Experimental Analysis of LP Scaling Methods Based on Circuit Imbalance Minimization

Jakub Komárek
Computer Science Institute, Charles University, Prague, Czech Republic

Martin Koutecký
Computer Science Institute, Charles University, Prague, Czech Republic

Abstract

Linear programming (LP) is a fundamental problem with rich theory and wide applications. A ubiquitous technique in LP is *scaling*, where the input instance is transformed in some way to make its solution easier. Dadush et al. [STOC '20] have recently devised an algorithm which scales the columns of the constraint matrix of a linear program in a way that aims to minimize the *circuit imbalance measure*, a matrix condition number of growing theoretical interest. They show that this rescaling achieves favorable theoretical guarantees for certain LP algorithms.

We follow up on their work in an experimental manner. First, we have implemented their algorithm, overcoming several engineering obstacles. Next, we have used our implementation to obtain a rescaling of 142 publicly available instances. Finally, we have performed experiments evaluating the effects of the obtained rescalings on the runtime of real-world LP solvers, and we have evaluated their quality with regard to the circuit imbalance measure.

2012 ACM Subject Classification Theory of computation → Linear programming

Keywords and phrases Linear programming, scaling, circuit imbalance measure

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.18

Related Version *Full Version*: `https://dspace.cuni.cz/handle/20.500.11956/184426` [17]

Supplementary Material *Software (Source code and experiment data)*: `https://github.com/Jakoma02/lp-scaling-analysis`, archived at `swh:1:dir:0a90ab07fada3a81f5b7cf470f48db5a951dc655`

Funding Partially supported by project 22-22997S of GA ČR. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.
Martin Koutecký: Supported by Charles Univ. project UNCE 24/SCI/008.

1 Introduction

Linear programming (LP) is a core topic in theoretical computer science and optimization with many applications in practice. One of the possible approaches to improve LP algorithms is the use of *preconditiong* [7] or *scaling* [19, Section 7.2]. Dadush et al. [9] have shown that the complexity of the LP interior-point method of Vavasis and Ye [23] can be bounded in terms of a certain condition number κ of the constraint matrix A. This condition number can be changed by scaling individual columns of A by positive real values, and they present an algorithm which finds a positive diagonal matrix D such that the κ of the rescaled matrix approximates the minimum achievable value.

We follow up on their work by exploring its applicability in practice and describing the effects of the condition number on the performance of practical LP solvers, including ones not based on interior-point methods. The theoretical results of Dadush et al. [9] only directly apply to the particular interior-point method of Vavasis and Ye [23], so why do we test whether it speeds up other methods? We acknowledge this apparent "irrelevance";

our motivation is simply that the rescaling of Dadush et al. [9] is a significant theoretical contribution, and it is not *a priori* obvious whether and how it would affect the performance of real-world solvers.

1.1 The Circuit Imbalance Measure

The matrix condition numbers κ and κ^* as defined by Dadush et al. [9] are center to our work. Consider an LP $\min\{\mathbf{c}^\top \mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$. A *circuit* of the vector matroid of the columns of A is an inclusion-wise minimal set of indices of linearly dependent columns. For every circuit C of the vector matroid of A, we can consider its *elementary vector*, which is an element of the kernel of A that has C as its support. The *circuit imbalance measure* κ of matrix A is the maximum ratio of absolute values of nonzero elements over all elementary vectors of the vector matroid of A.

Scaling the columns of A does not change the set of circuits $\mathcal{C}$. However, the elementary vectors *do* change and so κ can also change. Viewing κ as a measure of how ill-conditioned the matrix A is, we ask for the best value of κ obtainable by choosing a suitable column scaling. This optimal value is denoted κ^*.

The interest in circuits and their imbalances has been steadily growing. Since circuits are a generalization of the edge directions of a polytope, bounds on the lengths of circuit walks have been of much interest [3, 4]. The computational complexity of finding good circuit steps has also been considered [18, 2, 5]. In the wake of the paper of Dadush et al. [9], circuit imbalances have been used to devise a fast first-order LP algorithm [8] or prove stronger bounds on circuit walks [10]. For more, we refer the reader to the excellent survey of Ekbatani et al. [11].

1.2 Our Contribution

Our **first** contribution is an implementation of the approximation rescaling algorithm of Dadush et al. [9]. We have discovered several non-obvious challenges, such as how exactly sparsity or floating-point arithmetic comes into play. We discuss the possible approaches and their trade-offs, paving the way for any possible further engineering effort. **Secondly**, we used the implementation to obtain an approximate rescaling of many problems in the benchmark sets Netlib [13] and MIPLIB [14], considering continuous relaxations in the latter case. Because of the high (even if polynomial) complexity of the algorithm and size of the mentioned problems there remain problem instances for which the computation of the rescaling using our implementation is infeasible in realistic timeframes. **Third**, we used the LP solvers GLPK[1], Gurobi [15], and SCIP [1] to evaluate the impact of the computed rescalings on the efficiency of the solvers. We have used all available solving methods and using both exact and floating-point arithmetic where available. Moreover, we discuss how successful the algorithm is in reducing the circuit imbalance measure κ.

Our main finding can be seen in Figure 1. On the one hand, the rescaling did not consistently speed-up any one solving method. On the other hand, for a great majority (79%) of instances, the rescaling sped-up *at least one* of the solving methods, showing possible promise for practical applicability. Moreover, the greatest achieved speed-up was by a factor of 13×. It also needs to be said that the compared solving times are sometimes very small (see the full data for details), possibly decreasing our confidence in meaningfulness of the measured time change.

[1] Open source software available at `http://www.gnu.org/software/glpk/glpk.html`

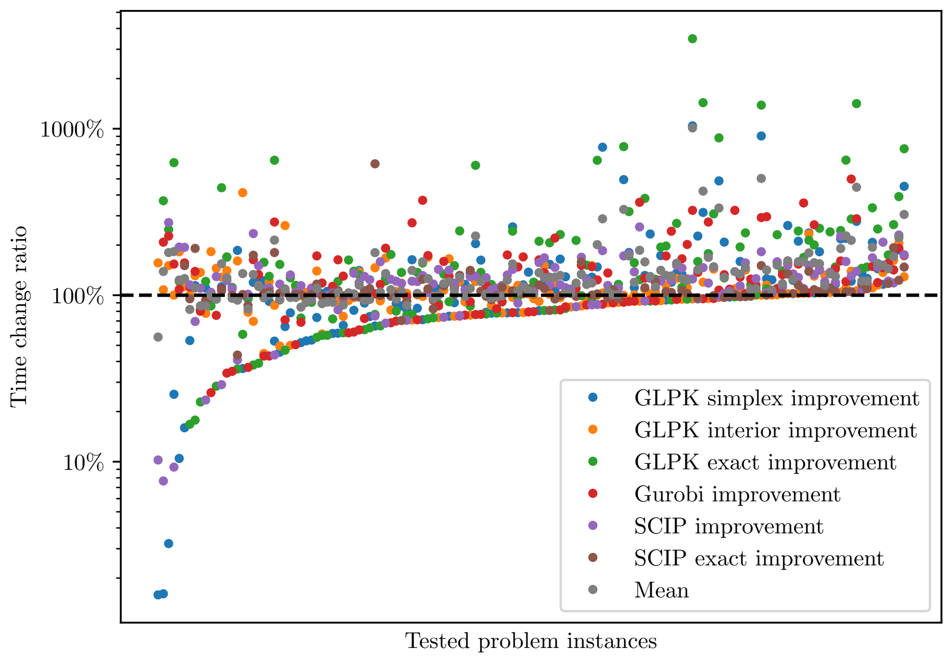

Figure 1 The results of the experiment. The rescaling improved the performance of at least one method for 112 out of the total 142 (79%) tested instances, and the best achieved speed-up was by a factor of 13×. However, the rescaling did not consistently help any particular method.

Because no single method consistently benefited from the rescaling, this may be regarded as a negative result. However, we still believe that positive results even for a minority of instances can point to potential applicability of this scaling approach. Conceivably, future research could identify features of instances for which this type of scaling helps, and if these instances could be recognized quickly, solving time improvements could be reaped. A major obstacle in this direction is the time necessary to compute the rescaling itself, which for all considered instances was vastly longer than the time it takes to solve the instance in the first place. However, our goal here was purely exploratory.

Paper Organization

This paper is based on the thesis [17], which contains many more details and helpful illustrations and examples.

2 Preliminaries

2.0.1 Notation

$\mathbb{R}$ denotes the set of real numbers, and $\mathbb{N}$ the set of natural numbers, including zero, and $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Matrices are written as $A \in \mathbb{R}^{m \times n}$ and vectors as $\mathbf{x} \in \mathbb{R}^n$.

