Computing the Spark: Mixed-Integer Programming for the (Vector) Matroid Girth Problem

Andreas M. Tillmann*

(draft, last updated March 7, 2018)

Abstract

We investigate the NP-hard problem of computing the spark of a matrix (i.e., the smallest number of linearly dependent columns), a key parameter in compressed sensing and sparse signal recovery. To that end, we identify polynomially solvable special cases, gather upper and lower bounding procedures, and propose several exact (mixed-)integer programming models and linear programming heuristics. In particular, we develop a branch & cut scheme to determine the girth of a matroid, focussing on the vector matroid case, for which the girth is precisely the spark of the representation matrix. Extensive numerical experiments demonstrate the effectiveness of our specialized algorithms compared to general-purpose black-box solvers applied to several mixed-integer programming models.

1 Introduction

The *spark* of a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is defined as the smallest number of linearly dependent columns, i.e.,

$$\operatorname{spark}(\boldsymbol{A}) \coloneqq \min\{\|\boldsymbol{x}\|_0 : \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\, \boldsymbol{x} \neq \boldsymbol{0}\}, \tag{1}$$

where $\|\boldsymbol{x}\|_0 \coloneqq |\{j : x_j \neq 0\}|$. In the context of compressed sensing (cf. [31, 36]) and the sparse recovery problem

$$\min \|\boldsymbol{x}\|_0 \quad \text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2}$$

the spark is essential regarding solution uniqueness; in particular, every k-sparse vector $\hat{\boldsymbol{x}}$ uniquely solves (2) with $\boldsymbol{b} \coloneqq \boldsymbol{A}\hat{\boldsymbol{x}}$ if and only if $2k < \operatorname{spark}(\boldsymbol{A})$. The spark is also a key parameter for identifying ambiguities in linear sensor arrays (see, e.g., [63]) or existence of unique tensor decompositions (by the relation to the Kruskal rank, which equals $\operatorname{spark}(\boldsymbol{A}) - 1$, see, e.g., [57, 56, 83]), verification of k-stability in the context of matrix completion (holds if and only if $\operatorname{spark}(\boldsymbol{A}) = n - k + 1$, see [84]), and other applications.

The spark of a matrix $\boldsymbol{A}$ is also known as the *girth* of the matroid associated with the columns of $\boldsymbol{A}$, usually called the *vector matroid* represented by $\boldsymbol{A}$. Spark computation for general (even integer-valued) $\boldsymbol{A}$ was shown to be NP-hard in [78], complementing previous intractability results for the girth of transversal matroids [64] and binary matroids (i.e., vector matroids over $\mathbb{F}_2$) [80].

Apparently, only two methods to directly determine $\operatorname{spark}(\boldsymbol{A})$ are known so far, namely the obvious brute-force approach and an incremental one employing a sequence of problems of the form (2), which are themselves NP-hard in general [38] and very challenging in practice [49]. In principle, via matroid duality, one could also employ algorithms designed for computing the cogirth (i.e., the girth of the dual matroid; for vector matroids in particular, see, e.g., [22, 53, 8, 7]). However, these require knowledge of the dual representation matrix, the computation of which introduces an undesired overhead and can reduce numerical reliability. In this paper, we explore several direct (mixed-)integer programming (MIP) approaches to spark computation and, in particular, develop a specialized branch & cut algorithm. Numerical experiments are carried out to compare the different formulations and to demonstrate the effectiveness of our specialized method.

The paper is organized as follows: After fixing some notation, we first give an overview of relevant previous work related to matroid girth computation (particularly w.r.t. vector matroids) and then take another look at

*A. M. Tillmann is with the Operations Research Group and the Visual Computing Institute at RWTH Aachen University, Germany (e-mail: tillmann@or.rwth-aachen.de).

the computational complexity of spark computation (Section 3); in particular, we identify some polynomially solvable special cases of this generally NP-hard task. In Section 4, we collect upper and lower bounds on the spark, and in Section 5 we propose several mixed-integer programming (MIP) formulations to compute the exact value. In Section 6, we introduce a pure integer (in fact, binary) program for the computation of the girth of any matroid accessible via an independence oracle, and develop a specialized branch & cut algorithm for the vector matroid case (i.e., spark computation) in Section 7. We present numerical experiments and comparisons of the different approaches in Section 8 and discuss some extensions and related problems in the concluding Section 9.

Throughout, we assume w.l.o.g. that $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ with $\operatorname{rank}(\boldsymbol{A}) = m < n$ and that $\boldsymbol{A}$ contains no all-zero column, which is arguably the natural situation and can always easily be established by simple preprocessing: If for some j, the column $\boldsymbol{A}_j = \boldsymbol{0}$, then $\operatorname{spark}(\boldsymbol{A}) = 1$ trivially, and in case of rank deficiency, redundant—i.e., linearly dependent—rows can be identified and removed a priori, cf. Section 7.1.

1.1 Notation & Preliminaries

We denote the nullspace (kernel) of a matrix $\boldsymbol{A}$ by $\mathcal{N}(\boldsymbol{A})$. The identity matrix will be denoted by $\boldsymbol{I}$, the all-ones vector by $\mathbb{1}$, and the all-zero vector or matrix by $\boldsymbol{0}$; the respective dimensions will always be clear from the context. A submatrix of a matrix $\boldsymbol{A}$ formed by sets of row and column indices R and C is denoted by $\boldsymbol{A}_{RC}$ (or $\boldsymbol{A}_{R,C}$); if either R or C is the whole row/column set, we write $\boldsymbol{A}_{\cdot C}$ or $\boldsymbol{A}_{R\cdot}$, respectively. Similarly, $\boldsymbol{x}_R$ is the restriction of a vector $\boldsymbol{x}$ to components indexed by R. Further, the j-th column of $\boldsymbol{A}$ will be denoted by $\boldsymbol{A}_j$ and its i-th row by $\boldsymbol{a}_i^\top$.

We denote the number of nonzeros in a vector $\boldsymbol{x}$ (the so-called "ℓ_0-norm") by $\|\boldsymbol{x}\|_0 \coloneqq |\operatorname{supp}(\boldsymbol{x})|$, where $\operatorname{supp}(\boldsymbol{x}) \coloneqq \{j : x_j \neq 0\}$ is the support of $\boldsymbol{x}$; the standard ℓ_p-norms for $1 \leq p \leq \infty$ are denoted by $\|\boldsymbol{x}\|_p$. For a positive integer $n \in \mathbb{N}$, we abbreviate $[n] \coloneqq \{1, 2, \ldots, n\}$. For $\boldsymbol{y} \in \mathbb{R}^n$ and $S \subseteq [n]$, we define $\boldsymbol{y}(S) \coloneqq \sum_{j\in S} y_j$. The complement of a set S w.r.t. a superset $T \supseteq S$ (that will always be clear from the context) is denoted by $\overline{S} \coloneqq T \setminus S$.

We assume familiarity with basic graph theory, linear algebra, and (mixed-)integer programming, and refer to [70] for details on matroid theory beyond the following brief introduction. A matroid $\mathcal{M} = (E, \mathcal{I})$ consists of a finite ground set E and a collection $\mathcal{I}$ of subsets of E, called *independent sets*, satisfying the following axioms: (I1) $\emptyset \in \mathcal{I}$, (I2) $I \in \mathcal{I}$ *and* $J \subseteq I \Rightarrow J \in \mathcal{I}$, and (I3) $I, J \in \mathcal{I}$ *and* $|I| < |J| \Rightarrow \exists e \in J \setminus I$ *such that* $I \cup \{e\} \in \mathcal{I}$. The inclusion-wise maximal independent sets of a matroid are its *bases*, the set of which we denote by $\mathcal{B}$. A matroid is actually fully characterized by its bases, so we may also write $\mathcal{M} = (E, \mathcal{B})$. Every basis $B \in \mathcal{B}$ has the same cardinality $|B| = r_{\mathcal{M}}$, the *rank* of the matroid. Similarly, the rank $r_{\mathcal{M}}(S)$ of a subset $S \subseteq E$ is the cardinality of a largest independent set contained in S. A subset of E that is not in $\mathcal{I}$ is called *dependent*, or sometimes *cycle*; we write $\mathcal{C}$ for the set of *circuits*, which are the inclusion-wise minimal dependent sets. Thus, the *girth* of a matroid is the minimum cardinality of its circuits. Also, note that every independent set can be extended to a basis and that every cycle contains a circuit.

The *dual matroid* $\mathcal{M}^*$ is a matroid over the same ground set whose bases are the complements (w.r.t. E) of the bases of $\mathcal{M}$, i.e., $\mathcal{M}^* = (E, \mathcal{B}^*)$ with $\mathcal{B}^* = \{\overline{B} : B \in \mathcal{B}\}$. The prefix "co" is used for the respective notions w.r.t. the dual matroid (e.g., cobasis, cocircuit, cogirth etc.), and an asterisk marks the respective symbols ($\mathcal{B}^*$, $\mathcal{C}^*$, and so on).

The *vector matroid* $\mathcal{M}[\boldsymbol{A}]$ is defined over the set of column indices of a *representation matrix* $\boldsymbol{A}$, and the notion of independence pertains to *linear* independence (of column subsets). In particular, nontrivial nullspace vectors with irreducible supports correspond to the circuits of this matroid (actually, their supports do, but for the sake of simplicity, we often neglect this distinction), i.e., the inclusion-wise minimal sets of linearly dependent columns. Thus, $\operatorname{spark}(\boldsymbol{A})$ is precisely the girth of $\mathcal{M}[\boldsymbol{A}]$. The bases of $\mathcal{M}[\boldsymbol{A}]$ are given by the column index sets of regular $m \times m$ submatrices of $\boldsymbol{A}$ (recall that we assume $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ with $r_{\mathcal{M}[\boldsymbol{A}]} = \operatorname{rank}(\boldsymbol{A}) = m < n$).

Since the girth of $\mathcal{M}[\boldsymbol{A}]$ is the same as the cogirth of the dual matroid $\mathcal{M}^*[\boldsymbol{A}]$, every scheme to compute the cogirth of vector matroids (also called the *cospark*) can, in principle, be employed to compute the spark, after determining a representation matrix for $\mathcal{M}^*[\boldsymbol{A}]$. Assuming w.l.o.g. that $\boldsymbol{A}$ is given in the so-called standard form $\boldsymbol{A} = (\boldsymbol{I}, \tilde{\boldsymbol{A}})$ (which can be obtained by transforming $\boldsymbol{A}$ to reduced row echelon form), then $\mathcal{M}^*[\boldsymbol{A}] = \mathcal{M}[\boldsymbol{B}^\top]$ for $\boldsymbol{B} \coloneqq (\tilde{\boldsymbol{A}}^\top, -\boldsymbol{I})^\top \in \mathbb{R}^{n\times(n-m)}$ with $\operatorname{rank}(\boldsymbol{B}) = n - m$; note that $\boldsymbol{B}$ is a basis matrix for the nullspace $\mathcal{N}(\boldsymbol{A})$, i.e., $\operatorname{span}(\boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-m}) = \mathcal{N}(\boldsymbol{A})$.

2 Related Work

In the literature, one can find various works investigating properties of and methods to determine circuits or cocircuits of matroids. Some noteworthy earlier results include a method to enumerate all circuits given knowledge of a matroid's set of bases, see [66], and an investigation of the complexity of testing several matroid properties (in particular, matroid uniformity, which for vector matroids translates to the notion of "full spark" [4], i.e., $\operatorname{spark}(\boldsymbol{A}) = m + 1$), see [71, 48]. The topic of enumerating circuits or cocircuits (in fact, equivalently, matroid hyperplanes, which are the complements of cocircuits) has been treated further in [52] and [73], respectively. Naturally, such approaches are of limited value for (co)girth computation, since for the latter we do not need to know *all* (co)circuits; given the potentially huge total number of (co)circuits, it would therefore seem quite wasteful to revert to explicit enumeration schemes in order to find one of smallest cardinality. The same can, of course, be said for naive brute-force/exhaustive rank-testing methods that essentially test all k-element subsets for varying k until the girth has been found and verified. A concrete such method is stated in [22, Algorithm 1]; that work also shows that if a matroid is disconnected (i.e., it can be expressed as the direct sum of matroids with smaller ground sets), rank-testing schemes to determine girth or cogirth can be applied to a set of smaller, independent subproblems to decrease the overall effort required.

For the special case of vector matroids, such matroidal decomposition amounts to finding an equivalent representation matrix exhibiting a (bordered) block-diagonal structure – the (co)girth of the full matroid is then given by the smallest (co)girth of those associated with the column-submatrices corresponding to the separate blocks. However, determining such a transformation, already when restricted to permutations of rows and columns (i.e., not considering potential benefits from further elementary row operations), is essentially equivalent to a (hyper-)graph partitioning problem, which is known to be NP-hard [10, 38], so that in practice, one resorts to heuristics. Experiments in [22, 13, 8] confirmed the positive effects of exploiting such (purely permutation-based) bordered block-diagonal decomposition in the context of algorithms to compute the cogirth of decomposable vector matroids. A mixed-integer programming model to compute the cogirth of a vector matroid was introduced in [53]; in Section 5.3 below, we will describe its application to the spark problem. The work [13] combined the MIP idea from [53] with the decomposition/rank-testing strategy from [22], outperforming both predecessor algorithms on decomposable instances. Nonetheless, since the potential benefits of matroid/matrix decomposition have been demonstrated before and, in fact, basically form a preprocessing step applicable to all (co)girth problems, we believe it should be treated as a separate problem of interest and therefore do not delve further into this aspect in the present paper. (It is also worth mentioning that, for many instances in the test set for our numerical experiments, the dual matroid representation matrices obtained to potentially apply cogirth algorithms appear not to be decomposable, at least by the usual permutation-based heuristics.)

Most related to the binary integer program we propose in Section 6 is an analogous IP model for the vector matroid cogirth problem set up in [8] (see also [7]). Although very similar in spirit, our "primal" formulation—tackling the girth/spark problem directly, avoiding the switch to a dual matroid perspective (and thus the potentially costly and numerically undesirable calculation of the dual representation matrix) in order to apply cogirth methods—has several advantages regarding the solution process: The primal viewpoint offers a set of propagation rules (to infer variable fixings and to prune/cut off branches of the search tree) that are not accessible for the cogirth IP from [8], and the relation to compressed sensing suggests, on the one hand, useful lower bounds (cf. Section 4) that further help in pruning, and on the other hand, efficient primal heuristics for the spark problem that help finding good or even optimal solutions early on. Those aspects, as well as further algorithmic components such as a branching rule, are discussed in detail in Sections 6 and 7; in particular, further justification of our direct approach versus using the cogirth IP on a dual representation matrix is provided in Remarks 17 and 19.

As mentioned before (e.g., in [34]), the spark can also be computed by solving a sequence of instances of (2) in which the right-hand side vector $\boldsymbol{b}$ represents each column of the matrix $\boldsymbol{A}$ in turn. These problems can be solved as box-constrained mixed-integer programs (see Section 5.1 for details); the only specialized exact solver for (2) we are aware of is the branch & cut scheme from [49], which unfortunately only appears to work well for rather small instances. In fact, we can also employ a spark solver to tackle problems of the form (2) for suitably sparse solutions, by means of the following result:

Proposition 1. *Let* $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ *with* $\operatorname{spark}(\boldsymbol{A}) \geq k$*, and let* $\hat{\boldsymbol{A}} \coloneqq (\boldsymbol{A}, \boldsymbol{A}\hat{\boldsymbol{x}})$ *for an* $\hat{\boldsymbol{x}}$ *with* $\|\hat{\boldsymbol{x}}\|_0 \leq k - 2$.

Then, a vector $\boldsymbol{x}^*$ *with* $\|\boldsymbol{x}^*\|_0 \leq k-2$ *and* $\boldsymbol{A}\boldsymbol{x}^*_{[n]} = \boldsymbol{A}\hat{\boldsymbol{x}}$ *can be recovered from the solution support of*

$$\operatorname{spark}(\hat{\boldsymbol{A}}) = \min\left\{\|\boldsymbol{x}\|_0 \,:\, \hat{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{0},\ \boldsymbol{x} \neq \boldsymbol{0}\right\}.$$

If $\operatorname{spark}(\boldsymbol{A}) \geq 2k+1$*, then* $\hat{\boldsymbol{x}}$ *is recoverable from the unique solution support of the above spark instance.*

Proof. Because $\|\hat{\boldsymbol{x}}\|_0 \leq k-2$, $\operatorname{spark}(\hat{\boldsymbol{A}}) \leq \|\hat{\boldsymbol{x}}\|_0 + 1 \leq k-1 < \operatorname{spark}(\boldsymbol{A})$. Since every $\boldsymbol{x} \neq \boldsymbol{0}$ with $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$ has at least k nonzero entries, $\operatorname{spark}(\hat{\boldsymbol{A}})$ is given by some $\boldsymbol{x}^* = (\tilde{\boldsymbol{x}}^\top, -1)^\top$ (i.e., the index of the last column of $\hat{\boldsymbol{A}}$ must be contained in every spark-defining circuit), so in particular, $\boldsymbol{A}\boldsymbol{x}^*_{[n]} = \boldsymbol{A}\tilde{\boldsymbol{x}} = \boldsymbol{A}\hat{\boldsymbol{x}}$. Moreover, if $\operatorname{spark}(\boldsymbol{A}) \geq 2k+1$, then any $\hat{\boldsymbol{x}}$ with $\|\hat{\boldsymbol{x}}\|_0 \leq k-2$ has $\|\hat{\boldsymbol{x}}\|_0 < \operatorname{spark}(\boldsymbol{A})/2$ and, consequently, is the unique solution to the instance of (2) given by $\boldsymbol{A}$ and $\boldsymbol{b} \coloneqq \boldsymbol{A}\hat{\boldsymbol{x}}$ (cf., e.g., [36]). Thus, in particular, $\operatorname{spark}(\hat{\boldsymbol{A}})$ has a unique solution $\boldsymbol{x}^* = (\tilde{\boldsymbol{x}}^\top, -1)^\top$ (up to scaling), i.e., $\operatorname{supp}(\hat{\boldsymbol{x}}) = \operatorname{supp}(\tilde{\boldsymbol{x}})$. □

This direct applicability to sparse recovery provides a further motivation to study models and specialized algorithms to compute the spark.

The special case of binary vector matroids, where all arithmetic is performed modulo 2, has received a lot of attention in coding theory, where the girth of a binary matroid is better known as the minimum distance of a (linear) code. Integer programming models and successful recent problem-specific branch & cut schemes are discussed, for instance, in [59, 44, 51, 74]. However, in the present paper, we focus on the general (non-binary) vector matroid case—i.e., spark computation—where none of those results are applicable. Nevertheless, the minimum distance of a binary linear code provides a lower bound on the spark of its parity check matrix (see, e.g., [60, Theorem 2]), so good (large-girth) codes also provide good (large-spark) measurement matrices for compressed sensing/sparse recovery – an observation that already led to several further explorations of this aspect, see, for instance, [30, 60, 82].

Finally, it should be mentioned that the binary IP and the resulting branch & cut method we put forth in Sections 6 and 7 actually provide a general framework for matroid girth computation whenever the matroid can be accessed via an independence oracle. (Most parts of the described branch & cut algorithm for the vector matroid case extend straightforwardly to the general situation.) In contrast, to the best of our knowledge, the only alternative for general matroids seems to be the exhaustive rank-testing procedure from [22], which relies on a rank oracle. While rank and independence oracles are well-known to be polynomially equivalent (one can be simulated by polynomially many calls to the other), either one may be more suitable for specific matroids and implementations. For instance, in our implementation for vector matroids, we test for independence by specialized Gaussian elimination routines operating on the considered column subset – thus, in particular, we can abort such a process as soon as dependence is detected, while a rank oracle implemented analogously would, by definition, necessarily have to transform *all* columns of the subset.

3 Remarks on the Computational Complexity

The decision problem associated with spark computation can be stated as follows.

$$\begin{aligned}\textsc{Spark}(\boldsymbol{A}, k) \coloneqq\ &\text{Given a matrix } \boldsymbol{A} \in \mathbb{Q}^{m\times n} \text{ and a positive integer } k \in \mathbb{N},\\ &\text{does there exist an } \boldsymbol{x} \in \mathbb{Q}^n \setminus \{\boldsymbol{0}\} \text{ with } \|\boldsymbol{x}\|_0 \leq k \text{ such that } \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}?\end{aligned}$$

This problem was shown to be (weakly) NP-complete in [78, Theorem 1] (see also [75]), even for all-integer $\boldsymbol{A}$. Consequently, computing $\operatorname{spark}(\boldsymbol{A})$ is NP-hard in general, so unless P=NP, there is no hope for polynomial-runtime solution algorithms. It is presently an open question whether this hardness holds in the strong sense (i.e., is Spark still NP-complete when all entries of the input matrix rather than only their encoding lengths are polynomially bounded by its dimensions?), which would further imply that no fully polynomial-time approximation scheme, nor an exact pseudo-polynomial algorithm, can exist unless P=NP, cf. [38].

In the following, we first point out connections of the spark decision problem to two related and well-studied problems—maximum feasible subsystem (MaxFS) and minimum unsatisfied linear relations (MinULR)—that may seem to yield strong NP-hardness of spark computation; while this unfortunately does not work out, we obtain new complexity results on unrestricted homogeneous variants of these problems based on the existing intractability result for the Spark problem. Then, we proceed to explore some polynomial-time solvable special cases of spark computation.

3.1 Relationship to Homogeneous MinULR/MaxFS and the Cospark

In [5], strong NP-completeness was claimed for the following two decision problems:

$\text{MaxFS}_0^=(\boldsymbol{B}, k) :=$ Given a matrix $\boldsymbol{B} \in \mathbb{Q}^{p\times q}$ and a positive integer $k \in \mathbb{N}$, is there a vector $\boldsymbol{q} \in \mathbb{Q}^q \setminus \{\boldsymbol{0}\}$ that satisfies at least k of the homogeneous equations $\boldsymbol{Bq} = \boldsymbol{0}$?

$\text{MinULR}_0^=(\boldsymbol{B}, k) :=$ Given a matrix $\boldsymbol{B} \in \mathbb{Q}^{p\times q}$ and a positive integer $k \in \mathbb{N}$, is there a vector $\boldsymbol{q} \in \mathbb{Q}^q \setminus \{\boldsymbol{0}\}$ that violates at most k of the homogeneous equations $\boldsymbol{Bq} = \boldsymbol{0}$?

(In fact, [5] only refers to $\text{MaxFS}_0^=$, but clearly, $\text{MinULR}_0^=$ and $\text{MaxFS}_0^=$ are exactly complementary and therefore equivalent when solved to optimality; see also [6, 9].)

