Near-Linear Time Algorithm for n-fold ILPs via Color Coding

Klaus Jansen
Department of Computer Science, Kiel University, Kiel, Germany
kj@informatik.uni-kiel.de

Alexandra Lassota
Department of Computer Science, Kiel University, Kiel, Germany
ala@informatik.uni-kiel.de

Lars Rohwedder
Department of Computer Science, Kiel University, Kiel, Germany
lro@informatik.uni-kiel.de

Abstract

We study an important case of ILPs $\max\{c^T x \mid \mathcal{A}x = b, l \le x \le u, x \in \mathbb{Z}^{nt}\}$ with $n \cdot t$ variables and lower and upper bounds $\ell, u \in \mathbb{Z}^{nt}$. In n-fold ILPs non-zero entries only appear in the first r rows of the matrix $\mathcal{A}$ and in small blocks of size $s \times t$ along the diagonal underneath. Despite this restriction many optimization problems can be expressed in this form. It is known that n-fold ILPs can be solved in FPT time regarding the parameters s, r, and Δ, where Δ is the greatest absolute value of an entry in $\mathcal{A}$. The state-of-the-art technique is a local search algorithm that subsequently moves in an improving direction. Both, the number of iterations and the search for such an improving direction take time $\Omega(n)$, leading to a quadratic running time in n. We introduce a technique based on Color Coding, which allows us to compute these improving directions in logarithmic time after a single initialization step. This leads to the first algorithm for n-fold ILPs with a running time that is near-linear in the number nt of variables, namely $(rs\Delta)^{\mathcal{O}(r^2 s + s^2)} L^2 \cdot nt \log^{\mathcal{O}(1)}(nt)$, where L is the encoding length of the largest integer in the input. In contrast to the algorithms in recent literature, we do not need to solve the LP relaxation in order to handle unbounded variables. Instead, we give a structural lemma to introduce appropriate bounds. If, on the other hand, we are given such an LP solution, the running time can be decreased by a factor of L.

2012 ACM Subject Classification Mathematics of computing → Integer programming

Keywords and phrases Near-Linear Time Algorithm, n-fold ILP, Color Coding

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.75

Category Track A: Algorithms, Complexity and Games

Related Version A full version can be found at `https://arxiv.org/abs/1811.00950`.

Funding This work was supported by DFG project JA 612/20-1.

1 Introduction

Solving integer linear programs of the form $\max\{c^T x \mid \mathcal{A}x = b, x \in \mathbb{Z}_{\ge 0}\}$ is one of the most fundamental tasks in optimization. This problem is very general and broadly applicable, but unfortunately also very hard. In this paper we consider n-fold ILPs, a class of integer linear programs with a specific block structure. This is, when non-zero entries appear only in the first r rows of $\mathcal{A}$ and in blocks of size $s \times t$ along the diagonal underneath. More precisely, the contraint matrix in an n-fold ILP has the form

$$\mathcal{A} = \begin{pmatrix} A_1 & A_2 & \dots & A_n \\ B_1 & 0 & \dots & 0 \\ 0 & B_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & B_n \end{pmatrix},$$

where $A_1, \dots, A_n$ are $r \times t$ matrices and $B_1, \dots, B_n$ are $s \times t$ matrices. In n-fold ILPs we also allow upper and lower bounds on the variables. Throughout the paper we subdivide a solution x into bricks of length t and denote by $x^{(i)}$ the i-th one. The corresponding columns in $\mathcal{A}$ will be called blocks.

Lately, n-fold ILPs received great attention [2, 7, 12, 14, 16] and were studied intensively due to two reasons. Firstly, many optimization problems are expressible as n-fold ILPs [5, 10, 12, 14]. Secondly, n-fold ILPs indeed can be solved much more efficiently than arbitrary ILPs [7, 10, 16]. The previously best algorithm has a running time of $(rs\Delta)^{\mathcal{O}(r^2s+rs^2)}L \cdot (nt)^2 \log^2(n \cdot t) + \mathrm{LP}$ and is due to Eisenbrand et al. [7]. Here LP is the running time required for solving the corresponding LP relaxation. This augmentation algorithm is the last one in a line of research, where local improvement/augmenting steps are used to converge to an optimal solution. Clever insights about the structure of the improving directions allow them to be computed fast. Nevertheless, the dependence on n in the algorithm above is still high. Indeed, in practice a quadratic running time is simply not suitable for large data sets [3, 6, 13]. For example when analyzing big data, large real world graphs as in telecommunication networks or DNA strings in biology, the duration of the computation would go far beyond the scope of an acceptable running time [3, 6, 13]. For this reason even problems which have an algorithm of quadratic running time are still studied from the viewpoint of approximation algorithms with the objective to obtain results in subquadratic time, even at the cost of a worse quality [3, 6, 13]. Hence, it is an intriguing question, whether the quadratic dependency on the number nt of variables can be eliminated. In this paper, we answer this question affirmatively. The technical novelty comes from a surprising area: We use a combinatorial structure called splitter, which has been used to derandomize Color Coding algorithms. It allows us to build a powerful data structure that is maintained during the local search and from which we can derive an improving direction in logarithmic time. Handling unbounded variables in an n-fold ILP is a non-trivial issue in the previous algorithms from literature. They had to solve the corresponding LP relaxation and use proximity results. Unfortunately, it is not known whether linear programming can be solved in near-linear time in the number of variables. Hence, it is an obstacle for obtaining a near-linear running time. We manage to circumvent the necessity of solving the LP by introducing artificial bounds as a function of the finite upper bounds and the right-hand side of the n-fold ILP.

Summary of Results

- We present an algorithm, which solves n-fold ILPs in time $(rs\Delta)^{\mathcal{O}(r^2s+s^2)}L \cdot nt \log^5(nt) + \mathrm{LP}$, where LP is the time to solve the LP relaxation of the n-fold ILP. This is the first algorithm with a near-linear dependence on the number of variables. The crucial step is to speed up the computation of the improving directions.
- We circumvent the need for solving the LP relaxation. This leads to a purely combinatorial algorithm with running time $(rs\Delta)^{\mathcal{O}(r^2s+s^2)}L^2 \cdot nt \log^7(nt)$.
- In the running times above the dependence on the parameters, i.e., $(rs\Delta)^{\mathcal{O}(r^2s+s^2)}$, improves on the function $(rs\Delta)^{\mathcal{O}(r^2s+rs^2)}$ in the previous best algorithms.