All vectors are regarded as column vectors, the notation $\mathbf{x}^\top$ is used to denote row vectors when needed. Vector elements are provided with indices and not written in bold font, as in $\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$. For $B \subseteq [n]$, let A_B denote the submatrix of A obtained by selecting exactly columns with indices in B and similarly let $\mathbf{v}_B$ denote the subvector of $\mathbf{v}$ obtained by selecting exactly the entries of $\mathbf{v}$ with indices in B. Whenever used with the index in bold font, $\mathbf{x_i}$ does *not* denote the i-th coordinate, but it is a vector itself. The notation $\mathrm{Diag}(\mathbf{d})$ means a diagonal matrix with elements of $\mathbf{d}$ on the diagonal. When using the $\leq$ and similar relations ($\geq$, $<$, $>$, $=$) on vectors, we mean that the relation holds coordinate-wise, i.e. $\mathbf{x} \leq \mathbf{y}$ means that $x_i \leq y_i$ for every coordinate i. Sets are denoted with capital letters. We use the union symbol strictly for the union of sets, e.g., $A \cup B$. For the set difference of A and B, we write $A \setminus B$. If $\mathbb{F}$ is arbitrary field, then the coordinate space whose elements are n-tuples of elements of $\mathbb{F}$ is denoted $\mathbb{F}^n$.

2.0.2 Linear Programming

For $A \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$, linear programming (LP) in *standard form* is the problem $\min\{\mathbf{c}^\top\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\, \mathbf{x} \geq \mathbf{0},\, \mathbf{x} \in \mathbb{R}^n\}$. The standard form is convenient in theoretical contexts, however, in real-world applications, the following *modelling form* is common. Let $A_{\text{ub}} \in \mathbb{R}^{m_1\times n}$, $A_{\text{eq}} \in \mathbb{R}^{m_2\times n}$, $\mathbf{b_{ub}} \in \mathbb{R}^{m_1}$, $\mathbf{b_{eq}} \in \mathbb{R}^{m_2}$, $\mathbf{c} \in \mathbb{R}^n$ and $\mathbf{l}, \mathbf{u} \in (\mathbb{R} \cup \{-\infty, \infty\})^n$ such that $\mathbf{l} \leq \mathbf{u}$. Then, a linear program in *modelling form* is

$$\min\left\{\mathbf{c}^\top\mathbf{x} \mid A_{\text{ub}}\mathbf{x} \leq \mathbf{b_{ub}},\, A_{\text{eq}}\mathbf{x} = \mathbf{b_{eq}},\, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{R}^n\right\}$$

It is not difficult to see that every LP in standard form is already in modelling form. What is potentially less clear is that also every LP in modelling form can be efficiently converted to an equivalent LP in standard form (using standard techniques, see, e.g., Gärtner and Matoušek [12, Section 4.1]), rendering the two forms easily interchangeable.

2.0.3 Solving Scaled Instances

The motivation for scaling comes from the ability to relate the optima of the original and the scaled instance, as shown here:

▶ **Proposition 1.** *Let $A \in \mathbb{R}^{m\times n}, \mathbf{b} \in \mathbb{R}^m, \mathbf{c} \in \mathbb{R}^n$ and let $D \in \mathbb{R}^{n\times n}$ be a diagonal matrix with positive entries on the diagonal. Let P be the standard-form LP $\min\left\{\mathbf{c}^\top\mathbf{x} \mid A\mathbf{x} = \mathbf{b},\, \mathbf{x} \geq \mathbf{0}\right\}$ and Q the standard-form LP $\min\left\{\mathbf{c}^\top D\mathbf{x} \mid (AD)\mathbf{x} = \mathbf{b},\, \mathbf{x} \geq \mathbf{0}\right\}$. Then $\mathbf{x}'$ is an optimum of P if and only if $D^{-1}\mathbf{x}'$ is an optimum of Q.*

Proof. Let $\mathbf{x}'$ be an optimum of P. Because D has positive diagonal entries, D^{-1} also has positive diagonal entries, so $D^{-1}\mathbf{x}' \geq \mathbf{0}$ holds. The equality constraint is also satisfied, which follows from $(AD)(D^{-1}\mathbf{x}') = A(DD^{-1})\mathbf{x}' = A\mathbf{x}' = \mathbf{b}$. This proves that $D^{-1}\mathbf{x}'$ is feasible for Q. What is left to show is the optimality of $D^{-1}\mathbf{x}'$ for Q. For contradiction consider that Q has a better solution $\mathbf{x}''$ such that $\mathbf{c}^\top D\mathbf{x}'' < \mathbf{c}^\top D(D^{-1}\mathbf{x}')$. Then, $D\mathbf{x}''$ is feasible for P and it holds that $\mathbf{c}^\top(D\mathbf{x}'') < \mathbf{c}^\top\mathbf{x}'$, contradicting that $\mathbf{x}'$ is an optimum of P. The proof of the opposite direction is symmetric. ◀

2.1 Circuit Imbalance Measures

We refer to the thesis [17, Section 2.3] or the book of Oxley [20] for the basics of matroid theory. In Section 1.1, we have already defined the notions of a circuit, its elementary vector, and the circuit imbalance measure κ. For a circuit C, let $\mathbf{g}^C$ denote its elementary vector, and for a matroid M, let $\mathcal{C}(M)$ denote the set of its circuits. The *pairwise circuit imbalance measure* for coordinate pair $i, j \in [n], i \neq j$, is the value

$$\kappa_{ij}^A(C) := \frac{|g_j^C|}{|g_i^C|} \text{ or } 0 \text{ if } g_i^C = 0, \quad \kappa_{ij}^A := \max\{\kappa_{ij}^A(C) \mid C \in \mathcal{C}(A), i, j \in C\}.$$

By convention we set $\kappa_{ij} = 0$ if there is no circuit supporting i and j.

Define the *circuit imbalance measure* to be $\kappa_A := \max\{\kappa_{ij}^A \mid i, j \in [n]\}$, and let $\kappa_A(C) := \max\{\kappa_{ij}^A(C) \mid i, j \in C\}$, which satisfies $\kappa_A = \max\{\kappa_A(C) \mid C \in \mathcal{C}(M(A))\}$.

Let $\mathbf{D}$ be the set of $n \times n$ real diagonal matrices with positive entries on the diagonal. The *optimal circuit imbalance measure* for A is the number $\kappa_A^* := \inf\{\kappa_{AD} \mid D \in \mathbf{D}\}$. If clear from context, we will omit the A subscript or superscript from κ and its variants. It is not a priori clear that the infimum is indeed attained, but we will later see that it in fact is.

2.2 Algorithm for Optimizing the Circuit Imbalance Measure

Here we give an overview of the scaling algorithm of Dadush et al. [9] that is implemented and used in our experiments. The purpose of the algorithm is to find a diagonal matrix D such that κ_{AD} is not much larger that κ_A^*.

First, it is not difficult to see that we can assume the vector matroid of A is connected, otherwise we can process its components separately.

Next, it follows from Tuncel [22] that approximating κ up to a factor of $2^{\mathcal{O}(m)}$ is NP-hard. The algorithm provided in the following text finds a $\hat{\kappa}_A$ satisfying

$$\hat{\kappa}_A \leq \kappa_A \leq n(\kappa_A^*)^2 \hat{\kappa}_A$$

in polynomial time. Note that this is not a contradiction to the NP-hardness claim, because the approximation ratio depends on κ_A^*.

The basic idea of the algorithm is as follows. One first shows that if we had the values κ_{ij}^A, we could efficiently compute the value of κ. The values κ_{ij} are of course still NP-hard to compute, but it is possible to efficiently find certain proxies and use these instead to obtain a good enough approximation $\hat{\kappa}$. The following theorem states what kind of proxies we are looking for, and what approximation do they yield; the theorem is stated in [9] differently due to the different focus of their work compared to ours, but its substance is the same:

▶ **Theorem 2** ([9, Theorem 2.5]). *For every $i, j \in [n]$ let $\hat{\kappa}_{ij}$ satisfy*

$$\hat{\kappa}_{ij} \leq \kappa_{ij} \leq (\kappa^*)^2 \hat{\kappa}_{ij}. \tag{1}$$

Define $\hat{\kappa} := \max\{\hat{\kappa}_{ij} \mid i, j \in [n]\}$. Then it holds that $\hat{\kappa} \leq \kappa \leq n(\kappa^*)^2 \hat{\kappa}$.

From now on, we will use $\hat{\kappa}_{ij}$ for any value satisfying (1). While possibly many values satisfy this, the same value is always meant when $\hat{\kappa}_{ij}$ occurs multiple times. Now we are left with the task of finding the pairwise approximations $\hat{\kappa}_{ij}$. The solution is actually simple: find any circuit that contains both i and j and estimate κ_{ij} based on this circuit:

▶ **Proposition 3** ([9, Corollary 2.13]). *Let $i, j \in [n]$ and let C be any circuit such that $i, j \in C$. Then $\kappa_{ij}(C) \leq \kappa_{ij} \leq (\kappa^*)^2 \kappa_{ij}(C)$.*

The task of finding such circuit will be the central topic of Section 2.2.1. Computing the diagonal matrix which witnesses an approximation $\hat{\kappa}^*$ of κ^* is more involved, but can still be done efficiently [9, Theorem 2.5]; let us reformulate their result in a more algorithmic way:

▶ **Definition 4** (Circuit ratio digraph). *The* circuit ratio digraph *is a directed graph* $\mathcal{C}G_{\bullet} := ([n], E)$*, where* $E := \{(i,j) \in [n]^2 \mid i \neq j, \kappa_{ij} > 0\}$.

▶ Remark 5. From Proposition 3 it follows that $\kappa_{ij} > 0$ if and only if $\hat{\kappa}_{ij} > 0$. The edge set above can therefore be equivalently defined as $E := \{(i,j) \in [n]^2 \mid i \neq j, \hat{\kappa}_{ij} > 0\}$.