The following observation therefore appears to point the way toward obtaining *strong* NP-hardness also for the spark computation problem:

Lemma 2. Spark *and* $\text{MinULR}_0^=$ *are polynomially equivalent.*

Proof. Let $\boldsymbol{A} \in \mathbb{Q}^{m\times n}$ (with $\text{rank}(\boldsymbol{A}) = m < n$) and $k \in \mathbb{N}$ be a given Spark-instance. Let $\boldsymbol{B} \in \mathbb{Q}^{n\times(n-m)}$ with $\text{rank}(\boldsymbol{B}) = n-m$ be a basis matrix for the nullspace of A, i.e., such that $\text{span}(\boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-m}) = \mathcal{N}(\boldsymbol{A})$; note that such $\boldsymbol{B}$ can be obtained as described in Section 1.1 in strongly polynomial time and with the encoding lengths of its entries polynomial in that of $\boldsymbol{A}$, by Gaussian elimination, cf. [43].

Then, since $\boldsymbol{B}$ has full column-rank (and hence the trivial nullspace $\{\boldsymbol{0}\}$), it holds that

$$\begin{aligned}
\text{Spark}(\boldsymbol{A}, k){=}\text{Yes} \quad &\Leftrightarrow \quad \exists \boldsymbol{x} \neq \boldsymbol{0} \,:\, \boldsymbol{Ax} = \boldsymbol{0},\ \|\boldsymbol{x}\|_0 \leq k \\
&\Leftrightarrow \quad \exists \boldsymbol{q} \neq \boldsymbol{0} \,:\, \boldsymbol{x} = \boldsymbol{Bq},\ \|\boldsymbol{x}\|_0 \leq k \\
&\Leftrightarrow \quad \exists \boldsymbol{q} \neq \boldsymbol{0} \,:\, \|\boldsymbol{Bq}\|_0 \leq k \\
&\Leftrightarrow \quad \text{MinULR}_0^=(\boldsymbol{B}, k){=}\text{Yes}.
\end{aligned}$$

Thus, an instance for the Spark problem can be transformed in polynomial time and space into one for the $\text{MinULR}_0^=$ problem (and vice versa, since $\boldsymbol{A}$ can be obtained from $\boldsymbol{B}$ analogously). □

Unfortunately, the claims from [5] are based on a flawed argument: Their reduction starts from the well-known strongly NP-complete problem "exact cover by 3-sets" (X3C, cf. problem [SP2] in [38]: Given a groundset $\mathcal{S}$ of $|\mathcal{S}| = 3q$ elements ($q \in \mathbb{N}$) and a collection $\mathcal{C}$ of m 3-element subsets of $\mathcal{S}$, decide whether there is an exact cover of $\mathcal{S}$ by sets from $\mathcal{C}$, i.e., a subcollection $\{C_1, \ldots, C_q\} \subseteq \mathcal{C}$ such that $\bigcup_{i=1}^q C_i = S$ and each element $s \in \mathcal{S}$ appears in exactly one set C_i, $1 \leq i \leq q$). Briefly, while the reduction in the proof of [5, Theorem 1] is correct up to an intermediate inhomogeneous MaxFS (or equivalent MinULR) problem, the corresponding nonzero right-hand-side vector is then simply inserted into the constructed matrix and it is claimed that any $\text{MinULR}_0^=$solution to the resulting homogeneous system must use this column. However, this is not true: The matrix for $\text{MinULR}_0^=$ constructed in [5] is

$$\boldsymbol{B} = \begin{pmatrix} \boldsymbol{A}_{\text{sc}} & -\mathbb{1} \\ \boldsymbol{I} & -\mathbb{1} \\ \boldsymbol{I} & \boldsymbol{0} \end{pmatrix},$$

where $\boldsymbol{A}_{\text{sc}} \in \{0,1\}^{3q\times m}$ encodes the X3C instance by setting $(\boldsymbol{A}_{\text{sc}})_{ij} = 1$ if and only if element s_i is contained in set C_j. It is easily verified that any vector $\boldsymbol{q}$ that selects just one column associated with a 3-set C_j (e.g., $\boldsymbol{q} = (1, 0, \ldots, 0)^\top$) leads to $\|\boldsymbol{Bq}\|_0 = 5$, regardless of the problem dimensions and whether the original X3C instance has a positive answer or not.

In [6], a follow-up paper to [5], such undesired "small but useless" solutions are excluded by an additional *explicit assumption* (see [6, Section 3]), but not *implicitly by construction*. Hence, the results from both [5] and [6] actually do *not* show unconditional NP-hardness for $\text{MinULR}_0^=$ and $\text{MaxFS}_0^=$, respectively.

In the following, we provide a valid NP-completeness proof for $\text{MinULR}_0^=$ and $\text{MaxFS}_0^=$ *without* any extra assumptions on the solution, based on the equivalence to the Spark decision problem.

Theorem 3. $\text{MinULR}_0^=$ *and* $\text{MaxFS}_0^=$ *are NP-complete.*

Proof. Since $\text{MaxFS}_0^=(\boldsymbol{B}, q-k) = \text{Yes} \Leftrightarrow \text{MinULR}_0^=(\boldsymbol{B}, k) = \text{Yes}$, it suffices to consider $\text{MinULR}_0^=$.

Containment in NP is clear: A certificate is a vector $\boldsymbol{q}$ such that $\|\boldsymbol{Bq}\|_0 \leq k$; since $\boldsymbol{q}$ is contained in the nullspace of the submatrix of $\boldsymbol{B}$ restricted to the (unknown) subset of rows corresponding to the zero components of $\boldsymbol{Bq}$, we may assume that $\boldsymbol{q}$ is rational and has encoding length polynomially bounded by that of the input. (This can be deduced, e.g., from properties of the Gaussian elimination method, cf. [43].) Then, obviously, one can compute $\boldsymbol{Bq}$ and verify $\|\boldsymbol{Bq}\|_0 \leq k$ in (input-)polynomial time.

To show hardness, we reduce from the Spark problem: By Lemma 2, an instance $(\boldsymbol{B}, k)$ to $\text{MinULR}_0^=$ can be constructed from a Spark instance $(\boldsymbol{A}, k)$ in poylnomial time and space such that $\text{Spark}(\boldsymbol{A}, k)=\text{Yes}$ if and only if $\text{MinULR}_0^=(\boldsymbol{B}, k)=\text{Yes}$. This equivalence completes the proof. □

As the additional assumption w.r.t. undesired solutions is not mentioned in the actual theorems of [5, 6], it can apparently be easily overlooked: The incorrectly implied strong NP-hardness of $\text{MinULR}_0^=$ was used, for instance, in [77, Theorem 1] to derive strong NP-hardness results about projection onto the set of cosparse vectors $\{\boldsymbol{z} : \|\boldsymbol{Bz}\|_0 \leq k\}$. Similarly, intractability of $\text{MaxFS}_0^=$ was exploited in [21, Proposition 4] to prove NP-hardness of deciding whether an $m \times n$ (integer) matrix contains a singular $m \times m$ submatrix. In light of the preceding discussion, such intractability results are "saved" by Theorem 3 above, although the "strong" part of the NP-hardness assertions is unfortunately lost.

Furthermore, note that the optimization version of $\text{MinULR}_0^=$ leads to the the notion of *cospark* (cf. [18]):

$$\text{cospark}(\boldsymbol{B}) \coloneqq \min\{\|\boldsymbol{Bq}\|_0 \,:\, \boldsymbol{q} \neq \boldsymbol{0}\} = \min\{k \,:\, \text{MinULR}_0^=(\boldsymbol{B}, k) = \text{Yes}\}. \tag{3}$$

Thus, when $\boldsymbol{B}$ spans $\mathcal{N}(\boldsymbol{A})$ as in the proof of Lemma 2, it holds that $\text{spark}(\boldsymbol{A}) = \text{cospark}(\boldsymbol{B})$. In terms of matroid theory, the cospark is precisely the *cogirth* of the vector matroid associated with $\boldsymbol{A}$, i.e., the girth of the corresponding dual matroid, which coincides with the vector matroid defined over the columns of $\boldsymbol{B}^\top$. Thus, the above results and the relationships between these notions immediately yield the following result, which, to the best of our knowledge, has not been stated (explicitly) in the literature before:

Corollary 4. *Computing the cogirth of a matroid is NP-hard, even for the special case of vector matroids (cospark computation).*

3.2 Polynomially Solvable Special Cases

Despite the general NP-hardness, the spark can be computed efficiently for certain matrix classes. For instance, if $\boldsymbol{A} \in \{0, \pm 1\}^{m \times n}$ is the oriented (vertex-edge-) incidence matrix of a graph G, then the corresponding vector matroid is a representation of the graphical matroid associated with G. (In case G is undirected, the matrix representation is obtained by orienting the edges arbitrarily.) For such $\boldsymbol{A}$, $\text{spark}(\boldsymbol{A})$ is simply the length of the shortest cycle in G, which can be computed in polynomial time, see, e.g., [46].

In fact, we can obtain the following more general result. Recall that $\boldsymbol{A}$ is called *unimodular* if the determinant of every $m \times m$ submatrix is either -1, 0 or 1.

Theorem 5. *If $\boldsymbol{A}$ is unimodular, then* $\text{spark}(\boldsymbol{A})$ *can be determined in polynomial time.*

For the proof, we employ the following auxiliary result, which (as well as its Corollary) may be of interest in its own right.

Proposition 6. *The sparse recovery problem* $\min\{\|\boldsymbol{x}\|_0 \,:\, \boldsymbol{Ax} = \boldsymbol{b}\}$ *can be solved in polynomial time if $\boldsymbol{b} \in \{0, \pm c\}^m$ for some $c \in \mathbb{Q}$ and the matrix $\boldsymbol{A}$ is unimodular.*

Proof. Assume w.l.o.g. that $c = 1$ (recall that the ℓ_0-norm is invariant under scaling). Moreover, clearly, if $\boldsymbol{A}$ is unimodular, then so is $(\boldsymbol{A}, -\boldsymbol{A})$. Thus, with $\boldsymbol{b} \in \{0, \pm 1\}^m \subset \mathbb{Z}^m$, it is well-known that the linear program (LP)

$$\min \mathbb{1}^\top \boldsymbol{x}^+ + \mathbb{1}^\top \boldsymbol{x}^- \quad \text{s.t.} \quad \boldsymbol{Ax}^+ - \boldsymbol{Ax}^- = \boldsymbol{b};\ \boldsymbol{x}^+, \boldsymbol{x}^- \geq 0 \tag{4}$$

has an integral optimal basic solution. In fact, the nonzeros in any basic solution of (4) are all either -1 or 1, as a simple consequence of Cramer's rule (applied to the subsystem associated with an optimal basis). Consequently, for optimal basic solutions, $\mathbb{1}^\top \boldsymbol{x}^+ + \mathbb{1}^\top \boldsymbol{x}^- = \|\boldsymbol{x}^+\|_0 + \|\boldsymbol{x}^-\|_0$. Moreover, for variable-split LPs such as (4), optimal basic variables $\hat{\boldsymbol{x}}^+$ and $\hat{\boldsymbol{x}}^-$ are complementary (i.e., $\hat{\boldsymbol{x}}_j^+ \hat{\boldsymbol{x}}_j^- = 0$ for all $j \in [n]$); thus, $\hat{\boldsymbol{x}} \coloneqq \hat{\boldsymbol{x}}^+ - \hat{\boldsymbol{x}}^-$ solves $\boldsymbol{A}\hat{\boldsymbol{x}} = \boldsymbol{b}$ and has $\|\hat{\boldsymbol{x}}\|_0 = \|\hat{\boldsymbol{x}}^+\|_0 + \|\hat{\boldsymbol{x}}^-\|_0$. It remains to recall that LPs can be solved in polynomial time (for rational data, by the ellipsoid algorithm or interior-point methods). □

Corollary 7. *If $\boldsymbol{A}$ is unimodular and $\boldsymbol{b} \in \{0, \pm c\}^m$ for some $c \in \mathbb{Q}$, ℓ_0-ℓ_1-equivalence always holds, i.e., solutions to* (2) *can be recovered by minimizing the ℓ_1-norm instead.*

Proof. The LP (4) is a popular reformulation of the basis pursuit problem $\min\{\|\boldsymbol{x}\|_1 : \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}\}$ (cf. [20]). Thus, if $\boldsymbol{A}$ is unimodular and $\boldsymbol{b} \in \{0, \pm 1\}^m$, then $\|\hat{\boldsymbol{x}}\|_1 = \|\boldsymbol{x}\|_0$ for *any* basic solution $\hat{\boldsymbol{x}}$ of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$; in particular, for an optimal basic solution $\hat{\boldsymbol{x}} = \hat{\boldsymbol{x}}^+ - \hat{\boldsymbol{x}}^-$ associated with an optimal basic solution $(\hat{\boldsymbol{x}}^+, \hat{\boldsymbol{x}}^-)$ of (4), $|\mathrm{supp}(\hat{\boldsymbol{x}})|$ coincides with the optimal value of the corresponding instance of (2). Since rescaling a vector leaves its support unchanged, the result carries over to the case $|c| \neq 1$ as well. □

Proof of Theorem 5. As mentioned in Section 2 (see also Section 5.1), $\mathrm{spark}(\boldsymbol{A})$ can be obtained by solving the n instances of (2) with matrices $\boldsymbol{A}^{(j)} \coloneqq \left(\boldsymbol{A}^\top, \boldsymbol{u}_j\right)^\top$, where $\boldsymbol{u}_j$ is the j-th unit vector, and right hand sides $\boldsymbol{u}_{m+1}$ for $j = 1, 2, \ldots, n$. If $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is unimodular, then so is $\boldsymbol{A}^{(j)}$, for any $j \in [n]$, as can easily be seen by developing the determinant of an arbitary $(m+1) \times (m+1)$ submatrix of $\boldsymbol{A}^{(j)}$ w.r.t. the last row. The claim now follows immediately from Proposition 6. □

Remark 8. *Since oriented incidence matrices of graphs are well-known to be (totally) unimodular, Theorem 5 yields the aforementioned polynomial-time spark computability for graphical matroids as a special case. The binary incidence matrices of undirected graphs G (i.e., where one does not impose an arbitrary orientation on each edge), are totally unimodular if and only if G is bipartite; thus, the spark can be computed in poly-time for such matrices as well. (Indeed, this is not news either: A bipartite graph contains no odd cycles. Therefore, circuit-defining nullspace vectors of its binary incidence matrix correspond only to even cycles in the graph, the shortest of which can also be found in poly-time by a combinatorial algorithm [68].) The vector matroids associated with (general) totally unimodular matrices are called* regular matroids*, cf. [70]; by Theorem 5, their girth can also be computed in poly-time.*

For non-bipartite undirected graphs, circuits of their binary incidence matrices correspond to even cycles and odd bowties (i.e., two distinct, simple odd cycles that are connected by a path, which may have length zero) [25, Theorem 6]. Since those matrices are not unimodular, we cannot invoke Theorem 5, and indeed, it seems to be unknown whether the shortest odd bowtie in an undirected non-bipartite graph can be found in poly-time. Thus, it is an open question whether the spark can be efficiently computed for such matrices and, by extension, for general binary matrices. (Note that a proof that this is not possible unless P=NP would likely settle the strong NP-completeness of Spark$(\boldsymbol{A}, k)$ *in the affirmative.)*

Finally, note that a result on ℓ_1-minimization (or LP) recovery of certain integral *sparse signals similar to Proposition 6 was recently shown in [58, Prop. 16].*

Remark 9. *The conditions in Theorem 5, Proposition 6 and Corollary 7 can be checked in polynomial time by testing unimodularity of $\boldsymbol{A}$, see [72, 79, 81]. Note also that ellipsoid or interior point LP methods may not compute a* basic *optimal solution (as simplex methods do), in which case the ℓ_0-ℓ_1-equivalence generally only pertains to the solution value; however, a basic solution can be obtained efficiently by crossover techniques, see, e.g., [16].*

4 Spark Bounds

There are various ways one can (efficiently) compute lower and upper bounds for the spark of a matrix. We gather several bounds in the following Proposition.

Proposition 10. *Let $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, w.l.o.g. with $\mathrm{rank}(\boldsymbol{A}) = m < n$ and $\boldsymbol{A}_j \neq \boldsymbol{0}$ for all $j \in [n]$.*

1. *(Trivial bounds)*
$$2 \leq \mathrm{spark}(\boldsymbol{A}) \leq m + 1$$
2. *(Lower bound via mutual coherence [32, 42])*
$$\mathrm{spark}(\boldsymbol{A}) \geq \left\lceil 1 + \tfrac{1}{\mu(\boldsymbol{A})} \right\rceil, \quad \text{where } \mu(\boldsymbol{A}) \coloneqq \max_{1 \leq i < j \leq n} \frac{\boldsymbol{A}_i^\top \boldsymbol{A}_j}{\|\boldsymbol{A}_i\|_2 \cdot \|\boldsymbol{A}_j\|_2}$$

3. *(Lower bound via coherence index [39])*

$$\operatorname{spark}(\boldsymbol{A}) \geq 1 + i_\mu(\boldsymbol{A}), \quad \text{where } i_\mu(\boldsymbol{A}) \coloneqq \min_{1\leq p\leq n} \Big\{ p \,:\, \sum_{i=1}^{p} \hat{\mu}_i \geq 1 \Big\},$$

where $\hat{\mu}_\ell$, $\ell = 1, \ldots, n(n-1)/2$, *is the sequence of column-pair coherence values* $\frac{\boldsymbol{A}_i^\top \boldsymbol{A}_j}{\|\boldsymbol{A}_i\|_2 \cdot \|\boldsymbol{A}_j\|_2}$ *sorted in non-increasing order, i.e.,* $\hat{\mu}_\ell \geq \hat{\mu}_k$ *for all* $\ell < k$.

4. *(Upper bound via* ℓ_1*-minimization)*

$$\text{Let } (j^*, \boldsymbol{x}^*) \text{ solve } \min_{j\in[n]} \big\{ \min\{\|\boldsymbol{x}\|_1 \,:\, \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\ x_j = 1\}\big\}. \text{ Then, } \operatorname{spark}(\boldsymbol{A}) \leq \|\boldsymbol{x}^*\|_0.$$

5. *(Probabilistic upper bounds)*
Let $\boldsymbol{z} \in \mathbb{R}^n$ *with i.i.d. Gaussian random entries (or a suitable other random vector). Then,*

$$\begin{aligned} \operatorname{spark}(\boldsymbol{A}) &\leq \min\{\|\boldsymbol{x}\|_0 \,:\, \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\ \boldsymbol{z}^\top \boldsymbol{x} = 1\} \\ &\leq \min\big\{\|\hat{\boldsymbol{x}}\|_0 : \hat{\boldsymbol{x}} \in \operatorname{Arg\,min}\{\|\boldsymbol{x}\|_1 \,:\, \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\ \boldsymbol{z}^\top \boldsymbol{x} = 1\}\big\}. \end{aligned}$$

With probability 1, *the first inequality holds with equality.*

6. *(Upper and lower bounds via column and row restriction, resp.)*

$$\operatorname{spark}(\boldsymbol{A}_{R\cdot}) \leq \operatorname{spark}(\boldsymbol{A}) \leq \operatorname{spark}(\boldsymbol{A}_{\cdot C}) \quad \forall\, R \subseteq [m],\ C \subseteq [n]$$

7. *(Lower bound via distinct cobases)*

$$\operatorname{spark}(\boldsymbol{A}) \geq \text{ number of mutually disjoint regular } (n-m)\times(n-m) \text{ submatrices of } \boldsymbol{B},$$

where $\boldsymbol{B} \in \mathbb{R}^{n\times(n-m)}$ *spans* $\mathcal{N}(\boldsymbol{A})$.

8. *(Lower bounds via restricted isometry constant)*

$$\operatorname{spark}(\boldsymbol{A}) \geq k+1 \quad \text{if and only if} \quad \delta_k(\boldsymbol{A}) < 1,$$

where $\delta_k(\boldsymbol{A})$ *is the* k*-th order restricted isometry constant of* $\boldsymbol{A}$, *i.e., the smallest* $\delta \geq 0$ *such that* $(1-\delta)\|\boldsymbol{x}\|_2^2 \leq \|\boldsymbol{A}\boldsymbol{x}\|_2^2 \leq (1+\delta)\|\boldsymbol{x}\|_2^2$ *for all* k*-sparse* $\boldsymbol{x}$. *Moreover, for* $\boldsymbol{A}$ *with unit-*ℓ_2*-norm columns and* $\delta_k(\boldsymbol{A}) < 1$, *it holds that*

$$\operatorname{spark}(\boldsymbol{A}) \geq \lceil (k-1)/\delta_k(\boldsymbol{A}) + 1 \rceil.$$

9. *(Lower bounds via nullspace constant)*

$$\operatorname{spark}(\boldsymbol{A}) \geq k+1 \quad \text{if and only if} \quad \alpha_k(\boldsymbol{A}) < 1,$$

where $\alpha_k(\boldsymbol{A}) \coloneqq \max\{\|\boldsymbol{x}_S\|_1 \,:\, \|\boldsymbol{x}\|_1 = 1,\ \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\ S \subseteq [n],\ |S| \leq k\}$ *is the* k*-th order nullspace constant of* $\boldsymbol{A}$.

