University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Adam Karczmarz

Data Structures and Dynamic Algorithms for Planar Graphs

PhD dissertation

Supervisor

dr hab. Piotr Sankowski
Institute of Informatics
University of Warsaw

November 2018

Author's declaration:
I hereby declare that this dissertation is my own work.

November 5, 2018 .....
date *Adam Karczmarz*

Supervisor's declaration:
the dissertation is ready to be reviewed.

November 5, 2018 .....
date *dr hab. Piotr Sankowski*

Abstract

In this thesis we show new data structures useful in designing planar graph algorithms.

First, we show an optimal data structure maintaining a planar graph subject to edge contractions. Specifically, our data structure explicitly maintains individual vertices' neighbors lists and supports constant-time adjacency queries on the stored graph. After each subsequent edge contraction, the data structure can be updated in amortized constant time. By applying the data structure, we obtain optimal algorithms for several planar graph problems, such as finding a unique perfect matching or computing maximal 3-edge connected subgraphs. We also show that by using our data structure in a black-box manner one obtains conceptually simple optimal algorithms for minimum spanning tree and 5-coloring in planar graphs.

Next, we study decremental reachability algorithms for planar directed graphs. In particular, we show a nearly-optimal decremental algorithm for single-source reachability for planar digraphs. We also show a trade-off for decremental transitive closure. For any $t \in [1, n]$, our decremental transitive closure algorithm has $\widetilde{O}(n^2/t)$ total update time and $\widetilde{O}(\sqrt{t})$ query time. Additionally, as a byproduct, we obtain nearly-linear time algorithms for several other static and dynamic reachability-related problems on planar directed graphs.

Finally, we consider the problem of computing shortest paths in so-called dense distance graphs, a basic building block for designing efficient planar graph algorithms. Fakcharoenphol and Rao proposed an efficient implementation of Dijkstra's algorithm (later called *FR-Dijkstra*) computing single-source shortest paths in a dense distance graph. At the heart of their implementation lies a data structure updating distance labels maintained by Dijkstra's algorithm and extracting minimum labeled vertices in $O(\log^2 n)$ amortized time per vertex. We show a more efficient data structure accomplishing the same task in $O\left(\frac{\log^2 n}{\log^2 \log n}\right)$ amortized time per vertex. This yields improved time bounds for all problems on planar graphs for which computing shortest paths in dense distance graphs is currently a bottleneck, such as maximum bipartite matching, single-source all-sinks maximum flow, and dynamic all-pairs shortest paths.

2012 ACM Subject Classification: Theory of computation → Data structures design and analysis, Graph algorithms analysis

Keywords: Planar graph algorithms, Data structures, Dynamic graph algorithms, Decremental reachability, Shortest paths

Streszczenie

W tej rozprawie pokazujemy nowe struktury danych użyteczne w projektowaniu algorytmów na grafach planarnych.

Po pierwsze, pokazujemy optymalną strukturę danych utrzymującą graf planarny zmieniany za pomocą kontrakcji krawędzi. Nasza struktura danych jawnie utrzymuje listy sąsiedztwa i stopnie poszczególnych wierzchołków, a także obsługuje zapytania o sąsiedztwo wierzchołków w czasie stałym. Po każdej kolejnej kontrakcji, struktura jest aktualizowana w zamortyzowanym czasie stałym. Używając jej, uzyskujemy optymalne algorytmy dla kilku problemów na grafach planarnych, takich jak znajdowanie unikalnego doskonałego skojarzenia czy obliczanie maksymalnych podgrafów 3-spójnych krawędziowo. Pokazujemy dodatkowo, że wykorzystując naszą strukturę danych, można uzyskać proste koncepcyjnie optymalne algorytmy dla problemów minimalnego drzewa rozpinającego i 5-kolorowania grafów planarnych.

Następnie zajmujemy się problemem dekrementalnej osiągalności w skierowanych grafach planarnych. W szczególności, pokazujemy prawie optymalny dekrementalny algorytm dla problemu osiągalności z jednego źródła. Uzyskujemy także algorytm dla problemu dekrementalnego domknięcia przechodniego. Dla dowolnego $t \in [1, n]$, algorytm ten ma całkowity czas zmian rzędu $\widetilde{O}(n^2/t)$ i czas zapytania rzędu $\widetilde{O}(\sqrt{t})$. Dodatkowo, niejako przy okazji, otrzymujemy prawie liniowe algorytmy dla kilku innych statycznych i dynamicznych problemów związanych z osiągalnością w skierowanych grafach planarnych.

W końcu, zajmujemy się problemem znajdowania najkrótszych ścieżek w tzw. gęstych grafach odległości. Gęste grafy odległości, wraz z algorytmami operującymi nimi, są podstawowym narzędziem używanym przy projektowaniu efektywnych algorytmów dla grafów planarnych. Fakcharoenphol i Rao zaproponowali podali bardzo efektywną implementację algorytmu Dijkstry (nazwaną później FR-Dijkstra) na gęstych grafach odległości. Serce tej implementacji stanowi struktura danych aktualizująca oszacowania odległości utrzymywane przez algorytm Dijkstry i wyłuskująca nieodwiedzony wierzchołek o minimalnym oszacowaniu w zamortyzowanym czasie $O(\log^2 n)$ na wierzchołek grafu. My pokazujemy bardziej efektywną strukturę danych, obsługującą te same operacje w zamortyzowanym czasie $O\left(\frac{\log^2 n}{\log^2 \log n}\right)$ na wierzchołek gęstego grafu odległości. W ten sposób uzyskujemy lepsze oszacowania złożoności czasowej wszystkich tych problemów na grafach planarnych, dla których obliczanie najkrótszych ścieżek w gęstym grafie odległości jest wąskim gardłem najlepszego znanego dotychczas algorytmu. Są to, przykładowo, maksymalne skojarzenie dwudzielne, maksymalny przepływ o wielu źródłach i ujściach, czy dynamiczne najkrótsze ścieżki między wszystkimi parami wierzchołków.

Tytuł rozprawy w języku polskim: Struktury danych i algorytmy dynamiczne dla grafów planarnych.

Słowa kluczowe: Algorytmy dla grafów planarnych, Struktury danych, Dynamiczne algorytmy grafowe, Dekrementalna osiągalność, Najkrótsze ścieżki

Contents

Chapter 1

Introduction

Obtaining provably efficient algorithms for the most basic graph problems like finding (shortest) paths or computing maximum matchings, fast enough to handle real-world-scale graphs (i.e., consisting of millions of vertices and edges), is a very challenging task. For example, in a very general regime of strongly-polynomial algorithms (see, e.g., [86]), we still do not know how to compute shortest paths in a real-weighted sparse directed graph significantly faster than in quadratic time, using the classical, but somewhat simple-minded, Bellman-Ford method.

One way to circumvent this problem is to consider more restricted computation models for graph algorithms. If, for example, we restrict ourselves to graphs with integral edge weights, we can improve upon the Bellman-Ford algorithm [18, 41]. Although these results are very deep algorithmically, their theoretical efficiency is still very far from the only known trivial linear lower bound on the actual time complexity of the negatively-weighted shortest path problem.

Another approach is to develop algorithms specialized for certain graph classes that appear in practice. Planar graphs constitute one of the most important and well-studied such classes. Many of the real-world networks can be drawn on a plane with no or few edge crossings. The examples include not very complex road networks and graphs considered in the domain of VLSI design. Complex road networks, although far from being planar, share with planar graphs some useful properties, like the existence of small separators [24]. Special cases of planar graphs, such as grids, appear often in the area of image processing (e.g., [10]).

And indeed, if we restrict ourselves to planar graphs, many of the classical polynomial-time graph problems, in particular computing shortest paths [47, 77] and maximum flows [6, 7, 25] in real-weighted graphs, can be solved either optimally or in nearly-linear time. The very rich combinatorial structure of planar graphs often allows breaking barriers that appear in the respective problems for general graphs by using techniques from computational geometry (e.g., [32]), or by applying sophisticated data structures, such as dynamic trees [6, 13, 25, 87].

In this thesis, we focus on the data-structural aspect of planar graph algorithmics. By this, we mean that rather than concentrating on particular planar graph problems, we study more abstract, "low-level" problems. Efficient algorithms for these problems can be used in a black-box manner to design algorithms for multiple specific problems at once. Such an approach allows us to improve upon many known complexity upper bounds for different planar graph problems simultaneously, without going into the specifics of these problems.

We also study *dynamic algorithms* for planar graphs, i.e., algorithms that maintain certain information about a dynamically changing graph (such as "is the graph connected?") much more efficiently than by recomputing this information from scratch after each update. We consider the *edge-update* model where the input graph can be modified only by adding or removing single edges. A graph algorithm is called *fully-dynamic* if it supports both edge insertions and edge deletions, and *partially dynamic* if it supports either only edge insertions (then we call it

incremental), or only edge deletions (then it is called *decremental*).

When designing dynamic graph algorithms, we care about the *update time*, i.e., the time needed by the algorithm to adapt to an elementary change of the graph, and *query time*, i.e., the time needed by the algorithm to recompute the requested portion of the maintained information. Sometimes, especially in partially dynamic settings, it is more convenient to measure the *total update time*, i.e., the total time needed by the algorithm to process any possible sequence of updates. For some dynamic problems, it is worth focusing on a more restricted *explicit maintenance* model where the entire maintained information is explicitly updated (so that the user is notified about the update) after each change. In this model the query procedure is trivial and thus we only care about the update time.

Note that there is actually no clear distinction between dynamic graph algorithms and graph data structures since dynamic algorithms are often used as black-boxes to obtain efficient *static* algorithms (e.g., [31]). For example, the *incremental connectivity problem*, where one needs to process queries about the existence of a path between given vertices, while the input undirected graph undergoes edge insertions, is actually equivalent to the *disjoint-set data structure* problem, also called the *union-find data structure* problem (see, e.g., [19]).

We concentrate mostly on the decremental model and obtain very efficient decremental algorithms for problems on unweighted planar graphs related to reachability and connectivity. We also apply our dynamic algorithms to static problems, thus confirming once again the data-structural character of these results.

In the following, let $G = (V, E)$ denote the input planar graph with n vertices. For clarity of this introduction, assume G is a simple graph. Then, by planarity, it has $O(n)$ edges. When we talk about general graphs, we denote by m the number of edges of the graph.

1.1 Contracting a Planar Graph

Edge contraction is one of the fundamental graph operations. Given an undirected graph and its edge e, contracting the edge e consists in removing it from the graph and merging its endpoints. The notion of contraction has been used to describe a number of prominent graph algorithms, including Edmonds' algorithm for computing maximum matchings [23], or Karger's minimum cut algorithm [60].

Edge contractions are of particular interest in planar graphs as a number of planar graph properties can be described using contractions. For example, it is well-known that a graph is planar precisely when it cannot be transformed into K_5 or $K_{3,3}$ by contracting edges, or removing vertices or edges (see e.g., [21]). Moreover, contracting an edge preserves planarity.

We would like to have at our disposal a data structure that performs contractions on the input planar graph and still provides access to the most basic information about our graph, such as the sizes of neighbors sets of individual vertices and the adjacency relation. While contraction operation is conceptually very simple, its efficient implementation is challenging. This is because it is not clear how to represent individual vertices' adjacency lists so that adjacency list merges, adjacency queries, and neighborhood size queries are all efficient. By using standard data structures (e.g., balanced binary search trees), one can maintain adjacency lists of a graph subject to contractions in polylogarithmic amortized time. However, in many planar graph algorithms this becomes a bottleneck.

As an example, consider the problem of computing a 5-coloring of a planar graph. There exists a very simple algorithm based on contractions [71] that only relies on a folklore fact that a planar graph has a vertex of degree no more than 5. However, linear-time algorithms solving this problem use some more involved planar graph properties [28, 71, 81]. For example, the algorithm by Matula et al. [71] uses the fact that every planar graph has either a vertex of

degree at most 4 or a vertex of degree 5 adjacent to at least four vertices, each having degree at most 11. Similarly, although there exists a very simple algorithm for computing a minimum spanning tree of a planar graph based on edge contractions, various different methods have been used to implement it efficiently [28, 69, 70].

Our Results

We show a data structure that can efficiently maintain a planar graph subject to edge contractions in linear total time, assuming the standard word-RAM model with word size $\Omega(\log n)$. It can report groups of parallel edges and self-loops that emerge. It also supports constant-time adjacency queries and maintains the neighbor lists and degrees explicitly. The data structure can be used as a black-box to implement planar graph algorithms that use contractions. As an example, it can be used to give clean and conceptually simple implementations of algorithms for computing 5-coloring or minimum spanning tree that do not manipulate the embedding. More importantly, by using our data structure, we give improved algorithms for a few problems in planar graphs. In particular, we obtain optimal algorithms for decremental 2-edge-connectivity, finding a unique perfect matching, and computing maximal 3-edge-connected subgraphs.

Related Work

The problem of detecting self-loops and parallel edges under contractions has been implicitly addressed by Giammarresi and Italiano [40] in their work on decremental (edge-, vertex-) connectivity in planar graphs. Their data structure uses $O(n \log^2 n)$ total time.

In their book, Klein and Mozes [63] showed that there exists a data structure maintaining a planar graph under edge contractions and deletions, and answering adjacency queries in $O(1)$ worst-case time. The update time is $O(\log n)$. This result is based on the work of Brodal and Fagerberg [11], who showed how to maintain a bounded-outdegree orientation of a dynamic planar graph so that edge insertions and deletions are supported in $O(\log n)$ amortized time.

Gustedt [43] showed an optimal solution to the union-find problem in the case when at any time the actual subsets form disjoint and connected subgraphs of a given planar graph G. In other words, in this problem the allowed unions correspond to the edges of a planar graph and the execution of a union operation can be seen as a contraction of the respective edge.

Technical Overview

As mentioned before, it is relatively easy to design a simple *vertex merging data structure* for general graphs that would process any sequence of contractions in $O(m \log^2 n)$ total time and support the same queries as our data structure in $O(\log n)$ time. To this end, one can store the neighbors lists of individual vertices as balanced binary search trees. Upon a contraction of an edge uv, or a more general operation of merging two (not necessarily adjacent) vertices u, v, the neighbors lists of u and v are merged by inserting the smaller set into the larger one (and detecting loops and parallel edges on the fly at no additional cost). If we used hash tables instead of balanced binary search trees, we could achieve $O(\log n)$ expected amortized update time and $O(1)$ query time. In fact, such an approach was used in [40].

To obtain the speed-up we take advantage of planarity. Our general idea is to partition the graph into small pieces and use the above simple-minded vertex merging data structures to solve our problem separately for each of the pieces and for the subgraph induced by the vertices contained in multiple pieces (the so-called boundary vertices). Due to the nature of edge contractions, we need to specify how the partition evolves when our graph changes.

The data structure builds a so-called r-division (see Chapter 2) $\mathcal{R} = P_1, P_2, \ldots$ of the input graph for $r = \log^4 n$, i.e., it partitions G into edge-disjoint pieces of size $O(r)$. The set $\partial\mathcal{R}$ of boundary vertices (i.e., those shared by at least two pieces) has size $O(n/\log^2 n)$. Let (V_0, E_0) denote the original graph, and (V, E) denote the current graph (after performing some number of contractions). Since edge contractions merge the vertices of G, V can be viewed as a partition of V_0. Then, when an edge $e = uv$, where $u, v \in V$, is contracted, the vertices u and v are replaced with their union $u \cup v$. Let us denote by $\phi : V_0 \to V$ the unique function such that for each initial vertex $v_0 \in V_0$, where $v_0 \in \phi(v_0)$. We use vertex merging data structures to detect parallel edges and self-loops in the "top-level" subgraph $G[\phi(\partial\mathcal{R})]$, which contains only edges between boundary vertices, and separately in the "bottom-level" subgraphs $G[\phi(V(P_i))] \setminus G[\phi(\mathcal{R})]$. At any time, each edge of G is contained in exactly one of the defined subgraphs, and thus, the distribution of responsibility for handling individual edges is based solely on the initial r-division.

However, such an assignment of responsibilities gives rise to additional difficulties. First, a contraction of an edge in a lower-level subgraph might cause some edges "flow" from this subgraph to the top-level subgraph (i.e., we may get new edges connecting boundary vertices). As such an operation turns out to be costly in our implementation, we need to prove that the number of such events throughout is only $O(n/\log^2 n)$.

Another difficulty lies in the need of keeping the individual data structures synchronized: when an edge of the top-level subgraph is contracted, pairs of vertices in multiple lower-level subgraphs might need to be merged. We cannot afford iterating through all the lower-level subgraphs after each contraction in $G[\phi(\partial\mathcal{R})]$. This problem is solved by maintaining a system of pointers between representations of the same vertex of V in different vertex-merging data structures and another clever application of the smaller-to-larger merge strategy.

Such a two-level data structure would yield a data structure with $O(n \log\log n)$ total update time. To obtain a linear time data structure, we further partition the pieces P_i and add another layer of maintained subgraphs on $O(\log^4 \log^4 n) = O(\log^4 \log n)$ vertices. These subgraphs are so small that we can precompute in $O(n)$ time the self-loops and parallel edges for every possible graph on $t = O(\log^4 \log n)$ vertices and every possible sequence of edge contractions.

We note that this overall idea of recursively reducing a problem with an r-division to a size when micro-encoding can be used has been previously exploited in [43] and [68] (Gustedt [43] did not use r-divisions, but his concept of a *patching* could be replaced with an r-division). Our data structure can be also seen as a solution to a more general version of the planar union-find problem studied by Gustedt [43]. However, maintaining the status of each edge e of the initial graph G (i.e., whether e has become a self-loop or a parallel edge) subject to edge contractions, turns out to be a serious technical challenge. For example, in [43], the requirements posed on the bottom-level union-find data structures are in a sense relaxed and it is not necessary for those to be synchronized with the top-level union-find data structure.

1.2 Decremental Reachability

In the *dynamic reachability* problem we are given a (directed) graph G subject to edge updates and the goal is to design a data structure that would allow answering queries about the existence of a path between a pair of query vertices $u, v \in V$.

Two variants of dynamic reachability are studied most often. In the *all-pairs* variant, our data structure has to support queries between arbitrary pairs of vertices. This variant is also called the *dynamic transitive closure* problem since a path $u \to v$ exists in G if uv is an edge of the transitive closure of G.

In the *single-source reachability* problem, a source vertex $s \in V$ is fixed from the very beginning and the only allowed queries are about the existence of a path $s \to v$, where $v \in V$.

If we work with undirected graphs, the dynamic reachability problem is called the *dynamic connectivity* problem. Note that in the undirected case a path $u \to v$ exists in G if and only if a path $v \to u$ exists in G.

State of the Art

Dynamic reachability in general directed graphs turns out to be a very challenging problem. First of all, it is computationally much more demanding than its undirected counterpart. For undirected graphs, fully-dynamic all-pairs algorithms with polylogarithmic amortized update and query bounds are known [48, 52, 95]. For directed graphs, on the other hand, in most settings (either single-source or all-pairs, either incremental, decremental or fully-dynamic) the best known algorithm has either polynomial update time or polynomial query time. The only exception is the incremental single-source reachability problem, for which a trivial extension of depth-first search [90] achieves $O(1)$ amortized update time.

One of the possible reasons behind such a big gap between the undirected and directed settings is that one needs only linear time and space to compute the connected components of an undirected graph and thus there exists a $O(n)$-space *static* data structure that can answer connectivity queries in undirected graphs in $O(1)$ time. On the other hand, the best known algorithm for computing the transitive closure runs in $\widetilde{O}(\min(n^\omega, nm)) = \widetilde{O}(n^2)$[1] time [14, 78].

So far, the best known bounds for fully-dynamic reachability are as follows. For dynamic transitive closure, there exist a number of algorithms with $O(n^2)$ update time and $O(1)$ query time [20, 82, 85]. These algorithms, in fact, maintain the transitive closure explicitly. There also exist a few fully-dynamic algorithms that are better for sparse graphs, each of which has $\Omega(n)$ amortized update time and query time which is $o(n)$ but still polynomial in n [83, 84, 85]. For the single-source variant, the only known non-trivial (i.e., other than recompute-from-scratch) algorithm has $O(n^{1.53})$ update time and $O(1)$ query time [85].

Algorithms with $O(nm)$ total update time are known for both incremental [53] and decremental [66, 83] transitive closure. Note that for sparse graphs this bound is only poly-logarithmic factors away from the best known static transitive closure upper bound [14].

All the known partially-dynamic single-source reachability algorithms work in the explicit maintenance model. As mentioned before, for incremental single-source reachability, an optimal (in the amortized sense) algorithm is known. Interestingly, the first algorithms with $O(mn^{1-\epsilon})$ total update time (where $\epsilon > 0$) have been obtained only recently [44, 45]. The best known algorithm to date has $\widetilde{O}(m\sqrt{n})$ total update time and is due to Chechik et al. [16].

Dynamic reachability has also been previously studied for planar graphs. Diks and Sankowski [22] showed a fully-dynamic transitive closure algorithm with $\widetilde{O}(\sqrt{n})$ update and query times, which works under the assumption that the graph is plane embedded and the inserted edges can only connect vertices sharing some adjacent face. Łącki [66] showed that one can maintain the strongly connected components of a planar graph under edge deletions in $O(n\sqrt{n})$ total time. By known reductions, it follows that there exists a decremental single-source reachability algorithm for planar graphs with $O(n\sqrt{n})$ total update time. Note that this bound matches the recent best known bound for general graphs [16] up to polylogarithmic factors.

Our Results

We show new decremental reachability algorithms for planar digraphs.

[1] We denote by $\widetilde{O}(f(n))$ the order $O(f(n)\operatorname{polylog} n)$.

For decremental single-source reachability, we obtain an almost optimal (up to polylogarithmic factors) algorithm explicitly maintaining the set of vertices reachable from the source. Our algorithm processes any online sequence of edge deletions in $O(n \log^2 n \log \log n)$ total time.

For decremental transitive closure, we obtain a randomized trade-off algorithm. For any chosen $t \in [1, n]$, our algorithm has $\widetilde{O}(n^2/t)$ total update time and $\widetilde{O}(\sqrt{t})$ query time. In particular, for $t = n$, our algorithm has polylogarithmic amortized update time and $\widetilde{O}(\sqrt{n})$ time. For $t = n^{2/3}$, we obtain an algorithm with $\widetilde{O}(n^{1/3})$ update and query time. To the best of our knowledge, this is the first dynamic algorithm for general planar digraphs answering arbitrary point-to-point queries with $O(n^{1/2-\epsilon})$ update and query bounds.

As a byproduct, we also obtain nearly-linear time planar algorithms for a few static and decremental reachability-related problems that have been previously studied only for general graphs, and for which no nearly-linear algorithms have been known. These include computing maximal 2-edge-connected subgraphs of a directed graph [15] and decremental maintenance of the set of so-called strong bridges of the graph [37].

Technical Overview

All our reachability-related algorithms are obtained using a uniform approach. We first construct a *recursive decomposition* of the initial graph G. A recursive decomposition is a tree-like hierarchy of subgraphs of a graph G (pieces) built by recursively partitioning G with $O(\sqrt{n})$-size cycle separators [72]. For each piece H of the decomposition, the set of its *boundary vertices* ∂H is defined to be the set of vertices of H shared with its complement $G - H$ (defined, in turn, as the subgraph of G induced by the edges $E(G) \setminus E(H)$). The used recursive decomposition algorithm guarantees that ∂H lies on a small number of faces of H and the size of ∂H is small compared to the size of H. Such decompositions proved very useful in obtaining nearly-linear time algorithms for planar graphs (e.g., [5, 8, 67]), as well as dynamic planar graph algorithms (e.g., [22]).

Subsequently, for each piece H, we explicitly maintain only certain parts of transitive closures of graphs H and $G-H$. Specifically, we are only interested in the existence of the paths between the vertices ∂H in H and $G-H$. This way, as we show, the total amount of information we store, which is $O(|\partial H|^2)$ per piece, is nearly-linear in n. We prove that these parts of the transitive closures of H and $G - H$ can be computed inductively by taking a transitive closure of the union of the corresponding parts of transitive closures stored in the children and the parent of H. Therefore, the needed information can be *maintained* inductively using a decremental transitive closure algorithm run on the corresponding data accompanying the children and the parent pieces of H.

In order to obtain an efficient algorithm maintaining the needed reachability information dynamically, as described above, we analyze and exploit the structural properties of a reachability matrix of a set of vertices[2] that, roughly speaking, lie on a constant number of faces of a plane graph (which is a property satisfied by vertices ∂H of each piece H in our decomposition). The matrix viewpoint allows us to obtain properties that are algorithmically useful and otherwise not easy to capture using the previously used *separating path* approach to reachability in planar digraphs [22, 89, 91]. Subsequently, we show that these properties can be used to simulate the randomized decremental transitive closure algorithm of Bernstein [4] very efficiently, so that the total update time of our recursive data structure is still nearly-linear in n.

Finally, we show that the maintained reachability information is sufficient to support reachability queries in $\widetilde{O}(\sqrt{n})$ time and explicitly maintain the strongly-connected components of G

[2] A reachability matrix of a set S is a submatrix of the transitive closure matrix consisting of rows and columns corresponding to vertices belonging to S.

under edge deletions. By known black-box reductions, this is sufficient to solve decremental single-source reachability in nearly-linear time. Using a technique of [76], we can reduce the point-to-point query time to $\widetilde{O}(\sqrt{t})$ at the cost of increasing the total update time to $\widetilde{O}(n^2/t)$.

Note that this way we obtain randomized algorithms. We also show that by using plane duality we can, in a sense, turn the decremental strongly-connected components problem into a certain incremental reachability problem, where the edges of a graph (fixed from the beginning) are switched-on in an online manner. This problem is solved using an analogous recursive data structure, but now the reachability information between the boundary vertices of individual pieces has to be maintained incrementally. As a result, we can replace Bernstein's decremental transitive closure algorithm with a conceptually simpler, "folklore" incremental algorithm which is more efficient and deterministic at the same time. Consequently, we obtain a deterministic decremental single-source reachability algorithm with $O(n \log^2 n \log\log n)$ total update time.

1.3 Shortest Paths in Dense Distance Graphs

In their breakthrough paper [27], Fakcharoenphol and Rao introduced the general concept of a *dense distance graph*. Let G be a non-negatively-weighted plane digraph and let U denote some subset of its "boundary" vertices lying on some $O(1)$ faces of G. Such graphs with a topologically nice boundary typically emerge after decomposing a plane graph using a cycle separator. For example, by using a cycle separator of Miller [72], one can decompose any n-vertex triangulated plane graph H into two subgraphs H_{in} and H_{out} such that (i) $H_{\text{in}} \cup H_{\text{out}} = H$; (ii) H_{in} and H_{out} are smaller than H by a constant factor; (iii) the set $U = V(H_{\text{in}}) \cap V(H_{\text{out}})$ has size $O(\sqrt{n})$, and lies both on a single face of H_{in} and on a single face of H_{out}.

We define a *distance clique* of G, denoted $\text{DC}(G)$, to be a complete graph on U such that the weight of an edge uv is equal to the length of the shortest path from u to v in G. A dense distance graph is a union of possibly many unrelated distance cliques $\text{DC}(G_1), \ldots, \text{DC}(G_q)$.

We note that such a definition of a dense distance graph (also used in [79]) is a bit more general than in [27], where a dense distance graph was only defined with respect to a recursive decomposition of G using cycle-separators. In fact, subsequently dense distance graphs have been also defined a bit differently with respect to so-called r-divisions [58], and even the two sides of a cycle-separator [63] (i.e., $\text{DC}(H_{\text{in}}) \cup \text{DC}(H_{\text{out}})$ in the above example). Our definition captures all these cases.

Fakcharoenphol and Rao [27] proposed an efficient implementation of Dijkstra's algorithm (later called *FR-Dijkstra*) computing single-source shortest paths in a dense distance graph. Their algorithm spends $O(b \log b \log n)$ time per distance clique with b vertices, even though a clique has b^2 edges. Here, n is the total number of vertices of the dense distance graph. Whereas Dijkstra's algorithm uses a priority queue to maintain its distance labels and extract a non-visited vertex with minimum label, a much more sophisticated data structure is used in FR-Dijkstra. This data structure is capable of relaxing many edges in a single step, by leveraging the fact that certain submatrices of the weight matrix of a distance clique constitute so-called *Monge matrices* [94].

Fakcharoenphol and Rao originally employed FR-Dijkstra to construct their dense distance graph recursively, and consequently solve the real-weighted single-source shortest paths problem on planar graphs in nearly-linear time. However, the applications of FR-Dijkstra proved much broader. As a result, it has become an important planar graph primitive used to obtain numerous breakthrough results in recent years. We briefly cover the most important of these results below.

The dense distance graphs and FR-Dijksta have been used to break the long-standing $O(n \log n)$ barrier for computing minimal s,t-cuts [55] in undirected planar graphs, and global

min-cuts in both undirected [65] and directed [74] planar graphs. Borradaile et al. [8] developed an oracle answering arbitrary min s,t-cut queries in a weighted undirected planar graph after only nearly-linear preprocessing. This result has been later generalized to bounded-genus graphs [5], thus proving the usefulness of FR-Dijkstra in more general graph classes.

The most sophisticated applications of FR-Dijkstra to date are probably those related to computing maximum flow in directed planar graphs. Borradaile et al. [7] gave a nearly-linear time max-flow algorithm for the case of multiple sources and multiple sinks, and consequently, a nearly-linear algorithm for maximum bipartite matching. Later, Łącki et al. [67] gave a nearly-linear time algorithm computing the maximum flow values between a specified source and all possible sinks.

Most recently, Asathulla et al. [3] used FR-Dijkstra to break through the $O(n^{3/2})$ barrier for the planar assignment problem with integer weights. Cabello [12] showed the first truly subquadratic algorithm for computing a diameter of a weighted planar graph. Even though it mainly builds on a new concept of additively-weighted Voronoi diagrams for planar graphs, dense distance graphs and FR-Dijkstra are still used extensively in his work. The diameter algorithm was later improved by Gawrychowski et al. [32] to run in $O(n^{5/3}\operatorname{polylog} n)$. Currently, the diameter algorithm of [32] does not require FR-Dijkstra, but it seems that using it would be again required if one gave a more efficient Voronoi diagrams construction algorithm for planar graphs.

Last but not least, FR-Dijkstra has been instrumental in obtaining virtually all sublinear update/query time *exact* dynamic algorithms for shortest paths, maximum flows and minimum cuts in planar graphs [27, 55, 58, 62, 65].

Related Work

Dense distance graphs are pivotal in designing efficient planar graph algorithms, and therefore obtaining fine-grained bounds for computing and manipulating them is an important research direction. Although a better algorithm (in comparison to the recursive method of [27]), running in $O((|V|+|U|^2)\log n)$ time, has been proposed for computing a distance clique [62], improving FR-Dijkstra itself proved very challenging and no progress over [27] has been made in the most general setting so far.

For the important case of a *dense distance graph over an r-division*, i.e., when the individual graphs G_i are the pieces of an *r-division with few holes* of a single planar graph (see e.g., [64]), Mozes et al. [75] gave an algorithm for computing single source shortest paths in $O\left(\frac{n}{\sqrt{r}}\log^2 r\right)$ time. The original FR-Dijkstra runs in $O\left(\frac{n}{\sqrt{r}}\log n\log r\right)$ time in that case. Hence, [75] does not improve over it in the case of $r=\operatorname{poly} n$ which emerges in many important applications, e.g., [3, 5, 7, 58, 67]. However, dense distance graphs over r-divisions with $r=\operatorname{polylog}(n)$ have also found applications, most notably in $O(n\log\log n)$ algorithms for minimum cuts [55, 65, 74]. Computing shortest paths in dense distance graphs is not a bottleneck in those algorithms, though. For other applications of dense distance graphs over r-divisions with small r, consult [75].

Our Results

We show an algorithm for computing single-source shortest paths in a dense distance graph with $O\left(b\left(\frac{\log^2 b}{\log^2\log b}+\log^\epsilon b\log n\right)\right)=O\left(b\frac{\log^2 n}{\log^2\log n}\right)$ time overhead per distance clique with b vertices, for any $\epsilon\in(0,1)$. Our algorithm is asymptotically faster than FR-Dijkstra in *all* cases. Specifically, for a dense distance graph defined over an r-division the algorithm runs in

$O\left(\frac{n}{\sqrt{r}}\frac{\log^2 n}{\log^2 \log n}\right)$ time.

Our result implies an immediate improvement by a factor of $O(\log^2 \log n)$ in the time complexity for a number of planar digraph problems such as multiple-source multiple-sink maximum flows, maximum bipartite matching [7], single-source all-sinks maximum flows [67] for which the best known time bounds were $O(n \log^3 n)$, i.e., already nearly-linear. It also yields polylog-logarithmic speed-ups to both preprocessing and query/update algorithms of dynamic algorithms for shortest paths and max-flows [55, 58, 62]. More generally, we make polylog-logarithmic improvements to all previous results (such as [3]), for which the bottleneck of the best known algorithm is computing shortest paths in a dense distance graph.

It should be noted that for small values of r, such as $r = \text{polylog}(n)$, our algorithm does not improve upon [75] for the case of a dense distance graph over an r-division.

Technical Overview

We treat the problem of computing shortest paths in a dense distance graph from a purely data-structural perspective. At a high level, instead of developing an entirely new shortest paths algorithm, we propose a new data structure for maintaining distance labels and extracting minimum labeled vertices in amortized $O\left(\frac{\log^2 b}{\log^2 \log b}\right)$ time, as opposed to $O(\log^2 b)$ time in [27].

In [27], a distance clique is first partitioned into *square* Monge matrices, each handling a subset of the clique's edges. For any such matrix, a separate data structure is used for relaxing the corresponding edges and extracting the labels possibly induced by these edge relaxations. Recall that in the case of Dijkstra's algorithm, the improvement from $O(m \log n)$ to $O(m + n \log n)$ time is obtained by noticing that relaxing edges is cheaper than extracting minimum labeled vertices. Consequently, one can use a Fibonacci heap [30] in place of a binary heap. We show that in the case of the data structure originally used in [27] for handling Monge matrices the situation is in a sense the opposite: label extractions can be made cheaper than edge relaxations. We make use of this fact by proposing a different than in [27], biased scheme of partitioning distance cliques into *rectangular* (as opposed to square) Monge matrices. Whereas in [27] the partition follows from a very natural idea of splitting a face boundary into halves, our partition is tailored to exploit this asymmetry between the cost of processing a row and the cost of processing a column. A more detailed overview can be found in Chapter 5, after all the needed notions are introduced.

1.4 Organization of This Thesis

In Chapter 2 we introduce the notation that we use throughout the thesis.

The data structure for maintaining a planar graph under contractions, along with its applications, is described in detail in Chapter 3.

Decremental reachability algorithms for planar graphs are described in Chapter 4.

In Chapter 5 we show our improved algorithm for computing single-source shortest paths in dense distance graphs and discuss some of its implications.

Finally, in Chapter 6 we highlight some open problems and possible further research directions that are closely related to this thesis' findings.

1.5 Articles Comprising This Thesis

The preliminary versions of the contents of this thesis have been included in the following conference papers.

- *Contracting a Planar Graph Efficiently*, joint work with Jacob Holm, Giuseppe F. Italiano, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski, published at ESA 2017 [49].
- *Decremental Single-Source Reachability in Planar Digraphs*, joint work with Giuseppe F. Italiano, Jakub Łącki, and Piotr Sankowski, published at STOC 2017 [54].
- *Decremental Transitive Closure and Shortest Paths for Planar Digraphs and Beyond*, published at SODA 2018 [59].
- *Improved Bounds for Shortest Paths in Dense Distance Graphs*, joint work with Paweł Gawrychowski, published at ICALP 2018 [33].

1.6 Acknowledgments

First of all, I would like to thank my supervisor, Piotr Sankowski, for all the support, many inspiring discussions, and giving me a lot of freedom in pursuing these topics.

I would like to thank Jakub Łącki with whom I collaborated the most, for many fruitful discussions, sharing the passion for dynamic graphs, and often serving as a role-model for me.

I would like to thank Tomasz Kociumaka for sharing the office room with me most of the time, helping a lot with bureaucracy, answering tons of stupid questions, and being a usual companion during many conference trips.

I am very grateful to Giuseppe F. Italiano for hosting me during a few memorable visits to Rome.

I would also like to thank the remaining co-authors of my publications: Paweł Gawrychowski, Jacob Holm, and Eva Rotenberg.

Finally, I would like to thank my family, and especially my wonderful wife who is the most important to me. While working on (the contents of) this thesis I achieved so much more in my private life and this truly kept me happy throughout this time.

My research was supported by the Polish National Science Centre grant no. 2014/13/B/ST6/01811 and the Polish National Science Centre scholarship "Etiuda" no. 2017/24/T/ST6/00036.

Chapter 2

Preliminaries

2.1 Graphs

Let $G = (V, E)$ be a graph. We study both *undirected* and *directed* graphs (*digraphs*). If not stated explicitly, it should be clear from the context at all times whether the graphs we consider are undirected or directed.

If G is undirected, then each edge $e \in E$ is formally a pair $(\{u, v\}, \mathrm{id}(e))$ where $u, v \in V$ and $\mathrm{id}(e)$ is an identifier that is unique among the identifiers of all elements of E. When G is directed, each edge $e \in E$ is a pair $((u, v), \mathrm{id}(e))$ where $u, v \in V$. If $u = v$, then e is called a *self-loop*. If there are two distinct edges $e_1, e_2 \in E$ such that the first coordinates of e_1 and e_2 are equal, we say that e_1 and e_2 are *parallel* to each other. If the graph contains no parallel edges and no self-loops, we call it *simple*.

The identifiers $\mathrm{id}(e)$ serve two purposes. First, they are used to distinguish between the parallel edges of G. Moreover, sometimes we use them to describe a correspondence between the edges of one graph G_1 and the (subset of) edges of some other graph G_2. The edges of two graphs correspond to each other if they have equal identifiers. Sometimes it is convenient to view the identifiers as integers 1 through $|E|$.

For simplicity, we often use the notation uv to denote one of the edges connecting vertices u and v, i.e., when the first coordinate of the edge is $\{u, v\}$ if G is undirected, or (u, v) if G is directed. Hence, uv and vu have the same meaning if G is undirected, whereas uv and vu are different if G is directed. We write $uv = e \in E$ to refer to some specific edge uv. A vertex $w \in V$ is *incident* to an edge $uv = e \in E$ if $w \in \{u, v\}$.

We often deal with multiple different graphs at once. For a graph G, we let $V(G)$ and $E(G)$ denote the vertex and edge set of G, respectively.

If G is undirected, for each vertex $u \in V(G)$ we define $N_G(u) = \{v : uv \in E(G), u \neq v\}$ to be the *neighbor set* of v in G. On the other hand, if G is directed, then for each vertex u we define $N_G^{\mathrm{out}}(u) = \{v : uv \in E(G), u \neq v\}$ and $N_G^{\mathrm{in}}(u) = \{v : vu \in E(G), u \neq v\}$.

Let G_1, G_2 be two graphs. $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ is called the *union* of G_1 and G_2, whereas $G_1 \setminus G_2 = (V(G_1), E(G_1) \setminus E(G_2))$ is their *difference*.

We write $G + uv$ to denote the graph G with some edge uv added. For $E' \subseteq E(G)$, we write $G - E'$ to denote the graph $(V(G), E(G) \setminus E')$. For brevity, when $e \in E(G)$, we write $G - e$ to denote $G - \{e\}$, i.e., the graph obtained from G by removing a single edge e.

We consider both *unweighted* and *weighted* graphs. If a graph G is weighted, then we assume a function $w_G : E(G) \to \mathbb{R}$ is given. We can view unweighted graphs as weighted graphs with $w_G \equiv 1$.

Subgraphs. A graph G' is called a subgraph of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. For $S \subseteq V(G)$, we denote by $G[S]$ the *induced subgraph* $(S, \{e : uv = e \in E(G), \{u, v\} \subseteq S\})$. For $E' \subseteq E(G)$, we denote by $G[E']$ the *edge-induced subgraph* $(V_{E'}, E')$, where $V_{E'} \subseteq V(G)$ is the set of all vertices of G incident to at least one edge of E'. If G' is a subgraph of G, we denote by $G - G'$ the edge-induced subgraph $G[E(G) \setminus E(G')]$. Note that $G - G'$ is equal to the graph $G - E(G')$ with isolated vertices removed.

Paths and cycles. A *path* $P \subseteq G$ is a subgraph whose edges $E(P)$ can be ordered $e_1, \ldots, e_k$ such that if $e_i = u_i v_i$, then for each $i = 2, \ldots, k$ we have $u_i = v_{i-1}$. Such P is also called a $u_1 \to v_k$ path. A path P is *simple* if $u_i \neq u_j$ for $i \neq j$. A *cycle* is a path such that $u_1 = v_k$. A *simple cycle* is a cycle that is a simple path.

For $u, v \in V(G)$, we say that v is *reachable* from u if there exists a $u \to v$ path in G.

The *length* $\ell(P)$ of a path P is defined as $\sum_{i=1}^{k} w_G(e_i)$. If P is unweighted, then clearly $\ell(P) = k$.

A *shortest* $u \to v$ *path* is a $u \to v$ path with minimal length. Shortest paths are guaranteed to exist if the graph does not contain any cycles of negative length. This condition is trivially satisfied if the edge weights are non-negative.

Suppose that G contains no cycles of negative length. For $u, v \in V$, the *distance* $\delta_G(u, v)$ is the length of a shortest $u \to v$ path in G if v is reachable from u, and $\delta_G(u, v) = \infty$ otherwise.

Transitive closure and transitive reduction. Let G be an unweighted directed graph. The *transitive closure* G^+ of G is a simple directed graph on $V(G)$ such that $uv \in E(G^+)$ if and only if $u \neq v$ and v is reachable from u in G.

An edge $uv = e \in E(G)$ is called a 1-*cut edge* if v is not reachable from u in $G - e$.

A *transitive reduction* G^- of a directed graph G is a directed graph on $V(G)$ such that $G^+ = (G^-)^+$ and G^- has a minimum number of edges. In general, G^- may not be uniquely defined. However, if we limit ourselves to acyclic graphs then the following property holds.

Lemma 2.1.1 ([2])**.** *Let G be an acyclic digraph. Then G^- is unique and is a subgraph of G consisting exactly of the 1-cut edges of G.*

Connectivity and components. Let G be a directed graph. If u reachable from v and v is reachable from u, then u and v are *strongly connected*. Clearly, strong-connectivity is an equivalence relation on $V(G)$. The *strongly connected components* of G are the subgraphs of G induced by the equivalence classes of this relation. G is *strongly connected* if it has only one strongly connected component.

An edge $uv = e \in E(G)$ is called *intra-SCC* if u and v are strongly-connected. Otherwise, e is said to be an *inter-SCC* edge.

If G is undirected, v is reachable from u if and only if u and v are strongly connected. In this case we say that u and v are *connected*. We define *connected components* for both directed and undirected graphs: these are the strongly-connected components of the graph obtained from G by ignoring the edge directions. A graph G is *connected* if it consists of a single connected component.

Cutsets and higher edge-connectivity. Let G be an undirected graph. A subset $C \subseteq E(G)$ is called a *cutset* of G if $G - C$ has more connected components than G. A cutset C of G is *minimal* if no proper subset of C is a cutset of G. A k-*cutset* is a cutset of size k. The only element of a 1-cutset is traditionally called a *bridge*.

Two vertices $u, v \in V(G)$ are *k-edge-connected* if there exists k edge-disjoint paths between them. Equivalently, by Menger's theorem (see, e.g., [21]), u and v are k-edge-connected if u and v are in the same connected component of $G - C$ for any $(k-1)$-cutset C of G. It is known that k-edge-connectivity is an equivalence relation on $V(G)$. The *k-edge-connected components* are the equivalence classes of this relation. The graph G is called *k-edge connected* if all pairs of its vertices are k-edge-connected, or equivalently, if G has no $(k-1)$-cutsets.

For $k \geq 3$, the (subgraphs induced by) k-edge-connected components of G are generally not equal to the *maximal k-edge-connected subgraphs* of G. It might be the case that for some two vertices u, v of G that are k-edge-connected all sets of k edge-disjoint paths between them use vertices from outside the k-edge-connected component of u and v. On the other hand, if G' is a maximal k-edge-connected subgraph of G then clearly all vertices of G' lie in the same k-edge-connected component of G.

2-edge-strong-connectivity in directed graphs. Let G be a directed graph. We say that $u, v \in V(G)$ are 2-edge-strongly-connected if there are two edge-disjoint paths between u and v and two edge-disjoint paths between v and u. Equivalently, u and v are 2-edge-strongly-connected if u, v are strongly-connected in $G - e$ for any $e \in E(G)$. 2-edge-strong-connectivity is also an equivalence relation on $V(G)$ [38].

Similarly as was the case for k-edge-connectivity in undirected graphs for $k \geq 3$, (the subgraphs induced by) 2-edge-strongly-connected components of G are generally not equal to the *maximal 2-edge-strongly-connected subgraphs* of G.

We call $e \in E(G)$ a *strong bridge* (see, e.g., [15, 38]) if $G - e$ has more strongly-connected components than G.

Edge contraction. For $e \in E(G)$ that is not a self-loop, we denote by G/e the graph obtained by contracting e. We will often look at contraction from the following perspective: as a result of contracting e, all edge endpoints equal to x or y are replaced with some new vertex z. In some cases it is convenient to assume $z \in \{x, y\}$. This yields a 1-to-1 correspondence between the edges of $G - e$ and the edges of G/e. Formally, we assume that the contraction preserves edge identifiers, i.e., $e_1 \in E(G - e)$ and $e_2 \in E(G/e)$ are corresponding if and only if $\mathrm{id}(e_1) = \mathrm{id}(e_2)$.

Note that contracting an edge may introduce parallel edges and self-loops. For example, if G is undirected, then for each edge that is parallel to e in G, there is a self-loop in G/e. And for each cycle consisting of 3 edges that contains e in G, there is a pair of parallel edges in G/e.

A graph G' is called a *minor* of G if it can be obtained from G by performing on G a sequence of edge deletions, edge contractions, and vertex deletions.

2.2 Model of Computation

We assume the standard word-RAM model with word size $\Omega(\log n)$, where n is the number of vertices of the input graph.

However, all our algorithms for weighted graphs work even if the edge weights are *real numbers*. Whereas we can manipulate integers that fit into a single word using arithmetical and bitwise operations, the edge weights can only be manipulated by performing arithmetical operations and comparisons on them.

In our algorithms we sometimes use randomization. We say that a probabilistic statement holds *with high probability* if it holds with probability at least $1 - n^{\beta}$, where $\beta > 0$ is a real constant that can be fixed arbitrarily.

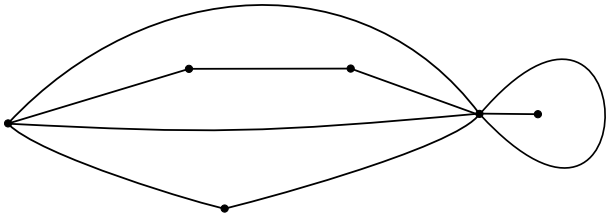

Figure 2.1: A semi-strict graph with 6 vertices and 5 faces.

2.3 Planar Graphs.

A *plane embedding* of a graph is a mapping of its vertices to distinct points and of its edges to non-crossing curves in the plane. We say that G is *plane embedded* (or *plane*, in short) if some embedding of G is assumed.

A graph G is *planar* if there exists a plane embedding of G. It is known that if G is planar then all its subgraphs and minors are also planar.

For any vertex v of a plane graph G we define an *edge ring* of v to be a circular list of all edges incident to v, ordered either clockwise or counter-clockwise according to the embedding of these edges around v. An edge ring can start with any edge incident to v.

A *face* of a connected plane G is a maximal open connected set of points that are not in the image of any vertex or edge in the embedding of G. There is exactly one *unbounded* face.

The *bounding cycle* $\mathrm{cyc}(f)$ of a bounded (unbounded, respectively) face f is a sequence of edges bounding f in clockwise (counterclockwise, respectively) order. Here, we ignore the directions of edges. An edge can appear in a bounding cycle at most twice.

We denote by $E(f)$ the set of edges that appear at least once in the sequence $\mathrm{cyc}(f)$. We denote by $V(f)$ the set of vertices incident to at least one edge of $E(f)$. We sometimes say that $v \in V(f)$ *lies* on the face f.

The *size* of a face is defined to be the length of its bounding cycle. We say that G is *triangulated* if all its faces have size 3.

The face is called *simple* if no edge appears in its bounding cycle twice and the subgraph induced by the edges of the bounding cycle forms a simple cycle.

By Jordan Curve Theorem, a Jordan curve $\mathcal{C}$ partitions $\mathbb{R}^2 \setminus \mathcal{C}$ into two connected regions, a bounded one B and an unbounded one U. We say that a set of points P is *strictly inside* (*strictly outside*) $\mathcal{C}$ if and only if $P \subseteq B$ ($P \subseteq U$, respectively). P is *weakly inside* (*weakly outside*) if and only if $P \subseteq B \cup \mathcal{C}$ ($P \subseteq U \cup \mathcal{C}$, respectively). We sometimes identify simple cycles of G and bounding cycles of simple faces of G with the respective Jordan curves obtained by concatenating the embeddings of the subsequent edges of these cycles.

We call a plane graph G *semi-strict* [63] if the bounding cycle of each of its faces has length at least 3 (see Figure 2.1). It is known that any undirected simple planar graph G with n vertices has at most $3n-6$ edges (a simple planar digraph can have twice as much since uv and vu are not parallel for digraphs). A folklore proof of this fact (see e.g. [21]) uses Euler's formula and easily extends to semi-strict plane graphs. Therefore, a semi-strict plane graph with n vertices has at most $3n-6$ edges as well. Consequently, for simple planar and semi-strict plane graphs G we have $|E(G)| = O(|V(G)|)$.

Duality. The *dual graph* of a connected plane graph G, denoted by G^*, is a plane graph whose set of vertices is the set of faces of G. Moreover, for each edge $uv = e \in E(G)$, G^* contains a *dual edge* e^* between the faces on the two sides of e. We assume $\mathrm{id}(e) = \mathrm{id}(e^*)$. When G is directed, e^* is directed from the face left of e (looking from u in the direction of v) to the face

right of e.

For $E_1 \subseteq E(G)$, we denote by E_1^* the set $\{e^* : e \in E_1\}$.

We often use the fact that removing an edge in the primal graph corresponds to contracting its dual edge in the dual graph. Formally:

Fact 2.3.1. *Let G be a connected plane graph and $e \in E(G)$. Then $(G - e)^* = G^*/e^*$.*

There is a well-known correspondence between the minimal cutsets of a plane graph and simple cycles in its dual graph.

Fact 2.3.2. *Let G be an undirected connected plane graph and let $C \subseteq E(G)$. C is a minimal cutset of G if and only if C^* is a simple cycle in G^*.*

By Fact 2.3.2 it follows that $e \in E(G)$ (where G is plane and undirected) is a bridge if and only if e^* is a self-loop of G^*. Similarly, a pair of edges $\{e_1, e_2\} \subseteq E(G)$ is a 2-cutset if and only if e_1^* and e_2^* are parallel in G^*.

Embedding planar graphs. Every time we develop an algorithm working on a plane graph G, we only assume that we can access the edge rings and bounding cycles of all the vertices and faces, respectively, of both G and the dual graph G^*. Fortunately, the known linear-time planarity testing algorithms, e.g., [9, 17, 51], all output such edge rings and bounding cycles as a byproduct.

We never access the actual embedding of G (i.e., the points and curves) when implementing the algorithms, but use it only for analysis.

Plane subgraphs and boundary vertices. If H is a subgraph of a plane graph G, we assume that H is also plane and it inherits the embedding of G.

We define the set of *boundary vertices* ∂H of H as $\partial H = V(H) \cap V(G - H)$.

A face of H that is not a face of G is called a *hole* of H. A *simple hole* is a hole of H that is a simple face of H.

Lemma 2.3.3. *Let H be a connected subgraph of a connected plane graph G. Then for any $v \in \partial H$, v lies on some hole of H.*

Proof. Recall that $G - H$ is edge-induced, so from $v \in \partial H = V(H) \cap V(G - H)$ it follows that there exists an edge $e = vw$ such that $e \in E(G)$ and $e \notin E(H)$. Since H and $H + e$ are both subgraphs of G, there exists a face f of H which contains the embedding of e and $\{v, w\} \subseteq V(f)$. But no face of G contains the embedding of any edge of G so f is not a face of G, hence it is a hole of H. □

Plane graph separators and divisions. Let $G = (V, E)$ and let $x : V \to \mathbb{R}_{\geq 0}$ be a vertex weight function. For $X \subseteq V$ we define $x(X) := \sum_{v \in X} x(v)$. Let c be a positive real number less than 1. A set $S \subseteq V$ is called a *c-balanced separator* of G with respect to x if $V \setminus S$ can be split into disjoint parts A, B such that are no edges in G with one endpoint in A and the other in B, and $\max(x(A), x(B)) \leq c \cdot x(V)$.

Lemma 2.3.4 ([72])**.** *Let G be a triangulated connected plane graph with n vertices and let $x : V(G) \to \mathbb{R}_{\geq 0}$. In linear time one can compute a simple cycle $C \subseteq G$ of length at most $2\sqrt{2n}$ (also called a* simple cycle separator*) such that $V(C)$ is a $\frac{2}{3}$-balanced separator of G with respect to x.*

Let G be a simple plane graph with n vertices. For any $r \in [1, n]$, an *r-division* $\mathcal{R}$ of G is a collection of $O(n/r)$ edge-induced subgraphs of G, called *pieces*, whose union is G and such that each piece P has $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices. We denote by $\partial\mathcal{R}$ the set $\bigcup_{P\in\mathcal{R}} \partial P$, also called the *boundary* of r-division $\mathcal{R}$. Clearly, $|\partial\mathcal{R}| = O(n/\sqrt{r})$.

If the pieces of an r-division $\mathcal{R}$ are edge-disjoint, we call it an *r-partition*.

Theorem 2.3.5 ([29, 42])**.** *Let G be a simple plane graph and let $n = |V(G)|$. For any $r \in [1, n]$, an r-partition of G can be computed in $O(n)$ time.*

Suppose now that G is plane embedded. An *r-division with few holes* is an r-division with an additional property that each piece is connected, has $O(1)$ holes, and for each hole h of P, $V(h) \subseteq \partial P$.

Theorem 2.3.6 ([64])**.** *Let G be a simple triangulated connected plane graph with n vertices. For any $r \in [1, n]$, an r-division with few holes of G can be computed in $O(n)$ time.*[1]

2.4 Data-Structural Toolbox

Priority queues. We assume that priority queues store elements with real keys. A priority queue H supports the following set of operations.

- Insert(e, k) – insert an element e with key k into H.
- Extract-Min() – delete an element $e \in H$ with the smallest key and return e.
- Decrease-Key(e, k) – given an element $e \in H$, decrease the key of e to k. If the current key of e is smaller than k, do nothing.
- Min-Key() – return the smallest key in H.

Formally, we assume that each call Insert(e, k) also produces a "handle" which can be later used to point the call Decrease-Key to a place inside H where e is being kept. In our applications, the elements stored in a priority queue are always distinct and thus for brevity we skip the details of using handles later on.

Fredman and Tarjan [30] showed a data structure called *the Fibonacci heap*, which can perform Extract-Min in amortized $O(\log n)$ time and all the remaining operations in amortized $O(1)$ time. Here n is the current size of the queue. In the following sections, we assume that each priority queue is implemented as a Fibonacci heap.

Predecessor searching. Let S be some fixed totally ordered set such that for any $s \in S$ we can compute the *rank* of s, i.e., the number $|\{y \le s : y \in S\}|$, in constant time.

A *dynamic predecessor/successor data structure* maintains a subset R of S and supports the following operations.

- Insertion of some $s \in S$ into R.
- Deletion of some $s \in R$.

[1] Klein et al. [64] do not explicitly state that for each piece P of the r-division produced by their algorithm, each hole of P consists solely of vertices of ∂P. However, as noted by Nussbaum [79], this is indeed the case since in their r-division, each face of G is a face of exactly one piece and thus each edge of a hole of a piece is also an edge of some other piece.

- $\textsc{Pred}(s)$ ($\textsc{Succ}(s)$) – for some $s \in S$, return the largest (smallest respectively) element r of R such that $r \leq s$ ($r \geq s$ respectively).

Van Emde Boas [93] showed that using $O(|S|)$ space we can perform each of these operations in $O(\log\log|S|)$ time. Therefore, whenever we use a dynamic predecessor/successor data structure in the following sections, we can assume that all the above operations can be performed in $O(\log\log|S|)$ time.

Balanced binary search trees. We often use balanced binary search trees (balanced BSTs; e.g., *splay trees* [88]) to represent dynamic sets R over various totally ordered universes S whose elements can be compared in constant time. We assume that if $R \subseteq S$ is stored in a balanced binary search tree, then we can add elements to R, remove elements from R and perform predecessor/successor searches on R in $O(\log|R|)$ time. A balanced binary search tree storing R uses only $O(|R|)$ space. It also allows to iterate through all elements of R in $O(|R|)$ time.

Hence, in comparison to a dynamic predecessor data structure, a balanced binary search is more space-efficient and does not require computation of ranks in S at a cost of exponentially slower time costs of individual operations.

Chapter 3

Maintaining a Planar Graph Under Contractions

In this chapter we describe an optimal data structure that can efficiently maintain a planar graph subject to edge contractions. It can report groups of parallel edges and self-loops that emerge in an online fashion. It also supports constant-time adjacency queries and maintains the neighbor lists and degrees explicitly.

Outline. This chapter is organized as follows. In Section 3.1 we define precisely what kind of information our data structure maintains and specify the set of operations that our data structure supports. Then, in Section 3.2 we present some applications of our data structure: we obtain linear time algorithms for a few problems related to connectivity, matchings, and colorings. In Section 3.3 we give a detailed implementation and performance analysis of our data structure.

3.1 The Data Structure Interface

In this section we specify the set of operations that our data structure supports so that it fits our applications. It proves beneficial to look at the graph undergoing contractions from two perspectives.

1. The *adjacency viewpoint* allows us to track the neighbor sets of the individual vertices as if G was simple at all times.

2. The *edge status viewpoint* allows us to track, for all the original edges E_0, whether they have become self-loops or parallel edges, and also track how E_0 is partitioned into classes of pairwise-parallel edges.

Let $G_0 = (V_0, E_0)$ be an undirected planar graph used to initialize the data structure. Recall that any contraction alters both the set of vertices and the set of edges of the graph. Throughout, we let $G = (V, E)$ denote the *current* version of the graph, unless otherwise stated.

Each edge $e \in E$ can be either a self-loop, an edge parallel to some other edge $e' \neq e$ (we call such an edge *parallel*), or an edge that is not parallel to any other edge of G (we call it *simple* in this case). An edge $e \in E$ that is simple might either get contracted, or might change into a parallel edge as a result of contracting other edges. Similarly, a parallel edge might either get contracted or might change into a self-loop. Note that during contractions, neither can a parallel edge ever become simple, nor can a self-loop become parallel.

Observe that parallelism is an equivalence relation on the edges of G. Once two edges e_1, e_2 connecting vertices $u, v \in V$ become parallel, they remain parallel until some edge e_3 (possibly equal to e_1 or e_2) parallel to both of them gets contracted. However, groups of parallel edges might merge (Figure 3.1), and this might also be a valuable piece of information.

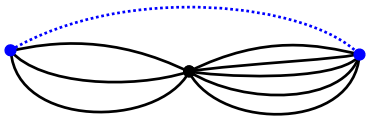

Figure 3.1: Contracting the blue dotted edge will merge two groups of parallel edges.

To succinctly describe how the groups of parallel edges change, we report parallelism in a directed manner as follows. Each group $Y \subseteq E$ of parallel edges in G is assumed to have its *representative* edge $\alpha(Y)$. For $e \in Y$ we define $\alpha(e) = \alpha(Y)$. When two groups of parallel edges $Y_1, Y_2 \subseteq E$ merge as a result of a contraction, the data structure chooses $\alpha(Y_i)$ for some $i \in \{1, 2\}$ to be the new representative of the group $Y_1 \cup Y_2$, and reports an ordered pair $\alpha(Y_{3-i}) \to \alpha(Y_i)$. We call each such pair a *directed parallelism*. After such an event, $\alpha(Y_{3-i})$ will not be reported as a part of a directed parallelism anymore. The choice of i can also be made according to some fixed strategy, e.g., if the edges are assigned weights $w(\cdot)$, then we may choose $\alpha(Y_i)$ so that $w(\alpha(Y_i)) \leq w(\alpha(Y_{3-i}))$. This is convenient in what Klein and Mozes [63] call *strict optimization problems*, such as the minimum spanning tree problem, where we can discard one of any two parallel edges based only on these edges.

Note that at any point of time the set of directed parallelisms reported so far can be seen as a forest of rooted trees $\mathcal{T}$ over E, such that each tree T of $\mathcal{T}$ represents a group Y of parallel edges of G. The root of T is equal to $\alpha(Y)$.

When some edge is contracted, all edges parallel to it are reported as self-loops. Clearly, each edge e is reported as a self-loop at most once. Moreover, it is reported as a part of a directed parallelism $e \to e'$, where $e' \neq e$, at most once.

Recall that edge contraction preserves edge identifiers (see Section 2.1) of all edges except the contracted edge. Hence, the set $\mathrm{id}(E)$ can be easily seen to be equal to the set $\mathrm{id}(E_0)$ with the contracted edges' identifiers removed. Technically speaking, in the implementation we refer to the edges of E using their identifiers. However, the endpoints of the edge with a given identifier generally change in time. Our data structure also provides access to the "current" endpoints of each edge $e \in E$.

Since the vertices V can actually be viewed as sets forming a partition of V_0, we also need some way of labeling V. For $v \in V$, denote by v' the label of v. Let V' be the sets of labels such that there is a 1-1 correspondence between V and V'. Initially, $V = \{\{v\} : v \in V_0\}$ and $\{v\}' = v$, so $V' \subseteq V_0$. When an edge uv is contracted, the vertices $u, v \in V$ are merged. Then, the labels u' and v' are invalidated, and the data structure computes a new label $s = (u \cup v)'$. The set V' is replaced by $V' \setminus \{u', v'\} \cup \{s\}$.

We are now ready to define the complete interface of our data structure.

- $\textsc{Init}(G_0 = (V_0, E_0), w)$ – initialize the data structure and report all the initial self-loops and directed parellelisms. Here, w is an optional weight function.
- $(s, P, L) := \textsc{Contract}(e)$, for $e \in E$ – contract the edge e. Let $e = uv$. The call $\textsc{Contract}(e)$ returns a new label s corresponding to the vertex resulting from merging u and v, and two lists P and L of new directed parallelisms and self-loops, respectively, reported as a result of contraction of e.
- $\textsc{Vertices}(e)$, for $e \in E$ – return $\{u', v'\} \subseteq V'$ such that $e = uv$.

