Two-Commodity Flow Is Equivalent to Linear Programming Under Nearly-Linear Time Reductions

Ming Ding
ETH Zürich, Switzerland

Rasmus Kyng
ETH Zürich, Switzerland

Peng Zhang
Rutgers University, Piscataway, NJ, USA

Abstract

We give a nearly-linear time reduction that encodes any linear program as a 2-commodity flow problem with only a small blow-up in size. Under mild assumptions similar to those employed by modern fast solvers for linear programs, our reduction causes only a polylogarithmic multiplicative increase in the size of the program, and runs in nearly-linear time. Our reduction applies to high-accuracy approximation algorithms and exact algorithms. Given an approximate solution to the 2-commodity flow problem, we can extract a solution to the linear program in linear time with only a polynomial factor increase in the error. This implies that any algorithm that solves the 2-commodity flow problem can solve linear programs in essentially the same time. Given a directed graph with edge capacities and two source-sink pairs, the goal of the 2-commodity flow problem is to maximize the sum of the flows routed between the two source-sink pairs subject to edge capacities and flow conservation. A 2-commodity flow problem can be formulated as a linear program, which can be solved to high accuracy in almost the current matrix multiplication time (Cohen-Lee-Song JACM'21). Our reduction shows that linear programs can be approximately solved, to high accuracy, using 2-commodity flow as well.

Our proof follows the outline of Itai's polynomial-time reduction of a linear program to a 2-commodity flow problem (JACM'78). Itai's reduction shows that exactly solving 2-commodity flow and exactly solving linear programming are polynomial-time equivalent. We improve Itai's reduction to nearly preserve the problem representation size in each step. In addition, we establish an error bound for approximately solving each intermediate problem in the reduction, and show that the accumulated error is polynomially bounded. We remark that our reduction does not run in strongly polynomial time and that it is open whether 2-commodity flow and linear programming are equivalent in strongly polynomial time.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Linear programming

Keywords and phrases Two-Commodity Flow Problems, Linear Programming, Fine-Grained Complexity

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.54

Category Track A: Algorithms, Complexity and Games

Related Version *Extended Version*: `https://arxiv.org/abs/2201.11587` [6]

Funding *Rasmus Kyng*: The research leading to these results has received funding from grant no. 200021 204787 of the Swiss National Science Foundation.

1 Introduction

Multi-commodity maximum flow is a very well-studied problem, which can be formulated as a linear program. In this paper, we show that general linear programs can be very efficiently encoded as a multi-commodity maximum flow programs. Many variants of multi-commodity flow problems exist. We consider one of the simplest directed variants, 2-commodity maximum through-put flow. Given a directed graph with edge capacities and two source-sink pairs, this problem requires us to maximize the sum of the flows routed between the two source-sink pairs, while satisfying capacity constraints and flow conservation at the remaining nodes. In the rest of the paper, we will simply refer to this as *the 2-commodity flow problem*. We abbreviate this problem as 2CF. Our goal is to relate the hardness of solving 2CF to that of solving linear programs (LPs). 2-commodity flow is easily expressed as a linear program, so it is clearly no harder than solving LPs. We show that the 2-commodity flow problem can encode a linear program with only a polylogarithmic blow-up in size, when the program has polynomially bounded integer entries and polynomially bounded solution norm. Our reduction runs in nearly-linear time. Given an approximate solution to the 2-commodity flow problem, we can recover, in linear time, an approximate solution to the linear program with only a polynomial factor increase in the error. Our reduction also shows that an exact solution to the flow problem yields an exact solution to the linear program.

Multi-commodity flow problems are extremely well-studied and have been the subject of numerous surveys [23, 38, 40, 1, 50], in part because a large number of problems can be expressed as variants of multi-commodity flow. Our result shows a very strong form of this observation: In fact, general linear programs can be expressed as 2-commodity flow problems with essentially the same size. Early in the study of these problems, before a polynomial-time algorithm for linear programming was known, it was shown that the *undirected* 2-commodity flow problem can be solved in polynomial time [16]. In fact, it can be reduced to two undirected single commodity maximum flow problems. In contrast, directed 2-commodity flow problems were seemingly harder, despite the discovery of non-trivial algorithms for some special cases [8, 9].

Searching for Multi-Commodity Flow Solvers. Alon Itai [17] proved a polynomial-time reduction from linear programming to 2-commodity flow, before a polynomial-time algorithm for linear programming was known. For decades, the only major progress on solving multi-commodity flow came from improvements to general linear program solvers [24, 19, 43, 48]. Leighton et al. [33] showed that undirected capacitated k-commodity flow in a graph with m edges and n vertices can be approximately solved in $\widetilde{O}(kmn)$ time, completely routing all demands with $1+\epsilon$ times the optimal congestion, albeit with a poor dependence on the error parameter ϵ. This beats solve-times for linear programming in sparse graphs for small k, even with today's LP solvers that run in current matrix multiplication time, albeit with much worse error. This result spurred a number of follow-up works with improvements for low-accuracy algorithms [14, 11, 35]. Later, breakthroughs in achieving almost- and nearly-linear time algorithms for undirected single-commodity maximum flow also lead to faster algorithms for undirected k-commodity flow [21, 45, 41], culminating in Sherman's introduction of area-convexity to build a $\widetilde{O}(mk\epsilon^{-1})$ time algorithm for approximate undirected k-commodity flow [46].

Solving Single-Commodity Flow Problems. Single commodity flow problems have been an area of tremendous success for the development of graph algorithms, starting with an era of algorithms influenced by early results on maximum flow and minimum cut [12] and later the development of powerful combinatorial algorithms for maximum flow with polynomially bounded edge capacities [7, 10, 15]. Later, a breakthrough nearly-linear time algorithm for electrical flows by Spielman and Teng [47] lead to the *Laplacian paradigm*. A long line of work explored direct improvements and simplifications of this result [25, 26, 22, 42, 28, 18]. This also motivated a new line of research on undirected maximum flow [3, 31, 21, 45], which in turn lead to faster algorithms for directed maximum flow and minimum cost flow [36, 37, 34, 20, 49, 13] building on powerful tools using mixed-ℓ_2, ℓ_p-norm minimizing flows [27] and inverse-maintenance ideas [2]. Certain developments are particularly relevant to our result: For a graph $G = (V, E)$ these works established high-accuracy algorithms with $\widetilde{O}(|E|)$ running time for computing electrical flow [47] and $O(|E|^{4/3})$ running time for unit capacity directed maximum flow [36, 20], and $\widetilde{O}(\min(|E|^{1.497}, |E| + |V|^{1.5}))$ running time for directed maximum flow with general capacities [13, 49].

Solving General Linear Programs. As described in the previous paragraphs, there has been tremendous success in developing fast algorithms for single-commodity flow problems and undirected multi-commodity flow problems, albeit in the latter case only in the low-accuracy regime (as the algorithm running times depend polynomially on the error parameter). In contrast, the best known algorithms for directed multi-commodity flow simply treat the problem as a general linear program, and use a solver for general linear programs.

The fastest known solvers for general linear programs are based on interior point methods [19], and in particular central path methods [43]. Recently, there has been significant progress on solvers for general linear programs, but the running time required to solve a linear program with roughly n variables and $\widetilde{O}(n)$ constraints (assuming polynomially bounded entries and polytope radius) is stuck at the $\widetilde{O}(n^{2.372\cdots})$, the running time provided by LP solvers that run in current matrix multiplication time [4]. Note that this running time is in the RealRAM model, and this algorithm cannot be translated to fixed point arithmetic with polylogarithmic bit complexity per number without additional assumptions on the input, as we describe the paragraphs on numerical stability below. To compare these running times with those for single-commodity maximum flow algorithms on a graph with $|V|$ vertices and $|E|$ edges, observe that in a sparse graph with $|E| = \widetilde{O}(|V|)$, by writing the maximum flow problem as a linear program, we can solve it using general linear program solvers and obtain a running time of $\widetilde{O}(|V|^{2.372\cdots})$, while the state-of-the art maximum flow solver obtains a running time of $\widetilde{O}(|V|^{1.497})$ on such a sparse graph. On dense graphs with $|E| = \Theta(|V|^2)$, the gap is smaller but still substantial: The running time is $\widetilde{O}(|V|^2)$ using maximum flow algorithms vs. $\widetilde{O}(|V|^{2.372\cdots})$ using general LP algorithms.

