Fast Algorithms for Separable Linear Programs

Sally Dong[*]
University of Washington
sallyqd@uw.edu

Gramoz Goranci
University of Vienna
gramoz.goranci@univie.ac.at

Lawrence Li[†]
University of Toronto
lawrenceli@cs.toronto.edu

Sushant Sachdeva[‡]
University of Toronto
sachdeva@cs.toronto.edu

Guanghao Ye[§]
Massachusetts Institute of Technology
ghye@mit.edu

October 26, 2023

Abstract

In numerical linear algebra, considerable effort has been devoted to obtaining faster algorithms for linear systems whose underlying matrices exhibit structural properties. A prominent success story is the method of generalized nested dissection [Lipton-Rose-Tarjan'79] for separable matrices. On the other hand, the majority of recent developments in the design of efficient linear program (LP) solves do not leverage the ideas underlying these faster linear system solvers nor consider the separable structure of the constraint matrix.

We give a faster algorithm for separable linear programs. Specifically, we consider LPs of the form $\min_{\mathbf{A}\boldsymbol{x}=\boldsymbol{b},\boldsymbol{\ell}\leq\boldsymbol{x}\leq\boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{x}$, where the graphical support of the constraint matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ is $O(n^\alpha)$-separable. These include flow problems on planar graphs and low treewidth matrices among others. We present an $\widetilde{O}((m + m^{1/2+2\alpha}) \log(1/\epsilon))$ time algorithm for these LPs, where ϵ is the relative accuracy of the solution.

Our new solver has two important implications: for the k-multicommodity flow problem on planar graphs, we obtain an algorithm running in $\widetilde{O}(k^{5/2}m^{3/2})$ time in the high accuracy regime; and when the support of $\mathbf{A}$ is $O(n^\alpha)$-separable with $\alpha \leq 1/4$, our algorithm runs in $\widetilde{O}(m)$ time, which is nearly optimal. The latter significantly improves upon the natural approach of combining interior point methods and nested dissection, whose time complexity is lower bounded by $\Omega(\sqrt{m}(m + m^{\alpha\omega})) = \Omega(m^{3/2})$, where ω is the matrix multiplication constant. Lastly, in the setting of low-treewidth LPs, we recover the results of [DLY21a] and [GS22] with significantly simpler data structure machinery.

Contents

1 Introduction

Linear programming (LP) is a widely used technique for solving a broad range of problems that emerge in optimization, operations research, and computer science, among others. LP solvers have been a subject of research for many years, both from theoretical as well as practical perspectives. This has led to the development of several algorithmic gems such as the Simplex algorithm [Dan51], ellipsoid algorithm [Kha80] and interior point method [Kar84], to name a few.

Fast solvers for LPs via interior point methods have received considerable attention recently, especially in the theoretical computer science community. A series of improvements culminated in the recent breakthrough work of Cohen, Lee and Song [CLS21], which shows that any linear program $\min_{\mathbf{A}x=b, l\leq x \leq u} c^\top x$ with n constraints and m variables can be solved in $\widetilde{O}(m^\omega \log(1/\varepsilon))$ time, where ε is the accuracy parameter and $\omega \approx 2.3715$ is the matrix multiplication exponent [DWZ22, WXXZ23]. When $\mathbf{A}$ is a dense matrix, their running time is almost optimal as it nearly matches the $O(m^\omega)$ algorithm for solving a linear system $\mathbf{A}x = b$, which is a sub-problem of linear programming. However, the case when $\mathbf{A}$ is a sparse matrix is equally important, since the constraint matrices of many LP instances that arise in practical applications happen to be sparse.

A widely-used method for identifying structures in a sparse matrix $\mathbf{A}$ involves associating a graph with its non-zero pattern, which captures the interactions between the equations in the system. In this paper, we are interested in when said graph is *separable*; we use a weighted-version of the definition as is common in literature, such as [HKRS97]:

Definition 1.1 (Separable graphs). A (hyper-)graph $G = (V, \mathcal{E})$ is n^α-*separable* for some $\alpha \in [0, 1]$ if there exists constants $b \in (0, 1)$ and $c > 0$, such that for any vertex weight assignment w, the vertices of G can be partitioned into S, A and B such that $|S| \leq c \cdot |V|^\alpha$, there are no edges between A and B, and $\max\{w(A), w(B)\} \leq b \cdot w(V)$. We call S the *(b-)balanced vertex separator* of H (with respect to w).

A notable case is $\alpha = 1/2$, which includes the family of planar and bounded-genus graphs [LT79]. It has also been empirically observed that road networks have separators of size $n^{1/3}$[DSW14, SS15].

Building upon the seminal work of George [Geo73], Lipton Tarjan and Rose [LRT79] introduced the generalized nested dissection algorithm, which solves the linear system $\mathbf{A}x = b$ in $O(m + m^{\alpha\omega})$ time when $\mathbf{A}$ is a symmetric-positive definite matrix and the associated graph is $O(n^\alpha)$-separable. When $\alpha < 1$, this algorithm outperforms the canonical $O(m^\omega)$-time algorithm for general linear systems. Motivated by this, we ask the natural question of how to leverage the structures in the constraints to speed up linear programming:

Are there faster LP solvers for the class of problems where the constraint matrix $\mathbf{A}$ can be represented by an $O(n^\alpha)$-separable graph?

Given the constraint matrix $\mathbf{A}$, [LRT79] associates with it the unique graph whose adjacency matrix has the same non-zero pattern as $\mathbf{A}$. In the context of linear programs, we define the *dual graph* $G_{\mathbf{A}}$ of a constraint matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ to be the hypergraph with vertex set $\{1, \ldots, n\}$ corresponding to the rows of $\mathbf{A}$ and hyper-edges $\{e_1, \ldots, e_m\}$, such that vertex i is in hyperedge e_j if $\mathbf{A}_{i,j} \neq 0$.

In this paper, we present a faster solver for LPs whose dual graph is separable.

Theorem 1.2. *Given a linear program* $\min \{c^\top x \; : \; \mathbf{A}x = b, l \leq x \leq u\}$*, where* $\mathbf{A} \in \mathbb{R}^{n \times m}$ *is a full-rank matrix with* $n \leq m$*, suppose the dual graph* $G_{\mathbf{A}}$ *is* $O(n^\alpha)$*-separable with* $\alpha \in [0, 1]$*, and a balanced separator is computable in* $T(n)$ *time.*

Suppose that r is the inner radius of the polytope, namely, there is $\boldsymbol{x}$ such that $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ and $\boldsymbol{l} + r \leq \boldsymbol{x} \leq \boldsymbol{u} - r$. Let $L = \|\boldsymbol{c}\|_2$ and $R = \|\boldsymbol{u} - \boldsymbol{l}\|_2$. Then, for any $0 < \varepsilon \leq 1/2$, we can find a feasible $\boldsymbol{x}$ with high probability such that

$$\boldsymbol{c}^\top \boldsymbol{x} \leq \min_{\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{x} + \varepsilon \cdot LR,$$

in time

$$\widetilde{O}\left((m + m^{1/2 + 2\alpha}) \cdot \log(R/(r\varepsilon)) + T(n)\right).$$

Our result should be compared against the natural $\widetilde{O}(m^{1/2}(m + m^{\alpha\omega}))$ runtime, which directly follows from the fact that IPM-based methods require $\widetilde{O}(\sqrt{m})$ iterations, each of which can be implemented in $O(m + m^{\alpha\omega})$ time using the nested dissection [LRT79, AY10] algorithm. For linear programs whose dual graph are $O(n^\alpha)$-separable with $\alpha \leq 1/4$, our algorithm achieves $\widetilde{O}(m \log(1/\varepsilon))$ time, which is optimal up to poly-logarithmic factors.

We would like to emphasize that $\widetilde{O}(m + m^{1/2 + 2\alpha})$ represents a natural barrier for the (robust) IPM-based approaches. At a high level, each iteration of IPM involves performing matrix operations using the inverse of an $O(m^\alpha) \times O(m^\alpha)$ matrix[1]. Even if one is given access to said inverse, multiplying a vector against it takes at least $\Omega(m^{2\alpha})$, showing that improving upon the $m^{2\alpha}$ factor will require significantly new ideas in the design and analysis of robust Interior Point Methods. Obtaining an LP solver whose time complexity is $\widetilde{O}(m + m^{\alpha\omega})$, which would in turn nearly match the time complexity for solving linear systems with recursively separable structure, remains an outstanding open problem [GKK86].

An immediate application of Theorem 1.2 is a faster algorithm for solving the (fractional) k-commodity flow problem on planar graphs to high accuracy. For general sparse graphs, an $\widetilde{O}((km)^\omega)$ time algorithm for this problem follows by the recent linear program solvers that run in matrix multiplication time [CLS21, vdB20]. It is known that solving the k-commodity flow problem is as hard as linear programming [Ita78, DKZ22], suggesting that additional structural assumptions on the input graph are necessary to obtain faster algorithms. As shown in the theorem below, our result achieves a polynomial speed-up when the input graph is planar.

Theorem 1.3. *Given a minimum-cost k-multicommodity flow problem on a planar graph $G = (V, E)$ on n vertices and m edges, with edge-vertex incidence matrix $\mathbf{B}$, integer edge capacities $\boldsymbol{u} \in \mathbb{R}_{\geq 0}^E$, integer costs $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k \in \mathbb{R}^E$ and integer demands $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_k \in \mathbb{R}^E$ for each commodity, we can solve the LP*

$$
\begin{aligned}
\min \quad & \sum_{i=1}^k \boldsymbol{c}_i^\top \boldsymbol{f}_i \\
s.t \quad & \mathbf{B}^\top \boldsymbol{f}_i = \boldsymbol{d}_i \qquad \forall i \in [k] \\
& \sum_{i=1}^k \boldsymbol{f}_i \leq \boldsymbol{u} \\
& \boldsymbol{f}_i \geq \boldsymbol{0} \qquad \forall i \in [k]
\end{aligned}
\tag{1.1}
$$

to ϵ accuracy in $\widetilde{O}(k^{2.5} m^{1.5} \log(M/\varepsilon))$ time, where M is an upper on the absolute values of $\boldsymbol{u}, \boldsymbol{c}, \boldsymbol{d}$.

Our main result also has the important advantage of recovering and simplifying the recent work by Dong, Lee and Ye [DLY21a] and Gu and Song [GS22] who obtain fast solvers for LPs whose constraint matrix has bounded treewidth.

---

[1]For $O(n^\alpha)$-separable graph, where $\alpha < 1$, it is known that $m = O(n)$, see e.g., [LRT79].

Theorem 1.4. *Suppose we have a linear program with the same setup as Theorem 1.2, and we are given a tree-decomposition of the dual graph $G_{\mathbf{A}}{}^2$ of width τ. Then we can solve the linear program in time*

$$\widetilde{O}(m\tau^2 \log(R/(\varepsilon r))) \text{ or } \widetilde{O}(m\tau^{(\omega+1)/2} \log(R/(\varepsilon r))).$$

1.1 Previous work

It is known that 2-commodity flow is as hard as linear programming [Ita78]. Recently, [DKZ22] showed a linear-time reduction from linear programs to *sparse k-commodity flow* instance, indicating that sparse k-commodity flow instances are hard to solve. This has led to renewed interest in solving k-commodity flow in restricted settings, with the authors of [vdBZ23] making progress on dense graphs.

Linear programming solvers. The quest for understanding the computational complexity of linear programming has a long and rich history in computer science and mathematics. Since the seminal works of Khachiyan [Kha80] and later Karmarkar [Kar84], who were the first to prove that LPs can be solved in polynomial time, the interior point method and its subsequent variants have become the central methods for efficiently solving linear programs with provable guarantees. This has led to a series of refined and more efficient IPM-based solvers [Ren88, Vai96, NN91, LS19, LSZ19, CLS21, JSWZ21], which culminated in the recent breakthrough work of Cohen, Lee, and Song [CLS21] who showed that an LP solver whose running time essentially matches the matrix multiplication cost, up to small low-order terms. In a follow-up work, Brand [vdB20] managed to derandomize their algorithm while retaining the same time complexity.

A problem closely related to this paper is solving LPs when the support of the constraint matrix has bounded treewidth τ. Dong, Lee and Ye [DLY21a] showed that such structured LPs can be solved in $\widetilde{O}(m\tau^2)$, which is near-linear when τ is poly-logarithmic in the parameters of the input.

High-accuracy and approximate multi-commodity flow. As mentioned above, it is known that we can solve multicommodity flow in the high-accuracy regime using linear programming. For a graph with n nodes, m edges, and k commodities, the underlying constraint matrix has km variables and $kn + m$ equality constraints. Thus, using the best-known algorithms for solving linear programs [CLS21, vdB20], one can achieve a runtime time complexity of $\widetilde{O}((km)^\omega)$ for solving multi-commodity flow. In the special case of dense graphs, Brand and Zhang [vdBZ23] recently showed an improved algorithm achieving $\widetilde{O}(k^{2.5}\sqrt{m}n^{\omega-1/2})$ runtime.

In the approximate regime, Leighton et al. [LSM+91] show that $(1 + \epsilon)$ multi-commodity flow on *undirected graphs* can be solved in $\widetilde{O}(kmn)$, albeit with a rather poor dependency on ϵ. This result led to several follow-up improvements in the low-accuracy regime [GK07, Fle00, Mad10]. Later on, breakthrough works in approximating single commodity max flow in nearly-linear time were also extended to the k-commodity flow problem on undirected graphs [KLOS14, She13, Pen16], culminating in the work of Sherman [She17] who achieved an $\widetilde{O}(mk\epsilon^{-1})$ time algorithm for the problem.

Multi-commodity flow on planar graphs. The multi-commodity flow problem on planar graphs was studied in the 1980s, but there has not been much interest in it until most recently. Results in the past focused on finding conditions under which solutions existed [OS81], or finding simple algorithms

---

[2]We can view the hypergraph $G_{\mathbf{A}}$ as a graph, where we interpret each hyper-edge as a clique, and consider its treewidth as usual.

in even more restricted settings, with the authors of [MNS85] demonstrating that the problem could be solved in $O(kn + n^2(\log n)^{1/2})$ time if the sources and sinks were all on the outer face of the graph. More recently, [KK13] studied the all-or-nothing version of planar multi-commodity flow, where flows have to be integral, and demonstrate that an $O(1)$-approximation could be achieved in polynomial time.

Max flow and min-cost flow on general graphs. In what follows, we will focus on surveying only *exact* algorithms for max-flow and min-cost flow on general graphs. For earlier developments on these problems, including fast approximation algorithms, we refer the reader to the following works [KRT94, AMO88, CKM+11, She13, KLOS14, Pen16, ST18, BGS21], and the references therein.

An important view, unifying almost all recent max-flow or min-cost flow developments, is interpreting max-flow as the problem of finding one unit of s-t flow that minimizes the ℓ_∞ congestion of the flow vector. Motivated by the near-linear Laplacian solver of Spielman and Teng [ST04] (which in turn can be used to solve the problem of finding one unit of s-t flow that minimizes the ℓ_2 congestion), and the fact that the gap between ℓ_∞ and ℓ_2 is roughly $O(\sqrt{m})$, Daitch and Spielman [DS08] showed how to implement the IPM for solving min-cost flows in $\widetilde{O}(m^{3/2})$ time.

Follow-up works initially made progress on the case of unit capacitated graphs, with the work of Madry [Mad13] achieving an $\widetilde{O}(m^{10/7})$ time algorithm for max flow and thus being the first to break the 3/2-exponent barrier in the runtime. The running time was later improved to $O(m^{4/3+o(1)})$ and it was generalized to the min-cost flow problem [AMV22, KLS20].

For general, polynomially bounded capacities, Brand et al. [vdBLL+21] gave an improved algorithm for dense graphs that runs in $\widetilde{O}(m + n^{3/2})$. In the sparse graph regime, Gao, Liu and Peng [GLP21] were the first to break the 3/2-exponent barrier by giving an $\widetilde{O}(m^{3/2-1/128})$ time algorithm, which was later improved to $\widetilde{O}(m^{3/2-1/58})$ [vdBGJ+21]. Very recently, the breakthrough work of Chen et al. [CKL+22] shows that the min-cost flow problem can be solved in $\widetilde{O}(m^{1+o(1)})$, which is optimal up to the subpolynomial term.

Max flow and min-cost flow on planar graphs. The study of flows on planar graphs dates back to the celebrated work of Ford and Fulkerson [FF56] who showed that for the case of s, t-planar graphs[3], there is an $O(n^2)$ time algorithm for max flow. This was subsequently improved to $O(n \log n)$ by Itai and Shiloach [IS79] and finally to $O(n)$ by Henzinger et al [HKRS97], the latter building upon a prior work of Hassin [Has81].

For general planar graphs, there have been two lines of work focusing on the undirected and the directed version of the problem respectively. In the first setting, Reif [Rei83] (and later Hassin and Johnson [HJ85]) gave an $O(n \log^2 n)$ time algorithm. The state-of-the-art algorithm is due to Italiano et al. [INSW11] and achieves $O(n \log \log n)$ runtime. Weihe [Wei97] gave the first speed-up for directed planar max flow running in $O(n \log n)$ time. However, his algorithm required some assumptions on the connectivity of the input graph. Later on, Borradaile and Klein [Bor08] gave an $O(n \log n)$ algorithm for general planar directed graphs. Generalization of planar graphs, e.g., graphs of bounded genus have also been studied in the context of the max flow problem. The work of Chambers et al. [CEFN23] showed that these graphs also admit near-linear time max flow algorithms.

Imai and Iwano [II90] obtained an $O(n^{1.594} \log M)$ min-cost flow algorithm for graphs that are $O(\sqrt{n})$ recursively separable. For the min-cost flow problem on planar graphs with unit capacities, Karczmarz and Sankowski [KS19] gave an $O(n^{4/3})$ algorithm. Very recently, Dong et al. [DGG+22]

---

[3] planar graphs where s and t lie on the same face

showed that the min-cost flow on planar directed graphs with polynomially bounded capacities admits an $\widetilde{O}(n)$ time algorithm, which is optimal up to polylogarithmic factors.

1.2 Technical overview

Our algorithm framework builds on the work of Dong-Gao-Goranci-Lee-Peng-Sachdeva-Ye on planar min-cost flow [DGG+22]. We solve our linear program using the robust interior point method used in [DLY21a, DGG+22], where we maintain feasible primal and dual solutions $\boldsymbol{x}$ and $\boldsymbol{s}$ to the linear program that converge to the optimal solution over $\widetilde{O}(\sqrt{m})$-many steps of IPM. At every step, we want to move our solutions in the direction of steepest descent of the objective function. To stay close to the central path and avoid violating the capacity lower and upper bounds, the IPM controls the weights $\mathbf{W}$ on the variables and the step direction $\boldsymbol{v}$, in order to limit the magnitude of the update to a variable as it approach its bounds. Both $\mathbf{W}$ and $\boldsymbol{v}$ are defined to be entry-wise dependent on the current solution $\boldsymbol{x}$ and $\boldsymbol{s}$. To maintain feasibility of the solutions, we apply the weighted projection $\mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$ matrix to the desired step direction $\boldsymbol{v}$, which ensures the resulting $\boldsymbol{x}$ and $\boldsymbol{s}$ after a step remain in their respective feasible subspaces. In robust IPMs, we also maintain entry-wise approximations $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$ to $\boldsymbol{x}$ and $\boldsymbol{s}$, and use these approximations to compute $\boldsymbol{w}, \boldsymbol{v}$, and $\mathbf{P}_{\boldsymbol{w}}$ at every step. By limiting the updates to $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$, robust IPMs achieve efficient runtimes.

The key challenge in the RIPM framework is to implement each step efficiently, specifically, computing the projection $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$, as well as updating $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$. Similar to [DGG+22], we use a *separator tree* to recursively factor the term $\mathbf{A}\mathbf{W}\mathbf{A}^\top$ in $\mathbf{P}_{\boldsymbol{w}}$ via nested dissection and recursive Schur complements. However, there are several challenges in applying the framework from [DGG+22] to general linear programs: In flow problems, the constraint matrix $\mathbf{A}$ is the vertex-edge incidence matrix of the underlying graph, and therefore $\mathbf{A}\mathbf{W}\mathbf{A}^\top$, as well as all recursive Schur complements along the separator tree, are weighted Laplacian matrices, for which we have efficient nearly-linear-time solvers [ST04] and sparse approximations to Schur complements [KS16, GHP18]. This allows [DGG+22] to work with an approximate $\widetilde{\mathbf{P}}_{\boldsymbol{w}} \approx \mathbf{P}_{\boldsymbol{w}}$ efficiently, with implicit access via a collection of approximate Schur complements that can be viewed as sparse Laplacians.

In the context of general separable linear programs, we do not have fast solvers or sparse approximate Schur complements, so instead, we must maintain the collection of Schur complements and their inverses explicitly, by computing them in a bottom-up fashion using the separator tree. To bound the update time, we show that a rank-k update to $\mathbf{A}\mathbf{W}\mathbf{A}^\top$ induced by changes in $\mathbf{W}$ corresponds to rank-k updates to all the recursive Schur complements.

Our second contribution is the dynamic data structures to maintain the implicit representations of $\boldsymbol{x}, \boldsymbol{s}$, which can be viewed as a significant refinement of those from [DGG+22]. We recall the notion of *tree operators* introduced in [DGG+22] and define an analogous *inverse tree operator*, and give simplified modular data structures to maintain $\boldsymbol{x}, \boldsymbol{s}$ using the tree and inverse tree operator. Specifically, we demonstrate more cleanly the power of nested dissection and the recursive subgraph structure in supporting efficient lazy updates to the IPM solutions.

Our third technical contribution is the definition of a fine-grained separator tree which we call the (a, b, λ)-separator tree. The parameters are defined based on the parameters of separable graphs, but they also capture important characteristics of other classes such as low-treewidth graphs. These trees guarantee that at any node, we are able to separate not only the associated graph region, but also the boundary of the region. We use them to maintain the tree operators from the implicit representations, and a careful analysis of node and boundary sizes allows us to conclude that the maintenance can be performed efficiently.

Finally, we note that this work recovers the treewidth LP result of [DLY21a] and [GS22] with significantly lower technical complexity. Whereas [DLY21a] constructs an *elimination tree* to directly

compute the Cholesky factorization of $\mathbf{AWA}^\top = \mathbf{LL}^\top$, we use a *separator tree* to recursively factor $\mathbf{AWA}^\top$. There is a key difference in the two tree constructions, which we believe this paper is correct in: To construct an elimination tree, [DLY21a] finds a balanced vertex separator S of $G_\mathbf{A}$, *remove S from $G_\mathbf{A}$ yielding two disconnected subgraphs H_1, H_2*, recursively construct the elimination tree for H_1 and H_2, and attach them as children to a vertical path of length $|S|$ corresponding to the vertices of S. When the treewidth of $\mathbf{A} \in \mathbb{R}^{n \times m}$ is t, this process results in an elimination tree of height $\widetilde{O}(t)$ where each node corresponds to a vertex of $G_\mathbf{A}$, which can then be used to identify explicit coordinates in the Cholesky factor $\mathbf{L}$ to update when $\mathbf{W}$ changes. Next, an extremely involved transformation using *heavy-light decomposition* is needed to turn the elimination tree into a *sampling tree* of height $O(\log n)$, in order to facilitate the sampling of entries from some implicit vector of the form $\mathbf{A}^\top \mathbf{L}^{-\top} \boldsymbol{z}$ (required for maintaining $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$). In contrast, to construct our separator tree, we find a balanced vertex separator S of $G_\mathbf{A}$, but *include S in both subgraphs that are recursed on*, and partition the hyperedges in $\mathcal{E}(S)$ arbitrarily between the two subgraphs. The resulting separator tree has height $O(\log n)$, where each node corresponds to a subgraph of $G_\mathbf{A}$, and each level of the tree gives a partition of the columns of $\mathbf{A}$. This recursive partitioning gives a cleaner, recursive rather than brute-force factorization of $\mathbf{AWA}^\top$, and leads to a significant difference in the data structures. When $\mathbf{W}$ changes at a step, [DLY21a] updates the Cholesky factorization by processing one changed coordinate at a time ([GS22] processes one block at a time), so that the data structure update time is linear in the number of new coordinates. On the other hand, our separator tree allows us to update $\mathbf{W}$ in one pass through the tree and yields a sublinear dependence on the number of new coordinates. Moreover, as each node in our separator tree naturally corresponds to a subset of columns of $\mathbf{A}$, we can use it in a much more straight-forward manner to sample coordinates of $\mathbf{A}^\top \mathbf{L}^{-\top} \boldsymbol{z}$.

2   Preliminaries

General Notations.   *We assume all matrices and vectors in an expression have matching dimensions.* That is, we will trivially pad matrices and vectors with zeros when necessary. This abuse of notation is unfortunately unavoidable as we will be considering lots of submatrices and subvectors.

An event holds with high probability if it holds with probability at least $1 - n^c$ for arbitrarily large constant c. The choice of c affects guarantees by constant factors.

We use boldface lowercase variables to denote vectors, and boldface uppercase variables to denote matrices. We use $\|\boldsymbol{v}\|_2$ to denote the 2-norm of vector $\boldsymbol{v}$ and $\|\boldsymbol{v}\|_\mathbf{M}$ to denote $\sqrt{\boldsymbol{v}^\top \mathbf{M} \boldsymbol{v}}$. We use $\mathrm{nnz}(\boldsymbol{v})$ to denote the number of non-zero entries in the vector $\boldsymbol{v}$, equivalently, it is the zero-norm. For any vector $\boldsymbol{v}$ and scalar x, we define $\boldsymbol{v} + x$ to be the vector obtained by adding x to each coordinate of $\boldsymbol{v}$ and similarly $\boldsymbol{v} - x$ to be the vector obtained by subtracting x from each coordinate of $\boldsymbol{v}$. We use $\mathbf{0}$ for all-zero vectors and matrices where dimensions are determined by context.

For an index set A, we use $\mathbf{1}_A$ for the vector with value 1 on coordinates in A and 0 everywhere else. We use $\mathbf{I}$ for the identity matrix and $\mathbf{I}_S$ for the identity matrix in $\mathbb{R}^{S \times S}$. For any vector $\boldsymbol{x} \in \mathbb{R}^S$, $\boldsymbol{x}|_C$ denotes the sub-vector of $\boldsymbol{x}$ supported on $C \subseteq S$; *more specifically, $\boldsymbol{x}|_C \in \mathbb{R}^S$, where $\boldsymbol{x}_i = 0$ for all $i \notin C$*.

For any matrix $\mathbf{M} \in \mathbb{R}^{A \times B}$, we use the convention that $\mathbf{M}_{C,D}$ denotes the sub-matrix of $\mathbf{M}$ supported on $C \times D$ where $C \subseteq A$ and $D \subseteq B$. When $\mathbf{M}$ is not symmetric and only one subscript is specified, as in $\mathbf{M}_D$, this denotes the sub-matrix of $\mathbf{M}$ supported on $A \times D$. To keep notations simple, $\mathbf{M}^{-1}$ will denote the inverse of $\mathbf{M}$ if it is an invertible matrix and the Moore-Penrose pseudo-inverse otherwise.