Outline of New Techniques

We will briefly elaborate the main technical novelty in this paper. Let x be some feasible, non-optimal solution for the n-fold ILP. It is clear that when y^* is an optimal solution for $\max\{c^T y \mid \mathcal{A}y = 0, \ell - x \le y \le u - x, y \in \mathbb{Z}^{nt}\}$, then $x + y^*$ is optimal for the initial n-fold ILP. In other words, y^* is a particularly good improving step. A sensible approximation of y^* is to consider directions y of small size and multiplying them by some step length, i.e., find some $\lambda \cdot y$ with $\|y\|_1 \le k$ for a value k depending only on Δ, r, and s. This implies that at most k of the n blocks are used for y. If we randomly color the blocks with k^2 colors, then with high probability at most one block of every color is used. This reduces the problem to choosing a solution of a single brick for every color and to aggregate them. We add data structures for every color to implement this efficiently. There is of course a chance that the colors do not split y perfectly. We handle this by using a deterministic structure of multiple colorings (instead of one) such that it is guaranteed that at least one of them has the desired property.

Related Work

The first XP-time algorithm for solving n-fold integer programs is due to De Loera et al. [5] with a running time of $n^{g(\mathcal{A})} L$. Here $g(\mathcal{A})$ denotes a so-called Graver complexity of the constraint matrix $\mathcal{A}$ and L is the encoding length of the largest number in the input. This algorithm already uses the idea of iterative converging to the optimal solution by finding improving directions. Nevertheless, the Graver complexity appears to be huge even for small n-fold integer linear programs and thus this algorithm was of no practical use [10]. The exponent of this algorithm was then greatly improved by Hemmecke et al. [10] to a constant factor yielding the first cubic time algorithm for solving n-fold ILPs. More precisely, the running time of their algorithm is $\Delta^{\mathcal{O}(t(rs+st))} L \cdot (nt)^3$, i.e., FPT-time parameterized over Δ, r, s, and t. Lately, two more breakthroughs were obtained. One of the results is due to Koutecký et al. [16], who gave a strongly polynomial algorithm with running time $\Delta^{\mathcal{O}(r^2 s + rs^2)} (nt)^6 \cdot \log(nt) + LP$. Here LP is the running time for solving the corresponding LP relaxation, which is possible in strongly polynomial time, since the entries of the matrix are bounded. Simultaneously, Eisenbrand et al. reduced in [7] the running time from a cubic factor to a quadratic one by introducing new proximity and sensitivity results. This leads to an algorithm with running time $(\Delta rs)^{\mathcal{O}(r^2 s + rs^2)} L \cdot (nt)^2 \log^2(nt) + LP$. Note that both results require only polynomial dependency on t.

As for applications, n-fold ILPs are broadly used to model various problems such as string, flow or scheduling problems. We refer to the works [5, 10, 11, 12, 15, 19] and the references therein for an overview.

Structure of the Document

In Section 2 we introduce the necessary preliminaries. Section 3 gives the algorithm for efficiently computing the augmenting steps. This is then integrated into an algorithm for n-fold ILPs in Section 4. At first we require finite variable bounds and then discuss how to eliminate this requirement using the solution of the LP relaxation. Finally, in Section 5 we discuss how to handle infinite variable bounds without the LP relaxation and give new structural results.

2 Preliminaries

In the following we introduce n-fold ILPs formally and state the main results regarding them. Further we familiarize splitters, a technique known from Color Coding.

▶ **Definition 1.** *Let $n, r, s, t \in \mathbb{N}$. Furthermore let $A_1, \ldots, A_n$ be $r \times t$ integer matrices and $B_1, \ldots, B_n$ be $s \times t$ integer matrices. Then the n-fold matrix $\mathcal{A}$ is of following form:*

$$\mathcal{A} = \begin{pmatrix} A_1 & A_2 & \ldots & A_n \\ B_1 & 0 & \ldots & 0 \\ 0 & B_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & B_n \end{pmatrix}.$$

The matrix $\mathcal{A}$ is of dimension $(r+n\cdot s)\times n\cdot t$. We will divide $\mathcal{A}$ into blocks of size $(r+n\cdot s)\times t$. Similarly, the variables of a solution x are partitioned into bricks of length t. This means each brick $x^{(i)}$ corresponds to the columns of one submatrix A_i and therefore also B_i. Given $c, \ell, u \in \mathbb{Z}^{n\cdot t}$ and $b \in \mathbb{Z}^{r+n\cdot s}$, the corresponding n-fold Integer Linear Programming problem is defined by:

$$\max\{c^T x \mid \mathcal{A}x = b, \ell \le x \le u,\ x \in \mathbb{Z}^{n\cdot t}\}.$$

The main idea for the state-of-the-art algorithms relies on some insight about the Graver basis of n-fold ILPs, which are special elements of the kern of $\mathcal{A}$. More formally, we introduce the following definitions:

▶ **Definition 2.** *The kern of a matrix A is defined as the set of integral vectors $x \in \mathbb{Z}^{\ltimes\cdot\approx}$ with $Ax = 0$. We write $\mathrm{kern}(A)$ for them.*

▶ **Definition 3.** *A Graver basis element g is a minimal element of $\mathrm{kern}(A)$. An element is minimal, if it is not the sum of two sign-compatible elements $u, v \in \mathrm{kern}(A)$.*

Here, sign-compatible means that $u_i \cdot v_i \ge 0$ for every i.

▶ **Theorem 4** ([4]). *Let $A \in \mathbb{Z}^{n\times m}$ and let $x \in \mathrm{kern}(A)$. Then there exist $2n-1$ Graver basis elements $g_1, \ldots, g_{2n-1}$, which are sign-compatible with x such that*

$$x = \sum_{i=1}^{2n-1} \lambda_i g_i$$

for some $\lambda_1, \ldots, \lambda_{2n-1} \in \mathbb{Z}_{\ge 0}$.