- NEIGHBORS(u'), for $u' \in V'$ – return a pointer to the list $\{(v', \alpha(uv)) : v \in N_G(u)\}$.
- DEG(u'), for $u' \in V$ – find the number of neighbors of u in G.
- EDGE(u', v'), for $u', v' \in V$ – if $uv \in E$, then return $\alpha(uv)$. Otherwise, return **nil**.

The following theorem summarizes the performance of our data structure.

Theorem 3.1.1. *Let $G = (V, E)$ be an undirected planar graph with $|V| = n$ and $|E| = m$. There exists a deterministic data structure supporting* EDGE, VERTICES, NEIGHBORS *and* DEG *in $O(1)$ worst-case time, and whose initialization and any sequence of* CONTRACT *operations take $O(n + m)$ total time. The data structure supports iterating through the neighbor list of a vertex with $O(1)$ overhead per element.*

3.2 Applications

3.2.1 Optimal Decremental 2-Edge-Connectivity and Unique Matchings

In the *decremental 2-edge connectivity* problem, the goal is to design a data structure that supports queries about the existence of two edge-disjoint paths between a pair of given vertices subject to edge deletions. Giammarresi and Italiano [40] showed a data structure for this problem with $O(\log n)$ amortized update time and $O(1)$ query time.

Theorem 3.2.1. *Let $G = (V, E)$ be a simple planar graph and let $n = |V|$. There exists a deterministic data structure that maintains G subject to edge deletions and can answer 2-edge connectivity queries in $O(1)$ time. Its total update time is $O(n)$.*

Proof. Denote by G_0 the initial graph. Suppose without loss of generality that G_0 is connected. Let $B(H)$ denote the set of all bridges of a graph H. Note that two vertices u, v are in the same 2-edge-connected component of G if and only if they are in the same connected component of the graph $(V, E \setminus B(G))$.

Observe that if e is a bridge of G, then deleting e from G does not influence the 2-edge-components of G. Hence, when a bridge e is deleted, we may ignore this deletion. We denote by G' be the graph obtained from G_0 by the same sequence of deletions as G, but ignoring the bridge deletions. This way, G' is connected at all times and the 2-edge-connected components of G' and G are the same. It is also easy to see that $E(G) \setminus B(G) = E(G') \setminus B(G')$ and $B(G) = B(G') \cap E(G)$. Moreover, the set $E(G')$ shrinks in time, whereas $B(G')$ only grows.

First we show how the set $B(G')$ is maintained. Recall that $e \in E(G')$ is a bridge of G' if and only if e^* is a self-loop of $(G')^*$. We build the data structure of Theorem 3.1.1 for $(G')^*$, which initially equals G_0^*. As deleting a non-bridge edge e of G' translates to a contraction of a non-loop edge e^* in $(G')^*$, we can maintain $B(G')$ in $O(n)$ total time by detecting self-loops in $(G')^*$.

Denote by H the graph $(V, E(G') \setminus B(G'))$. To support 2-edge connectivity queries, we maintain the graph H with the decremental connectivity data structure of Łącki and Sankowski [68]. This data structure maintains a planar graph subject to edge deletions in linear total time and supports connectivity queries in $O(1)$ time. When an edge e is deleted from G, we first check whether it is a bridge and if so, we do nothing. If e is not a bridge, the set $E(G')$ shrinks and thus we remove the edge e from H. The deletion of e might cause the set $B(G')$ to grow. Any new edge of $B(G')$ is also removed from H afterwards.

To finish the proof, note that each 2-edge connectivity query on G translates to a single connectivity query in H. All the maintained data structures have $O(n)$ total update time. □

As an almost immediate consequence of Theorem 3.2.1, we improve upon [31] and obtain an optimal algorithm for the *unique perfect matching* problem when restricted to planar graphs.

Corollary 3.2.2. *Given a simple planar graph $G = (V, E)$ with $n = |V|$, in $O(n)$ time we can find a unique perfect matching of G or detect that the number of perfect matchings in G is not* 1*.*

Proof sketch. The algorithm by Gabow et al. [31] for this problem runs in $O(n \log n)$ time. The algorithm has the following two bottlenecks and otherwise runs in linear time.

1. Maintaining the set of bridges of G under edge deletions.
2. Maintaining the sizes of connected components of G under edge deletions. Specifically, one has to be able to query the size of a component containing given $v \in V$ in $O(1)$ time.

We can remove the former bottleneck by proceeding as in the proof of Theorem 3.2.1. The latter bottleneck can be dealt with by extending the data structure for decremental connectivity in planar graphs due to Łącki and Sankowski [68]. This data structure computes an r-partition $\mathcal{R}$ of the input graph, and based on it defines a skeleton graph. Roughly speaking, the skeleton graph is defined as follows. We say that a connected component is interesting if it contains a boundary vertex. Thus, each connected component is either interesting or fully contained within one piece of the r-division (in which case it is handled recursively).

The skeleton graph represents all interesting connected components of the graph. It has vertices of two types, namely it contains all boundary vertices of the r-division and, for each interesting component C and each piece containing vertices of C, one auxiliary vertex representing the intersection of C and the piece. Such an auxiliary vertex may correspond to multiple vertices in the entire graph.

The skeleton graph has $O(n/\sqrt{r})$ vertices, and for each vertex the data structure explicitly maintains the identifier of its connected component. In order to extend the data structure to maintain the sizes of the components, it suffices to maintain, for each auxiliary vertex, the number of vertices in the entire graph, that it corresponds to. From the algorithm, it follows that this information can be updated without impacting the overall running time. □

3.2.2 Maximal 3-Edge-Connected Subgraphs

Lemma 3.2.3. *Let $k \geq 2$. Suppose there exists a data structure D_k maintaining a planar graph H under edge contractions and reporting edges of H participating in some cycle of length i, where $2 \leq i \leq k$, in an online manner. Denote by $O(f_k(n, m))$ the total time needed by D_k to execute any sequence of contractions on a graph with n vertices and m edges.*

Then, there exists an algorithm computing the maximal $(k+1)$-edge-connected subgraphs of a planar graph G in $O(f_k(n, m) + n + m)$ time.

Proof. Assume without loss of generality that G is connected – otherwise we could handle the connected components of G separately. Recall that each simple cycle of length i, $1 \leq i \leq k$, in G^* corresponds to a minimal i-cutset in G.

First, analogously as in Theorem 3.2.1, we build for G a data structure D_1 maintaining the set of bridges of G under deletions of non-bridge edges. Recall that this data structure needs $O(n + m)$ total time to operate. We also build a data structure D_k for G^*.

Clearly, if G does not have cutsets of size at most k, it is $(k+1)$-edge connected. Suppose some edge e participates in some cutset of size i, $2 \leq i \leq k$, in G. Then e participates in some cutset of size no more than i in every subgraph of G containing e. As a result, any subgraph of G containing e is not $(k+1)$-edge-connected. We may thus safely discard e: the maximal $(k+1)$-edge-connected subgraphs of G and $G - e$ are the same.

The above observation leads to a simple algorithm for computing maximal $(k+1)$-edge-connected subgraphs. We maintain a queue Q of non-bridge edges e that participate in some cutset of G of size no more than k. As long as $Q \neq \emptyset$, we extract some e of Q and remove it from G. We issue the removal of e to D_1. Some new edges might be reported as bridges – these are removed from Q. Recall that removal of a non-bridge edge e in G corresponds to contraction of e^* in G^*. We thus issue a contraction to D_k, after which new non-bridge edges can be marked as participating in cycles of length at least 2 and at most k of G^*. These newly marked edges in G^* are added to Q.

Once the algorithm finishes, let G' be the remaining graph G. The edges of G' can be partitioned into two sets: the set B of bridges and the set X of edges whose duals do not participate in any cycle of length at most k in G'^*. We show that the connected components of (V, X), i.e., the 2-edge-connected components of G', are $(k+1)$-edge-connected. Take some such component H and suppose it is not $(k+1)$-edge-connected. Then it has a cutset C of size at most k. Moreover, $|C| > 1$ and C is not a cutset of G'. Hence, there exist some two vertices $x, y \in V(H)$ such that there is a simple path P from x to y in G' and $P \not\subseteq H$. We conclude that P has to contain some edge from B. However, no simple path between two vertices of the same 2-edge-connected component of G' can contain a bridge, a contradiction. □

Lemma 3.2.4. *The maximal 3-edge-connected subgraphs of a planar graph can be computed in linear time.*

Proof. Recall that, by Theorem 3.1.1, we can maintain any planar graph H under edge contractions in linear total time so that the edges participating in 2-cycles, i.e., parallel edges, are reported in an online fashion. To finish the proof, we apply Lemma 3.2.3. □

3.2.3 Simple Linear-Time Algorithms

Finally, we present two examples showing that Theorem 3.1.1 might be a useful black-box in designing linear time algorithms for planar graphs.

Example 3.2.5. *Every planar graph G can be 5-colored in linear time.*

Proof. A textbook proof of the 5-color theorem proceeds by induction as follows (see Figure 3.2). By Euler's formula, each planar graph has a vertex u with at most 5 different neighbors. The case when u has less than 5 neighbors is easy: let $uv = e \in E(G)$, for any $v \in N_G(u)$. We can color G/e inductively, uncontract the edge e, and finally recolor u with a color not used among the vertices $N_G(u)$. When, however, u has exactly 5 neighbors, there exist two neighbors of x, y of u such that x and y are not adjacent as otherwise G would contain K_5 and therefore G would not be planar. We could thus obtain a planar graph G' by contracting both some $e_x = ux$ and some $e_y = uy$. After inductively coloring G' and "uncontracting" e_x and e_y, we obtain a coloring of G that is valid, except that x, y, and u have the same colors assigned. Thus, at most 4 colors are used among the neighbors of u and we recolor u to the remaining color in order to get a valid coloring of G.

Note that this proof can be almost literally converted into a linear time 5-coloring algorithm (see Algorithm 1 for pseudocode) using the data structure of Theorem 3.1.1 built for G. We only need to maintain a subset Q of vertices of G with at most 5 neighbors. The subset Q can be easily maintained in linear total time since all vertices that potentially change their neighbor sets after the call `contract`(e) are endpoints of the reported parallel edges. □

Example 3.2.6. *A minimum spanning tree of a planar graph G can be computed in linear time.*

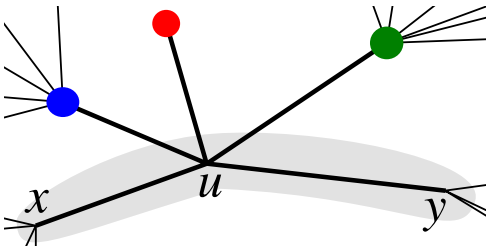

Figure 3.2: The degree ≤ 5 vertex and its two independent neighbors may be colored using the remaining two colors.

Algorithm 1 A linear-time 5-coloring algorithm for planar graphs.

```
1:  procedure Contract-And-Update-Queue(e)
2:      {u', v'} := Vertices(e)
3:      Q := Q \ {u', v'}
4:      (s, P, L) := Contract(e)
5:      for w ∈ {s} ∪ ⋃{Vertices(e1) : (e1 → e2) ∈ P} do
6:          if w ∉ Q and Deg(w) ≤ 5 then
7:              Q := Q ∪ {w}
8:      return s
9:
10: procedure Color()
11:     if Q = ∅ then
12:         return
13:     u := any element of Q
14:     Q := Q \ {u}
15:     Z := Neighbors(u)
16:     if Deg(u) ≥ 1 and Deg(u) ≤ 4 then
17:         (v, e) := any element of Z
18:         s := Contract-And-Update-Queue(e)
19:         Color()
20:         C[v] := C[s]
21:     else if Deg(u) = 5 then
22:         (x, e_x), (y, e_y) := any two elements of Z such that Edge(x, y) = nil
23:         Contract-And-Update-Queue(e_x)
24:         s := Contract-And-Update-Queue(e_y)
25:         Color()
26:         C[x] := C[s]
27:         C[y] := C[s]
28:     C[u] := any color of ({1, 2, 3, 4, 5} \ {C[w] : (w, ·) ∈ Z})
29:
30: function 5-Color(G)
31:     Init(G)
32:     Q := {v' ∈ V' : Deg(v') ≤ 5}
33:     C ← an array indexed with vertices, with values from the set {1, 2, 3, 4, 5}
34:     Color()
35:     return C
```

Algorithm 2 A linear-time minimum spanning tree algorithm for planar graphs.

```
1: function MST(G)
2:     Init(G, w_G).                    ▷ Each time e' → e is reported, we have w_G(e') ≥ w_G(e).
3:     Q := {v' ∈ V' : Deg(v') ≤ 5}
4:     T := ∅
5:     while Q ≠ ∅ do
6:         u := any element of Q
7:         Q := Q \ {u}
8:         if Deg(u) ≥ 1 then
9:             e := an edge such that (v, e) ∈ Neighbors(u) and w_G(e) is minimal
10:            T := T ∪ {e}
11:            Contract-And-Update-Queue(e)                    ▷ See Algorithm 1.
12:    return T
```

Proof. Observe that by the cut property of a minimum spanning tree (see, e.g., [19]), for any vertex $u \in V(G)$ and an edge e of minimum weight among the edges adjacent to u, there exists a minimum spanning tree T of G such that $e \in T$.

This observation can be turned into an efficient algorithm as follows. Again we build the data structure of Theorem 3.1.1, this time keeping in mind that a representative edge of each set of parallel edges should be an edge of minimum weight. We maintain the subset $Q \subseteq V(G)$ containing the vertices with no more than 5 neighbors. We repeatedly pick any $u \in Q$, find the minimum cost edge $e = uv$ adjacent to u (in $O(1)$ time), include e in the constructed minimum spanning tree, and subsequently contract e. We stop when G consists of a single vertex.

The set S can be updated after a contraction analogously as in Example 3.2.5 (see Algorithm 2). By Theorem 3.1.1, the total running time of this algorithm is linear. □

3.3 Maintaining a Planar Graph Under Contractions

In this section we prove Theorem 3.1.1. We first assume all edges to have equal weights. Supporting weights will be discussed later on.

We start by reducing our problem to the case when the graph G is initially simple and of bounded degree. We build a graph H from G by introducing, for each edge $uv = e \in E$, an auxiliary vertex w_e and an edge $e' = w_e v$, and changing the endpoint v of e to w_e. Observe that after this transformation H is simple and planar, and thus has $O(n + m)$ vertices and edges.

Let us now compute the embedding of H in linear time [9, 17, 51]. We build a graph H' from H by replacing each vertex v of H of degree at least 4 with a path P_v of length $\deg(v) - 1$ consisting of newly introduced vertices and edges, and replacing the endpoint v of the i-th edge e_i in the edge ring of v in H with the i-th vertex of the path P_v. After this step, all the vertices of H' have degree no more than 3. Moreover, H' is still planar, simple, and has $O(n+m)$ edges and vertices. Let us call auxiliary the edges of the form $e' = w_e v$ and those constituting the paths P_v.

We now build a data structure for maintaining H' under contractions. We first perform Contract(e') on all the introduced auxiliary edges. Note that after these contractions the graph H in fact becomes equal to G (up to relabeling the vertices). Moreover, no auxiliary edge is reported either as a self-loop or as a part of a directed parallelism. Only the original edges of G are possibly reported as self-loops or parallel edges.

By the described reduction, in the remaining part of this section we can assume that G is initially simple and of bounded degree. We denote by n the number of vertices of this modified

graph G, and since G then has $O(n)$ edges, we can forget about the parameter m.

3.3.1 A Vertex Merging Data Structure

We first consider a more general problem, which we call the *bordered vertex merging* problem. The data structure presented below will constitute a basic building block of the multi-level data structure. Let us now describe the data structure for the bordered vertex merging problem in detail.

Suppose we have a dynamic *simple* planar graph $G = (V, E)$ and a *border set* $B \subseteq V$. Assume G is initially equal to $G_0 = (V_0, E_0)$ and no edge of E_0 connects two vertices of B. The data structure supports the following update operations.

- Merge (or in other words, an identification) of two vertices $u, v \in V$ ($u \neq v$) such that the resulting graph is still planar. If at least one of u, v is in B, the resulting vertex s is also in B after the merge, i.e., $B := B \setminus \{u, v\} \cup \{s\}$. Otherwise, if $\{u, v\} \cap B = \emptyset$, then $s \notin B$ and B does not change either.
- Insertion of an edge $e = uv$ (where $uv \notin E$ is not required) such that $G + uv$ is planar.

Let $\tilde{E}$ be the set of edges inserted using an update operation of the latter type. After each update operation the data structure performs a "cleanup" on G so that the following invariants are satisfied:

1. G is planar and simple.
2. No edge of E has both its endpoints in B.

To satisfy the first invariant, the data structure reports and removes the parallel edges and self-loops that emerge. Once reported, each set of parallel edges is merged into one representative edge. The reporting of parallel edges is done in the form of directed parallelisms, as described in Section 3.1. Again, it is easy to see that each edge of $E_0 \cup \tilde{E}$ is reported as the first coordinate of a directed parallelism at most once.

Observe that some modifications might also break the second invariant: both an edge insertion and a merge might introduce an edge e with both endpoints in B. We call such an edge a *border edge*. Each border edge e that is not a self-loop is reported and deleted from (or not inserted to) G. Note that an edge e may be first reported parallel (in a directed parallelism of the form $e' \to e$, where $e' \neq e$) and then reported border.

Clearly, merging vertices alters the set V by replacing two vertices u, v with a single vertex s. It is useful to additionally require that in fact:

- $s \in \{u, v\}$, i.e., *we always merge one vertex into the other*. This way, we have $V \subseteq V_0$ at all times.
- If $u \in B$ and $v \notin B$, then $s = u$. In other words, we always merge a non-border vertex into a border vertex.

As the merges proceed, each vertex of G corresponds to a set of vertices of the initial graph G_0. Let $\phi : V_0 \to V_0$ be a mapping such that for $a \in V_0$ $\phi(a)$ is a vertex of the *current* vertex set "containing" a. In this section we formally define V as $\phi(V_0)$. Let $\phi^{-1} : V \to 2^{V_0}$ be a reverse mapping such that $\phi^{-1}(v) = \{v_0 \in V_0 : \phi(v_0) = v\}$.

At any time, the edges of E constitute a subset of $E_0 \cup \tilde{E}$ in the following sense: for each $e = xy \in E$ there exists an edge $e' = uv \in E_0 \cup \tilde{E}$ such that $\mathrm{id}(e) = \mathrm{id}(e')$ and vertices u and v have been merged into x and y, respectively. Note that by our assumption that we merge one

vertex into another and $V \subseteq V_0$, we can also equivalently say that $\phi(u) = x$ and $\phi(v) = y$, even though when e' was inserted we already could have $|\phi^{-1}(u)| > 1$ or $|\phi^{-1}(v)| > 1$.

Apart from reporting self-loops, parallel edges, and border edges, the data structure is required to explicitly maintain the mappings ϕ and ϕ^{-1}, and also support constant-time queries regarding the representative edge connecting any two vertices u, v of the current vertex set V (if such an edge exists).

Graph representation. The data structure for the bordered vertex merging problem internally maintains G using the data structure of the following lemma for planar graphs.

Lemma 3.3.1 ([11])**.** *There exists a deterministic, linear-space data structure maintaining a dynamic simple planar graph H with n vertices so that:*

- *adjacency queries in H can be performed in $O(1)$ worst-case time,*
- *edge insertions and deletions can be performed in $O(\log n)$ amortized time.*

The data structure can be initialized in $O(n)$ time.

Fact 3.3.2. *The data structure of Lemma 3.3.1 can be easily extended so that:*

- *for each $v \in V$, a doubly-linked list storing $N_H(v)$ is maintained within the same bounds,*
- *for each pair (x, y) of vertices adjacent in H, some auxiliary data associated with (x, y) can be accessed and updated in $O(1)$ worst-case time.*

In addition to the data structure of Lemma 3.3.1 representing G, for each unordered pair x, y of vertices adjacent in G, we maintain an edge $\alpha(x, y) = e$ where e is the unique edge in E connecting x and y. Recall that in fact $\alpha(x, y)$ corresponds to some of the original edges of E_0 or one of the inserted edges $\tilde{E}$. By Fact 3.3.2, we can access $\alpha(x, y)$ in constant time.

The mapping ϕ is stored in an array, whereas the sets $\phi^{-1}(\cdot)$ – in doubly-linked lists.

Suppose a merge of u and v, $u, v \in V$, is issued and we decide to merge u into v. In terms of the operations supported by the data structure of Lemma 3.3.1, we need to remove each edge ux and insert an edge vx unless v has been adjacent to x before.

More precisely, to update our representation, we only need to perform the following steps:

- For each $v_0 \in \phi^{-1}(u)$, set $\phi(v_0) = v$ and add v_0 to $\phi^{-1}(v)$.
- Compute the list $N_u = \{(x, \alpha(u, x)) : x \in N_G(u)\}$. Remove all edges adjacent to u from G. For each $(x, \alpha(u, x)) \in N_u$, $x \neq v$, check whether $x \in N_G(v)$ (this can be done in $O(1)$ time, by Lemma 3.3.1). If so, report the parallelism $\alpha(u, x) \to \alpha(v, x)$. Otherwise, if vx is not a border edge, insert an edge vx to G and set $\alpha(v, x) = \alpha(u, x)$. If, on the other hand, $v \in B$ and $x \in B$ (i.e., vx is a border edge), report $\alpha(u, x)$ as a border edge.

Observe that the order of updates issued to the data structure of Lemma 3.3.1 guarantees that the graph this data structure stores is planar at all times. After the update procedure ends, the data structure again represents the graph G.

Recall that if exactly one of the merged vertices (say u) belongs to B, then we are required to merge v into u. However, the decision whether we merge u into v or v into u, when both u and v are both border vertices or both non-border vertices, heavily affects the efficiency of the data structure. Our strategy will be to always pick a vertex (say u) with a smaller set $\phi^{-1}(u)$, and merge u into v.

In order to handle the update operation of the second type, i.e., an insertion of a new edge $e = xy$, we first check whether xy is a border edge. If so, we discard e and report it. Otherwise,

we check whether x and y are adjacent in G. If so, we report the parallelism $e \to \alpha(x, y)$. If not, we add the edge xy to G and set $\alpha(x, y) = e$.

Lemma 3.3.3. *Let G be a graph initially equal to a simple planar graph $G_0 = (V_0, E_0)$ such that $n = |V_0|$. There is a deterministic data structure for the bordered vertex merging problem that processes any sequence of modifications of G_0 in $O((n + f) \log^2 n + m^*)$ total time, where m^* is the total number of edge insertions, and f is the total number of insertions of edges connecting non-adjacent vertices.*

Proof. By Lemma 3.3.1, building the initial representation takes $O(n \log n)$ time as we insert $O(n)$ edges to G. The reporting of parallel edges and border edges takes $O(n + m^*)$ time since each (initial or inserted) edge is reported as a border edge or occurs as the first coordinate of a reported directed parallelism at most once.

Also note that, by Lemma 3.3.1, an insertion of a parallel edge costs $O(1)$ time, for a total of $O(m)$ time over all insertions, as G is not updated in that case. Recall that, by Fact 3.3.2, accessing and updating values $\alpha(x, y)$, for $xy \in E(G)$, takes $O(1)$ time.

The total cost of maintaining the representation of G is $O(g \log n)$, where g is the total number of edge updates to the data structure of Lemma 3.3.1. We now prove that $g = O((n + f) \log n)$. To this end, we look at the merge of u into v from a different perspective: instead of removing an edge $e = ux$ and inserting an edge vx, imagine that we simply change an endpoint u of e to v but the edge itself does not lose its identity. Then, new edges in G are only created either during the initialization or by inserting an edge connecting the vertices that have not been, at that moment, adjacent in G. Hence, there are $O(n + f)$ creations of new edges.

Consider some edge $e = xy$ of G immediately after its creation. Denote by $q(e)$ the pair $(|\phi^{-1}(x)|, |\phi^{-1}(y)|)$. The value of $q(e)$ always changes when some endpoint of e is updated. Suppose a merge of u into v ($u \neq v$) causes the change of some endpoint u of e to v. Either we have $u \notin B$ and $v \in B$, or $|\phi^{-1}(v)| \geq |\phi^{-1}(u)|$ before the merge. The former situation can happen at most once per each endpoint of e since we always merge a non-border vertex into a border vertex if such case arises. In the latter case, on the other hand, one coordinate of $q(e)$ grows at least by a factor of 2, and clearly this can happen at most $O(\log n)$ times as the size of any $\phi^{-1}(x)$ is never more than n. Since there are $O(n + f)$ "created" edges, and each such edge undergoes $O(\log n)$ endpoint updates, indeed we have $g = O((n + f) \log n)$.

A very similar argument can be used to show that the total time needed to maintain the mapping ϕ along with the reverse mapping ϕ^{-1} is $O(n \log n)$. □

A micro variant. In order to obtain an optimal data structure, we need a specialized version of the bordered vertex merging data structure that handles small graphs in linear total time.

Suppose we restrict our problem even more by disallowing inserting new edges into G. Additionally, assume we are allowed to perform some preprocessing in time $O(n)$. Then, due to a monotonous nature of allowed operations on G, when the size of G_0 is very small compared to n, we can maintain G faster than by using the data structure of Lemma 3.3.3.

Lemma 3.3.4. *After preprocessing in $O(n)$ time, we can repeatedly solve the bordered vertex merging problem without edge insertions for simple planar graphs G_0 with $t = O(\log^4 \log^4 n)$ vertices in $O(t)$ time.*

Proof. Let $f(n) = c \log^4 \log^4 n$ for some $c > 0$. We use the preprocessing time to simulate every possible sequence of updates on every possible graph $G_0 = (V_0, E_0)$ with no more than $f(n)$ vertices and each possible $B \subseteq V_0$. The simulation allows us to precompute, for each step, the list of self-loops and directed parallelisms to be reported.

We identify the vertices V_0 with the set $\{1, \ldots, t\}$ and assume that edges of E_0 are assigned identifiers from the set $\{1, \ldots, |E_0|\}$ so that $e = uv \in E_0$ is assigned an identifier equal to the position of the pair (u, v) in the sorted list $\{(u, v) : u < v, uv \in E_0\}$.

Any possible graph G_0 can be encoded with $t^2 = O(f(n)^2)$ bits representing the adjacency matrix of G_0. For a given G_0 with t vertices, each possible $B \subseteq V_0$ can be easily encoded with additional $t = f(n)$ bits. On a graph G initially equal to G_0, at most t merges can be performed. Clearly, a single operation on G can be encoded as a pair of affected vertices, i.e., $O(\log t)$ bits. Each possible (not necessarily maximal) sequence S of modifications of G can be thus encoded with additional $O(t \log t) = O(f(n)^2)$ bits. We conclude that each triple (G_0, B, S) can be encoded with $O(f(n)^2)$ bits and thus there are no more than $O(2^{\text{poly}(f(n))})$ such triples.

For each triple $\psi = (G_0, B, S)$, we do the following:

- We compute its bit encoding $z(\psi)$.
- We use the data structure D of Lemma 3.3.3 to simulate the sequence of updates S on a graph G initially equal to G_0 and a border set B.
- Afterwards, a record $Q[z(\psi)]$ is filled with the following information:
  - mappings ϕ and ϕ^{-1} computed by D,
  - the lists of border edges and directed parallelisms that were reported after the last modification of the sequence S.
  - the bit encodings $z(\psi')$ of all the triples $\psi' = (G_0, B, S')$ such that S' extends S by a single update.

For each triple $\psi = (G_0, B, S)$, all the needed information can be clearly computed in time polynomial in $f(n)$. Hence, in total we need $O(2^{\text{poly}(f(n))})$ time to compute all the necessary information. As $O(\text{poly}(f(n))) = o(\log n)$, any bit encoding $z(\psi)$ is an integer of order $O(n)$.

Now, to handle any sequence of modifications on a graph G_0 with at most $f(n)$ vertices and a border set $B \subseteq V_0$, we first compute in linear time the bit encoding $z(\psi^*)$ of $\psi^* = (G_0, B, S)$, where initially $S = \emptyset$. Each modification Y is executed as follows: we use the information in $Q[z(\psi^*)]$ to find the bit encoding $z(\psi')$ of the configuration $\psi' = (G_0, B, S \cup \{Y\})$, and we move from the configuration ψ^* to ψ'. Next, we read from $Q[z(\psi')]$ which edges should be reported as parallel edges or border edges. As we only move between the configurations by updating the bit encoding of the current configuration and possibly report edges, the whole sequence of updates takes time linear in the size of G_0. Clearly, the record $Q[z(\psi^*)]$ can be used to access the mappings ϕ and ϕ^{-1} in constant time. □

3.3.2 A Multi-Level Data Structure

Recall that our goal is to maintain a planar graph G under contractions, where G is simple and has constant degree. Below we describe in detail how to take advantage of graph partitioning and bordered vertex merging data structures to obtain a linear total time solution.

We build an r-partition $\mathcal{R} = \{P_1, P_2, \ldots, \}$ of G with $r = \log^4 n$, where $n = |V_0|$. Then, for each piece $P_i \in \mathcal{R}$, we build an r-partition $\mathcal{R}_i = \{P_{i,1}, P_{i,2}, \ldots\}$ of P_i with $r = \log^4 \log^4 n$. By Theorem 2.3.5, building all the necessary pieces takes $O(n)$ time in total. Since G_0 is of constant degree, any vertex $v \in V_0$ is contained in $O(1)$ pieces of $\mathcal{R}$. Similarly, for any $v \in P_i$, v is contained in $O(1)$ pieces of $\mathcal{R}_i$.

As G undergoes contractions, we can view its vertex set as a partition of V_0. Initially, $V = \{\{v\} : v \in V_0\}$, and when an edge uv is contracted ($u, v \in V$), the resulting vertex is $u \cup v$. Denote by $\phi : V_0 \to V$ the unique mapping such that for each $v \in V_0$, $v \in \phi(v)$. Of course,

initially $\phi(v) = \{v\}$ for each $v \in V_0$. We will never store the mapping ϕ explicitly; we use it only for the data structure's description and analysis.

Let $\overline{G} = (V, \overline{E})$ denote the maximal simple subgraph of G, i.e., the graph G with self-loops discarded and each group Y of parallel edges replaced with a single edge $\alpha(Y)$. The key component of our data structure is a 3-level set of (possibly micro-) bordered vertex merging data structures $\Pi = \{\pi\} \cup \{\pi_i : P_i \in \mathcal{R}\} \cup \{\pi_{i,j} : P_i \in \mathcal{R}, P_{i,j} \in \mathcal{R}_i\}$. The data structures Π form a tree such that π is the root, $\{\pi_i : P_i \in \mathcal{R}\}$ are the children of π, and $\{\pi_{i,j} : P_{i,j} \in \mathcal{R}_i\}$ are the children of π_i. For a data structure $\mathcal{D} \in \Pi$, let $par(\mathcal{D})$ be the parent of $\mathcal{D}$ and let $A(\mathcal{D})$ be the set of ancestors of $\mathcal{D}$. We call the value $\ell(\mathcal{D}) = |A(\mathcal{D})|$ the *level* of $\mathcal{D}$. The data structures of levels 0 and 1 are stored as data structures of Lemma 3.3.3, whereas the data structures of level 2 are stored as micro structures of Lemma 3.3.4.

Each data structure $\mathcal{D} \in \Pi$ has a set $V_{\mathcal{D}} \subseteq V_0$ of *interesting vertices*, defined as follows: $V_\pi = \partial\mathcal{R}$, $V_{\pi_i} = \partial P_i \cup \partial\mathcal{R}_i$, and $V_{\pi_{i,j}} = V(P_{i,j})$. The data structure $\mathcal{D}$ maintains a certain subgraph $G_{\mathcal{D}}$ of $\overline{G}$ defined inductively as follows (recall that we defined $G_1 \setminus G_2$ to be a graph containing all vertices of G_1 and edges of G_1 that do not belong to G_2):

$$G_{\mathcal{D}} = \overline{G}[\phi(V_{\mathcal{D}})] \setminus \left(\bigcup_{\mathcal{D}' \in A(\mathcal{D})} G_{\mathcal{D}'} \right).$$

Fact 3.3.5. *For any $\mathcal{D} \in \Pi$, $G_{\mathcal{D}}$ is a minor of G_0.*

Fact 3.3.6. *For any $uv = e \in \overline{E}$, there exists $\mathcal{D} \in \Pi$ such that $e \in E(G_{\mathcal{D}})$.*

Proof. Let u_0, v_0 be the initial endpoints of e. Initially $e \in P_{i,j}$ for some i, j. Observe that since $\{\phi(u_0), \phi(v_0)\} \subseteq V(G_{\pi_{i,j}})$, e is contained in $G_{\pi_{i,j}}$ or some of its ancestors. □

For each $\mathcal{D} \in \Pi$, we define the set of *ancestor vertices* $AV_{\mathcal{D}} = V_{\mathcal{D}} \cap \left(\bigcup_{\mathcal{D}' \in A(\mathcal{D})} V_{\mathcal{D}'} \right)$. Now we discuss more precisely what it means for the bordered vertex merging data structure $\mathcal{D}$ to *maintain* the graph $G_{\mathcal{D}}$. The vertex set used to initialize $\mathcal{D}$ is $V_{\mathcal{D}}$. Observe that the vertices of $V_{\mathcal{D}}$ come from an n-element set V_0, and thus each of them might need $\Theta(\log |V_0|)$ bits to be represented. In Section 3.3.1, however, we assumed implicitly that we can use linear-space arrays indexed with the vertices of the graph, i.e., that each vertex can be encoded using a number of bits logarithmic in the size of the vertex set. We circumvent this technical problem by introducing, for each $\mathcal{D} \in \Pi$, a separate two-way mapping between $V_{\mathcal{D}}$ and the set $\{1, \ldots, |V_{\mathcal{D}}|\}$, which makes it possible for the bordered vertex merging data structure to internally use the vertices $\{1, \ldots, |V_{\mathcal{D}}|\}$ and allows us to translate these vertices to $V_{\mathcal{D}}$ in constant time. For π, such a mapping can be implemented using an ordinary array, whereas for the data structures π_i and $\pi_{i,j}$, we can use deterministic integer dictionaries of Pătraşcu and Thorup [80] (since these mappings are of $O(r) = O(\text{polylog}\, n)$ size). In the following, we forget about this technical issue, and assume that the initial vertex set of each data structure $\mathcal{D} \in \Pi$ is $V_{\mathcal{D}}$.

Each $\mathcal{D} \in \Pi$ is initialized with the graph $G_{\mathcal{D}}$, so that the vertex $v \in V_{\mathcal{D}}$ in the data structure corresponds to the vertex $\phi(v) = \{v\}$ in $G_{\mathcal{D}}$. We initially set $B_{\mathcal{D}}$, the border set of $\mathcal{D}$, to $AV_{\mathcal{D}}$. We write $\phi_{\mathcal{D}}, \phi_{\mathcal{D}}^{-1}$ to denote the mappings ϕ, ϕ^{-1} maintained by $\mathcal{D} \in \Pi$, respectively. Initially, $\phi_{\mathcal{D}}(v) = v$ and $\phi_{\mathcal{D}}^{-1}(v) = \{v\}$ for each $v \in V_{\mathcal{D}}$.

Throughout a sequence of contractions, we maintain the following invariants for any $\mathcal{D} \in \Pi$:

- There is a 1-1 mapping between the sets $\phi(V_{\mathcal{D}})$ and $\phi_{\mathcal{D}}(V_{\mathcal{D}})$ such that for the corresponding vertices $x \in \phi(V_{\mathcal{D}})$ and $y \in \phi_{\mathcal{D}}(V_{\mathcal{D}})$ we have $\phi_{\mathcal{D}}^{-1}(y) = x \cap V_{\mathcal{D}}$, and if $x \in \phi(AV_{\mathcal{D}})$, then $y \in B_{\mathcal{D}}$. We also say that x is *represented* in $\mathcal{D}$ in this case.

- There is an edge $xy \in E(G_{\mathcal{D}})$ if and only if there is an edge $x_{\mathcal{D}}y_{\mathcal{D}}$ in the graph maintained by $\mathcal{D}$, where $x_{\mathcal{D}}, y_{\mathcal{D}} \in \phi_{\mathcal{D}}(V_{\mathcal{D}})$ are the corresponding vertices of x and y, respectively.

In other words, the graph maintained by $\mathcal{D}$ is isomorphic to $G_{\mathcal{D}}$, but can technically use a different vertex set.

Observe that in $G_{\mathcal{D}}$ there are no edges between the vertices $\phi(AV_{\mathcal{D}})$. The following fact describes how this is reflected in the data structure $\mathcal{D}$.

Fact 3.3.7. *In the graph stored in $\mathcal{D}$, no two vertices of $B_{\mathcal{D}}$ are adjacent.*

Note that since the sets $V_{\mathcal{D}}$ and $V_{\mathcal{D}'}$ might overlap for $\mathcal{D} \neq \mathcal{D}'$, the vertices of V can be represented in multiple data structures.

Now we formulate a few useful properties following from the above invariants.

Lemma 3.3.8. *Suppose for $v \in V$ we have $v \in V(G_{\mathcal{D}_1})$ and $v \in V(G_{\mathcal{D}_2})$. Then, $v \in V(G_{\mathcal{D}})$, where $\mathcal{D}$ is the lowest common ancestor of $\mathcal{D}_1$ and $\mathcal{D}_2$.*

Proof. We first prove that for $i \neq j$, $\phi(V(P_i)) \cap \phi(V(P_j)) \subseteq \phi(\partial\mathcal{R})$. Assume the contrary. Then, there exists such $w \in \phi(V(P_i)) \cap \phi(V(P_j))$ that $w \notin \phi(\partial\mathcal{R})$. But since G undergoes contractions only, for $x \in V$, $G_0[x]$ is a connected subgraph of G_0. Thus, $G_0[w]$ is connected and contains both some vertex of P_i and some vertex of P_j. But each path from $V(P_i)$ to $V(P_j)$ in G_0 has to go through a vertex of $\partial\mathcal{R}$, by the definition of an r-partition. Hence $\partial\mathcal{R} \cap w \neq \emptyset$ and $w \in \phi(\partial\mathcal{R})$, a contradiction.

Analogously one can prove that for any i and $j \neq k$, $\phi(V(P_{i,j})) \cap \phi(V(P_{i,k})) \subseteq \phi(\partial\mathcal{R}_i)$.

Suppose that $v \in V(G_{\mathcal{D}_1}) \cap V(G_{\mathcal{D}_2})$. If for some $i \neq j$ we have $\mathcal{D}_1 \in \{\pi_i\} \cup \bigcup_k \{\pi_{i,k}\}$ and $\mathcal{D}_2 \in \{\pi_j\} \cup \bigcup_k \{\pi_{j,k}\}$ then $v \in \phi(V(P_i)) \cap \phi(V(P_j))$ and we conclude $v \in \phi(\partial\mathcal{R})$. Hence, $v \in V(G_\pi)$. Analogously we prove that if $\mathcal{D}_1 = \pi_{i,j}$ and $\mathcal{D}_2 = \pi_{i,k}$ for some $j \neq k$, then $v \in V(G_{\pi_i})$. □

By Lemma 3.3.8, each vertex $v \in V$ is represented in a unique data structure of minimal level, a lowest common ancestor of all data structures where v is represented. We denote such a data structure by $\mathcal{D}(v)$.

Lemma 3.3.9. *For any $\mathcal{D} \in \Pi$, the vertices $\{v : \mathcal{D}(v) = \mathcal{D}\}$ are represented in $\mathcal{D}$ by the vertices from $\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus B_{\mathcal{D}} \subseteq V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$.*

Proof. Let $v \in V$ be such that $\mathcal{D}(v) = \mathcal{D}$. We have $v \notin V(G_{par(\mathcal{D})})$, so $v \in \phi(V_{\mathcal{D}}) \setminus \phi(AV_{\mathcal{D}})$. Since there is a 1-1 correspondence between $\phi(AV_{\mathcal{D}})$ and $B_{\mathcal{D}}$, v is represented in $\mathcal{D}(v)$ by a vertex of $\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus B_{\mathcal{D}}$.

$\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus B_{\mathcal{D}} \subseteq V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$ follows from the initial condition $B_{\mathcal{D}} = AV_{\mathcal{D}}$ and the fact that bordered vertex merging data structures always merge one vertex into another, and additionally non-border vertices into border vertices. □

Lemma 3.3.10. *Let $uv = e \in \overline{E}$ and $\ell(\mathcal{D}(u)) \geq \ell(\mathcal{D}(v))$. Then, $e \in E(G_{\mathcal{D}(u)})$ and either $\mathcal{D}(u) = \mathcal{D}(v)$ or $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$.*

Proof. If $\{u, v\} \subseteq \phi(\partial\mathcal{R})$, then clearly $\ell(\mathcal{D}(u)) = \ell(\mathcal{D}(v)) = 0$, $e \in E(G_\pi)$, and the lemma holds. Moreover, no $G_{\mathcal{D}}$ such that $\mathcal{D}$ is a descendant of π can contain the edge uv.

Otherwise, u does not belong to $\phi(\partial\mathcal{R})$. Consequently, by Fact 3.3.6 and Lemma 3.3.8, there exists exactly one i such that u is a vertex of some graph $G_{\mathcal{D}}$, and e is an edge of some graph $G_{\mathcal{D}'}$, where $\mathcal{D}, \mathcal{D}'$ are data structures in the subtree of Π rooted of π_i. If $v \notin \phi(\partial\mathcal{R})$, v cannot be a vertex of any $G_{\mathcal{D}''}$, where $\mathcal{D}''$ is in the subtree of π_j, $j \neq i$.

Again, if $\{u, v\} \subseteq \phi(\partial P_i \cup \partial \mathcal{R}_i)$, then $uv \in E(G_{\pi_i})$, $\mathcal{D}(u) = \pi_i$, and no descendant of G_{π_i} can contain uv. If not, we analogously get that there might exist at most one $G_{\pi_{i,j}}$ containing the edge uv and the vertex u. If $v \notin \phi(\partial P_i \cup \partial \mathcal{R}_i)$, then only $G_{\pi_{i,j}}$ can contain the vertex v.

In all cases, $\mathcal{D}(u) = \mathcal{D}(v)$ or $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$. □

Lemma 3.3.11. *Let uv be an edge of some $G_{\mathcal{D}}$, $\mathcal{D} \in \Pi$. If $\{u, v\} \subseteq V(G_{\mathcal{D}'})$, where $\mathcal{D}' \neq \mathcal{D}$, then $\mathcal{D}'$ is a descendant of $\mathcal{D}$ and both u and v are represented as border vertices of $\mathcal{D}'$.*

Proof. Suppose without loss of generality that $\ell(\mathcal{D}(u)) \geq \ell(\mathcal{D}(v))$. By Lemma 3.3.10, we have $\mathcal{D} = \mathcal{D}(u)$.

Let $\{u, v\} \subseteq V(G_{\mathcal{D}'})$. If $\mathcal{D}'$ is a descendant of $\mathcal{D}$, then $\{u, v\} \subseteq \phi(AV_{\mathcal{D}'})$, and by the invariants maintained by our data structure, u and v are represented by the vertices of $B_{\mathcal{D}'}$.

Suppose $\mathcal{D}' \neq \mathcal{D}$ and $\mathcal{D}'$ is not a descendant of $\mathcal{D}$. Then, by Lemma 3.3.8, the lowest common ancestor of $\mathcal{D}'$ and $\mathcal{D}$ contains the vertex u and is an ancestor of $\mathcal{D} = \mathcal{D}(u)$, a contradiction. □

Lemma 3.3.12. *Let $v \in \phi(V_{\mathcal{D}})$, where $\mathcal{D} \in \Pi$. Let $v_{\mathcal{D}}$ be the vertex representing v in $\mathcal{D}$. Then, v is represented in $O(|\phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}})|)$ data structures $\mathcal{D}'$ such that $par(\mathcal{D}') = \mathcal{D}$.*

Proof. Let $\mathcal{D}'$ be a child of $\mathcal{D}$. If v is represented in $\mathcal{D}'$, then $v \in \phi(V_{\mathcal{D}}) \cap \phi(V_{\mathcal{D}'})$. It follows that as $G_0[v]$ is a connected subgraph of G_0, it contains a path between some vertex $x \in V_{\mathcal{D}}$ and some vertex $y \in V_{\mathcal{D}'}$. Assume $x \notin V_{\mathcal{D}'}$. If $\mathcal{D}' = \pi_i$, then in fact we have $x \in V(P_j)$ for $j \neq i$, and any path from x to y has to go through some vertex $z \in \partial P_i$. As $\partial P_i \subseteq V_{\mathcal{D}} \cap V_{\mathcal{D}'}$, there exists a vertex from $V_{\mathcal{D}} \cap V_{\mathcal{D}'}$ in the set v. Similarly, if $\mathcal{D}' = \pi_{i,j}$, there exists a vertex of $\partial P_{i,j}$ in the set v and we again obtain $v \cap V_{\mathcal{D}} \cap V_{\mathcal{D}'} \neq \emptyset$.

Recall that we maintain the invariant $\phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}}) = v \cap V_{\mathcal{D}}$. Therefore, $\phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}}) \cap V_{\mathcal{D}'} \neq \emptyset$. However, for each $w \in \phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}})$, there are only $O(1)$ child data structures $\mathcal{D}'$ such that $w \in V_{\mathcal{D}'}$, by the constant degree assumption. It follows that there are at most $O(|\phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}})|)$ data structures $\mathcal{D}'$ such that $par(\mathcal{D}') = \mathcal{D}$ and $\phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}}) \cap V_{\mathcal{D}'} \neq \emptyset$, which in turn means that there are at most $O(|\phi_{\mathcal{D}}^{-1}(v_{\mathcal{D}})|)$ data structures $\mathcal{D}'$ such that v is also represented in $\mathcal{D}'$. □

Auxiliary components. We need the following auxiliary components for each $\mathcal{D} \in \Pi \setminus \{\pi\}$.

- For each $x \in B_{\mathcal{D}}$, a pointer $\beta_{\mathcal{D}}(x)$ to $y \in \phi_{par(\mathcal{D})}(V_{par(\mathcal{D})})$ such that x and y represent the same vertex of the maintained graph G.
- A dictionary $\gamma_{\mathcal{D}}$ mapping each $x \in \phi_{par(\mathcal{D})}(V_{par(\mathcal{D})})$, such that its corresponding vertex $x' \in V$ is also represented in $\mathcal{D}$, to a vertex $y \in B_{\mathcal{D}}$ corresponding to x' in $\mathcal{D}$. Note that $\gamma_{\mathcal{D}}$ has always size $O(r) = O(\log^4 n)$ and its keys are integers fitting in a single word. Therefore, each $\gamma_{\mathcal{D}}$ can be implemented using a dynamic integer dictionary of Pătraşcu and Thorup [80]. This allows us to perform insertions, deletions, and lookups on $\gamma_{\mathcal{D}}$ deterministically in $O(1)$ worst-case time.

Another component of our data structure is the forest $\mathcal{T}$ of reported parallelisms: for each reported parallelism $e \to \alpha(e)$, we make e a child of $\alpha(e)$ in $\mathcal{T}$. Note that the forest $\mathcal{T}$ allows us to go through all the edges parallel to $\alpha(e)$ in time linear in their number.

We also need some way to index the vertices of V as upon a contraction our data structure should return a label of a new vertex. Namely, a vertex $v \in V$ is assigned the same label as is assigned to its corresponding vertex v' in the data structure $\mathcal{D}(v)$. By Lemma 3.3.9, this label of v comes from the set $\phi_{\mathcal{D}(v)}(V_{\mathcal{D}(v)}) \setminus B_{\mathcal{D}(v)} \subseteq V_{\mathcal{D}(v)} \setminus AV_{\mathcal{D}(v)}$. Since the sets of the form $V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$ are pairwise disjoint for distinct $\mathcal{D}$, such a labeling scheme makes it trivial to find the data structure $\mathcal{D}(v)$ based on the label of v only. We set $V' = \bigcup_{\mathcal{D} \in \Pi}(\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus B_{\mathcal{D}})$.

Lemma 3.3.13. *For $v_0 \in V_0$, we can compute the label of $\phi(v_0)$ and find $\mathcal{D}(\phi(v_0))$ in $O(1)$ time.*

Proof. Let $P_{i,j}$ be any piece such that $v_0 \in V(P_{i,j})$. First, we can compute the representation $x = \phi_{\pi_{i,j}}(v_0)$ of $\phi(v_0)$ in $\pi_{i,j}$ in $O(1)$ time as the data structure $\pi_{i,j}$ stores the mapping $\phi_{\pi_{i,j}}$ explicitly. Set $\mathcal{D} = \pi_{i,j}$.

Next, if $x \in B_{\mathcal{D}}$ (or, technically speaking, if $x \in AV_{\mathcal{D}}$), we follow the pointer $\beta_{\mathcal{D}}(x)$ to the data structure of lower level and repeat if needed until we reach the data structure $\mathcal{D}(\phi(v_0))$. As the tree of data structures has 3 levels, we follow $O(1)$ pointers. □

Lemma 3.3.14. *Let $v \in \phi(V_{\mathcal{D}})$ and suppose we are given $v_{\mathcal{D}}$ – the vertex representing v in $\mathcal{D}$. For any $\mathcal{D}'$, such that $par(\mathcal{D}') = \mathcal{D}$, we can compute the vertex v' representing v in $G_{\mathcal{D}'}$ (or detect that such v' does not exist) in $O(1)$ time.*

Proof. If $\gamma_{\mathcal{D}'}$ contains the key $v_{\mathcal{D}}$, we return $\gamma_{\mathcal{D}'}[v_{\mathcal{D}}]$. Recall that the lookup cost of $\gamma_{\mathcal{D}'}$ is constant. □

Implementing the operations. We now describe how to implement the operation $(s, P, L) :=$ CONTRACT(e), where $uv = e \in E$, $u, v \in V$. Suppose the initial endpoints of e were $u_0, v_0 \in V_0$. First, we iterate through the tree $T_e \in \mathcal{T}$ containing e to find $\alpha(e)$. By Lemma 3.3.13, we can find the vertices u, v, along with the respective data structures $\mathcal{D}(u), \mathcal{D}(v)$, based on u_0, v_0 in $O(1)$ time. Assume without loss of generality that $\ell(\mathcal{D}(u)) \geq \ell(\mathcal{D}(v))$. By Lemma 3.3.10, $\alpha(e)$ is an edge of $G_{\mathcal{D}(u)}$. Although we are asked to contract e, we conceptually contract $\alpha(e)$ by issuing a merge of the vertices corresponding to u and v, respectively, in $\mathcal{D}(u)$. To reflect that we were actually asked to contract e, we include all the edges of $T_e \setminus \{e\}$ in L as self-loops. The merge might make $\mathcal{D}(u)$ report some parallelisms $e_1 \to e_2$. In such a case our data structure reports $e_1 \to e_2$ (by including it in P) and update the forest $\mathcal{T}$.

We also have to reflect the contraction of e in all the required data structures $\mathcal{D} \in \Pi$ and update the auxiliary data structures so that our invariants are again satisfied. If, before the contraction, both u and v were the vertices of some $G_{\mathcal{D}}$, $\mathcal{D} \neq \mathcal{D}(u)$, then by Lemma 3.3.11, $\mathcal{D}$ is a descendant of $\mathcal{D}(u)$. Moreover, observe that if both u and v are vertices of some $G_{\mathcal{D}}$, where $\ell(\mathcal{D}) = 2$ and $\ell(\mathcal{D}(u)) = 0$, then both u and v are also vertices of the graph $G_{par(\mathcal{D})}$. Hence, the representations of u and v have to be merged in $\mathcal{D}(u)$, some children of $\mathcal{D}(u)$, and some children of these children of $\mathcal{D}(u)$. However, by Lemma 3.3.11, the merges to be performed in the descendants of $\mathcal{D}(u)$ are always between border vertices, and thus cannot cause these data structures report new border edges (that is, e.g., those with both endpoints in $B_{\mathcal{D}_i}$ for $\mathcal{D}_i$).

The merge of u and v might also create some new edges $e' = xy$ between the vertices $\phi(AV_{\mathcal{D}(u)})$ in $G_{\mathcal{D}(u)}$. Note that since the vertices $\phi(AV_{\mathcal{D}(u)})$ are represented by $B_{\mathcal{D}(u)}$ in $\mathcal{D}(u)$, in this case $\mathcal{D}(u)$ reports $e'_{\mathcal{D}(u)} = x_{\mathcal{D}(u)}y_{\mathcal{D}(u)}$, where $\mathrm{id}(e'_{\mathcal{D}(u)}) = \mathrm{id}(e')$, as a border edge and also we know that $\ell(\mathcal{D}(x)) < \ell(\mathcal{D}(u))$ and $\ell(\mathcal{D}(y)) < \ell(\mathcal{D}(u))$. Hence, e' should end up in some of the ancestors of $\mathcal{D}(u)$. To reflect this, we insert $e'_{par(\mathcal{D}(u))} = \beta_{\mathcal{D}(u)}(x_{\mathcal{D}(u)})\beta_{\mathcal{D}(v)}(y_{\mathcal{D}(u)})$, where $\mathrm{id}(e'_{par(\mathcal{D}(u))}) = \mathrm{id}(e')$, to $par(\mathcal{D}(u))$ unless it would become a border edge in $par(\mathcal{D}(u))$. In that case an edge connecting the representations of x and y is inserted to the grandparent of $\mathcal{D}$. It is also possible that e' will be reported a parallel edge in some of the ancestors of $\mathcal{D}$: in such a case an appropriate directed parallelism is added to P.

By the above discussion, the edges can only move up the data structure tree, between $\mathcal{D}(u)$ and its ancestors, whereas the merges of the representations of u and v have to be performed in $\mathcal{D}(u)$ and some of its descendants. We now describe how to perform these merges and update the auxiliary data structures.

Let $\mathcal{D}$ be some data structure where the vertices $u_\mathcal{D}$ and $v_\mathcal{D}$, corresponding to u and v, are to be merged. Suppose the merge has already been performed in all the relevant ancestors of $\mathcal{D}$ and the values $\beta_\mathcal{D}(\cdot)$ and $\gamma_\mathcal{D}(\cdot)$ are updated. Specifically, if $u \cup v$ is represented in $par(\mathcal{D})$ by x, then $\beta_\mathcal{D}(u_\mathcal{D}) = x$ if $u_\mathcal{D} \in B_\mathcal{D}$ and $\beta_\mathcal{D}(v_\mathcal{D}) = x$ if $v_\mathcal{D} \in B_\mathcal{D}$. This condition is automatically satisfied for the first data structure to be updated, i.e., $\mathcal{D} = \mathcal{D}(u)$ since, by the fact that uv was located in $G_{\mathcal{D}(u)}$, we had $u_\mathcal{D} \notin B_\mathcal{D}$ before the merge and no merges have been issued (when handling this contraction) to the ancestors of $\mathcal{D}(u)$.

Let us issue the merge to $\mathcal{D}$ and assume $\mathcal{D}$ merges $u_\mathcal{D}$ into $v_\mathcal{D}$. This is without loss of generality since for $\mathcal{D} = \mathcal{D}(u)$, $u_\mathcal{D} \notin B_\mathcal{D}$ and for $\mathcal{D} \neq \mathcal{D}(u)$ we have $\{u_\mathcal{D}, v_\mathcal{D}\} \subseteq B_\mathcal{D}$. If $\beta_\mathcal{D}(v_\mathcal{D}) \neq \textbf{nil}$, we set $\gamma_\mathcal{D}(\beta_\mathcal{D}(v_\mathcal{D})) = v_\mathcal{D}$. By a similar argument as in the proof of Lemma 3.3.3, we can afford to iterate through $\phi_\mathcal{D}^{-1}(u_\mathcal{D})$ without increasing the asymptotic performance of the $u_\mathcal{D}$-into-$v_\mathcal{D}$ merge performed by $\mathcal{D}$ as long as we spend only $O(\log |V_\mathcal{D}|)$ additional time per each element of $\phi_\mathcal{D}^{-1}(u_\mathcal{D})$. By Lemma 3.3.12, there are $O(|\phi_\mathcal{D}^{-1}(u_\mathcal{D})|)$ data structures $\mathcal{D}_1, \mathcal{D}_2, \ldots$ that are the children of $\mathcal{D}$ and contain the representation of u. For each such $\mathcal{D}_i$, we first use the dictionary $\gamma_{\mathcal{D}_i}$ to find the vertex $u_{\mathcal{D}_i}$ representing u in $\mathcal{D}_i$, and update $\beta_{\mathcal{D}_i}(u_{\mathcal{D}_i})$ to $v_\mathcal{D}$. Then, using Lemma 3.3.14, we check whether $v \in V(G_{\mathcal{D}_i})$ in $O(1)$ time. If not, we set $\gamma_{\mathcal{D}_i}(v_\mathcal{D})$ to $u_{\mathcal{D}_i}$. Otherwise, let $v_{\mathcal{D}_i}$ represent v in $\mathcal{D}_i$. At this point, the values $\beta_{\mathcal{D}_i}(\cdot)$ are updated so we can issue a merge of $u_{\mathcal{D}_i}$ and $v_{\mathcal{D}_i}$ to $\mathcal{D}_i$ and handle this merge recursively. The merge may also cause $\mathcal{D}_i$ report some parallelisms. We handle them as described above in the case of the data structure $\mathcal{D}(u)$.

The vertex s to be returned by the call $\textsc{Contract}(e)$ can be computed, after the update procedure finishes, in $O(1)$ time using Lemma 3.3.13 for some original endpoint $u_0 \in V_0$ of e.

Note that all the performed merges and edge insertions are only used to make the graphs represented by the data structures satisfy their definitions. Fact 3.3.5 implies that the represented graphs remain planar at all times.

The key invariants are satisfied by construction and the fact that the bordered vertex merging data structures always merge non-border vertices into border vertices.

We now describe how the other operations are implemented. To compute $u', v' \in V$ such that $\{u', v'\} = \textsc{Vertices}(e)$, where $e \in E$, we use Lemma 3.3.13. u' is the label of $\phi(u_0)$ and v' is the label of $\phi(v_0)$, where u_0, v_0 are the initial endpoints of e. Clearly, this takes $O(1)$ time.

To maintain the values $\textsc{Deg}(v')$, $v' \in V$, before each call $(s, P, L) := \textsc{Contract}(e)$ we compute $\{u', v'\} = \textsc{Vertices}(e)$ and simply set $\textsc{Deg}(s) := \textsc{Deg}(u') + \textsc{Deg}(v') - 1$ after the call. Additionally, for each directed parallelism $e_1 \to e_2$ we decrease $\textsc{Deg}(x)$ and $\textsc{Deg}(y)$ by one, where $\{x, y\} = \textsc{Vertices}(e_1)$.

For each $u' \in V'$, we maintain a doubly-linked list $\mathcal{E}(u') = \{\alpha(uv) : uv \in \overline{E}\}$. Additionally, for each $e \in \overline{E}$ we store the pointers to the two occurrences of e in the lists $\mathcal{E}(\cdot)$. Again, after a call $(s, P, L) := \textsc{Contract}(e)$, where $e = uv$ and $\{u', v'\} = \textsc{Vertices}(e)$, we set $\mathcal{E}(s)$ to be a concatenation of the lists $\mathcal{E}(u')$ and $\mathcal{E}(v')$. Finally, we remove all the occurrences of edges $\{\alpha(e)\} \cup \{e_1 : (e_1 \to e_2) \in P\}$ from the lists $\mathcal{E}(\cdot)$. Now, the implementation of the pointer to $\textsc{Neighbors}(u')$ is easy as the endpoints of the edges in $\mathcal{E}(u')$ not equal to u (obtained using the operation $\textsc{Vertices}$) form exactly the labels of the set $N_G(u)$. We have proved the following.

Lemma 3.3.15. *The operations* $\textsc{Vertices}$, $\textsc{Deg}$ *and* $\textsc{Neighbors}$ *run in* $O(1)$ *worst-case time.*

We now show how to implement the operation $\textsc{Edge}(u', v')$ in $O(1)$ time. Recall that $\mathcal{D}(u)$ and $\mathcal{D}(v)$, along with the representations of u and v in these respective data structures, can be computed in constant time based on the labels u', v' only. By Lemma 3.3.10, the edge $\alpha(uv)$ can be contained in either $\mathcal{D}(u)$ or $\mathcal{D}(v)$, whichever has greater level. Suppose without loss of generality that $\ell(\mathcal{D}(u)) \geq \ell(\mathcal{D}(v))$. Again, by Lemma 3.3.10, $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$. Recall that Lemma 3.3.14 allows us to compute the representation of a vertex in a child data

structure $\mathcal{D}'$ in $O(1)$ time. Thus, we can find v in $\mathcal{D}(u)$ by applying Lemma 3.3.14 at most twice. Finally, given representations of u and v in $\mathcal{D}(u)$, the edge $\alpha(uv)$ can be found in $O(1)$ time by issuing a query to the bordered vertex merging data structure $\mathcal{D}(u)$.

Lemma 3.3.16. *The operation* Edge *runs in $O(1)$ worst-case time.*

The following lemma summarizes the total time spent on updating all the vertex merging data structures Π and is proved in Section 3.3.3.

Lemma 3.3.17. *The cost of all operations on the data structures $\mathcal{D} \in \Pi$ is $O(n)$.*

Proof of Theorem 3.1.1. Recall that we first reduced the general case to the case when G is simple and bounded-degree, and has $O(n+m)$ vertices and edges.

To initialize our data structure, we initialize all the data structures $\mathcal{D} \in \Pi$ and the auxiliary components. This takes $O(n+m)$ time. The time needed to perform any sequence of operations Contract is proportional to the total time used by the data structures Π as the cost of maintaining the auxiliary components can be charged to the operations performed by the individual structures of Π. By Lemma 3.3.17, this time is linear.

By combining the above with Lemmas 3.3.15 and 3.3.16, the theorem follows. □

3.3.3 Running Time Analysis

Lemma 3.3.18. *Let $\mathcal{D} \in \Pi$. After the initialization of $\mathcal{D}$, only the edges initially contained in the graphs represented by the descendants of $\mathcal{D}$ might be inserted into $\mathcal{D}$, each at most once.*

Proof. Note that whenever we report a border edge in $\mathcal{D}$, we insert it to $par(\mathcal{D})$. □

To bound the operating time of our data structure, we need to analyze, for any $\mathcal{D} \in \Pi$ and any sequence of edge contractions, the number of changes to $E(G_{\mathcal{D}})$ that result in a costly operation of inserting an edge connecting non-adjacent vertices into the underlying bordered vertex merging data structure $\mathcal{D}$.