For any vector $\boldsymbol{v}$, we use the corresponding capitalized letter $\mathbf{V}$ to denote the diagonal matrix

with $\boldsymbol{v}$ on the diagonal.

For two positive semi-definite matrices $\mathbf{L}_1$ and $\mathbf{L}_2$, we write $\mathbf{L}_1 \approx_t \mathbf{L}_2$ if $e^{-t}\mathbf{L}_1 \preceq \mathbf{L}_2 \preceq e^t\mathbf{L}_1$, where $\mathbf{A} \preceq \mathbf{B}$ means $\mathbf{B} - \mathbf{A}$ is positive semi-definite. Similarly we define $\geq_t$ and $\leq_t$ for scalars, that is, $x \leq_t y$ if $e^{-t}x \leq y \leq e^t x$.

When multiplying two matrices of differing sizes, say an $m \times n$ matrix with an $n \times k$ matrix, we decompose both matrices into blocks of size $\min\{m, n, k\}$. We then perform block matrix multiplication, with fast matrix multiplication used for the multiplication of two blocks. For example, multiplying a $m \times n$ matrix with an $n \times n$ matrix, with $m \geq n$, takes $(m/n)(n^\omega)$ time.

Trees. For a tree $\mathcal{T}$, we write $H \in \mathcal{T}$ to mean H is a node in $\mathcal{T}$. We write $\mathcal{T}_H$ to mean the complete subtree of $\mathcal{T}$ rooted at H. We say a node A is an ancestor of H and H is a descendant of A if H is in the subtree rooted at A, and $H \neq A$. Given a set of nodes $\mathcal{H}$, we use $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$ to denote the set of all nodes in $\mathcal{T}$ that are ancestors of some node in $\mathcal{H}$ unioned with $\mathcal{H}$.

The *level* of a node in a tree has the following properties: the root is at level 0; the maximum level is one less than the height of the tree; and the level of a node must be at least one greater than the level of its parent, but this difference does not have to be equal to one. We may assign levels to nodes arbitrarily as long as the above is satisfied. We use $\mathcal{T}(i)$ to denote the collection of all nodes at level i in tree $\mathcal{T}$.

IPM data structures. When we discuss data structures in the context of the IPM, step 0 means the initialization step. For $k > 0$, step k means the k-th iteration of the while-loop in SOLVE (Algorithm 2); that is, it is the k-th time we update the current solutions. For any vector or matrix $\boldsymbol{x}$ used in the IPM, we use $\boldsymbol{x}^{(k)}$ to denote the value of $\boldsymbol{x}$ at the end of the k-th step.

In all procedures in these data structures, we assume inputs are given by the set of changed coordinates and their values, *compared to the previous input.* Similarly, we output a vector by the set of changed coordinates and their values, compared to the previous output. This can be implemented by checking memory for changes.

3   Overview of RIPM framework

In this section, we set up the general framework for solving a linear program using a robust IPM. We show that if the projection matrix from the IPM can be maintained efficiently based on the structure of its sparsity pattern, then the overall IPM can be implemented efficiently.

3.1   Robust interior point method

Theorem 3.1 (RIPM). *Consider the linear program*

$$\min_{\mathbf{A}\boldsymbol{x}=\boldsymbol{b},\, \boldsymbol{l}\leq\boldsymbol{x}\leq\boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{x}$$

with $\mathbf{A} \in \mathbb{R}^{n \times m}$. We are given a scalar $r > 0$ such that there exists *some interior point $\boldsymbol{x}_\circ$ satisfying* $\mathbf{A}\boldsymbol{x}_\circ = \boldsymbol{b}$ *and $\boldsymbol{l} + r \leq \boldsymbol{x}_\circ \leq \boldsymbol{u} - r$. Let $L = \|\boldsymbol{c}\|_2$ and $R = \|\boldsymbol{u} - \boldsymbol{l}\|_2$. For any $0 < \varepsilon \leq 1/2$, the algorithm RIPM (Algorithm 2) finds $\boldsymbol{x}$ such that $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$ and*

$$\boldsymbol{c}^\top \boldsymbol{x} \leq \min_{\mathbf{A}\boldsymbol{x}=\boldsymbol{b},\, \boldsymbol{l}\leq\boldsymbol{x}\leq\boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{x} + \varepsilon L R.$$

Furthermore, the algorithm has the following properties:

- *Each call of SOLVE involves $O(\sqrt{m}\log m \log(\frac{mR}{\epsilon r}))$-many steps, and $\bar{t}$ is only updated $O(\log m \log(\frac{mR}{\epsilon r}))$-many times.*

- *In each step of SOLVE, the coordinate i in $\boldsymbol{w}, \boldsymbol{v}$ changes only if $\overline{\boldsymbol{x}}_i$ or $\overline{\boldsymbol{s}}_i$ changes.*

- *In each step of SOLVE, $h\|\boldsymbol{v}\|_2 = O(\frac{1}{\log m})$.*

- *Line 19 to Line 21 takes $O(K)$ time in total, where K is the total number of coordinate changes in $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$.*

We note that this algorithm only requires access to $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}})$, but not $(\boldsymbol{x}, \boldsymbol{s})$ during the main while loop. Hence, $(\boldsymbol{x}, \boldsymbol{s})$ can be implicitly maintained via any data structure. We only require $(\boldsymbol{x}, \boldsymbol{s})$ explicitly when returning the approximately optimal solution at the end of the algorithm Line 26.

3.2 Projection operators

At step k of SOLVE with step direction $\boldsymbol{v}^{(k)}$ and weights $\boldsymbol{w}$ (we drop its superscript $^{(k)}$ for convenience), recall we define the projection matrix

$$\mathbf{P}_{\boldsymbol{w}} \overset{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}.$$

We want to make the primal and dual updates

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + h^{(k)}\mathbf{W}^{1/2}\boldsymbol{v}^{(k)} - h^{(k)}\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)},$$
$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \bar{t}h^{(k)}\mathbf{W}^{-1/2}\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}.$$

The first term for the primal update is straightforward to maintain, so we may ignore it without loss of generality. After this reduction, we see that the primal and dual updates are analogous. In the remainder of this section, we show how to maintain $\boldsymbol{x}$ undergoing the update

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + h^{(k)}\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}.$$

First, observe that $\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}}$ is an operator dependent on the dynamic weights $\boldsymbol{w}$, which motivates us to formalize this problem setting:

Definition 3.2 (Dynamic linear operator, update complexity)**.** Let $\boldsymbol{w}$ be a dynamic vector. We say $\mathbf{M}$ is a *dynamic linear operator dependent on* $\boldsymbol{w}$ if $\mathbf{M}$ is a function of $\boldsymbol{w}$. Let $\boldsymbol{w}^{(k)}$ be the value of $\boldsymbol{w}$ at step k, then we use $\mathbf{M}^{(k)}$ to denote the corresponding value of $\mathbf{M}$ at step k.

Suppose exists a data structure that dynamically maintains $\mathbf{M}$ and $\boldsymbol{w}$, such that at every step k, if $\boldsymbol{w}^{(k-1)}$ and $\boldsymbol{w}^{(k)}$ differ on K coordinates, then the data structure can update $\mathbf{M}^{(k-1)}$ to $\mathbf{M}^{(k)}$ in $f(K)$ time. Then we say $\mathbf{M}$ *has update complexity* f.

Next, we define two types of dynamic operators dependent on the weights $\boldsymbol{w}$ from the IPM: the *inverse tree operator* ∇ and the *tree operator* $\boldsymbol{\Delta}$. For linear programs with separable structures, they should crucially combine so that throughout algorithm, we have

$$\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}} = \boldsymbol{\Delta}\nabla. \tag{3.1}$$

3.2.1 Operators on a tree

In this section, we fix $\mathcal{T}$ to be a *constant-degree* rooted tree with root node G, called the *operator tree*. Let each node $H \in \mathcal{T}$ be associated with a set F_H, where the F_H's are pairwise disjoint. Let each *leaf* node $L \in \mathcal{T}$ be further associated with a *non-empty* set $E(L)$, where the $E(L)$'s are pairwise disjoint. For a non-leaf node H, define $E(H) \overset{\text{def}}{=} \bigcup_{\text{leaf } L \in \mathcal{T}_H} E(L)$. Finally, define $E \overset{\text{def}}{=} E(G) = \bigcup_{\text{leaf } L \in \mathcal{T}} E(L)$ and $V \overset{\text{def}}{=} \bigcup_{H \in \mathcal{T}} F_H$.

We define two special classes of linear operators that build on the structure of $\mathcal{T}$. The advantage of these operators lie in their decomposability, which allows them to be efficiently maintained.

Definition 3.3 (Inverse tree operator)**.** Let $\mathcal{T}$ be an operator tree with the associated sets as above. We say a linear operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$ is an *inverse tree operator supported on* $\mathcal{T}$ if there exists a linear *edge operator* ∇_H for each non-root node H in $\mathcal{T}$, corresponding to the edge from H to its parent, such that ∇ can be decomposed as

$$\nabla = \sum_{\text{leaf } L, \text{ node } H \,:\, L \in \mathcal{T}_H} \mathbf{I}_{F_H} \nabla_{H \leftarrow L},$$

where $\nabla_{H \leftarrow L}$ is defined as follows: If $L = H$, then $\nabla_{H \leftarrow L} \overset{\text{def}}{=} \mathbf{I}$; otherwise, suppose the path in $\mathcal{T}$ from leaf L to node H is given by $(H_t \overset{\text{def}}{=} L, H_{t-1}, \ldots, H_1, H_0 \overset{\text{def}}{=} H)$, then

$$\nabla_{H \leftarrow L} \overset{\text{def}}{=} \nabla_{H_1} \cdots \nabla_{H_{t-1}} \nabla_{H_t}.$$

To maintain ∇, it will suffice to maintain ∇_H at each non-root node H in $\mathcal{T}$.

Intuitively, when applying an inverse tree operator to a vector $\boldsymbol{v} \in \mathbb{R}^E$, $\boldsymbol{v}$ is partitioned according to the leaves of $\mathcal{T}$, and then the edge operators are applied sequentially along the tree edges in a bottom-up fashion. It is natural to then also define the opposite process, where edge operators are applied along the tree edges in a top-down fashion.

Definition 3.4 (Tree operator)**.** Let $\mathcal{T}$ be an operator tree with the associated sets as above. We say a linear operator $\nabla : \mathbb{R}^V \mapsto \mathbb{R}^E$ is *tree operator supported on* $\mathcal{T}$ if there exists a linear edge operator ∇_H for each non-root node H in $\mathcal{T}$, corresponding to the edge from H to its parent, such that ∇ can be decomposed as

$$\boldsymbol{\Delta} \overset{\text{def}}{=} \sum_{\text{leaf } L, \text{ node } H \,:\, L \in \mathcal{T}_H} \boldsymbol{\Delta}_{L \leftarrow H} \mathbf{I}_{F_H}.$$

where $\boldsymbol{\Delta}_{H \leftarrow L}$ is defined as follows: If $L = H$, then $\boldsymbol{\Delta}_{L \leftarrow H} \overset{\text{def}}{=} \mathbf{I}$. Otherwise, suppose the path in $\mathcal{T}$ from node H to leaf L is given by $(H_t \overset{\text{def}}{=} L, H_{t-1}, \ldots, H_0 \overset{\text{def}}{=} H)$, then

$$\boldsymbol{\Delta}_{L \leftarrow H} \overset{\text{def}}{=} \boldsymbol{\Delta}_{H_t} \cdots \boldsymbol{\Delta}_{H_2} \boldsymbol{\Delta}_{H_1}.$$

We define the complexity of a tree (and inverse tree) operator to be parameterized by the number of edge operators applied.

Definition 3.5 (Query complexity)**.** Let $\boldsymbol{\Delta} \overset{\text{def}}{=} \{\boldsymbol{\Delta}_H : H \in \mathcal{T}\}$ be a tree (or inverse tree) operator on tree $\mathcal{T}$. Suppose for any set $\mathcal{H}$ of K distinct non-root nodes in $\mathcal{T}$, and any two families of K vectors indexed by $\mathcal{H}$, $\{\boldsymbol{u}_H : H \in \mathcal{H}\}$ and $\{\boldsymbol{v}_H : H \in \mathcal{H}\}$, the total time to compute $\{\boldsymbol{u}_H^\top \boldsymbol{\Delta}_H : H \in \mathcal{H}\}$ and $\{\boldsymbol{\Delta}_H \boldsymbol{v}_H : H \in \mathcal{H}\}$ is bounded by $f(K)$. Then we say $\boldsymbol{\Delta}$ has query complexity f for some function f.

Without loss of generality, we may assume $f(0) = 0$, $f(k) \geq k$, and f is concave.

By examining the definition of the inverse tree and tree operator, we see they are related.

Lemma 3.6. *If $\mathbf{\Delta}$ is a tree operator on $\mathcal{T}$, then $\mathbf{\Delta}^\top$ is an inverse tree operator on $\mathcal{T}$, where its edge operators are obtained from $\mathbf{\Delta}$'s edge operators by taking a transpose. Furthermore, $\mathbf{\Delta}$ and ∇ have the same query and update complexity.* $\qquad\square$

3.3 Implicit representations of the solution

Assuming we have dynamic inverse tree and tree operators ∇ and $\mathbf{\Delta}$ on tree $\mathcal{T}$ dependent on $\boldsymbol{w}$ such that $\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}} = \mathbf{\Delta}\nabla$, we can now state how to abstractly maintain the implicit representation of the solutions throughout SOLVE (Algorithm 2). Specifically, we want to maintain the solution $\boldsymbol{x}$, and at every step k, carry out an update of the form

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + h^{(k)}\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}. \tag{3.2}$$

We design a data structure MAINTAINREP to accomplish this, by:

- At the start of SOLVE, initializing the data structure using the procedure INITIALIZE with $\boldsymbol{x} = \boldsymbol{x}^{(\mathrm{init})}$,

- At each step k, updating the weights $\boldsymbol{w}$ in the data structure using the procedure REWEIGHT, followed by updating $\boldsymbol{x}$ according to Eq. (3.2) using the procedure MOVE,

- At the end of SOLVE, outputing the final $\boldsymbol{x}$ using the procedure EXACT.

The key to designing an efficient data structure is to make use of the structure of the operators. Due to their decomposition along $\mathcal{T}$, we can update the operators and apply them to vectors without exploring all of $\mathcal{T}$ every time.

Theorem 3.7 (Implicit representation maintenance). *Let $\boldsymbol{w}$ be the weights changing at every step of SOLVE (Algorithm 2). Suppose there exists dynamic inverse tree and tree operators ∇ and $\mathbf{\Delta}$ on tree $\mathcal{T}$ both dependent on $\boldsymbol{w}$ such that $\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}} = \mathbf{\Delta}\nabla$ throughout the IPM. Let Q be the max of the query complexity of the tree and inverse tree operator, and let U be the max of the update complexity of the two operators. Suppose $\mathcal{T}$ has constant degree and height η. Then there is a data structure MAINTAINREP that satisfies the following invariants at the end of step k:*

- *It explicitly maintains the dynamic weights $\boldsymbol{w}$ and step direction $\boldsymbol{v}$ from the current step.*

- *It explicitly maintains scalar c and vectors $\boldsymbol{z}^{(\mathrm{step})}, \boldsymbol{z}^{(\mathrm{sum})}$, which together represent the implicitly-maintained vector $\boldsymbol{z} \overset{\mathrm{def}}{=} c\boldsymbol{z}^{(\mathrm{step})} + \boldsymbol{z}^{(\mathrm{sum})}$. At the end of step k, $\boldsymbol{z}^{(\mathrm{step})} = \nabla^{(k)}\boldsymbol{v}^{(k)}$, and*

$$\boldsymbol{z} = \sum_{i=1}^{k} h^{(i)}\nabla^{(i)}\boldsymbol{v}^{(i)}.$$

- *It implicitly maintains $\boldsymbol{x}$ so that at the end of step k,*

$$\boldsymbol{x} = \boldsymbol{x}^{(\mathrm{init})} + \sum_{i=1}^{k} h^{(i)}\mathbf{\Delta}^{(i)}\nabla^{(i)}\boldsymbol{v}^{(i)},$$

where $\boldsymbol{x}^{(\mathrm{init})}$ is some initial value set at the start of SOLVE.

The data structure supports the following procedures and runtimes:

- INITIALIZE($\mathbf{\Delta}, \nabla, \boldsymbol{v}^{(\text{init})} \in \mathbb{R}^m, \boldsymbol{w}^{(\text{init})} \in \mathbb{R}^m_{>0}, \boldsymbol{x}^{(\text{init})} \in \mathbb{R}^m$): *Preprocess and set* $\boldsymbol{x} \leftarrow \boldsymbol{x}^{(\text{init})}$.
  The procedure runs in $O(U(m) + Q(m))$ *time.*

- REWEIGHT($\delta_{\boldsymbol{w}} \in \mathbb{R}^m_{>0}$): *Update the weights to* $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta_{\boldsymbol{w}}$.
  The procedure runs in $O(U(\eta K) + Q(\eta K))$ *total time, where* $K = \text{nnz}(\delta_{\boldsymbol{w}})$.

- MOVE($h \in \mathbb{R}, \delta_{\boldsymbol{v}} \in \mathbb{R}^m$): *Update the current step direction to* $\boldsymbol{v} \leftarrow \boldsymbol{v} + \delta_{\boldsymbol{v}}$. *Update the implicit representation of* $\boldsymbol{x}$ *to reflect the following change in value:*

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + h\mathbf{\Delta}\nabla\boldsymbol{v}.$$

  The procedure runs in $O(Q(\eta K))$ *time, where* $K = \text{nnz}(\delta_{\boldsymbol{v}})$.

- EXACT: *Output the current exact value of* $\boldsymbol{x}$ *in* $O(Q(m))$ *time.*

3.4 Solution approximation

In the IPM, one key operation is to maintain the solution vector $\overline{\boldsymbol{x}}$ that is close to $\boldsymbol{x}$ throughout the algorithm. (Analogously for the slack $\overline{\boldsymbol{s}}$ close to $\boldsymbol{s}$.) Since we have implicit representations of the solution $\boldsymbol{x}$ from MAINTAINREP, we now show how to maintain $\overline{\boldsymbol{x}}$ close to $\boldsymbol{x}$. To accomplish this, we use a meta data structure that solves this in a more general setting introduced in [DGG+22].

Theorem 3.8 (Approximate vector maintenance with tree operator [DGG+22]). *Let* $0 < \rho < 1$ *be a failure probability. Suppose* $\mathbf{\Delta} \in \mathbb{R}^{m \times n}$ *is a tree operator with query complexity* Q *and supported on a constant-degree tree* $\mathcal{T}$ *with height* η. *There is a randomized data structure MAIN-TAINAPPROX that takes as input the dynamic weights* $\boldsymbol{w}$ *and the dynamic* $\boldsymbol{x}$ *implicitly maintained according to Theorem 3.7 at every step, and explicitly maintains the approximation* $\overline{\boldsymbol{x}}$ *to* $\boldsymbol{x}$ *satisfying* $\left\|\mathbf{W}^{-1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\right\|_\infty \leq \delta$ *at every step with probability* $1 - \rho$.

Suppose $\|\mathbf{W}^{(k)^{-1/2}}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)})\|_2 \leq \beta$ *for all steps* k. *Furthermore, suppose* $\boldsymbol{w}$ *is a function of* $\overline{\boldsymbol{x}}$ *coordinate-wise. Then, for each* $\ell \geq 0$, $\overline{\boldsymbol{x}}$ *admits* $2^{2\ell}$ *coordinate changes every* 2^ℓ *steps. Over* N *total steps, the total cost of the data structure is*

$$\widetilde{O}(\eta^3(\beta/\delta)^2 \log^3(mN/\rho)) \left(Q(m) + \sum_{k=1}^N Q(S^{(k)}) + \sum_{\ell=0}^{\log N} \frac{N}{2^\ell} \cdot Q(2^{2\ell}) \right), \tag{3.3}$$

where $S^{(k)}$ *is the number of nodes* H *where* $\mathbf{\Delta}_H$ *or* $\boldsymbol{u}_H$ *in the implicit representation of* $\boldsymbol{x}$ *changed at step* k.

3.5 Main theorem for the RIPM framework

We are now ready to state and prove the main result in this framework.

Theorem 3.9 (RIPM framework). *Consider an LP of the form*

$$\min_{\boldsymbol{x} \in \mathcal{P}} \boldsymbol{c}^\top \boldsymbol{x} \quad where \quad \mathcal{P} = \{\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \ \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\} \tag{3.4}$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$. *For any vector* $\boldsymbol{w}$, *let* $\mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$, *and suppose there exists dynamic tree and inverse tree operators* $\mathbf{\Delta}$ *and* ∇ *dependent on* $\boldsymbol{w}$, *such that* $\mathbf{W}^{1/2}\mathbf{P}_{\boldsymbol{w}} = \mathbf{\Delta}\nabla$. *Let* U *be the update complexity of* $\mathbf{\Delta}$ *and* ∇, *and let* Q *be their query complexity. Let* r *and* $R = \|\boldsymbol{u} - \boldsymbol{l}\|_2$ *be*

the inner and outer radius of $\mathcal{P}$, and let $L = \|\boldsymbol{c}\|_2$. Then, there is a data structure to solve Eq. (3.4) to εLR accuracy with probability $1 - 2^{-m}$ in time

$$\widetilde{O}\left(\eta^4 \sqrt{m} \log(\frac{R}{\varepsilon r}) \cdot \sum_{\ell=0}^{\frac{1}{2}\log m} \frac{U(2^{2\ell}) + Q(2^{2\ell})}{2^\ell}\right).$$

Proof of Theorem 3.9. We implement the IPM algorithm using the data structures from Sections 3.3 and 3.4, and bound the cost of each operations of the data structures. For simplicity, we only discuss the primal variables in this proof, but the slack variables are analogous. We use one copy of MAINTAINREP to maintain $\boldsymbol{x}$, and one copy of MAINTAINAPPROX to maintain $\overline{\boldsymbol{x}}$. At each step, we perform the implicit update of $\boldsymbol{x}$ using MOVE and update $\boldsymbol{w}$ using REWEIGHT in MAINTAINREP. We construct the explicit approximations $\overline{\boldsymbol{x}}$ using APPROXIMATE in MAINTAINAPPROX.

Theorem 3.8 shows that throughout the IPM, for each $\ell \geq 0$, there are $2^{2\ell}$ coordinate changes to $\overline{\boldsymbol{x}}$ every 2^ℓ steps. Since $\boldsymbol{w}$ is a function of $\overline{\boldsymbol{x}}$ coordinate-wise, there are also $2^{2\ell}$ coordinate changes in $\boldsymbol{w}$ every 2^ℓ steps. Similarly, we observe that $\boldsymbol{v}$ is defined as a function of $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{s}}$ coordinate-wise, so there are $O(2^{2\ell})$ coordinate changes to $\boldsymbol{v}$ every 2^ℓ steps. Then Theorem 3.7 shows that the total runtime over N steps for the MAINTAINREP data structure is

$$\widetilde{O}(U(m) + Q(m)) + \widetilde{O}\left(\sum_{\ell=0}^{\log N} \frac{N}{2^\ell} \cdot \left(U(\eta \cdot 2^{2\ell}) + Q(\eta \cdot 2^{2\ell})\right)\right). \tag{3.5}$$

Theorem 3.8 shows that the total runtime over N steps for MAINTAINAPPROX is

$$\widetilde{O}(\eta^3 (\beta/\delta)^2 \log^3(mN/\rho))\left(Q(m) + \sum_{k=1}^{N} Q(S^{(k)}) + \sum_{\ell=0}^{\log N} \frac{N}{2^\ell} \cdot Q(2^{2\ell})\right), \tag{3.6}$$

where the variables are defined as in the theorem statement. By examining Theorem 3.7, we see that when a coordinate of $\boldsymbol{w}$ or $\boldsymbol{v}$ changes, the implicit representation of $\boldsymbol{x}$ admits updates at $O(\eta)$-many nodes. Combined with the concavity of Q, we can bound

$$\sum_{k=1}^{N} Q(S^{(k)}) \leq O(\eta) \cdot \sum_{\ell=0}^{\log N} \frac{N}{2^\ell} \cdot Q(2^{2\ell}).$$

Theorem 3.1 guarantees that there are $N = \sqrt{m} \log m \log(\frac{mR}{\varepsilon r})$ total IPM steps, and at each step k, we have $h^{(k)} \left\|\mathbf{W}^{(k)-1/2}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)})\right\|_2 = h^{(k)} \left\|\boldsymbol{v}^{(k)} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}^{(k)}\right\|_2 \leq O(\frac{1}{\log m})$, so we can set $\beta = O(\frac{1}{\log m})$. By examining Algorithm 2, we see it suffices to set $\delta = O(\frac{1}{\log m})$. We choose the failure probability ρ to be appropriately small, e.g. 2^{-m}. Finally, we conclude that the overall runtime of the IPM framework is

$$\widetilde{O}\left(\eta^4 \sqrt{m} \log(\frac{R}{\varepsilon r}) \cdot \sum_{\ell=0}^{\frac{1}{2}\log m} \frac{U(2^{2\ell}) + Q(2^{2\ell})}{2^\ell}\right),$$

where the terms for intialization times have been absorbed. $\qquad\square$

4 From separator tree to projection operators

In this section, we explore the separable structure of the dual graph $G_{\mathbf{A}}$ of the LP constraint matrix $\mathbf{A}$, and use these properties to help define and maintain the tree operator and inverse tree operator as needed for the IPM framework from Section 3.

Throughout this section, we fix $\mathbf{A} \in \mathbb{R}^{n \times m}$, so that the dual graph $G_{\mathbf{A}} = (V, E)$ has n vertices, m hyperedges. Additionally, let ρ denote the max hyperedge size in $G_{\mathbf{A}}$; equivalently, ρ is the column sparsity of $\mathbf{A}$.

4.1 Separator tree

The notion of using a *separator tree* to represent the recursive decomposition of a separable graph is well-established in literature, c.f [EGIS96, HKRS97]. In our work, we use the following definition:

Definition 4.1 (Separator tree). Let G be a hypergraph with n vertices, m hyperedges, and max hyperedge size ρ. A *separator tree* $\mathcal{S}$ for G is a *constant-degree* tree whose nodes represent a recursive decomposition of G based on balanced separators.

Formally, each node of $\mathcal{S}$ is a *region* (edge-induced subgraph) H of G; we denote this by $H \in \mathcal{S}$. At a node H, we define subsets of vertices $\partial H, S(H), F_H$, where ∂H is the set of *boundary vertices* of H, i.e. vertices with neighbours outside H in G; $S(H)$ is a balanced vertex separator of H; and F_H is the set of *eliminated vertices* at H. Furthermore, let $E(H)$ denote the edges contained in H.

The nodes and associated vertex sets are defined in a top-down manner as follows:

1. The root of $\mathcal{S}$ is the node $H = G$, with $\partial H = \emptyset$ and $F_H = S(H)$.

2. A non-leaf node $H \in \mathcal{S}$ has a constant number of children whose union is H. The children form a edge-disjoint partition of H, and the intersection of their vertex sets is a balanced separator $S(H)$ of H. Define the set of eliminated vertices at H to be $F_H \overset{\text{def}}{=} S(H) \setminus \partial H$.