▶ **Theorem 6** ([9, Theorem 2.5]). *Consider the optimization program*

$$\begin{aligned} &\min t \\ &\hat{\kappa}_{ij} d_j / d_i \leq t \quad \forall (i,j) \in E(\mathcal{C}G_{\bullet}) \\ &\quad \mathbf{d} > 0. \end{aligned} \tag{2}$$

Let $\hat{\mathbf{d}}$ *be the optimal value of* $\mathbf{d}$ *for* (2) *and* $D := \operatorname{Diag}(\hat{\mathbf{d}})$*. Then* $\kappa_{AD} \leq (\kappa_A^*)^3$.

▶ Remark 7. Note that (2) can be made into an LP by taking the logarithms of all variables.

In the following, let $\hat{\mathbf{d}}$ always denote the optimal value of $\mathbf{d}$ for (2). This optimization problem can be solved in polynomial time using LP techniques. There also exists an equivalent combinatorial description for the rescaling, which will yield additional insights.

▶ **Theorem 8** ([9, Theorem 2.5]). *Consider a weighted directed graph* G *defined by* $V(G) := [n]$, $E(G) := \{(i,j) \in [n]^2 \mid i \neq j, \hat{\kappa}_{ij} > 0\}$*, and* $w_G(e) := \hat{\kappa}_{ij}$ *for all* $e \in E(G)$*. Define* $\hat{t}$ *as the maximum geometric mean of a cycle in* G*, that is*

$$\hat{t} := \max \left\{ \left(\sum_{e \in C} w_G(e) \right)^{\frac{1}{|C|}} \middle| C \text{ is a cycle in } G \right\}.$$

Let H *be a weighted directed graph defined by* $V(H) := V(G) \cup \{r\}$*,*
$E(H) := E(G) \cup \{(r,i) \mid i \in [n]\}$*, and* $w_H(e) := \begin{cases} \log \hat{t} - \log \hat{\kappa}_{ij} & \text{for } e \in E(H) \text{ s.t. } r \notin e, \\ 0 & \text{for } e \in E(H) \text{ s.t. } r \in e. \end{cases}$

Let s_i *denote the length of the shortest path from* r *to* i *in* H*, and define* $d_i := \exp(s_i)$ *for all* $i \in [n]$*. Then, it holds that* $\mathbf{d} = \hat{\mathbf{d}}$.

▶ Remark 9. Note that while H has negative edge weights, it has no negative cycles, because $\hat{t}$ is defined as the maximum geometric mean of weights over all cycles in G. The graph H has the same cycles as G, so for every cycle C of H it must hold that $|C| \log \hat{t} \geq \sum_{(i,j) \in C} \log \hat{\kappa}_{ij}$. Therefore, the Bellman-Ford shortest path algorithm can be used to determine the length of the shortest path from r to i in H, and, in particular, the values s_i are well-defined.

2.2.1 Finding Circuits

What remains is to describe how to find circuits C_{ij} that include both i and j. Since we only consider connected matroids, such circuits always exist. We will now describe the algorithm from [9, Theorem 2.14] to find these circuits.

Suppose we have a basis B of $M(A)$. We start by computing the set of all fundamental circuits $\mathcal{F}_B$ with respect to B. If there is a circuit $C \in \mathcal{F}_B$ with $i, j \in \mathcal{F}_B$, we set $C_{ij} := C$ and we are done with this pair i, j.

Otherwise, we construct an undirected graph G with $V(G) = B$ and edges between elements that are contained in a common fundamental circuit. Because $M(A)$ is connected, so is G. First assume $i, j \notin B$, and recall that, for an element $e \notin B$, $C(B, e)$ denotes the fundamental circuit of e with respect to B. We find a shortest path in G between any vertex in the set $C(B,i) \setminus \{i\}$ and any vertex in the set $C(B,j) \setminus \{j\}$. This can be represented by the sequences $V := \{v_1, v_2, \ldots, v_{\ell+1}\} \subseteq [n] \setminus B$, $v_1 = i, v_{\ell+1} = j$, and $U = \{u_1, u_2, \ldots, u_\ell\} \subseteq B$. The set V corresponds to vertices of G and the set U to edges of G. More precisely, we set $v_i := b$ for some $b \in B$ such that the i-th edge on the path connects vertices in $C(B,b)$.

It is shown in [9] that the set $S := (B \setminus U) \cup V$ contains a unique circuit C that contains all v_t's, including i and j.

If $\{i,j\} \cap B \neq \emptyset$, assume w.l.o.g. that $i \in B$. Then, for any choice of $V = \{v_1, v_2, \ldots, v_{\ell+1}\}$ and $U = \{u_1, u_2, \ldots, u_\ell\}$ as before such that $i \in C(B, v_1)$ and $i \notin C(B, v_t)$ for $t > 1$, the unique circuit in $(B \setminus U) \cup V$ also contains i. If both $i, j \in B$, we proceed in a similar manner with j. For more details see the original algorithm description [9].

2.2.2 Finding a unique circuit in a set S

It remains to show how to find the unique circuit in S together with its elementary vector. There are standard tools of linear algebra that can be utilized for this task, but [9] also provides an alternative solution. We provide both of these solutions here because comparing their efficiency and suitability in practical settings will be an important topic of Section 3.2.

The first method is to solve the linear system $A_S\mathbf{x} = \mathbf{0}$ using any generic linear equation solver, and pronounce the result to be the unique circuit contained in S. This approach is justified by the next Lemma:

▶ **Lemma 10.** *Let $A \in \mathbb{R}^{n \times m}$ and let $S \subseteq [n]$ such that $M(A_S)$ contains a unique circuit C. Then,* $\mathrm{Ker}(A_S)$ *has dimension 1. Let the basic vector of* $\mathrm{Ker}(A_S)$ *(unique up to scaling) be* $\mathbf{v}$*. Then,* $\mathbf{v}$ *is an elementary vector for C and $C = \mathrm{supp}(\mathbf{v})$.*

Proof. We will only prove that $\dim(\mathrm{Ker}(A_S)) = 1$, the rest of the lemma statement follows in a straightforward way. Let C be a circuit of $M(A_S)$ and let $\mathbf{v_1}$ be its elementary vector. For contradiction, suppose that $\dim(\mathrm{Ker}(A_S)) \geq 2$. That implies that there exists another vector $\mathbf{v_2} \in \mathrm{Ker}(A_S)$ that is linearly independent from $\mathbf{v_1}$.

Let i be any element such that $i \in C$. If $(v_2)_i = 0$, define $\mathbf{v_3} := \mathbf{v_2}$. Otherwise, let $\mathbf{v_3} := \mathbf{v_2} - \frac{(v_1)_i}{(v_2)_i}\mathbf{v_1}$. In both cases, $i \notin \mathrm{supp}(\mathbf{v_3})$. Because $\mathbf{v_3} \in \mathrm{Ker}(A_S)$, $\mathrm{supp}(\mathbf{v_3})$ is a dependent set in $M(A_S)$. Let C' be any inclusion-wise minimal dependent subset of $\mathrm{supp}(\mathbf{v_3})$.

Because $i \in C$ and $i \notin C'$, it must hold that $C \neq C'$, which is a contradiction with the claim that $M(A_S)$ has exactly one circuit. ◀

The second approach is advantageous if the task is solved repeatedly for the same matrix but different subsets S, which is exactly our situation. It requires converting the matrix A into the row-reduced echelon form (RREF) $A' = (I_m|H)$, but this step is independent of the choice of S and only has to be done once in the course of the algorithm:

▶ **Proposition 11** ([9, Inside proof of Theorem 2.14])**.** *Define an elementary vector* $\mathbf{g}$ *for a circuit C using the following procedure: Set $g_{v_1} := 1$. Using Gauss-Jordan elimination, convert A to the form $A' = (I_m|H)$. Now for each $t \in [\ell]$, the row of A'_S corresponding to u_t contains only two nonzero entries: $A'_{u_t v_t}$ and $A'_{u_t v_{t+1}}$. Propagate the value g_{v_1} by assigning $g_{v_2}, g_{v_3}, \ldots, g_{v_{\ell+1}}$. Uniquely extend* $\mathbf{g}$ *to the indices in $B \cap S$ for any basis B so that* $\mathbf{g} \in \mathrm{Ker}(A)$*. Then,* $\mathbf{g}$ *is an elementary vector for C and it holds that $C = \mathrm{supp}(\mathbf{g})$.*

2.2.3 LP Dataset

To evaluate algorithms related to LP, we will use two publicly available collections of instances. The first and older one is Netlib [13]. In order to widen the set of instances considered here, we also consider the larger and more modern repository MIPLIB [14], which contains mixed integer programs (MIP). Because this work only directly applies to LPs and not MIPs, we consider the *LP relaxations* of MIPLIB instances.

2.2.4 Problem Format

Both Netlib and MIPLIB problems are in MPS format, which is the industry standard for linear programming problems. Without going into unnecessary detail, let us mention that MPS is very similar to a sparsely represented modelling LP form. Reading the problem file into a convenient representation will be discussed later. Most of the considered problems are very sparse and therefore using a sparse representation is crucial.