Consider some sequence S of k edge contractions on G. We abuse notation a bit and identify each $v_0 \in V_0$ with $\{v_0\}$. Let $G_i = (V_i, E_i)$ (for $i = 0, 1, \ldots, k$) be the graph G after i contractions. Denote by $e_i = u_i v_i$, where $u_i, v_i \in V_{i-1}$, the edge involved in the i-th contraction. We have $V_i = V_{i-1} \setminus \{u_i, v_i\} \cup \{u_i \cup v_i\}$. Moreover, let $\phi_i : V_0 \to V_i$ be the mapping ϕ after i contractions of S. Denote by $\overline{G_i}$ the graph $\overline{G}$ (recall that $\overline{G}$ is a "simple" version of G) after i contractions.

Let $W \subseteq V_0$. For $i > 0$, let $\Delta_i^W \subseteq E(\overline{G_i}[\phi_i(W)])$ be the set of edges that we would need to add to $\overline{G_{i-1}}[\phi_{i-1}(W)]$, after possibly merging u_i and v_i in $G_{i-1}[\phi_{i-1}(W)]$, in order to obtain $\overline{G_i}[\phi_i(W)]$.

We now define Δ_i^W more precisely. For any $E' \subseteq E_{i-1}$ define $r_i(E')$ to be the set obtained by taking all edges $e \in E'$ and adding it to $r_i(E')$, but with all endpoints of e equal to u_i or v_i replaced with $u_i \cup v_i$. Using this notation, we formally have

$$\Delta_i^W = E\left(\overline{G_i}[\phi_i(W)]\right) \setminus r_i\left(E\left(\overline{G_{i-1}}[\phi_{i-1}(W)]\right)\right).$$

Clearly, for any $x, y \in \phi_i(W)$, there is at most one edge xy in Δ_i^W. Let

$$\Psi(W) = \sum_{i=1}^{k} |\Delta_i^W|.$$

Observe that we perform exactly $\Psi(V_\pi) = \Psi(\partial\mathcal{R})$ costly insertions on the data structure π. Similarly, for each π_j, we perform no more than $\Psi(V_{\pi_j})$ costly insertions since some of the edges

of $\Delta_i^{V_{\pi_j}}$ are actually not inserted as they constitute border edges in π_j. Generally speaking, for any $\mathcal{D} \in \Pi$ we perform at most $\Psi(V_{\mathcal{D}})$ costly insertions on $\mathcal{D}$.

Lemma 3.3.19. *For any $W \subseteq V_0$, $\Psi(W) = O(|W|)$.*

Proof. The idea behind the proof is to analyze how many "costly" insertions would we perform if we maintained semi-strict graphs instead of simple graphs in our data structures. It turns out that this quantity is easier to track and additionally constitutes an upper bound on the quantity we actually care about.

Fix some plane embedding of G_0. We define semi-strict versions $G_0^W, G_1^W, \ldots, G_k^W$ of graphs $G_0[\phi_0(W)], G_1[\phi_1(W)], \ldots, G_k[\phi_k(W)]$, respectively, so that:

- $G_0^W = G_0[W]$. Recall that G_0 is simple, and thus its subgraph $G_0[W]$ is also simple and in particular semi-strict.
- If $\{u_i, v_i\} \cap \phi_{i-1}(W) = \emptyset$, then $G_i^W = G_{i-1}^W$.
- If $\{u_i, v_i\} \subseteq \phi_{i-1}(W)$, then we obtain G_i^W from G_{i-1}^W by first contracting $u_i v_i$. The vertices u_i, v_i are merged into $u_i \cup v_i$. For any triangular face $f = u_i v_i x_i$ of G_{i-1}^W (there can be between 0 and 2 such faces since G_{i-1}^W is semi-strict), the contraction introduces a face $f' = (u_i \cup v_i)x_i$ of length 2. We remove one of these edges $(u_i \cup v_i)x_i$ from G_i^W so that the face f' is merged with any neighboring face and G_i^W becomes semi-strict.
- If $|\{u_i, v_i\} \cap \phi_{i-1}(W)| = 1$, suppose without loss of generality that $u_i \in \phi_{i-1}(W)$ and $v_i \notin \phi_{i-1}(W)$ (the case when $v_i \in \phi_{i-1}(W)$ is symmetrical). Pick a maximal pairwise non-parallel subset F_i of such edges $v_i b_i$ of G_{i-1} that $b_i \in \phi_{i-1}(W)$ and $u_i b_i \notin E(G_{i-1}^W)$. Let G_i^W be obtained from the following subgraph of G_{i-1}:

  $$X_i = (\phi_{i-1}(W) \cup \{v_i\}, E(G_{i-1}^W) \cup \{e_i\} \cup F_i)$$

  by contraction of $u_i v_i$. Observe that, by definition of X_i, the contraction of $u_i v_i$ in X_i does not introduce new parallel edges and, as a result, G_i^W is semi-strict.

The graphs G_i^W are defined in such a way that $E(G_i^W) \subseteq E_i$ and for any $x, y \in V_i$, $xy \in E(\overline{G_i}[\phi_i(W)])$ if and only if $xy \in E(G_i^W)$.

Let $E'_{i-1} = r_i(E(G_{i-1}^W))$. Take some x, y such that $xy \in \Delta_i^W$. Note that since $\Delta_i^W = E\left(\overline{G_i}[\phi_i(W)]\right) \setminus r_i\left(E\left(\overline{G_{i-1}}[\phi_{i-1}(W)]\right)\right)$, we have $xy \in E(G_i^W) \setminus E'_{i-1}$. Since there are no parallel edges in Δ_i^W, we obtain

$$|\Delta_i^W| \leq |E(G_i^W) \setminus E'_{i-1}|.$$

Hence, in order to prove $\Psi(W) = O(|W|)$, it is thus sufficient to show

$$|E(G_1^W) \setminus E'_0| + |E(G_2^W) \setminus E'_1| + \ldots + |E(G_k^W) \setminus E'_{k-1}| = O(|W|).$$

As each G_i^W is semi-strict, $|E(G_i^W)| \leq 3|V(G_i^W)| = 3|\phi_i(W)| \leq 3|W|$. Moreover, as any contraction in a semi-strict graph decreases the number of edges by at most 3, we have $|E'_{i-1} \setminus E(G_i^W)| \leq 3$. In fact, by the definition of G_i^W, we can have $E'_{i-1} \not\subseteq E(G_i^W)$ only if $\{u_i, v_i\} \subseteq \phi_{i-1}(W)$, i.e., when $|V(G_i^W)| < |V(G_{i-1}^W)|$. This may happen for at most $|W|$ values of i as $|V(G_0^W)| = W$. Denote the set of these values i as I.

We have

$$\begin{aligned}
\Psi(W) &\leq \sum_{i=1}^{k} |E(G_i^W) \setminus E'_{i-1}| \\
&= \sum_{i=1}^{k} |E(G_i^W) \setminus (E(G_i^W) \cap E'_{i-1})| \\
&= \sum_{i=1}^{k} |E(G_i^W)| - |E(G_i^W) \cap E'_{i-1}| \\
&\leq \sum_{i=1}^{k} |E(G_i^W)| - \sum_{i \in \{1,\ldots,k\} \setminus I} |E'_{i-1}| - \sum_{i \in I} (|E'_{i-1}| - 3) \\
&= \sum_{i=1}^{k} |E(G_i^W)| - \sum_{i=1}^{k} |E'_{i-1}| + 3|I| \\
&= \sum_{i=1}^{k} |E(G_i^W)| - \sum_{i=1}^{k} |E(G_{i-1}^W)| + 3|I| \\
&= |E(G_k^W)| - |E(G_0^W)| + 3|W| \\
&\leq 6|W| \\
&= O(|W|).
\end{aligned}$$

□

Proof of Lemma 3.3.17. Recall that by Lemma 3.3.3, the cost of any sequence of operations on $\mathcal{D} \in \{\pi\} \cup \{\pi_i : P_i \in \mathcal{R}\}$ is $O((|V_\mathcal{D}| + f_\mathcal{D}) \log^2 |V_\mathcal{D}| + m_\mathcal{D})$ where $m_\mathcal{D}$ is the total number of times an edge is inserted into $\mathcal{D}$, and $f_\mathcal{D}$ is the number of "costly" insertions connecting non-adjacent vertices. By Lemma 3.3.18, $m_\pi = O(|E_0|)$ and $m_{\pi_i} = O(|E(P_i)|)$. By Lemma 3.3.19, $f_\mathcal{D} = \Psi(V_\mathcal{D}) = O(|V_\mathcal{D}|)$. We have $|V_\pi| = O(n/\log^2 n)$, and thus the total time used by π is $O(n)$. Similarly, we have $|V_{\pi_i}| = O(\log^4 n/\log^2 \log^4 n)$, and the total time used by $O(n/\log^4 n)$ data structures π_i is $O(n/\log^2 \log^4 n + \sum_i |E(P_i)|) = O(n)$.

By Lemma 3.3.4, after $O(n)$ preprocessing, the total time used by each $\pi_{i,j}$ is $O(|V(P_{i,j})|)$, and thus, summed over all i, j, we again obtain $O(n)$ time. □

3.3.4 Supporting Edge Weights

In this section we show how to modify the data structure of Section 3.3 by adding additional layer so that, given a weight function $w : E_0 \to \mathbb{R}$, for each reported directed parallelism $\alpha(Y_{3-i}) \to \alpha(Y_i)$ (see Section 3.1), we have $w(\alpha(Y_i)) \leq w(\alpha(Y_{3-i}))$.

We maintain an array δ defined as follows. Let Y be a group of parallel edges represented by a tree $T \in \mathcal{T}$. Then, $\delta[\alpha(Y)]$ is equal to an edge $e \in T$ such that $w(e)$ is minimum. Initially, for each $e \in E_0$ we have $\delta(e) = e$. To maintain the invariant posed on δ throughout any sequence of contractions, we do the following.

Suppose the data structure of Section 3.3 reports a parallelism $\alpha(Y_1) \to \alpha(Y_2)$. Then, if $\delta[\alpha(Y_1)] \leq \delta[\alpha(Y_2)]$, the weight supporting layer reports $\delta[\alpha(Y_2)] \to \delta[\alpha(Y_1)]$ instead and sets $\delta[\alpha(Y_2)] = \delta[\alpha(Y_1)]$. On the other hand, when $\delta[\alpha(Y_1)] > \delta[\alpha(Y_2)]$, we only report $\delta[\alpha(Y_1)] \to \delta[\alpha(Y_2)]$ instead.

The layer also modifies the behavior of operations Edge and Neighbors: whenever they return some edge $\alpha(uv)$, the user is given the edge $\delta[\alpha(uv)]$ instead.

Chapter 4

Decremental Reachability in Planar Digraphs

In this chapter we study the problem of maintaining the reachability information for a given planar directed graph G when G is subject to edge deletions.

We assume that our input graph G is initially simple (and hence $|E(G)| = O(|V(G)|)$). This is without much loss of generality since removing self-loops of parallel edges does not influence, for any $u, v \in V(G)$, whether a path $u \to v$ exists in G. However, by applying various reductions and using auxiliary graphs, we sometimes internally use non-simple graphs for convenience.

Outline. All algorithms in this chapter build upon a recursive decomposition of a graph G. We define the decomposition that we need along with all its structural and quantitative properties in Section 4.1. That section also describes in detail how to compute such a decomposition.

Next, in Section 4.2 we describe the structural properties of reachability in plane graphs that we later exploit algorithmically.

Section 4.3 contains the main technical part of this chapter. In that section we solve an abstract problem of maintaining the transitive closure of a union of two graphs of the form $H^+[U]$ where U, roughly speaking, lies on a small number of faces of H, when those two graphs undergo partially dynamic updates. We give both incremental and decremental transitive closure algorithms as they both prove useful later on, even though our input graph G undergoes edge deletions only.

In Section 4.4 we combine the recursive decomposition with the decremental transitive closure algorithm of Section 4.3 and obtain a randomized decremental transitive closure algorithm with $\tilde{O}(n)$ total update time and $\tilde{O}(\sqrt{n})$ query time (here, n is the number of vertices of G).

Next, in Section 4.5 we show a simple extension of the dynamic algorithm of Section 4.4 that allows us to decrementally maintain certain useful information about individual edges of G in nearly-linear total time. This information can used in an almost straightforward way to solve other decremental reachability-related problems on a planar digraph such as single-source reachability, maintaining the strongly-connected components, or maintaining the transitive reduction (the last one only if G happens to be acyclic).

Section 4.6 shows how plane duality can be used to obtain some of the extensions of Section 4.5 more efficiently and without resorting to randomization. This is where we take advantage of the incremental transitive closure algorithm developed in Section 4.3, which is deterministic and more efficient than its decremental counterpart.

Finally, in Section 4.7 we show a generalization of the decremental transitive closure algorithm from Section 4.4. Namely, we show a trade-off data structure that, given any parameter $t \in [1, n]$, supports edge deletions in $\tilde{O}(n/t)$ amortized time and queries in $\tilde{O}(\sqrt{t})$ time.

4.1 Simple Recursive Decomposition.

Let G be a plane graph and let $n = |V(G)|$. A *simple recursive decomposition* of G is a collection of *edge-induced, connected subgraphs* of G (called *pieces*) organized in a binary tree $\mathcal{T}_G$. We write $H \in \mathcal{T}_G$ to denote that H, which is a subgraph of G, is a piece, or in other words, a node of $\mathcal{T}_G$.

The root of $\mathcal{T}_G$ is the graph G itself. Each non-leaf piece $H \in \mathcal{T}_G$ has exactly two children $\text{child}_1(H), \text{child}_2(H)$, such that $H = \text{child}_1(H) \cup \text{child}_2(H)$ and $E(\text{child}_1(H)) \cap E(\text{child}_2(H)) = \emptyset$.

Let $H \in \mathcal{T}_G$ be any piece. We define the *level* $\ell(H)$ of a piece H to be the number of edges on the path from the root to H. We define the *height* $\chi(H)$ of a piece H to be the maximum number of edges on the path from H to a leaf piece. We also denote by $\mathcal{T}_G(H)$ the subtree of $\mathcal{T}_G$ rooted at the node H.

The *separator* $\text{Sep}(H)$ of a piece is defined as $\emptyset$ if H is a leaf of $\mathcal{T}_G$ and

$$\text{Sep}(H) = V(\text{child}_1(H)) \cap V(\text{child}_2(H))$$

otherwise.

The tree $\mathcal{T}_G$ has the following additional properties:

(1) The height of the decomposition $\mathcal{T}_G$ is $O(\log n)$.

(2) There are $O(n)$ leaf subgraphs in $\mathcal{T}_G$ and each leaf subgraph has $O(1)$ vertices.

(3) There exists a constant $\rho > 1$ such that for any $H \in \mathcal{T}_G$, $|\partial H| = O\left(\sqrt{n/\rho^{\ell(H)}}\right)$.

(4) $\sum_{H \in \mathcal{T}_G} |\partial H|^2 = O(n \log n)$.

(5) For any $H \in \mathcal{T}_G$, H has $O(1)$ holes, all of them *simple* and *pairwise vertex-disjoint.*

(6) For any $H \in \mathcal{T}_G$, each hole of H consists solely of the vertices of ∂H.

Before showing how to compute a simple recursive decomposition $\mathcal{T}_G$, let us make some simple but useful observations about $\mathcal{T}_G$.

Fact 4.1.1. *Every $e \in E(G)$ is contained in a unique leaf subgraph of $\mathcal{T}_G$.*

Proof. Observe that the edge set of each non-leaf piece is partitioned among its two children. □

Lemma 4.1.2. *Let $H \in \mathcal{T}_G$ be a non-root piece. Let P be its parent and let S be its sibling. Then,*

$$\partial H = \text{Sep}(P) \cup (V(H) \cap \partial P).$$

Proof. We have:

$$\begin{aligned}
\partial H &= V(H) \cap V(G - H) \\
&= V(H) \cap V(S \cup (G - P)) \\
&= (V(H) \cap V(S)) \cup (V(H) \cap V(G - P)) \\
&= \text{Sep}(P) \cup (V(H) \cap V(P) \cap V(G - P)) \\
&= \text{Sep}(P) \cup (V(H) \cap \partial P).
\end{aligned}$$

□

Lemma 4.1.3. *Let $\mathcal{T}_G$ be a simple recursive decomposition and let $H \in \mathcal{T}_G$. Then, the vertices of the holes of H are precisely the boundary vertices of H.*

Proof. Combine Lemma 2.3.3 and property (6) of a simple recursive decomposition. □

Unfortunately, not all plane graphs admit simple decompositions. However, we show that one can "extend" a plane graph so that constructing a simple decomposition is possible. The remainder of this section is devoted to proving the following theorem.

Theorem 4.1.4. *Let $G = (V, E)$ be a simple, connected and triangulated plane graph with n vertices. In $O(n \log n)$ time one can construct a (not necessarily simple) plane graph $G' = (V', E_0 \cup E' \cup E^{\times})$, along with its simple recursive decomposition $\mathcal{T}_{G'}$ such that:*

- *V' is a disjoint union of sets $S(v)$, $v \in V$, where $v \in S(v)$.*
- *The sets E_0, E' and $E^{\times}$ are pairwise disjoint.*
- *The connected components of (V', E_0) are exactly the sets $S(v)$.*
- *The edges of each $G'[S(v)]$ can be oriented in such a way that $G'[S(v)]$ becomes strongly-connected.*
- *If $uv = e \in E$, then there exists an edge $u'v' = e' \in E'$ such that $\mathrm{id}(e) = \mathrm{id}(e')$, $u' \in S(u)$ and $v' \in S(v)$.*
- *$|V'| = O(n)$ and $|E_0 \cup E' \cup E^{\times}| = O(n)$.*

Our algorithm resembles [8, 27, 64] (in that the graph G is recursively separated using Miller's cycle separator (Lemma 2.3.4) until we reach $O(1)$-size pieces) and builds upon these algorithms, but is more involved due to the stronger properties that we pursue (in [8, 27, 64], the holes of individual pieces are not required to be simple and pairwise vertex-disjoint).

To compute a simple decomposition, we proceed as follows. Our recursive decomposition procedure maintains the following invariants about the input piece H of each its recursive call:

1. H is a connected simple plane graph.
2. H has $O(1)$ holes.
3. The holes of H are simple and pairwise disjoint.
4. Each hole of H consists solely of vertices of ∂H.

Moreover, as we will gradually change G (i.e., the root piece of the decomposition tree) during the decomposition process, we maintain the invariant that immediately before each recursive call the root of the decomposition G has only faces of size 2 or 3.

These invariants are clearly satisfied when the recursive procedure is first called on G as G is simple, connected, triangulated, $\partial G = \emptyset$, and G has no holes at all.

Let H be the graph to be decomposed and suppose $|V(H)| > n_0$, where n_0 is a sufficiently large integer, to be set later, so that various inequalities that we will need hold all at once. Let H^{Δ} be a graph obtained from H by putting a vertex v_h inside each hole h of H and connecting it (by edges embedded inside h) with all vertices of h. Denote by A^{Δ} the set of auxiliary "hole" vertices of H^{Δ}. Clearly, $V(H^{\Delta}) = V(H) \cup A^{\Delta}$ and $H \subseteq H^{\Delta}$. As H^{Δ} is simple and triangulated, we can use Lemma 2.3.4 to compute a simple cycle $C(H)$ that is additionally a balanced separator of H^{Δ} with respect to some weight function $x : V(H^{\Delta}) \to \mathbb{R}_{\geq 0}$. Throughout, we sometimes use $C(H)$ to refer to the underlying curve (or, in other words, the embedding) of the cycle $C(H)$ of H^{Δ}. Note that since $C(H)$ is a simple cycle in H^{Δ}, the curve $C(H)$ can go through each hole h of H at most once (by passing through $v_h \in A^{\Delta}$ at most once).

First, suppose we simply set $\mathrm{child}_1(H)$ ($\mathrm{child}_2(H)$) to be the part of H weakly inside (weakly outside, respectively) the curve $C(H)$. Consider $\mathrm{child}_1(H)$ and let us construct a cycle $C_1(H)$ by replacing each pair of consecutive edges uv_h, v_hw of $C(H)$ such that $v_h \in A^\Delta$, with the unique subpath of the bounding cycle of h that goes from u to w weakly inside (weakly outside, respectively) the curve $C(H)$. It might happen that $C_i(H) = C(H)$ if $C(H)$ does not go through any vertices of A^Δ.

Observe that $C_1(H)$ is actually a face of $\mathrm{child}_1(H)$ and there is a 1-1 correspondence between the vertices $A^\Delta \setminus V(C(H))$ inside $C(H)$ (or equivalently, inside $C_1(H)$) and the remaining holes of $\mathrm{child}_1(H)$ (excluding $C_1(H)$ if it happens to be a hole). Hence, if H has ℓ holes, $\mathrm{child}_1(H)$ has at most $\ell + 1$ holes.

Note that in fact $\mathrm{child}_1(H)$ is equal to the part of H weakly inside $C_1(H)$ and $C_1(H) \subseteq \mathrm{child}_1(H)$. Let $x, y \in V(\mathrm{child}_1(H))$. Since H is connected, there is a path P from x to y in H. For each maximal subpath Q of P that consists of edges not in $\mathrm{child}_1(H)$, the endpoints a, b of Q are vertices of $C_1(H)$ as $V(\mathrm{child}_1(H)) \cap V(\mathrm{child}_2(H)) \subseteq V(C_1(H))$. Hence, we can replace Q with a path from a to b that uses the edges of $C_1(H)$ only. This way we conclude that there also exists a path from x to y in $\mathrm{child}_1(H)$, or equivalently, $\mathrm{child}_1(H)$ is connected.

Analogously we argue that $\mathrm{child}_2(H)$ is connected and has at most $\ell + 1$ holes.

Unfortunately, if we use the curve $C(H)$ to separate H as described above, the holes of $\mathrm{child}_i(H)$ might still be not necessarily pairwise disjoint or non-simple.

We say that a vertex $v \in \partial H \cap V(C(H))$ lying on a hole h (recall h is uniquely defined, by the invariants that H satisfies and Lemma 4.1.3) of H is *bad*, if:

1. either $v_h \notin C(H)$,
2. or u, v_h, w are consecutive vertices on $C(H)$ and $v \notin \{u, w\}$.

The first type of a bad vertex leads to non-disjoint holes in either $\mathrm{child}_1(H)$ or $\mathrm{child}_2(H)$, whereas the second leads to a non-simple hole in some child of H. Observe that if there are no bad vertices, the face $C_i(H)$ of $\mathrm{child}_i(H)$ (defined as before) is disjoint with other holes of $\mathrm{child}_i(H)$, and $C_i(H)$ is simple. All remaining holes of $\mathrm{child}_i(H)$ are simple and pairwise disjoint by the fact that they are holes of H as well. Consequently, our strategy will be to transform H slightly so that bad vertices are eliminated.

While there exists a bad vertex $v \in V(h)$ of H, we perform the following operation. Let $e_1, \ldots, e_q, e_{q+1} = e_1$ be the clockwise edge ring of v (in G) defined so that e_1 and e_i ($i > 2$) lie on the bounding cycle of h and $e_2, \ldots, e_{i-1} \in E(G - H)$. Such a cyclical rotation and an index i of the edge ring of v exist and are uniquely defined by the invariant that the holes of H are simple and disjoint.

We first introduce a copy e'_1 of the edge e_1 embedded between e_q and e_1 in this order. We also introduce a copy e'_i of the edge e_i embedded between e_i and e_{i+1} in this order. The copies are given fresh identifiers $\mathrm{id}(e'_1)$, $\mathrm{id}(e'_i)$. Afterwards, we split v into two vertices v and v' and connect them by a pair of new edges e', e'' so that the old edge ring of v is also split as follows:

- the clockwise edge ring of v (in G) is now $e_1, \ldots, e_i, e', e''$,
- the clockwise edge ring of v' (in G) is now $e'', e', e'_i, e_{i+1}, \ldots, e_q, e'_1$.

The identifiers of the edges of the split edge ring are unchanged. This operation is performed consistently on H and all its (weak) ancestors H', and also on H^Δ (in H^Δ the edge ring of the vertex v will become e_1, vv_h, e_i, e', e'').

Let H' be any weak ancestor of H. Observe that if $v \in \partial H'$ then v is still a boundary vertex of H' after such an operation. Vertex v', on the other hand, has no adjacent edges in $G - H$,

and hence no adjacent edges in $G - H'$. As a result, we have $v' \notin V(G - H')$ (since $G - H'$ is edge-induced) and consequently $v' \notin \partial H'$. Observe that the transformation does not alter the holes of any of the ancestors H' (in terms of their bounding cycles) and only introduces two new triangular faces $e_1 e_1' e'$, $e_i e_i' e''$ and one face $e' e''$ of size 2 in each of these ancestors H' (including H and G).

The last step of the transformation is to replace v with v' on $C(H)$ and the edges e_1, e_i (if they appear on $C(H)$) with the edges e_1', e_i' respectively. Afterwards, $C(H)$ is still a simple cycle (of the same size) in H^Δ, but $v \notin C(H)$ and thus v is no longer bad.

Once there are no more bad vertices, setting $\mathrm{child}_1(H)$ and $\mathrm{child}_2(H)$ to be the part of H that is weakly inside and weakly outside of $C(H)$, respectively, produces pieces with simple and pairwise disjoint holes. We have $\mathrm{Sep}(H) = V(\mathrm{child}_1(H)) \cap V(\mathrm{child}_2(H)) = V(C(H)) \setminus A^\Delta$ and the size of $V(C(H))$ did not increase as a result of to the performed transformations, so we have $|\mathrm{Sep}(H)| = O(\sqrt{|V(H)|})$.

Note that setting $\mathrm{child}_i(H)$ to be the weak interior or weak exterior of $C(H)$ can generally lead to $E(\mathrm{child}_1(H)) \cap E(\mathrm{child}_2(H)) \neq \emptyset$, which violates our definition of a recursive decomposition. We circumvent this problem as follows. After the previous transformations, but before initializing $\mathrm{child}_1(H)$ and $\mathrm{child}_2(H)$, in H and all its ancestors, for each $e \in E(H) \cap E(C(H))$ we introduce an edge e' parallel to e and connecting the same endpoints, so that ee' forms a face of size 2. We make e embedded inside the curve $C(H)$ (i.e., we "bend" the embedding of the original e so that it lies inside the curve $C(H)$, as opposed to *on* the curve $C(H)$) and embed e' is outside $C(H)$. As a result, the embedding of H now intersects with the curve $C(H)$ only in the vertices of $C(H)$. Observe that this transformation does not change the status of the individual faces of the affected graphs: each face of size at least 3 is a hole if and only if it was a hole before the transformation. Also, all the affected graphs are still connected.

We now argue why each hole of $\mathrm{child}_1(H)$ consists solely of vertices $\partial\mathrm{child}_1(H)$ (the argument for $\mathrm{child}_2(H)$ is analogous). Recall that by invariants posed on H and Lemma 4.1.3, before transforming H, ∂H was equal to the set of vertices of holes of H. As we argued before, this is still true after the transformation of H which did not change either the boundary of H or its holes. If h is a hole of H as well, then by the invariant posed on H, it consists solely of vertices of $\partial H \cap V(\mathrm{child}_1(H)) \subseteq \partial\mathrm{child}_1(H)$, by Lemma 4.1.2. On the other hand, if h is not the hole of H, it is uniquely defined and consists of vertices of $\mathrm{Sep}(H)$ and vertices W of holes of H crossed by the curve $C(H)$ such that these vertices are inside $C(H)$. Since again by Lemma 4.1.2, $\mathrm{Sep}(H) \subseteq \partial\mathrm{child}_1(H)$ and by the invariants combined with Lemma 4.1.3, $W \subseteq \partial H \cap V(\mathrm{child}_1(H)) \subseteq \partial\mathrm{child}_1(H)$, we conclude that in fact $V(h) \subseteq \partial\mathrm{child}_1(H)$.

Now we discuss how we pick the weight function $x : V(H^\Delta) \to \mathbb{R}_{\geq 0}$ used to separate each piece H based on its level $\ell(H)$. We assign the weights as follows (recall that the simple cycle separator algorithm is run on H^Δ before any transformations):

- If $\ell(H) \equiv 0 \pmod 3$, we set $x(v) = [v \in V(H)]$ for $v \in V(H^\Delta)$.
- If $\ell(H) \equiv 1 \pmod 3$, we set $x(v) = [v \in \partial H]$ for $v \in V(H^\Delta)$.
- If $\ell(H) \equiv 2 \pmod 3$, we set $x(v) = [v \in A^\Delta]$ for $v \in V(H^\Delta)$.

Note that this strategy indeed guarantees that the number of holes remains $O(1)$ since if $\ell(H) \equiv 2 \pmod 3$, all holes h of H such that $v_h \in V(C(H))$ are replaced with a single new hole in the children of H, and the number of remaining holes on each side of $C(H)$ drops at least by a factor of $\frac{3}{2}$.

Denote by G' the root piece of the obtained decomposition, after the decomposition of G ends (in the following we assume G is the initial, input graph). For $v \in V$, we set $S(v)$ to be $\{v\}$ combined with the set of vertices v' of root piece G' that were split out of v during the process,

the set of vertices split out of all v' and so on. It is clear that each time a vertex is split, the two obtained vertices are connected by a pair of parallel edges. We include all these parallel edges in the set E_0. Hence, $G'[S(v)]$ is connected for each $v \in S(v)$ and in fact the connected components of (V', E_0) are exactly the sets $S(v)$. Each introduced pair of parallel edges can be oriented to form a directed cycle of length 2, and therefore $G'[S(v)]$ can be easily oriented to be strongly-connected.

For each original edge $uv = e \in E$, e may change its endpoints as a consequence of vertex splits, but it is easy to see that, throughout the process, the former endpoint always belongs to $S(u)$ and the latter always belongs to $S(v)$. Since the identifiers of edges connecting vertices from different sets $S(u), S(v)$ are never changed, it is easy to see that the edge $e' \in E(G')$ such that $\mathrm{id}(e') = \mathrm{id}(e)$ connects a vertex from $S(u)$ with a vertex of $S(v)$. These edges form the set E', i.e., $E' = \{e' \in E(G') : \mathrm{id}(e') \in \mathrm{id}(E)\}$.

We define $E^\times$ to be the set of remaining auxiliary edges, i.e., $E^\times = E(G') \setminus (E_0 \cup E')$.

Now that we described our construction and argued that the invariants are preserved in the children of each piece, which results in the obtained decomposition being simple, we need to prove that the quantitative properties (1), (2), (3), and (4) of the recursive decomposition are also satisfied.

Let $f = 2\sqrt{2}$ and $\gamma = \frac{2}{3}$. First, we introduce a notion of *initial* and *extra* vertices of each piece H of the obtained decomposition. The initial vertices $V_0(H) \subseteq V(H)$ are the vertices of H before H is transformed and further decomposed, or in other words, $V_0(H)$ is the set of vertices of H at the beginning of the recursive call on H. Here, by $V(H)$ we denote the final set of vertices of H, after the decomposition of H ends. Recall that if $|V_0(H)| \geq n_0$, then a simple cycle separator $C(H)$ is computed, and at most $|V(C(H))| \leq f\sqrt{|V_0(H)|}$ new vertices are introduced to make sure there are no bad vertices and the decomposition algorithm is run on the parts of H on the two sides of $C(H)$. The children of H are further decomposed and each new vertex introduced in a descendant piece of H is also added to H. When the algorithm terminates, $V_0(H) \subseteq V(H)$, but $V(H)$ can be generally larger than $V_0(H)$. We call the vertices $V(H) \setminus V_0(H)$ extra vertices of H. Observe that the set ∂H is never altered by the transformations of H or its descendants (i.e., ∂H is fixed before H is further decomposed) so we have $\partial H \subseteq V_0(H)$.

Let n_0 be such that $\gamma n_0 + 5f\sqrt{n_0} < n_0$.

Lemma 4.1.5. *Suppose a recursive call is being performed on a graph H with n initial vertices. Then, $\chi(H) = O(n)$ and all pieces in the decomposition of H have $O(n)$ initial vertices.*

Proof. First note that if a piece H has n initial vertices, the transformations on H never add more than $\min(f\sqrt{n}, n)$ vertices. Hence, a child of any piece has no more than twice as many initial vertices than that piece. Moreover, if the level of H (in $\mathcal{T}_G$) is divisible by 3, we perform a balanced partition of H with respect to its initial vertices and then add at most $f\sqrt{n}$ extra vertices to H before initializing the children of H. Thus, the children of H have at most $\gamma n + 2f\sqrt{n}$ initial vertices in this case.

Suppose H' is any nearest descendant of H such that $\ell(H') \equiv 1 \pmod 3$ (with respect to $\mathcal{T}_{G'}$). Observe that the parent of H' can have no more than $n + f\sqrt{n} + f\sqrt{2n} < n + 3f\sqrt{n}$ initial vertices. Hence, H' can have at most $\gamma(n + 3f\sqrt{n}) + 2f\sqrt{n} < \gamma n + 5f\sqrt{n} < n$ initial vertices. Consequently, on any H-to-leaf path of the decomposition of H, the pieces H' at levels $\ell(H') \equiv 1 \pmod 3$ (with respect to $\mathcal{T}_{G'}$) have decreasing numbers of initial vertices. We thus have $\chi(H) \leq 3n + 4 = O(n)$.

Note also that each piece H' in the decomposition of H has no more than $4n = O(n)$ initial vertices as either H' or some of the two nearest ancestors of H' is either equal to H or has level congruent to 1 modulo 3 and hence has no more than n initial vertices. □

Lemma 4.1.6. *There exist constants K, d such that $K \geq 1$ is an integer, $d \geq 1$, and such that for any n_0 satisfying $\gamma n_0 + 5f\sqrt{n_0} < n_0$, there exists $n^* \geq n_0$ satisfying the following property.*

Suppose a recursive call is being performed on a graph H with $n \geq n^*$ initial *vertices and b boundary vertices. Then, after K levels of recursive decomposition according to the described rules, we obtain a decomposition of H into $q \leq 2^K$ leaf graphs $H_1, \ldots, H_q$ such that:*

- *each H_i has n'_i initial vertices and $n_i \leq n'_i \leq n_i + d\sqrt{n} < \frac{1}{2}n$,*
- *each H_i has at most b'_i boundary vertices and $b_i \leq b'_i \leq b_i + d\sqrt{n}$,*
- *$\sum_{i=1}^{q} n_i \leq n$, and $n_i \in [0, \frac{1}{3}n]$,*
- *$\sum_{i=1}^{q} b_i \leq b$, and $b_i \in [0, \frac{1}{3}b]$,*
- *for each intermediate graph H' that is decomposed to obtain $H_1, \ldots, H_q$ (i.e., H' is a descendant of H and an ancestor of some H_i), H' has at most $b + d\sqrt{n}$ boundary vertices.*

Proof. Consider the recursion tree of this algorithm called on a graph H, but truncated to the first $K+1$ levels, where K is divisible by 3. The graph H is considered to be at level 0 in this truncated tree. Let n^* be such that $n + 2Kf\sqrt{n} \leq 4n$ for any $n \geq n^*$.

Let us analyze the maximum number of initial vertices N_i of some i-th level piece in this truncated tree. First, we have $N_i \leq N_{i-1} + f\sqrt{N_{i-1}}$ for $0 < i \leq K$. If $\ell(H) + i \equiv 1 \pmod 3$, then we have $N_i \leq \gamma N_{i-1} + 2f\sqrt{N_{i-1}}$ (see the proof of Lemma 4.1.5).

Let us now prove $N_i \leq n + 2if\sqrt{n}$. The bound is valid for $i = 0$ as $N_0 = n$. Proceeding inductively and using $n \geq n^*$, for $i > 0$ we obtain

$$\begin{aligned} N_i &\leq N_{i-1} + f\sqrt{N_{i-1}} \\ &\leq n + 2(i-1)f\sqrt{n} + f\sqrt{n + 2(i-1)f\sqrt{n}} \\ &\leq n + 2(i-1)f\sqrt{n} + f\sqrt{4n} \\ &= n + 2if\sqrt{n}. \end{aligned}$$

Consequently, for $i \leq K$ we obtain

$$N_i \leq n + 2if\sqrt{n} \leq n + 2Kf\sqrt{n} \leq 4n$$

and, as a result, $N_i \leq N_{i-1} + 2f\sqrt{n}$. For $\ell(H) + i \equiv 1 \pmod 3$, in turn, we have $N_i \leq \gamma N_{i-1} + 4f\sqrt{n}$.

Using this, we now prove a stronger bound $N_i \leq \gamma^{\lfloor i/3 \rfloor} n + 3if\sqrt{n}$. The bound holds trivially for $i = 0, 1, 2$. Suppose $i \geq 3$. For $\ell(H) + i \equiv 0 \pmod 3$, we have

$$\begin{aligned} N_i &\leq N_{i-1} + 2f\sqrt{n} \\ &\leq N_{i-2} + 4f\sqrt{n} \\ &\leq \gamma N_{i-3} + 8f\sqrt{n} \\ &\leq \gamma^{\lfloor i/3 \rfloor} n + 3(i-3)f\sqrt{n} + 8f\sqrt{n} \\ &\leq \gamma^{\lfloor i/3 \rfloor} n + 3if\sqrt{n}. \end{aligned}$$

If $\ell(H) + i \equiv 1 \pmod 3$, we have

$$\begin{aligned} N_i &\leq \gamma N_{i-1} + 4f\sqrt{n} \\ &\leq \gamma(N_{i-2} + 2f\sqrt{n}) + 4f\sqrt{n} \\ &\leq \gamma(N_{i-3} + 4f\sqrt{n}) + 4f\sqrt{n} \\ &\leq \gamma \cdot \gamma^{\lfloor (i-3)/3 \rfloor} n + 3(i-3)f\sqrt{n} + 8f\sqrt{n} \\ &\leq \gamma^{\lfloor i/3 \rfloor} n + 3if\sqrt{n}. \end{aligned}$$

Finally, if $\ell(H) + i \equiv 2 \pmod 3$, we have

$$\begin{aligned} N_i &\leq N_{i-1} + 2f\sqrt{n} \\ &\leq \gamma N_{i-2} + 6f\sqrt{n} \\ &\leq \gamma(N_{i-3} + 2f\sqrt{n}) + 6f\sqrt{n} \\ &\leq \gamma \cdot \gamma^{\lfloor (i-3)/3 \rfloor} n + 3(i-3)f\sqrt{n} + 8f\sqrt{n} \\ &\leq \gamma^{\lfloor i/3 \rfloor} n + 3if\sqrt{n}. \end{aligned}$$

Now let B_i be the maximum number of boundary vertices of a level-i piece in the considered truncated tree. As we have $B_i \leq B_{i-1} + f\sqrt{N_{i-1}} \leq B_{i-1} + 2f\sqrt{n}$ for all $0 < i \leq K$ and $B_i \leq \gamma B_{i-1} + 2f\sqrt{n}$ if $\ell(H) + i \equiv 2 \pmod 3$, we can proceed analogously as for N_i and prove that $B_i \leq \gamma^{\lfloor i/3 \rfloor} b + 2if\sqrt{n}$.

Denote by $V_1(H)$ the set $\bigcup_{i=1}^{q} V_0(H_i)$. We obviously have $V_0(H) \subseteq V_1(H) \subseteq V(H)$. Note that each vertex of $V_1(H) \setminus V_0(H)$ is introduced when a bad vertex of some non-leaf (in the truncated tree) piece is eliminated. As for each piece this happens at most $f\sqrt{4n} = 2f\sqrt{n}$ times and the number of non-leaf pieces is at most $q - 1$, we obtain

$$|V_1(H)| = |V_0(H)| + |V_1(H) \setminus V_0(H)| \leq n + 2qf\sqrt{n} \leq n + 2^{K+1} f\sqrt{n}.$$

Now let us set $K \geq 6$ divisible by 3 such that $\gamma^{K/3} \leq \frac{1}{3}$, $d = 2^{K+1}f + 1$ and n^* large enough so that the following inequalities are all satisfied for $n \geq n^*$:

- $2^{K+1} f\sqrt{n} > n_0$,
- $\gamma^{K/3} n + d\sqrt{n} < \frac{1}{2} n$,
- $n + 2Kf\sqrt{n} \leq 4n$,
- $2^{2K+1} f\sqrt{n} \leq n$.

Define now $X_i = V(H_i) \cap V(H - H_i)$. Since $\partial H = V(H) \cap V(G - H)$ and $\partial H_i = V(H_i) \cap V(G - H_i)$, we have $X_i \subseteq \partial H_i \subseteq V_0(H_i)$ and $\partial H_i \setminus X_i = V(H_i) \cap (V(G - H_i) \setminus V(H - H_i)) \subseteq V(H_i) \cap V(G - H) \subseteq \partial H \subseteq V_0(H)$. Note that if $v \in V(H_i) \cap V(H_j)$, where $i \neq j$, then $v \in X_i \cap X_j$. Hence, any vertex of H_i not contained in X_i cannot be found in any other H_j. We now prove the existence of the numbers $n_1, \ldots, n_q$.

Let $p \in \{1, \ldots, q\}$ be such that $|V_0(H_p)| \geq 2^{K+1} f\sqrt{n}$. Such p indeed exists as some leaf piece has size at least $\frac{n}{q} \geq \frac{n}{2^K} \geq 2^{K+1} f\sqrt{n}$ and we assumed $n \geq n^*$.

Observe that we have $|X_i| \leq 2Kf\sqrt{n}$ since

$$X_i \subseteq \bigcup \{\mathrm{Sep}(A) : A \text{ is an ancestor of } H_i \text{ in the truncated tree}\},$$

H_i has at most K ancestors A, and we have $\mathrm{Sep}(A) \leq f\sqrt{4n} = 2f\sqrt{n}$. Let X_i' be any subset of $V_0(H_i)$ of size $\min(|V_0(H_i)|, \lceil 2^{K+1} f\sqrt{n} \rceil)$ such that $X_i \subseteq X_i'$. We set $n_i = |V_0(H_i) \setminus X_i'|$. It follows that H_i has at most $n_i + \lceil 2^{K+1} f\sqrt{n} \rceil \leq n_i + d\sqrt{n}$ initial vertices. The sets $V_0(H_i) \setminus X_i$, where $i \neq p$, and $V_0(H_p)$ are all pairwise disjoint, and are all contained in $V_1(H)$, hence

$$\begin{aligned} \sum_{i=1}^{q} n_i &= \sum_{i=1}^{q} |V_0(H_i) \setminus X_i'| \\ &\leq \sum_{\substack{1 \leq i \leq q \\ i \neq p}} |V_0(H_i) \setminus X_i| + |V_0(H_p)| - 2^{K+1} f\sqrt{n} \\ &\leq |V_1(H)| - 2^{K+1} f\sqrt{n} \\ &\leq n. \end{aligned}$$

Moreover, if H_i is not at level K of the truncated tree, then $|V(H_i)| < n_0 < 2^{K+1} f\sqrt{n^*} \leq 2^{K+1} f\sqrt{n}$. Hence, $X_i' = V_0(H_i)$ in that case. Therefore, we have either $n_i = 0 \leq \frac{1}{3}n$ if $X_i' = V_0(H_i)$ or otherwise

$$\begin{aligned} n_i &= |V_0(H_i)| - \lceil 2^{K+1} f\sqrt{n} \rceil \\ &\leq N_K - 2^K f\sqrt{n} \\ &\leq \gamma^{K/3} n + 3Kf\sqrt{n} - 2^K f\sqrt{n} \\ &\leq \gamma^{K/3} n \\ &\leq \frac{1}{3} n. \end{aligned}$$

Similarly we prove that the numbers $b_1, \ldots, b_q$ within the required bounds do exist. In this case we set $b_i = |\partial H_i \setminus Y_i'|$, where Y_i' is now defined to be any subset of ∂H_i such that $X_i \subseteq Y_i'$ and $|Y_i'| = \min(|\partial H_i|, \lceil 2Kf\sqrt{n} \rceil)$. Again, the sets $\partial H_i \setminus Y_i'$ are pairwise disjoint and are all contained in ∂H, thus $\sum_{i=1}^{q} b_i \leq b$. Observe that if the level of H_i in the truncated tree is less than K, then $|\partial H_i| \leq |V(H_i)| \leq n_0 \leq 2Kf\sqrt{n}$, and hence $Y_i' = \partial H_i$. If $Y_i' = \partial H_i$, then $b_i = 0$; otherwise (if H_i is at level K and $Y_i' \neq \partial H_i$), we have $b_i \leq B_K - 2Kf\sqrt{n} \leq \gamma^{K/3} b \leq \frac{1}{3} b$. □

We now analyze a recurrence that will prove very useful when arguing about our decomposition algorithm.

Lemma 4.1.7. *Let K, d (independent of n_0) and n^* (dependent on n_0) be constants as in Lemma 4.1.6. Let $u, z \geq 1$ and $\beta \in [0, \frac{1}{2}] \cup \{1\}$ be some real constants.*

Let $T_{z,\beta} : \mathcal{T}_{G'} \to \mathbb{R}_{\geq 0}$ be a function satisfying:

$$\begin{aligned} &T_{u,z,\beta}(H) = un^\beta + zb^{2\beta} && \text{if } H \in \mathcal{T}_G \text{ is a leaf,} \\ &T_{u,z,\beta}(H) = un^\beta + zb^{2\beta} + T_{u,z,\beta}(\text{child}_1(H)) + T_{u,z,\beta}(\text{child}_2(H)) && \text{otherwise.} \end{aligned}$$

where $n = |V_0(H)|$ and $b = |\partial H|$. If n_0 is sufficiently large (dependent on u, z and β), then:

$$T_{u,z,\beta}(H) = \begin{cases} O(n + b\log n) & \text{for } \beta \in [0, \frac{1}{2}], \\ O(n\log n + b^2) & \text{for } \beta = 1. \end{cases}$$

Proof. Let $T(n, b)$ be defined as the maximum possible value of $T_{u,z,\beta}(H)$ for any simple recursive decomposition $\mathcal{T}_{G'}$ and $H \in \mathcal{T}_{G'}$ such that $|V_0(H)| = n$ and $|\partial H| = b$. Let $Q = 2^K$. By Lemma 4.1.5 we have $T(n, b) = 2^{O(n)} \cdot (u + z) \cdot O(n^{2\beta})$, hence each $T(n, b)$ is finite. Let

$$M = \max_{\substack{1 \leq n < n^* \\ 0 \leq b \leq n}} T(n, b).$$

Clearly, $M \geq 1$.

Observe that by Lemmas 4.1.5 and 4.1.6, when we decompose a piece H with $n \geq n^*$ vertices and b boundary vertices, each of weak descendants H' of H at levels $\ell(H)$ through $\ell(H) + K$ has $O(n)$ vertices and at most $b + d\sqrt{n}$ boundary vertices. Hence, for each of these pieces H' we would add an $O(un^\beta + (b + d\sqrt{n})^{2\beta}) = O(n^\beta + b^{2\beta})$ term when computing $T_{u,z,\beta}(H)$ directly using its definition. Hence, there exists a constant $p \geq 1$ such that if $n \geq n^*$, then we have

$$T(n, b) \leq Qp(n^\beta + b^{2\beta}) + \max \left\{ \sum_{i=1}^{q} T(n_i', b_i') \right\},$$

where the maximum is taken over:

- all possible q, $1 \leq q \leq Q = 2^K$,

- all possible q-tuples of integers $n_1, \ldots, n_q \in [0, \frac{1}{3}n]$ such that $\sum_{i=1}^{q} n_i \le n$,
- all possible q-tuples of integers $n'_1, \ldots, n'_q$ such that $n'_i \in [n_i, n_i + d\sqrt{n}]$.
- all possible q-tuples of integers $b_1, \ldots, b_q \in [0, \frac{1}{3}b]$ such that $\sum_{i=1}^{q} b_i \le b$,
- all possible q-tuples of integers $b'_1, \ldots, b'_q$ such that $b'_i \in [b_i, b_i + d\sqrt{n}]$.

We first inductively prove that for $\beta \in [0, \frac{1}{2}]$, if n_0 is sufficiently large, we have

$$T(n, b) \le M \cdot \max(n - xn^{0.6} + yb \log_2 n, 1)$$

for all n, where $y = 2Qp$ and $x = \frac{3^{0.6}}{2-3^{0.6}} \ge 1$. Clearly the bound holds for $n \le n^*$ by the definition of M.

Let n_0 be sufficiently large so that the following inequalities hold for all $n \ge n_0$:

- $n^{0.6} + \frac{2}{3}n \le n - x\sqrt{n}$,
- $2Q(p + yd)\sqrt{n} \log_2 n \le n^{0.6}$

Suppose now that $n \ge n^*$ and the bound $T(n, b) \le M \cdot \max(n - xn^{0.6} + yb \log_2 n, 1)$ holds for all $n' < n$. For fixed $n'_1, \ldots, n'_q, b'_1, \ldots, b'_q$, let $S = \{i : 1 \le i \le q \wedge n'_i - x(n'_i)^{0.6} + yb \log_2 n'_i \ge 1\}$. We have:

$$\begin{aligned}
T(n,b) &\le Qp(n^\beta + b^{2\beta}) + \max\left\{\sum_{i=1}^{q} T(n'_i, b'_i)\right\} \\
&\le Qp(\sqrt{n} + b) + \max\left\{\sum_{i=1}^{q} T(n'_i, b'_i)\right\} && \beta \le \frac{1}{2} \\
&= Qp(\sqrt{n} + b) + \max\left\{\sum_{i\in S} T(n'_i, b'_i) + \sum_{i\in\{1,\ldots,q\}\setminus S} T(n'_i, b'_i)\right\} \\
&\le Qp(\sqrt{n} + b) + \max\left\{\sum_{i\in S} T(n'_i, b'_i)\right\} + QM && q \le Q \\
&\le Qp(\sqrt{n} + b) + QM + M \cdot \max\left\{\sum_{i\in S} \left(n'_i - x(n'_i)^{0.6} + yb'_i \log_2 n'_i\right)\right\} && \text{induction} \\
&\le M \cdot \left(2Qp(\sqrt{n} + b) + \max\left\{\sum_{i\in S} \left(n'_i - x(n'_i)^{0.6} + yb'_i \log_2 n'_i\right)\right\}\right) && M, p, n \ge 1 \\
&\le M \cdot \left(2Qp(\sqrt{n} + b) + \max\left\{\sum_{i\in S} \left(n_i - xn_i^{0.6} + yb_i \log_2\left(\frac{1}{2}n\right)\right)\right\}\right. && n_i \le n'_i \le \frac{1}{2}n \\
&\qquad \left. +2yQd\sqrt{n} \log_2 n\right) \\
&\le M \cdot \left(2Q(p + yd)\sqrt{n} \log_2 n + b(2Qp + y \log_2 n - y)\right. \\
&\qquad \left. + \max\left\{\sum_{i\in S} \left(n_i - xn_i^{0.6}\right)\right\}\right) \\
&\le M \cdot \left(n^{0.6} + by \log_2 n + \max\left\{\sum_{i\in S} n_i - x\sum_{i\in S} n_i^{0.6}\right\}\right) && n \ge n_0, y = 2Qp
\end{aligned}$$

Now suppose the maximum above is attained when $\sum_{i\in S} n_i < \frac{2}{3}n$. We then have:

$$\begin{aligned} n^{0.6} + \max\left\{\sum_{i\in S} n_i - x\sum_{i\in S} n_i^{0.6}\right\} &\le n^{0.6} + \max\left\{\sum_{i\in S} n_i\right\} \\ &\le n^{0.6} + \frac{2}{3}n \\ &\le n - xn^{0.6}. \end{aligned}$$

If on the other hand the maximum is attained when $n \ge \sum_{i\in S} n_i \ge \frac{2}{3}n$, then

$$n^{0.6} + \max\left\{\sum_{i\in S} n_i - x\sum_{i\in S} n^{0.6}\right\} \le n - \left(\frac{2}{3^{0.6}}x - 1\right)n^{0.6} = n - xn^{0.6}.$$

The last inequality above is by the fact that $\sum_{i\in S} n_i^{0.6}$ is minimized (for real numbers $n_1, \ldots, n_q$) subject to $\sum_{i\in S} n_i \ge \frac{2}{3}n$ and $0 \le n_i \le \frac{1}{3}n$ when $\sum_{i\in S} n_i = \frac{2}{3}n$ and the individual n_i's ($i \in S$) are largest possible, i.e., two of them are equal to $\frac{1}{3}n$, and all other are equal to 0. Hence, $\sum_{i\in S} n_i^{0.6} \ge 2\left(\frac{n}{3}\right)^{0.6}$.

Now we move on to the case $\beta = 1$. We prove that $T(n,b) \le xn\log_2 n + yb^2$, where $y = 3Qp$ and $x = \max(M, Qp + 2yQd^2 + 1)$. Let n_0 be sufficiently large so that the inequality $xQd\sqrt{n}\log_2 n \le n$ holds for all $n \ge n_0$.

Clearly, since $x \ge M$, the bound $T(n,b) \le xn\log_2 n + yb^2$ holds for all $n < n^*$. Now suppose $n \ge n^*$ and assume the bound holds for all $n' < n$. We have

$$\begin{aligned} T(n,b) &= Qp(n+b^2) + \max\left\{\sum_{i=1}^{q} T(n_i', b_i')\right\} \\ &\le Qp(n+b^2) + \max\left\{\sum_{i=1}^{q} xn_i'\log_2(n_i') + y(b_i')^2\right\} && n_i' < n \text{ and induction} \\ &\le Qp(n+b^2) + \max\left\{\sum_{i=1}^{q} x(n_i + d\sqrt{n})\log_2\frac{n}{2} + y(b_i + d\sqrt{n})^2\right\} && n_i' \le n_i + d\sqrt{n} \le \frac{1}{2}n \\ &\le Qp(n+b^2) + \max\left\{\sum_{i=1}^{q} x(n_i + d\sqrt{n})\log_2\frac{n}{2} + 2y(b_i^2 + d^2 n)\right\} && (e+f)^2 \le 2e^2 + 2f^2 \\ &\le (Qp + 2yQd^2)n + Qpb^2 + xQd\sqrt{n}\log_2 n \\ &\quad + \max\left\{\sum_{i=1}^{q} xn_i\log_2\frac{n}{2} + 2yb_i^2\right\} \\ &\le (Qp + 2yQd^2 + 1 + x\log_2 n - x)n + Qpb^2 + 2y\max\left\{\sum_{i=1}^{q} b_i^2\right\} && \sum_{i=1}^{q} n_i \le n,\ n \ge n_0 \\ &\le (Qp + 2yQd^2 + 1 + x\log_2 n - x)n + \left(Qp + \frac{2}{3}y\right)b^2 && \sum_{i=1}^{q} b_i^2 \le \frac{1}{3}b^2 \\ &= xn\log_2 n + \left(Qp + \frac{2}{3}y\right)b^2 && x = Qp + 2yQd^2 + 1 \\ &= xn\log_2 n + yb^2. && y = 3Qp \end{aligned}$$

In the above, the inequality $\sum_{i=1}^{q} b_i^2 \le \frac{1}{3}b^2$ is justified by the fact that, by the convexity of the function x^2, the sum $\sum_{i=1}^{q} x_i^2$ is maximized subject to $\sum_{i=1}^{q} x_i \le X$ and $0 \le x_i \le \frac{1}{3}X$

(for reals $x_1, \ldots, x_q$) if $x_1 = x_2 = x_3 = \frac{1}{3}X$ and $x_i = 0$ for $i > 3$. In other words, under these constraints we have $\sum_{i=1}^{q} x_i^2 \leq 3 \cdot \frac{X^2}{3^2} = \frac{X^2}{3}$. Therefore, $\sum_{i=1}^{q} b_i^2 \leq \frac{1}{3}b^2$.

Hence, the bound holds for all n. We have proved inductively that for $\beta = 1$ we indeed have $T(n, b) = O(n \log n + b^2)$. □

In the following, let n^*, K, d be as in Lemma 4.1.6. The following Lemmas 4.1.8 and 4.1.9 prove properties (1) and (2), respectively.

Lemma 4.1.8. *Let $H \in \mathcal{T}_{G'}$ and $n = |V_0(H)|$. Then $\chi(H) = O(\log n)$.*

Proof. Denote by $h(n)$ the maximum height of a decomposition of an n-vertex piece. By Lemma 4.1.5, $h(n) = O(n)$ and hence $h(n)$ is finite. We prove inductively that $h(n) \leq y + K \log_2 n$, where $y = \max_{1 \leq n < n^*} h(n) = O(1)$. The bound holds trivially for $n < n^*$.

Suppose now that it holds for all $n' < n$, where $n \geq n^*$. Then, by Lemma 4.1.6:

$$\begin{aligned}
h(n) &\leq K + \max_{(n_i')_{i=1}^q} \left\{ \max_{1 \leq i \leq q} h(n_i') \right\} && (n_i')_{i=1}^q \text{ as in Lemma 4.1.6} \\
&\leq K + y + K \cdot \max_{(n_i')_{i=1}^q} \left\{ \max_{1 \leq i \leq q} \log_2(n_i') \right\} \\
&\leq K + y + K \log_2 \left(\frac{1}{2} n \right) && \text{monotonicity of } \log_2 n \text{ and } n_i' < \frac{1}{2}n \\
&\leq y + K \log_2 n.
\end{aligned}$$

We conclude that indeed $h(n) = O(\log n)$ and hence $\chi(H) \leq h(n) = O(\log n)$. □

Lemma 4.1.9. *Let $H \in \mathcal{T}_{G'}$ and $n = |V_0(H)|$. Then $\mathcal{T}_{G'}(H)$ has $O(n)$ leaves and each leaf has $O(1)$ vertices.*

Proof. Since the decomposition stops only when the piece to be decomposed has size less than n_0, each leaf has no more than $n_0 = O(1)$ vertices.

Now consider $N(H)$, the total number of pieces in the decomposition of H, i.e., $N(H) = |\mathcal{T}_G(H)|$. We have $N(H) = 1$ if H is a leaf and $N(H) = 1 + N(\text{child}_1(H)) + N(\text{child}_2(H))$ otherwise. Hence, in terms of the notation of Lemma 4.1.7, $N = T_{1,0,0}$. Consequently, by Lemma 4.1.7, $N(H) = O(n)$.

Since the number $L(H)$ of leaf pieces in $\mathcal{T}_{G'}(H)$ is no more than $|\mathcal{T}_{G'}(H)|$, $L(H) = O(n)$. □

Lemma 4.1.10. *Let $H_0 \in \mathcal{T}_{G'}$ and $n = |V_0(H_0)|$. There exists a constant $\rho > 1$ such that for any $H \in \mathcal{T}_{G'}(H_0)$, H has $O(n/\rho^{\ell(H) - \ell(H_0)})$ initial vertices.*

Proof. Let $s \geq 1$ be an integer such that for any graph H' with n' initial vertices, the decomposition of H' has height no more than sn'. Similarly, let $g \geq 1$ be a constant such that for any H' with n' initial vertices, all pieces in the decomposition of H' have no more than gn' initial vertices. Both constants are guaranteed to exist by Lemma 4.1.5.

Let $\ell'(H) = \ell(H) - \ell(H_0)$. Let $\rho = 2^{1/K}$. We first prove that if $|V_0(H)| \geq n^*$, then $|V_0(H)| \leq g\rho^{K - \ell'(H)} n$. We use induction on $\ell'(H)$. For $0 \leq \ell'(H) < K$, we have $|V_0(H)| \leq gn \leq gn \cdot \rho^{K - \ell'(H)}$ and the bound clearly holds.

Now suppose $\ell'(H) \geq K$ and that the bound holds for all ancestors of H. Let H' be the ancestor of H at level $\ell(H) - K$. Then, by Lemma 4.1.6 applied to H',

$$|V_0(H)| \leq \frac{1}{2}|V_0(H')| = \frac{1}{\rho^K}|V_0(H')| \leq \frac{1}{\rho^K} \cdot g\rho^{K - \ell'(H')} n = \frac{1}{\rho^K} \cdot g\rho^{2K - \ell'(H)} n = g\rho^{K - \ell'(H)} n.$$

Now suppose $|V_0(H)| < n^*$. If all ancestors of H in $\mathcal{T}_{G'}(H_0)$ have less than n^* vertices, then by Lemma 4.1.5, $\ell'(H) \leq sn \leq sn^*$ and thus $\rho^{-\ell'(H)+sn^*} \geq 1$. Thus, by Lemma 4.1.5,

$$|V_0(H)| \leq gn \leq g\rho^{K-\ell'(H)+sn^*}n.$$

Otherwise, let H' be the nearest ancestor of H (in $\mathcal{T}_{G'}(H_0)$) of size at least n^*. Let H'' be the child of H' that is an ancestor of H (or is equal to H). By Lemma 4.1.5, we have $\chi(H'') \leq s \cdot (n^* - 1)$. From

$$\ell'(H) \leq \ell'(H'') + \chi(H'') = \ell'(H') + 1 + \chi(H'') \leq \ell'(H') + s \cdot n^*$$

and $|V_0(H')| \leq g\rho^{K-\ell'(H')}n$, we obtain

$$1 \leq |V_0(H')| \leq g\rho^{K-\ell'(H)+sn^*}n.$$

By Lemma 4.1.5 applied to H', we conclude $|V_0(H)| \leq g|V_0(H')| \leq g^2\rho^{K-\ell'(H)+sn^*}n$.

Hence, for all pieces H, regardless of the size of $V_0(H)$, we have

$$|V_0(H)| \leq g^2\rho^{K+sn^*} \cdot n/\rho^{\ell(H)-\ell(H_0)} = O(n/\rho^{\ell(H)-\ell(H_0)}). \quad \square$$

Lemma 4.1.11. *Let $H_0 \in \mathcal{T}_{G'}$, $n = |V_0(H_0)|$ and $b = |\partial H_0|$. Let ρ be the constant from Lemma 4.1.10. For any $H \in \mathcal{T}_{G'}(H_0)$, $|\partial H| = O\left(\sqrt{n/\rho^{\ell(H)-\ell(H_0)}} + b/\rho^{\ell(H)-\ell(H_0)}\right)$.*

Proof. We proceed analogously as we did when bounding $|V_0(H)|$. Let s and g be defined as in the proof of Lemma 4.1.10 and let $\ell'(H) = \ell(H) - \ell(H_0)$. Recall that in the proof of Lemma 4.1.10 we set $\rho = 2^{1/K}$.