Proof. The first three statements are well-known and the fourth is immediate from scalability of nullspace vectors and the basis pursuit (ℓ_1-norm minimization) approach to ℓ_0-norm minimization problems (see also Section 5.1 below for the corresponding sequence of ℓ_0-problems to exactly compute the spark that are here replaced by basis pursuit subproblems). For item 5, note that a suitable random vector $\boldsymbol{z}$ (e.g., with entries drawn i.i.d. from the standard normal distribution $\mathcal{N}(0,1)$) is almost surely not contained in any specific linear subspace; thus, with probability 1, $\boldsymbol{z}$ will not be orthogonal to every vector $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})\backslash\{\boldsymbol{0}\}$ with $\|\boldsymbol{x}\|_0 \leq \operatorname{spark}(\boldsymbol{A})$. This shows the claimed equality under high probability (in the (measure-zero) event that $\boldsymbol{z}^\top \boldsymbol{x} = 0$ for all spark-defining $\boldsymbol{x}$, the inequality becomes strict); the second inequality is clear since minimizing the ℓ_1-norm instead of the ℓ_0-norm could produce solutions with support size larger than $\operatorname{spark}(\boldsymbol{A})$ (in either event). The bounds in item 6 are immediately clear from observing that $\mathcal{N}(\boldsymbol{A}_{\cdot C}) \subseteq \mathcal{N}(\boldsymbol{A}) \subseteq \mathcal{N}(\boldsymbol{A}_{R\cdot})$ for all $R \subseteq [m]$ and all $C \subseteq [n]$. The bound in item 7 follows from matroid arguments, cf. Lemma 13 later. The first

bound in item 8 follows directly from the NP-hardness proof in [75, 78] for computing restricted isometry constants (the key point being that $\delta_k(\boldsymbol{A}) < 1$ implies, in particular, that the smallest singular value of all k-column submatrices of $\boldsymbol{A}$ is positive, signifying that all such column subsets are linearly independent), and the second one combines the sufficient condition $\delta_\ell(\boldsymbol{A}) < 1$ for $\operatorname{spark}(\boldsymbol{A}) \geq \ell + 1$ to hold with the fact that for $\ell \geq k$ (and $\boldsymbol{A}$ with normalized columns), $\delta_\ell(\boldsymbol{A}) \leq \delta_k(\boldsymbol{A})(\ell - 1)/(k - 1)$, see [55]. Finally, the bound in item 9 is a direct consequence of the NP-hardness proof of [78, Theorem 5] for computing the nullspace constant (briefly, if for all $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})$ with $\|\boldsymbol{x}\|_1 = 1$, $\|\boldsymbol{x}_S\|_1 < 1$ for all k-element subsets $S \subseteq [n]$, then every nullspace vector must have at least $k + 1$ nonzero elements, and vice versa). □

Naturally, the support size of any nonzero vector in the nullspace of $\boldsymbol{A}$ provides an upper bound on $\operatorname{spark}(\boldsymbol{A})$ as well. Further lower bounds can be obtained, for instance, from the LP relaxations of the (mixed-)integer programs we put forth in the next sections.

Also, note that although the restricted isometry constant (RIC) is (strongly) NP-hard to compute in general, the second bound in item 8 above may still be of use if the exact value of $\delta_k(\boldsymbol{A})$ is known for a small (constant) k; indeed, the bound generalizes that from item 2, which is recovered for $k = 2$, since $\delta_2(\boldsymbol{A}) = \mu(\boldsymbol{A})$ (if $\|\boldsymbol{A}_j\|_2 = 1 \; \forall j \in [n]$). Moreover, in the first bound in item 8, one could replace the standard RIC by the lower asymmetric RIC, i.e., dropping the right-hand inequality in the stated definition of $\delta_k(\boldsymbol{A})$, as the connection to column linear independence hinges solely on the left-hand part (but note that the second bound is only proven for the standard, symmetric RIC). Given that both RIC and nullspace constant are NP-hard to compute themselves (cf. [78, 75]), the associated spark bounds from Proposition 10 are likely not very convenient for practical purposes. Indeed, besides obviously prohibitive brute-force calculations, there currently appears to be only one more sophisticated exact algorithm each, namely a mixed-integer semidefinite programming (SDP) method for the RIC [37] and a branch & bound tree-search algorithm for the nullspace constant [23], both of which can only handle small instances (and orders k) efficiently. One could instead resort to available SDP or LP relaxations to compute useful bounds for δ_k or α_k, respectively (see [50, 27, 28, 26]), but these also take considerable running time and are thus inadequate for the purpose of *quick*, possibly repeated, application within a framework to compute the spark.

Nevertheless, we shall employ several of the other lower and upper bounds listed in the above proposition in our branch & cut algorithm, see Section 7.

5 Mixed-Integer Programming Formulations

In this section, we introduce several different approaches to computing the spark, all of which can be formulated as mixed-integer programs (MIPs) and may thus be solved with an arbitrary general-purpose black-box MIP solver. We also remark further on some related models from the literature (for cospark rather than spark computation) that were already mentioned in Section 2 and discuss some practical aspects; computational comparisons are deferred to the experimental evaluation in Section 8.

5.1 Computing the Spark via ℓ_0-Minimization Subproblems

The first approach employs a sequence of ℓ_0-minimization problems (cf. [32] or [78, Remark 3]):

$$\begin{aligned} \operatorname{spark}(\boldsymbol{A}) &= 1 + \min\big\{\, \min\{\|\boldsymbol{x}\|_0 \,:\, \boldsymbol{A}_{\cdot[n]\setminus j}\boldsymbol{x} = \boldsymbol{A}_j\} \,:\, j \in [n]\big\} \\ &= \min\big\{\, \underbrace{\min\{\|\boldsymbol{x}\|_0 \,:\, \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\; x_j = 1\}}_{=:(\mathrm{P}_0^j)} \,:\, j \in [n]\big\} \end{aligned} \tag{5}$$

Here, we fix one variable at a time to 1 (w.l.o.g. – by nullspace vector scalability, any other nonzero value would do as well), thereby excluding the trivial all-zero solution; thus, the constraint $\boldsymbol{x} \neq \boldsymbol{0}$ is circumvented by reverting to these n subproblems. In matroid terms, this approach is equivalent to finding the smallest circuit through each element in turn, in order to then identify the overall minimum achievable cardinality.

For the above scheme, we need an exact method for problems of the form (2). To the best of our knowledge, there are only two such exact approaches at the time of writing (not counting simple brute force): The straightforward, "folklore" Big-M MIP

$$\min\big\{\mathbb{1}^\top \boldsymbol{y} \,:\, \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},\;\; -M\boldsymbol{y} \leq \boldsymbol{x} \leq M\boldsymbol{y},\; \boldsymbol{y} \in \{0,1\}^n\big\}, \tag{6}$$

where M is a sufficiently large positive constant, and the specialized branch & cut algorithm from [49]. The numerical experiments in [49] for the latter method indicate that it can only handle relatively small instances efficiently. On the other hand, a valid value for M in (6) that provably does not cut off all optimal solutions of (2) is, in general, not known a priori. Here, however, the scalability of nullspace vectors allows us to enforce the bounds $-\mathbb{1} \leq \boldsymbol{x} \leq \mathbb{1}$ for our subproblems (i.e., we may use $M = 1$ w.l.o.g.); thus, we focus on the second approach here:

Proposition 11. *It holds that*

$$\begin{aligned}\operatorname{spark}(\boldsymbol{A}) &= \min\big\{\min\{\|\boldsymbol{x}\|_0 \,:\, \boldsymbol{A}\boldsymbol{x}=\boldsymbol{0},\ x_j=1,\ -\mathbb{1}\leq\boldsymbol{x}\leq\mathbb{1}\} \,:\, j\in[n]\big\}\\ &= \min\big\{\underbrace{\min\big\{\mathbb{1}^\top\boldsymbol{y} \,:\, \boldsymbol{A}\boldsymbol{x}=\boldsymbol{0},\ x_j=1,\ y_j=1,\ -\boldsymbol{y}\leq\boldsymbol{x}\leq\boldsymbol{y},\ \boldsymbol{y}\in\{0,1\}^n\big\}}_{=:(\mathrm{P}_0^j)'} \,:\, j\in[n]\big\}. \end{aligned} \tag{7}$$

Proof. Observe that for any $j \in [n]$, the solution value of $(\mathrm{P}_0^j)'$ may be worse than that of (P_0^j), but for $k \in \operatorname{Arg\,max}\{|x_\ell^*| \,:\, \ell \in \operatorname{supp}(\boldsymbol{x}^*)\}$ for a spark-defining vector $\boldsymbol{x}^*$, setting $x_k = y_k = 1$ keeps the rescaled vector $-\mathbb{1} \leq (\operatorname{sign}(x_k^*)/|x_k^*|)\boldsymbol{x}^* \leq \mathbb{1}$ (along with $y_j^* = 1 \Leftrightarrow x_j^* \neq 0$) as a solution to $(\mathrm{P}_0^k)'$. Thus, for at least one subproblem (P_0^j) whose solution yields $\operatorname{spark}(\boldsymbol{A})$, the fixing actually *implies* the bounds explicitly enforced in $(\mathrm{P}_0^j)'$, whence at least this optimal solution is not lost. □

Naturally, along the subproblem sequence, we may also restrict to finding strictly better solutions than the current best known one, thereby typically rendering several subproblems infeasible (which may be detectable quicker than it takes to fully solve the resp. subproblem). Moreover, to simplify the subproblems, we can carefully remove certain columns (i.e., eliminate the corresponding variables): In case of solving (P_0^j)-subproblems directly, for every $k \in \operatorname{supp}(\hat{\boldsymbol{x}}^j)$, where $\hat{\boldsymbol{x}}^j$ is the solution to the (perhaps already modified) j-th subproblem, we can eliminate x_j from the respective k-th subproblem. To see this, note that (P_0^k) either yields the same solution support $\operatorname{supp}(\hat{\boldsymbol{x}}^j)$ or a different (in particular, smaller, if the search was restricted to strictly better than previous solutions) one that does not contain j. Analogously, when tackling the MIP sequence (7), x_j can be eliminated from the subproblems for all $k \in \operatorname{supp}(\hat{\boldsymbol{x}}^j) \cap \{i \,:\, |\hat{x}_i^j| = 1\}$. As this argument holds from each subproblem to the next, every remaining index has associated with it a list of variables that may be eliminated from the respective subproblem once its turn to be solved has come.

In fact, along with similar variable removals that do not even require previous subproblem solutions, the number of subproblems can itself be reduced:

Proposition 12. *Let* $\underline{s} \leq \operatorname{spark}(\boldsymbol{A})$*, let* $\pi = (\pi_1, \ldots, \pi_n)$ *be any permutation of* $[n]$ *and let* $\sigma \leq \max\{i \in [n] : \operatorname{rank}(\boldsymbol{A}_{\cdot\{\pi_{n-i+1},\ldots,\pi_n\}}) = i\}$*. Then,*

$$\operatorname{spark}(\boldsymbol{A}) = \min\big\{\min\{\|\boldsymbol{x}\|_0 \,:\, \boldsymbol{A}\boldsymbol{x}=\boldsymbol{0},\ x_{\pi_k}=1,\ x_{\pi_j}=0\ \forall\, j<k\} \,:\, k\in[n-\max\{\underline{s},\sigma+1\}+1]\big\}. \tag{8}$$

Proof. W.l.o.g., let $\pi = (1, 2, \ldots, n)$. Since $\operatorname{spark}(\boldsymbol{A}) \geq \underline{s}$, subproblems for $k > n - \underline{s} + 1$ contain fewer than $\underline{s}$ variables, and similarly, $\boldsymbol{A}_{\cdot\{n-\sigma+1,\ldots,n\}}$ has full column-rank σ; therefore, these subproblems are infeasible and need not be considered at all. Moreover, we can fix all $x_j = 0$ for $j < k$ in the k-th subproblem because, for any spark-defining vector $\boldsymbol{x}^*$, the subproblem associated with the first (smallest) index k in $\operatorname{supp}(\boldsymbol{x}^*)$ retains $\boldsymbol{x}^*$ as its solution. □

Note, however, that Proposition (12) is only applicable when working with direct solvers of ℓ_0-minimization problems (2) or when using the general Big-M formulation (6) (instead of $(\mathrm{P}_0^j)'$) in (7), as the choice $M = 1$ is easily seen to be generally *not* valid in conjunction with the 0-fixings in each subproblem of (8).

Nevertheless, Proposition 12 forms the basis of a branching rule devised for our specialized branch & cut solver, which employs a binary IP model without explicit $\boldsymbol{x}$-variables and therefore avoids the associated scaling (or Big-M) issues, see Sections 6 and 7.3.

5.2 Direct MIP Formulation

With the following model, we can avoid having to solve a sequence of (2) problems:

$$\min\big\{\mathbb{1}^\top\boldsymbol{y} \,:\, \boldsymbol{A}\boldsymbol{x}=\boldsymbol{0},\ -\boldsymbol{y}+2\boldsymbol{z}\leq\boldsymbol{x}\leq\boldsymbol{y},\ \mathbb{1}^\top\boldsymbol{z}=1,\ \boldsymbol{x}\in\mathbb{R}^n,\ \boldsymbol{y}\in\{0,1\}^n,\ \boldsymbol{z}\in\{0,1\}^n\big\} \tag{9}$$

Here, the choice which variable is set to 1 does not have to be made a priori but is left to the solver; it is encoded in the binary variable vector $\boldsymbol{z}$. Like in the $(\mathrm{P}_0^j)'$ problems, the model also enforces nullspace vector scaling to within the bounds $\pm\mathbb{1}$.

5.3 Matroid-Dual MIP Formulations

The spark of $\boldsymbol{A}$ can be computed as cospark($\boldsymbol{B}$), where $\boldsymbol{B}$ is any matrix whose columns span the nullspace of $\boldsymbol{A}$; we can compute one such $\boldsymbol{B}$ (actually, the standard representation matrix of $\mathcal{M}^*[\boldsymbol{A}] = \mathcal{M}[\boldsymbol{B}^\top]$) as described in the proof of Lemma 2. Once $\boldsymbol{B}$ has been obtained, we can use a MIP model similar to (9) that was put forth in [53] (see also [13]) for the computation of the cospark: Let $\tilde{\boldsymbol{B}}$ denote $\boldsymbol{B}$ after normalizing the rows to unit ℓ_1-norm; then, spark($\boldsymbol{A}$) equals

$$\min\big\{\mathbb{1}^\top\boldsymbol{y} \,:\, -\boldsymbol{y} \le \tilde{\boldsymbol{B}}\boldsymbol{q} \le \boldsymbol{y},\ -\mathbb{1}+2\boldsymbol{z} \le \boldsymbol{q} \le \mathbb{1},\ \mathbb{1}^\top\boldsymbol{z}=1,\ \boldsymbol{q}\in\mathbb{R}^{n-m},\ \boldsymbol{y}\in\{0,1\}^n,\ \boldsymbol{z}\in\{0,1\}^{n-m}\big\}. \quad (10)$$

As an alternative, we propose the following MIP that has $2(n-m)$ fewer constraints (but m more binary variables) than (10), and clearly also computes cospark($\boldsymbol{B}$) (compare with (9)):

$$\min\big\{\mathbb{1}^\top\boldsymbol{y} \,:\, -\boldsymbol{y}+2\boldsymbol{z} \le \boldsymbol{B}\boldsymbol{q} \le \boldsymbol{y},\ \mathbb{1}^\top\boldsymbol{z}=1,\ \boldsymbol{q}\in\mathbb{R}^{n-m},\ \boldsymbol{y}\in\{0,1\}^n,\ \boldsymbol{z}\in\{0,1\}^n\big\}. \quad (11)$$

Of course, one could also use the row-rescaled $\tilde{\boldsymbol{B}}$ in (11).

Recalling (3), note that these MIPs can also be seen as direct formulations of the optimization problem associated with $\textsc{MinULR}_0^=$ (for given $\boldsymbol{B}$).

Compared to the direct "primal" formulation (9), the "dual" MIP (10) from [53] has fewer variables but more constraints, whereas our modification (11) in fact has both fewer variables and constraints than (9). However, as pointed out before, the required computation of $\boldsymbol{B}$ can introduce or increase undesired numerical instability to the matroid-dual approaches, as well as some runtime overhead. Preliminary computational experiments indicated that (9) was significantly faster than either of the above MIPs to compute the spark, even disregarding the overhead induced by computing $\boldsymbol{B}$ and on instances where numerical problems did not appear to arise and the difference in model sizes leans in favor of (at least) (11). Therefore, we do not include the cospark-based models (10) and (11) in the numerical experiments reported on in Section 8 later. Further computational studies comparing (10) with (11), or whether the "primal" spark models offer preferable alternatives to cospark methods even if one is interested in the cospark to begin with, go beyond the scope of this paper and are thus left for future consideration.

5.4 Replacing the Nontriviality Constraint by Fixing the ℓ_1-Norm

Clearly, $\boldsymbol{x} \neq \boldsymbol{0}$ if and only if $\|\boldsymbol{x}\| \neq 0$ for any norm. Indeed, the model (9) can be seen to enforce $\|\boldsymbol{x}\|_\infty = 1$. Another MIP can be obtained by replacing $\boldsymbol{x} \neq \boldsymbol{0}$ in the spark problem by the constraint $\|\boldsymbol{x}\|_1 = 1$. In order to obtain a linear formulation, we need to split the variable vector $\boldsymbol{x}$ into its positve and negative parts ($\boldsymbol{x} = \boldsymbol{x}^+ - \boldsymbol{x}^-$ with $\boldsymbol{x}^+ = \max\{\boldsymbol{0}, \boldsymbol{x}\}$ and $\boldsymbol{x}^- = \max\{\boldsymbol{0}, -\boldsymbol{x}\}$); then, $\|\boldsymbol{x}\|_1 = \mathbb{1}^\top\boldsymbol{x}^+ + \mathbb{1}^\top\boldsymbol{x}^-$. Accordingly, we use binary variable vectors $\boldsymbol{y}^+$ and $\boldsymbol{y}^-$ to model the supports of $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$, respectively; complementarity of $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ is enforced by requiring $\boldsymbol{y}^+ + \boldsymbol{y}^- \le \mathbb{1}$. Moreover, by fixing the ℓ_1-norm to 1, $\boldsymbol{0} \le \boldsymbol{x}^\pm \le \mathbb{1}$ holds automatically, so we can use the constraints $0 \le \boldsymbol{x}^\pm \le \boldsymbol{y}^\pm$. In total, we obtain the following MIP to compute spark($\boldsymbol{A}$):

$$\begin{aligned} \min\quad & \mathbb{1}^\top\boldsymbol{y}^+ + \mathbb{1}^\top\boldsymbol{y}^- && (12)\\ \text{s.t.}\quad & \boldsymbol{A}\boldsymbol{x}^+ - \boldsymbol{A}\boldsymbol{x}^- = 0, \quad \mathbb{1}^\top\boldsymbol{x}^+ + \mathbb{1}^\top\boldsymbol{x}^- = 1,\\ & \boldsymbol{y}^+ + \boldsymbol{y}^- \le \mathbb{1}, \quad \boldsymbol{0} \le \boldsymbol{x}^+ \le \boldsymbol{y}^+, \quad \boldsymbol{0} \le \boldsymbol{x}^- \le \boldsymbol{y}^-\\ & \boldsymbol{y}^+ \in \{0,1\}^n, \quad \boldsymbol{y}^- \in \{0,1\}^n. \end{aligned}$$

Besides having more variables and constraints than all previous models, a noteworthy drawback of the above model is that it contains symmetry: If $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})$, then also $-\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})$, so for every feasible point $(\boldsymbol{x}^+, \boldsymbol{x}^-, \boldsymbol{y}^+, \boldsymbol{y}^-)$, there is an equivalent feasible point with the entries of $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ and those of $\boldsymbol{y}^+$ and $\boldsymbol{y}^-$ simply interchanged. Much of this symmetry can be eliminated by enforcing $\mathbb{1}^\top\boldsymbol{y}^+ \ge \mathbb{1}^\top\boldsymbol{y}^-$, which removes symmetry w.r.t. vectors $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})$ that have different absolute sums of positive and negative entries (but cannot do the same for "balanced" solutions).

Preliminary experiments showed that, even with the latter symmetry-breaking constraints included, solving (12) takes significantly more time to solve than all other considered approaches, even for very easy instances. Therefore, we will not consider this formulation any further. (As mixed-integer nonlinear problems are typically even harder to solve in practice than linear MIPs, neither do we consider nonlinear variants of the above model, as could be obtained by using, e.g., the ℓ_2-norm for the nontriviality constraint.)

6 Matroid Girth Formulation as a Hitting Set Problem

The spark computation problem can also be recast as a hitting set problem. To that end, it is useful to again consider the matroid interpretation, i.e., that computing $\operatorname{spark}(\boldsymbol{A})$ is equivalent to finding the girth of the vector matroid $\mathcal{M}[\boldsymbol{A}]$ associated with the columns of $\boldsymbol{A}$.

We can characterize the dependent sets of a matroid as follows.

Lemma 13. *Let $\mathcal{M}$ be a matroid over E. Then, $C \subseteq E$ is a dependent set of $\mathcal{M}$ if and only if $C \cap \overline{B} \neq \emptyset$ for all bases B of $\mathcal{M}$.*

Proof. Corollary 2.1.20 in [70] asserts that the circuits of $\mathcal{M}$ are precisely the *inclusion-wise minimal* subsets $C \subseteq E$ with $C \cap B^* \neq \emptyset$ for all cobases $B^* \in \mathcal{B}^*$. Since every dependent set contains a circuit and $\overline{B} \in \mathcal{B}^*$ if and only if $B \in \mathcal{B}$, the claim thus follows immediately. □

Note that Lemma 13 implies Proposition 10, item 7: Any dependent set C must contain at least one index from the complement of every basis—i.e., every cobasis—so the number of mutually disjoint cobases provides a lower bound on the number of elements in every cycle (and hence, in particular, in every circuit).

Based on Lemma 13, we can compute the girth of a matroid $\mathcal{M}$ over E as

$$\min \left\{ \mathbb{1}^\top \boldsymbol{y} \,:\, \boldsymbol{y}(\overline{B}) = \sum_{j \notin B} y_j \geq 1 \quad \text{for all bases } B \text{ of } \mathcal{M},\ \boldsymbol{y} \in \{0,1\}^{|E|} \right\} \tag{13}$$

and, in particular, we obtain that

$$\operatorname{spark}(\boldsymbol{A}) = \min \left\{ \mathbb{1}^\top \boldsymbol{y} \,:\, \boldsymbol{y}(\overline{B}) \geq 1 \quad \text{for all bases } B \text{ of } \mathcal{M}[\boldsymbol{A}],\ \boldsymbol{y} \in \{0,1\}^n \right\}. \tag{14}$$

Such models are known as *hitting set problems*, since feasible solutions are required to "hit" (intersect) every set from a certain set class.