How Hard Is It to Solve Multi-Commodity Flow? The many successes in developing high-accuracy algorithms for single-commodity flow problems highlight an important open question: Can multi-commodity flow be solved to high accuracy faster than general linear programs? We rule out this possibility, by proving that any linear program (assuming it is polynomially bounded and has integer entries) can be encoded as a multi-commodity flow problem in nearly-linear time. This implies that any improvement in the running time of (high-accuracy) algorithms for sparse multi-commodity flow problems would directly translate to a faster algorithm for solving sparse linear programs to high accuracy, with only a polylogarithmic increase in running time.

Previous work by Kyng and Zhang [30] had shown that fast algorithms for multi-commodity flow were unlikely to arise from combining interior point methods with special-purpose linear equation solvers. Concretely, they showed that the linear equations that arise in interior point methods for multi-commodity flow are as hard to solve as arbitrary linear equations. This ruled out algorithms following the pattern of the known fast algorithms for high-accuracy single-commodity flow problems. However, it left open the broader question if some other family of algorithms could succeed. We now show that, in general, a separation between multi-commodity flow and linear programming is not possible.

1.1 Background: Numerical Stability of Linear Program Solvers and Reductions

Current research on fast algorithms for solving linear programs generally relies on assuming bounds on (1) the size of the program entries and (2) the norm of all feasible solutions. Generally, algorithm running time depends logarithmically on these quantities, and hence to make these factors negligible, entry size and feasible solution norms are assumed to be polynomially bounded, for example in [4]. We will refer to a linear program satisfying these assumptions as *polynomially bounded*. More precisely, we say a linear program with N non-zero coefficients is *polynomially bounded* if it has coefficients in the range $[-X, X]$ and $\|\boldsymbol{x}\|_1 \leq R$ for all feasible $\boldsymbol{x}$ (i.e. the polytope of feasible solutions has radius of R in ℓ_1 norm), and $X, R \leq O(N^c)$ for some constant c. In fact, if there exists a feasible solution $\boldsymbol{x}$ satisfying $\|\boldsymbol{x}\|_1 \leq R$, then we can add a constraint $\|\boldsymbol{x}\|_1 \leq R$ to the LP (which can be rewritten as linear inequality constraints) so that in the new LP, all feasible solutions have ℓ_1 norm at most R. This only increases the number of nonzeros in the LP by at most a constant factor.

Interior Point Methods and Reductions With Fixed Point Arithmetic. Modern fast interior point methods for linear programming, such as [4], are analyzed in the RealRAM model. In order to implement these algorithms using *fixed point arithmetic* with polylogarithmic bit complexity per number, instead of RealRAM, additional assumptions are required. For example, this class of algorithms relies on computing matrix inverses, and these must be approximately representable using polylogarithmic bit complexity per entry. This is not possible, if the inverses have exponentially large entries, which may occur even in polynomially bounded linear programs. For example, consider a linear program feasibility problem $\boldsymbol{Ax} \leq \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}$, with constraint matrix $\boldsymbol{A} \in \mathbb{R}^{2n \times 2n}$ given by

$$\boldsymbol{A}(i,j) = \begin{cases} 1 & \text{if } i < n \text{ and } i = j \\ -2 & \text{if } i < n \text{ and } i+1 = j \\ 2 & \text{if } i \geq n \text{ and } i = j \\ -1 & \text{if } i \geq n \text{ and } i+1 = j \\ 0 & \text{o.w.} \end{cases}$$

Such a linear program is polynomially bounded for many choices of $\boldsymbol{b}$, e.g. $\boldsymbol{b} = \boldsymbol{e}_{2n}$. Unfortunately, for the vector $\boldsymbol{x} \in \mathbb{R}^{2n}$ given by

$$\boldsymbol{x}(i) = \begin{cases} 2^{-i-1} & \text{if } i \leq n \\ 0 & \text{o.w.} \end{cases}$$

we have $\boldsymbol{Ax} = 2^{-n}\boldsymbol{e}_n$, and from this one can see that $\boldsymbol{A}^{-1}$ must have entries of size at least $\Omega(2^n/n)$. This will cause algorithms such as [4] to perform intermediate calculations with n bits per number, increasing the running time by a factor roughly n.

Modern interior point methods can be translated to fixed precision arithmetic with various different assumptions leading to different per entry bit complexity (see [4] for a discussion of one standard sufficient condition). Furthermore, if the problem has polynomially bounded condition number (when appropriately defined), then we expect that polylogarithmic bit complexity per entry should suffice, at least for highly accurate approximate solutions, by relying on fast stable numerical linear algebra [5], although we are not aware of a complete analysis of this translation.

If a linear program with integer entries is solved to sufficiently small additive error, the approximate solution can be converted into an exact solution, e.g. see [43, 32, 4] for a discussion of the necessary precision and for a further discussion of numerical stability properties of interior point methods. Polynomially bounded linear programs with integer coefficients may still require exponentially small additive error for this rounding to succeed.

We give a reduction from general linear programming to 2-commodity flow, and like [4], we assume the program is polynomially bounded to carry out the reduction. We also use an additional assumption, namely that the linear program is written using integral entries[1]. We do not make additional assumptions about polynomially bounded condition number of problem. This means we can apply our reduction to programs such as the one above, despite [4] not obtaining a reasonable running time on such programs using fixed point arithmetic.

Our analysis of our reduction uses the RealRAM model like [4] and other modern interior point method analysis, however, it should be straightforward to translate our reduction and error analysis to fixed point arithmetic with polylogarithmically many bits, because all our mappings are simple linear transformations, and we never need to compute or apply a matrix inverse.

Rounding Linear Programs to Have Integer Entries. It is possible to give some fairly general and natural sufficient conditions for when a polynomially bounded linear program can be rounded to have integral entries, one example of this is having a polynomially bounded *Renegar's condition number.* Renegar introduced this condition number for linear programs in [44]. For a given linear program, suppose that perturbing the entries of the program by at most δ each does not change the feasibility of the the linear program, and let δ^* be the largest such δ. Let U denote the maximum absolute value of entries in the linear program. Then $\kappa = U/\delta^*$ is Renegar's condition number for the linear program.

Suppose we are given a polynomially bounded linear program $\max\{\boldsymbol{c}^\top \boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}, \boldsymbol{x} \ge \boldsymbol{0}\}$ (also referred to as $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$), with polytope radius at most R, and Renegar's condition number κ also bounded by a polynomial. We wish to compute a vector $\boldsymbol{x} \ge \boldsymbol{0}$ with an ϵ additive error on each constraint and in the optimal value. We can reduce this problem for instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ to a polynomially bounded linear program instance with integral input numbers. Specifically, we round the entries of $\boldsymbol{A}$ down to $\tilde{\boldsymbol{A}}$ and those of $\boldsymbol{b}, \boldsymbol{c}$ up to $\tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}}$ all by at most $\min\{\frac{\epsilon}{3R}, \frac{U}{\kappa R}\}$ such that each entry of $\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}}$ only needs a logarithmic number of bits. Suppose $\tilde{\boldsymbol{x}}$ is a solution to $(\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}})$ with $\frac{\epsilon}{3}$ additive error on each constraint and in the optimal value. Then,

$$\boldsymbol{A}\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} + (\boldsymbol{A} - \tilde{\boldsymbol{A}})\tilde{\boldsymbol{x}} \le \boldsymbol{b} + \epsilon\boldsymbol{1}, \qquad \boldsymbol{c}^\top \tilde{\boldsymbol{x}} \ge \tilde{\boldsymbol{c}}^\top \tilde{\boldsymbol{x}}^* - \frac{2}{3}\epsilon$$

where $\boldsymbol{1}$ is the all-one vector, $\tilde{\boldsymbol{x}}^*$ is an optimal solution to $(\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}})$. In addition, the optimal value of $(\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}})$ is greater than or equal to that of $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$. So, $\tilde{\boldsymbol{x}}$ is a solution to $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ with ϵ additive error as desired. Since each entry of $\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{\boldsymbol{c}}$ has a logarithmic number of bits,

[1] W.l.o.g., by scaling, this is the same as assuming the program is written with polynomially bounded fixed precision numbers of the form k/D where k is an integer, and D is an integral denominator shared across all entries, and both k and D are polynomially bounded.

we can scale all of them to polynomially bounded integers without changing the feasible set and the optimal solutions. Thus we see that if we restrict ourselves to polynomially linear programs with polynomially bounded Renegar's condition number, and we wish to solve the program with small additive error, we can assume without loss of generality that the program has integer coefficients.