First we prove that for a sufficiently large constant $u \geq d$ (to be determined later, d is as in Lemma 4.1.6), if $|V_0(H)| \geq n^*$, then

$$|\partial H| \leq u\sqrt{\rho^K} \cdot \sqrt{n/\rho^{\ell'(H)}} + b \cdot \rho^{K-\ell'(H)}.$$

We proceed by induction on $\ell'(H)$. Indeed, if $0 \leq \ell'(H) < K$, then by Lemma 4.1.6 applied to H_0:

$$|\partial H| \leq |\partial H_0| + d\sqrt{n} = b + d\sqrt{n} \leq u\sqrt{n}\sqrt{\rho^{K-\ell'(H)}} + b \cdot \rho^{K-\ell'(H)}.$$

Now suppose $\ell'(H) \geq K$ and that the bound holds for all ancestors of K. Let H' be the ancestor of H at level $\ell(H) - K$. Then, by Lemmas 4.1.6 and 4.1.10 (both applied to H'), for some constant $t > 1$ we have:

$$\begin{aligned}
|\partial H| &\leq \frac{1}{3}|\partial H'| + d\sqrt{|V_0(H')|} \\
&\leq \frac{1}{3}u\sqrt{n\rho^{K-\ell'(H')}} + \frac{1}{3}b \cdot \rho^{K-\ell'(H')} + d\sqrt{tn\rho^{-\ell'(H')}} \\
&= \sqrt{n}\left(\frac{1}{3}u\sqrt{\rho^{2K-\ell'(H)}} + d\sqrt{t}\sqrt{\rho^{K-\ell'(H)}}\right) + \frac{1}{3}b \cdot \rho^{2K-\ell'(H)}.
\end{aligned}$$

Since $\frac{1}{3}b \cdot \rho^{2K-\ell'(H)} = \frac{\rho^K}{3}b\rho^{K-\ell'(H)} = \frac{2}{3}b\rho^{K-\ell'(H)} \leq b\rho^{K-\ell'(H)}$, it is enough to pick u such that

$$\frac{1}{3}u\sqrt{\rho^{2K-\ell'(H)}} + d\sqrt{t}\sqrt{\rho^{K-\ell'(H)}} \leq u \cdot \sqrt{\rho^{K-\ell'(H)}}.$$

Equivalently,

$$\frac{1}{3}u\sqrt{\rho^K} + d\sqrt{t} \leq u.$$

As $\rho = 2^{1/K}$, it is sufficient to set $u = \frac{d\sqrt{t}}{1-\frac{1}{3}\sqrt{2}} \geq d$.

Now consider the case when $|V_0(H)| < n^*$. Then if H has no ancestors with at least n^* vertices in $\mathcal{T}_{G'}(H_0)$ then by Lemma 4.1.5, $\ell'(H) \leq sn \leq sn^*$ and thus $\rho^{\ell'(H)-sn^*} \leq 1$ and $\sqrt{tn/\rho^{\ell'(H)-sn^*}} \geq 1$.

Similarly, if H has the nearest ancestor H' with at least n^* vertices, then $\ell'(H') \geq \ell'(H) - sn^*$, and by Lemma 4.1.10, $1 \leq |V_0(H')| \leq tn/\rho^{\ell'(H')} \leq tn/\rho^{\ell'(H)-sn^*}$. We can conclude $\sqrt{tn/\rho^{\ell'(H)-sn^*}} \geq 1$ also in this case.

Therefore,

$$|\partial H| < n^* \leq n^*\sqrt{tn/\rho^{\ell'(H)-sn^*}} = n^*\sqrt{t\rho^{sn^*}} \cdot \sqrt{n/\rho^{\ell'(H)}}.$$

Hence, by combining the obtained bounds, for all pieces H we obtain

$$|\partial H| \leq \left(u\sqrt{\rho^K} + n^*\sqrt{t\rho^{sn^*}}\right) \cdot \sqrt{n/\rho^{\ell'(H)}} + b \cdot \rho^{K-\ell'(H)} = O\left(\sqrt{n/\rho^{\ell(H)-\ell(H_0)}} + b/\rho^{\ell(H)-\ell(H_0)}\right).$$

□

Lemma 4.1.11 proves property (3) of the obtained decomposition. The below Lemma 4.1.12 proves property (4).

Lemma 4.1.12. *Let $n = |V(G)| = |V_0(G')|$. Then, $\sum_{H \in \mathcal{T}_{G'}} |\partial H|^2 = O(n \log n)$.*

Proof. Let $S_2(H) = \sum_{H' \in \mathcal{T}_{G'}(H)} |\partial H'|^2$. Observe that $S_2(H) = |\partial H|^2$ if H is a leaf and $S_2(H) = |\partial H|^2 + S_2(\text{child}_1(H)) + S_2(\text{child}_2(H))$ otherwise. Therefore, we have $S_p = T_{0,1,1}$, where $T_{0,1,1}$ is defined as in Lemma 4.1.7. Consequently, since $\partial G = \emptyset$, the lemma follows by the bound of Lemma 4.1.7. □

Lemma 4.1.13. *Let $H \in \mathcal{T}_{G'}$. Then, $|V(H)| = O(|V_0(H)|)$.*

Proof. Note that $V(H)$ is equal to the union of initial vertex sets of the leaves of $\mathcal{T}_{G'}(H)$. However, by Lemma 4.1.9, there are $O(|V_0(H)|)$ leaves in $\mathcal{T}_{G'}(H)$ and each has $O(1)$ initial vertices. Therefore, the union of initial vertex sets of the leaves of $\mathcal{T}_{G'}(H)$ has $O(|V_0(H)|)$ vertices and $|V(H)| = O(|V_0(H)|)$. □

Corollary 4.1.14. *Let $n = |V_0(G)|$. Then, the obtained root graph G' has $O(n)$ vertices and edges.*

Proof. Since each leaf of $\mathcal{T}_{G'}$ is additionally simple, it has $O(1)$ edges and hence by Lemma 4.1.13, the entire graph G' has $O(n)$ edges. □

Lemma 4.1.15. *The running time of the decomposition algorithm on an n-vertex simple, connected, and triangulated graph G is $O(n \log n)$.*

Proof. Again, the worst-case running time $T(H)$ of the decomposition algorithm on a piece H can be described as $O(n)$ for a leaf piece H and $O(n) + T(\text{child}_1(H)) + T(\text{child}_2(H))$ otherwise. Hence, by Lemma 4.1.7, we obtain $T(H) = O(|V_0(H)| \log |V_0(H)|)$. Consequently, the running time of the algorithm on a n-vertex graph G is $O(n \log n)$. □

4.2 Structural Properties of Reachability in Plane Digraphs

In this section we present the structural properties of reachability in planar digraphs that we exploit later in this chapter. We show how to efficiently represent reachability information between a set of vertices that, roughly speaking, lie on a constant number of faces.

In fact, our analysis is more general and extends to sets of vertices that lie on a constant number of *separator curves*, which we define below. In the following definition, we consider undirected graphs or directed graphs where edge directions are ignored.

Definition 4.2.1. *Let G be a plane embedded graph. A Jordan curve $\mathcal{C}$ is a* separator curve *of G if and only if one of the following holds:*

1. *each connected component of G lies either weakly inside $\mathcal{C}$ or strictly outside $\mathcal{C}$;*
2. *each connected component of G lies either weakly outside $\mathcal{C}$ or strictly inside $\mathcal{C}$.*

Moreover, for each $uv = e \in E(G)$, the embedding of e either constitutes a contiguous fragment of $\mathcal{C}$ or intersects with $\mathcal{C}$ only at its endpoints u, v.

Fact 4.2.2. *Let f be a cycle bounding some simple face of a plane embedded connected graph G. Then, the closed curve defined by the embedding of f is a separator curve.*

We focus on separator curves instead of faces because for any subgraph H of G, any separator curve of G is also a separator curve of H. However, clearly, a face of G might no longer be a face of H.

Definition 4.2.3. *Let P be some set totally ordered by $\prec$. An* interval *over the set P is a set of the form $[x, y]_P = \{z \in P : x \preceq z \preceq y\}$, where $x, y \in P$ and $x \preceq y$.*

Consider a plane digraph G and let $U = U_1 \cup \ldots \cup U_\ell$, $U \subseteq V$, be a set of vertices of G that lie on $\ell = O(1)$ *pairwise disjoint* separator curves $\mathcal{C}_1, \ldots, \mathcal{C}_\ell$ of G. We have $U_i = U \cap \mathcal{C}_i$, which implies $U_i \cap U_j = \emptyset$ for $i \neq j$. For each considered set U satisfying these properties, we fix *any* (out of possibly many) total order $\prec$ on the elements of U which satisfies the following property. Consider the sequence $S(U)$ of elements of U sorted according to $\prec$. Then, elements of each U_i form a contiguous fragment of $S(U)$ and are sorted in clockwise order (with respect to $\mathcal{C}_i$).

For $X, Y \subseteq U$ we also write $X \prec Y$ if for each $x \in X$, $y \in Y$, we have $x \prec y$. Note that $X \prec Y$ implies $X \cap Y = \emptyset$.

Definition 4.2.4. *Let A be a binary matrix with both its rows and columns indexed with the vertices of U. The rows and columns of A are ordered according to the order $\prec$. We say that A is a* reachability matrix *for U in G if and only if for each $u, v \in U$, $A_{u,v} = 1$ holds if and only if there exists a path $u \to v$ in G.*

Definition 4.2.5. *Let $X, Y \subseteq U$ and let A be the reachability matrix of U. A binary matrix $A^{X,Y}$ with rows indexed with X and columns indexed with Y (ordered according to $\prec$) is called a* reachability submatrix *for X, Y if and only if $A^{X,Y}_{x,y} = A_{x,y}$ for all $x \in X$ and $y \in Y$.*

Definition 4.2.6. *Let $X, Y \subseteq U$. The subset of Y containing those y such that for some $x \in X$, $A^{X,Y}_{x,y} = 1$ is called the set of* active columns *of $A^{X,Y}$ and is denoted by* $\mathrm{act}(A^{X,Y})$.

Definition 4.2.7. *For a set $\mathcal{A} = \{A^{S_1,T_1}, \ldots, A^{S_k,T_k}\}$ of reachability submatrices of A and a row $s \in U$, we define a* row projection $\mathrm{r}\pi_s(\mathcal{A})$ *to be the subset $\{A^{S_j,T_j} \in \mathcal{A} : s \in S_j\}$.*

Similarly, for $t \in U$, we define a column projection $\mathrm{c}\pi_t(\mathcal{A}) = \{A^{S_j,T_j} \in \mathcal{A} : t \in T_j\}$.

The following lemma provides a decomposition of the reachability matrix $A = A^U$ used in the next sections in a black-box manner.

Lemma 4.2.8. *Let $M = |U|$. The reachability matrix A of U can be partitioned into a set $\mathcal{A} = \{A^{S_1,T_1}, \ldots, A^{S_k,T_k}\}$ of reachability submatrices such that:*

1. *for each $s,t \in U$, $s \neq t$, there exists exactly one such $A^{S_j,T_j} \in \mathcal{A}$ that $s \in S_j$ and $t \in T_j$,*
2. *for each $A^{S_j,T_j} \in \mathcal{A}$ and $\overline{T} = \mathrm{act}(A^{S_j,T_j})$, the ones in each row of $A^{S_j,\overline{T}}$ form $O(1)$ blocks,*
3. *for any $s \in U$ and $t \in U$, the sets $\mathrm{r}\pi_s(\mathcal{A})$ and $\mathrm{c}\pi_t(\mathcal{A})$ have size $O(\log M)$.*

The sets S_j and T_j that define the partition depend only on the sets $U_1, \ldots, U_t$ and the total order $\prec$. The partition $\mathcal{A}$ can be computed in $O(M^2)$ time.

Observe that the fact that the partition of A in Lemma 4.2.8 only depends on the sets $U_1, \ldots, U_t$ and the order $\prec$ actually means that $\mathcal{A}$ computed for G preserves its properties if we replace G with any subgraph $H \subseteq G$, $V(H) = V$. This is justified by the fact that each $\mathcal{C}_i$ is also a separator curve of H.

We will also need the following extension of Lemma 4.2.8 which captures how the sets of active columns of individual rows of the matrices of the partition $\mathcal{A}$ when G is subject to edge deletions.

Lemma 4.2.9. *Let G, U, A and $\mathcal{A}$ be as in Lemma 4.2.8. Suppose G undergoes edge deletions. Fix some $A^{S,T} \in \mathcal{A}$ and let $\overline{T} = \mathrm{act}(A^{S,T})$. Then, for any $s \in S$, the ones in row s of $A^{S,\overline{T}}$ can be* covered *by $O(1)$ blocks,* which only shrink in time.

Formally, let $G = G_0 \supseteq G_1 \supseteq \ldots \supseteq G_t$ be the subsequent versions of G undergoing edge deletions. For $i = 0, \ldots, t$, denote by A_i the matrix A and by $\overline{T}_i$ the set $\mathrm{act}(A^{S,T})$ when $G = G_i$.

Then for each i and $s \in S$, the subset of columns $C^i_s \subseteq \overline{T}_i$ containing ones in row s of $A_i^{S,\overline{T}_i}$ can be covered by $k_s = O(1)$ (possibly empty) sets $\mathcal{J}^i_s = J^i_1, \ldots, J^i_{k_s}$ such that $J^i_1 \cup \ldots \cup J^i_{k_s} = C^i_s$, and:

- *Each J^i_j is either empty or is of the form $[x^i_j, y^i_j]_{\overline{T}_i}$.*
- *For any $i > 0$ and all $j = 1, \ldots, k_s$, $J^i_j \subseteq J^{i-1}_j$, i.e., either J^i_j is empty or $x^{i-1}_j \preceq x^i_j$ and $y^i_j \preceq y^{i-1}_j$.*

The remaining part of this section is devoted to proving Lemmas 4.2.8 and 4.2.9.

Definition 4.2.10. *The reachability submatrix $A^{S,T}$ is called* bipartite *if S and T lie on a single separator curve (i.e., $S, T \subseteq U_i$ for some $i \in \{1, \ldots, \ell\}$) and either $S \prec T$ or $T \prec S$.*

Lemma 4.2.11 (Monge property)**.** *Let $A^{S,T}$ be a bipartite reachability submatrix of U. Let $a, b \in S$ and $c, d \in T$ be such that $a \prec b$ and $c \prec d$. Suppose $A^{S,T}_{a,c} = 1$ and $A^{S,T}_{b,d} = 1$. Then $A^{S,T}_{a,d} = 1$ and $A^{S,T}_{b,c} = 1$ also hold.*

Proof. We have $S, T \subseteq U_i$ for some i. Without loss of generality, assume that $S \prec T$ and that all connected components of G lie either weakly inside of $\mathcal{C}_i$ or strictly outside $\mathcal{C}_i$. Since a, b, c, d lie on $\mathcal{C}_i$, the connected components of G containing any of these vertices are all weakly inside $\mathcal{C}_i$. As $A^{S,T}_{a,c} = 1$, there exists a simple path $P = a \to c$ contained entirely weakly inside $\mathcal{C}_i$. Similarly, there exists a path $Q = b \to d$ contained weakly inside $\mathcal{C}_i$. By planarity of G, we conclude that the paths P and Q must have a common vertex w (see Figure 4.1). Note that w is reachable from both a and b. Analogously, both c and d are reachable from w. Thus, there also exist paths $a \to d$ and $b \to d$ in G. □

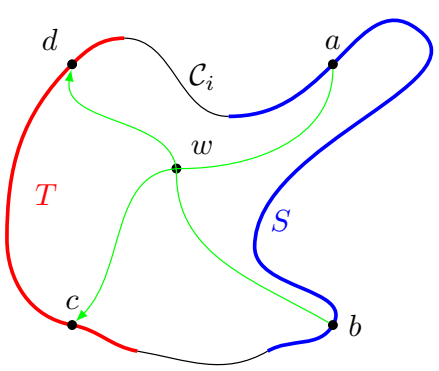

Figure 4.1

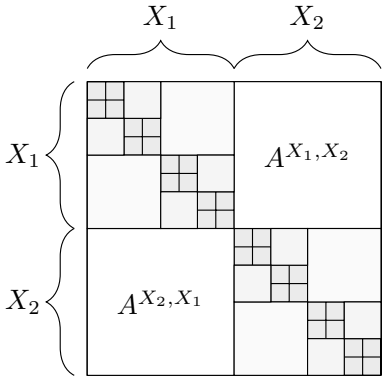

Figure 4.2

Lemma 4.2.12. *Let $A^{S,T}$ be a bipartite reachability submatrix of U. Let $\overline{T} = \mathrm{act}(A^{S,T})$. Let $s \in S$ be some row of $A^{S,\overline{T}}$ containing at least one non-zero entry. Then, the columns $C_s \subseteq \overline{T}$ containing ones in row s of $A^{S,\overline{T}}$ form a single interval over $\overline{T}$, i.e., $C = [l_s, r_s]_{\overline{T}}$, for some $l_s, r_s \in \overline{T}$, $l_s \preceq r_s$.*

This property is preserved when G undergoes edge deletions and the interval $[l_s, r_s]_{\overline{T}}$ shrinks in time.

Proof. Suppose there exist $a, b, c \in \overline{T}$ such that $a \prec b \prec c$ and $A^{S,\overline{T}}_{s,a} = 1$, $A^{S,\overline{T}}_{s,b} = 0$ and $A^{S,\overline{T}}_{s,c} = 1$. By the definition of $\overline{T}$, there exists some other row $s' \in S, s' \neq s$, such that $A^{S,\overline{T}}_{s',b} = 1$. Without loss of generality let us assume $s \prec s'$ (the case when $s' \prec s$ is symmetric). Then we have both $A^{S,\overline{T}}_{s,a} = 1$ and $A^{S,\overline{T}}_{s',b} = 1$. By Lemma 4.2.11, this implies $A^{S,\overline{T}}_{s,b} = 1$, a contradiction.

Clearly, the above holds for every subgraph of G so when G undergoes deletions, the ones in row s form a single interval over $\overline{T}$ until the row s becomes all-zero. Since the ones in $A^{S,T}$ only disappear in time, the interval after a deletion has to be contained in the interval before the deletion. □

Lemma 4.2.13. *Let $m = |U_i|$. The reachability submatrix A^{U_i} can be partitioned into a set $\mathcal{A}^{U_i} = \{A^{S_1,T_1}, \ldots, A^{S_k,T_k}\}$ of bipartite reachability submatrices such that:*

1. *for each $s, t \in U_i$, $s \neq t$, there exists exactly one such $A^{S_j,T_j} \in \mathcal{A}^{U_i}$ that $s \in S_j$ and $t \in T_j$,*
2. *for any $s \in U_i$ and $t \in U_i$, the sets $\mathrm{r}\pi_s(\mathcal{A}^{U_i})$ and $\mathrm{c}\pi_t(\mathcal{A}^{U_i})$ have size $O(\log m)$.*

The sets S_j and T_j that define the partition do not depend on the entries of A^{U_i}. The partition $\mathcal{A}^{U_i}$ can be computed in $O(m^2)$ time.

Proof. We give a recursive procedure to construct the partition $\mathcal{A}^X$, for $X \subseteq U_i$. If $|X| = 1$, the procedure exits immediately with $\mathcal{A}^X = \{A^X\}$. Otherwise, let $X = \{x_1, \ldots, x_k\}$, where $x_1 \prec \ldots \prec x_k$ and $k \geq 2$. Let $q = \lfloor (k+1)/2 \rfloor$. Set $X_1 = \{x_1, \ldots, x_q\}$ and $X_2 = \{x_{q+1}, \ldots, x_k\}$. Note that $X_1 \prec X_2$. Thus, both A^{X_1,X_2} and A^{X_2,X_1} are bipartite reachability submatrices. We add these matrices to the partition and recurse on the subsets X_1 and X_2 (see Figure 4.2).

Note that for any $s, t \in U_i$, $s \neq t$, the last recursive call with both s and t in the input set X places the entries $A^{U_i}_{s,t}$ and $A^{U_i}_{t,s}$ in the bipartite reachability submatrices A^{X_1,X_2} and A^{X_2,X_1}, respectively.

Fix $x \in X$ and assume $|X| = k$. Let $f(k)$ be the number of bipartite reachability submatrices that are produced by the recursive algorithm and contain a row (column) corresponding to x. We have $f(k) \leq f(\lceil k/2 \rceil) + 1$ and thus it is easy to see that $f(k) = O(\log k)$. Thus, we conclude that for any s, $|\mathrm{r}\pi_s(\mathcal{A}^{U_i})| = O(\log m)$ and similarly $|\mathrm{c}\pi_s(\mathcal{A}^{U_i})| = O(\log m)$.

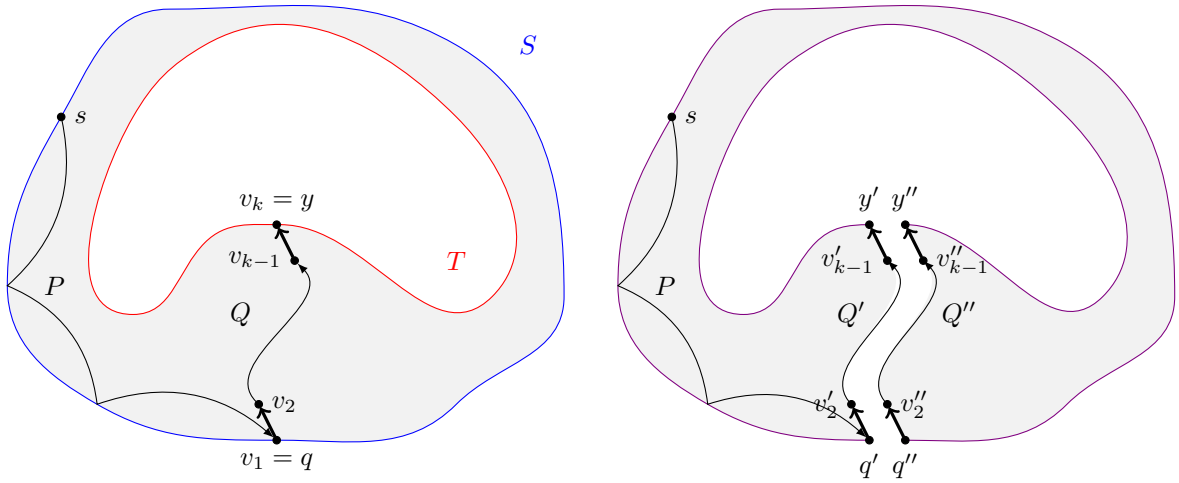

Figure 4.3

The recursive procedure runs in time that is proportional to the total size of matrices that are produced. Recall that for each $s, t \in U_i$, $s \neq t$, there exists exactly one such $A^{S_j,T_j} \in \mathcal{A}^{U_i}$ that $s \in S_j$ and $t \in T_j$. It follows that the total running time is $O(m^2)$. □

An analogue of Lemma 4.2.12 can be also shown for reachability submatrices A^{U_i,U_j}, where $i \neq j$.

Lemma 4.2.14. *Let $A^{S,T}$ be a reachability submatrix for the sets $S = U_i, T = U_j$, where $i \neq j$. Let $\overline{T} = \mathrm{act}(A^{S,T})$. For each row s of $A^{S,\overline{T}}$, the ones in that row can be covered by $O(1)$ intervals over $\overline{T}$ that shrink over time when G undergoes edge deletions.*

Proof. Let s be some row of $A^{S,T}$ containing at least one non-zero entry. Equivalently, there exists a path $s \longrightarrow y$ in G for some $y \in T$. Without loss of generality, assume that all connected components of G lie either weakly inside $\mathcal{C}_i$ or strictly outside $\mathcal{C}_i$. As $s \in \mathcal{C}_i$ and y is in the same connected component of G as s, y lies weakly inside $\mathcal{C}_i$. However, $y \in \mathcal{C}_j$ and $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset$, so entire $\mathcal{C}_j$ lies strictly inside $\mathcal{C}_i$. In particular, all vertices of T lie strictly inside $\mathcal{C}_i$.

Recall that G undergoes edge deletions and let $G_0, G_1, \ldots$ be its versions over time. Let G_l be the last version such that the row s in $A^{S,T}$ contains at least one non-zero entry.

Let $P = s \longrightarrow y$ be the shortest directed path from s to a vertex of T in G_l. Note that the path P exists in all graphs $G = G_0, \ldots, G_l$ for which the row s in $A^{S,T}$ is not all-zero and P does not exist in G_{l+1}. Clearly, P is simple (otherwise it would not be shortest in G_l) and y is the only vertex of $V(P) \cap T$ (again, otherwise it would not be the shortest).

Let us again focus on G. Let q be the last vertex of P such that $q \in S$ and denote by Q the directed subpath $q = v_1 \ldots v_k = y$ of P. By the definitions of a separator curve and path P, Q is weakly inside $\mathcal{C}_i$, weakly outside $\mathcal{C}_j$, and $V(Q) \cap \mathcal{C}_i = \{q\}$, $V(Q) \cap \mathcal{C}_j = \{y\}$.

Let G' be obtained by "cutting" G along the path Q as follows. We split each vertex v_i of Q into two vertices v'_i and v''_i that inherit the edges of v_i emanating strictly left or right of Q, respectively. The created vertices are connected with directed paths $Q' = v'_1 \ldots v'_k$ and $Q'' = v''_1 \ldots v''_k$ (see Figure 4.3). Let $U^* = (S \cup T) \setminus Q \cup Q' \cup Q''$. Note that all the vertices of U^* lie on a single separator curve $\mathcal{C}^*$ of G'. This curve can be used to impose some clockwise order $\prec_s$ on U^*. Let us pick the order so that $(S \setminus Q) \prec_s Q'' \prec_s (T \setminus Q) \prec_s Q'$.

First we prove that any $t \in T \setminus \{y\}$ is reachable from q in G if and only if t is reachable from either q' or q'' in G'. The "if" part is trivial. Suppose there is a path $R = q \longrightarrow t$ in G

and let z be the last vertex that R and Q have in common. Note that z is not the last vertex of R (by $V(Q) \cap T = \{y\}$) and let the edge going out of z emanate left of R (without loss of generality, the case when that edge emanates right of R is symmetric). The subpath $z \longrightarrow t$ of R has only its starting point in $V(Q)$ and thus corresponds to a path $z' \longrightarrow t$ in G'. Therefore, there exists a path $q' \to z' \to t$ in G'.

Next we prove that if $s \neq q$, then any $t \in T \setminus \{y\}$ is reachable from s in G if and only if t is reachable from either q' or q'' or s in G'. To see the "if" part, note that t is reachable from q' or q'' in G' if and only if it is reachable from q in G, whereas q is reachable from s in G. For the "only if" part, suppose there is a path $s \longrightarrow t$ in G. Either this path intersects with Q and then t is reachable from q in G (which, equivalently, means that t is reachable from q' or q'' in G'), or the path does not intersect with Q and is thus preserved in G'.

Now let $S^* = S \setminus \{q\} \cup \{q', q''\}$ and $T^* = T \setminus \{y\}$. Observe that the sets S^*, T^* constitute contiguous fragments of the cycle containing the elements of U^* on $\mathcal{C}^*$ and $S^* \prec_s T^*$. Consider the bipartite reachability submatrix A^{S^*,T^*} in the graph G'. We show that $\mathrm{act}(A^{S,T}) = \mathrm{act}(A^{S^*,T^*}) \cup \{y\}$. Clearly, $\mathrm{act}(A^{S^*,T^*}) \cup \{y\} \subseteq \mathrm{act}(A^{S,T})$. Let $x \in \mathrm{act}(A^{S,T})$. There exists a path $S \longrightarrow x$ in G, where $x \in T \setminus \{y\}$. If this path does not intersect with Q, then it also exists in G'. Otherwise, x is reachable from q in G and thus is reachable from either q' or q'' in G', so clearly $x \in \mathrm{act}(A^{S^*,T^*})$.

By Lemma 4.2.12, the ones in each row of $A^{S^*,\overline{T}^*}$ (with columns ordered with $\prec_s$), where $\overline{T}^* = \mathrm{act}(A^{S^*,T^*})$ form at most one interval over $\overline{T}^*$. Moreover, this interval of ones only shrinks in time in versions $G_0, \ldots, G_l$.

Now let B be the matrix $A^{S^*,\overline{T}^*}$ but with columns ordered according to the original order $\prec$ imposed on U. Since the order $\prec_s$ restricted to T^* is just a cyclical shift of the order $\prec$ restricted to $T \setminus \{y\}$, and thus the order $\prec_s$ restricted to $\overline{T}^*$ is a cyclical shift of the order $\prec$ restricted to $\overline{T} \setminus \{y\}$, we conclude that for each row of B the ones in that row:

- either form a single interval over $\overline{T}^*$ ordered by $\prec$, and this interval also shrinks in time in versions $G_0, \ldots, G_l$,
- or they form two intervals over $\overline{T}^*$ ordered by $\prec$ such that one of these intervals starts with the first column of $\overline{T}^*$ and the other ends with the last column of $\overline{T}^*$. These two intervals also only shrink in time in versions $G_0, \ldots, G_l$.

Observe that the set of ones in the row s of $A^{S,\overline{T}}$ can be seen as the union of:

- a single one in the column y (since we have $y \in \overline{T}$),
- the set of ones in the row q' of B,
- the set of ones in the row q'' of B,
- if $s \neq q$, the set of ones in the row s of B.

Each of the above contributes at most two intervals over $\overline{T} \setminus \{y\}$ that only shrink over time. These intervals, if interpreted as intervals over $\overline{T}$, might contain the column y. This is not a problem though because in fact y was chosen in such a way that the one in column y of row s is there in all the versions $G_0, \ldots, G_l$ for which s can reach some vertex of T.

In total, the ones in the row s of $A^{S,T}$ can be covered by $O(1)$ intervals over $\overline{T}$ that shrink in time. □

Remark 4.2.15. *Note that in the above proof of Lemma 4.2.14, for each row s we have shown the existence of the interval cover that shrinks over time non-constructively as we referred to some future version of the graph. However, this will turn out to be sufficient for our needs.*

Proof of Lemmas 4.2.8 and 4.2.9. We first partition A into ℓ^2 reachability submatrices A^{U_i,U_j}, for all $i,j \in \{1,\ldots,\ell\}$. Each A^{U_i,U_i} is then partitioned using Lemma 4.2.13, whereas each A^{U_i,U_j}, for $i \neq j$, is included in $\mathcal{A}$ without further partitioning. As each $u \in U$ belongs to exactly one U_i, $|\mathrm{r}\pi_u(\mathcal{A})| = |\mathrm{c}\pi_u(\mathcal{A})| = O(\log|U_i|) + \ell - 1 = O(\log m)$. By Lemmas 4.2.12 and 4.2.14, for all $A^{S,T} \in \mathcal{A}$, the ones in each row of $A^{S,\overline{T}}$, where $\overline{T} = \mathrm{act}(A^{S,T})$, can be covered with $O(1)$ intervals over $\overline{T}$ that only shrink in time. The running time of this procedure is clearly $O\left(m^2 + \sum_i |U_i|^2\right) = O(m^2)$. □

4.3 Partially-Dynamic Monge Transitive Closure

Let G be a plane embedded digraph and let $U = U_1 \cup \ldots U_\ell \subseteq V(G)$ be a set of vertices lying on $O(1)$ *fixed*, pairwise disjoint separator curves $\mathcal{C}_1,\ldots,\mathcal{C}_\ell$ of G, as was the case in Section 4.2.

We say that the graph G is *decremental* if it is subject to edge deletions. We say that G is *U-incremental* if it is subject to edge insertions but only such that they respect the embedding of G and do not break the property that each $\mathcal{C}_i$ is a separator curve of G.

In this section we consider the following problem. For $i = 1,2$, let G_i be a plane digraph such that $U_i \subseteq V(G_i)$ lies on $O(1)$ fixed, pairwise disjoint separator curves of G_i. Suppose that either both G_1 and G_2 are decremental, or G_1 is U_1-incremental and G_2 is U_2-incremental. Our goal is to maintaining the transitive closure of the union of $G_1^+[U_1]$ and $G_2^+[U_2]$ subject to batches of decremental (incremental) updates to these graphs resulting from deleting (inserting, respectively) edges of G_1 and G_2. Efficient algorithm for these partially-dynamic problems will turn out to be the key to obtaining fast decremental reachability algorithms for planar graphs.

The remainder of this section is devoted to proving the following two theorems.

Theorem 4.3.1. *For $i = 1,2$. let G_i be a plane digraph that is U_i-incremental. Let $M = |U_1 \cup U_2| = \mathrm{poly}(n)$. Then, we can maintain the transitive closure $(G_1^+[U_1] \cup G_2^+[U_2])^+$ subject to batches of incremental updates to the transitive closures $G_1^+[U_1]$ and $G_2^+[U_2]$ (resulting from inserting edges to G_1 and G_2, respectively) in $O(M^2 \log n \log\log n)$ total time.*

Theorem 4.3.2. *For $i = 1,2$, let G_i be a plane digraph that is decremental. Let $M = |U_1 \cup U_2| = \mathrm{poly}(n)$. Then, we can maintain the transitive closure $(G_1^+[U_1] \cup G_2^+[U_2])^+$ subject to batches of decremental updates to the transitive closures $G_1^+[U_1]$ and $G_2^+[U_2]$ (resulting from deleting edges from G_1 and G_2, respectively) in $O(M^2 \log^4 n)$ total time. The algorithm is Monte Carlo randomized and is correct with probability at least $1 - n^{-d}$, for any constant $d \geq 1$.*

4.3.1 Auxiliary Data Structures for Reachability Matrices

Similarly as in Section 4.2, let G be a plane digraph and let $U \subseteq V(G)$ lie on a set of $O(1)$ disjoint separator curves of G. Let $\prec$ be the order imposed on U, as described in Section 4.2. Let A be the reachability matrix of U in G and $M = |U|$. Let $\mathcal{A}$ be the partition of A of Lemma 4.2.8.

Suppose G is either decremental or U-incremental. We would like to maintain the following auxiliary data structures for each $A^{S,T} \in \mathcal{A}$:

- The set $\overline{T} = \mathrm{act}(A^{S,T})$.
- For all $s \in S$, the sets $\mathrm{Out}_s(A^{S,T}) = \{x : A^{S,T}_{s,x} = 1\}$ along with its representation $\mathrm{Ival}_s(A^{S,T})$ as $O(1)$ *disjoint* and *non-empty* intervals over $\overline{T}$. Such a representation consists of k disjoint intervals $[x_1,y_1]_{\overline{T}},\ldots,[x_k,y_k]_{\overline{T}}$, such that $\mathrm{Out}_s(A^{S,T}) = \bigcup_{i=1}^k [x_i,y_i]_{\overline{T}}$. Here, the number k might change in time, but is always bounded by a constant.

Lemma 4.3.3. *Suppose G is either decremental or U-incremental and after each change that results in a change to the reachability matrix A we are given the updates to A accordingly.*

Then, in $O(M^2 \log n)$ total time, we can maintain the above auxiliary data structures for all elements $A^{S,T}$ of the partition $\mathcal{A}$.

Additionally, if G is decremental*, then for all $s \in S$, each edge deletion issued to G can only split or shrink the intervals of $\mathrm{Ival}_s(A^{S,T})$, but never merge them. Specifically, at any moment of time, if we denote by $\mathrm{Ival}_s(A^{S,T})$ the representation after an edge deletion, and by $\mathrm{Ival}'_s(A^{S,T})$ the representation before that deletion, then for any $J \in \mathrm{Ival}_s(A^{S,T})$, there is exactly one $J' \in \mathrm{Ival}'_s(A^{S,T})$ such that $J \subseteq J'$ (here, J' is assumed to be an interval over $\overline{T}'$, where $\overline{T}' = \mathrm{act}(A^{S,T})$ before the deletion).*

Proof. By Lemma 4.2.8, we can compute $\mathcal{A}$ in $O(M^2)$ time. Observe that by Lemma 4.2.8, this partition is independent of the values in the cells of A so it does not need to be updated.

Note that the values in the cells of A can either only go from 1 to 0, or only from 0 to 1, so clearly there are at most $O(M^2)$ updates to A throughout. Whenever a cell of A changes, we update the corresponding cell in some $A^{S,T}$. Again by Lemma 4.2.8, for each cell of A there is exactly one $A^{S,T} \in \mathcal{A}$ that contains that cell. While updating the matrices, it is also easy to update the set $\mathrm{act}(A^{S,T})$ for each $A^{S,T} \in \mathcal{A}$. We store $\mathrm{act}(A^{S,T})$ in a balanced binary search tree ordered by $\prec$.

For each row $s \in A^{S,T}$, we store the set $\mathrm{Out}_s(A^{S,T})$ in a balanced binary tree, again ordered by $\prec$. This set can be updated in logarithmic time when some cell in this row changes its value. In total, maintaining these sets takes $O(M^2 \log n)$ time.

Let $\overline{T} = \mathrm{act}(A^{S,T})$. We initialize each set $\mathrm{Ival}_s(A^{S,T})$ with a unique *minimum-size* set of disjoint intervals $[x, y]_{\overline{T}}$ such that $\bigcup \mathrm{Ival}_s(A^{S,T}) = \mathrm{Out}_s(A^{S,T})$. By Lemma 4.2.8 and minimality, initially $\mathrm{Ival}_s(A^{S,T})$ has $O(1)$ intervals.

In the following we describe how the set $\mathrm{Ival}_s(A^{S,T})$ is updated when some value in row s of $A^{S,T}$ changes. Define $\mathrm{Ival}'_s(A^{S,T})$, $\mathrm{Out}'_s(A^{S,T})$ and $\overline{T}'$ to be the respective sets *before* the deletion. We maintain the invariant that $|\mathrm{Ival}'_s(A^{S,T})| = O(1)$. Clearly, the invariant is satisfied before the first update to the row s of A. We then construct the new representation $\mathrm{Ival}_s(A^{S,T})$ as follows.

First consider the case when G is U-incremental. Then clearly $\overline{T}' \subseteq \overline{T}$ and $\mathrm{Out}'_s(A^{S,T}) \subseteq \mathrm{Out}_s(A^{S,T})$. For each $[x_i, y_i]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T})$, we construct a minimum set I_i of $O(|\overline{T} \setminus \overline{T}'|)$ intervals over $\overline{T}$ such that $\bigcup I_i = [x_i, y_i]_{\overline{T}} \setminus (\overline{T} \setminus \overline{T}')$. Let

$$I^*_s = \left(\bigcup \left\{ I_i : [x_i, y_i]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T}) \right\}\right) \cup \left\{ [z, z]_{\overline{T}} : z \in \mathrm{Out}_s(A^{S,T}) \setminus \mathrm{Out}'_s(A^{S,T}) \right\}.$$

Note that since $(\overline{T} \setminus \overline{T}') \cap \mathrm{Out}'_s(A^{S,T}) = \emptyset$, $\bigcup \{I_i : [x_i, y_i]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T})\} = \mathrm{Out}'_s(A^{S,T})$, and therefore $\bigcup I^*_s = \mathrm{Out}_s(A^{S,T})$ and the elements of I^*_s are disjoint. Since $|\mathrm{Ival}'_s(A^{S,T})| = O(1)$, we have

$$|I^*_s| = |\mathrm{Ival}'_s(A^{S,T})| \cdot |\overline{T} \setminus \overline{T}'| + |\mathrm{Out}_s(A^{S,T}) \setminus \mathrm{Out}'_s(A^{S,T})| = O(|\overline{T} \setminus \overline{T}'|) + |\mathrm{Out}_s(A^{S,T}) \setminus \mathrm{Out}'_s(A^{S,T})|.$$

Observe that by first sorting the intervals I^*_s, we can easily obtain a minimal set $\mathrm{Ival}_s(A^{S,T})$ of intervals over $\overline{T}$, such that $\bigcup \mathrm{Ival}_s(A^{S,T}) = I^*_s = \mathrm{Out}_s(A^{S,T})$ in $O(|I^*_s| \log n)$ time. By the minimality of $\mathrm{Ival}_s(A^{S,T})$ and Lemma 4.2.8, we conclude $|\mathrm{Ival}_s(A^{S,T})| = O(1)$.

To bound the total time needed to update the sets $\mathrm{Ival}_s(A^{S,T})$ over all updates when G is U-incremental, note that the sum of terms $|\overline{T} \setminus \overline{T}'|$ over all updates is at most $|T|$. Similarly, the sum of terms $|\mathrm{Out}_s(A^{S,T}) \setminus \mathrm{Out}'_s(A^{S,T})|$ over all updates is no more than $|T|$. Hence, the total time needed to maintain the sets $\mathrm{Ival}_s(A^{S,T})$ is $O(|T| \log n)$. Summing these total time

costs over all $A^{S,T} \in \mathcal{A}$ and $s \in S$, we obtain the bound $O(M^2 \log n)$ as the total number of cells in all matrices $A^{S,T} \in \mathcal{A}$ is $O(M^2)$.

Let us now focus on the case when G is decremental. This case is a bit more involved because we pursue stronger properties of how the sets $\mathrm{Ival}_s(A^{S,T})$ change. For any $[x,y]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T})$ we just split $[x,y]_{\overline{T}'}$ into a minimum number of intervals $[a,b]_{\overline{T}}$, where $[a,b]_{\overline{T}} \subseteq [x,y]_{\overline{T}'}$, such that $[a,b]_{\overline{T}} \subseteq \mathrm{Out}_s(A^{S,T})$. Observe that splitting of each interval can be done in time nearly linear in time number of elements removed from $\mathrm{Out}'_s(A^{S,T})$ after the edge deletion. Namely, the endpoints of new intervals that should replace $[a,b]_{\overline{T}}$ can be found by issuing certain predecessor/successor queries to the BST storing the already-up-to-date set $\mathrm{Out}_s(A^{S,T})$ in $O\left(|\mathrm{Out}'_s(A^{S,T}) \setminus \mathrm{Out}_s(A^{S,T})| \log n\right)$ time.

Clearly, from the algorithm computing $\mathrm{Ival}_s(A^{S,T})$ it follows that the intervals shrink or split but never merge. We only need to prove that the size of $\mathrm{Ival}_s(A^{S,T})$ is $O(1)$ at all times. Note that since the total number of entries in all matrices $A^{S,T}$ is M^2, it will follow that the total time needed to maintain these sets is $O(M^2 \log n)$.

By Lemma 4.2.9, there exists a collection $\mathcal{J}_s$ of $k_s = O(1)$ "shrinking" intervals $[x_1,y_1]_{\overline{T}}, \ldots, [x_k,y_k]_{\overline{T}}$ that *cover* $\mathrm{Out}_s(A^{S,T})$, i.e., $\bigcup \mathcal{J}_s = \mathrm{Out}_s(A^{S,T})$, when time passes. However, we do not know how this collection looks like exactly and how it behaves when G undergoes deletions. What we can prove, however, is that at all times each of these shrinking intervals $[x_i,y_i]_{\overline{T}}$ is contained in exactly one $[a,b]_{\overline{T}} \in \mathrm{Ival}_s(A^{S,T})$ and each $[a,b]_{\overline{T}} \in \mathrm{Ival}_s(A^{S,T})$ contains some $[x_i,y_i]_{\overline{T}} \in \mathcal{J}_s$. Observe that a simple consequence of this fact is that $|\mathrm{Ival}_s(A^{S,T})| \leq |\mathcal{J}_s| = O(1)$.

We prove this property inductively. Obviously, it holds at the beginning as we initialize $\mathrm{Ival}_s(A^{S,T})$ with a minimal set of disjoint intervals whose union is $\mathrm{Out}_s(A^{S,T})$. Suppose some edge deletion happens. Let $\mathcal{J}'_s$ be the collection of Lemma 4.2.9 before the deletion, and $\mathcal{J}_s$ – after the deletion. By the induction hypothesis, the collection $\mathcal{J}'_s$ can be partitioned into disjoint classes $\mathcal{J}'_{[c,d]}$, containing intervals of $\mathcal{J}'_s$ that were contained in each $[c,d]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T})$. Let $[a,b]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T})$. Suppose our algorithm breaks $[a,b]_{\overline{T}'}$ into intervals $[a_1,b_1]_{\overline{T}}, \ldots, [a_q,b_q]_{\overline{T}}$ after the deletion. By the minimality of this partition, after the deletion, for all i, $1 \leq i < q$, there exists $z \in \overline{T}$, $b_i \prec z \prec a_{i+1}$ such that $z \notin \mathrm{Out}_s(A^{S,T})$. Clearly, for all $[x,y]_{\overline{T}} \in \mathcal{J}_{[a,b]}$, we have either $z \prec x$ or $y \prec z$. Here, $\mathcal{J}_{[a,b]}$ are the shrunk versions (recall Lemma 4.2.9) of intervals $\mathcal{J}'_{[a,b]}$. Therefore, each $[x,y]_{\overline{T}} \in \mathcal{J}_{[a,b]}$ intersects with at most one interval $[a_i,b_i]_{\overline{T}}$ and by the definition of $\mathcal{J}'_{[a,b]}$, $[x,y]_{\overline{T}'} \subseteq [a,b]_{\overline{T}'}$. However, recall that $\bigcup \mathcal{J}_s = \bigcup \mathrm{Ival}_s(A^{S,T}) = \mathrm{Out}_s(A^{S,T})$ so in fact each $[x,y]_{\overline{T}} \in \mathcal{J}_{[a,b]}$ *is contained* in exactly one interval $[a_i,b_i]_{\overline{T}}$. On the other hand, since both $\mathcal{J}_s$ and $\mathrm{Ival}_s(A^{S,T})$ cover $\mathrm{Out}_s(A^{S,T})$, and $\mathrm{Ival}_s(A^{S,T})$ is minimal, for each $[a_i,b_i]_{\overline{T}}$, some $[x,y]_{\overline{T}} \in \mathcal{J}_{[a,b]}$ is contained in $[a_i,b_i]_{\overline{T}}$. □

Remark 4.3.4. *Assume the notation from Lemma 4.3.3. We will often need to access both* $\mathrm{Ival}_s(A^{S,T})$ *(the intervals after) and* $\mathrm{Ival}'_s(A^{S,T})$ *(before) to capture the difference between these two sets. We further assume that both these sets are maintained: when a new update comes,* $\mathrm{Ival}'_s(A^{S,T})$ *is set to* $\mathrm{Ival}_s(A^{S,T})$*, and* $\mathrm{Ival}_s(A^{S,T})$ *is computed as described in the proof of Lemma 4.3.3.*

Remark 4.3.5. *Assume the notation from Lemma 4.3.3. Recall that if G is decremental, the intervals of* $\mathrm{Ival}_s(A^{S,T})$ *only shrink or split. When an interval* $[x,y]_{\overline{T}'} \in \mathrm{Ival}'_s(A^{S,T})$ *shrinks, we assume it maintains its identity in* $\mathrm{Ival}_s(A^{S,T})$*. When an interval* $[x,y]_{\overline{T}'}$ *splits into* $k > 1$ *intervals* $[x_1,y_1]_{\overline{T}}, \ldots, [x_k,y_k]_{\overline{T}}$*, we assume that* $[x_1,y_1]$ *takes over the identity of* $[x,y]_{\overline{T}'}$ *and the intervals* $[x_2,y_2]_{\overline{T}}, \ldots, [x_k,y_k]_{\overline{T}}$ *are* created*. This way, for a fixed s, only $O(1)$ intervals are ever created in* $\mathrm{Ival}_s(A^{S,T})$*.*

Proof. Note that $|\mathrm{Ival}_s(A^{S,T})|$ is equal to $|\mathrm{Ival}'_s(A^{S,T})|$ plus the number of intervals created, minus the number of intervals of $\mathrm{Ival}'_s(A^{S,T})$ that shrink to an empty interval. Observe that an interval of $\mathcal{J}'_s$ can only shrink to an empty interval when some of the intervals of $\mathcal{J}'_s$ becomes empty in $\mathcal{J}_s$. This might only happen $O(1)$ times. However, in the proof of Lemma 4.3.3 we have $|\mathrm{Ival}_s(A^{S,T})| \leq |\mathcal{J}_s| = O(1)$ and $|\mathcal{J}_s|$ is constant in time. Therefore, a new interval can be created in $\mathcal{J}_s$ only $O(1)$ times. □

4.3.2 Fast Breadth-First Search in a Union of Reachability Matrices

In this section we show how Lemma 4.2.8, along with the data structures of Section 4.3.1, can be used to find paths in a graph that, roughly-speaking, is a union of several reachability matrices that satisfy the assumptions of Lemma 4.2.8. This algorithm resembles FR-Dijkstra [27] (see also Chapter 5) and will prove very useful in the following sections of this chapter.

Lemma 4.3.6. *For $i = 1, \ldots, p$, let G_i be a plane digraph with vertices V_i. Let U_i be a subset of V_i lying on a constant number of pairwise disjoint separator curves of G_i.*

Suppose that for each i we are given the reachability matrix A_i of U_i in G_i, along with its partition $\mathcal{A}_i$ of Lemma 4.2.8, and the auxiliary data of Lemma 4.3.3.

Define $G = \bigcup_{i=1}^{p} G_i^+[U_i]$ and let $|V(G)| = O(\mathrm{poly}(n))$. Then, for any $s \in V(G)$, we can perform a breadth-first search from s (i.e., compute the distances $\delta_G(s, v)$, for all $v \in V$) in $O(|V(G)| \log n \log\log n) = O\left((\sum_{i=1}^{p} |U_i|) \log n \log\log n\right)$ time.

Proof. We maintain the set $W \subseteq V(G)$ of vertices discovered as reachable from s during the breadth-first search. Initially, $W = \{s\}$. Moreover, for each $v \in V(G)$, we maintain the value $D(v)$ which will eventually store the distance from s to v. Initially, $D(s) = 0$ and $D(v) = \infty$ for $v \neq s$.

The algorithm repeatedly chooses an unprocessed vertex $v \in W$ in fifo-order, and computes the set $N_G^{\mathrm{out}}(v) \setminus W$ containing exactly the vertices reachable from s through v by a single edge that have not been yet discovered. Then, for each $x \in N_G^{\mathrm{out}}(v) \setminus W$, x is inserted into W and we set $D(x) = D(v) + 1$.

It is clear that the entire computation, excluding the computation of sets $N_G^{\mathrm{out}}(v) \setminus W$, takes $O(|V(G)|)$ time. To compute the sets $N_G^{\mathrm{out}}(v) \setminus W$ efficiently, we proceed as follows. Recall that for each graph $G_i^+[U_i]$ that comprises G, we maintain the partition $\mathcal{A}_i$ of A_i of Lemma 4.2.8, and some auxiliary components from Lemma 4.3.3. For all i, and each $A^{S,T} \in \mathcal{A}_i$, our algorithm initializes and maintains the set $\mathrm{act}(A^{S,T}) \setminus W$ in a dynamic predecessor data structure [93]. The total size of these sets is $O(\sum_i^p |U_i| \log |U_i|) = O(|V(G)| \log n)$, by Lemma 4.2.8.

First, any $\mathrm{act}(A^{S,T}) \setminus W$ is possibly updated when $\mathrm{act}(A^{S,T})$ shrinks. Such updates take $O\left(\sum_i^p |U_i| \log n \log\log n\right)$ time in total. When some new vertex v is inserted into W, for all i such that $v \in U_i$ we go through all of $O(\log U_i)$ matrices $A^{S,T} \in \mathrm{c}\pi_v(\mathcal{A}_i)$ and remove v from $\mathrm{act}(A^{S,T}) \setminus W$. Such updates also take $O\left(\sum_i^p |U_i| \log n \log\log n\right)$ time in total since each vertex of U_i is inserted into W at most once. Now, observe that $N_G^{\mathrm{out}}(v) \setminus W$ can be expressed as

$$\bigcup_{\substack{U_i : v \in U_i \\ A^{S,T} \in \mathrm{r}\pi_v(\mathcal{A}_i)}} \left(\mathrm{Out}_v(A^{S,T}) \setminus W\right),$$

whereas for any $v \in U_i$ and $A^{S,T} \in \mathrm{r}\pi_v(\mathcal{A}_i)$ we have

$$\mathrm{Out}_v(A^{S,T}) \setminus W = (\mathrm{act}(A^{S,T}) \setminus W) \cap \mathrm{Ival}_v(A^{S,T}).$$

Thus, to compute $N_G^{\mathrm{out}}(v) \setminus W$, it is enough to intersect $O(q \log n)$ sets $\mathrm{act}(A^{S,T}) \setminus W$ with $O(1)$ intervals over $\mathrm{act}(A^{S,T})$ where $q \leq p$ is the number of sets U_i such that $v \in U_i$.

As the sets $\mathrm{act}(A^{S,T}) \setminus W$ are stored as dynamic predecessor data structures, the elements of each intersection can be reported in $O(\log\log n)$ time per element. Hence, this step can be implemented in $O((|N_G^{\mathrm{out}}(v) \setminus W| + q)\log n \log\log n)$ time. In other words, we spend additional $O(q \log n \log\log n)$ time to process v and $O(\log n \log\log n)$ time per each vertex inserted to W. Summing over all vertices v, the total time spent in this step during the search is $O\left(\sum_i^p |U_i| \log n \log\log n\right)$.

We conclude that the whole procedure takes $O\left(\sum_{i=1}^p |U_i| \log n \log\log n\right)$ time. □

Corollary 4.3.7. *For $i = 1, \ldots, p$, let G_i, V_i, U_i, A_i and $\mathcal{A}_i$ be as in Lemma 4.3.6. Let H be some (not necessarily planar) directed graph.*

Define $G = H \cup \bigcup_{i=1}^p G_i^+[U_i]$ and let $|V(G)| = O(\mathrm{poly}(n))$ and $|E(H)| = O(\mathrm{poly}(n))$. Then, we can compute distances from s to all vertices of G in $O\left((|V(G) + |E(H)|)\log n \log\log n\right)$ time.

Proof. Note that each edge $uv = e_j \in E(H)$ can be treated as two-vertex graph G_{p+j} with $U_{p+j} = \{u, v\}$. For any embedding of e there clearly exists a Jordan curve going through u and v and not crossing this embedding of e. Hence, U_{p+j} lies on a constant number of disjoint separator curves of G_{p+j}. Therefore, we can apply Lemma 4.3.6. □

4.3.3 Incremental Algorithm

Queue-Based Incremental Transitive Closure Algorithm

We first show and analyze a simple queue-based algorithm for updating the transitive closure of a (general) graph after a set of edges is added. Its pseudocode is given as Algorithm 3. The algorithm should be considered folklore, but for the sake of completeness we describe it in a detailed way as we need its efficient implementation.

The transitive closure algorithm is based on the following idea. Whenever it determines that a vertex b is reachable from a vertex a, it infers that every vertex reachable from b by an edge is also reachable from a and every vertex that has an edge to a can also reach b. Then, it propagates this information using a queue.

Algorithm 3 The queue-based incremental transitive closure.

Require: A digraph G, such that $F \subseteq E(G)$ and the transitive closure $H = (G - F)^+$.
Ensure: $H = G^+$.

```
1:  function UpdateTransitiveClosure(G, H, F)
2:      Q := empty queue
3:      for uv ∈ F do
4:          if uv ∉ E(H) then
5:              E(H) := E(H) ∪ {uv}
6:              Q.Enqueue(uv)
7:      while Q is not empty do
8:          ab := Q.Dequeue
9:          for x ∈ N_G^out(b) \ N_H^out(a) do
10:             E(H) := E(H) ∪ {ax}
11:             Q.Enqueue(ax)
12:         for x ∈ N_G^in(a) \ N_H^in(b) do
13:             E(H) := E(H) ∪ {xb}
14:             Q.Enqueue(xb)
```

Lemma 4.3.8. *Algorithm 3 is correct.*

Proof. First we show that $pq \in E(H)$ only if there is a path $p \to q$ in G. This is satisfied initially by the fact that $H = (G - F)^+$. Note that the algorithm adds uv to H in line 5 if $uv \in E(G)$ and thus there is a path $u \to v$ in G. Hence, before the first iteration of the while loop, $pq \in E(H)$ only if $pq \in E(G^+)$ and Q only contains edges pq such that $pq \in E(G^+)$. Now we show that this condition is satisfied after all subsequent iterations of the while loop. Indeed, the algorithm adds ax to H in line 10 when $ab \in E(G^+)$ (since ab has been removed from Q) and $bx \in E(G)$. Similarly, it adds xb to H in line 13 when $ab \in E(G^+)$ and $xa \in E(G)$. In both cases it is easy to see that if pq is added to H (and to Q) then there exists a path $p \to q$ in G.

It remains to show that if adding F causes q to be reachable from p, then the algorithm adds the edge pq to H (we assume that q was not reachable from p before adding edges F to G). Let P be some $p \to q$ path in G. We say that an edge e of P has a detour avoiding F if a subpath of P containing e can be replaced with a path (detour) that does not contain edges of F and the obtained path still connects p and q. As long as P has an edge of F that has a detour avoiding F we replace its respective subpath with the detour. This process terminates as at each step we reduce the number of edges of F on P. Once the process terminates, P contains some edge uv of F as there is no path from p to q that does not contain edges of F. Moreover, uv does not have a detour avoiding F.

Let $P' = p' \to q'$ be a subpath of P that contains the edge uv. We have that $p'q' \notin E(H) = E((G - F)^+)$ before the algorithm was run as P' could not be replaced with a detour. We now use induction on the length of P' to show that the algorithm adds $p'q'$ to H.

The case when $k = 1$ is easy because the only valid path P' of length 1 is the edge uv itself: uv is added to H in line 5. Let us now consider a path P' of length $k > 1$. Either the first or the last edge of P' is not equal to uv. Without loss of generality, let us assume that it is the last one and denote it by rq'. By the induction hypothesis, the algorithm adds the edge $p'r$ to H. This means that it then adds $p'r$ to the queue. Consider the iteration of the while loop when $p'r$ is removed from the queue, that is $ab = p'r$. Clearly, $q' \in N_G^{\text{out}}(b) = N_G^{\text{out}}(r)$ as rq' is an edge on a path in G. There are two cases to consider. If $q' \in N_H^{\text{out}}(a) = N_H^{\text{out}}(p')$, then we already have $p'q' \in E(H)$. Otherwise, $q' \in N_G^{\text{out}}(b) \setminus N_H^{\text{out}}(a)$ and then $aq' = p'q'$ is added to H in line 10 of some iteration of the for-loop. This completes the proof. □

The following lemma highlights which part of Algorithm 3 can be sped up.

Lemma 4.3.9. *The running time of Algorithm 3, excluding the time needed to compute the sets $N_G^{\text{out}}(b) \setminus N_H^{\text{out}}(a)$ and $N_G^{\text{in}}(a) \setminus N_H^{\text{in}}(b)$, is proportional to the number of edges added to H (or constant, if no edges are added to H).*

Proof. Let k be the total number of edges added to H by the algorithm. Observe that just before the algorithm adds ax to H in line 10, we have $ax \notin E(H)$ as $x \notin N_H^{\text{out}}(a)$. The similar reasoning applies to line 13. Thus, the total running time of the two for loops is proportional to k.

Moreover, the number of elements added to the queue Q is at most k. This implies that the total number of iterations of the while loop is at most k, and the lemma follows. □

Proof of Theorem 4.3.1

For $i = 1, 2$, let us identify the graph $G_i^+[U_i]$ with the reachability submatrix of U_i in G_i. In order to handle each update efficiently, we use Algorithm 3 for a graph $G = G_1^+[U_1] \cup G_2^+[U_2]$ initially equal to $(U_1 \cup U_2, \emptyset)$. Specifically, initially and after each updated we require that $H = (G_1^+[U_1] \cup G_2^+[U_2])^+$. When some G_i is updated and thus we are given a batch F of new edges that have been added to $G_i^+[U_i]$, we run UpdateTransitiveClosure(G, H, F). Moreover, we leverage the special structure of matrices $G_i^+[U_i]$ to reduce the total time spent

on computing the sets $N_G^{\text{out}}(b) \setminus N_H^{\text{out}}(a)$ and $N_G^{\text{in}}(a) \setminus N_H^{\text{in}}(b)$ in lines 9 and 12. In the following we only deal with the former set; in order to compute the latter, one needs to proceed symmetrically.

We start by computing the partition $\mathcal{A}_i = \{A^{S_1,T_1}, \ldots, A^{S_k,T_k}\}$ of Lemma 4.2.8 of the matrix $G_i^+[U_i]$. This is possible since U_i lies on $O(1)$ fixed, pairwise disjoint separator curves of G_i. While $G_i^+[U_i]$ undergoes incremental updates, we also maintain the data structures accompanying the matrices $A^{S,T} \in \mathcal{A}_i$ described in Section 4.3.1. This takes $O(M^2 \log n)$ total time, by Lemma 4.3.3.

For each $a \in U_1 \cup U_2$ and $A^{S,T} \in \mathcal{A}_1 \cup \mathcal{A}_2$ the algorithm stores a *reachability candidates set* $\text{can}_a(A^{S,T}) \subseteq \text{act}(A^{S,T})$. The algorithm maintains the invariant that

$$\text{can}_a(A^{S,T}) = \text{act}(A^{S,T}) \setminus N_H^{\text{out}}(a).$$

Whenever a column $t \in T$ of $A^{S,T}$ becomes active, we check whether $at \in E(H)$ and if not, we insert t into $\text{can}_a(A^{S,T})$. Once we insert at to H, t is removed from $\text{can}_a(A^{S,T})$ (and, clearly, never added again as we never remove edges from H in Algorithm 3). Each $\text{can}_a(A^{S,T})$ is stored as a dynamic predecessor data structure [93].

Lemma 4.3.10. *The sets of reachability candidates can be maintained in $O(M^2 \log n \log\log n)$ total time.*

Proof. Fix $a \in U_1 \cup U_2$. For each $A^{S,T} \in \mathcal{A}_1 \cup \mathcal{A}_2$, each element of T is added to and removed from $\text{can}_a(A^{S,T})$ at most once. If $\mathcal{A}_1 \cup \mathcal{A}_2 = \{A^{S_1,T_1}, \ldots, A^{S_k,T_k}\}$ then the total number of operations is $\sum_{i=1}^k |T_i|$. By Lemma 4.2.8, for each $b \in U_1 \cup U_2$ there are $O(\log M) = O(\log n)$ matrices $A^{S,T} \in \mathcal{A}_1 \cup \mathcal{A}_2$ such that $b \in T$. Thus we make at most $\sum_{i=1}^k |T_i| = O(M \log n)$ operations for a fixed a, which gives $O(M^2 \log n)$ operations in total. Since each operation on a dynamic predecessor data structure takes $O(\log\log n)$ time, we conclude that maintaining reachability candidates sets takes $O(M^2 \log n \log\log n)$ time. □

We are now ready to show how to speed up line 9 of Algorithm 3. The goal is to compute the set $N_G^{\text{out}}(b) \setminus N_H^{\text{out}}(a)$ efficiently. Algorithm 3 traverses this set, once it is computed, but this does not affect the running time considerably. Only the computation of the set could be slow. This means that it suffices to compute the set in time which is, say, almost linear in its size. Our algorithm, roughly speaking, uses the property that $N_G^{\text{out}}(b)$ is represented by a small number of intervals, so computing an intersection with the set $N_G^{\text{out}}(b)$ is easy. Recall that $\text{r}\pi_b(\mathcal{A}_i)$ is the subset of $\mathcal{A}_i$ consisting of matrices which contain the row b.