Many results for n-fold ILPs rely on the fact that the ℓ_1-norm of Graver basis elements for n-fold matrices are small. The best bound known for the ℓ_1-norm is due to Eisenbrand et al. [7].

▶ **Theorem 5** ([7]). *The ℓ_1-norm of the Graver basis elements of an n-fold matrix $\mathcal{A}$ is bounded by $\mathcal{O}(rs\Delta)^{rs}$.*

Next, we will introduce a technique called splitters (see e.g. [17]), which has its origins in the FPT community and was used to derandomize the Color Coding technique [1]. So far it has not been used with n-fold ILPs. We refer the reader to the outline of techniques in the introduction for the idea on how we apply the splitters.

▶ **Definition 6.** *An (n, k, ℓ) splitter is a family of hash functions F from $\{1, \ldots, n\}$ to $\{1, \ldots, \ell\}$ such that for every $S \subseteq \{1, \ldots, n\}$ with $|S| = k$, there exists a function $f \in F$ that splits S evenly, that is, for every $j, j' \leq \ell$ we have $|f^{-1}(j) \cap S|$ and $|f^{-1}(j') \cap S|$ differ by at most 1.*

If $\ell \geq k$, the above means that there is some hash function that has no collisions when restricted to S. Interestingly, there exist splitters of very small size.

▶ **Theorem 7** ([1, 18]). *There exists an (n, k, k^2) splitter of size $k^6 \log(k) \log(n)$ which is computable in time $k^6 \log(k) \cdot n \log(n)$.*

We note that an alternative approach to the result above is to use FKS hashing. Although it has an extra factor of $\log(n)$, it is particularly easy to implement.

▶ **Theorem 8** (Corollary 2 and Lemma 2, [9]). *Define for every prime $q < k^2 \log(n)$ and prime $p < q$ the hash function $x \mapsto 1 + (p \cdot (x \mod q) \mod k^2)$. This is an (n, k, k^2) splitter of size $\mathcal{O}(k^4 \log^2(n))$.*

We remark that a hash function from $\{1, \ldots, n\}$ to $\{1, \ldots, \ell\}$ naturally corresponds to a partition of the set $\{1, \ldots, n\}$ into exactly ℓ subsets.

3 Efficient Computation of Improving Directions

The backbone of our algorithm is the efficient computation of augmenting steps. The important aspect is the fact that we can update the augmenting steps very efficiently if the input changes only slightly. In other words, whenever we change the current solution by applying an augmenting step, we do not have to recompute the next augmenting step from scratch. The augmenting steps depend on a partition of the bricks. In the following we define the notion of a best step based on a fixed partition. Later, we will independently find steps for a number of partitions and take the best among them.

▶ **Definition 9.** *Let P be a partition of the n bricks into k^2 disjoint sets $P_1, P_2, \ldots, P_{k^2}$. Let $\bar{u} \in \mathbb{Z}^{nt}_{\geq 0}$ and $\bar{\ell} \in \mathbb{Z}^{nt}_{\leq 0}$ be some upper and lower bounds on the variables (not necessarily the same as in the n-fold ILP). A (P, k)-best step is an optimal solution of the system below. We slightly abuse notation by using P_j or bricks $S_j \in P_j$ for the indices of variables contained in them.*

$$\begin{aligned}
\max\ & c^T x \\
\mathcal{A}x &= 0 \\
\sum_{i \in S_j} |x_i| &\leq k && \forall j \in \{1, \ldots, k^2\} \\
x_i &= 0 && \forall j \in \{1, \ldots, k^2\}, i \in P_j \setminus \{S_j\} \\
\bar{\ell} \leq x &\leq \bar{u} \\
x &\in \mathbb{Z}^{nt} \\
S_j &\in P_j && \forall j \in \{1, \ldots, k^2\}.
\end{aligned}$$

This means a (P, k)-best step is an element of $\operatorname{kern}(\mathcal{A})$, which uses only one brick of every $P_j \in P$. Within that brick the norm of the solution must be at most k. Later, we will choose k as the upper bound for the ℓ_1-norm of a Graver basis element, i.e., $k = \mathcal{O}(rs\Delta)^{rs}$.

▶ **Theorem 10.** *Consider the problem of finding a (P, k)-best step in an n-fold matrix where the lower and upper bounds $\overline{u}, \overline{\ell}$ can change. This problem can be solved initially in time $k^{\mathcal{O}(r)} \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot nt$ and then in $k^{\mathcal{O}(r)} \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot \log(nt)$ update time whenever the bounds of a single variable change.*

Proof. Let P be a partition of the bricks from matrix $\mathcal{A}$ into k^2 disjoint sets $P_1, P_2, \dots, P_{k^2}$. Solving the (P, k)-best step problem requires that from each set $P_j \in P$ we choose at most one brick and set this brick's variables. All variables in other bricks of P_j must be 0.

Let x be a (P, k)-best step and let $x^{(j)}$ have the values of x in variables of P_j and 0 in all other variables. Then by definition, $\|x^{(j)}\|_1 \leq k$. This implies that the right-hand side regarding $x^{(j)}$, that is to say, $\mathcal{A}x^{(j)}$, is also small. Since the absolute value of an entry in $\mathcal{A}$ is at most Δ, we have that $\|\mathcal{A}x^{(j)}\|_\infty \leq k\Delta$. Let a_i be the i-th row of $\mathcal{A}$. If $i > r$, then $a_i x^{(j)} = 0$. This is because $\mathcal{A}x = 0$ and a_i has all its support either completely inside P_j or completely outside P_j. Meaning, the value of $\mathcal{A}x^{(j)}$ is one of the $(2k\Delta + 1)^r$ many values we get by enumerating all possibilities for the first r rows. Furthermore, since P has only k^2 sets, the partial sum $\mathcal{A}(x^{(1)} + \cdots + x^{(j)})$ is always one of $(2k^3\Delta + 1)^r = (k\Delta)^{\mathcal{O}(r)}$ many candidates.