3 Algorithm Implementation

We use SageMath [21] for the implementation of the algorithm. SageMath uses Python and can be used directly from Python code, which gives the advantage of a variety of available scientific packages. SageMath itself has support for working with both densely and sparsely represented matrices over both exact and inexact fields, working with graphs, and some support for working with matroids. These capabilities make SageMath a good fit for our goal, which is exploratory rather than aiming for practical efficiency.

Despite Python being relatively slow, this is likely not a major bottleneck since a large portion of the run-time of our implementation is spent inside library functions that have heavily optimized low-level implementations. Thus, the potential performance gain of re-implementing the same algorithm in a lower-level language is unlikely to be large.

3.1 Linear Programming Considerations and Challenges

When reading an LP instance represented in the MPS format, we deal with two considerations. First, how to even read MPS and handle it efficiently? Second, the MPS format is akin to the modelling LP form representation, but our algorithm assumes the standard form of LP. A direct solution would be to perform the polynomial conversion from modelling to the standard form; we point out that we can do a bit better by stopping the conversion earlier.

3.1.1 Reading The Problem Matrix from MPS

Several Python libraries allow reading LPs from the MPS format. The problem with most of these libraries is that it is not easy to access the problem matrix itself. In order to avoid writing an MPS parser from scratch, we utilize the open-source `scikit-glpk` [2] Python package. This package provides Python bindings for GLPK and allows to read the problem into modelling form as sparse SciPy matrices and vectors, which can be easily converted to sparse matrices and vectors native to SageMath.

[2] Available from `https://github.com/mckib2/scikit-glpk`

Prior to our work, this package was not able to read all MPS files that are used in Netlib and MIPLIB. To allow efficient reading all of the problems, we needed to make some changes to the library, which we contributed back to the original package[3].

The first modification of the library was fixing a bug caused by earlier modifications of the library that rendered the MPS reading and writing capabilities defunct. The second modification addressed the fact that the MPS reader inside GLPK sometimes internally represents constraints bounded from both sides, i.e. $b_1 \leq \mathbf{a}^\top \mathbf{x} \leq b_2$. As this does not directly correspond to a constraint in modelling form, the library did not support reading the problems whose internal GLPK representation contained such a constraint. We have modified the library to split any such constraint into the two constraints $b_1 \leq \mathbf{a}^\top \mathbf{x}$ and $b_2 \geq \mathbf{a}^\top \mathbf{x}$. Lastly, we have significantly improved the library performance for working with sparse matrices. The original implementation used an intermediate dense representation when combining two matrices into one and only then converted the resulting matrix back into a sparsely represented one. By using a sparse implementation for the combining step, we avoid the need for the costly dense conversion.

3.1.2 Solving LP Form Mismatch

The standard procedure to convert an LP in modelling form to an LP in standard form turns inequality constraints into equality constraints and turns generic bounds into non-negativity bounds. Here we show that converting the bounds to the form $\mathbf{x} \geq \mathbf{0}$ is not necessary and it suffices to arrive at the more general form of *box constraints* $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ for vectors $\mathbf{l}, \mathbf{u} \in (\mathbb{R} \cup \{-\infty, +\infty\})^n$ satisfying $\mathbf{l} \leq \mathbf{u}$, since this step does not change the value of κ nor the value of κ^*. We call an LP $\{\min \mathbf{c}^\top \mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$ to be in *semi-standard form*. Our main result here is the following:

▶ **Theorem 12.** *Let A be a problem matrix of an LP in semi-standard form and A' a problem matrix for the same LP converted to standard form. Then, $\kappa_A = \kappa_{A'}$ and $\kappa^*_A = \kappa^*_{A'}$.*

We will break down the proof of Theorem 12 into several simpler statements.

The textbook algorithm for converting an LP in semi-standard form to standard form only uses the following operations:

(O1) Take a variable x_i and express it as $x_i = x_i^+ - x_i^-$ for new variables $x_i^+, x_i^- \geq 0$. Let, without loss of generality,

$$A = \left(\begin{array}{c|c} & \vdots \\ \bar{A} & A_{*i} \\ & \vdots \end{array} \right), \qquad \mathbf{x} = \left(\begin{array}{c} \vdots \\ \hat{\mathbf{x}} \\ \vdots \\ \hline x_i \end{array} \right).$$

The problem matrix after this operation is then

$$A' = \left(\begin{array}{cc|c} & \vdots & \vdots \\ \bar{A} & A_{*i} & -A_{*i} \\ & \vdots & \vdots \end{array} \right).$$

[3] https://github.com/mckib2/scikit-glpk/pull/24
https://github.com/mckib2/scikit-glpk/pull/25
https://github.com/mckib2/scikit-glpk/pull/23

(O2) Take a bound $\pm x \geq l$ and turn it into a constraint. If we assume the same layout of $\mathbf{x}$ as in (O1), the problem matrix after this operation is

$$A' = \left(\begin{array}{ccc|c} & & \vdots & 0 \\ & \bar{A} & A_{*i} & \vdots \\ & & \vdots & 0 \\ \hline 0 \cdots\cdots 0 & & \pm 1 & 1 \end{array}\right).$$

We will show that performing neither operation changes the values κ, κ^*. By induction it then follows that performing the conversion algorithm also does not change these values.

▶ **Lemma 13.** $\kappa_A \geq 1$.

Proof. Let us choose any $C \in M(A)$. For all $i, j \in \binom{C}{2}$, both values $\kappa_{ij}^A(C)$ and $\kappa_{ji}^A(C)$ are considered when determining the value of $\kappa_A(C)$. Because $\kappa_{ij}^A = {}^1/_{\kappa_{ji}^A}$, it must be the case that $\max\{\kappa_{ij}^A(C), \kappa_{ji}^A(C)\} \geq 1$. Therefore, the maximum over all pairs of i, j is also greater than 1, meaning that $\kappa_A(C) \geq 1$ for any circuit C. By taking maximum over all circuits, we get the lemma statement. ◀

▶ **Lemma 14.** *Performing operation (O1) does not change the value of κ.*

Proof. Let A, A' be problem matrices before and after performing the operation respectively. Firstly we will show the inequality $\kappa_A \leq \kappa_{A'}$. Let $C \in \mathcal{C}(M(A))$ be the circuit that $\kappa_A = \kappa_A(C)$ and let $\mathbf{g}^C = (\bar{\mathbf{g}}^{\mathbf{C}}, g_i)^\top$ be its elementary vector. Define the vector $\mathbf{g}^{\mathbf{C}'} := (\bar{\mathbf{g}}^{\mathbf{C}}, g_i, 0)^\top$. For this vector it holds that

$$\left(\begin{array}{ccc|c} & & \vdots & \vdots \\ & \bar{A} & A_{*i} & -A_{*i} \\ & & \vdots & \vdots \end{array}\right) \left(\begin{array}{c} \vdots \\ \bar{\mathbf{g}}^{\mathbf{C}} \\ \vdots \\ g_i \\ \hline 0 \end{array}\right) = A\mathbf{g}^C = \mathbf{0}.$$

We will show that $C' = \operatorname{supp}(\mathbf{g}^{\mathbf{C}'})$ is a circuit of $M(A')$. From the above it directly follows that C' is a dependent set. For contradiction assume that there exists a dependent set of $M(A')$ denoted by $C'' \subset C'$. Let $(\bar{\mathbf{g}}^{\mathbf{C}''}, 0)^\top$ be an elementary vector for this set. Then, $\operatorname{supp}(\bar{\mathbf{g}}^{\mathbf{C}''}) \subset C$ is an independent set of $M(A)$, which contradicts that C is a circuit.

Because $\mathbf{g}^{\mathbf{C}'}$ has the same non-zeroes as $\mathbf{g}^C$, we have $\kappa_A(C) = \kappa_{A'}(C')$. Thus,

$$\kappa_A = \kappa_A(C) = \kappa_{A'}(C') \leq \kappa_{A'}.$$

For the other inequality, consider any $C' \in \mathcal{C}(M(A'))$ with its elementary vector $\mathbf{g}^{\mathbf{C}'}$. Because the last two columns of A' are linearly dependent, only two scenarios occur for $\mathbf{g}^{\mathbf{C}'}$:

1. Either $\mathbf{g}^{\mathbf{C}'} = (\bar{\mathbf{g}}^{\mathbf{C}'}, g_i, 0)^\top$ or $\mathbf{g}^{\mathbf{C}'} = (\bar{\mathbf{g}}^{\mathbf{C}'}, 0, -g_i)^\top$. In that case, $(\bar{\mathbf{g}}^{\mathbf{C}'}, g_i)^\top$ is an elementary vector for some vector $C \in \mathcal{C}(M(A))$ and it holds that $\kappa_{A'}(C') = \kappa_A(C)$.
2. Otherwise, $\mathbf{g}^{\mathbf{C}'} = (\mathbf{0}, g_i, g_i)^\top$. Then clearly $\kappa_{A'}(C') = 1$.

For any circuit C' it therefore holds that $\kappa_{A'}(C') \leq \max\{\kappa_A, 1\}$. By using the previously stated Lemma 13, we get the inequality $\kappa_{A'}(C') \leq \kappa_A$.