Lemma 4.3.11. *Let*

$$Q = \bigcup_{i \in \{1,2\}} \bigcup_{A^{S,T} \in \text{r}\pi_b(\mathcal{A}_i)} \left(\text{can}_a(A^{S,T}) \cap \text{Out}_b(A^{S,T})\right).$$

Then $Q = N_G^{\text{out}}(b) \setminus N_H^{\text{out}}(a)$.

Proof. By Lemma 4.2.8 and the definition of the sets $\text{Out}_b(A^{S,T})$, we have

$$\begin{aligned} N_G^{\text{out}}(b) &= \bigcup_{i \in \{1,2\}} \bigcup_{A^{S,T} \in \text{r}\pi_b(\mathcal{A}_i)} \text{Out}_b(A^{S,T}) \\ &= \bigcup_{i \in \{1,2\}} \bigcup_{A^{S,T} \in \text{r}\pi_b(\mathcal{A}_i)} \text{act}(A^{S,T}) \cap \text{Out}_b(A^{S,T}). \end{aligned}$$

Moreover $\mathrm{can}_a(A^{S,T}) = \mathrm{act}(A^{S,T}) \setminus N_H^{\mathrm{out}}(a)$. Hence

$$\begin{aligned}
Q &= \bigcup_{i\in\{1,2\}} \bigcup_{A^{S,T}\in \mathrm{r}\pi_b(\mathcal{A}_i)} \mathrm{can}_a(A^{S,T}) \cap \mathrm{Out}_b(A^{S,T}) \\
&= \bigcup_{\substack{i\in\{1,2\} \\ A^{S,T}\in \mathrm{r}\pi_b(\mathcal{A}_i)}} (\mathrm{act}(A^{S,T}) \setminus N_H^{\mathrm{out}}(a)) \cap \mathrm{Out}_b(A^{S,T}) \\
&= \left(\bigcup_{\substack{i\in\{1,2\} \\ A^{S,T}\in \mathrm{r}\pi_b(\mathcal{A}_i)}} (\mathrm{act}(A^{S,T}) \cap \mathrm{Out}_b(A^{S,T})) \right) \setminus N_H^{\mathrm{out}}(a) \\
&= N_G^{\mathrm{out}}(b) \setminus N_H^{\mathrm{out}}(a).
\end{aligned}$$

□

Lemma 4.3.12. *$Q = N_G^{\mathrm{out}}(b) \setminus N_H^{\mathrm{out}}(a)$ can be computed in $O(|Q| \log\log n + \log n)$ time.*

Proof. By Lemma 4.3.11, it suffices to compute

$$\bigcup_{i\in\{1,2\}} \bigcup_{A^{S,T}\in \mathrm{r}\pi_b(\mathcal{A}_i)} \mathrm{can}_a(A^{S,T}) \cap \mathrm{Out}_b(A^{S,T}).$$

Since $|\mathrm{r}\pi_b(\mathcal{A}_i)| = O(\log n)$ (by Lemma 4.2.8), this is a sum of $O(\log n)$ sets of the form $\mathrm{can}_a(A^{S,T}) \cap \mathrm{Out}_b(A^{S,T})$. Moreover, these sets are disjoint since for the matrices $A^{S,T}$ such that $A^{S,T} \in \mathrm{r}\pi_b(\mathcal{A}_i)$, the sets T are disjoint.

Let us focus on computing $\mathrm{can}_a(A^{S,T}) \cap \mathrm{Out}_b(A^{S,T})$. Recall that $\mathrm{Out}_b(A^{S,T}) = \mathrm{Ival}_b(A^{S,T})$, where $\mathrm{Ival}_b(A^{S,T})$ is a set of $O(1)$ disjoint intervals over $\mathrm{act}(A^{S,T})$.

However, we also have $\mathrm{can}_a(A^{S,T}) \subseteq \mathrm{act}(A^{S,T})$. As a result, to compute the intersection it suffices to take elements of $\mathrm{can}_a(A^{S,T})$ that are contained in the intervals describing $\mathrm{Ival}_b(A^{S,T})$. Since $\mathrm{can}_a(A^{S,T})$ is represented as dynamic predecessor data structure, finding a single element of $\mathrm{can}_a(A^{S,T})$ contained in a given interval can be done in $O(\log\log n)$ time. Thus, we spend $O(\log\log n)$ time on computing each element of Q plus $O(\log n)$ time to consider $O(\log n)$ sets that comprise the sum. □

Proof of Theorem 4.3.1. By Lemmas 4.3.3 and 4.3.10 the total cost of computing and updating the auxiliary data structures of Section 4.3.1 for graphs $G_i^+[U_i]$ and the reachability candidates is $O(M^2 \log n \log\log n)$. By Lemma 4.3.9, the total running time of Algorithm 3 excluding the cost of lines 9 and 12 is $O(M^2)$. By Lemma 4.3.12, it takes $O(q \log\log n + \log n)$ to execute each of these lines, assuming that they compute a set of size q. Since each such set is then traversed by the algorithm, the $O(q \log\log n + \log n) = O((q+1)\log n)$ overhead implies that the transitive closure algorithm runs in $O(M^2 \log n)$ total time. Thus, the overall running time is $O(M^2 \log n \log\log n)$. □

4.3.4 Decremental Algorithm

Our strategy will be again to adjust an algorithm developed for general graphs to our needs. We base our approach on the decremental $(1+\epsilon)$-approximate all-pairs shortest paths algorithm of Bernstein [4]. Even though this algorithm is designed for a more general problem of computing shortest paths, its simplified version – which, as we will see in this section, basically follows from replacing all usages of the so-called h-SSSP data structure (also due to Bernstein [4]) with so-called Even-Shiloach tree (see Theorem 4.3.13 below) – in fact solves the decremental transitive closure problem in $\widetilde{O}(nm)$ total time.

Theorem 4.3.13 (Even-Shiloach tree [26, 46, 61])**.** *Given an unweighted directed graph G and a source $s \in V$, we can maintain the set of vertices of G at distance no more than d from s subject to edge deletions in $O(md)$ total time.*

The advantage of the simplified algorithm of Bernstein [4], which we review below, over other known $\tilde{O}(mn)$ total update time decremental transitive closure algorithms [66, 83] is that it essentially has a single bottleneck which is the efficiency of the Even-Shiloach tree. Thus, giving a faster implementation of the Even-Shiloach algorithm for our graph of special form, almost immediately yields a faster decremental algorithm maintaining $(G_1^+[U_1] \cup G_2^+[U_2])^+$. This property of Bernstein's algorithm, however, comes at a cost of using Monte-Carlo randomization and assuming that the adversary is oblivious to the random choices made internally by the data structure.

The Decremental Transitive Closure Algorithm of Bernstein [4].

Our description uses some parts of Section 6 of [4]. Some key claims about the transitive closure algorithm of Bernstein are are only stated here. For full proofs consult [4].

In this section we will also $\overrightarrow{\text{es}}(G, s, d)$ ($\overleftarrow{\text{es}}(G, s, d)$) to denote the set of vertices v of G at distance no more than d from s (such that s is at distance no more than d from v, respectively), but emphasizing that it is maintained using the data structure of Theorem 4.3.13 (used on the reverse graph of G, respectively).

Assume for simplicity that n is a power of 2 and let $q = \log_2 n$. Let $A_0 = V$. For each $i = 1, \ldots, q$, construct A_i from A_{i-1} by picking half of vertices of A_{i-1} uniformly at random. This way, A_i is a random subset of $\frac{n}{2^i}$ vertices, and in particular, $|A_q| = 1$.

Let d be the constant of Theorem 4.3.2 and set $c = \lceil d + 5 \rceil$. The algorithm sets up various graphs depending on each other (without cyclic dependencies) and updates them accordingly after each edge deletion so that they still satisfy their definitions.

In the following, denote by $E(\overrightarrow{\text{es}}(H, v, d))$ the set $\{vy : y \in \overrightarrow{\text{es}}(H, v, d)\}$, and similarly let $E(\overleftarrow{\text{es}}(H, v, d)) = \{xv : x \in \overleftarrow{\text{es}}(H, v, d)\}$. Set $A_{q+1} = \emptyset$ and $D_{q+1} = \emptyset$. There are $q + 1$ *steps* numbered 0 through q.

For each step $k = q$ down to 0, we do the following:

- For each $v \in A_k$, let

$$G_{v,k} = (V, E(G) \cup \{vy : y \in A_{k+1}, vy \in D_{k+1}\} \cup \{xv : x \in A_{k+1}, xv \in D_{k+1}\}).$$

- For each $v \in A_k$, maintain $\overrightarrow{\text{es}}(G_{v,k}, v, d_k)$ and $\overleftarrow{\text{es}}(G_{v,k}, v, d_k)$ for $d_k = \min(n, \lceil c \cdot 2^{k+1} \ln n \rceil)$. Using this, maintain an edge set

$$D_k = \bigcup_{v \in A_k} E(\overrightarrow{\text{es}}(G_{v,k}, v, d_k)) \cup E(\overleftarrow{\text{es}}(G_{v,k}, v, d_k)).$$

Suppose an edge e is removed from G. The update procedure is very simple: we only need to make all the maintained graphs, edge sets, and Even-Shiloach trees again satisfy their definition, which is done as follows. Deleting an edge of G and issuing this deletion to relevant Even-Shiloach trees may shrink the sets $\overrightarrow{\text{es}}(G_{v,q} = G, v, n)$ and $\overleftarrow{\text{es}}(G_{v,q} = G, v, n)$ for the unique element $v \in A_q$. That, in turn, can cause an edge deletion from D_q which clearly influences the graphs $G_{v,q-1}$, where $v \in A_{q_1}$, which in turn influences the edge sets $E(\overrightarrow{\text{es}}(G_{v,q-1}, v, d))$ comprising D_{q-1}, and so on.

To check whether there exists a path $u \to v$ in G, it is sufficient to check whether $uv \in D_0$, i.e., D_0 contains exactly the edges of G^+, as proved below.

Theorem 4.3.14 ([4]). *With probability at least $1 - n^{-d}$, the above algorithm maintains the edges D_0 of the transitive closure G^+ subject to edge deletions issued to G. The total update time is $O(mn \log^2 n)$.*

Proof sketch. The proof starts by noting a well-known fact (Lemma 4.4 in [4]), that if P is any fixed simple path with at least $\min(n, \lceil cn \ln n/r \rceil)$ vertices, then the set S of r vertices of V chosen uniformly at random contains at least one vertex of S with probability at least $1 - n^{-c}$.

One can first observe that if $uv \in D_k$ (for any $k = q, \ldots, 0$), then there exists a $u \to v$ path in G. Suppose without loss of generality that $u \in A_k$. Then indeed, uv is put into D_k if there exists a $u \to v$ path in $G_{u,k}$ and such a path is either entirely contained in G, or (if $k < q$) is a concatenation of a single edge uu' of D_{k+1}, where $u' \in A_{k+1}$, with some $u' \to v$ path from G. Since $uu' \in D_{k+1}$ certifies that there exists a $u \to u'$ path in G, we can apply induction.

Subsequently, one can prove inductively that for any moment of time t (recall that there are at most $m \leq n^2$ edge deletions, hence $t \leq n^2$), for any $k = q, \ldots, 0$, and for any $(u, v) \in (A_k \times V) \cup (V \times A_k)$, if some path $P(t, u, v) = u \to v$ exists in G, then $uv \in D_k$ with probability at least $1 - (q-k)n^{-c}$. In particular, since $A_0 = V$, by the union bound it follows that D_0 is the edge set of the transitive closure of G at any time with probability at least $1 - n^2 \cdot n^2 \cdot qn^{-c} \geq 1 - n^{-c+5} \geq 1 - n^{-d}$.

The claim clearly holds for $k = q$ since for $k = q$ we maintain the sets of all the vertices reachable from v and all vertices that can reach v where v is the unique element of A_q.

Let $k < q$ and suppose the claim holds for $k' = k + 1$. We focus on proving the inductive step in the case $(u, v) \in A_k \times V$ since the case $(u, v) \in V \times A_k$ is completely analogous.

If $P(t, u, v)$ has no more than $d_k = \min(n, \lceil c \cdot 2^{k+1} \ln n \rceil)$ vertices, then clearly there exists a $u \to v$ path of length less than d_k in $G_{u,k}$ since $G \subseteq G_{u,k}$. Suppose that $P(t, u, v)$ has more than d_k vertices and consider the subpath $P'(t, u, v) = u' \to v$ of $P(t, u, v)$ consisting of its d_k last vertices. Since A_{k+1} is completely random from the point of view of the path $P'(t, u, v)$, with probability at least $1 - n^{-c}$, $P'(t, u, v)$ contains a vertex of $v' \in A_{k+1}$. It follows that there exists a path $u \to v'$ in G. Hence, by the inductive assumption, $uv' \in D_{k+1}$ with probability at least $1 - (q-k-1)n^{-c}$. Let $P''(t, u, v)$ be the $v' \to v$ subpath of $P'(t, u, v)$. Then $uv' \in E(G_{u,k})$ and $v' \to v = P''(t, u, v) \subseteq G \subseteq G_{u,k}$, so we conclude that there exists a path $u \to v$ of length at most d_k in $G_{u,k}$ with probability at least $1 - (q - k)n^{-c}$. By the definition of D_k, it follows that $uv \in D_k$ with the same probability.

To bound the total update time of this algorithm, first note that for each $k = q, \ldots, 0$, we maintain $O(n/2^k)$ Even-Shiloach trees on graphs with $m + O(n)$ edges up to depth $d_k = O(2^k \log n)$. Hence, by Theorem 4.3.13, the total update time for a fixed k is $O(mn \log n)$. Since there are $O(\log n)$ different values of k, we conclude that the total update time is $O(mn \log^2 n)$. □

A Fast Even-Shiloach Tree Implementation.

This section is devoted to showing how to speed-up the decremental transitive closure algorithm of Theorem 4.3.14 for a special case that fits our applications.

Theorem 4.3.15. *Let G_1, U_1, G_2, U_2 and M be as in Theorem 4.3.2. Let $U = U_1 \cup U_2$. Let $s_1, \ldots, s_p$ be some vertices of U and let $F_1, \ldots, F_p \subseteq U \times U$ be sets of edges such that initially $|F_i| = O(M)$. Let $d \leq M$ be a positive integer.*

Suppose G_1 and G_2 undergo edge deletions and we are given a batch of updates to $G_1^+[U_1]$ or $G_2^+[U_2]$ resulting from each deletion. Also assume each set F_i is subject to deletions. For $i = 1, \ldots, p$, let

$$H_i = G_1^+[U_1] \cup G_2^+[U_2] \cup (U, F_i).$$

Then, we can explicitly maintain the sets $\mathcal{B}_i = \{v \in U : \delta_{H_i}(v, s_i) \leq d\}$, *for all* $i = 1, \ldots, p$, *in* $O((M^2 + Mpd) \log^2 n)$ *total time.*

For $j = 1, 2$, we identify each $G_j^+[U_j]$ with its adjacency matrix, which is in fact the reachability matrix of U_j in G_j. Note that G_j and U_j satisfy the properties required by Lemma 4.2.8. We first compute the partition $\mathcal{A}_j$ of $G_j^+[U_j]$ in $O(M^2)$ time. We also maintain the auxiliary information of Lemma 4.3.3 about the elements of $\mathcal{A}_j$ subject to updates to $G_j^+[U_j]$. This takes $O(M^2 \log n)$ time in total.

Observe that for each $i = 1, \ldots, p$, the graph H_i is subject to an edge deletion whenever an edge is deleted from either $G_1^+[U_1]$, $G_2^+[U_2]$, or F_i.

For each $i = 1, \ldots, p$ and $u \in U$, we explicitly maintain the value $D_i(u)$ called a *level* of u in H_i. At all times (even during the update procedure) we will guarantee that:

(1) $D_i(u) \in \{0, 1, \ldots, d, \infty\}$,

(2) $D_i(u)$ never decreases,

(3) for any $uv \in E(H_i)$, $D_i(u) \leq D_i(v) + 1$.

Observe that this way each value $D_i(u)$ can change at most $d+1$ times. Hence, the total number of times any level $D_i(u)$ changes is $O(Mpd)$.

The Even-Shiloach algorithm. We now refer to the Even-Shiloach algorithm [26, 61], which constitutes the base of our algorithm. Suppose we run the Even-Shiloach algorithm for each $i = 1, \ldots, p$. For a single source s_i, this algorithm also maintains the levels $D_i(v)$ that satisfy the properties (1)-(3) of how they change.

After the initialization (which involves computing shortest paths to s_i) and after processing each update to $E(H_i)$, the Even-Shiloach algorithm guarantees that:

- $D_i(u) = \delta_{H_i}(u, s_i)$ if $\delta_{H_i}(u, s_i) \leq d$,
- $D_i(u) = \infty$ otherwise.

We call a vertex v *good* with respect to i if and only if either $v = s_i$ and $D_i(s_i) = 0$, $D_i(v) = \infty$, or there exists an edge $vx \in E(H_i)$ such that $D_i(v) = D_i(x) + 1$. It is easy to see that when the values $D_i(v)$ constitute correct distances from s_i, then all vertices are good.

The Even-Shiloach update procedure can be summarized as follows. Suppose the set $E(H_i)$ is updated first. While there exists some vertex v that is not good:

- choose some vertex v that is not good and has the smallest level $D_i(v)$,
- increase the level $D_i(v)$ by 1 (or set $D_i(v) = \infty$ if $D_i(v)$ was already equal to d).

For a proof that after this update procedure ends the levels are the same as distances to s_i, see [26, 61].

Suppose that the information which vertices are good is known to us and is updated accordingly. Then, the remaining part of the Even-Shiloach algorithm can be implemented as follows. Store the vertices that are not good in a priority queue keyed by their levels. Since each step of the update procedure involves increasing some level, the total update time for a fixed i can be seen to be $O(|V(H_i)| \cdot d \log n) = O(Md \log n)$, for a total of $O(Mpd \log n)$ over all graphs $H_1, \ldots, H_p$.

The remainder of this section is devoted to showing how to maintain the vertices that are not good with respect to all valid indices i in $O((M^2 + Mpd) \log^2 n)$ total time subject to the levels change issued by the Even-Shiloach algorithm. From this, Theorem 4.3.15 will follow.

Data structure components. In this paragraph we describe the components our decremental algorithm stores. First, we need the following data-structure which can be implemented quite easily using the known dynamic orthogonal range searching data structures (e.g., [73]).

Lemma 4.3.16. *Let Z be some set of size z and let $\prec$ be a total order on Z. There exists a data structure for maintaining a set Q of elements keyed with intervals over Z, supporting the following operations.*

1. *Insert an element q with $\mathrm{key}(q) = I$ to Q.*
2. *Delete some element q of Q.*
3. *Query 1: let each of A, B be either $\emptyset$ or an interval over Z and suppose $A \subseteq B$. Report all elements $q \in Q$ satisfying $A \subseteq \mathrm{key}(q)$ and $B \not\subseteq \mathrm{key}(q)$.*
4. *Query 2: let each of A, B be either $\emptyset$ or an interval over Z and suppose $A \subseteq B$. Report all elements $q \in Q$ satisfying $\mathrm{key}(q) \not\subseteq A$ and $\mathrm{key}(q) \subseteq B$.*

The insertions and deletions take $O(\log z)$ time, whereas the query takes $O(k + \log z)$ time, where k is the number of reported elements.

Proof. We can identify the set Z with integers $\{1, \ldots, z\}$. Thus, we can in fact work with intervals over $\{1, \ldots, z\}$.

We map any interval $[a, b]$ over $\{1, \ldots, z\}$ to a point $p([a, b]) = (z + 1 - a, b)$ of the plane. This way, for two intervals $[a, b]$ and $[c, d]$, $[a, b]$ contains $[c, d]$ if and only if $a \leq c$ and $b \geq d$, which is equivalent to $z + 1 - a \geq z + 1 - c$ and $b \geq d$, i.e., $p([a, b])$ *dominates* $p([c, d])$ in the sense that it has both coordinates not smaller than the respective coordinates of $p([c, d])$.

Hence, the keys of all the elements of Q are mapped to a point set P on the plane. Let $p(q) = A = (x, y)$ be some query point. Then:

- All points of P that A dominates lie in a single orthogonal rectangle, namely $[0, x] \times [0, y]$.
- Hence, all points that A *does not dominate* lie in the sum of two disjoint rectangles $[x + 1, z] \times [0, z]$ and $[0, x] \times [y + 1, z]$.
- Similarly, all points that dominate A lie in a single rectangle $[x, z] \times [y, z]$.
- All points that do not dominate A lie in the union of two disjoint orthogonal rectangles.

Note that both required queries ask for points of P lying in the intersection of two regions of one of the above forms. Hence, both queries can be reduced to reporting points of $O(1)$ disjoint rectangles.

To finish the proof, we apply the existing fully dynamic two-dimensional orthogonal range reporting data structure [73]. This data structure has $O(\log z)$ update time and $O(\log z + k)$ query time, where k is the number of reported points. □

Set $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. The following components are all inductively defined in terms of values $D_i(u)$, the matrices of $\mathcal{A}$ along with the accompanying data structures of Lemma 4.3.3, and the previously defined components. The dependencies between these components in fact form a DAG: once some component changes its state, it automatically causes updates to data structures that depend on it so that every component complies its definition. Hence, when describing the individual components, at the same time we also explain how they are updated.

- For each $i = 1, \ldots, p$, $j = 0, \ldots, d$, set $B_{i,j} = \{v \in V(H_i) : D_i(v) \leq j\}$. We explicitly maintain all sets $B_{i,j}$: these can be easily updated whenever some value $D_i(u)$ changes.

Recall that the values $D_i(u)$ only increase, and in fact each increases at most $d+1$ times. As a result, each $B_{i,j}$ *only shrinks* and the total time used to keep all sets $B_{i,j}$ updated is $O(Mpd)$.

- For all $i = 1, \ldots, p$, $j = 0, \ldots, d$, and $A^{S,T} \in \mathcal{A}$, we maintain the set $B^{S,T}_{i,j} = \mathrm{act}(A^{S,T}) \cap B_{i,j}$. Recall that by Lemma 4.2.8, for any $u \in U$, $|\mathrm{c}\pi_u(\mathcal{A})| = O(\log n)$. Hence, each u can be included in at most $O(pd \log n)$ sets $B^{S,T}_{i,j}$. It follows that the total size of all the sets $B^{S,T}_{i,j}$ is $O(Mpd \log n)$. Clearly, these sets can be initialized in $O(Mpd \log n)$ time. Also, they can be easily updated whenever some set $\mathrm{act}(A^{S,T})$ shrinks (which happens $O(M \log n)$ times in total), or when some $B_{i,j}$ shrinks ($O(Mpd)$ times in total). Hence, all sets $B^{S,T}_{i,j}$ can be maintained in $O(Mpd \log n)$ total time.

- For all $i = 1, \ldots, p$, $j = 0, \ldots, d$, and $A^{S,T} \in \mathcal{A}$, we maintain the set $Q^{S,T}_{i,j}$ containing the maximal intervals $[a_i, b_i]_T$ such that for all $x \in T$ satisfying $a_i \preceq x \preceq b_i$ we have $x \in T \setminus B^{S,T}_{i,j}$. As $B^{S,T}_{i,j}$ only shrinks in time, $\bigcup Q^{S,T}_{i,j}$ only grows. We store each $Q^{S,T}_{i,j}$ in a balanced binary search tree ordered by left endpoints. These sets can be easily computed in $O(Mpd \log^2 n)$ time.

  We now explain how $Q^{S,T}_{i,j}$ is updated when an element x is deleted from $B^{S,T}_{i,j}$. Namely, at most two intervals of $Q^{S,T}_{i,j}$ (with endpoints neighboring x in T) are removed, and one new interval – containing the removed intervals and x – is inserted. Hence, the total time spent on maintaining all the sets $Q^{S,T}_{i,j}$ can be charged to the number of changes to the sets $B^{S,T}_{i,j}$ times $O(\log n)$ and therefore is again $O(Mpd \log^2 n)$.

- For each $A^{S,T} \in \mathcal{A}$, we maintain a data structure $L^{S,T}$ of Lemma 4.3.16. For each valid i, j, and $[a_k, b_k]_T \in Q^{S,T}_{i,j}$, $L^{S,T}$ contains an element (i, j) keyed with $[a_k, b_k]_T$. As discussed above, the number of times an element is inserted or removed into any of sets $Q^{S,T}_{i,j}$ is $O(Mpd \log n)$. Thus, the data structures $L^{S,T}$ can also be initialized and maintained in $O(Mpd \log^2 n)$ total time because by Lemma 4.3.16, insertions and deletions to $L^{S,T}$ cost $O(\log n)$ time.

- For each $A^{S,T} \in \mathcal{A}$, we maintain a data structure $P^{S,T}$ of Lemma 4.3.16. Let $\overline{T} = \mathrm{act}(A^{S,T})$. For each $s \in S$, and $[x_k, y_k]_{\overline{T}} \in \mathrm{Ival}_s(A^{S,T})$, there is an element $(s, [x_k, y_k]_{\overline{T}})$ keyed with $[x_k, y_k]_T$ in $P^{S,T}$ at all times. Recall that by Lemma 4.3.3, the sets $\mathrm{Ival}_s(A^{S,T})$ are stored explicitly and undergo $O(M^2)$ changes in total. Thus, the data structures $P^{S,T}$ can also be initialized and maintained in $O(M^2 \log n)$ total time because by Lemma 4.3.16, insertions and deletions to $P^{S,T}$ cost $O(\log n)$ time.

- For each $i = 1, \ldots, p$, $j = 0, \ldots, d$, and $A^{S,T} \in \mathcal{A}$, let $\overline{T} = \mathrm{act}(A^{S,T})$. For all $s \in S$ and $[x_k, y_k]_{\overline{T}} \in \mathrm{Ival}_s(A^{S,T})$, we maintain the *cover bit* $\phi_{i,j}([x_k, y_k]_{\overline{T}})$ equal to 1 if and only if $[x_k, y_k]_T \subseteq \bigcup Q^{S,T}_{i,j}$ (i.e., if and only if for some $[a, b]_T \in Q^{S,T}_{i,j}$, $[x_k, y_k]_T \subseteq [a, b]_T$) and 0 otherwise. Note that as the sets $\bigcup Q^{S,T}_{i,j}$ only grow and the intervals $[x_k, y_k]_{\overline{T}}$ only shrink (once created, see Lemma 4.3.3), the cover bits (once initialized) can only go from 0 to 1.

Now we discuss how the cover bits are maintained. A cover bit can change for two reasons.

Updating cover bits after $\mathrm{Ival}_s(A^{S,T})$ changes. Let $\overline{T}' = \mathrm{act}(A^{S,T}), \overline{T}$ and $\mathrm{Ival}'_s(A^{S,T})$, $\mathrm{Ival}_s(A^{S,T})$ be the corresponding sets before and after the deletion of a batch of edges from $G_1^+[U_1] \cup G_2^+[U_2]$ (see Remark 4.3.4). First, if the interval $[x, y]_{\overline{T}'}$ is not deleted from $\mathrm{Ival}'_s(A^{S,T})$, it is split into $k \geq 1$ intervals $[x_1, y_1]_{\overline{T}}, \ldots, [x_k, y_k]_{\overline{T}}$ in $\mathrm{Ival}_s(A^{S,T})$. Recall that by Remark 4.3.5,

we assume that $[x_1, y_1]_{\overline{T}}$ inherits the identity of $[x, y]_{\overline{T}'}$. By that we also mean that it inherits all the cover bits of $[x, y]_{\overline{T}'}$. Each of the intervals $[x_2, y_2]_{\overline{T}}, \ldots, [x_k, y_k]_{\overline{T}}$ is *created* and we initialize all the cover bits of these intervals to be equal to the cover bits of $[x, y]_{\overline{T}'}$. As by Remark 4.3.5, no more than $O(1)$ in intervals are ever created in a single set $\mathrm{Ival}_s(A^{S,T})$, $O(Mpd \log n)$ cover bits are ever initialized. Observe that at this point the cover bits of all $[x_l, y_l]_{\overline{T}}$ $(l = 1, \ldots, k)$ may be incorrect (are equal to 0 but should be set to 1) as they were set to the cover bits of a larger interval $[x, y]_{\overline{T}'}$. We fix them for each $[x_l, y_l]_{\overline{T}}$ as follows.

Suppose we knew the set X of pairs (i, j) for which the cover bit $\phi_{i,j}([x_l, y_l]_{\overline{T}})$ should be switched from 0 to 1. Then, by going through $(a, b) \in X$ and just setting $\phi_{a,b}([x_l, y_l]_{\overline{T}}) = 1$, we would spend $O(Mpd)$ time in total as each of $O(Mpd)$ cover bits that is ever initialized would be switched from 0 to 1 at most once. Hence, we would like to find the set X.

Let Y be the set of elements $(i, j) \in L^{S,T}$ such that (i, j) is included in the set Y if and only if $[x_l, y_l]_T \subseteq \mathrm{key}((i, j))$ and $[x, y]_T \not\subseteq \mathrm{key}((i, j))$. Observe that in $L^{S,T}$ there can be multiple elements (i, j), but all of them have disjoint keys. Thus, by the definition of Y, at most one can be included in Y.

We now prove that $X = Y$. Take some $(i, j) \in Y$ and let $\mathrm{key}((i, j)) = [f, g]_T$. Then, $[f, g]_T \in Q_{i,j}^{S,T}$. Hence, $[x_l, y_l]_T \subseteq [f, g]_T$ and thus $\phi_{i,j}^{S,T}([x_l, y_l]_{\overline{T}})$ should be set to 1. We also have $[x, y]_T \not\subseteq [f, g]_T$. Note that $\emptyset \neq [x_l, y_l]_T \subseteq [f, g]_T \cap [x, y]_T$, so by the maximality of $[f, g]_T$ in $Q_{i,j}^{S,T}$ we conclude that there exists some $z \in T$ such that $z \in [x, y]_T \setminus \bigcup Q_{i,j}^{S,T}$. Therefore, $[x, y]_T \not\subseteq \bigcup Q_{i,j}^{S,T}$, i.e., $\phi_{i,j}^{S,T}([x, y]_{\overline{T}'})$ was indeed set to 0, so $(i, j) \in X$.

We have proved $Y \subseteq X$. Now let $(i, j) \in X$. After the deletion $\phi_{i,j}([x_l, y_l]_{\overline{T}})$ should be set to 1 if and only if $[x_l, y_l]_{\overline{T}} \subseteq \bigcup Q_{i,j}^{S,T}$. Equivalently, as each $Q_{i,j}^{S,T}$ is a disjoint union of intervals over T, there exists a *unique* $[c_1, d_1]_T \subseteq Q_{i,j}^{S,T}$ such that $[x_l, y_l]_T \subseteq [c_1, d_1]_T$. Recall that since the cover bits were correct before the update, $\phi_{i,j}([x, y]_{\overline{T}'}) = 0$ if and only if $[x, y]_{\overline{T}'} \not\subseteq \bigcup Q_{i,j}^{S,T}$, and in particular $[x, y]_T \not\subseteq [c_1, d_1]_T$. As $[c_1, d_1]_T \in Q_{i,j}^{S,T}$, there is an element (i, j) with key $[c_1, d_1]_T$ in $L^{S,T}$. Moreover, we have $[x_l, y_l]_T \subseteq [c_1, d_1]_T$ and $[x, y]_T \not\subseteq [c_1, d_1]_T$ which proves $(i, j) \in Y$.

It remains to note that by Lemma 4.3.16, we can find $Y = X$ in $O(|X| + \log n)$ time. Hence, by the previous discussion, computation of all sets X throughout any sequence of edge deletions takes $O(\sum |X| + M^2 \log n) = O((Mpd + M^2) \log n)$ time in total.

Updating cover bits when the sets $Q_{i,j}^{S,T}$ change. Suppose some set $Q_{i,j}^{S,T}$ undergoes a change. Let again $\overline{T} = \mathrm{act}(A^{S,T})$. Recall that an elementary change on $Q_{i,j}^{S,T}$ involves inserting to $Q_{i,j}^{S,T}$ one new interval $[a, b]_T$ and deleting from $Q_{i,j}^{S,T}$ l $(0 \leq l \leq 2)$ intervals $[a_1, b_1]_T, \ldots, [a_l, b_l]_T$ where $[a_i, b_i]_T$ are pairwise disjoint and for each i, $[a_i, b_i]_T \subseteq [a, b]_T$. For simplicity, we only consider the case when $l = 2$. The cases $l = 0, 1$ can be handled by setting the non-existent intervals to $\emptyset$.

We would thus like to find the set X of all pairs $(s, [x_k, y_k]_{\overline{T}})$ such that $s \in S$, $[x_k, y_k]_{\overline{T}} \in \mathrm{Ival}_s(A^{S,T})$, and the cover bit $\phi_{i,j}^{S,T}([x_k, y_k]_{\overline{T}})$ has to be updated from 0 to 1.

For $g = 1, \ldots, l$, let Y_g be the subset of elements $e = (s, [x_k, y_k]_{\overline{T}})$ of $P^{S,T}$ such that $\mathrm{key}(e) \subseteq [a, b]_T$ and $\mathrm{key}(e) \not\subseteq [a_g, b_g]_T$. Let $Y = Y_1 \cup Y_2$. The proof that $X = Y$ proceeds completely analogously to that of the previous paragraph, so we skip it. Again, by Lemma 4.3.16, we find each set Y_i in $O(|Y_i| + \log n) = O(|X| + \log n)$ time. Recall that the sets $Q_{i,j}^{S,T}$ undergo $O(Mpd \log n)$ updates in total. The total time spent on the computation of sets X is hence $O(\sum |X| + Mpd \log^2 n) = O(Mpd \log^2 n)$.

Further components. We now continue describing our data structure.

- For each $i = 1, \ldots, p$, $A^{S,T} \in \mathcal{A}$, let $\overline{T} = \mathrm{act}(A^{S,T})$. For all $s \in S$ and $[x_k, y_k]_{\overline{T}} \in \mathrm{Ival}_s(A^{S,T})$, we also define the *cover rank* $\alpha_i([x_k, y_k]_{\overline{T}})$ to be the minimum j such that $\phi_{i,j}([x_k, y_k]_{\overline{T}}) = 0$, i.e., the minimum j such that $[x_k, y_k]_{\overline{T}}$ is not covered by any interval of $Q_{i,j}^{S,T}$. If no such j exists, we set $\alpha_i([x_k, y_k]_{\overline{T}}) = \infty$.

  Keeping that in mind, the cover ranks can be maintained in total time $O(Mpd \log n)$, which can be shown as follows. For for each $s \in S$, by Lemma 4.3.3, only $O(1)$ intervals are ever created in $\mathrm{Ival}_s(A^{S,T})$. Since s is a row of $O(\log n)$ matrices $A^{S,T}$, there are no more than $O(Mp \log n)$ cover ranks ever initialized. Note that the as the cover bits go only from 0 to 1, the cover ranks can only increase. Each cover rank is thus maintained in $O(d)$ time.

- For each $i = 1, \ldots, p$ and $u \in U$, we maintain the set $Z_i(u) = \{D_i(x) : ux \in F_i\}$ in a balanced binary tree, augmented so that the value $\beta_i(u) = \min Z_i(u)$, called the *edge rank*, is updated efficiently when some set $Z_i(u)$ changes. If $Z_i(u) = \emptyset$, we set $\beta_i(u) = \infty$.

  The sets $Z_i(u)$ are maintained as follows. First, when $uv \in F_i$ is deleted, we remove $D_i(v)$ from $Z_i(u)$. Such updates take $O\left(\sum_{i=1}^{p} |F_i| \log n\right) = O(Mp \log n)$ time in total. When some value $D_i(x)$ changes, we go through all edges $ux \in F_i$ and update the value of $D_i(x)$ in $Z_i(u)$ accordingly. Denote by $\deg_i(x)$ the number of edges of F_i with an endpoint in x. As each $D_i(x)$ changes only $O(d)$ times, we spend $O(\deg_i(x) d \log n)$ time in total on updating the values $Z_i(\cdot)$ when the values $D_i(x)$ change. Since $\sum_{x \in U} \deg_i(x) = O(|F_i|) = O(M)$, this again gives us $O(Mpd \log n)$ total time used on updating the values $Z_i(\cdot)$ for all i.

- Finally, for all $i = 1, \ldots, p$ and $u \in U$, we maintain the *rank*

$$\mu_i(u) = \min \left(\{\beta_i(u)\} \cup \bigcup_{\substack{A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A}) \\ J \in \mathrm{Ival}_u(A^{S,T})}} \{\alpha_i(J)\} \right).$$

  Note that the rank $\mu_i(u)$ can recomputed from scratch in $O(\log n)$ time when some edge rank or some cover rank changes. Both are updated at most $O(Mpd \log n)$ times, and thus the total time needed to maintain ranks is also $O(Mpd \log^2 n)$.

Initialization. As discussed before, we first set up the partitions $\mathcal{A}_1, \mathcal{A}_2$ of the matrices $G_1^+[U_1]$ and $G_2^+[U_2]$, respectively, set $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ and initialize all the auxiliary data structures of Section 4.3.1 accompanying $\mathcal{A}$ in $O(M^2 \log n)$ time.

Note that we already described how to inductively initialize each component of our data structure once we know the values $D_i(v)$. Recall that each of these initialization costs was either $O(M^2 \log n)$ or $O(Mpd \log^2 n)$.

In remains to initialize each $D_i(v)$ to the value $\delta_{H_i}(v, s_i)$. In order to find the values $\delta_{H_i}(\cdot, s_i)$, for each $i = 1, \ldots, p$, we run the breadth-first-search on H_i. By Lemma 4.3.6 and Corollary 4.3.7, this can be done in $O((|U| + |F_i|) \log n \log \log n) = O(M \log^2 n)$ time. Thus, running p breadth-first searches takes $O(Mp \log^2 n)$ time. The overall cost of initialization can be easily seen to be $O(Mpd \log^2 n + M^2 \log n)$.

Finding vertices that are not good. We now show the key property of the maintained ranks that allows us to identify vertices that are not good.

Lemma 4.3.17. *For any $i = 1, \ldots, p$, a vertex $u \in U$ is good with respect to i if and only if*

$$\mu_i(u) = D_i(u) - 1.$$

Proof. Let u be good with respect to i. Since the Even-Shiloach algorithm guarantees that for each $ux \in E(H_i)$, $D_i(u) \leq D_i(x) + 1$, then we also have $D_i(u) \leq \min\{D_i(x) : ux \in E(H_i)\} + 1$. Recall that u is good with respect to i if and only if for some edge $ux \in E(H_i)$ we have

$$D_i(u) = D_i(x) + 1 \geq \min\{D_i(x) : ux \in E(H_i)\} + 1.$$

Hence, in fact u is good if and only if

$$D_i(u) = \min\{D_i(x) : ux \in E(H_i)\} + 1,$$

and it is sufficient to show that $\mu_i(u) = \min\{D_i(x) : ux \in E(H_i)\}$. We have

$$\begin{aligned}\min\{D_i(x) : ux \in E(H_i)\} &= \min\left(\{D_i(x) : ux \in F_i\} \cup \{D_i(x) : ux \in E(G_1^+[U_1] \cup G_2^+[U_2])\}\right) \\ &= \min\left(\beta_i(u), \min\{D_i(x) : ux \in E(G_1^+[U_1] \cup G_2^+[U_2])\}\right).\end{aligned}$$

Hence, by the definition of the rank, it is enough to prove that:

$$\min\{D_i(x) : ux \in E(G_1^+[U_1] \cup G_2^+[U_2])\} = \min_{\substack{A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A}) \\ J \in \mathrm{Ival}_u(A^{S,T})}} \{\alpha_i(J)\}.$$

Recall that

$$\{x : ux \in E(G_1^+[U_1] \cup G_2^+[U_2])\} = \bigcup_{A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A})} \mathrm{Out}_u(A^{S,T}) = \bigcup_{A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A})} \bigcup \mathrm{Ival}_u(A^{S,T}).$$

Therefore, it is enough to prove that

$$\min\left\{D_i(x) : x \in \bigcup_{\substack{A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A}) \\ J \in \mathrm{Ival}_u(A^{S,T})}} J\right\} = \min \bigcup_{\substack{A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A}) \\ J \in \mathrm{Ival}_u(A^{S,T})}} \{\alpha_i(J)\}.$$

Consequently, this can be done by proving that for any $u \in U$, $A^{S,T} \in \mathrm{r}\pi_u(\mathcal{A})$, and $J \in \mathrm{Ival}_u(A^{S,T})$:

$$\min\{D_i(x) : x \in J\} = \alpha_i(J).$$

Recall that $\alpha_i(J)$ is defined as $\min\{j : \phi_{i,j}(J) = 0\}$, or equivalently

$$\begin{aligned}\alpha_i(J) &= \min\{j : \phi_{i,j}(J) = 0\} \\ &= \min\left\{j : J \not\subseteq \bigcup Q_{i,j}^{S,T}\right\} \\ &= \min\left\{j : J \not\subseteq T \setminus B_{i,j}^{S,T}\right\} \\ &= \min\left\{j : J \cap B_{i,j}^{S,T} \neq \emptyset\right\} \\ &= \min\left\{j : J \cap B_{i,j} \cap \mathrm{act}(A^{S,T}) \neq \emptyset\right\} \\ &= \min\{j : J \cap B_{i,j} \neq \emptyset\},\end{aligned}$$

since we have $J \subseteq \mathrm{act}(A^{S,T})$. Finally, we obtain

$$\alpha_i(J) = \min\{j : J \cap B_{i,j} \neq \emptyset\} = \min\{j : J \cap \{v : D_i(v) \leq j\} \neq \emptyset\}.$$

Therefore, we have $J \cap \{v : D_i(v) \leq \alpha_i(J)\} \neq \emptyset$ and $J \cap \{v : D_i(v) \leq \alpha_i(J) - 1\} = \emptyset$. Hence, for some vertex $u \in J$, $D_i(u) = \alpha_i(J)$, and for all $v \in J$, $D_i(v) \geq \alpha_i(J)$. Therefore, we conclude that in fact

$$\min\{D_i(x) : x \in J\} = \alpha_i(J)$$

holds. □

Proof of Theorem 4.3.15. Observe that Lemma 4.3.17 gives us a very simple characterization of good vertices in terms of the values we maintain. We have shown that the time needed to maintain all the components of our data structure is $O((Mpd+M^2)\log^2 n)$. Hence, maintaining the good vertices with respect to all values i when the values $D_i(v)$ are updated also takes $O((Mpd + M^2)\log^2 n)$ time. □

Proof of Theorem 4.3.2.

In our case we run the algorithm of Theorem 4.3.14 on the graph $H = G_1^+[U_1] \cup G_2^+[U_2]$. Observe that in the algorithm of Bernstein (Theorem 4.3.14), each instance of Even-Shiloach tree is run on a graph H with $O(M)$ additional edges with endpoints in $U_1 \cup U_2$ added. Moreover, for each step $k = q, \ldots, 0$ of the algorithm of Theorem 4.3.14, there are p instances of Even-Shiloach tree maintained, each up to depth d, for numbers p, d satisfying $pd = O(M \log n)$. Apart from maintaining Even-Shiloach trees, Bernstein's algorithm it runs in $O(M^2 \log n)$ time.

We maintain the Even-Shiloach trees of each step using the data structure of Theorem 4.3.15. Since $pd = O(M \log n)$ in each step, and there are $O(\log n)$ steps, the total update time of the algorithm is

$$O(M^2 \log n + (M^2 + M^2 \log n)\log^2 n \cdot \log n) = O(M^2 \log^4 n).$$

□

4.4 Decremental Planar Transitive Closure With Near-Linear Total Update Time

In this section we show a decremental all-pairs reachability algorithm for planar graphs with nearly-linear total update time. Formally, we prove the following Theorem.

Theorem 4.4.1. *Let G be a simple planar digraph. Then, there exists a decremental transitive closure algorithm with $O(\log^5 n)$ amortized update time (over all the edge deletions), and $O(\sqrt{n} \log n \log\log n)$ worst-case query time. The algorithm is Monte Carlo randomized and is correct with high probability.*

4.4.1 Reachability Preserving Reductions

In Theorem 4.4.1 we only assume that our input digraph G is simple and planar. However, it will be more convenient for us to work with *plane* digraphs with additional special properties, such as triangulation or bounded-degree. Consequently, we will repeatedly reduce our general problem of maintaining various reachability information about G to the same problem on a graph with more structure.

Definition 4.4.2. *Let $G = (V, E)$ be a planar digraph. Let $G' = (V', E_0 \cup E' \cup E^\times)$ be a digraph such that V' is a disjoint union of sets $S(v)$, where $v \in V$ and such that $v \in S(v)$, and the sets $E_0, E', E^\times$ are pairwise disjoint. We say that G' preserves reachability of G if:*

1. *The components of (V', E_0) are exactly the sets $S(v)$, $v \in V$.*
2. *The strongly-connected components of (V', E_0) are exactly the sets $S(v)$, $v \in V$.*
3. *We have $uv = e \in E$ if and only if there exists $u'v' = e' \in E'$ such that $\mathrm{id}(e) = \mathrm{id}(e')$, $u' \in S(u)$ and $v' \in S(v)$.*

Let *reachability preserving reduction* be a procedure that takes the input graph G and produces a more structured graph G' that additionally preserves the reachability of G.

The following lemma shows that we can combine reachability preserving reductions sequentially.

Lemma 4.4.3. *Let $G = (V, E)$ be a digraph. Suppose $G'_1 = (V'_1, E_{0,1} \cup E'_1 \cup E'^{\times}_1)$ preserves reachability of G and $G'_2 = (V'_2, E_{0,2} \cup E'_2 \cup E'^{\times}_2)$ preserves reachability of G'_1.*

Then $G'_2 = G' = (V'_2, E_0 \cup E' \cup E^{\times})$ preserves reachability of G, where:

$$E_0 = E_{0,2} \cup \{e' \in E'_2 : e \in E_{0,1} \wedge \mathrm{id}(e) = \mathrm{id}(e')\},$$
$$E' = \{e' \in E'_2 : e \in E'_1 \wedge \mathrm{id}(e) = \mathrm{id}(e')\},$$
$$E^{\times} = E^{\times}_2 \cup \{e' \in E'_2 : e \in E^{\times}_1 \wedge \mathrm{id}(e) = \mathrm{id}(e')\}.$$

Proof. Since we have $\{\mathrm{id}(e) : e \in E(G'_1)\} = \{\mathrm{id}(e) : e \in E'_2\}$ and $\{\mathrm{id}(e) : e \in E\} = \{\mathrm{id}(e) : e \in E'_1\}$, E_0, E' and $E^{\times}$ are pairwise disjoint, $E_0 \cup E' \cup E^{\times} = E(G'_2)$ and $\{\mathrm{id}(e) : e \in E\} = \{\mathrm{id}(e) : e \in E'\}$.

Let the strongly connected components of $(V'_1, E_{0,1})$ be the sets $S_1(v)$, $v \in V$. Let the strongly connected components of $(V'_2, E_{0,2})$ be the sets $S_2(v')$, $v' \in V'_1$. We now prove that the (strongly) connected components of (V'_2, E_0) are the sets $S(v) = \bigcup_{v' \in S_1(v)} S_2(v')$.

First, since $E_{0,2} \subseteq E_0$, each strongly connected component of $(V'_2, E_{0,2})$ is a subset of some strongly connected component of (V'_2, E_0). We have $u''v'' = e'' \in E_0 \setminus E_{0,2}$ if and only if there exists an edge $u'v' = e \in E_{0,1}$ satisfying $\mathrm{id}(e) = \mathrm{id}(e')$ and $u'' \in S_2(u')$ and $v'' \in S_2(v')$. Subsequently, we can say more generally that a path $u' \to v'$ exists in $(V'_1, E_{0,1})$ if and only if a path from $S_2(u')$ to $S_2(v')$ exists in (V'_2, E_0). But a path $u' \to v'$ exists in $(V'_1, E_{0,1})$ if and only if $v' \to u'$ also exists in this graph, or, equivalently, $\{u', v'\} \in S_1(v)$ for some $v \in V$. So finally a path from $S_2(u')$ to $S_2(v')$ exists in (V'_2, E_0) if and only if a path from $S_2(v')$ to $S_2(u')$ exists in (V'_2, E_0), and if and only if $\{u', v'\} \in S_1(v)$ for some $v \in V$. This proves that the strongly connected components of (V'_2, E_0) are indeed the sets $S(v) = \bigcup_{v' \in S_1(v)} S_2(v')$.

By the definitions of E' and E'_2, we have $u''v'' = e'' \in E'$ if and only if there exists an edge $u'v' = e' \in E'_1$ such that $\mathrm{id}(e') = \mathrm{id}(e'')$, $u'' \in S_2(u')$, and $v'' \in S_2(v')$. By the definition of E'_1, such an edge e' exists if and only if there exists an edge $uv = e \in E$ such that $\mathrm{id}(e) = \mathrm{id}(e')$,$u' \in S_1(u)$, and $v' \in S_1(v)$. Since $S(u) = \bigcup_{u' \in S_1(u)} S_2(u')$ and $S(v) = \bigcup_{v' \in S_1(v)} S_2(v')$, $u'' \in S(u)$ and $v'' \in S(v)$. □

We now discuss four useful reachability preserving reductions.

Lemma 4.4.4. *Let G be a simple digraph. In linear time one can compute a graph G' larger than G by a constant factor, that preserves the reachability of G and is connected.*

Proof. We obtain $G' = (V, E \cup E^{\times})$ by adding a minimal set $E^{\times}$ of $k - 1$ edges that would make G connected where k is the number of connected components of G. □

Lemma 4.4.5. *Let G be a simple and connected plane digraph. In linear time one can compute a simple and connected plane digraph G' larger than G by a constant factor that preserves the reachability of G and has constant degree.*

Proof. We compute $G' = (V', E_0 \cup E')$ based on $G = (V, E)$ as follows. Let v be a vertex of G with an edge ring $e^v_1, \ldots, e^v_k$. If $k \geq 3$, define $C(v)$ to be a directed cycle $v_1 v_2 \ldots v_k$ where $v_1 = v$ and $C(v)$ contains directed edges $v_i v_{i+1}$ for each $i = 1, \ldots, k$, and $v_{k+1} = v_1$. If $k \leq 2$, on the other hand, we set $C(v) = \{v\}$ and $v_1 = v_k = v$.

Define $V' = \bigcup_{v \in V} V(C(v))$ and $E_0 = \bigcup_{v \in V} E(C(v))$. We also let $S(v) = V(C(v))$ – clearly for each v $G'[S(v)]$ is strongly connected and $v \in S(v)$.

For any $uv = e \in E$, suppose the tail of e appears as the i-th edge in the edge ring of u and its head appears as the j-th edge in the edge ring of v. We include the edge $e' = u_i v_j$ in E' and set $\mathrm{id}(e') = \mathrm{id}(e)$. Clearly, the edge e' goes from a vertex in $S(u)$ to a vertex of $S(v)$.

Since G' is obtained from G be expanding vertices of G into cycles, the embedding of G' can be easily devised from the embedding of G and thus G' is also plane. Finally, observe that each vertex of G' has degree no more than 3. □

Lemma 4.4.6. *Let G be a simple and connected plane digraph. Let F be some subset of faces of G such that the bounding cycle of each face of F has length at least 4. Then, in linear time one can triangulate each face of F by only adding edges inside F so that the obtained graph $G' = (V, E \cup E^{\times})$ preserves the reachability and the degree of each vertex grows by a constant factor.*

Proof. We obtain G' from G by triangulating each face $f \in F$, where $V(f) = w_1, \ldots, w_k$, $k > 3$, by adding $k - 3$ edges inside it so that no vertex v gets added more than 2 edges per its occurrence on the bounding cycle of f. This can be achieved, e.g., by adding edges $w_1w_3, w_3w_k, w_kw_4, w_4w_{k-1}, \ldots$ and so on. These edges are included in $E^{\times}$. Afterwards, all faces of F are triangulated.

Consequently, since the total number of occurrences of a vertex v on the face bounding cycles is $\deg(v)$, we have $\deg_{G'}(v) \leq 3 \deg_G(v)$. Clearly, we have $|E(G')| = O(|V| + |E|) = O(n)$. □

Lemma 4.4.7. *Let $G = (V, E)$ be a simple, connected, and triangulated digraph. In $O(n \log n)$ time one can compute a graph G' that preserves the reachability of G and is larger than G by a constant factor, and its simple decomposition $\mathcal{T}_{G'}$.*

Proof. We apply Theorem 4.1.4 to the undirected graph $\bar{G}$ obtained from G by turning directed edges into undirected edges and consequently obtain a graph $\bar{G}' = (V', \bar{E}_0, \bar{E}', \bar{E^{\times}})$ along with its simple decomposition $\mathcal{T}_{\bar{G}'}$ in $O(n \log n)$ time. We obtain G' from $\bar{G}'$ by restoring edge directions of each $\bar{e} \in E(\bar{G}')$ in each piece $H \in \mathcal{T}_{\bar{G}'}$ that contains $\bar{e}$, as follows.

- The edges of $\bar{E}_0$ are oriented so that each $G'[S(v)]$, where $v \in V$, is strongly-connected. This can be easily done by the construction of Theorem 4.1.4.
- For each $uv = e \in E$, let $u'v' = e' \in \bar{E}'$ be such that $\mathrm{id}(e) = \mathrm{id}(\bar{e}) = \mathrm{id}(e')$, $u' \in S(u)$, and $v' \in S(v)$. We orient $\bar{e}$ from u' to v'.
- For each $\bar{e} \in \bar{E^{\times}}$, we orient it arbitrarily.

Clearly, by the properties of $\bar{G}'$ guaranteed by Theorem 4.1.4, the obtained graph G' preserves the reachability of G and has a simple decomposition $\mathcal{T}_{G'}$. □

4.4.2 The Data Structure

Lemma 4.4.8. *Let G be planar digraph and suppose $G' = (V', E_0 \cup E' \cup E^{\times})$ preserves reachability of G. Then, the decremental transitive closure problem on G can be reduced to the decremental transitive closure problem on G'.*

Proof. We first initialize the decremental transitive closure algorithm with the graph G'. Next, we issue deletions of all the edges of $E^{\times}$. Afterwards, G' contains exactly $|E|$ edges that are not in E_0. The edges of E_0 are never removed from G', whereas a deletion of $e \in E$ from G is translated to a deletion of e' from G', where $\mathrm{id}(e') = \mathrm{id}(e)$.

At any time, since $(V, E_0) \subseteq G'$ and $G'[S(v)]$ is strongly-connected for any $v \in V$, one can easily see that a path $u \to v$ exists in G if and only if a path $u' \to v'$ exists in G', where u'

is any vertex of $S(u)$ and v' is any vertex of $S(v)$. Hence, a reachability query on G can be translated to a single reachability query on G' (regarding a different pair of vertices). □

In the remaining part of this section we assume that G is in fact connected, plane-embedded, and we are given its simple recursive decomposition $\mathcal{T}_G$. To actually perform the reduction to that case, we use Lemma 4.4.8 with graph G' obtained from G by subsequently applying Lemma 4.4.4, computing some plane embedding of G in linear time [9, 17, 51], triangulating G using Lemma 4.4.6, and finally using Lemma 4.4.7. The reduction takes $O(n \log n)$ time, by Lemma 4.4.7. The obtained graph G' is larger than the original graph G by a constant factor only so we may ignore this overhead.

Although the graph G undergoes edge updates, the decomposition $\mathcal{T}_G$ will evolve only in a very limited way. Namely, suppose an edge e is removed from G. Then, e is also removed from all the pieces $H \in \mathcal{T}_G$ that initially contained e. We assume that the sets $V(H)$ and ∂H are fixed before any updates to G happen, and do not change throughout, even though the individual pieces may cease to be edge-induced (i.e., some vertices of $V(H)$ may become isolated). Moreover, even though we have $\partial H = V(H) \cap V(G - H)$ initially, since ∂H does not change and $V(G - H)$ shrinks in time, we only guarantee $V(H) \cap V(G - H) \subseteq \partial H$. This is sufficient for our needs, though. Clearly, edge deletions cannot break any of the properties (1), (2), (3) and (4) of the simple recursive decomposition of Section 4.1.

However, now edge deletions may change the holes of individual pieces and possibly break properties (5) and (6) of a simple decomposition. It turns out that this is not a problem, though. Since our recursive decomposition is initially simple, for each $H \in \mathcal{T}_G$, ∂H lies on $O(1)$ separator curves of H. This property remains true when G undergoes edge deletions (without changing the separator curves) since a separator curve of G is also a separator curve of any subgraph of G. Thus, we may fix some order $\prec_H$ on ∂H precisely as described at the beginning of Section 4.2.

In the following, when we write $\mathcal{T}_G$, we always refer to the *current* decomposition, as explained above.

Given $H \in \mathcal{T}_G$, we define two directed graphs $\mathcal{R}(H)$, $\mathrm{In}(H)$ on the subsets of $V(H)$. Both of these graphs are induced subgraphs of the transitive closure of H.

- If H is non-leaf, then
$$\mathcal{R}(H) = H^+[\partial H \cup \mathrm{Sep}(H)],$$
i.e., for $u, v \in \partial H \cup \mathrm{Sep}(H)$ there is a directed edge uv in $\mathcal{R}(H)$ if and only if there is a path $u \to v$ in H. Otherwise, for a leaf H we set $\mathcal{R}(H) = H^+$.
- $\mathrm{In}(H) = H^+[\partial H]$. Note that $\mathrm{In}(H)$ is a subgraph of $\mathcal{R}(H)$.

Lemma 4.4.9. *Let $H \in \mathcal{T}_G$ be a non-leaf piece. Then $\mathcal{R}(H) = (\mathrm{In}(\mathrm{child}_1(H)) \cup \mathrm{In}(\mathrm{child}_2(H)))^+$.*

Proof. Let $H_i = \mathrm{child}_i(H)$ for $i = 1, 2$. By Lemma 4.1.2, we have $\partial H \cup \mathrm{Sep}(H) = \partial H_1 \cup \partial H_2$. Hence, we only need to show that for $u, v \in \partial H \cup \mathrm{Sep}(H)$ there is a path $P = u \to v$ in H if and only if there is a path $u \to v$ in $\mathrm{In}(H_1) \cup \mathrm{In}(H_2)$.

The " $\Longleftarrow$ " direction is obvious because each edge in $\mathrm{In}(H_i)$ corresponds to a path in H_i.

To prove the " $\Longrightarrow$ " part, split P into maximal paths $P_1, \ldots, P_q$ fully contained in either H_1 or H_2. Let P_j go from a_j to b_j and suppose that P_j is fully contained in H_i. Then a_j is either equal to $u \in \partial H_i$ if $j = 1$ or is the same as the last vertex of P_{j-1}, and hence $a_j \in V(H_1) \cap V(H_2) = \mathrm{Sep}(H) \subseteq \partial H_i$. Similarly we prove that $b_j \in \partial H_i$. By the definition of $\mathrm{In}(H_i)$, there is an edge $a_j b_j$ in $\mathrm{In}(H_i)$. We conclude that there is a directed path $ua_2 \ldots a_q v$ in $\mathrm{In}(H_1) \cup \mathrm{In}(H_2)$. □

For each $H \in \mathcal{T}_G$, we maintain the graphs $\mathcal{R}(H)$ and $\mathrm{In}(H)$ subject to edge deletions issued to G. If H is a leaf, we can afford recomputing $\mathcal{R}(H)$ from scratch after the deletion of each edge e that is also contained in $E(H)$. Since H has $O(1)$ edges initially, the recomputation of $\mathcal{R}(H) = H^+$ takes $O(1)$ time and happens $O(1)$ times in total. Since $\mathcal{T}_G$ has $O(n)$ leaves, the total time needed to maintain the graphs $\mathcal{R}(H)$ for leaf pieces H is $O(n)$.

Suppose H is not a leaf piece. Then by Lemma 4.4.9, $\mathcal{R}(H) = (\mathrm{In}(\mathrm{child}_1(H)) \cup \mathrm{In}(\mathrm{child}_2(H)))^+$. Hence, it is possible to compute (and maintain) $\mathcal{R}(H)$ based only on the graphs $\mathrm{In}(\mathrm{child}_1(H))$ and $\mathrm{In}(\mathrm{child}_2(H))$, and the updates they undergo. Similarly, since for any $H \in \mathcal{T}_G$, $\mathrm{In}(H)$ is a subgraph of $\mathcal{R}(H)$, $\mathrm{In}(H)$ is updated easily based only on the updates to $\mathcal{R}(H)$. We say that $\mathcal{R}(H)$ *depends* on $\mathrm{In}(\mathrm{child}_1(H))$ and $\mathrm{In}(\mathrm{child}_2(H))$, whereas $\mathrm{In}(H)$ depends on $\mathcal{R}(H)$.

Since for $i = 1, 2$, $\partial\mathrm{child}_i(H)$ lies on $O(1)$ pairwise disjoint separator curves of $\mathrm{child}_i(H)$, we can use Theorem 4.3.2 for maintaining $\mathcal{R}(H)$. Specifically, by setting for $i = 1, 2$, $G_i = \mathrm{child}_i(H)$ and $U_i = \partial\mathrm{child}_i(H)$, we conclude that if we are given the updates to $\mathrm{In}(\mathrm{child}_1(H))$ and $\mathrm{In}(\mathrm{child}_2(H))$ after each edge deletion that actually changes these graphs, we can maintain $\mathcal{R}(H)$ in $O(|\partial\mathrm{child}_1(H) \cup \partial\mathrm{child}_2(H)|^2 \log^4 n) = O((|\partial\mathrm{child}_1(H)|^2 + |\partial\mathrm{child}_2(H)|^2) \log^4 n)$ time. The total time needed to maintain all the graphs $\mathcal{R}(H)$, if we are given the updates the respective graphs $\mathrm{In}(\mathrm{child}_1(H))$ and $\mathrm{In}(\mathrm{child}_2(H))$, through all pieces $H \in \mathcal{T}_G$, is, by property (4) of $\mathcal{T}_G$,

$$O\left(\sum_{H \in \mathcal{T}_G} |\partial H|^2 \log^4 n\right) = O(n \log^5 n).$$

In order to be sure that all the $O(n)$ maintained graphs $\mathcal{R}(H)$ are correct with high probability, it is enough to set the constant d from Theorem 4.3.2 to $d = \beta + 2$.