Hence to find a (P, k)-best step we can restrict our search to solutions whose partial sums stay in this range. To do so, we set up a graph containing $k^2 + 2$ layers $L_0, L_1, \dots L_{k^2}, L_{k^2+1}$. An example is given in figure 1. The first layer L_0 will consist of just one node marking the starting point with partial sum zero. Similarly, the last layer L_{k^2+1} will just contain the target point also having partial sum zero, since a (P, k)-best step is an element of kern$(\mathcal{A})$. Each layer L_j with $1 \leq j \leq k^2$ will contain $(2k^3\Delta + 1)^r$ many nodes, each representing one possible value of $\mathcal{A}(x^{(1)} + \cdots x^{(j)})$. Two points v, w from adjacent layers L_{j-1}, L_j will be connected if the difference of the corresponding partial sums, namely $w - v$, can be obtained by a solution y of variables from only one brick of P_j (with $\|y\|_1 \leq k$). The weight of the edge will be the largest gain for the objective function $c^T y$ over all possible bricks. Hence, it could be necessary to compute and compare up to n values for each P_j and each difference in the partial sums to insert one edge into the graph. Finally, we just have to find the longest path in this graph as it corresponds to a (P, k)-best step. The out-degree of each node is bounded by $(2k^3\Delta + 1)^r$ since at most this many nodes are reachable in the next layer. Therefore the overall number of edges is bounded by

$$(k^2 + 2) \cdot (2k^3\Delta + 1)^r \cdot (2k^3\Delta + 1)^r = (k\Delta)^{\mathcal{O}(r)}.$$

Using the Bellman-Ford algorithm we can solve the Longest Path problem for a graph with N vertices and M edges in time $\mathcal{O}(N \cdot M)$ as the graph is directed and acyclic. This gives a running time of $(k\Delta)^{\mathcal{O}(r)} \cdot (k\Delta)^{\mathcal{O}(r)} = (k\Delta)^{\mathcal{O}(r)}$ for solving the problem. Constructing the graph, however, requires solving a number of IPs of the form

$$\begin{aligned} &\max c'^T x \\ &\begin{pmatrix} A_j \\ B_j \end{pmatrix} x = \begin{pmatrix} b' \\ 0 \end{pmatrix} \\ &\|x\|_1 \leq k \\ &\ell' \leq x \leq u' \\ &x \in \mathbb{Z}^t, \end{aligned}$$

where $b' \in \mathbb{Z}^r$ is the corresponding right-hand side of the top rows and ℓ', u', c' are the upper and lower bounds, and the objective of the block. This is an IP with $r + s + 1$ constraints, t variables, lower and upper bounds, and entries of the matrix bounded by Δ in absolute value.

Using the algorithm by Eisenbrand and Weismantel [8], solving one of them requires time

$$t \cdot \mathcal{O}(r+s+1)^{r+s+4} \cdot \mathcal{O}(\Delta)^{(r+s+1)(r+s+4)} \cdot \log^2((r+s+1)\Delta) + \mathrm{LP} = t \cdot \Delta^{\mathcal{O}(r^2+s^2)} + \mathrm{LP},$$

where LP is the time for solving the LP relaxation. Note that strictly speaking the constraint $\|x\|_1 \leq k$ is not linear, but by standard construction the ILP can easily be transformed into an equivalent one replacing every variable x_i with a composition $x_i^+ - x_i^-$ of two positive variables. Then the ℓ_1-norm is simply the sum over all variables. This does not affect the asymptotic running time.

Furthermore, a little thought allows us to reduce the dependency on t to a logarithmic one: Since the number of constraints in the ILP above is very small, there are only $\Delta^{\mathcal{O}(r+s)}$ many different columns. Because of the cardinality constraint $\|x\|_1 \leq k$, we only have to consider $2k$ many variables of each type of column, namely: The k many with $u'_i > 0$ and maximal c'_i and the k many with $\ell'_i < 0$ and minimal c'_i.

If some solution uses a variable not in this set, then by pigeonhole principle there is a variable with the same column values and a superior objective value and which can be increased/decreased. We can reduce the variable outside this set and increase the corresponding variable inside this set until all variables outside the set are 0. We can use an appropriate data structure (e.g. AVL trees) to maintain a set of all variables with $u'_i > 0$ ($\ell'_i < 0$) such that we can find the k best among them in time $\mathcal{O}(k\log(t))$. Whenever the bounds of some variable change, we might have to add or remove entries, which also takes only logarithmic time. After initialization in time $\mathcal{O}(nt)$ (in total for all bricks) solving such an IP can therefore be implemented in time

$$k\log(t) + 2k\Delta^{\mathcal{O}(r+s)}\Delta^{\mathcal{O}(r^2+s^2)} + \mathrm{LP} \leq k\log(t)\Delta^{\mathcal{O}(r^2+s^2)} + \mathrm{LP} \leq k\log(t)\Delta^{\mathcal{O}(r^2+s^2)}.$$

The last inequality holds, because using Tardos' algorithm [20] LPs can be solved in time polynomial in the encoding size of the matrix, which can be bounded by $2k\Delta^{\mathcal{O}(r+s)} \cdot (r+s) \cdot \log(\Delta)$. This is dominated by the other term. The number of IPs to solve is at most n times the number of edges, since we have to compare the values of up to n bricks. This gives a running time of

$$\mathcal{O}(nt) + n \cdot (k\Delta)^{\mathcal{O}(r)} \cdot \log(t) \cdot \Delta^{\mathcal{O}(r^2+s^2)} \leq nt \cdot k^{\mathcal{O}(r)} \cdot \Delta^{\mathcal{O}(r^2+s^2)}$$

for constructing the graph. To obtain the update time from the premise of the theorem, it is perfectly fine to solve the Longest Path problem again, but we cannot construct the graph from scratch. However, in order to construct the graph we still have to find the best value over all bricks for each edge. Fortunately, if only a few bricks are updated (in their lower and upper bounds) it is not necessary to recompute all values. Each edge corresponds to a particular $P_j \in P$ and a fixed right-hand side (a possible value of $\mathcal{A}x^{(j)}$). We require an appropriate data structure $\mathcal{D}_e$ for every edge e, which supports fast computation of the operations FindMax, Insert, and Delete. Again, an AVL tree computes each of these operations in time $\mathcal{O}(\log(N))$, where N is the number of elements. In $\mathcal{D}_e$ we store pairs (v, i) where i is a brick in P_j and v is the maximum gain of brick i for the right-hand side of e. The pairs are stored in lexicographical order. Since there are at most n bricks in P_j, the data structure will have at most n elements. Initially, we can build $\mathcal{D}_e$ in time $nt \cdot \Delta^{\mathcal{O}(r^2+s^2)}$ (this is replicated for each edge). Now consider a change to the instance. Recall that we are looking at changes that affect only a single brick, namely the upper and lower bounds within that brick change. We are going to update the data structure $\mathcal{D}_e$ (for each edge) to reflect the changes and we are going to recompute the edge value of each edge e using $\mathcal{D}_e$. Then we