Consider C'' such that $\kappa_{A'} = \kappa_{A'}(C'')$. Then it is holds that

$$\kappa_{A'} = \kappa_{A'}(C'') \leq \kappa_A.$$ ◀

▶ **Lemma 15.** *For any $D \in \mathbf{D}$, $\mathcal{C}(M(A)) = \mathcal{C}(M(AD))$.*

Proof. Multiplying vectors by nonzero constants in a set of vectors does not change the linear dependence of the set. Therefore, rescaling columns with matrix D does not change dependent sets nor inclusion-wise minimal dependent sets. ◀

▶ **Lemma 16.** *Performing operation (O1) does not change the value of κ^*.*

Proof. Let A, A' be matrices as in the previous proof. We will start by showing that $\kappa^*_{A'} \leq \kappa^*_A$. Consider $D \in \mathbf{D}$ such that $\kappa^*_A = \kappa_{AD}$. Define

$$D' := \left(\begin{array}{ccc|c} & & 0 & 0 \\ & \bar{D} & \vdots & \vdots \\ & & 0 & \vdots \\ & & d_i & 0 \\ \hline 0 & \cdots\cdots & 0 & d_i \end{array}\right).$$

Consider the circuit $C' \in \mathcal{C}(M(A'D'))$ such that $\kappa_{A'D'} = \kappa_{A'D'}(C')$ with an elementary vector $\mathbf{g}^{\mathbf{C}'}$.

Using Lemma 15, we again know that this circuit can be of one of two types:

1. If $\mathbf{g}^{\mathbf{C}'} = (\bar{\mathbf{g}}^{\mathbf{C}'}, g_i, 0)^\top$ or $\mathbf{g}^{\mathbf{C}'} = (\bar{\mathbf{g}}^{\mathbf{C}'}, 0, -g_i)^\top$, then the vector $\mathbf{g}^C := (\bar{\mathbf{g}}^{\mathbf{C}'}, g_i)^\top$ is an elementary vector for a circuit $C \in \mathcal{C}(M(AD))$. Because this vector shares the same nonzero values with $\mathbf{g}^{\mathbf{C}'}$, it holds that
$$\kappa^*_{A'} \leq \kappa_{A'D'} = \kappa_{A'D'}(C') = \kappa_{AD}(C) \leq \kappa_{AD} = \kappa^*_A.$$
2. Otherwise, $\mathbf{g}^{\mathbf{C}'} = (\mathbf{0}, g_i, g_i)^\top$. In that case, $\kappa_{A'D'}(C') = 1$. By Lemma 13,
$$\kappa^*_{A'} \leq \kappa_{A'D'} = \kappa_{A'D'}(C') = 1 \leq \kappa_{AD} = \kappa^*_A.$$

To prove the other inequality, consider the $(n+1) \times (n+1)$ diagonal matrix with positive entries on the diagonal

$$D' = \left(\begin{array}{c|c} & 0 \\ \bar{D}' & \vdots \\ & 0 \\ \hline 0 \cdots\cdots 0 & d_{n+1} \end{array}\right)$$

such that $\kappa^*_{A'} = \kappa_{A'D'}$.

Define $D := \bar{D}'$ and consider $C \in \mathcal{C}(M(AD))$ such that $\kappa_{AD} = \kappa_{AD}(C)$ together with the elementary vector $\mathbf{g}^C$. Then, the vector $\mathbf{g}^{\mathbf{C}'} := (\mathbf{g}^C, 0)^\top$ is an elementary vector of a circuit in $M(A'D')$ with $\kappa_{A'D'}(C') = \kappa_{AD}(C)$. Therefore it must hold that

$$\kappa^*_A \leq \kappa_{AD} = \kappa_{AD}(C) = \kappa_{A'D'}(C') \leq \kappa_{A'D'} = \kappa^*_{A'}.$$ ◀

▶ **Lemma 17.** *Performing operation (O2) does not change the value of κ.*

Proof. Let A, A' be matrices as before, $C' \in \mathcal{C}(M(A'))$ such that $\kappa_{A'} = \kappa_{A'}(C')$ and its elementary vector $\mathbf{g}^{\mathbf{C}'}$. For the elementary vector it must hold that

$$\left(\begin{array}{cc|c} & & 0 \\ & A & \vdots \\ & & 0 \\ \hline 0 \cdots\cdots 0 & \pm 1 & 1 \end{array}\right) \left(\begin{array}{c} \vdots \\ \bar{\mathbf{g}}^{\mathbf{C}'} \\ \vdots \\ \hline g_{n+1} \end{array}\right) = \left(\begin{array}{c} \vdots \\ \mathbf{0} \\ \vdots \\ \hline 0 \end{array}\right).$$

This can be split into two conditions

$$\bar{\mathbf{g}}^{\mathbf{C}'} \in \mathrm{Ker}(A)$$
$$g_{n+1} = \pm g_n.$$

We will show that $\mathbf{g}^C := \bar{\mathbf{g}}^{\mathbf{C}'}$ is an elementary vector of a circuit $C := \mathrm{supp}(\bar{\mathbf{g}}^{\mathbf{C}'}) \in \mathcal{C}(M(A))$. We already know that C is dependent. For contradiction let us now suppose that there exists another dependent set $\hat{C} \subset C$. This circuit can be extended to a circuit $\hat{C}' \in \mathcal{C}(A')$ such that $\hat{C}' \subset C'$, contradicting that C' is a circuit. The set C is therefore maximal dependent.

Vectors $\mathbf{g}^{\mathbf{C}'}$ and $\mathbf{g}^C$ have the same nonzero absolute values, which yields $\kappa_A(C) = \kappa_{A'}(C')$. Consequently,

$$\kappa_{A'} = \kappa_{A'}(C') = \kappa_A(C) \le \kappa_A.$$

For showing the opposite inequality, it suffices to observe that every elementary vector $\mathbf{g}^C$ for a circuit $C \in \mathcal{C}(A)$ can be uniquely extended to an elementary vector $\mathbf{g}^{\mathbf{C}'}$ for a circuit $C' \in \mathcal{C}(A')$ such that $\kappa_A(C) = \kappa_{A'}(C')$. Specifically for C such that $\kappa_A = \kappa_A(C)$, it then follows that

$$\kappa_A = \kappa_A(C) = \kappa_{A'}(C') \le \kappa_{A'}. \qquad \blacktriangleleft$$

▶ **Lemma 18.** *Performing operation (O2) does not change the value of κ^*.*

Proof. We again denote the matrices A and A' before and after performing the operation. First let us show that $\kappa^*_{A'} \ge \kappa^*_A$. Let D' be a diagonal matrix with positive numbers on the diagonal such that $\kappa^*_{A'} = \kappa_{A'D'}$. The scaled matrix can be written as

$$A'D' = \left(\begin{array}{ccc|c} & A\bar{D}' & & \begin{matrix}0\\ \vdots \\ 0\end{matrix} \\ \hline 0 \cdots\cdots 0 & & \pm d_n & d_{n+1} \end{array}\right).$$

For any elementary vector $\mathbf{g}^{\mathbf{C}'} = (\bar{\mathbf{g}}^{\mathbf{C}'}, g_{n+1})^\top$ of a circuit in $\mathcal{C}(M(A'D'))$ it must hold that

$$\bar{\mathbf{g}}^{\mathbf{C}'} \in \mathrm{Ker}(A\bar{(}D'))$$
$$g_{n+1} = \pm \tfrac{d_n}{d_{n+1}} g_n.$$

Let $C \in \mathcal{C}(A\bar{D})$ be a circuit such that $\kappa_{A\bar{D}}(C) = \kappa_{A\bar{D}}$ with an elementary vector $\mathbf{g}^C$. Define $\mathbf{g}^{\mathbf{C}'} := (\mathbf{g}^C, \pm\frac{d_n}{d_{n+1}} g_n)^\top$ and observe that $C' := \mathrm{supp}(\mathbf{g}^{\mathbf{C}'}) \in \mathcal{C}(M(A'D'))$.

Because $\mathbf{g}^{\mathbf{C}'}$ contains all the non-zero elements that $\mathbf{g}^C$ contains, it must be the case that $\kappa_{A'D'}(C') \ge \kappa_{A\bar{D}}(C)$, leaving us with

$$\kappa^*_{A'} = \kappa_{A'D'} \ge \kappa_{A'D'}(C') \ge \kappa_{A\bar{D}}(C) = \kappa_{A\bar{D}} \ge \kappa^*_A.$$

For the other direction, consider $D \in \mathbf{D}$ such $\kappa^*_A = \kappa_{AD}$. Choose D' such that

$$D' := \left(\begin{array}{cc|c} \bar{D} & \begin{matrix}0\\ \vdots \\ 0\end{matrix} & \begin{matrix}0\\ \vdots \\ \vdots \\ 0\end{matrix} \\ & d_n & 0 \\ \hline 0 \cdots\cdots\cdots 0 & & d_n \end{array}\right).$$

Consider the circuit $C' \in \mathcal{C}(M(A'D'))$ such that $\kappa_{A'D'}(C') = \kappa_{A'D'}$ together with its elementary vector $\mathbf{g}^{\mathbf{C}'} = (\bar{\mathbf{g}}^{\mathbf{C}'}, g_{n+1})^\top$. Plugging into the conditions above, it must hold that

$$\bar{\mathbf{g}}^{\mathbf{C}'} \in \mathrm{Ker}(AD)$$
$$g_{n+1} = \pm g_n.$$

We can observe that $\bar{\mathbf{g}}^{\mathbf{C}'}$ is an elementary vector of a circuit $C \in \mathcal{C}(M(AD))$. The vectors $\mathbf{g}^{\mathbf{C}'}$ and $\bar{\mathbf{g}}^{\mathbf{C}'}$ have the same nonzero absolute values, therefore $\kappa_{A'D'}(C') = \kappa_{AD}(C)$. By putting everything together, we get

$$\kappa^*_A = \kappa_{AD} \geq \kappa_{AD}(C) = \kappa_{A'D'}(C') = \kappa_{A'D'} \geq \kappa^*_{A'}.$$ ◀

Combining Lemmas 14, 16, 17 and 18 concludes the proof of Theorem 12.