1.2 Previous Work

Our paper follows the proof by Itai [17] that linear programming is polynomial-time reducible to 2-commodity flow. However, it is also inspired by recent works on hardness for structured linear equations [30] and packing/covering LPs [29], which focused on obtaining nearly-linear time reductions in somewhat related settings. These works in turn were motivated by the last decade's substantial progress on fine-grained complexity for a range of polynomial time solvable problems, e.g. see [51]. Also notable is the result by Musco et al. [39] on hardness for matrix spectrum approximation.

1.3 Our Contributions

In this paper, we explore the hardness of 2-commodity maximum throughput flow, which for brevity we refer to as the 2-commodity flow problem or 2CF. We relate the difficulty of 2CF to that of linear programming (LP) by developing an extremely efficient reduction from the former to the latter. The main properties of our reduction are described by the informal theorem statement below. We give a formal statement of Theorem 1.1 as Theorem 3.1 in Section 3.

▶ **Theorem 1.1** (Main Theorem (Informal)). *Consider any polynomially bounded linear program* $\max\{\boldsymbol{c}^\top \boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}\}$ *with integer coefficients and N non-zero entries. In nearly-linear time, we can convert this linear program to a 2-commodity flow problem which is feasible if and only if the original program is. The 2-commodity flow problem has $\widetilde{O}(N)$ edges and has polynomially bounded integral edge capacities. Furthermore, any solution to the 2-commodity flow instance with at most ϵ additive error on each constraint and value at most ϵ from the optimum can be converted to a solution to the original linear program with additive error $\widetilde{O}(\mathrm{poly}(N)\epsilon)$ on each constraint and similarly value within $\widetilde{O}(\mathrm{poly}(N)\epsilon)$ of the optimum.*

This implies that, for any constant $a > 1$, if any 2-commodity flow instance with polynomially bounded integer capacities can be solved with ϵ additive error in time $\widetilde{O}\left(|E|^a \cdot \mathrm{poly}\log(1/\epsilon)\right)$, then any polynomially bounded linear program can be solved to ϵ additive error in time $\widetilde{O}\left(N^a \cdot \mathrm{poly}\log(1/\epsilon)\right)$.

Note that in our definition any 2CF problem is already an LP, and so no reduction in the other direction is necessary. Our notion of approximate solutions to LPs and 2CF problems is also such that treating a 2CF problem as an LP and solving it approximately ensures that the 2CF is approximately solved w.r.t. to the our approximate solution definition for 2CF.

We obtain Theorem 1.1, our main result, by making several improvements to Itai's reduction from LP to 2CF. Recall that a linear program with N non-zero coefficients is polynomially bounded if it has coefficients in the range $[-X, X]$ and $\|\boldsymbol{x}\|_1 \leq R$ for all feasible $\boldsymbol{x}$, where $X, R \leq O(N^c)$ for some constant c. Firstly, while Itai produced a 2CF with the number of edges on the order of $\Theta\left(N^2 \log^2 X\right)$, we show that an improved gadget can reduce this to $O\left(N \log X\right)$. Thus, in the case of polynomially bounded linear programs, where $\log X = O(\log N)$, we get an only polylogarithmic multiplicative increase in the number of non-zero entries from N to $\widetilde{O}(N)$, whereas Itai had an increase in the number of nonzeros by a factor $\widetilde{O}(N)$, from N to $\widetilde{O}(N^2)$.

Secondly, Itai used very large graph edge capacities that require $O\left((N \log X)^{1.01}\right)$ many bits *per edge*, letting the capacities grow exponentially given an LP with polynomially bounded entries. We show that when the feasible polytope radius R is bounded, we can ensure capacities remain a polynomial function of the initial parameters N, R, and X. In the important case of polynomially bounded linear programs, this means the capacities stay polynomially bounded.

Thirdly, while Itai only analyzed the chain of reductions under the case with exact solutions, we generalize the analysis to the case with approximate solutions by establishing an error analysis along the chain of reductions. We show that the error only grows polynomially during the reduction. Moreover, to simplify our error analysis, we observe that additional structures can be established in many of Itai's reductions. For instance, we propose the notion of a *fixed flow network*, which consists of a subset of edges with equal lower and upper bound of capacity. It is a simplification of Itai's (l, u) network with general capacity (both lower and upper bounds on the amount of flow).

Open Problems. Our reductions do not suffice to prove that a strongly polynomial time algorithm for 2-commodity flow would imply a strongly polynomial time algorithm for linear programming. In a similar vein, it is unclear if a more efficient reduction could exist for the case of linear programs that are not polynomially bounded. We leave these as very interesting open problems.

Finally, our reductions do not preserve the "shape" of the linear program, in particular, a dense linear program may be reduced to a sparse 2-commodity flow problem with a similar number of edges as there are non-zero entries in the original program. It would be interesting to convert a dense linear program into a dense 2-commodity flow problem, e.g. to convert a linear program with m constraints and n variables (say, $m \leq n$) into a 2-commodity flow problem with $\widetilde{O}(n)$ edges and $\widetilde{O}(m)$ vertices.

2 Preliminaries

2.1 Notation

Matrices and Vectors. We use parentheses to denote entries of a matrix or a vector: Let $\boldsymbol{A}(i, j)$ denote the (i, j)th entry of a matrix $\boldsymbol{A}$, and let $\boldsymbol{x}(i)$ denote the ith entry of a vector $\boldsymbol{x}$. Given a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, we use $\boldsymbol{a}_i^\top$ to denote the ith row of a matrix $\boldsymbol{A}$ and $\mathrm{nnz}(\boldsymbol{A})$ to denote the number of nonzero entries of $\boldsymbol{A}$. Without loss of generality, we assume that $\mathrm{nnz}(\boldsymbol{A}) \geq \max\{m, n\}$. For any vector $\boldsymbol{x} \in \mathbb{R}^n$, we define $\|\boldsymbol{x}\|_{\max} = \max_{i \in [n]} |\boldsymbol{x}(i)|$, $\|\boldsymbol{x}\|_1 = \sum_{i \in [n]} |\boldsymbol{x}(i)|$. For any matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, we define $\|\boldsymbol{A}\|_{\max} = \max_{i,j} |\boldsymbol{A}(i, j)|$.

We define a function X that takes an arbitrary number of matrices $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_{k_1}$, vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{k_2}$, and scalars $K_1, \ldots, K_{k_3}$ as arguments, and returns the maximum of $\|\cdot\|_{\max}$ of all the arguments, i.e.,

$$\begin{aligned} &X(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_{k_1}, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{k_2}, K_1, \ldots, K_{k_3}) \\ &\qquad = \max\left\{\|\boldsymbol{A}_1\|_{\max}, \ldots, \|\boldsymbol{A}_{k_1}\|_{\max}, \|\boldsymbol{b}_1\|_{\max}, \ldots, \|\boldsymbol{b}_{k_2}\|_{\max}, |K_1|, \ldots, |K_{k_3}|\right\}. \end{aligned}$$

2.2 Problem Definitions

In this section, we formally define the problems that we use in the reduction. These problems fall into two categories: one is related to linear programming and linear equations, and the other is related to flow problems in graphs. In addition, we define the errors for approximately solving these problems.

2.2.1 Linear Programming and Linear Equations With Positive Variables

For the convenience of our reduction, we define linear programming as a "decision" problem. We can solve the optimization problem $\max\{\boldsymbol{c}^\top \boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}, \boldsymbol{x} \ge \boldsymbol{0}\}$ by binary searching its optimal value via the decision problem.

