \leq 3c\sqrt{r}$, and there exist at most $3h$ faces of $P$ such that any $v \in \partial P$ lies on one of these faces, called holes of $P$. Moreover, each edge of $G^+$ is contained in at most two pieces of $R$: this is satisfied initially since the $r$-division of [64] forms a partition of faces of $G^+$.

Suppose that the removal of an edge $e = uv$ is issued to $G$. We then remove $e$ from each of the at most two pieces $P$ containing it. If $P$ is disconnected afterwards, we replace it with two connected pieces $P_1, P_2$. Otherwise, since removing $e$ merges two faces of $P$, the total number of holes of $P$ does not increase. If, on the other hand, a new edge $e = uw$ is inserted, we add a new piece $P_e$ consisting of a single edge $e$ to $R$. Adding $P_e$ may cause an endpoint of $e$, say $u$, to become a boundary vertex of $R$. If $u$ was not a boundary vertex before the insertion, it had to be a vertex of a single piece $P_u$. At this point, since a new boundary vertex emerges in $P_u$, we might have $\partial P_u > 3c\sqrt{r}$ or $\partial P$ might no longer lie on at most $3h$ faces of $P_u$. However, there surely exist some $3h + 1$ faces whose vertices include the whole set $\partial P_u$, and $|\partial P_u| \leq 3c\sqrt{r} + 1$. To fix our invariants, we first compute a cycle separator $C$ of $P_u$ wrt. the boundary vertices of $P_u$ in $O(r)$ time, and replace $P_u$ with two pieces $P_{u,1}, P_{u,2}$ — the two subgraphs of $P_u$ induced by vertices weakly on one side of $C$. Clearly, the vertices of $C$ become new boundary vertices afterwards. Subsequently, we similarly break each of $P_{u,1}, P_{u,2}$ further into two parts using a cycle separator wrt. the holes of this piece (see [64] for details). Each of the at most four resulting pieces has at most $\frac{3}{2}(3c\sqrt{r} + 1) + 2\sqrt{2}\sqrt{r} \leq (2c + 2\sqrt{2} + 2)\sqrt{r} \leq 3c\sqrt{r}$ boundary vertices, and at most $\frac{3}{2} \cdot (3h + 1) + 1 \leq 2h + 2 \leq 3h$ holes.

Observe that a single edge update can introduce $O(1)$ new pieces of size $O(r)$ in the maintained $r$-division, and the sizes of the existing pieces do not increase. For each of the affected pieces we recompute the auxiliary data structures in $O(T(r))$ time. As a result, after $O(n/r)$ updates, there are still $O(n/r)$ pieces, each of size $O(r)$ and with $O(\sqrt{r})$ boundary vertices distributed over $O(1)$ holes of that piece. Consequently, after every $\Omega(n/r)$ updates, we reinitialize $R$ for the current graph $G$ in $O\left(n + \frac{n}{r} \cdot T(r)\right)$ time. Hence, the amortized time to update $R$ is $O(r + T(r))$.

Finally, observe that our data structure can be modified in a standard way (see e.g., [5]) to have $O(r + T(r))$ worst-case update time bound instead of just an amortized one. This is possible since our update procedure actually takes $O(r + T(r))$ worst-case time apart from once every $k = \Omega(n/t)$ updates when the whole data structure is rebuilt in $O\left(n + \frac{n}{r} \cdot T(r)\right)$.
worst-case time. To this end we apply the time-slicing technique. We use two copies of our data structure switching their roles every \(k/2\) updates. One copy is for handling at most \(k/2\) updates and answering queries, and another is being gradually reinitialized in chunks of \(\Omega(r + T(r))\) time (of either initialization or updates replayed) in the background.

\[\text{B Empty Voronoi Cells}\]

Here, we briefly explain why the algorithm of [23] that underlies Theorem 7 works irrespective of whether there are empty Voronoi cells.

We need a few more definitions. Let \(f^*\) be the vertex of the dual graph of \(G\) corresponding to the face \(f\), where the sites lie. For an additively weighted Voronoi diagram \(VD(S, \omega)\) over a triangulated graph (possibly apart from face \(f\)), we call a face whose incident vertices belong to three different Voronoi cells trichromatic. Let \(VD_0^*(S, \omega)\) be the forest obtained by considering the dual edges of \(G\) whose endpoints belong to different Voronoi cells. Then, we obtain \(VD_1^*(S, \omega)\) by repeatedly contracting edges of the forest that have an endpoint of degree 2. We obtain \(VD_2^*(S, \omega)\) by creating one copy of \(f^*\) for each of its incident edges – inheriting only that edge. The vertices of \(VD_2^*(S, \omega)\) that are not copies of \(f^*\) are in one-to-one correspondence with the trichromatic faces of \(G\), other than \(f\). If \(VD_2^*(S, \omega)\) is a tree then we are done. Otherwise, is some cell of \(VD(S, \omega)\) is empty, \(VD_2^*(S, \omega)\) might be a forest. However, as shown in Section 6 of [41], \(VD_2^*(S, \omega)\) can be turned into the sought tree \(VD^*(S, \omega)\) in \(O(|S|)\) time.