To guarantee that all the graphs are updated before we start updating their dependent graphs, we proceed as follows. During the initialization, we fix some topological order on the graph describing the dependencies between various graphs that we maintain (the dependencies between the graphs $\mathcal{R}(\cdot)$, $\mathrm{In}(\cdot)$, and other useful graphs that we will define in Section 4.5, are depicted in Figure 4.4). When an edge e is removed from G, we create a priority queue Q of graphs that potentially need updates. The elements of Q are keyed by their position in the fixed topological order. First, the unique graph $\mathcal{R}(H)$ such that H is a leaf and $e \in E(H)$, is pushed to Q. We repeatedly pop a graph Z out of Q and process either the edge deletion (in the case when $Z = \mathcal{R}(H)$ and H is a leaf) or the changes to the graphs that Z depends on. If the graph Z actually changes after switching e on, we push to Q all graphs Z' such that Z' directly depends on Z. The correctness of this update procedure follows form the fact that the dependencies do not form cycles.

Observe that since for each maintained graph Z, there are at most two graphs Z' (again, see Figure 4.4) that depend directly on M, the total number of times a graph Z is inserted into Q is proportional to the total number of changes to the maintained graphs $\mathcal{R}(\cdot)$ and $\mathrm{In}(\cdot)$. This quantity is clearly bounded by their total size, i.e., $O\left(\sum_{H \in \mathcal{T}_G} |\partial H|^2 + |E(G)|\right) = O(n \log n)$. Thus, the cost of priority queue operations on Q is $O(n \log^2 n)$. We have thus proved the following lemma.

Lemma 4.4.10. *The total time needed to maintain all the graphs $\mathcal{R}(H)$ and $\mathrm{In}(H)$, where $H \in \mathcal{T}_G$, subject to any sequence of edge deletions issued to G, is $O(n \log^5 n)$. The algorithm is correct with high probability.*

Before we show how the maintained graphs can be used to answer reachability queries on G, we need one more definition. Let $H \in \mathcal{T}_G$. For $v \in V(H)$, let H_v be the leftmost leaf piece in

$\mathcal{T}_G(H)$ such that $v \in V(H_v)$. Then we define

$$\mathcal{R}^v(H) = \mathcal{R}(H_v) \cup \bigcup_{\substack{H' \in \mathcal{T}_G(H) \\ H' \neq H_v \\ H_v \in \mathcal{T}_G(H')}} \mathrm{In}(\mathrm{child}_1(H')) \cup \mathrm{In}(\mathrm{child}_2(H')).$$

In other words, $\mathcal{R}^v(H)$ is the union of $\mathcal{R}(H_v)$ and all graphs $\mathrm{In}(\mathrm{child}_1(H')) \cup \mathrm{In}(\mathrm{child}_2(H'))$, where $H' \neq H_v$ lies on the path from H to H_v in $\mathcal{T}_G$.

Lemma 4.4.11. *Let $H \in \mathcal{T}_G$, $u \in \partial H \cup \mathrm{Sep}(H)$ and $v \in V(H)$. Then there exists a path $u \to v$ ($v \to u$) in H if and only if there exists a path $u \to v$ ($v \to u$, respectively) in $\mathcal{R}^v(H)$.*

Proof. We only consider $u \to v$ paths; the proof for $v \to u$ paths is analogous. We proceed by induction on $\chi(H)$. Clearly, if $\chi(H) = 0$, i.e., H is a leaf piece, the lemma holds since $\mathcal{R}^v(H) = \mathcal{R}^v(H_v) = \mathcal{R}(H_v) = H^+$.

Suppose H is a non-leaf piece. The " $\Longleftarrow$ " part is easy since $\mathcal{R}^v(H)$ is a union of subgraphs of transitive closures over some subgraphs of H.

To prove the " $\Longrightarrow$ " part, let $P = u \to v$ be a simple path in H. Split P into two paths P_1P_2, where $P_1 = u \to s$ and $P_2 = s \to v$ are such that $s \in \partial H \cup \mathrm{Sep}(H)$ and all subsequent vertices of P_2 *do not* belong to $\partial H \cup \mathrm{Sep}(H)$. Note that P_1 and P_2 can both be of length 0. Also, if P_2 has at least one edge, then $v \notin \partial H \cup \mathrm{Sep}(H)$.

Since $u, s \in \partial H \cup \mathrm{Sep}(H)$ and there exists a path $u \to s$ in H, we have $us \in E(\mathcal{R}(H))$. Recall that by Lemma 4.4.9, $\mathcal{R}(H) = (\mathrm{In}(\mathrm{child}_1(H)) \cup \mathrm{In}(\mathrm{child}_2(H)))^+$. Hence, there exists a path $P_1 = u \to s$ in $(\mathrm{In}(\mathrm{child}_1(H)) \cup \mathrm{In}(\mathrm{child}_2(H)) \subseteq \mathcal{R}^v(H)$.

Also note that if P_2 has at least one edge, then all its vertices except s belong to $V(H) \setminus (\partial H \cup \mathrm{Sep}(H)) \subseteq V(H) \setminus \mathrm{Sep}(H)$. Hence, P_2 is entirely contained in the unique piece $\mathrm{child}_i(H)$ such that $v \in V(\mathrm{child}_i(H))$. We obtain that H_v is a descendant of $\mathrm{child}_i(H)$ and hence $\mathcal{R}^v(H) = \mathcal{R}(H) \cup \mathcal{R}^v(\mathrm{child}_i(H))$. By the induction hypothesis, a path $s \to v$ exists in $\mathcal{R}^v(\mathrm{child}_i(H))$. As $\mathcal{R}^v(\mathrm{child}_i(H)) \subseteq \mathcal{R}^v(H)$, the lemma follows. □

Lemma 4.4.12. *Let $H \in \mathcal{T}_G$. For any $u, v \in V(H)$, there exists a path $u \to v$ in H if and only if there exists a path $u \to v$ in $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$.*

Proof. We again proceed by induction on $\chi(H)$. If H is a leaf, then $\mathcal{R}^u(H) = \mathcal{R}^v(H) = H^+$, and the lemma holds trivially. Suppose H is a non-leaf piece. The " $\Longleftarrow$ " part is easy since $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$ is a union of subgraphs of transitive closures over some subgraphs of H.

Let P be a path $u \to v$ in H. If P does not go through a vertex of $\mathrm{Sep}(H)$, then it is fully contained in $\mathrm{child}_i(H)$ for some $i \in \{1, 2\}$, and both H_u and H_v are descendants of $\mathrm{child}_i(H)$ (in fact, we have $\{u, v\} \cap V(\mathrm{child}_{3-i}(H)) = \emptyset$). Then by the induction hypothesis, there exists a path $u \to v$ in $\mathcal{R}^u(\mathrm{child}_i(H)) \cup \mathcal{R}^v(\mathrm{child}_i(H))$ which is a subgraph of $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$ in this case.

Otherwise P goes through a vertex $s \in \mathrm{Sep}(H)$. By Lemma 4.4.11, there exist paths: $u \to s$ in $\mathcal{R}^u(H)$ and $s \to v$ in $\mathcal{R}^v(H)$. Their concatenation is clearly contained in $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$. □

For each matrix $\mathrm{In}(H)$[1], where $H \in \mathcal{T}_G$, we also maintain the auxiliary data structures of Section 4.3.1. By Lemma 4.3.3, for all pieces H this takes $O\left(\sum_{H \in \mathcal{T}_G} |\partial H|^2 \log n\right) = O(n \log^2 n)$ additional time.

Lemma 4.4.13. *Let $u, v \in V(H)$, where $H \in \mathcal{T}_G$. We can test whether v is reachable from u in $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$ in $O\left(\sqrt{n/\rho^{\ell(H)}} \log n \log\log n\right)$ time.*

[1]Recall that we sometimes identify the graphs that we maintain with their respective adjacency matrices.

Proof. Note that $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$ is a union of $O(\log n)$ graphs of the form $\mathrm{In}(H') = H^*[\partial H']$, for which we maintain the partition of Lemma 4.2.8 and auxiliary data of Lemma 4.3.3, and two graphs $\mathcal{R}(H_u), \mathcal{R}(H_v)$ of constant size. Hence, $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$ satisfies the requirements of Lemma 4.3.6 and Corollary 4.3.7, and we can find the vertices reachable from u in $\mathcal{R}^u(H) \cup \mathcal{R}^v(H)$ using breadth-first-search in $O(|V(\mathcal{R}^u(H) \cup \mathcal{R}^v(H)) + O(1)| \log n \log\log n + 1)$ time. To finish the proof, observe that

$$\begin{aligned}
|V(\mathcal{R}^u(H) \cup \mathcal{R}^v(H))| &= O(1) + \sum_{\substack{H' : H' \in \mathcal{T}_G(H) \\ H_u \in \mathcal{T}_G(H') \vee H_v \in \mathcal{T}_G(H') \\ H' \neq H_u \wedge H' \neq H_v}} |\partial \mathrm{child}_1(H')| + |\partial \mathrm{child}_2(H')| \\
&\leq O(1) + 2 \sum_{i=\ell(H)}^{\ell(H_u)} O\left(\sqrt{n/\rho^i}\right) + 2 \sum_{i=\ell(H)}^{\ell(H_v)} O\left(\sqrt{n/\rho^i}\right) \\
&= O\left(\sqrt{n/\rho^{\ell(H)}}\right).
\end{aligned}$$

□

Corollary 4.4.14. *Let $u, v \in V(G)$. We can test whether v is reachable from u in G in $O(\sqrt{n} \log n \log\log n)$ time.*

Proof. Apply Lemma 4.4.13 for $H = G$. □

Note, however, that in the query algorithm described in Lemma 4.4.13 the graph $\mathcal{R}^{u\prime}(H) \cup \mathcal{R}^{v\prime}(H)$ is never built explicitly: we only access $O(\log n)$ needed matrices $\mathrm{In}(\cdot)$ and the auxiliary data structures accompanying them. This proves Theorem 4.4.1.

4.5 Extensions of The Decremental Transitive Closure

In this section we show some additional interesting results concerning decremental reachability in planar digraphs.

4.5.1 Maintaining the Status of Individual Edges

Lemma 4.5.1. *Let G be a simple planar digraph and suppose $G' = (V', E_0 \cup E' \cup E^\times)$ preserves reachability of G. Then the problems of:*

- *maintaining the set of intra-SCC edges of G,*
- *maintaining the set of 1-cut edges of G,*
- *maintaining the set of strong bridges of G,*

under edge deletions can be reduced to the respective problems on G'.

Proof. Analogously as in Lemma 4.4.8, we first initialize the decremental algorithm with the graph G'. Next, we issue deletions of all the edges of $E^\times$. Afterwards, G' contains exactly $|E|$ edges that are not in E_0. The edges of E_0 are never removed from G', whereas the deletion of each $e \in E$ from G is translated to a deletion of e' from G', where $\mathrm{id}(e') = \mathrm{id}(e)$.

Recall from the proof of Lemma 4.4.8, that for any $u, v \in V(G)$ a path $u \to v$ exists in G if and only if a path $u' \to v'$ exists in G', where u' is any vertex of $S(u)$ and v' is any vertex of $S(v)$. Hence $e \in E$ is an inter-SCC edge, a 1-cut edge, or a strong bridge of G if and only if $e' \in E'$, $\mathrm{id}(e') = \mathrm{id}(e)$, and e' is an inter-SCC edge, a 1-cut edge, or a strong bridge, respectively, of G'. Note that G' might have more 1-cut edges or strong bridges in the set E_0, but those are not of our interest. □

We now show how to extend the data structure of Section 4.4 so that it additionally maintains the sets of intra-SCC edges, 1-cut edges, and strong bridges of a planar graph G under edge deletions. By applying the same set of reachability preserving reductions as we did in Section 4.4, and applying Lemma 4.5.1, we may assume that G is initially plane and has a simple recursive decomposition $\mathcal{T}_G$.

We start with the following lemma, from which it follows that for each $H \in \mathcal{T}_G$, ∂H lies on $O(1)$ separator curves of $G - H$ throughout the process.

Lemma 4.5.2. *Let $\mathcal{T}_G$ be a simple recursive decomposition and let $H \in \mathcal{T}_G$. A cycle bounding a hole of H is a separator curve of $G - H$.*

Proof. Note that the holes of H can be seen as unions of original faces of G, merged by removing the edges of $G - H$ from G. Thus, each edge of $G - H$ lies inside some unique hole h of H. Let $E_h \subseteq E(G - H)$ be the subset of edges lying inside h.

Let $\mathcal{C}_h$ be the cycle bounding a hole h of H. Assume without loss of generality that h is a bounded face. Then, the interior of $\mathcal{C}_h$ is the same as the interior of h (the case when h is unbounded is analogous; we replace each occurrence of "inside of $\mathcal{C}_h$" with "outside of $\mathcal{C}_h$"). By Fact 4.2.2 and since H is connected, for each $\mathcal{C}_h$, H lies weakly outside $\mathcal{C}_h$. If h is the only hole of H, then clearly $G - H = G[E_h]$ lies weakly on one side of $\mathcal{C}_h$ and the Lemma holds.

Assume now that h is not the only hole and let $h' \neq h$ be some other hole of H. As h' and h are disjoint, h' lies strictly outside $\mathcal{C}_h$. Thus, all edges of $E_{h'}$ lie strictly outside $\mathcal{C}_h$. Hence, $V(G[E_h]) \cap V(G[E_{h'}]) = \emptyset$ for $h' \neq h$. To conclude, note that for each weakly connected component of $G - H$ the edges of that component are contained in a unique subset E_h. □

Remark 4.5.3. *The assumption that $\mathcal{T}_G$ is simple is crucial to proving Lemma 4.5.2, which does not hold if the holes of H are not pairwise-disjoint or not necessarily simple.*

For each piece $H \in \mathcal{T}_G$, apart from the graphs $\mathcal{R}(H)$ and $\mathrm{In}(H)$, we also maintain a graph

$$\mathrm{Ex}(H) = (G - H)^+[\partial H]$$

under edge deletions.

Lemma 4.5.4. *Let $H \in \mathcal{T}_G$ be a non-root piece. Denote by P and S the parent and the sibling of H in $\mathcal{T}_G$, respectively. Then,* $\mathrm{Ex}(H) = (\mathrm{Ex}(P) \cup \mathrm{In}(S))^+[\partial H]$.

Proof. By the definition of a boundary, we have $V(G - P) \cap V(S) \subseteq V(G - P) \cap V(P) \subseteq \partial P$ and similarly $V(G-P) \cap V(S) \subseteq V(G-S) \cap V(S) \subseteq \partial S$. Hence, $V(G-P) \cap V(S) \subseteq \partial P \cap \partial S$.

We only need to show that for $u, v \in \partial H$, there is a path $P = u \to v$ in $G - H$ if and only if there is a path $u \to v$ in $\mathrm{Ex}(P) \cup \mathrm{In}(S)$. The " $\Longleftarrow$ " direction is obvious as each edge in $\mathrm{In}(S)$ corresponds to a path in S, each edge in $\mathrm{Ex}(P)$ corresponds to a path in $G - P$ and $G - H = (G - P) \cup S$.

To prove the " $\Longrightarrow$ " part, split P into maximal paths $P_1, \ldots, P_q$ fully contained in either $G - P$ or S. Let P_j go from a_j to b_j and suppose that P_j is fully contained in S. Then a_j is either equal to $u \in \partial H \cap V(S) \subseteq \partial S$ if $j = 1$, or is the same as the last vertex of P_{j-1}. Hence, $a_j \in V(G-P) \cap V(S) \subseteq \partial S$. Similarly we prove that $b_j \in \partial S$. By the definition of $\mathrm{In}(S)$, there is an edge $a_j b_j \in E(\mathrm{In}(S))$. Analogously, for each subpath $P_j = a_j b_j$ fully contained in $G - P$, both a_j and b_j belong to ∂P, and by the definition of $\mathrm{Ex}(P)$, there is an edge $a_j b_j$ in $\mathrm{Ex}(P)$. We conclude that there is a directed path $u a_2 \ldots a_q v$ in $\mathrm{Ex}(P) \cup \mathrm{In}(S)$. □

By Lemma 4.5.4, for each non-leaf piece H, the graph $\mathrm{Ex}(\mathrm{child}_i(H))$ can be computed inductively based on the "nearby" graphs $\mathrm{In}(\mathrm{child}_{3-i}(H))$ and $\mathrm{Ex}(H)$. In other words, $\mathrm{Ex}(\mathrm{child}_i(H))$

depends on the graphs $\mathrm{In}(\mathrm{child}_{3-i}(H))$ and $\mathrm{Ex}(H)$. Consequently, for maintaining the graphs $\mathrm{Ex}(\mathrm{child}_i(H))$ we may use the same strategy as we did for maintaining the graphs $\mathcal{R}(\cdot)$ in Section 4.4. Namely, each $\mathrm{Ex}(\mathrm{child}_i(H))$ is maintained using Theorem 4.3.2 applied to $G_1 = \mathrm{child}_{3-i}(H), U_1 = \partial\mathrm{child}_{3-i}(H), G_2 = G - H, U_2 = \partial H$. Then, we have $G_1^+[U_1] = \mathrm{In}(\mathrm{child}_{3-i}(H))$ and $G_2^+[U_2] = \mathrm{Ex}(H)$. Note that U_2 indeed lies on $O(1)$ pairwise disjoint separator curves of G_2, by Lemma 4.5.2.

Hence, the total time needed to maintain the graph $\mathrm{Ex}(\mathrm{child}_i(H))$ subject to deletions to the graphs $\mathrm{In}(\mathrm{child}_{3-i}(H))$ and $\mathrm{Ex}(H)$ is $O\left((|\mathrm{child}_1(H)|^2 + |\mathrm{child}_2(H)|^2)\log^4 n\right)$. Through all non-leaf pieces $H \in \mathcal{T}_G$, this sums up to $O(n\log^5 n)$ total time. In other words, maintaining the graphs $\mathrm{Ex}(\cdot)$ does not affect the asymptotic total update time of the data structure of Section 4.4.

Now, for each *leaf* piece $L \in \mathcal{T}_G$, and each $e \in E(L)$, define $\mathcal{L}_e(L) = (G-e)^*[V(L)]$. The proof of the following lemma is completely analogous to the proofs of Lemmas 4.4.9 and 4.5.4, and is thus omitted.

Lemma 4.5.5. *For any leaf piece $L \in \mathcal{T}_G$, and $e \in E(L)$, $\mathcal{L}_e(L) = (\mathrm{Ex}(L) \cup (L-e))^+$.*

Each graphs $\mathcal{L}_e(L)$ is recomputed from scratch each time either L or $\mathrm{Ex}(L)$ changes (recall that L changes when some of its edges is deleted). As each leaf L piece is of constant size, there are $O(n)$ relevant (L, e) pairs. Note that the edge sets of both graphs $L - e$ and $\mathrm{Ex}(L)$ can only shrink $O(1)$ times in total. Therefore, we can maintain the graphs $\mathcal{L}_e(L)$ subject to edge deletions issued to G based on the respective graphs $\mathrm{Ex}(L)$ and $L - e$ in $O(n)$ total time.

The update procedure when some edge is removed from G is the same as in Section 4.4. The only difference is that now we maintain more graphs for each piece of the decomposition. The dependencies between the data structures maintaining the graphs $\mathrm{In}(\cdot)$, $\mathcal{R}(\cdot)$, $\mathrm{Ex}(\cdot)$ and $\mathcal{L}_e(\cdot)$ we use, imposed by Lemmas 4.4.9, 4.5.4 and 4.5.5, are clearly not cyclic. See Figure 4.4.

Lemma 4.5.6. *Let $e = uv$ be contained in a leaf piece $L \in \mathcal{T}_G$. Then:*

- *e is an intra-SCC edge if and only if $vu \in E(\mathcal{L}_e(L))$.*
- *e is a 1-cut edge if and only if $uv \notin E(\mathcal{L}_e(L))$,*
- *e is a strong bridge if and only if $uv \notin E(\mathcal{L}_e(L))$ and $vu \in E(\mathcal{L}_e(L))$.*

Proof. Trivial by the definition of $\mathcal{L}_e(L)$. □

Theorem 4.5.7. *Let G be a planar digraph. Then, in $O(n\log^5 n)$ total time, we can maintain the sets of intra-SCC edges of G, 1-cut edges of G, and strong bridges of G under edge deletions issued to G. The algorithm is Monte Carlo randomized and is correct with high probability.*

Proof. Observe that by Lemma 4.5.6, we can maintain intra-SCC edges, 1-cut-edges, and strong bridges of G decrementally by simply tracking when certain edges of the graphs $\mathcal{L}_e(L)$ disappear.

The total time needed to maintain the graphs $\mathrm{In}(\cdot)$, $\mathcal{R}(\cdot)$, $\mathrm{Ex}(\cdot)$, and $\mathcal{L}_e(\cdot)$ is $O(n\log^5 n)$ as discussed previously. □

4.5.2 Applications

In this section we describe some implications of Theorem 4.5.7. We show that maintaining the status of individual edges under edge deletions can be used in a black-box manner to obtain decremental algorithms for several other reachability problems on planar graphs.

Lemma 4.5.8. *Let G be a simple planar digraph. Suppose we can maintain the intra-SCC edges of G under edge deletions. Then, in $O(n\log n)$ additional total time we can explicitly maintain the identifiers of strongly-connected components of all vertices of G under edge deletions.*

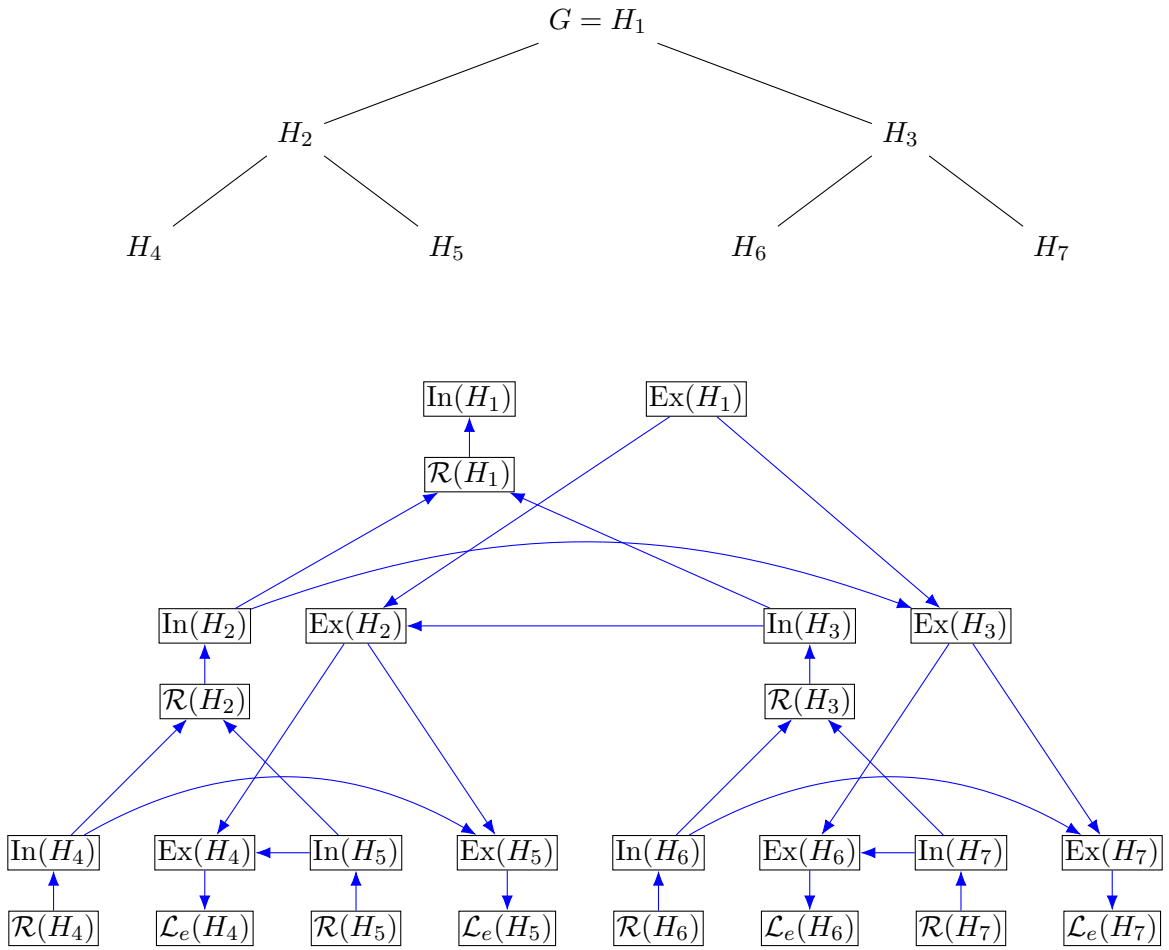

Figure 4.4: An example decomposition $\mathcal{T}_G$ (top) and the dependencies between the corresponding matrices $\mathrm{In}(H), \mathcal{L}(H), \mathrm{Ex}(H), \mathcal{L}_e(H)$, for $H \in \mathcal{T}_G$ (bottom).

Proof. Define $\bar{G}$ to be the undirected graph obtained from G by first removing inter-SCC edges and then ignoring edge directions, i.e., $\bar{G}$ contains only intra-SCC edges of G with their directions ignored. Note that the connected components of $\bar{G}$ are exactly the same as the strongly-connected components of G.

Observe that the set of intra-SCC edges of G only shrinks if G is subject to edge deletions, hence $\bar{G}$ is also subject to edge deletions only. Since we maintain the set of intra-SCC edges of G under edge deletions, we can maintain the set of edges of $\bar{G}$ under edge deletions in additional $O(n)$ time.

To maintain the identifiers of strongly connected components of vertices of G, we can equivalently maintain the identifiers of connected components of the graph $\bar{G}$, which is clearly also planar. This can be achieved by using the planar decremental connectivity algorithm of Łącki and Sankowski [68, Lemma 3] on $\bar{G}$. The total update time of this algorithm is $O(n \log n)$. □

Lemma 4.5.9. *Let G be a simple planar digraph. Suppose we can maintain the intra-SCC edges of G under edge deletions. Let $s \in V(G)$. Then we can maintain the set of vertices reachable from s when G is subject to edge deletions in $O(n \log n)$ additional total time.*

Proof. First let us explicitly maintain the strongly-connected component identifiers of all vertices of G under edge deletions. This takes $O(n \log n)$ additional time, by Lemma 4.5.8. Since

we maintain intra-SCC edges explicitly, we can easily maintain the set of inter-SCC edges of G as well.

A *condensation* of a directed graph G, denoted by $cond(G)$, is a graph obtained from G by contracting all its strongly-connected components. The vertices of $cond(G)$ are sets of vertices of G contained in the corresponding strongly connected components. There is a 1-to-1 correspondence between the edges of $cond(G)$ and the inter-SCC edges of G. Note that $cond(G)$ is a directed acyclic graph (DAG).

Our goal is to maintain the set of vertices of $cond(G)$ which are reachable from the vertex of $cond(G)$ containing the source vertex s. The high-level idea is that in order to maintain strongly-connected components of G we use the decremental strong-connectivity algorithm, whereas to maintain the set of reachable vertices in $cond(G)$ we use the fact that it is a DAG, which makes the problem much easier.

We first describe a simple dynamic single-source reachability algorithm for DAGs. This algorithm maintains the set of vertices reachable from a fixed source and supports two types of updates. The first update removes a single edge, whereas the second one replaces a vertex v with an acyclic subgraph H. This happens in three steps. First, some number of new vertices are added to the maintained graph G (vertex v is not deleted). Second, some edges of G, whose endpoint is v may change this endpoint to one of the newly added vertices. Third, new edges can be added, but their endpoints can only be the newly added vertices and v. Also, adding these edges may not introduce cycles.

Note that both operations are in a sense decremental as once a vertex becomes unreachable from the source, it never becomes reachable again. The set of vertices reachable from the source can be easily maintained by iteratively applying the following principle: if a vertex distinct from the source has no incoming edges, it is not reachable from the source and thus can be deleted from G. The resulting algorithm runs in time which is linear in the size of the original graph and the number of vertices and edges added in the course of the algorithm. See [66] for details.

It remains to describe how to maintain $cond(G)$ and funnel the updates to $cond(G)$ to the dynamic single-source reachability algorithm for DAGs. Recall that we maintain the inter-SCC edges and strongly connected components of G explicitly. Whenever an inter-SCC edge of G is deleted, it has a corresponding edge in $cond(G)$ and to update $cond(G)$ it suffices to remove this corresponding edge. Otherwise, suppose a strongly connected component C decomposes into strongly-connected components $C_1, \ldots, C_k$. Without loss of generality assume that C_1 is the largest one (in the sense of the number of vertices) of these strongly-connected components. Thus, to update $cond(G)$ we add one new vertex for each of $C_2, C_3, \ldots, C_k$. We do not need to add a vertex corresponding to C_1 as we update the vertex corresponding to C (reuse it) so that it represents C_1. Some edges that were incident to C need to be updated as after the edge deletion their endpoint is one of $C_2, \ldots, C_k$. In order to do that, we iterate through all edges of G incident to vertices contained in $C_2, \ldots, C_k$. Similarly, by iterating through all these edges we may add all new inter-SCC edges that appear as a result of the edge deletion.

It follows that the total running time of the algorithm is dominated by the initial graph size and the total time of iterating through edges in the process of handling an edge deletion. An edge uw is considered only when the size of the strongly-connected component containing either u or w halves. Thus, we spend $O(\log n)$ time for each edge, which gives $O(n \log n)$ total time. □

Lemma 4.5.10. *Let G be a planar DAG. Suppose we can maintain the set of 1-cut edges of G when G is subject to edge deletions. Then, we can maintain the transitive reduction G^- of G in $O(n)$ additional time.*

Proof. By Lemma 2.1.1, it is indeed sufficient to decrementally maintain the 1-cut edges of G.

□

Lemma 4.5.11. *Let G be a planar digraph. Suppose we can maintain the sets of intra-SCC edges and strong bridges of G under edge deletions. Then, in $O(n)$ additional total time we can maintain the set of 2-edge-strongly-connected subgraphs of G under edge deletions.*

Proof. It is known that the maximal 2-edge-strongly-connected subgraphs of a directed graph can be found by repeatedly removing all the inter-SCC edges and strong bridges of G until none are left [39], and then taking the connected components of the obtained graph. Hence, we can use the fact that we maintain the sets of inter-SCC edges and strong bridges of G under deletions to not only compute the maximal 2-edge-strongly-connected subgraphs (by repeatedly detecting and deleting the arising inter-SCC edges and strong bridges) but also to maintain them subject to edge deletions. □

Theorem 4.5.12. *Let G be a planar digraph. In $O(n \log^5 n)$ total time we can maintain the following when G is subject to edge deletions:*

- *the strongly connected components of G,*
- *the set of vertices reachable from some source $s \in V(G)$,*
- *the transitive reduction of G if G is acyclic,*
- *the maximal 2-edge-strongly-connected subgraphs of G.*

The algorithm is Monte-Carlo randomized and correct with high probability.

Proof. Combine Theorem 4.5.7 with Lemmas 4.5.8, 4.5.9, 4.5.10 and 4.5.11. □

4.6 Faster Deterministic Decremental Single-Source Reachability

By Theorem 4.5.12, the decremental single-source reachability problem on planar graphs can be solved in $O(n \log^4 n)$ total time. The algorithm is Monte Carlo randomized.

In this section we show that by exploiting plane graph duality, we can develop a faster algorithm that is additionally deterministic. Formally, we prove the following theorem.

Theorem 4.6.1. *Let G be a planar digraph. Then, in $O(n \log^2 n \log\log n)$ total time we can maintain the sets of intra-SCC edges and strong bridges of G when G is subject to edge deletions.*

By combining Theorem 4.6.1 with Lemmas 4.5.8, 4.5.9 and 4.5.11, we obtain the following result.

Corollary 4.6.2. *Let G be a planar digraph. In $O(n \log^2 n \log\log n)$ total time we can maintain the following when G is subject to edge deletions:*

- *the strongly connected components of G,*
- *the set of vertices reachable from some source $s \in V(G)$,*
- *the maximal 2-edge-strongly-connected subgraphs of G.*

Proof. Recall that the reductions of Lemmas 4.5.8, 4.5.9, and 4.5.11 did not use randomization and required only $O(n \log n)$ additional total time. □

We use the following lemmas.

Lemma 4.6.3 (folklore, see, e.g., [57])**.** *Let G be a plane digraph and $e \in E(G)$. Then, e is an inter-SCC edge of G if and only if e^* is an intra-SCC edge of G^*.*

Lemma 4.6.4. *Let G be a plane digraph and let $e \in E(G)$ be an intra-SCC edge of G. Then, e is a strong bridge of G if and only if e^* is* not *a 1-cut edge in G^*.*

Proof. By Lemma 4.6.3, $e^* = ab$ is an inter-SCC edge of G^*, or equivalently, there is no path $b \to a$ in G^*.

Suppose first that e^* is not a 1-cut edge in G^*. Then there exists a path $P = a \to b$ in $G^* - e^*$. If all edges of P are intra-SCC in G^*, then all vertices of P, in particular a and b, are in the same strongly-connected component of G^*. This contradicts the fact that e^* is inter-SCC in G^*. Hence, there exists an inter-SCC edge $f = xy$ on P. However, the edges of P form a directed cycle in G^*/e^* so all the edges of P, in particular f, are intra-SCC in G^*/e^*. By Lemma 4.6.3, we conclude that f is intra-SCC in G, whereas it is inter-SCC in $G - e$. This proves that removing e from G breaks some strongly-connected component of G. In other words, e is a strong bridge.

Now suppose $e = uv$ is a strong bridge of G. Then, there is no path $u \to v$ in $G - e$ but there is a path $Q = v \to u$ in $G - e$. Again, there exists an edge g on Q that is inter-SCC in $G - e$ as otherwise u and v would be in the same strongly-connected component of $G - e$. Since Qe forms a cycle in G, g is intra-SCC in G. Equivalently, by Lemma 4.6.3, g^* is inter-SCC in G^* and intra-SCC in G^*/e^*. We conclude that there exists a simple cycle C in G^*/e^* that g lies on. The edges of C, however, do not form a cycle in G^*. Note that this can only happen if the edges of C form a $a \to b$ path in G^*. But clearly $e^* \notin E(C)$ and hence there exists a $a \to b$ path in $G^* - e^*$. The lemma follows. □

By Lemmas 4.6.3 and 4.6.4, instead of maintaining intra-SCC edges and strong bridges of G under edge deletions, we can equivalently maintain inter-SCC and 1-cut edges (that are simultaneously inter-SCC edges), respectively, in G^* *under contractions*. Note that since G^* undergoes contractions only, any of its edges *can never start* being an inter-SCC or 1-cut edge; it can only cease being one.

In the following we forget about the primal and dual graphs and concentrate on maintaining inter-SCC and 1-cut edges under edge contractions in a non-necessarily simple planar graph G (note that the dual graph may be not simple even if the primal is simple), i.e., we set $G := G^*$.

The first step is to handle self-loops and parallel edges separately. A self-loop of G is always an intra-SCC edge and thus will never become an inter-SCC edge. Hence, we can remove self-loops from G at the very beginning. As far as parallel edges are concerned, note that an edge is inter-SCC if and only if all its parallel edges are inter-SCC. Also observe that if an edge has a parallel edge, it is certainly not a 1-cut edge (and hence it will never start being one). From each set of pairwise parallel edges we may thus simply remove all but one, keeping in mind that if we removed at least one, then the edge that we kept is not a 1-cut edge even if our algorithm designed for simple graphs will say so. From this point on, we assume that G is initially simple.

The next step is to reduce our problem to the case when G has a simple recursive decomposition. We again apply the same set of reductions as we did in Section 4.4, and in $O(n \log n)$ time obtain a constant-factor larger graph $G' = (V', E_0 \cup E' \cup E^\times)$ that preserves reachability of G, along with a simple recursive decomposition $\mathcal{T}_{G'}$. Recall that the first step of the reduction of Lemma 4.5.1 was to get rid of the edges $E^\times$ by issuing edge deletions in the initialization phase. However, we cannot do that safely using edge contractions. Instead, for each $H \in \mathcal{T}_G$ we simply remove all edges of $E(H) \cap E^\times$ from that piece. Afterwards, the key properties, that $V(H) \cap V(G - H) \subseteq \partial H$ and ∂H lies on $O(1)$ simple and pairwise-disjoint separator curves of

both H and $G - H$, are preserved (we used the same fact in Section 4.4.2). Clearly, the graph $(V', E_0 \cup E')$ also preserves the reachability of G and the inter-SCC and 1-cut edges that lie in E' in this graph correspond to inter-SCC and 1-cut edges in G, even if these graphs undergo contractions of the corresponding edges. We thus set $G := (V', E_0 \cup E')$ in the following and assume that we are given a decomposition $\mathcal{T}_G$ as discussed above.

Now, we further simplify our problem using the following observation.

Lemma 4.6.5. *Let G be a digraph and let $uv = e \in E(G)$. Let $e' \in E(G)$ and $e' \neq e$. Then, e' is an inter-SCC edge (a 1-cut edge) of G/e if and only if e' is an inter-SCC edge (a 1-cut edge, respectively) of $G + vu$.*

Proof. Note that from the point of view of reachability information in G, contraction of an edge $e = uv$, which is effectively a merge of vertices u and v, is equivalent to making u and v strongly connected. Since $uv \in E(G)$, adding the edge vu makes u and v strongly-connected. □

By the above lemma, instead of maintaining inter-SCC edges and 1-cut edges of $G = (V, E_0)$ under contractions, we can alternatively track which edges of the initial edge set E_0 of G are inter-SCC and 1-cut under *insertions of reverse edges*.

We solve this incremental problem with basically the same approach as we used in the decremental setting in Section 4.5. For all $H \in \mathcal{T}_G$ we also maintain the graphs $\mathcal{R}(H)$, $\mathrm{In}(H)$ and $\mathrm{Ex}(H)$, and if H is a leaf piece, additionally the graphs $\mathcal{L}_e(H)$ for each $e \in E(H) \cap E_0$. These graphs are defined identically as in Sections 4.4.2 and 4.5 and can be computed inductively using Lemmas 4.4.9, 4.5.4, and 4.5.5. Whereas in Section 4.5 a deletion of an edge $uv = e \in E(G)$ was handled by deleting it from a unique leaf H_e that contained it and propagating changes using the dependencies graph (Figure 4.4), here the insertion of a reverse edge $e' = vu$ is handled by adding e' to H_e and propagating changes. By Lemma 4.5.6, we can decide which edges of G are inter-SCC and 1-cut by just looking at appropriate graphs $\mathcal{L}_e(L)$.

The only difference is that now the graphs $\mathcal{R}(\cdot)$, $\mathrm{In}(\cdot)$, $\mathrm{Ex}(\cdot)$ and $\mathcal{L}_e(\cdot)$ need to be maintained under insertions of reverse edges to G, or equivalently, under batches of incremental updates to the graphs they depend on. To achieve that, we use Theorem 4.3.1 exactly as we used Theorem 4.3.2 in Sections 4.4.2 and 4.5 for efficiently maintaining all the needed graphs. We only need to argue why for any $H \in \mathcal{T}_G$, the graphs H and $G - H$ are both ∂H-incremental (see the definition of ∂H-incremental at the beginning of Section 4.3).

Recall that G is a subgraph of a graph G' which had an actual simple recursive decomposition $\mathcal{T}_{G'}$. Hence, for each edge $uv = e \in E_0 \subseteq E(G')$, the unique leaf piece $H'_e \in \mathcal{T}_{G'}$ that contained that edge had a face f_e (a non-hole) of G' on at least one side of e in H_e (by the fact that the holes of H_e were simple and pairwise disjoint). Therefore, f_e was actually a face of all pieces $H' \in \mathcal{T}_{G'}$ that contained e. Suppose we obtain a graph G'' from G' by adding, for each edge $uv = e \in E_0$, a reverse edge $e' = vu$ to G' and all pieces $H' \in \mathcal{T}_{G'}$ such that $e \in E(H')$, and embedding it inside f_e (so that the added reverse edges embedded in a single face of G' do not cross). Then the holes of any piece piece $H'' \in \mathcal{T}_{G''}$ are the same as in the corresponding piece $H' \in \mathcal{T}_{G'}$ from which H'' was obtained. We conclude (by Lemma 4.5.2) that $\partial H''$ lies on $O(1)$ simple and pairwise-disjoint separator curves of both H'' and $G'' - H''$.

Now, note that by inserting into $H \in \mathcal{T}_G$ a reverse edge e' for a number of edges $e \in E_0 \cap E(H)$, we always maintain an invariant that $H \subseteq H''$ and $G - H \subseteq G'' - H''$. But $\partial H = \partial H''$ and the holes of H'' (equal to the initial holes of H) are separator curves of H and $G - H$ at all times. Hence, both graphs H and $G - H$ are ∂H-incremental.

Finally, in order to finish the proof of Theorem 4.6.1, observe that the total update time of

this algorithm is, by Theorem 4.3.1,

$$O\left(\sum_{H\in\mathcal{T}_G} |\partial H|^2 \log n \log\log n\right) = O(n\log^2 n \log\log n).$$

4.7 A Trade-Off for Planar Decremental Transitive Closure

In this section we show how to obtain a decremental transitive closure algorithm for planar graphs with faster query time, at a cost of slower update time and larger space consumption. The general approach we use is due to Mozes and Sommer [76] who used it to develop the so-called *cycle-MSSP* data structure, which they subsequently leveraged to show a space-time trade-off for exact distance oracles for planar graphs.

In the following, let $t \in [1, n]$ be a parameter. We first apply Lemmas 4.4.4, 4.4.5 and 4.4.6, so that we can assume that G is simple, connected, plane-embedded, triangulated and of constant degree.

We start by computing an r-division with few holes of G for $r = t$. By Theorem 2.3.6, this can be done in linear time. First, for each piece P separately we build the data structure $\mathcal{D}_P$ of Theorem 4.4.1. Each time an edge of G contained initially in P is removed, we issue the deletion of e to that data structure. This way, given $u, v \in V(P)$, we can check whether there exists a path $u \to v$ contained entirely in P in $O(\sqrt{t}\log n \log\log n)$ time. The total time needed to maintain all the data structures $\mathcal{D}_P$ is clearly $O(n/t \cdot t\log^5 n = n\log^5 n)$.

Suppose now that we want to check whether there exists a path $u \to v$ in G, where $u \in V(P)$. Observe that if no such path is contained in P, any $u \to v$ path has to go through a vertex of ∂P lying some hole h out of $O(1)$ holes of P.

For each hole h of P *separately*, we proceed as follows. Suppose without loss of generality that h is the unbounded face of P, i.e., that P lies inside h. First, $\mathrm{cyc}(h)$ is not necessarily a simple cycle. To handle this case, we proceed similarly to [58, Section 5.1]. We will modify the graph G, but let the set ∂P be fixed. Let us set $E'_h = E$. First, we duplicate the edges of h that appear twice on $\mathrm{cyc}(h)$ and include the introduced duplicates in the set $E^{\times}_h$. We subsequently split each vertex v that appears multiple times on $\mathrm{cyc}(h)$ (and modify h accordingly) until no such vertex remains, i.e., until $\mathrm{cyc}(h)$ becomes a simple cycle in G. Specifically, while some vertex v appears on $\mathrm{cyc}(h)$ $k > 1$ times, we choose some occurrence of v on h arbitrarily. Let $e_1, \ldots, e_k$ be the clockwise edge ring of v. Let e_i, e_j be the two consecutive edges of $\mathrm{cyc}(h)$ adjacent to that occurrence of v. We split v into vertices v, v' connected by a pair of directed edges $e' = vv', e'' = v'v$ (included in the set $E_{h,0}$) embedded inside the hole h so that the new edge ring of v is $e', e'', e_{j+1}, \ldots, e_{i-1}$ and the new edge ring of v' is $e'', e', e_i, \ldots, e_j$. The edges $e_i, \ldots, e_j$ change one of its endpoints from v to v' but otherwise preserve their identifiers. Afterwards, v appears on $\mathrm{cyc}(h)$ $k-1$ times whereas v' appears there once.

Note that we do not actually change the edges of h; we might only change some of their endpoints. Observe that we introduced $O(|E(h)|)$ new vertices (all of them on h) and edges in this process, and therefore the size of the graph increased by a constant factor only. Moreover, by the constant degree assumption $O(|E(h)|) = O(|V(h)|)$ and thus h has $O(|\partial P|) = O(\sqrt{t})$ vertices. When the process ends, h is simple. In the following, suppose h is simple.

Let $G_h = (V'_h, E_{h,0} \cup E'_h \cup E^{\times}_h)$ be the obtained graph. Denote by $S(v)$ the set of vertices split out of v. Note that G_h preserves the reachability of G (see Definition 4.4.2): the strongly connected components of $(V_h, E_{h,0})$ are exactly the sets $S(v)$ and for each $uv = e \in E$, there exists $u'v' = e' \in E'_h$ such that $u' \in S(u)$, $v' \in S(v)$ and $\mathrm{id}(e') = \mathrm{id}(e)$.

We again triangulate G_h without adding new vertices, as in Lemma 4.4.6 (the added edges are directed arbitrarily and included in $E^{\times}_h$).

Recall that we were originally interested in finding paths in G that go through $\partial P \cap V(h)$ when G is subject to edge deletions (here G is the original graph, whereas h is a simple cycle in G_h). Since G_h preserves the reachability of G and $\bigcup_{v \in \partial P \cap V(h)} S(v) = V(h)$, we can equivalently look for paths in G_h that go through $V(h)$, when G_h is subject to edge deletions.

Let $G_{h,1}$ ($G_{h,2}$) be the graph G_h with vertices strictly outside (strictly inside, respectively) h removed. Observe that $G_{h,1}$ is not necessarily of size $O(t)$ (as P was) since there could be other pieces located both inside h and inside holes of P. Note that both $G_{h,1}$ and $G_{h,2}$ (seen as subgraphs of G_h) have at most one hole (bounded by the cycle h; we will also call it h) and are connected. Clearly, for any $x, y \in V(G)$, a path $x \to y$ exists in G if and only if a path $x \to y$ exists in $G_{h,1} \cup G_{h,2}$. Moreover, $V(G_{h,1}) \cap V(G_{h,2}) = V(h)$.

Let us now apply the reduction of Lemma 4.4.7 to $G_{h,i}$, for $i = 1, 2$, with the following small difference. When decomposing $G_{h,i}$, we redefine a *hole* of a piece H to be a face of H that is either not a face of $G_{h,j}$, or is equal to h. For $H \in \mathcal{T}_{G_{h,i}}$, we also redefine ∂H to be equal to $V(H) \cap V(G_h - H) = V(H) \cap (V(G_{h,i} - H) \cup V(h))$ instead of $V(H) \cap V(G_{h,i} - H)$. Then h is the only hole of $G_{h,i}$ and $\partial G_{h,i} = V(h)$.

Although the construction of Theorem 4.1.4 assumed that the input graph is triangulated, the recursive decomposition procedure in that construction only required that for any input piece H to be decomposed:

- H was simple and connected,
- H had $O(1)$ simple and pairwise vertex-disjoint holes,
- all vertices of holes of H were boundary vertices of H,
- all the faces of H that were not holes had size no more than 3.

Clearly, all these requirements are met for $G_{h,i}$ if we assume our changed definition of boundary vertices and holes. Moreover, one can easily see that if we use our modified definition of boundary vertices, Lemma 4.1.2 still holds. So, we can compute the boundary vertices of the children pieces without any changes. Since h is a hole of $G_{h,i}$, for any piece $H \in \mathcal{T}_{G_{h,i}}$, the vertices $V(h) \cap V(H)$ also lie on holes of H. Therefore, Lemma 2.3.3 remains true. Since $|\partial G_{h,i}| = O(\sqrt{t}) = O(\sqrt{n})$, all the quantitative properties of the boundary vertices sets of a recursive decomposition (proved in Lemmas 4.1.6, 4.1.11 and 4.1.12) are still satisfied.

This way, in $O(n \log n)$ time we can obtain a graph $G'_{h,i}$ that preserves the reachability of $G_{h,i}$ and additionally has a simple recursive decomposition $\mathcal{T}_{G'_{h,i}}$. Moreover, the construction of Theorem 4.1.4 guaranteed that after the decomposition procedure is called on a piece, its holes (and, as a byproduct, its boundary vertices) are never altered. Therefore, without loss of generality we may assume that $G_{h,i}$ has a simple recursive decomposition $\mathcal{T}_{G_{h,i}}$ and simultaneously $G_{h,i}$ has a hole bounded by the simple cycle h.

Let $i \in \{1, 2\}$. For each $H \in \mathcal{T}_{G_{h,i}}$, we again maintain the graphs $\mathrm{In}(H)$ and $\mathcal{R}(H)$, defined as in Section 4.4, under edge deletions issued to $G_{h,i}$. For $H \in \mathcal{T}_{G_{h,i}}$, let us define $\partial^* H = \partial H \cup V(h)$.

Lemma 4.7.1. *For any $H \in \mathcal{T}_{G_{h,i}}$, $\partial^* H$ lies on $O(1)$ disjoint separator curves of both H and $G_{h,i} - H$.*

Proof. Let $Y_H = V(h) \setminus V(H)$. Since $V(h) \cap V(H) \subseteq \partial H$, we actually have $\partial^* H = \partial H \cup Y_H$ and $\partial H \cap Y_H = \emptyset$.

As proven in Fact 4.2.2 and Lemma 4.5.2, the bounding cycles of the holes of H are all separator curves of both H and $G_{h,i} - H$. Hence, since H has $O(1)$ holes, the set ∂H lies on

$O(1)$ disjoint separator curves of both H and $G_{h,i} - H$. If $Y_h = \emptyset$, $\partial H = \partial^* H$ and we would be done at this point.

We now prove that if $Y_h \neq \emptyset$, then there exists a separator curve $\mathcal{C}$ of both H and $G_{h,i} - H$, disjoint with the bounding cycles of the holes of H, such that the vertices Y_H lie on $\mathcal{C}$. Without loss of generality assume that $i = 1$, i.e., $G_{h,i}$ lies weakly inside the curve h. Since H and $G_{h,i} - H$ are both subgraphs of $G_{h,i}$, they both lie weakly inside h. Since $Y_H \cap V(H) = \emptyset$, no vertex of the holes of H is contained in Y_H. Set $\mathcal{C}$ to be a Jordan curve going through the vertices of Y_H (and no other vertices of $G_{h,i}$) in the same (cyclic) order as the vertices Y_H appear on h and such that each part of $\mathcal{C}$ connecting two neighboring vertices of Y_H in that order is embedded strictly outside h. Since h is a simple hole of $G_{h,i}$ and $G_{h,i}$ is entirely weakly inside h, such a $\mathcal{C}$ indeed exists.

Observe that H and $G_{h,i} - H$ both lie weakly inside $\mathcal{C}$ (since $G_{h,i}$ lies weakly inside $\mathcal{C}$) so $\mathcal{C}$ is a separator curve of them both. Moreover, $\mathcal{C} \cap V(H) = \emptyset$ and thus $\mathcal{C}$ is disjoint with the holes of H.

To conclude, the bounding cycles of the holes of H, together with $\mathcal{C}$, form the sought $O(1)$ disjoint separator curves of both H and $G_{h,i} - H$ that $\partial^* H$ lies on. □

We call a piece $H \in \mathcal{T}_{G_{h,i}}$ *marked* if the inequality $|\partial H'| + \sqrt{|V_0(H')|} \geq |V(h)|$ holds for all ancestors $H' \neq H$ of H in $\mathcal{T}_{G_{h,i}}$ Observe that, by this definition, if a piece H is marked, then its parent is also marked. Also, since $\partial G_{h,i} = V(h)$, the root piece $G_{h,i}$ is clearly marked.

For marked pieces H, we also maintain the graph

$$\mathrm{Ex}^*(H) = (G_{h,i} - H)^+[\partial^* H].$$

Lemma 4.7.2. *Let $H \in \mathcal{T}_{G_{h,i}}$ be a marked piece and let P, S be the parent and the sibling of H in $\mathcal{T}_{G_{h,i}}$, respectively. Then,* $\mathrm{Ex}^*(H) = (\mathrm{Ex}^*(P) \cup \mathrm{In}(S))^+[\partial^* H]$.

Proof. We only need to show that for $u, v \in \partial^* H$ there is a path $Q = u \to v$ in $G_{h,i} - H$ if and only if there is a path $u \to v$ in $\mathrm{Ex}^*(P) \cup \mathrm{In}(S)$. The easier " $\Longleftarrow$ " direction is handled analogously as in the proof of Lemma 4.5.4.

To prove the " $\Longrightarrow$ " part, recall that $G_{h,i} - H = (G_{h,i} - P) \cup S$ and split Q into maximal paths $Q_1, \ldots, Q_q$ fully contained in either $G_{h,i} - P$ or S. Let Q_j go from a_j to b_j and suppose Q_j is fully contained in S. Then a_j is either equal to u if $j = 1$, and then

$$a_j = u \in V(S) \cap \partial^* H \subseteq V(S) \cap (V(H) \cup V(h)) \subseteq V(S) \cap (V(G_{h,i} - S) \cup V(h)) = \partial S,$$

or a_j is the same as the last vertex of Q_{j-1} and hence

$$a_j \in V(G_{h,i} - P) \cap V(S) \subseteq V(G_h - S) \cap V(S) = \partial S.$$

Similarly we prove that $b_j \in \partial S$. By the definition of $\mathrm{In}(S)$, there is an edge $a_j b_j$ in $\mathrm{In}(S)$.

Analogously, for each subpath $Q_j = a_j b_j$ fully contained in $G_{h,i} - P$, if $j > 1$, then

$$a_j \in V(G_{h,i} - P) \cap V(S) \subseteq V(G_h - P) \cap V(P) = \partial P \subseteq \partial^* P,$$

or otherwise $a_j = a_1 = u \in V(G_{h,i} - P) \cap \partial^* H = V(G_{h,i} - P) \cap (\partial H \cup V(h))$. If $u \in V(h)$, then clearly $u \in \partial^* P$. Otherwise, we have

$$u \in V(G_{h,i} - P) \cap \partial H \subseteq V(G_h - P) \cap V(H) \subseteq V(G_h - P) \cap V(P) = \partial P \subseteq \partial^* P.$$

Similarly we prove that $b_j \in \partial^* P$. By the definition of $\mathrm{Ex}^*(P)$, there is an edge $a_j b_j$ in $\mathrm{Ex}^*(P)$.

We conclude that there is a directed path $u a_2 \ldots a_q v$ in $\mathrm{Ex}^*(P) \cup \mathrm{In}(S)$. □

By Lemmas 4.7.1 and 4.7.2, we can maintain the graphs $\mathrm{Ex}^*(H)$ for marked pieces H inductively using Theorem 4.3.2. The total time needed to maintain these graphs is

$$
\begin{aligned}
O\left(\sum_{\substack{H\in\mathcal{T}_{G_{h,i}}\\ H \text{ marked}\\ H=\mathrm{child}_i(P)\\ S=\mathrm{child}_{3-i}(P)}} (|\partial^* P|^2 + |\partial S|^2)\log^4 n\right) &= O\left(\sum_{\substack{H\in\mathcal{T}_{G_{h,i}}\\ H \text{ marked}\\ H=\mathrm{child}_i(P)}} |\partial^* P|^2 \log^4 n\right) + O(n\log^5 n)\\
&= O\left(\sum_{\substack{H\in\mathcal{T}_{G_{h,i}}\\ H \text{ marked}\\ H=\mathrm{child}_i(P)}} (|\partial P| + |V(h)|)^2 \log^4 n\right) + O(n\log^5 n)\\
&= O\left(\sum_{\substack{H\in\mathcal{T}_{G_{h,i}}\\ H \text{ marked}\\ H=\mathrm{child}_i(P)}} \left(|\partial P| + \sqrt{|V_0(P)|}\right)^2 \log^4 n\right) + O(n\log^5 n)\\
&= O\left(\log^4 n \cdot \sum_{H\in\mathcal{T}_{G_{h,i}}} (|\partial H|^2 + |V_0(H)|)\right) + O(n\log^5 n)\\
&= O(n\log^5 n).
\end{aligned}
$$

In the last step we used the fact that, by Lemma 4.1.7, $\sum_{H\in\mathcal{T}_{G_{h,i}}} \left(|\partial H|^2 + |V_0(H)|\right) = O(n\log n)$.

Similarly as in Section 4.4.2, for $H \in \mathcal{T}_{G_{h,i}}$ let us define $H_{h,i,w}$ to be the leftmost leaf piece in the subtree $\mathcal{T}_{G_{h,i}}(H)$ such that $w \in V(H_{h,i,w})$. For $H \in \mathcal{T}_{G_{h,i}}$ such that $w \in V(H)$, let

$$
\mathcal{R}^w_{h,i}(H) = \mathcal{R}(H_{h,i,w}) \cup \bigcup_{\substack{H'\in\mathcal{T}_{G_{h,i}}(H)\\ H'\neq H_{h,i,w}\\ H_{h,i,w}\in\mathcal{T}_{G_{h,i}}(H')}} \mathrm{In}(\mathrm{child}_1(H')) \cup \mathrm{In}(\mathrm{child}_2(H')).
$$

These definitions would actually match the definitions of H_w and $\mathcal{R}^w(H)$ of Section 4.4.2 if we substituted $G := G_{h,i}$. Also, by proceeding similarly as in the proof of Lemma 4.4.13 and taking into account the bound from Lemma 4.1.11, we can easily prove that

$$
|V(\mathcal{R}^w_{h,i}(H))| = O\left(\sqrt{|V_0(H)|} + |\partial H|\right).
$$

Analogously as in Section 4.4.2, by Lemma 4.4.11, for $w \in V(H)$ and $y \in \partial H \cup \mathrm{Sep}(H)$, where $H \in \mathcal{T}_{G_{h,i}}$, a path $w \to y$ ($y \to w$) exists in H if and only if a path $w \to y$ ($y \to w$, respectively) exists in $\mathcal{R}^w_{h,i}(H)$.

Let us now define the graph $G_{h,u,v}$ to be a union of certain graphs we maintain. For $i = 1, 2$, include in $G_{h,u,v}$ the following graphs:

- $\mathrm{In}(G_{h,i})$,
- For $w \in \{u, v\}$, if $w \in V(G_{h,i})$, let $A_{h,i,w}$ be the nearest weak ancestor of $H_{h,i,w}$ in $\mathcal{T}_{G_{h,i}}$ that is marked. Then, include in $G_{h,u,v}$ the graphs:
  - $\mathcal{R}^w_{h,i}(A_{h,i,w})$,

– $\mathrm{Ex}^*(A_{h,i,w})$.

We now show that the number of vertices of $G_{h,u,v}$ if $O(\sqrt{t})$. First, recall that $\partial G_{h,i} = O(\sqrt{t})$. Since $A_{h,i,w}$ is the nearest marked ancestor, either $A_{h,i,w} = H_{h,i,w}$ or $|\partial A_{h,i,w}| + \sqrt{|V_0(A_{h,i,w})|} < |V(h)| = O(\sqrt{t})$. In both cases $|\partial A_{h,i,w}| = O(\sqrt{t})$. By that we conclude $|\partial^* A_{h,i,w}| \leq |\partial A_{h,i,w}| + |V(h)| = O(\sqrt{t})$ and $|V(\mathcal{R}^w_{h,i}(A_{h,i,w}))| = O(\sqrt{t})$. Hence, we indeed have $|V(G_{h,u,v})| = O(\sqrt{t})$.

Lemma 4.7.3. *Let $u, v \in V(G_h)$. Then the following conditions hold.*

1. *If v is reachable from u in $G_{h,u,v}$, then there is a path $u \to v$ in G_h.*
2. *If a path $u \to v$ going through a vertex of $V(h)$ exists in G, then there exists a path $u \to v$ in $G_{h,u,v}$.*

Proof. Let us first note that a path $u \to v$ exists in G_h if and only if it exists in $G_{h,1} \cup G_{h,2}$. Then, the first part is clear as each edge xy of $G_{h,u,v}$ certifies that a path $x \to y$ exists in some subgraph of either $G_{h,1}$ or $G_{h,2}$.

To prove the second part, consider some simple path $P = u \to v$ going through a vertex $V(h)$ and split it into maximal paths $P_1, \ldots, P_p$ such that the internal (i.e., non-endpoint) vertices of each P_j are not contained in $V(h)$. If P_j consists of a single edge, then it is clearly contained in $G_{h,1}$ or $G_{h,2}$ (or possibly in both). Otherwise, P_j contains a vertex of $V(G_h) \setminus V(h)$, and thus P_j is contained in exactly one of $G_{h,1}$ or $G_{h,2}$. Moreover, for $i \in [2, p)$, P_j has both endpoints in $V(h)$.

Fix some j and suppose without loss of generality that a path $P_j = x_j \to y_j$ is contained in $G_{h,i}$. If $j \in [2, p)$, then $\{x_j, y_j\} \subseteq V(h) \subseteq \partial G_{h,i}$, and it follows that there is an edge $x_j y_j$ in $\mathrm{In}(G_{h,i}) \subseteq G_{h,u,v}$.

Now consider $j = 1$ (the case $j = p$ is symmetric). Then $x_1 = u$. We cannot have $\{x_1, y_1\} \cap V(h) = \emptyset$ since $u \to v$ is supposed to pass through a vertex of $V(h)$ and no internal vertices of P_1 belong to $V(h)$. Assume without loss of generality that $y_1 \in V(h)$ (this is certainly the case when $p \geq 2$, but if $p = 1$, it might happen that $x_1 = u \in V(h)$ and $y_1 \notin V(h)$ – then we follow the symmetric "$j = p$" proof).

Split P_1 into maximal paths $Q_1, \ldots, Q_q$, where $Q_k = a_k \to b_k$, contained entirely in either $A_{h,i,u}$ or $G_{h,i} - A_{h,i,u}$. For any $k < q$,

$$b_k \in V(A_{h,i,u}) \cap V(G_{h,i} - A_{h,i,u}) \subseteq V(A_{h,i,u}) \cap V(G_h - A_{h,i,u}) = \partial A_{h,i,u}.$$

Similarly, we have $a_k \in \partial A_{h,i,u}$ for $k > 1$.

