2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS) | 979-8-3503-1894-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/FOCS57990.2023.00036

Faster High Accuracy Multi-Commodity Flow from Single-Commodity Techniques

Jan van den Brand
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
vdbrand@gatech.edu

Daniel J. Zhang
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
dzhang381@gatech.edu

***Abstract*—Since the development of efficient linear program solvers in the 80s, all major improvements for solving multi-commodity flows to high accuracy came from improvements to general linear program solvers. This differs from the single commodity problem (e.g. maximum flow) where all recent improvements also rely on graph specific techniques such as graph decompositions or the Laplacian paradigm.**

This phenomenon sparked research to understand why these graph techniques are unlikely to help for multi-commodity flow. [Kyng and Zhang FOCS'17] reduced solving multi-commodity Laplacians to general linear systems and [Ding, Kyng, and Zhang ICALP'22] showed that general linear programs can be reduced to 2-commodity flow. However, the reductions create sparse graph instances, so improvement to multi-commodity flows on denser graphs might exist.

We show that one can indeed speed up multi-commodity flow algorithms on non-sparse graphs using graph techniques from single-commodity flow algorithms. This is the first improvement to high accuracy multi-commodity flow algorithms that does not just stem from improvements to general linear program solvers. In particular, using graph data structures from recent min-cost flow algorithm by [Brand, Lee, Liu, Saranurak, Sidford, Song, and Wang STOC'21] based on the celebrated expander decomposition framework, we show that 2-commodity flow on an n-vertex m-edge graph can be solved deterministically in $\widetilde{O}(\sqrt{m}n^{\omega-1/2})$ time for current bounds on fast matrix multiplication $\omega \approx 2.372$, improving upon the previous fastest algorithms with $\widetilde{O}(m^{\omega})$ [Cohen, Lee, and Song STOC'19] and $\widetilde{O}(\sqrt{m}n^2)$ [Kapoor and Vaidya'96] time complexity. For general k commodities, our algorithm runs in $\widetilde{O}(k^{2.5}\sqrt{m}n^{\omega-1/2})$ time.

***Index Terms*—G.1.6.g Linear programming, E.1 Data Structures, G.2.2.a Graph algorithms, G.2.2.d Network problems**

I. Introduction

The multi-commodity flow problem arises when more than one commodity must be routed in a shared network. These types of problems occur often in traffic-, distribution-, and communication-systems, as well as in network and VLSI design. Formally, we are given a (possibly directed) graph $G = (V, E)$ with edge capacities $u \in \mathbb{R}^E_{>0}$ and multiple source-sink pairs (one for each commodity) and the task is to route a flow for each source-sink pair, such that the sum of the flows do not exceed the edge capacities. In the maximum through-put variant, the task is to maximize the sum of flows routed between the pairs, whereas in the minimum cost variant, each source-sink pair has a demand that must be satisfied and has costs assigned to the edges. The task here is to route the flow such that it minimizes the cost.

There exist various algorithms that solve these problems to $(1+\epsilon)$ accuracy [2]–[9]. On undirected graphs, the problem can even be solved in nearly-linear time [7]–[10]. However, all these algorithms are either low-accuracy (i.e. their complexity has a $\mathrm{poly}(1/\epsilon)$ dependence), or they consider relaxations such as flows without capacities minimizing a mixed $\ell_{q,p}$-norm for constant p, q [11]. To obtain high-accuracy solutions (with $\mathrm{polylog}(1/\epsilon)$ time complexity dependence), the multi-commodity flow problems can be phrased as a linear program (LP) and thus solved via efficient linear programming solvers. For instance, on sparse graphs, the current fastest algorithm for multi-commodity flow are the recent matrix multiplication time linear program solvers [12]–[15], running in $\widetilde{O}((km)^{\omega})$ time[1] on m-edge graphs with k commodities. Before the advent of polynomial time LP solvers, Hu developed a polynomial time algorithm for multi-commodity flow on undirected graphs [19]. But since linear programs became efficient to solve, all improvements for solving multi-commodity flow to high accuracy stem from improvements to LP solvers [12]–[15], [20]–[26]. In particular, no improvements were made via graph specific techniques.

This differs a lot from the developments that occurred for single-commodity flows (i.e. the classical maximum flow and min-cost flow problems). While recent developments rely on continuous optimization methods similar to LP solvers, they do use many powerful graph specific techniques [11], [25], [27]–[38]. For example, while the central path method reduces solving general LPs to solving a sequence of linear systems, applying this method to single-commodity flows results in a sequence of Laplacian systems. Such a system can be solved in near-linear time using Laplacian solvers [39]. This powerful framework of combining continuous optimization methods with Laplacian solvers is often referred to as "Laplacian paradigm". Additionally, even the central path method itself can be accelerated for the special case when the LP is a single-

Full version [1] available at https://arxiv.org/abs/2304.12992.

[1]The matrix exponent m^{ω} is the number of operations required to multiply two $m \times m$ matrices. Currently $\omega \leq 2.372$ [16], [17]. We write $\omega(\cdot,\cdot,\cdot)$ for the rectangular matrix multiplication exponent [18]. Here $n^{\omega(a,b,c)}$ is the complexity for multiplying an $n^a \times n^b$ matrix by $n^b \times n^c$ matrix.

commodity flow instance [28]–[33]. Every recent improvement for single-commodity flows stems from combining continuous optimization techniques with graph specific tools. Yet somehow, none of these tools translate from single-commodity to multi-commodity problems. Even when extending to just two commodities, these tools have not found any application in the high accuracy regime. This has raised the following question:

Can single-commodity techniques be used to improve high-accuracy multi-commodity flow algorithms?

This sparked research to better understand why so far these graph techniques could not help for the high accuracy multi-commodity flow problem. Kyng and Zhang showed that an equivalent of the Laplacian paradigm for multi-commodity flows is unlikely [40]. They showed that any linear system can be reduced to a 2-commodity Laplacian. So despite near-linear time solvers for Laplacian systems, such solvers are unlikely to exist for 2-commodity Laplacians unless we can solve all linear systems in near linear time. Thus it is unlikely that we can speed up the central path method in a similar way as the Laplacian paradigm did for single-commodity flow. While this is only an argument against one specific algorithmic approach, there is also evidence that 2-commodity flows in general are hard. Itai showed that any linear program can be reduced to exact 2-commodity flow [41], and recently Ding, Kyng, Zhang [42] extended this result to the high-accuracy regime. Thus any algorithm that solves 2-commodity flow to high accuracy can also be used to solve general linear programs to high accuracy. The reduction by [42] produces a 2-commodity flow instance on a sparse graph where the number of edges corresponds to the number of non-zeros in the LP. So, if one can get an algorithm using graph techniques that is competitive with general purpose linear program solvers on the 2-commodity flow problem on sparse graphs, then this algorithm would also be competitive on general sparse LPs, despite the LP not having any kind of graph structure. With these insights from [40]–[42], it seems that improvements to multi-commodity flow via graph techniques are impossible. However, since these reductions produce sparse graphs, single-commodity techniques might still lead to improvements on dense graphs.

In this work we show that this is indeed possible. We obtain the first improvement to the multi-commodity flow problem via graph specific single-commodity techniques since the development of efficient LP solvers in the 80s. This is the first improvement that does not just stem from improvements to general linear program solvers.

Our algorithm combines algebraic methods from general purpose linear program solvers [12], [14], [15] with the dynamic expander decomposition framework from dynamic graph theory [43]–[48]. This graph technique was also used in all recent improvements to single-commodity flows (either directly or inside a Laplacian solver) [11], [25], [27]–[38].

A. High Accuracy Results

As mentioned before, multi-commodity flows can be solved to high-accuracy using linear program solvers. For an n-node m-edge graph with k commodities, the respective linear program has km variables and $kn + m$ equality-constraints. Using state-of-the-art linear program solvers, the following complexities can be achieved for solving multi-commodity flow with polynomially bounded edge capacities $U \leq \operatorname{poly}(n)$ and polynomially bounded error $\epsilon = 1/\operatorname{poly}(n)$: $\widetilde{O}((km)^{\omega})$ time [12], $\widetilde{O}(k^{2.5}\sqrt{m}n^2)$ time [24], $\widetilde{O}((kn+m)^{2.5})$ time [25], [26][2].

Using expander decomposition, we improve these complexities on graphs that are at least slightly dense. This is the first improvement to multi-commodity flow that stems from single-commodity techniques. All previous improvements [12], [20]–[26] stem from improvements to general LP solvers.

A conceptual insight of our work is that graph techniques can lead to faster multi-commodity flow algorithms despite increasing evidence to the contrary. Our improvements are possible, because previous impossibility results hold only for sparse graphs. We show that dense multi-commodity flow is more combinatorial in nature than previously thought.