Since there can be exponentially many bases, one cannot, or may not want to, work with the above models in their entirety. Nevertheless, following the classical branch & cut idea, we can turn to generating hitting set constraints (also called *set covering inequalities*) dynamically in a framework that repeatedly solves relaxed subproblems containing far fewer than the whole set of constraints. For this, we need a *separation* method to identify violated constraints efficiently. Fortunately, it turns out that the separation problem w.r.t. (13) *is* tractable for matroids with polynomial-time independence oracles, i.e., those for which it can be decided in polynomial time whether a given set of elements is independent or not:

Theorem 14. *Given $\hat{\boldsymbol{y}} \in [0,1]^n$, a basis B of $\mathcal{M}$ minimizing $\hat{\boldsymbol{y}}(\overline{B})$ can be found in oracle-polynomial time $\mathcal{O}(n \log(n) + n\mathcal{T}_{\mathcal{M}})$, where $\mathcal{T}_{\mathcal{M}}$ is the runtime of the matroid's independence oracle.*

Proof. There exists a basis $B \in \mathcal{B}$ of $\mathcal{M}$ such that $\hat{\boldsymbol{y}}(\overline{B}) < 1$ if and only if the optimization problem

$$\min \hat{\boldsymbol{y}}(\overline{B}) \quad \text{s.t.} \quad B \in \mathcal{B}$$

has optimal value strictly smaller than one, or equivalently (rewriting the covering inequalities $\boldsymbol{y}(\overline{B}) \geq 1$ as $\boldsymbol{y}(B) \leq \mathbb{1}^\top \boldsymbol{y} - 1$ by subtracting the equation $\boldsymbol{y}(\overline{B}) + \boldsymbol{y}(B) = \mathbb{1}^\top \boldsymbol{y}$), if and only if

$$\gamma \coloneqq \max\{\hat{\boldsymbol{y}}(B) \,:\, B \in \mathcal{B}\} > \mathbb{1}^\top \hat{\boldsymbol{y}} - 1.$$

Since computing γ amounts to maximizing a linear function over the set of bases of a matroid, the greedy algorithm is well-known to be guaranteed to find the optimum (cf. [70]): Traverse the entries of $\hat{\boldsymbol{y}}$ in non-increasing order to build an optimal basis $\hat{B}$, i.e., starting from $\emptyset$, include the next index—corresponding to

an element of the matroid ground set E—if and only if independence of the collected elements is maintained; as soon as $r_{\mathcal{M}}$ independent elements have been gathered thus, a basis was found and the procedure may be stopped. Clearly, sorting $\hat{\boldsymbol{y}}$ can be done in $\mathcal{O}(n\log(n))$ (e.g., using quicksort), and at most n calls to the independence oracle will be needed until an optimal basis is constructed. □

Consequently, the separation problem w.r.t. (14) can be solved in strongly polynomial time for rational representation matrices $\boldsymbol{A}$ of vector matroids:

Corollary 15. *Let $\boldsymbol{A} \in \mathbb{Q}^{m\times n}$. Given $\hat{\boldsymbol{y}} \in [0,1]^n$, a basis B of $\mathcal{M}[\boldsymbol{A}]$ minimizing $\hat{\boldsymbol{y}}(\overline{B})$ can be found in $\mathcal{O}(n(\log(n)+m^3))$ time.*

Proof. For $\mathcal{M}[\boldsymbol{A}]$, one may use Gaussian elimination to determine whether a given collection of columns is linearly independent. Since $r_{\mathcal{M}[\boldsymbol{A}]} = \operatorname{rank}(\boldsymbol{A}) = m < n$, the sets needed to be checked for independence during the greedy algorithm will never contain more than m elements, and Gaussian elimination on rational $m \times m$ matrices can be implemented so as to take only strongly polynomial time $\mathcal{O}(m^3)$, cf. [43]. The result now follows from Theorem 14. □

Moreover, it is important to note that binary points are indeed separated exactly:

Corollary 16. *Given $\hat{\boldsymbol{y}} \in \{0,1\}^n$, a basis B of $\mathcal{M}$ such that $\hat{\boldsymbol{y}}(\overline{B}) < 1$ exists if and only if $\hat{\boldsymbol{y}}$ is the incidence vector of an independent set.*

Proof. By Lemma 13, all incidence vectors of dependent sets satisfy all covering inequalities, so a binary vector can *violate* some such inequality if and only if the associated set (i.e., its support) is *not* dependent. □

Based on Corollaries 15 and 16, we implemented a branch & cut scheme that dynamically generates maximally violated covering inequalities for LP relaxation solutions. The algorithm is described in-depth in the next section.

Remark 17. *An analogous model to compute the* cospark *of a matrix was employed in [8], and can also be derived (and used to obtain* $\operatorname{spark}(\boldsymbol{A}) = \operatorname{cospark}(\boldsymbol{B})$*) as follows: Applying Lemma 13 to the dual matroid $\mathcal{M}^*[\boldsymbol{A}] = \mathcal{M}[\boldsymbol{B}^\top]$ shows that every codependent set intersects every basis (see also [70, Prop. 2.1.19]), so*

$$\operatorname{cospark}(\boldsymbol{B}) = \min\left\{\mathbb{1}^\top\boldsymbol{y} \,:\, \boldsymbol{y}(B) \geq 1 \quad \text{for all bases } B \text{ of } \mathcal{M}[\boldsymbol{B}^\top],\ \boldsymbol{y} \in \{0,1\}^n\right\}. \tag{15}$$

(Naturally, this model can be generalized to the cogirth problem for arbitrary matroids, analogously to (13) *and* (14)*.) However, apart from avoiding potential numerical issues due to computing a dual representation matrix in order to work with the above IP* (15)*, there are several reasons to focus on the direct formulation* (14) *to compute the spark: A branch & cut algorithm to solve these formulations will rely on lots of checks whether certain sets are independent (in particular, bases), in the greedy separation scheme and other algorithmic components. The matrices $\boldsymbol{A}$ in the spark problem, particularly in the context of compressed sensing, often have far fewer rows than columns; then, a basis has size $m \ll n$, whereas a basis for the dual matroid has size $n - m \gg m$, so we can expect those independence tests to be significantly less time-consuming in the direct "primal" formulation. Further incentives (such as propagation rules) to use* (14) *will be discussed in the next section; in fact, we did run some tests with the alternative separation method that finds a cobasis using $\boldsymbol{B}$ (traversing indices in non-decreasing order of $\hat{\boldsymbol{y}}$-values as in [8]), but as expected, this approach—which can be seen as mimicking the branch & cut algorithm from [8] in our framework—deteriorated performance, even when further algorithmic parts requiring knowledge of $\boldsymbol{A}$ (e.g., propagation) remained activated. Consequently, we will not consider the approach to compute* $\operatorname{spark}(\boldsymbol{A})$ *as* $\operatorname{cospark}(\boldsymbol{B})$ *via solving* (15) *any further.*

7 Branch & Cut Algorithm

In the following, we describe the key ingredients to our branch & cut approach to compute the spark, based on the IP formulation (14). Very briefly recalling the general procedure, a branch & cut scheme repeatedly solves (usually, LP) relaxations combined with separating and adding new (general-purpose and/or problem-specific) cutting planes, creating smaller disjoint subproblems by branching, propagating variable fixings

and bounds to reduce subproblem and search tree sizes, and further algorithmic techniques; for a broader background, see, e.g., [67] or textbooks on (mixed-)integer programming and discrete and combinatorial optimization.

We implemented our branch & cut algorithm in `C`, using the open-source MIP-framework SCIP [62] that—unlike proprietary codes—allows access to virtually all parts of the solution process and relatively easy integration of own plugins for branching, heuristics, propagation, and constraint handling (particularly, cut separation), among others. LP relaxations can be solved with the solver SoPlex that comes with SCIP, or with external solvers linked to it. Many general-purpose tools are already included in SCIP; thus, if not combined with own algorithmic ideas, it can also be used as a stand-alone general-purpose solver.

7.1 Preprocessing

Recall that throughout, we assume that $\boldsymbol{A}$ has full row-rank $m < n$ and contains no all-zero columns. These assumptions can be guaranteed by simple preprocessing, as already mentioned in the introduction:

- Traverse the columns to check if they all contain at least one nonzero entry. If some $\boldsymbol{A}_j = \boldsymbol{0}$, terminate with the optimum $\operatorname{spark}(\boldsymbol{A}) = 1$ and an optimal solution for (14) with $y_j = 1$, $y_\ell = 0$ for all $\ell \neq j$.
- Identify redundant rows and eliminate them from the problem. To that end, we can transform (a copy of) the given matrix $\boldsymbol{A}$ into row-echelon form (REF); the number r of "steps" of the REF matrix is the rank of $\boldsymbol{A}$. Thus, either $r = n \leq m$ and the spark problem is infeasible ($\boldsymbol{A}$ has full column-rank and therefore the trivial nullspace $\mathcal{N}(\boldsymbol{A}) = \{\boldsymbol{0}\}$), or $r \leq m$ and $r < n$, so it suffices to work with the REF-induced $r \times n$ submatrix instead of the complete $m \times n$ given matrix.

Further simple preprocessing steps apply to very sparse rows in $\boldsymbol{A}$: If for some $i \in [m]$, $\|\boldsymbol{a}_i^\top\|_0 = 1$, i.e., the i-th row only has a single nonzero entry (say $a_{ij} \neq 0$), then, clearly, $x_j = 0$ for all $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})$. Thus, such columns and rows can be removed from the problem a priori. Similarly, suppose $\|\boldsymbol{a}_i^\top\|_0 = 2$ for some $i \in [m]$, with $a_{ij} \neq 0$ and $a_{ik} \neq 0$. Then, it must hold that $a_{ij}x_j = -a_{ik}x_k$ for any $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{A})$; in particular, $x_j \neq 0$ if and only if $x_k \neq 0$, so for the associated binary variables, we get $y_j = y_k$. Hence, we can set $\boldsymbol{A}_j \leftarrow \boldsymbol{A}_j - (a_{ij}/a_{ik})\boldsymbol{A}_k$ and change the objective function coefficient for y_j from 1 to 2 (since y_j in the new problem now represents two variables from the original formulation) and remove column k and row i from the problem; thus, the original ℓ_0-norm objective of the spark problem (in one of its (M)IP reformulations) becomes a weighted ℓ_0-norm.

Nevertheless, our implementation does not check such cases explicitly in preprocessing, as they are recognized and treated by our propagation (or cut separation) routines anyway, cf. Sections 7.4 and 7.5.4.

7.2 Primal Heuristics

Finding feasible primal solutions (i.e., circuits) heuristically can naturally help the solver by providing upper bound improvements. Oftentimes, good solutions can be found relatively quickly; the hard part is bringing up the lower (dual) bounds to match the primal bound and thereby certify optimality. Our implementation contains two variants of basis pursuit heuristics. In the first one, we solve the standard LP reformulation

$$\min \mathbb{1}^\top \boldsymbol{x}^+ + \mathbb{1}^\top \boldsymbol{x}^- \quad \text{s.t.} \quad \boldsymbol{A}\boldsymbol{x}^+ - \boldsymbol{A}\boldsymbol{x}^- = 0,\ \boldsymbol{z}^\top \boldsymbol{x}^+ - \boldsymbol{z}^\top \boldsymbol{x}^+ = 1,\ \boldsymbol{x}^\pm \geq 0 \tag{16}$$

of the ℓ_1-minimization problem $\min\{\|\boldsymbol{x}\|_1 : \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0},\ \boldsymbol{z}^\top \boldsymbol{x} = 1\}$ with a random vector $\boldsymbol{z}$ (cf. Prop. 10, item 5) and extract a circuit contained in $\operatorname{supp}(\boldsymbol{x}) = \operatorname{supp}(\boldsymbol{x}^+) \cup \operatorname{supp}(\boldsymbol{x}^-)$ as the primal solution. In deeper levels of the search tree (in our implementation, depth at least 10), we call this heuristic on reduced problems in which all columns/variables corresponding to model variables y_j that are fixed to zero in the current node are removed. (We also experimented with another variant that omits the random linear constraint and instead adds $x_j^+ + x_j^- \geq 1$ for all j for which y_j was fixed to one, but this appeared to work less well.)

The second heuristic builds on Prop. 10, item 4: Starting with $J = [n]$, $j = 1 \in J$, solve the ℓ_1-minimization problem $\min\{\|\boldsymbol{x}_J\|_1 : \boldsymbol{A}_{\cdot J}\boldsymbol{x}_J = \boldsymbol{0},\ x_j = 1\}$ (reformulated as an LP akin to (16)), remove the solution support from J and proceed with the next $j \in J$, until $|J| \leq 1$. The smallest subproblem solution support size encountered in this sequences serves as an upper bound on $\operatorname{spark}(\boldsymbol{A})$. Due to solving a sequence of problems here, this heuristic is a bit more expensive than the simpler first one outlined above, so we only

execute it in the root node. In principle, of course, we could also incorporate variables fixings here to obtain a version suitable for use at any level of the branch & bound tree.

Additionally, whenever we encounter a circuit in the process of iteratively constructing a basis during separation of covering inequalities or during independence checks in propagation, we add it to the solution pool (if it improves the upper bound). In the majority of test cases, these three ways to obtain primal solutions led to early discovery of one or more optimal solutions.

Another heuristic that can, in principle, be used for arbitary matroids, is to enumerate fundamental circuits induced by a basis B: For all (or some) $j \notin B$, identify the respective unique circuit in $B \cup \{j\}$ (cf. [70, Cor. 1.2.6]). In the spark case, these circuits can be found as the solution supports (plus $\{j\}$) of the full-rank linear equation systems $\boldsymbol{A}_B \boldsymbol{x} = \boldsymbol{A}_j$. We found this routine to generate primal solutions to be similarly effective as the LP-based heuristics and due to slight runtime advantages enabled only the latter in our implementation.

7.3 Branching Rules

Once separation stops for the root LP relaxation (without having solved the problem already), the first level of the branch & bound tree is created by splitting the original problem into $\mathcal{O}(n)$ subproblems based on Proposition 12[1]: For all $k = 1, 2, \ldots, n - \underline{s} + 1$, where $\underline{s} \geq 2$ is the current best global lower bound for the IP objective (i.e., for spark($\boldsymbol{A}$)), create child nodes comprising the subproblems corresponding to the respective variable fixings

$$y_{\pi_k} \coloneqq 1, \quad y_{\pi_j} \coloneqq 0 \;\; \forall\, 1 \leq j < k,$$

where the permutation π reflects the root LP relaxation solution values $\boldsymbol{y}^*$ such that $y^*_{\pi_1} \geq \cdots \geq y^*_{\pi_n}$. This order takes the tentative "importance" of variables as expressed by the root LP solution into account and appeared superior to the canonical order $(1, 2, \ldots, n)$ in all our experiments.

For the deeper levels of the search tree, we revert to standard variable branching (i.e., creating two child nodes with some y_j fixed to either 0 or 1). In fact, it turned out that elaborate branching schemes (particularly, variants of strong branching) were often counterproductive when applied in the branch & cut method for (14), spending significant runtime on apparently unsuccessfully trying to gauge which branch node is likely to be the most useful to pursue next. Therefore, we switch to the very basic "most infeasible" branching, which simply branches on a variable whose fractional LP relaxation value is farthest from both 0 and 1 (i.e., closest to 1/2).

Remark 18. *For the spark MIP models discussed in Section 5, applying an analogous branching scheme is less convenient, as it is unknown a priori how (to which nonzero value) to fix x_k when fixing $y_k = 1$ so that the rest of the $\boldsymbol{x}$-variables that will be nonzero in an optimal circuit through k could still obey their bounds. Nonetheless, for* (9) *(or similarly for* (11)*), we could create root children with $x_k = y_k = z_k = 1$ and $z_j = 0$ $\forall\, j \neq k$, for $k \in [n]$, and thus at least eliminate all the $\boldsymbol{z}$-variables at the first level of the search tree.*

7.4 Propagation and Pruning Rules

In a given node of the branch & bound tree, the current variable fixings can be exploited for domain propagation and subtree pruning. We implemented a set of rules specific to the spark problem that turned out to be helpful, sometimes crucial, in reducing the search space and speeding up the method in practice. In that regard, a first important observation is that for the formulation (14) based on Lemma 13, the characteristic vectors of *all* dependent sets are feasible; however, we are only intereset in circuits, i.e., those dependent sets that are inclusion-wise minimal. We summarize our propagation routine in Algorithm 1; throughout, let J_0 and J_1 denote the index sets of variables fixed to 0 or 1, respectively, and let $U \coloneqq [n] \setminus (J_0 \cup J_1)$ be the presently unfixed variable indices, in the current node.

The initialization and updates of the lower bounds $\underline{s}$ in Algorithm 1 incorporate several of the bounds given in Proposition 10, combined with local information (i.e., current variable fixings). In particular, evaluating the lower bounds based on coherence (cf. Prop. 10, items 2 and 3) for $\boldsymbol{A}_{J_1 \cup U}$ alone can yield better local bounds than that obtained from the full matrix. Moreover, note that the propagation scheme ensures that J_1 is always kept linearly independent – if a $j \in U$ exists such that $J_1 \cup \{j\}$ is linearly dependent

[1] We do not include σ as defined in Prop. 12 to omit further nodes, as they will be pruned during propagation anyway.

Algorithm 1 Propagation Method for Spark Branch & Cut (based on (14))

1: Initialize $\underline{s}$ and $\overline{s}$ to current best (local) lower and upper spark bound, resp.
2: **if** $\underline{s} \geq \overline{s}$ **or** $|J_1| + |U| < \underline{s}$ **or** $|J_1| \geq \overline{s} - 1$ **then**
3: stop: cut off current node (branch contains no improving solutions, or too many 0- or 1-fixings, resp.)
4: **repeat**
5: **if** $\exists i \in [m], j \in U: \ \boldsymbol{A}_{i,J_1 \cup (U \setminus \{j\})} = \boldsymbol{0}^\top$, $a_{ij} \neq 0$ **then**
6: fix y_j to 0: $J_0 \coloneqq J_0 \cup \{j\}$, $U \coloneqq U \setminus \{j\}$
7: **if** $\exists i \in [m], k \in J_1, j \in U: \ \boldsymbol{A}_{i,(J_1 \setminus \{k\}) \cup (U \setminus \{j\})} = \boldsymbol{0}^\top$, $a_{ij} \neq 0$, $a_{ik} \neq 0$ **then**
8: **if** $J_1 \cup \{j\}$ is linearly dependent **then**
9: J_1 is a circuit $\rightsquigarrow$ found new primal solution $\hat{\boldsymbol{y}}$ with $\hat{\boldsymbol{y}}_{J_1 \cup \{j\}} = \mathbb{1}$, $\hat{\boldsymbol{y}}_{[n] \setminus (J_1 \cup \{j\})} = \boldsymbol{0}$
10: stop: cut off current node (branch contains no better solution than $\hat{\boldsymbol{y}}$)
11: fix y_j to 1: $J_1 \coloneqq J_1 \cup \{j\}$, $U \coloneqq U \setminus \{j\}$
12: **if** $\exists i \in [m], k \in J_1: \ \boldsymbol{A}_{i,(J_1 \setminus \{k\}) \cup U)} = \boldsymbol{0}^\top$, $a_{ik} \neq 0$ **then**
13: stop: cut off current node (infeasible branch: $\boldsymbol{a}_{i,J_1 \cup U}^\top \boldsymbol{x}_{J_1 \cup U} = 0$ impossible with $x_k \neq 0 \Leftrightarrow k \in J_1$)
14: update $\underline{s}$ (if new variable fixings were found)
15: **if** $\underline{s} \geq \overline{s}$ **or** $|J_1| + |U| < \underline{s}$ **or** $|J_1| \geq \overline{s} - 1$ **then**
16: stop: cut off current node (no improving solutions in branch, or too many 0- or 1-fixings, resp.)
17: **until** no further variable fixings could be inferred
18: **if** $\underline{s} \leq |J_1| + 1 < m + 1$ **and** $\exists j \in U: \ J_1 \cup \{j\}$ is linearly dependent **then**
19: $J_1 \cup \{j\}$ is a circuit $\rightsquigarrow$ found new primal solution $\hat{\boldsymbol{y}}$ with $\hat{\boldsymbol{y}}_{J_1 \cup \{j\}} = \mathbb{1}$, $\hat{\boldsymbol{y}}_{[n] \setminus (J_1 \cup \{j\})} = \boldsymbol{0}$
20: stop: cut off current node (branch contains no better solution than $\hat{\boldsymbol{y}}$)
21: update $\underline{s} \coloneqq \max\{\underline{s}, |J_1| + 2\}$
22: **if** $\underline{s} \geq \overline{s}$ **or** $|J_1| + |U| < \underline{s}$ **then**
23: stop: cut off current node (no improving solutions in branch, or too many 0-fixings, resp.)
24: **if** $|J_1| + |U| \leq m$ **and** $J_1 \cup U$ is linearly independent **then**
25: stop: cut off current node (branch contains no feasible solutions)

(and hence, by induction, a circuit), the method prunes the current node and pursues the corresponding branch no further, as it cannot yield better solutions. Exploiting this independence of J_1, the lower bound update also ensures that $\underline{s} \geq |J_1|+1$ initially; after having checked that $J_1 \cup \{j\}$ remains linearly independent for all $j \in U$, we can even employ $\underline{s} \geq |J_1| + 2$. (In particular, this also asserts that after the next branching decision, in the node corresponding to a new 1-fixing, J_1 is again independent initially.)

Furthermore, we could make use of the dual matroid representation matrix $\boldsymbol{B}$ to detect when too many (and the "wrong") variables have been fixed to 0: If J_0 is part of the complement of a circuit support, it must hold that $\boldsymbol{B}_{J_0 \cdot} \boldsymbol{q} = \boldsymbol{0}$ for some $\boldsymbol{q} \neq \boldsymbol{0}$, but if $\boldsymbol{B}_{J_0 \cdot}$ has full column rank $n-m$, this homogeneous equation is only solved by the forbidden all-zero vector. Hence, in this case, the current branch contains no feasible solutions of (14) and can be cut off. However, this check may only become effective if $|J_0| \geq n - m = \operatorname{rank}(\boldsymbol{B})$, i.e., in deep levels of the search tree. In our experiments, we found it hardly ever led to actual cut-offs but was computationally rather expensive; as it therefore appears overall to be unhelpful in improving the solution progress, we have deactivated it in our code and omitted it from Algorithm 1 (also because it would require us to explicitly compute $\boldsymbol{B}$, which in general, we do not need and wish to avoid for numerical reasons).

Finally, note that the inferences in Steps 5-6 and 12-13 of Algorithm 1 are, in fact, local versions of the case of sparsity-1-rows described earlier in Section 7.1.

Remark 19. *The propagation and cut-off rules are based on the properties of circuits and thus make direct use of linear dependencies among matrix columns. The ability to exploit such information makes the direct approach* (14) *structurally more attractive than the cogirth IP model* (15) *considered in [8]. There, most of this information is "hidden" in the "co" of "cocircuit" – the approach builds on the characterization of cocircuits as the (inclusion-wise) minimal sets with nonempty intersection with every basis and is essentially blind to the fact that cocircuits are circuits of the dual matroid and, as such, offer structural properties that can aid the solution process. Therefore, although the spark and cospark hitting set formulations are very similar (as are the associated optimal greedy separation routines for covering inequalities), the model* (14) *and our algorithm based on it offer clear advantages over the matroid-dual formulation, even disregarding the previously mentioned potential numerical issues and runtime overhead that come with computing* $\boldsymbol{B}$.

7.5 Valid Inequalities and Cut Separation

The covering inequalities $\boldsymbol{y}(\overline{B}) \geq 1$ are generated by greedily finding a basis, traversing columns in descending order of $\boldsymbol{y}$-values, as described in Section 6. To strengthen the formulation, it is desirable to find other inequalities that are valid for our problem, preferably ones that define facets of the polytope that forms the feasible set of the LP relaxation.