▶ **Definition 2.1** (Linear Programming (LP)). *Given a matrix $\boldsymbol{A} \in \mathbb{Z}^{m\times n}$, vectors $\boldsymbol{b} \in \mathbb{Z}^m$ and $\boldsymbol{c} \in \mathbb{Z}^n$, an integer K, and $R \ge \max\{1, \max\{\|\boldsymbol{x}\|_1 : \boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}, \boldsymbol{x} \ge \boldsymbol{0}\}\}$, we refer to the LP problem for $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R)$ as the problem of finding a vector $\boldsymbol{x} \in \mathbb{R}^n_{\ge 0}$ satisfying*

$$\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b} \text{ and } \boldsymbol{c}^\top \boldsymbol{x} \ge K$$

if such an $\boldsymbol{x}$ exists and returning "infeasible" otherwise.

We will reduce linear programming to linear equations with nonnegative variables (LEN), and then to linear equations with nonnegative variables and small integer coefficients (k-LEN).

▶ **Definition 2.2** (Linear Equations with Nonnegative Variables (LEN)). *Given $\boldsymbol{A} \in \mathbb{Z}^{m\times n}$, $\boldsymbol{b} \in \mathbb{Z}^m$, and $R \ge \max\{1, \max\{\|\boldsymbol{x}\|_1 : \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \ge \boldsymbol{0}\}\}$, we refer to the LEN problem for $(\boldsymbol{A}, \boldsymbol{b}, R)$ as the problem of finding a vector $\boldsymbol{x} \in \mathbb{R}^n_{\ge 0}$ satisfying $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ if such an $\boldsymbol{x}$ exists and returning "infeasible" otherwise.*

▶ **Definition 2.3** (k-LEN (k-LEN)). *The k-LEN problem is an LEN problem $(\boldsymbol{A}, \boldsymbol{b}, R)$ where the entries of $\boldsymbol{A}$ are integers in $[-k, k]$ for some given $k \in \mathbb{Z}_+$.*

We employ the following additive error notion. We append a letter "A" to each problem name to denote its approximation version, e.g., LP Approximate Problem is abbreviated to LPA.

▶ **Definition 2.4** (Approximation Errors). *We always require $\boldsymbol{x} \ge \boldsymbol{0}$. In addition,*
1. *Error in objective: $\boldsymbol{c}^\top \boldsymbol{x} \ge K$ is relaxed to $\boldsymbol{c}^\top \boldsymbol{x} \ge K - \epsilon$;*
2. *Error in constraint:*
   - a. *The inequality constraint $\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}$ is relaxed to $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} \le \epsilon \boldsymbol{1}$, where $\boldsymbol{1}$ is the all-1 vector;*
   - b. *The equality constraint $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is relaxed to $\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_\infty \le \epsilon$.*

Based on Definition 2.4, we can define the approximate version of LP. The approximate version of LEN and k-LEN can be found in the full version of the paper [6].

▶ **Definition 2.5** (LP Approximate Problem (LPA)). *An LPA instance is given by an LP instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R)$ and an error parameter $\epsilon \in [0, 1]$, which we collect in a tuple $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R, \epsilon)$. We say an algorithm solves the LPA problem, if, given any LPA instance, it returns a vector $\boldsymbol{x} \ge \boldsymbol{0}$ such that*

$$\boldsymbol{c}^\top \boldsymbol{x} \ge K - \epsilon$$
$$\boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b} + \epsilon \boldsymbol{1}$$

where $\boldsymbol{1}$ is the all-1 vector, or it correctly declares that the associated LP instance is infeasible.

2.2.2 Flow Problems

A *flow network* is a directed graph $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of edges, together with a vector of edge capacities $\boldsymbol{u} \in \mathbb{Z}^{|E|}_{>0}$ that upper bound the amount of flow passing each edge. A *2-commodity flow network* is a flow network together with two source-sink pairs $s_i, t_i \in V$ for each commodity $i \in \{1, 2\}$.

Given a 2-commodity flow network $(G = (V, E), \boldsymbol{u}, s_1, t_1, s_2, t_2)$, a *feasible 2-commodity flow* is a pair of flows $\boldsymbol{f}_1, \boldsymbol{f}_2 \in \mathbb{R}^{|E|}_{\ge 0}$ that satisfies

1. capacity constraint: $\boldsymbol{f}_1(e) + \boldsymbol{f}_2(e) \leq \boldsymbol{u}(e),\ \forall e \in E$, and
2. conservation of flows: $\sum_{u:(u,v)\in E} \boldsymbol{f}_i(u,v) = \sum_{w:(v,w)\in E} \boldsymbol{f}_i(v,w),\ \forall i \in \{1,2\}, v \in V \setminus \{s_i, t_i\}$[2].

Similar to the definition of LP, we define 2-commodity flow problem as a decision problem. We can solve a decision problem by solving the corresponding optimization problem.

▶ **Definition 2.6** (2-Commodity Flow Problem (2CF)). *Given a 2-commodity flow network $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2)$ together with $R \geq 0$, we refer to the 2CF problem for $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2, R)$ as the problem of finding a feasible 2-commodity flow $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq \boldsymbol{0}$ satisfying*

$$F_1 + F_2 \geq R$$

if such flows exist and returning "infeasible" otherwise.

To reduce LP to 2CF, we need a sequence of variants of flow problems.

▶ **Definition 2.7** (2-Commodity Flow with Required Flow Amount (2CFR)). *Given a 2-commodity flow network $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2)$ together with $R_1, R_2 \geq 0$, we refer to the 2CFR for $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2, R_1, R_2)$ as the problem of finding a feasible 2-commodity flow $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq \boldsymbol{0}$ satisfying*

$$F_1 \geq R_1, \quad F_2 \geq R_2$$

if such flows exist and returning "infeasible" otherwise.

▶ **Definition 2.8** (Fixed Flow Constraints). *Given a set $F \subseteq E$ in a 2-commodity flow network, we say the flows $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq \boldsymbol{0}$ satisfy* fixed flow constraints on F *if*

$$\boldsymbol{f}_1(e) + \boldsymbol{f}_2(e) = \boldsymbol{u}(e),\ \forall e \in F.$$

Similarly, given a set $F \subseteq E$ in a 1-commodity flow network, we say the flow $\boldsymbol{f} \geq \boldsymbol{0}$ satisfies fixed flow constraints on F *if*

$$\boldsymbol{f}(e) = \boldsymbol{u}(e),\ \forall e \in F.$$

▶ **Definition 2.9** (2-Commodity Fixed Flow Problem (2CFF)). *Given a 2-commodity flow network $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2)$ together with a subset of edges $F \subseteq E$, we refer to the 2CFF problem for the tuple $(G, F, \boldsymbol{u}, s_1, t_1, s_2, t_2)$ as the problem of finding a feasible 2-commodity flow $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq \boldsymbol{0}$ which also satisfies the fixed flow constraints on F if such flows exist and returning "infeasible" otherwise.*

▶ **Definition 2.10** (Selective Fixed Flow Problem (SFF)). *Given a 2-commodity network $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2)$ together with three edge sets $F, S_1, S_2 \subseteq E$, we refer to the SFF problem for $(G, F, S_1, S_2, \boldsymbol{u}, s_1, t_1, s_2, t_2)$ as the problem of finding a feasible 2-commodity flow $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq \boldsymbol{0}$ such that for each $i \in \{1,2\}$, flow $\boldsymbol{f}_i(e) > 0$ only if $e \in S_i$, and $\boldsymbol{f}_1, \boldsymbol{f}_2$ satisfy the fixed flow constraints on F, if such flows exist, and returning "infeasible" otherwise.*

▶ **Definition 2.11** (Fixed Homologous Flow Problem (FHF)). *Given a flow network with a single source-sink pair $(G, \boldsymbol{u}, s, t)$ together with a collection of disjoint subsets of edges $\mathcal{H} = \{H_1, \ldots, H_h\}$ and a subset of edges $F \subseteq E$ such that F is disjoint from all the sets in $\mathcal{H}$, we refer to the FHF problem for $(G, F, \mathcal{H}, \boldsymbol{u}, s, t)$ as the problem of finding a feasible flow $\boldsymbol{f} \geq \boldsymbol{0}$ such that*

[2] Note that for commodity i, this constraint includes the case of $v \in \{s_{\bar{i}}, t_{\bar{i}}\}, \bar{i} = \{1,2\} \backslash i$.

$$\boldsymbol{f}(e_1) = \boldsymbol{f}(e_2), \quad \forall e_1, e_2 \in H_k, 1 \leq k \leq h,$$

and $\boldsymbol{f}$ satisfies the fixed flow constraints on F, if such flows exist, and returning "infeasible" otherwise.