3.2 Practical Aspects of Dealing With the Vector Matroid

To perform computations with the vector matroid and to understand their efficiency, it is important to understand how the relevant operations on vector matroids can be implemented. These details have great implications for the practicality and scalability of the experiments.

3.2.1 Exactness of Computation

When performing computations with real numbers on modern computers, a floating point (FP) number representation is typically used. While this representation is convenient in many cases, practical calculations then suffer from an inherent precision loss. We will show that this causes issues when using FP computations for working with the vector matroid of a matrix. Not only can small numerical differences in values of the matrix completely change the structure of its vector matroid, but they may also arbitrarily change both the values κ and κ^*.

Example 19 shows that inaccuracies introduced by FP computations may lead an algorithm to falsely proceed as if some set S is independent when it in fact contains a circuit.

▶ **Example 19** (Floating-point arithmetics and vector matroid). For $a \in \mathbb{R}$ consider the matrix

$$A := \begin{pmatrix} a & a & 4a \\ 0 & a & a \end{pmatrix}.$$

Because $3 \cdot A_{*1} + A_{*2} = A_{*3}$, $\{1,2,3\}$ is not an independent set of $M(A)$. However, when using floating-point operations $\cdot_F$ and $+_F$, one may get the result

$$3 \cdot_F \begin{pmatrix} a \\ 0 \end{pmatrix} +_F \begin{pmatrix} a \\ a \end{pmatrix} = \begin{pmatrix} 4a + \varepsilon_1 \\ a + \varepsilon_2 \end{pmatrix}$$

for small values of ε_1 and ε_2. This might lead an algorithm relying on floating-point arithmetics to proceed as if all the three columns were linearly independent and to conclude that $\{1,2,3\}$ is an independent set of $M(A)$.

The obvious remedy is using exact arithmetic, but this is undesirable for performance reasons. We have tested an alternative approach based on trying to (quickly) find a candidate for a circuit using p-bit precision computation for increasing values of p, while verifying potential candidate circuits by exact computation. Unfortunately, this approach turned out to be impractical because all attempted choices for the precision parameters led to unreasonably high iteration counts. As a matter of fact, the algorithm converged in reasonable time only for very small problem instances. Since we have not found a way to work with the vector matroid using floating-point arithmetics, we use SageMath's support for exact rational numbers.

Not only is working with the vector matroid without using exact arithmetics tricky to say the least, but we also observe that small changes in A can inflict large changes in both κ and κ^*. This can be dangerous if we obtain a nearly optimal rescaling of a matrix and then try to use it in an inexact context (e.g., run an LP solver that uses FP arithmetics).

Example 20 shows that FP inaccuracies may make an algorithm operate as if $\kappa_A = C$ for an arbitrarily large C, despite $\kappa_A = 1$. Moreover, Example 21 shows how not only κ^* but also the scaling matrix D is sensitive to inaccuracies.

▶ **Example 20** (Floating-point arithmetic and κ)**.** For $a, C \in \mathbb{R}^+$ consider the matrix

$$A := \begin{pmatrix} a & a & Ca+\varepsilon \\ a & a & Ca \end{pmatrix}.$$

The only circuit of $M(A)$ is $\{1,2\}$ and $\mathbf{g} = (1,-1,0)^\top$ is its elementary vector, implying $\kappa_A = 1$. An algorithm based on floating-point numbers could however falsely believe that $\{2,3\}$ is a circuit with an elementary vector $(0, C, -1)^\top$ and therefore that $\kappa_A = C$.

▶ **Example 21** (Floating-point arithmetics and κ^*)**.** The vector matroid of matrix

$$A := \begin{pmatrix} \bar\kappa & 1 & \bar\kappa & 0 \\ \bar\kappa & \bar\kappa^2 & 0 & \bar\kappa^2 \\ \varepsilon & 0 & 0 & 0 \end{pmatrix}$$

for $\bar\kappa > 1$ has a single circuit $\{2,3,4\}$. The scaling matrix

$$D := \begin{pmatrix} 1 & & & \\ & \bar\kappa & & \\ & & 1 & \\ & & & \bar\kappa \end{pmatrix} \qquad \text{gives} \qquad AD = \begin{pmatrix} \bar\kappa & \bar\kappa & \bar\kappa & 0 \\ \bar\kappa & \bar\kappa^3 & 0 & \bar\kappa^3 \\ \varepsilon & 0 & 0 & 0 \end{pmatrix}.$$

The only circuit then has an elementary vector $\mathbf{g} = (0,1,-1,-1)^\top$, and thus $\kappa^*_A = \kappa_{AD} = 1$.

An algorithm using floating-point numbers may however give results as if $\varepsilon = 0$ due to loss of precision, effectively making $\kappa^*_A = \bar\kappa$.

If we were to use the exactly computed scaling matrix D in an inexact context (assuming ε is treated as 0), we get $\kappa_{AD} = \bar\kappa^2$. This shows that not only is the value κ^* sensitive to inaccuracies, but the same is also true for the scaling matrix D.

3.2.2 Relevant Matroid Operations

For the purposes of the algorithm, the following matroid operations are relevant:

- Components() splits the vector matroid to matroids for its components,
- Basis() finds any basis of the vector matroid,
- FundamentalCircuits(B) finds all fundamental circuits with respect to the basis B,
- ElementaryVector(C) finds an elementary vector for the circuit C and
- UniqueCircuit(B, U, V) finds a circuit contained in $(B \setminus U) \cup V$ that is guaranteed to be unique, where $U \subseteq B$, $V \subseteq [n] \setminus B$.

3.2.3 Utilizing SageMath

SageMath comes with the `LinearMatroid` class which natively provides most of the operations, and allows easy implementation of the remaining ones. If `M` is an instance of `LinearMatroid`, then COMPONENTS() is implemented by `M.components()`, BASIS() is implemented by `M.basis()` and FUNDAMENTALCIRCUITS(B) can be implemented by using the method `M.fundamental_circuit(B, e)` for every non-basic element e.

The operations ELEMENTARYVECTOR(C) and UNIQUECIRCUIT(B, U, V) can be implemented either using the method in Lemma 10 or the one in Proposition 11. In both cases, we will work directly with the matrix A. In the first case this can be done using `A.matrix_from_columns(C).right_kernel().basis()[0]` for ELEMENTARYVECTOR(C) and `A.matrix_from_columns(B.difference(U).union(V)).right_kernel().basis()[0].support()` for implementing the operation UNIQUECIRCUIT(B, U, V).

The latter option is a bit more involved. Firstly, the matrix A needs to be converted to RREF using Gauss-Jordan elimination. Then, we construct a "vector finding graph" that will allow for efficient implementation of the procedure.

▶ **Definition 22** (Vector finding graph). *Let A be a matrix in RREF such that $M(A)$ is connected, let U, V and B be sets as above and let $S := (B \setminus U) \cup V$. Consider the submatrix $\bar{A}$ obtained by selecting the columns in S and the rows where columns in U have their only nonzero-value (this value is 1). Let $\overline{uv}$ denote the indices such that $\bar{A}_{\overline{uv}}$ corresponds to A_{uv}.*

Then, the vector finding graph *of A is a weighed directed graph G such that*

$$\begin{aligned} V(G) &:= [n], \\ E(G) &:= \left\{ (u,v) \in [n]^2 \,\middle|\, \begin{array}{l} \textit{there is a row } r_{\overline{uv}} \textit{ in } \bar{A} \textit{ whose only nonzero values} \\ \textit{are in columns } \bar{u} \textit{ and } \bar{v} \end{array} \right\}, \\ w_G((u,v)) &:= -\frac{\bar{A}_{r_{\overline{uv}}\bar{u}}}{\bar{A}_{r_{\overline{uv}}\bar{v}}}. \end{aligned}$$

This graph can be constructed in time $\mathcal{O}(nm)$. Its meaning is that for an elementary vector $\mathbf{g}$ of the unique circuit contained in S it holds that if $(u, v) \in E(G)$, then $\frac{g_v}{g_u} = w_G((u, v))$.

The procedure from Proposition 11 can be implemented as follows: start with an empty vector $\mathbf{w}$ and set $w_{v_1} \leftarrow 1$. Then, run a depth-first search (DFS) from v_1 in the vector finding graph and for every discovered edge (i, j), set $w_j \leftarrow w_i \cdot w_G((i, j))$. Processing edges in this order guarantees that every edge (i, j) is used only after the value w_i is known, that everything that can be propagated, and that no value g_j is computed more that once. Lastly, set values to indices in $B \cap S$ such that $\mathbf{w} \in \mathrm{Ker}(A)$. This is done by performing the matrix-vector multiplication $\mathbf{w}_{B\cap S} \leftarrow -(A\mathbf{w})_{B\cap S}$, which is a relatively simpler operation than solving a generic system of linear equations as in Lemma 10.