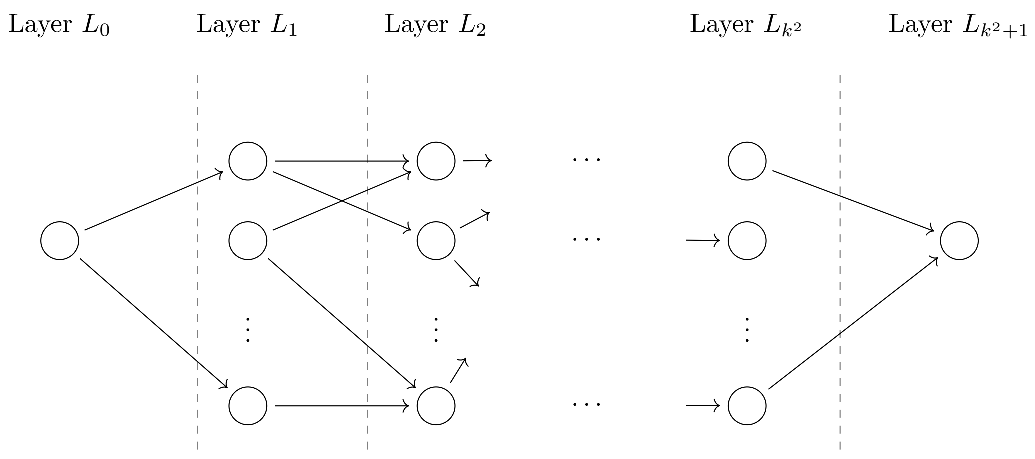

Figure 1 This figure shows an example for a layered graph obtained while solving the (P, k)-best step problem. There are $k^2 + 2$ layers, visually separated by gray dashed lines. This includes one source layer L_0, one target layer L_{k^2+1} both with just a single node representing the zero sum. Further there are k^2 layers with $(2k^3\Delta + 1)^r$ nodes each, where in one layer the nodes stand for all reachable partial sums. Two points v, w from adjacent layers L_{j-1}, L_j will be connected if the difference of the corresponding partial sums, namely $w - v$, can be obtained by a solution y of variables from only one brick of P_j (with $\|y\|_1 \le k$). The weight of the edge will be the largest gain for the objective function $c^T y$ over all possible bricks. For the sake of clarity both the values of the nodes and the edges are not illustrated.

simply solve the Longest Path problem again. Let $P_j \in P$ be the set that contains the brick i that has changed in some variable. We only have to consider edges from L_{j-1} to L_j, since none of the other edges are affected by the change. For a relevant edge e we compute the previous value v and current value v' that the brick i would produce (before and after the bounds have changed). In $\mathcal{D}_e$ we have to remove (v, i) and insert (v', i). Both operations need only $O(\log(n))$ time. Then the running time to update $\mathcal{D}_e$ for one edge is

$$k \log(t) \cdot \Delta^{\mathcal{O}(r^2+s^2)} + \mathcal{O}(\log(n)) \le k \log(nt) \cdot \Delta^{\mathcal{O}(r^2+s^2)}.$$

In order to update the edge value of e using $\mathcal{D}_e$, we simply have to find the maximum element in $\mathcal{D}_e$, which again takes time $\mathcal{O}(\log(n))$. To summarize, the total time to update the (P, k)-best step after a change to a single brick consists of (1) updating each $\mathcal{D}_e$, (2) finding the maximum in each $\mathcal{D}_e$, and (3) solving the Longest Path problem. We conclude that the update time is

$$\begin{aligned} &k \log(nt) \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot (k\Delta)^{\mathcal{O}(r)} + \log(n) \cdot (k\Delta)^{\mathcal{O}(r)} + (k\Delta)^{\mathcal{O}(r)} \\ &\qquad \le k^{\mathcal{O}(r)} \Delta^{\mathcal{O}(r^2+s^2)} \cdot \log(nt). \end{aligned}$$ ◀

4 The Augmenting Step Algorithm

In this section we will assume that all lower and upper bounds are finite and give a complete algorithm for this case. Later, we will explain how to cope with infinite bounds. We start by showing how to converge to an optimal solution when an initial feasible solution is given. To compute the initial solution, we also apply this algorithm on a slightly modified instance. The approach resembles the procedure in previous literature, although we apply the results from the previous section to speed up the computation of augmenting steps.

Let x be a feasible solution for the n-fold ILP, in particular $\mathcal{A}x = b$. Let x^* be an optimal one. Theorem 4 states that we can decompose the difference vector $x' = x^* - x$ into at most $2nt - 1$ weighted Graver basis elements, that is

$$x' = x^* - x = \sum\nolimits_{j=1}^{2nt-1} \lambda_j g_j.$$

For intuition, consider the following simple approach (this is similar to the algorithm by Hemmecke et al. [10]). Suppose we are able to guess the best vector $\lambda_i g_i = \text{argmax}_j \{c^T(\lambda_j g_j)\}$ regarding the gain for the objective function. This pair of step length λ_i and Graver element g_i is called the Graver best step. Then we can augment the current solution x by adding $\lambda_i g_i$ to it, i.e., we set $x \leftarrow x + \lambda_i g_i$. Feasibility follows because all g_j are sign-compatible. This procedure is repeated until no improving step is possible and therefore x must be optimal. In each iteration this decreases the gap to the optimal solution by a factor of at least $1 - 1/(2nt)$ by the pigeonhole principle. It may be costly to guess the precise Graver best step, but for our purposes it will suffice to find an augmenting step that is approximately as good.