▶ **Definition 2.12** (Fixed Pair Homologous Flow Problem (FPHF)). *An FPHF is an FHF problem $(G, F, \mathcal{H}, \boldsymbol{u}, s, t)$ where every set in $\mathcal{H}$ has size 2.*

Now, we define errors for the above flow problems.

▶ **Definition 2.13** (Approximation Errors). *We always require flows $\boldsymbol{f} \geq \boldsymbol{0}$ and $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq \boldsymbol{0}$. In addition,*

1. *Error in congestion: the capacity constraints are relaxed to:*

   $$\boldsymbol{f}_1(e) + \boldsymbol{f}_2(e) \leq \boldsymbol{u}(e) + \epsilon, \quad \forall e \in E.$$

   There are several variants corresponding to different flow problems.

   a. *If $e \in F$ is a fixed-flow edge, the fixed-flow constraints are relaxed to*

   $$\boldsymbol{u}(e) - \epsilon \leq \boldsymbol{f}_1(e) + \boldsymbol{f}_2(e) \leq \boldsymbol{u}(e) + \epsilon, \quad \forall e \in F$$

   b. *If G is a 1-commodity flow network, we replace $\boldsymbol{f}_1(e) + \boldsymbol{f}_2(e)$ by $\boldsymbol{f}(e)$.*

2. *Error in demand: the conservation of flows is relaxed to*

   $$\left| \sum_{u:(u,v)\in E} \boldsymbol{f}_i(u,v) - \sum_{w:(v,w)\in E} \boldsymbol{f}_i(v,w) \right| \leq \epsilon, \ \forall v \in V \backslash \{s_i, t_i\}, i \in \{1,2\} \qquad (1)$$

   There are several variants of this constraint corresponding to different flow problems.

   a. *If the problem is with flow requirement F_i, then besides Eq. (1), we add demand constraints for s_i and t_i with respect to commodity i:*

   $$\left| \sum_{w:(s_i,w)\in E} \boldsymbol{f}_i(s_i,w) - F_i \right| \leq \epsilon, \qquad \left| \sum_{u:(u,t_i)\in E} \boldsymbol{f}_i(u,t_i) - F_i \right| \leq \epsilon, \ i \in \{1,2\} \qquad (2)$$

   b. *If G is a 1-commodity flow network, Eq. (1) can be simplified as*

   $$\left| \sum_{u:(u,v)\in E} \boldsymbol{f}(u,v) - \sum_{w:(v,w)\in E} \boldsymbol{f}(v,w) \right| \leq \epsilon, \ \forall v \in V \backslash \{s,t\}$$

3. *Error in type: the selective constraints are relaxed to*

   $$\boldsymbol{f}_{\bar{i}}(e) \leq \epsilon, \quad \forall e \in S_i, \ \bar{i} = \{1,2\} \backslash i.$$

4. *Error in (pair) homology: the (pair) homologous constraints are relaxed to*

   $$|\boldsymbol{f}(e_1) - \boldsymbol{f}(e_2)| \leq \epsilon, \ \forall e_1, e_2 \in H_k, H_k \in \mathcal{H}.$$

Based on Definition 2.13, we define the approximate version of 2CF. Again, the approximate version of the rest flow problems can be found in the full version of the paper [6].

▶ **Definition 2.14** (2CF Approximate Problem (2CFA)). *A 2CFA instance is given by a 2CF instance $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2, R)$ and an error parameter $\epsilon \in [0,1]$, which we collect in a tuple $(G, \boldsymbol{u}, s_1, t_1, s_2, t_2, R, \epsilon)$. We say an algorithm solves the 2CFA problem, if, given any 2CFA instance, it returns a pair of flows $\boldsymbol{f}_1, \boldsymbol{f}_2 \geq 0$ that satisfies*

$$\boldsymbol{f}_1(e) + \boldsymbol{f}_2(e) \leq \boldsymbol{u}(e) + \epsilon, \ \forall e \in E \tag{3}$$

$$\left| \sum_{u:(u,v)\in E} \boldsymbol{f}_i(u,v) - \sum_{w:(v,w)\in E} \boldsymbol{f}_i(v,w) \right| \leq \epsilon, \ \forall v \in V \backslash \{s_i, t_i\}, i \in \{1,2\} \tag{4}$$

$$\left| \sum_{w:(s_i,w)\in E} \boldsymbol{f}_i(s_i,w) - F_i \right| \leq \epsilon, \qquad \left| \sum_{u:(u,t_i)\in E} \boldsymbol{f}_i(u,t_i) - F_i \right| \leq \epsilon, \ i \in \{1,2\} \tag{5}$$

where $F_1 + F_2 = R$ [3]; or it correctly declares that the associated 2CF instance is infeasible. We refer to the error in (3) as error in congestion, error in (4) and (5) as error in demand.

3 Main Results

▶ **Theorem 3.1.** *Given an LPA instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R, \epsilon^{lp})$ where $\boldsymbol{A} \in \mathbb{Z}^{m\times n}, \boldsymbol{b} \in Z^m, \boldsymbol{c} \in \mathbb{Z}^n, K \in \mathbb{Z}, \epsilon^{lp} \geq 0$ and $\boldsymbol{A}$ has $\mathrm{nnz}(\boldsymbol{A})$ nonzero entries, we can reduce it to a 2CFA instance $(G = (V,E), \boldsymbol{u}, s_1, t_1, s_2, t_2, R^{2cf}, \epsilon^{2cf})$ in time $O(\mathrm{nnz}(\boldsymbol{A}) \log X)$ where $X = X(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K)$, such that*

$$|V|, |E| = O(\mathrm{nnz}(\boldsymbol{A}) \log X),$$
$$\|\boldsymbol{u}\|_{\max}, R^{2cf} = O(\mathrm{nnz}^3(\boldsymbol{A}) R X^2 \log^2 X),$$
$$\epsilon^{2cf} = \Omega\left(\frac{1}{\mathrm{nnz}^7(\boldsymbol{A}) R X^3 \log^6 X}\right) \epsilon^{lp}.$$

If the LP instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R)$ has a solution, then the 2CF instance $(G^{2cf}, \boldsymbol{u}^{2cf}, s_1, t_1, s_2, t_2, R^{2cf})$ has a solution. Furthermore, if $\boldsymbol{f}^{2cf}$ is a solution to the 2CFA (2CF) instance, then in time $O(\mathrm{nnz}(\boldsymbol{A}) \log X)$, we can compute a solution $\boldsymbol{x}$ to the LPA (LP, respectively) instance, where the exact case holds when $\epsilon^{2cf} = \epsilon^{lp} = 0$.

Our main theorem immediately implies the following corollary.

▶ **Corollary 3.2.** *If we can solve any 2CFA instance $(G = (V,E), \boldsymbol{u}, s_1, t_1, s_2, t_2, R^{2cf}, \epsilon)$ in time $O\left(|E|^c \operatorname{poly}\log\left(\frac{\|\boldsymbol{u}\|_1}{\epsilon}\right)\right)$ for some small constant $c \geq 1$, then we can solve any LPA instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R, \epsilon)$ in time $O\left(\mathrm{nnz}^c(\boldsymbol{A}) \operatorname{poly}\log\left(\frac{\mathrm{nnz}(\boldsymbol{A}) R X(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K)}{\epsilon}\right)\right)$.*

3.1 Overview of Our Proof

In this section, we will explain how to reduce an LP instance to a 2-commodity flow (2CF) instance by a chain of efficient reductions. In each step, we reduce a decision problem P to a decision problem Q. We guarantee that (1) the reduction runs in nearly-linear time[4], (2) the

[3] If we encode 2CF as an LP instance, and approximately solve the LP with at most ϵ additive error. Then, the approximate solution also agrees with the error notions of 2CF, except that we get $F_1 + F_2 \geq R - \epsilon$ instead of $F_1 + F_2 \geq R$. This inconsistency can be eliminated by setting $\epsilon' = 2\epsilon$, and slightly adjusting F_1, F_2 to F_1', F_2' such that $F_1' + F_2' \geq R$. This way, we obtain an approximate solution to 2CF with at most ϵ' additive error.