Theorem I.1. *For any $0 \leq \mu \leq 1$, given a k-commodity instance on graph $G = (V, E)$ with integer edge capacities $u \in [0, U]^E$ and source sink pairs $(s_1, t_1), ..., (s_k, t_k) \in V \times V$, we can solve maximum through-put commodity flow* deterministically *up to additive error $\epsilon > 0$ in time*

$$\widetilde{O}\left(\left(n^{\omega-1/2} + n^{\omega(1,1,\mu)-\mu/2} + n^{1+\mu} + n\log\frac{U}{\epsilon}\right) k^{2.5}\sqrt{m}\log\frac{U}{\epsilon}\right).$$

For current bounds on $\omega(\cdot,\cdot,\cdot)$ [16]–[18], and polynomially bounded u, ϵ^{-1}, this is

$$\widetilde{O}(k^{2.5}\sqrt{m}n^{\omega-1/2}).$$

[40]–[42] show that multi-commodity flow is already hard for two commodities, so let us for now focus on $k = 2$. Our new algorithm improves upon previous work whenever the graph is at least slightly dense. On sparse graphs, we match the $\widetilde{O}(m^{\omega})$-time[3] algorithm by [12]–[15] up to the extra $\sqrt{mn}\log(U/\epsilon)$ term, though this term does not matter for polynomially bounded U and $1/\epsilon$. On dense graphs, we improve upon the $\widetilde{O}(\sqrt{m}n^2 \log U/\epsilon)$-time algorithm by [24].

We remark that the reduction of Ding, Kyng, Zhang [42] creates 2-commodity flow instances on sparse graphs. So our algorithm does not improve upon general purpose LP solvers, but using their reduction, our algorithm can solve LPs within

[2]For single-commodity flows, Lee and Sidford's algorithm [25] runs in $\sqrt{n}$ iterations. However, for multi-commodity flow the number of iterations of [25] is $\sqrt{m + kn} > \sqrt{m}$, see e.g. [26, Section 7.4].

[3]Like our results, these algorithms run in $\widetilde{O}(m^{\omega})$ time for current bounds on matrix multiplication. In particular, [12]–[14] have the same $\widetilde{O}(n^{\omega-1/2} + n^{\omega(1,1,\mu)-\mu/2} + n^{1+\mu})$ dependency as our Theorem I.1.

the same $\widetilde{O}(m^{\omega}\log\delta^{-1})$-time complexity[4] as other state-of-the-art LP solvers [12]–[15] on sparse LPs. In particular, if we were able to improve upon [12]–[15] for 2-commodity flow on sparse graphs, then our algorithm would improve general LP solvers as well. This gives strong evidence that improvements via single-commodity techniques might only be possible for at least slightly dense graphs.

Ding, Kyng and Zhang [42] posed the open problem whether their reduction from LP to 2-commodity flow could be modified to maintain the "shape" of the LP; If a tall sparse LP is given (n rows, m columns, $n \le m \le n^2$) can one construct a multi-commodity flow instance on $\widetilde{O}(m)$ edges and $\widetilde{O}(n)$ vertices? If such a reduction exists, then our multi-commodity flow algorithm would beat state-of-the-art LP solvers when $m \le n^{1.254}$. So either there is exciting opportunity to improve general LP solvers using graph theoretic expander decomposition techniques, or our result can be interpreted as evidence that such a reduction is not possible.

Our result can also solve the minimum cost variant of k-commodity flow.

Theorem I.2. *For any $0 \le \mu \le 1$, given a k-commodity instance on graph $G = (V, E)$ with integer edge capacities $u \in [0, U]^E$, integer costs $c_1, ..., c_k \in [-C, C]^E$ and integer demands $\sigma_1, ..., \sigma_k \in [-U, U]^V$, we can solve minimum-cost commodity flow* deterministically *up to additive error $\epsilon > 0$ in time*

$$\widetilde{O}\left(\left(n^{\omega-1/2} + n^{\omega(1,1,\mu)-\mu/2} + n^{1+\mu} + n\log\frac{CU}{\epsilon}\right) k^{2.5}\sqrt{m}\log\frac{CU}{\epsilon}\right).$$

For current bounds on ω [16]–[18] and polynomially bounded ϵ^{-1}, C, U, this is

$$\widetilde{O}(k^{2.5}\sqrt{m}n^{\omega-1/2}).$$

The returned flows $f_1, ..., f_k \in \mathbb{R}^E_{\ge 0}$ satisfy the demands approximately with (here $\mathbf{B} \in \mathbb{R}^{E\times V}$ is the incidence matrix)

$$\|\mathbf{B}^\top f_i - \sigma_i\|_1 \le \epsilon \textit{ for } i = 1, ..., k.$$

We remark that the small error w.r.t. the demands also occurs when solving multi-commodity flow with high accuracy LP solvers such as [12]–[15].

B. Techniques

Here we summarize the techniques used by our multi-commodity flow algorithm and how they relate to previous general purpose LP solvers. A more detailed outline of the techniques is given in Section II. Our algorithm is based on general purpose LP solvers which we accelerate for the multi-commodity flow problem using single-commodity flow techniques. To highlight our ideas we start with a quick summary on how general purpose LP solvers were accelerated for single-commodity flow and then explain how a similar improvement can be made for multi-commodity flow.

[4]These LP solvers return an approximate solution with additive δ-error.

In the line of work of [34], [35], [49], it was shown that a linear program with constraint matrix of dimension $m \times n$ (with $m \ge n$) can be solved in $\widetilde{O}(mn + n^{2.5})$ time.[5] This algorithm uses data structures that work for general matrices. If the linear program is a single-commodity flow problem, then the additional graph structure of the involved matrices allows for faster data structures using graph techniques. This resulted in an $\widetilde{O}(m + n^{1.5})$ time algorithm (i.e. saved a factor n) for single-commodity flow [35]. A natural question is if a similar speed up is possible when extending from single to multi-commodity flow. For this, we will focus on each term (mn and $n^{2.5}$) separately and discuss how they can be improved for multi-commodity flows.

The two terms mn and $n^{2.5}$ generally come from the following two factors: (i) How many iterations does the algorithm take, and the time required to solve a linear system in each iteration (this is where the $\widetilde{O}(n^{2.5})$ term came from), (ii) How much time is required for a certain "heavy hitter problem" which we describe next. (This is where the $\widetilde{O}(mn)$ term came from)

a) Heavy Hitter: LP solvers based on the central path method must compute a matrix vector product of the form $\mathbf{A}h$ in each iteration. Here $\mathbf{A}$ is a fixed matrix and h is some new vector given in each iteration. It was shown in [34], [49] that it suffices to just detect large entries of this product, a task they refer to as "heavy hitters".

In [49], this heavy hitter task requires $\widetilde{O}(mn)$ total time over all iterations and [34] shows that for the special case where $\mathbf{A}$ is an edge-vertex-incidence matrix, it can be solved in just $\widetilde{O}(m)$ total time over all iterations.

For our application of k-commodity flow, this matrix $\mathbf{A}$ will be of the following form: The rows of $\mathbf{A}$ are given by Kronecker-products. For each edge $(u, v) \in E$ we have $k + 1$ rows in $\mathbf{A}$ given by $\mathbf{W}^{(u,v)} \otimes (e_u^\top - e_v^\top)$ for some matrix $\mathbf{W}^{(u,v)} \in \mathbb{R}^{(k+1)\times(k+1)}_{>0}$ and $e_u, e_v \in \mathbb{R}^n$ standard unit vectors. So matrix $\mathbf{A}$ can be seen as an incidence matrix where instead of real edge weights, we now have $(k + 1)^2$-dimensional edge weights $\mathbf{W}^{(u,v)}$. This weight matrix $\mathbf{W}^{(u,v)}$ has a very specific structure given by the following definition. We show that the heavy hitter task is efficiently solvable on matrices of the following form.

Definition I.3. *Given graph $G = (V, E)$ and vectors $h^{(u,v)} \in \mathbb{R}^{k+1}_{>0}$ for each $(u, v) \in E$. We call the matrix $\mathbf{M} \in \mathbb{R}^{(k+1)m\times(k+1)n}$ with rows given by*

$$\left((\mathbf{H}^{(u,v)})^{1/2}\left(\mathbf{I} - \frac{\mathbf{1}_{k+1}h^\top}{\|h\|_1}\right)\right) \otimes (e_u^\top - e_v^\top) \in \mathbb{R}^{k+1\times(k+1)n}$$

a k-commodity incidence matrix.[6]

Here $\mathbf{H}^{(u,v)}$ is diagonal matrix with entries of $h^{(u,v)}$ on the diagonal, $\mathbf{I}$ is identity matrix, $\mathbf{1}_{k+1}$ is the $k + 1$ dimensional all 1-vector, and $\otimes$ is the Kronecker-product.