▶ Remark 23. It suffices to convert the matrix A to RREF only once in the course of the run of the algorithm and then reuse it for all calls of ELEMENTARYVECTOR and UNIQUECIRCUIT.

Upon inspection of the source code of SageMath, we have discovered that its vector matroid implementation cannot leverage matrix sparsity. In particular, the matrix is converted to an internal dense representation when constructing the vector matroid object. Because matrices of problems in Netlib and MIPLIB usually have many zeroes, representing them densely has great impact on both memory and runtime of the operations. It is therefore natural to explore other options that work with the matrix directly in its sparse representation.

3.2.4 Sparsity and RREF

While it might seem sufficient to modify SageMath's algorithm's internals to work with the sparse representation of A, there are more fundamental problems that make it impossible to fully avoid the memory blow-up caused by converting to a dense representation. The root of the problem lies in needing the RREF of matrix A. Even if A is sparse, converting it to RREF it is often not the case and the representation becomes inefficient. This is known as matrix *fill-in* and has been studied for example by Brayton et al. [6]. Because of this, the memory savings (and consequently time savings) that motivated the use of sparse matrices are not directly possible.

Note that while we can avoid using RREF explicitly by using the alternative method in Lemma 10, the SageMath implementation of vector matroids nevertheless echelonizes the matrix A internally. It might be possible to reorder the columns of A in a way which minimizes the fill-in, however, we are only aware of strategies for minimizing fill-in when computing LU and QR decompositions, but not for computing RREF.

3.2.5 Comparison of the Methods

The choice of a suitable implementation of the UniqueCircuit method depends on the available computational resources. The generic method of Lemma 10 based on sparse operations is a good fit for scenarios where available memory is limited. The specialized method of Proposition 11 internally uses a dense representation (due to SageMath's implementation of vector matroids) and exhibited higher memory usage during our tests, but had the benefit of considerably shorter run times.

We have opted for the specialized method; we believe that a practical implementation of the scaling algorithm should, in the context of the UniqueVector method, focus either on optimizing sparsity, or avoiding RREF altogether. Nevertheless, the generic and "natively" sparse method can conceivably be more practical for extremely memory-demanding problems.

3.3 Other Implementation Concerns

Besides working with the vector matroid, the algorithm also solves the maximum geometric mean problem for the pairwise imbalance measure digraph, and the single-source shortest path problem for finding the rescaling diagonal values.

3.3.1 Maximum Geometric Mean Problem

The maximum geometric mean problem can be converted to the minimum geometric mean problem in a straightforward way, and then further reduced to the minimum arithmetic mean problem by taking logarithms of all edge weights. This well-known problem can then be solved by Karp's algorithm [16].

We have not found an implementation of Karp's algorithm in SageMath, so we opted for our own implementation in plain Python. It may be possible to significantly speed up this step of the algorithm by choosing a lower-level language implementation, however, this procedure only represents a very small portion of the total runtime.

At this point in the algorithm we no longer need to use exact arithmetic; we will formally argue why this is not a problem later (Lemma 24).

3.3.2 Single-source Shortest Path Problem

The graph in which we need to find shortest paths from u to all other vertices v contains negative edges, so the natural choice is to use the Bellman-Ford algorithm. This is implemented in SageMath as the method `shortest_paths(u, by_weight=True, algorithm="Bellman-Ford_Boost")` of SageMath graph objects, and it internally calls a Boost implementation of Bellman-Ford which does not support weights that are SageMath-specific infinite-precision rational numbers.

By using inexact arithmetic, the shortest paths can only change by a very small amount. Because the shortest path lengths are exactly the diagonal elements of the obtained rescaling matrix, this causes us to have found a slightly different rescaling. The following lemma shows that this rescaling cannot be much worse than the one found when using exact values.

▶ **Lemma 24.** *Let $\alpha \geq 1$, and let $D \in \mathbf{D}$ and $D' \in \mathbf{D}$ satisfy $D'_{ii} \in \left[\frac{1}{\alpha} D_{ii}, \alpha D_{ii}\right]$ for every $i \in [n]$. Then, $\kappa_{AD'} \leq \alpha^2 \kappa_{AD}$.*

Proof. Consider the circuit C and indices i, j such that $\kappa_{AD'} = \kappa_{ij}^{AD'}(C)$. Let $\mathbf{g}^{\mathbf{AD}}$ and $\mathbf{g}^{\mathbf{AD}'}$ be the elementary vectors corresponding to C such that $\mathbf{g}^{\mathbf{AD}} \in \operatorname{Ker}(AD)$ and $\mathbf{g}^{\mathbf{AD}'} \in \operatorname{Ker}(AD')$. Then,

$$\kappa_{AD'} = \kappa_{ij}^{AD'}(C) = \left| \frac{g_j^{AD'}}{g_i^{AD'}} \right| \leq \frac{\alpha \left| g_j^{AD} \right|}{\alpha^{-1} \left| g_i^{AD} \right|} \leq \alpha^2 \kappa_{AD}. \qquad ◀$$

Even though the original graph is guaranteed to not contain negative cycles, inexact arithmetics may introduce slightly negative cycles. In that case, the Boost implementation fails. For this reason, we increase all edge weights by a value ε to compensate for such inaccuracies. thus making the shortest path up to $n \cdot \varepsilon$ longer.

4 Performance Comparison

Our main result is the evaluation of the performance of LP solvers both on the original problem and the problem after applying the approximate optimal rescaling. Note that while Dadush et al. [9] give some performance guarantees (indeed, that is their primary motivation), their relevance in our case is somewhat limited. Firstly, all claims in [9] are relevant for the interior point LP algorithm of Vavasis and Ye [23], but none of the tested algorithms directly use this method and, to the best of our knowledge, no theoretical claims have been made about the effects of the rescaling for other algorithms.

The second thing to note is that for problems with large κ^*, the rescaling may be very far from an optimal one. Still, the potentially weak guarantee is only a worst-case one, and the algorithm may do well on real-world instances nevertheless.

Lastly, despite the potential pitfalls of working with the circuit imbalance measure using non-exact arithmetic explained earlier, we also evaluate the performance of non-exact (floating-point) LP algorithms. We do so for completeness and because the vast majority of use cases of LP are in FP (not exact) arithmetic.

4.1 Exact Solvers

Some evaluated solvers can be configured to find the solution using exact arithmetic. However, even these solvers read its input from MPS files and at least some of them parse the numeric values in it into floating-point numbers and only then into an exact data type. Because of issues outlined earlier, a naive use of an exact solver will lead to problems with precision loss.

We propose the following workaround. Due to the properties of the IEEE 754 floating-point number arithmetic standard, multiplying an FP number by a (possibly negative) power of two does not cause precision loss if the mathematically correct result can be represented as an IEEE 754 FP number. Therefore, if we restrict the elements of our rescaling matrix to be such powers of two, the rescaled problem as loaded by the exact solver will be mathematically precisely the original problem with scaled columns of the problem matrix.

The next lemma shows that rounding down the rescaling matrix obtained by our algorithm to integral powers of two gives a rescaling not much worse than the original one:

▶ **Lemma 25.** *Let $D = \mathrm{Diag}(\mathbf{d}) \in \mathbf{D}$. Define $\hat{d}_i$ to be the nearest smaller integral power of two than d_i for every $i \in [n]$ and let $\hat{D} := \mathrm{Diag}(\hat{\mathbf{d}})$. Then, $\kappa_{A\hat{D}} \leq 2 \cdot \kappa_{AD}$.*

The proof of this lemma is almost identical to the proof of Lemma 24.

4.2 Experimental Methodology

We have measured solving times for both original and rescaled instances using the open-source solvers GLPK and SCIP [1] and using the proprietary solver Gurobi [15]. In all cases, garbage collection was disabled during the measuring process to keep it from interfering with the measurement. We exclude parsing time from the measurement. We did our best to disable any built-in scaling so as to not interfere with our scaling. We repeat every measurement 10 times and then consider the arithmetic mean to mitigate faulty measurements.

We include problem instances that were solved in a very short time (under 0.0003 seconds). As was pointed out by the reviewers, the time ratio for these instances might not be very meaningful. We are thankful for this remark and we will make a more careful selection of the measured results in a future journal version of this paper.

Our main interest is the relative effect the rescaling has on the solver run time. Let τ_o be the time that a solver needs to solve the original instance, and τ_r the time the solver needs to solve the rescaled instance. Define the *time ratio* of the rescaling as $\rho := \left(\frac{\tau_r}{\tau_o} \cdot 100\right)\%$. For floating-point solvers, we measure τ_r using the precise rescaling, while exact solvers we use the rescaling rounded to powers of two as described in Subsection 4.1.

To measure τ_o, an MPS file created from the problem converted to semi-standard form is used in contrast to using the original MPS files. This is done so that the difference between the two tested problems is solely in column scaling and structural differences of the matrices can not affect performance.