[4] Linear in the size of problem P, poly-logarithmic in the maximum magnitude of all the numbers that describe P, the feasible set radius, and the inverse of the error parameter if an approximate solution is allowed.

Table 1 A summary of notation used in the reduction from LP to 2CF.

Exact problems	Input	Output
LP (Def. 2.1)	$\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R$	$\boldsymbol{x}$
LEN (Def. 2.2)	$\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{R}$	$\tilde{\boldsymbol{x}}$
2-LEN (Def. 2.3)	$\bar{\boldsymbol{A}}, \bar{\boldsymbol{b}}, \bar{R}$	$\bar{\boldsymbol{x}}$
1-LEN (Def. 2.3)	$\hat{\boldsymbol{A}}, \hat{\boldsymbol{b}}, \hat{R}$	$\hat{\boldsymbol{x}}$
FHF (Def. 2.11)	$G^h, F^h, \mathcal{H}^h = \{H_1, \cdots, H_h\}, \boldsymbol{u}^h, s, t$	$\boldsymbol{f}^h$
FPHF (Def. 2.12)	$G^p, F^p, \mathcal{H}^p = \{H_1, \cdots, H_p\}, \boldsymbol{u}^p, s, t$	$\boldsymbol{f}^p$
SFF (Def. 2.10)	$G^s, F^s, S_1, S_2, \boldsymbol{u}^s, s_1, t_1, s_2, t_2$	$\boldsymbol{f}^s$
2CFF (Def. 2.9)	$G^f, F^f, \boldsymbol{u}^f, s_1, t_1, s_2, t_2$	$\boldsymbol{f}^f$
2CFR (Def. 2.7)	$G^r, \boldsymbol{u}^r, \bar{s}_1, \bar{t}_1, \bar{s}_2, \bar{t}_2, R_1, R_2$	$\boldsymbol{f}^r$
2CF (Def. 2.6)	$G^{2cf}, \boldsymbol{u}^{2cf}, \bar{\bar{s}}_1, \bar{\bar{t}}_1, \bar{\bar{s}}_2, \bar{\bar{t}}_2, R^{2cf}$	$\boldsymbol{f}^{2cf}$

size of Q is nearly-linear in the size of P, and (3) that P is feasible implies that Q is feasible, and an approximate solution to Q can be turned to an approximate solution to P with only a polynomial blow-up in error parameters, in linear time.

We follow the outline of Itai's reduction [17]. A summary of the problem notation used in the reduction from LP to 2CF is given in Table 1. Itai first reduced an LP instance to a 1-LEN instance (linear equations with nonnegative variables and ± 1 coefficients). A 1-LEN instance can be cast as a single-commodity flow problem subject to additional homologous constraints and fixed flow constraints (i.e., FHF). Then, Itai dropped these additional constraints step by step, via introducing a second commodity of flow and imposing lower bound requirements on the total amount of flows routed between the source-sink pairs. However, in the worst case, Itai's reduction from 1-LEN to FHF enlarges the problem size quadratically and is thus inefficient. One of our main contributions is to improve this step so that the problem representation size is preserved along the reduction chain.

Our second main contribution is an upper bound on the errors accumulated during the process of mapping an approximate solution to the 2CF instance to an approximate solution to the LP instance. We show that the error only grows by polynomial factors. Itai only considered exact solutions between these two instances.

We will explain the reductions based on the exact versions of the problems. At the end of this section, we will discuss some intuitions of our error analysis.

3.1.1 Reducing Linear Programming to Linear Equations With Nonnegative Variables and ± 1 Coefficients

Given an LP instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, K, R)$ where $R \geq \max\{1, \max\{\|x\|_1 : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}\}\}$, we want to compute a vector $\boldsymbol{x} \geq \boldsymbol{0}$ satisfying

$$\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{c}^\top \boldsymbol{x} \geq K$$

or to correctly declare infeasible. We introduce slack variables $\boldsymbol{s}, \alpha \geq \boldsymbol{0}$ and turn the above inequalities to equalities:

$$\boldsymbol{A}\boldsymbol{x} + \boldsymbol{s} = \boldsymbol{b}, \boldsymbol{c}^\top \boldsymbol{x} - \alpha = K$$

which is an LEN instance $(\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{b}}, \tilde{R})$. Comparing to Itai's proof, we need to track two additional parameters: the polytope radius R and the maximum magnitude of the input entries X.

We then reduce the LEN instance to linear equations with ± 2 coefficients (2-LEN) by bitwise decomposition. For each bit, we introduce a carry term. Different from Itai's reduction, we impose an upper bound for each carry variable. We show that this upper bound does not change problem feasibility and it guarantees that the polytope radius only increases polynomially. The following example demonstrates this process.

$$5x_1 + 3x_2 - 7x_3 = -1 \quad \Rightarrow \quad (x_1 + x_2 - x_3)2^0 + (x_2 - x_3)2^1 + (x_1 - x_3)2^2 = -1 \cdot 2^0$$

It can be decomposed to 3 linear equations, together with carry terms $(c_i - d_i)$, where $c_i, d_i \geq 0$:

$$\begin{aligned} x_1 + x_2 - x_3 - 2(c_0 - d_0) &= -1 \\ x_2 - x_3 + (c_0 - d_0) - 2(c_1 - d_1) &= 0 \\ x_1 - x_3 + (c_1 - d_1) &= 0 \end{aligned}$$

Next, we reduce the 2-LEN instance $(\bar{\boldsymbol{A}}, \bar{\boldsymbol{b}}, \bar{R})$ to a 1-LEN instance $(\hat{\boldsymbol{A}}, \hat{\boldsymbol{b}}, \hat{R})$ by replacing each ± 2 coefficient variable with two new equal-valued variables.

All the above three reduction steps run in nearly-linear time, and the problem sizes increase nearly-linearly.

3.1.2 Reducing Linear Equations With Nonnegative Variables and ± 1 Coefficients to Fixed Homologous Flow Problem

One of our main contributions is a linear-time reduction from 1-LEN to FHF (single-commodity fixed homologous flow problem). Our reduction is similar to Itai's reduction, but more efficient.

Itai observed that a single linear equation $\hat{\boldsymbol{a}}^\top \hat{\boldsymbol{x}} = \hat{b}$ with ± 1 coefficients can be represented as a fixed homologous flow network. We improve his construction by creating a *sparser* flow network in which the number of edges is proportional to the number of nonzero coefficients in the linear equations. Figure 1 depicts our gadget. Our gadget has a source vertex s, a sink vertex t, and two additional vertices J^+ and J^-. Each variable $\hat{\boldsymbol{x}}(i)$ with coefficient $\hat{\boldsymbol{a}}(i) \neq 0$ corresponds to an edge: There is an edge from s to J^+ if $\hat{\boldsymbol{a}}(i) = 1$, and there is an edge from s to J^- if $\hat{\boldsymbol{a}}(i) = -1$. The amount of flow passing this edge encodes the value of $\hat{\boldsymbol{x}}(i)$. Thus, the difference between the total amount of flow entering J^+ and that entering J^- equals to $\hat{\boldsymbol{a}}^\top \hat{\boldsymbol{x}}$. To force $\hat{\boldsymbol{a}}^\top \hat{\boldsymbol{x}} = \hat{b}$, we add two edges e_1, e_2 from J^+ to t and one edge e_3 from J^- to t; we require e_1 and e_3 to be homologous and require e_2 to be a fixed flow with value $\hat{b}$.

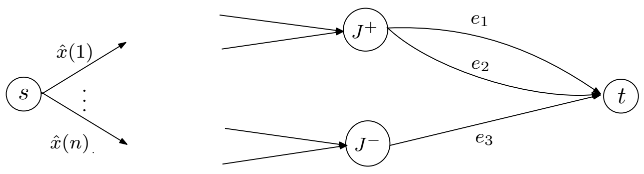

Figure 1 The gadget of reducing a single linear equation $\hat{\boldsymbol{a}}^\top \hat{\boldsymbol{x}} = \hat{b}$ with ± 1 coefficients to a fixed homologous flow network (FHF).