   The set $F_H \cup \partial H$ consists of all vertices in the boundary and separator, which can intuitively be interpreted as the *skeleton* of H. In later sections, we recursively construct graphs (matrices) on $F_H \cup \partial H$ which capture compressed information about all of H.

   By definition of boundary vertices, for a child D of H, we have $\partial D \overset{\text{def}}{=} (\partial H \cup S(H)) \cap V(D)$.

3. At a leaf node H, we define $S(H) = \emptyset$ and $F_H = V(H) \setminus \partial H$. (This convention allows leaf nodes to exist at different levels in $\mathcal{S}$.) The leaf nodes of $\mathcal{S}$ partition the edges of G.

We use η to denote the height of $\mathcal{S}$.

For a separator tree to be meaningful, the leaf node regions should be sufficiently small, to indicate that we have a good overall decomposition of the graph. Additionally, for our work, we want a more careful bound on the sizes of the skeleton of regions. This motivates the following refined definition:

Definition 4.2 $((a, b, \lambda)$-separator tree). Let G be a graph with n vertices, m edges, and max hyperedge size ρ. Let $a \in [0, 1]$ and $b \in (0, 1)$ be constants, and $\lambda \geq 1$ be an expression in terms of m, n, ρ. An (a, b, λ)-separator tree $\mathcal{S}$ for G is a separator tree satisfying the following additional properties:

1. There are at most $O(b^{-i})$ nodes at level i in $\mathcal{S}$,

Figure 4.1: An example of a separator tree. The bold edges denote the boundary of each component, ∂H while the dotted lines denote the separators $S(H)$. Note that $F_H = S(H) \setminus \partial H$ is defined differently on the leaves.

2. any node H at level i satisfies $|F_H \cup \partial H| \leq O(\lambda \cdot b^{ai})$,

3. a node H at level i is a leaf node if and only if $|V(H)| \leq O(\rho)$.

Intuitively, a and b come from the separability parameters of G, and λ is a scaling factor for node sizes in $\mathcal{S}$. Since there could be hyperedges of size ρ, regions of size ρ are not necessarily separable, so we set the region as a leaf.

We make extensive use of these properties in subsequent sections when computing runtimes.

4.2 Nested dissection using a separator tree

Let $\mathcal{S}$ be any separator tree for $G_{\mathbf{A}}$. In this section, we show how to use $\mathcal{S}$ to factor the matrix $\mathbf{L}^{-1} \stackrel{\text{def}}{=} (\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}$ recursively:

Definition 4.3 (Block Cholesky decomposition)**.** The *block Cholesky decomposition* of a symmetric matrix $\mathbf{L}$ with blocks indexed by F and C is:

$$\mathbf{L} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{L}_{C,F}(\mathbf{L}_{F,F})^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{F,F} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L},C) \end{bmatrix} \begin{bmatrix} \mathbf{I} & (\mathbf{L}_{F,F})^{-1}\mathbf{L}_{F,C} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}, \tag{4.1}$$

where the middle matrix in the decomposition is a block-diagonal matrix with blocks indexed by F and C, with the lower-right block being the *Schur complement* $\mathbf{Sc}(\mathbf{L},C)$ of $\mathbf{L}$ onto C:

$$\mathbf{Sc}(\mathbf{L},C) \stackrel{\text{def}}{=} \mathbf{L}_{C,C} - \mathbf{L}_{C,F}\mathbf{L}_{F,F}^{-1}\mathbf{L}_{F,C}. \tag{4.2}$$

Since $\mathbf{Sc}(\mathbf{L},C)$ is a symmetric matrix, we can recursively apply the decomposition Eq. (4.1) to it. By choosing the index sets F, C for each recursive step according to $\mathcal{S}$, we get a recursive decomposition of $\mathbf{L}^{-1}$:

Theorem 4.4 ($\mathbf{L}^{-1}$ factorization, c.f. [DGG$^+$22] Theorem 33). *Let $\mathcal{S}$ be the separator tree of $G_{\mathbf{A}}$ with height η. For each node $H \in \mathcal{S}$ with hyperedges $E(H)$, let $\mathbf{A}_H \in \mathbb{R}^{n \times m}$ denote the matrix $\mathbf{A}$ restricted to columns indexed by $E(H)$. Define*

$$\mathbf{L}[H] \overset{\text{def}}{=} \mathbf{A}_H \mathbf{W} \mathbf{A}_H^\top, \text{ and} \tag{4.3}$$

$$\mathbf{L}^{(H)} \overset{\text{def}}{=} \mathbf{Sc}(\mathbf{L}[H], F_H \cup \partial H). \tag{4.4}$$

Then, we have

$$\mathbf{L}^{-1} = \mathbf{\Pi}^{(\eta)\top} \cdots \mathbf{\Pi}^{(1)\top} \mathbf{\Gamma} \mathbf{\Pi}^{(1)} \cdots \mathbf{\Pi}^{(\eta)}, \tag{4.5}$$

where[4]

$$\mathbf{\Gamma} \overset{\text{def}}{=} \begin{bmatrix} \left(\sum_{H \in \mathcal{T}(\eta)} \left(\mathbf{L}^{(H)}_{F_H, F_H} \right)^{-1} \right) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \left(\sum_{H \in \mathcal{T}(0)} \left(\mathbf{L}^{(H)}_{F_H, F_H} \right)^{-1} \right) \end{bmatrix}, \tag{4.6}$$

and for $i = 1, \dots, \eta$,

$$\mathbf{\Pi}^{(i)} \overset{\text{def}}{=} \mathbf{I} - \sum_{H \in \mathcal{T}(i)} \mathbf{X}^{(H)}, \tag{4.7}$$

where $\mathcal{T}(i)$ is the set of nodes at level i in $\mathcal{T}$, the matrix $\mathbf{\Pi}^{(i)}$ is supported on $\bigcup_{H \in \mathcal{T}(i)} F_H \cup \partial H$ and padded with zeros to n-dimensions, and for each $H \in \mathcal{S}$,

$$\mathbf{X}^{(H)} \overset{\text{def}}{=} \mathbf{L}^{(H)}_{\partial H, F_H} \left(\mathbf{L}^{(H)}_{F_H, F_H} \right)^{-1}. \tag{4.8}$$

4.3 Projection operators definition

Suppose $\mathcal{S}$ is a separator tree for $G_{\mathbf{A}}$. In this subsection, we define the operator tree $\mathcal{T}$ based on $\mathcal{S}$, followed by the tree operator $\mathbf{\Delta}$ and inverse tree operator ∇ which will be supported on $\mathcal{T}$. Finally, we will show that our definitions indeed satisfy

$$\mathbf{W}^{1/2} \mathbf{P}_{\boldsymbol{w}} = \mathbf{\Delta} \nabla.$$

Recall that $\mathcal{S}$ is a constant-degree tree. The leaf nodes of $\mathcal{S}$ partition the hyperedges of $G_{\mathbf{A}}$, however, we do not have a bound on the number of hyperedges in a leaf node. In constructing $\mathcal{T}$, we simply want to modify $\mathcal{S}$ so that each leaf contains exactly one hyperedge. Specifically, for each leaf node $H \in \mathcal{S}$ containing $|E(H)|$ hyperedges, we construct a complete binary tree $\mathcal{T}_H^+$ rooted at H with $|E(H)|$ leaves, assign one hyperedge from $E(H)$ to one new leaf, and attach $\mathcal{T}_H^+$ at the node H. This construction yields the desired operator tree $\mathcal{T}$ whose height is within a $\log |E|$ factor of $\mathcal{S}$.

We define the tree operator $\mathbf{\Delta}$ on $\mathcal{T}$ follows: For non-root node H in $\mathcal{T}$, let

$$\mathbf{\Delta}_H \overset{\text{def}}{=} \begin{cases} \mathbf{I}_{F_H \cup \partial H} - \mathbf{X}^{(H)\top} & \text{if } H \text{ exists in } \mathcal{S} \\ \mathbf{W}_{E(H)}^{1/2} \mathbf{A}_H^\top & \text{if } H \text{ is a leaf node in } \mathcal{T} \\ \mathbf{I} & \text{else.} \end{cases} \tag{4.9}$$

Note that the first two cases are indeed disjoint by construction. We pad zeros to all matrices in order to arrive at the correct overall dimensions.

---

[4]We use a different definition of *level* compared to [DGG$^+$22]. In [DGG$^+$22], the root has level η in and leaf nodes have level 0, and in this paper, the root has level 0 and leaf nodes have level η. This is purely for notational convenience in later calculations, so this theorem is otherwise unaffected.

Lemma 4.5 (c.f. [DGG$^+$22], Lemma 59)**.** *Let $\boldsymbol{\Delta}$ be the tree operator as defined above. Then*

$$\boldsymbol{\Delta} = \mathbf{W}^{1/2} \mathbf{A}^{\top} \boldsymbol{\Pi}^{(\eta)\top} \cdots \boldsymbol{\Pi}^{(1)\top}. \tag{4.10}$$

$\square$

Next, we establish the query complexity of the tree operator:

Lemma 4.6. *Suppose L is the total number of leaf nodes in $\mathcal{S}$. The query complexity of $\boldsymbol{\Delta}$ is*

$$Q(K) = O\left(\rho K + \max_{\mathcal{H}:set\ of\ K\ leaves\ in\ \mathcal{S}} \sum_{H \in \mathcal{P}_{\mathcal{S}}(\mathcal{H})} |F_H \cup \partial H|^2\right)$$

for $K \leq L$, where $\mathcal{P}_{\mathcal{S}}(\mathcal{H})$ is the set of all nodes in $\mathcal{S}$ that are ancestors of some node in $\mathcal{H}$ unioned with $\mathcal{H}$. When $K > L$, then we define $Q(K) = Q(L)$.

Proof. First, we consider the query time $Q(1)$ for a single edge. Let $\boldsymbol{u}$ be any vector, and let H be a non-root node in $\mathcal{T}$. If H is a leaf node, then computing $\boldsymbol{\Delta}_H \boldsymbol{u}$ and $\boldsymbol{u}^{\top} \boldsymbol{\Delta}_H$ both take $O(\rho)$ time. If H exists in $\mathcal{S}$, then computing $\boldsymbol{\Delta}_H \boldsymbol{u}$ takes $O\left(|F_H|^2 + |\partial H||F_H|\right) \leq O\left(|F_H \cup \partial H|^2\right)$ time, since the bottleneck is naively computing $\mathbf{L}^{(H)}_{\partial H, F_H}\left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1} \boldsymbol{u}$. Therefore, $Q(1) = O\left(\rho + \max_{H \in \mathcal{S}} |F_H \cup \partial H|^2\right)$.

For $K > 1$, we can simply bound the query time for K distinct edges by

$$Q(K) = O\left(\rho K + \max_{\mathcal{H}:set\ of\ K\ nodes\ in\ \mathcal{S}} \sum_{H \in \mathcal{H}} |F_H \cup \partial H|^2\right).$$

Finally, note that we can take the summation over $H \in \mathcal{P}_{\mathcal{S}}(\mathcal{H})$ instead of $H \in \mathcal{H}$ for an upper bound. In this case, it suffices to take the max over sets of leaf nodes. $\square$

By taking the transpose of $\boldsymbol{\Delta}$, we get an inverse tree operator, and together, they give the projection matrix using Eq. (4.5).

Corollary 4.7. *Let $\nabla \stackrel{\text{def}}{=} \boldsymbol{\Delta}^{\top}$ be the inverse tree operator obtained from $\boldsymbol{\Delta}$ by transposing the edge and leaf operators. Then*

$$\mathbf{W}^{1/2} \mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2} \mathbf{W}^{1/2} \mathbf{A}^{\top} \mathbf{L}^{-1} \mathbf{A} \mathbf{W}^{1/2} = (\mathbf{W}^{1/2} \boldsymbol{\Delta}) \boldsymbol{\Gamma} \nabla. \tag{4.11}$$

$\square$

Remark 4.8. Without loss of generality, we have chosen to simplify our presentation and consider $\boldsymbol{\Delta} \nabla$ in place of $\mathbf{W}^{1/2} \boldsymbol{\Delta} \boldsymbol{\Gamma} \nabla$.

This is possible for two reasons: One, $\mathbf{W}^{1/2} \boldsymbol{\Delta}$ is a tree operator, which we can in fact maintain in the same time complexity as $\boldsymbol{\Delta}$. Two, $\boldsymbol{\Gamma}$ is a block-diagonal matrix, with a block for each $H \in \mathcal{T}$ that is indexed by F_H. It is straightforward to show we can maintain and apply $\boldsymbol{\Gamma} \nabla$ in the same time complexity as ∇.

4.4 Maintenance of projection operators

So far, we have defined the separator tree $\mathcal{S}$ for the graph $G_{\mathbf{A}}$, which we then used to define the operator tree $\mathcal{T}$, which supports the tree operator $\boldsymbol{\Delta}$ needed for the IPM framework. In this subsection, we discuss how to maintain $\mathbf{L}^{(H)}, \mathbf{Sc}(\mathbf{L}^{(H)}, \partial H)$, and $(\mathbf{L}_{F_H, F_H}^{(H)})^{-1}$ at each node $H \in \mathcal{S}$ using the data structure DYNAMICSC (Algorithm 1), as the weight vector $\boldsymbol{w}$ undergoes changes throughout the IPM. This will in turn allow us to maintain the tree operator $\boldsymbol{\Delta}$.

We begin with a lemma showing that given a symmetric matrix and a low-rank update, we can compute its new inverse and Schur complement quickly.

Lemma 4.9. *Let $\mathbf{L}' = \mathbf{L} + \mathbf{U}\mathbf{V} \in \mathbb{R}^{n \times n}$ be a symmetric matrix plus a rank-K update, where $\mathbf{U}$ and $\mathbf{V}^{\top}$ both have dimensions $n \times K$. Given $\mathbf{L}', \mathbf{U}, \mathbf{V}$, we can compute $\mathbf{L}'^{-1}$ in $O(n^2 K^{\omega-2})$ time.*

Additionally, suppose we are also given $\mathbf{L}^{-1}$ and $\mathbf{Sc}(\mathbf{L}, S)$ for an index set S. Then we can compute $\mathbf{Sc}(\mathbf{L}', S)$, $\mathbf{U}', \mathbf{V}'$ in $O(n^2 K^{\omega-2})$ time, so that $\mathbf{Sc}(\mathbf{L}, S) + \mathbf{U}'\mathbf{V}' = \mathbf{Sc}(\mathbf{L}', S)$, and $\mathbf{U}', \mathbf{V}'^{\top}$ both have K columns.

Proof. The Sherman-Morrison formula states

$$\mathbf{L}'^{-1} = \mathbf{L}^{-1} - \mathbf{L}^{-1}\mathbf{U}(\mathbf{I}_K + \mathbf{V}\mathbf{L}^{-1}\mathbf{U})^{-1}\mathbf{V}\mathbf{L}^{-1}.$$

The time to compute this update is dominated by the time required to multiply an $n \times n$ matrix with an $n \times K$ matrix, which is $O(n^2 K^{\omega-2})$.

For the second part of the lemma, recall that the Schur complement is defined to be:

$$\mathbf{Sc}(\mathbf{L}, C) \overset{\text{def}}{=} \mathbf{L}_{C,C} - \mathbf{L}_{C,F}\mathbf{L}_{F,F}^{-1}\mathbf{L}_{F,C}. \tag{4.12}$$

If we were to naively use this definition of the Schur complement to perform the updates and construct $\mathbf{U}'$ and $\mathbf{V}'^{\top}$, we will run into an issue where the rank of the new update blows up by a factor of 8, leading to an exponential blowup in the rank as we go up the levels recursively. Instead, we make use of the fact that the inverse of the Schur complement, $\mathbf{Sc}(\mathbf{L}, S)^{-1}$ is exactly the S, S submatrix of $\mathbf{L}^{-1}$ to control the rank of the updates.

We first apply the definition of Schur complement and then use the Sherman-Morrison formula to get

$$\begin{aligned}
\mathbf{Sc}(\mathbf{L}', S)^{-1} &= \mathbf{L}'^{-1}{}_{S,S} \\
&= \mathbf{L}^{-1}{}_{S,S} - \left(\mathbf{L}^{-1}\mathbf{U}(\mathbf{I}_K + \mathbf{V}\mathbf{L}^{-1}\mathbf{U})^{-1}\mathbf{V}\mathbf{L}^{-1}\right)_{S,S} \\
&= \mathbf{Sc}(\mathbf{L}, S)^{-1} - \mathbf{I}_S\mathbf{L}^{-1}\mathbf{U}(\mathbf{I}_K + \mathbf{V}\mathbf{L}^{-1}\mathbf{U})^{-1}\mathbf{V}\mathbf{L}^{-1}\mathbf{I}_S.
\end{aligned}$$

This gives us the new rank-K update $\mathbf{Sc}(\mathbf{L}', S)^{-1} = \mathbf{Sc}(\mathbf{L}, S)^{-1} + \mathbf{U}^*\mathbf{V}^*$ with

$$\begin{aligned}
\mathbf{U}^* &= -\mathbf{I}_S\mathbf{L}^{-1}\mathbf{U} \\
\mathbf{V}^* &= (\mathbf{I}_K + \mathbf{V}\mathbf{L}^{-1}\mathbf{U})^{-1}\mathbf{V}\mathbf{L}^{-1}\mathbf{I}_S.
\end{aligned}$$

We can now determine the Schur complement update by applying Sherman-Morrison again:

$$\mathbf{Sc}(\mathbf{L}', S) = \mathbf{Sc}(\mathbf{L}, S) - \mathbf{Sc}(\mathbf{L}, S)\mathbf{U}^*(\mathbf{I}_K + \mathbf{V}^*\mathbf{Sc}(\mathbf{L}, S)\mathbf{U}^*)^{-1}\mathbf{V}^*\mathbf{Sc}(\mathbf{L}, S).$$

This is a rank-K update $\mathbf{Sc}(\mathbf{L}', S) = \mathbf{Sc}(\mathbf{L}, S) + \mathbf{U}'\mathbf{V}'$ with

$$\begin{aligned}
\mathbf{U}' &= -\mathbf{Sc}(\mathbf{L}, S)\mathbf{U}^* \\
\mathbf{V}' &= (\mathbf{I}_K + \mathbf{V}^*\mathbf{Sc}(\mathbf{L}, S)\mathbf{U}^*)^{-1}\mathbf{V}^*\mathbf{Sc}(\mathbf{L}, S).
\end{aligned}$$

The time to compute $\mathbf{U}^*, \mathbf{V}^*, \mathbf{U}', \mathbf{V}'$ are all dominated by the time to multiply an $n \times n$ matrix with an $n \times K$ matrix, which is $O(n^2 K^{\omega-2})$. □

Now, we are ready to present the data structure for maintaining the Schur complement matrices along a separator tree.

Lemma 4.10. *Let $\boldsymbol{w}$ be the weights changing at every step of the IPM. Let $\mathcal{S}$ be any separator tree for $G_{\mathbf{A}}$. Recall $G_{\mathbf{A}}$ has n vertices, m hyperedges, and max hyperedge size ρ. Then the data structure DYNAMICSC (Algorithm 1) correctly maintains the matrices $\mathbf{L}^{(H)}, (\mathbf{L}_{F_H,F_H}^{(H)})^{-1}, \mathbf{Sc}(\mathbf{L}^{(H)}, \partial H)$ at every node $H \in \mathcal{S}$ dependent on $\boldsymbol{w}$ throughout the IPM. The data structure supports the following procedures and runtimes:*

- *INITIALIZE($\mathcal{S}, \boldsymbol{w}^{(\mathrm{init})} \in \mathbb{R}^m$): Set $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\mathrm{init})}$, and compute all matrices with respect to $\boldsymbol{w}$, in time*

$$O\left(\sum_{\text{leaf } H \in \mathcal{S}} |E(H)| \cdot |F_H \cup \partial H|^{\omega - 1} + \sum_{H \in \mathcal{S}} |F_H \cup \partial H|^{\omega} \right).$$

- *REWEIGHT($\delta_{\boldsymbol{w}} \in \mathbb{R}^m$): Update the weight vector to $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta_{\boldsymbol{w}}$, and update all the maintained matrices with respect to the new weights, in time*

$$O\left(\sum_{\text{leaf } H \in \mathcal{H}} \mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) \cdot |F_H \cup \partial H|^{\omega - 1} + \sum_{H \in \mathcal{H}} |F_H \cup \partial H|^2 \cdot K_H^{\omega - 2} \right).$$

where $\mathcal{H}$ is the set of nodes H with $\delta_{\boldsymbol{w}}|_{E(H)} \neq \boldsymbol{0}$, $K_H \overset{\mathrm{def}}{=} \min\{\mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}), |F_H \cup \partial H|\}$.

Proof. INITIALIZE is a special case of REWEIGHT, where the change in the weight vector is from $\boldsymbol{0}$ to $\boldsymbol{w}^{(\mathrm{init})}$, so we focus on a single call of REWEIGHT.

It suffices for REWEIGHT visits only nodes in $\mathcal{H}$, since if none of the edges in a region admits a weight update, then the matrices stored at the node remain the same by definition. Also note that $H \in \mathcal{H}$ implies all ancestors of H are also in $\mathcal{H}$.

Correctness. We use the superscript $^{(\mathrm{new})}$ on $\mathbf{L}^{(H)}$ to indicate that it is computed with respect to the new weights, and $^{(\mathrm{old})}$ otherwise. Recall that $\mathbf{L}^{(H)}$ is supported on $F_H \cup \partial H$.

We maintain some additional matrices at each node, in order to efficiently compute low-rank updates. Specifically, we use helper matrices $\mathbf{U}_H, \mathbf{V}_H$ at H, and guarantee that during a single REWEIGHT($\delta_{\boldsymbol{w}}$) call, after SCHURNODE($H, \delta_{\boldsymbol{w}}$) is run, they satisfy $\mathbf{Sc}(\mathbf{L}^{(H)^{(\mathrm{old})}}, \partial H) = \mathbf{Sc}(\mathbf{L}^{(H)^{(\mathrm{new})}}, \partial H) + \mathbf{U}_H \mathbf{V}_H$, and $\mathbf{U}_H, \mathbf{V}_H^{\top}$ both have at most K_H-many columns.

Now, we show inductively that after SCHURNODE($H, \delta_{\boldsymbol{w}}$) is run, all matrices at H, as well as all matrices at all descendants of H, are updated correctly: When H is leaf node, recall $\mathbf{L}^{(H)}$ is defined to be $\mathbf{L}[H] \overset{\mathrm{def}}{=} \mathbf{A}_H \mathbf{W}_{E(H)} \mathbf{A}_H^{\top}$, so clearly SCHURNODE updates $\mathbf{L}^{(H)}$ correctly, and the rank of the update is at most K_H. The remaining matrices at H are computed correctly by Lemma 4.9.

Inductively, when H is a non-leaf node, the recursive property of Schur complements (c.f. [DGG+22, Lemma 18]) allows us to write $\mathbf{L}^{(H)^{(\mathrm{new})}} = \sum_{\text{child } D \text{ of } H} \mathbf{Sc}(\mathbf{L}^{(D)^{(\mathrm{new})}}, \partial D)$ at every node $H \in \mathcal{S}$. This formula trivially shows that the update $\mathbf{L}^{(H)^{(\mathrm{new})}} - \mathbf{L}^{(H)^{(\mathrm{old})}}$ has rank $|F_H \cup \partial H|$ (ie. full-rank). Alternatively, if $\mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) \leq |F_H \cup \partial H|$, then by the guarantees on the helper

18

Algorithm 1 Data structure to maintain dynamic Schur complements

1: **data structure** DYNAMICSC
2: **private: member**
3:     Hypergraph $G_\mathbf{A}$ with incidence matrix $\mathbf{A}$
4:     $\boldsymbol{w} \in \mathbb{R}^m$: Dynamic weight vector
5:     $\mathcal{S}$: Separator tree of height η. Every node H of $\mathcal{S}$ stores:
6:         $F_H, \partial H$: Sets of eliminated vertices and boundary vertices of region H
7:         $E(H)$: Set of hyperedges of region H
8:         $\mathbf{L}^{(H)}, (\mathbf{L}^{(H)}_{F_H, F_H})^{-1}, \mathbf{Sc}(\mathbf{L}^{(H)}, \partial H),$: Matrices to maintain as a function of $\boldsymbol{w}$
9:         $\mathbf{L}^{(H)^{-1}}$: Additional inverse matrix to maintain as a function of $\boldsymbol{w}$
10:         $\mathbf{U}_H, \mathbf{V}_H$: Low-rank update at H, used in REWEIGHT
11:
12: **procedure** INITIALIZE($\mathcal{S}, \boldsymbol{w}^{(\text{init})} \in \mathbb{R}^m$)
13:     $\mathcal{S} \leftarrow \mathcal{S}, \boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{init})}$
14:     **for** level $i = \eta$ to 0 **do**
15:         **for** each node H at level i **do**
16:             $\mathbf{L}^{(H)}, (\mathbf{L}^{(H)}_{F_H, F_H})^{-1}, \mathbf{Sc}(\mathbf{L}^{(H)}, \partial H) \leftarrow \mathbf{0}, \mathbf{0}, \mathbf{0}$
17:             SCHURNODE($H, \boldsymbol{w}$)
18:         **end for**
19:     **end for**
20: **end procedure**
21:
22: **procedure** REWEIGHT($\delta_{\boldsymbol{w}} \in \mathbb{R}^m$)
23:     $\mathcal{H} \leftarrow$ set of nodes H in $\mathcal{S}$ where $\delta_{\boldsymbol{w}}|_{E(H)} \neq \mathbf{0}$
24:     **for** level $i = \eta$ to 0 **do**
25:         **for** each node $H \in \mathcal{H}$ at level i **do**
26:             SCHURNODE($H, \delta_{\boldsymbol{w}}$)
27:         **end for**
28:     **end for**
29:     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta_{\boldsymbol{w}}$
30: **end procedure**
31:
32: **procedure** SCHURNODE($H \in \mathcal{S}, \delta_{\boldsymbol{w}} \in \mathbb{R}^m$)
33:     **if** H is a leaf node **then**   ▷ rank of update $\leq \min\{\text{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}), |F_H \cup \partial H|\}$
34:         $\mathbf{L}^{(H)} \leftarrow \mathbf{L}^{(H)} + \mathbf{A}_H \text{diag}(\delta_{\boldsymbol{w}}|_{E(H)})\mathbf{A}_H^\top$
35:     **else if** $\text{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) \leq |F_H \cup \partial H|$ **then**   ▷ rank of update $\leq \sum_{\text{child } D} K_D \leq \text{nnz}(\delta_{\boldsymbol{w}}|_{E(H)})$
36:         $\mathbf{L}^{(H)} \leftarrow \mathbf{L}^{(H)} + \sum_{\text{child } D \text{ of } H} \mathbf{U}_D \mathbf{V}_D$
37:     **else**   ▷ rank of update $\leq |F_H \cup \partial H|$
38:         $\mathbf{L}^{(H)} \leftarrow \sum_{\text{child } D \text{ of } H} \mathbf{Sc}(\mathbf{L}^{(D)}, \partial D)$
39:     **end if**
40:     Let $K_H \stackrel{\text{def}}{=} \min\{\text{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}), |F_H \cup \partial H|\}$ ▷ upper bound on the rank of update to $\mathbf{L}^{(H)}$
41:     Compute $(\mathbf{L}^{(H)}_{F_H, F_H})^{-1}$ and $\mathbf{L}^{(H)^{-1}}$ by Lemma 4.9
42:     Compute $\mathbf{Sc}(\mathbf{L}^{(H)}, \partial H)$ and its rank-K_H update factorization $\mathbf{U}_H, \mathbf{V}_H$ by Lemma 4.9
43: **end procedure**

matrices, we have

$$\mathbf{L}^{(H)^{(\mathrm{new})}} = \sum_{\mathrm{child}\ D\ \mathrm{of}\ H} \mathbf{Sc}(\mathbf{L}^{(D)^{(\mathrm{new})}}, \partial D)$$

$$= \sum_{\mathrm{child}\ D\ \mathrm{of}\ H} \mathbf{Sc}(\mathbf{L}^{(D)^{(\mathrm{old})}}, \partial D) + \mathbf{U}_D \mathbf{V}_D$$

$$= \mathbf{L}^{(H)^{(\mathrm{old})}} + \sum_{\mathrm{child}\ D\ \mathrm{of}\ H} \mathbf{U}_D \mathbf{V}_D.$$

This gives a low-rank factorization of the update $\mathbf{L}^{(H)^{(\mathrm{new})}} - \mathbf{L}^{(H)^{(\mathrm{old})}}$ with rank at most $\sum_{\mathrm{child}\ D} K_D$, which we can show by induction is at most $\mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)})$. Since we have the correct low-rank update to $\mathbf{L}^{(H)}$, the remaining matrices at H again are computed correctly by Lemma 4.9.