[5]For simplicity we hide $\log U/\epsilon$ terms (for single and multi-commodity flow) and $\log\delta^{-1}$ terms (for general LP solvers) in this overview.

[6]The notion of a multi-commodity incidence matrix was defined in a different way in [40], however $\mathbf{M}^\top\mathbf{M}$ is a multi-commodity Laplacian as defined in [40].

Unlike heavy hitters from previous work [34], [35], [49] where the matrix was fixed (though scaling of rows was admissible), the matrix from Definition I.3 is allowed to change over time because we allow to update entries of any $d^{(u,v)}$.

Additionally, note that the matrix from Definition I.3 has $O(k)$ non-zero entries per row. Solving the heavy hitter problem on general sparse matrices is one of the major open problems in linear program solving. In particular, if heavy hitter could be solved in $O(km)$ total time on a matrix with m rows and k non-zero entries per row for $1 \leq k \leq n$, one would obtain a nearly linear time solver for sparse and sufficiently tall linear programs [35]. So far there are only nearly linear time algorithms for dense linear programs [35], [49]. Our heavy hitter for Definition I.3 is the first result for a structured special case of sparse matrices with > 2 non-zero entries per row. Unfortunately, the improvements are only for multi-commodity flow and do not extend to general sparse linear programs, because of the additional structure imposed on $\mathbf{W}^{(u,v)} = \left((\mathbf{H}^{(u,v)})^{1/2}\left(\mathbf{I} - \frac{\mathbf{1}_{k+1}h^\top}{\|h\|_1}\right)\right)$. It is interesting to see though, that the heavy hitter problem for multi-commodity incidence matrices seems to be easier than the heavy hitter problem for general sparse matrices. For comparison: solving multi-commodity Laplacian systems (such as $\mathbf{M}^\top\mathbf{M}x = b$ for $\mathbf{M}$ from Definition I.3) are as hard as general linear systems [40]. And solving linear programs are as hard as sparse multi-commodity flows [41], [42]. Yet, heavy hitter for multi-commodity incidence matrices (Definition I.3) do not seem to be as hard as general sparse heavy hitters. In particular, using the special structure of $\mathbf{W}^{(u,v)}$, we can reduce the heavy hitter task on multi-commodity incidence matrices to the heavy hitter task on classical incidence matrices.

The hardness of solving linear systems in multi-commodity Laplacians such as $\mathbf{M}^\top\mathbf{M}$ brings us to the next section.

b) Solving Linear Systems: In addition of detecting the heavy hitters of some matrix vector product $\mathbf{A}h$, modern linear program solvers must also solve a linear system in each iteration.

We already stated that tall LPs of size $m \times n$ ($m \gg n$) could be solved in $\widetilde{O}(mn + n^{2.5})$ time. Here the $n^{2.5}$ term comes from the LP solver internally running $\widetilde{O}(\sqrt{n})$ iterations and in each iteration solving a linear system (i.e $O(n^2)$ time to multiply a vector with some $n \times n$ sized matrix inverse). In the case of single-commodity flow, this linear system is given by a Laplacian matrix and by using Laplacian solvers the system the $O(n^2)$ cost can be reduced to just $\widetilde{O}(n)$ time.

A natural idea is to use same techniques to speed up multi-commodity flow. However, there are two problems for multi commodity flow that result in a slow down. First, the number of iterations is larger: while [25], [25], [34], [35] consider tall LPs of size $m \times n$, multi-commodity LPs have dimension $(km+m) \times (kn+m)$, so the number of iterations is $\sqrt{kn+m}$. Especially for small k there is no substantial speed up from using [25] compared to, let's say, the classic log-barrier method with $\sqrt{km}$ iterations. For simplicity we will use the simpler but slightly slower $\sqrt{km}$ iteration algorithm.

The second and bigger issue is that for multi-commodity flow, we cannot use a Laplacian solver to solve the linear system. [40] shows that for multi-commodity flow, solving this linear system is as hard as solving a general linear system, even for just $k = 2$ commodities. So it is unlikely that there is a fast solver using graph techniques similar to Laplacian solvers.

With these two issues, one would achieve only $\widetilde{O}(\sqrt{m}n^2)$ time for 2-commodity flow ($\sqrt{m}$ iterations and n^2 for solving the linear system), as was already achieved in [24].

Our algorithm also cannot use graph techniques to speed up solving this linear system. However, we show that because of the sparsity and structure of the linear program, we can solve the linear system in $\widetilde{O}(k^2 n^{\omega-1/2})$ amortized time per iteration instead of $O(k^2n^2)$. This uses projection maintenance techniques from [12]–[15], [50] which, when applied directly to the linear system, would solve it in $\widetilde{O}((km)^{\omega-1/2})$ time per iteration. The improvement from $\widetilde{O}((km)^{\omega-1/2})$ to $\widetilde{O}(k^2n^{\omega-1/2})$ comes from two observations: (i) The sparsity of the LP implies that any change to the linear system from one iteration to the next is very sparse. (ii) Most constraints of the linear program are of form $\sum_{i=1}^k f_i \leq u$ (i.e. the capacity constraints of k-commodity flow). These constraints induce a certain structure in the linear system that can be exploited to reduce the size onto a smaller $kn \times kn$ sized linear system for k-commodity flow. In particular, one can transform the linear system to be of shape $\mathbf{M}^\top\mathbf{M}x = b$ for matrix $\mathbf{M}$ being a multi-commodity incidence matrix as in Definition I.3. Property (ii) was previously used by [24] to accelerate Vaidya's and Karmarkar's LP solvers [20], [22] for k-commodity flow, which is why they achieved an $\widetilde{O}(k^{2.5}\sqrt{m}n^2)$ time algorithm. We now use this idea together with (i) to accelerate the projection maintenance/inverse maintenance techniques from [12]–[15], [50]. The next difficulty for obtaining a faster algorithm comes from projecting the smaller kn dimensional solution back onto $O(km)$ dimensional space without paying $O(km)$ time per iteration (and thus leading to an at best $\widetilde{O}((km)^{1.5}) \leq \widetilde{O}(k^{1.5}n^3)$ time algorithm, since we have $\widetilde{O}(\sqrt{km})$ iterations). We show that projecting the solution back onto $O(km)$ dimensional space is solved by the "heavy hitter" problem on multi-commodity incidence matrices.

C. Related work

While we focus on solving multi-commodity flow to high accuracy, related work also studies low accuracy regimes [2]–[9]. On undirected graphs, the "maximum concurrent flow" (i.e. maximizing F s.t. each commodity has at least F units of flow) for k commodities can be reduced to 2^{k-1} single commodity flows [10], [51]. Maximum concurrent flow can be solved by our algorithms by adding edges $t_i \to s_i$ of capacity F and negative cost for each of the k source/sink pairs, then binary searching for the maximum F. The maximum concurrent flow problem motivated the study of multi-commodity flow *without* capacities minimizing a mixed $\ell_{q,p}$-norm. For $q \to 1, p \to \infty$ this is equivalent to maximum concurrent flow. Chen and Ye [10] show that this problem is solvable to high-accuracy in almost-linear time for $1 \leq p \leq 2 \leq q$ with $p = \widetilde{O}(1)$, $1/(q-1) = O(1)$ by reducing it to single-commodity flow. Like our work, [10] also provides new directions for efficient high-accuracy multi-commodity flow algorithms, circumventing lower bound arguments [41], [42].

a) Linear programs and generalizations: Previous efficient multi-commodity flow algorithms all rely on linear programming techniques such as interior point methods which are then combined with data structures to reduce the time per iteration. This technique of combining interior point methods with data structures is commonly used in various LP solvers [12]–[15], [20], [22], [23], [26], [35], [49] and also finds application in generalization such as semi-definite programs [52]–[54] and general convex optimization via cutting planes [55], [56]. An especially powerful data structure framework is the inverse maintenance [12]–[15], [26], [50], [57], [58].

Converting high accuracy solutions for multi-commodity flow to exact solutions can be done with the same techniques as converting high accuracy solution of general LPs to exact solutions, since multi-commodity flow is just a special case. [59] presents a scheme that takes $O(m)$ approximate solutions and converts them to an exact solution. Another technique is to run the algorithm for small enough $\epsilon > 0$ such that one can round to the nearest corner of the polytope representing the feasible solution space, see e.g. [21], [24], [25] for a discussion on this.

b) Dynamic Expander Decomposition: The improvements of our algorithm use data structures based on dynamic expander decompositions [43], [48]. This is a fundamental tool in the area of dynamic graph algorithms, previously used to maintain properties of dynamic graphs such as connectivity [45]–[47], [60]–[62], distances [63]–[67], approximate flows [60], [62], [65], [66], or sparsifiers [44].