We can generalize this construction to encode a system of linear equations $\hat{\boldsymbol{A}}\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}}$, where $\hat{\boldsymbol{A}} \in \boldsymbol{Z}^{\hat{m} \times \hat{n}}, \hat{\boldsymbol{b}} \in \boldsymbol{Z}^{\hat{m}}$. Specifically, we create a gadget as above for each individual equation, and then glue all the source (sink) vertices for all the equations together as the source (sink, respectively) of the graph (see Figure 2). In addition to requiring e_i^+, e_i^- to be homologous for each single linear equation, to guarantee the variable values to be consistent in these

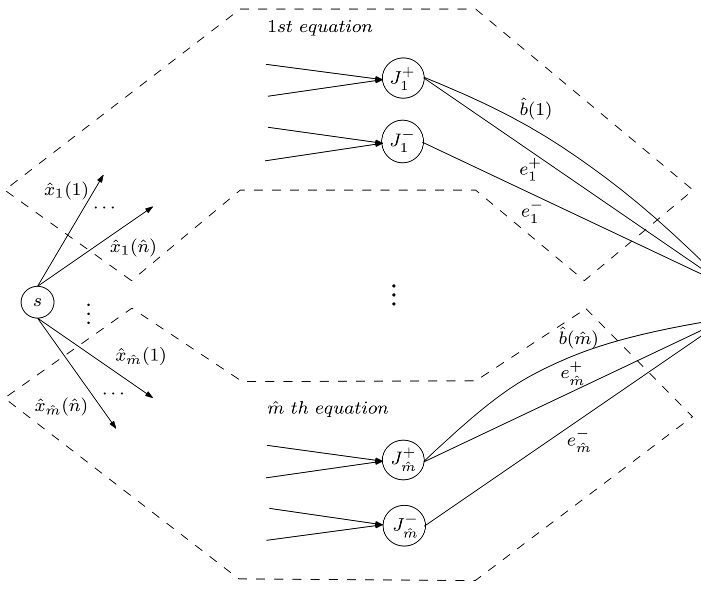

Figure 2 The reduction from 1-LEN to FHF.

equations, we also require the edges corresponding to the same variable $\hat{\boldsymbol{x}}(i)$ in different equations to be homologous, i.e., $\{\hat{x}_1(i), \ldots, \hat{x}_{\hat{m}}(i)\}$. We can check that the number of the vertices is linear in the number of equations; the number of the edges and the total size of the homologous sets are both linear in the number of nonzero coefficients of the linear equation system.

3.1.3 Dropping the Homologous and Fixed Flow Constraints

To reduce FHF to 2CF (2-commodity flow problem), we need to drop the homologous and fixed flow constraints. The reduction has three main steps.

Reducing FHF to SFF. Given an FHF instance, we can reduce it to a fixed homologous flow instance in which each homologous edge set has size 2 (FPHF). To drop the homologous requirement in FPHF, we introduce a second commodity of flow with source-sink pair (s_2, t_2), and for each edge, we carefully select the type(s) of flow that can pass through this edge. Specifically, given two homologous edges (v, w) and (y, z), we construct a constant-sized gadget (see Figure 3): We introduce new vertices vw, vw', yz, yz', construct a directed path

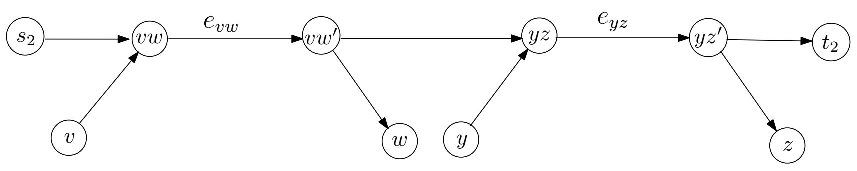

Figure 3 The gadget of reducing a pair of homologous edges $(v, w), (y, z)$ to a selective fixed flow network (SFF).

$P : s_2 \to vw \to vw' \to yz \to yz' \to t_2$, and add edges $(v, vw), (vw', w)$ and $(y, yz), (yz', z)$. Now, there is a directed path $P_{vw} : v \to vw \to vw' \to w$ and a directed path $P_{yz} : y \to yz \to yz' \to z$. Paths P and P_{vw} (P_{yz}) share an edge $e_{vw} = (vw, vw')$ $(e_{yz} = (yz, yz')$, respectively). We select e_{vw} and e_{yz} for both flow $\boldsymbol{f}_1^s$ and $\boldsymbol{f}_2^s$, select the rest of the edges along P for only $\boldsymbol{f}_2^s$, and select the rest of the edges along P_{vw}, P_{yz} for only $\boldsymbol{f}_1^s$. By this construction, in this gadget, we have $\boldsymbol{f}_2^s(e_{vw}) = \boldsymbol{f}_2^s(e_{yz})$ being the amount of flow routed in P, $\boldsymbol{f}_1^s(e_{vw})$ and $\boldsymbol{f}_1^s(e_{yz})$ being the amount of flow routed in P_{vw} and P_{yz}, respectively. Next, we choose e_{vw} and e_{yz} to be fixed flow edges with equal capacity; this guarantees the same amount of $\boldsymbol{f}_1^s$ is routed through P_{vw} and P_{yz}. The new graph is an SFF instance.

Reducing SFF to 2CFF. Next, we will drop the selective requirement of the SFF instance. For each edge (x, y) selected for flow i, we construct a constant-sized gadget (see Figure 4): We introduce two vertices xy, xy', construct a direct path $s_i \to xy' \to xy \to t_i$, and add

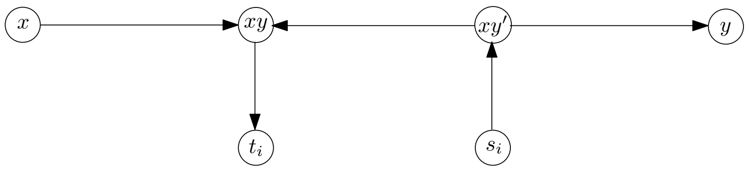

Figure 4 The gadget of reducing a selective edge (x, y) for commodity i to a 2-commodity fixed flow network (2CFF).

edge (x, xy) and (xy', y). This gadget simulates a directed path from x to y for flow $\boldsymbol{f}_i^f$, and guarantees no directed path from x to y for flow $\boldsymbol{f}_{\bar{i}}^f$ so that $\boldsymbol{f}_{\bar{i}}^f$ cannot be routed from x to y. We get a 2CFF instance.

Reducing 2CFF to 2CF. It remains to drop the fixed flow constraints. The gadget we will use is similar to that used in the last step. We first introduce new sources $\bar{s}_1, \bar{s}_2$ and sinks $\bar{t}_1, \bar{t}_2$. Then, for each edge (x, y) with capacity u, we construct a constant-sized gadget (see Figure 5).

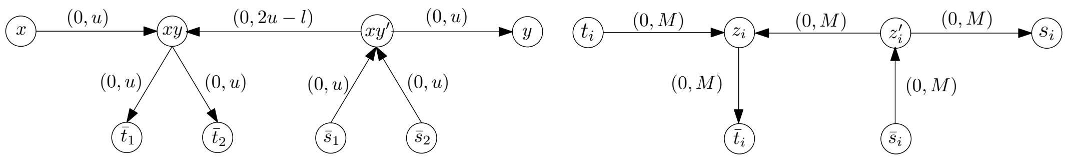

Figure 5 The gadget of reducing 2CFF to a 2-commodity flow with required flow amount (2CFR). l, u are lower and upper edge capacity of (x, y), respectively: $l = u$ if (x, y) is a fixed flow edge; $l = 0$ if (x, y) is a non-fixed flow edge.

We introduce two vertices xy, xy', add edges $(\bar{s}_1, xy'), (\bar{s}_2, xy'), (xy, \bar{t}_1), (xy, \bar{t}_2), (xy', xy)$, and $(x, xy), (xy', y)$. This simulates a directed path from x to y that both flow $\boldsymbol{f}_1^r$ and $\boldsymbol{f}_2^r$ can pass through. We let (xy', xy) have capacity u if (x, y) is a fixed flow edge and $2u$ otherwise; we let all the other edges have capacity u. Assume all the edges incident to the sources and the sinks are saturated, then the total amount of flows routed from x to y in this gadget must be u if (x, y) is a fixed flow edge and no larger than u otherwise. Moreover, since the original sources and sinks are no longer sources and sinks now, we have to satisfy the conservation of flows at these vertices. For each $i \in \{1, 2\}$, we create a similar gadget involving $\bar{s}_i, \bar{t}_i$ to simulate a directed path from t_i to s_i (the original sink and source), and let the edges incident to $\bar{s}_i, \bar{t}_i$ have capacity M, the sum of all the edge capacities in the 2CFF instance. This gadget guarantees that assuming the edges incident to $\bar{s}_i$ and $\bar{t}_i$ are saturated, the amount of flow routed from t_i to s_i through this gadget can be any number at most M. To force the above edge-saturation assumptions to hold, we require the amount of flow $\boldsymbol{f}_i^r$ routed from $\bar{s}_i$ to $\bar{t}_i$ to be no less than $2M$ for each $i \in \{1, 2\}$.