We will now describe how to guess λ_i. Since $x + \lambda_i g_i$ is feasible, we have that $\lambda_i g_i \leq u - x \leq u - \ell$ and $\lambda_i g_i \geq \ell - x \geq \ell - u$. Let $(g_i)_j \in \text{supp}(g_i)$ be some non-zero variable. If $(g_i)_j > 0$, then $\lambda_i \leq (\lambda_i g_i)_j \leq u_j - \ell_j$. Otherwise, $(g_i)_j < 0$ and $\lambda_i \leq -(\lambda_i g_i)_j \leq -(\ell_j - u_j) = u_j - \ell_j$. Hence, it suffices to check all values in the range $\{1, \ldots, \Gamma\}$, where $\Gamma = \max_j\{u_j - \ell_j\}$. Proceeding like in [7], we lower the time a bit further by not taking every value into consideration. Instead, we look at guesses of the form $\lambda' = 2^k$ for $k \in \{0, \ldots, \lfloor \log(\Gamma) \rfloor\}$. Doing so we lose a factor of at most 2 regarding the improvement of the objective function, since $c^T(\lambda' g_i) > 0.5 \cdot c^T(\lambda_i g_i)$ when taking $\lambda' = 2^{\lfloor \log(\lambda_i) \rfloor} > \lambda_i/2$. Fix λ' to the value above. Next we describe how to compute an augmenting step that is at least as good as $\lambda' g_i$. Note that g_i is a solution of

$$\begin{aligned} \mathcal{A}y &= 0 \\ \|y\|_1 &\leq k \\ \lceil \frac{\ell - x}{\lambda'} \rceil \leq y &\leq \lfloor \frac{u - x}{\lambda'} \rfloor, \end{aligned}$$

where $k = \mathcal{O}(rs\Delta)^{rs}$ is the bound on the norm of Graver elements from Theorem 5. Suppose we have guessed some partition $P = \{P_1, \ldots, P_{k^2}\}$ of the bricks such that of each P_j only a single brick has non-zero variables in g_i. Clearly, the augmenting step $\lambda' y^*$, where y^* is a (P, k)-best step with bounds $\overline{\ell} = \lceil \frac{\ell - x}{\lambda'} \rceil$ and $\overline{u} = \lfloor \frac{u - x}{\lambda'} \rfloor$ would be at least as good as $\lambda' g_i$. Indeed Theorem 10 explains how to compute such a (P, k)-best step dynamically and when we add $\lambda' g_i$ to x we only change the bounds of at most k^3 many variables. Hence, it is very efficient to recompute (P, k)-best steps until we have converged to the optimal solution. However, valid choices of λ' and P might be different in every iteration. Regarding λ', we simply compute (P, k)-best steps for each of the $\mathcal{O}(\log(\Gamma))$ many guesses and take the best among them. We proceed similarly for P. We guess a small number of partitions and guarantee that always at least one of them is valid. For this purpose we employ splitters. More precisely, we compute a (n, k, k^2) splitter of the n bricks. Since g_i has a norm bounded by k, it can also only use at most k bricks. Therefore the splitter always contains a partition $P = \{P_1, \ldots, P_{k^2}\}$ where g_i only uses a single brick in every P_j.

To recap, in every iteration we solve a (P, k)-best step problem for every guess λ' and every partition P in the splitter and take the overall best solution as an improving direction $\lambda' y^*$. Then we update our solution x by adding $\lambda' y^*$ onto it. At most k^2 many bricks change (and within each brick only k variables can change) and therefore we can efficiently recompute the (P, k)-best steps for every guess for the next iteration. This way we guarantee that we improve the solution by a factor of at least $1 - 1/(4nt)$ in every iteration. The explicit running time of these steps will be analyzed in the next theorem.

Recall that we still have to find an initial solution. This solution can indeed be computed by using the augmenting step algorithm described above. Roughly speaking, we modify our n-fold matrix by introducing new variables, such that it admits a trivial initial solution. Introducing a new objective function which penalizes using these new variables, an optimal solution clearly corresponds to an initial solution of the original n-fold ILP. The detailed procedure is given in the full version of this paper.

▶ **Theorem 11.** *The dynamic augmenting step algorithm described above computes an optimal solution for the n-fold Integer Linear Program problem in time $(rs\Delta)^{\mathcal{O}(r^2s+s^2)} \cdot \mathcal{O}(L^2 \cdot nt\log^5(nt))$ when finite variable bounds are given for each variable. Here L is the encoding length of the largest occurring number in the input.*

Proof. Due to Theorem 4 we know that the difference vector of an optimal solution x^* to our current solution x, i.e. $x' = x^* - x$, can be decomposed into $2nt - 1$ weighted Graver basis elements. Hence, if we adjust our solution x with the Graver best step, we reduce the gap between the value of an optimal solution and our current solution by a factor of at least $1 - 1/(2nt)$ due to the pigeonhole principle. Our algorithm finds an augmenting step that is at least half as good as the Graver best step. Therefore, the gap to the optimal solution is still reduced by at least a factor of $1 - 1/(4nt)$.

Regarding the running time we first have to compute the splitter. Theorem 7 says, that this can be done in time $k^{\mathcal{O}(1)} \cdot n\log(n) = (rs\Delta)^{\mathcal{O}(rs)} \cdot n\log(n)$. Next we have to try all values for the weight λ. Due to our step-length we get $\mathcal{O}(\log(\Gamma))$ guesses. Recall that Γ denotes the largest difference between an upper bound and the corresponding lower bound, i.e., $\Gamma = \max_j\{u_j - \ell_j\}$. Fixing one, we have to find the best improving direction regarding each of the $((rs\Delta)^{\mathcal{O}(rs)})^{\mathcal{O}(1)}\log(n) = (rs\Delta)^{\mathcal{O}(rs)}\log(n)$ partitions. In the first iteration we have to set up the tables in time $k^{\mathcal{O}(r)} \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot nt = (rs\Delta)^{\mathcal{O}(r^2s)} \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot nt$ by computing the gain for each possible summand for each set and setting up the data structure. In each following iteration we update each table and search for the optimum in time $k^{\mathcal{O}(r)} \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot \log(nt) = (rs\Delta)^{\mathcal{O}(r^2s)} \cdot \Delta^{\mathcal{O}(r^2+s^2)} \cdot \log(nt)$. Now it remains to bound the number I of iterations needed to converge to an optimal solution. To obtain such a bound we calculate:

$$1 > (1 - 1/(4nt))^I |c^T(x^* - x)|.$$

By reordering the term, we get

$$I < \frac{-\log(|c^T(x^* - x)|)}{\log(1 - 1/(4nt))}.$$

As $\log(1 + x) = \Theta(x)$, we can bound $\log(1 - 1/(4nt))$ by $\Theta(-1/(4nt))$ and thus

$$I < \mathcal{O}(\frac{-\log(|c^T(x^* - x)|)}{-1/(4nt)}) \leq \mathcal{O}(4nt\log(|c^T(x^* - x)|)).$$