Recently, dynamic expander decomposition has been used to develop data structure that accelerate single-commodity flow algorithms (i.e. max flow, min-cost flow but also bipartite matching, transshipment etc.) [11], [35], [37], [38]. This line of work recently culminated in an almost-linear time algorithm for single-commodity flow [11]. Another important tool for single-commodity flow algorithms [25], [27]–[33], [35] are Laplacian system solvers which run in nearly-linear time and have an extensive history of research [39], [68]–[76].

D. Organization

We start by giving a technical overview in Section II. There we present the main technical ideas, sketch the proof for our algorithm, and all tools and required data structures. The full proof is given in the full version [1].

E. Preliminaries

We use $\widetilde{O}$ to hide $\operatorname{polylog}(m, k)$ factors and $\widehat{O}$ to hide sub-polynomial $m^{o(1)}$ factors. We write $[n]$ for the interval $\{1, ..., n\}$. Given two vectors $u, v \in \mathbb{R}^m$, all operations are defined elementwise, e.g. uv is an elementwise product so $(uv)_i = u_i \cdot v_i$ for all $i \in [m]$. Likewise $(u/v)_i = u_i/v_i$ for all $i \in [m]$. We also extend all scalar operations to be elementwise on vectors so, for example, $(\sqrt{v})_i = \sqrt{v_i}$ for all $i \in [m]$. For an $\alpha \in \mathbb{R}$ we write $v + \alpha$ for adding α to each entry of v, so $(v + \alpha)_i = v_i + \alpha$ for all $i \in [m]$.

For vectors $x, s, d, g \in \mathbb{R}^m$ we write $\mathbf{X}, \mathbf{S}, \mathbf{D}, \mathbf{G}$ for the $m \times m$ diagonal matrices with the respective vectors on the diagonal, e.g., $\mathbf{X}_{i,i} = x_i$ for all $i \in [m]$.

Given $\epsilon > 0$, $\alpha, \beta \in \mathbb{R}$, we write $\alpha \approx_\epsilon \beta$ when $\exp(-\epsilon)\alpha \leq \beta \leq \exp(\epsilon)\alpha$. For small ϵ we have that $\exp(\pm\epsilon)$ is roughly $(1 \pm \epsilon)$ so the $\approx_\epsilon$ notation can be considered to reflect $(1 \pm \epsilon)$ approximations. The definition via the exponential function allows for the following transitive property: if $\alpha \approx_\epsilon \beta$ and $\beta \approx_\delta \gamma$ then $\alpha \approx_{\epsilon+\delta} \gamma$. We also extend the notation to vectors, so $u \approx_\epsilon v$ means $u_i \approx_\epsilon v_i$ for all i.

II. Technical Overview

Our algorithm for solving multi-commodity flow comes from improving linear program solvers by developing data structures. One of these data structure problems is the "heavy hitter problem". Here one must preprocess a matrix $\mathbf{M}$ and then support queries where for any given vector v, one must detect all large entries of the product $\mathbf{M}v$. Solving the heavy hitter problem on sparse matrices is an open problem. We make the first contribution for very structured sparse matrices as given in Definition I.3 which we refer to as multi-commodity incidence matrices. Since our data structure relies on the specific structure of the matrix, we must first prove that we indeed have a structure as in Definition I.3. The first two subsections of this overview will recap how linear programs are solved via the robust central path method which will also prove the specific structure of the heavy hitter task.

In the first subsection II-A, we define the LP structure of k-commodity flow. We then give an overview for how to use the robust central path method to solve linear programs efficiently in subsection II-B. There we also sketch why the $\widetilde{O}((km)^\omega)$ complexity[7] from [12]–[15] can be reduced to $\widetilde{O}(k^{2.5}(\sqrt{m}n^{\omega-1/2} + m^{3/2}))$ for k-commodity flow. Finally, subsection II-C describes how to improve $\widetilde{O}(k^{2.5}(\sqrt{m}n^{\omega-1/2} + m^{3/2}))$ to $\widetilde{O}(k^{2.5}\sqrt{m}n^{\omega-1/2})$ time, by developing an efficient heavy hitter data structure on multi-commodity incidence matrices. This last step uses techniques from single commodity flows as we show that heavy hitters on multi-commodity incidence matrices can be reduced to heavy hitters on classical incidence matrices, which in [34] was solved using dynamic expander decomposition.

A. k-Commodity LP

Given a graph $G = (V, E)$, let $\mathbf{B} \in \mathbb{R}^{E \times V}$ be the edge-vertex incidence matrix. Let $u \in \mathbb{R}^E_{>0}$ be edge capacities, $c_1, ..., c_k \in \mathbb{R}^E$ be the costs and $\sigma_1, ...\sigma_k \in \mathbb{R}^V$ be the demand vectors for the k commodities. Then, we can write the min-cost variant of k-commodity flow as the LP

$$\min_{x_1,\ldots,x_k} \sum_{i=1}^{k} c_i^\top x_i \text{ subject to}$$

[7]For simplicity we hide $\log(U/\epsilon)$ factors in this overview.

$$\mathbf{B}^\top x_i = \sigma_i \text{ for all } 1 \le i \le k,$$
$$\sum_{i=1}^k x_i \le u, \quad x_1, ..., x_k \ge 0$$

where $x_1, ..., x_k \in \mathbb{R}^E$ are the flows for the k commodities. This can be written as an LP in standard form by introducing a slack variable $x_{k+1} \ge 0$ with $\sum_{i=1}^{k+1} x_i = u$ and writing $x = (x_1, ..., x_{k+1}) \in \mathbb{R}^{(k+1)E}$. The primal LP $\mathcal{P}$ (and its dual $\mathcal{D}$) are given by

$$\begin{array}{rlrl} (\mathcal{P}) & \min_x c^\top x & (\mathcal{D}) & \max_y \sigma^\top y \\ & \mathcal{B}^\top x = \sigma, & & \mathcal{B}y + s = c, \\ & x \ge 0 & & s \ge 0 \end{array}$$

$$\text{where } \mathcal{B} := \begin{bmatrix} \mathbf{B} & & 0 & \mathbf{I} \\ & \ddots & & \vdots \\ 0 & & \mathbf{B} & \mathbf{I} \\ 0 & ... & 0 & \mathbf{I} \end{bmatrix}$$
$$\sigma := (\sigma_1, ..., \sigma_k, u) \in \mathbb{R}^{kV+E}$$
$$c := (c_1, ..., c_k, 0) \in \mathbb{R}^{(k+1)E}$$

Using [12], [14], [15] to solve this LP would take $\widetilde{O}((km)^\omega)$ time since $\mathcal{B}$ is of size $(k+1)m \times (kn+m)$. Kapoor and Vaidya [24] solve this LP in $\widetilde{O}(k^{2.5}\sqrt{m}n^2)$ time while Lee and Sidford solve it in $\widetilde{O}((kn+m)^{2.5})$ time [25], [26]. These are the fastest algorithms for solving k-commodity flow to high accuracy and none of them use the graph structure of $\mathbf{B}$. In particular, if we were to replace $\mathbf{B}$ by any other $m \times n$ matrix $\mathbf{A}$ with $\mathrm{nnz}(\mathbf{A})$ non-zero entries, [12], [24], [25], [26] would still run in $\widetilde{O}((km)^\omega)$, $\widetilde{O}(\sqrt{km}(k^2n^2 + k\,\mathrm{nnz}(\mathbf{A})))$, or $\widetilde{O}(\sqrt{kn+m}(k\,\mathrm{nnz}(\mathbf{A}) + (kn+m)^2)$ time respectively.

Let us call this type of LP a "*k-commodity LP*" since it is an LP with k commodities $x_1, ..., x_k$ that do not necessarily represent a flow as $\mathbf{A}$ does not have to be an incidence matrix.