7.5.1 Set Covering Perspective

To that end, since the complete model (14), i.e., including *all* constraints, is a set covering problem, we can, in principle, draw on the large body of polyhedral results regarding set covering polytopes, see, e.g., the good (though perhaps a bit outdated) survey in [17, Section 1.9], and the classical references therein. However, many interesting results on facet-defining inequalities are tied to certain structured submatrices of the (often sparse) constraint system, which in our case is very dense (every covering inequality includes all but m of the n variables) and thus somewhat unlikely to exhibit highly useful substructures. Moreover, the separation problems for significant inequality classes of set covering polytopes are computationally hard [17]. For the related cospark IP (15), which can also—analogously to (14)—be seen as a set covering problem, a class of valid inequalities from [12] of the form $\boldsymbol{a}^\top \boldsymbol{y} \geq 2$ with $\boldsymbol{a} \in \{0,1,2\}^n$ was considered in the branch & cut method in [8], but their experiments showed that (violated) cuts from this class were hardly ever found. Therefore, weighing effort for separation based on the already known/generated subset of all constraints against the potential to significantly reduce the search space, we only incorporated one of the known classes of set covering facets into our branch & cut scheme:

$$\boldsymbol{y}(\overline{B} \cup \{k\}) \geq 2 \qquad \text{for a basis } B \text{ and some } k \in B. \tag{17}$$

These inequalities can be derived as a special case of generalized antiweb (or generalized cycle) inequalities, cf. [17], and are valid if $B \setminus \{k\} \cup \{j\}$ is a basis for every $j \in \overline{B}$. In fact, they can also be conceived from aggregation of the covering inequalites associated with those $n - m$ bases: Summing up $\boldsymbol{y}(\overline{B}) \geq 1$ and $\boldsymbol{y}(B \setminus \{k\} \cup \{j\}) \geq 1$ for $k \in B$ and every $j \notin B$, we obtain that for binary $\boldsymbol{y}$,

$$(n-m)\boldsymbol{y}(\overline{B} \cup \{k\}) \geq n - m + 1 \quad \Leftrightarrow \quad \boldsymbol{y}(\overline{B} \cup \{k\}) \geq \lceil (n-m+1)/(n-m) \rceil = 2,$$

where the right-hand side can be rounded up since $\boldsymbol{y} \in \{0,1\}^n$ only has integral coefficients here. Note that the cuts (17) belong to the same class as the valid inequalities considered in [8] for the cospark problem ($\boldsymbol{a}^\top \boldsymbol{y} \geq 2$ with $\boldsymbol{a} \in \{0,1,2\}^n$), with two advantages: In (17), all coefficients on the left-hand side are actually in $\{0,1\}$, whereas the routine to obtain inequalities described in [8] is likely to lead to one or more 2-coefficients, yielding weaker cuts. Moreover, said routine requires inspection of covering inequalities that have already been added to the model, whereas we can generate a cut of the form (17) without including the covering inequalities it aggregates into the model. (In fact, in our experiments, it proved beneficial to add one cut (17) rather than all corresponding $n - m + 1$ covering inequalities.)

In our implementation, we first greedily construct a basis B to solve the covering inequality separation problem. If this procedure did not "skip" any columns (due to linear dependence on those already included in the basis being built), we pick the last index k added to B (i.e., one with smallest corresponding entry in the LP relaxation solution) to maximize cut violation and test whether $B \setminus \{k\} \cup \{j\}$ is also a basis, for every $j \notin B$. If so, we add the corresponding cut (17). (Further choices of k are not considered.)

7.5.2 Matrix Sparsity Pattern Cuts

Similarly to some propagation rules, valid inequalities to be added to the IP (14) can be derived straightforwardly from the sparsity pattern of the representation matrix: For every $j \in \operatorname{supp}(\boldsymbol{a}_i^\top)$, we must have either $y_j = 0$ or $y_j = 1$ *and* $\boldsymbol{y}(\operatorname{supp}(\boldsymbol{a}_i^\top) \setminus \{j\}) \geq 1$. This disjunction can be expressed as one inequality for each such j:

$$-y_j + \boldsymbol{y}\Big(\operatorname{supp}\left(\boldsymbol{a}_i^\top\right) \setminus \{j\}\Big) = -y_j + \sum_{\substack{k \in \operatorname{supp}(\boldsymbol{a}_i^\top), \\ k \neq j}} y_k \geq 0 \quad \forall\, i \in [m]. \tag{18}$$

It is well-known (cf., e.g., [17]) that if an inequality $\boldsymbol{a}^\top \boldsymbol{y} \geq \alpha$ defines a facet of a set covering polytope other than the upper bound constraints $y_j \leq 1$, then all its coefficients are nonnegative. Thus, the inequalities (18)

will not be globally facet-defining, but the one for $j \in \operatorname{supp}(\boldsymbol{a}_i^\top)$ might become important in the subproblems where y_j is fixed to 1.

As with parity-check inequalities in the special case of binary vector matroids (see, e.g., [44, 51]), redundancy could be of help here, i.e., additional linearly dependent rows may be used to infer further information in the form of the above cuts (and in propagation). Therefore, in our implementation, we add all of the above "sparsity pattern cuts" to the initial (root) LP, in particular also including those associated with rows that are removed by preprocessing (where only $\operatorname{rank}(\boldsymbol{A})$ rows are kept). We presently do not, however, generate further redundant rows.

7.5.3 Solution Exclusion, Bound, and Cocircuit-Intersection Cuts

In [8], the feasible solution exclusion inequalities from [14] were included in their cospark branch & cut method and reported to be helpful. For our algorithm, we could not make the same empirical observation for these cuts; similarly, spherical and cylindrical cuts from [61] or canonical cuts from [11] could also serve to cut off feasible or infeasible binary solutions, but mostly did not appear to help either.

Further valid inequalities can be derived from upper and lower bounds: For any basis B, we can impose that $\boldsymbol{y}(B) \leq \overline{s} - 1$, which excludes dependent sets of size exceeding the current best upper bound $\overline{s}$. For a similar idea on how to make use of *lower* bounds $\underline{s}$, note that the incidence vector of any subset $C \subseteq [n]$ (with $|C| \leq n - \underline{s}$) obeys the disjunction that either $\boldsymbol{y}(C) \geq 1$ or $\boldsymbol{y}(C) = 0$ *and* $\boldsymbol{y}(\overline{C}) \geq \underline{s}$, which gives rise to the inequality $\underline{s}\boldsymbol{y}(C) + \boldsymbol{y}(\overline{C}) \geq \underline{s}$. (Similarly, cuts of the form $\boldsymbol{y}(C) + (|C|/2)\boldsymbol{y}(\overline{C}) \geq |C|$ may be used to encourage picking at least two columns from the complement of a circuit C if not choosing C itself, as is necessary w.r.t. optimality unless C is already optimal.)

Finally, we also tested the addition of cuts based on the following property of matroid circuits:

Lemma 20. *[70, Prop. 2.1.11] For any circuit C and cocircuit C^* of a matroid, it holds that $|C \cap C^*| \neq 1$.*

From this, we can derive cuts similar to (18) to express that either $|C \cap C^*| = 0$ or $|C \cap C^*| \geq 2$:

$$-y_j + \boldsymbol{y}(C^* \setminus \{j\}) \geq 0 \qquad \forall\, j \in C^*.$$

Note that even without access to the dual matroid (via its representation matrix $\boldsymbol{B}$), we can construct cocircuits as the complements of matroid hyperplanes (cf. [70]); for the vector matroid $\mathcal{M}[\boldsymbol{A}]$, $C^* = \overline{H}$ for an inclusion-wise maximal set H of column indices such that $\operatorname{rank}(\boldsymbol{A}_{\cdot H}) = \operatorname{rank}(\boldsymbol{A}) - 1 = m - 1$.

Alas, like the different kinds of solution exclusion inequalities, none of these other cuts proved useful overall, whence their addition is not included in our final implementation.

7.5.4 Local Cuts

Many classes of valid inequalities can be modified to incorporate local information (variable fixings), yielding cuts that are valid only in the local branch. For instance, we could explicitly add $\boldsymbol{y}(U) \geq 2$ as a local cut, as our propagation scheme either cuts off the present node or implies exactly that at least two more columns (of those still unfixed) are needed to achieve local feasibility. In fact, this cut can be strengthened to the cardinality cut $\boldsymbol{y}(U) \geq \max\{2, \underline{s}_{\text{local}} - |J_1|\}$, using the best available local lower bound $\underline{s}_{\text{local}}$. However, cardinality cuts like this are known to often hinder the solution progress, and indeed, the above-described cuts only sped up the overall solution process for a few, but had an adversary effect on the majority of considered test instances. Thus, their inclusion is disabled in our branch & cut implementation by default.

The case of having only two nonzero coefficients in a row of $\boldsymbol{A}$ was described in Section 7.1 already: In this case, we can merge the variables, as they are necessarily equal in all feasible solutions. We realize this aspect by adding local cuts $y_j = y_k$ whenever $\|\boldsymbol{a}_{i,J_1}^\top\|_0 = 0$ and $\operatorname{supp}(\boldsymbol{a}_{i,U}^\top) = \{j, k\}$ for some $i \in [m]$; in the root node, i.e., if $\|\boldsymbol{a}_i^\top\|_0 = 2$, equating the corresponding variables is implied by the sparsity pattern cuts (18) associated with row i. (Note that (18) would, in principle, also enforce $y_j = y_k$ locally, but these inequalities may at some point be removed from the model due to inactive-constraint aging; actually, forcing them to remain and omitting the local cuts proved inferior in preliminary test runs.)

7.6 Further Implementation Details

For all independence tests, we use dedicated Gaussian elimination procedures that avoid unnecessary repetition of compuations where possible and employ column pivoting to enhance numerical stability. For instance, to test whether $J_1 \cup \{j\}$ forms a circuit for some $j \in U$ during propagation, we bring $\boldsymbol{A}_{\cdot J_1}$ into row echelon form (storing the row operation coefficients in-place below the "steps" of the transformed matrix, which would have zeros there) so that to test for dependence of $J_1 \cup \{j\}$, we only need to apply the transformations to column $\boldsymbol{A}_j$ and check whether another nonzero pivot can be found or not. Similarly, we can detect with relatively little effort whenever inferred variable fixings lead to a circuit. Such an elimination scheme is also used in the main separation routine, i.e., the greedy method to construct a basis with minimal left-hand side value in the associated covering inequality, as well as for construction of cuts (17). We use a numerical tolerance of 10^{-9} for comparisons (in particular, α with $|\alpha| < 10^{-9}$ is treated as zero).

The inner products of each pair of (normalized) matrix columns is computed only once and subsequently used for quicker computation of local mutual coherence and coherence index values, respectively.

In our branch & cut algorithm based on (14), the many other general-purpose primal heuristics available in the SCIP framework were rarely successful; in particular, we turned off all diving heuristics as well as most rounding and several other heuristics[2] as they consumed an appreciable amount of running time while hardly ever producing a feasible solution. Similarly, SCIP only very rarely found (violated) general-purpose inequalities (such as zero-half or general Gomory cuts) for the LP subproblems encountered throughout the solving process; yet, we did not turn off any of SCIP's own separation routines explicitly as the runtime overhead seemed overall negligible.

Regarding covering inequalities and others described in Section 7.5, we build a *cut pool* that collects all cuts found so far and keeps them stored until they have been inactive in 100 000 subsequent separation calls. (Note that only a fraction of the inequalities is typically active locally and included in the local LP relaxation, which is therefore kept relatively small and quick to solve; the choice which constraints to include in the LP is left to the SCIP framework.) In separation, we first check all pool cuts and then only run our separation routines if none of the stored cuts were violated. Furthermore, we restricted separation of new cuts to the root node and to enforcing of integral (LP) solutions only – recall that for the latter, one always either finds a (maximally) violated cut or certifies feasibility (cf. Corollary 16), the latter case allowing to immediately prune the node. The very first (root) LP contains only the sparsity pattern inequalities (18); covering (and other) cuts are not added before it was solved once.

8 Numerical Experiments

We compare our specialized branch & cut solver for (14) with the direct MIP formulation (9); the other formulations detailed in Section 5 were found to be inferior to (9) in preliminary experiments. Our numerical experiments comprise both tests with purely open-source software (our implementation in SCIP 4.0.1 using SoPlex 3.0.1 as the LP solver vs. SCIP with SoPlex applied to the MIP (9)) as well as results obtained with propriety software (our implementation, using CPLEX 12.7.1 [2] as LP solver, vs. solving (9) with CPLEX or Gurobi 7.5.0 [1], resp.). Moreover, we tested a modification of our solver that combines the MIP formulation (9) with our various branch & cut tools (recall that both models include binary support variables to which propagation rules, cut separation etc. can be applied—with small changes such as using the alternative branching rule mentioned in Remark 18; we omit further details for the sake of brevity—also when the underlying main model is the MIP (9) rather than the pure IP (14)); this is again a SCIP implementation, tested with LP solvers SoPlex and CPLEX both.

As mentioned earlier, preliminary tests showed that the models (10), (11) and (12) all yielded significantly worse solver performance than the direct MIP formulation (9), and similarly, that taking a dual approach within our branch & cut framework (in particular, separating w.r.t. (15) rather than (14), thus mimicking the proposed cogirth algorithm from [8]) also resulted in notable performance deterioration of the method compared to our original (primal) scheme. Moreover, we tested three variants of the sequential ℓ_0-minimization approach (cf. Section 5.1), realized by combining our root-node branching rules with depth-first node selection when employing SCIP (using SoPlex) to solve (9) or when working with our branch & cut solver based on (14) or adapted to (9), respectively. However, it turned out that overall, none of these

[2]Namely, *randrounding, simplerounding, rounding, feaspump, locks, rens, rins, crossover* and *oneopt.*

variants were able to improve on the respective original versions (where the ℓ_0-minimization problems are implicitly present in the branch & bound tree, too, but are not actually solved in a sequential fashion).

Therefore, and to save space, we will not present detailed computational results for any of these other approaches, and (in Section 8.2 below) only discuss the outcome of numerical experiments with the models and solvers listed at the beginning of the present section. We left all respective stand-alone MIP solver's parameters at their default values, except lowering the integrality feasibility tolerance of CPLEX and Gurobi to 10^{-6} to match that of SCIP (also, for Gurobi, the default 10^{-5} led to numerical failure on a few of the non-binary/ternary instances); the constraint feasibility tolerance is 10^{-6} for all solvers. (In particular, this implies solvers accept $\boldsymbol{x}$ with $\|\boldsymbol{A}\boldsymbol{x}\|_\infty < 10^{-6}$; perhaps more conservatively, albeit not directly comparable, our Gaussian elimination independence checks treat absolute values down to 10^{-9} as nonzeros.) All computations were carried out on a desktop computer (8-core Intel Core i7-6700 CPU @ 3.40 GHz, 8 MB cache, 16 GB main memory) running Linux, using a single thread. For all solvers, we set a time limit of one hour (3 600 seconds) per instance.

Our implementations, as well as the test set described below, are available online from the author's homepage[3].

8.1 Test Instances

We conducted experiments with a variety of matrices; the full test set details are summarized in Tables 1, 2 and 3. Instances 1–41 are deterministic compressed sensing matrices constructed based on [47] or [29] as described in [35], or built according to [69], respectively; all of these matrices are binary or ternary (up to column scaling/normalization) and were designed to exhibit small mutual coherence values and/or favorable restricted isometry properties. Instances 42–68 are parity check matrices for binary linear codes from the online database [45]; we included some of these all-binary matrices here because such matrices, particularly low-density parity check (LDPC) matrices, have been shown to also allow for efficient sparse recovery in the context of compressed sensing tasks. Finally, instances 69–73 are non-binary LDPC parity check matrices (also taken from the database [45]), instances 74–82 and 83–91 are obtained from the respective matrices indicated in the "notes" column of Table 3 by replacing nonzero entries with numbers from $\{-1, 1\}$ or $\{-10, -9, \ldots, -1, 1, \ldots, 10\}$ (drawn uniformly at random), respectively, and instances 92–100 are partial Hadamard matrices (m rows drawn uniformly at random from an $n \times n$ Hadamard matrix).

8.2 Computational Results

The results of our numerical experiments are summarized in Tables 4 to 10.

First, in Table 4, we gather general information about the overall solver performances: Average runtimes, numbers of search tree nodes (including root node) and optimality gap estimates at termination (here defined as $100\% \cdot (\overline{s} - \underline{s})/\underline{s}$, where $\overline{s}$ is the best known upper spark bound and $\underline{s}$ the best known *integral* lower bound), the number of instances that were solved to optimality or ran into the time limit as well as the number of times a solver was fastest or reached the smallest final gap, respectively. Here, and in all other tables, running times and gap percentages as well as all reported averages are rounded to the nearest first significant digit (decimal); moreover, boldface numbers indicate the best achieved values over all solvers w.r.t. the respective measure, and slanted ones signify the best values achieved by using only non-commercial solver software.

Table 4 reveals that each one of our specialized branch & cut (B & C) solver variants can solve significantly more instances from the test set to optimality than any of the black-box MIP solvers SCIP, CPLEX or Gurobi. Moreover, they also reach smaller optimality gap values for instances on which they run into the time limit. In total, 17 instances could not be solved by any solver within the 1-hour time limit, and only 42 instances were solved by all solvers. Notably, SCIP (using SoPlex as the LP solver) could not even solve half of the instances, and even the commercial solvers only solved 58% (Gurobi) or 66% (CPLEX) of the instances within the given time limit, whereas our B & C codes were able to solve between 75% and 82% of the test cases. Comparing overall running times, applying our B & C with CPLEX as LP solver to the spark MIP formulation (9) outperforms all other variants. Regarding running times only on the subset of test cases for which all solvers succeeded (which thus, by definition, exhibits a bias towards easier instances), variants of our approach still beat the black-box solvers in almost all pair-to-pair comparisons for the three different time averages presented. A similar situation can be observed with respect to the number of search tree

[3] `http://www.graphics.rwth-aachen.de/person/265/`

Table 1: *Test set part 1 – Deterministic compressed sensing matrices.*

inst. no.	m	n	rank r	μ	# nonzeros/(rn)	notes
1	28	50	24	3/5	0.1592	binary, from [47, 35]
2	36	50	33	1/2	0.1073	binary, from [47, 35]
3	41	100	36	1/2	0.1314	binary, from [47, 35]
4	60	100	57	1/4	0.0661	binary, from [47, 35]
5	36	200	35	1/2	0.0554	binary, from [47, 35]
6	88	200	85	1/4	0.0452	binary, from [47, 35]
7	120	200	117	1/4	0.0333	binary, from [47, 35]
8	59	300	57	1/3	0.0506	binary, from [47, 35]
9	41	400	36	2/3	0.1313	binary, from [47, 35]
10	120	400	117	1/4	0.0332	binary, from [47, 35]
11	56	500	51	1/2	0.1011	binary, from [47, 35]
12	109	500	107	1/3	0.0275	binary, from [47, 35]
13	120	500	117	1/4	0.0333	binary, from [47, 35]
14	56	750	51	1/2	0.1011	binary, from [47, 35]
15	77	750	71	1/2	0.0950	binary, from [47, 35]
16	83	750	79	2/5	0.0598	binary, from [47, 35]
17	25	50	18	2/5	0.2167	binary, from [29, 35]
18	49	100	39	2/7	0.1605	binary, from [29, 35]
19	25	125	21	2/5	0.2000	binary, from [29, 35]
20	49	200	41	2/7	0.1527	binary, from [29, 35]
21	25	625	21	3/5	0.2000	binary, from [29, 35]
22	55	121	51	1/5	0.0909	binary, from [69]
23	65	169	61	1/5	0.0769	binary, from [69]
24	68	289	65	1/4	0.0588	binary, from [69]
25	114	361	109	1/6	0.0526	binary, from [69]
26	115	529	111	1/5	0.0435	binary, from [69]
27	93	961	91	1/3	0.0323	binary, from [69]
28	49	168	49	1/7	0.1429	ternary, from [35]
29	121	396	121	1/11	0.0910	ternary, from [35]
30	225	720	225	1/15	0.0667	ternary, from [35]
31	119	200	115	1/7	0.0609	ternary, from [35], based on [47]
32	41	400	41	1/3	0.0732	ternary, from [35], based on [47]
33	119	400	119	1/7	0.0588	ternary, from [35], based on [47]
34	49	500	49	1/3	0.0612	ternary, from [35], based on [47]
35	143	500	143	1/7	0.0490	ternary, from [35], based on [47]
36	59	750	59	1/3	0.0509	ternary, from [35], based on [47]
37	49	100	44	1/7	0.1591	ternary, from [35], based on [29]
38	121	200	112	1/11	0.0982	ternary, from [35], based on [29]
39	361	400	344	1/19	0.0552	ternary, from [35], based on [29]
40	121	500	114	1/11	0.0965	ternary, from [35], based on [29]
41	361	500	344	1/19	0.0552	ternary, from [35], based on [29]

Table 2: *Test set part 2 – Binary Parity check matrices from [45]. Construction parameters as stated in [45]: $N = n$ (except when stated otherwise), $K = n - r$, rate as given in column "notes".*

inst. no.	m	n	rank r	μ	# nonzeros/(rn)	notes
42	28	49	25	0.2500	0.1429	array code (rate 0.49)
43	44	121	41	0.2500	0.0909	array code (rate 0.66)
44	55	121	51	0.2000	0.0909	array code (rate 0.58)
45	52	169	49	0.2500	0.0769	array code (rate 0.71)
46	65	169	61	0.2000	0.0769	array code (rate 0.64)
47	76	361	73	0.2500	0.0526	array code (rate 0.80)
48	8	255	8	0.9354	0.5020	BCH code (rate 0.97)
49	9	511	9	0.9428	0.5010	BCH code (rate 0.98)
50	10	1023	10	0.9487	0.5005	BCH code (rate 0.99)
51	48	96	48	0.3333	0.0642	TU KL LDPC code (rate 1/2)
52	70	120	70	0.5000	0.0417	multi-edge type LDPC code ($N = 100$, $H = 20$, rate 1/2)
53	84	144	84	0.7071	0.0327	multi-edge type LDPC code ($N = 120$, $H = 24$, rate 1/2, v0)
54	140	240	140	0.7071	0.0196	multi-edge type LDPC code ($N = 200$, $H = 40$, rate 1/2)
55	588	1008	588	0.7071	0.0047	multi-edge type LDPC code ($N = 840$, $H = 168$, rate 1/2)
56	32	128	32	0.9186	0.3086	polar code (rate 3/4)
57	64	128	64	0.8292	0.2188	polar code (rate 1/2)
58	93	155	91	0.3333	0.0323	Tanner $(3, 5)$ LDPC code
59	96	576	96	0.8165	0.0347	WiMAX LDPC code (rate 5/6)
60	144	576	144	0.5000	0.0255	WiMAX LDPC code (rate 3/4 B)
61	112	672	112	0.8165	0.0298	WiMAX LDPC code (rate 5/6)
62	128	768	128	0.5774	0.0260	WiMAX LDPC code (rate 5/6)
63	224	1344	224	0.8165	0.0149	WiMAX LDPC code (rate 5/6)
64	96	192	96	0.5000	0.0330	Wimax-like LDPC code (rate 1/2)
65	120	240	120	0.5000	0.0264	Wimax-like LDPC code (rate 1/2)
66	168	336	168	0.5000	0.0189	Wimax-like LDPC code (rate 1/2)
67	192	384	192	0.5000	0.0165	WRAN LDPC code (rate 1/2)
68	160	480	160	0.5774	0.0211	WRAN LDPC code (rate 2/3)