As the maximal difference between the current solution x and an optimal one x^* can be at most the maximal value of c times the largest number in between the bounds for each variable, we get $|c^T(x^* - x)| \leq nt\max_i |c_i| \cdot \Gamma$ and thus

$$I < \mathcal{O}(4nt\log(|c^T(x^* - x)|)) \leq \mathcal{O}(nt\log(nt\max_i |c_i| \cdot \Gamma)) \leq \mathcal{O}(nt\log(nt\Gamma\max_i |c_i|)).$$

Let L denote the encoding length of largest integer in the input. Clearly 2^L bounds the largest absolute value in c and thus we get

$$I < \mathcal{O}(nt \log(nt\Gamma \max_i |c_i|)) = \mathcal{O}(nt \log(nt\Gamma 2^L)) = \mathcal{O}(nt \log(nt\Gamma 2^L)).$$

Hence after this amount of steps by always improving the gain by a factor of at least $1 - 1/(4nt)$ we close the gap between the initial solution and an optimal one. Given this, we can now bound the overall running time with:

$$\begin{aligned}
&\underbrace{(rs\Delta)^{\mathcal{O}(rs)} \cdot n\log(n)}_{\text{Splitter}} + \underbrace{(rs\Delta)^{\mathcal{O}(rs)} \log(n)}_{\text{Partitions}} \cdot \underbrace{(rs\Delta)^{\mathcal{O}(r^2 s)} \cdot (rs\Delta)^{\mathcal{O}(r^2+s^2)} \cdot nt}_{\text{First Iteration}} + \\
&\underbrace{\mathcal{O}(nt\log(nt\Gamma 2^L))}_{I} \cdot \underbrace{\mathcal{O}(\log(\Gamma))}_{\lambda\text{ Guesses}} \cdot \underbrace{(rs\Delta)^{\mathcal{O}(rs)} \log(n)}_{\text{Partitions}} \cdot \underbrace{(rs\Delta)^{\mathcal{O}(r^2 s)} \cdot (rs\Delta)^{\mathcal{O}(r^2+s^2)} \cdot \log(nt))}_{\text{Update Time}} \\
&= \mathcal{O}((nt\log(nt\Gamma 2^L)) \cdot \mathcal{O}(\log(\Gamma)) \cdot (rs\Delta)^{\mathcal{O}(r^2 s+s^2)} \cdot \log^2(nt) \\
&= (rs\Delta)^{\mathcal{O}(r^2 s+s^2)} \cdot \mathcal{O}(\log^2(\Gamma + 2^L) \cdot nt\log^3(nt)).
\end{aligned}$$

Here *Splitter* denotes the time to compute the initial set $\mathcal{P}$ of partitions and *Partitions* denotes the cardinality of $\mathcal{P}$. *First Iteration* is the time to solve the first iteration of the (P, k)-best step problem. Further λ *Guesses* is the number of guesses we have to do to get the right weight and lastly *Update Time* is the time needed to solve each following (P, k)-best step including updating the bounds and data structures.

Note, that we still have to argue about finding the initial solution, since in the construction of the modified n-fold ILP the parameters slightly change. The length of a brick expand to $t' = t + r + s$, as shown in the full version of this paper. This, however, can be hidden in the $\mathcal{O}$-Notation of $(rs\Delta)^{\mathcal{O}(r^2 s+s^2)}$. Further, Γ', the biggest difference in upper and lower bounds can change as we introduced new variables admitting new bounds. The difference between the bounds of old variables does not change. For the new variables, however, the difference can be as large as $\|b'\|_\infty$. Thus we bound this value by

$$\|b'\|_\infty = \|b - \mathcal{A}\ell\|_\infty \le \|b\|_\infty + \|\mathcal{A}\ell\|_\infty \le \|b\|_\infty + \Delta \cdot \|\ell\|_1 \le \mathcal{O}(\Delta \cdot nt \cdot 2^L).$$

We conclude that the running time for finding an initial solution (and also the overall running time) is

$$\begin{aligned}
(rs\Delta)^{\mathcal{O}(r^2 s+s^2)} \mathcal{O}(\log^2(\Gamma' + 2^L) nt \log^3(nt)) &= (rs\Delta)^{\mathcal{O}(r^2 s+s^2)} \mathcal{O}(\log^2(\Delta 2^L nt) nt \log^3(nt)) \\
&= (rs\Delta)^{\mathcal{O}(r^2 s+s^2)} \cdot L^2 nt \log^5(nt). \qquad \blacktriangleleft
\end{aligned}$$

Handling Infinite Bounds

Remark, that if no finite bounds are given for all variables, we have to introduce some artificial bounds first. Here we can proceed as in [7], where first the LP relaxation is solved to obtain an optimal fractional solution z^*. Using the proximity results from [7], we know that an optimal integral solution x^* exists such that $\|x^* - z^*\|_1 \le nt(rs\Delta)^{\mathcal{O}(rs)}$. This allows us to introduce artificial upper bounds for the unbounded variables. Remark that this comes at the price of solving the corresponding relaxation of the n-fold Integer Linear Program problem. However we also lessen the dependency from L^2 to L as the finite upper and lower bounds can also be bounded more strictly due to the same proximity result. This yields an overall running time of $(rs\Delta)^{\mathcal{O}(r^2 s+s^2)} \cdot L \cdot nt \log^5(nt) + \text{LP}$. Nevertheless, solving this LP can be very costly, indeed it is not clear if a potential algorithm even runs in time linear in n.