Assume $Q_k = a_k \to b_k$ is contained entirely in $A_{h,i,u}$. Note that if $k = q$, then $b_q \in V(h) \cap V(A_{h,i,u}) \subseteq \partial A_{h,i,u}$. Consequently, $b_k \in \partial A_{h,i,u}$ for all k. Now, if $k = 1$, then $a_1 = u$ and hence by Lemma 4.4.11 for $\mathcal{R}^u_{h,i}(A_{h,i,u})$, it follows that there exists a path $(u = a_1) \to b_1$ in $\mathcal{R}^u_{h,i}(A_{h,i,u}) \subseteq G_{h,u,v}$. If $k > 1$, then $\{a_k, b_k\} \in \partial A_{h,i,u}$, so there is a path $a_k \to b_k$ in $\mathcal{R}(A_{h,i,u})$. Observe that if $A_{h,i,u}$ is a leaf piece, then $\mathcal{R}^u_{h,i}(A_{h,i,u}) = \mathcal{R}(A_{h,i,u}) = A_{h,i,u}$. Otherwise, since $\mathcal{R}(A_{h,i,u}) = (\mathrm{In}(\mathrm{child}_1(A_{h,i,u})) \cup \mathrm{In}(\mathrm{child}_2(A_{h,i,u}))^+$, a path $a_k \to b_k$ exists in $\mathrm{In}(\mathrm{child}_1(A_{h,i,u})) \cup \mathrm{In}(\mathrm{child}_2(A_{h,i,u})) \subseteq \mathcal{R}^u_{h,i}(A_{h,i,u})$. We conclude that indeed if Q_k is contained entirely in $A_{h,i,u}$, then there exists a path $a_k \to b_k$ in $G_{h,i,u}$.

Now suppose $Q_k = a_k \to b_k$ is contained entirely in $G_{h,i} - A_{h,i,u}$. We show that $a_k b_k$ is an edge of $\mathrm{Ex}^*(A_{h,i,u})$ in this case.

First note that if $k = 1$, then by definition of $H_{h,i,u}$ we have $u \in V(H_{h,i,u})$. Thus,

$$a_k = a_1 = u \in V(H_{h,i,u}) \cap V(G_{h,i} - A_{h,i,u}) \subseteq V(A_{h,i,u}) \cap V(G_h - A_{h,i,u}) = \partial A_{h,i,u}.$$

So in fact $a_k \in \partial A_{h,i,u} \subseteq \partial^* A_{h,i,u}$ regardless of k. If $k = q$, then $b_k \in V(h)$ and we trivially have $b_k \in \partial A_{h,i,u} \cup V(h) = \partial^* A_{h,i,u}$. We conclude that $\{a_k, b_k\} \subseteq \partial^* A_{h,i,u}$ regardless of k, and therefore $a_k b_k$ is indeed an edge of $\mathrm{Ex}^*(A_{h,i,u}) \subseteq G_{h,u,v}$.

Since for each Q_k there is a path $a_k \to b_k$ in $G_{h,u,v}$, $G_{h,u,v}$ contains a path $u = x_1 \to y_1$ as well. Finally, $G_{h,u,v}$ contains a path $x_j \to y_j$ for any $j = 1, \ldots, p$, and thus it also contains a path $u \to v$. □

Recall that in order to answer a general query about a path $u \to v$, where $u \in V(P)$, we first look for such a path in P, by querying $\mathcal{D}_P$, in $O(\sqrt{t} \log n \log\log n)$ time. Then, for each of $O(1)$ holes h of P, we look for a path $u \to v$ in $G_{h,u,v}$. By Lemma 4.7.3, this is enough to decide whether $u \to v$ exists in the original graph G. Observe that each $G_{h,u,v}$ is composed of $O(1)$ edges plus a number of graphs of the form $H^+[U]$, where U lies on $O(1)$ simple and pairwise-disjoint separator curves of a plane graph H. Hence, by Corollary 4.3.7, we can search for a $u \to v$ path in $G_{h,u,v}$ in $O(\sqrt{t} \log n \log\log n)$ time.

The total time needed to construct each $\mathcal{T}_{G_{h,i}}$ and maintain all the needed graphs $\mathrm{In}(\cdot)$, $\mathcal{R}(\cdot)$, $\mathrm{Ex}^*(\cdot)$, is $O\left(\sum_{H \in \mathcal{T}_{G_{h,i}}} |\partial H|^2 \log^4 n\right) = O(n \log^5 n)$. The number of valid pairs (h, i) is clearly $O(n/t)$ so the total update time (and the required space) of the whole data structure is $O(n^2/t \cdot \log^5 n)$.

Theorem 4.7.4. *For any $t \in [1, n]$, there exists a decremental transitive closure algorithm for planar graphs with $O(n^2/t \cdot \log^5 n)$ total update time and $O(\sqrt{t} \log n \log\log n)$ query time.*

The algorithm is Monte-Carlo randomized and is correct with high probability.

Corollary 4.7.5. *There exists a decremental transitive closure algorithm for planar graphs with $\widetilde{O}(n^{1/3})$ amortized update time and $\widetilde{O}(n^{1/3})$ query time.*

The algorithm is Monte-Carlo randomized and is correct with high probability.

Proof. Set $t = n^{2/3}$ and apply Theorem 4.7.4. □

Chapter 5

Improved Algorithm for Shortest Paths in Dense Distance Graphs

In this chapter we study the problem of computing shortest paths in so-called *dense distance graphs*, a basic building block for designing efficient planar graph algorithms.

For $i = 1, \ldots, q$, let G_i be a plane, non-negatively *weighted* digraph. Let U_i be a subset of vertices of G_i lying on $O(1)$ faces of G_i. We define a *distance clique* of G_i, denoted $\mathrm{DC}(G_i)$, to be a complete graph on U_i such that for each $uv \in E(\mathrm{DC}(G_i))$, $w_{\mathrm{DC}(G_i)}(uv) = \delta_G(u, v)$.

A dense distance graph is a union of possibly many unrelated distance cliques. Let

$$\mathrm{DDG} = \bigcup_{i=1}^{q} \mathrm{DC}(G_i).$$

We also set $V = U_1 \cup \ldots \cup U_q$ and $n = |V|$.

Formally, we solve the following abstract problem. Suppose we are given q distance cliques $\mathrm{DC}(G_1), \ldots, \mathrm{DC}(G_q)$ explicitly. Assume that we are allowed to preprocess each $\mathrm{DC}(G_i)$ once in time asymptotically no more than the time used to construct it, which is clearly $\Omega(|U_i|^2)$. To the best of our knowledge, in all known applications when U_i does not necessarily lie on a *single* face of G_i this time is $\Omega((|V(G_i)| + |U_i|^2) \log |V(G_i)|)$, which is the time needed to compute the multiple-source shortest-path data structure [13, 62] for $O(1)$ faces of G_i and compute $|U_i|^2$ pairwise distances using it. After the preprocessing stage we may need to handle multiple single-source shortest-path computations on DDG. Specifically, given $s \in V$, we are asked to compute the distances $\delta_{\mathrm{DDG}}(s, v)$ for all $v \in V$.

Note that this problem can be also viewed as a generalization of the problem from Section 4.3.2. There, instead of distance cliques, we were given reachability cliques and we also wanted to compute distances from a single-source in a union of these cliques. We could in fact reduce the problem of Section 4.3.2 to computing shortest paths in DDG if we replaced unweighted edges with 0-weight edges. Then, for each $uv \in E(\mathrm{DC}(G_i))$ we would have $w_{\mathrm{DC}(G_i)}(uv) = 0$ if $uv \in G_i^+[U_i]$ and $w_{\mathrm{DC}(G_i)}(uv) = \infty$ otherwise. However, the special structure of the graphs $G_i^+[U_i]$ allowed us to develop a simpler and faster algorithm for that case in Section 4.3.2

The main goal of this chapter is to prove the following theorem, which is an improvement over the original algorithm of Fakcharoenphol and Rao [27], so-called FR-Dijkstra.

Theorem 5.0.1. *The single-source shortest paths computations in DDG can be performed in* $O\left(\sum_{i=1}^{q} |U_i| \frac{\log^2 n}{\log^2 \log n}\right)$ *time. The required preprocessing time per each* G_i *is* $O(|U_i|^2)$ *if* U_i *lies on a single face of* G_i*, and* $O\left((|V(G_i)| + |U_i|^2) \log |V(G_i)|\right)$ *otherwise.*

In the following we will focus on the case when each U_i in fact lies on $O(1)$ *simple and pairwise-disjoint* faces of G_i. This is without much loss of generality: for some important applications (e.g., [7]) this is actually sufficient, whereas for others (e.g., [8, 67]), depending on the application, we may appropriately extend the graphs (and subgraphs) that we work on, as we did multiple times in Chapter 4 (see also [58]), so that this condition is eventually satisfied.

Outline. This chapter is organized as follows. In Section 5.1 we introduce the matrix notation that we use and state some important properties of Monge matrices, a central notion to this chapter.

Since the main findings of this chapter are technically involved, in Section 5.2 we start with an overview of our shortest paths algorithm and sketch how it reduces to a certain data-structural problem about reporting column minima of a *staircase* Monge matrix in an online fashion. We also discuss the main ideas behind out improved data structure for this problem.

In Sections 5.3, 5.4, and 5.5, we develop increasingly more powerful data structures for reporting column minima in online Monge matrices. Each of these data structures is used in a black-box manner in the following section.

In Section 5.6 we give the details of our shortest paths algorithm. In Section 5.7 we discuss how the algorithm of Section 5.6 can be useful in the case of arbitrary (i.e., possibly negative) edge weights.

Section 5.8 covers the most important applications of FR-Dijkstra and discusses how our result influences these applications.

5.1 Monge Matrices and Their Minima

In this chapter we define a *matrix* to be a partial function $\mathcal{M} : R \times C \to \mathbb{R}$ where R (called *rows*) and C (called *columns*) are some totally ordered finite sets. Set $R = \{r_1, \ldots, r_k\}$ and $C = \{c_1, \ldots, c_l\}$ where $r_1 \leq \ldots \leq r_k$ and $c_1 \leq \ldots \leq c_l$. If for $r_i, r_j \in R$ we have $r_i \leq r_j$, we also say that r_i is (weakly) *above* r_j and r_j is (weakly) *below* r_i. Similarly, when we have $c_i < c_j$, we say that c_i is *to the left* of c_j and c_j is *to the right* of c_i.

For $r \in R$ and $c \in C$, we denote by $\mathcal{M}_{r,c}$ an *element* of $\mathcal{M}$. An element is the value of $\mathcal{M}$ on pair (r, c), if defined.

For $R' \subseteq R$ and $C' \subseteq C$ we define $\mathcal{M}(R', C')$ to be a *submatrix* of $\mathcal{M}$. $\mathcal{M}(R', C')$ is a partial function on $R' \times C'$ satisfying $\mathcal{M}(R', C')_{r,c} = \mathcal{M}_{r,c}$ for any $(r, c) \in R' \times C'$ such that $\mathcal{M}_{r,c}$ is defined. We sometimes abuse this notation by writing $\mathcal{M}(R', c')$ or $\mathcal{M}(r', C')$ when R' or C' are single-element, i.e., when $R' = \{r'\}$ or $C' = \{c'\}$.

The *minimum* of a matrix $\min\{\mathcal{M}\}$ is defined as the minimum value of the partial function $\mathcal{M}$. The *column minimum* of $\mathcal{M}$ in column c is defined as $\min\{\mathcal{M}(R, \{c\})\}$.

We call a matrix $\mathcal{M}$ *rectangular* if $\mathcal{M}_{r,c}$ is defined for every $r \in R$ and $c \in C$. A matrix is called *staircase* (*flipped staircase*) if $|R| = |C|$ and $\mathcal{M}_{r_i,c_j}$ is defined if and only if $i \leq j$ ($i \geq j$ respectively).

Finally, a *subrectangle* of $\mathcal{M}$ is a rectangular matrix $\mathcal{M}(\{r_a, \ldots, r_b\}, \{c_x, \ldots, c_y\})$ where $1 \leq a \leq b \leq k$, $1 \leq x \leq y \leq l$. We define a *subrow* to be a subrectangle with a single row.

For a matrix $\mathcal{M}$ and a function $d : R \to \mathbb{R}$, define the *offset matrix* $\mathrm{off}(\mathcal{M}, d)$ to be a matrix $\mathcal{M}'$ such that for all r, c, for which $\mathcal{M}_{r,c}$ is defined, we have $\mathcal{M}'_{r,c} = \mathcal{M}_{r,c} + d(r)$.

We say that a matrix $\mathcal{M}$ with rows R and columns C is a *Monge matrix*, if for each $r_1, r_2 \in R$, $r_1 \leq r_2$ and $c_1, c_2 \in C$, $c_1 \leq c_2$, such that all elements $\mathcal{M}_{r_1,c_1}, \mathcal{M}_{r_1,c_2}, \mathcal{M}_{r_2,c_1}, \mathcal{M}_{r_2,c_2}$ are defined, the *Monge property* holds, i.e., we have

$$\mathcal{M}_{r_2,c_1} + \mathcal{M}_{r_1,c_2} \leq \mathcal{M}_{r_1,c_1} + \mathcal{M}_{r_2,c_2}.$$

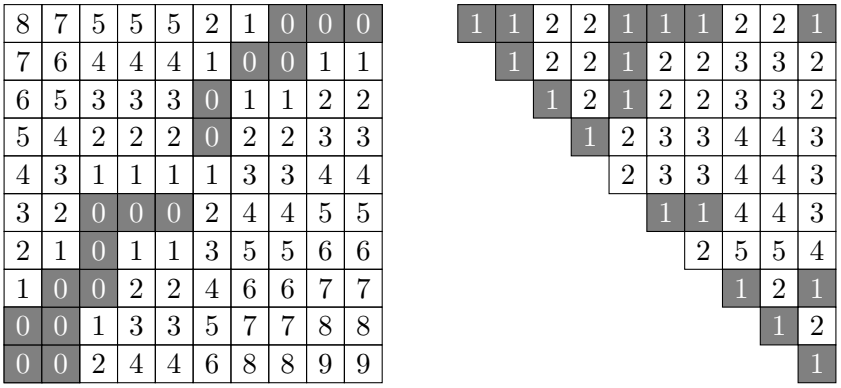

Figure 5.1: Example 10×10 Monge matrices: a rectangular one to the left and a staircase one to the right. The gray cells contain the column minima of the respective columns.

Fact 5.1.1. *Let $\mathcal{M}$ be a Monge matrix. For any $R' \subseteq R$ and $C' \subseteq C$, $\mathcal{M}(R', C')$ is also a Monge matrix.*

Fact 5.1.2. *Let $\mathcal{M}$ be a rectangular Monge matrix. Assume that for some $c \in C$ and $r \in R$, $\mathcal{M}_{r,c}$ is a column minimum of c. Then:*

1. *for each column $c^- < c$, there exists a row $r^- \geq r$ such that $\mathcal{M}_{r^-,c^-}$ is a column minimum of c^-,*
2. *for each column $c^+ > c$, there exists a row $r^+ \leq r$ such that $\mathcal{M}_{r^+,c^+}$ is a column minimum of c^+.*

Proof. We only prove the former claim as the proof of the latter is analogous. Suppose all the column minima of c^- lie in rows above r. Then, by the Monge property, for some $r' < r$ we have $\mathcal{M}_{r,c^-} + \mathcal{M}_{r',c} \leq \mathcal{M}_{r',c^-} + \mathcal{M}_{r,c}$. But $\mathcal{M}_{r',c^-} < \mathcal{M}_{r,c^-}$ and $\mathcal{M}_{r,c} \leq \mathcal{M}_{r',c}$, so we conclude $\mathcal{M}_{r,c^-} + \mathcal{M}_{r',c} < \mathcal{M}_{r,c^-} + \mathcal{M}_{r',c}$, a contradiction. □

Fact 5.1.3. *Let $\mathcal{M}$ be a Monge matrix with rows R and let $d : R \to \mathbb{R}$. Then, off$(\mathcal{M}, d)$ is also a Monge matrix.*

Proof. Let $r_1 \leq r_2$ be two rows and let $c_1 \leq c_2$ be two columns such that $\mathcal{M}_{r_1,c_1}$,M_{r_1,c_2},M_{r_2,c_1} and M_{r_2,c_2} are all defined. We have

$$\begin{aligned}\text{off}(\mathcal{M}, d)_{r_2,c_1} + \text{off}(\mathcal{M}, d)_{r_1,c_2} &= \mathcal{M}_{r_2,c_1} + \mathcal{M}_{r_1,c_2} + d(r_1) + d(r_2) \\ &\leq \mathcal{M}_{r_1,c_1} + \mathcal{M}_{r_2,c_2} + d(r_1) + d(r_2) \\ &= \text{off}(\mathcal{M}, d)_{r_1,c_1} + \text{off}(\mathcal{M}, d)_{r_2,c_2}.\end{aligned}$$

□

Fact 5.1.4. *Let $\mathcal{M}$ be a rectangular Monge matrix and assume R is partitioned into disjoint blocks $\mathcal{R} = R_1, \ldots, R_a$ such that each R_i is a contiguous group of subsequent rows and each R_i is above R_{i+1}. Assume also that the set C is partitioned into blocks $\mathcal{C} = C_1, \ldots, C_b$ so that C_i is to the left of C_{i+1}. Then, a matrix $\mathcal{M}'$ with rows $\mathcal{R}$ and columns $\mathcal{C}$ defined as*

$$\mathcal{M}'_{R_i,C_j} = \min\{\mathcal{M}(R_i, C_j)\},$$

is also a Monge matrix.

Proof. Let i_1, i_2, j_1, j_2 be such that $1 \leq i_1 \leq i_2 \leq a$ and $1 \leq j_1 \leq j_2 \leq b$. We need to prove

$$\mathcal{M}'_{R_{i_2},C_{j_1}} + \mathcal{M}'_{R_{i_1},C_{j_2}} \leq \mathcal{M}'_{R_{i_1},C_{j_1}} + \mathcal{M}'_{R_{i_2},C_{j_2}}.$$

If $i_1 = i_2$ or $j_1 = j_2$, then the inequality is trivial so suppose $i_1 < i_2$ and $j_1 < j_2$.

Let $\mathcal{M}_{r_1,c_1}$ be any minimum element of $\mathcal{M}(R_{i_1}, C_{j_1})$ and let $\mathcal{M}_{r_2,c_2}$ be any minimum element of $\mathcal{M}(R_{i_2}, C_{j_2})$. Since R_{i_1} is above R_{i_2} and C_{j_1} is to the left of C_{j_2}, $r_1 < r_2$ and $c_1 < c_2$. By the Monge property of $\mathcal{M}$ and the minimality of the values $\mathcal{M}_{r_1,c_1}$ and $\mathcal{M}_{r_2,c_2}$ we obtain:

$$\begin{aligned}
\mathcal{M}'_{R_{i_2},C_{j_1}} + \mathcal{M}'_{R_{i_1},C_{j_2}} &= \min\{\mathcal{M}(R_{i_2}, C_{j_1})\} + \min\{\mathcal{M}(R_{i_1}, C_{j_2})\} \\
&\leq \mathcal{M}_{r_2,c_1} + \mathcal{M}_{r_1,c_2} \\
&\leq \mathcal{M}_{r_1,c_1} + \mathcal{M}_{r_2,c_2} \\
&= \min\{\mathcal{M}(R_{i_1}, C_{j_1})\} + \min\{\mathcal{M}(R_{i_2}, C_{j_2})\} \\
&= \mathcal{M}'_{R_{i_1},C_{j_1}} + \mathcal{M}'_{R_{i_2},C_{j_2}}.
\end{aligned}$$

□

Fact 5.1.5. *Let $\mathcal{M}$ be a rectangular Monge matrix. For $r \in R$, define $C_r \in C$ to be the set of columns having one of their column minima in row r. Then:*

1. *C_r is contiguous, that is, either $C_r = \emptyset$ or $C_r = \{c_a, \ldots, c_b\}$ for some $1 \leq a \leq b \leq l$.*
2. *If $r_1 > r_2$ and both C_{r_1} and C_{r_2} are non-empty, i.e., $C_{r_1} = \{c_{a_1}, \ldots, c_{b_1}\}$ and $C_{r_2} = \{c_{a_2}, \ldots, c_{b_2}\}$ for some $a_1 \leq b_1$, $a_2 \leq b_2$, then $a_1 \leq a_2$ and $b_1 \leq b_2$.*

Proof. Let us start with the former claim. Suppose the contrary, i.e., that there exist x, y, z, $1 \leq x < y < z \leq l$ such that $c_x \in C_r$, $c_y \notin C_r$ and $c_z \in C_r$. Let a column minimum of c_y lie in some row r'. If $r' < r$, then $\mathcal{M}_{r,c_y} + \mathcal{M}_{r',c_z} > \mathcal{M}_{r',c_y} + \mathcal{M}_{r,c_z}$, a contradiction. Otherwise, if $r' > r$, we conclude $\mathcal{M}_{r',c_x} + \mathcal{M}_{r,c_y} > \mathcal{M}_{r,c_x} + \mathcal{M}_{r',c_y}$, also a contradiction.

For the latter claim, we only prove $a_1 \leq a_2$ as proving $b_1 \leq b_2$ is analogous. Suppose the contrary, i.e., $a_2 < a_1$. Then, by $\mathcal{M}_{r_1,c_{a_2}} > \mathcal{M}_{r_1,c_{a_1}}$ and $\mathcal{M}_{r_2,c_{a_1}} \geq \mathcal{M}_{r_2,c_{a_2}}$, we get $\mathcal{M}_{r_1,c_{a_2}} + \mathcal{M}_{r_2,c_{a_1}} > \mathcal{M}_{r_2,c_{a_2}} + \mathcal{M}_{r_1,c_{a_1}}$, a contradiction with the Monge property. □

Remark 5.1.6. *The statements of Facts 5.1.2 and 5.1.5 could be simplified if we either assumed that the column minima in the considered Monge matrices are unique, or introduced some tie-breaking rule. However, this would lead to a number of similar assumptions in terms of priority queue keys and path lengths in the following sections, which in turn would complicate the description. Thus, we do not use any simplifying assumptions about the column minima.*

5.2 Shortest Paths in a Dense Distance Graph: an Overview

The single-source shortest paths in DDG are computed with an optimized implementation of Dijkstra's algorithm. Recall that Dijkstra's algorithm run from the source s grows a set S of *visited* vertices of the graph, such that the lengths $d(v)$ of the shortest paths $s \to v$ for $v \in S$ are already known. Initially $S = \{s\}$ and we repeatedly choose a vertex $y \in V \setminus S$ such that the value (a distance estimate) $z(y) := \min_{x \in S}\{d(x) + w_{\mathrm{DDG}}(xy) : xy = e \in E(DDG)\}$ is smallest. The vertex y is then added to S with $d(y) = z(y)$. The vertices $y \in V \setminus S$ are typically stored in a priority queue with keys $z(y)$, which allows to choose the best y efficiently.

Since the vertices of U_i lie on $O(1)$ faces of a planar digraph G_i, we can exploit the fact that many of the shortest paths represented by $\mathrm{DC}(G_i)$ have to cross. Formally, this is captured by the following lemma proved in Section 5.6.

Lemma 5.2.1 ([77]). *Each DC(G_i) can be decomposed into $O(1)$ (possibly flipped) staircase Monge matrices D_i of at most $|U_i|$ rows and columns. For each $u, v \in U_i$ we have:*

- *for each $\mathcal{M} \in D_i$ such that $\mathcal{M}_{u,v}$ is defined, $\mathcal{M}_{u,v} \geq w_{DC(G_i)}(uv)$.*
- *there exists $\mathcal{M} \in D_i$ such that $\mathcal{M}_{u,v}$ is defined and $\mathcal{M}_{u,v} = w_{DC(G_i)}(uv)$.*

The decomposition can be computed in $O(|U_i|^2)$ time if U_i lies on a single face of G_i, and in $O((|V(G_i)| + |U_i|^2) \log |V(G_i)|)$ time otherwise.

In other words, the weight matrix of DC(G_i) can be partitioned into a constant number of staircase Monge matrices. Consequently, a natural approach to maintaining the minimum distance estimates $z(y)$, for $y \notin S$, is to split the work needed to accomplish this task between the individual matrices $\mathcal{M} \in \bigcup_{i=1}^{q} D_i$ that encode the edges of DDG. Then, it is sufficient to design a data structure reporting the column minima of the offset matrix off$(\mathcal{M}, d)$ in an online fashion. Specifically, the data structure has to handle *row activations* intermixed with extractions of the column minima in non-decreasing order. Once Dijkstra's algorithm establishes the distance $d(v)$ to some vertex v, the row of off$(\mathcal{M}, d)$ corresponding to v is activated and becomes available to the data structure. This row contains values $d(v) + \mathcal{M}_{v,w}$, where $\mathcal{M}_{v,w}$ is, by Lemma 5.2.1, no less than the length of the edge vw in DDG. Alternatively, a minimum in some column corresponding to v (in the revealed part of off$(\mathcal{M}, d)$) may be used by Dijkstra's algorithm to establish a new distance label $z(v) = d(v)$, even though not all rows of off$(\mathcal{M}, d)$ have been revealed so far. In this case, we can guarantee that all the inactive rows of off$(\mathcal{M}, d)$ contain entries not smaller than $d(v)$, and hence we can safely extract the column minimum of off$(\mathcal{M}, d)$.

Such an approach was also used by Fakcharoenphol and Rao [27] and Mozes et al. [75], who both dealt with staircase Monge matrices by using a recursive partition into *square* Monge matrices, which are easier to handle. In particular, Fakcharoenphol and Rao showed that a sequence of row activations and column minima extractions can be performed on an $m \times m$ square Monge matrix in $O(m \log m)$ time. The recursive partition assigns each row and column to $O(\log m)$ square Monge matrices. As a result, in [27] the total time for handling all the square matrices is $O(m \log^2 m)$. We provide the details and the pseudocode of the above shortest path algorithm in Section 5.6.

The Data Structure. An improved data structure reporting the column minima of an online offset staircase Monge matrix is achieved in three steps, presented in the following three sections in a bottom-up fashion. Below we sketch the main ideas behind these steps.

Our first component is a refined data structure for handling row activations and column minima extractions on a *rectangular* Monge matrix described in Section 5.3. We show a data structure supporting any sequence of operations on a $k \times l$ matrix in $O\left(k \frac{\log m}{\log \log m} + l \log m\right)$ total time, where $m = \max(k, l)$.

The second step is to relax the requirements posed on a data structure handling rectangular $k \times l$ Monge matrices. It is motivated by the following observation. Let $\Delta > 0$ be an integer. Imagine we have found the minima of l/Δ evenly spread *pivot* columns $c_1, \ldots, c_{l/\Delta}$. Denote by $r_1, \ldots, r_{l/\Delta}$ the rows containing the corresponding minima. Fact 5.1.2 implies that for any column c' lying between c_i and c_{i+1}, we only have to look for a minimum of c' in rows $r_i, \ldots, r_{i+1}$. Thus, the minima in the remaining columns can be found in $O(k\Delta + l)$ total time. In Section 5.4 we show how to adapt this idea to an online setting that fits our needs. The columns are partitioned into $O(l/\Delta)$ *blocks* of size at most Δ. Each block is conceptually contracted to a single column: an entry in row r is defined as the minimum in row r over the contracted

columns. For sufficiently small values of Δ, such a minimum can be computed in $O(1)$ time using the data structure of [34]. Locating a block minimum can be seen as an introduction of a new pivot column. We handle the block matrix, which by Fact 5.1.4 is also Monge, with the data structure of Section 5.3 and prove that the total time needed to correctly report all the column minima is $O\left(k\frac{\log m}{\log\log m} + k\Delta + l + \frac{l}{\Delta}\log m\right)$. In particular, for $\Delta = \log^{1-\epsilon} m$ this bound becomes $O\left(k\frac{\log m}{\log\log m} + l\log^{\epsilon} m\right)$.

Finally, in Section 5.5 we exploit the asymmetry of per-row and per-column costs of the developed block data structure for rectangular matrices, by using a different partition (than in [27]) of an $m \times m$ staircase Monge matrix. Our partition is biased towards columns, i.e., the matrix is split into *rectangular* (as opposed to square) Monge matrices, each with roughly poly-logarithmically more columns than rows. Consequently, the total number of rows in these matrices is $O\left(m\frac{\log m}{\log\log m}\right)$, whereas the total number of columns is only slightly larger, i.e., $O\left(m\log^{1+\epsilon} m\right)$. This yields a data structure handling any sequence of row activations and column minima extractions of an offset staircase Monge matrices in $O\left(m\frac{\log^2 m}{\log^2\log m}\right)$ total time.

5.3 Online Column Minima of a Rectangular Offset Monge Matrix

Let $\mathcal{M}_0$ be a rectangular $k \times l$ Monge matrix. Let $R = \{r_1, \ldots, r_k\}$ and $C = \{c_1, \ldots, c_l\}$ be the sets of rows and columns of $\mathcal{M}_0$, respectively. Set $m = \max(k, l)$.

Let $d : R \to \mathbb{R}$ be an offset function and set $\mathcal{M} = \text{off}(\mathcal{M}_0, d)$. By Fact 5.1.3, $\mathcal{M}$ is also a Monge matrix. Our goal is to design a data structure capable of reporting the column minima of $\mathcal{M}$ in increasing order of their values. However, the function d is not entirely revealed beforehand, as opposed to the matrix $\mathcal{M}_0$. There is an initially empty, growing set $\overline{R} \subseteq R$ containing the rows for which $d(r)$ is known. Alternatively, the set $\overline{R}$ can be seen as "active" rows of $\mathcal{M}$ that can be accessed by the data structure. There is also a set $\overline{C} \subseteq C$ containing the remaining columns for which we have not reported the minima yet. Initially, $\overline{C} = C$, and $\overline{C}$ shrinks over time. We also provide a mechanism to guarantee that the rows that have not been revealed do not influence the smallest of the column minima of $\mathcal{M}(R, \overline{C})$.

The exact set of operations we support is the following:

- Activate-Row(r), where $r \in R \setminus \overline{R}$ – add r to the set $\overline{R}$.
- Lower-Bound() – compute the number $\min\{\mathcal{M}(\overline{R}, \overline{C})\}$.
- Ensure-Bound-And-Get() – inform the data structure that we indeed have $\min\{\mathcal{M}(R\setminus\overline{R}, C)\} \geq \min\{\mathcal{M}(\overline{R}, \overline{C})\} =$ Lower-Bound(), that is, the smallest element of $\mathcal{M}(R, \overline{C})$ does not depend on the values of $\mathcal{M}$ located in rows $R\setminus\overline{R}$. It is the responsibility of the user to guarantee that this condition is in fact satisfied.

  Such claim implies that for some column $c \in \overline{C}$ we have $\min\{\mathcal{M}(R, c)\} = \min\{\mathcal{M}(\overline{R}, \overline{C})\}$, which in turn means that we are able to find the minimum element in column c. The function returns any such c and removes it from the set $\overline{C}$.
- Current-Min-Row(c), where $c \in C$ – compute r, where $r \in \overline{R}$ is a row such that $\min\{\mathcal{M}(\overline{R}, c)\} = \mathcal{M}_{r,c}$. If $\overline{R} = \emptyset$, return **nil**. Note that c is not necessarily in $\overline{C}$.

  Additionally, we require Current-Min-Row to have the following property: once the column c is moved out of $\overline{C}$, Current-Min-Row(c) always returns the same row. Moreover, between any two consecutive calls that change either $\overline{R}$ or $\overline{C}$ (Activate-Row or

ENSURE-BOUND-AND-GET), for $c_1, c_2 \in C$ such that $c_1 < c_2$ we have

$$\text{CURRENT-MIN-ROW}(c_1) \geq \text{CURRENT-MIN-ROW}(c_2).$$

Note that ACTIVATE-ROW increases the size of $\overline{R}$ and thus cannot be called more than k times. Analogously, ENSURE-BOUND-AND-GET decreases the size of $\overline{C}$ so it cannot be called more than l times. Actually, in order to reveal all the column minima with this data structure, the operation ENSURE-BOUND-AND-GET has to be called *exactly* l times.

5.3.1 The Components

Subrow minimum query data structure. Given $r \in \overline{R}$ and a, b, $1 \leq a \leq b \leq l$, a subrow minimum query $S(r, a, b)$ computes a column $c \in \{c_a, \ldots, c_b\}$ such that $\mathcal{M}_{r,c} = \min\{\mathcal{M}(r, \{c_a, \ldots, c_b\})\}$. We use the following theorem of Gawrychowski et al. [34].

Theorem 5.3.1 ([34])**.** *Given a $k \times l$ rectangular Monge matrix $\mathcal{M}$, a data structure supporting subrow minimum queries in $O(\log\log(k+l))$ time can be constructed in $O(l \log k)$ time.*

Recall that $\mathcal{M} = \text{off}(\mathcal{M}_0, d)$. Adding the offset $d(r)$ to all the elements in row r of $\mathcal{M}_0$ does not change the relative order of elements in row r. Hence, the answer to a subrow minimum query $S(r, a, b)$ in $\mathcal{M}$ is the same as the answer to $S(r, a, b)$ in $\mathcal{M}_0$.

We build a data structure of Theorem 5.3.1 for $\mathcal{M}_0$ and assume that any subrow minimum query in $\mathcal{M}$ can be answered in $O(\log\log m)$ time.

Column groups. The set C is internally partitioned into disjoint, contiguous *column groups* $\mathcal{C}_1, \ldots, \mathcal{C}_q$ (where $\mathcal{C}_1$ is the leftmost and $\mathcal{C}_q$ is the rightmost) so that $\bigcup_i \mathcal{C}_i = C$.

As the groups constitute contiguous segments of columns, we can represent the partition with a subset $F \subseteq C$ containing the first columns of individual groups. Each group is identified with its leftmost column. We use a dynamic predecessor data structure [93] (see also Section 2.4) for maintaining the set F. The first column of the group containing column c can be thus found by calling $F.\text{PRED}(c)$ in $O(\log\log m)$ time. Such representation also allows to split groups and merge neighboring groups in $O(\log\log m)$ time.

Potential row sets. For each $\mathcal{C}_i$ we store a set $P(\mathcal{C}_i) \subseteq \overline{R}$ called a *potential row set*. Between consecutive operations, the potential row sets satisfy the following invariants:

P.1 For any $c \in \mathcal{C}_i$ there exists a row $r \in P(\mathcal{C}_i)$ such that $\min\{\mathcal{M}(\overline{R}, c)\} = \mathcal{M}_{r,c}$.

P.2 The size of any set $P(\mathcal{C}_i)$ is less than 2α, where $\alpha = \sqrt{\log m}$.

P.3 For any $i < j$ and any $r \in P(\mathcal{C}_i)$, $r' \in P(\mathcal{C}_j)$, we have $r \geq r'$.

The sets $P(\mathcal{C}_i)$ are stored as balanced binary search trees sorted bottom to top. As by Fact 5.1.1 $\mathcal{M}(\overline{R}, C)$ is a Monge matrix, from Fact 5.1.2 it follows that these invariants can be indeed satisfied.

Lemma 5.3.2. *For any choice of column groups we have $\sum_i |P(\mathcal{C}_i)| = O(k+l)$.*

Proof. By invariant P.3 we also have $|P(\mathcal{C}_i) \cap P(\mathcal{C}_{i+1})| \leq 1$ and thus the sum of sizes of sets $P(\mathcal{C}_i)$ is $O(k+l)$. □

We also use the following auxiliary data structures. The union of sets $P(\mathcal{C}_i)$ is stored in a dynamic predecessor/successor data structure U. We also have an auxiliary array `last` mapping each row $r \in \overline{R}$ to the rightmost column group $\mathcal{C}_i$ such that $r \in P(\mathcal{C}_i)$ (if such group exists).

Lemma 5.3.3. *An insertion or deletion of some r to $P(\mathcal{C}_i)$ (along with the update of the auxiliary structures) can be performed in $O(\log \alpha + \log\log m) = O(\log\log m)$ time.*

Proof. The cost of updating the binary search tree is $O(\log |P(\mathcal{C}_i)|) = O(\log \alpha)$, whereas updating the predecessor structure U takes $O(\log\log m)$ time. Updating the array `last` upon insertion is trivial. When a row r is deleted and $\texttt{last}[r] \neq \mathcal{C}_i$, $\texttt{last}[r]$ does not have to be updated. Otherwise, we check if $r \in P(\mathcal{C}_{i-1})$ and set $\texttt{last}[r]$ to either $\mathcal{C}_{i-1}$ or **nil**. □

Special handling of columns with known minima. We require that for each column c being moved out of $\overline{C}$, a row y_c such that $\min\{\mathcal{M}(R, c)\} = \mathcal{M}_{y_c,c}$ is computed. In order to ensure that Current-Min-Row has the described consistent behavior, we guarantee that starting at the moment of deletion of c from $\overline{C}$ there exists a group $\mathcal{C}$ consisting of a single element c such that $P(\mathcal{C}) = \{y_c\}$. We call such groups *done*.

Main priority queue. A priority queue H contains an element c for each $c \in \overline{C}$. The queue H satisfies the following invariants between any two operations.

H.1 For each $c \in \overline{C}$, the key of c in H is greater than or equal to $\min\{\mathcal{M}(\overline{R}, c)\}$.

H.2 For each group $\mathcal{C}_j$ that is not done, there exists such column $c_j \in \mathcal{C}_j$ that the key of c_j in H is equal to $\min\{\mathcal{M}(\overline{R}, c_j)\} = \min\left\{\mathcal{M}(\overline{R}, \mathcal{C}_j)\right\}$.

We maintain invariant H.1 implicitly, each time setting the key of a column c to either ∞ or some value $\mathcal{M}_{r,c}$, where $r \in \overline{R}$. This is justified by the fact that the value $\min\{\mathcal{M}(\overline{R}, c)\}$ does not increase over time.

Note that by invariants H.1 and H.2, the key at the top of H is in fact equal to $\min\{\mathcal{M}(\overline{R}, \overline{C})\}$. Hence, Lower-Bound can be implemented trivially in $O(1)$ time.

Lemma 5.3.4. *We can ensure that the invariant H.2 is satisfied for a single group $\mathcal{C}_j$ in $O(\alpha \log\log m)$ time.*

Proof. We perform $O(|P(\mathcal{C}_j)|) = O(\alpha)$ subrow minimum queries on $\mathcal{M}$ to compute for each $r \in P(\mathcal{C}_j)$ some column $c \in \mathcal{C}_j$ such that $\mathcal{M}_{r,c} = \min\{\mathcal{M}(r, \mathcal{C}_j)\}$. As each subrow minimum query takes $O(\log\log m)$ time, this takes $O(\alpha \log\log m)$ in total. For each computed c, we decrease the key of c in H to $\mathcal{M}_{r,c}$ in $O(1)$ time. Note that by invariant P.1, some $\mathcal{M}_{r,c}$ is in fact equal to $\min\{\mathcal{M}(\overline{R}, \mathcal{C}_j)\}$. □

5.3.2 Implementing the Operations

Initialization. First, we build the data structure of Theorem 5.3.1 in $O(l \log m)$ time. Then, an element c with key ∞ is inserted into H for each $c \in C$. When the first row r is activated, we create a single group $\mathcal{C} = C$ with $P(\mathcal{C}) = \{r\}$. Using Lemma 5.3.4 we ensure that invariant H.2 is satisfied.

Current-Min-Row. The data structure F is used to identify the group $\mathcal{C}$ containing the column c. If $c \in C \setminus \overline{C}$, then the group c is done and we return the only element of $P(\mathcal{C})$. Otherwise, we spend $O(|P(\mathcal{C})|) = O(\alpha)$ time to find the topmost row of $P(\mathcal{C})$ that contains a minimum of c. By Fact 5.1.2 and invariant P.3, returning the topmost row of $P(\mathcal{C})$ guarantees that for $c_1 \leq c_2$, Current-Min-Row$(c_1) \geq$ Current-Min-Row(c_2). The total running time is thus $O(\alpha + \log\log m) = O(\alpha)$.

Ensure-Bound-And-Get. Let c_i be the top element of H and let r^* be the row returned by CURRENT-MIN-ROW(c_i). Invariant H.2 guarantees that we have

$$\min\{\mathcal{M}(\overline{R}, \overline{C})\} = \min\{\mathcal{M}(\overline{R}, c_i)\} = \mathcal{M}_{r^*,c_i} = H.\text{MIN-KEY}(),$$

By the precondition of ENSURE-BOUND-AND-GET, we conclude that in fact

$$\mathcal{M}_{r^*,c_i} = \min\{\mathcal{M}(R, \overline{C})\}.$$

By invariant H.2, H.EXTRACT-MIN() returns the column c_i. With a single query to F, we find the current group of c_i, $\mathcal{C} = \{c_a, \ldots, c_i, \ldots, c_b\}$. First, we need to create a single-column group $\mathcal{C}^* = \{c_i\}$, mark it done, and set $P(\mathcal{C}^*) = \{r^*\}$. We thus split $\mathcal{C}$ into at most three groups $\mathcal{C}^- = \{c_a, \ldots, c_{i-1}\}$, $\mathcal{C}^*$, and $\mathcal{C}^+ = \{c_{i+1}, \ldots, c_b\}$. By Fact 5.1.2, we can safely set $P(\mathcal{C}^-) = \{r \in P(\mathcal{C}) : r \geq r^*\}$ and $P(\mathcal{C}^+) = \{r \in P(\mathcal{C}) : r \leq r^*\}$. The split of $\mathcal{C}$ requires $O(1)$ operations on F, whereas by Lemma 5.3.3, replacing the set $P(\mathcal{C})$ with the sets $P(\mathcal{C}^-), P(\mathcal{C}^*), P(\mathcal{C}^+)$ takes $O(\alpha(\log\log m + \log\alpha))$ time. The last step is to fix invariant H.2 for the newly created groups. This takes $O(\alpha \log\log m)$, by Lemma 5.3.4. Thus, taking into account the $O(\log m)$ cost of performing H.EXTRACT-MIN, ENSURE-BOUND-AND-GET takes $O(\log m + \alpha(\log\alpha + \log\log m)) = O(\log m)$ time.

Before we describe how ACTIVATE-ROW is implemented, we need the following lemma.

Lemma 5.3.5. *Let $\mathcal{M}$ be a $u \times v$ rectangular Monge matrix with rows $\{r_1, \ldots, r_u\}$ and columns $C = \{c_1, \ldots, c_v\}$. For any $i \in [1, u]$, in $O\left(u\frac{\log v}{\log u}\right)$ time we can find such $c_s \in C$ that:*

1. *some minima of columns $c_1, \ldots, c_s$ lie in rows $r_{i+1}, \ldots, r_u$,*
2. *some minima of columns $c_{s+1}, \ldots, c_v$ lie in rows $r_1, \ldots, r_i$.*

Proof. Let $R = \{r_1, \ldots, r_u\}$. Aggarwal et al. [1] proved the following theorem. The algorithm they found was nicknamed *the SMAWK algorithm*.

Theorem 5.3.6 ([1])**.** *One can compute the bottommost column minima of a rectangular $k \times l$ Monge matrix in $O(k + l)$ time.*

If $u \geq v$, we can find the column minima for each column of matrix $\mathcal{M}$ using the SMAWK algorithm in $O(u)$ time. Picking the right c_s is straightforward in this case.

Assume $u < v$. We first pick a set $C' = \{c'_1, \ldots, c'_u\}$ of u evenly spread columns of C, including the leftmost and the rightmost column. By Fact 5.1.1, $\mathcal{M}(R, C')$ is also a Monge matrix. The SMAWK algorithm is then used to obtain the bottommost rows $r'_1, \ldots, r'_u$ containing the column minima of $c'_1, \ldots, c'_u$ in $O(u)$ time. By Fact 5.1.2, we have $r'_1 \geq \ldots \geq r'_u$. We then find some j such that $r'_j \geq r_i \geq r'_{j+1}$. The sought column c_s can now be found by proceeding recursively on the matrix

$$\mathcal{M}' = \mathcal{M}(R, \{c'_j, \ldots, c'_{j+1}\}).$$

The matrix $\mathcal{M}'$ has still u rows, but it has only $O(v/u)$ columns.

At each recursive step we divide the size of the column set by $\Omega(u)$ so there are at most $\log_u v = \frac{\log v}{\log u}$ steps. Each step takes $O(u)$ time and hence we obtain the desired bound. □

Activate-Row. Assume we activate row r. At that point $r \notin P(\mathcal{C}_i)$ for any group $\mathcal{C}_i$. Our goal is to reorganize the column groups and their potential row sets so that the invariants P.1, P.2, P.3 and H.2 are again satisfied.

In the following, let us assume for convenience that $\mathcal{C}_0$ ($\mathcal{C}_{q+1}$) is a "sentinel" group that is done and we have $P(\mathcal{C}_0) = \{r_{+\infty}\}$ ($P(\mathcal{C}_{q+1}) = \{r_{-\infty}\}$, respectively). Also, for any r we have $r_{-\infty} < r < r_{+\infty}$.

Let us first observe that if there exists a group $\mathcal{C}^* = \{c^*\}$ that is done and $P(\mathcal{C}^*) = \{r^*\}$, $r^* > r$, then there is no point in updating the potential row sets for groups to the left of $\mathcal{C}^*$. Indeed, by Fact 5.1.5, for any column c' to the left of c^*, if c' has a minimum in row r, then it also has a minimum in row r^*. Similarly, if $r^* < r$, then there is no need to update potential row sets for groups to the right of $\mathcal{C}^*$. Therefore, the only groups whose potential row sets may become invalid after the activation of r lie between two consecutive groups that are done.

Denote by $\mathcal{C}_x, \ldots, \mathcal{C}_y$ the unique maximal contiguous set of groups such that:

- none of $\mathcal{C}_x, \ldots, \mathcal{C}_y$ is done,
- $\mathcal{C}_{x-1}$ is done and $r' > r$ where r' is the only element of $P(\mathcal{C}_{x-1})$,
- $\mathcal{C}_{y+1}$ is done and $r' < r$ where r' is the only element of $P(\mathcal{C}_{y+1})$.

If this set is empty, then clearly for all columns c, $\min\{\mathcal{M}(\overline{R} \cup \{r\}, c)\} = \min\{\mathcal{M}(\overline{R}, c)\}$ and hence the potential row set do not require any adjustments.

Consider some i, $x \leq i \leq y$. $\mathcal{C}_i$ can fall into exactly one of three categories.

C.1 For each $c \in \mathcal{C}_i$ we have $\mathcal{M}_{r,c} \leq \min\{\mathcal{M}(P(\mathcal{C}_i), c)\}$,

C.2 For some two columns $c_1, c_2 \in \mathcal{C}_i$ we have $\mathcal{M}_{r,c_1} < \min\{\mathcal{M}(P(\mathcal{C}_i), c_1)\}$ and $\mathcal{M}_{r,c_2} > \min\{\mathcal{M}(P(\mathcal{C}_i), c_2)\}$.

C.3 For each $c \in \mathcal{C}_i$, we have $\mathcal{M}_{r,c} \geq \min\{\mathcal{M}(P(\mathcal{C}_i), c)\}$ and for some $c' \in \mathcal{C}_i$ we have $\mathcal{M}_{r,c'} > \min\{\mathcal{M}(P(\mathcal{C}_i), c')\}$

Fact 5.1.5 guarantees that row r contains column minima for a (possibly empty) interval of columns of $\mathcal{M}(\overline{R} \cup \{r\}, C)$. As the groups do not overlap, this implies that the groups in category **C.1** form a (possibly empty) interval of groups $\mathcal{C}_a, \ldots, \mathcal{C}_b$, where $x \leq a \leq b \leq y$. Moreover, there can be at most two category **C.2** groups, if they exist, namely $\mathcal{C}_{a-1}$ and/or $\mathcal{C}_{b+1}$ (in this case $a - 1 \geq x$ and/or $b + 1 \leq y$, respectively).

Observe that we can decide if $\mathcal{C}_i$ falls into category **C.1** in $O(|P(\mathcal{C}_i)|) = O(\alpha)$ time by checking, equivalently, if r contains column minima of both the leftmost and the rightmost columns of $\mathcal{C}_i$. This follows from Fact 5.1.5.

Moreover, if r is below all the rows of $P(\mathcal{C}_i)$ or above all the rows of $P(\mathcal{C}_i)$, then, by looking only at the leftmost and rightmost columns of $\mathcal{C}_i$, we can precisely detect the category of $\mathcal{C}_i$ in $O(\alpha)$ time.

Note that, as invariant P.3 holds before the activation of r, there can be at most one group $\mathcal{C}_-^+$ of $\mathcal{C}_x, \ldots, \mathcal{C}_y$, such that $P(\mathcal{C}_-^+)$ contains rows both above and below r.

Equipped with these observations, we proceed as follows. We first find the rightmost group $\mathcal{C}_i$ such that for some $r' \in P(\mathcal{C}_i)$ we have $r' > r$. This can be done in $O(\log \log m)$ time by setting $\mathcal{C}_i = \texttt{last}(U.\textsc{Succ}(r))$. By Fact 5.1.5, if there is any group $\mathcal{C}'$ in categories **C.1** or **C.2**, then one of the groups $\mathcal{C}_i$, $\mathcal{C}_{i+1}$ also falls into **C.1** or **C.2**. We may thus find all groups $\mathcal{C}_a, \ldots, \mathcal{C}_b$ in category **C.1** by moving both to the left and to the right of $\mathcal{C}_i$. The groups $\mathcal{C}_a, \ldots, \mathcal{C}_b$ are replaced with a single group $\mathcal{C}^*$ spanning all their columns and $P(\mathcal{C}^*)$ is set to $\{r\}$. If the group $\mathcal{C}_{a-1}$ ($\mathcal{C}_{b+1}$ respectively) exists, we insert r into $P(\mathcal{C}_{a-1})$ ($P(\mathcal{C}_{b+1})$) only if this group is either in fact $\mathcal{C}_-^+$ or is in category **C.2**. After such insertions, both invariants P.2 and P.3 may become violated.

Invariant P.3 can only be violated if the group $\mathcal{C}_-^+$ existed, was not in category **C.1**, and also there exists some other group with r in its potential row set. Since it is impossible that r was inserted into potential row sets of groups both to the left and to the right of $\mathcal{C}_-^+$, suppose without loss of generality that some $\mathcal{C}'$ is to the right of $\mathcal{C}_-^+$ and $r \in P(\mathcal{C}')$. In $O(\alpha)$ time we

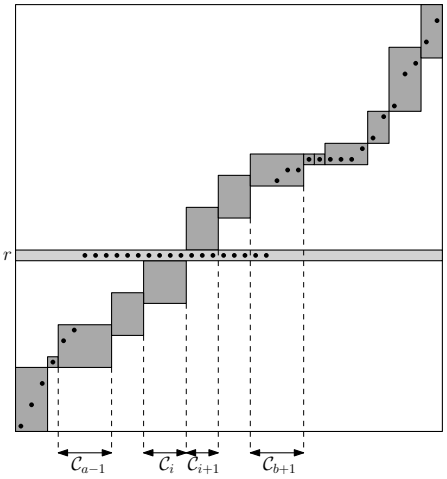

Figure 5.2: Updating the column groups and the corresponding potential row sets after activating row r. The rectangles conceptually show the potential row sets. The rows of $\overline{R}$ that are not contained in any potential row set are omitted in the picture. The dots represent the column minima. Note that it might happen that $P(\mathcal{C}_i)$ contains rows both above and below r.

can check if r contains the column minimum of the rightmost column of $\mathcal{C}_-^+$ in $\mathcal{M}(\overline{R} \cup \{r\}, \mathcal{C}_-^+)$. If so, by Facts 5.1.2 and 5.1.5, we can delete from $P(\mathcal{C}_-^+)$ all the rows above r (recall that r contains a column minimum for the leftmost column of $\mathcal{C}'$). Otherwise, by Fact 5.1.5 and since r contains some column minimum of $\mathcal{M}(R \cup \{r\}, \mathcal{C}')$, we can safely delete r from $P(\mathcal{C}_-^+)$. Hence, we fix invariant P.3 in $O(\alpha(\log\log m + \log\alpha))$ time.

Invariant P.2 is violated if $|P(\mathcal{C}_{a-1})| = 2\alpha$ or $|P(\mathcal{C}_{b+1})| = 2\alpha$. In that case algorithm of Lemma 5.3.5 is used to split the relevant group $\mathcal{C}_z$, for $z \in \{a-1, b+1\}$, into groups $\mathcal{C}'_z, \mathcal{C}''_z$ such that $|P(\mathcal{C}'_z)| = |P(\mathcal{C}''_z)| = \alpha$.

We spend $O(\alpha(\log\log m + \log\alpha)) = O(\alpha \log\log m)$ time on identifying, accessing and updating each group that falls into categories **C.2** or **C.3**. There are $O(1)$ such groups, as discussed above. Also, by Lemma 5.3.4, it takes $O(\alpha \log\log m)$ time to fix invariant H.2 for (possibly split) groups $\mathcal{C}_{a-1}$, $\mathcal{C}_{b+1}$ and $\mathcal{C}^*$.

In order to bound the running time of the remaining steps, i.e., handling the groups of category **C.1** and splitting the groups that break invariant P.2, we introduce two types of credits for each element inserted into sets $P(\mathcal{C}_i)$:

- an $O(\log\log m)$ *identification credit*,
- an $O\left(\frac{\log m}{\log \alpha}\right)$ *splitting credit*.

The identification credit is used to pay for successfully verifying that some group $\mathcal{C}_i$ falls into category **C.1** and deleting all the elements of $P(\mathcal{C}_i)$. Indeed, as discussed above, we spend $O(|P(\mathcal{C}_i)| \log\alpha) = O(|P(\mathcal{C}_i)| \log\log m)$ time on this. As $P(\mathcal{C}_i)$ is not empty, we can charge the cost of merging $\mathcal{C}_i$ with some other group to some arbitrary element of $P(\mathcal{C}_i)$. Recall that merging and splitting groups takes $O(\log\log m)$ time.

Finally, consider performing a split of $P(\mathcal{C}_i)$ of size 2α. As the sets $P(\mathcal{C}_i)$ only grow by inserting single elements, there exist at least α elements of $P(\mathcal{C}_i)$ that never took part in any split. We use the total $O\left(\alpha\frac{\log m}{\log\alpha}\right)$ total credit of those elements to pay for the split.

To sum up, the time needed to perform k operations Activate-Row is $O\left(k\alpha(\log\log m) + I(\log\log m + \frac{\log m}{\log\alpha})\right)$, where I is the total number of insertions to the sets

$P(\mathcal{C}_i)$. Since Ensure-Bound-And-Get incurs $O(l)$ insertions in total, $I = O(k + l)$. As we have previously set $\alpha = \sqrt{\log m}$, we obtain the following lemma.

Lemma 5.3.7. *Let $\mathcal{M}$ be a $k \times l$ offset Monge matrix. There exists a data structure initialized in $O(k + l \log m)$ time, supporting* Lower-Bound *in $O(1)$ time and both* Current-Min-Row *and* Ensure-Bound-And-Get *in $O(\log m)$ time. Additionally, any sequence of operations* Activate-Row *can be performed in $O\left((k+l)\frac{\log m}{\log\log m}\right)$ total time, where $m = \max(k, l)$.*

5.4 Online Column Minima of a Block Monge Matrix

Let $\mathcal{M} = \text{off}(\mathcal{M}_0, d)$, R, C, l, k, m be defined as in Section 5.3. In this section we consider the problem of reporting the column minima of a rectangular offset Monge matrix but in a slightly different setting. Again, we are given a fixed rectangular Monge matrix $\mathcal{M}_0$ and we also have an initially empty, growing set of rows $\overline{R} \subseteq R$ for which the offsets $d(\cdot)$ are known. Let $\Delta > 0$ be an integral parameter not larger than l. We partition C into a set $\mathcal{B} = \{B_1, \ldots, B_b\}$ of at most $\lceil l/\Delta \rceil$ blocks, each of size at most Δ. The columns in each B_i constitute a contiguous fragment of $c_1, \ldots, c_l$, and each block B_i is to the left of B_{i+1}. We also maintain a shrinking subset $\overline{\mathcal{B}} \subseteq \mathcal{B}$ containing the blocks B_i such that the minima $\min\{\mathcal{M}(R, B_i)\}$ are not yet known. More formally, for each $B_i \in \mathcal{B} \setminus \overline{\mathcal{B}}$ we have $\min\{\mathcal{M}(R, B_i)\} = \min\{\mathcal{M}(\overline{R}, B_i)\}$. Initially $\overline{\mathcal{B}} = \mathcal{B}$.

For each $c \in C$ not contained in the blocks of $\overline{\mathcal{B}}$, the data structure explicitly maintains the *current minimum*, i.e., the value $\min\{\mathcal{M}(\overline{R}, c)\}$. Moreover, when a new row is activated, we provide the user with columns of $\bigcup(\mathcal{B} \setminus \overline{\mathcal{B}})$ for which the current minima have changed.

For blocks $\overline{\mathcal{B}}$, the data structure only maintains the value $\min\{\mathcal{M}(\overline{R}, \bigcup\overline{\mathcal{B}})\}$. Once the user can guarantee that $\min\{\mathcal{M}(R, \bigcup\overline{\mathcal{B}})\}$ does not depend on the "hidden offsets" of rows $R \setminus \overline{R}$, the data structure moves a block $B_i \in \overline{\mathcal{B}}$ such that $\min\{\mathcal{M}(R, \bigcup\overline{\mathcal{B}})\} = \min\{\mathcal{M}(\overline{R}, B_i)\}$ out of $\overline{\mathcal{B}}$ and makes it possible to access the current minima of columns of B_i.

More formally, we support the following set of operations:

- Activate-Row(r), where $r \in R \setminus \overline{R}$ – add r to the set $\overline{R}$.
- Block-Lower-Bound() – return $\min\{\mathcal{M}(\overline{R}, \bigcup\overline{\mathcal{B}})\}$.
- Block-Ensure-Bound() – tell the data structure that indeed

  $$\min\{\mathcal{M}(R \setminus \overline{R}, C)\} \geq \text{Block-Lower-Bound}() = \min\{\mathcal{M}(\overline{R}, B_i)\},$$

  for some $B_i \in \overline{\mathcal{B}}$, i.e., that the smallest element of $\mathcal{M}(R, \bigcup\overline{\mathcal{B}})$ does not depend on the entries of $\mathcal{M}$ located in rows $R \setminus \overline{R}$. Again, it is the responsibility of the user to guarantee that this condition is in fact satisfied.

  As the minimum of $\mathcal{M}(R, B_i)$ can now be computed, B_i is removed from $\overline{\mathcal{B}}$.
- Current-Min(c), where $c \in C$ – for $c \in \bigcup(\mathcal{B} \setminus \overline{\mathcal{B}})$, return the explicitly maintained $\min\{\mathcal{M}(\overline{R}, \{c\})\}$. For $c \in \bigcup\overline{\mathcal{B}}$, set Current-Min($c$) $= \infty$.

Additionally, the data structure provides access to the queue Updates containing the columns $c \in \bigcup(\mathcal{B} \setminus \overline{\mathcal{B}})$ such that the most recent call to either Activate-Row or Block-Ensure-Bound resulted in a change (or an initialization, if $c \in B_i$ and the last update was Block-Ensure-Bound which moved B_i out of $\overline{\mathcal{B}}$) of the value Current-Min(c).

Note that there can be at most k calls to Activate-Row and no more than $\lceil l/\Delta \rceil$ calls to Block-Ensure-Bound.

5.4.1 The Components

Infrastructure for short subrow minimum queries. In this section we assume that for any $r \in R$ and $1 \leq a, b \leq l$, $b - a + 1 \leq \Delta$, it is possible to compute an answer to a subrow minimum query $S(r, a, b)$ (see Section 5.3) on matrix $\mathcal{M}_0$ (equivalently: $\mathcal{M}$) in constant time. We call such a subrow minimum query *short*.

Block minima matrix. Define a $k \times b$ matrix $\mathcal{M}'$ with rows R and columns $\mathcal{B}$, such that for all $r_i \in R$ and $B_j \in \mathcal{B}$,

$$\mathcal{M}'_{r_i,B_j} = \min\{\mathcal{M}(r_i, B_j)\}.$$

Lemma 5.4.1. *$\mathcal{M}'$ is a Monge matrix and its entries can be accessed in $O(1)$ time.*

Proof. As we assume that we can perform short subrow minima queries in $O(1)$ time, and every block spans at most Δ columns, we can access the elements of $\mathcal{M}'$ in constant time. Fact 5.1.4 implies that $\mathcal{M}'$ is also a rectangular Monge matrix. □

Equipped with Lemma 5.4.1, we build the data structure of Section 5.3 for matrix $\mathcal{M}'$. For brevity, we identify the matrix $\mathcal{M}'$ and its associated data structure. We use the dot notation to denote operations acting on specific matrices, e.g., $\mathcal{M}_i$.Lower-Bound.

Exact minima array. For each column $c \in \bigcup(\mathcal{B} \setminus \overline{\mathcal{B}})$, the value

$$\texttt{cmin}(c) = \min\{\mathcal{M}(\overline{R}, c)\}$$

is stored explicitly. The operation Current-Min(c) simply returns $\texttt{cmin}(c)$.

Rows containing the block minima. For each $B_j \in (\mathcal{B} \setminus \overline{\mathcal{B}})$ we store the value

$$y_j = \mathcal{M}'.\textsc{Current-Min-Row}(B_j).$$

Note that the data structure of Section 5.3 guarantees that for $B_i, B_j \in (\mathcal{B} \setminus \overline{\mathcal{B}})$ such that $i < j$, we have $y_i \geq y_j$. The set of defined y_j's grows over time. Additionally, we store this set in a dynamic predecessor/successor data structure Y.

We also use two auxiliary arrays `first` and `last` indexed with the rows of R. $\texttt{first}(r)$ ($\texttt{last}(r)$) contains the leftmost (rightmost respectively) block B_j such that $y_j = r$. Updating these arrays in $O(1)$ time each time $\overline{\mathcal{B}}$ shrinks (i.e., when $\mathcal{B} \setminus \overline{\mathcal{B}}$ grows) is straightforward.

Row candidate sets. Two subsets D_0 and D_1 of $\overline{R}$ are maintained. The set D_j for $j = 0, 1$ contains the rows of $\overline{R}$ that may still prove useful when computing the initial value of $\texttt{cmin}(c)$ for $c \in \bigcup\{B_i : B_i \in \overline{\mathcal{B}} \wedge i \bmod 2 = q\}$. More formally, before and after each operation, for each such c, D_j contains a row r such that $\min\{\mathcal{M}(\overline{R}, c)\} = \mathcal{M}_{r,c}$.

The sets D_0, D_1 are also stored in dynamic predecessor structures.

Remark 5.4.2. *There is a subtle reason why we keep two row candidate sets D_0, D_1, responsible for even and odd blocks respectively, instead of one. Being able to separate two neighboring blocks of each group with a block from the other group will prove useful in the amortized analysis of the cost of the operation* Block-Ensure-Bound.

5.4.2 Implementing the Operations

Block-Ensure-Bound. The preconditions of this operation ensure that it is valid to call $\mathcal{M}'$.Ensure-Bound-And-Get(), which in response returns some B_j. At this point we find the row y_j containing the minimum of $\mathcal{M}(R, B_j)$ using $\mathcal{M}'$.Current-Min-Row(B_j). The data structure Y and the arrays `first` and `last` are updated accordingly.