Table 3: *Test set part 3 – Non-binary parity check matrices from [45] (construction parameters given in column "notes"), modified matrices with entries in* $\{-1, 0, 1\}$ *or* $\{-10, \ldots, 10\}$ *(obtained from those indicated in column "notes", abbreviated* ± 1 *or* ± 10*, resp.), and partial Hadamard matrices.*

inst. no.	m	n	rank r	μ	# nonzeros/(rn)	notes
69	32	64	32	0.9924	0.9995	$N = 512$, $K = 256$, GF 256 (d2)
70	12	72	12	0.9988	1.0000	$N = 576$, $K = 480$, GF 256
71	36	72	36	0.9821	1.0000	$N = 576$, $K = 288$, GF 256 (d2)
72	16	96	16	0.9927	0.9968	$N = 576$, $K = 480$, GF 64
73	48	96	48	0.9939	0.9996	$N = 576$, $K = 288$, GF 64 (d1)
74	60	100	60	0.8212	0.0667	± 1, sparsity pattern from inst. 4
75	109	500	109	0.9150	0.0275	± 1, sparsity pattern from inst. 12
76	120	500	120	0.8656	0.0333	± 1, sparsity pattern from inst. 13
77	77	750	77	0.7391	0.1039	± 1, sparsity pattern from inst. 15
78	49	200	47	0.6687	0.1489	± 1, sparsity pattern from inst. 20
79	44	121	44	0.8344	0.0910	± 1, sparsity pattern from inst. 43
80	55	121	55	0.6303	0.0910	± 1, sparsity pattern from inst. 44
81	70	120	70	0.9328	0.0417	± 1, sparsity pattern from inst. 52
82	32	128	32	0.9762	0.3086	± 1, sparsity pattern from inst. 56
83	60	100	60	0.8212	0.0667	± 10, sparsity pattern from inst. 4
84	109	500	109	0.9150	0.0275	± 10, sparsity pattern from inst. 12
85	120	500	120	0.8656	0.0333	± 10, sparsity pattern from inst. 13
86	77	750	77	0.7391	0.1039	± 10, sparsity pattern from inst. 15
87	49	200	47	0.6687	0.1489	± 10, sparsity pattern from inst. 20
88	44	121	44	0.8344	0.0910	± 10, sparsity pattern from inst. 43
89	55	121	55	0.6303	0.0910	± 10, sparsity pattern from inst. 44
90	70	120	70	0.9328	0.0417	± 10, sparsity pattern from inst. 52
91	32	128	32	0.9762	0.3086	± 10, sparsity pattern from inst. 56
92	16	128	16	0.7500	1.0000	partial Hadamard
93	32	128	32	0.4375	1.0000	partial Hadamard
94	16	256	16	0.8750	1.0000	partial Hadamard
95	32	256	32	0.4375	1.0000	partial Hadamard
96	64	256	64	0.3438	1.0000	partial Hadamard
97	32	512	32	0.6250	1.0000	partial Hadamard
98	64	512	64	0.4063	1.0000	partial Hadamard
99	64	1024	64	0.4063	1.0000	partial Hadamard
100	128	1024	128	0.2500	1.0000	partial Hadamard

nodes: Here, the B & C version utilizing SoPlex, applied to the MIP (9), exhibits the smallest node numbers (w.r.t. all three average types) both on the whole test set as well as restricted to the instances solved by all codes, closely followed by the variant using CPLEX instead of SoPlex. In fact, although the former solves 7 instances less than the latter, it also reaches the best optimality gap estimates on instances it could not solve within the time limit (though the total average gap numbers indicate that larger outliers exist for the SoPlex-using variant than for the one employing CPLEX); overall, the CPLEX-using variant of our B & C code (applied to the MIP (9)), gives the best gap values after at most 1 hour in 95% of the cases. All variants of our B & C scheme and CPLEX turned out to be the fastest method on similar numbers of instances, with CPLEX and our B & C (with CPLEX as LP solver) applied to the IP (14) displaying a slight advantage overall. Comparing the B & C variants based on either the pure-binary IP (14) with their "siblings" based on the MIP (9), the additional information accessible by explicitly including the constraints $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$ (and $-\boldsymbol{y} + 2\boldsymbol{z} \leq \boldsymbol{x} \leq \boldsymbol{y}$, $\mathbb{1}^\top \boldsymbol{z} = 1$) apparently more than makes up for the increase in model size/variables – the MIP-based versions deliver consistently better overall results, though with a few exceptions and sometimes only slightly so. Using CPLEX instead of SoPlex seems to be more beneficial for the MIP-based variant than for the IP-based one. Finally, comparing only the two proprietary stand-alone MIP solvers, it is easily seen that Gurobi appears to perform significantly worse than CPLEX on our kind of problems; we therefore do not include details on test runs with Gurobi in the remainder of the evaluation of our computational experiments. (We do, however, keep including SCIP with SoPlex in the comparison, as it is the only open-source stand-alone MIP solver considered in this paper. In fact, since our B & C codes are implemented in the SCIP framework, the most direct conclusions regarding the impact of our specialized solver components can be drawn from comparing our codes with SCIP, all using SoPlex as LP solver.)

Tables 5–7 and 8–10 contain detailed computational results for the different instance types (cf. Tables 1–3) for all purely open-source solvers or those using CPLEX, respectively. For every algorithm/solver, we report the running times (in seconds), finished solving nodes, final lower and upper bounds ($\underline{s}$ and $\overline{s}$, respectively) as well as the estimated time $t_{\overline{s}}$ at which the (first) best primal-feasible solution was found; moreover, in each of these tables, the last three rows state arithmetic, geometric and shifted geometric means (with shifts 10 for runtimes and 100 for node numbers), respectively.

The detailed numerical results confirm the overall impression gleaned from Table 4: On all test set parts, our B & C codes achieve the best results w.r.t. both running times and solving nodes. Notably, using SoPlex as LP solver, the IP-based B & C is faster than its MIP-based counterpart on the deterministic compressed sensing matrix and parity-check matrix instances, with the situation reversed on the third test set part with

Table 4: *Summary of solver performance measures over the whole test set. DNF stands for "did not finish", i.e., when the time limit was reached. The three runtime and node values given per solver are, from top to bottom, arithmetic mean, geometric mean and shifted geometric means (with shift factor* 10 *or* 100*, respectively), the three average gap values are arithmetic and geometric means as well as the medians.*

solver/model	avg. time		avg. nodes		# solved	# DNF	avg. gap (%)		# fastest	# best gap
	total	solved by all	total	solved by all			total	DNF only		
SCIP w/ SoPlex, (9)	2137.5	247.6	698315.7	163853.7	46	54	347.9	644.3	17	46
	435.1	241.6	46290.2	2898.1			—	331.4		
	617.4	49.0	70209.2	7729.5			38.1	287.5		
B & C w/ SoPlex, (14)	1227.3	227.1	308901.5	110825.9	*75*	*25*	111.7	446.4	*35*	77
	47.6	***54.5***	4798.6	601.3			—	202.4		
	169.2	19.4	11803.5	2528.4			***0.0***	271.4		
B & C w/ SoPlex, (9)	*1105.6*	*29.7*	***205317.1***	***11633.0***	*75*	*25*	*85.8*	***343.3***	34	*82*
	47.7	69.3	***2057.5***	***157.7***			—	***149.5***		
	145.8	*8.7*	***5182.7***	***642.6***			***0.0***	***200.0***		
CPLEX, (9)	1329.5	38.8	860507.0	83371.8	66	34	205.9	640.9	**43**	66
	47.6	97.7	18191.4	729.0			—	250.2		
	142.2	**5.6**	25076.8	1276.6			**0.0**	300.0		
GUROBI, (9)	1648.3	97.5	851772.2	71593.0	58	42	221.9	528.2	23	58
	169.7	176.6	39837.4	3587.0			—	263.1		
	303.2	22.8	44398.4	4280.4			**0.0**	287.5		
B & C w/ CPLEX, (14)	1091.3	226.1	323640.1	115510.9	76	24	107.1	446.8	**43**	79
	51.3	54.6	4605.7	593.8			—	205.2		
	142.5	18.9	11418.0	2504.7			**0.0**	269.3		
B & C w/ CPLEX, (9)	**872.3**	**19.8**	284445.3	11786.9	**82**	**18**	**64.2**	356.8	37	**95**
	35.0	55.2	2649.6	165.6			—	179.6		
	98.8	6.1	6681.1	677.8			**0.0**	202.9		

non-binary or -ternary (and apparently harder) instances. Both these variants beat SCIP (with SoPlex) as a stand-alone solver w.r.t. all performance measures on all test set parts; the MIP-based B & C using SoPlex moreover exhibits the overall lowest average numbers of solving nodes. In terms of running time, large improvements can be gained by employing CPLEX instead of SoPlex, both regarding black-box solution of (9) (SCIP vs. CPLEX) and for the B & C codes. Here, it appears that the additional information explicitly provided by the MIP formulation can be used particularly beneficially by CPLEX rather than SoPlex: It turns out that the MIP-based B & C version using CPLEX is almost always the overall fastest algorithm on each test set part, especially w.r.t. the shifted geometric mean running times, which is arguably the most important average time measure considered as it reduces the influence of easy instances.

It can also be observed that our B & C codes solve all ternary compressed sensing matrix instances (28–41) in the root node already; this is explained by the fact that the mutual coherence lower bounds for the spark turned out to be sharp in these cases, and our LP based basis pursuit heuristic finds an optimal solution immediately. Without exploiting these tools (lower bounding procedure and primal heuristic), these instances still remain mostly easy for CPLEX, but SCIP (with SoPlex) struggles with several of them and could not solve them within one hour. Nevertheless, even discarding instances 28–41, the conclusions about best achieved node numbers and running times can be upheld also on test set part 1, see Table 11.

The efficiency of the IP-based B & C on test set parts 1 and 2 is most likely related to the sparsity of the test matrices: In both parts, the matrices are mostly quite sparse (often between 5% and 10% nonzero entries), which makes the propagation rules particularly effective and conceivably also positively influences the Gaussian elimination routines. While these parts are also present in the MIP-based B & C, the additional constraints and variables to be handled there appear to result in slight runtime overhead compared to the IP-based version, at least on the usually very sparse (often around 2% nonzeros only) LDPC matrices. In contrast, on test set part 3 with mostly, and indeed often almost completely, dense matrices, the MIP-based B & C is clearly superior to the IP-based version, regardless of the employed LP solver. This shows that especially when not too much (sparsity) structure from the matrix itself can be exploited, the explicit inclusion of the MIP formulation (9) rather than working solely based on (14) is worthwhile despite increasing the model size (in particular, number of variables) and moving from a pure IP to a MIP.

Regarding the times until the respective best solutions were obtained (columns labeled "$t_{\overline{s}}$"), it can be observed that the B & C variants often, though not always, achieve the best values, which demonstrates the effectiveness of our primal heuristics. Indeed, for almost all instances, the best solution in a B & C variant was found by either one of our LP-based heuristics or during Gaussian elimination (in separation or propagation). On instances that could not be solved within the time limit, particularly on test set part 3, the primal bounds obtained by our B & C solvers are often larger than those obtained by CPLEX, and to a lesser extent, also

Table 5: *Experimental results on part 1 of the test set (cf. Table 1) for the MIP formulation* (9) *solved by SCIP using SoPlex as LP solver and for our branch & cut algorithms based on* (14) *or* (9) *(also employing SoPlex), respectively.*

	SCIP with SoPlex, MIP (9)					B & C with SoPlex, IP (14)					B & C with SoPlex, MIP (9)				
inst.	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$
1	0.3	3	6	6	0.1	0.2	129	6	6	<0.1	0.3	51	6	6	<0.1
2	4.8	17365	12	12	0.6	4.5	8581	12	12	<0.1	2.2	2659	12	12	1.8
3	41.3	55526	8	8	0.3	10.9	11115	8	8	<0.1	4.8	2309	8	8	<0.1
4	149.6	194471	12	12	0.8	9.5	6292	12	12	3.0	6.6	2324	12	12	4.8
5	1.0	2	4	4	<0.1	0.7	243	4	4	<0.1	3.4	201	4	4	<0.1
6	3600.0	913948	4	12	3.3	137.6	31633	12	12	0.2	121.5	14169	12	12	0.2
7	3600.0	637838	4	15	3.1	72.2	10385	15	15	<0.1	112.2	7673	15	15	<0.1
8	607.3	256631	6	6	0.7	5.7	1774	6	6	<0.1	47.7	301	6	6	<0.1
9	43.3	18497	4	4	0.7	5.7	663	4	4	0.1	50.4	401	4	4	0.2
10	3600.0	350018	2	12	71.0	3202.6	291916	12	12	58.0	3168.8	129021	12	12	314.2
11	3600.0	313531	3	6	8.8	132.9	42291	6	6	1.5	201.9	4197	6	6	46.2
12	3600.0	242456	2	6	2.3	15.1	1266	6	6	0.1	215.0	501	6	6	0.1
13	3600.0	63815	3	8	14.9	79.6	5413	8	8	1.4	508.5	501	8	8	8.2
14	3600.0	157498	2	6	18.1	1208.6	222205	6	6	0.7	959.7	15475	6	6	29.9
15	3600.0	78438	2	6	26.5	433.6	45255	6	6	232.3	502.1	774	6	6	11.8
16	3600.0	46830	2	6	8.4	105.0	12153	6	6	15.5	580.0	1357	6	6	7.0
17	0.5	247	6	6	0.1	0.4	796	6	6	<0.1	0.4	473	6	6	<0.1
18	5.9	3349	8	8	0.6	11.1	9911	8	8	<0.1	2.6	577	8	8	0.3
19	19.5	27286	6	6	0.4	11.6	20161	6	6	<0.1	4.9	3224	6	6	<0.1
20	259.0	74120	8	8	2.5	941.6	626743	8	8	0.4	17.5	2011	8	8	0.1
21	3600.0	1153914	3	4	2.6	45.4	20998	4	4	0.3	108.1	626	4	4	1.1
22	1.4	5	10	10	0.5	28.9	18667	10	10	<0.1	4.0	122	10	10	<0.1
23	1632.9	1135323	12	12	6.8	2400.0	1128555	12	12	0.2	357.6	132754	12	12	18.0
24	3600.0	817511	4	10	15.4	1703.8	455438	10	10	0.1	1533.9	265709	10	10	60.0
25	3600.0	373584	2	23	536.0	3600.0	321919	10	38	104.3	3600.0	309729	12	38	2.9
26	3600.0	227956	2	12	71.9	3600.0	309097	9	12	4.5	3600.0	112890	10	12	51.7
27	3600.0	64509	2	6	1.8	368.3	28687	6	6	0.4	1220.8	962	6	6	0.4
28	19.6	3941	8	8	0.7	<0.1	1	8	8	<0.1	<0.1	1	8	8	<0.1
29	3600.0	85197	2	12	3.4	<0.1	1	12	12	<0.1	0.1	1	12	12	<0.1
30	3600.0	7448	2	16	12.1	0.1	1	16	16	<0.1	0.5	1	16	16	0.4
31	3.1	4	8	8	<0.1	<0.1	1	8	8	<0.1	<0.1	1	8	8	<0.1
32	5.3	2	4	4	0.2	<0.1	1	4	4	<0.1	<0.1	1	4	4	<0.1
33	1772.6	473659	8	8	1.7	<0.1	1	8	8	<0.1	0.1	1	8	8	<0.1
34	11.7	34	4	4	0.2	0.1	1	4	4	<0.1	0.1	1	4	4	<0.1
35	3600.0	113466	3	8	2.7	<0.1	1	8	8	<0.1	0.1	1	8	8	<0.1
36	3600.0	638837	3	4	1.0	0.1	1	4	4	0.1	0.1	1	4	4	<0.1
37	0.5	3	8	8	0.2	<0.1	1	8	8	<0.1	<0.1	1	8	8	<0.1
38	36.3	3375	12	12	0.1	<0.1	1	12	12	<0.1	0.1	1	12	12	<0.1
39	71.1	5419	20	20	0.3	<0.1	1	20	20	<0.1	0.2	1	20	20	0.2
40	3600.0	59685	3	12	6.5	<0.1	1	12	12	<0.1	0.2	1	12	12	0.2
41	3600.0	33853	3	20	4.8	0.1	1	20	20	<0.1	0.3	1	20	20	0.3
arithm.	1870.4	210965.7				442.3	88592.7				*413.1*	***24658.7***			
geom.	231.7	14479.4				***5.4***	625.1				7.3	***183.7***			
sh. geom.	378.3	25335.6				*39.3*	3028.9				43.1	***879.6***			

Table 6: *Experimental results on part 2 of the test set (cf. Table 2) for the MIP formulation* (9) *solved by SCIP using SoPlex as LP solver and for our branch & cut algorithms based on* (14) *or* (9) *(also employing SoPlex), respectively.*

	SCIP with SoPlex, MIP (9)					B & C with SoPlex, IP (14)					B & C with SoPlex, MIP (9)				
inst.	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$
42	0.2	2	8	8	0.1	0.5	952	8	8	<0.1	0.4	50	8	8	<0.1
43	360.1	509936	10	10	2.6	125.4	103832	10	10	<0.1	65.3	52840	10	10	<0.1
44	1.5	5	10	10	0.4	30.4	18833	10	10	<0.1	4.1	122	10	10	<0.1
45	1300.5	1183209	10	10	1.3	388.5	206024	10	10	<0.1	180.5	92884	10	10	0.3
46	2002.8	1264346	12	12	8.2	2355.1	1108199	12	12	0.9	364.6	132616	12	12	17.4
47	3600.0	802270	3	10	23.1	2965.1	650416	10	10	<0.1	3600.0	382519	9	10	158.5
48	4.1	394	3	3	0.3	0.2	1	3	3	0.1	0.3	1	3	3	0.2
49	21.5	4981	3	3	1.2	0.6	1	3	3	0.5	0.9	2	3	3	0.7
50	187.1	21875	3	3	0.6	3.0	1	3	3	3.0	5.1	2	3	3	4.1
51	3600.0	6062750	11	14	3.0	604.2	551531	14	14	53.3	380.3	247429	14	14	3.2
52	137.7	114363	11	11	2.2	2.5	1174	11	11	1.9	7.2	1191	11	11	0.2
53	330.0	304227	10	10	3.4	1.1	345	10	10	0.8	7.7	599	10	10	<0.1
54	3600.0	546045	4	13	19.9	21.4	1385	13	13	1.4	112.4	3076	13	13	84.7
55	3600.0	22682	2	18	48.5	2171.1	9141	18	18	12.5	3600.0	1619	12	19	82.7
56	0.8	3	4	4	0.2	1.9	3759	4	4	0.1	2.5	129	4	4	<0.1
57	127.6	107575	8	8	1.3	1445.3	324421	8	8	0.1	4.5	615	8	8	0.2
58	3600.0	1824513	6	20	204.0	1693.6	665539	20	20	0.2	2121.1	460244	20	20	<0.1
59	3600.0	156463	3	7	37.7	368.8	23551	7	7	98.7	729.1	15936	7	7	468.0
60	3600.0	149227	2	24	2397.0	2738.7	108827	9	9	2560.7	3600.0	63662	7	108	1203.0
61	3600.0	122719	2	7	58.4	363.4	25007	7	7	27.6	1117.7	14055	7	7	146.5
62	3600.0	78132	2	7	110.0	581.9	19924	7	7	416.0	1578.6	8063	7	7	48.5
63	3600.0	26694	2	13	1514.0	2591.2	22597	7	7	1657.9	3600.0	2197	3	7	2082.0
64	3600.0	868649	4	14	38.8	157.3	55639	14	14	35.1	423.9	48585	14	14	191.7
65	3600.0	405305	4	13	54.9	85.1	16605	13	13	47.5	390.1	15285	13	13	323.6
66	3600.0	204022	3	16	113.0	912.9	88796	16	16	462.3	3600.0	45564	13	16	3034.5
67	3600.0	108360	3	15	865.0	419.8	31752	15	15	60.1	1933.1	31090	15	15	636.7
68	3600.0	151219	2	12	850.0	3600.0	184881	10	14	3513.1	3600.0	40327	9	14	2868.9
arithm.	2165.7	557035.8				*875.1*	156312.3				1149.2	***61507.5***			
geom.	491.8	48247.2				*107.4*	10839.6				125.8	***3693.3***			
sh. geom.	715.5	71823.0				*200.5*	18395.9				222.0	***6413.8***			

Table 7: *Experimental results on part 3 of the test set (cf. Table 3) for the MIP formulation* (9) *solved by SCIP using SoPlex as LP solver and for our branch & cut algorithms based on* (14) *or* (9) *(also employing SoPlex), respectively.*