This completes the correctness proof.

Runtime. Consider the runtime of the procedure SCHURNODE$(H, \delta_{\boldsymbol{w}})$ at a node H: If H is a leaf node, then computing the update to $\mathbf{L}^{(H)}$ involves multiplying a $|F_H \cup \partial H| \times \mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)})$-sized matrix with its transpose (Line 33). Note that if $|F_H \cup \partial H| > \mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)})$, then this runtime can be absorbed into the runtime expression for the remaining steps of the procedure, since $K_H = \mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)})$. Otherwise, we use fast matrix multiplication which takes $O(\mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) \cdot |F_H \cup \partial H|^{\omega-1})$ time. If H is a non-leaf node, there are two cases for the update to $\mathbf{L}^{(H)}$ in the algorithm. The first case (Line 35) takes $O\left(|F_H \cup \partial H| \cdot (\sum K_D)\right) \leq O(|F_H \cup \partial H| \cdot K_H)$ time, and the second case (Line 37) takes $O(|F_H \cup \partial H|^2)$ time. Computing the other matrices at any node H takes $O\left(|F_H \cup \partial H|^2 \cdot K_H^{\omega-2}\right)$ time by Lemma 4.9.

The runtime of REWEIGHT$(\delta_{\boldsymbol{w}})$ is therefore given by

$$\sum_{H \in \mathcal{H}} \text{SCHURNODE}(H, \delta_{\boldsymbol{w}})\ \text{time}$$

$$= O\left(\sum_{\mathrm{leaf}\ H \in \mathcal{H}} \mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) \cdot |F_H \cup \partial H|^{\omega-1} + \sum_{H \in \mathcal{H}} |F_H \cup \partial H|^2 \cdot K_H^{\omega-2} \right).$$

For INITIALIZE, we further simplify the expression using $\mathrm{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) = |E(H)|$ and $K_H \leq |F_H \cup \partial H|$. $\qquad \square$

4.5 Projection operator complexities

In this subsection, we summarize the runtime complexities for the tree operator, in the special case when $\mathcal{S}$ is a (a, b, λ)-separator tree for $G_{\mathbf{A}}$. Parametrizing the separator tree this way allows us to write the runtime expressions using geometric series. For non-negative x, we use the standard bound $\sum_{i=\ell}^u x^i \leq O(x^\ell + x^u)$. When it is clear $x < 1$, we bound $\sum_{i=\ell}^u x^i \leq O(x^\ell)$.

Lemma 4.11. *Suppose $\mathcal{S}$ is an (a, b, λ)-separator tree for $G_{\mathbf{A}}$ on n vertices, m edges, with max hyperedge size ρ, where $a \in [0, 1]$ and $b \in (0, 1)$. Let η denote the height of $\mathcal{S}$, and let L denote the number of leaf nodes. Let $\mathbf{\Delta}$ be the tree operator on $\mathcal{T}$ as defined in Section 4.3. Then there is a data structure to maintain $\mathbf{\Delta}$ as a function of the weights $\boldsymbol{w}$ throughout SOLVE, so that:*

- *The data structure can be initialize in time*

$$O\left(\rho^{\omega-1} m + \lambda^\omega \cdot \left(1 + (b^{a\omega-1})^\eta\right)\right). \tag{4.13}$$

- *The query complexity of $\boldsymbol{\Delta}$ is*

$$Q(K) = O\left(\rho K + \lambda^2 \left(1 + (\min\{K, L\})^{1-2a}\right)\right) \tag{4.14}$$

- *When $a < 1$, the update complexity of $\boldsymbol{\Delta}$ is $U(K) =$*

$$\rho^{\omega-1} K + \lambda^2 \min\{K, \lambda\}^{\omega-2} + \begin{cases} \lambda^2 K^{1-2a} & \text{if } K \leq \lambda \\ \lambda^2 K^{1-2a} + \lambda^{\frac{\omega-1}{1-\alpha}} K^{\frac{1-\alpha\omega}{1-\alpha}} & \text{if } \lambda < K \leq \lambda \cdot b^{(a-1)\eta} \\ \lambda^\omega \cdot b^{(a\omega-1)\eta} & \text{if } K > \lambda \cdot b^{(a-1)\eta}. \end{cases} \tag{4.15}$$

When $a = 1$, the update complexity is $U(K) = \rho^{\omega-1} K + \lambda^2 \min\{K, \lambda\}^{\omega-2}$.

Proof. The data structure we use to maintain $\boldsymbol{\Delta}$ is precisely the data structure DYNAMICSC with respect to $\mathcal{S}$.

Initialization time. We use the runtime expression for INITIALIZE in DYNAMICSC (Lemma 4.10) combined with the parameters of the (a, b, λ)-separator tree. For any H, we have $|F_H \cup \partial H| \leq \rho$, so $\sum_{\text{leaf } H \in \mathcal{S}} |E(H)| \cdot |F_H \cup \partial H|^{\omega-1} \leq \rho^{\omega-1} m$. Moreover,

$$\sum_{H \in \mathcal{S}} |F_H \cup \partial H|^\omega \leq \sum_{i=0}^{\eta} b^{-i} \left(\lambda \cdot b^{ai}\right)^\omega \leq O(\lambda^\omega) \cdot \left[1 + (b^{a\omega-1})^\eta\right].$$

Query complexity. We substitute the (a, b, λ)-separator tree bounds in Lemma 4.6, to conclude that the query complexity of $\boldsymbol{\Delta}$ is

$$Q(K) = O\left(\rho K + \sum_{H \in \mathcal{P}_\mathcal{S}(\mathcal{H})} (\lambda \cdot b^{ai})^2\right),$$

where $\mathcal{H}$ is any set of K leaf nodes in $\mathcal{S}$. We group terms according to their node level, and note that there are $\min\{K, b^{-i}\}$-many terms at any level i, so

$$= O\left(\rho K + \sum_{i=0}^{\eta} \min\{K, b^{-i}\} \cdot (\lambda \cdot b^{ai})^2\right)$$

$$= O(\rho K) + O(\lambda^2) \cdot \left(\sum_{i=0}^{-\log_b K} b^{-i} \cdot b^{2ai} + K \cdot \sum_{i=-\log_b K}^{\eta} b^{2ai}\right)$$

Note that K can be at most L in the summation, so we have

$$= O\left(\rho K + \lambda^2(1 + (\min\{K, L\})^{1-2a})\right).$$

Update complexity. When $\boldsymbol{w}$ changes, we update $\boldsymbol{\Delta}$ by invoking REWEIGHT$(\delta_{\boldsymbol{w}})$ in DYNAMICSC, where $\delta_{\boldsymbol{w}}$ denotes the change in $\boldsymbol{w}$. By Lemma 4.10, the runtime for the fixed $\delta_{\boldsymbol{w}}$ is

$$\sum_{\text{leaf } H: \delta_{\boldsymbol{w}}|_{E(H)} \neq \boldsymbol{0}} \text{nnz}(\delta_{\boldsymbol{w}}|_{E(H)}) \cdot |F_H \cup \partial H|^{\omega-1} + \sum_{\text{node } H: \delta_{\boldsymbol{w}}|_{E(H)} \neq \boldsymbol{0}} |F_H \cup \partial H|^2 \cdot K_H^{\omega-2}. \tag{4.16}$$

21

Hence, the update complexity of $\mathbf{\Delta}$ is the max of the above expression taken over all choices of $\delta_{\mathbf{w}}$. For any leaf node H, we upper bound $|F_H \cup \partial H|^{\omega-1} \leq \rho^{\omega-1}$, and therefore the first summation is at most $\rho^{\omega-1}K$.

For the second summation, we substitute in the (a, b, λ)-separator tree bounds, and group terms according to their node level. Let $\mathcal{S}(i)$ denotes all nodes at level i in $\mathcal{S}$. Then for any $\delta_{\mathbf{w}}$, we have

$$\sum_{\text{node } H:\, \delta_{\mathbf{w}}|_{E(H)} \neq \mathbf{0}} |F_H \cup \partial H|^2 \cdot K_H{}^{\omega-2} \leq \sum_{i=0}^{\eta} \left((\lambda \cdot b^{ai})^2 \cdot \sum_{H \in \mathcal{S}(i)} K_H{}^{\omega-2} \right), \qquad (4.17)$$

where the K_H's are non-negative integers satisfying $\sum_{H \in \mathcal{S}(i)} K_H \leq K$ and $K_H \leq |F_H \cup \partial H| \leq \lambda \cdot b^{ai}$ for $H \in \mathcal{S}(i)$. We are interested in upper bounding Eq. (4.17). At any level i, there are b^{-i} nodes, and the sum is maximized when all the K_H's are equal. Depending on the relationship between K and the level i, we have the following three cases:

- If $K \leq b^{-i}$, that is, the total update rank is less than the number of nodes at the level, then the sum is maximized if $K_H = 1$ for K-many nodes, and $K_H = 0$ for the rest.

- If $b^{-i} < K \leq b^{-i} \cdot (\lambda \cdot b^{ai})$, the sum is upper bounded by setting $K_H = K/b^{-i}$.

- If $K > O(b^{-i}) \cdot (\lambda \cdot b^{ai})$, the sum is upper bounded by setting $K_H = \lambda \cdot b^{ai}$.

Then, we can bound the summation term in Eq. (4.17) by

$$\sum_{\substack{0 \leq i \leq \eta \\ K \leq b^{-i}}} K(\lambda \cdot b^{ai})^2 + \sum_{\substack{0 \leq i \leq \eta: \\ b^{-i} < K \leq \lambda \cdot b^{(a-1)i}}} (\lambda \cdot b^{ai})^2 \cdot b^{-i} \cdot (Kb^i)^{\omega-2} + \sum_{\substack{0 \leq i \leq \eta: \\ K > \lambda \cdot b^{(a-1)i}}} b^{-i} \cdot (\lambda \cdot b^{ai})^\omega$$

$$\leq \lambda^2 K \sum_{i=-\log_b K}^{\eta} b^{2ai} + \lambda^2 K^{\omega-2} \sum_{i=\frac{\log_b(K/\lambda)}{a-1}}^{-\log_b K} b^{(2a+\omega-3)i} + \lambda^\omega \sum_{i=0}^{\frac{\log_b(K/\lambda)}{a-1}} b^{(a\omega-1)i}.$$

We need to further consider different cases for the possible values of K, which affects the summation indices. If $\log_b(K/\lambda) < 0$, i.e. $K < \lambda$, the expression simplifies to

$$\lambda^2 K^{1-2a} + \lambda^2 K^{\omega-2}.$$

If $0 \leq \log_b(K/\lambda)/(a-1) \leq \eta$, i.e. $\lambda \leq K \leq \lambda \cdot b^{(a-1)\eta}$, the expression simplifies to

$$\lambda^2 K^{1-2a} + \lambda^{\frac{\omega-1}{1-\alpha}} K^{\frac{1-\alpha\omega}{1-\alpha}} + \lambda^\omega.$$

And lastly, if $\log_b(K/\lambda)/(a-1) > \eta$, i.e $K > \lambda \cdot b^{(a-1)\eta}$, the expression simplifies to

$$\lambda^\omega + \lambda^\omega \cdot b^{(a\omega-1)\eta}.$$

We combine the cases to arrive at the overall update complexity, having implicitly assumed that $\alpha < 1$. When $\alpha = 1$, the summation in Eq. (4.17) is maximized when $K_H = \min\{K, \lambda\} \cdot b^i$ for H at level i. Then we can upper bound the summation term by

$$\sum_{i=0}^{\eta} (\lambda \cdot b^i)^2 \cdot b^{-i} \cdot (\min\{K, \lambda\} \cdot b^i)^{\omega-2} \leq O(\lambda^2 \min\{K, \lambda\}^{\omega-2}).$$

$\square$

22

5 Proofs of main theorems

For our main theorems, it remains to show that we can construct an appropriate (a, b, λ)-separator tree for $G_\mathbf{A}$ in each of the scenarios: when $G_\mathbf{A}$ is n^α-separable; when $\mathbf{A}$ is the constraint matrix for a planar k-multicommodity flow instance; and when $G_\mathbf{A}$ has a tree decomposition of width τ. Then, we apply Lemma 4.11 to the separator tree get the complexity of the tree operator, which we combine with Theorem 3.9 to conclude the overall IPM running times.

5.1 Proof of Theorem 1.2

First, we show how to construct a separator tree for an n^α-separable graph, by modifying the proof from [Fed87].

Lemma 5.1. *Suppose $G_\mathbf{A}$ is a graph on n vertices and m edges. If $G_\mathbf{A}$ is n^α-separable for $\alpha < 1$, then $G_\mathbf{A}$ admits an (α, b, cn^α)-separator tree, where $b \in (0, 1)$ and $c > 0$ are some constants. Furthermore, if a balanced vertex separator for $G_\mathbf{A}$ can be computed in $T(n)$ time, then the separator tree can be computed in $\widetilde{O}(T(n))$ time.*

Proof. Let $b' \in (0, 1)$ and $c' = 1$ (without loss of generality) be the parameters for $G_\mathbf{A}$ being n^α-separable. In the separator tree construction process, assume inductively that we have constants $b \in (0, 1)$ and $c > 0$, both to be chosen later, such that for any node H at level i, we have $|V(H)| \leq b^i n$ and $|\partial H| \leq cn^\alpha \cdot b^{\alpha i}$. In the base case at the root node, we have $i = 0$, and $|V(G_\mathbf{A})| \leq n$ and $|\partial G_\mathbf{A}| = 0 \leq cn^\alpha$.

We show how to construct the nodes at level $i + 1$. Let H be an already-constructed node at level i. There are three cases:

1. If H satisfies $|V(H)| \leq b^{i+1}n$ and $|\partial H| \leq cn^\alpha \cdot b^{\alpha(i+1)}$, put a copy of H as its only child at level $i + 1$.

2. If $|V(H)| \geq b^{i+1}n$, then assign a weight of 1 to all vertices, find a balanced vertex separator $S(H)$, and partition H accordingly into H_1 and H_2. Let us consider H_1; the analogous holds for H_2.

   By definition of separability, we know $|V(H_1)| \leq b' \cdot |V(H)| + |V(H)|^\alpha \leq b \cdot |V(H)| \leq b^{i+1}n$ as long as $b \in (b', 1)$. If $|\partial H_1| \leq c \cdot |V(H_1)|^\alpha$, then we can upper bound this expression by $cn^\alpha \cdot b^{\alpha(i+1)}$, and we are done.

   On the other hand, if $|\partial H_1| > c \cdot |V(H_1)|^\alpha$, then by definition of boundary, we have $|\partial H_1| \leq |\partial H| + |S(H)| \leq (c + 1)n^\alpha \cdot b^{\alpha i}$ using the guarantees at H. Next, we assign a weight of 1 to vertices in ∂H_1 and 0 to all other vertices, find a balanced separator $S(H_1)$ of H_1 with respect to these weights, and create two children D_1, D_2 of H_1 accordingly. Then, for $j = 1, 2$, we have

$$\begin{aligned}
|\partial D_j| &\leq b \cdot |\partial H_1| + |V(H_1)|^\alpha \\
&\leq b(c + 1)n^\alpha \cdot b^{\alpha i} + n^\alpha \cdot b^{\alpha(i+1)} \\
&\leq \left(b^{1-\alpha} \cdot \frac{c+1}{c} + \frac{1}{c} \right) \cdot cn^\alpha \cdot b^{\alpha(i+1)},
\end{aligned}$$

   As long as c is large enough so the expression in the parentheses to be less than 1. In this case, observe that we can add $S(H_1)$ to the balanced separator $S(H)$, and set D_1 and D_2 directly as the children of H.

3. If $|V(H)| \leq b^{(i+1)n}$ and $|\partial H| \geq cn^\alpha \cdot b^{\alpha(i+1)}$, then we apply case 2 with H_1 being H.

23

So we have shown inductively that at the end of this construction, any node H at level i satisfies $|V(H)| \leq b^i n$ and $|\partial H| \leq cn^\alpha \cdot b^{\alpha i}$. It follows that $|F_H \cup \partial H| \leq |S(H)| + |\partial H| = O(cn^\alpha \cdot b^{\alpha i})$.

Next, we show that there are only $O(b^{-i})$ nodes at level i. Let $L_i(n)$ denote the total number of boundary vertices with multiplicities, when carrying out the construction starting on a graph of size n and ending when each leaf node H satisfies the level-i assumptions. We can recursively write

$$L_i(k) = \sum_{j=1}^{4} L_i(b_j k + 3ck^\alpha), \qquad \text{if } k > Cb^i n$$

$$B_i(k) = 1 \qquad\qquad\qquad\qquad \text{else.}$$

where $\sum b_j = 1$, each $b_j \leq b'$, and C is a positive constant we choose. To see this, note that a node of size k has at most four children in the construction; the separator is of size $3ck^\alpha$ since we may need to compute up to three separators each of size ck^α and take their union; and child j has at most $b_j k$ vertices that are not from the separator. Solving the recursion yields $L_i(k) \leq k/(Cb^i n) - \gamma k^\alpha$ for some constant $\gamma > 0$. Therefore, there are at most $L_i(n) \leq O(b^{-i})$ nodes at level i.

Finally, it is straightforward to see that the separator tree can be computed in $\widetilde{O}(T(n))$ time, since the node sizes decrease by a geometric factor as we proceed down the tree during construction. $\qquad\square$

Proof of Theorem 1.2. We consider the cases when $\alpha = 1$ and $\alpha < 1$ separately.

All hypergraphs are trivially n-separable with max hyperedge size $\rho = n$. In this case, let $\mathcal{S}$ be the separator tree consisting of simply one node representing $G_{\mathbf{A}}$, which is a $(1, 1/2, n)$-separator tree. By Lemma 4.11, the tree operator data structure can be initialized in $O(m^\omega)$ time; the query complexity is $Q(K) = O(nK + n^2)$, and the update complexity is $U(K) = O(n^{\omega-1}K + n^2 K^{\omega-2})$.

We apply Theorem 3.9 to get the overall runtime:

$$\widetilde{O}\left(\sqrt{m}\log(\frac{R}{\varepsilon r}) \cdot \sum_{\ell=0}^{\frac{1}{2}\log m} \frac{n2^{2\ell} + n^2 + n^2 2^{2\ell(\omega-2)}}{2^\ell}\right) = \widetilde{O}\left(\sqrt{m}n^2 \log(\frac{R}{\varepsilon r})\right).$$

If $G_{\mathbf{A}}$ is n^α-separable for $\alpha < 1$, then by Lemma 5.1, $G_{\mathbf{A}}$ admits a (α, b, cn^α)-separator tree computable in $\widetilde{O}(n)$ time. In this case, $\rho = O(1)$, and $\eta = O(\log_{1/b} n)$. Plugging the parameters into Lemma 4.11, we get the following tree operator runtimes:

- The data structure can be initialize in $O\left(m + n^{\alpha\omega}(1 + n^{1-\alpha\omega})\right) \leq O(m + m^{\alpha\omega})$ time.

- The query complexity is $Q(K) \leq O(K + n^{2\alpha}(1 + K^{1-2\alpha}))$.

- The update complexity is

$$U(K) \leq O\left(K + n^{2\alpha}\min\{K, n^\alpha\}^{\omega-2} + n^{2\alpha}K^{1-2\alpha} + n^{\frac{\alpha(\omega-1)}{1-\alpha}}K^{\frac{1-\alpha\omega}{1-\alpha}} \cdot \mathbb{1}_{K \geq n^\alpha}\right).$$

We apply Theorem 3.9 to get the overall runtime.

$$\widetilde{O}\left(\sqrt{m}\log(\frac{R}{\varepsilon r})\right) \cdot \sum_{\ell=0}^{\frac{1}{2}\log m} \frac{2^{2\ell} + n^{2\alpha} + n^{\alpha\omega} \cdot \mathbb{1}_{2^{2\ell} > n^\alpha} + n^{2\alpha}2^{(1-2\alpha)2\ell} + n^{\frac{\alpha(\omega-1)}{1-\alpha}}2^{\frac{1-\alpha\omega}{1-\alpha}2\ell} \cdot \mathbb{1}_{2^{2\ell} > n^\alpha}}{2^\ell}$$

$$= \widetilde{O}\left(\sqrt{m}\log(\frac{R}{\varepsilon r})\right) \cdot \left(\sqrt{m} + n^{2\alpha} + n^{\alpha\omega-\frac{\alpha}{2}} + n^{2\alpha}m^{1-2\alpha-\frac{1}{2}} + n^{\frac{\alpha(\omega-1)}{1-\alpha}}\left(n^{\frac{\alpha(1-\alpha\omega)}{1-\alpha}-\frac{\alpha}{2}} + m^{\frac{1-\alpha\omega}{1-\alpha}-\frac{1}{2}}\right)\right)$$

$$= \widetilde{O}\left(\left(m + m^{\frac{1}{2}+2\alpha}\right) \cdot \log(\frac{R}{\varepsilon r})\right),$$

where in the last step, we used the fact $\alpha\omega - \frac{\alpha}{2} \leq 2\alpha$. $\qquad\square$

5.2 Proof of Theorem 1.3

Let $G = (V, E)$ denote the planar graph for the original problem, with $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. First, we write the LP in Eq. (1.1) in standard form by adding slack variables $s \in \mathbb{R}^E$:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{k} c_i^\top f_i \\
\text{s.t} \quad & B^\top f_i = d_i \qquad \forall i \in [k] \\
& \sum_{i=1}^{k} f_i + s = u \\
& f_i \geq 0 \qquad \forall i \in [k] \\
& s \geq 0
\end{aligned}
\tag{P'}
$$

Let A denote the full constraint matrix of P'. Then

$$
A = \left[
\begin{array}{cccc|c}
B^\top & 0 & \cdots & 0 & 0 \\
0 & B^\top & & \vdots & \vdots \\
& & \ddots & & \\
0 & 0 & \cdots & B^\top & 0 \\
\hline
I & I & \cdots & I & I
\end{array}
\right] \in \mathbb{R}^{(kn+m) \times (k+1)m}
\tag{5.1}
$$

where the top left part of A contains k copies of $B^\top$ in block-diagonal fashion, and all the identity matrices are of dimension $m \times m$. The dual graph of $B^\top$ is precisely G. Let G_A be the dual graph of A.

First, we describe G_A: It contains k independent copies of the vertices V, which we label with $V^i = (v_1^i, \ldots, v_n^i)$, so that v_j^i is a copy of $v_j \in V$. Additionally, G_A contains m vertices $u_1, \ldots, u_m$, where the vertex u_i is identified with edge $e_i \in E$. For each edge $e_i \in E$ with endpoints v_{i_1}, v_{i_2}, there are k hyper-edges in G_A of the form $\{v_{i_1}^\ell, v_{i_2}^\ell, u_i\}$ for $\ell = 1, \ldots, k$. Additionally, there are m hyper-edges $f_1, \ldots, f_m$ where f_i contains only the vertex u_i.

Next, we show how to construct an appropriate separator tree efficiently.

Claim 5.2. G_A *admits a* $(\frac{1}{2}, b, kn^{1/2})$-*separator tree that can be computed in* $O(kn \log n)$ *time.*

Proof. Let G be the original planar graph which is $\sqrt{n}$-separable, and let $\tilde{\mathcal{S}}$ be the $(\frac{1}{2}, b, n^{1/2})$-separator tree for G constructed using Lemma 5.1 in $O(n \log n)$ time by [LRT79]. We show how to construct a $(\frac{1}{2}, b, kn^{1/2})$-separator tree $\mathcal{S}$ for G_A based on $\tilde{\mathcal{S}}$. Without loss of generality, we ignore the hyper-edges $f_1, \ldots f_m$ in this construction.

Intuitively, $\mathcal{S}$ will have the same tree structure as $\tilde{\mathcal{S}}$, but each node will be larger by a factor of $O(k)$ due to the k copies of G in G_A. For each $\tilde{H} \in \tilde{\mathcal{S}}$, we construct a corresponding $H \in \mathcal{S}$ as follows: if $v_j \in \tilde{H}$, then $v_j^i \in H$ for all $i \in [k]$; if $e_j \in E(\tilde{H})$, i.e. both endpoints of e_j are in $\tilde{H}$, add u_j to H. Since the k copies $v_j^1, \ldots, v_j^k$ are always grouped together, we will refer to them together as v_j in G_A as well.

Let us show that this is indeed a $(\frac{1}{2}, b, kn^{1/2})$-separator tree. Suppose H is a node with children D_1 and D_2 in $\mathcal{S}$, corresponding to nodes $\tilde{H}, \tilde{D}_1, \tilde{D}_2$ in $\tilde{\mathcal{S}}$. Let $S(H) \stackrel{\text{def}}{=} V(D_1) \cap V(D_2)$, then $v_j \in S(H)$ iff $v_j \in S(\tilde{H})$, and $u_j \in S(H)$ iff $e_j \in E(S(\tilde{H}))$ for all values of j. It is straightforward to see that $S(H)$ is indeed a separator of H. When it comes to the set of boundary vertices, we see $v_j \in \partial H$ iff

25

$v_j \in \partial \tilde{H}$, and if $u_j \in \partial H$ with v_{j_1}, v_{j_2} being the two endpoints of e_j, then v_{j_1}, v_{j_2} are both in ∂H. Since G is a planar graph, the number of edges in $\tilde{H}$ is on the same order as the number of vertices, so we conclude that $|V(H)| \le O(k) \cdot |V(\tilde{H})|$, and similarly, $|F_H \cup \partial H| \le O(k) \cdot |F_{\tilde{H}} \cup \partial \tilde{H}|$. Since node sizes in $\mathcal{S}$ have increased by a factor of $O(k)$ compared to $\tilde{\mathcal{S}}$, we conclude $\mathcal{S}$ is a $(\frac{1}{2}, b, kn^{1/2})$-separator tree.