The algorithm of [23] for computing \(VD^*(S, \omega)\) implicitly assumes that all Voronoi cells are non-empty, and hence that \(VD_2^*(S, \omega)\) is a tree. This algorithm performs FR-Dijkstra computations on \(G\) in order to compute the trichromatic faces of \(G\), and a representation of shortest paths from the sites to the vertices incident to these trichromatic faces. Then, \(VD_2^*(S, \omega)\) can be straightforwardly retrieved from this representation. If we run the same algorithm without assuming that there are no non-empty Voronoi cells, the sites with empty Voronoi cells can be easily read from the shortest paths tree obtained from FR-Dijkstra (more specifically, these are the sites whose some ancestor in this tree is also in \(S\)). Since the proof of correctness of the algorithm of [23] for computing the trichromatic faces actually only requires the set \(S\) to lie on a single face of \(G\) (and not necessarily to be equal to \(V(f)\)), we can re-run it with \(S\) pruned from “empty” sites. From all the trichromatic faces and the corresponding shortest paths from sites to their incident vertices, we can retrieve the forest \(VD_2^*(S, \omega)\) in the same way as if it were a tree. Finally, we can then turn this forest into the sought tree \(VD^*(S, \omega)\) in \(O(|S|)\) time as in [41].

\[\text{C Negative Edges in the Fully Dynamic SSSP Algorithm}\]

Recall that \(p : V(H) \to \mathbb{R}\) is called a feasible price function of \(H\) if \(\text{dist}_H(u, v) + p(u) - p(v) \geq 0\) for all \(u, v \in V(H)\). A feasible price function is guaranteed to exist if \(H\) contains no negative-cost cycle. It is well-known that, provided that a graph \(H\) is strongly connected, a vector of distances from any vertex of \(H\) constitutes a feasible price function of \(H\).

As in the fully dynamic all-pairs algorithm of [56], we maintain functions \(\phi : \partial R \to \mathbb{R}\) and \(\phi_P : \partial P \to \mathbb{R}\), where \(P \in R\), such that \(\phi\) is some feasible price function of \(G\) restricted to \(\partial R\), and each \(\phi_P\) is a feasible price function of \(P\).

Since single-source shortest paths in planar graphs with negative weights can be computed in \(O(n \log^2 n)\) time [65, 70], each \(\phi_P\) can be seen as an accompanying data structure of piece \(P\) computable in \(O(r \log^2 r)\) time and maintained by the fully dynamic \(r\)-division algorithm.
The functions $\phi_P$ allow to treat individual graphs $P$ and $P_h$ as non-negatively weighted when computing all the needed DDGs and MSSP data structures, and also point location data structures $L(P)$.

It is known [40, 56] that the FR-Dijkstra algorithm (as in Lemma 3) can handle negative weights in DDG($\mathcal{H}$) with no asymptotic overhead if a feasible price function on DDG($\mathcal{H}$) is provided. Therefore, we would like to have a feasible price function on DDG(cone$_\mathcal{R}(s)$) to compute distances from $s$ in DDG(cone$_\mathcal{R}(s)$) needed by the update algorithm. We can extend $\phi$ from $\partial \mathcal{R}$ to all vertices in DDG(cone$_\mathcal{R}(s)$) as follows. Note that all pieces in cone$_\mathcal{R}(s)$ except of those in $\mathcal{R}$ have their parents also in cone$_\mathcal{R}(s)$. We call those pieces $H \in$ cone$_\mathcal{R}(s)$ for which we know the value of $\phi$ on all of $\partial H$ processed. Initially, only the pieces $P \in \mathcal{R}$ are processed by the definition of $\phi$. While there are still unprocessed pieces, we take any unprocessed piece $H$ whose parent $A \in \mathcal{T}(P)$ has already been processed. Let $H'$ be the sibling of $H$ in $\mathcal{T}(P)$. Observe that $\phi_P$ is a feasible price function of $A$ as well. We extend $\phi$ to boundary vertices of $H, H'$ by computing shortest paths on DDG($\{H, H'\}$) from vertices $\partial A$, with the initial distance to each $v \in \partial A$ set to $\phi(v)$, and using FR-Dijkstra (Lemma 3) with price function $\phi_P$. This way, only the initial distances of $\partial A$ are possibly negative from the point of view of FR-Dijkstra. This does not constitute a problem for either Dijkstra’s algorithm or FR-Dijkstra though (see [56]; one can treat the initial distances as weights of edges going out of a super-source; these weights can be all increased by the same large value to be made positive). One can show from the definition of $\phi$ that the values of $\phi$ on $\partial A$ will not be altered and the computed distances form a feasible price function on $\partial H \cup \partial H'$ in $G$. Hence, given that $\partial A \subseteq \partial H \cup \partial H'$, we can process the children $H, H'$ of a processed piece $A$ in $O(|\partial H| + |\partial H'| \log^2 n)$ time. Summing over all pieces, we obtain by Lemma 5 that extending $\phi$ to all $V(\text{DDG}(\text{cone}_\mathcal{R}(s)))$ takes $O(n/\sqrt{r} \log^2 n)$ time. This cost is therefore negligible.

Note that an edge deletion or weight increase cannot break the feasibility of $\phi$. We might need to recompute $\phi$ only upon insertion or weight decrease of some edge $uv$. As shown by Kaplan et al. [56], the new “global” price function $\phi'$ (or a negative cycle) can be found by computing distances from $v$ to $\partial \mathcal{R} \cup \{u\}$ in DDG(cone$_\mathcal{R}(u, v)$) before applying the insertion using FR-Dijkstra and the old price function $\phi$. This can be done in $O(n/\sqrt{r} \log^2 n)$ time by first extending the old $\phi$ to $V(\text{DDG}(\text{cone}_\mathcal{R}(u, v)))$ as described above and then running the single-source shortest paths algorithm of Lemma 3.