	SCIP with SoPlex, MIP (9)					B & C with SoPlex, IP (14)					B & C with SoPlex, MIP (9)				
inst.	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$
69	3600.0	9429070	8	10	63.3	3600.0	72099	6	10	1.8	514.8	1302967	10	10	0.4
70	47.2	150013	5	5	0.3	17.0	12107	5	5	0.1	6.2	22552	5	5	2.7
71	3600.0	6878277	7	10	8.4	3600.0	153174	7	10	0.4	302.0	748888	10	10	0.3
72	998.0	2656548	6	6	1.7	3600.0	145473	5	6	0.2	104.9	299448	6	6	5.3
73	3600.0	3188410	6	11	657.0	3600.0	57362	6	11	68.2	3600.0	4688901	8	11	47.1
74	3600.0	4459762	13	23	732.0	3600.0	2587712	14	49	2000.5	3600.0	1365077	16	41	858.4
75	2824.4	346801	6	6	10.0	13.3	1244	6	6	0.1	266.7	576	6	6	22.0
76	3600.0	262544	2	67	571.0	3600.0	104691	8	87	24.1	3600.0	50723	9	14	667.6
77	3600.0	388672	2	62	3394.0	3600.0	117259	3	66	17.2	3600.0	92981	6	61	17.1
78	3600.0	1483519	3	42	436.0	3600.0	2620475	6	44	0.3	3600.0	1769195	8	44	0.6
79	903.3	698628	10	10	101.0	1427.3	970943	10	10	1426.6	72.7	30906	10	10	65.7
80	3600.0	2554491	10	45	2827.0	3600.0	3316669	10	53	0.3	3600.0	1680809	12	51	1.1
81	99.0	79720	10	10	3.0	1.8	816	10	10	1.7	5.8	556	10	10	5.2
82	18.7	9678	4	4	4.9	1.6	2747	4	4	0.1	5.2	553	4	4	<0.1
83	3600.0	2334189	7	47	169.0	3600.0	2508833	14	52	<0.1	3600.0	780857	14	50	481.8
84	3600.0	388989	2	23	674.0	3600.0	77404	7	33	0.4	3600.0	36137	7	21	1271.7
85	3600.0	184724	2	84	1303.0	3600.0	101886	7	87	2.5	3600.0	53078	7	54	1022.1
86	3600.0	296195	2	65	54.1	3600.0	76658	3	68	43.4	3600.0	89191	3	61	24.2
87	3600.0	1449527	2	40	109.0	3600.0	1879067	6	43	1964.9	3600.0	1730125	8	44	0.3
88	3600.0	2774077	5	32	297.0	3600.0	3099744	10	36	3597.3	3600.0	1141427	10	36	23.9
89	3600.0	2168478	6	50	390.0	3600.0	3114930	10	52	0.2	3600.0	1129341	11	51	1.1
90	3600.0	2518493	6	22	271.0	3600.0	1811941	15	21	13.2	3600.0	585069	14	21	142.0
91	37.9	29619	4	4	16.2	2.0	3679	4	4	0.1	4.1	877	4	4	0.1
92	5.6	3959	4	4	0.1	8.0	10571	4	4	<0.1	0.7	587	4	4	<0.1
93	1.7	2	4	4	0.4	<0.1	1	4	4	<0.1	<0.1	1	4	4	<0.1
94	119.6	130055	4	4	0.1	293.2	52641	4	4	<0.1	6.6	4087	4	4	0.1
95	7.3	2	4	4	0.3	0.2	1	4	4	0.2	0.2	1	4	4	0.2
96	2133.1	118148	8	8	4.0	3600.0	5938	4	8	12.2	74.3	4117	8	8	66.4
97	2991.1	938317	4	4	1.5	3600.0	127033	3	4	1.3	34.0	514	4	4	1.4
98	3600.0	121885	2	8	57.2	3600.0	1201	4	8	4.5	3600.0	212093	7	8	2.6
99	3600.0	83133	1	8	69.7	3600.0	232	4	16	9.5	3600.0	37611	4	8	10.6
100	3600.0	16086	1	16	1541.0	3600.0	190	5	24	2292.8	3600.0	756	5	24	37.2
arithm.	2455.8	1441937.8				2791.5	719835.0				*2019.1*	***558125.0***			
geom.	879.8	198159.1				811.7	32855.5				*283.9*	***27742.2***			
sh. geom.	1016.4	253680.4				1711.6	45364.9				*557.1*	***38246.0***			

by SCIP (with SoPlex); here, the MIP-based B & C schemes and their IP-based counterparts are not clearly distinguished in terms of which one seems to more often find better solutions. Given that the heuristics in our B & C methods apparently work very well in general, the most likely explanation for the empirical findings that on hard instances, the stand-alone MIP solvers deliver better upper bounds than the B & C schemes, is the different inherent numerical accuracy the respective solvers work with: In particular, the stand-alone MIP solvers are satisfied with achieving $\|\boldsymbol{A}\boldsymbol{x}\|_\infty \leq 10^{-6}$ (w.r.t. the constraint $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$) whereas our B & C codes require pivot absolute values during Gaussian elimination linear (in-)dependence checks to be below 10^{-9} before accepting the corresponding solution as representing linearly dependent column subsets. Thus, it seems that though not directly comparable, the latter measure is a bit stricter, with the consequence that the B & C solvers accept potential solutions less often in numerically more challenging cases.

Generally, issues of numerical accuracy naturally limit the practical applicability of all solvers: The spark, like the notion of rank, is a discrete measure that is well-defined in exact arithmetic, but potentially problematic to evaluate in floating-point arithmetic. For parts 1 and 2 of the test set used for the present experimental evaluation, these issues appear to pose no problem for any solver, but on test set part 3 (where several instances exhibit relatively larger ratios between largest and smallest nonzero matrix coefficients, although all are still integers), one may already suspect issues of numerical precision to play a role, especially in light of the above-described differences in accepted primal solutions. In fact, precision limitations could be argued to make the whole concept of computing the spark for "less nice" matrices (in particular, say, sub-Gaussian random matrices that are often used in theoretical compressed sensing results) unreliable and less relevant for realistic applications, encouraging, e.g., the search for alternative measures that can be used to distinguish (sparse approximate) solutions to (noisy) linear measurements based on "geometric spread" (rather than a corresponding algrebaic viewpoint as represented by spark considerations), see [54]. On the other hand, disregarding possible application fields, numerical challenges are often unavoidable in the context of solving MIPs, and should therefore not be seen as a problem inherent to spark computation.

Yet more problematic than obtaining well-defined (in the numerical precision sense) primal solutions is, apparently, improving the lower bounds, up to eventually prove optimality of the best-known feasible solution. This can be deduced from the tables detailing the results of our computational experiments, by comparing the total running times needed to solve some instance with the respective time $t_{\overline{s}}$ until the (first)

Table 8: *Experimental results on part 1 of the test set (cf. Table 1) for the MIP formulation* (9) *solved by CPLEX and for our branch & cut algorithms based on* (14) *or* (9) *(employing CPLEX as LP solver), respectively.*

	CPLEX, MIP (9)					B & C with CPLEX, IP (14)					B & C with CPLEX, MIP (9)				
inst.	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$
1	<0.1	94	6	6	<0.1	0.1	113	6	6	<0.1	0.2	51	6	6	<0.1
2	1.1	5874	12	12	<0.1	5.8	7973	12	12	<0.1	2.1	2785	12	12	1.5
3	6.2	17038	8	8	<0.1	10.3	10897	8	8	<0.1	2.7	2509	8	8	0.8
4	10.3	29432	12	12	0.3	9.4	6435	12	12	0.1	3.7	2299	12	12	1.3
5	0.3	202	4	4	<0.1	0.3	369	4	4	<0.1	1.2	201	4	4	<0.1
6	582.3	872723	12	12	0.4	73.9	29617	12	12	<0.1	58.1	14219	12	12	0.3
7	208.8	179029	15	15	0.3	39.6	10063	15	15	<0.1	52.2	7659	15	15	<0.1
8	1.5	302	6	6	0.2	3.0	1706	6	6	<0.1	8.0	301	6	6	<0.1
9	2.2	402	4	4	0.1	2.4	473	4	4	0.2	12.0	401	4	4	0.2
10	3600.0	1742908	3	12	21.8	1139.8	258611	12	12	18.1	1199.9	129377	12	12	15.6
11	1098.6	452817	6	6	1.5	113.8	51973	6	6	0.5	81.6	5306	6	6	4.4
12	6.8	502	6	6	0.4	6.0	1224	6	6	0.4	34.5	501	6	6	0.5
13	17.8	715	8	8	12.1	31.8	5357	8	8	2.5	44.9	501	8	8	1.7
14	3600.0	767958	4	6	3.1	1056.8	307551	6	6	2.1	377.4	19136	6	6	14.2
15	26.9	916	6	6	16.6	407.0	65344	6	6	40.3	100.2	792	6	6	8.5
16	21.4	1294	6	6	8.1	61.8	14804	6	6	5.0	119.7	1367	6	6	11.0
17	<0.1	142	6	6	<0.1	0.5	734	6	6	<0.1	0.5	601	6	6	<0.1
18	0.7	923	8	8	<0.1	11.2	10453	8	8	<0.1	1.2	719	8	8	0.5
19	0.1	54	6	6	0.1	12.1	20199	6	6	<0.1	7.1	7552	6	6	<0.1
20	31.4	23850	8	8	<0.1	716.4	591915	8	8	0.4	11.4	3671	8	8	0.2
21	0.5	1	4	4	0.5	18.6	17968	4	4	0.5	21.0	627	4	4	0.9
22	<0.1	1	10	10	<0.1	27.4	19399	10	10	<0.1	1.3	122	10	10	<0.1
23	985.3	2651681	12	12	0.2	2069.1	1181681	12	12	<0.1	244.1	131639	12	12	4.3
24	3600.0	4489819	9	10	0.4	1040.5	516864	10	10	0.3	871.8	265310	10	10	15.0
25	3600.0	1802374	4	18	1855.1	3600.0	681280	10	38	0.5	3600.0	637513	12	23	266.0
26	161.0	42027	12	12	15.7	3600.0	671173	10	12	2.5	3600.0	318910	10	12	82.0
27	38.5	1922	6	6	0.8	150.1	18415	6	6	1.0	169.2	962	6	6	1.0
28	0.8	336	8	8	<0.1	0.1	1	8	8	<0.1	0.1	1	8	8	<0.1
29	15.1	792	12	12	0.5	1.2	1	12	12	1.2	1.3	1	12	12	1.3
30	108.3	2395	16	16	1.0	11.9	1	16	16	11.9	12.1	1	16	16	12.1
31	0.5	210	8	8	<0.1	<0.1	1	8	8	<0.1	0.1	1	8	8	<0.1
32	1.4	402	4	4	<0.1	0.1	1	4	4	0.1	0.1	1	4	4	0.1
33	6.2	402	8	8	0.3	0.4	1	8	8	0.4	0.5	1	8	8	0.5
34	2.4	506	4	4	0.1	0.2	1	4	4	0.2	0.2	1	4	4	0.2
35	17.4	1000	8	8	0.5	0.8	1	8	8	0.8	0.8	1	8	8	0.8
36	9.6	1500	4	4	0.2	0.4	1	4	4	0.4	0.5	1	4	4	0.5
37	0.2	200	8	8	<0.1	<0.1	1	8	8	<0.1	<0.1	1	8	8	<0.1
38	1.4	400	12	12	0.2	0.1	1	12	12	0.1	0.1	1	12	12	0.1
39	2.0	800	20	20	0.1	0.2	1	20	20	0.2	0.3	1	20	20	0.3
40	26.6	1000	12	12	5.2	0.5	1	12	12	0.5	0.6	1	12	12	0.6
41	36.1	1003	20	20	18.5	1.6	1	20	20	1.6	1.8	1	20	20	1.8
arithm.	434.9	319413.3				512.2	109819.6				**259.6**	37927.9			
geom.	9.9	2587.2				7.9	652.0				6.3	203.9			
sh. geom.	34.1	3690.2				32.9	3165.0				**25.0**	983.8			

Table 9: *Experimental results on part 2 of the test set (cf. Table 2) for the MIP formulation* (9) *solved by CPLEX and for our branch & cut algorithms based on* (14) *or* (9) *(employing CPLEX as LP solver), respectively.*

	CPLEX, MIP (9)					B & C with CPLEX, IP (14)					B & C with CPLEX, MIP (9)				
inst.	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$
42	<0.1	1	8	8	<0.1	0.7	926	8	8	<0.1	0.2	50	8	8	<0.1
43	0.4	1320	10	10	<0.1	114.9	107948	10	10	<0.1	55.4	53632	10	10	0.1
44	<0.1	1	10	10	<0.1	28.1	19757	10	10	<0.1	1.3	122	10	10	<0.1
45	0.5	689	10	10	0.1	278.9	204242	10	10	<0.1	146.1	93918	10	10	0.1
46	1.0	1503	12	12	0.3	2106.4	1188187	12	12	0.2	249.1	130740	12	12	5.9
47	11.5	6398	10	10	0.5	1851.5	805002	10	10	0.2	1863.1	436315	10	10	19.3
48	0.1	16	3	3	0.1	0.1	1	3	3	0.1	0.2	1	3	3	0.2
49	0.3	18	3	3	0.2	0.6	1	3	3	0.6	0.8	2	3	3	0.6
50	0.8	20	3	3	0.5	3.2	1	3	3	3.1	4.9	2	3	3	4.1
51	3600.0	5167948	11	14	7.7	580.6	497781	14	14	22.1	312.8	245450	14	14	21.0
52	9.5	24467	11	11	0.9	1.4	831	11	11	<0.1	4.2	1147	11	11	<0.1
53	10.5	14352	10	10	0.7	1.4	321	10	10	0.4	3.0	595	10	10	0.1
54	2171.3	2279777	13	13	3.3	10.0	1325	13	13	0.2	32.1	3159	13	13	19.3
55	3600.0	407815	2	18	170.5	1624.8	10331	18	18	4.4	3266.9	21723	18	18	1072.2
56	0.2	130	4	4	<0.1	1.9	4647	4	4	<0.1	0.9	129	4	4	<0.1
57	0.4	428	8	8	0.1	2302.6	240727	8	8	0.1	2.4	623	8	8	0.1
58	3600.0	4947751	16	20	1.6	1496.7	653127	20	20	0.2	1483.4	478002	20	20	222.6
59	3600.0	1011789	3	7	176.9	136.0	20749	7	7	1.4	245.9	14700	7	7	32.9
60	3600.0	859065	6	9	6.7	357.0	37605	9	9	108.6	875.1	62894	9	9	555.1
61	3600.0	1102604	3	7	126.4	154.8	17961	7	7	2.0	248.5	13881	7	7	27.4
62	3600.0	660529	3	7	87.4	338.7	23726	7	7	25.2	476.9	22621	7	7	437.4
63	3600.0	285314	3	7	8.3	701.2	13332	7	7	75.0	1632.2	13903	7	7	1360.0
64	3600.0	3558954	11	14	6.4	164.1	70654	14	14	68.3	219.2	46012	14	14	86.0
65	1269.4	1348282	13	13	7.4	37.8	11229	13	13	3.6	106.8	12523	13	13	65.7
66	3600.0	2375672	11	16	4.6	511.4	84887	16	16	56.6	1367.2	89421	16	16	802.0
67	3600.0	1787609	4	15	4.7	231.8	30789	15	15	51.9	486.0	30646	15	15	124.0
68	3600.0	1564222	3	12	609.2	868.6	145870	12	12	525.0	1949.0	104113	12	12	1412.6
arithm.	1728.7	1015062.0				**515.0**	155257.7				556.8	69493.5			
geom.	**50.4**	20002.7				70.0	9861.9				60.5	4767.0			
sh. geom.	209.7	36031.1				133.5	16763.0				**123.8**	8276.6			

Table 10: *Experimental results on part 3 of the test set (cf. Table 3) for the MIP formulation* (9) *solved by CPLEX and for our branch & cut algorithms based on* (14) *or* (9) *(employing CPLEX as LP solver), respectively.*

	CPLEX, MIP (9)					B & C with CPLEX, IP (14)					B & C with CPLEX, MIP (9)				
inst.	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$	time	nodes	$\underline{s}$	$\overline{s}$	$t_{\overline{s}}$
69	2023.7	2847610	10	10	180.6	3600.0	48837	6	10	1.4	531.9	1111915	10	10	27.8
70	31.6	101837	5	5	2.0	22.4	12905	5	5	0.4	7.2	23070	5	5	2.9
71	3600.0	4614860	9	10	0.9	3600.0	90994	6	10	0.2	440.5	757237	10	10	31.2
72	955.2	2250714	6	6	35.5	3600.0	140409	5	6	0.7	107.4	297654	6	6	12.6
73	3600.0	3355428	7	11	183.7	3600.0	44963	5	11	5.6	3600.0	5136425	8	11	286.6
74	3600.0	4030260	14	23	689.7	3600.0	2537697	14	22	3058.0	3600.0	2015849	16	39	238.8
75	31.2	8671	6	6	0.4	7.1	1174	6	6	0.6	41.7	582	6	6	10.5
76	3600.0	1257245	2	39	1154.7	3600.0	138412	8	83	5.7	3600.0	184105	10	14	76.7
77	3600.0	836880	2	56	555.5	3600.0	78322	3	64	25.4	3600.0	316057	6	65	88.5
78	3600.0	3537047	4	41	22.9	3600.0	2864353	6	44	1878.1	3600.0	2904818	8	44	0.3
79	480.0	581307	10	10	45.2	1442.5	1147169	10	10	1442.5	47.5	30854	10	10	40.3
80	3600.0	3646112	5	47	3460.0	3600.0	2944150	10	52	0.1	3600.0	2629253	13	50	308.1
81	1.3	2882	10	10	<0.1	0.8	427	10	10	<0.1	2.4	525	10	10	0.2
82	3.0	2742	4	4	1.7	1.7	3353	4	4	<0.1	2.1	535	4	4	0.1
83	3600.0	2806148	10	43	3492.4	3600.0	2724988	14	53	<0.1	3600.0	1372695	15	48	389.9
84	3600.0	1733628	2	20	19.5	3600.0	82353	7	33	0.6	3600.0	146124	7	20	16.2
85	3600.0	966302	2	61	3159.0	3600.0	160969	8	77	0.9	3600.0	180713	8	78	254.1
86	3600.0	849700	2	63	1048.7	3600.0	59759	3	67	27.2	3600.0	240455	3	57	16.0
87	3600.0	2179918	3	42	1856.0	3600.0	2110390	6	45	1.7	3600.0	2612244	8	44	0.6
88	3600.0	3318762	4	31	286.6	3600.0	3150436	10	36	<0.1	3600.0	1781083	10	37	371.6
89	3600.0	3701723	8	46	3531.8	3600.0	3247407	10	53	0.3	3600.0	1934173	11	51	307.2
90	3600.0	1182013	8	21	3399.6	3600.0	1875445	15	21	101.3	3600.0	994892	15	21	17.1
91	17.6	17180	4	4	0.8	2.3	4841	4	4	<0.1	2.6	853	4	4	0.2
92	0.4	519	4	4	<0.1	9.4	10167	4	4	<0.1	0.5	565	4	4	<0.1
93	0.3	256	4	4	<0.1	<0.1	1	4	4	<0.1	<0.1	1	4	4	<0.1
94	16.2	18183	4	4	0.1	309.3	52651	4	4	0.1	5.6	4827	4	4	0.1
95	1.3	512	4	4	0.2	0.2	1	4	4	0.2	0.2	1	4	4	0.2
96	70.3	23249	8	8	0.7	3600.0	5894	4	8	32.8	27.4	2817	8	8	0.3
97	10.9	1024	4	4	1.5	3600.0	130411	3	4	1.3	13.5	514	4	4	1.1
98	3600.0	1185089	3	8	2.3	3600.0	317	4	8	3.6	2718.0	219905	8	8	1.5
99	3600.0	339864	2	8	2.4	3600.0	91	4	8	94.6	3600.0	77751	6	10	8.7
100	3600.0	150419	2	22	2760.5	3600.0	158	5	40	45.3	3600.0	34664	8	16	10.8
arithm.	2138.8	1423377.6				2531.1	739670.1				**1923.4**	781661.1			
geom.	342.2	204336.2				468.4	29659.0				**204.3**	43152.7			
sh. geom.	535.8	209917.9				805.4	41892.4				**381.5**	59358.7			

Table 11: *Average runtimes and node numbers for all solvers and branch & cut variants on instances 1–27 (binary deterministic compressed sensing matrices only).*

	SCIP, (9)		B & C / SoPlex, (14)		B & C / SoPlex, (9)		CPLEX, (9)		B & C / CPLEX, (14)		B & C / CPLEX, (9)	
	time	node	time	node	time	node	time	node	time	node	time	node
arithm.	1969.1	267580.4	671.7	134529.1	*627.2*	***37444.1***	651.9	484629.6	879.4	166762.6	**393.6**	57593.7
geom.	271.0	33503.9	*59.3*	17608.2	66.3	***2743.1***	**14.9**	5459.6	109.9	18769.7	27.3	3212.4
sh. geom.	448.4	50086.2	*102.6*	18458.2	115.2	***3081.9***	57.8	8776.9	77.0	19697.7	**53.5**	3609.6

best solution was found, and also by the sometimes very large gaps at the end of the time limit for unsolved instances. Particularly the black-box solvers often seem to get stuck at values close to the trivial lower bounds for a long time. Naturally, instances with larger spark values can thus also be observed to be harder to solve than instances of similar dimensions and matrix density with lower optimal value; this can already be witnessed for the very small instances 1 and 2, see Tables 5 and 8. Indeed, the tabled results allow to conclude that for lower matrix density, larger spark values can be proven to be optimal (largely due to more efficient propagation), see, in particular, the results for sparse LDPC matrices in Tables 6 and 9 compared to the denser matrices with (likely) similarly large spark values seen in Tables 7 and 10.

For applications in sparse recovery guarantees, lower spark bounds are in fact more important than upper bounds: Statements like "$\boldsymbol{x}$ can be recovered from measurements $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ if $\|\boldsymbol{x}\|_0 < \operatorname{spark}(\boldsymbol{A})/2$" obviously remain valid when replacing $\operatorname{spark}(\boldsymbol{A})$ by a lower bound, but do not hold with upper bounds instead. At least in that light, it is noteworthy that the lower bounds our B & C solvers achieve (when running into the time limit) are significantly larger than those obtained by black-box optimization of (9) with SCIP or CPLEX (or Gurobi, too). This might be partially attributable to larger root lower bounds (e.g., from the mutual coherence bound in Proposition 10, item 2, and/or the initial sparsity-pattern inequalities (18)); however, note that "external" bounds like the mutual coherence one are not enforced in the model via universally usable cardinality constraints $\mathbb{1}^\top \boldsymbol{y} \geq \underline{s}$ but are used only in the propagation routine of our B & C implementations.

9 Concluding Remarks and Outlook

In this paper, we investigated algorithmic approaches to compute the spark of a matrix. The proposed matroid-based branch & cut schemes were demonstrated to outperform black-box optimization of MIP formulations of the spark problem on a large test set, with respect to several relevant measures such as number of instances solved to optimality, average running times or numbers of search tree nodes, or lower bounds/optimality gaps achieved after a one-hour time limit. The evaluation of the numerical results showed that our proposed primal heuristics are very often able to quickly find good or even optimal solutions, and that the propagation routines employed in our B & C solvers can significantly decrease the search space and speed up the overall solution process, especially for matrices with low density.