Finally, we can compute $\tilde{\mathcal{S}}$ for G in $O(n \log n)$ time, so we can compute $\mathcal{S}$ in $O(kn \log n)$ time. $\square$

We reduce our problem to minimum cost multi-commodity circulation problem in order to establish the existence of an interior point in the polytope, before invoking the RIPM in Theorem 3.1. For each commodity $i \in [k]$, we add extra vertices s_i and t_i. Let $\boldsymbol{d}_i$ be the demand vector of the i-th commodity. For every vertex v with $\boldsymbol{d}_{i,v} < 0$, we add a directed edge from s_i to v with capacity $-\boldsymbol{d}_{i,v}$ and cost 0. For every vertex v with $\boldsymbol{d}_{i,v} > 0$, we add a directed edge from v to t_i with capacity $\boldsymbol{d}_{i,v}$ and cost 0. Then, we add a directed edge from t_i to s_i with capacity $4kmM$ and cost $-4kmM$. The modified graph G' has only $2k$ extra vertices of the form s_i and t_i compared to $G_{\mathbf{A}}$, so we can construct a $(\frac{1}{2}, b, kn^{1/2} + 2k)$-separator tree for G' based on the $(\frac{1}{2}, kn^{1/2})$-separator tree for $G_{\mathbf{A}}$, where we include the extra vertices at every node of the tree.

To show the existence of the interior point, we remove all the directed edges that no single commodity flow from s_i to t_i can pass for any $i \in [k]$. This can be done by run BFS for k times which takes $O(km)$ time. For the interior point $\boldsymbol{f}$, we construct this finding a circulation $\boldsymbol{f}^{(e)}$ that passing through e and s_i, t_i for some i with flow value $1/(10km)$ for all the remaining edge e. Then, since the capacities are integers, we find a feasible $\boldsymbol{f}, \boldsymbol{s}$ with value at least $1/(10km)$. This shows the inner radius r of the polytope is at least $1/(10km)$. For the L and R, we note we can bound it by $O(kmM)$.

Let $\mathbf{A}'$ be the constraint matrix of the reduced problem with dual graph G'. The RIPM in Theorem 3.1 invokes the subroutine SOLVE twice. In the first run, we make a new constraint matrix by concatenating $\mathbf{A}'$ three times. One can check that the dual graph is G' with each edge duplicated three times, so the corresponding separator tree is straightforward to construct.

Now, we bounding the running time. The tree operator complexities are similar to the analysis in the previous section with an additional factor of k in the expression for λ. The initialization time is $O(km + (kn^{1/2})^\omega)$. The query complexity is $Q(K) = O(K + k^2 n)$. After simplifying, the update complexity is

$$U(K) = K + \begin{cases} k^2 n K^{\omega - 2} & \text{if } K \le kn^{1/2} \\ (kn^{1/2})^\omega & \text{else.} \end{cases}$$

Note that the number of variables is km. Plugging our choice of L, R, and r, by Theorem 3.9, the total runtime simplifies to

$$\widetilde{O}\left(k^{2.5} m^{1.5} \log(M/\varepsilon)\right).$$

5.3 Proof of Theorem 1.4

First, we show how to construct a $(0, 1/2, O(\tau \log n))$-separator tree $\mathcal{S}$ for $G_{\mathbf{A}}$ when we have a tree decomposition of $G_{\mathbf{A}}$ of width τ. At the root of $\mathcal{S}$, we can use the tree decomposition to compute a balanced separator S of $G_{\mathbf{A}}$ of size $O(\tau)$ in $\widetilde{O}(n\tau)$ time (c.f. [DLY21a, Theorem 4.17]), so that the two parts A and B of $G_{\mathbf{A}} \setminus S$ each have size at most $\frac{2}{3} n$. We construct two children of the root node on the vertex sets $A \cup S$ and $B \cup S$ respectively, and apply this procedure recursively until the nodes are of size at most 9τ.

Claim 5.3. *There are $O(n/\tau)$-many leaves at the end of this construction.*

Proof. Let $L(k)$ denote the number of leaves when starting the construction with a size k subgraph. We know $L(k) = 1$ if $k \leq 9\tau$, and $L(k) = L(k_1 + \tau) + L(k_2 + \tau)$ if $k > 9\tau$, where $k_1 + k_2 + \tau = k$ and $k_1, k_2 \leq 2/3k$. By induction, we can show that $L(k) \leq 2(k/\tau - 1)$ when $k > 2\tau$, where the balanced separator crucially ensures that the recursion does not reach the base case of $k \leq 2\tau$. $\qquad\square$

The resulting separator tree is binary, so there are at most 2^i nodes at level i. Since there are $L = O(n/\tau)$-many leaves, the height η is at most $\eta \leq \log_2(n/\tau)$. The boundary of a node H is contained in the union of balanced separators over its ancestors, so $|F_H \cup \partial H| \leq \tau\eta \leq O(\tau \log n)$. The max hyperedge size of $G_{\mathbf{A}}$ is $\rho = \tau$.

Using these values, we simplify the complexities in Lemma 4.11: The initialization time for the tree operator data structure is $\widetilde{O}\left(\tau^{\omega-1}m + \tau^{\omega}\left(1 + n/\tau\right)\right) = \widetilde{O}(\tau^{\omega-1}m)$. The query complexity of $\boldsymbol{\Delta}$ is $Q(K) = \widetilde{O}\left(\tau K + \tau^2 \min\{K, L\}\right)$. The update complexity of $\boldsymbol{\Delta}$ is

$$U(K) \leq \tau^{\omega-1}K + \begin{cases} \tau^2 K & \text{if } K \leq n \\ \tau^{\omega} & \text{if } K > n \end{cases}$$

Finally, we apply Theorem 3.9 to get the overall runtime, which is clearly bounded by

$$\widetilde{O}\left(\sqrt{m}\log(\frac{R}{\varepsilon r})\right) \cdot \sum_{\ell=0}^{\frac{1}{2}\log m} \frac{\tau^2 2^{2\ell}}{2^\ell} = \widetilde{O}\left(m\tau^2 \log(R/(\varepsilon r))\right).$$

To obtain the faster runtime given in [GS22], we use the data structure restarting trick: Recall MAINTAINAPPROX guarantees there are $2^{2\ell}$-many coordinate updates to $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{s}}$ every 2^ℓ steps, i.e. the number of coordinate updates grows superlinearly with respect to the total number of steps taken. By reinitializing MAINTAINAPPROX with the exact solution once in a while, we limit the total number of coordinate updates. In the proof of Theorem 3.9, we showed that running M steps of the RIPM takes

$$\widetilde{O}\left(U(m) + Q(m) + \eta^4 M \log(\frac{R}{\varepsilon r}) \cdot \sum_{\ell=0}^{\log M} \frac{U(2^{2\ell}) + Q(2^{2\ell})}{2^\ell}\right)$$

time, where $U(m) + Q(m)$ is the time to initialize the data structures and obtain the final exact solutions. There are $N = \sqrt{m}\log m \log(\frac{mR}{\varepsilon r})$-many total IPM steps, and we reinitialize the data structures every M steps. Then the total running time is (ignoring the big-O notation and log factors)

$$\frac{N}{M}\left(U(m) + Q(m) + M\sum_{\ell=0}^{\log M} \frac{U(2^{2\ell}) + Q(2^{2\ell})}{2^\ell}\right)$$
$$= \frac{\sqrt{m}}{M}\left(\tau^{\omega-1}m + \tau^2 M^2\right).$$

The expression is minimized by taking $M = \sqrt{m}\tau^{\frac{\omega-3}{2}}$, which gives an overall runtime of

$$\widetilde{O}\left(m\tau^{(\omega+1)/2}\log(R/(\varepsilon r))\right).$$

$\qquad\square$

Acknowledgements

References

[AMO88]   Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows*. Prentice Hall, 1988.

[AMV22]   Kyriakos Axiotis, Aleksander Mądry, and Adrian Vladu. Faster sparse minimum cost flow by electrical flow localization. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 528–539, 2022.

[AY10]   Noga Alon and Raphael Yuster. Solving linear systems through nested dissection. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 225–234. IEEE, 2010.

[BGS21]   Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental SSSP and approximate min-cost flow in almost-linear time. In *62st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*. IEEE, 2021.

[Bor08]   Glencora Borradaile. *Exploiting Planarity for Network Flow and Connectivity Problems*. Brown University, 2008.

[CEFN23] Erin W. Chambers, Jeff Erickson, Kyle Fox, and Amir Nayyeri. Minimum cuts in surface graphs. *SIAM J. Comput.*, 52(1):156–195, 2023.

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022.

[CKM+11] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 273–282, 2011.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[CLS21]   Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *Journal of the ACM (JACM)*, 68(1):1–39, 2021.

[Dan51]   George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13:339–347, 1951.

[DGG+22] Sally Dong, Yu Gao, Gramoz Goranci, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Guanghao Ye. Nested dissection meets ipms: Planar min-cost flow in nearly-linear time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 124–153. SIAM, 2022.

[DKZ22]   Ming Ding, Rasmus Kyng, and Peng Zhang. Two-commodity flow is equivalent to linear programming under nearly-linear time reductions. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 54:1–54:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[DLY21a]   Sally Dong, Yin Tat Lee, and Guanghao Ye. A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, pages 1784–1797. ACM, 2021.

[DLY21b]   Sally Dong, Yin Tat Lee, and Guanghao Ye. A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path. *arXiv preprint arXiv:2011.05365v2*, 2021.

[DS08]   Samuel I Daitch and Daniel A Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 451–460, 2008.

[DSW14]   Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, pages 271–282, Cham, 2014. Springer International Publishing.

[DWZ22]   Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing. *arXiv preprint arXiv:2210.10173*, 2022.

[EGIS96]   David Eppstein, Zvi Galil, Giuseppe F Italiano, and Thomas H Spencer. Separator based sparsification: I. planarity testing and minimum spanning trees. *journal of computer and system sciences*, 52(1):3–27, 1996.

[Fed87]   Greg N. Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[FF56]   Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[Fle00]   Lisa K Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, 2000.

[Geo73]   Alan George. Nested dissection of a regular finite element mesh. *SIAM journal on numerical analysis*, 10(2):345–363, 1973.

[GHP18]   Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate Schur Complement on separable graphs. In *26th Annual European Symposium on Algorithms, ESA 2018*, volume 112 of *LIPIcs*, pages 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[GK07]   Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007.

[GKK86]   I. Gohberg, T. Kailath, and I. Koltracht. Efficient solution of linear systems of equations with recursive structure. *Linear Algebra and its Applications*, 80:81–113, 1986.

[GLP21]   Yu Gao, Yang P. Liu, and Richard Peng. Fully dynamic electrical flows: Sparse maxflow faster than Goldberg-Rao. In *62st IEEE Annual Symposium on Foundations of Computer Science, FOCS2021*. IEEE, 2021.

[GS22]   Yuzhou Gu and Zhao Song. A faster small treewidth SDP solver. *CoRR*, abs/2211.06033, 2022.

[Has81]   Refael Hassin. Maximum flow in (s, t) planar networks. *Information Processing Letters*, 13(3):107, 1981.

[HJ85]   Refael Hassin and Donald B Johnson. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM Journal on Computing*, 14(3):612–624, 1985.

[HKRS97]   Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[II90]   Hiroshi Imai and Kazuo Iwano. Efficient sequential and parallel algorithms for planar minimum cost flow. In *Algorithms, International Symposium SIGAL '90, Tokyo, Japan*, volume 450 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 1990.

[INSW11]   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*, pages 313–322. ACM, 2011.

[IS79]   Alon Itai and Yossi Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.

[Ita78]   Alon Itai. Two-commodity flow. *J. ACM*, 25(4):596–611, 1978.

[JSWZ21]   Shunhua Jiang, Zhao Song, Omri Weinstein, and Hengjie Zhang. A faster algorithm for solving general lps. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 823–832. ACM, 2021.

[Kar84]   Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[Kha80]   Leonid G Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[KK13]   Ken-ichi Kawarabayashi and Yusuke Kobayashi. All-or-nothing multicommodity flow problem with bounded fractionality in planar graphs. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 187–196, 2013.

[KLOS14]   Jonathan A Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on discrete algorithms*, pages 217–226. SIAM, 2014.

[KLS20]   Tarun Kathuria, Yang P. Liu, and Aaron Sidford. Unit capacity maxflow in almost $o(m^{4/3})$ time. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 119–130, 2020.

[KRT94]   Valerie King, Satish Rao, and Rorbert Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[KS16]   Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians-fast, sparse, and simple. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 573–582. IEEE, 2016.

[KS19]   Adam Karczmarz and Piotr Sankowski. Min-cost flow in unit-capacity planar graphs. In *27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 66:1–66:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[LRT79]   Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979.

[LS19]   Yin Tat Lee and Aaron Sidford. Solving linear programs with sqrt(rank) linear system solves. *CoRR*, abs/1910.08033, 2019.

[LSM⁺91]   Tom Leighton, Clifford Stein, Fillia Makedon, Éva Tardos, Serge Plotkin, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. In *Proceedings of the twenty-third annual ACM symposium on Theory of Computing*, pages 101–111, 1991.

[LSZ19]   Yin Tat Lee, Zhao Song, and Qiuyi Zhang. Solving empirical risk minimization in the current matrix multiplication time. In *Conference on Learning Theory*, pages 2140–2157. PMLR, 2019.

[LT79]   Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[Mad10]   Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 121–130, 2010.

[Mad13]   Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 253–262. IEEE, 2013.

[MNS85]   Kazuhiko Matsumoto, Takao Nishizeki, and Nobuji Saito. An efficient algorithm for finding multicommodity flows in planar networks. *SIAM Journal on Computing*, 14(2):289–302, 1985.

[NN91]   Yurii Nesterov and Arkadi Nemirovsky. Acceleration and parallelization of the path-following interior point method for a linearly constrained convex quadratic problem. *SIAM Journal on Optimization*, 1(4):548–564, 1991.

[OS81]   Haruko Okamura and P.D. Seymour. Multicommodity flows in planar graphs. *Journal of Combinatorial Theory, Series B*, 31(1):75–81, 1981.

[Pen16]   Richard Peng. Approximate undirected maximum flows in o (m polylog (n)) time. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1862–1867. SIAM, 2016.

[Rei83]   John H Reif. Minimum *s-t* cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM Journal on Computing*, 12(1):71–81, 1983.

[Ren88]   James Renegar. A polynomial-time algorithm, based on newton's method, for linear programming. *Mathematical programming*, 40(1-3):59–93, 1988.

[She13]   Jonah Sherman. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 263–269. IEEE, 2013.

[She17]   Jonah Sherman. Area-convexity, linf regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 452–460, 2017.

[SS15]   Aaron Schild and Christian Sommer. On balanced separators in road networks. In Evripidis Bampis, editor, *Experimental Algorithms*, pages 286–297, Cham, 2015. Springer International Publishing.

[ST04]   Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90, 2004.

[ST18]   Aaron Sidford and Kevin Tian. Coordinate methods for accelerating linf regression and faster approximate maximum flow. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 922–933. IEEE, 2018.

[Vai96]   Pravin M Vaidya. A new algorithm for minimizing convex functions over convex sets. *Mathematical programming*, 73(3):291–341, 1996.

[vdB20]   Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 259–278. SIAM, 2020.

[vdBGJ⁺21] Jan van den Brand, Yu Gao, Arun Jambulapati, Yin Tat Lee, Yang P. Liu, Richard Peng, and Aaron Sidford. Faster maxflow via improved dynamic spectral vertex sparsifiers. *CoRR*, abs/2112.00722, 2021.

[vdBLL⁺21] Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, MDPs, and ℓ1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021.

[vdBZ23]   Jan van den Brand and Daniel Zhang. Faster high accuracy multi-commodity flow from single-commodity techniques. *arXiv e-prints*, pages arXiv–2304, 2023.

[Wei97]   Karsten Weihe. Maximum (s, t)-flows in planar networks in $O(|v| \log |v|)$ time. *Journal of Computer and System Sciences*, 55(3):454–475, 1997.

[WXXZ23]   Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. *arXiv preprint arXiv:2307.07970*, 2023.

A   Robust interior point method

For completeness, we include the robust interior point method from [DGG+22], developed in [DLY21b], which is a refinement of the methods in [CLS21, vdB20]. Although there are many other robust interior point methods, we simply refer to this method as RIPM. Consider a linear program of the form

$$\min_{\boldsymbol{x} \in \mathcal{P}} \boldsymbol{c}^\top \boldsymbol{x} \quad \text{where} \quad \mathcal{P} = \{\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \ \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\} \tag{A.1}$$

for some matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$. As with many other IPMs, RIPM follows the central path $\boldsymbol{x}(t)$ from an interior point ($t \gg 0$) to the optimal solution ($t = 0$):

$$\boldsymbol{x}(t) \stackrel{\text{def}}{=} \arg\min_{\boldsymbol{x} \in \mathcal{P}} \boldsymbol{c}^\top \boldsymbol{x} - t\phi(\boldsymbol{x}) \quad \text{where} \quad \phi(\boldsymbol{x}) \stackrel{\text{def}}{=} -\sum_i \log(\boldsymbol{x}_i - \boldsymbol{l}_i) - \sum_i \log(\boldsymbol{u}_i - \boldsymbol{x}_i),$$

where the term ϕ controls how close the solution $\boldsymbol{x}_i$ can be to the constraints $\boldsymbol{u}_i$ and $\boldsymbol{l}_i$. Following the central path exactly is expensive. Instead, RIPM maintains feasible primal and dual solution $(\boldsymbol{x}, \boldsymbol{s}) \in \mathcal{P} \times \mathcal{S}$, where $\mathcal{S}$ is the dual space given by $\mathcal{S} = \{\boldsymbol{s} : \mathbf{A}^\top \boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c} \text{ for some } \boldsymbol{y}\}$, and ensures $\boldsymbol{x}(t)$ is an approximate minimizer. Specifically, the optimality condition for $\boldsymbol{x}(t)$ is given by

$$\mu^t(\boldsymbol{x}, \boldsymbol{s}) \stackrel{\text{def}}{=} \boldsymbol{s}/t + \nabla\phi(\boldsymbol{x}) = \boldsymbol{0} \tag{A.2}$$
$$(\boldsymbol{x}, \boldsymbol{s}) \in \mathcal{P} \times \mathcal{S}$$

where $\mu^t(\boldsymbol{x}, \boldsymbol{s})$ measures how close $\boldsymbol{x}$ is to the minimizer $\boldsymbol{x}(t)$. RIPM maintains $(\boldsymbol{x}, \boldsymbol{s})$ such that

$$\|\gamma^t(\boldsymbol{x}, \boldsymbol{s})\|_\infty \leq \frac{1}{C \log m} \quad \text{where} \quad \gamma^t(\boldsymbol{x}, \boldsymbol{s})_i = \frac{\mu^t(\boldsymbol{x}, \boldsymbol{s})_i}{(\nabla^2\phi(\boldsymbol{x}))_{ii}^{1/2}}, \tag{A.3}$$

for some universal constant C. The normalization term $(\nabla^2\phi)_{ii}^{1/2}$ makes the centrality measure $\|\gamma^t(\boldsymbol{x}, \boldsymbol{s})\|_\infty$ scale-invariant in $\boldsymbol{l}$ and $\boldsymbol{u}$.

The key subroutine SOLVE takes as input a point close to the central path $(\boldsymbol{x}(t_{\text{start}}), \boldsymbol{s}(t_{\text{start}}))$, and outputs another point on the central path $(\boldsymbol{x}(t_{\text{end}}), \boldsymbol{s}(t_{\text{end}}))$. Each step of the subroutine decreases t by a multiplicative factor of $(1 - \frac{1}{\sqrt{m}\log m})$ and moves $(\boldsymbol{x}, \boldsymbol{s})$ within $\mathcal{P} \times \mathcal{S}$ such that $\boldsymbol{s}/t + \nabla\phi(\boldsymbol{x})$ is smaller for the current t. [DLY21b] proved that even if each step is computed approximately, IPM still outputs a point close to $(\boldsymbol{x}(t_{\text{end}}), \boldsymbol{s}(t_{\text{end}}))$ using $\widetilde{O}(\sqrt{m} \log(t_{\text{end}}/t_{\text{start}}))$ steps. See Algorithm 2 for a simplified version.

RIPM calls SOLVE twice. The first call to SOLVE finds a feasible point by following the central path of the following modified linear program

$$\min_{\substack{\mathbf{A}(\boldsymbol{x}^{(1)}+\boldsymbol{x}^{(2)}-\boldsymbol{x}^{(3)})=\boldsymbol{b} \\ \boldsymbol{l} \leq \boldsymbol{x}^{(1)} \leq \boldsymbol{u}, \ \boldsymbol{x}^{(2)} \geq \boldsymbol{0}, \ \boldsymbol{x}^{(3)} \geq \boldsymbol{0}}} \boldsymbol{c}^{(1)\top} \boldsymbol{x}^{(1)} + \boldsymbol{c}^{(2)\top} \boldsymbol{x}^{(3)} + \boldsymbol{c}^{(2)\top} \boldsymbol{x}^{(3)}$$

where $\boldsymbol{c}^{(1)} = \boldsymbol{c}$, and $\boldsymbol{c}^{(2)}, \boldsymbol{c}^{(3)}$ are some positive large vectors. The above modified linear program is chosen so that we know an explicit point on its central path, and any approximate minimizer to this new linear program gives an approximate central path point for the original problem. The second call to SOLVE finds an approximate solution by following the central path of the original linear program.

Algorithm 2 Robust Primal-Dual Interior Point Method from [DLY21b]

1: **procedure** RIPM($\mathbf{A} \in \mathbb{R}^{n \times m}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u}, \epsilon$)

2:     Define $L \stackrel{\text{def}}{=} \|\boldsymbol{c}\|_2$ and $R \stackrel{\text{def}}{=} \|\boldsymbol{u} - \boldsymbol{l}\|_2$

3:     Define $\phi_i(x) \stackrel{\text{def}}{=} -\log(\boldsymbol{u}_i - x) - \log(x - \boldsymbol{l}_i)$

4:     Define $\mu_i^t(\boldsymbol{x}, \boldsymbol{s}) \stackrel{\text{def}}{=} \boldsymbol{s}_i/t + \nabla \phi_i(\boldsymbol{x}_i)$

    ▷ Modify the linear program and obtain an initial $(\boldsymbol{x}, \boldsymbol{s})$ for modified linear program

5:     Let $t = 2^{21} m^5 \cdot \frac{LR}{128} \cdot \frac{R}{r}$

6:     Compute $\boldsymbol{x}_c = \arg\min_{\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}} \boldsymbol{c}^\top \boldsymbol{x} + t\phi(\boldsymbol{x})$ and $\boldsymbol{x}_\circ = \arg\min_{\mathbf{A}\boldsymbol{x} = \boldsymbol{b}} \|\boldsymbol{x} - \boldsymbol{x}_c\|_2$

7:     Let $\boldsymbol{x} = (\boldsymbol{x}_c, 3R + \boldsymbol{x}_\circ - \boldsymbol{x}_c, 3R)$ and $\boldsymbol{s} = (-t\nabla\phi(\boldsymbol{x}_c), \frac{t}{3R + \boldsymbol{x}_\circ - \boldsymbol{x}_c}, \frac{t}{3R})$

8:     Let the new matrix $\mathbf{A}^{\text{new}} \stackrel{\text{def}}{=} [\mathbf{A}; \mathbf{A}; -\mathbf{A}]$, the new barrier

$$\phi_i^{\text{new}}(x) = \begin{cases} \phi_i(x) & \text{if } i \in [m], \\ -\log x & \text{else.} \end{cases}$$

    ▷ Find an initial $(\boldsymbol{x}, \boldsymbol{s})$ for the original linear program

9:     $((\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}), (\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}, \boldsymbol{s}^{(3)})) \leftarrow \text{SOLVE}(\mathbf{A}^{\text{new}}, \phi^{\text{new}}, \boldsymbol{x}, \boldsymbol{s}, t, LR)$

10:     $(\boldsymbol{x}, \boldsymbol{s}) \leftarrow (\boldsymbol{x}^{(1)} + \boldsymbol{x}^{(2)} - \boldsymbol{x}^{(3)}, \boldsymbol{s}^{(1)})$

    ▷ Optimize the original linear program

11:     $(\boldsymbol{x}, \boldsymbol{s}) \leftarrow \text{SOLVE}(\mathbf{A}, \phi, \boldsymbol{x}, \boldsymbol{s}, LR, \frac{\epsilon}{4m})$

12:     **return** $\boldsymbol{x}$

13: **end procedure**

14: **procedure** SOLVE($\mathbf{A}, \phi, \boldsymbol{x}, \boldsymbol{s}, t_{\text{start}}, t_{\text{end}}$)

15:     Define $\alpha \stackrel{\text{def}}{=} \frac{1}{2^{20}\lambda}$ and $\lambda \stackrel{\text{def}}{=} 64\log(256m^2)$

16:     Let $t \leftarrow t_{\text{start}}, \overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}, \overline{\boldsymbol{s}} \leftarrow \boldsymbol{s}, \overline{t} \leftarrow t$

17:     **while** $t \geq t_{\text{end}}$ **do**

18:         $t \leftarrow \max((1 - \frac{\alpha}{\sqrt{m}})t, t_{\text{end}})$

19:         Update step size $h = -\alpha/\|\cosh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}))\|_2$ where γ is defined in Eq. (A.3)

20:         Update diagonal weight matrix $\mathbf{W} = \nabla^2\phi(\overline{\boldsymbol{x}})^{-1}$

21:         Update step direction $\boldsymbol{v}$ where $\boldsymbol{v}_i = \sinh(\lambda\gamma^{\overline{t}}(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}})_i) \cdot \mu^{\overline{t}}(\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}})_i$

22:         Implicitly update $\boldsymbol{x}, \boldsymbol{s}$, with $\mathbf{P}_{\boldsymbol{w}} \stackrel{\text{def}}{=} \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + h\mathbf{W}^{1/2}(\boldsymbol{v} - \mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}),$$
$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \overline{t}h\mathbf{W}^{-1/2}\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$$

23:         Explicitly update $\overline{\boldsymbol{x}}, \overline{\boldsymbol{s}}$ such that

$$\|\mathbf{W}^{-1/2}(\overline{\boldsymbol{x}} - \boldsymbol{x})\|_\infty \leq \alpha,$$
$$\|\mathbf{W}^{1/2}(\overline{\boldsymbol{s}} - \boldsymbol{s})\|_\infty \leq \overline{t}\alpha$$

24:         If $|\overline{t} - t| \geq \alpha\overline{t}$, update $\overline{t} \leftarrow t$

25:     **end while**

26:     **return** $(\boldsymbol{x}, \boldsymbol{s})$

27: **end procedure**

Theorem 3.1 (RIPM). *Consider the linear program*

$$\min_{\mathbf{A}\boldsymbol{x}=\boldsymbol{b},\, \boldsymbol{l}\leq\boldsymbol{x}\leq\boldsymbol{u}} \boldsymbol{c}^\top\boldsymbol{x}$$

with $\mathbf{A}\in\mathbb{R}^{n\times m}$. *We are given a scalar* $r>0$ *such that there exists some interior point* $\boldsymbol{x}_\circ$ *satisfying* $\mathbf{A}\boldsymbol{x}_\circ=\boldsymbol{b}$ *and* $\boldsymbol{l}+r\leq\boldsymbol{x}_\circ\leq\boldsymbol{u}-r$. *Let* $L=\|\boldsymbol{c}\|_2$ *and* $R=\|\boldsymbol{u}-\boldsymbol{l}\|_2$. *For any* $0<\varepsilon\leq 1/2$, *the algorithm RIPM (Algorithm 2) finds* $\boldsymbol{x}$ *such that* $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$, $\boldsymbol{l}\leq\boldsymbol{x}\leq\boldsymbol{u}$ *and*

$$\boldsymbol{c}^\top\boldsymbol{x}\leq \min_{\mathbf{A}\boldsymbol{x}=\boldsymbol{b},\, \boldsymbol{l}\leq\boldsymbol{x}\leq\boldsymbol{u}} \boldsymbol{c}^\top\boldsymbol{x}+\varepsilon LR.$$

Furthermore, the algorithm has the following properties:

- *Each call of* SOLVE *involves* $O(\sqrt{m}\log m\log(\frac{mR}{\epsilon r}))$*-many steps, and* $\bar{t}$ *is only updated* $O(\log m\log(\frac{mR}{\epsilon r}))$*-many times.*

- *In each step of* SOLVE, *the coordinate* i *in* $\boldsymbol{w},\boldsymbol{v}$ *changes only if* $\overline{\boldsymbol{x}}_i$ *or* $\overline{\boldsymbol{s}}_i$ *changes.*

- *In each step of* SOLVE, $h\|\boldsymbol{v}\|_2=O(\frac{1}{\log m})$.