Definition II.1. *Given $m \times n$ matrix $\mathbf{A}$, vectors $b_1, ..., b_k \in \mathbb{R}^n$, $c_1, ..., c_n \in \mathbb{R}^m$ we call the following primal and dual linear programs a k-commodity LP.*

$$\begin{array}{rlrl} (\mathcal{P}) & \min_x c^\top x & (\mathcal{D}) & \max_y b^\top y \\ & \mathcal{A}^\top x = b, & & \mathcal{A}y + s = c, \\ & x \ge 0 & & s \ge 0 \end{array}$$

$$\textit{where } \mathcal{A} := \begin{bmatrix} \mathbf{A} & & 0 & \mathbf{I} \\ & \ddots & & \vdots \\ 0 & & \mathbf{A} & \mathbf{I} \\ 0 & ... & 0 & \mathbf{I} \end{bmatrix}$$
$$b := (b_1, ..., b_k, u) \in \mathbb{R}^{kn+m}$$
$$c := (c_1, ..., c_k, 0) \in \mathbb{R}^{(k+1)m}$$

B. Robust IPM and Algebraic Techniques

In this subsection we show how techniques from [24] can be combined with the recent robust interior point framework and data structures from [12], [14], [15] to obtain a faster algorithm for k-commodity LPs. This leads to an algorithm with complexity $\widetilde{O}(k^{2.5}m^{1/2}(n^{\omega-1/2} + \mathrm{nnz}(\mathbf{A})))$ for sparse $\mathbf{A}$ with only $O(1)$ non-zeros per row. This first speed up relies only on algebraic techniques and works for any sparse k-commodity LP. In the later Section II-C, we show how to accelerate this algorithm further via graph based single-commodity flow techniques, when $\mathbf{A}$ is an incidence matrix. This then leads to our fast k-commodity flow algorithms Theorems I.1 and I.2.

a) Robust Central Path: Let us start with a quick recap of the robust central path method which has led to many improvements in LP solvers [13]–[15], [49] and single-commodity flow algorithms [34], [35], [37], [38]. The robust central path method is a variant of the classical central path method for solving LPs. Here, one has two iterates x (a feasible but not optimal solution to the primal LP $\mathcal{P}$ in Definition II.1) and s (the slack of the dual LP $\mathcal{D}$). There are $\widetilde{O}(\sqrt{km})$ iterations and in each iteration, both x and s are moved a bit towards the optimal solution. In the robust central path method, the movement of x and s are given by $x \leftarrow x + \delta_x$, $s \leftarrow s + \delta_s$ where

$$\delta_s = \mathcal{A}(\mathcal{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathcal{A})^{-1}\mathcal{A}^\top \overline{\mathbf{S}}^{-1}\overline{g} \tag{1}$$
$$\delta_x = \overline{\mathbf{S}}^{-1}\overline{g} - \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\delta_s.$$

where $\overline{\mathbf{X}}, \overline{\mathbf{S}}$ are diagonal matrices with $\overline{\mathbf{X}}_{i,i} \approx x_i, \overline{\mathbf{S}}_{i,i} \approx s_i$ for all i, and $\overline{g}$ is some vector specified by the robust central path method. Here $\overline{\mathbf{X}}, \overline{\mathbf{S}}, \overline{g}$ may change from one iteration to the next.

b) Dimension Reduction: Note that $(\mathcal{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathcal{A})$ in (1) is an $(m+kn) \times (m+kn)$ matrix (by definition of $\mathcal{A}$ in Definition II.1), so one would expect at least $\mathrm{poly}(m)$ time per iteration to multiply a vector with it. However, using the k-commodity structure of $\mathcal{A}$, one can reduce the dimension of the linear system onto a smaller inverse of size $kn \times kn$ by taking the Schur-complement. This was first observed in [24]. We briefly sketch how taking the Schur-complement reduces the dimension. We have

$$\mathcal{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathcal{A} = \left[\begin{array}{ccc|c} \mathbf{A}^\top\mathbf{D}_1\mathbf{A} & & 0 & \mathbf{A}^\top\mathbf{D}_1 \\ & \ddots & & \vdots \\ 0 & & \mathbf{A}^\top\mathbf{D}_k\mathbf{A} & \mathbf{A}^\top\mathbf{D}_k \\ \hline \mathbf{D}_1\mathbf{A} & \dots & \mathbf{D}_k\mathbf{A} & \mathbf{D}_\Sigma \end{array}\right] \tag{2}$$

where for ease of notation we defined $\mathbf{D}_i = \overline{\mathbf{X}}_i\overline{\mathbf{S}}_i^{-1}$ where $\overline{\mathbf{X}}_i, \overline{\mathbf{S}}_i$ are the $m \times m$ submatrices with rows in $[(i-1)m+1, im]$ for $i = 1, ..., k+1$, and $\mathbf{D}_\Sigma = \sum_{i=1}^{k+1}\mathbf{D}_i$. By taking the Schur-complement the inverse of (2) is given by Figure 1.

Note that for Figure 1, only the smaller $kn \times kn$ matrix $\mathbf{E}$ must be inverted[8] (since $\mathbf{D}_\Sigma$ is a diagonal matrix and thus trivial to invert). Further, multiplying the vector $\mathcal{A}^\top\overline{\mathbf{S}}^{-1}\overline{g}$ with

[8]If $\mathbf{A}$ is an incidence matrix, then this matrix $\mathbf{E}$ was referred to as a "multi-commodity Laplacian" in [40], and despite graph structure, solving a linear system in $\mathbf{E}$ is as hard as solving a general linear system. Further, after reordering rows and columns, $\mathbf{E}$ is exactly $\mathbf{M}^\top\mathbf{M}$ for $\mathbf{M}$ as in Definition I.3 where $h_i^{(u,v)}$ from Definition I.3 is the diagonal entry of $\mathbf{D}_i \in \mathbb{R}^{m\times m}$ corresponding to edge (u,v).

$$(\mathcal{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathcal{A})^{-1} = \left[\begin{array}{ccc|c} \mathbf{I} & & & 0\\ & \ddots & & \vdots \\ & & \mathbf{I} & 0 \\ \hline -\mathbf{D}_\Sigma^{-1}\mathbf{D}_1\mathbf{A} & \cdots & -\mathbf{D}_\Sigma^{-1}\mathbf{D}_k\mathbf{A} & \mathbf{I}\end{array}\right] \left[\begin{array}{ccc|c} & & & 0\\ & \mathbf{E}^{-1} & & \vdots\\ & & & 0\\ \hline 0 & \cdots & 0 & \mathbf{I}\end{array}\right] \left[\begin{array}{ccc|c} \mathbf{I} & & & -\mathbf{A}^\top\mathbf{D}_1\mathbf{D}_\Sigma^{-1}\\ & \ddots & & \vdots\\ & & \mathbf{I} & -\mathbf{A}^\top\mathbf{D}_k\mathbf{D}_\Sigma^{-1}\\ \hline 0&\cdots&0&\mathbf{D}_\Sigma^{-1}\end{array}\right]$$

$$\text{where } \mathbf{E} := \begin{bmatrix} \mathbf{A}^\top\mathbf{D}_1\mathbf{A} & & 0\\ & \ddots & \\ 0 & & \mathbf{A}^\top\mathbf{D}_k\mathbf{A}\end{bmatrix} - \begin{bmatrix}\mathbf{A}^\top\mathbf{D}_1\\ \vdots\\ \mathbf{A}^\top\mathbf{D}_k\end{bmatrix}\mathbf{D}_\Sigma^{-1}\begin{bmatrix}\mathbf{D}_1\mathbf{A} & \cdots & \mathbf{D}_k\mathbf{A}\end{bmatrix}$$

Fig. 1. Inverse of $\mathcal{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathcal{A}$.

$(\mathcal{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathcal{A})^{-1}$ as in (1) reduces to computing the following expression:

$$w = \mathbf{D}_\Sigma^{-1}\sum_{i=1}^{k+1}\overline{\mathbf{S}}_i^{-1} g_i$$

$$\begin{pmatrix} v_1\\ \vdots\\ v_k\end{pmatrix} = \mathbf{E}^{-1}\underbrace{\begin{pmatrix}\mathbf{A}^\top(\overline{\mathbf{S}}_1^{-1}g_1 - \mathbf{D}_1 w)\\ \vdots\\ \mathbf{A}^\top(\overline{\mathbf{S}}_k^{-1}g_k - \mathbf{D}_k w)\end{pmatrix}}_{=:u}.$$

Assume for now that $\mathbf{A}$ has sparse rows with $O(1)$ non-zero entries (as would be the case for k-commodity flow where $\mathbf{A}$ is an incidence matrix). If an entry of $\overline{\mathbf{X}}, \overline{\mathbf{S}}$, or $\overline{g}$ changes, then only $O(k^2)$ entries in $\mathbf{E}$ and u change, because of sparsity of rows of $\mathbf{A}$. Using a dynamic linear system data structure (e.g., [50]), one can maintain the solution of this linear system $\mathbf{E}^{-1}u$ efficiently under entry updates to $\mathbf{E}$ and u. In particular, an amortized complexity of $\widetilde{O}(k^2(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu/2} + n^{\omega-1/2}))$ per iteration of the central path method is possible for any trade-off parameter $0 \le \mu \le 1$. For current bounds on ω, this is just $\widetilde{O}(k^2 n^{\omega-1/2})$ amortized time per iteration. The only remaining problem is to compute the product with the matrix on the left of $\mathbf{E}^{-1}$ in Figure 1 and the product with $\mathcal{A}$ in definition of δ_s in (1).