The software versions used to perform the measurements were GLPK v4.65 (floating point simplex and interior point methods, and an exact simplex method), Gurobi v10.0.2 (floating-point default method), SCIP v8.0.3 (floating-point simplex) and an exact version of SCIP v8.0.0.1[4] (exact simplex). All solver settings unrelated to scaling or exact arithmetic are set to their default values. We thank the reviewers for noting that this makes it more difficult to draw conclusions about the results of Gurobi and SCIP, as they may use both simplex and interior point methods as appropriate for every instance. Because changing this setting would require running new experiments with different solver settings, we will consider this in a future journal version of this paper.

We used a computational cluster where each machine is equipped with AMD EPYC 7532 CPUs which have SPECfp2017 score of 6.9 per core. The solvers were limited to use a single CPU core.

[4] Available from `https://github.com/scipopt/scip/tree/exact-rational`

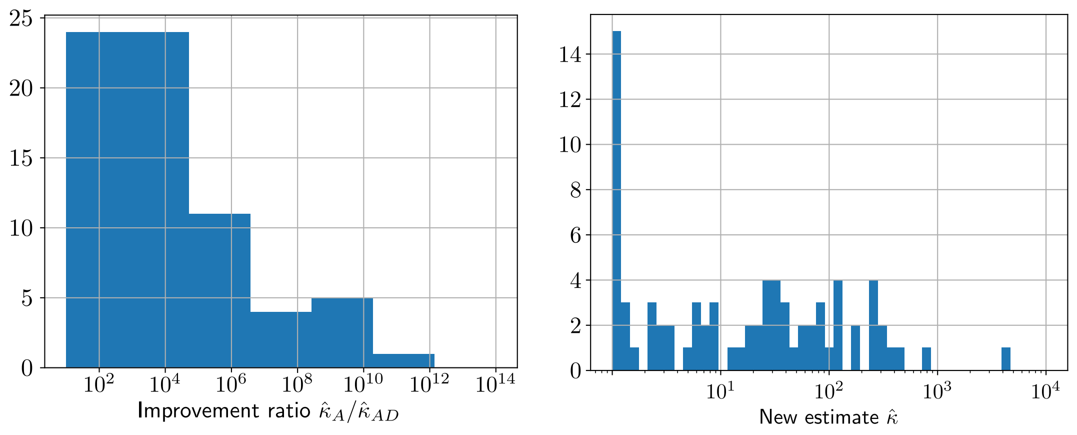

Figure 2 Two histograms depicting the improvement of $\hat{\kappa}$. In both plots, the y axis depicts the number of instances. On the left, the x axis depicts the relative improvement $\hat{\kappa}_A/\hat{\kappa}_{AD}$, where larger ratio represents bigger improvement. On the right, the x axis depicts the absolute new value of $\hat{\kappa}$, where smaller values mean smaller imbalance lower bound.

4.3 Measurement Results

We managed to compute the approximate rescaling for a total of 142 problem instances of MIPLIB and Netlib. We ran the measurements for all solution methods and all problem instances. We exclude measurements that have not finished in the allotted time.

The best average time ratio was attained for GLPK using an interior-point method with a value of 132%. The highest number of problem instances with time ratio under 100% was attained for Gurobi with 51 (36%) said problem instances.

The lowest (best) achieved time ratio is 7.6 %, meaning that applying the rescaling led to a 13× speed-up of the solver. On a similar note, for 112 of the problem instances the rescaling had a time ratio smaller than 100% for at least one of the solution methods, providing a speed-up in the runtime.

We have not observed strong correlations between speed-ups for the various solvers.

Detailed results of the experiment can be found in Figure 1. Columns of the scatter plot correspond to individual instances. In addition to time ratios for all solving methods, we have included the best and the mean time ratio for every problem instance.

4.3.1 Improvement of κ

Besides possible performance benefits, it is also natural to ask about the improvement of the circuit imbalance measure. Ideally, we would report statistics about the ratio κ_A/κ_{AD}, however, due to the intractability of computing these quantities, we resort to comparing the improvement with respect to the set of circuits $\mathcal{C}'$ computed by the algorithm. Specifically, recall that $\hat{\kappa}_A$ is a lower bound on κ_A based on $\mathcal{C}'$, and let $\hat{\kappa}_{AD}$ be the largest imbalance of any circuit in $\mathcal{C}'$ with respect to AD; thus, $\hat{\kappa}_{AD}$ gives a lower bound on κ_{AD}. Figure 2 shows the relative improvement (wrt. $\mathcal{C}'$) $\hat{\kappa}_A/\kappa_{\hat{AD}}$, and the (absolute) new estimate $\hat{\kappa}_{AD}$. Again, note that this is a lower bound.

Admittedly, comparing two lower bounds gives a fairly weak claim. It is however unclear whether there is much of an alternative: one could try generating (or sampling) many circuits to enlarge $\mathcal{C}'$ and get better bounds, but short of computing the κ's exactly, it is impossible to show (useful) definite bounds on the quality of the rescaling.

References

1 Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. URL: `http://www.optimization-online.org/DB_HTML/2021/12/8728.html`.

2 Steffen Borgwardt, Cornelius Brand, Andreas Emil Feldmann, and Martin Koutecký. A note on the approximability of deepest-descent circuit steps. *Oper. Res. Lett.*, 49(3):310–315, 2021. `doi:10.1016/J.ORL.2021.02.003`.

3 Steffen Borgwardt, Jesús A. De Loera, Elisabeth Finhold, and Jacob Miller. The hierarchy of circuit diameters and transportation polytopes. *Discret. Appl. Math.*, 240:8–24, 2018. `doi:10.1016/J.DAM.2015.10.017`.

4 Steffen Borgwardt, Tamon Stephen, and Timothy Yusun. On the circuit diameter conjecture. *Discret. Comput. Geom.*, 60(3):558–587, 2018. `doi:10.1007/S00454-018-9995-Y`.

5 Steffen Borgwardt and Charles Viss. A polyhedral model for enumeration and optimization over the set of circuits. *Discret. Appl. Math.*, 308:68–83, 2022. `doi:10.1016/J.DAM.2019.07.025`.

6 Robert K. Brayton, Fred G. Gustavson, and Ralph A. Willoughby. Some results on sparse matrices. *Math. Comp.*, 24:937–954, 1970. `doi:10.1090/s0025-5718-1970-0275643-8`.

7 Ke Chen. *Matrix preconditioning techniques and applications*, volume 19. Cambridge University Press, 2005.

8 Richard Cole, Christoph Hertrich, Yixin Tao, and László A. Végh. A first order method for linear programming parameterized by circuit imbalance. *CoRR*, abs/2311.01959, 2023. `doi:10.48550/ARXIV.2311.01959`.

9 Daniel Dadush, Sophie Huiberts, Bento Natura, and László A. Végh. A scaling-invariant algorithm for linear programming whose running time depends only on the constraint matrix. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 761–774, New York, NY, USA, June 2020. Association for Computing Machinery.

10 Daniel Dadush, Zhuan Khye Koh, Bento Natura, and László A. Végh. On circuit diameter bounds via circuit imbalances. In Karen I. Aardal and Laura Sanità, editors, *Integer Programming and Combinatorial Optimization - 23rd International Conference, IPCO 2022, Eindhoven, The Netherlands, June 27-29, 2022, Proceedings*, volume 13265 of *Lecture Notes in Computer Science*, pages 140–153. Springer, 2022. `doi:10.1007/978-3-031-06901-7_11`.

11 Farbod Ekbatani, Bento Natura, and László A Végh. Circuit imbalance measures and linear programming. *Surveys in Combinatorics*, pages 64–114, 2022.

12 Bernd Gärtner and Jiří Matoušek. *Understanding and using linear programming.* Universitext. Springer, 2007.

13 Gay, D. M. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society Committee on Algorithms Newsletter*, 13, 1985. Data available from `https://netlib.org/lp/data/`.

14 Ambros M. Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff T. Linderoth, Marco E. Lübbecke, Hans D. Mittelmann, Derya B. Özyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Math. Program. Comput.*, 13(3):443–490, 2021. `doi:10.1007/s12532-020-00194-3`.

15 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: `https://www.gurobi.com`.

16 Richard M. Karp. A characterization of the minimum cycle mean in a digraph. 23:309–311, 1978. `doi:10.1016/0012-365x(78)90011-0`.

17 Jakub Komárek. Experimental analysis of scaling methods for lp. 2023.

18 Jesús A. De Loera, Sean Kafer, and Laura Sanità. Pivot rules for circuit-augmentation algorithms in linear optimization. *SIAM J. Optim.*, 32(3):2156–2179, 2022. `doi:10.1137/21M1419994`.

19 István Maros. *Computational techniques of the simplex method*, volume 61. Springer Science & Business Media, 2002.

20 J. G. Oxley. *Matroid Theory*. Oxford University Press, New York, 1992.

21 The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.7)*, 2022. `https://www.sagemath.org`.

22 Levent Tunçel. Approximating the complexity measure of vavasis-ye algorithm is np-hard. *Math. Program.*, 86(1):219–223, 1999. `doi:10.1007/s101070050087`.

23 Stephen A. Vavasis and Yinyu Ye. A primal-dual interior point method whose running time depends only on the constraint matrix. 74:79–120, 1996. `doi:10.1007/bf02592148`.