- *Line 19 to Line 21 takes* $O(K)$ *time in total, where* K *is the total number of coordinate changes in* $\overline{\boldsymbol{x}},\overline{\boldsymbol{s}}$.

Proof. The number of steps follows from Theorem A.1 in [DLY21b], with the parameter $\boldsymbol{w}_i=\nu_i=1$ for all i. The number of coordinate changes in $\mathbf{W},\boldsymbol{v}$ and the runtime of Line 19 to Line 21 follows directly from the formula of $\mu^t(\boldsymbol{x},\boldsymbol{s})_i$ and $\gamma^t(\boldsymbol{x},\boldsymbol{s})_i$. For the bound for $h\|\boldsymbol{v}\|_2$, it follows from

$$h\|\boldsymbol{v}\|_2\leq\alpha\frac{\|\sinh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{x}},\overline{\boldsymbol{s}}))\|_2}{\|\cosh(\lambda\gamma^{\bar{t}}(\overline{\boldsymbol{x}},\overline{\boldsymbol{s}}))\|_2}\leq\alpha=O\left(\frac{1}{\log m}\right).$$

$\square$

B   Maintaining the implicit representation

In this section, we give the general data structure MAINTAINREP, which implicitly maintains a vector $\boldsymbol{x}$ throughout a call of SOLVE of Algorithm 2. We break up the representation into two parts, the first using the inverse tree operator, and the second using the tree operator.

First, we present some of the alternative decomposition properties of the tree operator.

Definition B.1 (Subtree operator). Let $\boldsymbol{\Delta}$ be a tree operator on $\mathcal{T}$. Recall $\mathcal{T}_H$ is the complete subtree of $\mathcal{T}$ rooted at H. We define the subtree operator $\boldsymbol{\Delta}^{(H)}$ at each node H to be

$$\boldsymbol{\Delta}^{(H)}\overset{\text{def}}{=}\sum_{\text{leaf }L\in\mathcal{T}_H}\boldsymbol{\Delta}_{L\leftarrow H}. \tag{B.1}$$

Corollary B.2. *Based on the above definitions, we have*

$$\boldsymbol{\Delta}=\sum_{H\in\mathcal{T}}\boldsymbol{\Delta}^{(H)}\mathbf{I}_{F_H}. \tag{B.2}$$

Furthermore, if H *has children* H_1,H_2, *then*

$$\boldsymbol{\Delta}^{(H)}=\boldsymbol{\Delta}^{(H_1)}\boldsymbol{\Delta}_{H_1}+\boldsymbol{\Delta}^{(H_2)}\boldsymbol{\Delta}_{H_2}. \tag{B.3}$$

35

The output of $\boldsymbol{\Delta}$ when restricted to $E(H)$ for a node $H \in \mathcal{T}$ can be written in two parts, which is useful for our data structures. The first part involves summing over all nodes in $\mathcal{T}_H$, ie. descendants of H and H itself, and the second part involves a sum over all ancestors of H.

Lemma B.3. *At any node $H \in \mathcal{T}$, we have*

$$\mathbf{I}_{E(H)}\boldsymbol{\Delta} = \sum_{D \in \mathcal{T}_H} \boldsymbol{\Delta}^{(D)}\mathbf{I}_{F_D} + \boldsymbol{\Delta}^{(H)} \sum_{\text{ancestor } A \text{ of } H} \boldsymbol{\Delta}_{H \leftarrow A}\mathbf{I}_{F_A}.$$

Proof. We consider the terms in the sum for $\boldsymbol{\Delta}$ that map into to $E(H)$, which is precisely the set of leaf nodes in the subtree rooted at H.

$$\mathbf{I}_{E(H)}\boldsymbol{\Delta} = \sum_{\text{leaf } L \in \mathcal{T}_H} \sum_{A : L \in \mathcal{T}_A} \boldsymbol{\Delta}_{L \leftarrow A}\mathbf{I}_{F_A}.$$

The right hand side involves a sum over the set $\{(L, A) \ : \ \text{leaf } L \in \mathcal{T}_H, L \in \mathcal{T}_A\}$. Observe that (L, A) is in this set if and only if A is a descendant of H, or $A = H$, or A is an ancestor of H. Hence, the summation can be written as

$$\sum_{\text{leaf } L \in \mathcal{T}_H} \sum_{\text{node } A \in \mathcal{T}_H} \boldsymbol{\Delta}_{L \leftarrow A}\mathbf{I}_{F_A} + \sum_{\text{leaf } L \in \mathcal{T}_H} \sum_{\text{ancestor } A \text{ of } H} \boldsymbol{\Delta}_{L \leftarrow A}\mathbf{I}_{F_A}.$$

The first term is precisely the first term in the lemma statement. For the second term, we can use the fact that A is an ancestor of H to expand $\boldsymbol{\Delta}_{L \leftarrow A} = \boldsymbol{\Delta}_{L \leftarrow H}\boldsymbol{\Delta}_{H \leftarrow A}$. Then, the second term is

$$\sum_{\text{leaf } L \in \mathcal{T}_H} \sum_{\text{ancestor } A \text{ of } H} \boldsymbol{\Delta}_{L \leftarrow H}\boldsymbol{\Delta}_{H \leftarrow A}\mathbf{I}_{F_A}$$

$$= \sum_{\text{leaf } L \in \mathcal{T}_H} \boldsymbol{\Delta}_{L \leftarrow H} \left(\sum_{\text{ancestor } A \text{ of } H} \boldsymbol{\Delta}_{H \leftarrow A}\mathbf{I}_{F_A} \right)$$

$$= \boldsymbol{\Delta}^{(H)} \left(\sum_{\text{ancestor } A \text{ of } H} \boldsymbol{\Delta}_{H \leftarrow A}\mathbf{I}_{F_A} \right),$$

by definition of $\boldsymbol{\Delta}^{(H)}$. $\qquad\qquad\square$

Now, we consider the cost of applying the inverse tree operator and the tree operator.

Lemma B.4. *Let $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$ be an inverse tree operator on $\mathcal{T}$ with query complexity Q. Given $\boldsymbol{v} \in \mathbb{R}^E$, we can compute $\nabla\boldsymbol{v}$ as well as $\boldsymbol{y}_H \overset{\text{def}}{=} \sum_{\text{leaf } L \in \mathcal{T}_H} \nabla_{H \leftarrow L}\boldsymbol{v}$ for all $H \in \mathcal{T}$ in $O(Q(\eta K))$ time, where $K = \text{nnz}(\boldsymbol{v})$ and η is the height of $\mathcal{T}$.*

Proof. Recall the definition

$$\nabla\boldsymbol{v} \overset{\text{def}}{=} \sum_{\text{leaf } L} \left(\sum_{H : L \in \mathcal{T}_H} \mathbf{I}_{F_H}\nabla_{H \leftarrow L} \right) \boldsymbol{v}.$$

At a leaf node L, if we have $\boldsymbol{v}_e = 0$ for all $e \in E(L)$, then we can ignore the term for L in the outer sum. So we can reduce $\nabla\boldsymbol{v}$ to consist of at most K terms in the outer sum. We can further rearrange the order of applying the edge operators so that each edge operator is applied at most once, and this naturally gives the values for all non-zero $\boldsymbol{y}_H$'s. We bound the overall runtime loosely by $O(Q(\eta K))$. $\qquad\qquad\square$

Unlike the inverse tree operator, the tree operator is applied downwards along a tree, and therefore we do not have non-trivial bounds on total number of edge operators applied. Instead, we have a more general bound:

Lemma B.5. *Let $\boldsymbol{\Delta} : \mathbb{R}^V \mapsto \mathbb{R}^E$ be a tree operator on $\mathcal{T}$ with query complexity Q. Given $\boldsymbol{z} \in \mathbb{R}^V$, we can compute $\boldsymbol{\Delta v}$ in $O(Q(|E|))$ time.*

Proof. We simply observe that we can compute $\boldsymbol{\Delta v}$ by applying each edge operator at most once. Since the leaf nodes partition the set E, we know in $\mathcal{T}$, there are $O(|E|)$ edge operators in total, so the overall time is at most $O(Q(|E|))$. □

With the appropriate partial computations taking advantage of the decomposition of ∇, we can maintain $\nabla \boldsymbol{v}$ efficiently for dynamic ∇ and $\boldsymbol{v}$. Specifically, we use the following property:

Lemma B.6. *Given a vector $\boldsymbol{v} \in \mathbb{R}^E$, let $\boldsymbol{y}_H \stackrel{\text{def}}{=} \sum_{\text{leaf } L \in \mathcal{T}_H} \nabla_{H \leftarrow L} \boldsymbol{v}$ for each $H \in \mathcal{T}$. If H has children H_1, H_2, then*

$$\boldsymbol{y}_H = \nabla_{H_1} \boldsymbol{y}_{H_1} + \nabla_{H_2} \boldsymbol{y}_{H_2}. \tag{B.4}$$

Furthermore,

$$\sum_{H \in \mathcal{T}} \mathbf{I}_{F_H} \boldsymbol{y}_H = \nabla \boldsymbol{v}. \tag{B.5}$$

□

Lemma B.7. *Let $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$ be an inverse tree operator with query complexity Q. Let $\nabla^{(\text{new})}$ be ∇ with K updated edge operators. Suppose we know $\nabla \boldsymbol{v}$, and we know $\boldsymbol{y}_H \stackrel{\text{def}}{=} \sum_{\text{leaf } L \in \mathcal{T}_H} \nabla_{H \leftarrow L} \boldsymbol{v}$ at all nodes H, then we can compute $(\nabla^{(\text{new})} - \nabla)\boldsymbol{v}$ and the $\boldsymbol{y}_H^{(\text{new})}$'s in $O(Q(\eta K))$ time.*

Proof. Observe that for a node $H \in \mathcal{T}$, if no edge operator in $\mathcal{T}_H$ was updated, then $\boldsymbol{y}_H$ remains the same. We use Eq. (B.4) to compute $\boldsymbol{y}_H^{(\text{new})}$ up the tree for the $O(\eta K)$-many nodes that admit changes, and then Eq. (B.5) to extract the change $(\nabla^{(\text{new})} - \nabla)\boldsymbol{v}$. □

Now we are ready for the complete data structure involving the inverse tree operator.

Theorem B.8 (Inverse tree operator data structure)**.** *Let $\boldsymbol{w} \in \mathbb{R}^m$ be the weights changing at every step of SOLVE, and let $\boldsymbol{v} \in \mathbb{R}^n$ be a dynamic vector. Suppose $\nabla : \mathbb{R}^m \mapsto \mathbb{R}^n$ is an inverse tree operator dependent on $\boldsymbol{w}$ supported on $\mathcal{T}$ with query complexity Q and update complexity U. Let η be the height of $\mathcal{T}$. Then the data structure INVERSETREEOP (Algorithm 3) maintains $\boldsymbol{z}^{(k)} \stackrel{\text{def}}{=} \sum_{i=1}^{k} h^{(i)} \nabla^{(i)} \boldsymbol{v}^{(i)}$ so that at the end of each step k, the variables in the algorithm satisfy*

- *$\boldsymbol{z} = c\boldsymbol{z}^{(\text{step})} + \boldsymbol{z}^{(\text{sum})}$,*

- *$\boldsymbol{z}^{(\text{step})} = \nabla \boldsymbol{v}$, and*

- *$\boldsymbol{y}_H = \sum_{\text{leaf } L \in \mathcal{T}_H} \nabla_{H \leftarrow L} \boldsymbol{v}$ for all nodes H.*

The data structure is initialized via INITIALIZE in $O(U(m) + Q(m))$ time. At step k, there is one call REWEIGHT($\delta_{\boldsymbol{w}}$) taking $O(U(K) + Q(\eta K))$ time, where $K = \text{nnz}(\delta_{\boldsymbol{w}})$, followed by one call of MOVE($h, \delta_{\boldsymbol{v}}$) taking $O(Q(\eta \cdot \text{nnz}(\delta_{\boldsymbol{v}})))$ time.

Algorithm 3 Dynamic data structure to maintain cumulative ∇v

1: **dynamic data structure INVERSETREEOP**
2: **member:**
3:     $\mathcal{T}$: tree supporting ∇ with edge operators on the edges
4:     $\boldsymbol{w} \in \mathbb{R}^m$: dynamic weight vector
5:     $\boldsymbol{v} \in \mathbb{R}^n$: dynamic vector
6:     $c, \boldsymbol{z}^{(\text{step})}, \boldsymbol{z}^{(\text{sum})} \in \mathbb{R}^n$: coefficient, result vectors
7:     $\boldsymbol{y}_H \in \mathbb{R}^n$ for each $H \in \mathcal{T}$: sparse partial computations
8:
9: **procedure** INITIALIZE$(\mathcal{T}, \boldsymbol{w}^{(\text{init})}, \boldsymbol{v}^{(\text{init})})$
10:     $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{init})}, \boldsymbol{v} \leftarrow \boldsymbol{v}^{(\text{init})}, c \leftarrow 0, \boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{0}$
11:     Initialize ∇ on $\mathcal{T}$ based on $\boldsymbol{w}$
12:     Compute $\nabla \boldsymbol{v}$ and $\boldsymbol{y}_H$'s, set $\boldsymbol{z}^{(\text{step})} \leftarrow \nabla \boldsymbol{v}$
13: **end procedure**
14:
15: **procedure** REWEIGHT$(\delta_{\boldsymbol{w}})$
16:     $\boldsymbol{w}^{(\text{new})} \leftarrow \boldsymbol{w} + \delta_{\boldsymbol{w}}$
17:     Let $\nabla^{(\text{new})}$ be the new tree operator using $\boldsymbol{w}^{(\text{new})}$
18:     $\boldsymbol{z}' \leftarrow (\nabla^{(\text{new})} - \nabla)\boldsymbol{v}$, and update $\boldsymbol{y}_H$'s $\triangleright$ Lemma B.7
19:     $\boldsymbol{z}^{(\text{step})} \leftarrow \boldsymbol{z}^{(\text{step})} + \boldsymbol{z}'$
20:     $\boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c \cdot \boldsymbol{z}'$
21:     $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{new})}, \nabla \leftarrow \nabla^{(\text{new})}$
22: **end procedure**
23:
24: **procedure** MOVE$(h, \delta_{\boldsymbol{v}})$
25:     Compute $\boldsymbol{z}' \stackrel{\text{def}}{=} \nabla \delta_{\boldsymbol{v}}$ and the $\boldsymbol{y}'_H \stackrel{\text{def}}{=} \sum_{\text{leaf } L \in \mathcal{T}_H} \nabla_{H \leftarrow L} \delta_{\boldsymbol{v}}$ for each node H $\triangleright$ Lemma B.4
26:     $\boldsymbol{z}^{(\text{step})} \leftarrow \boldsymbol{z}^{(\text{step})} + \boldsymbol{z}'$, and $\boldsymbol{y}_H \leftarrow \boldsymbol{y}_H + \boldsymbol{y}'_H$ for each node H
27:     $\boldsymbol{z}^{(\text{sum})} \leftarrow \boldsymbol{z}^{(\text{sum})} - c\boldsymbol{z}'$
28:     $c \leftarrow c + h$
29:     $\boldsymbol{v} \leftarrow \boldsymbol{v} + \delta_{\boldsymbol{v}}$
30: **end procedure**

Proof. In the data structure, we always maintain $\boldsymbol{z}^{(\text{step})}$ and the $\boldsymbol{y}_H$'s together. Specifically, at every step, we update the $\boldsymbol{y}_H$'s up the tree using the recursive property Eq. (B.4) only at the necessary nodes, and from the $\boldsymbol{y}_H$'s, we get $\boldsymbol{z}^{(\text{step})} = \sum_H \mathbf{I}_{F_H} \boldsymbol{y}_H$.

Consider INITIALIZE. At the end of the function, the variables satisfy

$$\boldsymbol{z} \stackrel{\text{def}}{=} c\boldsymbol{z}^{(\text{step})} + \boldsymbol{z}^{(\text{sum})} = 0 \cdot \nabla \boldsymbol{v} + \boldsymbol{0} = \boldsymbol{0},$$

and $\boldsymbol{z}^{(\text{step})} = \nabla \boldsymbol{v}$, as required.

Let us consider REWEIGHT. Let the superscript $^{(\text{new})}$ denote the value of an algorithm variable at the end of the function, and let no superscript denote the value at the start.

$$\begin{aligned}
\boldsymbol{z}^{(\text{new})} &= c^{(\text{new})}\boldsymbol{z}^{(\text{step})\,(\text{new})} + \boldsymbol{z}^{(\text{sum})\,(\text{new})} \\
&= c(\boldsymbol{z}^{(\text{step})} + \boldsymbol{z}') + \boldsymbol{z}^{(\text{sum})} - c\boldsymbol{z}' \\
&= c\boldsymbol{z}^{(\text{step})} + \boldsymbol{z}^{(\text{sum})},
\end{aligned}$$

$$\begin{aligned}
\text{and } \boldsymbol{z}^{(\text{step})\,(\text{new})} &= \boldsymbol{z}^{(\text{step})} + (\nabla^{(\text{new})} - \nabla)\boldsymbol{v} \\
&= \nabla^{(\text{new})}\boldsymbol{v},
\end{aligned}$$

as required. Similarly, let us consider MOVE:

$$\begin{aligned}
\boldsymbol{z}^{(\text{new})} &= c^{(\text{new})}\boldsymbol{z}^{(\text{step})\,(\text{new})} + \boldsymbol{z}^{(\text{sum})\,(\text{new})} \\
&= (c + h)(\boldsymbol{z}^{(\text{step})} + \boldsymbol{z}') + \boldsymbol{z}^{(\text{sum})} - c\boldsymbol{z}' \\
&= c\boldsymbol{z}^{(\text{step})} + \boldsymbol{z}^{(\text{sum})} + h\boldsymbol{z}^{(\text{step})},
\end{aligned}$$

$$\begin{aligned}
\text{and } \boldsymbol{z}^{(\text{step})\,(\text{new})} &= \boldsymbol{z}^{(\text{step})} + \nabla(\boldsymbol{v}^{(\text{new})} - \boldsymbol{v}) \\
&= \nabla \boldsymbol{v} + \nabla(\boldsymbol{v}^{(\text{new})} - \boldsymbol{v}) \\
&= \nabla \boldsymbol{v}^{(\text{new})},
\end{aligned}$$

which is exactly the update we want to make to $\boldsymbol{z}$, and the invariant we want to maintain.

The runtimes follow directly from Lemmas B.4 and B.7. $\qquad\square$

Next, we present the tree operator data structure, which is significantly more involved compared to the inverse tree operator. Applying the tree operator involves going down the tree to the leaves, which is too costly to do at every step. To circumvent the issue, we use lazy computations.

Theorem B.9 (Tree operator data structure). *Let $\boldsymbol{w} \in \mathbb{R}^m$ be the weights changing at every step of SOLVE. Suppose $\boldsymbol{\Delta} : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a tree operator dependent on $\boldsymbol{w}$ supported on $\mathcal{T}$ with query complexity Q and update complexity U. Let $\boldsymbol{z} \in \mathbb{R}^n$ be the vector maintained by Algorithm 3, so that at the end of step k, $\boldsymbol{z} = \sum_{i=1}^{k} h^{(i)} \nabla^{(i)} \boldsymbol{v}^{(i)}$. Then the data structure TREEOP (Algorithm 4) implicitly maintains $\boldsymbol{x}$ so that at the end of step k,*

$$\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(\text{init})} + \sum_{i=1}^{k} \boldsymbol{\Delta}^{(i)} \nabla^{(i)} \boldsymbol{v}^{(i)}.$$

The data structure is initialized via INITIALIZE in $O(U(m))$ time. At step k, there is one call to REWEIGHT$(\delta_{\boldsymbol{w}})$ taking $O(U(K) + Q(\eta K))$ time, where $K = \text{nnz}(\delta_{\boldsymbol{w}})$, followed by one call to MOVE$(\delta_{\boldsymbol{z}})$ taking $O(\text{nnz}(\delta_{\boldsymbol{z}}))$ time. At the end of SOLVE, $\boldsymbol{x}$ is returned via EXACT in $O(Q(m))$ time.

Algorithm 4 Dynamic data structure to maintain cumulative $\boldsymbol{\Delta z}$

1: **dynamic data structure** TREEOP
2: **member:**
3:     $\mathcal{T}$: tree supporting $\boldsymbol{\Delta}$
4:     $\boldsymbol{w} \in \mathbb{R}^m$: dynamic weight vector
5:     $\boldsymbol{z} \in \mathbb{R}^n$: dynamic vector
6:     $\boldsymbol{u}_H$ for each $H \in \mathcal{T}$: lazy pushdown computation vectors
7:
8: **procedure** INITIALIZE($\mathcal{T}, \boldsymbol{w}^{(\text{init})}, \boldsymbol{z}^{(\text{init})}, \boldsymbol{x}^{(\text{init})}$)
9:     $\boldsymbol{w} \leftarrow \boldsymbol{w}^{(\text{init})}, \boldsymbol{z} \leftarrow \boldsymbol{z}^{(\text{init})}$
10:     Initialize $\boldsymbol{\Delta}$ on $\mathcal{T}$ based on $\boldsymbol{w}$
11:     $\boldsymbol{u}_H \leftarrow \boldsymbol{0}$ for each non-leaf $H \in \mathcal{T}$
12:     $\boldsymbol{u}_H \leftarrow \boldsymbol{x}^{(\text{init})}|_{E(H)}$ for each leaf $H \in \mathcal{T}$
13: **end procedure**
14:
15: **procedure** REWEIGHT($\delta_{\boldsymbol{w}}$)
16:     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \delta_{\boldsymbol{w}}$
17:     Let $\boldsymbol{\Delta}^{(\text{new})}$ be the new tree operator wrt the new weights
18:     Let $\mathcal{H}$ be all nodes H where $\boldsymbol{\Delta}_H$ changed
19:     **for** $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ going down the tree level by level **do**
20:         PUSHDOWN(H)
21:     **end for**
22:     **for** $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ going down the tree level by level **do**
23:         $\boldsymbol{u}_H \leftarrow c\boldsymbol{z}|_{F_H}$
24:         PUSHDOWN(H)
25:     **end for**
26:     $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta}^{(\text{new})}$
27:     **for** $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ going down the tree level by level **do**
28:         $\boldsymbol{u}_H \leftarrow -c\boldsymbol{z}|_{F_H}$
29:         PUSHDOWN(H)
30:     **end for**
31: **end procedure**
32:
33: **procedure** MOVE($\delta_{\boldsymbol{z}}$)
34:     $\boldsymbol{z} \leftarrow \boldsymbol{z} + \delta_{\boldsymbol{z}}$
35: **end procedure**
36:
37: **procedure** EXACT
38:     **for** $H \in \mathcal{T}$ going down the tree level by level **do**
39:         $\boldsymbol{u}_H \leftarrow \boldsymbol{u}_H + \boldsymbol{z}|_{F_H}$
40:         PUSHDOWN(H)
41:     **end for**
42:     **return** $\boldsymbol{x}$ defined by $\boldsymbol{x}|_{E(H)} \stackrel{\text{def}}{=} \boldsymbol{u}_H$ at each leaf $H \in \mathcal{T}$
43: **end procedure**

Algorithm 5 Dynamic data structure to maintain cumulative $\mathbf{\Delta} z$, con't

---

1: **dynamic data structure** TREEOP
2: **procedure** PUSHDOWN($H \in \mathcal{T}$)
3:     **for** each child D of H **do**
4:         $\boldsymbol{u}_D \leftarrow \boldsymbol{u}_D + \mathbf{\Delta}_D \boldsymbol{u}_H$
5:     **end for**
6:     $\boldsymbol{u}_H \leftarrow \mathbf{0}$
7: **end procedure**

Proof. We will show that the data structure maintains the implicit representation via the identity

$$\boldsymbol{x} = c\mathbf{\Delta} z + \sum_{H \in \mathcal{T}} \mathbf{\Delta}^{(H)} \boldsymbol{u}_H, \tag{B.6}$$

where the RHS expression refers to the state of the variables at the end of step k during the algorithm.

At a high level, the variables $\mathbf{\Delta}$ and z in the data structure at step k represent the latest $\mathbf{\Delta}^{(k)}$ and $z^{(k)}$. We need to introduce additional vectors $\boldsymbol{u}_H$ at every node H which intuitively stores lazy computations at node H, in order to take advantage of the tree structure of $\mathbf{\Delta}$. The function PUSHDOWN performs the accumulated computation at H, and moves the result to its children nodes to be computed lazily at a later point. The next claim describes this process formally.

Claim B.10. *Let $H \in \mathcal{T}$ be a non-leaf node.* PUSHDOWN(H) *does not change the value of the implicit representation in Eq.* (B.6). *Also, at the end of the procedure,* $\boldsymbol{u}_H = \mathbf{0}$.

Proof. For any variable in the algorithm, we add the superscript $^{(\text{new})}$ to mean its state at the end of PUSHDOWN; if there is no superscript, then it refers to the state at the start.