For this, we prove in the full version [1] that the steps of the robust central path method (i.e., (1)) can be rewritten as follows. Compute $\delta_s = (\delta_s^1, ..., \delta_s^{k+1})$ where for $v_{k+1} := 0$

$$\delta_s^{(i)} = w + \mathbf{A}v_i - \sum_{j=1}^{k}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}v_j \quad \text{for } i = 1, ..., k+1 \tag{3}$$

So after computing the vectors $(v_1, ..., v_k) = \mathbf{E}^{-1}u$ in $\widetilde{O}(k^2 n^{\omega-1/2})$ amortized time, we are only left with with computing (3). In general, this would take $k^2\,\mathrm{nnz}(\mathbf{A})$ time. However, one can relax this problem to computing only the large entries of $\delta_s^{(i)}$, as outlined below.

c) Heavy Hitters: A common technique when solving linear programs via the robust central path method, is to use "heavy hitter" data structures. (See e.g. [34], [35], [37], [38], [49].) Note that we only need $\overline{\mathbf{X}}, \overline{\mathbf{S}}$ in (1) with $\overline{\mathbf{X}}_{i,i} \approx x_i$ and $\overline{\mathbf{S}}_{i,i} \approx s_i$ for all $1 \le i \le (k+1)m$. Thus we do not actually need to compute vectors x, s explicitly. Further, if an entry s_i did not change much from one iteration to the next, we can reuse the old value of $\overline{\mathbf{S}}_{i,i}$ in the next iteration. Only when s_i changes sufficiently, do we need to update $\overline{\mathbf{S}}_{i,i}$. This leads to the following data structure problem: in each iteration find for which $1 \le i \le (k+1)m$ the entries $|(\delta_s)_i| > \epsilon\overline{\mathbf{S}}_{i,i}$ for some parameter $\epsilon \in (0,1]$. That is, find the "heavy hitters" of the vector $\overline{\mathbf{S}}^{-1}\delta_s$. (And likewise $\overline{\mathbf{X}}^{-1}\delta_x$ but by (1) this is almost the same problem as $\overline{\mathbf{S}}^{-1}\delta_s$.) This allows for a speed-up of the algorithm, because not the entire vector δ_s must be computed. We explain in the next Section II-C how this data structure task can be solved efficiently using graph techniques when matrix $\mathbf{A}$ is an incidence matrix. This relies on data structures from single-commodity flow algorithms [34], [35] based on the expander decomposition framework.

C. Single-Commodity Techniques

As outlined in the previous subsection II-B, the remaining task for obtaining an efficient multi-commodity flow algorithm is to solve a "heavy hitter" data structure problem. The task is to find the large entries of the vector $\overline{\mathbf{S}}^{-1}\delta_s$ (as defined in (3)). The central path method guarantees that $\overline{\mathbf{X}\mathbf{S}} \approx t\mathbf{I}$ for some $t \in \mathbb{R}{>}\,0$, so finding entries larger than some ϵ in $\overline{\mathbf{S}}^{-1}\delta_x$ can be done by finding entries larger than $\epsilon\sqrt{t}$ in $\mathbf{D}^{1/2}\delta_s$ (reminder: $\mathbf{D} = \overline{\mathbf{X}}\ \overline{\mathbf{S}}^{-1}$) by

$$\mathbf{D}^{1/2}\delta_s = \overline{\mathbf{X}}^{1/2}\overline{\mathbf{S}}^{-1/2}\delta_s = (\overline{\mathbf{X}\mathbf{S}})^{1/2}\ \overline{\mathbf{S}}^{-1}\delta_s \approx \sqrt{t}\cdot\overline{\mathbf{S}}^{-1}\delta_s.$$

By definition of δ_s in (3)[9], we try to find large entries of

$$\mathbf{D}_i^{1/2}\left(\mathbf{A}v_i - \sum_{j=1}^{k}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}v_j\right) \text{ for each } i = 1, ..., k+1 \tag{4}$$

(where $\mathbf{D}_i$ is the i^{th} $m \times m$ diagonal subblock of $\mathbf{D}$.)

We remark that finding large entries of (4) is precisely the task we previously described in Section I-B (for a matrix as in Definition I.3). This is because (4) can be phrased as finding the large entries of a matrix-vector-product as in Figure 2.

To solve this heavy hitter problem, an intuitive idea would be to split the problem into smaller tasks on classical incidence

[9]For simplicity, we will from now on ignore the vector w in (3). Since the vector w is explicitly given, it's easy to check if $\mathbf{D}^{1/2}w$ has large entries. It's the sum of products with $\mathbf{A}$ for which finding the large entries is the bottleneck.

$$\left(\begin{bmatrix}\mathbf{D}_1^{1/2}\mathbf{A} & & 0\\ & \ddots & \\ 0 & & \mathbf{D}_{k+1}^{1/2}\mathbf{A}\end{bmatrix} - \begin{bmatrix}\mathbf{D}_1^{1/2}\mathbf{D}_1\mathbf{D}_\Sigma^{-1}\mathbf{A} & \dots & \mathbf{D}_1^{1/2}\mathbf{D}_{k+1}\mathbf{D}_\Sigma^{-1}\mathbf{A}\\ \vdots & \ddots & \vdots\\ \mathbf{D}_{k+1}^{1/2}\mathbf{D}_1\mathbf{D}_\Sigma^{-1}\mathbf{A} & \dots & \mathbf{D}_{k+1}^{1/2}\mathbf{D}_{k+1}\mathbf{D}_\Sigma^{-1}\mathbf{A}\end{bmatrix}\right)\begin{bmatrix}v_1\\ \vdots\\ v_{k+1}\end{bmatrix}$$

Fig. 2. Representation of (4) as a matrix-vector-product. After reordering rows and columns, the matrix is of form Definition I.3, where $h_i^{(u,v)}$ in Definition I.3 is the diagonal entry of $\mathbf{D}_i \in \mathbb{R}^{m\times m}$ that corresponds to edge $(u, v) \in E$.

matrices, e.g. instead of finding $1 \le i \le k+1$ and $1 \le \ell \le m$ where

$$|(\mathbf{D}_i^{1/2}\delta_s^{(i)})_\ell| = \left|\left(\mathbf{D}_i^{1/2}\left(\mathbf{A}v_i - \sum_{j=1}^{k+1}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}v_j\right)\right)_\ell\right| > \epsilon\sqrt{t} \quad (5)$$

we search for indices $1 \le i \le k+1$ and $1 \le \ell \le m$ where

$$\left|\left(\mathbf{D}_i^{1/2}\mathbf{A}v_i\right)_\ell\right| > \frac{\epsilon}{k+1}\sqrt{t} \text{ or}$$
$$\left|\left(\mathbf{D}_i^{1/2}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}v_j\right)_\ell\right| > \frac{\epsilon}{k+1}\sqrt{t} \text{ for any } 1 \le j \le k+1 \quad (6)$$

The problem with this approach is that it is unclear how many indices will be returned this way. Usually, the number of indices that satisfy (5) is bounded by $O(\|\mathbf{D}_j^{1/2}\delta_s^{(j)}\|_2/(\epsilon^2 t))$, and this norm is bounded by properties of the central path method. However, no such bound is given on the norms of the vectors in (6). We might return too many indices, because for some i, ℓ there might be two different j where (6) is satisfied, but one entry is a large positive value and the other is a large negative value. So in (5) the respective entry might still be small because of cancellation. As returning an index takes at least $O(1)$ time, we cannot bound the complexity for finding large entries in (6).

To solve heavy hitter task on (5), we find that there is actually a decomposition of the multi-commodity incidence matrix into smaller classical incidence matrices, where we can guarantee that such cancellations happen rarely.

Let us write $\delta_s^{(i)}$ in a slightly different form:

$$\mathbf{D}_i^{1/2}\delta_s^{(i)} = \mathbf{D}_i^{1/2}\left(\mathbf{A}v_i - \sum_{j=1}^{k+1}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}v_j\right)$$
$$= \sum_{j=1}^{k+1}\mathbf{D}_i^{1/2}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}(v_i - v_j)$$

(Here we used that $\mathbf{D}_\Sigma = \sum_{i=1}^{k+1}\mathbf{D}_i$.) Thus, we can replace the conditions in (6) by the following conditions

$$\left|\left(\mathbf{D}_i^{1/2}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}(v_i - v_j)\right)_\ell\right| > \frac{\epsilon}{k+1}\sqrt{t} \quad (7)$$
$$\text{for any } 1 \le j \le k+1$$

Unlike the terms in (6), we can bound the norms of above terms.