As the block B_j is moved out of $\overline{\mathcal{B}}$, we need to compute the initial values `cmin`(c) for $c \in B_j$. Let y_j^- be the row returned by $\mathcal{M}'$.Current-Min(B_{j-1}) if $j > 0$ and r_k otherwise. Similarly, set y_j^+ to be the row returned by $\mathcal{M}'$.Current-Min(B_{j+1}) if $j < b$ and r_1 otherwise. Clearly, $y_j^- \geq y_j \geq y_j^+$. First we prove that for each column $c \in B_j$, we have

$$\min\{\mathcal{M}(\overline{R}, c)\} = \min\{\mathcal{M}(\overline{R} \cap \{y_j^+, \ldots, y_j^-\}, c)\},$$

that is, the search for the minimum in column c can be limited to rows y_j^+ through y_j^-. By the definition of $\mathcal{M}'$, for some column $c_j \in B_j$, the minimum $\mathcal{M}(\overline{R}, c_j)$ is located in row y_j. Now assume that $c \in B_j$ is to the left of c_j. By Fact 5.1.2, some minimum of $\mathcal{M}(\overline{R}, c)$ is located in the rows of $\overline{R}$ (weakly) below y_j. If $j > 0$, then for some column $c_j^- \in B_{j-1}$ some minimum of $\mathcal{M}(\overline{R}, c_j^-)$ is located in row y_j^-. By Fact 5.1.2, the minimum of $\mathcal{M}(\overline{R}, c)$ is located in rows (weakly) above y_j^-. Analogously we prove that for $c \in B_j$ to the right of c_j, the minimum is located in rows y_j^+ through y_j.

We first add the rows y_j^-, y_j^+ to $D_{j \bmod 2}$. By the invariant posed on the set $D_{j \bmod 2}$ before the operation Block-Ensure-Bound and $\{y_j^-, y_j^+\} \subseteq D_{j \bmod 2}$, for each column $c \in B_j$ it suffices to only consider the elements $\mathcal{M}_{r,c}$, where $r \in D_{j \bmod 2} \cap \{y_j^+, \ldots, y_j^-\}$, as potential minima in column c. All such rows r can be found with $O(\log\log m)$ overhead per row using predecessor search on $D_{j \bmod 2}$. Now we prove that after performing this step for all columns $c \in B_j$, all such rows r except of y_j^- and y_j^+ can be safely removed from $D_{j \bmod 2}$. Indeed, let c_z be a column in some block $B_z \in \overline{\mathcal{B}}$ such that $z \equiv j \pmod 2$ and $z < j$. In fact, we have $z < j - 1$ and thus c_z is to the left of c_j^-. By Fact 5.1.5, if c_z has a minimum in row r, it also has a minimum in row y_j^-. Hence removing r from $D_{j \bmod 2}$ does not break the invariant posed on $D_{j \bmod 2}$ since $y_j^- \in D_{j \bmod 2}$. The proof of the case $z > j$ is analogous.

Let us now bound the total time spent on updating values `cmin`(c) during the calls Block-Ensure-Bound. For each column $c \in B_j$, all entries $\mathcal{M}_{r,c}$, where $r \in D_{j \bmod 2} \cap \{y_j^+, \ldots, y_j^-\}$, are examined as potential column minima. Alternatively, we can say that for each such row, we try to use it as a candidate for minima of $O(\Delta)$ columns. However, only two of these rows are not deleted from $D_{j \bmod 2}$ afterwards. If we assign a credit of Δ to each row inserted into $D_{j \bmod 2}$, this credits can be used to pay for considering all the rows except of y_j^- and y_j^+. Thus, the total time spent on testing candidates for the minima over all calls to Block-Ensure-Bound can be bounded by $O\left(\frac{l}{\Delta}\Delta + I\Delta\right)$, where I is the number of insertions to either D_0 or D_1. However, as we will see below, since Activate-Row(r) will only insert r to both D_0 and D_1, we have $I = \left(k + \frac{l}{\Delta}\right)$ and thus the total number of candidates tried by Block-Ensure-Bound is $O(k\Delta + l)$. The total cost spent on maintaining and traversing the sets D_0 and D_1 is $O\left((I + \frac{l}{\Delta})\log\log m\right) = O\left((k + \frac{l}{\Delta})\log\log m\right)$.

Activate-Row. Suppose we activate the row $r \in R \setminus \overline{R}$. The first step is to call $\mathcal{M}'$.Activate-Row(r) and add r to the sets D_0 and D_1. Observe that adding any row from $\overline{R}$ to D_0 or D_1 cannot break the invariants posed on these sets.

The introduction of the row r may change the minima of some columns $c \in \bigcup(\mathcal{B} \setminus \overline{\mathcal{B}})$. We now prove that there can be at most $O(\Delta)$ such changes. Recall that for each $B_i \in \mathcal{B} \setminus \overline{\mathcal{B}}$,

for some column $c_i \in B_i$ the minimum of $\mathcal{M}(R, c_i)$ is located in row y_i. Note that $r \neq y_i$ as r has just been activated. Let u be such that $y_u > r$. Then, for each block $B_j \in \mathcal{B} \setminus \overline{\mathcal{B}}$, where $j < u$, Fact 5.1.2 implies that all the columns of B_j have some of their minima in rows below y_u (or exactly at y_u) and thus the introduction of row r does not affect their minima. Analogously, if $y_v < r$, then the introduction of row r does not affect columns in blocks to the right of B_v. Hence, r can only affect the exact minima in at most two blocks: B_u, B_v where $u = \texttt{last}(Y.\textsc{Succ}(r))$ and $v = \texttt{first}(Y.\textsc{Pred}(r))$. The blocks can be found in $O(\log\log m)$ time, whereas updating the values $\texttt{cmin}(c)$ (along with pushing them to the queue Updates) takes $O(\Delta)$ time.

Let us bound the total running time of any sequence of operations Activate-Row and Block-Ensure-Bound. By Lemma 5.3.7, the time spent on executing operations on the data structure $\mathcal{M}'$ is $O\left(k\frac{\log m}{\log\log m} + \frac{l}{\Delta}\log m\right)$, whereas the time spent on maintaining the predecessor structures and updating the column minima is $O\left(k\Delta + k\log\log m + l + \frac{l}{\Delta}\log\log m\right)$. The following lemma follows.

Lemma 5.4.3. *Let $\mathcal{M} = \mathit{off}(\mathcal{M}_0, d)$ be a $k \times l$ rectangular offset Monge matrix. Let Δ be the block size. Assume we can perform subrow minima queries spanning at most Δ columns of $\mathcal{M}_0$ in $O(1)$ time. There exists a data structure initialized in $O(k+l+\frac{l}{\Delta}\log m)$ time and supporting both* Block-Lower-Bound *and* Current-Min *in $O(1)$ time. Any sequence of* Activate-Row *and* Block-Ensure-Bound *operations can be performed in $O\left(k\left(\frac{\log m}{\log\log m} + \Delta\right) + l + \frac{l}{\Delta}\log m\right)$ time, where $m = \max(k, l)$.*

5.5 Online Column Minima of a Staircase Offset Monge Matrix

In this section we show a data structure supporting a similar set of operations as in Section 5.3, but in the case when the matrices $\mathcal{M}_0$ and $\mathcal{M} = \mathrm{off}(\mathcal{M}_0, d)$ are staircase Monge matrices with m rows $R = \{r_1, \ldots, r_m\}$ and m columns $C = \{c_1, \ldots, c_m\}$. We still aim at reporting the column minima of $\mathcal{M}$ while the set $\overline{R}$ of revealed rows is extended and new bounds on the value $\min\{\mathcal{M}(R \setminus \overline{R}, C)\}$ are given.

In comparison to the data structure of Section 5.3, we loosen a bit the conditions posed on the operations Lower-Bound and Ensure-Bound-And-Get. Now, Lower-Bound might return a value smaller than $\min\{\mathcal{M}(\overline{R}, \overline{C})\}$ and a single call to Ensure-Bound-And-Get might not report any new column minimum at all. However, Ensure-Bound-And-Get can still only be called if $\min\{\mathcal{M}(R \setminus \overline{R}, C)\} \geq$ Lower-Bound() and the data structure we develop in this section guarantees that a bounded number of calls to Ensure-Bound-And-Get suffices to report all the column minima of $\mathcal{M}$. The exact set of supported operations is as follows.

- Activate-Row(r), where $r \in R \setminus \overline{R}$ – add r to the set $\overline{R}$.
- Lower-Bound() – return a number v such that $\min\{\mathcal{M}(\overline{R}, \overline{C})\} \geq v$. If $\overline{R} = \emptyset$ or $\overline{C} = \emptyset$, return ∞.
- Ensure-Bound-And-Get() – tell the data structure that the inequality

  $$\min\{\mathcal{M}(R \setminus \overline{R}, C)\} \geq \textsc{Lower-Bound}()$$

  holds. As for previous data structures, it is the responsibility of the user to guarantee that this condition is in fact satisfied.

  With this knowledge, the data structure may report some column $c \in \overline{C}$ such that $\min\{\mathcal{M}(R, c)\}$ is known. However, it's also valid to not report any new column minimum (in such case **nil** is returned) and only change the known value of Lower-Bound().

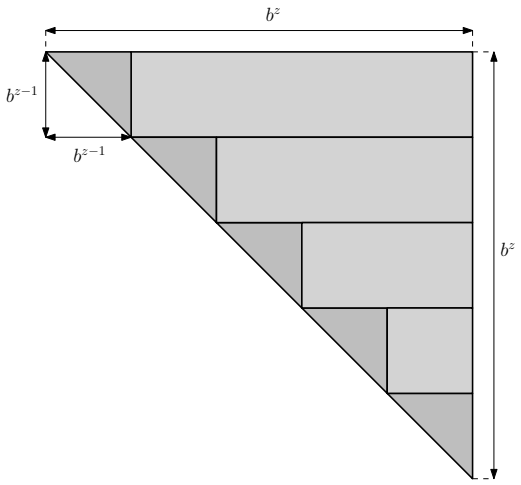

Figure 5.3: Schematic depiction of the biased partition used in Lemma 5.5.1.

- CURRENT-MIN(c), where $c \in C$ – if $c \in C \setminus \overline{C}$, return the known minimum in column c. Otherwise, return ∞.

5.5.1 Partitioning a Staircase Matrix into Rectangular Matrices

Before we describe the data structure, we prove the following lemma on partitioning staircase matrices into rectangular matrices.

Lemma 5.5.1. *For any $\epsilon \in (0,1)$, a staircase matrix $\mathcal{M}$ with m rows and m columns can be partitioned in $O(m \log^\epsilon m)$ time into $O(m \log^\epsilon m)$ non-overlapping rectangular matrices so that each row appears in $O\left(\frac{\log m}{\log\log m}\right)$ matrices of the partition, whereas each column appears in $O\left(\frac{\log^{1+\epsilon} m}{\log\log m}\right)$ matrices of the partition.*

Proof. Let $\epsilon \in (0,1)$ and set $b = \lfloor \log^\epsilon m \rfloor$. For $m > 1$, we have $b \geq 1$.

We first describe the partition for matrices $\mathcal{M}'$ with $m' = b^z$ rows $\{r'_1, \ldots, r'_{m'}\}$ and m' columns $\{c'_1, \ldots, c'_{m'}\}$, where $z \geq 0$. Our partition will be recursive. If $z = 0$, then $\mathcal{M}'$ is a 1×1 matrix and our partition consists of a single element $\mathcal{M}'$.

Assume $z > 0$. We partition $\mathcal{M}'$ into b staircase matrices $\mathcal{M}'^{\mathrm{s}}_1, \ldots, \mathcal{M}'^{\mathrm{s}}_b$ and $b-1$ rectangular matrices $\mathcal{M}'^{\mathrm{r}}_1, \ldots, \mathcal{M}'^{\mathrm{r}}_{b-1}$. For $i = 1, \ldots, b$, we set the i-th staircase matrix to be

$$\mathcal{M}'^{\mathrm{s}}_i = \mathcal{M}'(\{r'_{(i-1)b^{z-1}+1}, \ldots, r'_{ib^{z-1}}\},\ \{c'_{(i-1)b^{z-1}+1}, \ldots, c'_{ib^{z-1}}\}),$$

whereas for $j = 1, \ldots, b-1$, the j-th rectangular matrix is defined as

$$\mathcal{M}'^{\mathrm{r}}_i = \mathcal{M}'(\{r'_{(i-1)b^{z-1}+1}, \ldots, r'_{ib^{z-1}}\},\ \{c'_{ib^{z-1}+1}, \ldots, c'_{m'}\}).$$

See Figure 5.3 for a schematic depiction of such partition.

Each of matrices $\mathcal{M}'^{\mathrm{s}}_i$ is of size $b^{z-1} \times b^{z-1}$ and is then partitioned recursively.

Let us now compute the value $rowcnt(z)$ ($colcnt(z)$) defined as the maximum number of matrices in partition that some given row (column respectively) of a staircase matrix $\mathcal{M}'$ of size $b^z \times b^z$ appears in. Clearly, at the topmost level of recursion each row appears in exactly one staircase matrix $\mathcal{M}'^{\mathrm{s}}_i$ and at most one rectangular matrix $\mathcal{M}'^{\mathrm{r}}_j$. Thus, we have $rowcnt(0) = 1$ and $rowcnt(z+1) \leq rowcnt(z) + 1$, which easily implies $rowcnt(z) \leq z + 1$.

Each column appears in exactly one matrix $\mathcal{M}'^{\mathrm{s}}_i$ and no more than $b-1$ matrices $\mathcal{M}'^{\mathrm{r}}_j$. Hence, we have $colcnt(0) = 1$ and $colcnt(z+1) \leq colcnt(z) + b - 1$. We can thus conclude that $colcnt(z) \leq zb - z + 1$.

Analogously we can compute the value $rectcnt(z)$ denoting the total number of rectangular matrices in such a recursive partition. We have $rectcnt(0) = 1$ and $rectcnt(z+1) \leq b \cdot rectcnt(z) + b - 1$. An easy induction argument shows that $rectcnt(z) \leq 2b^z - 1$.

The partition for an arbitrary matrix $\mathcal{M}$ of m is obtained as follows. We find the smallest y such that $b^y \geq m$. We next find the recursive partition of matrix $\mathcal{M}^*$ which is defined as $\mathcal{M}$ padded so that it has b^y rows and b^y columns. The last step is to remove some number of dummy rightmost columns and bottommost rows from each rectangular matrix of the partition.

Now, each row of $\mathcal{M}$ appears in at most

$$y + 1 = O(\log_b m) = O\left(\frac{\log m}{\log b}\right) = O\left(\frac{\log m}{\epsilon \log\log m}\right) = O\left(\frac{\log m}{\log\log m}\right)$$

rectangular matrices of a partition. Each column of $\mathcal{M}$ appears in at most

$$yb - y + 1 \leq yb + 1 = O(b \log_b m) = O\left(\frac{b \log m}{\log b}\right) = O\left(\frac{\log^{1+\epsilon} m}{\epsilon \log\log m}\right) = O\left(\frac{\log^{1+\epsilon} m}{\log\log m}\right)$$

matrices of the partition. The partition consists of at most $2b^y - 1 = O(mb) = O(m \log^\epsilon m)$ rectangles. The time needed to compute the row and column intervals constituting the rows and columns of the individual matrices of the partition is $O(rectcnt(y)) = O(m \log^\epsilon m)$. □

5.5.2 The Components

Short subrow minimum queries infrastructure. Let $\Delta = \lceil \log^{1-\epsilon/2} m \rceil$. The following lemma allows us to use the data structure of Lemma 5.4.3 with block size Δ.

Lemma 5.5.2. *The staircase Monge matrix $\mathcal{M}_0$ can be preprocessed in $O(m\Delta \log m)$ time so that subrow minimum queries on $\mathcal{M}_0$ spanning at most Δ columns take $O(1)$ time.*

Proof. We use the following result of [34].

Lemma 5.5.3 ([34, Lemma 3], [35, Lemma 3])**.** *Let $y \leq m$ and $x = O(\log m)$, where m fits in a single word. Given a $y \times x$ rectangular Monge matrix $\mathcal{M}'$, one can construct in $O(x \log m)$ time an $O(x)$-space data structure supporting subrow minimum queries spanning* all *columns of $\mathcal{M}'$ in $O(1)$ time.*

Let q be the maximum integer such that $2^q < \Delta$. For $j \in [0, q]$ and $i \in [1, m - 2^j + 1]$, let $\mathcal{M}^j_i = \mathcal{M}_0(\{r_1, \ldots, r_i\}, \{c_i, \ldots, c_{i+2^j-1}\})$, i.e., $\mathcal{M}^j_i$ is a rectangular submatrix of $\mathcal{M}_0$ with columns $\{c_i, \ldots, c_{i+2^j-1}\}$ and all rows that have values defined for these columns. By Fact 5.1.1, $\mathcal{M}^j_i$ is a Monge matrix. For each $\mathcal{M}^j_i$, we build a data structure of Lemma 5.5.3. This takes

$$O\left(\sum_{j=0}^{q} \sum_{i=1}^{m-2^j+1} 2^j \log m\right) = O\left(\sum_{j=0}^{q} 2^j m \log m\right) = O(2^q m \log m) = O(m\Delta \log m)$$

time. Now we show how to handle a subrow minimum query $S(r, a, b)$ on $\mathcal{M}_0$, where $b - a + 1 \leq \Delta$. Let u be the greatest integer such that $2^u \leq b - a + 1$. Then we can cover our subrow minimum query with two possibly overlapping queries of length 2^u. Hence, to answer $S(r, a, b)$ it is enough to find the minimum in row r in $\mathcal{M}^u_a$ and the minimum in row r in $\mathcal{M}^u_{b-2^u+1}$ and return the smaller one. By Lemma 5.5.3, this takes $O(1)$ time. □

The partition of $\mathcal{M}$ into rectangular matrices. We partition the staircase Monge matrix $\mathcal{M}$ into $O(m \log^{\epsilon/2} m)$ non-overlapping *rectangular* Monge matrices $\mathcal{M}_1, \ldots, \mathcal{M}_q$ using Lemma 5.5.1. Each $\mathcal{M}_i$ is a subrectangle of $\mathcal{M}$, and each row r (column c) appears in a set W_r (W^c, respectively) of $O\left(\frac{\log m}{\log \log m}\right)$ ($O\left(\frac{\log^{1+\epsilon/2} m}{\log \log m}\right)$, respectively) subrectangles. Every element of $\mathcal{M}$ is covered by exactly one matrix $\mathcal{M}_i$. We precompute all the sets W_r and W^c after computing the partition.

We build the block data structure of Section 5.4 for each $\mathcal{M}_i$. For each $\mathcal{M}_i$, we use the same block size Δ. As each $\mathcal{M}_i$ is a subrectangle of $\mathcal{M}$, Lemma 5.5.2 guarantees that we can perform subrow minimum queries on $\mathcal{M}_i$ spanning at most Δ columns in $O(1)$ time. For brevity, we identify the matrix $\mathcal{M}_i$ and its associated data structure. We use the dot notation to denote operations acting on specific matrices, e.g., $\mathcal{M}_i$.Activate-Row.

For each matrix $\mathcal{M}_i$, we use notation analogous as in previous sections: R_i and C_i are the sets of rows and columns of $\mathcal{M}_i$, respectively. Let $k_i = |R_i|$ and $l_i = |C_i|$. Denote by $\overline{R}_i$ the set of active rows of $\mathcal{M}_i$.

Recall that the blocks of the matrix $\mathcal{M}_i$ are partitioned into two sets $\overline{\mathcal{B}}_i$ and $\mathcal{B}_i \setminus \overline{\mathcal{B}}_i$. Denote by $block(\mathcal{M}_i)$ the submatrix $\mathcal{M}_i(\overline{R}_i, \bigcup \overline{\mathcal{B}}_i)$ and by $exact(\mathcal{M}_i)$ the submatrix $\mathcal{M}_i(\overline{R}_i, \bigcup(\mathcal{B}_i \setminus \overline{\mathcal{B}}_i))$.

Main priority queue H. The core of our data structure is a priority queue H. At any time, H contains an element c for each column $c \in \overline{C}$ and at most one element $\mathcal{M}_i$ for each matrix $\mathcal{M}_i$. Thus the size of H never exceeds $O(m \log^{\epsilon/2} m)$. We maintain the following invariants after the initialization and each call Activate-Row or Ensure-Bound-And-Get resulting in $\overline{C} \neq \emptyset$:

H.1 For each $c \in \overline{C}$, the key of c in H is equal to

$$\min\{\mathcal{M}_i.\textsc{Current-Min}(c) : \mathcal{M}_i \in W^c\}.$$

H.2 For each $\mathcal{M}_i$ such that $block(\mathcal{M}_i)$ is not empty, the key of $\mathcal{M}_i$ in H is equal to

$$\min\{block(\mathcal{M}_i)\} = \mathcal{M}_i.\textsc{Block-Lower-Bound}().$$

Lemma 5.5.4. *Assume invariants H.1 and H.2 are satisfied. Then, H.Min-Key() $\leq \mathcal{M}(\overline{R}, \overline{C})$.*

Proof. Let $v = H.\textsc{Min-Key}()$. Assume the contrary, that there exists an element $\mathcal{M}_{r,c} < v$, where $r \in \overline{R}$ and $c \in \overline{C}$. Let $\mathcal{M}_i$ be the rectangular Monge matrix of the partition containing the element $\mathcal{M}_{r,c}$. If $c \in \bigcup \overline{\mathcal{B}}_i$, then $v > \mathcal{M}_{r,c} \geq \mathcal{M}_i.\textsc{Block-Lower-Bound}()$. But then the key of $\mathcal{M}_i$ in H is $\mathcal{M}_i.\textsc{Block-Lower-Bound}()$, a contradiction. Similarly, if $c \in \bigcup(\mathcal{B}_i \setminus \overline{\mathcal{B}}_i)$, then $\mathcal{M}_i.\textsc{Current-Min}(c) \leq \mathcal{M}_{r,c} < v$, a contradiction. □

5.5.3 Implementing the Operations

Initialization. At the initialization time we first build the short subrow minimum query data structure of Lemma 5.5.2. Then, the data structure of Lemma 5.4.3 is initialized for each $\mathcal{M}_i$. The total time needed to initialize these structures is thus

$$O\left(m\Delta \log m + m\frac{\log^{1+\epsilon/2} m}{\log \log m} + \frac{m}{\Delta}\frac{\log^{2+\epsilon/2} m}{\log \log m}\right) = O\left(m \log^{2-\epsilon/2} m\right).$$

Next, we insert into the priority queue H an element c with key ∞ for each $c \in C$ and an element $\mathcal{M}_i$ with key ∞ for each matrix $\mathcal{M}_i$. This takes additional $O(m \log^{\epsilon/2} m)$ time. Clearly, invariants H.1 and H.2 are satisfied immediately after the initialization.

Lower-Bound. By Lemma 5.5.4, the value $v = H.\textsc{Min-Key}()$ is a lower bound on the value $\min\{\mathcal{M}(\overline{R}, \overline{C})\}$. The function Lower-Bound returns v and thus works in $O(1)$ time.

Activate-Row. The call Activate-Row(r) may require changes to some keys of the entries of H in order to satisfy invariants H.1 and H.2. However, the activation of r does not alter what the functions $\mathcal{M}_i.\textsc{Current-Min}(c)$ or $\mathcal{M}_i.\textsc{Block-Lower-Bound}()$ return for matrices $\mathcal{M}_i \notin W_r$. For all $\mathcal{M}_i \in W_r$ we call $\mathcal{M}_i.\textsc{Activate-Row}(r)$. By Lemma 5.4.3, the columns c_j of $exact(\mathcal{M}_i)$ with changed minima can be read in linear time from $\mathcal{M}_i.\textsc{Updates}$. If $c_j \in \overline{C}$ and the current key of c_j in H is greater than $\mathcal{M}_i.\textsc{Current-Min}(c_j)$, we decrease key of c_j in H. Analogously, the call Activate-Row(r) can incur the change of $\mathcal{M}_i.\textsc{Block-Lower-Bound}()$ and thus we may need to decrease the key of $\mathcal{M}_i$ in H. In both cases, as the operation $H.\textsc{Decrease-Key}$ runs in $O(1)$ time, the time spent on decreasing keys in H is asymptotically no more than the running time of $\mathcal{M}_i.\textsc{Activate-Row}(r)$, and can be neglected.

Ensure-Bound-And-Get. Let $v = \textsc{Lower-Bound}() = H.\textsc{Min-Key}()$. Recall that the precondition of Ensure-Bound-And-Get requires

$$\min\{\mathcal{M}(R \setminus \overline{R}, \overline{C})\} \geq \min\{\mathcal{M}(R \setminus \overline{R}, C)\} \geq v.$$

Also, by Lemma 5.5.4, $\min\{\mathcal{M}(\overline{R}, \overline{C})\} \geq v$ so we can conclude that in fact

$$\min\{\mathcal{M}(R, \overline{C})\} = \min\left(\min\{\mathcal{M}(\overline{R}, \overline{C})\}, \min\{\mathcal{M}(R \setminus \overline{R}, \overline{C})\}\right) \geq v.$$

We now have two cases. First, if the top element of H is a column c, then from invariant H.1 we know that $c \in \overline{C}$ and for some $\mathcal{M}_j \in W^c$ we have:

$$\min\{\mathcal{M}(R, \overline{C})\} \geq \ v = \mathcal{M}_j.\textsc{Current-Min}(c) \geq \min\{\mathcal{M}(R, c)\}.$$

However, clearly $\min\{\mathcal{M}(R, c)\} \geq \min\{\mathcal{M}(R, \overline{C})\}$, so we conclude that the inequalities are in fact equalities and v is indeed the minimum in column c. In that case c is returned by Ensure-Bound-And-Get and c is removed from $\overline{C}$. It can be easily verified that after calling $H.\textsc{Extract-Min}()$ invariants H.1 and H.2 still hold. This case arises at most once for each column of C so the total cost of $H.\textsc{Extract-Min}$ calls for all columns is $O(m \log m)$.

The second case is when the top element of H is a matrix $\mathcal{M}_i$. In this case we return **nil** and do not alter the set $\overline{C}$. As $\mathcal{M}_i$ is a subrectangle of $\mathcal{M}$, by the precondition we have

$$\min\{\mathcal{M}_i(R_i \setminus \overline{R}, C_i)\} \geq \min\{\mathcal{M}(R \setminus \overline{R}, C)\} \geq v = \ \mathcal{M}_i.\textsc{Block-Lower-Bound}().$$

Hence, we can call $\mathcal{M}_i.\textsc{Block-Ensure-Bound}()$. Recall that this operation shrinks the set $\overline{\mathcal{B}}_i$ and thus we need to update H so that invariants H.1 and H.2 are satisfied. First we pop the entry $\mathcal{M}_i$ from H with $H.\textsc{Extract-Min}()$ in $O(\log m)$ time. Now, if $\overline{\mathcal{B}}_i \neq \emptyset$, we once again need to insert into H an element $\mathcal{M}_i$ with key $\mathcal{M}_i.\textsc{Block-Lower-Bound}()$ in order to satisfy invariant H.2. To satisfy invariant H.1, we decrease key of each $c_j \in \mathcal{M}_i.\textsc{Updates}$ to $\mathcal{M}_i.\textsc{Current-Min}(c_j)$ if appropriate. Again, as decreasing a key in H takes constant time, the time spent on decreasing column keys is asymptotically the same as the cost of the recent call to $\mathcal{M}_i.\textsc{Block-Ensure-Bound}$.

A call to $\mathcal{M}_i.\textsc{Block-Ensure-Bound}$ can happen at most $O(l_i/\Delta)$ times, so the additional time spent on updating H incurred by the calls to $\mathcal{M}_i.\textsc{Block-Ensure-Bound}$ is $O((l_i/\Delta) \cdot \log m)$. For the same reason, the call Ensure-Bound-And-Get returns **nil** at most $O\left(\sum_i^p l_i/\Delta\right)$ times. The total number of calls to Ensure-Bound-And-Get to compute all the column minima of $\mathcal{M}$ is thus $O\left(m \log^{\epsilon/2} m + \sum_i^p l_i/\Delta\right) = O\left(m \log^{\epsilon} m\right)$. The total cost of operations on H that were not charged to $\mathcal{M}_i.\textsc{Block-Ensure-Bound}$ calls is $O(m \log^{1+\epsilon} m)$.

Let us now compute the total time spent in the calls $\mathcal{M}_i$.Activate-Row and $\mathcal{M}_i$.Block-Ensure-Bound. We have:

$$\begin{aligned}\sum_i O\left(k_i\left(\Delta+\frac{\log m}{\log\log m}\right)+l_i+\frac{l_i}{\Delta}\log m\right) &= O\left(\frac{\log m}{\log\log m}\sum_i k_i+\log^{\epsilon/2} m\sum_i l_i\right)\\ &= O\left(m\left(\frac{\log m}{\log\log m}\right)^2+m\log^{1+\epsilon} m\right)\\ &= O\left(m\left(\frac{\log m}{\log\log m}\right)^2\right).\end{aligned}$$

Lemma 5.5.5. *Let $\mathcal{M}=\mathit{off}(\mathcal{M}_0,d)$ be an $m\times m$ offset staircase Monge matrix and let $\epsilon\in(0,1)$. There exists a data structure that can be initialized in $O\left(m\log^{2-\epsilon} m\right)$ time, supporting both* Lower-Bound *and* Current-Min *in $O(1)$ time. Any sequence of* Activate-Row *and* Ensure-Bound-And-Get *operations takes $O\left(m\frac{\log^2 m}{\log^2\log m}\right)$ total time. All the column minima are computed after $O(m\log^{\epsilon} m)$ calls to* Ensure-Bound-And-Get.

Remark 5.5.6. *Lemma 5.5.5 also holds for flipped staircase matrices.*

Proof. A flipped staircase matrix $\mathcal{M}'$ can be seen as a staircase matrix $\mathcal{M}$ with both the rows and columns reversed. Each subrow minimum query on $\mathcal{M}'$ translates easily into a single subrow minimum query on $\mathcal{M}$. □

5.6 Shortest Path in Dense Distance Graphs: Details

First, let us recall the following key lemma.

Lemma 5.2.1 ([77])**.** *Each $DC(G_i)$ can be decomposed into $O(1)$ (possibly flipped) staircase Monge matrices D_i of at most $|U_i|$ rows and columns. For each $u,v\in U_i$ we have:*

- *for each $\mathcal{M}\in D_i$ such that $\mathcal{M}_{u,v}$ is defined, $\mathcal{M}_{u,v}\geq w_{DC(G_i)}(uv)$.*
- *there exists $\mathcal{M}\in D_i$ such that $\mathcal{M}_{u,v}$ is defined and $\mathcal{M}_{u,v}=w_{DC(G_i)}(uv)$.*

The decomposition can be computed in $O(|U_i|^2)$ time if U_i lies on a single face of G_i, and in $O((|V(G_i)|+|U_i|^2)\log|V(G_i)|)$ time otherwise.

Proof. Let $f_1,\ldots,f_\ell$, $\ell=O(1)$, be the faces of G_i such that $U_i\subseteq V(f_1)\cup\ldots\cup V(f_\ell)$. We denote by $U_{i,j}$ the vertices of $U_i\cap V(f_j)\neq\emptyset$ in clockwise order. Moreover, we assume that for each $u,v\in U_i$ there exists a path $u\to v$ in G_i – this is without loss of generality since each graph G_i could be easily extended with bidirectional copies of edges of G_i with very large weights so that we can tell if a path actually exists by only looking at the weight of the shortest path.

We now describe the set of (flipped) staircase Monge matrices D_i. First, for each j we add to D_i a staircase matrix $\mathcal{M}^{j+}$ and a flipped staircase matrix $\mathcal{M}^{j-}$ with rows $U_{i,j}$ and columns $U_{i,j}$. The order imposed on the rows and columns is the clockwise order on the face f_j. For $u,v\in U_{i,j}$, $u\leq v$, we set $\mathcal{M}^{j+}_{u,v}=w_{\mathrm{DC}(G_i)}(uv)=\delta_{G_i}(u,v)$. For $u\geq v$, we set $\mathcal{M}^{j-}_{u,v}=w_{\mathrm{DC}(G_i)}(uv)=\delta_{G_i}(u,v)$. The matrices $\mathcal{M}^{j+}$ and $\mathcal{M}^{j-}$ represent the distances between the vertices of $U_{i,j}$. We now prove that both $\mathcal{M}^{j+}$ and $\mathcal{M}^{j-}$ are Monge. Let v,x,y,z be some nodes of $U_{i,j}$ in clockwise order.

Assume $\mathcal{M}^{j+}_{v,y} + \mathcal{M}^{j+}_{x,z} < \mathcal{M}^{j+}_{v,z} + \mathcal{M}^{j+}_{x,y}$, or, equivalently, $\delta_{G_i}(v,y) + \delta_{G_i}(x,z) < \delta_{G_i}(v,z) + \delta_{G_i}(x,y)$. As the vertices of $U_{i,j}$ lie on a single face of a planar graph G_i, any path $v \to y$ in G_i has to cross each path $x \to z$ in G_i. Specifically, a shortest path $P_1 = v \to y$ and a shortest path $P_2 = x \to z$ have some common vertex $u \in G_i$. Thus, the total length of paths $v \xrightarrow{P_1} u \xrightarrow{P_1} y$ and $x \xrightarrow{P_2} u \xrightarrow{P_2} z$ is $\delta_{G_i}(v,y) + \delta_{G_i}(x,z)$. But the paths $v \xrightarrow{P_1} u \xrightarrow{P_2} z$ and $x \xrightarrow{P_2} u \xrightarrow{P_1} y$ also have the same total length and that length cannot be less than $\delta_{G_i}(v,z) + \delta_{G_i}(x,y)$. This contradicts $\delta_{G_i}(v,y) + \delta_{G_i}(x,z) < \delta_{G_i}(v,z) + \delta_{G_i}(x,y)$ and thus proves that $\mathcal{M}^{j+}$ is indeed a staircase Monge matrix. The proof that $\mathcal{M}^{j-}$ is a flipped staircase Monge matrix is analogous. Both $\mathcal{M}^{j+}$ and $\mathcal{M}^{j-}$ are computed in $O(|U_{i,j}|^2)$ time.

Now we describe the matrices of D_i representing the distances between vertices of U_i lying on f_j and vertices of U_i lying on f_k (for $j \neq k$). Mozes and Wulff-Nilsen ([77], Section 4.4) showed that in $O((|V(G_i)| + |U_i|^2) \log |V(G_i)|)$ time one can compute two rectangular Monge matrices $\mathcal{M}^{j,k,\mathrm{L}}$ and $\mathcal{M}^{j,k,\mathrm{R}}$, with rows $U_{i,j}$ and columns $U_{i,k}$, such that for each $u \in U_{i,j}$ and $v \in U_{i,k}$ we have $w_{\mathrm{DC}(G_i)}(uv) = \min(\mathcal{M}^{j,k,\mathrm{L}}_{u,v}, \mathcal{M}^{j,k,\mathrm{R}}_{u,v})$ (for more details about this construction, see also [58], Section 5.3). Each square Monge matrix can be easily decomposed into a staircase Monge matrix and a flipped staircase Monge matrix. A rectangular Monge matrix, in turn, can be padded with either some number of copies of the last row or some number of copies of the last column in order to make it square. Thus, for each pair (k,l), $k \neq l$, we add to D_i four (flipped) staircase Monge matrices. In total, the set D_i has $2h + 4h(h-1) = O(1)$ staircase Monge matrices, each of size no more than $|U_i| \times |U_i|$. □

Our single source shortest paths algorithm on DDG (see Algorithm 4) maintains a growing subset $S \subseteq V$ of visited vertices and the values $d(x)$ for $x \in S$. Let us define $\mathcal{D}_i$ so that for each matrix $\mathcal{M} \in D_i$ there is a corresponding offset matrix $\mathcal{M}' = \mathrm{off}(\mathcal{M}, d)$ in $\mathcal{D}_i$ with the same rows and columns. We build a data structure of Lemma 5.5.5 for each matrix in $\bigcup_{i=1}^{q} \mathcal{D}_i$. The row u of $\mathcal{M} \in \mathcal{D}_i$ is activated (see Section 5.5) immediately once u is added to S. By Fact 5.1.3, the matrices of $\mathcal{D}_i$ are Monge matrices. The matrices $\mathcal{D}_i$ are never stored explicitly. Each entry is computed from the corresponding entry in D_i and the array d in $O(1)$ time every time it is accessed.

In comparison to Dijkstra's algorithm, we relax the invariant posed on the keys in the priority queue as we cannot afford storing exact values $z(y)$ as keys of the queue entries. Recall that a distance estimate is defined as $z(y) := \min_{x \in S}\{d(x) + w_{\mathrm{DDG}}(xy) : xy = e \in E(DDG)\}$. Instead, we conceptually maintain a threshold value T so that only the keys of vertices $y \in V \setminus S$ at distance less than T are equal to $z(y)$ and the remaining keys are no less than $z(y)$. The threshold is gradually increased, which in turn allows to call Ensure-Bound-And-Get on some matrices $\mathcal{M} \in \bigcup_i \mathcal{D}_i$ and obtain better bounds on the values $z(y)$.

Specifically, we have a priority queue H storing an element x for each $x \in V \setminus S$. Denote by $key(e)$ the key of an element $e \in H$. Let W_r (W^c) be the set of all matrices of $\bigcup_i \mathcal{D}_i$ containing the row r (the column c, respectively). For a data structure $\mathcal{M}$ of Lemma 5.5.5, denote by $C^*(\mathcal{M})$ the set of columns of $\mathcal{M}$ for which the minima have been already reported.

In our algorithm, we cannot afford to set $key(y)$ for each $y \in V \setminus S$ to

$$z(y) = \min_{x \in S}\{\min\{\mathcal{M}_{x,y} : \mathcal{M} \in W_x \cap W^y\}\}$$

as would Dijkstra's algorithm do. Instead, for $y \in V \setminus S$, $key(y)$ satisfies

$$key(y) = \min_{x \in S}\{\min\{\mathcal{M}_{x,y} : \mathcal{M} \in W_x, y \in C^*(\mathcal{M})\}\}.$$

Observe that the definition of $key(y)$ implies that if $key(y) \neq \infty$, then $key(y)$ is the length of some $s \to y$ path. Indeed, it is a minimum over some values $\mathcal{M}_{x,y}$, where $\mathcal{M} \in \bigcup_i \mathcal{D}_i$, and

the values in these matrices are always set to some actual lengths of paths from s. Moreover, we clearly have $key(y) \geq z(y)$ since the minimum in the definition of $key(y)$ is over a subset of values for which the minimum in the definition of $z(y)$ is taken.

We also add $O(q)$ special elements $\{\mathcal{M} : \mathcal{M} \in \bigcup_{i=1}^{q} \mathcal{D}_i\}$ to our priority queue H. At all times we have $key(\mathcal{M}) = \mathcal{M}.\textsc{Lower-Bound}()$. We also ensure that for each $x \in S$, in every $\mathcal{M} \in W_x$ row x is activated.

Observe that the above invariants imply that for $y \in V \setminus S$ we have

$$z(y) \geq \min(key(y), \min\{\mathcal{M}.\textsc{Lower-Bound}() : \mathcal{M} \in W^y\}). \tag{5.1}$$

Indeed, for each $x \in S$ and $\mathcal{M} \in W_x \cap W^y$ such that $y \notin C^*(\mathcal{M})$, by the definition of $\mathcal{M}.\textsc{Lower-Bound}$, we have $\min\{\mathcal{M}_{x,y} : y \notin C^*(\mathcal{M})\} \geq \mathcal{M}.\textsc{Lower-Bound}()$.

Algorithm 4 Pseudocode of our single-source shortest paths algorithm. The function Dijkstra returns a vector d containing the lengths of the shortest paths from s to all other vertices of DDG. We assume that each $\mathrm{DC}(G_i)$, for $i = 1, \ldots, q$, is preprocessed, so that we can access the matrices of sets $D_1, \ldots, D_q$.

```
1:  function Dijkstra(s)
2:      Initialize the data structures of Lemma 5.5.5 for each M ∈ ⋃_i D_i.
3:      H := empty priority queue
4:      S := ∅
5:      procedure Visit(x, val)
6:          S := S ∪ {x}
7:          d(x) := val
8:          for M ∈ W_x do
9:              M.Activate-Row(x)
10:             H.Decrease-Key(M, M.Lower-Bound())
11:     for x ∈ V \ {s} do
12:         d(x) := ∞
13:         H.Insert(x, ∞)
14:     for M ∈ ⋃_i D_i do
15:         H.Insert(M, ∞)
16:     Visit(s, 0)
17:     while S ≠ V  and  H.Min-Key() ≠ ∞ do
18:         v := H.Min-Key()
19:         Z := H.Extract-Min()
20:         if Z is a vertex of DDG then
21:             Visit(Z, v)
22:         else
23:             x := Z.Ensure-Bound-And-Get()
24:             if x ≠ nil and x ∉ S then
25:                 H.Decrease-Key(x, Z.Current-Min(x))
26:             H.Insert(Z, Z.Lower-Bound())
27:     return d
```

Let un now discuss the correctness of Algorithm 4. One can easily verify that the key invariants are satisfied before the first iteration of the **while** loop in line 17.

Assume the element that gets extracted from H in line 19 is some vertex $x \in V \setminus S$. We need to prove that x has the least value $z(x)$ among all vertices of $V \setminus S$ and that $z(x) = key(x)$. This

way we will consequently prove that our algorithm simulates Dijkstra's algorithm and therefore computes distances from s correctly.

As the keys of H include all keys $key(y)$ where $y \in V \setminus S$ and all keys $key(\mathcal{M})$ for the $O(q)$ data structures $\mathcal{M}$, by inequality 5.1, for each $y \in V \setminus S$ we have $z(y) \geq key(x)$. We conclude $z(y) \geq key(x) \geq z(x)$ and in particular $z(x) \geq key(x) \geq z(x)$. So, $z(x) = key(x)$. Consequently, x has the minimal $z(x)$ among all vertices in $V \setminus S$, and since there exists a path $s \to x$ of length $key(x)$, there exists a $s \to x$ path of length $z(x)$. Therefore, x could safely be the next vertex chosen by Dijkstra's algorithm. The procedure Visit is used to update the set S, the array d and all the keys of H affected by inserting x to S.

Sometimes it also may happen that the element extracted from H is some data structure Z. In this case no vertex is added to S but our algorithm still makes progress, as explained in the following. Note that we extract some previously unknown column minimum of Z with the call Z.Ensure-Bound-And-Get(). Recall that for this to be a legal operation on Z, we need to guarantee that all the rows of Z that are not active at that point contain only values not less than Z.Lower-Bound(). Notice that after each extraction of an element with key v from H we update some other keys in H to values not less than v. Thus, each extracted element has key not less than the previously extracted elements. In particular, we know that for each $y \in V \setminus S$, $d(y) \geq Z$.Lower-Bound(). For each $\mathcal{M} \in W_y$, the values in row y are not less than $d(y)$ and hence indeed $x = Z$.Ensure-Bound-And-Get() can be called. Since for a single data structure Z, Z.Ensure-Bound-And-Get() can be called only a finite number of times before Z.Lower-Bound() becomes ∞, this situation will happen only a finite number of times per each data structure Z. Subsequently, if $x \neq \mathbf{nil}$, a column minimum of Z has been found and the key of x is updated to satisfy the key invariants. Finally, Z is reinserted into H with the key equal to the new value of Z.Lower-Bound() (possibly ∞).

Let us now bound the running time of the function Dijkstra. By Lemma 5.5.5, the initialization, along with any sequence of operations Activate-Row and Block-Ensure-Bound, can be performed on $\mathcal{M} \in \mathcal{D}_i$ in $O\left(|U_i| \left(\frac{\log |U_i|}{\log \log |U_i|}\right)^2\right) = O\left(|U_i| \frac{\log^2 n}{\log^2 \log n}\right)$ time.

The time spent on extracting elements from H is $O(I \log n)$, where I is the number of insertions into H. Clearly, H never contains more than $O(|V| + q) = O(n)$ elements. Each vertex of DDG is inserted into H at most once, and, by Lemma 5.5.5, each data structure $\mathcal{M} \in \mathcal{D}_i$ is inserted into H at most $O(|U_i| \log^\epsilon |U_i|)$ times before it reports all the column minima and $\mathcal{M}$.Lower-Bound() is set to ∞. Hence, the total time spent on the operations H.Extract-Min is $O\left(\log n \sum_i |U_i| \log^\epsilon n\right)$. The operation H.Decrease-Key takes constant time and thus we can neglect the calls to H.Decrease-Key immediately after Activate-Row or Block-Lower-Bound. Taking into account the preprocessing of Lemma 5.2.1, we finally obtain the theorem claimed in the beginning of this chapter.

Theorem 5.0.1. *The single-source shortest paths computations in DDG can be performed in* $O\left(\sum_{i=1}^{q} |U_i| \frac{\log^2 n}{\log^2 \log n}\right)$ *time. The required preprocessing time per each* G_i *is* $O(|U_i|^2)$ *if* U_i *lies on a single face of* G_i*, and* $O\left((|V(G_i)| + |U_i|^2) \log |V(G_i)|\right)$ *otherwise.*

5.7 Price Functions and Supporting Negative Edge Weights

Let G be a weighted digraph with no negative-length cycles. Let $\phi : V \to \mathbb{R}$ be a *price function*. It is a well-known fact that for any price function ϕ adding $\phi(u) - \phi(v)$ to $w_G(uv)$ for each edge $uv \in E(G)$ preserves the shortest paths.

We say that ϕ is *feasible* if for any $uv \in E(G)$, $w_G(uv) + \phi(u) - \phi(v) \geq 0$. It is also a well-known fact that if a graph G contains no negative cycles, a feasible price function for G

exists. In fact, it is easy to show that the distances from an arbitrary source vertex in G form a feasible price function for G. This observation is the basis of the famous $O(mn + n^2 \log n)$ all-pairs shortest paths algorithm of Johnson [56] that first computes a feasible price function using Bellman-Ford method and then proceeds by running Dijkstra's algorithm from each source in order to compute all-pairs shortest paths.

Note that if a feasible price function ϕ_0 for DDG is available at the preprocessing step, the algorithm of Section 5.6 can support negative edges within the same bounds as in Theorem 5.0.1. To see that, note that if we set $w_{\mathrm{DC}(G_i)}(uv) := w_{\mathrm{DC}(G_i)}(uv) + \phi_0(u) - \phi_0(v)$, Lemma 5.2.1 still holds since it only relied on the fact that certain shortest paths cross and those are preserved after the edge weights' adjustment, as discussed above.

Moreover, the shortest path algorithm of Theorem 5.0.1 can be run with different feasible price functions without repeating the costly preprocessing. Recall that this preprocessing computed the decomposition of Lemma 5.2.1, which, again, only relied on the topology of the shortest paths in G_i (which is not affected by the price function at all), and not directly on the exact weights of edges of G_i.

5.8 Implications

The implications of Theorem 5.0.1 are numerous. In this section we mention the most important results, for which (repeatedly) running FR-Dijkstra is the actual bottleneck.

Maximum Flow. The multiple-source multiple-sink maximum flow algorithm [7], the single-source all-sinks maximum flow algorithm [67], and min-cut oracles for surface-embedded graphs [5] are arguably the most advanced applications of FR-Dijkstra to date. All of them use the so-called *flow-balancing procedure* developed by Borradaile et al. in [7] (see also [79]).

The very bottleneck of the flow-balancing procedure is the computation of single-source shortest paths on a graph H with vertex set X which consists of:

- a set of $O(|X|)$ edges P with endpoints in X,
- a dense distance graph $DDG = \mathrm{DC}(G_1) \cup \ldots \cup \mathrm{DC}(G_q)$ such that $q = O(1)$ and each $U_i \subseteq X$ lies on a *single* face of G_i.

However, what is interesting, the weights $w_H(e)$ of the edges $e \in P$ vary between the computations. Moreover, neither the edge-weights of DDG nor the weights of edges P need to be non-negative. Instead, for each computation, one is provided with a *feasible price function* $\phi : X \to \mathbb{R}$ for H, which, needless to say, may also be different for different instantiations of the procedure. Recall that a feasible price function ϕ satisfies $w_H(e) + \phi(u) - \phi(v) \geq 0$ for all edges $uv = e \in E(H)$ and adding $\phi(u) - \phi(v)$ to $w_H(e)$ for each edge e preserves the shortest paths. As noted in Section 5.7, given a feasible price function one can find shortest paths in a possibly negatively-weighted graph using Dijkstra's algorithm.

In [7], the shortest paths computations, as described above, are performed in $\Theta(|X| \log^2 |X| + |P| \log |X|) = \Theta(|X| \log^2 |X|)$ time using a modification of FR-Dijkstra of [58] originally designed (among others applications) to extend the fully-dynamic planar exact distance oracle [27, 62] to also support negative edge weights.

We now show how our algorithm of Section 5.6 can also be adjusted to work in this scenario and, as a result, improve over [7]. We first note that Lemma 5.2.1 assumed non-negative edge-weights so it seems we cannot use it. However, observe that in the proof of Lemma 5.2.1 the decomposition of $\mathrm{DC}(G_i)$ such that U_i was a subset of a single face of G_i did not depend on the edge weights at all, but only on the clockwise order of U_i on that face. As we only deal

with such sets U_i in the flow-balancing procedure, we are still allowed to compute the staircase matrix decomposition during the preprocessing as even the time needed to fill the distance cliques $\mathrm{DC}(G_i)$ is clearly $\Omega\left(\sum_i^q |U_i|^2\right)$.

Moreover, similarly as we proceeded in Corollary 4.3.7, each edge of P is treated as a distance clique of size 2. Clearly, this does not violate the requirements posed on a distance clique.

Finally, in Algorithm 4 we replace each $\mathcal{M} \in \bigcup_i \mathcal{D}_i$ with $\mathcal{M}'$ defined as

$$\mathcal{M}'_{u,v} = \mathcal{M}_{u,v} + \phi(u) - \phi(v)$$

for all relevant pairs (u, v). Such an adjustment clearly preserves the Monge property. We are not allowed to replace these matrices explicitly, though, as this would be too costly. Instead, we compute the individual values of each $\mathcal{M}'$ accessed by the data structures of Lemma 5.5.5 on the fly. This only increases the initialization and operation time of these data structures by a constant factor. Therefore, by Theorem 5.0.1, given the weights of edges P and ϕ, the total time needed to compute the single-source shortest paths in H is

$$O\left(\left(\sum_{i=1}^{q} |U_i| + |P|\right) \frac{\log^2 |X|}{\log^2 \log |X|}\right) = O\left(|X| \frac{\log^2 |X|}{\log^2 \log |X|}\right).$$

Each of the algorithms [5, 7, 67] recursively decomposes the input graph using cycle separators and runs the flow-balancing procedure at each level of the recursion. Hence, all three algorithms spend $\Omega(n \log^3 n)$ on these computations. However, only in [7] and [67] the flow-balancing procedure is the only bottleneck. As a result, we obtain the following.

Corollary 5.8.1. *The multiple-source multiple-sink max-flow, the bipartite matching, and the single-source all-sinks max-flow in a planar digraph can all be computed in $O\left(n \frac{\log^3 n}{\log^2 \log n}\right)$ time.*

Exact Distance Oracles. Mozes and Sommer [76] considered the following problem. Given a planar digraph $G = (V, E)$ with real edge lengths and space allocation $S \in [n \log \log n, n^{2-\epsilon}]$ ($\epsilon > 0$), construct a data structure of size $O(S)$ answering exact distance queries in G as efficiently as possible. Their trade-off data structure is very similar to our trade-off data structure for decremental transitive closure from Section 4.7, which was in fact heavily inspired by the distance oracle of Mozes and Sommer. Roughly speaking, it sets $S = n^2/t$ and for each piece H of $O(n/t)$ recursive decompositions it builds distance cliques defined similarly as the graphs $\mathrm{In}(H)$ and $\mathrm{Ex}^*(H)$ from Section 4.7.

At the heart of their query algorithm lies the basic version of FR-Dijkstra (without using price functions), and thus Theorem 5.0.1 implies a faster query algorithm.

Corollary 5.8.2. *Given a weighted planar digraph G, an $O(S)$-space exact distance oracle answering queries in $O\left(\frac{n}{\sqrt{S}} \frac{\log^2 n}{\log^{1/2} \log n}\right)$ time can be constructed in $O\left(S \frac{\log^3 n}{\log \log n}\right)$ time.*

Very recently, a much better space-query trade-off for exact distance oracles for planar graphs has been shown [36]. However, the oracle of Mozes and Sommer remains the most efficient solution if we require the preprocessing time to be nearly-linear in S.

Fully-Dynamic Distance and Max-Flow Oracles. In this problem we are given a plane digraph G with real edge lengths which undergoes embedding-preserving edge insertions and deletions. A straightforward combination of data structures of [55] and [58] results in a data structure supporting both edge set updates and queries in $O(n^{2/3} \log^{5/3} n)$ time even when negative edge lengths are allowed.

Computation of single-source shortest paths in a dense distance graph over an r-division and the recomputation of a dense distance graph using the multiple-source shortest paths data structure [13, 62] in $O(r \log r)$ time constitute the bottlenecks of the update procedure. The terms $O(r \log n)$ and $O\left(\frac{n}{\sqrt{r}} \frac{\log^2 n}{\log^2 \log n}\right)$ can be balanced for $r = n^{2/3} \frac{\log^{2/3} n}{\log^{4/3} \log n}$.

Corollary 5.8.3. *For a planar digraph G, a dynamic distance oracle supporting edge updates and queries in $O\left(n^{2/3} \frac{\log^{5/3} n}{\log^{4/3} \log n}\right)$ amortized time can be constructed in $O\left(n \frac{\log^2 n}{\log \log n}\right)$ time.*

As shown in [55], Corollary 5.8.3 also implies an $O(\log^{4/3} \log n)$ improvement of the fully dynamic algorithm for computing maximum s, t-flow values in an undirected plane graph.

Chapter 6

Open Problems

In Chapter 3 we showed an optimal solution to a certain data-structural problem on planar graphs. However, for the problems of decremental transitive closure and computing shortest paths in dense distance graphs, which we considered in Chapters 4 and 5 respectively, there is no good evidence to believe that our algorithms are even close to optimal.

Specifically, we find the following questions interesting:

1. Does there exist a decremental transitive closure algorithm for planar digraphs with $O(n^{1-\epsilon})$ amortized update time and $O(\text{polylog}\, n)$ query time, where $\epsilon > 0$? Recall that in our trade-off algorithm of Theorem 4.7.4 we could achieve polylogarithmic query time by setting $t = n$, but then the amortized update time would become $\widetilde{\Omega}(n)$.

   Note that it is not even clear why a polylogarithmic update/query time decremental transitive closure should not be possible: for planar graphs there exists a static linear-space reachability oracle with constant query time. The oracle can be built in optimal linear time [50]. Hence, in planar graphs, as opposed to general graphs, static reachability is computationally not harder than connectivity.

2. Does there exists a single-source shortest paths algorithm for dense distance graphs with $O(b \log^{2-\epsilon} n)$ overhead per distance clique with b-vertices, where $\epsilon > 0$? It seems that one possible approach to achieve such an algorithm could be to handle staircase Monge matrices more directly, without reducing them to a number of rectangular Monge matrices. However, it is not clear how to do it either.

3. Is it possible to obtain a faster algorithm for single-source shortest paths in dense distance graphs if we assume that the dense distance graph is integer-weighted? Note that in the case of general digraphs, if the edge weights are integral, it is indeed possible to improve upon Dijkstra's algorithm [92].

   Such an improved algorithm would imply, for example, a better algorithm for computing a maximum bipartite matching in a planar graph.

Bibliography

[1] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[2] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[3] Mudabir Kabir Asathulla, Sanjeev Khanna, Nathaniel Lahn, and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite perfect matching in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 457–476, 2018.

[4] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574, 2016.

[5] Glencora Borradaile, David Eppstein, Amir Nayyeri, and Christian Wulff-Nilsen. All-pairs minimum cuts in near-linear time for surface-embedded graphs. In *32nd International Symposium on Computational Geometry, SoCG 2016, June 14-18, 2016, Boston, MA, USA*, pages 22:1–22:16, 2016.

[6] Glencora Borradaile and Philip N. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *J. ACM*, 56(2):9:1–9:30, 2009.

[7] Glencora Borradaile, Philip N. Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. *SIAM J. Comput.*, 46(4):1280–1303, 2017.

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

[9] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273, 2004.

[10] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(9):1124–1137, 2004.

[11] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, pages 342–351, 1999.

[12] Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on*

Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pages 2143–2152, 2017.

[13] Sergio Cabello, Erin W. Chambers, and Jeff Erickson. Multiple-source shortest paths in embedded graphs. *SIAM J. Comput.*, 42(4):1542–1571, 2013.

[14] Timothy M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008.

[15] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1900–1918, 2017.

[16] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Łącki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\widetilde{O}(m\sqrt{n})$ total update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 315–324, 2016.

[17] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985.

[18] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{o}(m^{10/7} \log W)$ time (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 752–771, 2017.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[20] Camil Demetrescu and Giuseppe F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

[21] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[22] Krzysztof Diks and Piotr Sankowski. Dynamic plane transitive closure. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 594–604, 2007.

[23] Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, pages 449–467, 1965.

[24] David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *16th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2008, November 5-7, 2008, Irvine, California, USA, Proceedings*, page 16, 2008.

[25] Jeff Erickson. Maximum flows and parametric shortest paths in planar graphs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 794–804, 2010.

[26] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.

[27] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.

[28] Greg N. Frederickson. On linear-time algorithms for five-coloring planar graphs. *Inf. Process. Lett.*, 19(5):219–224, 1984.

[29] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[30] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[31] Harold N. Gabow, Haim Kaplan, and Robert Endre Tarjan. Unique maximum matching algorithms. *J. Algorithms*, 40(2):159–183, 2001.

[32] Pawel Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. Voronoi diagrams on planar graphs, and computing the diameter in deterministic $\tilde{O}(n^{5/3})$ time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 495–514, 2018.

[33] Pawel Gawrychowski and Adam Karczmarz. Improved bounds for shortest paths in dense distance graphs. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 61:1–61:15, 2018.

[34] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Submatrix maximum queries in monge matrices are equivalent to predecessor search. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 580–592, 2015.

[35] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Submatrix maximum queries in monge matrices are equivalent to predecessor search. *CoRR*, abs/1502.07663, 2015.

[36] Pawel Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 515–529, 2018.

[37] Loukas Georgiadis, Thomas Dueholm Hansen, Giuseppe F. Italiano, Sebastian Krinninger, and Nikos Parotsidis. Decremental data structures for connectivity and dominators in directed graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 42:1–42:15, 2017.

[38] Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-edge connectivity in directed graphs. *ACM Trans. Algorithms*, 13(1):9:1–9:24, 2016.

[39] Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-edge connectivity in directed graphs. *ACM Trans. Algorithms*, 13(1):9:1–9:24, 2016.

[40] Dora Giammarresi and Giuseppe F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 16(3):263–287, 1996.

[41] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995.

[42] Michael T. Goodrich. Planar separators and parallel polygon triangulation. *J. Comput. Syst. Sci.*, 51(3):374–389, 1995.

[43] Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theor. Comput. Sci.*, 203(1):123–141, 1998.

[44] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 674–683, 2014.

[45] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, Proceedings, Part I*, pages 725–736, 2015.

[46] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 664–672, 1995.

[47] Monika Rauch Henzinger, Philip N. Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.

[48] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[49] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, pages 50:1–50:15, 2017.

[50] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Planar reachability in linear space and constant time. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 370–389, 2015.

[51] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[52] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log \log n)^2)$ amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete 16-19*, pages 510–520, 2017.

[53] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(3):273–281, 1986.

[54] Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Piotr Sankowski. Decremental single-source reachability in planar digraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1108–1121, 2017.

[55] Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian 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 2011, San Jose, CA, USA, 6-8 June 2011*, pages 313–322, 2011.

[56] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[57] Ming-Yang Kao. Linear-processor NC algorithms for planar directed graphs I: strongly connected components. *SIAM J. Comput.*, 22(3):431–459, 1993.

[58] Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and partial monge matrices, and their applications. *ACM Trans. Algorithms*, 13(2):26:1–26:42, 2017.

[59] Adam Karczmarz. Decremental transitive closure and shortest paths for planar digraphs and beyond. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 73–92, 2018.

[60] David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA.*, pages 21–30, 1993.

[61] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91, 1999.

[62] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 146–155, 2005.

[63] Philip N. Klein and Shay Mozes. Optimization algorithms for planar graphs, 2017.

[64] Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 505–514, 2013.

[65] Jakub Łącki, and Piotr Sankowski and. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 155–166, 2011.

[66] Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Alg.*, 9(3):27:1–27:15, 2013.

[67] Jakub Łącki and Yahav Nussbaum and Piotr Sankowski and Christian Wulff-Nilsen. Single source - all sinks max flows in planar digraphs. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 599–608, 2012.

[68] Jakub Łącki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. *Theory Comput. Syst.*, 61(4):1037–1053, 2017.

[69] Martin Mareš. Two linear time algorithms for mst on minor closed graph classes. *Archivum mathematicum*, 40(3):315–320, 2002.

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

[71] David W. Matula, Yossi Shiloach, and Robert E. Tarjan. Two linear-time algorithms for five-coloring a planar graph. Technical report, Stanford University, Stanford, CA, USA, 1980.

[72] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. Syst. Sci.*, 32(3):265–279, 1986.

[73] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006.

[74] Shay Mozes, Kirill Nikolaev, Yahav Nussbaum, and Oren Weimann. Minimum cut of directed planar graphs in $O(n \log \log n)$ time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 477–494, 2018.

[75] Shay Mozes, Yahav Nussbaum, and Oren Weimann. Faster shortest paths in dense distance graphs, with applications. *Theor. Comput. Sci.*, 711:11–35, 2018.

[76] Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 209–222, 2012.

[77] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n\log^2 n/\log\log n)$ time. In *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part II*, pages 206–217, 2010.

[78] J. Ian Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971.

[79] Yahav Nussbaum. *Network flow problems in planar graphs.* PhD thesis, 2014.

[80] Mihai Patrascu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 166–175, 2014.

[81] Neil Robertson, Daniel P. Sanders, Paul D. Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 571–575, 1996.

[82] Liam Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1):6:1–6:16, 2008.

[83] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.

[84] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM J. Comput.*, 45(3):712–733, 2016.

[85] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 509–517, 2004.

[86] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency.* Number t. 1 in Algorithms and Combinatorics. Springer, 2003.

[87] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[88] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[89] Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings*, pages 372–383, 1993.

[90] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[91] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

[92] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.*, 69(3):330–353, 2004.

[93] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[94] Wikipedia. Monge array — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Monge_array`, 2018. [online; accessed 02-11-2018].

[95] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1757–1769, 2013.