Regarding the present B & C algorithm and implementation, further improvements may be possible w.r.t. several aspects: As the propagation rules are particularly effective for sparse matrices, one may consider a *matrix sparsification* preprocessing step, in which the input matrix $\boldsymbol{A}$ is "sparsified" by elementary row operations; the sparser linear combinations of rows could either replace existing (denser) ones of $\boldsymbol{A}$ or be added as (redundant) rows to the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$ underlying the spark IP or MIP model. However, matrix sparsification is itself an NP-hard problem—indeed, its optimal solution contains a spark-defining vector (of another matrix), cf. [33, 75]— and also hard to approximate [64, 41, 76], with only a handful of known heuristics (see, e.g., [3, 65, 19, 24, 40, 15]). The practicality of such an approach may therefore be somewhat doubtful. Matrix sparsification could, on the other hand, also be beneficial regarding possible problem decomposition by permutation-based heuristic methods: The sparser the matrix, the likelier it intuitively becomes to find a permutation of rows and columns that yields a bordered block-diagonal structure by which the original instance could then be split into smaller subproblems. Returning to the present B & C implementions, it may also be worthwhile to take the matrix density into account for deciding whether to even perform certain propagation steps or include certain valid inequalities: If a matrix row is "too dense" (in a sense then to be defined), the sparsity-pattern based propagation rules will likely not be successful and one could save the effort of trying; similarly, the number and density of the cuts (18) increases with the density of the associated matrix row – for fully dense rows $\boldsymbol{a}_i^\top$, the cuts are essentially meaningless. These aspects currently do not find consideration, but could speed up the algorithms for dense instances such as the partial Hadamard matrices from test set part 3 (which, coincidentally, would actually be amenable to simple greedy matrix sparsification, as relatively large parts of rows are identical up to the sign).

Beyond the still open question whether spark computation is NP-hard in the *strong* sense, several further aspects remain open for future research consideration. As pointed out in Sections 6 and 7, our branch & cut solver can be straightforwardly adapted to the computation of the girth for other matroids, where obviously, propagation rules and cutting planes that exploit the sparsity pattern of a matrix are only applicable in case of representable matroids (i.e., those that can be expressed as vector matroids w.r.t. an appropriate representation matrix) as is the MIP formulation (9) itself.

One example of an important matroid class to consider is that of *binary matroids*, which are vector matroids represented by binary matrices for which linear independence is defined over the binary field $\mathbb{F}_2$. The girth of binary matroids (represented by a binary matrix $\boldsymbol{A} \in \{0,1\}^{m\times n}$),

$$\begin{aligned} & \min \mathbb{1}^\top \boldsymbol{x} \quad \text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{0} \bmod 2,\ \boldsymbol{x} \in \{0,1\}^n \\ = \quad & \min \mathbb{1}^\top \boldsymbol{x} \quad \text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} = 2\boldsymbol{z},\ \boldsymbol{x} \in \{0,1\}^n,\ \boldsymbol{z} \in \mathbb{Z}^m. \end{aligned}$$

is known as the minimum (Hamming) distance of binary linear codes and thus finds application in coding theory and cryptography; it is also NP-hard to compute [80]. Recent years brought forth several works proposing integer programming formulations and branch & cut schemes for binary matroid girth computation, along with studies of the polyhedral structure of the polytope underlying the respective LP relaxations, see, e.g., [74, 51, 44, 59]. However, our hitting-set model (13) and matroid-based algorithmic ingredients appear to be new in this context and could combine beneficially with existing approaches.

Another problem of interest in the context of compressed sensing is to verify whether a given matrix $\boldsymbol{A}$ is "full spark", i.e., whether $\operatorname{spark}(\boldsymbol{A}) = m + 1$. For a general matroid $\mathcal{M}$, this amounts to asking whether $\mathcal{M}$ is *uniform*, i.e., whether the girth of $\mathcal{M}$ equals $r_{\mathcal{M}} + 1$. Uniformity verification was shown in [71, 48] to be a challenging problem in general, and also for the special case of vector matroids, full spark verification is known to be (co)NP-complete [78, Corollary 3]. Indeed, using one of the models to compute the spark (or, more generally, girth) discussed in the present paper might not be the best approach for this purpose:

As our numerical experiments show, the algorithmic challenge appears to largely lie in improving the lower bounds, whereas finding a minimum circuit can often be achieved quickly. In case of a full spark matrix, a smallest circuit is trivial to find (just take any $m+1$ columns), but to prove that one indeed could not do any better, the lower bound would have to be brought all the way up to $m+1$. (This problem essentially persists also if one switches to an infeasible-model approach that includes the constraint $\|\boldsymbol{x}\|_0 \leq m$.) Thus, achieving practically efficient methods for the full spark—or uniformity—verification problem apparently requires further attention and research efforts. To that end, further matroid duality arguments might become important; for instance, a matroid is uniform if and only if its dual is also uniform (which appears to be "folklore" knowledge and indeed has a simple proof; for vector matroids and the associated full-spark notion, it is shown in [54]), or that a disjoint circuit-cocircuit-pair can exist if and only if the respective matroids are not uniform (which follows easily from Lemma 20). Indeed, based on these observations, the full-spark property can be decided by solving an IP feasibility problem:

Proposition 21. *A matroid $\mathcal{M} = (E, \mathcal{B})$ is uniform if and only if there exist no $\boldsymbol{y}, \boldsymbol{z}, \boldsymbol{q} \in \{0,1\}^{|E|}$ such that*

$$\boldsymbol{y}(\overline{B}) \geq 1 \;\; \forall\, B \in \mathcal{B}, \quad \boldsymbol{z}(B) \geq 1 \;\; \forall\, B \in \mathcal{B}, \quad \mathbb{1}^\top \boldsymbol{q} = 0, \quad \boldsymbol{y} + \boldsymbol{z} - \mathbb{1} \leq \boldsymbol{q}, \quad \boldsymbol{q} \leq \boldsymbol{y}, \quad \boldsymbol{q} \leq \boldsymbol{z}.$$

Proof. The binary vectors $\boldsymbol{y}$ obeying $\boldsymbol{y}(\overline{B}) \geq 1$ for all $B \in \mathcal{B}$ are the incidence vectors of cycles, in particular, including circuits. Analogously, the binary vectors $\boldsymbol{z}$ obeying $\boldsymbol{z}(B) \geq 1$ for all $B \in \mathcal{B}$ are the incidence vectors of cocycles, including cocircuits. Thus, if and only if $\mathcal{M}$ is not uniform, we could find $\boldsymbol{y}, \boldsymbol{z} \in \{0,1\}^{|E|}$ such that $\boldsymbol{y}^\top \boldsymbol{z} = 0$ (thus, a disjoint circuit-cocircuit-pair). This bilinear constraint can be linearized exactly using the well-known McCormick envelope techniques: The binary variables $\boldsymbol{q}$ satisfying the constraints $\boldsymbol{y} + \boldsymbol{z} - \mathbb{1} \leq \boldsymbol{q}$, $\boldsymbol{q} \leq \boldsymbol{y}$ and $\boldsymbol{q} \leq \boldsymbol{z}$ are easily seen to express $q_j = y_j z_j$ for all $j \in [|E|]$, so $\boldsymbol{y}^\top \boldsymbol{z} = 0$ holds if and only if $\mathbb{1}^\top \boldsymbol{q} = 0$. □

Note that in Proposition 21, we do not need explicit access to the dual matroid, so in particular, in the vector matroid case, there is no need to compute the dual representation matrix. It is, however, beyond the scope of the present paper to further investigate the full-spark/uniformity verification problem and the possible merits of different approaches such as the one given above; we therefore leave it for future consideration as well.

Finally, let us mention that it would be interesting to see how the spark algorithms proposed in this work fare in practice when applied to solving sparse reconstruction problems of the form (2). Is it beneficial to view the problem of finding a sparsest solution to a given linear equation system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ as the problem of finding the smallest-cardinality circuit of $\mathcal{M}[(\boldsymbol{A}, \boldsymbol{b})]$ that contains the last column (associated with $\boldsymbol{b}$), rather than directly searching for sparse $\boldsymbol{x}$ that satisfy $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ (as done in [49], with a branch & cut method based on a related hitting-set-type reformulation of (2) for which covering inequality separation is NP-hard)?

Acknowledgments

The author would like to thank Marc Pfetsch for inspiring discussions in the early stages of this research effort and his support in getting started working with the SCIP framework, and the authors of [35] and [69] for kindly providing code for their respective deterministic compressed sensing matrix construction routines.

References

[1] *Gurobi Optimizer (7.5.0)*, `www.gurobi.com`.

[2] *IBM ILOG CPLEX Optimization Studio (12.7.1)*, `www.ibm.com/products/ilog-cplex-optimization-studio`.

[3] A. J. Hoffman and S. T. McCormick, *A Fast Algorithm That Makes Matrices Optimally Sparse*, in Progress in Combinatorial Optimization, W. R. Pulleyblank, ed., Academic Press, 1984, pp. 185–196.

[4] B. Alexeev, J. Cahill, and D. G. Mixon, *Full spark frames*, J. Fourier Anal. Appl., 18 (2012), pp. 1167–1194.

[5] E. Amaldi and V. Kann, *The complexity and approximability of finding maximum feasible subsystems of linear relations*, Theoret. Comput. Sci., 147 (1995), pp. 181–210.

[6] E. Amaldi and V. Kann, *On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems*, Theoret. Comput. Sci., 209 (1998), pp. 237–260.

[7] J. D. Arellano, *Algorithms to Find the Girth and Cogirth of a Linear Matroid*, PhD thesis, Rice University, 2014.

[8] J. D. Arellano and I. V. Hicks, *Degree of redundancy of linear systems using implicit set covering*, IEEE Trans. Automation Sci. Eng., 11 (2014), pp. 274–279.

[9] S. Arora, L. Babai, J. Stern, and Z. Sweedyk, *The hardness of approximate optima in lattices, codes, and systems of linear equations*, J. Comput. System Sci., 54 (1997), pp. 317–331.

[10] C. Aykanat, A. Pinar, and U. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM J. Sci. Comput., 25 (2004), pp. 1860–1879.

[11] E. Balas and R. Jeroslow, *Canonical cuts on the unit hypercube*, SIAM J. Appl. Math., 23 (1972), pp. 61–69.

[12] E. Balas and S. M. Ng, *On the set covering polytope: I. All the facets with coefficients in 0, 1, 2*, Math. Program., 43 (1989), pp. 57–69.

[13] M. Bausal, K. Kianfar, Y. Ding, and E. Moreno-Centeno, *Hybridization of bound-and-decompose and mixed integer feasibility checking to measure redundancy in structured linear systems*, IEEE Trans. Automation Sci. Eng., 10 (2013), pp. 1151–1157.

[14] J. E. Beasley and K. Jornsten, *Enhancing an algorithm for set covering problems*, European J. Oper. Res., 58 (1992), pp. 293–300.

[15] M. W. Berry, M. T. Heath, I. Kaneka, M. Lawo, R. J. Plemmons, and R. C. Ward, *An Algorithm to Compute a Sparse Basis of the Null Space*, Numerische Mathematik, 47 (1985), pp. 483–504.

[16] R. E. Bixby and M. J. Saltzman, *Recovering an optimal LP basis from an interior point solution*, Oper. Res. Lett., 15 (1994), pp. 169–178.

[17] R. Borndörfer, *Aspects of Set Packing, Partitioning, and Covering*, Doctoral dissertation, TU Berlin, Germany, 1998.

[18] E. Candès and T. Tao, *Decoding by linear programming*, IEEE Trans. Inform. Theory, 51 (2005), pp. 4203–4215.

[19] S. F. Chang and S. T. McCormick, *A Hierarchical Algorithm for Making Sparse Matrices Sparser*, Mathematical Programming, 56 (1992), pp. 1–30.

[20] S. S. Chen, D. L. Donoho, and M. A. Saunders, *Atomic decomposition by basis pursuit*, SIAM J. Sci. Comput., 20 (1998), pp. 33–61.

[21] A. Chistov, H. Fournier, L. Gurvits, and P. Koiran, *Vandermonde matrices, NP-completeness, and transversal subspaces*, Found. Comput. Math., 3 (2003), pp. 421–427.

[22] J. J. Cho, Y. Chen, and Y. Ding, *On the (co)girth of a connected matroid*, Discrete Appl. Math., 155 (2007), pp. 2456–2470.

[23] M. Cho, K. V. Mishra, and W. Xu, *New algorithms for verifying the null space condition in compressed sensing.* arXiv:1604.02769 [cs.IT], 2016.

[24] T. F. Coleman and A. Pothen, *The Sparse Null Space Basis Problem.* Tech. Rep. TR 84-598, Cornell University, Ithaca, NY, USA, 1984.

[25] P. Damaschke, O. Eğecioğlu, and L. Molokov, *Fixed-parameter tractability of error correction in graphical linear systems*, in Proc. WALCOM 2013, S. K. Ghosh and T. Tokuyama, eds., vol. 7748 of LNCS, Springer, 2013, pp. 245–256.

[26] A. d'Aspremont, F. Bach, and L. E. Ghaoui, *Optimal solutions for sparse principal component analysis*, J. Mach. Learn. Res., 9 (2008), pp. 1269–1294.

[27] A. d'Aspremont and L. E. Ghaoui, *Testing the nullspace property using semidefinite programming*, Math. Program., 127 (2011), pp. 123–144.

[28] A. d'Aspremont, L. E. Ghaoui, M. I. Jordan, and G. R. G. Lanckriet, *A direct formulation for sparse PCA using semidefinite programming*, SIAM Rev., 49 (2007), pp. 434–448.

[29] R. A. DeVore, *Deterministic constructions of compressed sensing matrices*, J. Complexity, 23 (2007), pp. 918–925.

[30] A. G. Dimakis, R. Smarandache, and P. O. Vontobel, *LDPC codes for compressed sensing*, IEEE Trans. Inform. Theory, 58 (2012), pp. 3093–3114.

[31] D. L. Donoho, *Compressed sensing*, IEEE Trans. Inform. Theory, 52 (2006), pp. 1289–1306.

[32] D. L. Donoho and M. Elad, *Optimally sparse representation in general (non-orthogonal) dictionaries via ℓ^1 minimization*, Proc. Natl. Acad. Sci. USA, 100 (2003), pp. 2197–2202.

[33] S. Egner and T. Minkwitz, *Sparsification of Rectangular Matrices*, Journal of Symbolic Computation, 26 (1998), pp. 135–149.

[34] M. Elad, *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*, Springer, Heidelberg, Germany, 2010.

[35] T. Fischer, *Konstruktion von dünn besetzten Sensing-Matrizen [in German].* Diploma Thesis, TU Darmstadt, Germany, 2012.

[36] S. Foucart and H. Rauhut, *A Mathematical Introduction to Compressive Sensing*, Appl. Numer. Harmon. Anal., Birkhäuser, 2013.

[37] T. Gally and M. E. Pfetsch, *Computing restricted isometry constants via mixed-integer semidefinite programming.* Optimization Online Preprint, 2016, `http://www.optimization-online.org/DB_HTML/2016/04/5395.html`.

[38] M. R. Garey and D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 19790.

[39] W. Gharibi, *An improved lower bound of the spark with application*, Int. J. Distrib. Parallel Syst., 3 (2012).

[40] J. R. Gilbert and M. T. Heath, *Computing a Sparse Basis for the Null Space*, SIAM Journal on Algebraic and Discrete Methods, 8 (1987), pp. 446–459.

[41] L.-A. Gottlieb and T. Neylon, *Matrix sparsification and the sparse null space problem*, Algorithmica, 76 (2016), pp. 426–444.

[42] R. Gribonval and M. Nielsen, *Sparse representations in unions of bases*, IEEE Trans. Inform. Theory, 49 (2003), pp. 3320–3325.

[43] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, vol. 2 of Algorithms Combin., Springer, 2nd ed., 1993.

[44] M. Helmling, S. Ruzika, and A. Tanatmis, *Mathematical programming decoding of binary linear codes: Theory and algorithms*, IEEE Trans. Inform. Theory, 58 (2012), pp. 4753–4769.

[45] M. Helmling, S. Scholl, F. Gensheimer, T. Dietz, K. Kraft, S. Ruzika, and N. Wehn, *Database of channel codes and ML simulation results*, 2017, `www.uni-kl.de/channel-codes`.

[46] A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM J. Comput., 7 (1978), pp. 413–423.

[47] M. A. Iwen, *Simple deterministically constructible RIP matrices with sublinear fourier sampling requirements*, in Proc. CISS 2009, 2009, pp. 870–875.

[48] P. M. Jensen and B. Korte, *Complexity of matroid property algorithms*, SIAM J. Comput., 11 (1982), pp. 184–190.

[49] S. Jokar and M. E. Pfetsch, *Exact and approximate sparse solutions of underdetermined linear equations*, SIAM J. Sci. Comput., 31 (2008), pp. 23–44.

[50] A. Juditsky and A. Nemirovski, *On verifiable sufficient conditions for sparse signal recovery via $\ell 1$ minimization*, Math. Program., 127 (2011), pp. 57–88.

[51] A. B. Keha and T. Duman, *Minimum distance computation of ldpc codes using a branch and cut algorithm*, IEEE Trans. Commun., 58 (2010), pp. 1072–1079.

[52] L. Khachiyan, E. Boros, K. Elbassioni, V. Gurvich, and K. Makino, *On the complexity of some enumeration problems for matroids*, SIAM J. Discrete Math., 19 (2006), pp. 966–984.

[53] K. Kianfar, A. Pourhabib, and Y. Ding, *An integer programming approach for analyzing the measurement redundancy in structured linear systems*, IEEE Trans. Automation Sci. Eng., 8 (2011), pp. 447–450.

[54] E. King, *Algebraic and geometric spread in finite frames*, in Proc. SPIE 9597, Wavelets and Sparsity XVI, 2015, p. 95970B.

[55] P. Koiran and A. Zouzias, *On the certification of the restricted isometry property.* arXiv:1103.4984 [cs.CC], 2011.

[56] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500.

[57] J. B. Kruskal, *Three-way arrays: Rank and uniqueness of trilinear decompositions*, Linear Algebra Appl., 18 (1977), pp. 95–138.

[58] J.-H. Lange, M. E. Pfetsch, B. M. Seib, and A. M. Tillmann, *Sparse recovery with integrality constraints.* arXiv:1608.08678 [cs.IT], 2016.

[59] P. Lisoněk and L. Trummer, *Algorithms for the minimum weight of linear codes*, Adv. Math. Commun., 10 (2016), pp. 195–207.

[60] X.-J. Liu and S.-T. Xia, *Constructions of quasi-cyclic measurement matrices based on array codes*, in Proc. ISIT 2013, 2013, pp. 479–483.

[61] N. Maculan, E. M. Macambira, and C. C. de Souza, *Geometrical cuts for 0–1 integer programming*, Tech. Report IC-02-006, Instituto de Computacao, Universidade Estadual de Campinas, 2002.

[62] S. J. Maher, T. Fischer, T. Gally, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig, *The SCIP Optimization Suite 4.0*, Tech. Report 17-12, ZIB, Takustr.7, 14195 Berlin, 2017.

[63] A. Manikas and C. Proukakis, *Modeling and estimation of ambiguities in linear arrays*, IEEE Trans. Signal Process., 46 (1998), pp. 2166–2179.

[64] S. T. McCormick, *A Combinatorial Approach to some Sparse Matrix Problems*, PhD thesis, Stanford University, 1983.

[65] S. T. McCormick, *Making Sparse Matrices Sparser: Computational Results*, Mathematical Programming, 49 (1990), pp. 91–111.

[66] E. Minieka, *Finding the circuits of a matroid*, J. Res. Natl. Bureau of Standards B, 80B (1976), pp. 337–342.

[67] J. E. Mitchell, *Branch-and-cut algorithms for combinatorial optimization problems*, in Handbook of Applied Optimization, P. M. Pardalos and M. G. C. Resende, eds., Oxford Univ. Press, 2002, pp. 65–77.

[68] B. Monien, *The complexity of determining a shortest cycle of even length*, Computing, 31 (1983), pp. 355–369.

[69] R. R. Naidu, P. Jampana, and C. S. Sastry, *Deterministic compressed sensing matrices: Construction via euler squares and applications*, IEEE Trans. Signal Process., 64 (2016), pp. 3566–3575.

[70] J. G. Oxley, *Matroid Theory*, Oxf. Grad. Texts Math., Oxford Univ. Press, 2nd ed., 2011.

[71] G. C. Robinson and D. J. A. Welsh, *The computational complexity of matroid properties*, Math. Proc. Cambridge Philos. Soc., 87 (1980), pp. 29–45.

[72] P. D. Seymour, *Decomposition of regular matroids*, J. Combin. Theory Ser. B, 28 (1980), pp. 305–359.

[73] P. D. Seymour, *A note on hyperplane generation*, J. Combin. Theory Ser. B, 61 (1994), pp. 88–91.

[74] A. Tanatmis, S. Ruzika, H. W. Hamacher, M. Punekar, F. Kienle, and N. Wehn, *Valid inequalities for binary linear codes*, in Proc. ISIT 2009, 2009, pp. 2216–2220.

[75] A. M. Tillmann, *Computational Aspects of Compressed Sensing*, Doctoral dissertation, TU Darmstadt, Germany, 2013.

[76] A. M. Tillmann, *On the Computational Intractability of Exact and Approximate Dictionary Learning*, IEEE Signal Processing Letters, 22 (2015), pp. 45–49.

[77] A. M. Tillmann, R. Gribonval, and M. E. Pfetsch, *Projection onto the cosparse set is NP-hard*, in Proc. ICASSP 2014, 2014, pp. 7148–7152.

[78] A. M. Tillmann and M. E. Pfetsch, *The computational complexity of the restricted isometry property, the nullspace property, and related concepts in compressed sensing*, IEEE Trans. Inform. Theory, 60 (2014), pp. 1248–1259.

[79] K. Truemper, *A decomposition theory for matroids. V. Testing of matrix total unimodularity*, J. Combin. Theory Ser. B, 49 (1990), pp. 241–281.

[80] A. Vardy, *The intractability of computing the minimum distance of a code*, IEEE Trans. Inform. Theory, 43 (1997), pp. 1757–1766.

[81] M. Walter and K. Truemper, *Implementation of a unimodularity test*, Math. Program. Comput., 5 (2013), pp. 57–73.

[82] J. Zhang, G. Han, and Y. Fang, *Deterministic construction of compressed sensing matrices from protograph LDPC codes*, IEEE Signal Process. Lett., 22 (2015), pp. 1960–1964.

[83] X.-D. Zhang, *Matrix Analysis and Applications*, Cambridge Univ. Press, 2017.

[84] Z. Zhu, A. M.-C. So, and Y. Ye, *Fast and near-optimal matrix completion via randomized basis pursuit*, in Proc. 5th ICCM, L. Ji, Y. S. Poon, L. Yang, and S.-T. Yau, eds., vol. 51 of AMS/IP Stud. Adv. Math., AMS and International Press, 2012, pp. 859–882.