Lemma II.2. *For any $v_1, ..., v_{k+1} \in \mathbb{R}^n$, $d_1, ..., d_{k+1} \in \mathbb{R}^m_{>0}$ let $d_\Sigma = \sum_{i=1}^{k+1} d_i$. Write $\mathbf{D}_i$ and $\mathbf{D}_\Sigma$ for the diagonal matrices with d_i and d_Σ on the diagonal. Then*

$$\sum_{i=1}^{k+1}\sum_{j=1}^{k+1}\left\|\mathbf{D}_i^{1/2}\mathbf{D}_j\mathbf{D}_\Sigma^{-1}\mathbf{A}(v_i - v_j)\right\|_2^2$$
$$\le 4\cdot\sum_{i=1}^{k+1}\left\|\mathbf{D}_i^{1/2}\sum_{j=1}^{k+1}\mathbf{D}_j\mathbf{D}_\Sigma\mathbf{A}(v_i - v_j)\right\|_2^2$$

Since the number of large entries in (7) can be bounded w.r.t the norms in Lemma II.2, we can bound how many entries are large. At the same time, we have

$$\sum_{i=1}^{k+1}\left\|\mathbf{D}_i^{1/2}\sum_{j=1}^{k+1}\mathbf{D}_j\mathbf{D}_\Sigma\mathbf{A}(v_i - v_j)\right\|_2^2$$
$$= \sum_{i=1}^{k+1}\|\mathbf{D}_i^{1/2}\delta_s^{(i)}\|_2^2 = \|\mathbf{D}\delta_s\|_2^2$$

which is bounded by guarantees of the central path method. So we never return too many large entries in (7). Finally, we use the heavy hitter data structure for classical incidence matrices from [34], originally developed for the purpose of solving single-commodity flows.

a) Summary: In summary, we can solve the heavy hitter problem on multi-commodity incidence matrices by splitting it into $O(k^2)$ instances on classical incidence matrices. This is quite different from the inverse subroutine (i.e., the data structure to invert the multi-commodity Laplacian $\mathbf{E}$ in Figure 1): solving multi-commodity Laplacians are unlikely be efficiently reducible to regular Laplacian systems [40] unless there is some breakthrough in sparse linear system solving. This demonstrates that while any LP can be reduced to multi-commodity flow [41], [42], some aspects of solving multi-commodity flow on dense graphs can actually be reduced to the single commodity case and accelerated using graph techniques.

Acknowledgement

We would like to thank Yin Tat Lee, Aaron Sidford, Rasmus Kyng, and Richard Peng for helpful discussions. This work was partially done while Jan van den Brand was at UC Berkeley and the Max Planck Institute for Informatics. Part of this research was funded by ONR BRC grant N00014-18-1-2562, and by the Simons Institute for the Theory of Computing through a Simons-Berkeley Postdoctoral Fellowship, and by the Max Planck Institute.

REFERENCES

[1] J. van den Brand and D. Zhang, “Faster high accuracy multi-commodity flow from single-commodity techniques,” *CoRR*, vol. abs/2304.12992, 2023.

[2] F. T. Leighton, F. Makedon, S. A. Plotkin, C. Stein, É. Tardos, and S. Tragoudas, “Fast approximation algorithms for multicommodity flow problems,” *J. Comput. Syst. Sci.*, vol. 50, no. 2, pp. 228–243, 1995, announced at STOC’91.

[3] L. Fleischer, “Approximating fractional multicommodity flow independent of the number of commodities,” *SIAM J. Discret. Math.*, vol. 13, no. 4, pp. 505–520, 2000, announced at FOCS’99.

[4] N. Garg and J. Könemann, “Faster and simpler algorithms for multicommodity flow and other fractional packing problems,” *SIAM J. Comput.*, vol. 37, no. 2, pp. 630–652, 2007, announced at FOCS’98.

[5] A. Madry, “Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms,” in *STOC*. ACM, 2010, pp. 121–130.

[6] J. A. Kelner, G. L. Miller, and R. Peng, “Faster approximate multicommodity flow using quadratically coupled flows,” in *STOC*. ACM, 2012, pp. 1–18.

[7] J. Sherman, “Nearly maximum flows in nearly linear time,” in *FOCS*. IEEE Computer Society, 2013, pp. 263–269.

[8] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, “An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations,” in *SODA*. SIAM, 2014, pp. 217–226.

[9] R. Peng, “Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time,” in *SODA*. SIAM, 2016, pp. 1862–1867.

[10] L. Chen and M. Ye, “High-accuracy multicommodity flows via iterative refinement,” *CoRR*, vol. abs/2304.11252, 2023.

[11] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva, “Maximum flow and minimum-cost flow in almost-linear time,” in *FOCS*. IEEE, 2022.

[12] M. B. Cohen, Y. T. Lee, and Z. Song, “Solving linear programs in the current matrix multiplication time,” *J. ACM*, vol. 68, no. 1, pp. 3:1–3:39, 2021, announced at STOC’19.

[13] Y. T. Lee, Z. Song, and Q. Zhang, “Solving empirical risk minimization in the current matrix multiplication time,” in *COLT*, ser. Proceedings of Machine Learning Research, vol. 99. PMLR, 2019, pp. 2140–2157.

[14] J. v. d. Brand, “A deterministic linear program solver in current matrix multiplication time,” in *SODA*. SIAM, 2020, pp. 259–278.

[15] S. Jiang, Z. Song, O. Weinstein, and H. Zhang, “A faster algorithm for solving general lps,” in *STOC*. ACM, 2021, pp. 823–832.

[16] R. Duan, H. Wu, and R. Zhou, “Faster matrix multiplication via asymmetric hashing,” *CoRR*, vol. abs/2210.10173, 2022.

[17] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, “New bounds for matrix multiplication: from alpha to omega,” *CoRR*, vol. abs/2307.07970, 2023.

[18] F. L. Gall and F. Urrutia, “Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor,” in *SODA*. SIAM, 2018, pp. 1029–1046.

[19] T. C. Hu, “Multi-commodity network flows,” *Operations research*, vol. 11, no. 3, pp. 344–360, 1963.

[20] N. Karmarkar, “A new polynomial-time algorithm for linear programming,” *Combinatorica*, vol. 4, no. 4, pp. 373–396, 1984, announced at STOC’84.

[21] J. Renegar, “A polynomial-time algorithm, based on newton’s method, for linear programming,” *Math. Program.*, vol. 40, no. 1-3, pp. 59–93, 1988.

[22] P. M. Vaidya, “An algorithm for linear programming which requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ arithmetic operations,” in *STOC*. ACM, 1987, pp. 29–38.

[23] ——, “Speeding-up linear programming using fast matrix multiplication (extended abstract),” in *FOCS*. IEEE Computer Society, 1989, pp. 332–337.

[24] S. Kapoor and P. M. Vaidya, “Speeding up karmarkar’s algorithm for multicommodity flows,” *Math. Program.*, vol. 73, pp. 111–127, 1996.

[25] Y. T. Lee and A. Sidford, “Path finding methods for linear programming: Solving linear programs in $O(\sqrt{rank})$ iterations and faster algorithms for maximum flow,” in *55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2014, pp. 424–433.

[26] ——, “Efficient inverse maintenance and faster algorithms for linear programming,” in *FOCS*. IEEE Computer Society, 2015, pp. 230–249.

[27] S. I. Daitch and D. A. Spielman, “Faster approximate lossy generalized flow via interior point algorithms,” in *STOC*. ACM, 2008, pp. 451–460.

[28] A. Madry, “Navigating central path with electrical flows: From flows to matchings, and back,” in *FOCS*. IEEE Computer Society, 2013, pp. 253–262.

[29] M. B. Cohen, A. Madry, P. Sankowski, and A. Vladu, “Negative-weight shortest paths and unit capacity minimum cost flow in õ $(m^{10/7} \log W)$ time (extended abstract),” in *SODA*. SIAM, 2017, pp. 752–771.

[30] A. Madry, “Computing maximum flow with augmenting electrical flows,” in *FOCS*. IEEE Computer Society, 2016, pp. 593–602.

[31] K. Axiotis, A. Madry, and A. Vladu, “Faster sparse minimum cost flow by electrical flow localization,” in *FOCS*. IEEE, 2021, pp. 528–539.

[32] Y. P. Liu and A. Sidford, “Faster energy maximization for faster maximum flow,” in *STOC*. ACM, 2020, pp. 803–814.