We show the claim for when H has two children H_1, H_2. Note that $\mathbf{\Delta}$ and z are not touched by PUSHDOWN, so we may ignore the term $c\mathbf{\Delta} z$ in Eq. (B.6). Then,

$$\sum_{H' \in \mathcal{T}} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'}^{(\text{new})}$$

$$= \mathbf{\Delta}^{(H)} \boldsymbol{u}_H^{(\text{new})} + \sum_{i=1,2} \mathbf{\Delta}^{(H_i)} \boldsymbol{u}_{H_i}^{(\text{new})} + \sum_{H' \in \mathcal{T}, H' \neq H, H_1, H_2} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'} \qquad \text{(expand terms)}$$

$$= \sum_{i=1,2} \mathbf{\Delta}^{(H_i)} (\boldsymbol{u}_{H_i} + \mathbf{\Delta}_{H_i} \boldsymbol{u}_H) + \sum_{H' \in \mathcal{T}, H' \neq H, H_1, H_2} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'} \qquad \text{(substitute values)}$$

$$= \sum_{i=1,2} \mathbf{\Delta}^{(H_i)} \mathbf{\Delta}_{H_i} \boldsymbol{u}_H + \sum_{H \in \mathcal{T}, H' \neq H} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'}$$

$$= \mathbf{\Delta}^{(H)} \boldsymbol{u}_H + \sum_{H \in \mathcal{T}, H' \neq H} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'} \qquad \text{(by Eq. (B.3))}$$

$$= \sum_{H' \in \mathcal{T}} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'},$$

so the implicit representation of $\boldsymbol{x}$ has not changed in value. $\qquad \square$

This claim can be generalized from $H \in \mathcal{T}$ to $\mathcal{H} \subseteq \mathcal{T}$; we omit the full details. Next, we show that the implicit representation of $\boldsymbol{x}$ by Eq. (B.6) is correctly maintained after reweight.

Claim B.11. *After the k-th call* REWEIGHT, *the value of $\boldsymbol{x}$ is unchanged, while the value of $\mathbf{\Delta}$ is updated to $\mathbf{\Delta}^{(k)}$ which is a function of $\boldsymbol{w}^{(k)}$.*

Proof. We begin by observing that if $H \notin \mathcal{P}_\mathcal{T}(\mathcal{H})$, then $\mathbf{\Delta}^{(H)(\text{new})} = \mathbf{\Delta}^{(H)}$ by definition, as there are no edges in $\mathcal{T}_H$ with updated operators.

At a high level, we traverse the subtree $\mathcal{P}_\mathcal{T}(\mathcal{H})$ three rounds and perform PUSHDOWN at every node. During the first round, we simply push down the current $\boldsymbol{u}_H$ values at each node H. By Claim B.10, we know this does not change the value of the implicit representation.

During the second round, we first initialize $\boldsymbol{u}_H \leftarrow c\boldsymbol{z}|_{F_H}$ at each node $H \in \mathcal{P}_\mathcal{T}(\mathcal{H})$, and then perform PUSHDOWN. Since PUSHDOWN does not affect the value of the implicit representation, we can use the initial change in $\boldsymbol{u}_H$ to determine the overall change in the implicit representation. Crucially, note that we perform PUSHDOWN using the old tree operator. So, the change in value of the implicit representation is given by

$$+c \sum_{H \in \mathcal{P}_\mathcal{T}(\mathcal{H})} \mathbf{\Delta}^{(H)}\boldsymbol{z}|_{F_H}.$$

After the second round of PUSHDOWN, we update the tree operator $\mathbf{\Delta}$ to $\mathbf{\Delta}^{(\text{new})}$. Note that $\mathbf{\Delta}^{(H)}$ changes if and only if $H \in \mathcal{P}_\mathcal{T}(\mathcal{H})$, and in this case, $\boldsymbol{u}_H = \mathbf{0}$. So, updating the tree operator at this point induces a change in the value of the implicit representation of

$$c\mathbf{\Delta}^{(\text{new})}\boldsymbol{z} - c\mathbf{\Delta}\boldsymbol{z} = c \sum_{H \in \mathcal{T}} \left(\mathbf{\Delta}^{(H)(\text{new})} - \mathbf{\Delta}^{(H)}\right)\boldsymbol{z}|_{F_H} = c \sum_{H \in \mathcal{P}_\mathcal{T}(\mathcal{H})} \left(\mathbf{\Delta}^{(H)(\text{new})} - \mathbf{\Delta}^{(H)}\right)\boldsymbol{z}|_{F_H}.$$

During the third round, we initialize $\boldsymbol{u}_H \leftarrow -c\boldsymbol{z}|_{F_H}$ at each node $H \in \mathcal{P}_\mathcal{T}(\mathcal{H})$ and perform PUSHDOWN. Similar to the first round, the change to the value of the implicit representation induced by this round is given by

$$-c \sum_{H \in \mathcal{P}_\mathcal{T}(\mathcal{H})} \mathbf{\Delta}^{(H)(\text{new})}\boldsymbol{z}|_{F_H}.$$

The sum of the changes from each of the three rounds is exactly $\mathbf{0}$, so we conclude the value of the implicit representation did not change. $\qquad\square$

Finally, we consider the other functions.

For INITIALIZE, we see that by substituting the values assigned during INITIALIZE and applying the definition from Eq. (B.2), we have

$$c\mathbf{\Delta}\boldsymbol{z} + \sum_{H \in \mathcal{T}} \mathbf{\Delta}^{(H)}\boldsymbol{u}_H = \boldsymbol{x}^{(\text{init})} + \mathbf{\Delta}\boldsymbol{z},$$

where $\mathbf{\Delta}$ is the initial $\mathbf{\Delta}^{(\text{init})}$ and $\boldsymbol{z}$ is the initial $\boldsymbol{z}^{(\text{init})}$, which is exactly how we want to initialize $\boldsymbol{x}$.

For MOVE, we see the value of $\boldsymbol{x}$ is incremented by $\mathbf{\Delta}^{(k)}(\boldsymbol{z}^{(k)} - \boldsymbol{z}^{(k-1)})$ after the step k. By definition of $\boldsymbol{z}$, we know $\boldsymbol{z}^{(k)} - \boldsymbol{z}^{(k-1)} = h^{(k)}\nabla^{(k)}\boldsymbol{v}^{(k)}$, so we conclude MOVE correctly makes the update $h^{(k)}\mathbf{\Delta}^{(k)}\nabla^{(k)}\boldsymbol{v}^{(k)}$.

For EXACT, we calculate the value of $\boldsymbol{x}$ explicitly by performing the computation $\sum_{H \in \mathcal{T}} \mathbf{\Delta}^{(H)}(\boldsymbol{u}_H + \boldsymbol{z}|_{F_H})$ using a sequence of PUSHDOWN's down the tree. The final answer $\boldsymbol{x}$ is stored in parts in the $\boldsymbol{u}_H$'s along the leaf nodes.

Note that by definition of the query complexity of $\mathbf{\Delta}$, PUSHDOWN uses $O(Q(1))$ time. The remaining runtimes are straightforward. $\qquad\square$

Finally, we combine Algorithm 3 and Algorithm 4 to get the overall data structure MAINTAINREP for maintaining $\boldsymbol{x}$ throughout SOLVE as given by Eq. (3.2). We omit the pseudocode implementation.

Proof of Theorem 3.7. We use one copy of INVERSETREEOP, which maintains $z \stackrel{\text{def}}{=} cz^{(\text{step})} + z^{(\text{sum})}$. We want to use TREEOP to maintain z which is given in two terms by INVERSETREEOP. To resolve this, we can simply use two copies of the data structure and track the two terms in z separately; then we correctly maintain x. During SOLVE, at step k, we first call REWEIGHT and MOVE in INVERSETREEOP, followed by REWEIGHT and MOVE in each copy of TREEOP. The runtimes follow in a straightforward manner. □

C Maintaining vector approximation

We include this section for completeness; all techniques are from [DGG+22].

Recall at every step of the IPM, we want to maintain approximate vectors $\overline{x}, \overline{s}$ so that

$$\left\| \mathbf{W}^{-1/2}(\overline{x} - x) \right\|_\infty \leq \delta \quad \text{and} \quad \left\| \mathbf{W}^{1/2}(\overline{s} - s) \right\|_\infty \leq \delta'$$

for some error tolerances δ and δ'.

In the previous section, we showed how to use MAINTAINREP to maintain x implicitly throughout SOLVE in the IPM. In this section, we give a data structure to efficiently maintain an approximate vector $\overline{x}$ to the x from MAINTAINREP, so that at every step,

$$\left\| \mathbf{W}^{-1/2} (\overline{x} - x) \right\|_\infty \leq \delta.$$

In the remainder of this section, we crucially assume that w is a function of $\overline{x}$ coordinate-wise, which is indeed satisfied by the RIPM framework.

Remark C.1. In our problem setting, we do not have full access to the exact vector x. The algorithms in the next two subsections however will refer to x for readability and modularity. We observe that access to x is limited to two types: accessing the JL-sketches of specific subvectors, and accessing exact coordinates and other specific subvectors of sufficiently small size.

In Appendix C.1, we reduce the problem of maintaining $\overline{x}$ to detecting coordinates of x with large changes. In Appendix C.2, we detect coordinates of x with large changes using a sampling technique on a constant-degree tree, where Johnson-Lindenstrauss sketches of subvectors of x are maintained at each node the tree. In Appendix C.3, we show how to compute and maintain the necessary collection of JL-sketches on the operator tree $\mathcal{T}$; in particular, we do this efficiently with only an implicit representation of x. Finally, we put the three parts together to prove Theorem 3.8.

For notational simplicity, we use $\mathbf{D} \stackrel{\text{def}}{=} \mathbf{W}^{-1/2}$. Recall we use the superscript $^{(k)}$ to denote the variable at the end of step k; that is, $\mathbf{D}^{(k)}$ and $x^{(k)}$ are the values of $\mathbf{D}$ and x at the end of step k. Step 0 is the initialization step.

C.1 Reduction to change detection

In this section, we show that in order to maintain an approximation $\overline{x}$ to some vector x, it suffices to detect coordinates of x that change a lot.

We make use of dyadic intervals. At step k, for each ℓ such that $k \equiv 0 \bmod 2^\ell$, we find the index set $I_\ell^{(k)}$ that contains all coordinates i of x such that $x_i^{(k)}$ changed significantly compared to $x_i^{(k-2^\ell)}$, that is, compared to 2^ℓ steps ago. Formally:

Definition C.2. At step k of the IPM, for each ℓ such that $k \equiv 0 \bmod 2^\ell$, we define

$$I_\ell^{(k)} \stackrel{\text{def}}{=} \{ i \in [n] : \mathbf{D}_{ii}^{(k)} \cdot |x_i^{(k)} - x_i^{(k-2^\ell)}| \geq \frac{\delta}{2 \lceil \log m \rceil},$$

$$\text{and } \overline{x}_i \text{ has not been updated after the } (k-2^\ell)\text{-th step}\}.$$

We show how to find the sets $I_\ell^{(k)}$ with high probability in the next section. Assuming the correct implementation, we have the following data structure for maintaining the desired approximation $\overline{\boldsymbol{x}}$:

Algorithm 6 Data structure AbstractMaintainApprox, Part 1

1: **data structure** AbstractMaintainApprox
2: **private : member**
3:    $\mathcal{T}$: constant-degree rooted tree with height η and m leaves   ▷ leaf i corresponds to $\boldsymbol{x}_i$
4:    $w \overset{\text{def}}{=} \Theta(\eta^2 \log(\frac{m}{\rho}))$: sketch dimension
5:    $\boldsymbol{\Phi} \sim \mathbf{N}(0, \frac{1}{w})^{w \times m}$: JL-sketch matrix
6:    $\delta > 0$: additive approximation error
7:    k: current step
8:    $\overline{\boldsymbol{x}} \in \mathbb{R}^m$: current valid approximate vector
9:    $\{\boldsymbol{x}^{(j)} \in \mathbb{R}^m\}_{j=0}^k$: list of previous inputs
10:   $\{\mathbf{D}^{(j)} \in \mathbb{R}^{m \times m}\}_{j=0}^k$: list of previous diagonal scaling matrices
11:
12: **procedure** Initialize($\mathcal{T}, \boldsymbol{x} \in \mathbb{R}^m, \mathbf{D} \in \mathbb{R}^{m \times m}_{>0}, \rho > 0, \delta > 0$)
13:   $\mathcal{T} \leftarrow \mathcal{T}$, $\delta \leftarrow \delta$, $k \leftarrow 0$
14:   $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}, \boldsymbol{x}^{(0)} \leftarrow \boldsymbol{x}, \mathbf{D}^{(0)} \leftarrow \mathbf{D}$
15:   sample $\boldsymbol{\Phi} \sim \mathbf{N}(0, \frac{1}{w})^{w \times m}$
16: **end procedure**
17:
18: **procedure** Update($\boldsymbol{x}^{(\text{new})} \in \mathbb{R}^m, \mathbf{D}^{(\text{new})} \in \mathbb{R}^{m \times m}_{>0}$)
19:   $k \leftarrow k + 1$, $\boldsymbol{x}^{(k)} \leftarrow \boldsymbol{x}^{(\text{new})}$, $\mathbf{D}^{(k)} \leftarrow \mathbf{D}^{(\text{new})}$
20: **end procedure**
21:
22: **procedure** Approximate
23:   $I \leftarrow \emptyset$
24:   **for all** $0 \le \ell < \lceil \log m \rceil$ such that $k \equiv 0 \bmod 2^\ell$ **do**
25:     $I_\ell^{(k)} \leftarrow$ FindLargeCoordinates(ℓ)
26:     $I \leftarrow I \cup I_\ell^{(k)}$
27:   **end for**
28:   **if** $k \equiv 0 \bmod 2^{\lceil \log m \rceil}$ **then**
29:     $I \leftarrow [m]$ ▷ Update $\overline{\boldsymbol{x}}$ in full every $2^{\lceil \log m \rceil}$ steps
30:   **end if**
31:   $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k)}$ for all $i \in I$
32:   **return** $\overline{\boldsymbol{x}}$
33: **end procedure**

Lemma C.3 (Approximate vector maintenance). *Suppose FindLargeCoordinates(ℓ) is a procedure in AbstractMaintainApprox that correctly computes the set $I_\ell^{(k)}$ at the k-th step. Then the deterministic data structure AbstractMaintainApprox in Algorithm 6 maintains an approximation $\overline{\boldsymbol{x}}$ of $\boldsymbol{x}$ with the following procedures:*

- *Initialize($\mathcal{T}, \boldsymbol{x}, \mathbf{D}, \rho > 0, \delta > 0$): Initialize the data structure at step 0 with tree $\mathcal{T}$, initial vector $\boldsymbol{x}$, initial diagonal scaling matrix $\mathbf{D}$, target additive approximation error δ, and success probability $1 - \rho$.*

- UPDATE($\boldsymbol{x}^{(\text{new})}, \mathbf{D}^{(\text{new})}$): *Increment the step counter and update vector $\boldsymbol{x}$ and diagonal scaling matrix $\mathbf{D}$.*

- APPROXIMATE: *Output a vector $\overline{\boldsymbol{x}}$ such that $\|\mathbf{D}(\boldsymbol{x} - \overline{\boldsymbol{x}})\|_\infty \le \delta$ for the latest $\boldsymbol{x}$ and $\mathbf{D}$ with probability at least $1 - \rho$.*

At step k, the procedure UPDATE is called, followed by APPROXIMATE. Suppose $\|\mathbf{D}^{(k)}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)})\|_2 \le \beta$ for all steps k, and $\mathbf{D}$ is a function of $\overline{\boldsymbol{x}}$ coordinate-wise. Then, for each $\ell \ge 0$, the data structure updates $O(2^{2\ell}(\beta/\delta)^2 \log^2 m)$ coordinates of $\overline{\boldsymbol{x}}$ every 2^ℓ steps.

Proof of Lemma C.3. The failure case arises from FINDLARGECOORDINATES. Assuming FIND-LARGECOORDINATES returns the correct set of coordinates, we prove the correctness of APPROXIMATE.

Fix some coordinate $i \in [m]$ and fix some step k. Suppose the latest update to $\overline{\boldsymbol{x}}_i$ is $\overline{\boldsymbol{x}}_i \leftarrow \boldsymbol{x}_i^{(k')}$ at step k'. So $\mathbf{D}_{ii}^{(d)}$ is the same for all $k \ge d > k'$, and i is not in the set $I_\ell^{(d)}$ returned by FINDLARGECOORDINATES for all $k \ge d > k'$. Since we set $\overline{\boldsymbol{x}} \leftarrow \boldsymbol{x}$ every $2^{\lceil \log m \rceil}$ steps by Line 29, we know $k - 2^{\lceil \log m \rceil} \le k' < k$. Using dyadic intervals, we can define a sequence $k_0, k_1, \ldots, k_s$ with $s \le 2 \lceil \log m \rceil$, where $k' = k_0 < k_1 < k_2 < \cdots < k_s = k$, each $k_{j+1} - k_j$ is a power of 2, and $(k_{j+1} - k_j) \mid k_{j+1}$. Hence, we have

$$\boldsymbol{x}_i^{(k)} - \overline{\boldsymbol{x}}_i^{(k)} = \boldsymbol{x}_i^{(k_s)} - \overline{\boldsymbol{x}}_i^{(k_0)} = \boldsymbol{x}_i^{(k_s)} - \boldsymbol{x}_i^{(k_0)} = \sum_{j=0}^{s-1} \left(\boldsymbol{x}_i^{(k_{j+1})} - \boldsymbol{x}_i^{(k_j)} \right).$$

We know that $\mathbf{D}_{ii}^{(d)}$ is the same for all $k \ge d > k'$. By the guarantees of FINDLARGECOORDINATES, we have

$$\mathbf{D}_{ii}^{(k)} \cdot |\boldsymbol{x}_i^{(k_{j+1})} - \boldsymbol{x}_i^{(k_j)}| = \mathbf{D}_{ii}^{(k_{j+1})} \cdot |\boldsymbol{x}_i^{(k_{j+1})} - \boldsymbol{x}_i^{(k_j)}| \le \frac{\delta}{2 \lceil \log m \rceil}$$

for all $0 \le j < s$. Summing over all $j = 0, 1, \ldots, s - 1$ gives

$$\mathbf{D}_{ii}^{(k)} \cdot |\boldsymbol{x}_i^{(k)} - \overline{\boldsymbol{x}}_i^{(k)}| \le \delta.$$

Hence, we have $\|\mathbf{D}(\boldsymbol{x} - \overline{\boldsymbol{x}})\|_\infty \le \delta$.

Fix some ℓ with $k \equiv 0 \bmod 2^\ell$. We bound the number of coordinates in $I_\ell^{(k)}$. For any $i \in I_\ell^{(k)}$, we know $\mathbf{D}_{ii}^{(j)} = \mathbf{D}_{ii}^{(k)}$ for all $j > k - 2^\ell$ because $\overline{\boldsymbol{x}}_i$ did not change in the meanwhile. By definition of $I_\ell^{(k)}$, we have

$$\mathbf{D}_{ii}^{(k)} \cdot \sum_{j=k-2^\ell}^{k-1} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}| \ge \mathbf{D}_{ii}^{(k)} \cdot |\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)}| \ge \frac{\delta}{2 \lceil \log m \rceil}.$$

Using $\mathbf{D}_{ii}^{(j)} = \mathbf{D}_{ii}^{(k)}$ for all $j > k - 2^\ell$ again, the above inequality yields

$$\frac{\delta}{2 \lceil \log m \rceil} \le \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|$$

$$\le \sqrt{2^\ell \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)2} |\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|^2}. \qquad \text{(by Cauchy-Schwarz)}$$

45

Squaring and summing over all $i \in I_\ell^{(k)}$ gives

$$\Omega\left(\frac{2^{-\ell}\delta^2}{\log^2 m}\right)|I_\ell^{(k)}| \leq \sum_{i \in I_\ell^{(k)}} \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)2}|\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|^2$$

$$\leq \sum_{i=1}^{m} \sum_{j=k-2^\ell}^{k-1} \mathbf{D}_{ii}^{(j+1)2}|\boldsymbol{x}_i^{(j+1)} - \boldsymbol{x}_i^{(j)}|^2$$

$$\leq 2^\ell \beta^2,$$

where we use $\|\mathbf{D}^{(j+1)}(\boldsymbol{x}^{(j+1)} - \boldsymbol{x}^{(j)})\|_2 \leq \beta$ at the end. Hence, we have

$$|I_\ell^{(k)}| = O(2^{2\ell}(\beta/\delta)^2 \log^2 m).$$

In other words, for each $\ell \geq 0$, we update $|I_\ell^{(k)}|$-many coordinates of $\overline{\boldsymbol{x}}$ at step k when $k \equiv 0 \mod 2^\ell$. So we conclude that for each $\ell \geq 0$, we update $O(2^{2\ell}(\beta/\delta)^2 \log^2 m)$-many coordinates of $\overline{\boldsymbol{x}}$ every 2^ℓ steps. $\qquad\square$

C.2 From change detection to sketch maintenance

Now we discuss the implementation of FindLargeCoordinates(ℓ) to find the set $I_\ell^{(k)}$ in Line 25 of Algorithm 6. We accomplish this by repeatedly sampling a coordinate i with probability proportional to $\left(\mathbf{D}_{ii}^{(k)}(\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)})\right)^2$, among all coordinates i where $\overline{\boldsymbol{x}}_i$ has not been updated since 2^ℓ steps ago. With high probability, we can find all indices in $I_\ell^{(k)}$ in this way efficiently. To implement the sampling procedure, we make use of a data structure based on segment trees [CLRS09] along with sketching based on the Johnson-Lindenstrauss lemma.

Formally, we define the vector $\boldsymbol{q} \in \mathbb{R}^m$ where $\boldsymbol{q}_i \overset{\text{def}}{=} \mathbf{D}_{ii}^{(k)}(\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_i^{(k-2^\ell)})$ if $\overline{\boldsymbol{x}}_i$ has not been updated after the $(k - 2^\ell)$-th step, and $\boldsymbol{q}_i = 0$ otherwise. Our goal is precisely to find all large coordinates of $\boldsymbol{q}$.

Let $\mathcal{T}$ be a constant-degree rooted tree with m leaves, where leaf i represents coordinate $\boldsymbol{q}_i$, which we call a *sampling tree*. For each node $u \in \mathcal{T}$, we define $E(u) \subseteq [m]$ to be the set of indices of leaves in the subtree rooted at u. We make a random descent down $\mathcal{T}$, in order to sample a coordinate i with probability proportional to $\boldsymbol{q}_i^2$. At a node u, for each child u' of u, the total probability of the leaves under u' is given precisely by $\left\|\boldsymbol{q}|_{E(u')}\right\|_2^2$. We can estimate this by the Johnson-Lindenstrauss lemma using a sketching matrix $\boldsymbol{\Phi}$. Then we randomly move from u down to child u' with probability proportional to the estimated value. To tolerate the estimation error, when reaching some leaf node representing coordinate i, we accept with probability proportional to the ratio between the exact probability of i and the estimated probability of i. If i is rejected, we repeat the process from the root again independently.

Lemma C.4. *Assume that $\|\mathbf{D}^{(k+1)}(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)})\|_2 \leq \beta$ for all IPM steps k. Let $\rho < 1$ be any given failure probability, and let $M_\ell \overset{\text{def}}{=} \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho))$ be the number of samples Algorithm 6 takes. Then with probability $\geq 1 - \rho$, during the k-th call of Approximate, Algorithm 6 finds the set $I_\ell^{(k)}$ correctly. Furthermore, the while-loop in Line 41 happens only $O(1)$ times in expectation per sample.*

Algorithm 6 Data structure ABSTRACTMAINTAINAPPROX, Part 2

34: **procedure** FINDLARGECOORDINATES(ℓ)

35:　　$\triangleright$ $\overline{\mathbf{D}}$: diagonal matrix such that

$$\overline{\mathbf{D}}_{ii} = \begin{cases} \mathbf{D}_{ii}^{(k)} & \text{if } \overline{\boldsymbol{x}}_i \text{ has not been updated after the } (k - 2^\ell)\text{-th step} \\ 0 & \text{otherwise.} \end{cases}$$

36:　　$\triangleright$ $\boldsymbol{q} \stackrel{\text{def}}{=} \overline{\mathbf{D}}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^\ell)})$　　　　　　　　　　　　　$\triangleright$ vector to sample coordinates from

37:

38:　　$I \leftarrow \emptyset$　　　　　　　　　　　　　　　　　　　　　$\triangleright$ set of candidate coordinates

39:　　**for** $M_\ell \stackrel{\text{def}}{=} \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(m/\rho))$ iterations **do**

40:　　　　$\triangleright$ Sample coordinate i of $\boldsymbol{q}$ w.p. proportional to $\boldsymbol{q}_i^2$ by random descent down $\mathcal{T}$ to a leaf

41:　　　　**while true do**

42:　　　　　　$u \leftarrow \text{root}(\mathcal{T}), p_u \leftarrow 1$

43:　　　　　　**while** u is not a leaf node **do**

44:　　　　　　　　Sample a child u' of u with probability

$$\mathbf{P}(u \rightarrow u') \stackrel{\text{def}}{=} \frac{\|\boldsymbol{\Phi}_{E(u')}\boldsymbol{q}\|_2^2}{\sum_{\text{child } u'' \text{ of } u} \|\boldsymbol{\Phi}_{E(u'')}\boldsymbol{q}\|_2^2}$$

$\triangleright$ let $\boldsymbol{\Phi}_{E(u)} \stackrel{\text{def}}{=} \boldsymbol{\Phi}\mathbf{I}_{E(u)}$ for each node u

45:　　　　　　　　$p_u \leftarrow p_u \cdot \mathbf{P}(u \rightarrow u')$

46:　　　　　　　　$u \leftarrow u'$

47:　　　　　　**end while**

48:　　　　　　**break** with probability $p_{\text{accept}} \stackrel{\text{def}}{=} \left\|\boldsymbol{q}|_{E(u)}\right\|^2 / (2 \cdot p_u \cdot \|\boldsymbol{\Phi}\boldsymbol{q}\|_2^2)$

49:　　　　**end while**

50:　　　　$I \leftarrow I \cup E(u)$

51:　　**end for**

52:　　**return** $\{i \in I \,:\, \boldsymbol{q}_i \geq \frac{\delta}{2\lceil \log m \rceil}\}$.

53: **end procedure**

Proof. The proof is similar to Lemma 6.17 in [DLY21b]. We include it for completeness. For a set S of indices, let $\mathbf{I}_S$ be the $m \times m$ diagonal matrix that is one on S and zero otherwise.