Thus, it may even dominate the running time of solving the n-fold ILP with finite upper bounds. Fortunately we can circumvent the necessity of solving the LP as we will describe in the following section using new structural results.

▶ **Theorem 12.** *The dynamic augmenting step algorithm described above computes an optimal solution for the n-fold Integer Linear Program problem in time $(rs\Delta)^{\mathcal{O}(r^2s+s^2)} \cdot L \cdot nt \log^5(nt) + LP$ when some variables have infinite upper bounds. Here LP is the running time to solve the corresponding relaxation of the n-fold ILP problem.*

5 Bounds on ℓ_1-norm

In the following, we state that even with infinite variable bounds in an n-fold ILP there always exists a solution of small norm (if the n-fold ILP has a finite optimum). Therefore, we can apply the algorithm for finite variable bounds by replacing every infinite one with this value. However, due to space restrictions, the proofs are omitted. They can be found in the full version of this paper.

▶ **Lemma 13.** *If the n-fold ILP is feasible and $\overline{y}$ is some vector satisfying the variable bounds, then there exists a feasible solution x with $\|x\|_1 \leq \mathcal{O}(rs\Delta)^{rs+1} \cdot (\|\overline{y}\|_1 + \|b\|_1)$*

▶ **Lemma 14.** *If the n-fold ILP is bounded and feasible, then there exists an optimal solution x with $\|x\|_1 \leq (rs\Delta)^{\mathcal{O}(rs)} \cdot (\|b\|_1 + nt\zeta)$, where ζ denotes the largest absolute value among all finite variable bounds.*

This yields an alternative approach to solving the LP relaxation, because now we can simply replace all infinite bounds with $\pm(rs\Delta)^{\mathcal{O}(rs)} \cdot nt \cdot 2^L$. Then we can apply the algorithm that works only on finite variable bounds. The new encoding length L' of the largest integer in the input can be bounded by

$$L' \leq \log((rs\Delta)^{\mathcal{O}(rs)} \cdot 2^L \cdot nt) \leq \mathcal{O}(rs \cdot \log(rs\Delta) \cdot L \cdot \log(nt)).$$

This way we obtain the following.

▶ **Corollary 15.** *We can compute an optimal solution for an n-fold ILP in time $(rs\Delta)^{\mathcal{O}(r^2s+s^2)} \cdot L^2 \cdot nt \log^7(nt)$.*

In a similar way, we can derive the following bound on the sensitivity of an n-fold ILP. This bound is not needed in our algorithm, but may be of independent interest, since it implies small sensitivity for problems that can be expressed as n-fold ILPs.

▶ **Theorem 16.** *Let x be an optimal solution of an n-fold ILP with right-hand side b, in particular, $\mathcal{A}x = b$. If the right hand side changes to b' and the n-fold ILP still has a finite optimum, then there exists an optimal solution x' for b' ($\mathcal{A}x' = b'$) with $\|x - x'\|_1 \leq \mathcal{O}(rs\Delta)^{rs} \cdot \|b - b'\|_1$.*

It is notable that this bound does not depend on n. This is in contrast to the known bounds for the distance between LP and ILP solutions of an n-fold ILP [7].

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995. Previously appeared in STOC 1994.

2 Katerina Altmanová, Dušan Knop, and Martin Koutecký. Evaluating and Tuning n-fold Integer Programming. In *17th International Symposium on Experimental Algorithms*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:14, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

3 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM Journal on Computing*, 41(6):1635–1648, 2012. Previously appeared in STOC 2009.

4 William Cook, Jean Fonlupt, and Alexander Schrijver. An integer analogue of Carathéodory's theorem. *Journal of Combinatorial Theory, Series B*, 40(1):63–70, 1986.

5 Jesús A. De Loera, Raymond Hemmecke, Shmuel Onn, and Robert Weismantel. N-fold integer programming. *Discrete Optimization*, 5(2):231–241, 2008.

6 Doratha E. Drake and Stefan Hougardy. A linear time approximation algorithm for weighted matchings in graphs. *Journal of the ACM Transactions on Algorithms*, 1(1):107–122, 2005.

7 Friedrich Eisenbrand, Christoph Hunkenschröder, and Kim-Manuel Klein. Faster Algorithms for Integer Programs with Block Structure. In *45th International Colloquium on Automata, Languages, and Programming*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:13, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

8 Friedrich Eisenbrand and Robert Weismantel. Proximity results and faster algorithms for integer programming using the steinitz lemma. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 808–816. Society for Industrial and Applied Mathematics, 2018.

9 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):538–544, 1984. Previously appeared in FOCS 1982.

10 Raymond Hemmecke, Shmuel Onn, and Lyubov Romanchuk. N-fold integer programming in cubic time. *Mathematical Programming*, pages 1–17, 2013.

11 Raymond Hemmecke, Shmuel Onn, and Robert Weismantel. N-fold integer programming and nonlinear multi-transshipment. *Optimization Letters*, 5(1):13–25, 2011.

12 Klaus Jansen, Kim-Manuel Klein, Marten Maack, and Malin Rau. Empowering the Configuration-IP - New PTAS Results for Scheduling with Setups Times. In *ITCS*, volume 124 of *LIPIcs*, pages 44:1–44:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

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

14 Dušan Knop and Martin Kouteckỳ. Scheduling meets n-fold integer programming. *Journal of Scheduling*, pages 1–11, 2017.

15 Dušan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n-fold Integer Programming and Applications. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, pages 54:1–54:14, 2017.

16 Martin Koutecký, Asaf Levin, and Shmuel Onn. A Parameterized Strongly Polynomial Algorithm for Block Structured Integer Programs. In *45th International Colloquium on Automata, Languages, and Programming*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 85:1–85:14, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

17 Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and Near-Optimal Derandomization. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 182–191, 1995.

18 Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and near-optimal derandomization. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 182–191. IEEE, 1995.

19 Shmuel Onn and Pauline Sarrabezolles. Huge Unimodular n-Fold Programs. *SIAM J. Discrete Math.*, 29(4):2277–2283, 2015.

20 Éva Tardos. A Strongly Polynomial Algorithm to Solve Combinatorial Linear Programs. *Operations Research*, 34(2):250–256, 1986. `doi:10.1287/opre.34.2.250`.