Now, this instance is close to a 2CF instance except that we require a lower bound for each flow value instead of a lower bound for the sum of two flow values. To handle this, we introduce new sources $\bar{\bar{s}}_1, \bar{\bar{s}}_2$ and for each $i \in \{1, 2\}$, we add an edge $(\bar{\bar{s}}_i, \bar{s}_i)$ with capacity $2M$, the lower bound required for the value of $\boldsymbol{f}_i^r$.

One can check that in each reduction step, the reduction time is nearly linear and the problem size increases nearly linearly. In addition, given a solution to the 2CF instance, one can construct a solution to the LP instance in nearly linear time.

3.1.4 Computing an Approximate Linear Program Solution From an Approximate 2-Commodity Flow Solution

We establish an error bound for mapping an approximate solution to 2CFA to an approximate solution to LPA. Below we outline the intuition behind our error analysis for flow problems. We will keep track of multiple types of error (e.g., error in congestion, demand, selective types, and homology depending on the problem settings). Now suppose we reduce problem P to problem Q using a certain gadget, and then we map a solution to Q back to a solution to P. We observe that each error notion of P is an additive accumulation of the multiple error notions of Q. This is because we have to map the flows of Q passing through a gadget including multiple edges back to a flow of P passing through a single edge. Each time we remove an edge, various errors related to this edge and incident vertices get transferred to its neighbors. Thus, the total error accumulation by the solution mapping can be polynomially bounded by the number of edges. So, the final error only increases polynomially.

References

1 Cynthia Barnhart, Niranjan Krishnan, and Pamela H. Vance. Multicommodity flow problems. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 2354–2362. Springer US, Boston, MA, 2009. doi:10.1007/978-0-387-74759-0_407.

2 L. Chen, G. Goranci, M. Henzinger, R. Peng, and T. Saranurak. Fast Dynamic Cuts, Distances and Effective Resistances via Vertex Sparsifiers. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146, November 2020. doi:10.1109/FOCS46700.2020.00109.

3 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.

4 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.

5 James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, 2007.

6 Ming Ding, Rasmus Kyng, and Peng Zhang. Two-commodity flow is equivalent to linear programming under nearly-linear time reductions. *arXiv preprint*, 2022. arXiv:2201.11587.

7 Efim A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

8 James R. Evans. A combinatorial equivalence between A class of multicommodity flow problems and the capacitated transportation problem. *Mathematical Programming*, 10(1):401–404, December 1976. doi:10.1007/BF01580684.

9 James R. Evans. The simplex method for integral multicommodity networks. *Naval Research Logistics Quarterly*, 25(1):31–37, March 1978. doi:10.1002/nav.3800250104.

10 Shimon Even and R. Endre Tarjan. Network flow and testing graph connectivity. *SIAM journal on computing*, 4(4):507–518, 1975.

11 Lisa K. Fleischer. Approximating Fractional Multicommodity Flow Independent of the Number of Commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, January 2000. doi:10.1137/S0895480199355754.

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

13 Yu Gao, Yang P. Liu, and Richard Peng. Fully Dynamic Electrical Flows: Sparse Maxflow Faster Than Goldberg-Rao. *arXiv:2101.07233 [cs]*, January 2021. arXiv:2101.07233.

14 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, January 2007. doi:10.1137/S0097539704446232.

15 Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, September 1998. doi:10.1145/290179.290181.

16 T. C. Hu. Multi-Commodity Network Flows. *Operations Research*, 11(3):344–360, June 1963. doi:10.1287/opre.11.3.344.

17 Alon Itai. Two-commodity flow. *Journal of the ACM (JACM)*, 25(4):596–611, 1978.

18 Arun Jambulapati and Aaron Sidford. Ultrasparse Ultrasparsifiers and Faster Laplacian System Solvers. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 540–559. SIAM, 2021.

19 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.

20 Tarun Kathuria, Yang P. Liu, and Aaron Sidford. Unit Capacity Maxflow in Almost $ O (m$\hat{}${$4/3$}$) $ Time. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 119–130. IEEE, 2020.

21 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.

22 Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, pages 911–920, 2013.

23 Jeff L. Kennington. A Survey of Linear Cost Multicommodity Network Flows. *Operations Research*, 26(2):209–236, 1978.

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

25 Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching Optimality for Solving SDD Linear Systems. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, pages 235–244, USA, October 2010. IEEE Computer Society. doi:10.1109/FOCS.2010.29.

26 Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly-m log n time solver for sdd linear systems. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 590–598. IEEE, 2011.

27 Rasmus Kyng, Richard Peng, Sushant Sachdeva, and Di Wang. Flows in almost linear time via adaptive preconditioning. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 902–913, 2019.

28 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.

29 Rasmus Kyng, Di Wang, and Peng Zhang. Packing LPs are hard to solve accurately, assuming linear equations are hard. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 279–296. SIAM, 2020.

30 Rasmus Kyng and Peng Zhang. Hardness results for structured linear systems. *SIAM Journal on Computing*, 49(4):FOCS17–280, 2020.

31 Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, pages 755–764, 2013.

32 Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in o (vrank) iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433. IEEE, 2014.

33 T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast Approximation Algorithms for Multicommodity Flow Problems. *Journal of Computer and System Sciences*, 50(2):228–243, April 1995. doi:10.1006/jcss.1995.1020.

34 Yang P. Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 803–814, 2020.

35 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*, STOC '10, pages 121–130, New York, NY, USA, June 2010. Association for Computing Machinery. doi:10.1145/1806689.1806708.

36 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.

37 Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016.

38 Thomas Magnanti, R Ahuja, and J Orlin. Network flows: theory, algorithms, and applications. *PrenticeHall, Upper Saddle River, NJ*, 1993.

39 Cameron Musco, Praneeth Netrapalli, Aaron Sidford, Shashanka Ubaru, and David P. Woodruff. Spectrum Approximation Beyond Fast Matrix Multiplication: Algorithms and Hardness. *arXiv:1704.04163 [cs, math]*, January 2019. arXiv:1704.04163.

40 A. Ouorou, P. Mahey, and J.-Ph. Vial. A Survey of Algorithms for Convex Multicommodity Flow Problems. *Management Science*, 46(1):126–147, 2000.

41 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.

42 Richard Peng and Daniel A. Spielman. An efficient parallel solver for SDD linear systems. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, pages 333–342, 2014.

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

44 James Renegar. Incorporating Condition Measures into the Complexity Theory of Linear Programming. *SIAM Journal on Optimization*, 5(3):506–524, August 1995. doi:10.1137/0805026.

45 Jonah Sherman. Nearly Maximum Flows in Nearly Linear Time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 263–269, October 2013. doi:10.1109/FOCS.2013.36.

46 Jonah Sherman. Area-convexity, $l\infty$ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 452–460, 2017.

47 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*, STOC '04, pages 81–90, New York, NY, USA, June 2004. Association for Computing Machinery. doi:10.1145/1007352.1007372.

48 Pravin M. Vaidya. Speeding-up linear programming using fast matrix multiplication. In *30th Annual Symposium on Foundations of Computer Science*, pages 332–337. IEEE Computer Society, 1989.

49 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*, STOC 2021, pages 859–869, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3406325.3451108.

50 I.-Lin Wang. Multicommodity network flows: A survey, Part I: Applications and Formulations. *International Journal of Operations Research*, 15(4):145–153, 2018.

51 Virginia Vassilevska Williams and R. Ryan Williams. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *Journal of the ACM*, 65(5):27:1–27:38, August 2018. doi:10.1145/3186893.