[33] T. Kathuria, Y. P. Liu, and A. Sidford, “Unit capacity maxflow in almost $o(m^{4/3})$ time,” in *FOCS*. IEEE, 2020, pp. 119–130.

[34] J. v. d. Brand, Y. T. Lee, D. Nanongkai, R. Peng, T. Saranurak, A. Sidford, Z. Song, and D. Wang, “Bipartite matching in nearly-linear time on moderately dense graphs,” in *FOCS*. IEEE, 2020, pp. 919–930.

[35] J. v. d. Brand, Y. T. Lee, Y. P. Liu, T. Saranurak, A. Sidford, Z. Song, and D. Wang, “Minimum cost flows, mdps, and ℓ_1)-regression in nearly linear time for dense instances,” in *STOC*. ACM, 2021, pp. 859–869.

[36] S. Dong, Y. Gao, G. Goranci, Y. T. Lee, R. Peng, S. Sachdeva, and G. Ye, “Nested dissection meets ipms: Planar min-cost flow in nearly-linear time,” in *SODA*. SIAM, 2022, pp. 124–153.

[37] J. v. d. Brand, Y. Gao, A. Jambulapati, Y. T. Lee, Y. P. Liu, R. Peng, and A. Sidford, “Faster maxflow via improved dynamic spectral vertex sparsifiers,” in *STOC*. ACM, 2022, pp. 543–556.

[38] Y. Gao, Y. P. Liu, and R. Peng, “Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao,” in *FOCS*. IEEE, 2021, pp. 516–527.

[39] D. A. Spielman and S. Teng, “Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems,” in *STOC*. ACM, 2004, pp. 81–90.

[40] R. Kyng and P. Zhang, “Hardness results for structured linear systems,” *SIAM J. Comput.*, vol. 49, no. 4, 2020, announced at FOCS’17.

[41] A. Itai, “Two-commodity flow,” *J. ACM*, vol. 25, no. 4, pp. 596–611, 1978.

[42] M. Ding, R. Kyng, and P. Zhang, “Two-commodity flow is equivalent to linear programming under nearly-linear time reductions,” in *ICALP*, ser. LIPIcs, vol. 229. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 54:1–54:19.

[43] Y. Hua, R. Kyng, M. P. Gutenberg, and Z. Wu, “Maintaining expander decompositions via sparse cuts,” *CoRR*, vol. abs/2204.02519, 2022.

[44] A. Bernstein, J. v. d. Brand, M. P. Gutenberg, D. Nanongkai, T. Saranurak, A. Sidford, and H. Sun, “Fully-dynamic graph sparsifiers against an adaptive adversary,” in *ICALP*, ser. LIPIcs, vol. 229. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:20.

[45] D. Nanongkai and T. Saranurak, “Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $o(n^{1/2-\epsilon})$-time,” in *STOC*. ACM, 2017, pp. 1122–1129.

[46] C. Wulff-Nilsen, “Fully-dynamic minimum spanning forest with improved worst-case update time,” in *STOC*. ACM, 2017, pp. 1130–1143.

[47] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen, “Dynamic minimum spanning forest with subpolynomial worst-case update time,” in *FOCS*. IEEE Computer Society, 2017, pp. 950–961.

[48] T. Saranurak and D. Wang, “Expander decomposition and pruning: Faster, stronger, and simpler,” in *SODA*. SIAM, 2019, pp. 2616–2635.

[49] J. v. d. Brand, Y. T. Lee, A. Sidford, and Z. Song, “Solving tall dense linear programs in nearly linear time,” in *STOC*. ACM, 2020, pp. 775–788.

[50] J. v. d. Brand, “Unifying matrix data structures: Simplifying and speeding up iterative algorithms,” in *SOSA*. SIAM, 2021, pp. 1–13.

[51] B. Rothschild and A. B. Whinston, “Feasibility of two commodity network flows,” *Oper. Res.*, vol. 14, no. 6, pp. 1121–1129, 1966.

[52] H. Jiang, T. Kathuria, Y. T. Lee, S. Padmanabhan, and Z. Song, “A faster interior point method for semidefinite programming,” in *FOCS*. IEEE, 2020, pp. 910–918.

[53] B. Huang, S. Jiang, Z. Song, R. Tao, and R. Zhang, “Solving sdp faster: A robust ipm framework and efficient implementation,” *CoRR*, vol. abs/2101.08208, 2021.

[54] S. Jiang, B. Natura, and O. Weinstein, “A faster interior-point method for sum-of-squares optimization,” in *ICALP*, ser. LIPIcs, vol. 229. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 79:1–79:20.

[55] Y. T. Lee, A. Sidford, and S. C. Wong, “A faster cutting plane method and its implications for combinatorial and convex optimization,” in *FOCS*. IEEE Computer Society, 2015, pp. 1049–1065.

[56] H. Jiang, Y. T. Lee, Z. Song, and S. C. Wong, "An improved cutting plane method for convex optimization, convex-concave games, and its applications," in *STOC*. ACM, 2020, pp. 944–953.

[57] P. Sankowski, "Subquadratic algorithm for dynamic shortest distances," in *COCOON*, ser. Lecture Notes in Computer Science, vol. 3595. Springer, 2005, pp. 461–470.

[58] J. v. d. Brand, D. Nanongkai, and T. Saranurak, "Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds," in *FOCS*. IEEE Computer Society, 2019, pp. 456–480.

[59] D. Dadush, B. Natura, and L. A. Végh, "Revisiting tardos's framework for linear programming: Faster exact solutions using approximate solvers," in *FOCS*. IEEE, 2020, pp. 931–942.

[60] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak, "A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond," in *FOCS*. IEEE, 2020, pp. 1158–1167.

[61] W. Jin and X. Sun, "Fully dynamic s-t edge connectivity in subpolynomial time (extended abstract)," in *FOCS*. IEEE, 2021, pp. 861–872.

[62] G. Goranci, H. Räcke, T. Saranurak, and Z. Tan, "The expander hierarchy and its applications to dynamic graph algorithms," in *SODA*. SIAM, 2021, pp. 2212–2228.

[63] J. Chuzhoy, "Decremental all-pairs shortest paths in deterministic near-linear time," in *STOC*. ACM, 2021, pp. 626–639.

[64] J. Chuzhoy and T. Saranurak, "Deterministic algorithms for decremental shortest paths via layered core decomposition," in *SODA*. SIAM, 2021, pp. 2478–2496.

[65] J. Chuzhoy and S. Khanna, "A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems," in *STOC*. ACM, 2019, pp. 389–400.

[66] A. Bernstein, M. P. Gutenberg, and T. Saranurak, "Deterministic decremental SSSP and approximate min-cost flow in almost-linear time," in *FOCS*. IEEE, 2021, pp. 1000–1008.

[67] ——, "Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing," in *FOCS*. IEEE, 2020, pp. 1123–1134.

[68] M. B. Cohen, J. A. Kelner, R. Kyng, J. Peebles, R. Peng, A. B. Rao, and A. Sidford, "Solving directed laplacian systems in nearly-linear time through sparse LU factorizations," in *FOCS*. IEEE Computer Society, 2018, pp. 898–909.

[69] R. Kyng and S. Sachdeva, "Approximate gaussian elimination for laplacians - fast, sparse, and simple," in *FOCS*. IEEE Computer Society, 2016, pp. 573–582.

[70] R. Kyng, Y. T. Lee, R. Peng, S. Sachdeva, and D. A. Spielman, "Sparsified cholesky and multigrid solvers for connection laplacians," in *STOC*. ACM, 2016, pp. 842–850.

[71] Y. T. Lee, R. Peng, and D. A. Spielman, "Sparsified cholesky solvers for SDD linear systems," *CoRR*, vol. abs/1506.08204, 2015.

[72] R. Peng and D. A. Spielman, "An efficient parallel solver for SDD linear systems," in *STOC*. ACM, 2014, pp. 333–342.

[73] M. B. Cohen, R. Kyng, G. L. Miller, J. W. Pachocki, R. Peng, A. B. Rao, and S. C. Xu, "Solving SDD linear systems in nearly $m\log^{1/2}n$ time," in *STOC*. ACM, 2014, pp. 343–352.

[74] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, "A simple, combinatorial algorithm for solving SDD systems in nearly-linear time," in *STOC*. ACM, 2013, pp. 911–920.

[75] I. Koutis, G. L. Miller, and R. Peng, "A nearly-m log n time solver for SDD linear systems," in *FOCS*. IEEE Computer Society, 2011, pp. 590–598.

[76] ——, "Approaching optimality for solving SDD linear systems," *SIAM J. Comput.*, vol. 43, no. 1, pp. 337–354, 2014.