We first prove that Line 48 breaks with probability at least $\frac{1}{4}$. By the choice of w, Johnson–Lindenstrauss lemma shows that $\|\mathbf{\Phi}_{E(u)}\boldsymbol{q}\|_2^2 = (1 \pm \frac{1}{9\eta})\|\mathbf{I}_{E(u)}\boldsymbol{q}\|_2^2$ for all $u \in \mathcal{T}$ with probability at least $1 - \rho$. Therefore, the probability we move from a node u to its child node u' is given by

$$\mathbf{P}(u \to u') = \left(1 \pm \frac{1}{3\eta}\right) \frac{\|\mathbf{I}_{E(u')}\boldsymbol{q}\|_2^2}{\sum_{u'' \text{ is a child of } u} \|\mathbf{I}_{E(u'')}\boldsymbol{q}\|_2^2} = \left(1 \pm \frac{1}{3\eta}\right) \frac{\|\mathbf{I}_{E(u')}\boldsymbol{q}\|_2^2}{\|\mathbf{I}_{E(u)}\boldsymbol{q}\|_2^2}.$$

Hence, the probability the walk ends at a leaf $u \in \mathcal{T}$ is given by

$$p_u = \left(1 \pm \frac{1}{3\eta}\right)^\eta \frac{\|\mathbf{I}_u\boldsymbol{q}\|_2^2}{\|\boldsymbol{q}\|_2^2} = (1 \pm \frac{1}{3\eta})^\eta \frac{\|\boldsymbol{q}|_{E(u)}\|^2}{\|\boldsymbol{q}\|_2^2}.$$

Now, p_{accept} on Line 48 is at least

$$p_{\text{accept}} = \frac{\|\boldsymbol{q}|_{E(u)}\|^2}{2 \cdot p_u \cdot \|\mathbf{\Phi}\boldsymbol{q}\|_2^2} \geq \frac{\|\boldsymbol{q}|_{E(u)}\|^2}{2 \cdot (1 + \frac{1}{3\eta})^\eta \frac{\|\boldsymbol{q}|_{E(u)}\|^2}{\|\boldsymbol{q}\|_2^2} \cdot \|\mathbf{\Phi}\boldsymbol{q}\|_2^2} \geq \frac{\|\boldsymbol{q}\|_2^2}{2 \cdot (1 + \frac{1}{3\eta})^\eta \|\mathbf{\Phi}\boldsymbol{q}\|_2^2} \geq \frac{1}{4}.$$

On the other hand, we have that $p_{\text{accept}} \leq \frac{\|\boldsymbol{q}\|_2^2}{2(1 - \frac{1}{3\eta})^\eta \|\mathbf{\Phi}\boldsymbol{q}\|_2^2} < 1$ and hence this is a valid probability.

Next, we note that u is accepted on Line 48 with probability

$$p_{\text{accept}} p_u = \frac{\|\boldsymbol{q}|_{E(u)}\|^2}{2 \cdot \|\mathbf{\Phi}\boldsymbol{q}\|_2^2}.$$

Since $\|\mathbf{\Phi}\boldsymbol{q}\|_2^2$ remains the same in all iterations, this probability is proportional to $\|\boldsymbol{q}|_{E(u)}\|^2$. Since the algorithm repeats when u is rejected, on Line 50, u is chosen with probability exactly $\|\boldsymbol{q}|_{E(u)}\|^2 / \|\boldsymbol{q}\|^2$.

Now, we want to show the output set is exactly $\{i \in [n] : |\boldsymbol{q}_i| \geq \frac{\delta}{2\lceil \log m \rceil}\}$. Let S denote the set of indices where $\overline{\boldsymbol{x}}$ did not update between the $(k - 2^\ell)$-th step and the current k-th step. Then

$$\begin{aligned}
\|\boldsymbol{q}\|_2 &= \|\mathbf{I}_S\mathbf{D}^{(k)}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^\ell)})\|_2 \\
&\leq \sum_{i=k-2^\ell}^{k-1} \|\mathbf{I}_S\mathbf{D}^{(k)}(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})\|_2 \\
&= \sum_{i=k-2^\ell}^{k-1} \|\mathbf{I}_S\mathbf{D}^{(i+1)}(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})\|_2 \\
&\leq \sum_{i=k-2^\ell}^{k-1} \|\mathbf{D}^{(i+1)}(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})\|_2 \\
&\leq 2^\ell \beta,
\end{aligned}$$

where we used $\mathbf{I}_S\mathbf{D}^{(i+1)} = \mathbf{I}_S\mathbf{D}^{(k)}$, because $\overline{\boldsymbol{x}}_i$ changes whenever $\mathbf{D}_{ii}$ changes at a step. Hence, each leaf u is sampled with probability at least $\|\boldsymbol{q}|_{E(u)}\|^2 / (2^\ell \beta)^2$. If $|\boldsymbol{q}_i| \geq \frac{\delta}{2\lceil \log m \rceil}$, and $i \in E(u)$ for a leaf node u, then the coordinate i is not in I with probability at most

$$\left(1 - \frac{\|\boldsymbol{q}|_{E(u)}\|^2}{(2^\ell \beta)^2}\right)^{M_\ell} \leq \left(1 - \frac{1}{2^{2\ell+2}(\beta/\delta)^2 \lceil \log m \rceil^2}\right)^{M_\ell} \leq \frac{\rho}{m},$$

by our choice of M_ℓ. Hence, all i with $|\boldsymbol{q}_i| \geq \frac{\delta}{2\lceil \log m \rceil}$ lies in I with probability at least $1 - \rho$. This proves that the output set is exactly $I_\ell^{(k)}$ with probability at least $1 - \rho$. $\qquad\square$

Remark C.5. In Algorithm 6, we only need to compute $\|\boldsymbol{\Phi}_{E(u)}\boldsymbol{q}\|_2^2$ for $O(M_\ell)$ many nodes $u \in \mathcal{T}$. Furthermore, the randomness of the sketch is not leaked and we can use the same random sketch $\boldsymbol{\Phi}$ throughout the algorithm. This allows us to efficiently maintain $\boldsymbol{\Phi}_{E(u)}\boldsymbol{q}$ for each $u \in \mathcal{T}$ throughout the IPM.

C.3 Sketch maintenance

In FINDLARGECOORDINATES in the previous subsection, we assumed the existence of a constant degree sampling tree $\mathcal{T}$, and for the dynamic vector $\boldsymbol{q}$, the ability to access $\boldsymbol{\Phi}_{E(u)}\boldsymbol{q}$ at each node $u \in \mathcal{T}$ and $\boldsymbol{q}|_{E(u)}$ at each leaf node u.

In this section, we consider when the required sampling tree is the operator tree $\mathcal{T}$ supporting a tree operator $\boldsymbol{\Delta}$, and the vector $\boldsymbol{q}$ is $\boldsymbol{x} \overset{\text{def}}{=} \boldsymbol{\Delta z} + \sum_{H \in \mathcal{T}} \boldsymbol{\Delta}^{(H)}\boldsymbol{u}_H$, where each of $\boldsymbol{\Delta}, \boldsymbol{z}$ and the $\boldsymbol{u}_H$'s undergo changes at every IPM step. We present a data structure that implements two features efficiently on $\mathcal{T}$:

- access $\boldsymbol{x}|_{E(H)}$ at every leaf node H,

- access $\boldsymbol{\Phi}_{E(H)}\boldsymbol{x}$ at every node H, where $\boldsymbol{\Phi}_{E(H)}$ is $\boldsymbol{\Phi}$ restricted to columns given by $E(H)$.

Lemma C.6. *Let $\mathcal{T}$ be a constant degree rooted tree with height η supporting tree operator $\boldsymbol{\Delta}$ with query complexity Q. Let $w = \Theta(\eta^2 \log(\frac{m}{\rho}))$ be as defined in Algorithm 6, and let $\boldsymbol{\Phi} \in \mathbb{R}^{w \times m}$ be a JL-sketch matrix. Then MAINTAINSKETCH (Algorithm 7) is a data structure that maintains $\boldsymbol{\Phi x}$, where $\boldsymbol{x}$ is implicitly represented by*

$$\boldsymbol{x} \overset{\text{def}}{=} \boldsymbol{\Delta z} + \sum_{H \in \mathcal{T}} \boldsymbol{\Delta}^{(H)}\boldsymbol{u}_H.$$

The data structure supports the following procedures:

- INITIALIZE*(operator tree $\mathcal{T}$, implicit $\boldsymbol{x}$): Initialize the data structure and compute the initial sketches in $O(Q(wm))$ time.*

- UPDATE*($\mathcal{H} \subseteq \mathcal{T}$): Update all the necessary sketches in $O(w \cdot Q(\eta|\mathcal{H}|))$ time, where $\mathcal{H}$ is the set of all nodes H where $\boldsymbol{u}_H$ or $\boldsymbol{\Delta}_H$ changed.*

- ESTIMATE*($H \in \mathcal{T}$): Return $\boldsymbol{\Phi}_{E(H)}\boldsymbol{x}$.*

- QUERY*($H \in \mathcal{T}$): Return $\boldsymbol{x}|_{E(H)}$.*

If we call QUERY on N nodes, the total runtime is $O(Q(w\eta N))$.

If we call ESTIMATE along a sampling path (by which we mean starting at the root, calling estimate at both children of a node, and then recursively descending to one child until reaching a leaf), and then we call QUERY on the resulting leaf, and we repeat this N times with no updates during the process, then the total runtime of these calls is $O(Q(w\eta N))$.

We note that $\boldsymbol{\Delta z} = \sum_{H \in \mathcal{T}} \boldsymbol{\Delta}^{(H)}\boldsymbol{z}|_{F_H}$. For simplicity, it suffices to give the algorithm for sketching the simpler $\boldsymbol{x} \overset{\text{def}}{=} \sum_{H \in \mathcal{T}} \boldsymbol{\Delta}^{(H)}\boldsymbol{u}_H$.

Algorithm 7 Data structure for maintaining $\mathbf{\Phi x}$, Part 1

1: **data structure** MAINTAINSKETCH
2: **private : member**
3:     $\mathcal{T}$: rooted constant degree tree, where at every node H, there is
4:         $\mathbf{S}^{(H)} \in \mathbb{R}^{w \times |F_H \cup \partial H|}$: sketched subtree operator $\mathbf{\Phi \Delta}^{(H)}$
5:         $\boldsymbol{t}^{(H)} \in \mathbb{R}^w$: sketched vector $\mathbf{\Phi} \sum_{H' \in \mathcal{T}_H} \mathbf{\Delta}^{(H')} \boldsymbol{u}_{H'}$
6:     $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$: JL-sketch matrix
7:     $\mathbf{\Delta} \in \mathbb{R}^{m \times n}$: dynamic tree operator on $\mathcal{T}$
8:     $\boldsymbol{u}_H$ at every $H \in \mathcal{T}$: dynamic vectors
9:
10: **procedure** INITIALIZE(tree $\mathcal{T}$, $\mathbf{\Phi} \in \mathbb{R}^{w \times m}$, tree operator $\mathbf{\Delta}$, $\boldsymbol{u}_H$ for each $H \in \mathcal{T}$)
11:     $\mathbf{\Phi} \leftarrow \mathbf{\Phi}, \mathcal{T} \leftarrow \mathcal{T}, \mathbf{\Delta} \leftarrow \mathbf{\Delta}, \boldsymbol{u}_H \leftarrow \boldsymbol{u}_H$ for each $H \in \mathcal{T}$
12:     $\mathbf{S}^{(H)} \leftarrow \mathbf{0}$, $\boldsymbol{t}^{(H)} \leftarrow \mathbf{0}$ for each $H \in \mathcal{T}$
13:     UPDATE($V(\mathcal{T})$)
14: **end procedure**
15:
16: **procedure** UPDATE($\mathcal{H} \stackrel{\text{def}}{=}$ set of nodes admitting implicit representation changes)
17:     **for** $H \in \mathcal{P}_\mathcal{T}(\mathcal{H})$ going up the tree level by level **do**
18:         $\mathbf{S}^{(H)} \leftarrow \sum_{\text{child } D \text{ of } H} \mathbf{S}^{(D)} \mathbf{\Delta}_D$
19:         $\boldsymbol{t}^{(H)} \leftarrow \mathbf{S}^{(H)} \boldsymbol{u}_H + \sum_{\text{child } D \text{ of } H} \boldsymbol{t}^{(D)}$
20:     **end for**
21: **end procedure**
22:
23: **procedure** SUMANCESTORS($H \in \mathcal{T}$)
24:     **if** UPDATE has not been called since the last call to SUMANCESTORS(H) **then**
25:         **return** the result of the last SUMANCESTORS(H)
26:     **end if**
27:     **if** H is the root **then return 0**
28:     **end if**
29:     **return** $\mathbf{\Delta}_H(\boldsymbol{u}_P + \text{SUMANCESTORS}(P))$ ▷ P is the parent of H
30: **end procedure**
31:
32: **procedure** ESTIMATE($H \in \mathcal{T}$)
33:     Let $\boldsymbol{y}$ be the result of SUMANCESTORS(H)
34:     **return** $\mathbf{S}^{(H)} \boldsymbol{y} + \boldsymbol{t}^{(H)}$
35: **end procedure**
36:
37: **procedure** QUERY(leaf $H \in \mathcal{T}$)
38:     **return** $\boldsymbol{u}_H + \text{SUMANCESTORS}(H)$
39: **end procedure**

Proof. Let us consider the correctness of the data structure, starting with the helper function SumAncestors. We implement it using recursion and memoization as it is crucial for bounding subsequent runtimes.

Claim C.7. *SumAncestors($H \in \mathcal{T}$) returns $\sum_{ancestor\ A\ of\ H} \boldsymbol{\Delta}_{H \leftarrow A} \boldsymbol{u}_A$.*

Proof. At the root, there are no ancestors, hence we return the zero matrix. When H is not the root, suppose P is the parent of H. Then we can recursively write

$$\sum_{ancestor\ A\ of\ H} \boldsymbol{\Delta}_{H \leftarrow A} \boldsymbol{u}_A = \boldsymbol{\Delta}_H \left(\boldsymbol{u}_P + \sum_{ancestor\ A\ of\ P} \boldsymbol{\Delta}_{P \leftarrow A} \boldsymbol{u}_A \right).$$

The procedure implements the right hand side, and is therefore correct. $\qquad \square$

Assuming we correctly maintain $\mathbf{S}^{(H)} \stackrel{\text{def}}{=} \boldsymbol{\Phi} \boldsymbol{\Delta}^{(H)}$ and $\boldsymbol{t}^{(H)} \stackrel{\text{def}}{=} \boldsymbol{\Phi} \sum_{H' \in \mathcal{T}_H} \boldsymbol{\Delta}^{(H')} \boldsymbol{u}_{H'}$ at every node H, Estimate and Query return the correct answers by the tree operator decomposition given in Lemma B.3.

For Update, note that if a node H is not in $\mathcal{H}$ and it has no descendants in $\mathcal{H}$, then by definition, the sketches at H are not changed. Hence, it suffices to update the sketches only at all nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$. We update the nodes from the bottom of $\mathcal{T}$ upwards, so that when we are at a node H, all the sketches at its descendant nodes are correct. Therefore, by definition, the sketches at H is also correct.

Now we consider the runtimes:

Initialize: It sets the sketches to $\mathbf{0}$ in $O(wm)$ time, and then calls Update to update the sketches everywhere on $\mathcal{T}$. By the correctness runtime of Update, this step is correct and runs in $\widetilde{O}(Q(wm))$ time.

Update(set of nodes $\mathcal{H}$ admitting implicit representation changes): First note that $|\mathcal{P}_{\mathcal{T}}(\mathcal{H})| \leq \eta|\mathcal{H}|$. For each node $H \in \mathcal{H}$ with children D_1, D_2, Line 18 multiplies each row of $\mathbf{S}^{(D_1)}$ with $\boldsymbol{\Delta}_{(D_1,H)}$, each row of $\mathbf{S}^{(D_2)}$ with $\boldsymbol{\Delta}_{D_2}$, and sums the results. Summing over w-many rows and over all nodes in $\mathcal{P}_{\mathcal{T}}(\mathcal{H})$, we see the total runtime of Line 18 is $O(Q(w\eta|\mathcal{H}|))$.

Line 19 multiply each row of $\mathbf{S}^{(H)}$ with a vector and then performs a constant number of additions of w-length vectors. Since $\mathbf{S}^{(H)}$ is computed for all $H \in \mathcal{P}_{\mathcal{T}}(\mathcal{H})$ in $O(Q(w\eta|\mathcal{H}|))$ total time, this must also be a bound on their number of total non-zero entries. Since each $\mathbf{S}^{(H)}$ is used once in Line 19 for a matrix-vector multiplication, the total runtime of Line 19 is $O(Q(w\eta|\mathcal{H}|))$.

All other lines are not bottlenecks.

Overall Estimate and Query time along N sampling paths: We show that if we call Estimate along N sampling paths each from the root to a leaf, and we call Query on the leaves, the total cost is $O(Q(w\eta N))$:

Suppose the set of nodes visited is given by $\mathcal{H}$, then $|\mathcal{H}| \leq \eta N$. Since there is no update, and Estimate is called for a node only after it is called for its parent, we know that SumAncestors(H) is called exactly once for each $H \in \mathcal{H}$. Each SumAncestor(H) multiplies a unique edge operator $\boldsymbol{\Delta}_{(H,P)}$ with a vector. Hence, the total runtime of SumAncestors is $Q(|\mathcal{H}|)$.

Finally, each Query applies a leaf operator to the output of a unique SumAncestors call, so the overall runtime is certainly bounded by $O(Q(|\mathcal{H}|))$. Similarly, each Estimate multiplies $\mathbf{S}^{(H)}$ with the output of a unique SumAncestors call. This can be computed as w-many vectors each multiplied with the SumAncestors output. Then two vectors of length w are added. Summing over all nodes in $\mathcal{H}$, the overall runtime is $O(Q(w|\mathcal{H}|)) = O(Q(w\eta N))$.

QUERY time on N leaves: Since this is a subset of the work described above, the runtime must also be bounded by $O(Q(w\eta N))$.

$\square$

C.4 Proof of Theorem 3.8

We combine the previous three subsections for the overall approximation procedure. It is essentially ABSTRACTMAINTAINAPPROX in Algorithm 6, with the abstractions replaced by a data structure implementation. We omit the pseudocode and simply describe the functions.

Theorem 3.8 (Approximate vector maintenance with tree operator [DGG+22])**.** *Let $0 < \rho < 1$ be a failure probability. Suppose $\mathbf{\Delta} \in \mathbb{R}^{m \times n}$ is a tree operator with query complexity Q and supported on a constant-degree tree $\mathcal{T}$ with height η. There is a randomized data structure MAIN-TAINAPPROX that takes as input the dynamic weights $\boldsymbol{w}$ and the dynamic $\boldsymbol{x}$ implicitly maintained according to Theorem 3.7 at every step, and explicitly maintains the approximation $\overline{\boldsymbol{x}}$ to $\boldsymbol{x}$ satisfying $\left\|\mathbf{W}^{-1/2}(\boldsymbol{x} - \overline{\boldsymbol{x}})\right\|_\infty \leq \delta$ at every step with probability $1 - \rho$.*

Suppose $\|\mathbf{W}^{(k)-1/2}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)})\|_2 \leq \beta$ for all steps k. Furthermore, suppose $\boldsymbol{w}$ is a function of $\overline{\boldsymbol{x}}$ coordinate-wise. Then, for each $\ell \geq 0$, $\overline{\boldsymbol{x}}$ admits $2^{2\ell}$ coordinate changes every 2^ℓ steps. Over N total steps, the total cost of the data structure is

$$\widetilde{O}(\eta^3 (\beta/\delta)^2 \log^3(mN/\rho)) \left(Q(m) + \sum_{k=1}^{N} Q(S^{(k)}) + \sum_{\ell=0}^{\log N} \frac{N}{2^\ell} \cdot Q(2^{2\ell}) \right), \tag{3.3}$$

where $S^{(k)}$ is the number of nodes H where $\mathbf{\Delta}_H$ or $\boldsymbol{u}_H$ in the implicit representation of $\boldsymbol{x}$ changed at step k.

Proof. We apply Lemma C.3 using $\boldsymbol{x}$ maintained by MAINTAINREP and $\mathbf{D} \stackrel{\text{def}}{=} \mathbf{W}^{-1/2}$ from SOLVE. We create $O(\log m)$ copies of MAINTAINSKETCH (Lemma C.6), so that for each $0 \leq \ell \leq O(\log m)$, we have one copy $\texttt{sketch}_{\ell,x}$ which maintains sketches of $\mathbf{\Phi}\overline{\mathbf{D}}\boldsymbol{x}^{(k)}$ at step k, and one copy $\texttt{sketch}_\ell$ which maintains sketches of $\mathbf{\Phi}\overline{\mathbf{D}}\boldsymbol{x}^{(k-2^\ell)}$ at step $k \geq 2^\ell$, where $\overline{\mathbf{D}}$ is defined so $\overline{\mathbf{D}}_{i,i} = \mathbf{D}_{i,i}$ if $\overline{\boldsymbol{x}}_i$ has not been updated after the $k - 2^\ell$-th step, and $\overline{\mathbf{D}}_{i,i} = 0$ otherwise (as needed in Algorithm 6). Note that $\overline{\mathbf{D}}$ can be absorbed into the tree operator in the implicit representation of $\boldsymbol{x}$, so Lemma C.6 does indeed apply.

To access sketches of the vector $\boldsymbol{q} \stackrel{\text{def}}{=} \overline{\mathbf{D}}(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-2^\ell)})$ as needed in FINDLARGECOORDINATES in Algorithm 6, we can simply access the corresponding sketch in $\texttt{sketch}_{\ell,x}$ and $\texttt{sketch}_\ell$, and then take the difference.

We now describe each procedure in words, and then prove their correctness and runtime.

INITIALIZE$(\mathcal{T}, \boldsymbol{x}, \mathbf{D}, \rho, \delta)$: This procedure implements the initialization of ABSTRACTMAINTAINAP-PROX to approximate the dynamic vector $\boldsymbol{x}$ which is given implicitly. The initialization steps described in Algorithm 6 takes $O(wm)$ time. Then, we initialize the $O(\log m)$ copies of MAINTAINSKETCH in $O(Q(wm)\log m)$ time by Lemma C.6.

UPDATE$(\boldsymbol{x}^{(\text{new})}, \mathbf{D}^{(\text{new})})$: To implement UPDATE, it suffices to update all the sketching data structures. Let us fix ℓ, and consider the update time for $\texttt{sketch}_{\ell,x}$ and $\texttt{sketch}_\ell$.

Lemma C.3 shows that throughout SOLVE, there are $O(2^{2\ell}(\beta/\delta)^2 \log^2 m)$-many coordinate updates to $\overline{\boldsymbol{x}}$ every 2^ℓ steps. Since $\mathbf{D}$ is a function of $\overline{\boldsymbol{x}}$ coordinate-wise, $\overline{\boldsymbol{x}}_i = \boldsymbol{x}_i^{(k-1)}$ for all i where $\mathbf{D}_{ii}^{(k)} \neq \mathbf{D}_{ii}^{(k-1)}$ by Line 31. The diagonal matrix $\overline{\mathbf{D}}$ is the same as $\mathbf{D}$, except $\overline{\mathbf{D}}_{ii}$ is temporarily

52

zeroed out for 2^ℓ steps after $\overline{\boldsymbol{x}}_i$ changes at a step. So, the overall number of coordinate changes to $\overline{\mathbf{D}}$ is $O(2^{2\ell})$-many every 2^ℓ steps.

Let $S^{(k)}$ denote the number of nodes H where $\boldsymbol{\Delta}_H$ or $\boldsymbol{u}_H$ in the implicit representation of $\boldsymbol{x}$ changed at step k. Additionally, since the sketching data structures maintain some variant of $\overline{\mathbf{D}}\boldsymbol{x}$ (where $\overline{\mathbf{D}}$ is viewed as absorbed in the tree operator), every coordinate change in $\overline{\mathbf{D}}$ implies an edge operator update. Now we apply Lemma C.6 to conclude that the total time for all UPDATE calls for $\texttt{sketch}_{\ell,x}$ and $\texttt{sketch}_\ell$ over N steps is:

$$O(1) \cdot \left(\sum_{k=1}^{N} Q\left(w\eta S^{(k)}\right) + \frac{N}{2^\ell} \cdot Q(w\eta \cdot 2^{2\ell}) \right) \le O(w\eta) \cdot \left(\sum_{k=1}^{N} Q(S^{(k)}) + \frac{N}{2^\ell} \cdot Q(2^{2\ell}) \right).$$

We then sum this over all ℓ to get the total update time for the sketching data structures.

APPROXIMATE: There are two operations to be implemented in the subroutine FINDLARGECO-ORDINATES(ℓ): Accessing $\boldsymbol{\Phi}_{E(u)}\boldsymbol{q}$ at a node u, and accessing $\boldsymbol{q}|_{E(u)}$ at a leaf node u. For the first, we call $\texttt{sketch}_{\ell,x}.\text{ESTIMATE}(u) - \texttt{sketch}_\ell.\text{ESTIMATE}(u)$. For the second, we call $\texttt{sketch}_{\ell,x}.\text{QUERY}(u) - \texttt{sketch}_\ell.\text{QUERY}(u)$.

To set $\overline{\boldsymbol{x}}_i$ as $\boldsymbol{x}_i^{(k)}$ for a single coordinate at step k as needed in Line 31, we find the leaf node H containing the edge e, and call $\texttt{sketch}_{0,x}.\text{QUERY}(H)$. This returns the sub-vector $\boldsymbol{x}^{(k)}|_{E(H)}$, from which we can extract $\boldsymbol{x}_i^{(k)}$ and set $\overline{\boldsymbol{x}}_i$ to be the value. This line is not a bottleneck in the runtime.

We compute the total runtime over N APPROXIMATE calls. For every $\ell \ge 0$, we call FIND-LARGECOORDINATES(ℓ) once every 2^ℓ steps, for a total of $N/2^\ell$ calls. In a single call, $M_\ell \stackrel{\text{def}}{=} \Theta(2^{2\ell}(\beta/\delta)^2 \log^2 m \log(mN/\rho))$ sampling paths are explored in the $\texttt{sketch}_\ell$ and $\texttt{sketch}_{\ell,x}$ data structures by Lemma C.4, where a sampling path correspond to one iteration of the while-loop. This takes a total of $O(Q(w\eta M_\ell))$ time by Lemma C.6. Therefore, for every fixed ℓ, the total time for all FINDLARGECOORDINATES(ℓ) calls is

$$\frac{N}{2^\ell} \cdot O\left(Q(w\eta M_\ell)\right).$$

The total time for all LARGECOORDINATES calls is obtained by summing over all values of $\ell = 0, \ldots, \log N$. To achieve overall failure probability at most ρ, it suffices to set the failure probability of each call to be $O(\rho/N)$. $\qquad\square$

We sum up the initialization time, update and approximate time for all values of $\ell = 0, \ldots, \log N$ and over N total steps of SOLVE, to get the overall runtime of the data structure:

$$\widetilde{O}(Q(wm)) + O(w\eta) \sum_{k=1}^{N} Q(S^{(k)}) + O(w\eta) \sum_{\ell=0}^{\log N} \frac{N}{2^\ell}\left(Q(2^{2\ell}) + Q(M_\ell)\right)$$

$$= \widetilde{O}(\eta^3(\beta/\delta)^2 \log^3(mN/\rho)) \left(Q(m) + \sum_{k=1}^{N} Q(S^{(k)}) + \sum_{\ell=0}^{\log N} \frac{N}{2^\ell} \cdot Q(2^{2\ell}) \right).$$

53